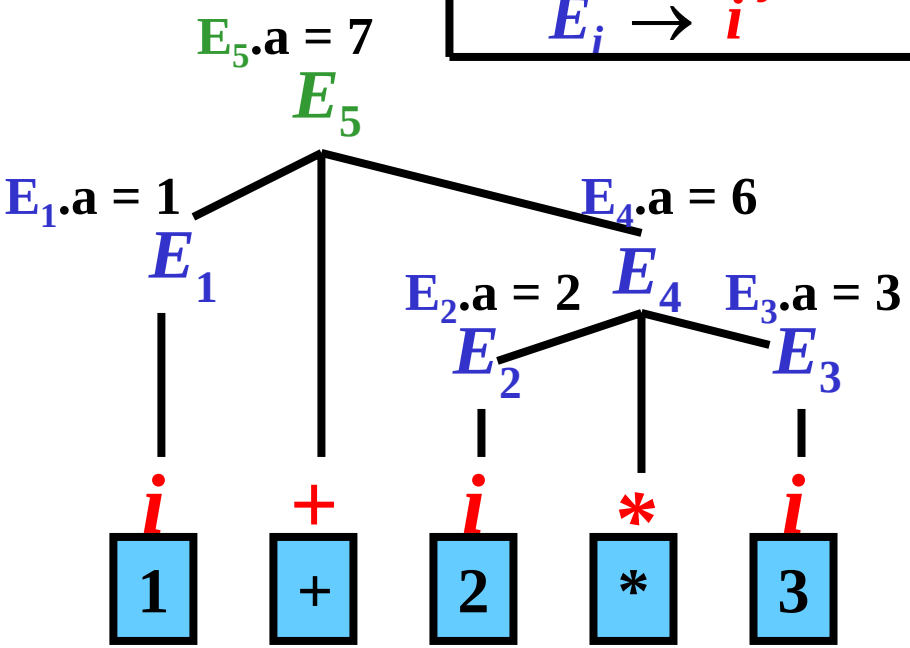
Part VII.
Syntax Directed Translation and Intermediate Code

Syntax-Directed Translation

Gist: *Semantic actions* are attached to gramatical rules. Most importantly, these actions make intermediate code generation and type checking.

Example:

Rule:	Semantic Action:
$E_i \rightarrow E_j + E_k$	{ E_i.a := E_j.a + E_k.a}
$E_i \rightarrow E_j * E_k$	{ E_i.a := E_j.a * E_k.a}
$E_i \rightarrow (E_j)$	{ E_i.a := E_j.a }
$E_i \rightarrow i$	{ E_i.a := i.val }



Rule:	Action:
$E_1 \rightarrow i$	E_1.a := i.val
$E_2 \rightarrow i$	E_2.a := i.val
$E_3 \rightarrow i$	E_3.a := i.val
$E_4 \rightarrow E_2 * E_3$	E_4.a := E_2.a*E_3.a
$E_5 \rightarrow E_1 + E_4$	E_5.a := E_1.a+E_4.a

Intermediate Code: Three–Address Code

- Instruction in **three–address code** (**3AC**) has the form:

(o, ☛a, ☛b, ☛r)

- **o** – operator (+, –, *, …)
- **a** – operand 1 (☛***a*** = address of ***a***)
- **b** – operand 2 (☛***b*** = address of ***b***)
- **r** – result (☛***r*** = address of ***r***)

Examples:

```
(:=   ,  a,   , c )  …  c := a
(+    ,  a, b, c )  …  c := a + b
(not  ,  a,   , b )  …  b := not(a)
(goto,   ,   , L1)  …  goto L1
(goto,   a,   , L1)  …  if a = true then goto L1
(lab  , L1,   ,   )  …  label L1:
```

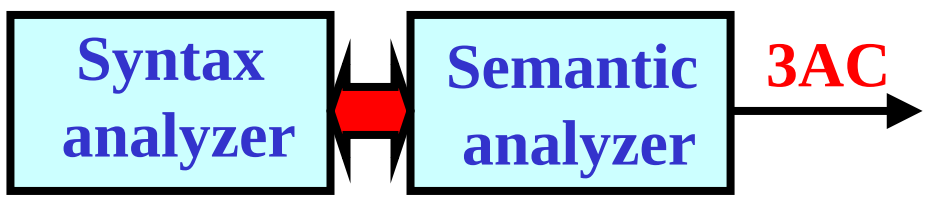
Syntax-Directed Generation of 3AC

Basic approaches:

1) Parser directs the creation of an ***abstract-syntax tree* (AST)** , which is then converted to **3AC**.

2) Parser directs the creation of a ***postfix notation* (PN)** , which is then converted to **3AC**.

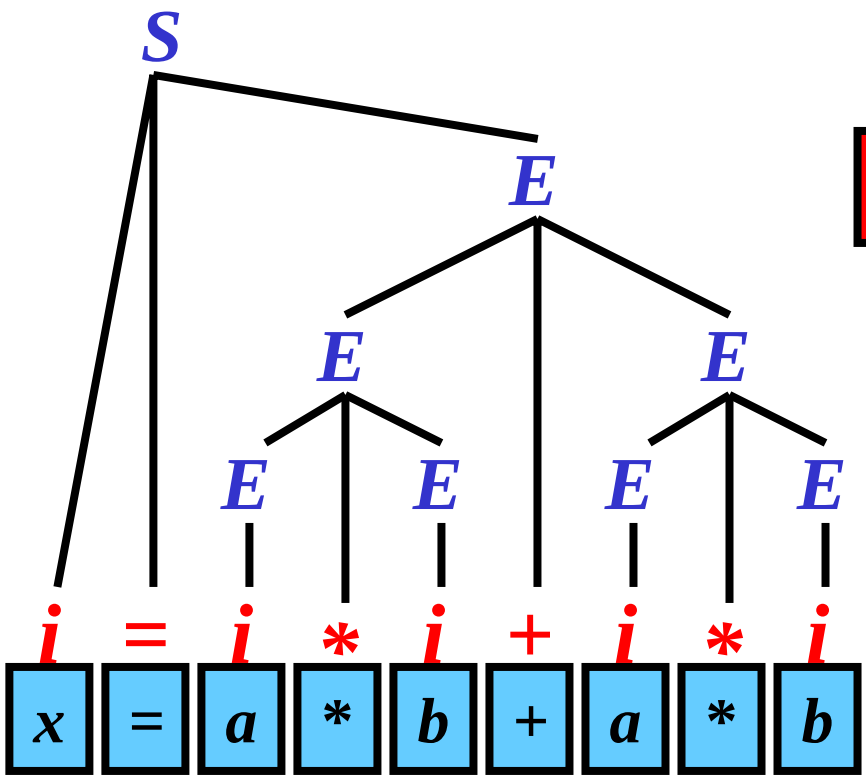
3) A parser directs the creation of **3AC.**



From a Parse Tree (PT) to an AST: Example

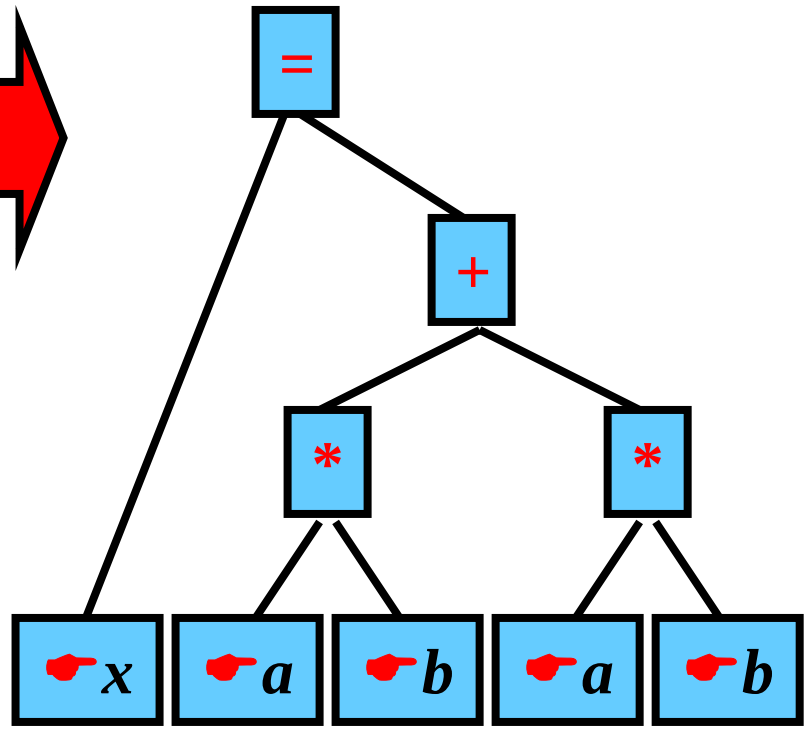
- **PT for**

  ***x = a*b + a*b*:**



- **AST for**

  ***x = a*b + a*b*:**



Generation of AST

Gist: A parser simulates the construction of PT and, simultaneously, calls some semantic actions to create AST.

