

# **Kapitola II.**

## **Úvod do překladačů**

# Překladač

- **Vstup:** Zdrojový program
  - **Výstup:** Cílový program
- 
- **Metoda:**
    - Překladač čte *zdrojový program* (napsaný ve zdrojovém jazyce) a překládá ho na *cílový program* (napsaný v cílovém jazyce)
    - Zdrojový a cílový program je vzájemně *funkčně ekvivalentní*.

# Struktura překladače: Logické fáze

`Position := Initial + Rate * 60`

**Lexikální analyzátor**

`Id1 := Id2 + Id3 * 60`

**Syntaktický analyzátor**

```

      :=
     / \
  Id1 /  \
      +   \
     / \   \
  Id2 /  \ * 60
      \
     Id3
  
```

**Sémantický analyzátor**

```

      :=
     / \
  Id1 /  \
      +   \
     / \   \
  Id2 /  \ * IntToReal
      \      |
     Id3     60
  
```

**Generátor vnitřního kódu**

```

T1  := IntToReal (60)
T2  := Id3 * T1
T3  := Id2 + T2
Id1 := T3
  
```

**Optimalizátor**

```

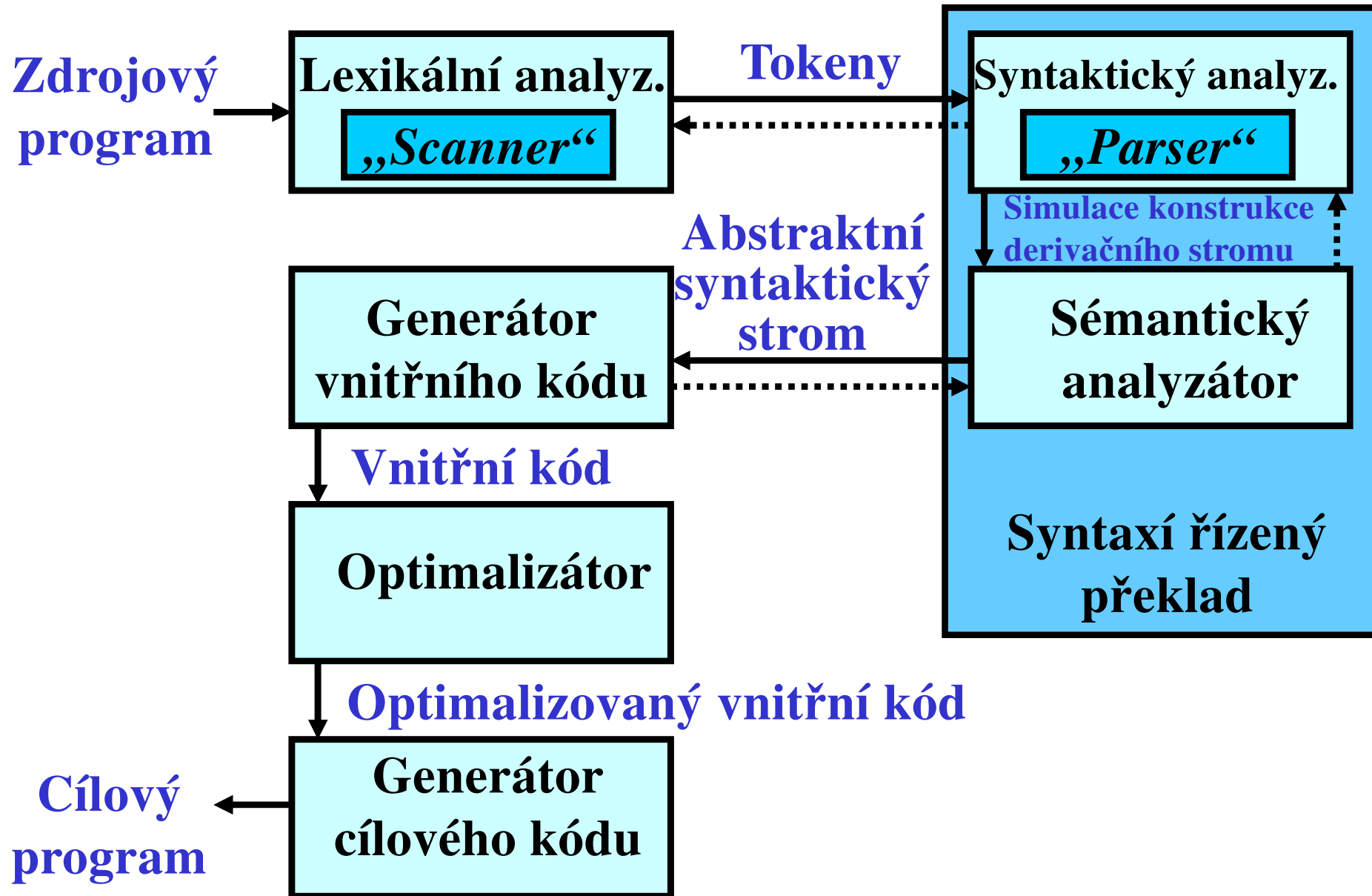
T1  := Id3 * 60.0
Id1 := Id2 + T1
  
```

**Generátor cílového kódu**

```

fmov R2 , Id3
fmul R2 , #60.0
fmov R3 , Id2
fadd R2 , R3
fmov Id1, R2
  
```

# Struktura překladače: Konstrukce



# Jazyky a překladače

**Teoretický pohled na formální jazyk:**

$$\Sigma = \{a, b\}, L = \{a^n b^n : n \geq 0\}$$

**Otázka:**  $aabb \in L$  ?

---

**Praktický pohled na formální jazyk:**

$$\Sigma = \{begin, end, id, :=, *, ;, \dots\},$$

$L_{Pascal}$  = Programovací jazyk Pascal

**Otázka:**  $begin\ id\ :=\ id\ *\ id;\ end;\ \in L_{Pascal}$  ?

**ANO:** Program je OK  $\Rightarrow$   
Vytvoř cílový program

**NE:** Program není v pořádku  $\Rightarrow$   
Najdi, kde jsou chyby.

# Lexikální analyzátor (Scanner)

- **Vstup:** Zdrojový program
  - **Výstup:** Řetězec tokenů
- 
- **Metoda:**
    - Zdrojový program je rozdělen na *lexémy* = logicky oddělené lexikální jednotky – (identifikátory, čísla, klíčová slova, operátory,...)
    - Lexémy jsou reprezentovány *tokeny*
    - Některé tokeny mohou mít atributy
-

# Lexikální analyzátor: Příklad

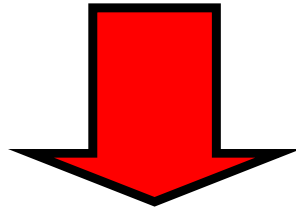
**Zdrojový program:**

```
Position := Initial + Rate * 60
```

# Lexikální analyzátor: Příklad

**Zdrojový program:**

`Position := Initial + Rate * 60`



**Lexémy:**

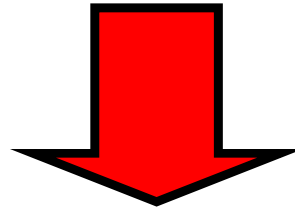
<code>Position</code>	<code>:=</code>	<code>Initial</code>	<code>+</code>	<code>Rate</code>	<code>*</code>	<code>60</code>
-----------------------	-----------------	----------------------	----------------	-------------------	----------------	-----------------



# Lexikální analyzátor: Příklad

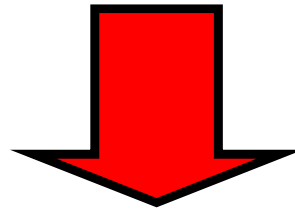
## Zdrojový program:

`Position := Initial + Rate * 60`



## Lexémy:

Position	:=	Initial	+	Rate	*	60
----------	----	---------	---	------	---	----



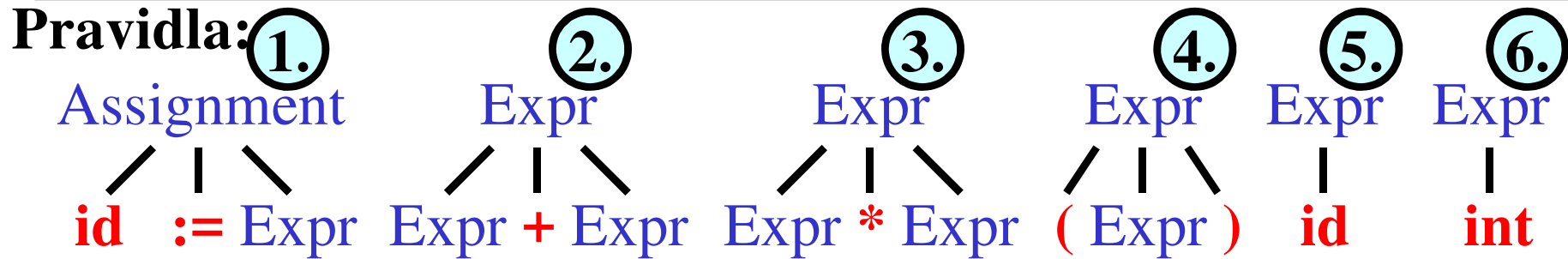
## Tokeny:

<u>id</u> ↪ Position	<u>:=</u>	<u>id</u> ↪ Initial	<u>+</u>	<u>id</u> ↪ Rate	<u>*</u>	<u>int</u> 60
-------------------------	-----------	------------------------	----------	---------------------	----------	------------------

# Syntaktický analyzátor (Parser)

- **Vstup:** Řetězec tokenů
  - **Výstup:** Simulace konstrukce derivačního stromu
- 
- **Metoda:**
  - Syntaktický analyzátor kontroluje, zda řetězec tokenů reprezentuje syntakticky správně napsaný program.
  - Pokud je k danému řetězci tokenů nalezen *derivační strom*, program je správný, jinak ne.
  - Simulace konstrukce derivačního stromu je založena na gramatických pravidlech.
  - Dva přístupy: Shora dolů a zdola nahoru.
-

