Introdução à Compilação

Sumário

Introdução à Compilação

Compilador

Efetua a tradução de um arquivo gerando um formato de saída adequado.

$$\text{Linguagem Fonte} \rightarrow \text{Linguagem Destino (ou Alvo)}$$

Característica importante

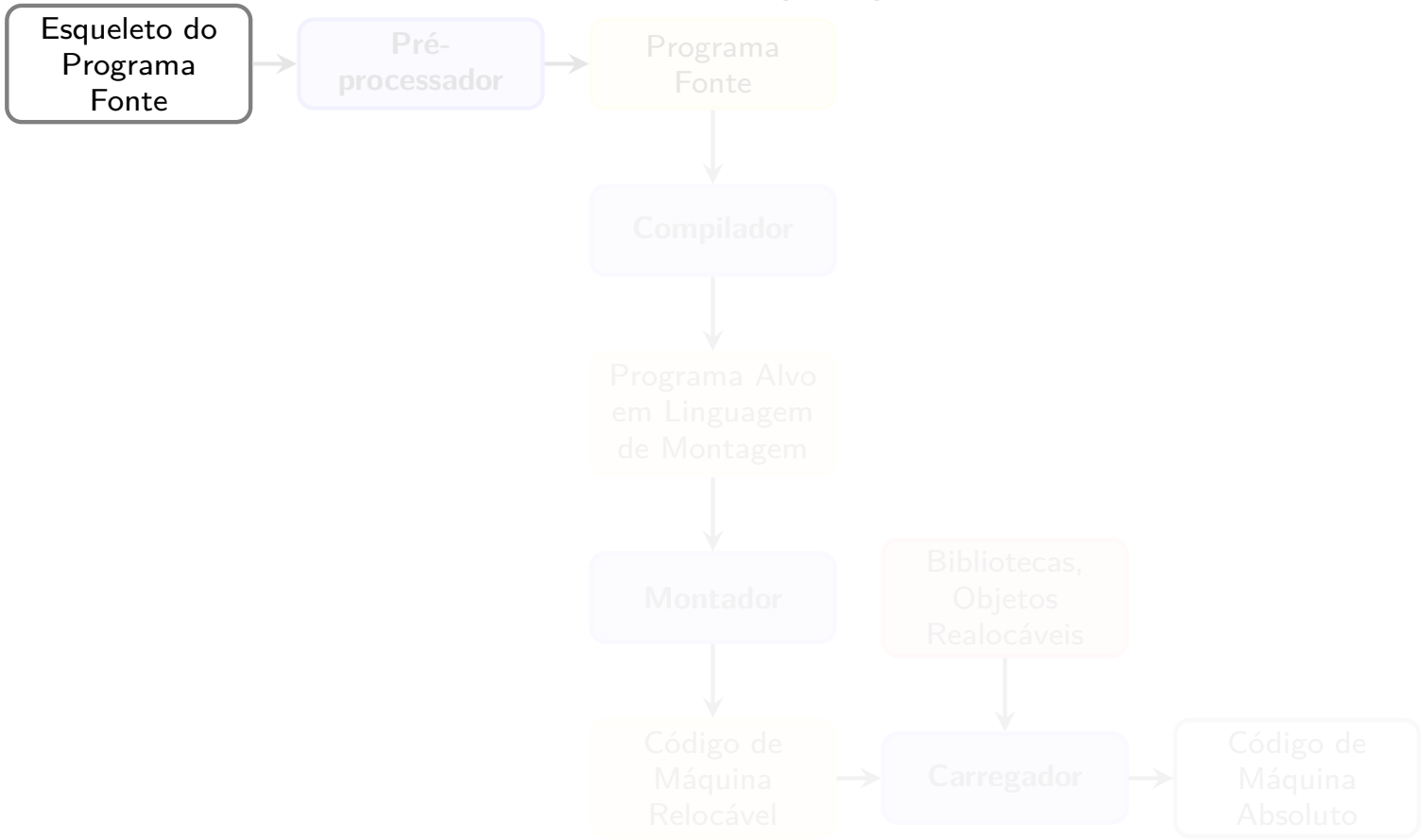
Detectar e relatar (durante o processo de tradução) a presença de erros no programa fonte.