Example:

Rule:	Semantic Action:
$S \rightarrow i := E_k$	{ $S.a$:= MakeTree('=', $i.a$, $E_k.a$) }
$E_i \rightarrow E_j + E_k$	{ $E_i.a$:= MakeTree('+', $E_j.a$, $E_k.a$) }
$E_i \rightarrow E_j * E_k$	{ $E_i.a$:= MakeTree('*', $E_j.a$, $E_k.a$) }
$E_i \rightarrow (E_j)$	{ $E_i.a$:= $E_j.a$ }
$E_i \rightarrow i$	{ $E_i.a$:= MakeLeaf($i.a$) }

Notes:

• **MakeTree(*o*, *a*, *b*)** creates a new node ***o***, attaches sons ***a*** (left) and ***b***, and returns a pointer to node ***o***

• **MakeLeaf(*i.a*)** creates a new node ***i.a*** (***i.a*** is a symbol-table address) and returns a pointer to this new node

AST Generation: Example 1/2

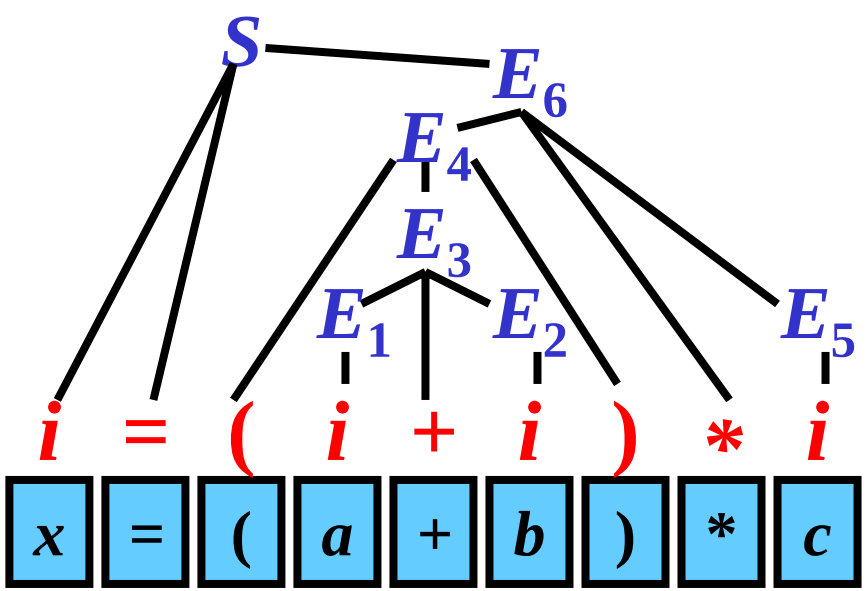
Pushdown	Input	Rule	Semantic action
$	$i = (i + i) * i$$		
$i	= (i + i) * i$		
$i =	(i + i) * i$		
$i = (	i + i) * i$		
$i = (i	+ i) * i$	$E_1 \rightarrow i$	$E_1.a$:= MakeLeaf($i.a$)
$i = ($E_1$	+ i) * i$		
$i = ($E_1$ +	i) * i$		
$i = ($E_1$ + i	) * i$	$E_2 \rightarrow i$	$E_2.a$:= MakeLeaf($i.a$)
$i = ($E_1$ + E_2	) * i$	$E_3 \rightarrow E_1+E_2$	$E_3.a$:=MakeTree(‘+’, $E_1.a$, $E_2.a$)
$i = ($E_3$	) * i$		
$i = ($E_3$)	* i$	$E_4 \rightarrow (E_3)$	$E_4.a$:= $E_3.a$
$i = E_4	* i$		
$i = E_4 *	i$		
$i = E_4 * i	$	$E_5 \rightarrow i$	$E_5.a$:= MakeLeaf($i.a$)
$i = E_4 * E_5	$	$E_6 \rightarrow E_4*E_5$	$E_6.a$:=MakeTree(‘*’, $E_4.a$, $E_5.a$)
$i = E_6	$	$S \rightarrow i = E_6$	$S.a$:= MakeTree(‘=’, $i.a$, $E_6.a$)
$S	$		

Bottom-Up parsing **Semantic actions**

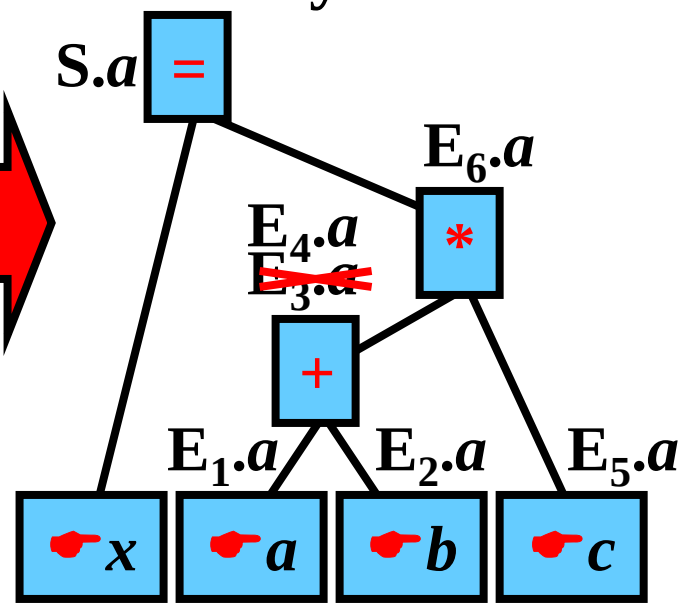
AST Generation: Example 2/2

Rule:	Semantic Action:
$E_1 \rightarrow i$	$E_1.a$:= MakeLeaf($i.a$)
$E_2 \rightarrow i$	$E_2.a$:= MakeLeaf($i.a$)
$E_3 \rightarrow E_1 + E_2$	$E_3.a$:= MakeTree(‘+’, $E_1.a$, $E_2.a$)
$E_4 \rightarrow (E_3)$	$E_4.a$:= $E_3.a$
$E_5 \rightarrow i$	$E_5.a$:= MakeLeaf($i.a$)
$E_6 \rightarrow E_4 * E_5$	$E_6.a$:= MakeTree(‘*’, $E_4.a$, $E_5.a$)
$S \rightarrow i = E_6$	$S.a$:= MakeTree(‘=’, $i.a$, $E_6.a$)

Simulated Parse tree:

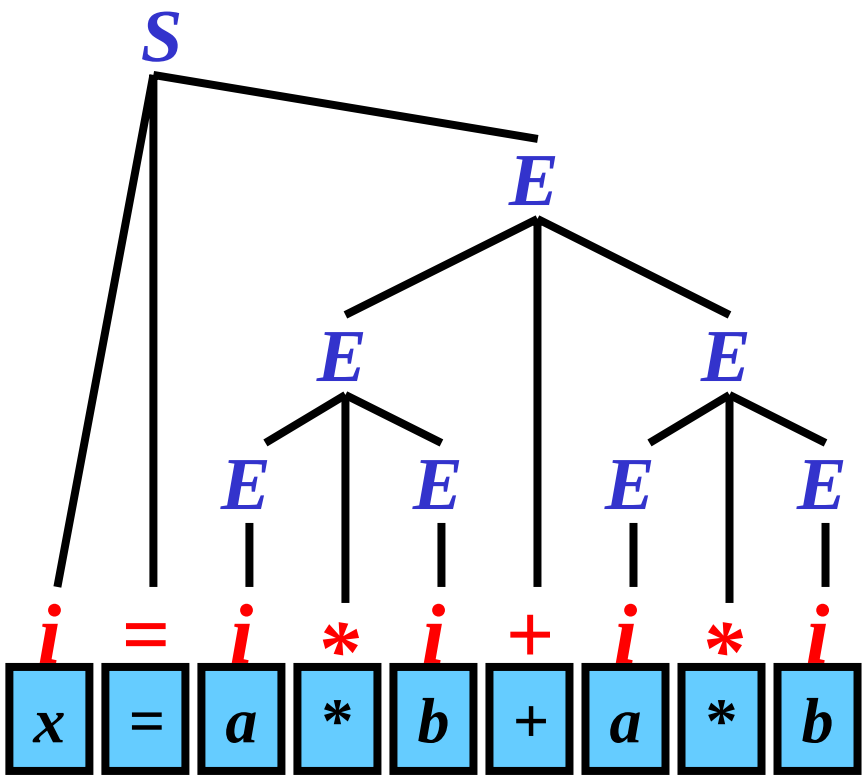


Abstract syntax tree:

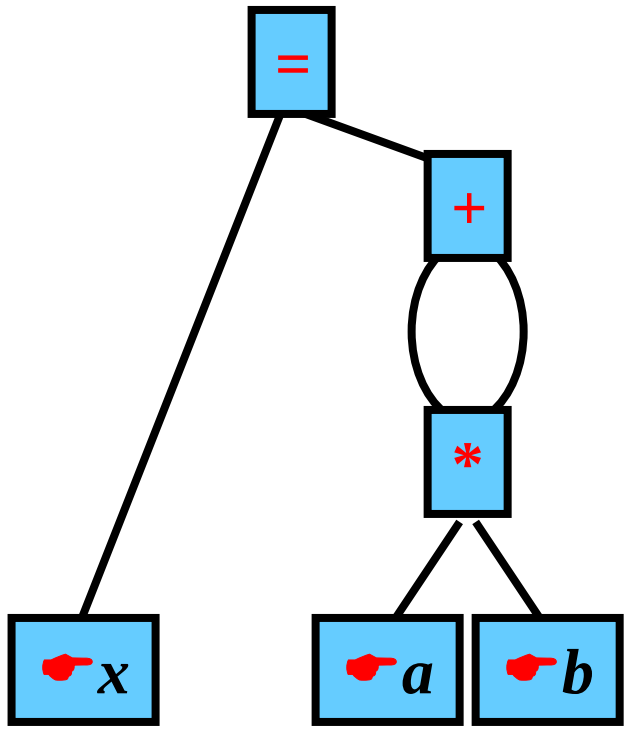


Direct Acyclic Graph(DAG): Example

- **Parse tree for**
  ***x = a*b + a*b*:**



- **DAG for**
  ***x = a*b + a*b*:**



Note: DAG has no redundant nodes.

Postfix Notation

Gist: Every operator occurs behind its operands.

Example:

Infix notation	Postfix notation
$a + b$	$a\ b\ +$
$a = b$	$a\ b\ =$
if C **then** S_1 **else** S_2	$C\ S_1\ S_2$ **if-then-else**

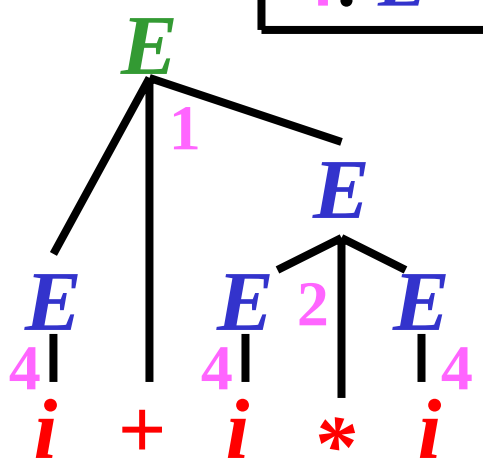
Note: Postfix notation is achievable by the postorder traversal of AST.

Infix to Postfix Directed by a BU Parser

Gist: Semantic actions produce the postfix version of the tokenized source program.

Example:

Rule:	Semantic Action:
1: $E \rightarrow E+E$	{*generate*(‘+’)}
2: $E \rightarrow E*E$	{*generate*(‘*’)}
3: $E \rightarrow (E)$	{ - }
4: $E \rightarrow i$	{*generate*(*i.a*) }



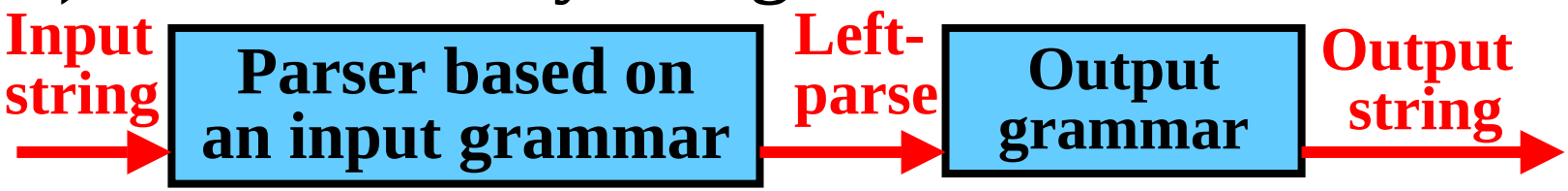
Input: a + b * c

Output: a b c * +

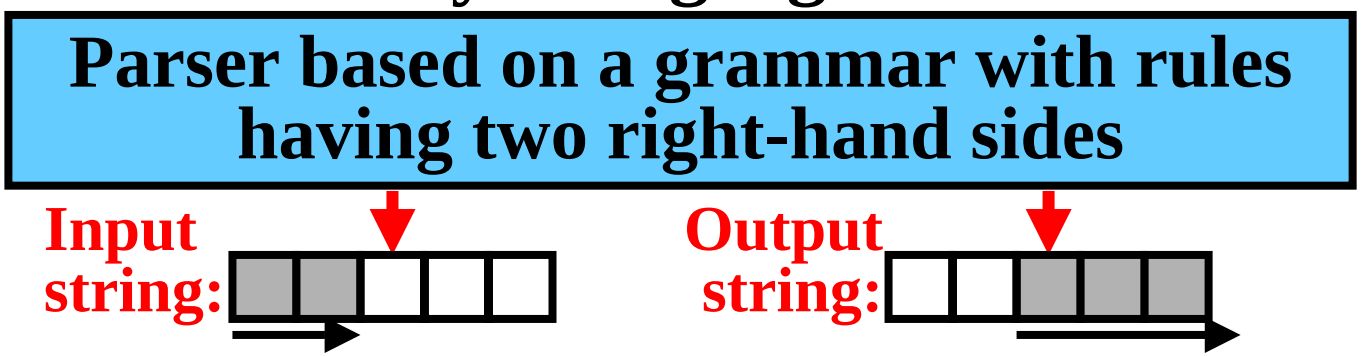
Translation Grammars

Gist: Translation grammars translate input strings to output strings

1) Translation by two grammars:



2) Translation by a single grammar

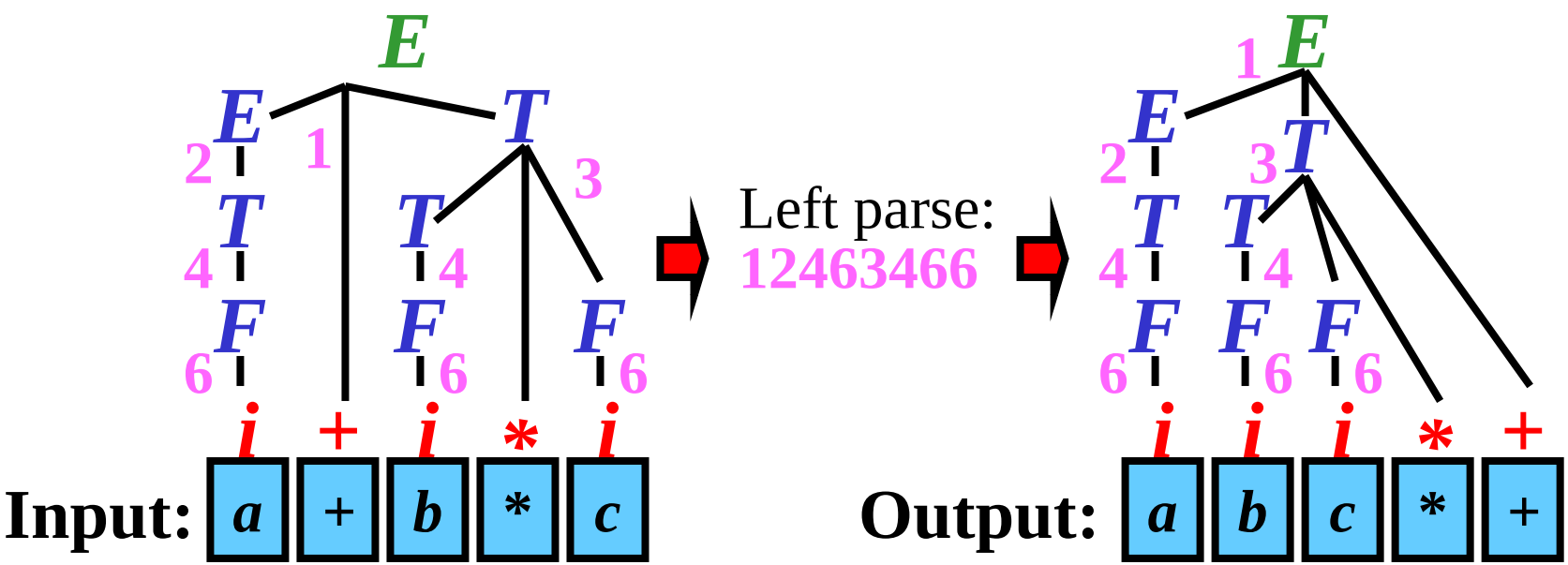


Note: During the parse of an input string, a simultaneous generation f an output string occurs

Two-Grammar Translation

Infix to postfix translation:

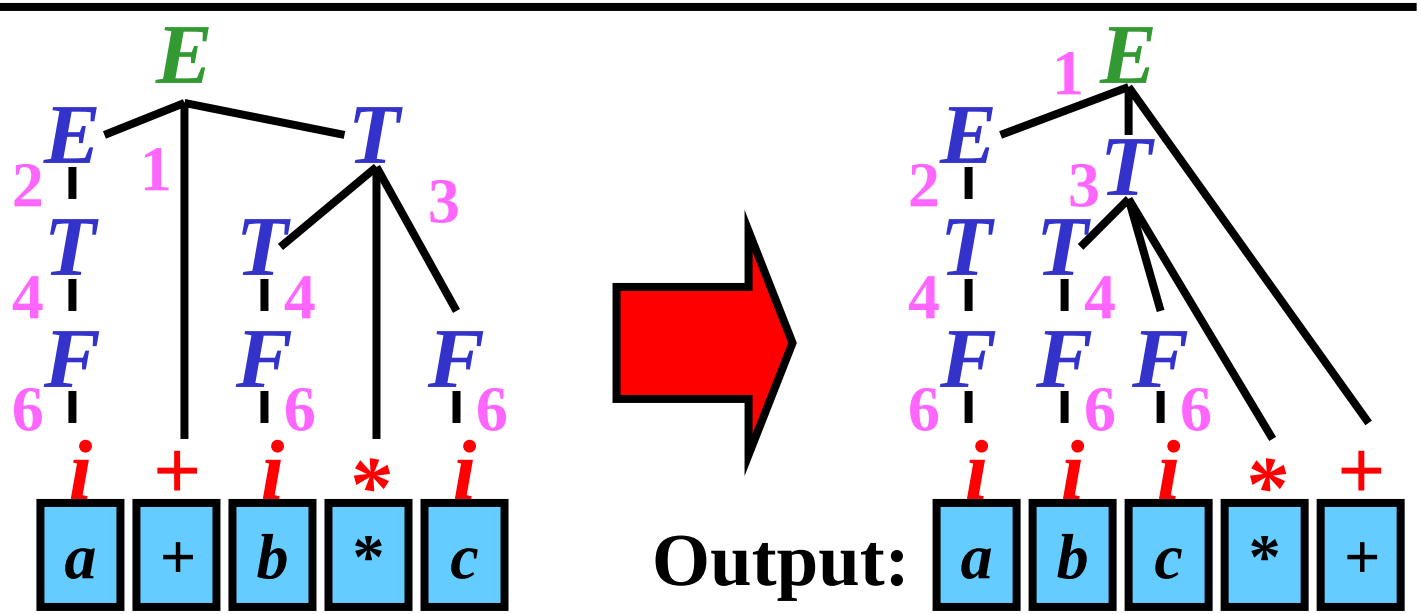
Rules of G_1	Rules of G_2
1: $E \rightarrow E+T$	1: $E \rightarrow ET+$
2: $E \rightarrow T$	2: $E \rightarrow T$
3: $T \rightarrow T*F$	3: $T \rightarrow TF*$
4: $T \rightarrow F$	4: $T \rightarrow F$
5: $F \rightarrow (E)$	5: $F \rightarrow E$
6: $F \rightarrow i$	6: $F \rightarrow i$



One-Grammar Translation

Infix to postfix translation:

Rule	Tran. Element
1: $E \rightarrow E+T$	$ET+$
2: $E \rightarrow T$	T
3: $T \rightarrow T*F$	$TF*$
4: $T \rightarrow F$	F
5: $F \rightarrow (E)$	E
6: $F \rightarrow i$	i

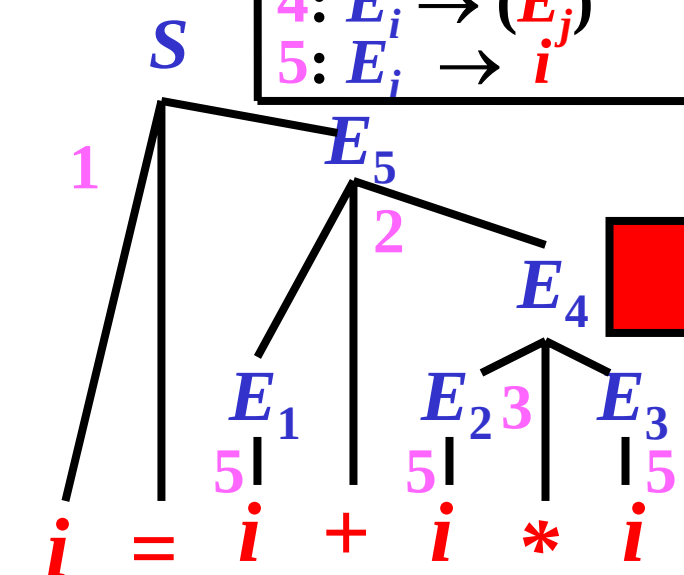


Direct Generation of 3AC

Gist: BU parser directs the generation of 3AC directly.

Example:

Rule:	Semantic Action:
1: $S \rightarrow i = E_k$	{ generate('=', E_k.loc, , i.loc)}
2: $E_i \rightarrow E_j + E_k$	{ generate('+', E_j.loc, E_k.loc, E_i.loc)}
3: $E_i \rightarrow E_j * E_k$	{ generate('*', E_j.loc, E_k.loc, E_i.loc)}
4: $E_i \rightarrow (E_j)$	{ generate('=', E_j.loc, , E_i.loc)}
5: $E_i \rightarrow i$	{ generate('=', i.loc, , E_i.loc)}



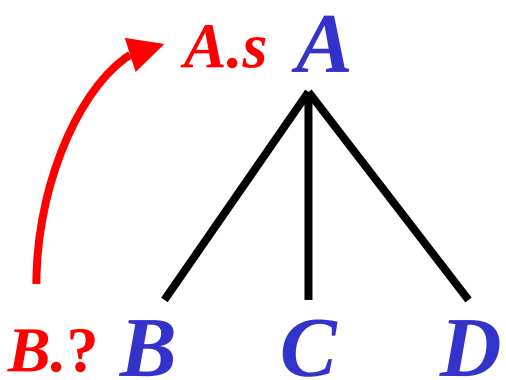
Input: x = a + b * c

Output:

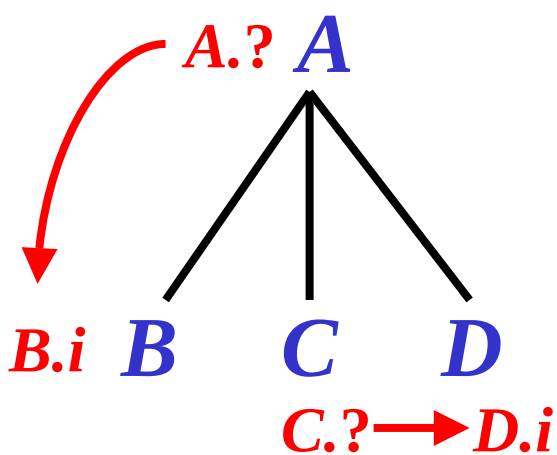
('=', ☛a, , E_1.loc)
('=', ☛b, , E_2.loc)
('=', ☛c, , E_3.loc)
('*', E_2.loc,E_3.loc,E_4.loc)
('+', E_1.loc,E_4.loc,E_5.loc)
('=', E_5.loc, , ☛x)

Top-Down Translation: Introduction

- **LL-grammar with attributes**
- **Two pushdown:**
  - **parser pushdown × semantic pushdown**
- **Two type of attributes:**
  - **synthesized:**
    (from children to parent)



iherited:
(from parent to children
or between siblings)