# Syntaktický analyzátor: Příklad



Derivační  
strom

Assignment

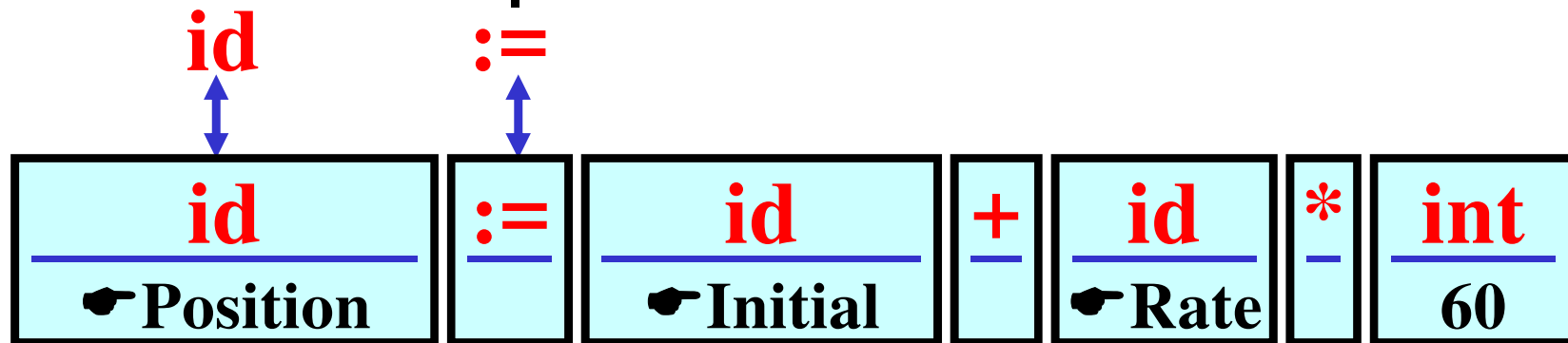
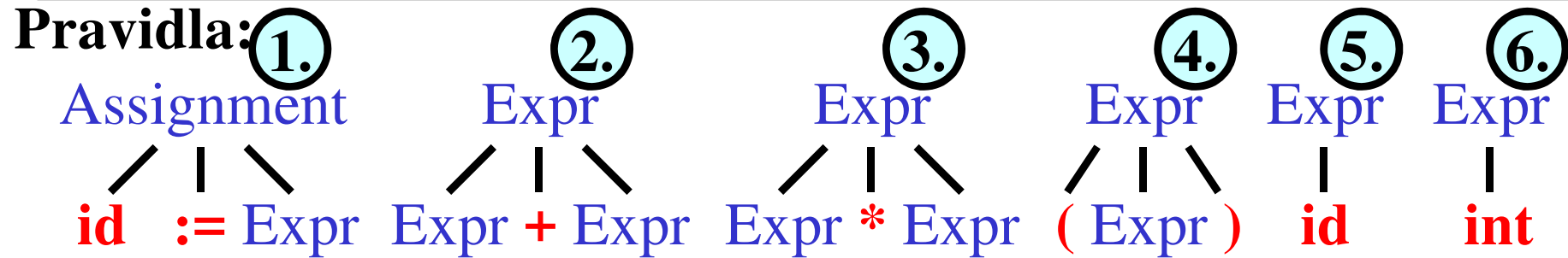
Úkol:

Simuluj derivační strom  
užitím gramatických pravidel

<b>id</b>	<b>:=</b>	<b>id</b>	<b>+</b>	<b>id</b>	<b>*</b>	<b>int</b>
☛ Position		☛ Initial		☛ Rate		60