Classificação

- Passo simples $\times$ Passo múltiplo
- Depuração
- Otimização

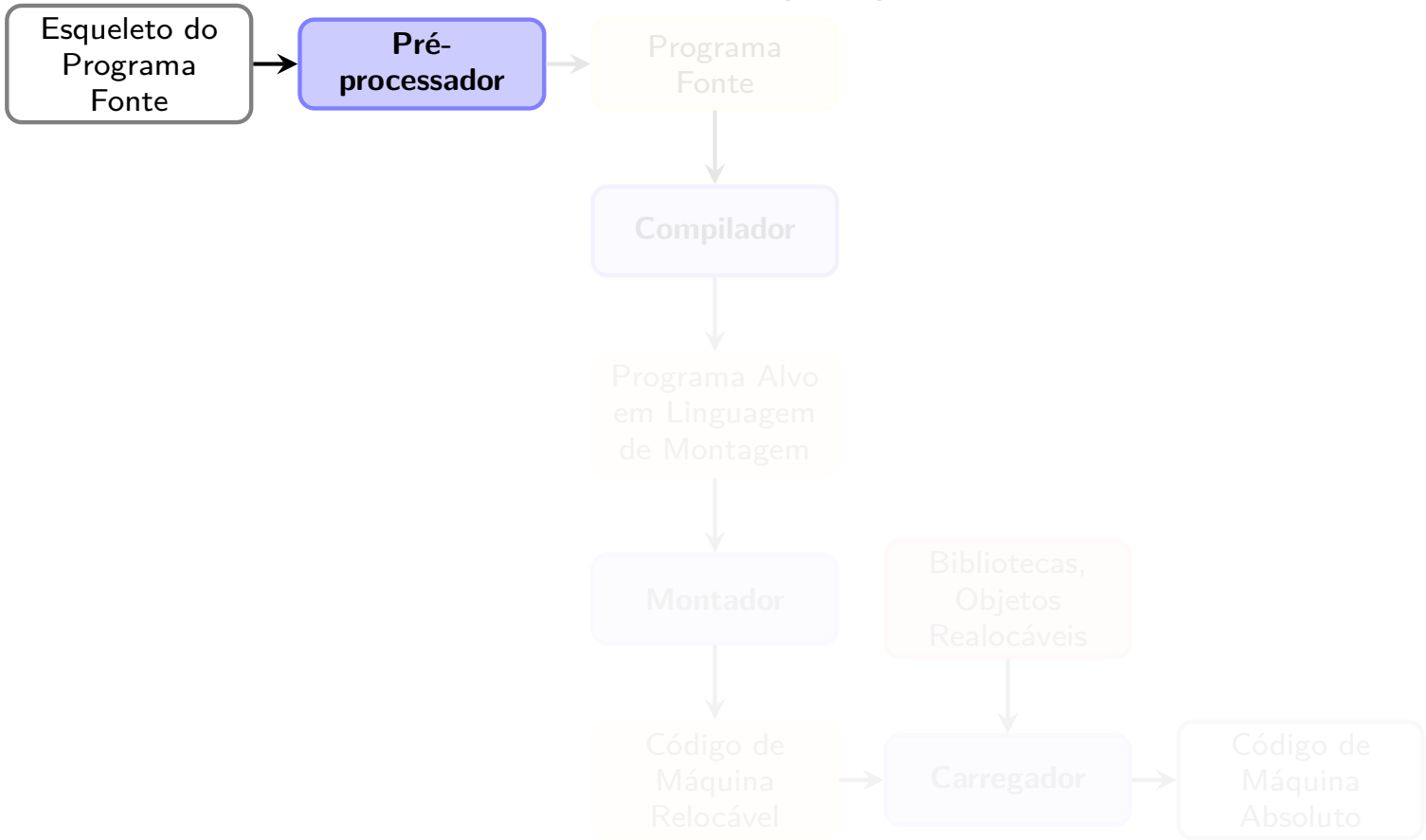
Introdução à Compilação

Sistema de Processamento de Linguagem



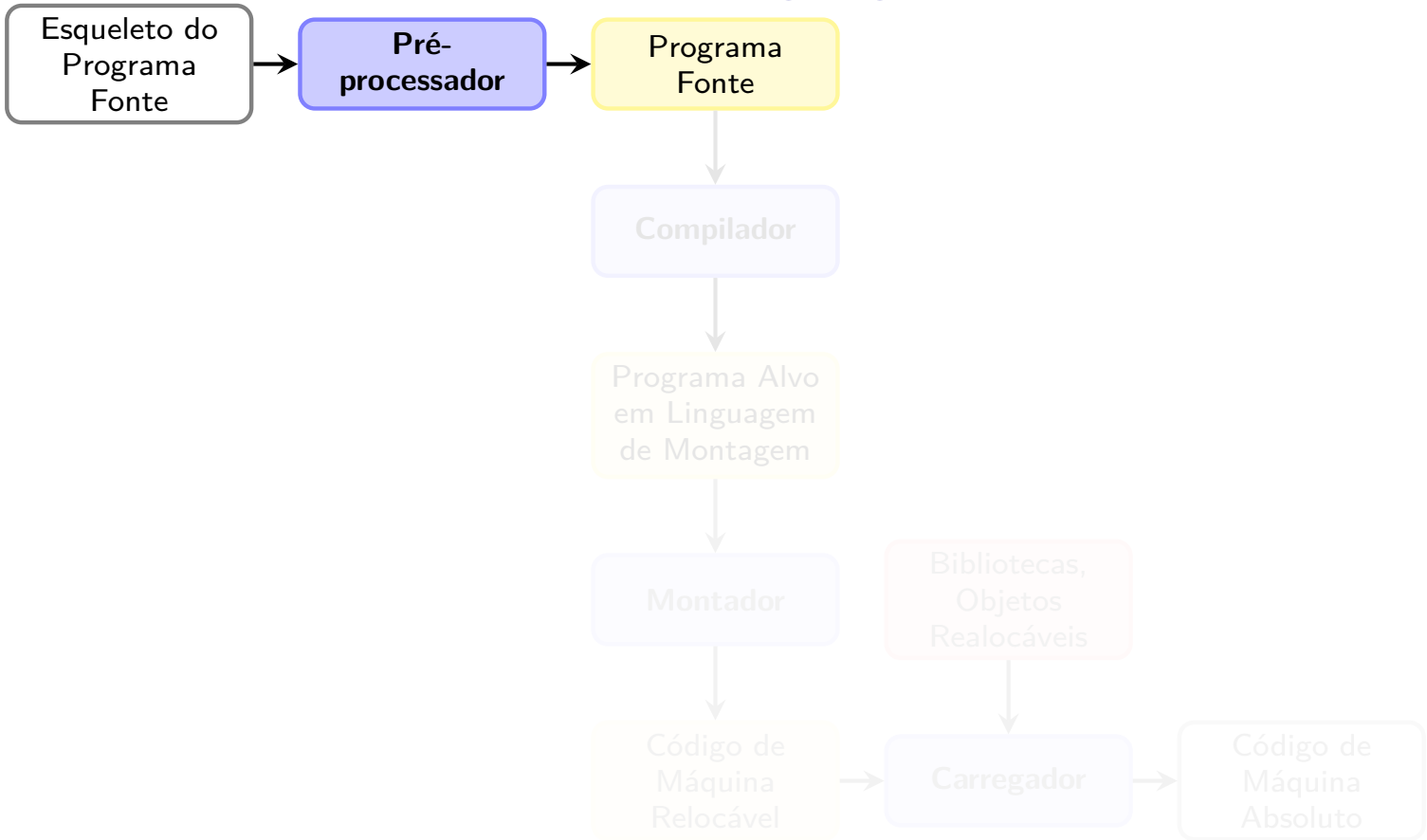
Introdução à Compilação

Sistema de Processamento de Linguagem



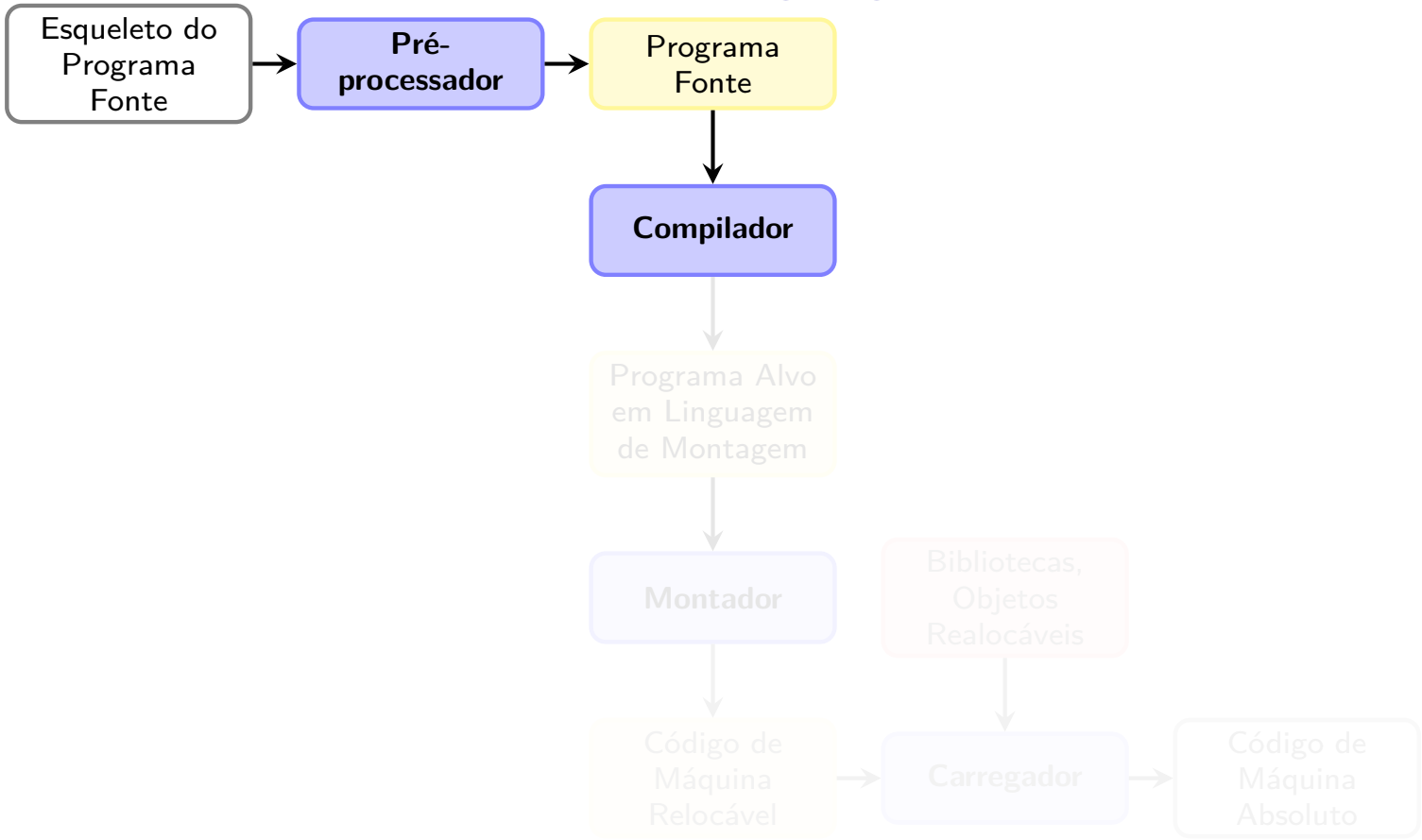
Introdução à Compilação

Sistema de Processamento de Linguagem



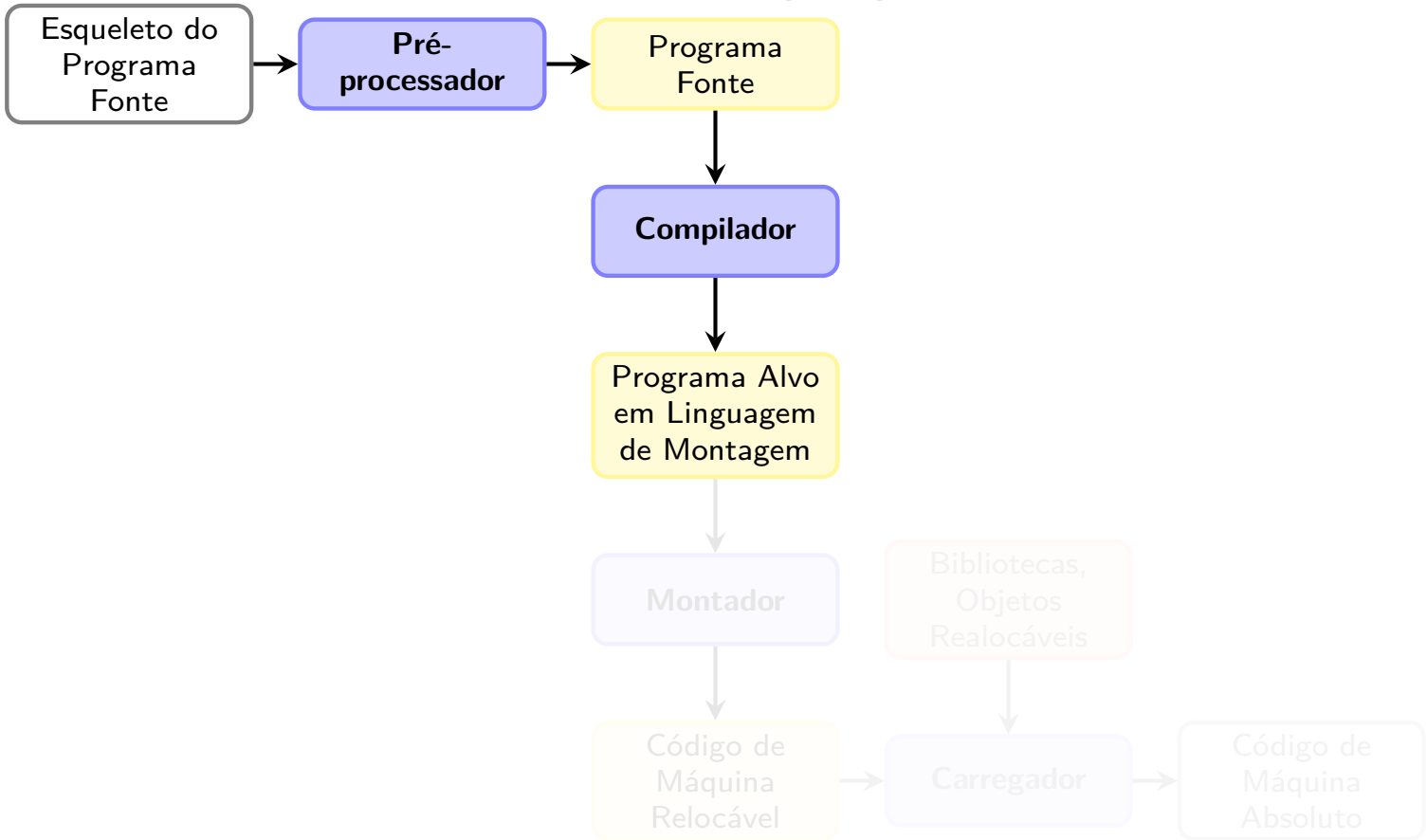
Introdução à Compilação

Sistema de Processamento de Linguagem



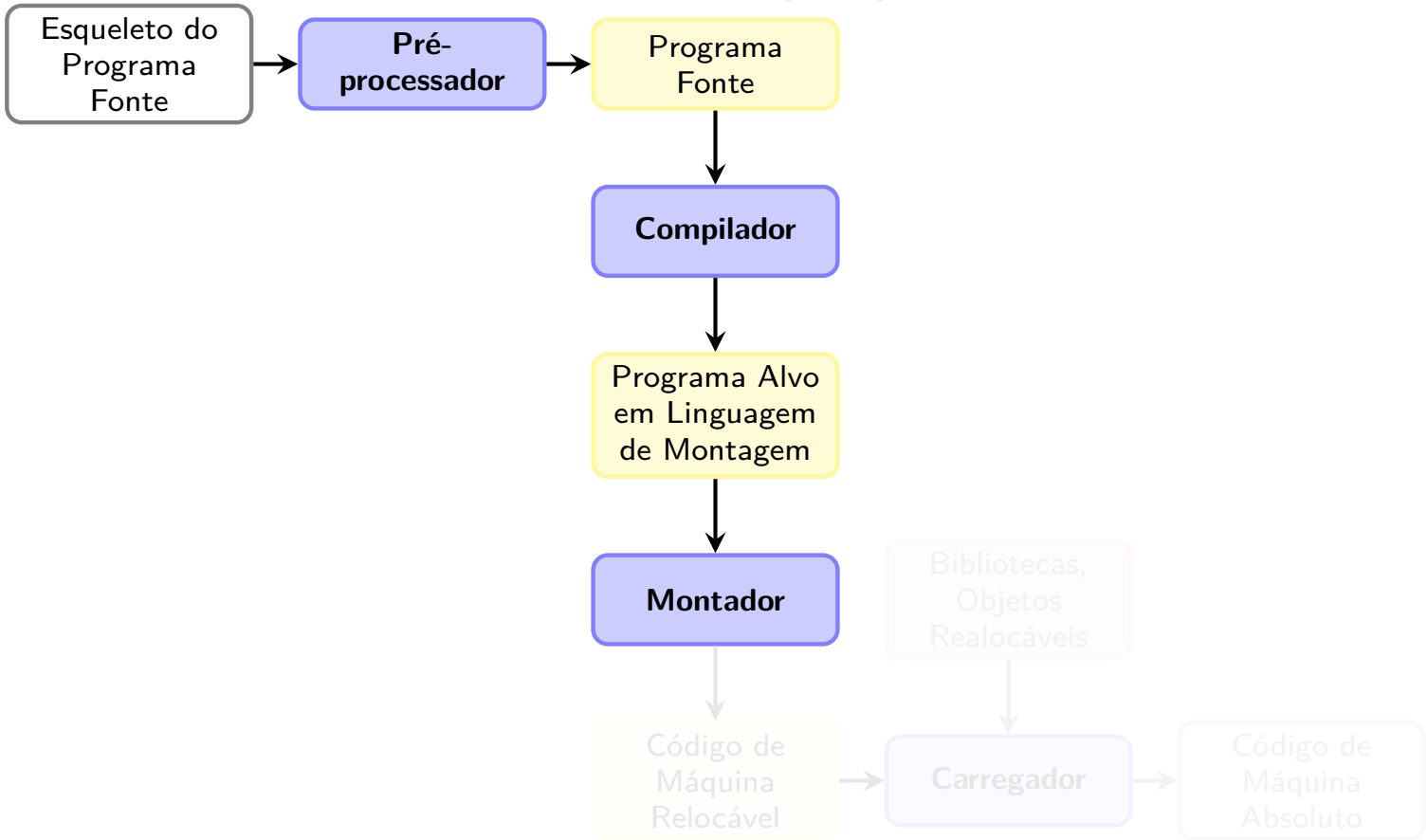
Introdução à Compilação

Sistema de Processamento de Linguagem



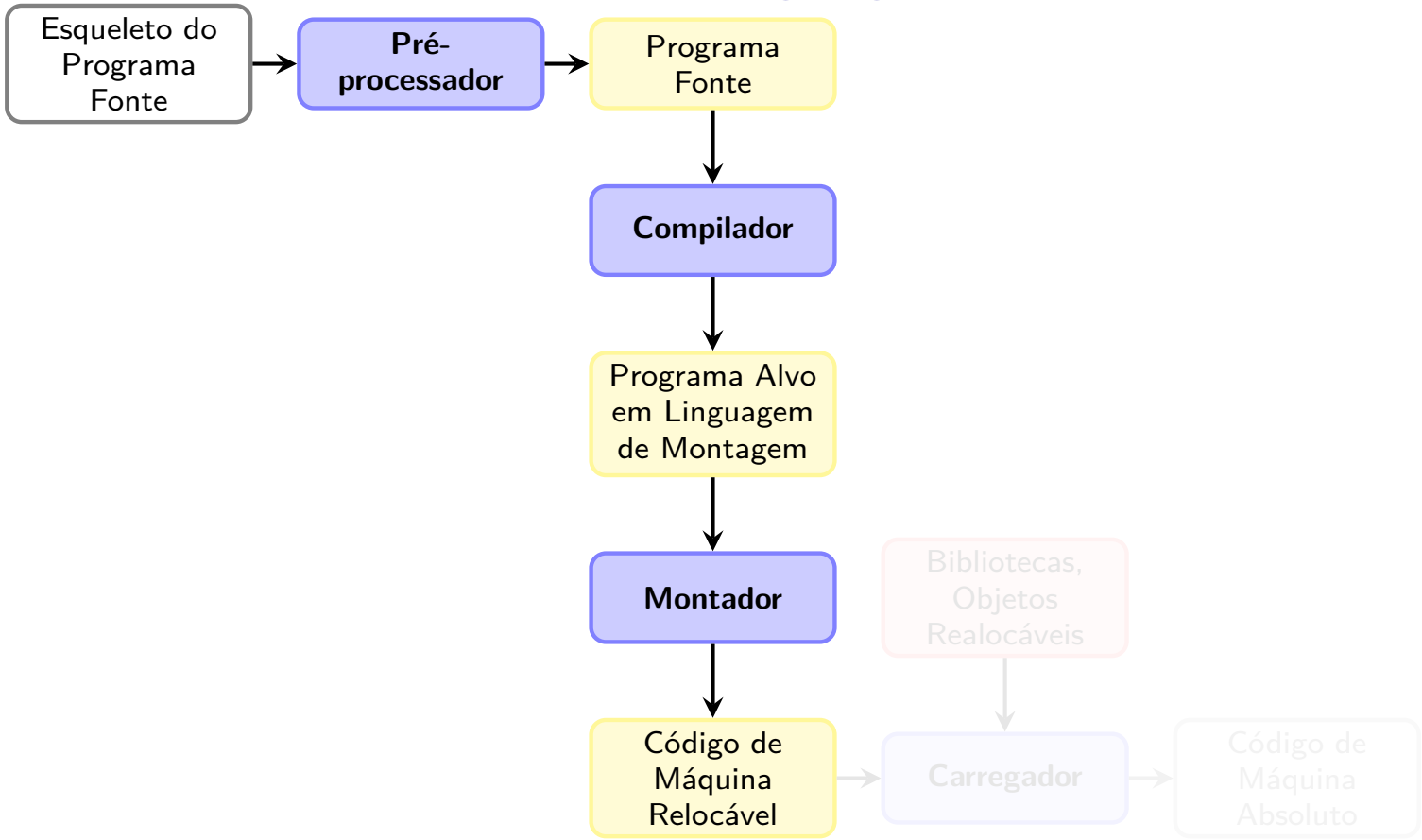
Introdução à Compilação

Sistema de Processamento de Linguagem



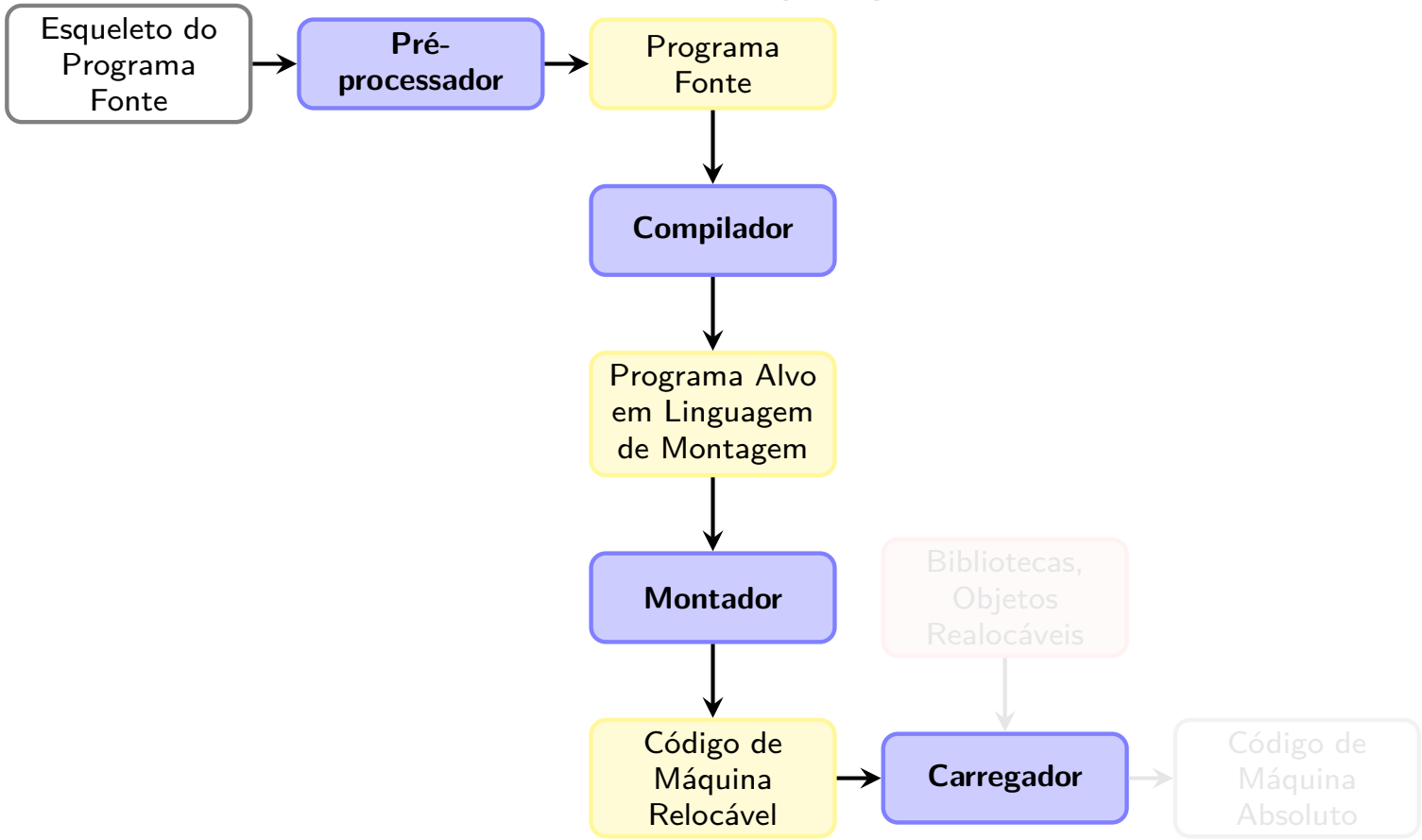
Introdução à Compilação

Sistema de Processamento de Linguagem