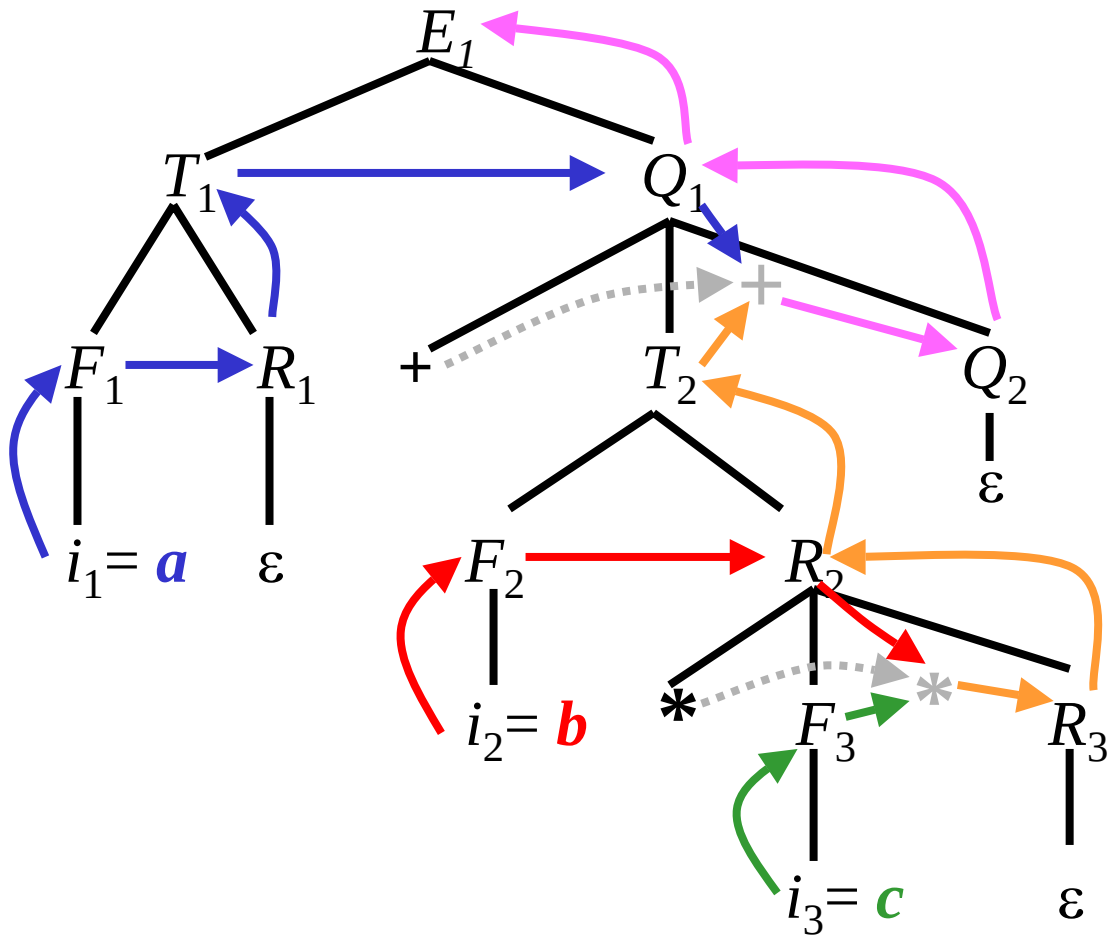


Top-Down Translation: Expressions

Grammar:

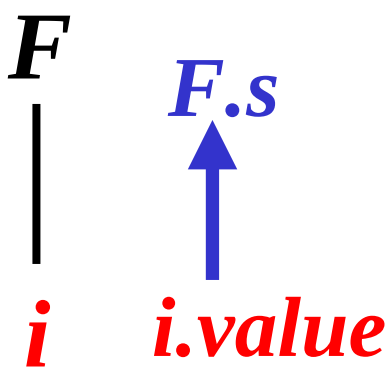
$E \rightarrow TQ$
$Q \rightarrow +TQ$
$Q \rightarrow \varepsilon$
$T \rightarrow FR$
$R \rightarrow *FR$
$R \rightarrow \varepsilon$
$F \rightarrow (E)$
$F \rightarrow i$

Parse tree for *a* + *b* * *c*:



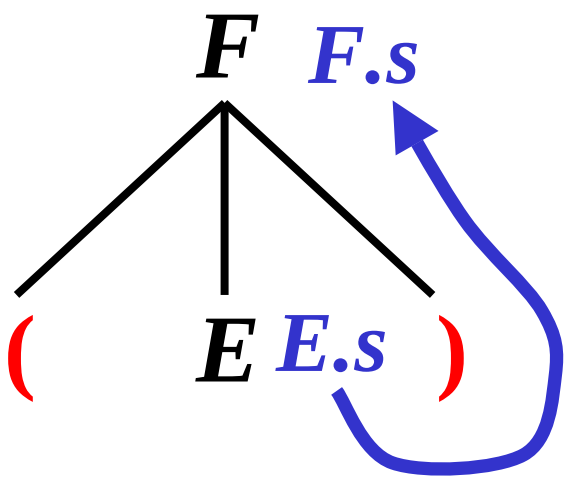
Expressions: Variable & Parentheses

Variable:



$F \rightarrow i \ \{F.s := i.value\}$

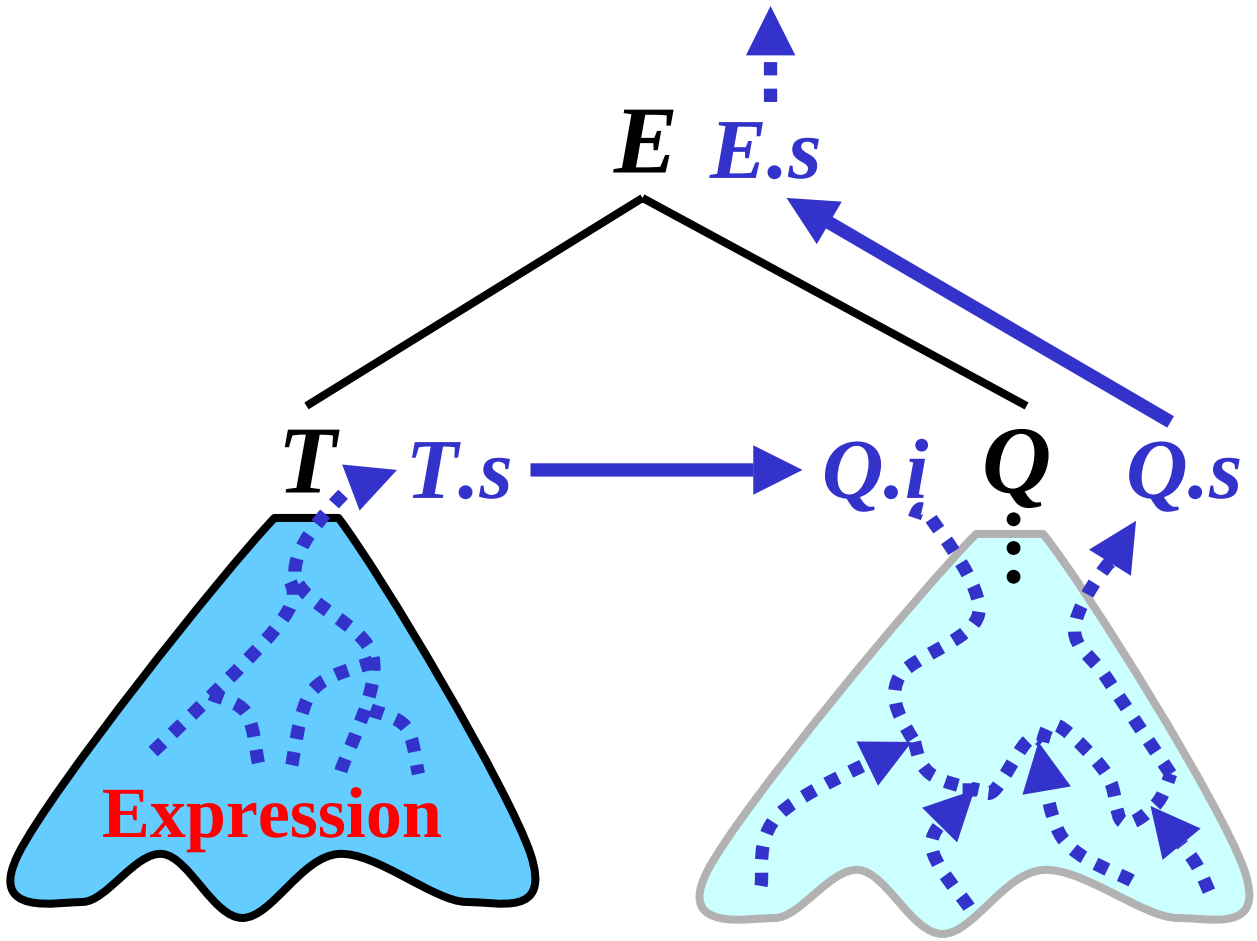
Parentheses:



$E \rightarrow (F \ \{F.s := E.s\} \ )$

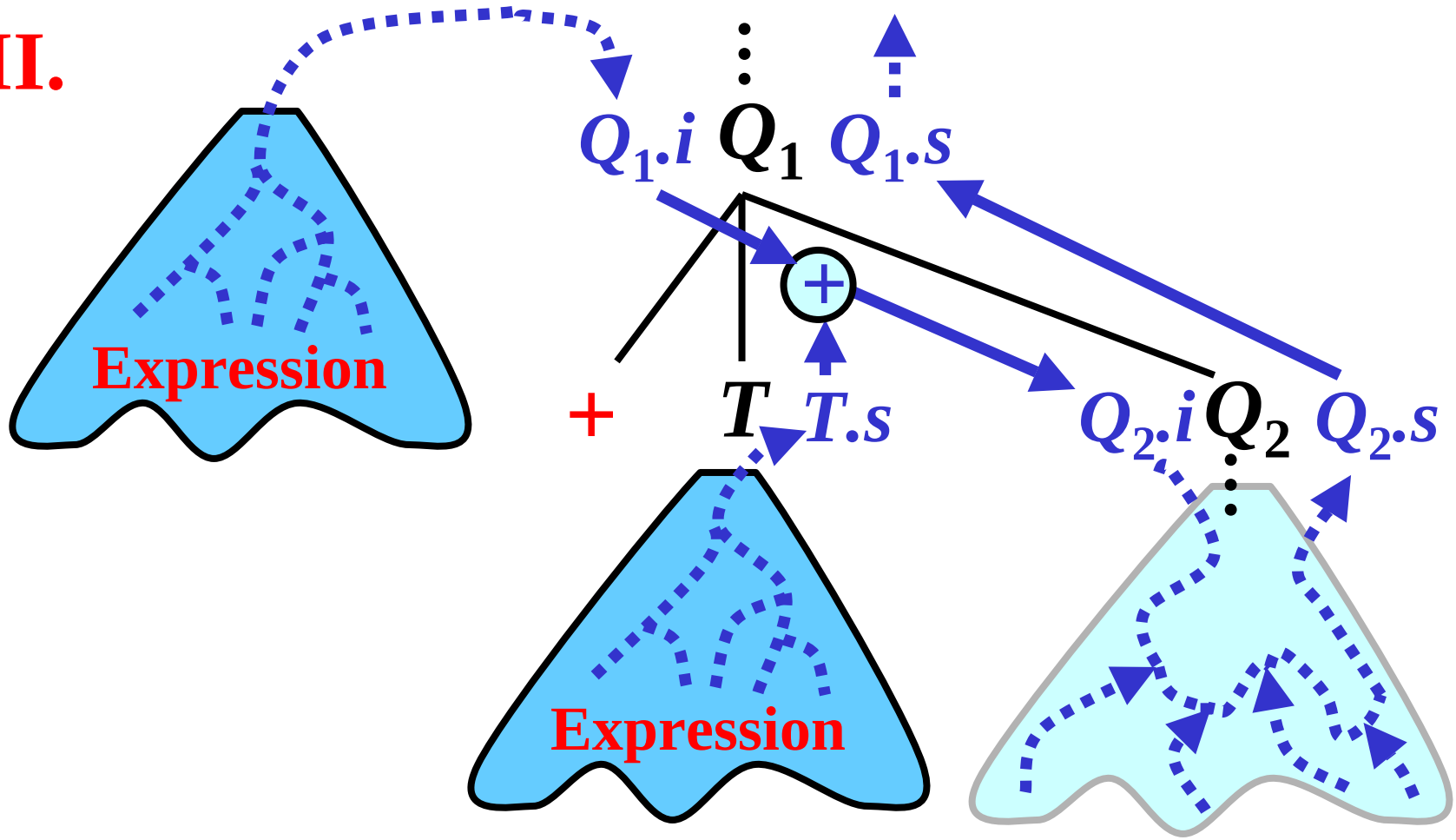
Expressions: Addition 1/4

I.



$$E \rightarrow T \{ Q.i := T.s \}\ Q \{ E.s := Q.s \}$$

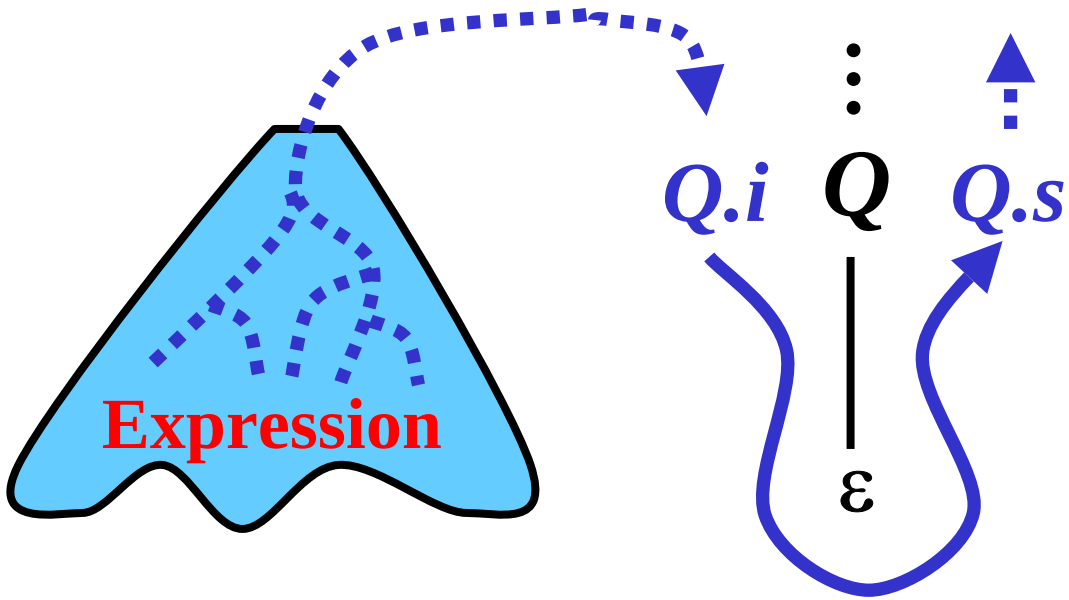
Expressions: Addition 2/4



$$Q_1 \rightarrow +T \{ Q_2.i := Q_1.i + T.s \} Q_2 \{ Q_1.s := Q_2.s \}$$

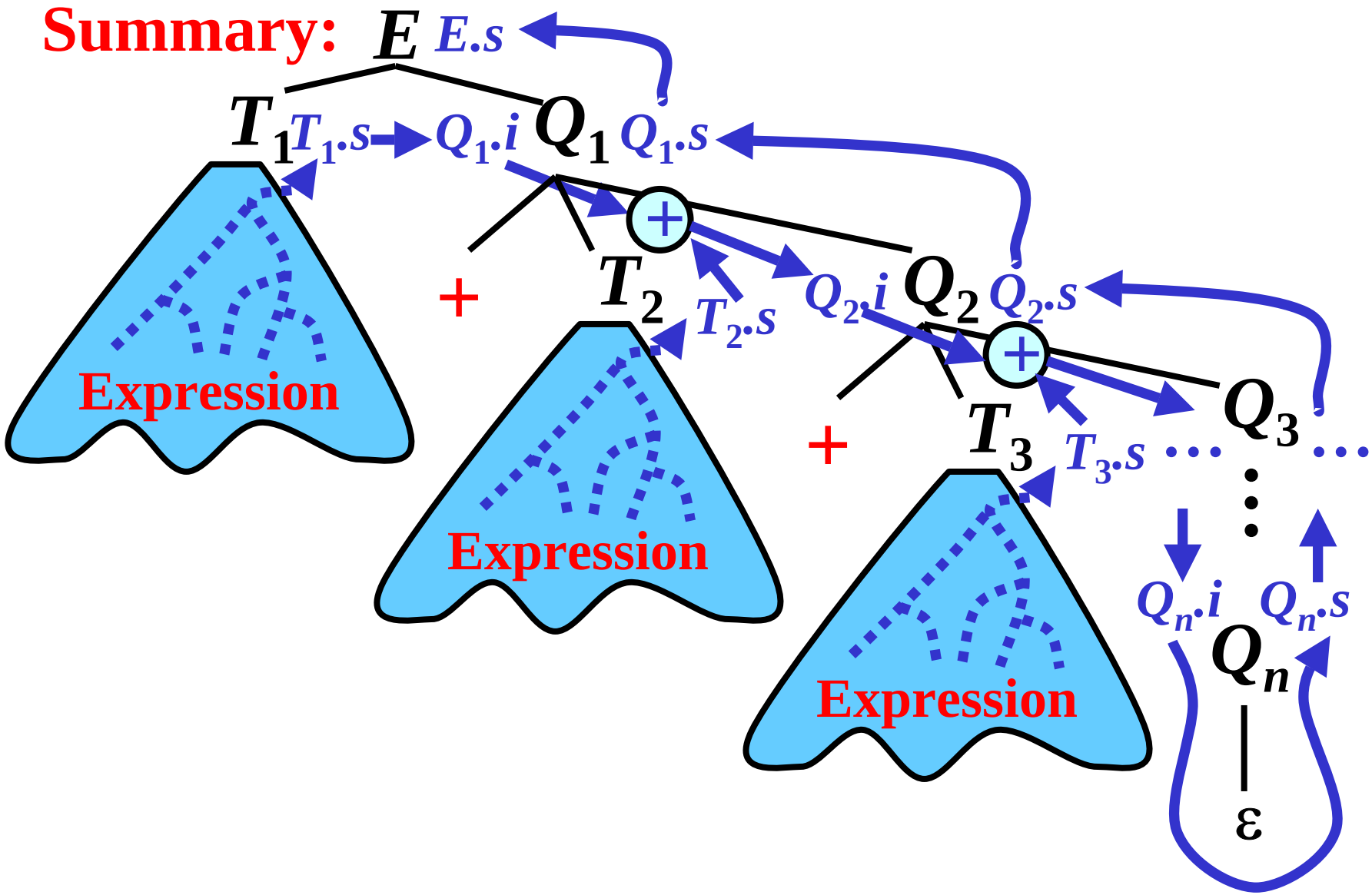
Expressions: Addition 3/4

III.



