

Part II.

Introduction to

Compilers

Compiler

- **Input:** Source program
 - **Output:** Target program
-
- **Method:**
 - A compiler reads a *source program* (in source language) and translates them into *target program* (in target language).
 - Source and target programs are *functionally equivalent*.

Structure of Compiler: Phases

`Position := Initial + Rate * 60`

Lexical analyzer

`Id1 := Id2 + Id3 * 60`

Syntax analyzer

```

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

Semantic analyzer

```

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

Intermediate code generator

```

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

Optimizer

```

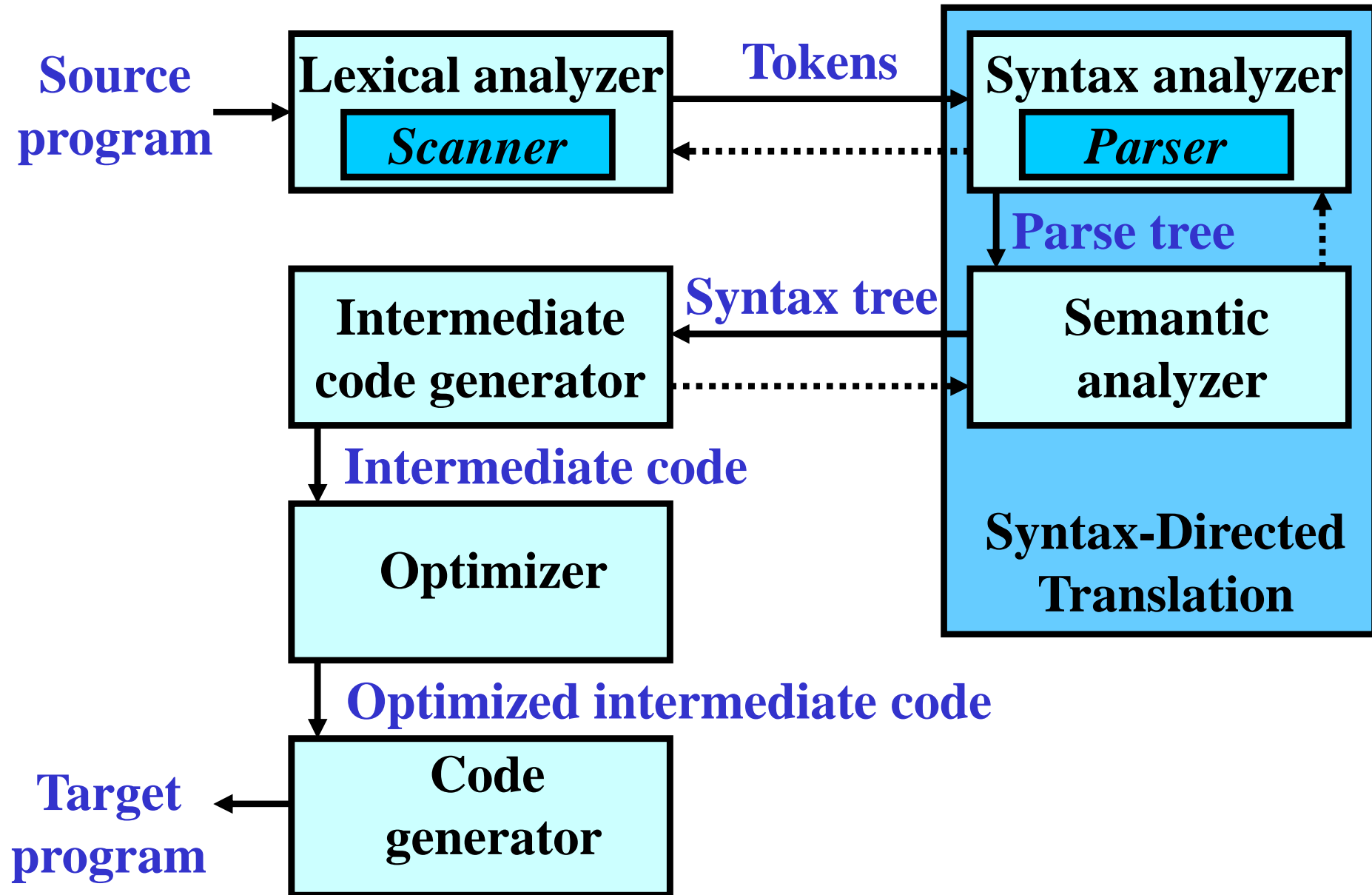
T1  := Id3 * 60.0
Id1 := Id2 + T1
  
```

Code generator

```

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

Structure of Compiler: Construction



Languages and Compilers

Theoretical view.

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

Question: $aabb \in L$?

Practical view.

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

L_{Pascal} = Programming Language Pascal

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

YES: Program is **OK** \Rightarrow
Create a target program

NO: Program is **not OK** \Rightarrow
Handle the errors

Lexical analyzer (Scanner)

- **Input:** Source program
 - **Output:** String of tokens
-
- **Method:**
 - Source program is broken into *lexemes* = logically cohesive lexical entities – (identifiers, numbers, key-words, operators,...)
 - Lexemes are represented by uniform *tokens*
 - Some tokens have *attributes*
-

Lexical analyzer: Example

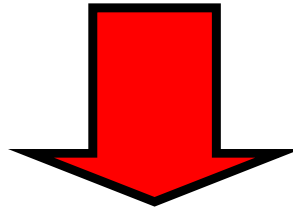
Source program:

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

Lexical analyzer: Example

Source program:

`Position := Initial + Rate * 60`



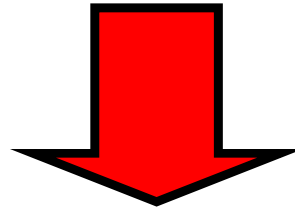
Lexemes:

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

Lexical analyzer: Example

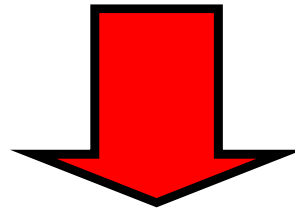
Source program:

`Position := Initial + Rate * 60`



Lexemes:

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



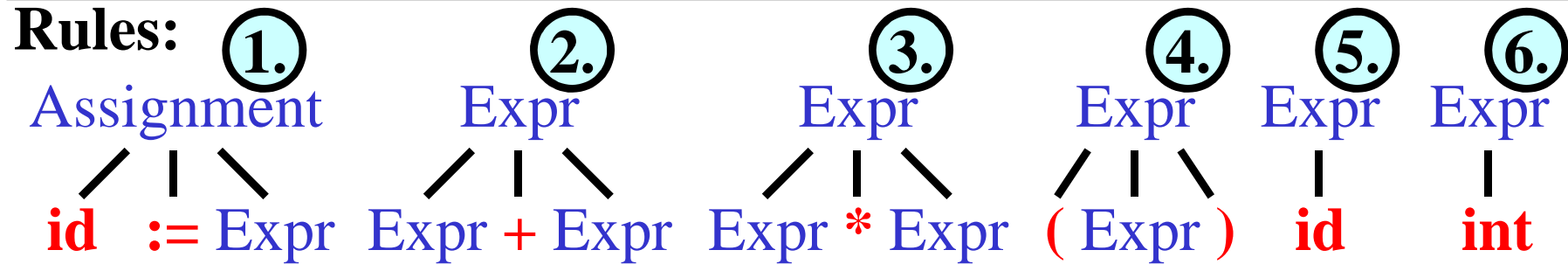
Tokens:

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

Syntax analyzer (Parser)

- **Input:** String of tokens
 - **Output:** Parse tree
-
- **Method:**
 - Parser verifies that the string of tokens represents a syntactically well-formed program
 - If it finds a *parse tree* for the string, it is correct; otherwise, it is not
 - Construction of tree is based on grammatical rules
 - Two approach: top-down and bottom-up
-

Syntax analyzer: Example



**Parse
Tree:**

Assignment

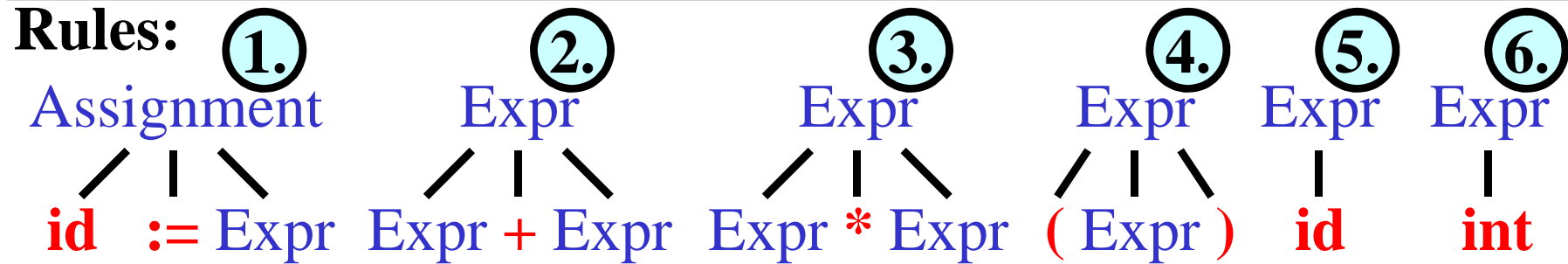
Task:

Create a parse tree
from a grammatical rules !