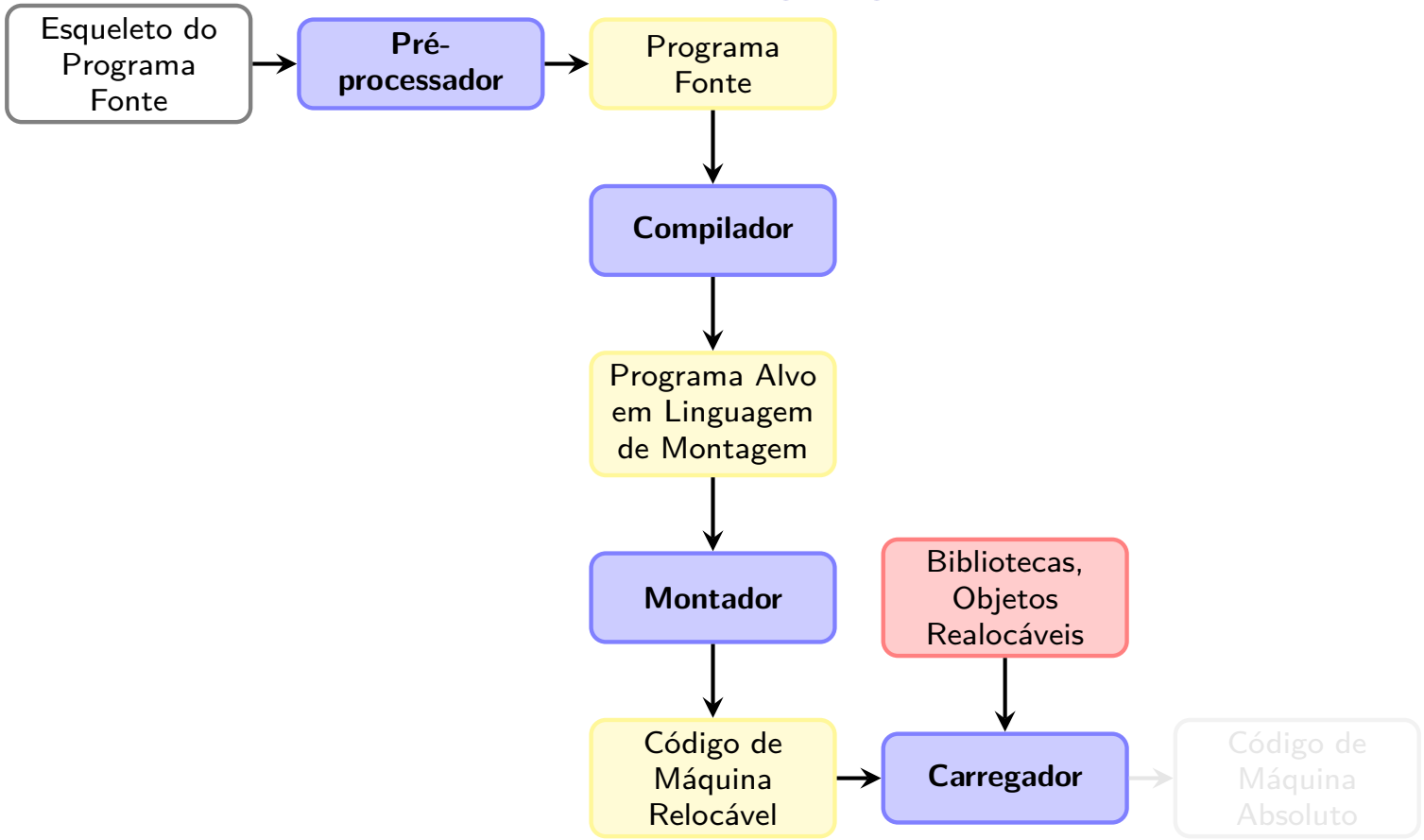
Introdução à Compilação

Sistema de Processamento de Linguagem



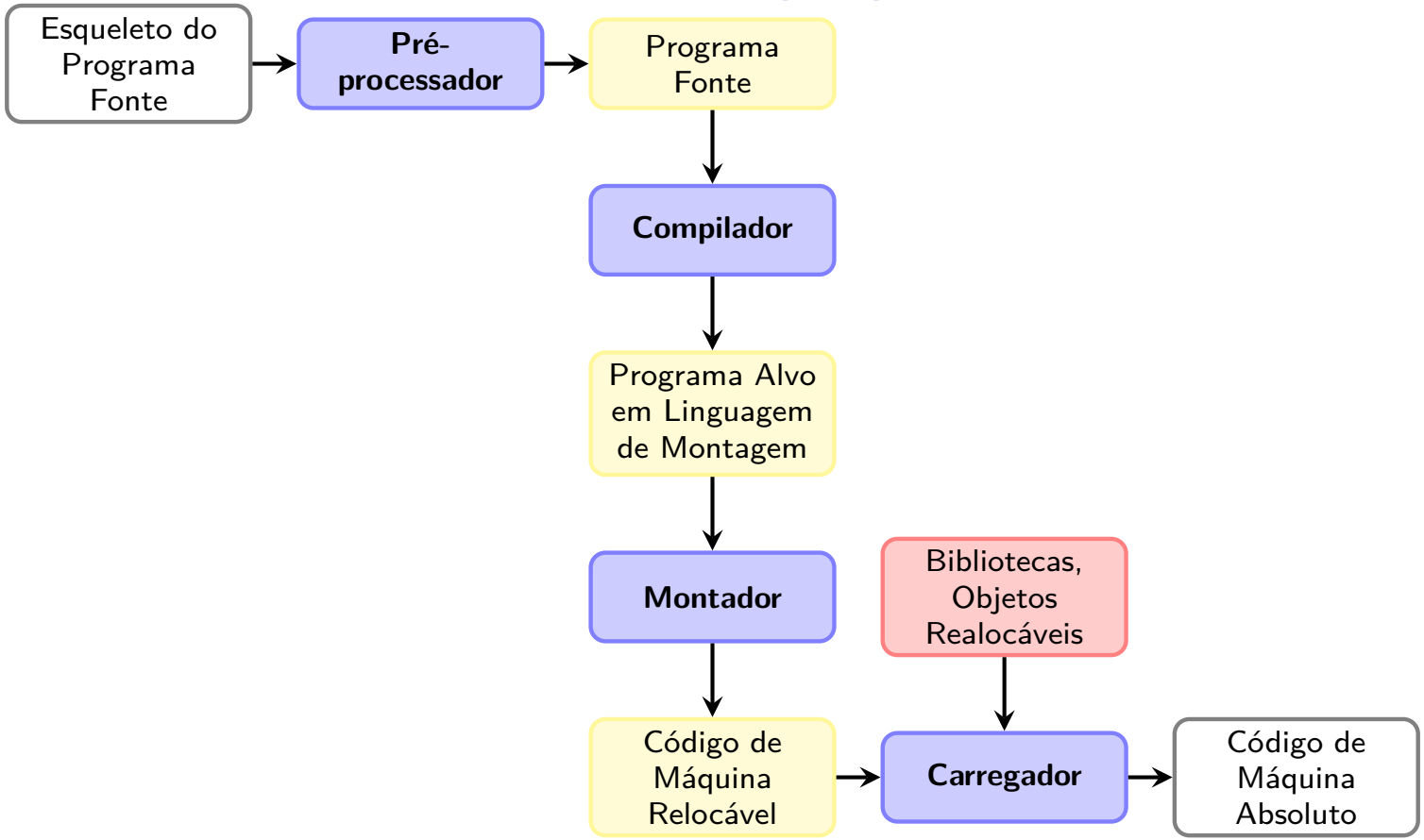
Introdução à Compilação

Sistema de Processamento de Linguagem



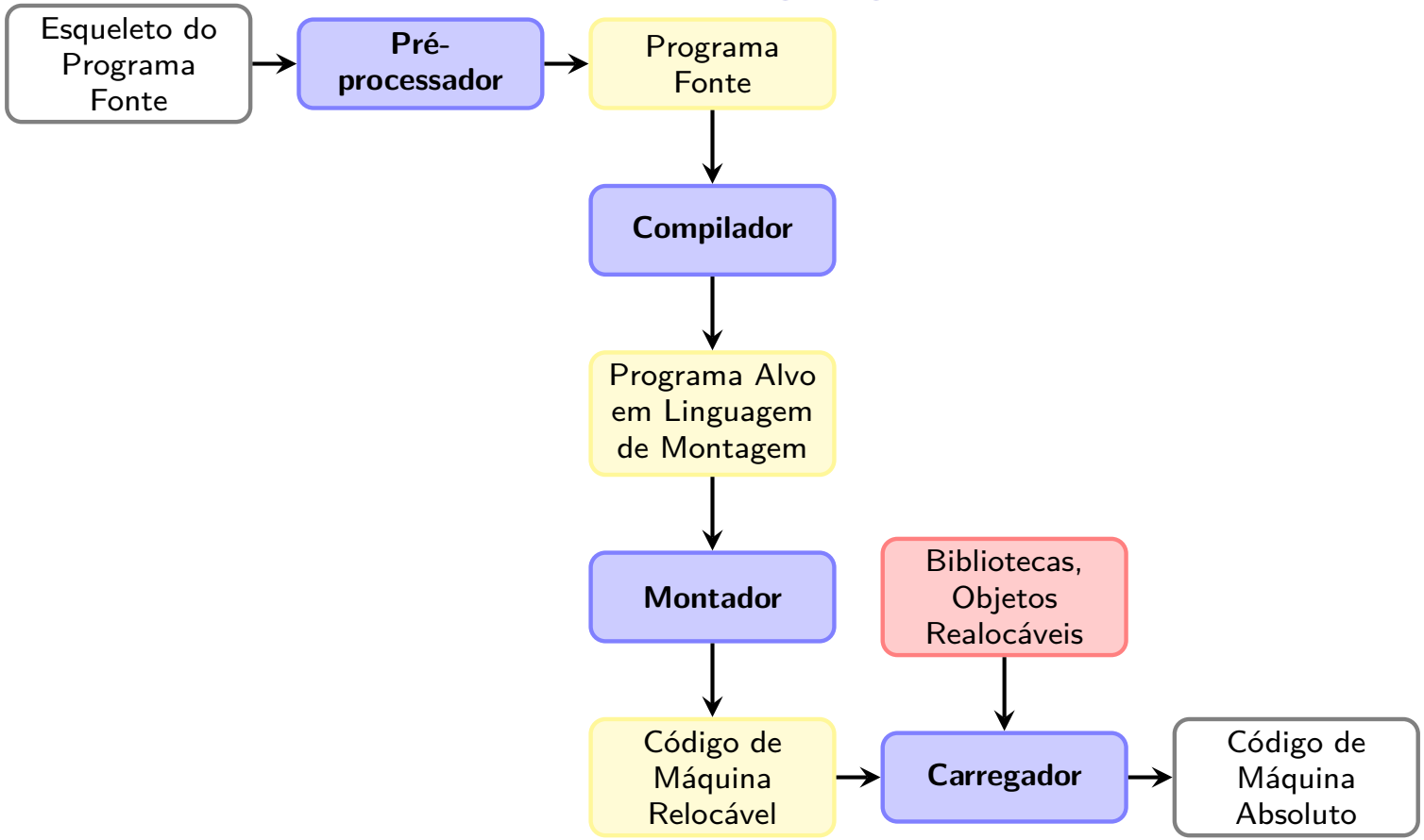
Introdução à Compilação

Sistema de Processamento de Linguagem



Introdução à Compilação

Sistema de Processamento de Linguagem



Introdução à Compilação

Modelo: Análise x Síntese

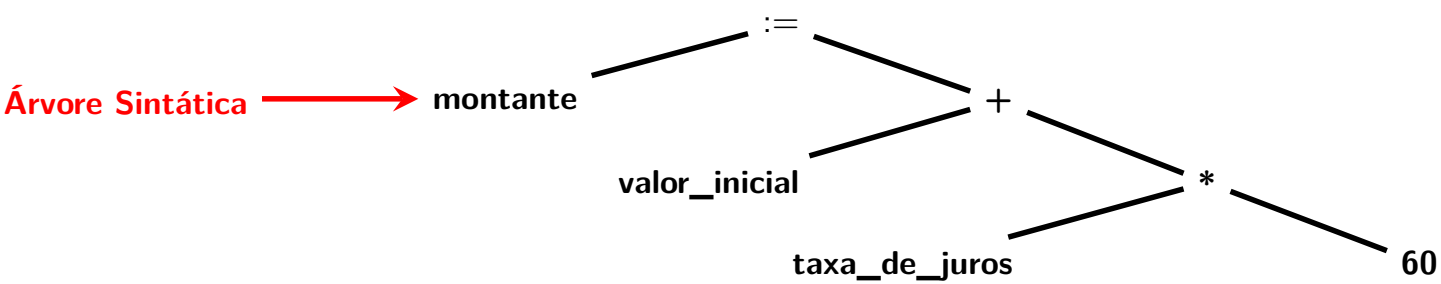
Processo de compilação é realizado em duas etapas: **Análise** × **Síntese**.

Análise

Etapa responsável por subdividir o programa fonte em diversas "construções" criando uma representação intermediária.

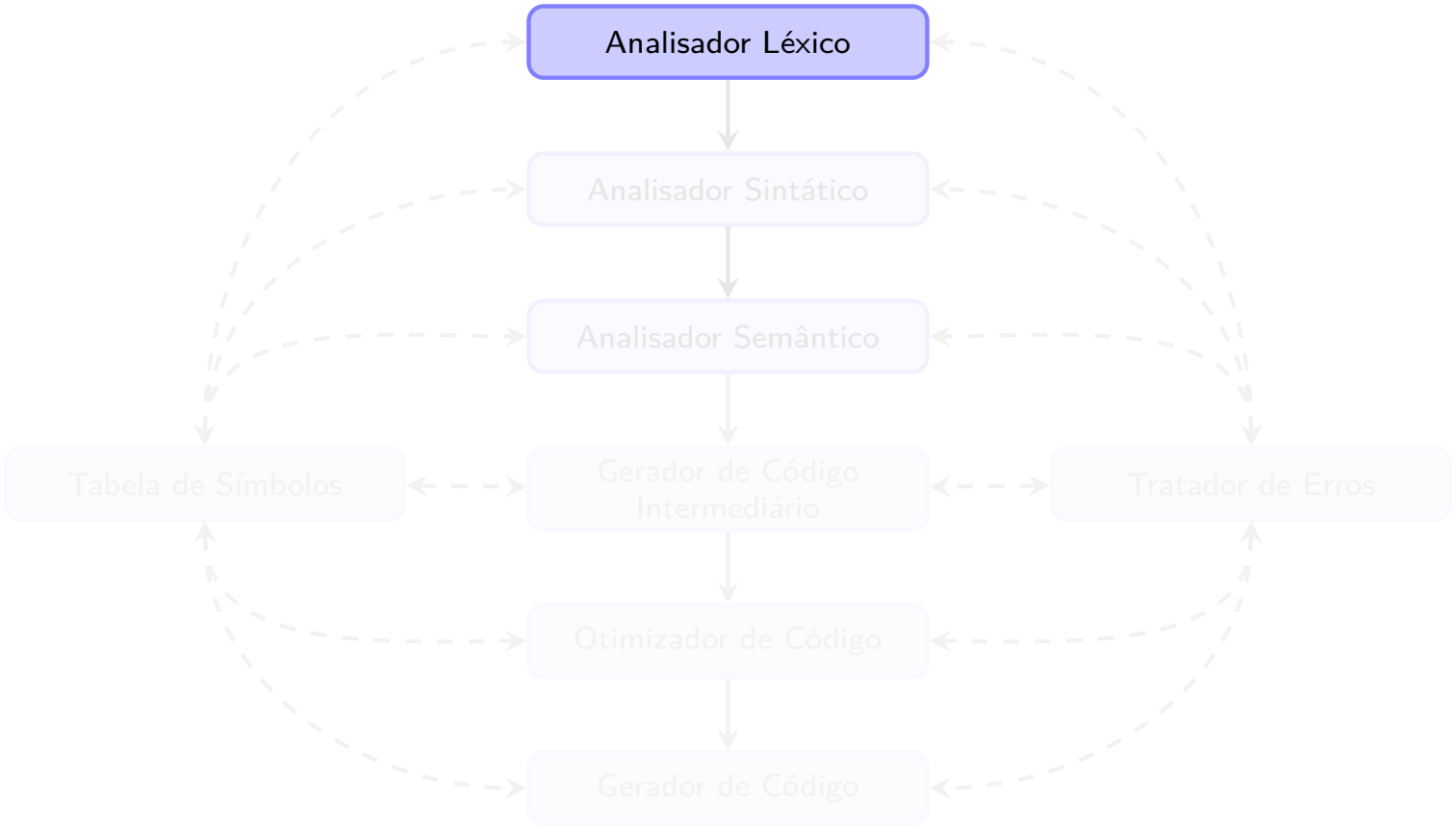
Síntese

Etapa responsável por gerar o programa destino (ou alvo) a partir de uma representação intermediária.