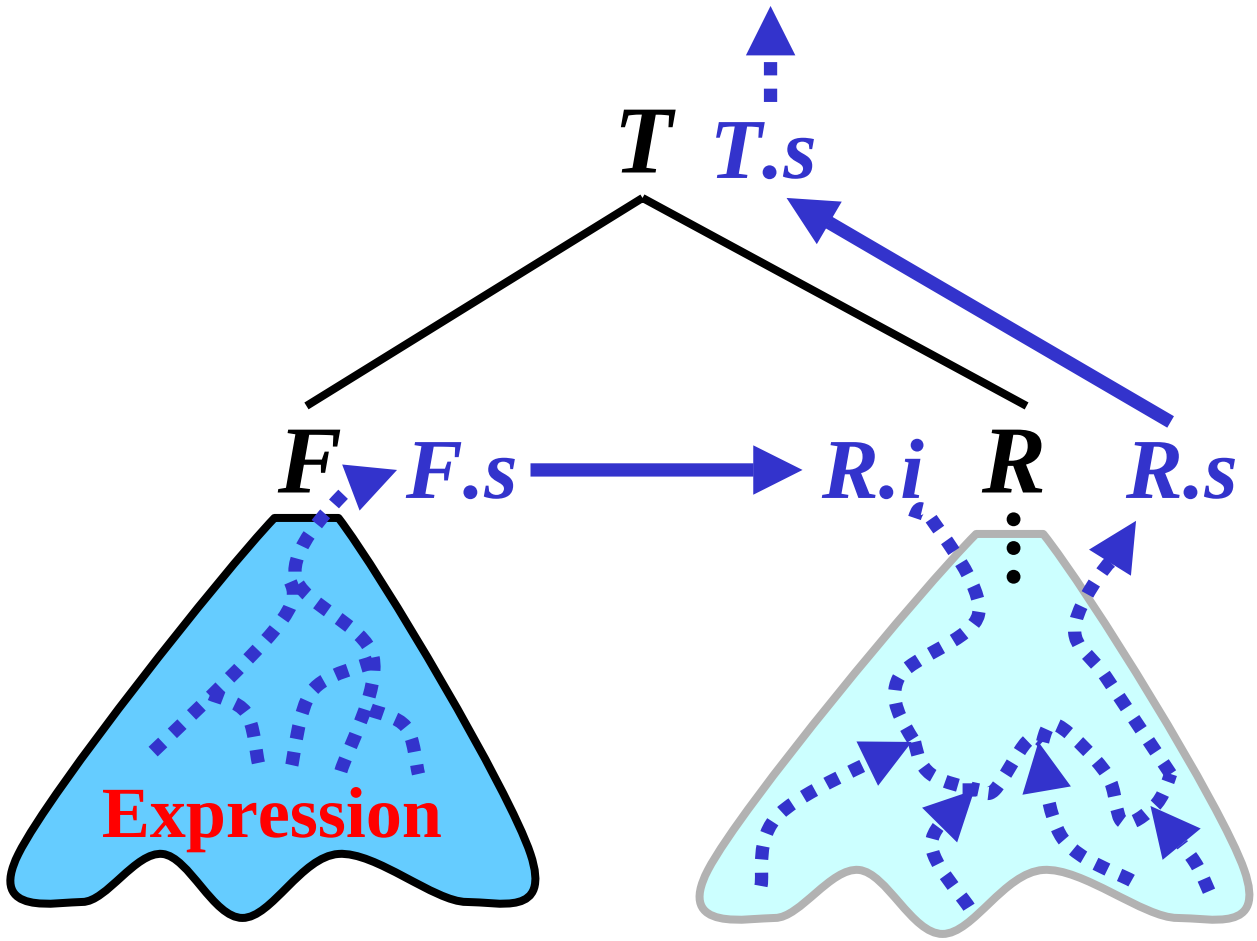
$Q \rightarrow \varepsilon \quad \{Q.s := Q.i\}$

Expressions: Addition 4/4



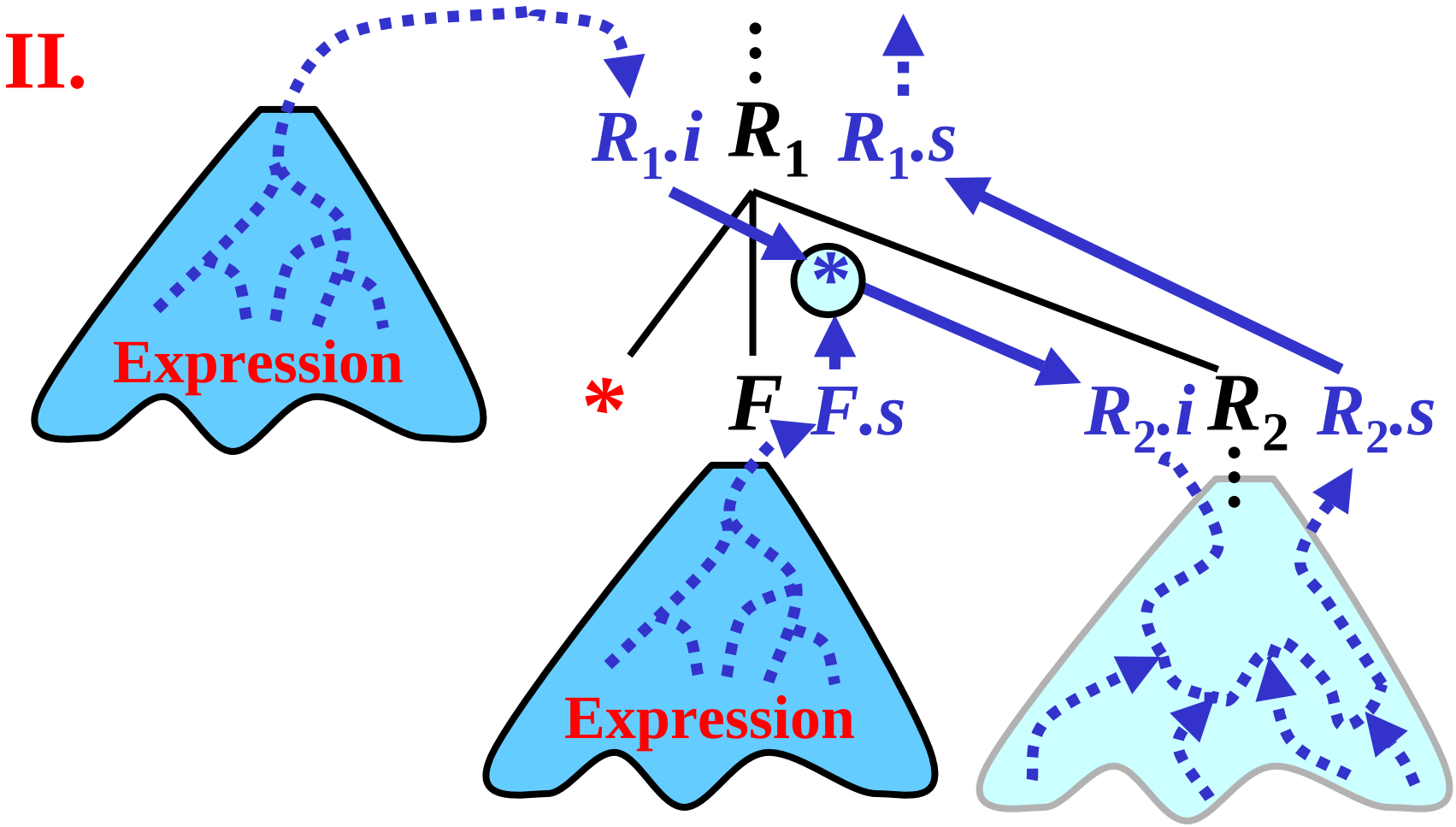
Expressions: Multiplication 1/4

I.