Tokens

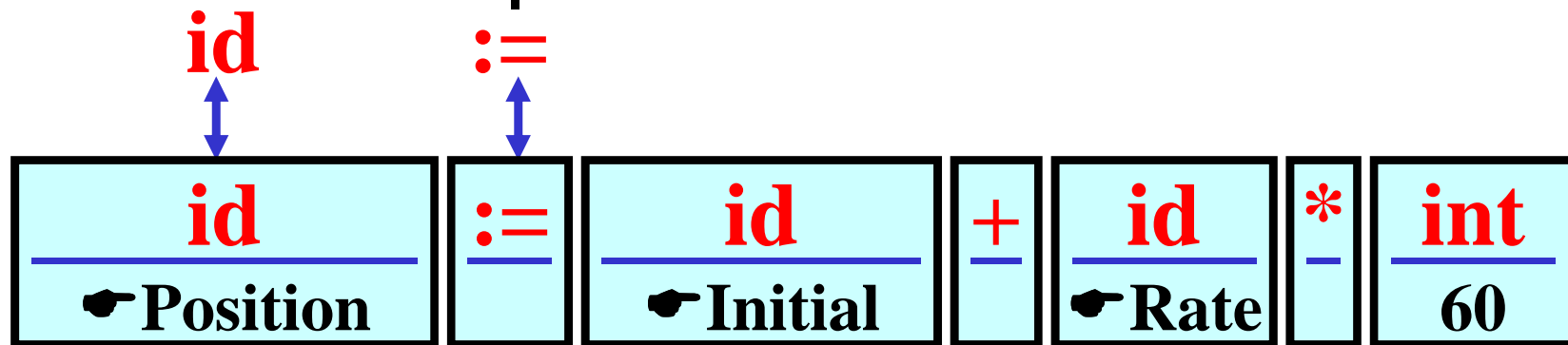
Syntax analyzer: Example



Parse Tree:

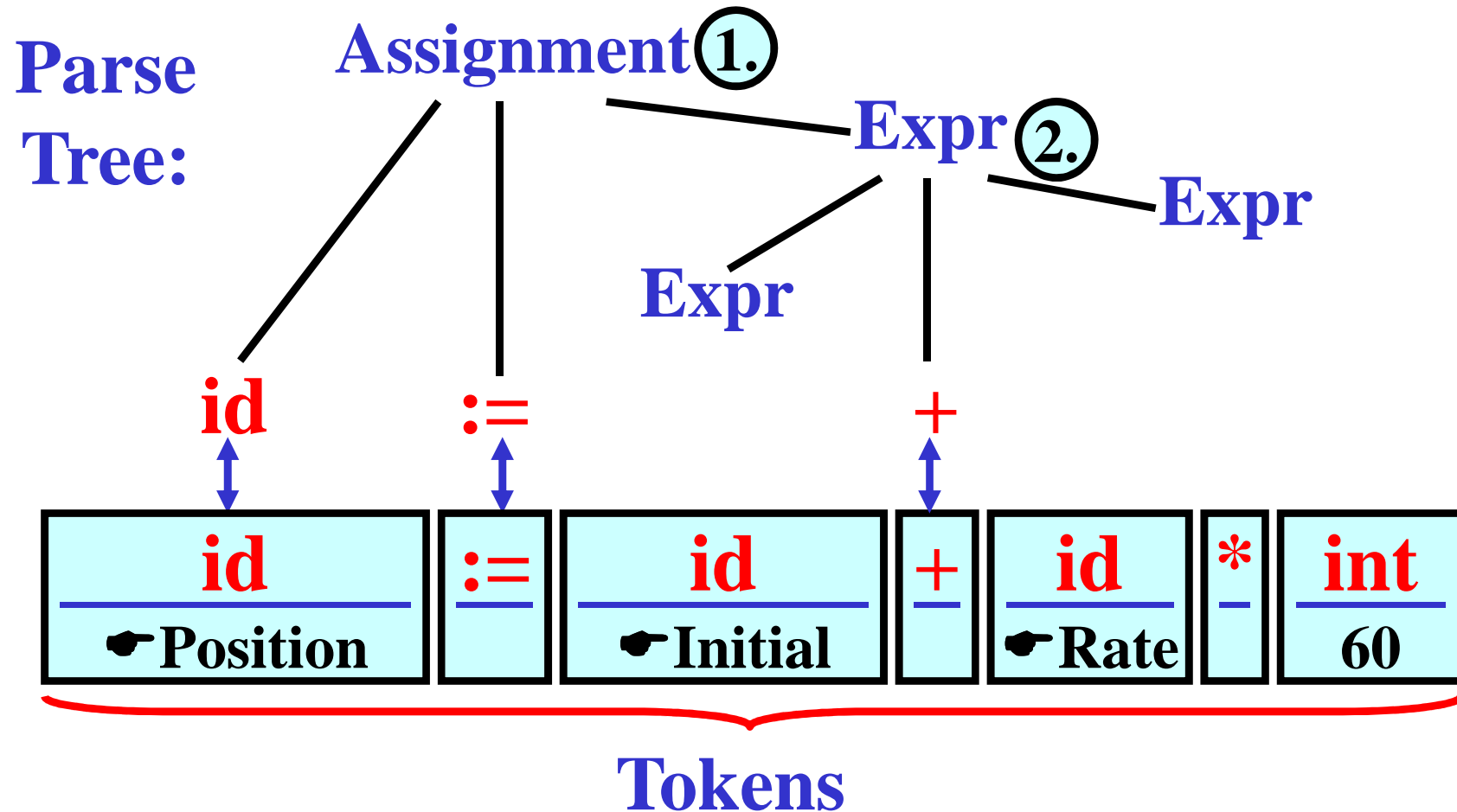
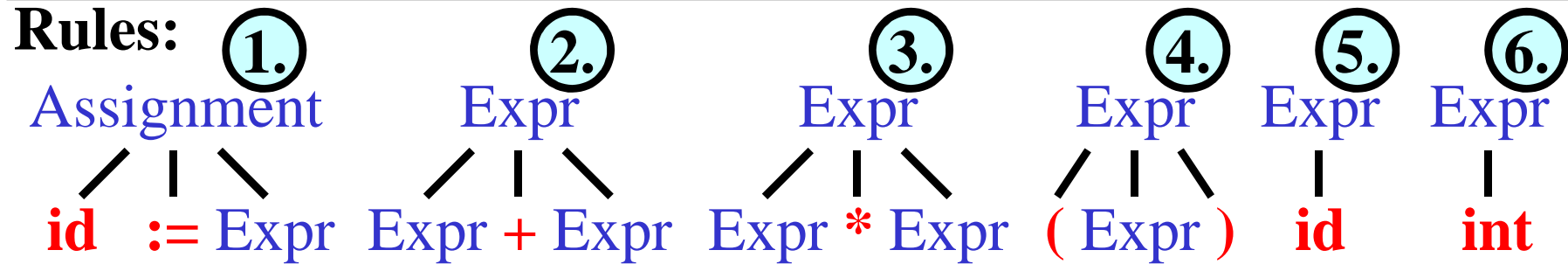
Assignment ①.

Expr

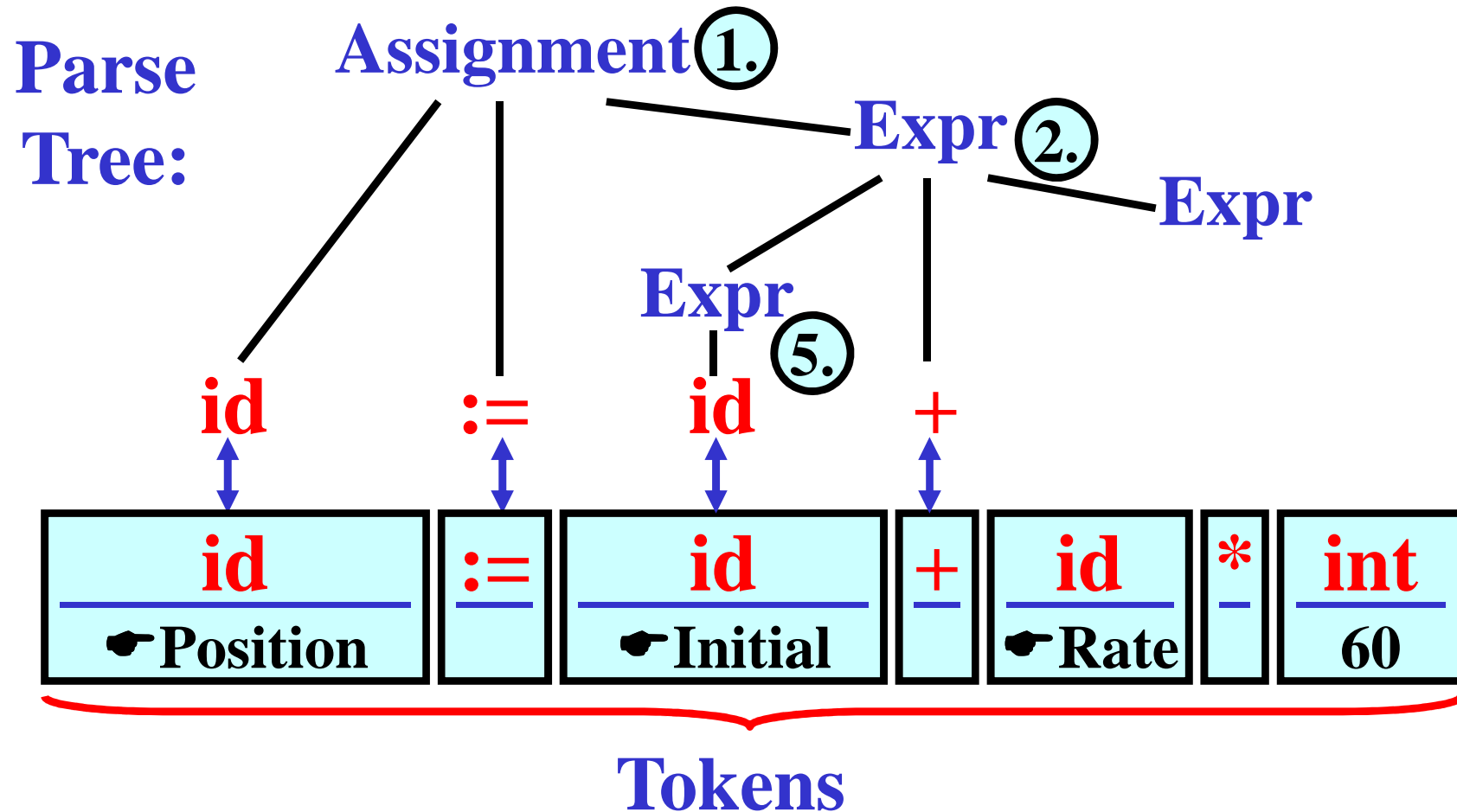
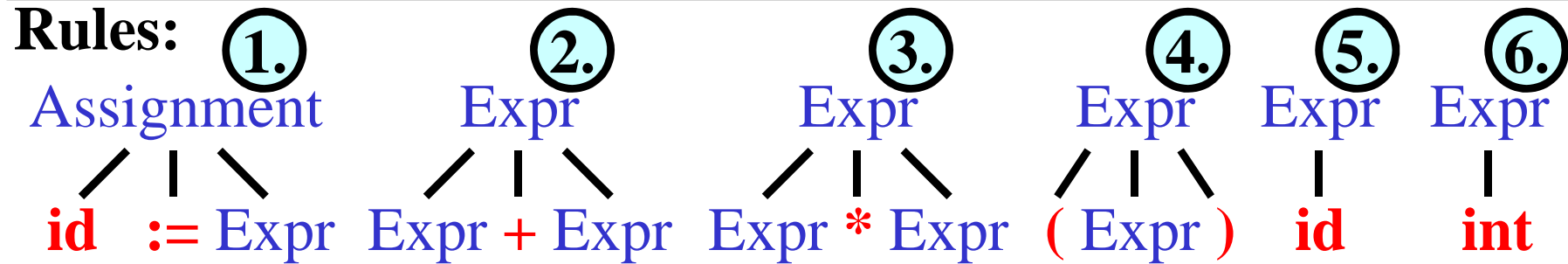