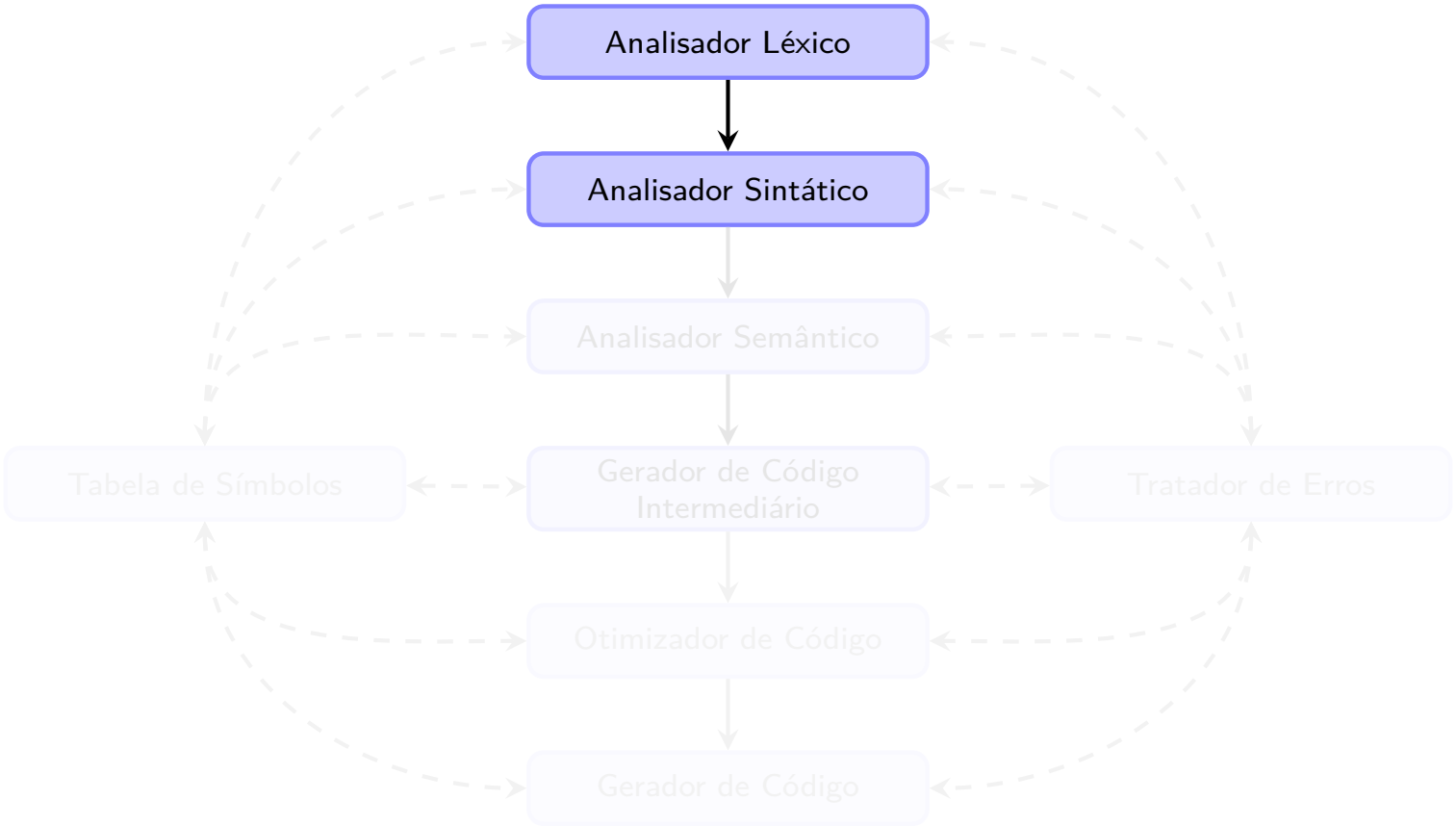
Introdução à Compilação

Fases do Modelo: Análise × Síntese



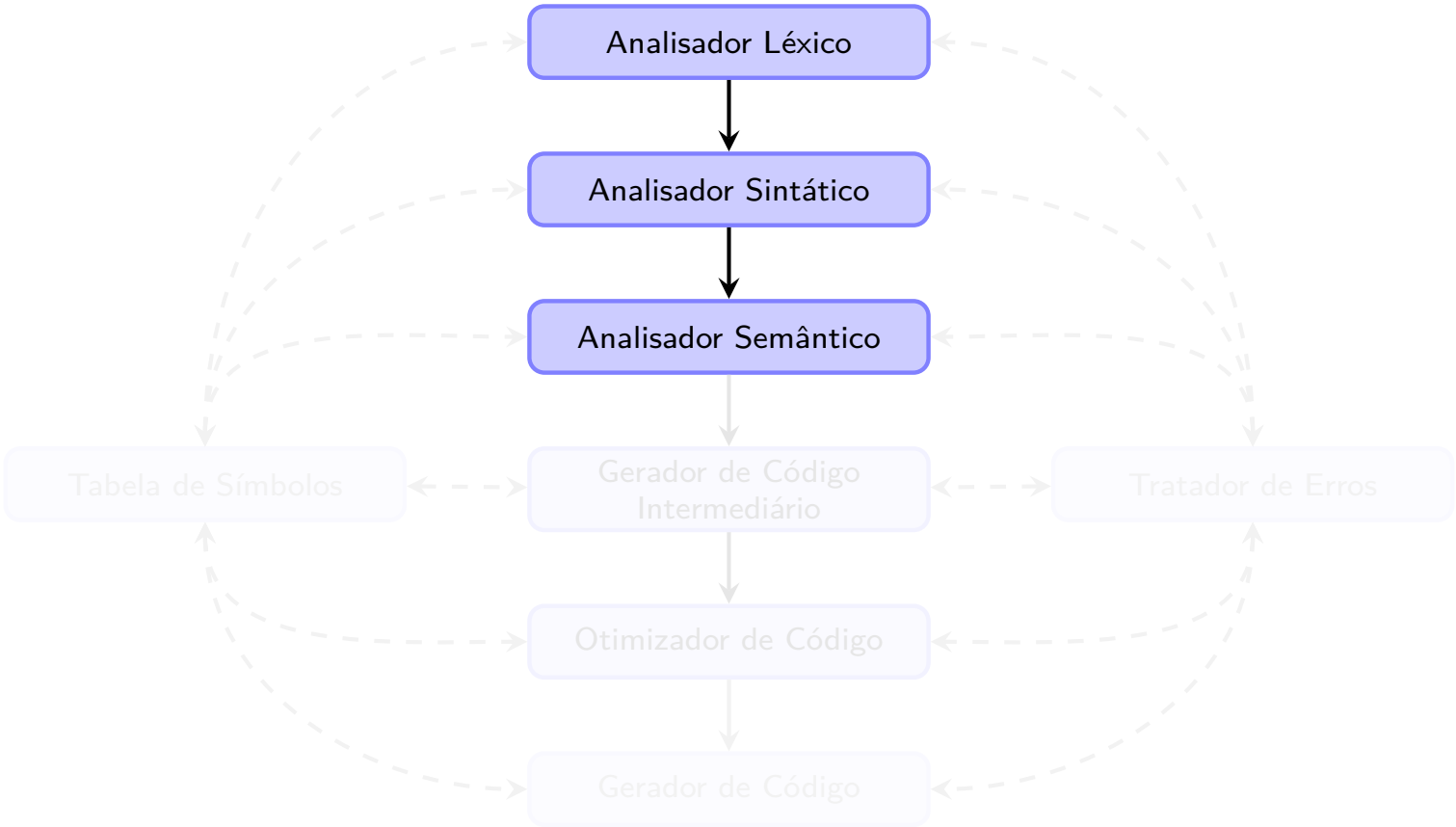
Introdução à Compilação

Fases do Modelo: Análise × Síntese



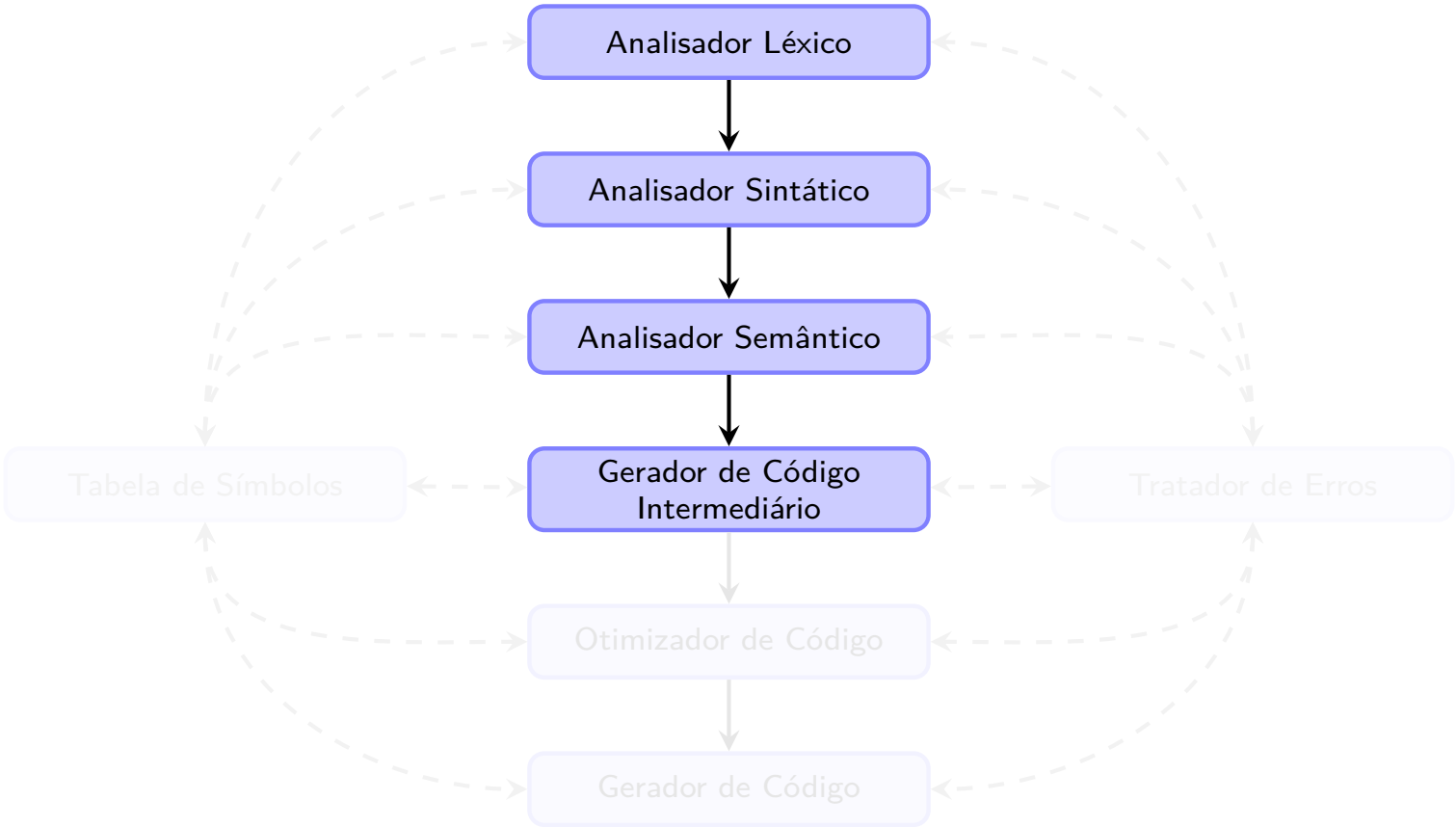
Introdução à Compilação

Fases do Modelo: Análise × Síntese



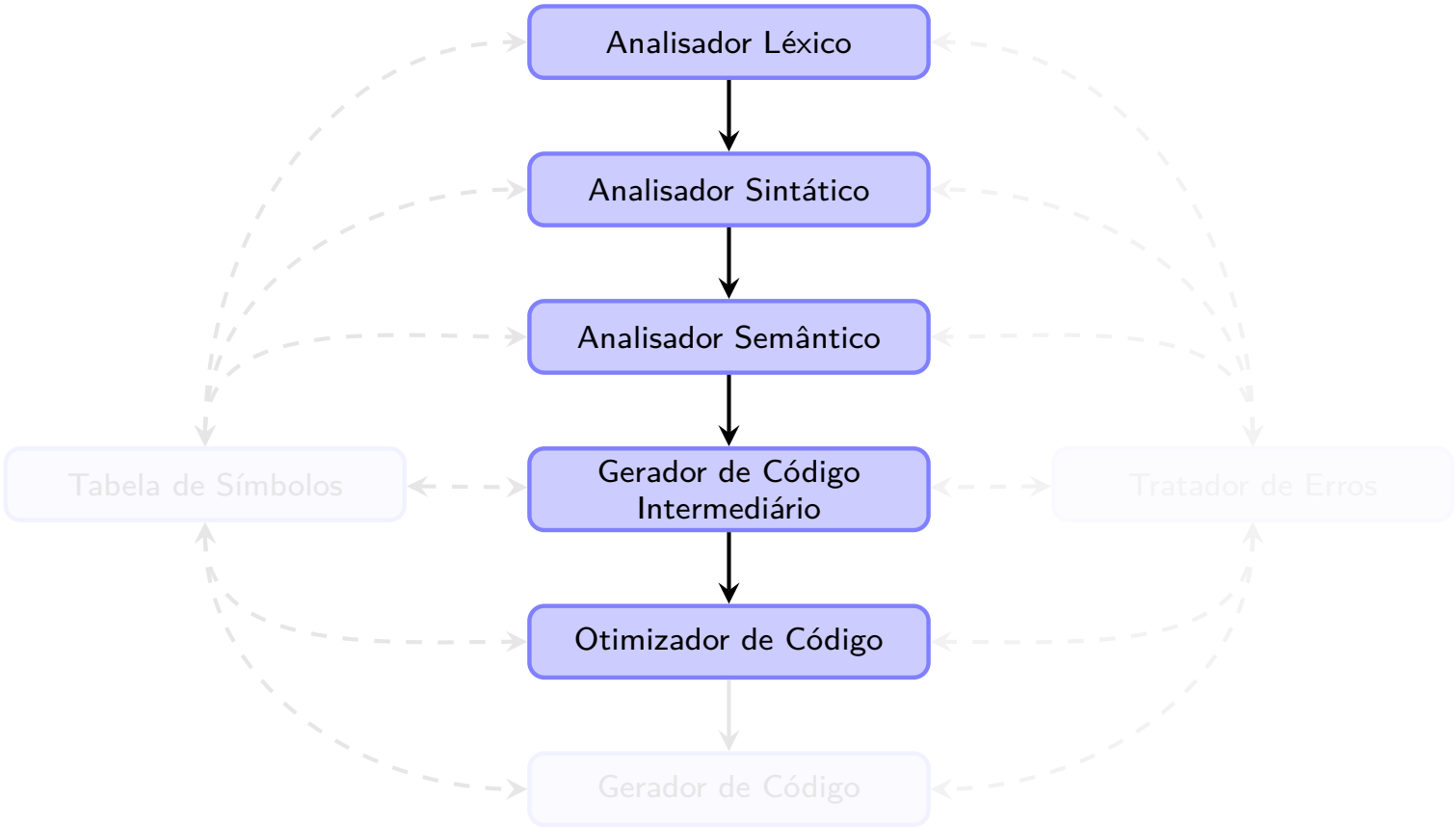
Introdução à Compilação

Fases do Modelo: Análise × Síntese



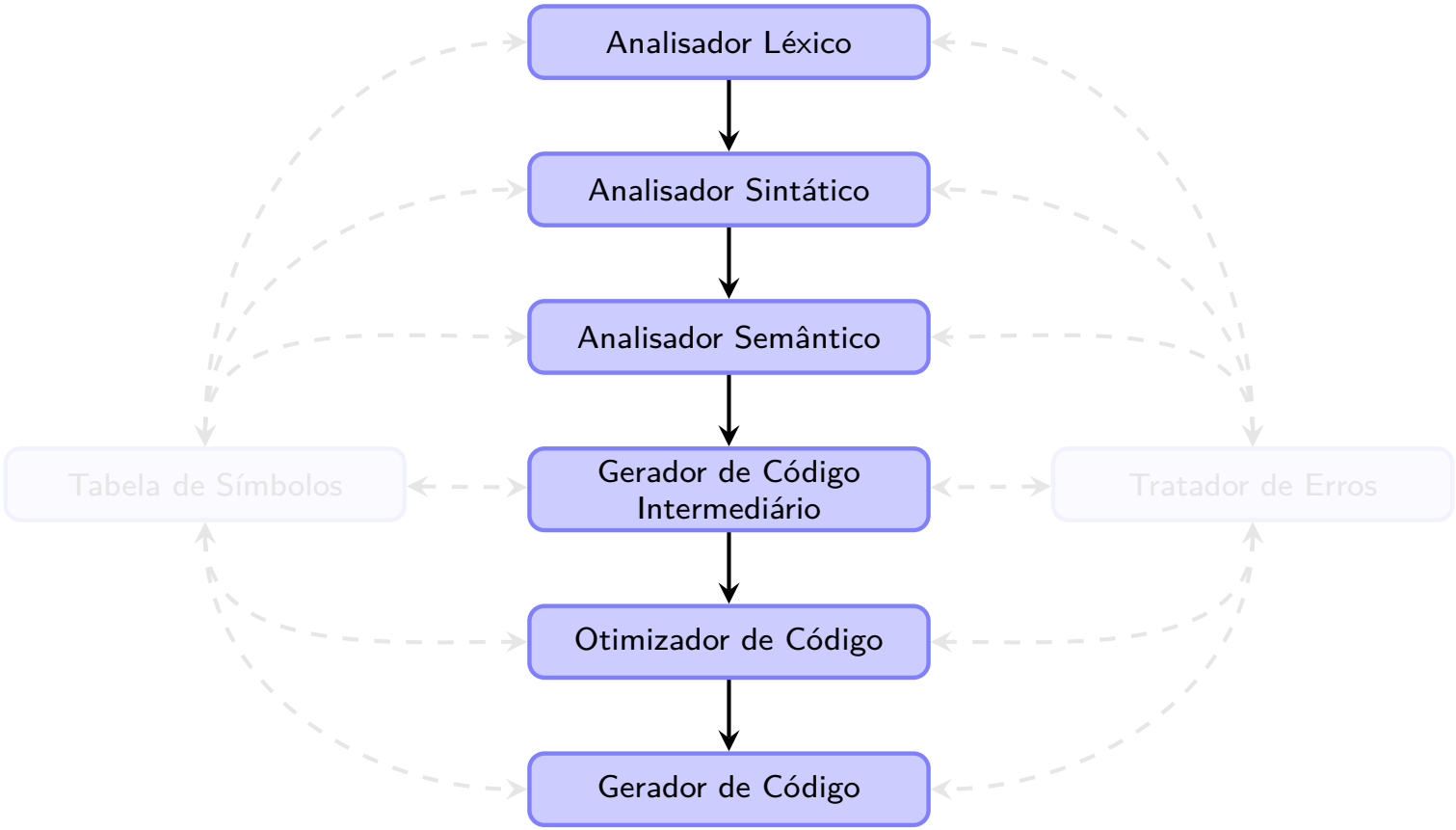
Introdução à Compilação

Fases do Modelo: Análise × Síntese



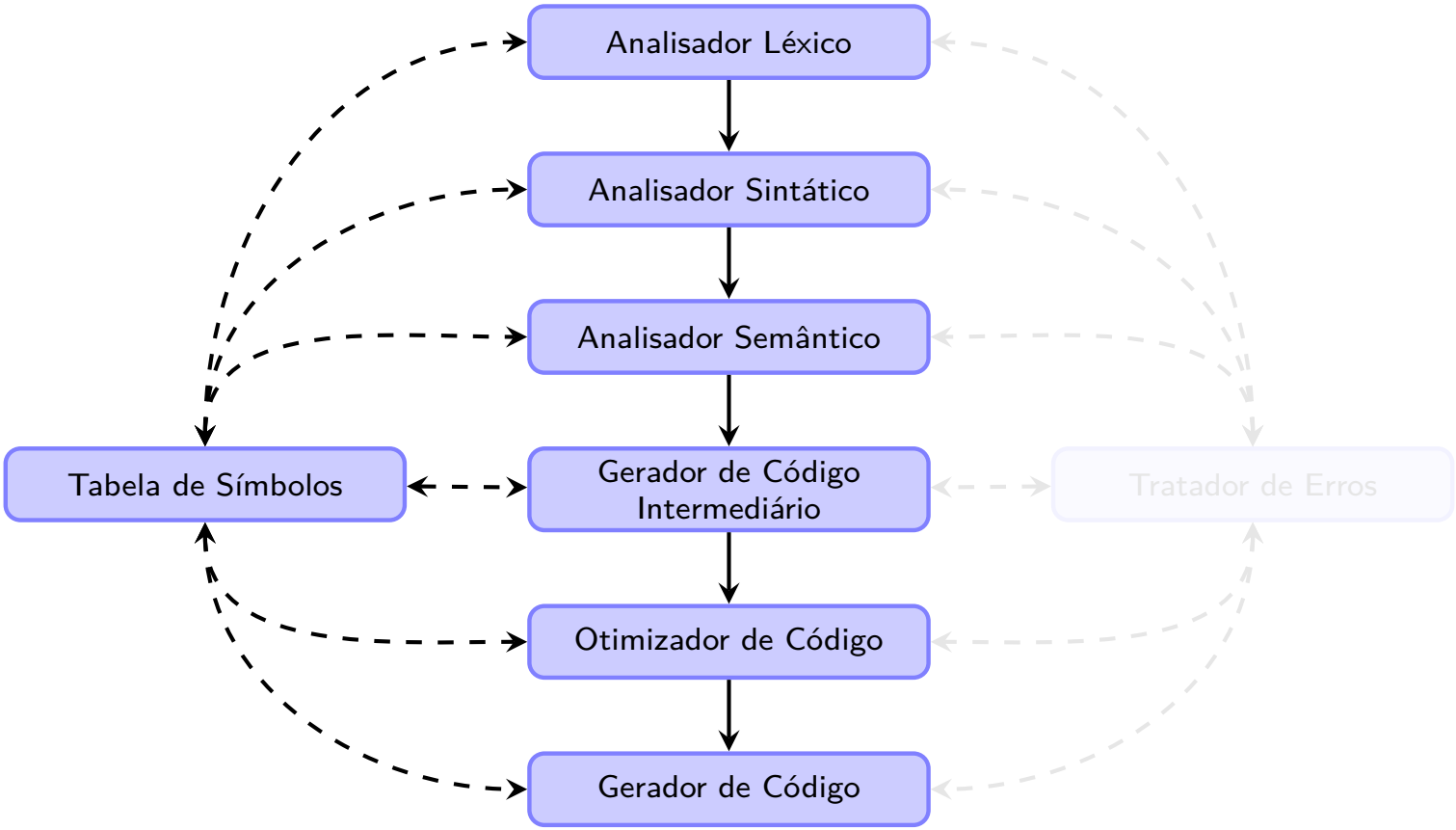
Introdução à Compilação

Fases do Modelo: Análise × Síntese



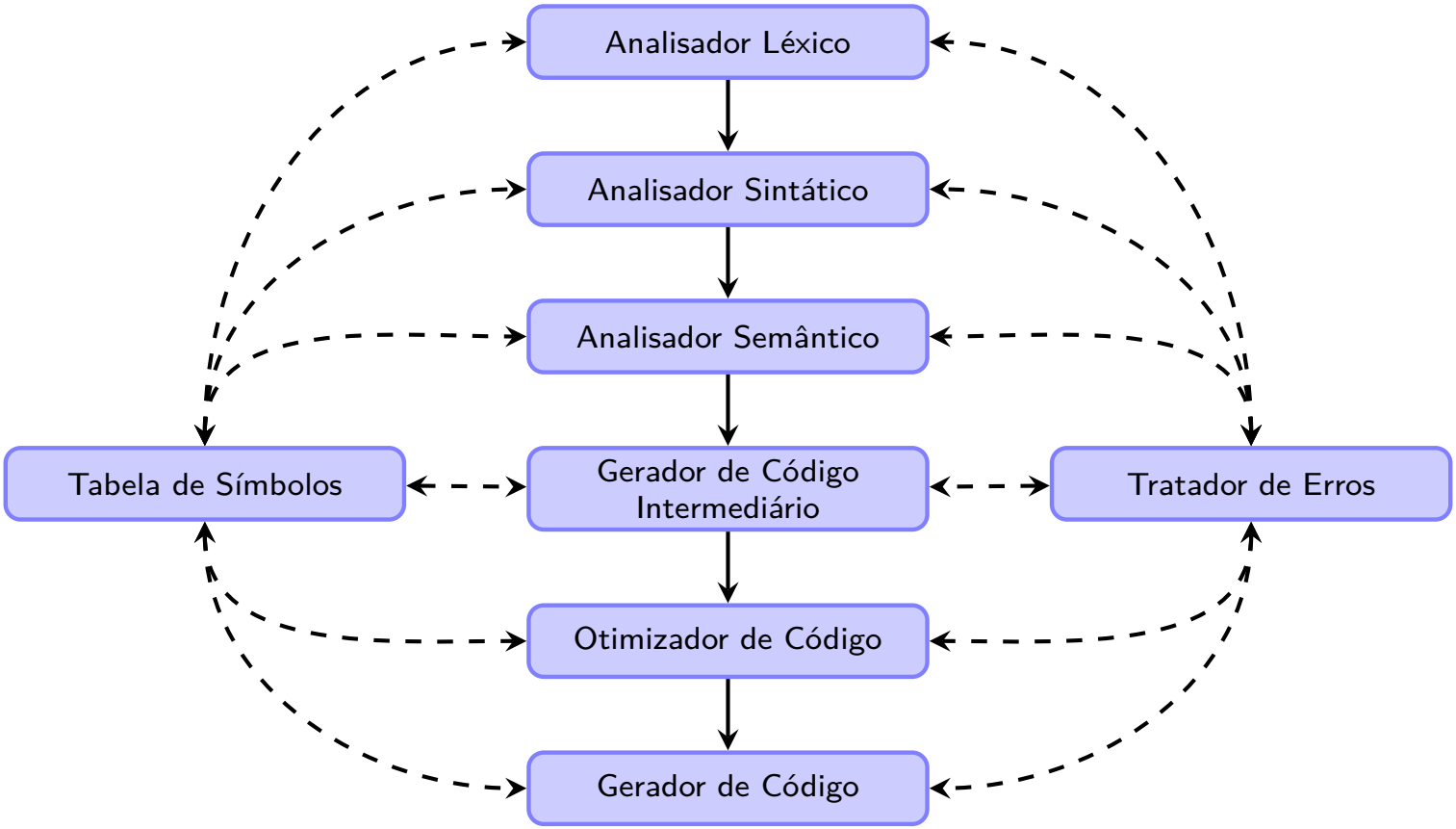
Introdução à Compilação

Fases do Modelo: Análise × Síntese



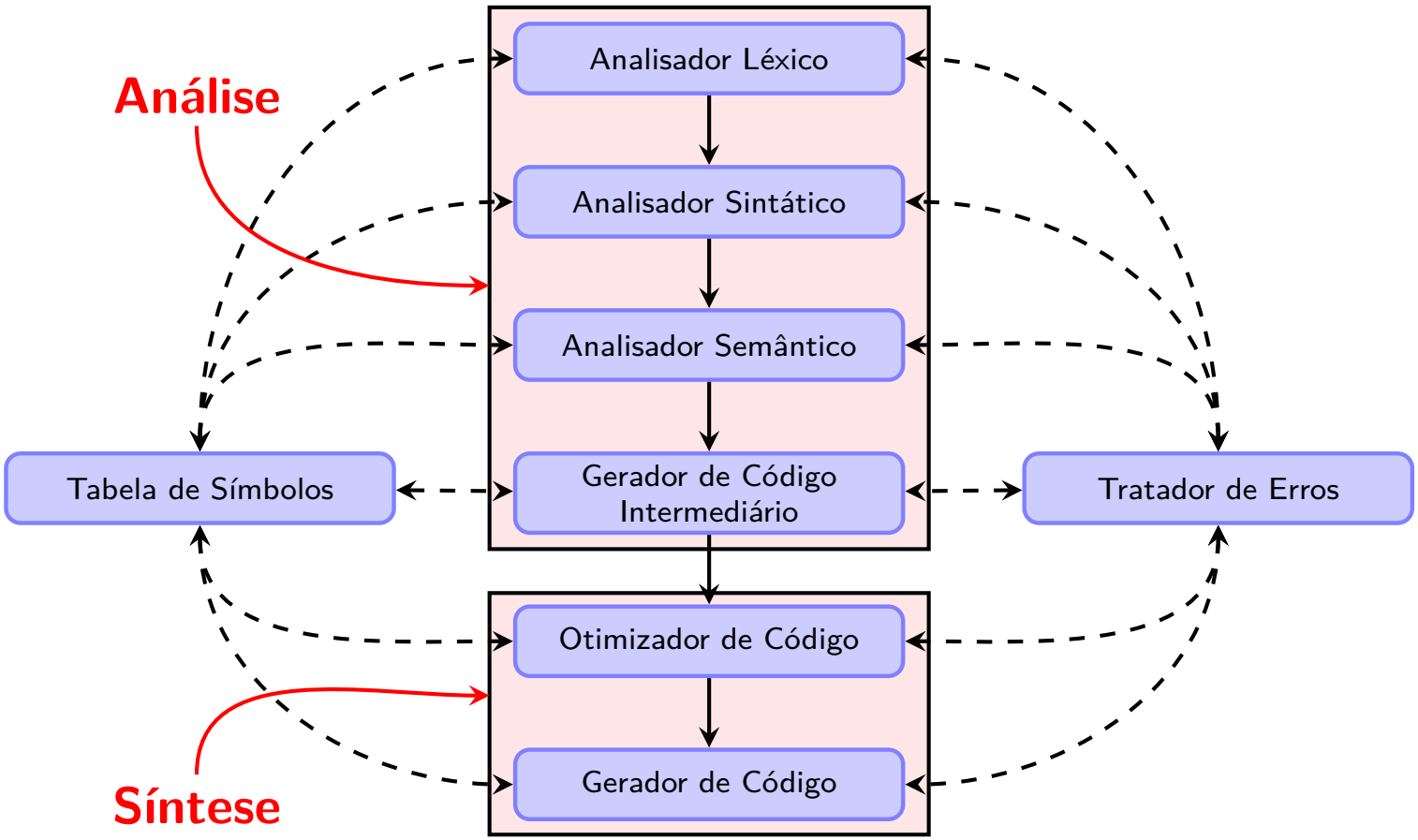
Introdução à Compilação

Fases do Modelo: Análise × Síntese



Introdução à Compilação

Fases do Modelo: Análise × Síntese



Análise

Análise Léxica (Análise Linear ou *Scanning*)

Análise léxica é responsável por agrupar as sequências de caracteres do programa fonte em *tokens* (unidades léxicas).

Token

Token é uma sequência de caracteres do programa fonte com um significado coletivo e representa uma classe de elementos léxicos, por exemplo: identificador, palavra-chave, operador, etc.

Lexema

Lexema é a sequência de caracteres que formam uma determinada *token*.

Análise

Exemplo de Análise Léxica

Os caracteres do seguinte enunciado:

montante := valor_inicial + taxa_de_juros * 60

poderiam ser agrupados nos seguintes *tokens*:

1. identificador **montante**
2. operador de atribuição **:=**
3. identificador **valor_inicial**
4. operador de adição **+**
5. identificador **taxa_de_juros**
6. operador de multiplicação *****
7. número inteiro **60**

Análise

Exemplo de Análise Léxica

Os caracteres do seguinte enunciado:

montante := valor_inicial + taxa_de_juros * 60

poderiam ser agrupados nos seguintes *tokens*:

1. identificador **montante**
2. operador de atribuição **:=**
3. identificador **valor_inicial**
4. operador de adição **+**
5. identificador **taxa_de_juros**
6. operador de multiplicação *****
7. número inteiro **60**

Análise

Exemplo de Análise Léxica

Os caracteres do seguinte enunciado:

montante := valor_inicial + taxa_de_juros * 60

poderiam ser agrupados nos seguintes *tokens*:

1. identificador **montante**
2. operador de atribuição **:=**
3. identificador **valor_inicial**
4. operador de adição **+**
5. identificador **taxa_de_juros**
6. operador de multiplicação *****