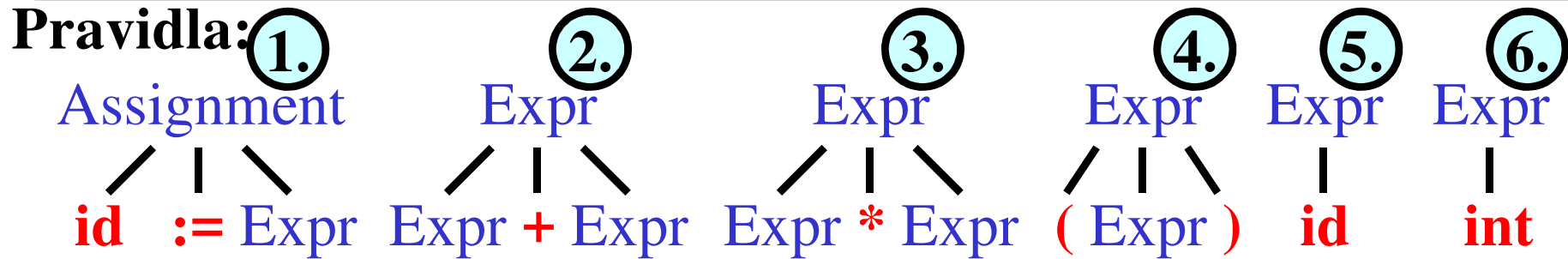
Tokeny

# Syntaktický analyzátor: Příklad

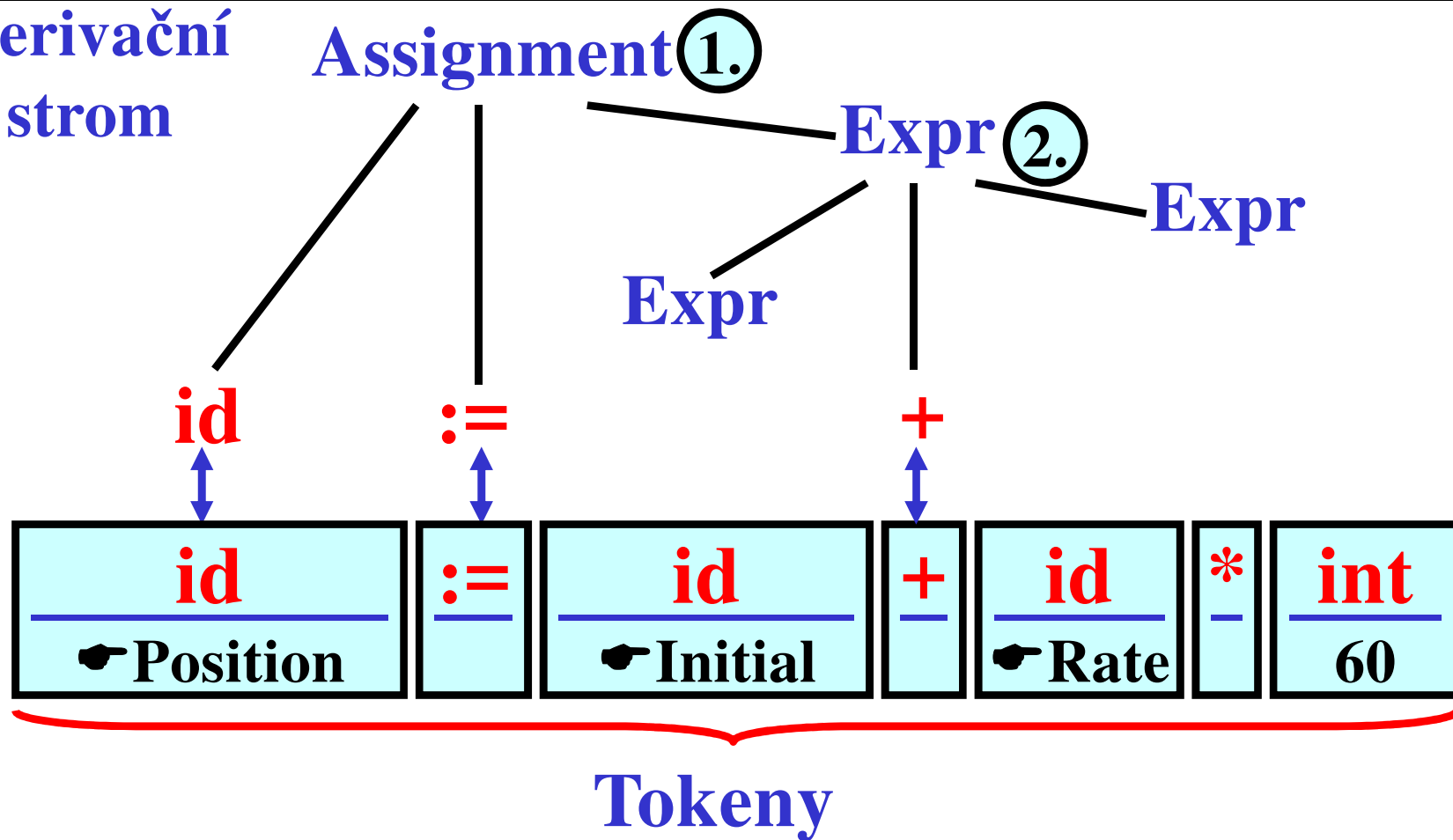


**Tokeny**

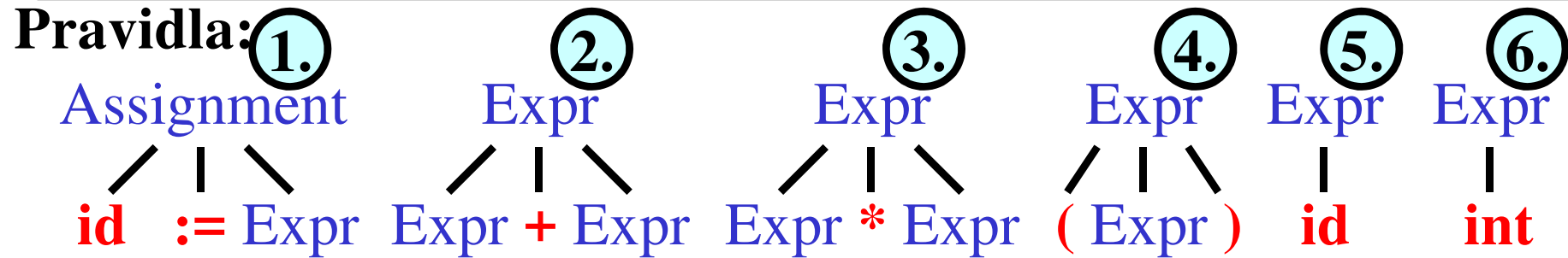
# Syntaktický analyzátor: Příklad



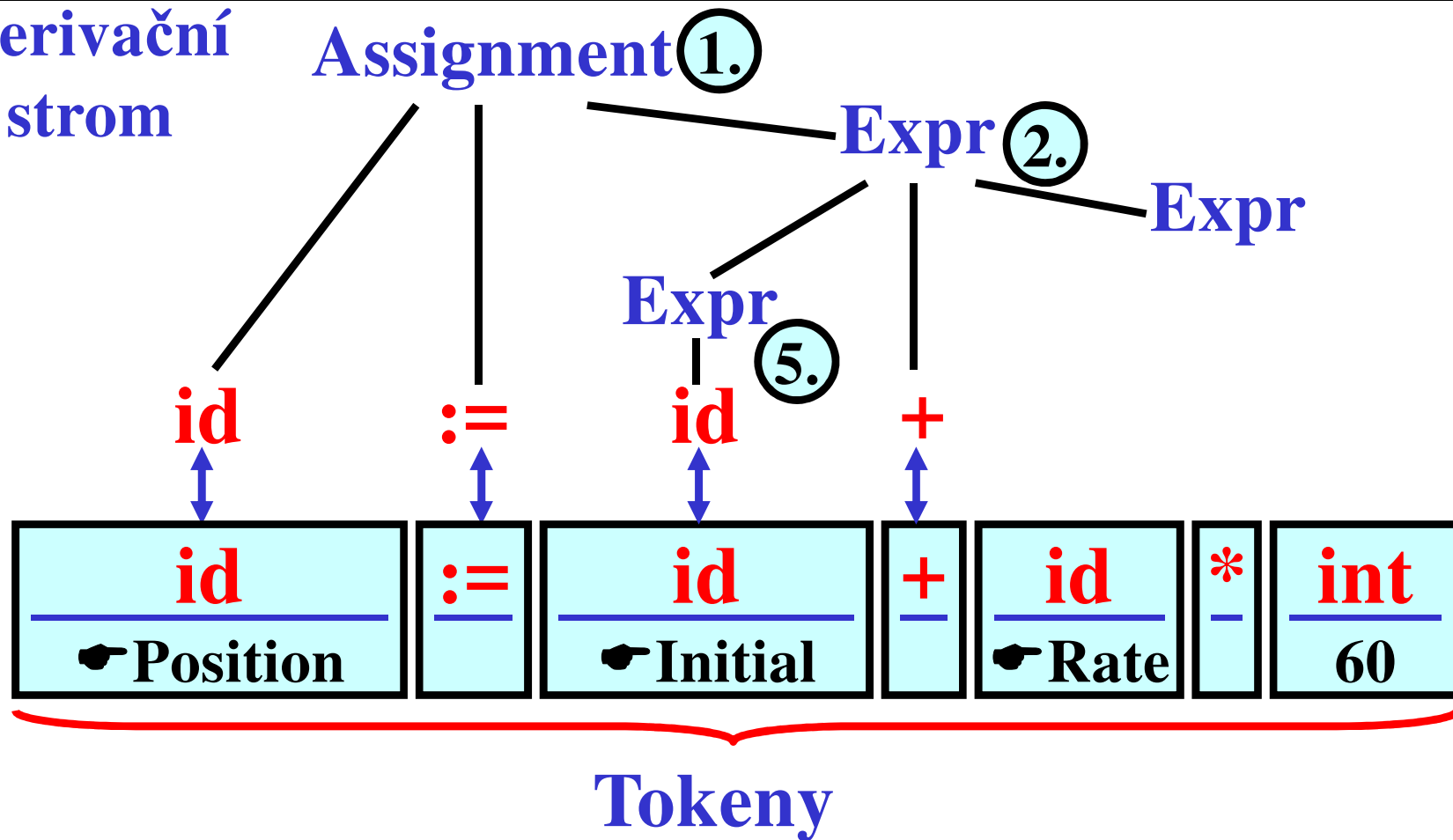
Derivační  
strom



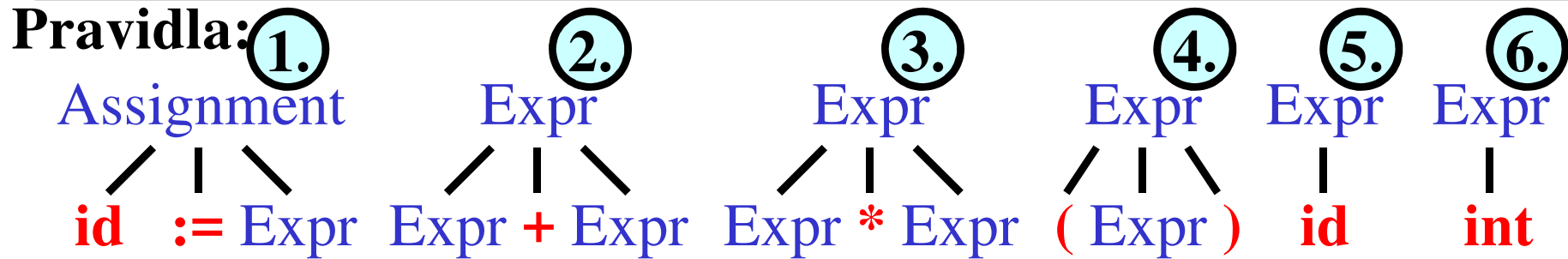
# Syntaktický analyzátor: Příklad



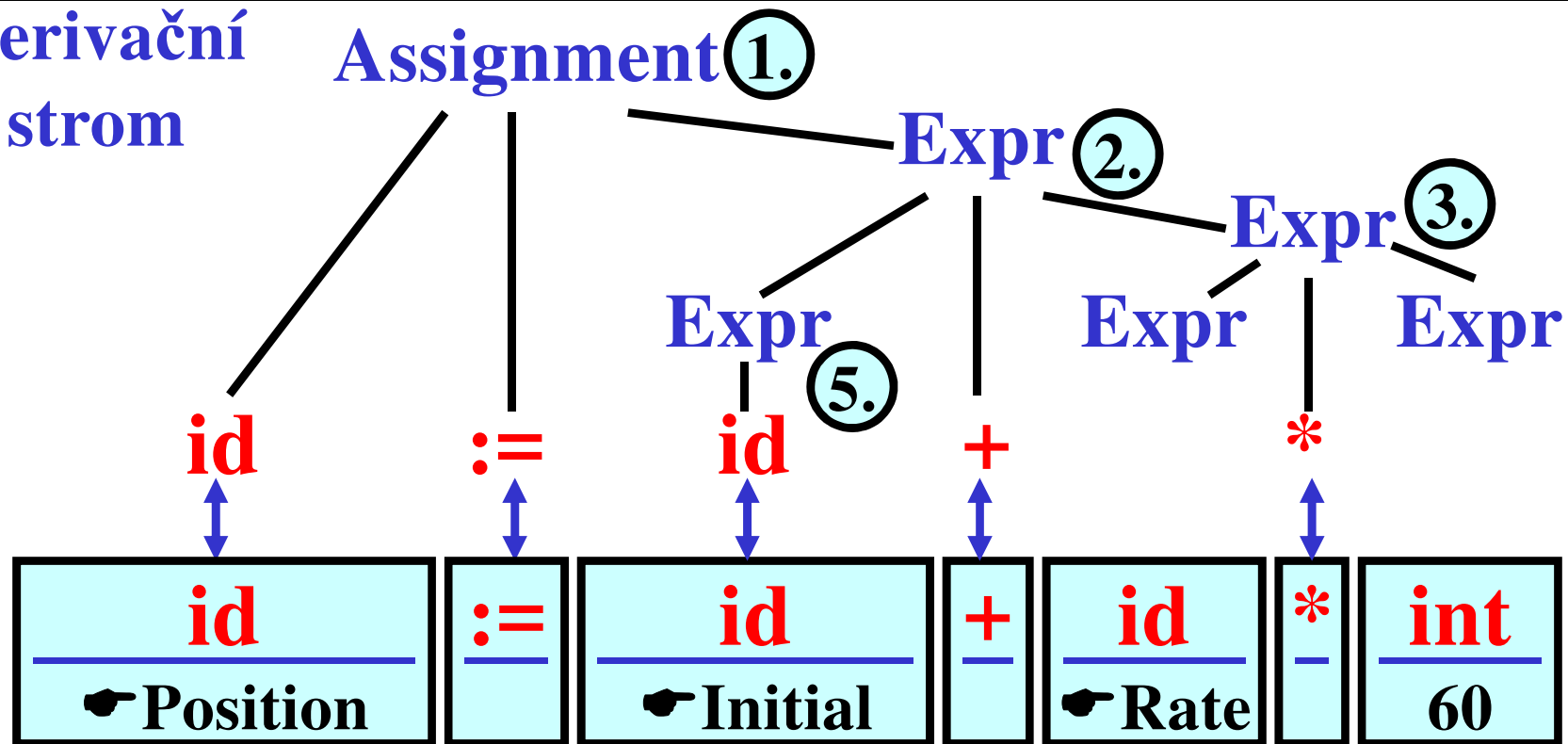
Derivační  
strom



# Syntaktický analyzátor: Příklad

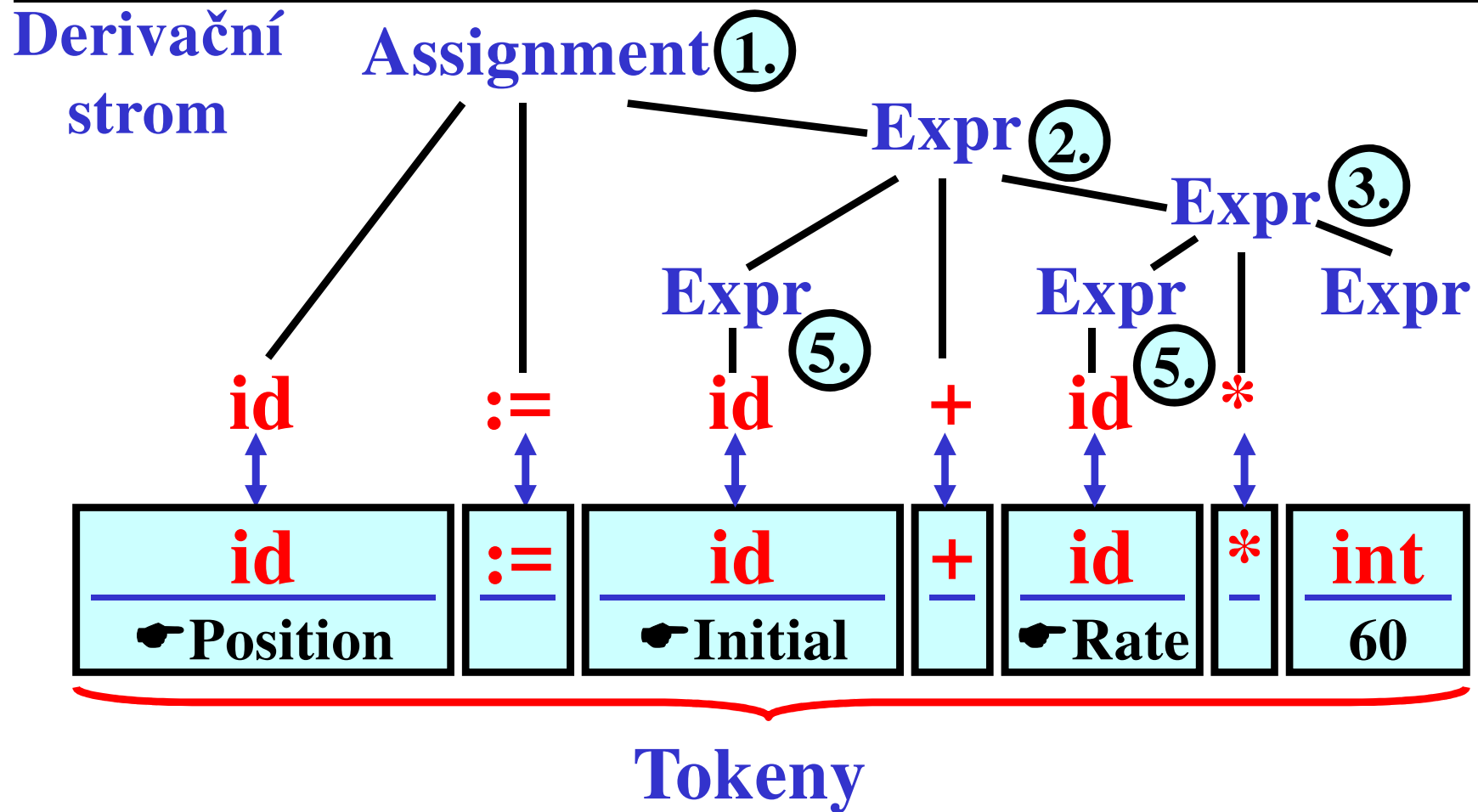
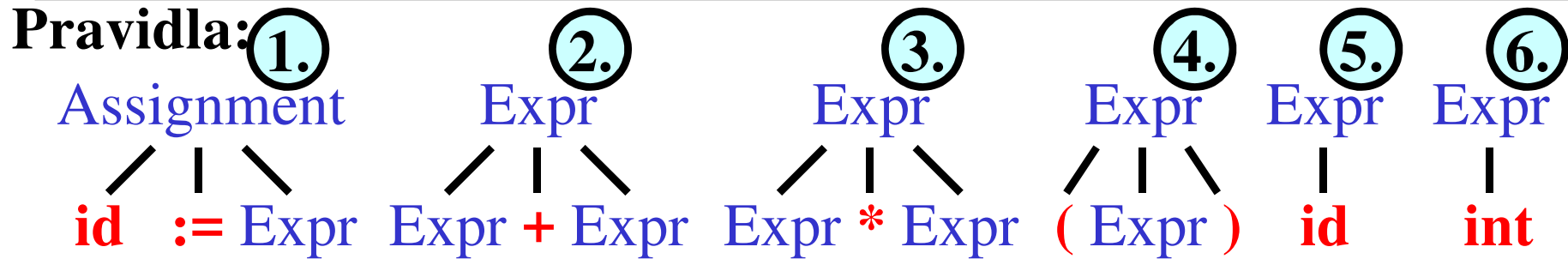


Derivační  
strom



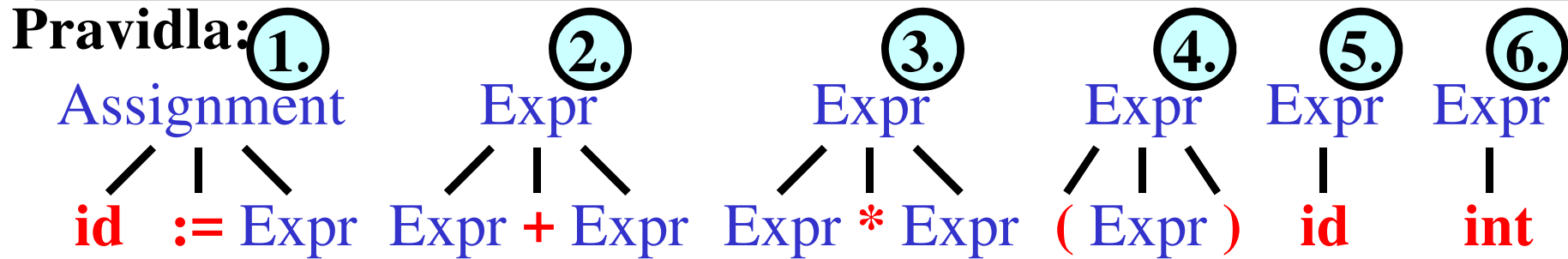