Tokens

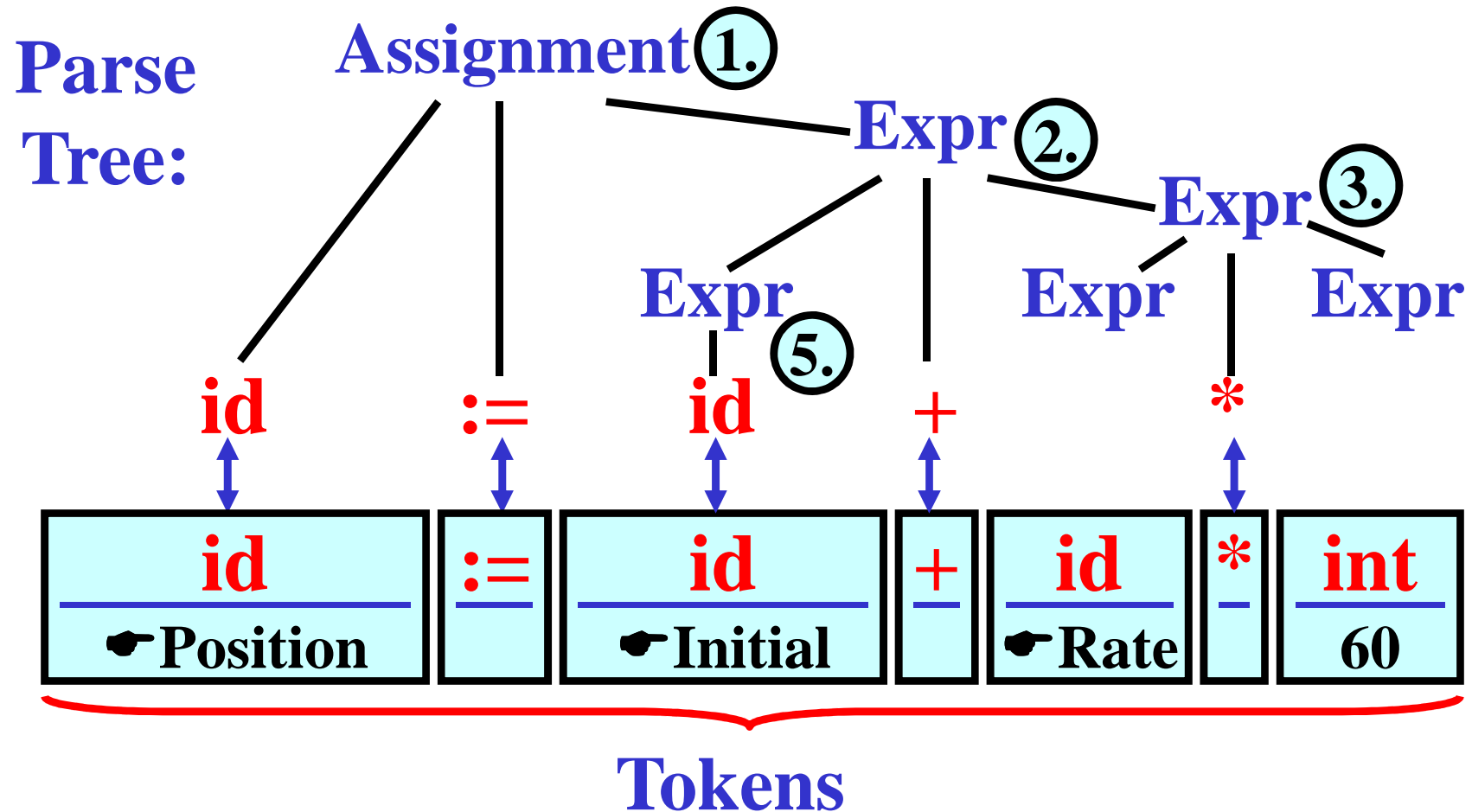
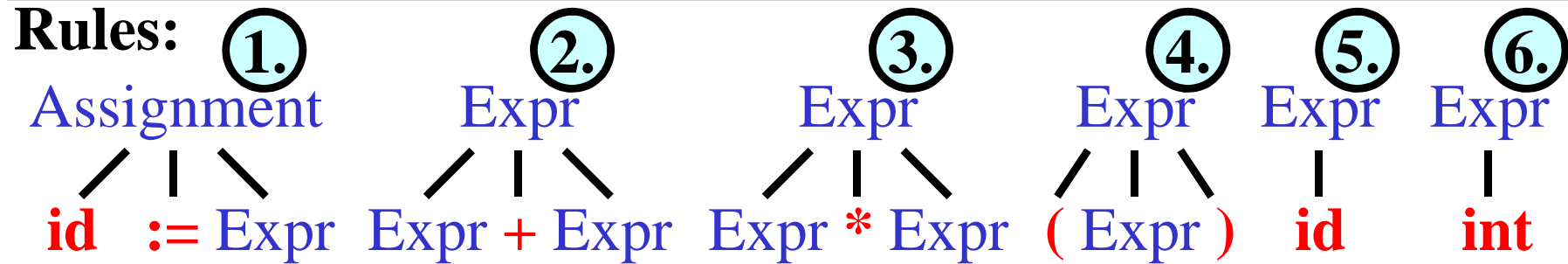
Syntax analyzer: Example