7. número inteiro **60**

Análise

Exemplo de Análise Léxica

Os caracteres do seguinte enunciado:

montante := valor_inicial + taxa_de_juros * 60

poderiam ser agrupados nos seguintes *tokens*:

1. identificador **montante**
2. operador de atribuição **:=**
3. identificador **valor_inicial**
4. operador de adição **+**
5. identificador **taxa_de_juros**
6. operador de multiplicação *
7. número inteiro **60**

Análise

Exemplo de Análise Léxica

Os caracteres do seguinte enunciado:

montante := valor_inicial + taxa_de_juros * 60

poderiam ser agrupados nos seguintes *tokens*:

1. identificador **montante**
2. operador de atribuição **:=**
3. identificador **valor_inicial**
4. operador de adição **+**
5. identificador **taxa_de_juros**
6. operador de multiplicação ***
7. número inteiro **60**

Análise

Exemplo de Análise Léxica

Os caracteres do seguinte enunciado:

montante := valor_inicial + taxa_de_juros * 60

poderiam ser agrupados nos seguintes *tokens*:

1. identificador **montante**
2. operador de atribuição **:=**
3. identificador **valor_inicial**
4. operador de adição **+**
5. identificador **taxa_de_juros**
6. operador de multiplicação *****
7. número inteiro **60**

Análise

Exemplo de Análise Léxica

Os caracteres do seguinte enunciado:

montante := valor_inicial + taxa_de_juros * 60

poderiam ser agrupados nos seguintes *tokens*:

1. identificador **montante**
2. operador de atribuição **:=**
3. identificador **valor_inicial**
4. operador de adição **+**
5. identificador **taxa_de_juros**
6. operador de multiplicação ***
7. número inteiro **60**

Análise

Exemplo de Análise Léxica

Os caracteres do seguinte enunciado:

montante := valor_inicial + taxa_de_juros * 60

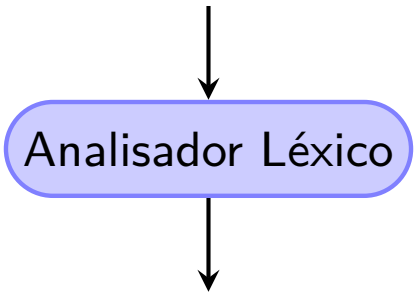
poderiam ser agrupados nos seguintes *tokens*:

1. identificador **montante**
2. operador de atribuição **:=**
3. identificador **valor_inicial**
4. operador de adição **+**
5. identificador **taxa_de_juros**
6. operador de multiplicação *****
7. número inteiro **60**

Análise

Exemplo de Análise Léxica

montante := valor_inicial + taxa_de_juros * 60



$\mathbf{id_1\ op_{:=}\ id_2\ op_{+}\ id_3\ op_{*}\ num_{60}}$

OU

$\mathbf{id_1 := id_2 + id_3 * 60}$

Análise

Análise Sintática (Análise Hierárquica ou *Parsing*)

Análise sintática é responsável por agrupar os *tokens* do programa fonte em frases gramaticais.

O agrupamento é feito de forma hierárquica e representado geralmente por uma árvore gramatical.