Tokeny

# Syntaktický analyzátor: Příklad

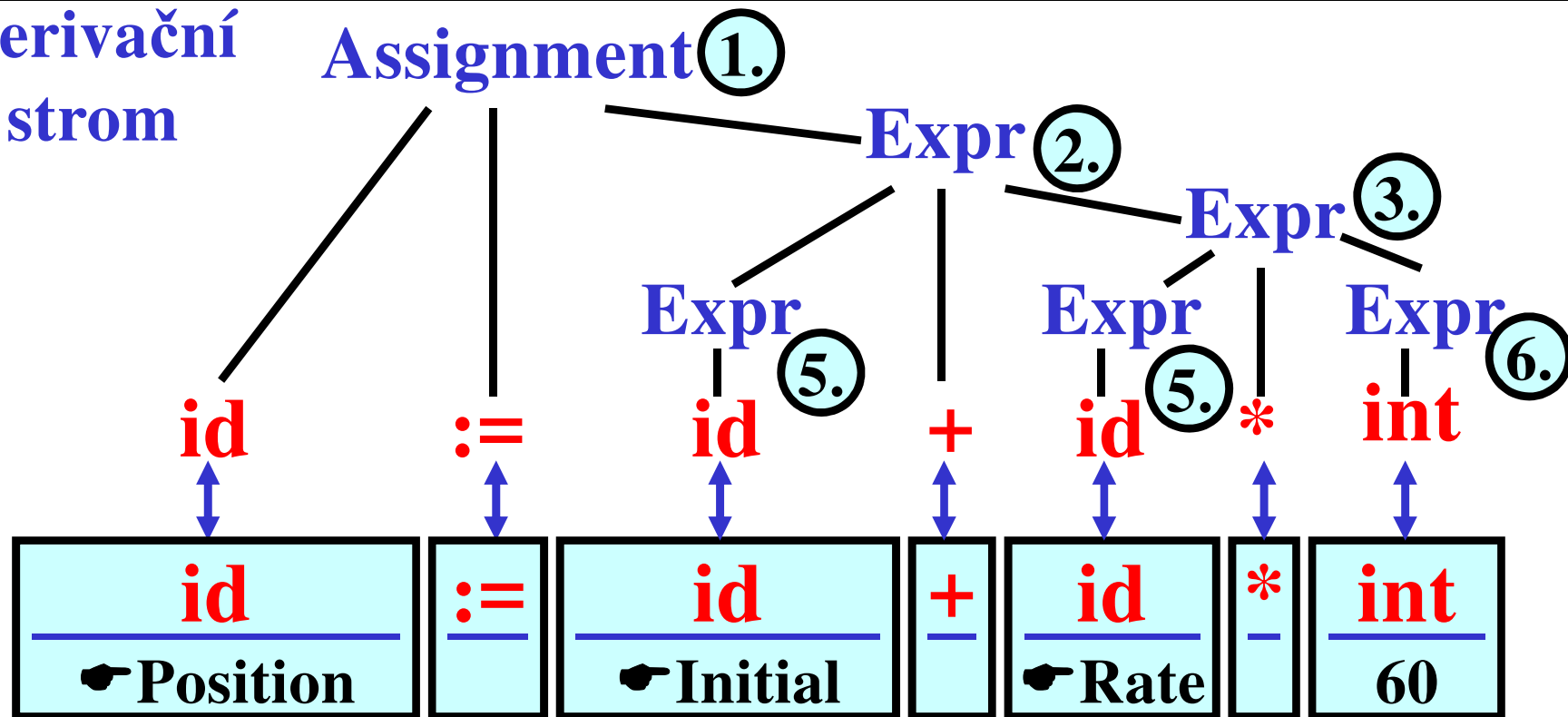




# Syntaktický analyzátor: Příklad



Derivační  
strom



Tokeny

# Sémantický analyzátor

- **Vstup:** Simulace konstrukce derivačního stromu
  - **Výstup:** Abstraktní syntaktický strom
- 

- **Metoda:**

- Sémantický analyzátor kontroluje sémantické aspekty programu:

- *kontrola typů*, při které může provádět implicitní konverze (např. int-to-real)
  - *kontrola deklarací proměnných*

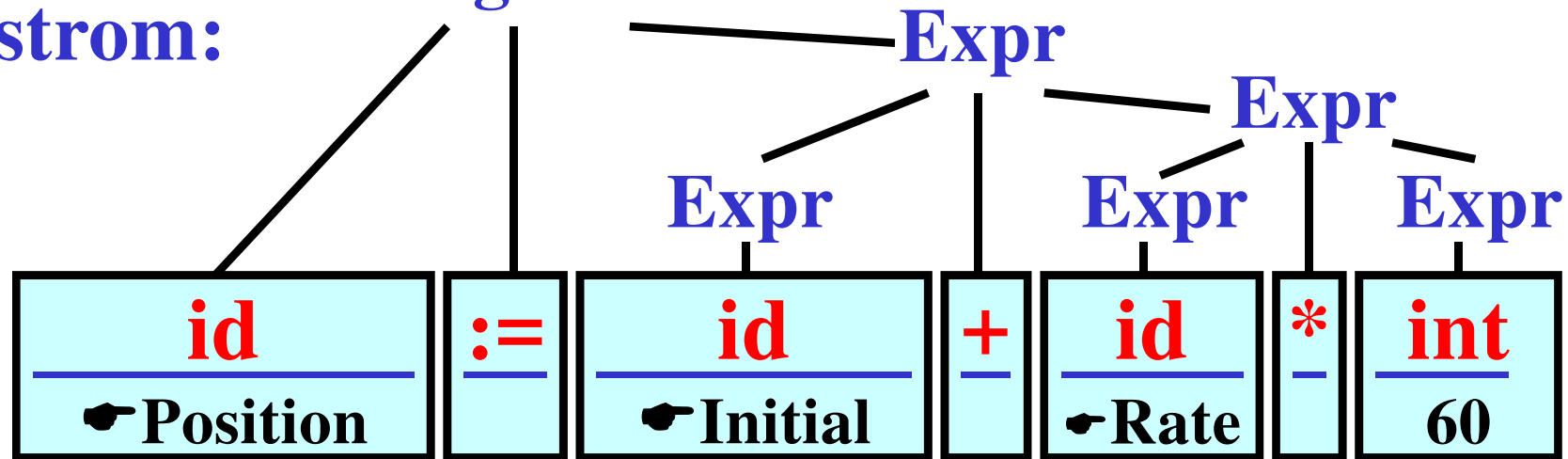
- **Syntaxí řízený překlad:**

Syntaktický analyzátor řídí:

- Provádění sémantických akcí
  - Generování abstraktního syntaktického stromu

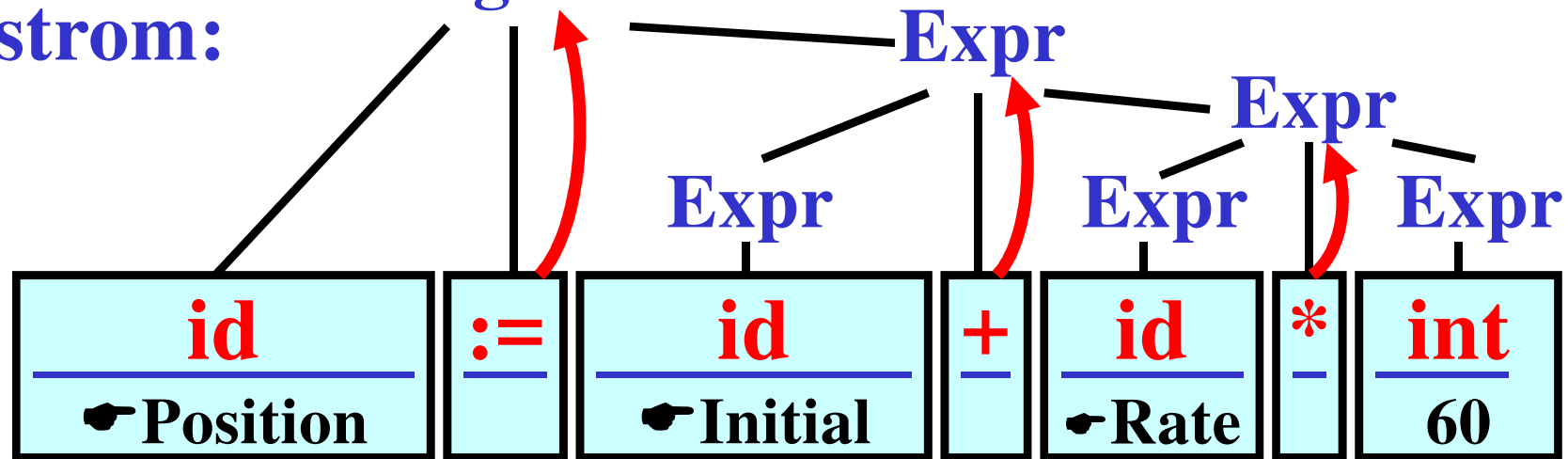
# Syntaxí řízený překlad: Příklad

Derivační  
strom:



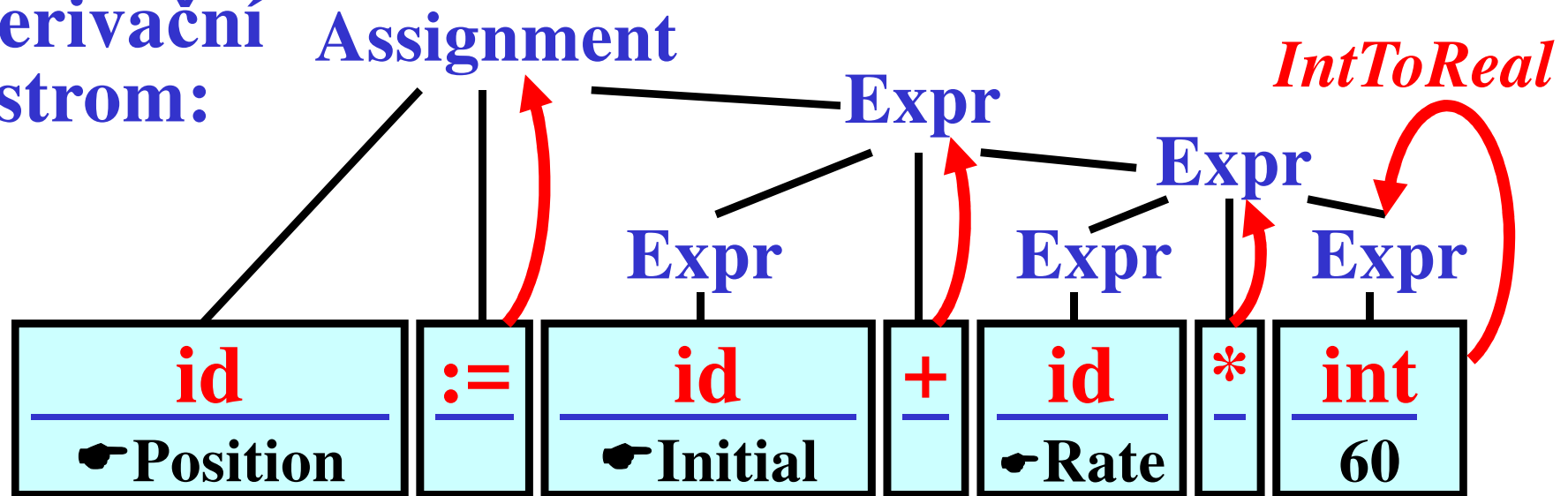
# Syntaxí řízený překlad: Příklad

Derivační  
strom:



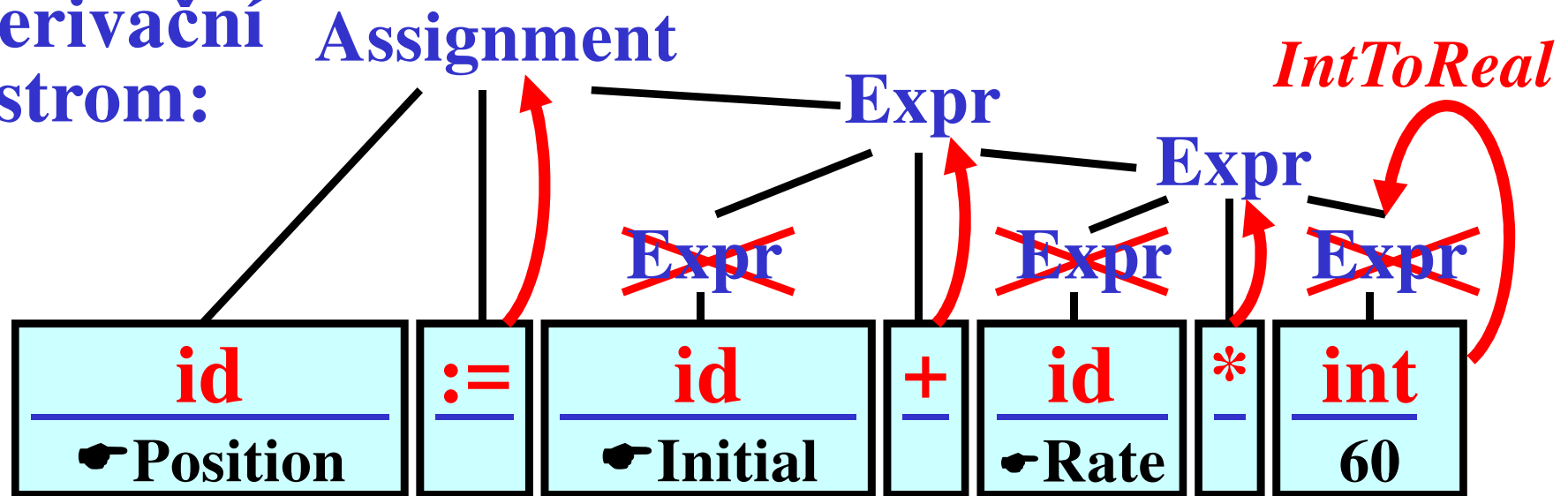
# Syntaxí řízený překlad: Příklad

Derivační  
strom:



# Syntaxí řízený překlad: Příklad

Derivační  
strom:



# Syntaxí řízený překlad: Příklad

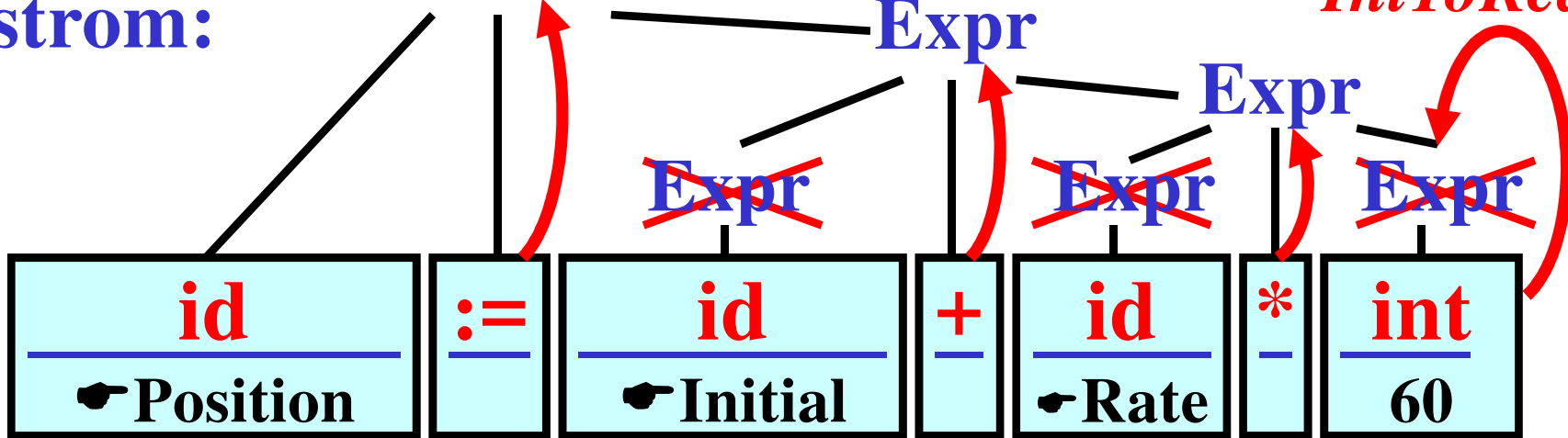
# Derivační strom:

# Assignment

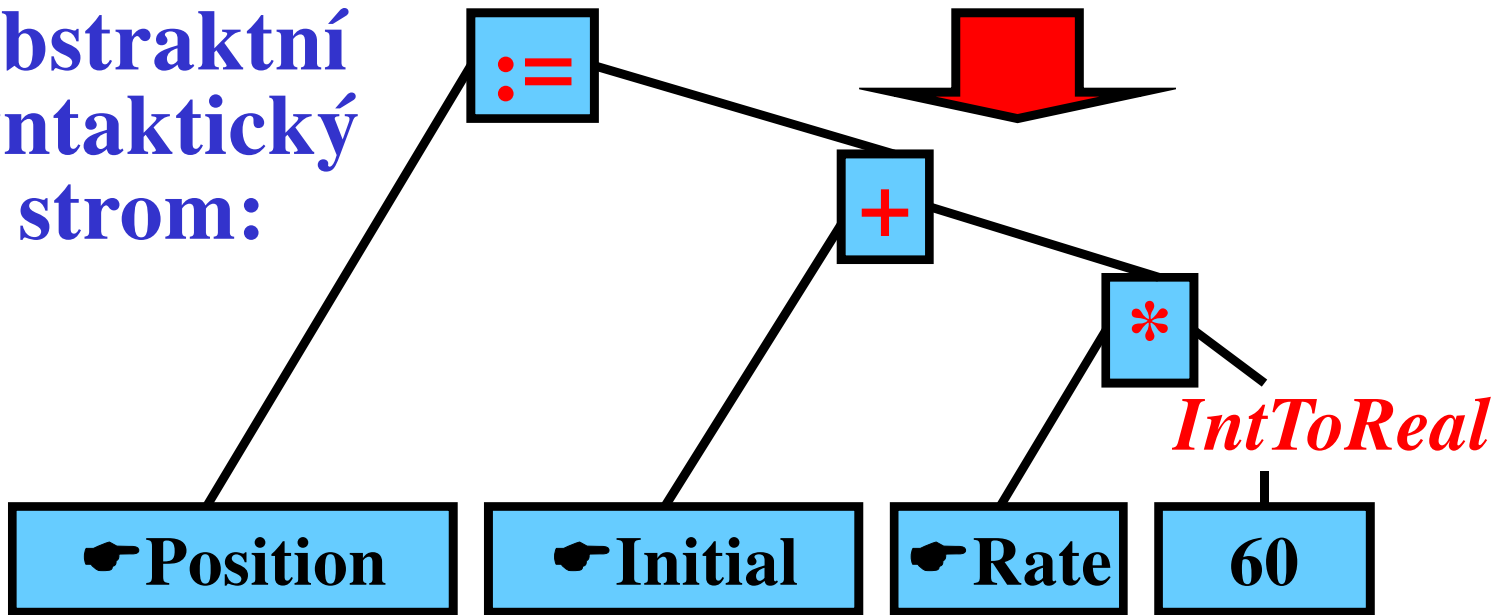
# Expr

# Expr

# *IntToReal*



# Abstraktní syntaktický strom:



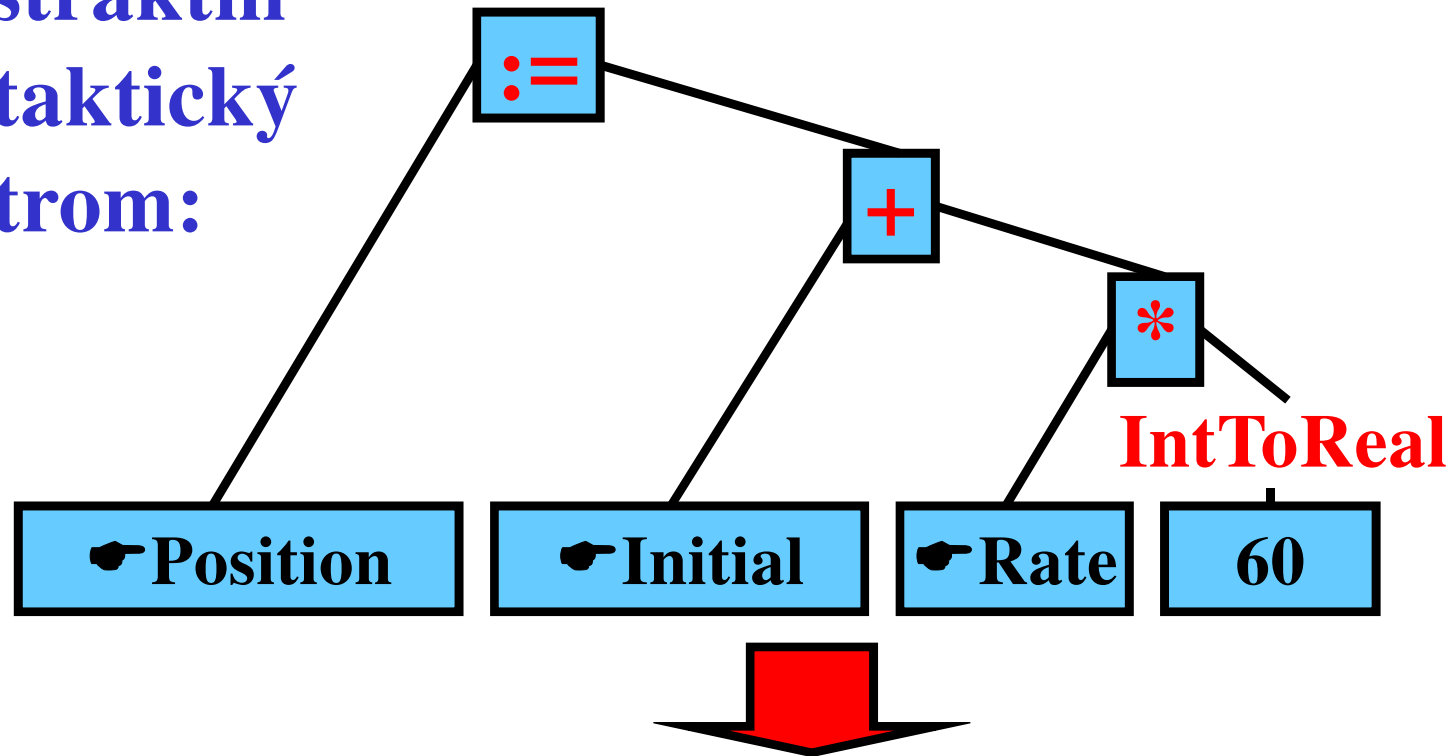
# Generátor vnitřního kódu

- **Vstup:** Abstraktní syntaktický strom
  - **Výstup:** Vnitřní kód
- 
- **Metoda:**
    - Generátor vnitřního kódu vytváří vnitřní reprezentaci programu nazývanou *vnitřní kód* (většinou 3-adresný kód) z následujících důvodů:
      - jednotnost
      - přímý překlad do cílového programu je složitý a „neprůhledný“
      - vnitřní kód lze snadno optimalizovat



# Generátor vnitřního kódu: Příklad

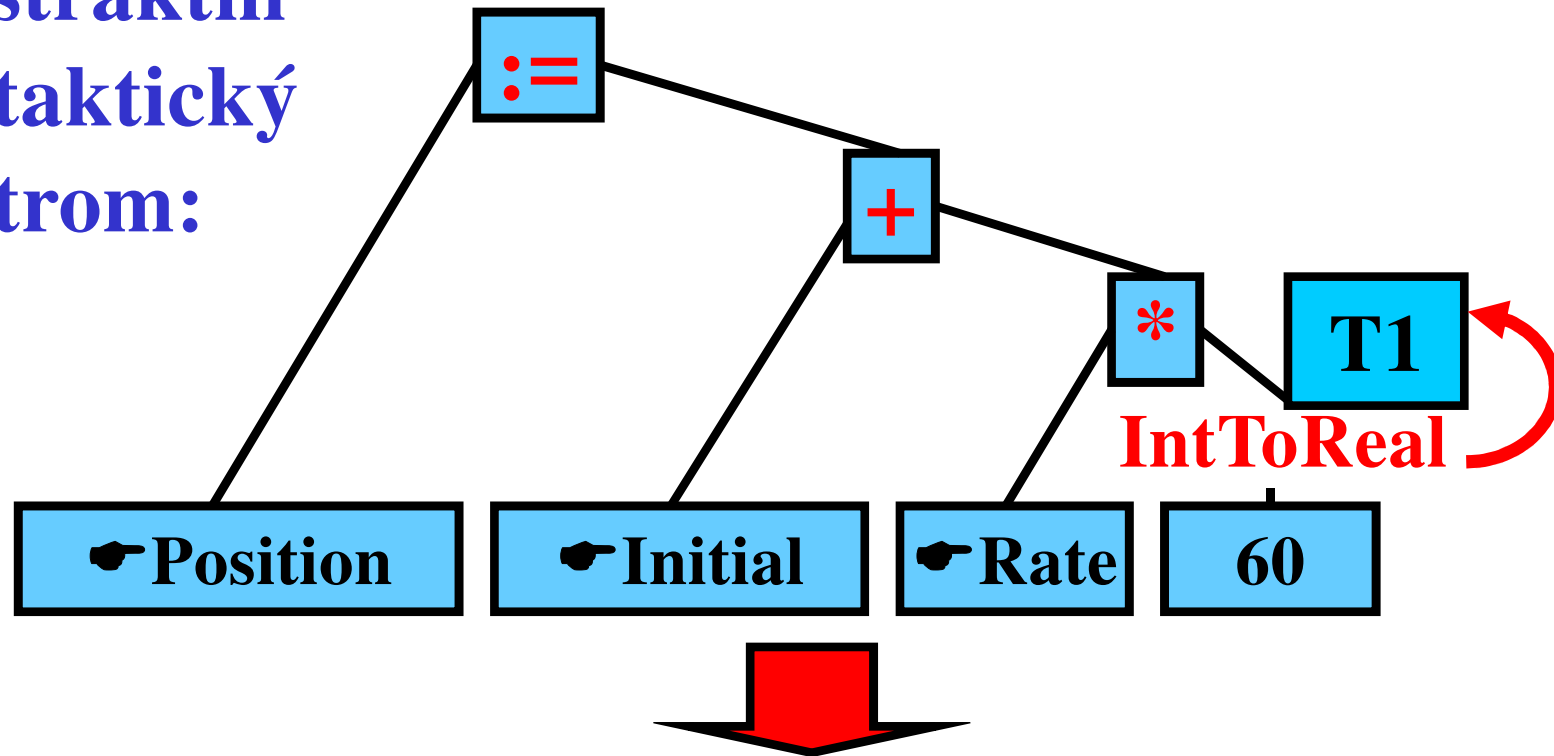
Abstraktní  
syntaktický  
strom:



Vnitřní kód:

# Generátor vnitřního kódu: Příklad

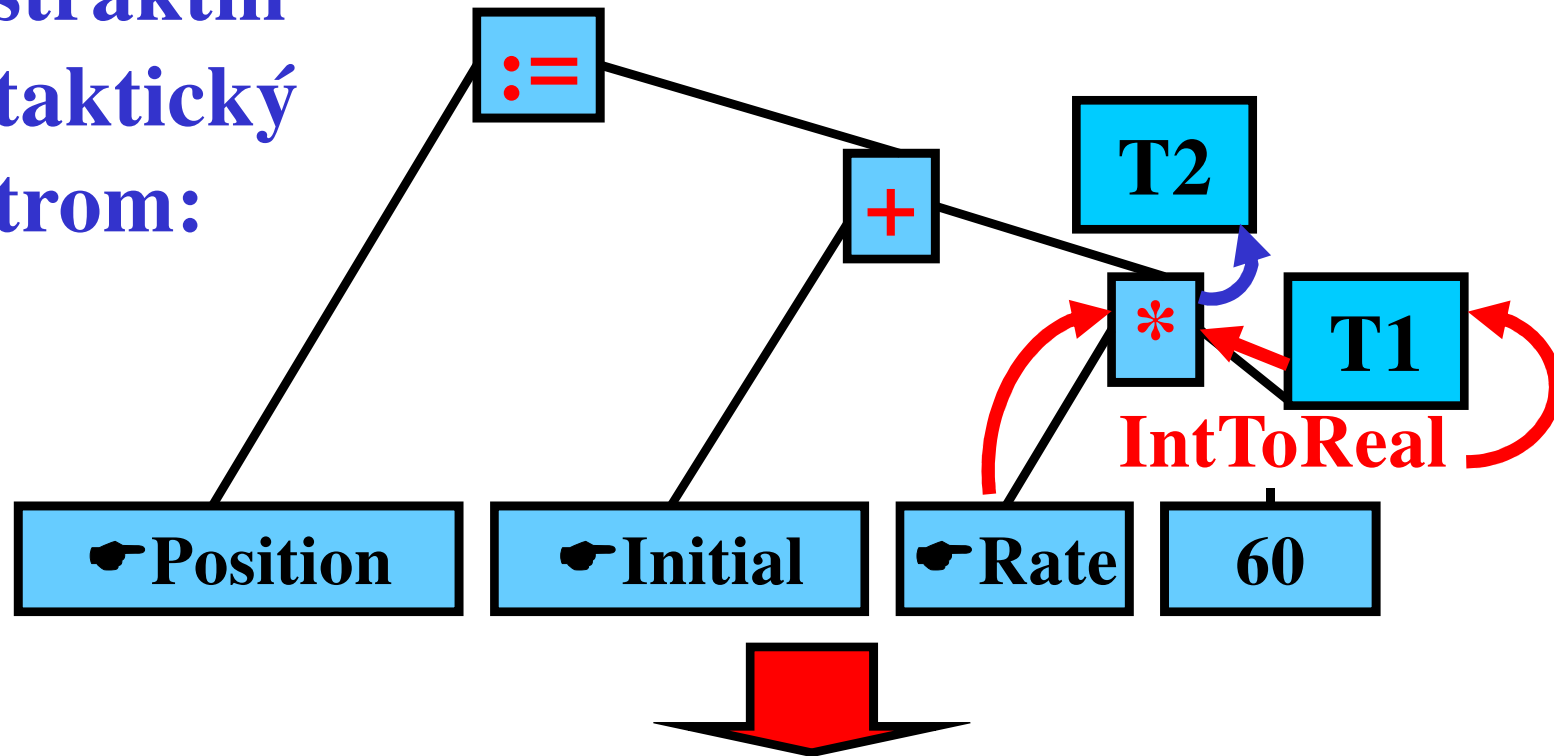
Abstraktní  
syntaktický  
strom:



Vnitřní kód: `T1 := IntToReal(60)`

# Generátor vnitřního kódu: Příklad

Abstraktní  
syntaktický  
strom:



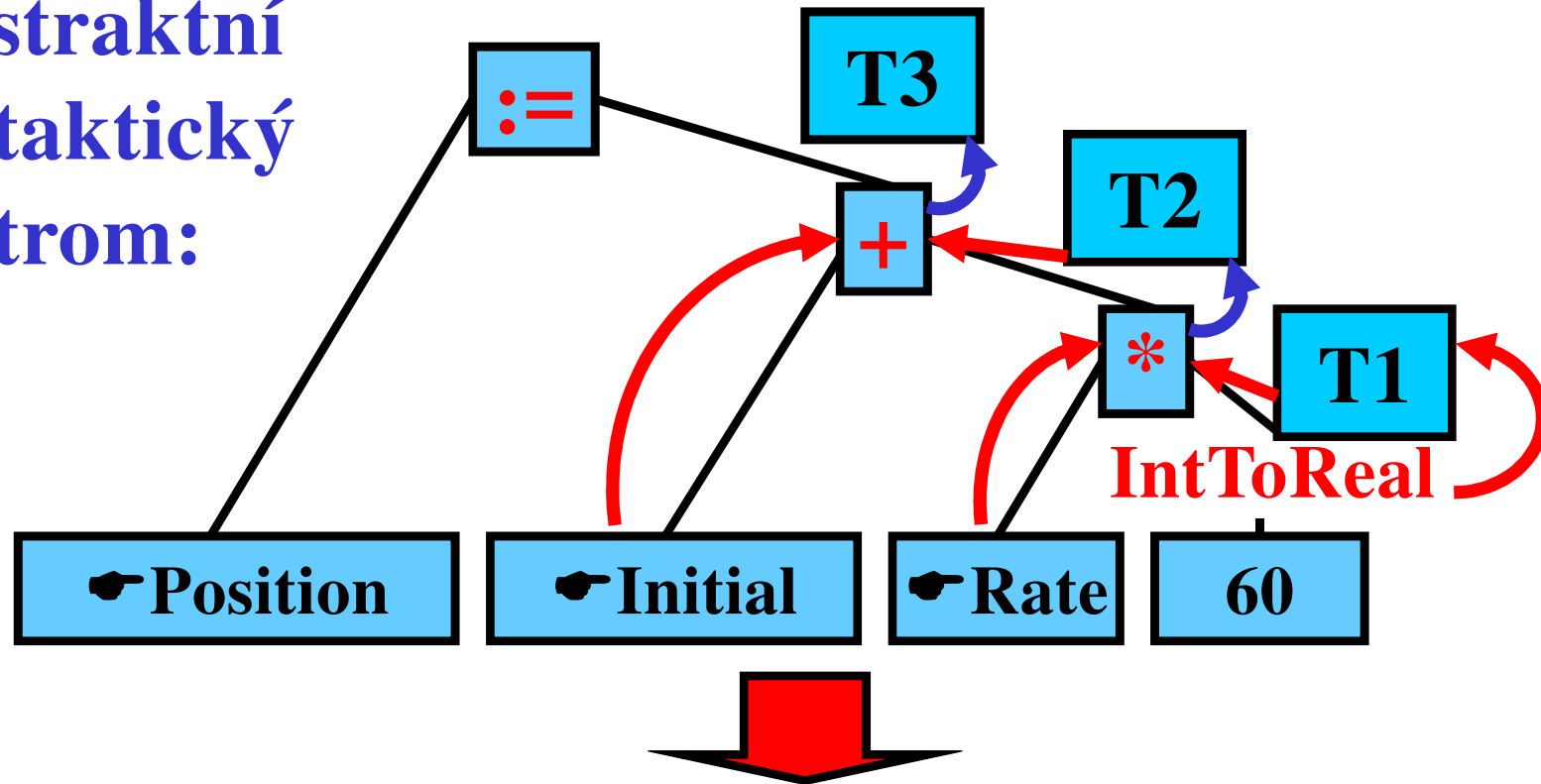
Vnitřní kód:

```

T1 := IntToReal(60)
T2 := Rate * T1
  
```

# Generátor vnitřního kódu: Příklad

Abstraktní  
syntaktický  
strom:



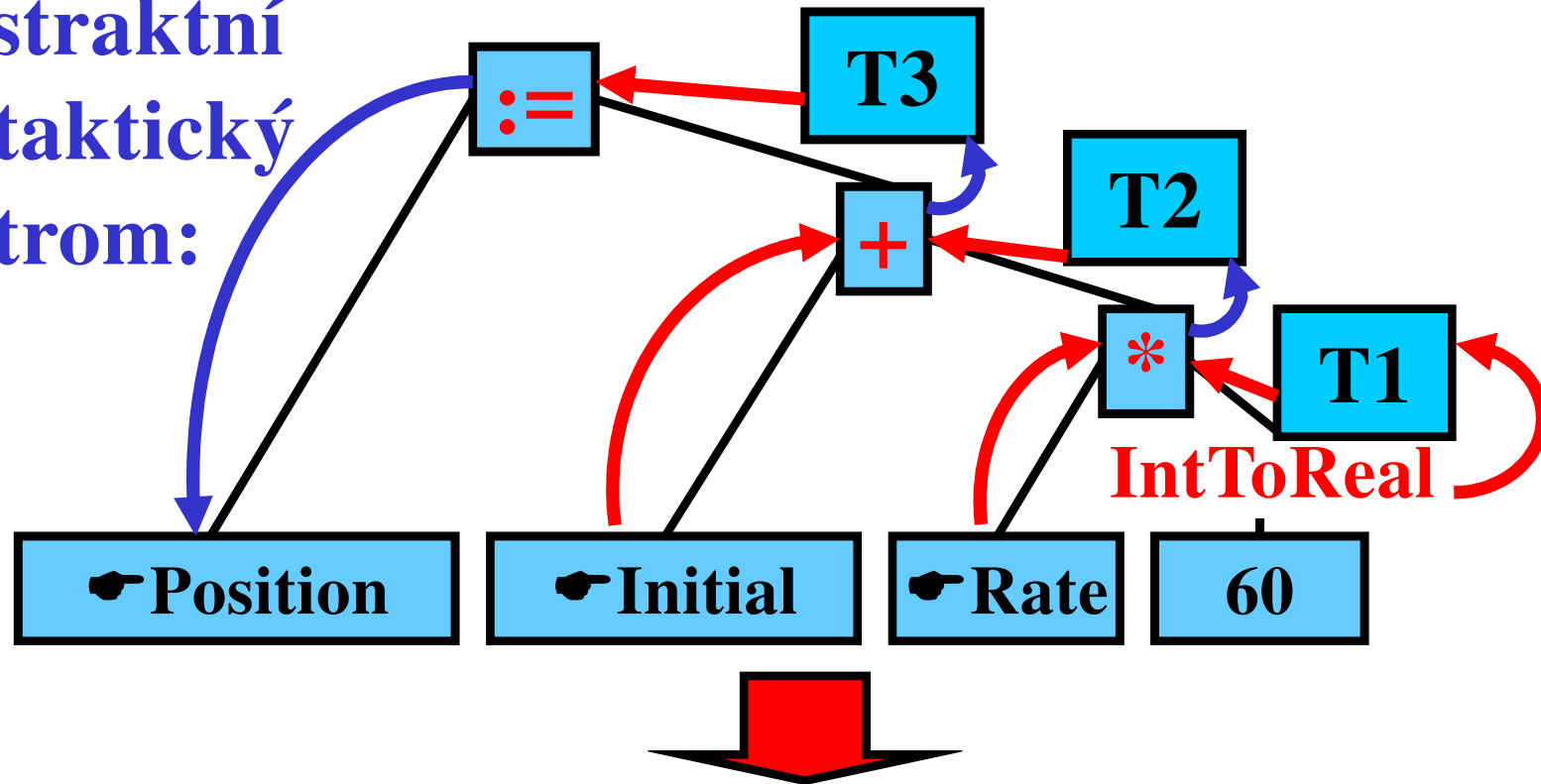
Vnitřní kód:

```

T1 := IntToReal(60)
T2 := ☛Rate * T1
T3 := ☛Initial + T2
  
```

# Generátor vnitřního kódu: Příklad

Abstraktní  
syntaktický  
strom:



Vnitřní kód:

```

T1 := IntToReal(60)
T2 := ⌡Rate * T1
T3 := ⌡Initial + T2
⌡Positon := T3
  
```

# Optimalizátor

- **Input:** Vnitřní kód
  - **Output:** Optimalizovaný vnitřní kód
- 

- **Metoda:**

- Optimalizátor upraví vnitřní kód tak, aby byl efektivnější. Tento upravený kód je nazýván *optimalizovaný vnitřní kód*:

- **Šíření konstanty:**  $(a := 1; b := 2; c := a + b \Rightarrow c := 3)$

*Pozn.: Proměnné  $a$ ,  $b$  nejsou již dále v programu použity*

- **Šíření kopírováním:**  $(b := a; c := b; d := c \Rightarrow d := a)$

*Pozn.: Proměnné  $b$ ,  $c$  nejsou již dále v programu použity*

- **Eliminace mrtvého kódu:**  $(\text{while false do } \dots \Rightarrow \text{odstranit})$

⋮

---

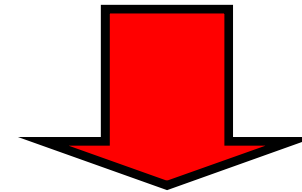
**Pozn.:** Některé překladače optimalizátor nemají

# Optimalizátor: Příklad

Vnitřní kód:

```
T1 := IntToReal(60)
T2 := ⬦Rate * T1
T3 := ⬦Initial + T2
⬦Positon := T3
```

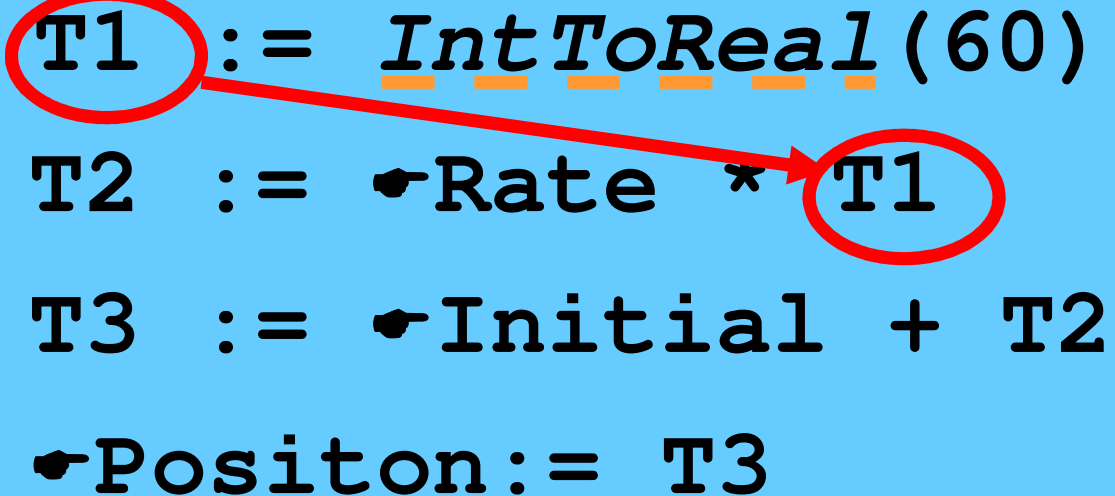
Optimalizovaný vnitřní kód:



# Optimalizátor: Příklad

Vnitřní kód:

```
T1 := IntToReal(60)
T2 := ⌘Rate * T1
T3 := ⌘Initial + T2
⌘Positon := T3
```



Optimalizovaný vnitřní kód:

```
T2 := ⌘Rate * 60.0
```

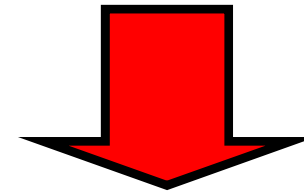
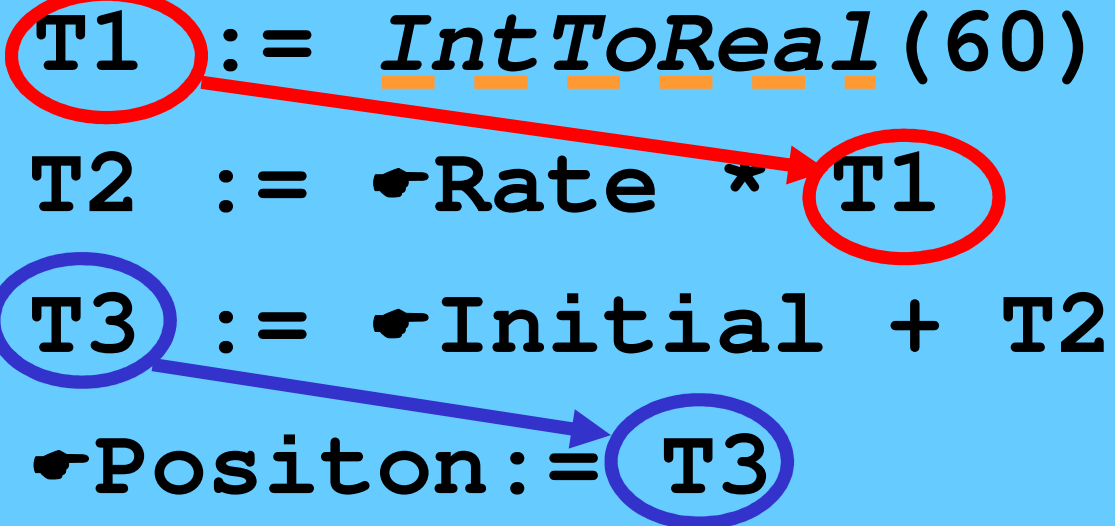




# Optimalizátor: Příklad

Vnitřní kód:

```
T1 := IntToReal(60)
T2 := ⬦Rate * T1
T3 := ⬦Initial + T2
⬦Positon := T3
```



Optimalizovaný vnitřní kód:

```
T2 := ⬦Rate * 60.0
⬦Positon := ⬦Initial + T2
```

# Generátor cílového kódu

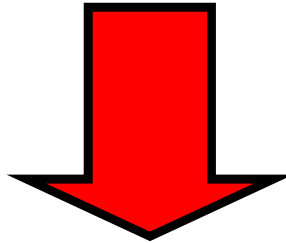
- **Vstup:** Optimalizovaný vnitřní kód
  - **Výstup:** Cílový program
- 
- **Metoda:**
    - Optimalizovaný vnitřní kód je převeden na *cílový program*
    - Cílový program je zapsán v cílovém jazyce
    - V praxi je cílovým jazykem většinou assembler nebo strojový kód

# Generátor cílového kódu: Příklad

## Optimalizovaný vnitřní kód

```
T2 := ⌡Rate * 60.0  
⌡Positon := ⌡Initial + T2
```

Cílový program:



# Generátor cílového kódu: Příklad

## Optimalizovaný vnitřní kód

```
T2 := ⌡Rate * 60.0  
⌡Positon := ⌡Initial + T2
```

## Cílový program:

```
fmov R2, ⌡Rate  
fmul R2, #60.0
```

$R2 \cong T2$

# Generátor cílového kódu: Příklad

## Optimalizovaný vnitřní kód

```
T2 := ⌡Rate * 60.0  
⌡Positon := ⌡Initial + T2
```

## Cílový program:

```
fmov R2, ⌡Rate  
fmul R2, #60.0  
fmov R3, ⌡Initial  
fadd R2, R3  
fmov ⌡Position, R2
```

$R2 \cong T2$