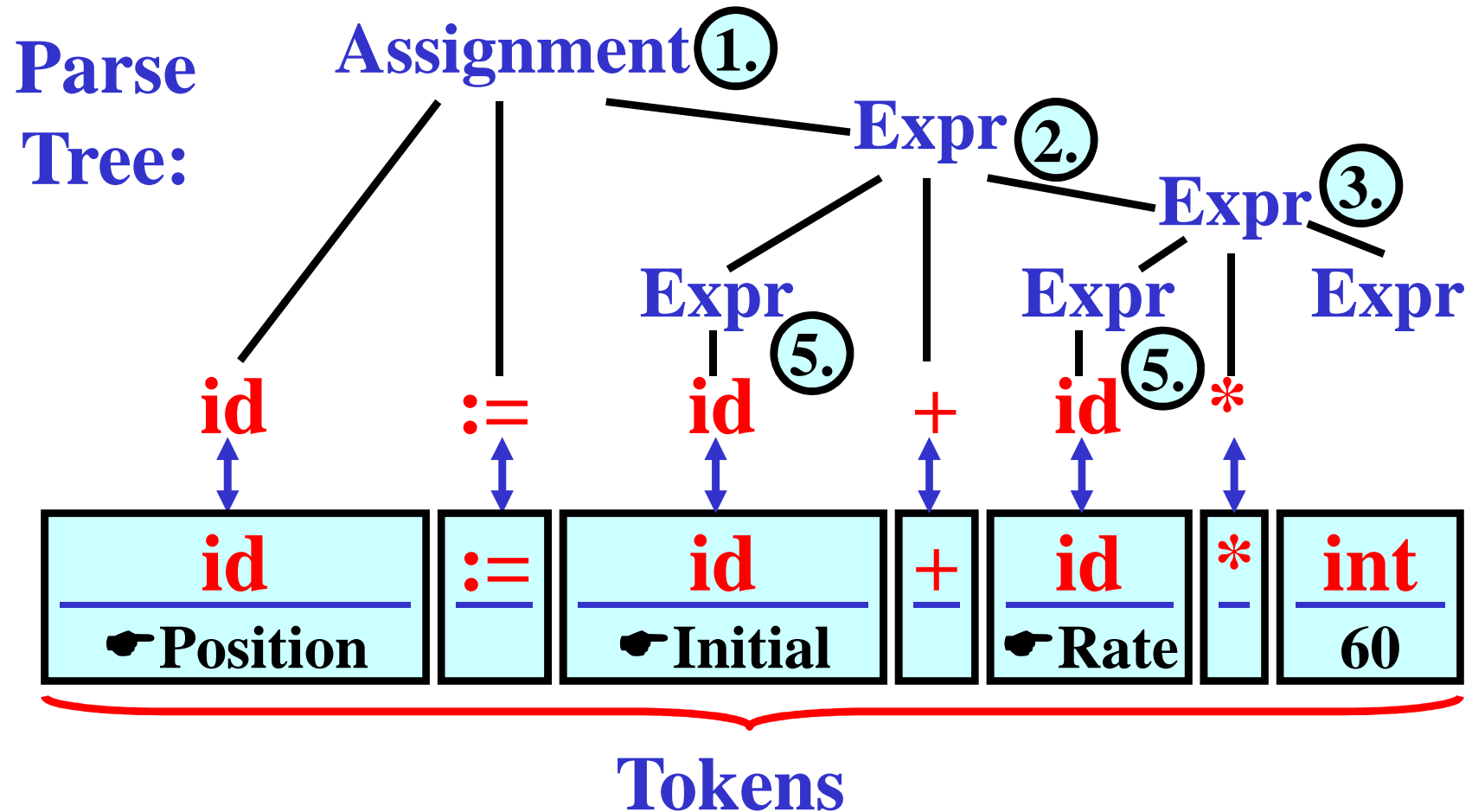
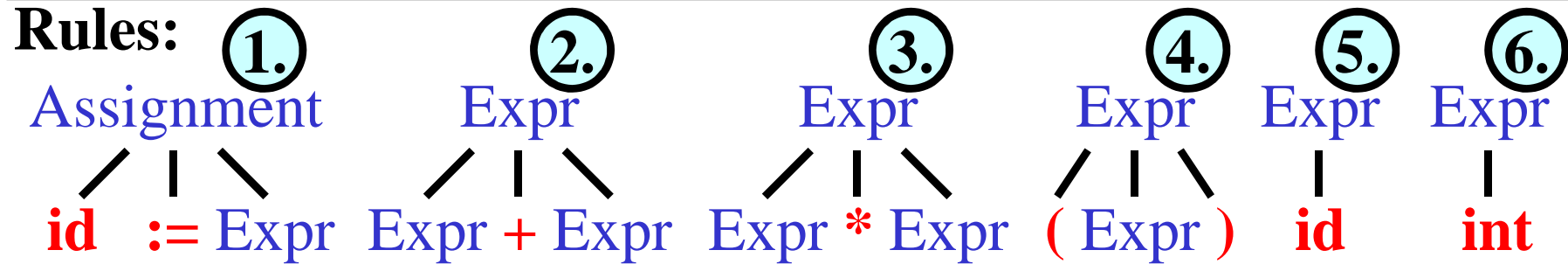
Syntax analyzer: Example



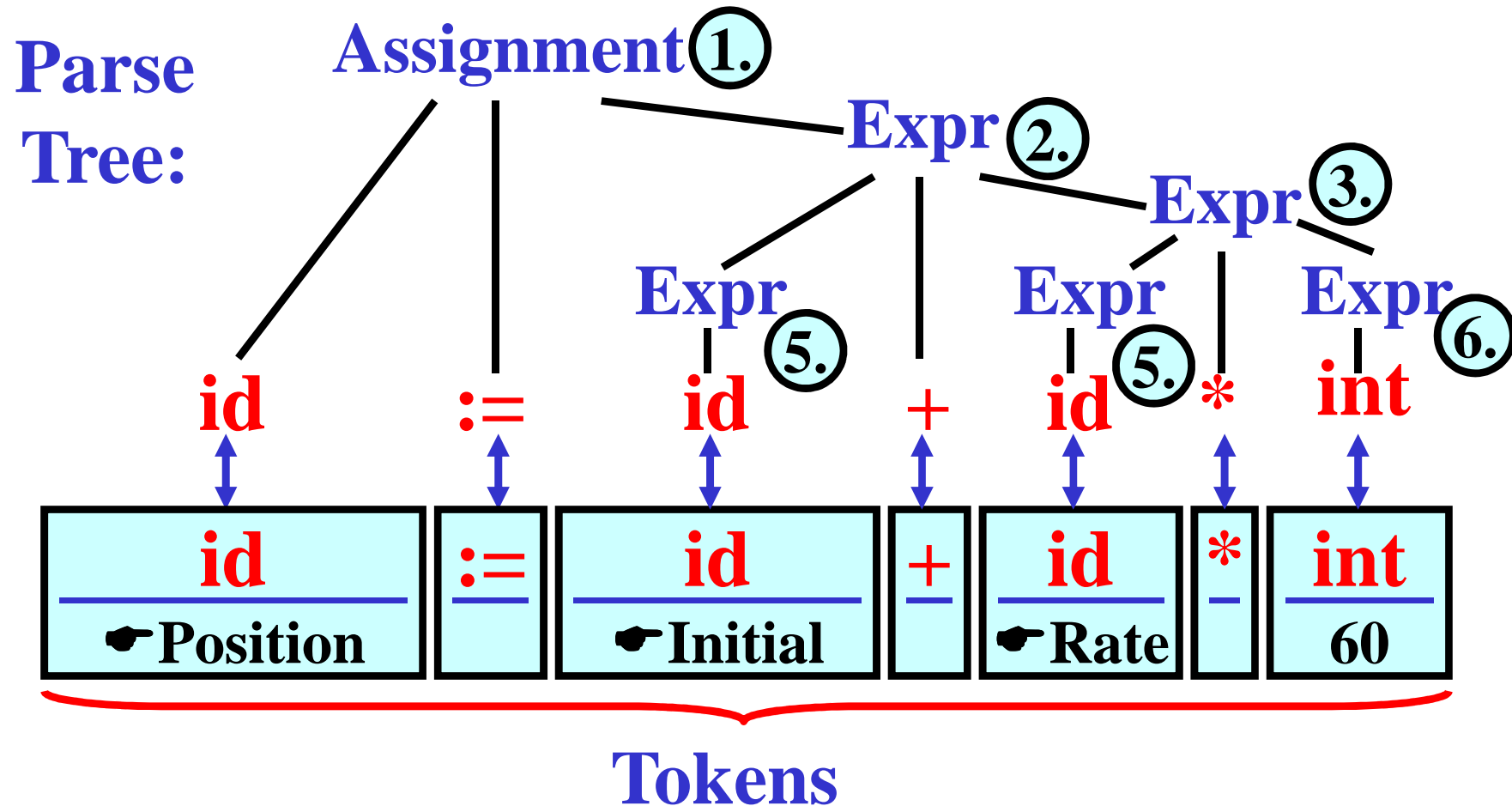
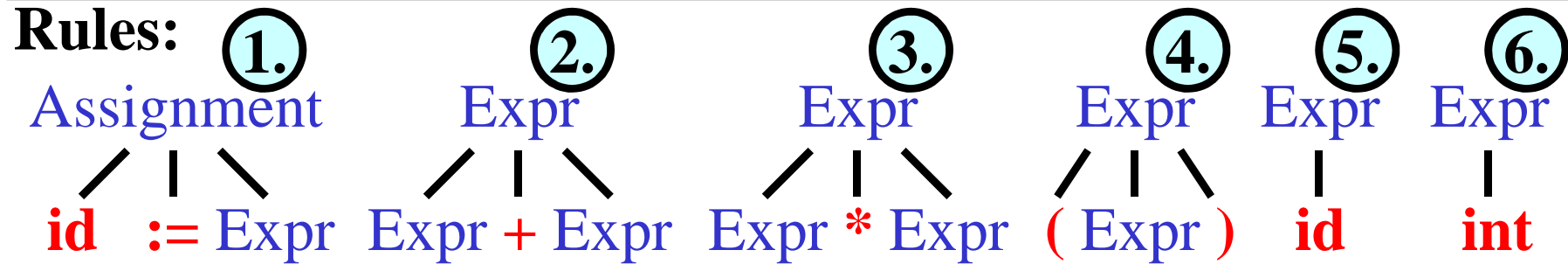
Syntax analyzer: Example



Syntax analyzer: Example



Syntax analyzer: Example

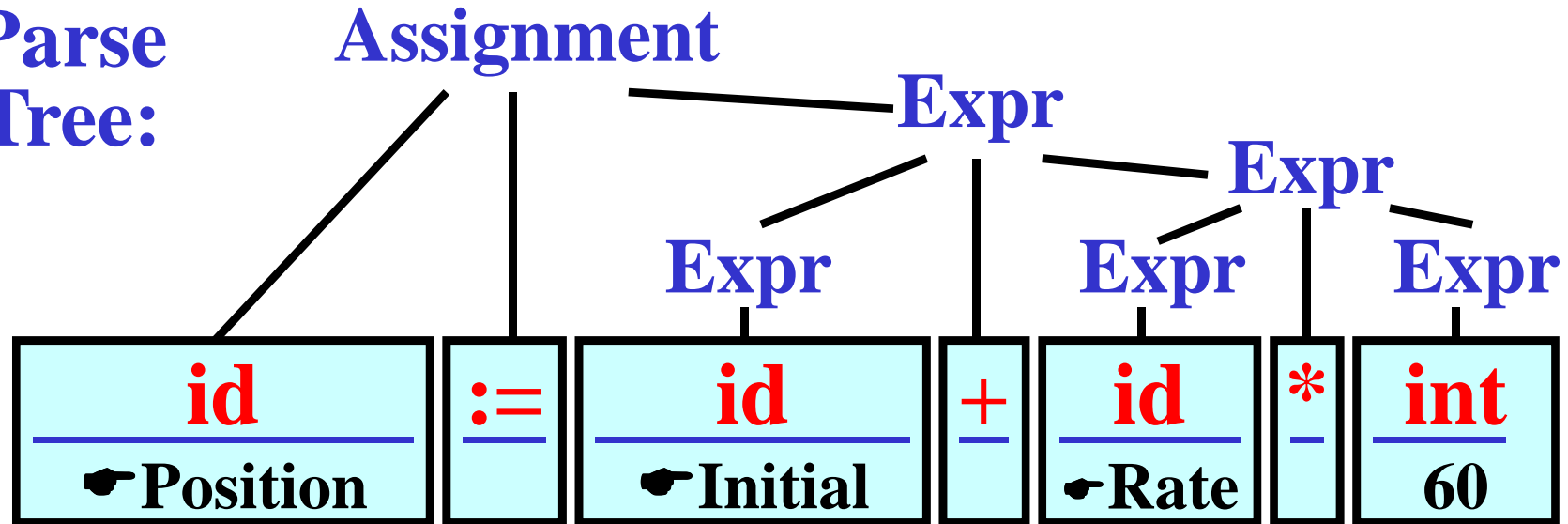


Semantic analyzer

- **Input:** Parse tree
 - **Output:** Abstract syntax tree
-
- **Method:**
 - Semantic analyzer checks semantic aspects:
 - *type checking*, which may imply conversions (for example int-to-real)
 - *checking declaration of variables*
 - **Syntax-Directed Translation:**
Parser controls:
 - Semantic actions
 - Generation of syntax tree