Árvore Gramatical (Árvore de Derivação ou *Parse Tree*)

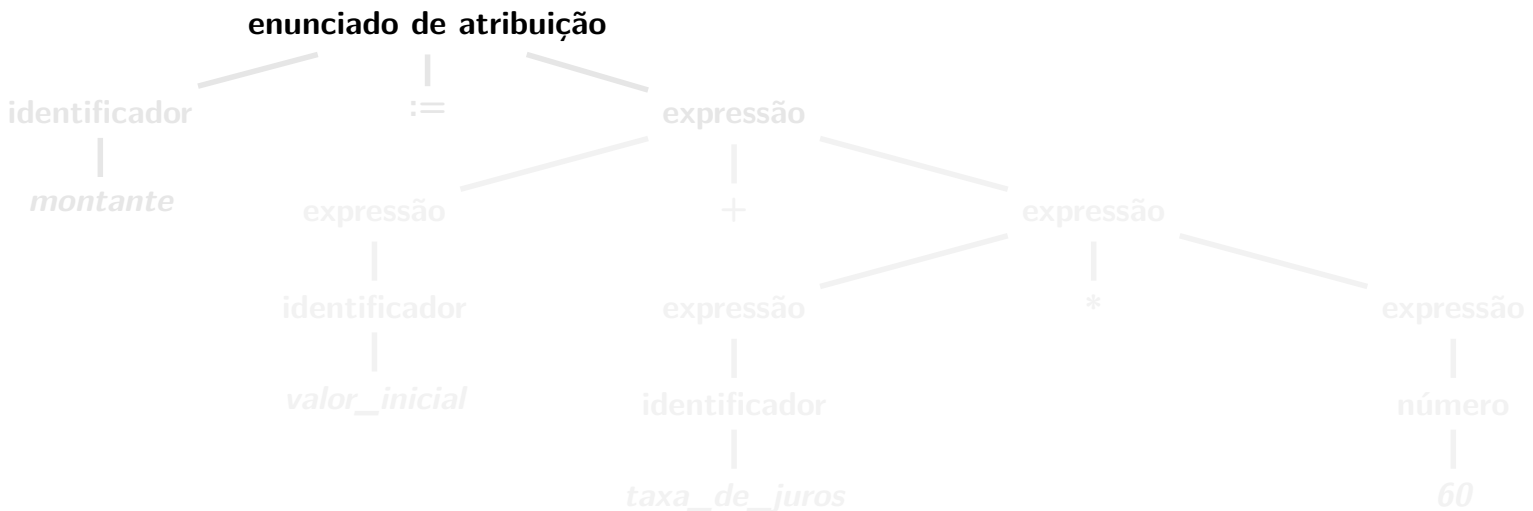
Análise

Análise Sintática (Análise Hierárquica ou *Parsing*)

Análise sintática é responsável por agrupar os *tokens* do programa fonte em frases gramaticais.

O agrupamento é feito de forma hierárquica e representado geralmente por uma árvore gramatical.

Árvore Gramatical (Árvore de Derivação ou *Parse Tree*)



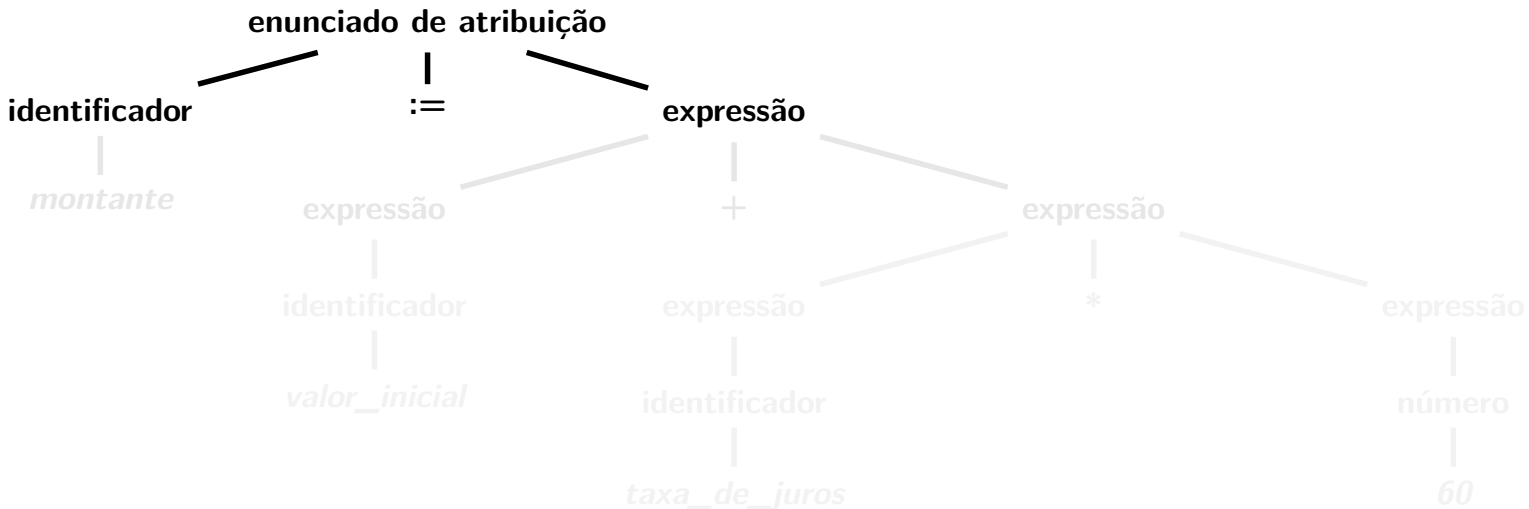
Análise

Análise Sintática (Análise Hierárquica ou *Parsing*)

Análise sintática é responsável por agrupar os *tokens* do programa fonte em frases gramaticais.

O agrupamento é feito de forma hierárquica e representado geralmente por uma árvore gramatical.

Árvore Gramatical (Árvore de Derivação ou *Parse Tree*)



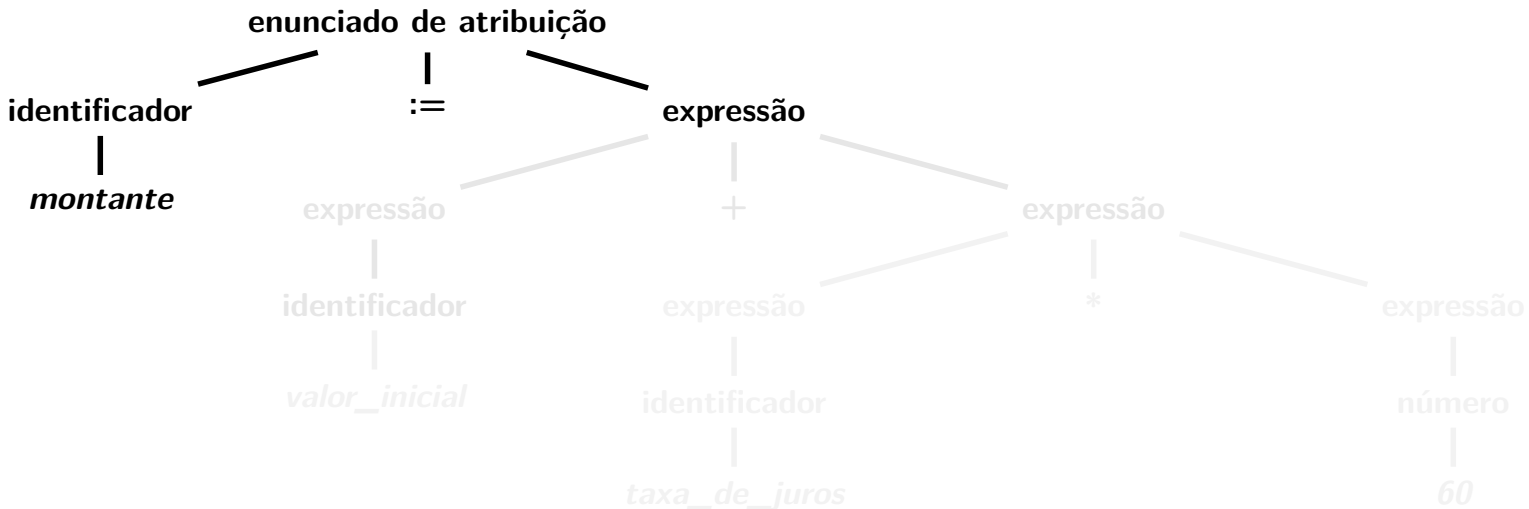
Análise

Análise Sintática (Análise Hierárquica ou *Parsing*)

Análise sintática é responsável por agrupar os *tokens* do programa fonte em frases gramaticais.

O agrupamento é feito de forma hierárquica e representado geralmente por uma árvore gramatical.

Árvore Gramatical (Árvore de Derivação ou *Parse Tree*)



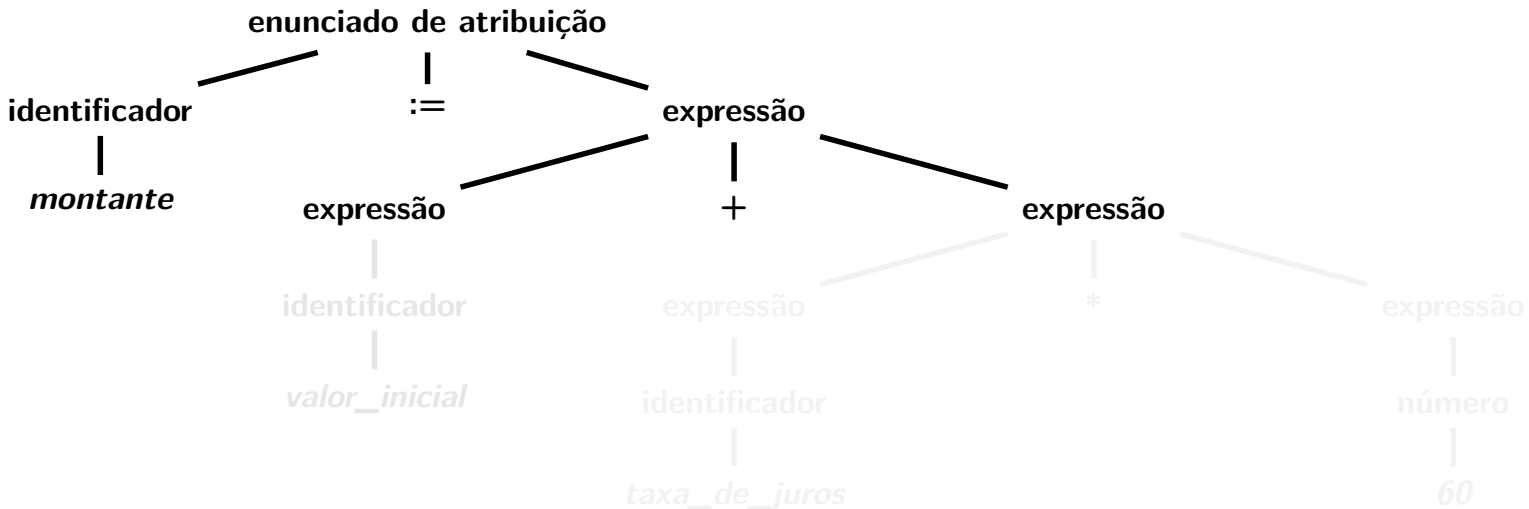
Análise

Análise Sintática (Análise Hierárquica ou *Parsing*)

Análise sintática é responsável por agrupar os *tokens* do programa fonte em frases gramaticais.

O agrupamento é feito de forma hierárquica e representado geralmente por uma árvore gramatical.

Árvore Gramatical (Árvore de Derivação ou *Parse Tree*)



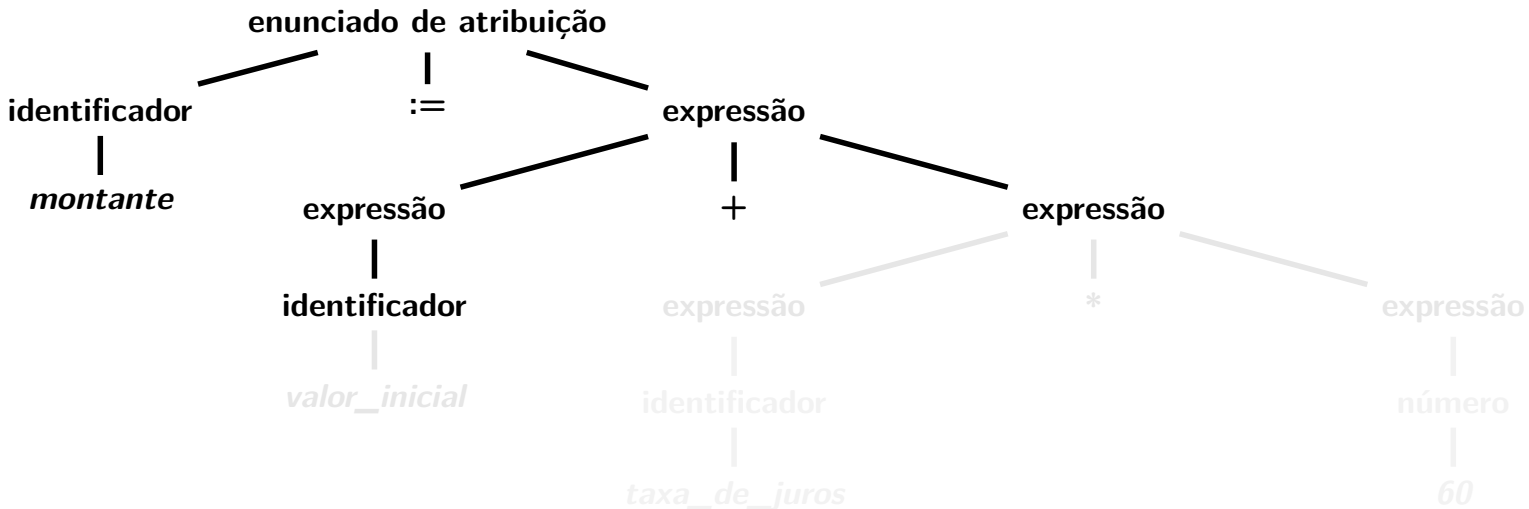
Análise

Análise Sintática (Análise Hierárquica ou *Parsing*)

Análise sintática é responsável por agrupar os *tokens* do programa fonte em frases gramaticais.

O agrupamento é feito de forma hierárquica e representado geralmente por uma árvore gramatical.

Árvore Gramatical (Árvore de Derivação ou *Parse Tree*)



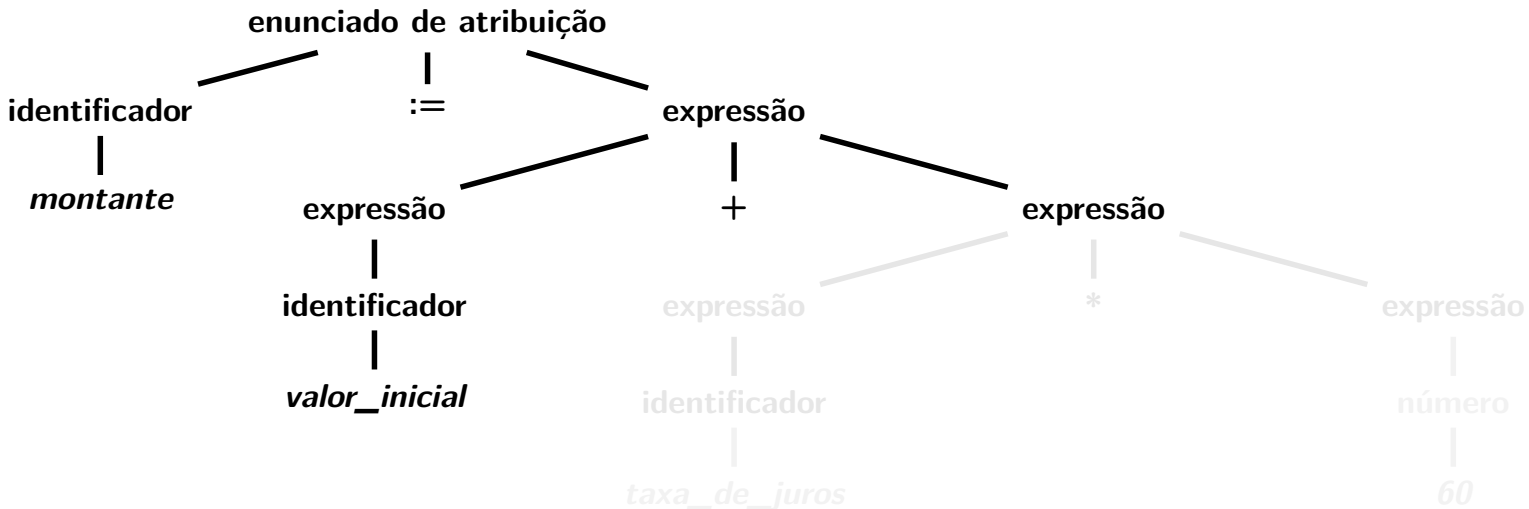
Análise

Análise Sintática (Análise Hierárquica ou *Parsing*)

Análise sintática é responsável por agrupar os *tokens* do programa fonte em frases gramaticais.

O agrupamento é feito de forma hierárquica e representado geralmente por uma árvore gramatical.

Árvore Gramatical (Árvore de Derivação ou *Parse Tree*)



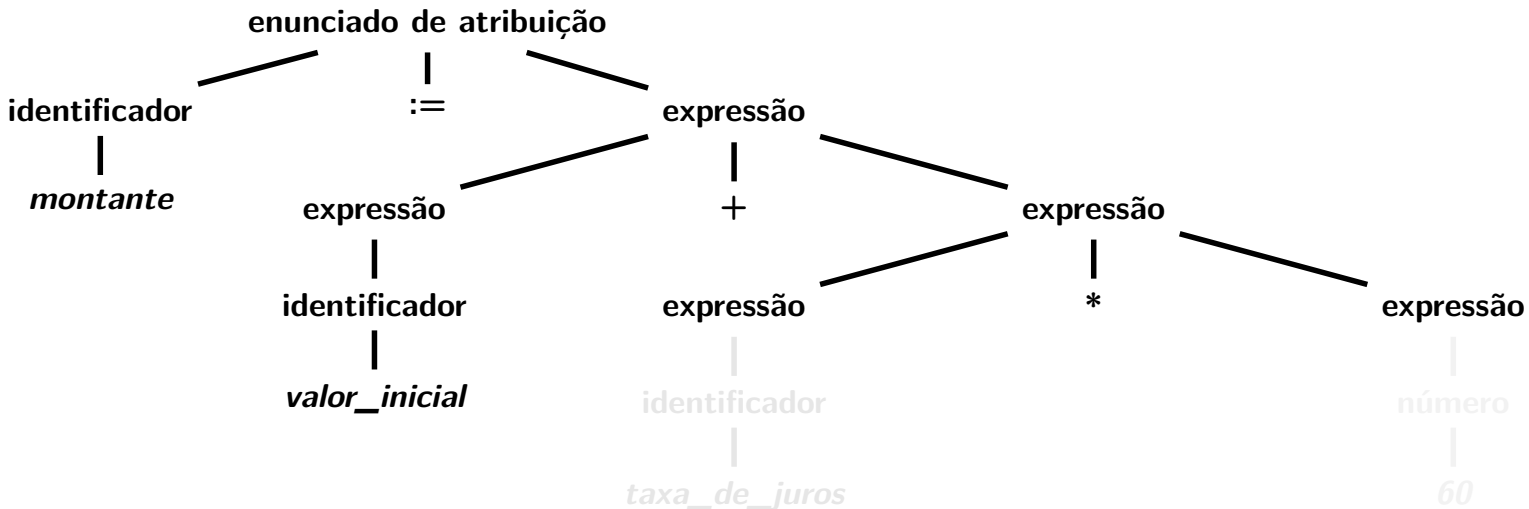
Análise

Análise Sintática (Análise Hierárquica ou *Parsing*)

Análise sintática é responsável por agrupar os *tokens* do programa fonte em frases gramaticais.

O agrupamento é feito de forma hierárquica e representado geralmente por uma árvore gramatical.

Árvore Gramatical (Árvore de Derivação ou *Parse Tree*)



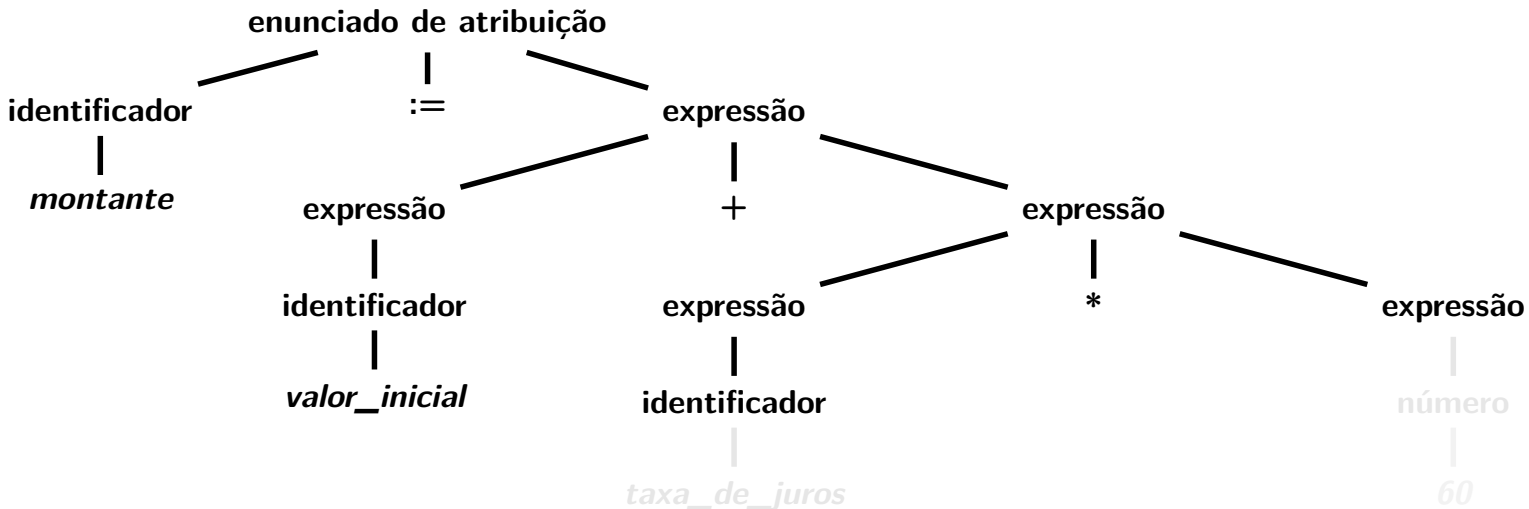
Análise

Análise Sintática (Análise Hierárquica ou *Parsing*)

Análise sintática é responsável por agrupar os *tokens* do programa fonte em frases gramaticais.

O agrupamento é feito de forma hierárquica e representado geralmente por uma árvore gramatical.

Árvore Gramatical (Árvore de Derivação ou *Parse Tree*)



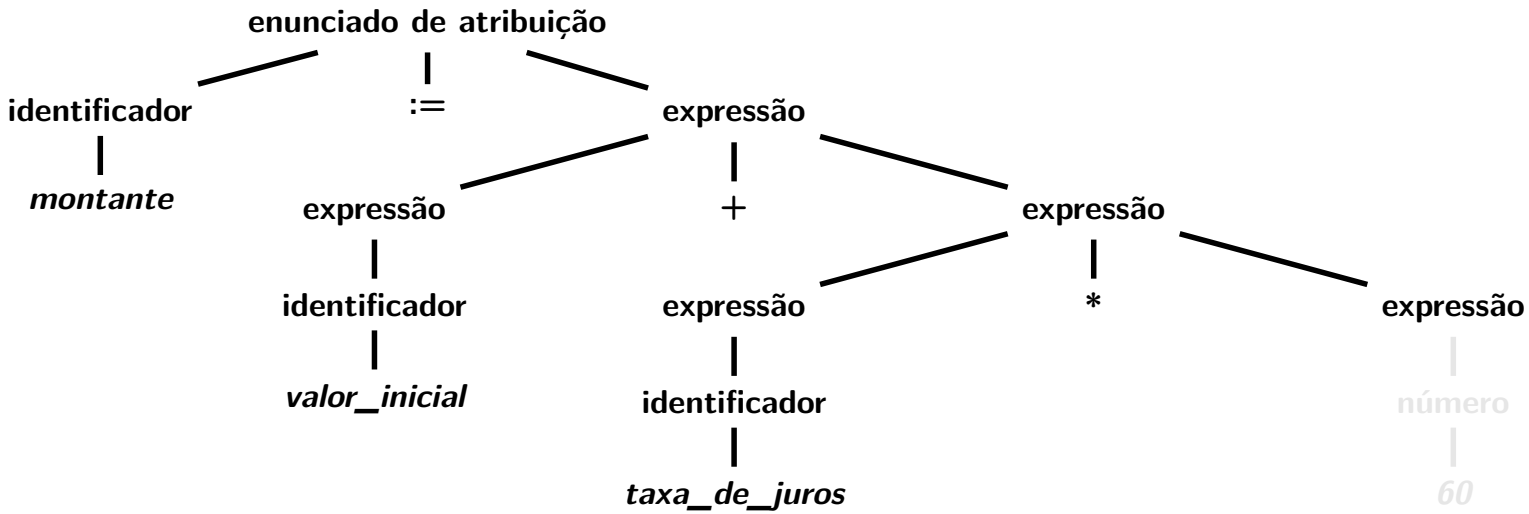
Análise

Análise Sintática (Análise Hierárquica ou *Parsing*)

Análise sintática é responsável por agrupar os *tokens* do programa fonte em frases gramaticais.

O agrupamento é feito de forma hierárquica e representado geralmente por uma árvore gramatical.

Árvore Gramatical (Árvore de Derivação ou *Parse Tree*)



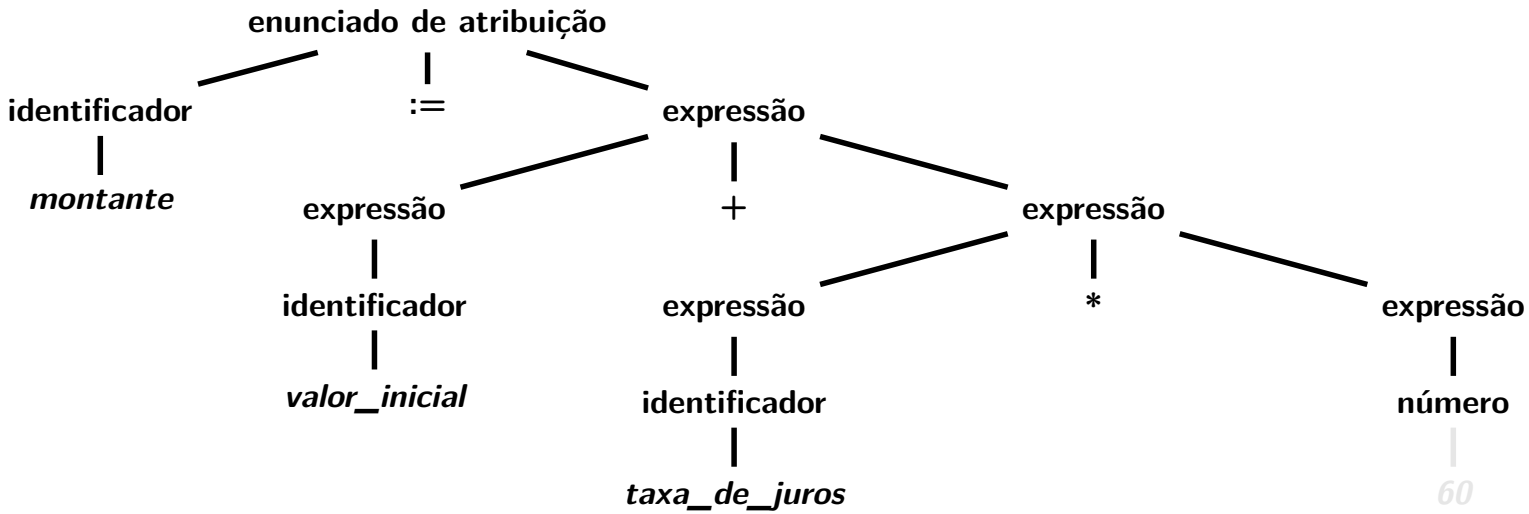
Análise

Análise Sintática (Análise Hierárquica ou *Parsing*)

Análise sintática é responsável por agrupar os *tokens* do programa fonte em frases gramaticais.

O agrupamento é feito de forma hierárquica e representado geralmente por uma árvore gramatical.

Árvore Gramatical (Árvore de Derivação ou *Parse Tree*)



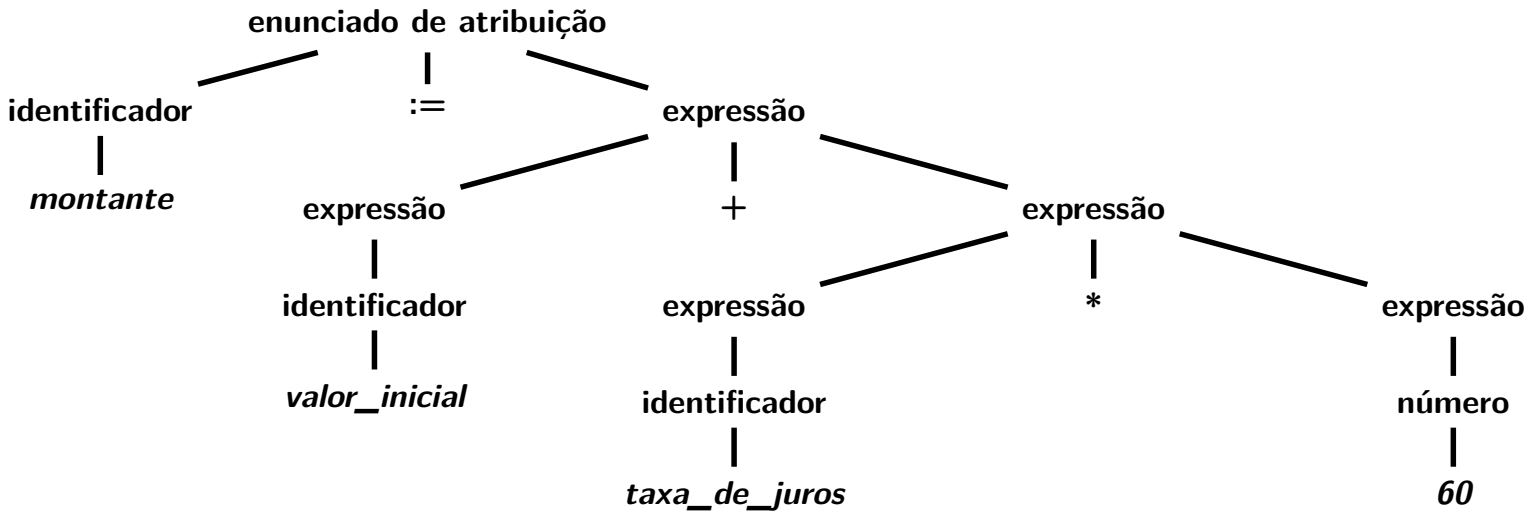
Análise

Análise Sintática (Análise Hierárquica ou *Parsing*)

Análise sintática é responsável por agrupar os *tokens* do programa fonte em frases gramaticais.

O agrupamento é feito de forma hierárquica e representado geralmente por uma árvore gramatical.

Árvore Gramatical (Árvore de Derivação ou *Parse Tree*)



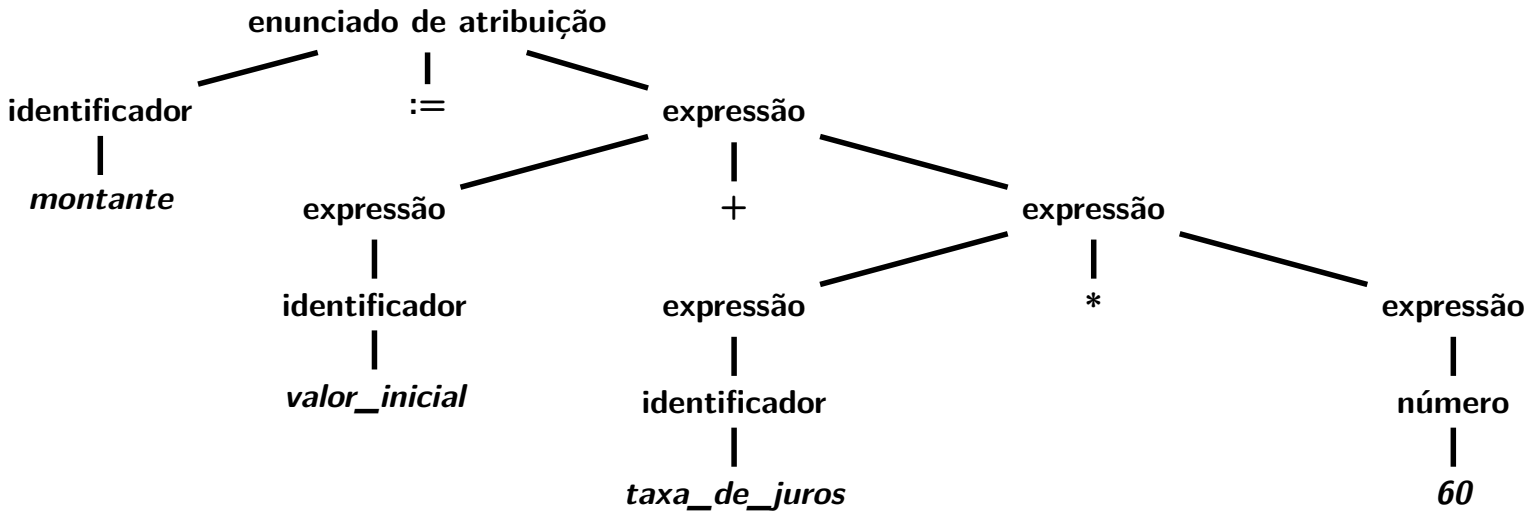
Análise

Análise Sintática (Análise Hierárquica ou *Parsing*)

Análise sintática é responsável por agrupar os *tokens* do programa fonte em frases gramaticais.