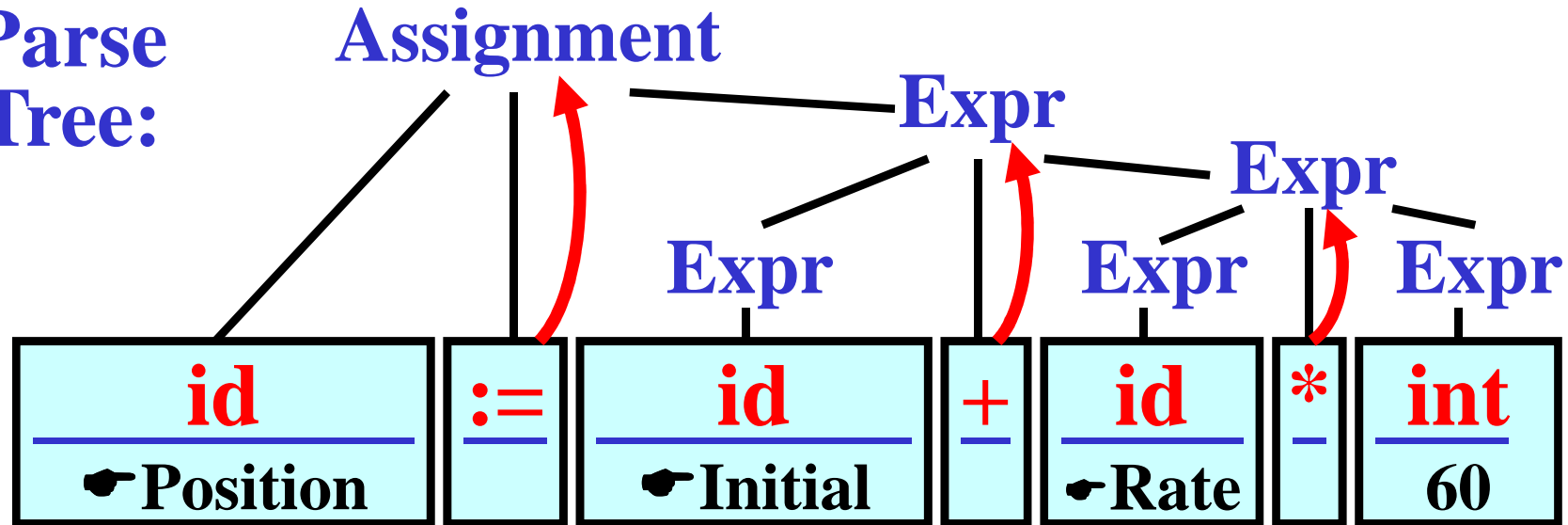
Syntax-Directed Translation: Example

Parse
Tree:



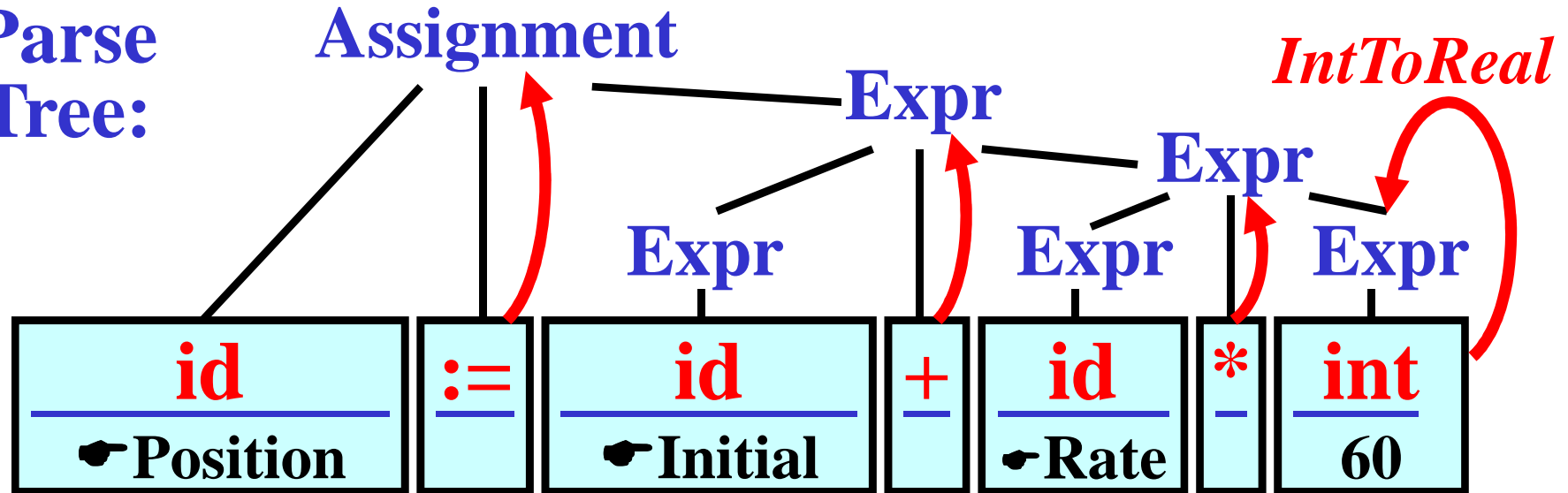
Syntax-Directed Translation: Example

Parse
Tree:



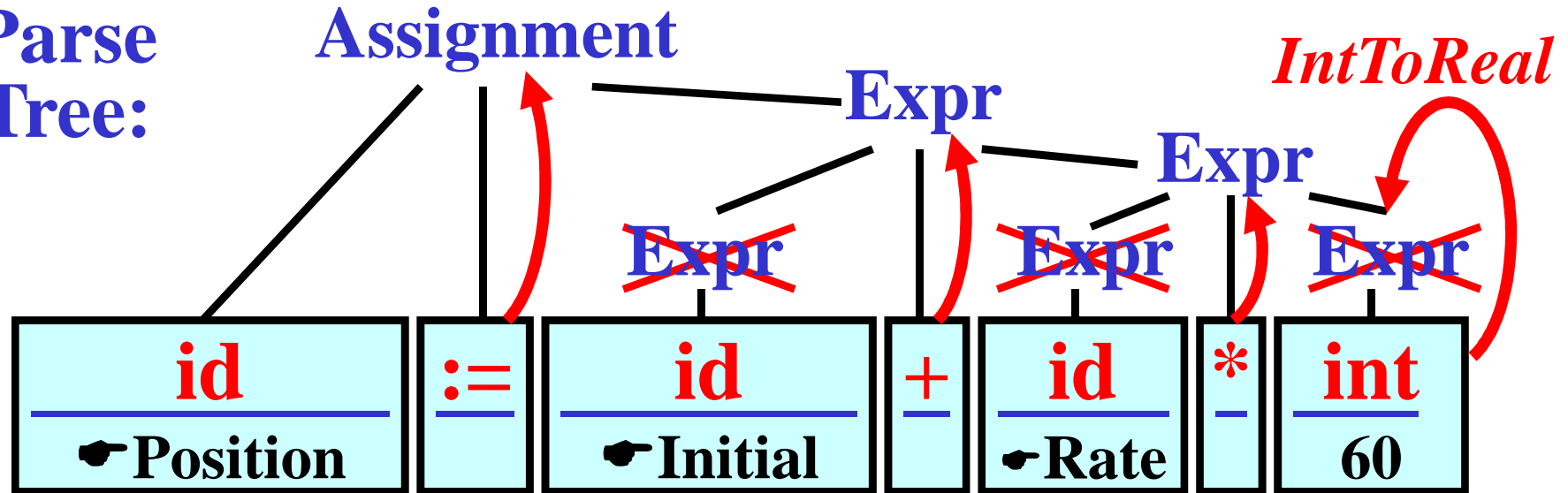
Syntax-Directed Translation: Example

Parse
Tree:

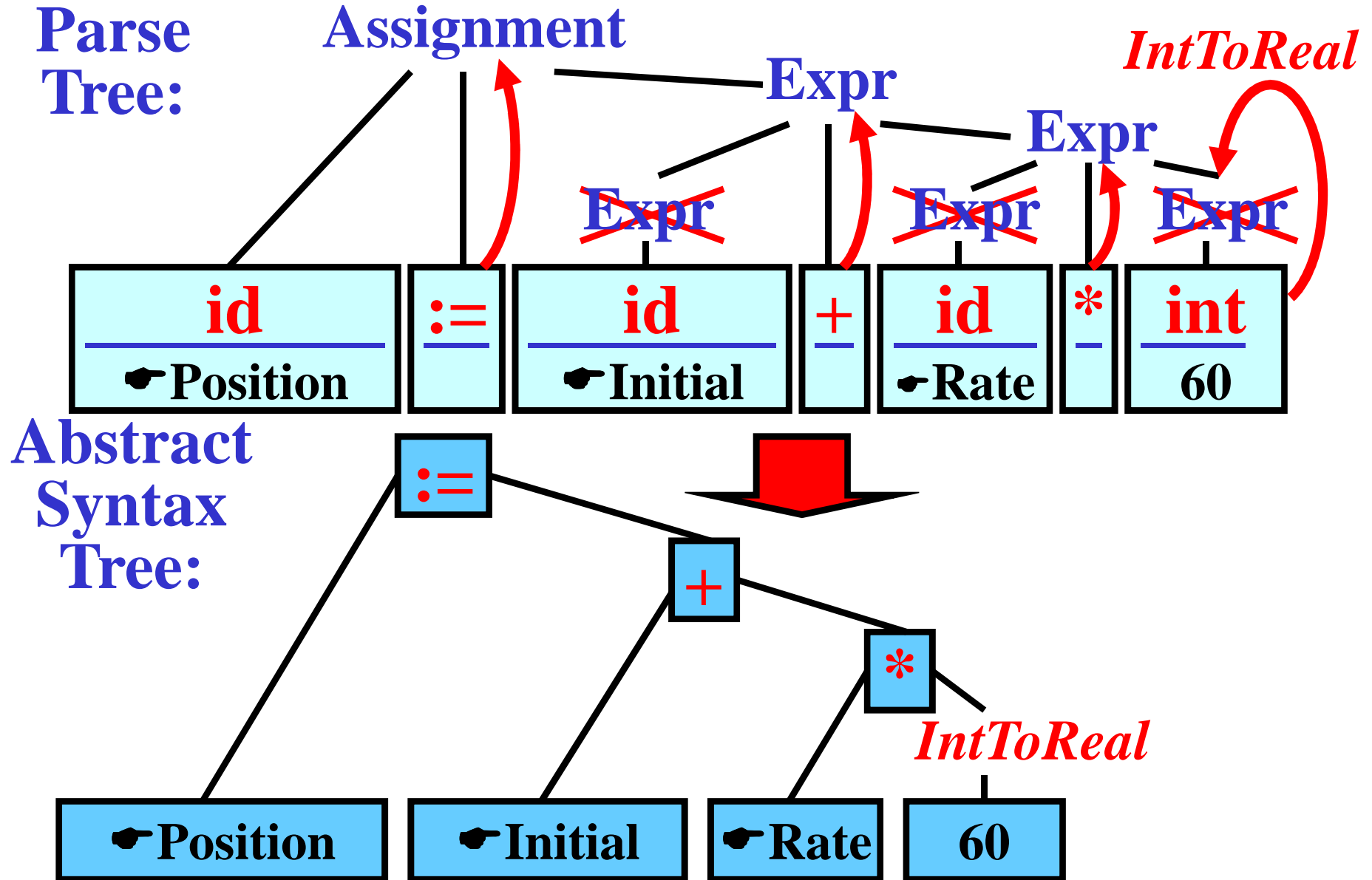


Syntax-Directed Translation: Example

Parse
Tree:



Syntax-Directed Translation: Example

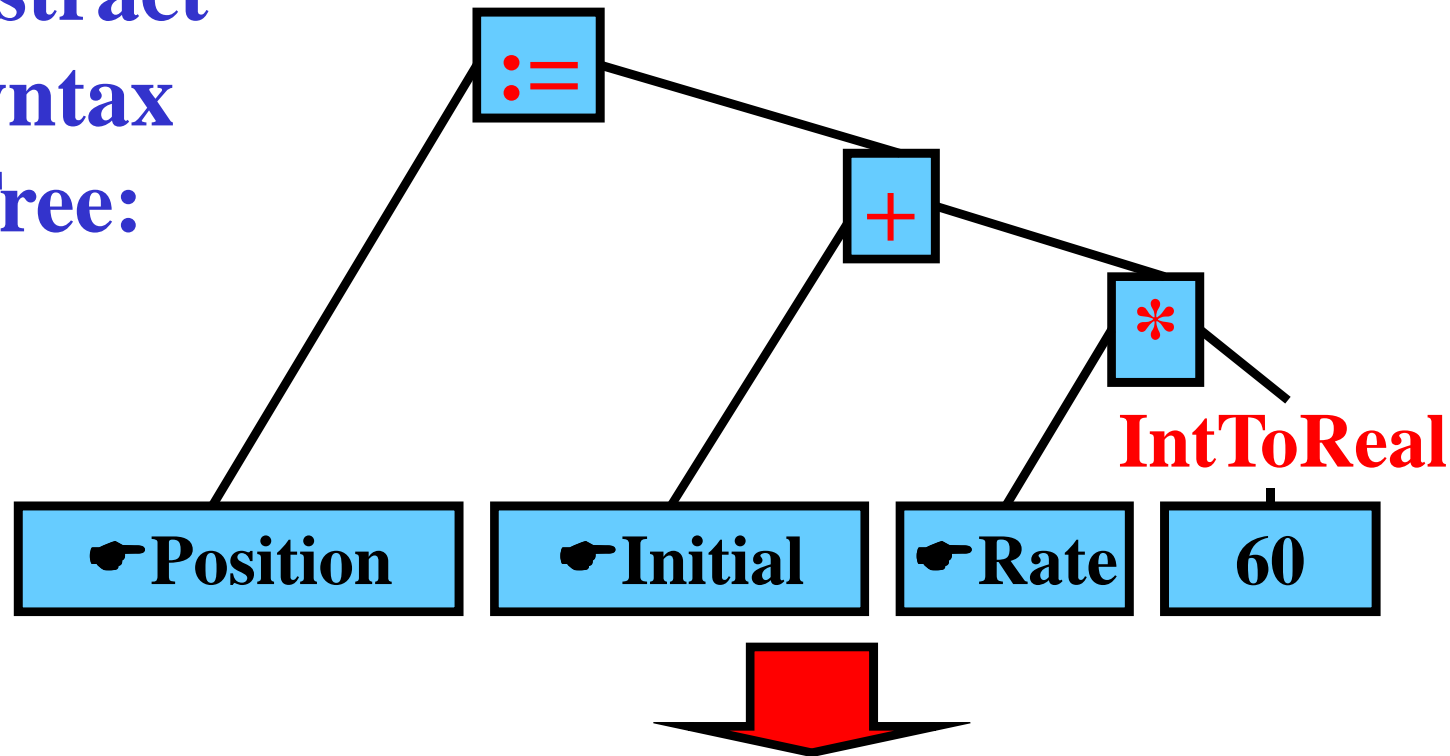


Intermediate code generator

- **Input:** Abstract syntax tree
 - **Output:** Intermediate code
-
- **Method:**
 - Intermediate code generator produces the internal version of target program called *intermediate code* for these reasons:
 - uniformity
 - direct translation to target program is difficult and “rough”
 - optimization