O agrupamento é feito de forma hierárquica e representado geralmente por uma árvore gramatical.

Árvore Gramatical (Árvore de Derivação ou *Parse Tree*)



Análise

Estrutura Hierárquica

A estrutura hierárquica de um programa é usualmente expressa por meio de **regras recursivas**.

Exemplo de Regras Recursivas

As seguintes regras poderiam fazer parte da definição de expressões:

1. Qualquer **identificador** é uma expressão;
2. Qualquer **número** é uma expressão;
3. Se $\mathbf{expr}_1$ e $\mathbf{expr}_2$ são expressões, então:
   - $\mathbf{expr}_1 + \mathbf{expr}_2$
   - $\mathbf{expr}_1 * \mathbf{expr}_2$
   - $(\mathbf{expr}_1)$

   também são expressões.

Análise

Estrutura Hierárquica

A estrutura hierárquica de um programa é usualmente expressa por meio de **regras recursivas**.

Exemplo de Regras Recursivas

As seguintes regras poderiam fazer parte da definição de expressões:

1. Qualquer **identificador** é uma expressão;
2. Qualquer **número** é uma expressão;
3. Se $\mathbf{expr}_1$ e $\mathbf{expr}_2$ são expressões, então:
   - $\mathbf{expr}_1 + \mathbf{expr}_2$
   - $\mathbf{expr}_1 * \mathbf{expr}_2$
   - $(\mathbf{expr}_1)$

   também são expressões.

Análise

Estrutura Hierárquica

A estrutura hierárquica de um programa é usualmente expressa por meio de **regras recursivas**.

Exemplo de Regras Recursivas

As seguintes regras poderiam fazer parte da definição de expressões:

1. Qualquer **identificador** é uma expressão;
2. Qualquer **número** é uma expressão;
3. Se $\mathbf{expr}_1$ e $\mathbf{expr}_2$ são expressões, então:
   - $\mathbf{expr}_1 + \mathbf{expr}_2$
   - $\mathbf{expr}_1 * \mathbf{expr}_2$
   - $(\mathbf{expr}_1)$

   também são expressões.

Análise

Estrutura Hierárquica

A estrutura hierárquica de um programa é usualmente expressa por meio de **regras recursivas**.

Exemplo de Regras Recursivas

As seguintes regras poderiam fazer parte da definição de expressões:

1. Qualquer **identificador** é uma expressão;
2. Qualquer **número** é uma expressão;
3. Se $\mathbf{expr_1}$ e $\mathbf{expr_2}$ são expressões, então:
   - $\mathbf{expr_1} + \mathbf{expr_2}$
   - $\mathbf{expr_1} * \mathbf{expr_2}$
   - $(\mathbf{expr_1})$

   também são expressões.

Análise

Exemplo de Regras Recursivas (cont.)

As regras (1) e (2) são as regras base (não recursivas), já a regra (3) define expressões (recursivamente) em termos de operadores aplicados a outras expressões.

Dessa forma, teríamos:

- Pela regra (1), **valor_inicial** e **taxa_de_juros** são expressões;
- Pela regra (2), **60** é uma expressão;
- Pela regra (3), **taxa_de_juros * 60** é uma expressão; e
- Pela regra (3), **valor_inicial + taxa_de_juros * 60** é uma expressão.

Análise

Exemplo de Regras Recursivas (cont.)

As regras (1) e (2) são as regras base (não recursivas), já a regra (3) define expressões (recursivamente) em termos de operadores aplicados a outras expressões.

Dessa forma, teríamos:

- Pela regra (1), **valor_inicial** e **taxa_de_juros** são expressões;
- Pela regra (2), **60** é uma expressão;
- Pela regra (3), **taxa_de_juros * 60** é uma expressão; e
- Pela regra (3), **valor_inicial + taxa_de_juros * 60** é uma expressão.

Análise

Exemplo de Regras Recursivas (cont.)

As regras (1) e (2) são as regras base (não recursivas), já a regra (3) define expressões (recursivamente) em termos de operadores aplicados a outras expressões.

Dessa forma, teríamos:

- Pela regra (1), **valor_inicial** e **taxa_de_juros** são expressões;
- Pela regra (2), **60** é uma expressão;
- Pela regra (3), **taxa_de_juros * 60** é uma expressão; e
- Pela regra (3), **valor_inicial + taxa_de_juros * 60** é uma expressão.

Análise

Exemplo de Regras Recursivas (cont.)

As regras (1) e (2) são as regras base (não recursivas), já a regra (3) define expressões (recursivamente) em termos de operadores aplicados a outras expressões.

Dessa forma, teríamos:

- Pela regra (1), **valor_inicial** e **taxa_de_juros** são expressões;
- Pela regra (2), **60** é uma expressão;
- Pela regra (3), **taxa_de_juros * 60** é uma expressão; e
- Pela regra (3), **valor_inicial + taxa_de_juros * 60** é uma expressão.

Análise

Exemplo de Regras Recursivas (cont.)

As regras (1) e (2) são as regras base (não recursivas), já a regra (3) define expressões (recursivamente) em termos de operadores aplicados a outras expressões.

Dessa forma, teríamos:

- Pela regra (1), **valor_inicial** e **taxa_de_juros** são expressões;
- Pela regra (2), **60** é uma expressão;
- Pela regra (3), **taxa_de_juros * 60** é uma expressão; e
- Pela regra (3), **valor_inicial + taxa_de_juros * 60** é uma expressão.

Análise

Exemplo de Regras Recursivas (cont.)

Similarmente, várias linguagens definem enunciados de forma recursiva, por exemplo:

1. Se $\mathbf{id}_1$ é um identificador e $\mathbf{expr}_1$, uma expressão, então

$$\mathbf{id}_1 := \mathbf{expr}_1$$

é um enunciado (de atribuição).

2. Se $\mathbf{expr}_1$ é uma expressão e $\mathbf{cmd}_1$, um enunciado, então

$$\mathbf{while}\ (\mathbf{expr}_1)\ \mathbf{do}\ \mathbf{cmd}_1$$

$$\mathbf{if}\ (\mathbf{expr}_1)\ \mathbf{then}\ \mathbf{cmd}_1$$

são enunciados (de repetição e alternativa, respectivamente).

Análise

Exemplo de Regras Recursivas (cont.)

Similarmente, várias linguagens definem enunciados de forma recursiva, por exemplo:

1. Se $\mathbf{id_1}$ é um identificador e $\mathbf{expr_1}$, uma expressão, então

$$\mathbf{id_1} := \mathbf{expr_1}$$

é um enunciado (de atribuição).

2. Se $\mathbf{expr_1}$ é uma expressão e $\mathbf{cmd_1}$, um enunciado, então

$$\mathbf{while}\ (\mathbf{expr_1})\ \mathbf{do}\ \mathbf{cmd_1}$$

$$\mathbf{if}\ (\mathbf{expr_1})\ \mathbf{then}\ \mathbf{cmd_1}$$

são enunciados (de repetição e alternativa, respectivamente).

Análise

Exemplo de Regras Recursivas (cont.)

Similarmente, várias linguagens definem enunciados de forma recursiva, por exemplo:

1. Se $\mathbf{id_1}$ é um identificador e $\mathbf{expr_1}$, uma expressão, então

$$\mathbf{id_1} := \mathbf{expr_1}$$

é um enunciado (de atribuição).

2. Se $\mathbf{expr_1}$ é uma expressão e $\mathbf{cmd_1}$, um enunciado, então

$$\mathbf{while}\ (\mathbf{expr_1})\ \mathbf{do}\ \mathbf{cmd_1}$$

$$\mathbf{if}\ (\mathbf{expr_1})\ \mathbf{then}\ \mathbf{cmd_1}$$

são enunciados (de repetição e alternativa, respectivamente).

Análise

Análise Semântica

Análise semântica é responsável por verificar a existência de erros semânticos no programa fonte e obtrr informações sobre tipos para a fase de geração de código intermediário.

Utiliza-se a estrutura hierárquica produzida durante a análise sintática para se determinar os operadores e operandos das expressões e enunciados.

Verificação de Tipos (*Type Checking*)

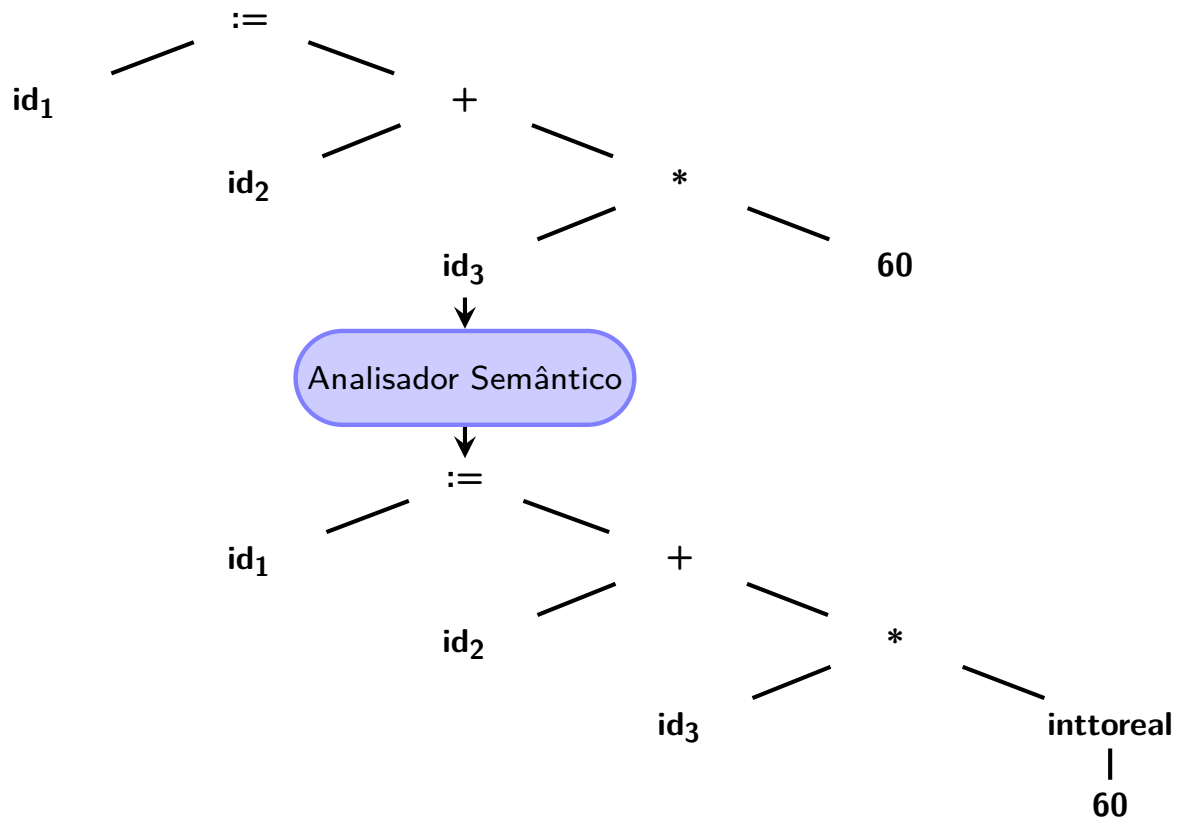
Verificação de tipos é um componente importante da análise semântica.

Verificação de tipos é responsável por checar se cada operador possui os operandos que são permitidos pela especificicação da linguagem.

Deve-se relatar erros, por exemplo, na atribuição de um valor real para uma variável de tipo inteiro ou, se a linguagem permitir, realizar a coerção de tipo.

Análise

Exemplo de Análise Semântica



Análise

Geração de Código Intermediário

Após as análises sintática e semântica, alguns compiladores geram um representação intermediária do programa fonte.

Pode-se entender essa representação intermediária como um programa para uma máquina abstrata.

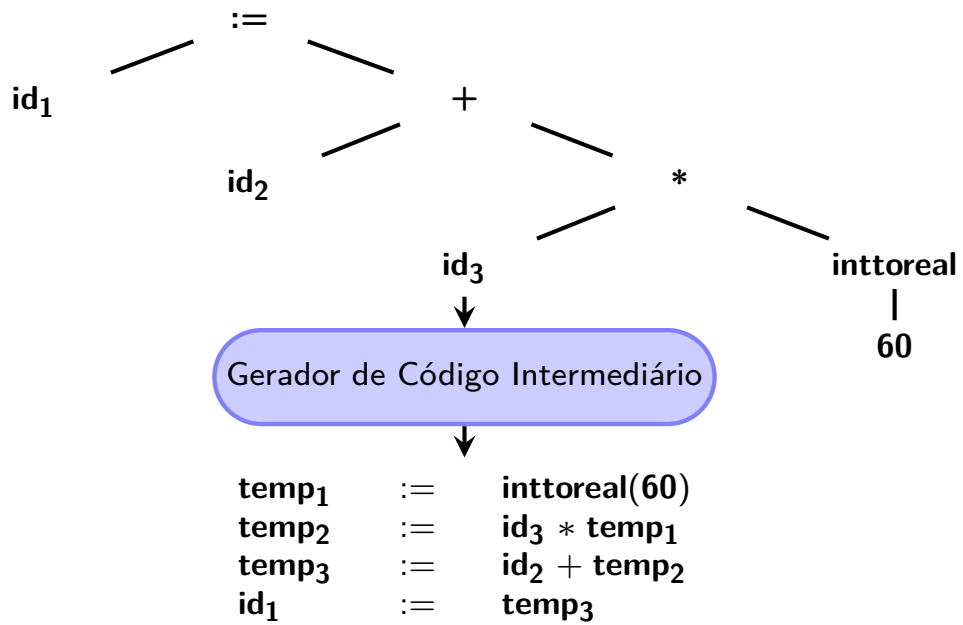
Essa representação deve ser fácil de produzir e fácil de traduzir no programa alvo e pode ter uma variedade de formas.

Uma possível forma de representação intermediária é chamada de "*código de três endereços*" – que é como uma linguagem de montagem para uma máquina, na qual cada localização de memória possa atuar como um registrador.

O *código de três endereços* consiste em uma sequência de instruções em que cada uma delas possui no máximo 3 operandos.

Análise

Exemplo de Geração de Código Intermédiário



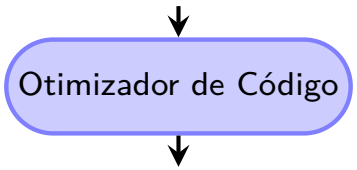
Síntese

Otimização de Código

A fase de otimização de código procura melhorar o código intermediário, de modo que o resultado seja um código de máquina mais rápido em tempo de execução ou econômico no uso de memória.

Exemplo de Otimização de Código

$$
\begin{array}{lcl}
\mathbf{temp_1} & := & \mathbf{inttoreal(60)} \\
\mathbf{temp_2} & := & \mathbf{id_3} * \mathbf{temp_1} \\
\mathbf{temp_3} & := & \mathbf{id_2} + \mathbf{temp_2} \\
\mathbf{id_1} & := & \mathbf{temp_3}
\end{array}
$$



$$
\begin{array}{lcl}
\mathbf{temp_1} & := & \mathbf{id_3} * \mathbf{60.0} \\
\mathbf{id_1} & := & \mathbf{id_2} + \mathbf{temp_1}
\end{array}
$$

Síntese

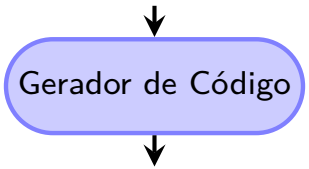
Geração de Código

A geração de código representa a fase final do compilador e consiste normalmente na produção código relocável ou código de montagem.

As posições de memória são selecionadas para cada uma das varáveis usadas no programa e, então, as instruções intermediárias são traduzidas em um sequência de instruções de máquina que realizam a mesma tarefa.

Exemplo de Geração de Código

$$\begin{array}{lll} \mathbf{temp_1} & := & \mathbf{id_3} * \mathbf{60.0} \\ \mathbf{id_1} & := & \mathbf{id_2} + \mathbf{temp_1} \end{array}$$




```
MOVF   id3,   R2
MULF   #60.0, R2
MOVF   id2,   R1
ADDF   R2,    R1
MOVF   R1,    id1
```