Intermediate code generator: Example

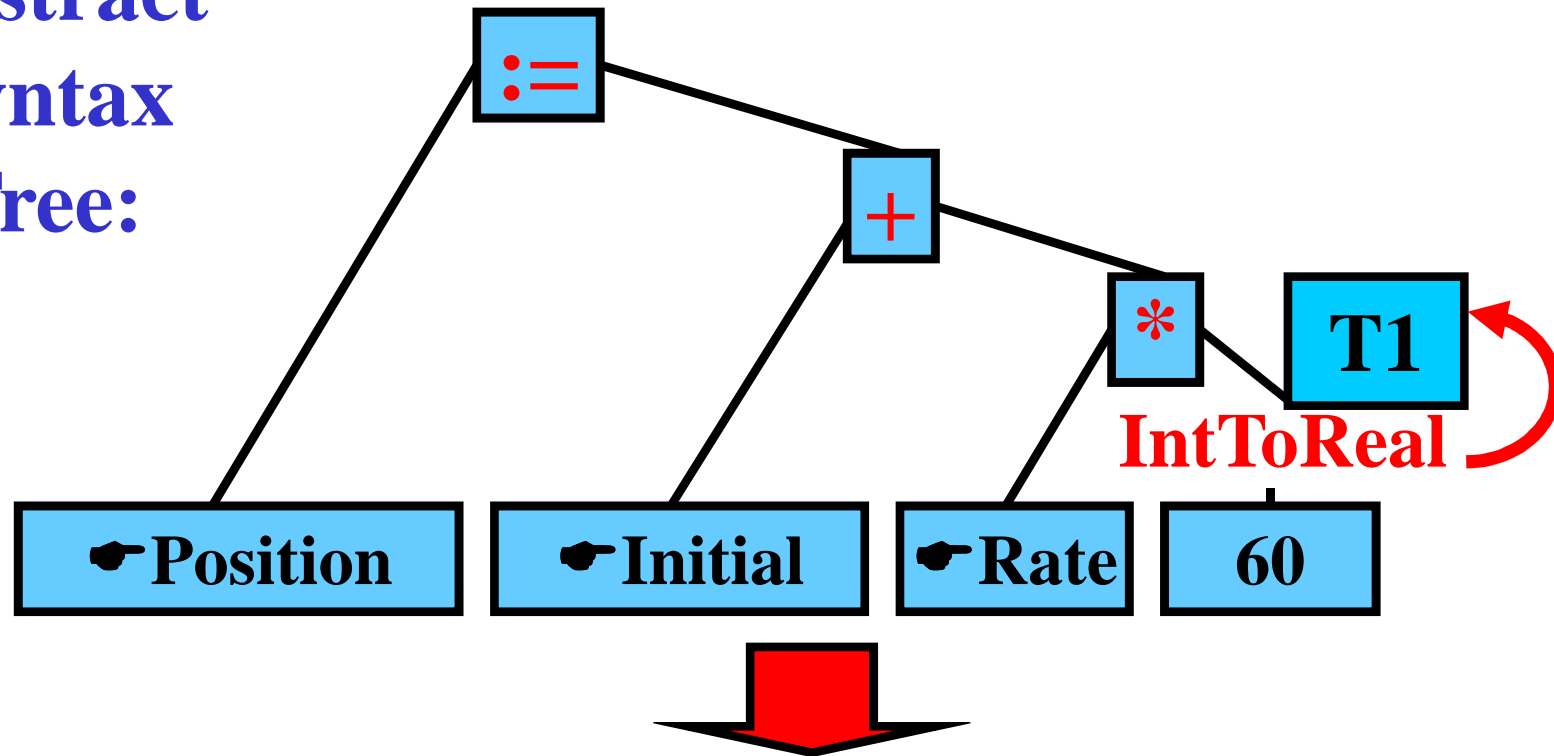
**Abstract
Syntax
Tree:**



**Intermediate
code:**

Intermediate code generator: Example

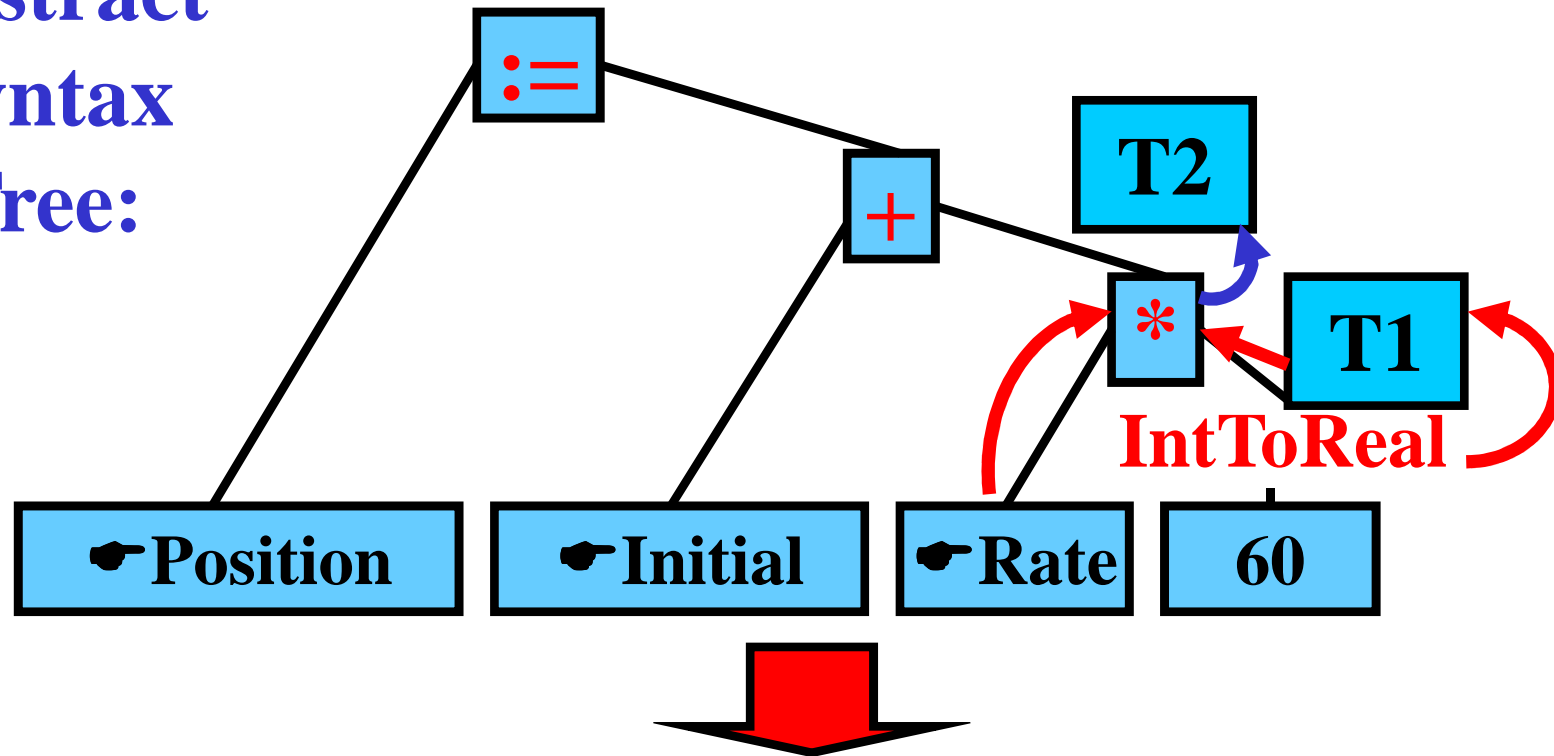
Abstract
Syntax
Tree:



Intermediate code: `T1 := IntToReal(60)`

Intermediate code generator: Example

Abstract
Syntax
Tree:



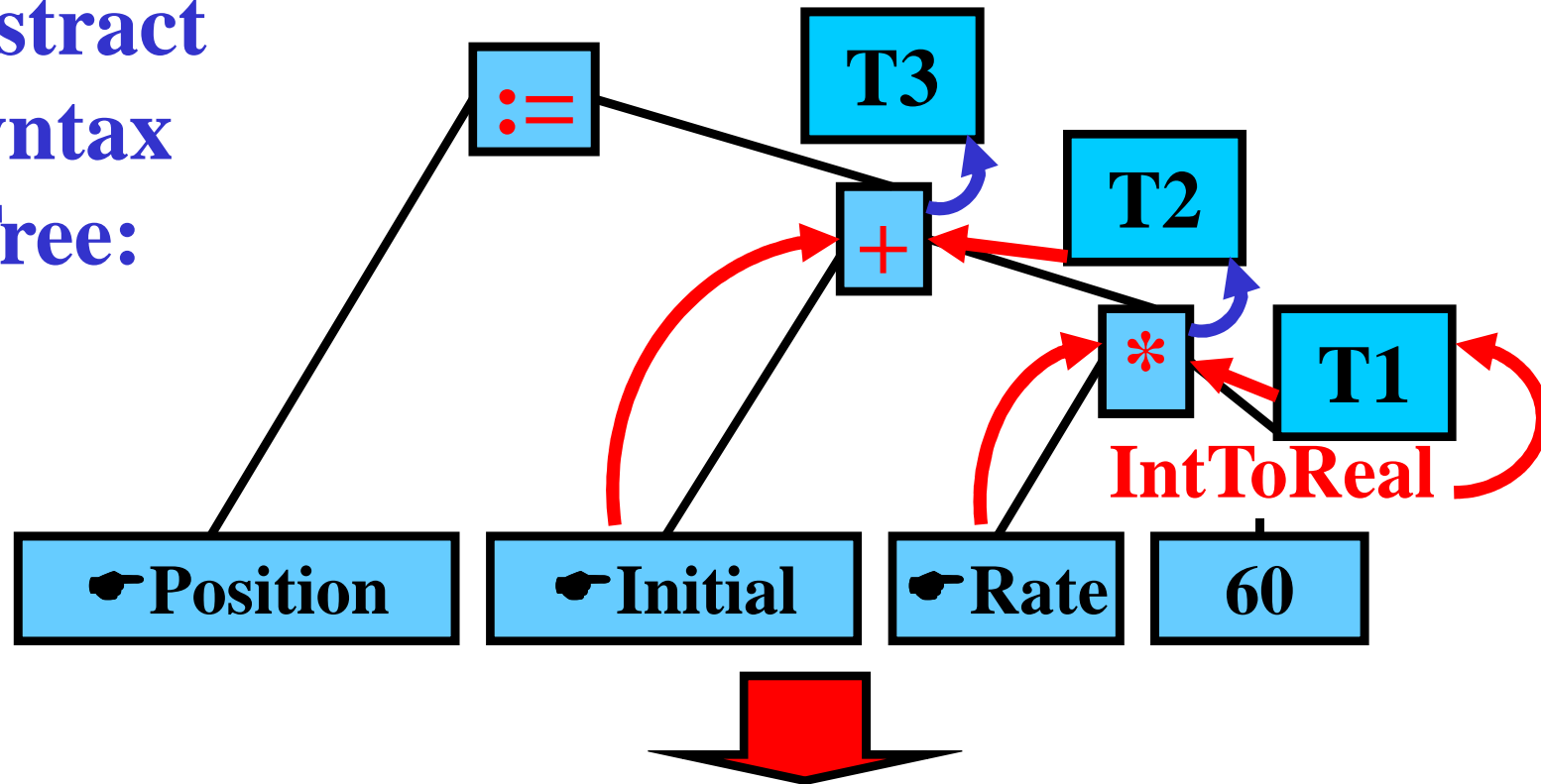
Intermediate code:

```

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

Intermediate code generator: Example

Abstract
Syntax
Tree:



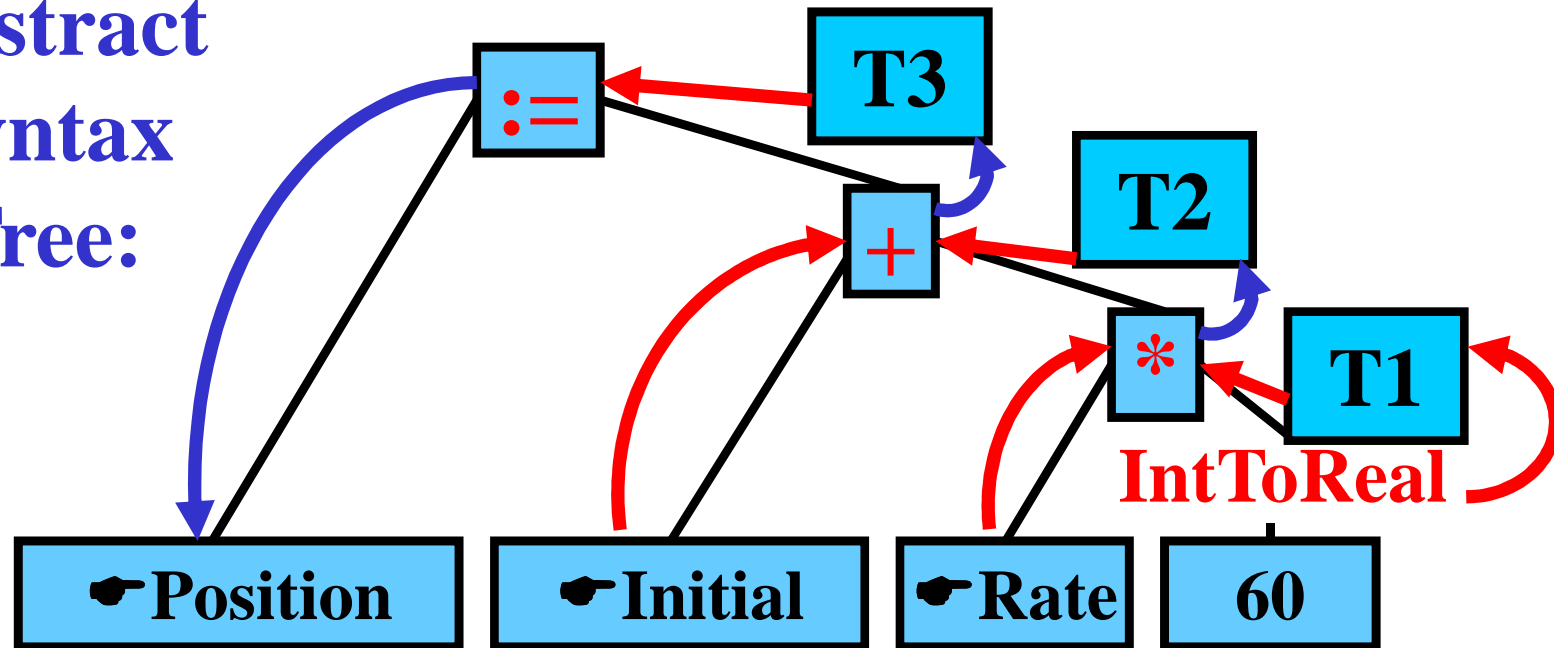
Intermediate
code:

```

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

Intermediate code generator: Example

Abstract
Syntax
Tree:



Intermediate
code:

```

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

Optimizer

- **Input:** Intermediate code
 - **Output:** Optimized intermediate code
-
- **Method:**
 - Optimizer makes more efficient version of intermediate code called *optimized intermediate code*:
 - **Constant propagation:** $(a := 1; b := 2; c := a + b \Rightarrow c := 3)$
Note: Variables a, b have no next use
 - **Copy propagation:** $(b := a; c := b; d := c \Rightarrow d := a)$
Note: Variables b, c have no next use
 - **Dead code elimination:** $(\text{while false do } \dots \Rightarrow \text{nothing})$
 - \vdots
-

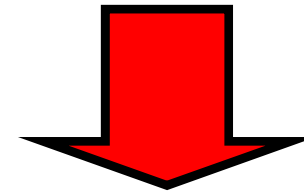
Note: Some compilers have no optimizer

Optimizer: Example

Intermediate
code:

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

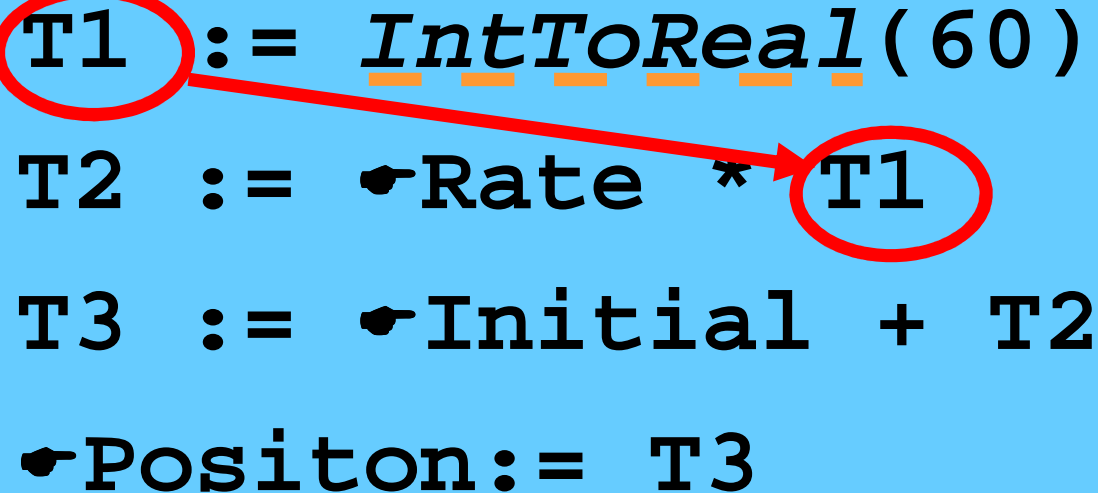
Optimized intermediate code:



Optimizer: Example

Intermediate
code:

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



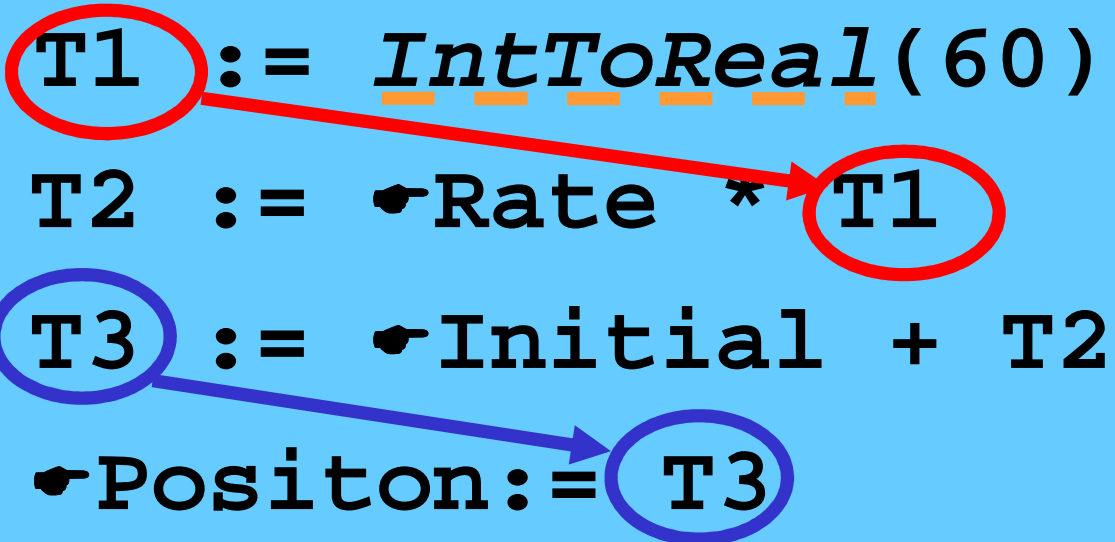
Optimized intermediate code:

```
T2 := ⌘Rate * 60.0
```


Optimizer: Example

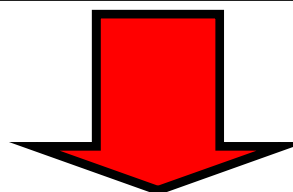
Intermediate
code:

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



Optimized intermediate code:

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



Code Generator

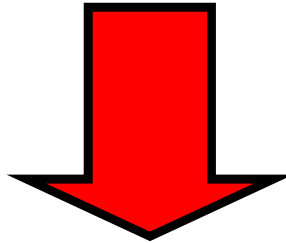
- **Input:** Optimized intermediate code
 - **Output:** Target program
-
- **Method:**
 - Optimized intermediate code is converted to *target program*
 - Target program is written in target language
 - In reality, target language is assembly or machine language

Code Generator: Example

Optimized intermediate code:

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

Target program:



Code Generator: Example

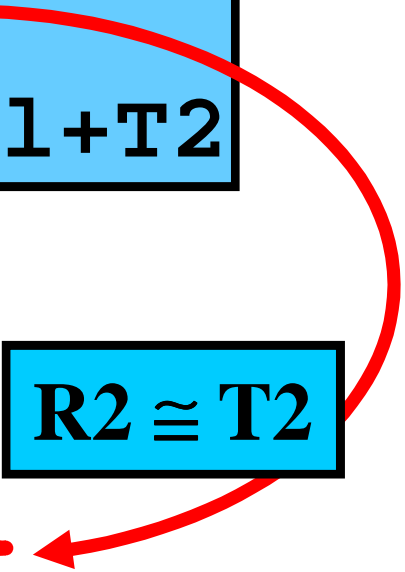
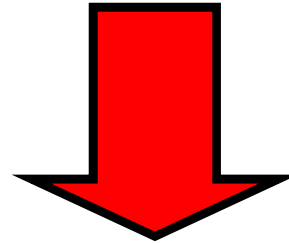
Optimized intermediate code:

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

Target program:

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

$R2 \cong T2$



Code Generator: Example

Optimized intermediate code:

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

Target program:

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

$R2 \cong T2$