$$T \rightarrow F \{ R.i := F.s \} R \{ T.s := R.s \}$$

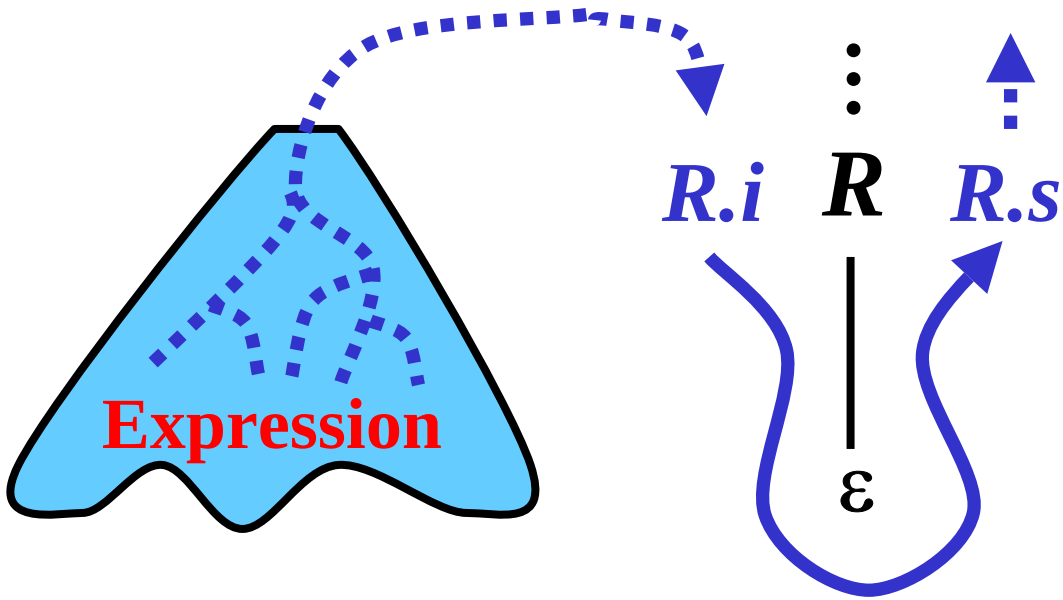
Expressions: Multiplication 2/4



$$R_1 \rightarrow {*}F \ \{ \ R_2.i := R_1.i * F.s \ \} \ Q_2 \{ R_1.s := R_2.s \ \}$$

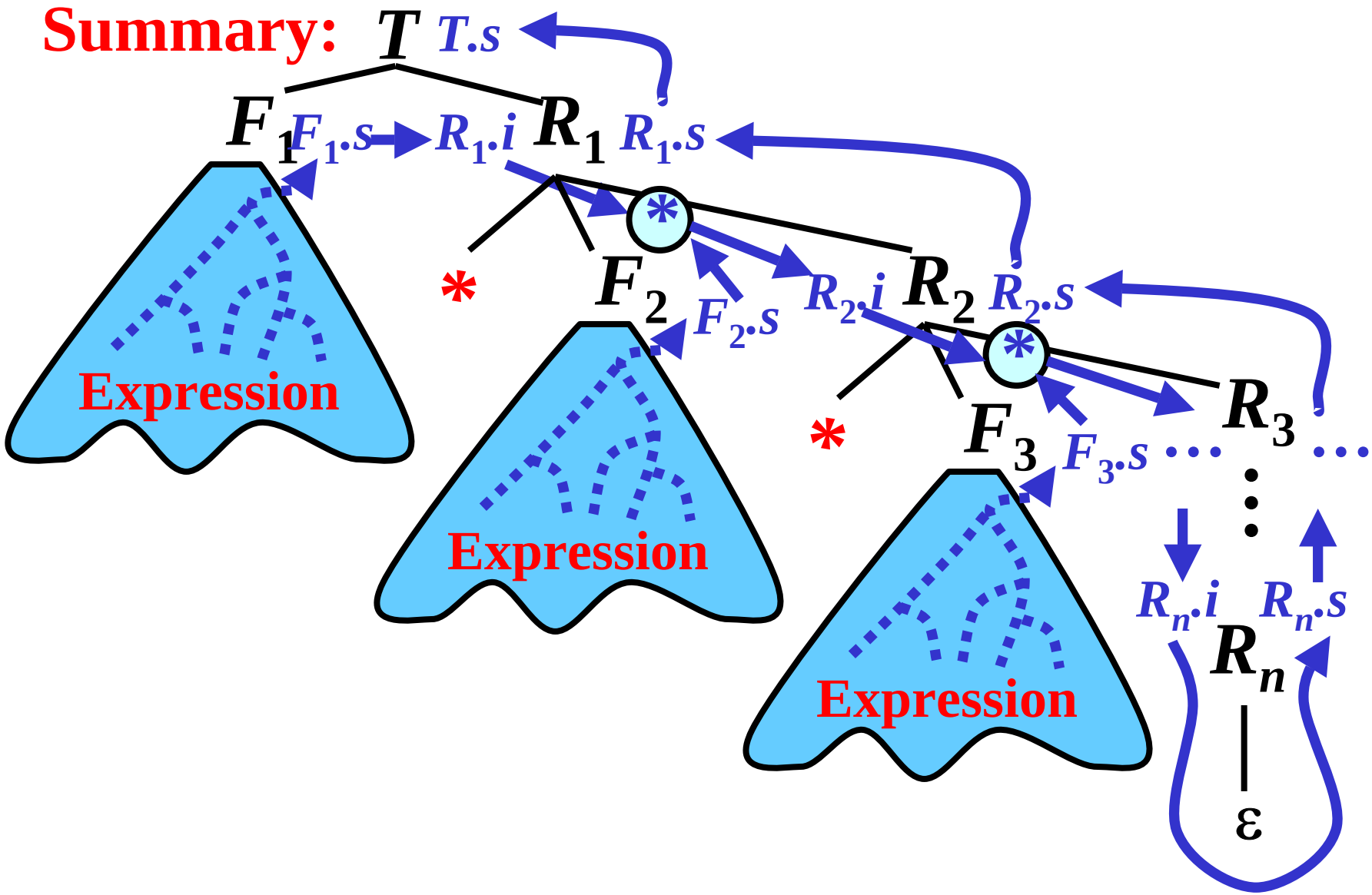
Expressions: Multiplication 3/4

III.



$R \rightarrow \varepsilon \quad \{R.s := R.i\}$

Expressions: Multiplication 4/4



Grammar for Expressions: Summary

1. $E \rightarrow T \; \{Q.i := T.s\} \; Q \; \{E.s := Q.s\}$
2. $Q_1 \rightarrow +T \; \{Q_2.i := Q_1.i + T.s\} \; Q_2 \; \{Q_1.s := Q_2.s\}$
3. $Q \rightarrow \varepsilon \; \{Q.s := Q.i\}$
4. $T \rightarrow F \; \{R.i := F.s\} \; R \; \{T.s := R.s\}$
5. $R_1 \rightarrow *F \; \{R_2.i := R_1.i * F.s\} \; R_2 \; \{R_1.s := R_2.s\}$
6. $R \rightarrow \varepsilon \; \{R.s := R.i\}$
7. $F \rightarrow (E \; \{F.s := E.s\} \;)$
8. $F \rightarrow i \; \{F.s := i.value\}$

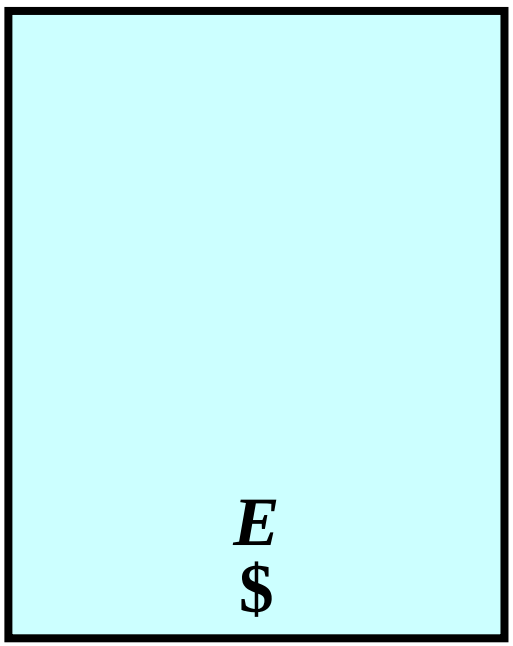
Evaluation of Expressions: Example 1/16

Example for $a + b$, where $a.value = 10$, $b.value = 20$

Input: $i_1 + i_2$ \$

Rule: $E \rightarrow T_1 \{Q_1.i := T_1.s\}\ Q_1\ \{E.s := Q_1.s\}$

Parser pushdown:



Semantic pushdown:

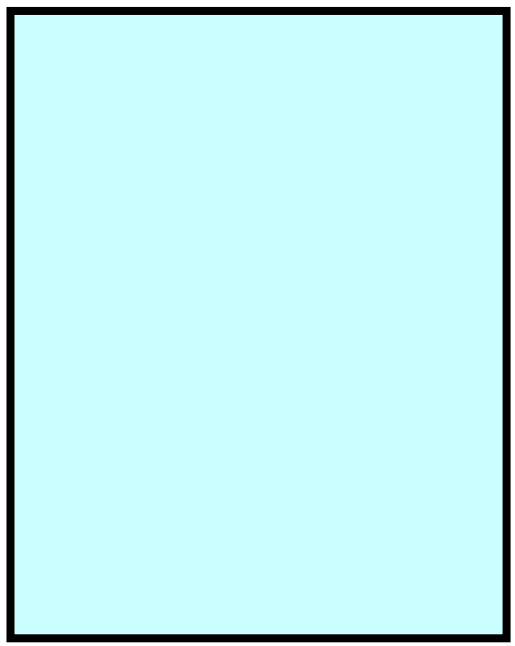


Illustration:

Evaluation of Expressions: Example 2/16

Example for $a + b$, where $a.value = 10$, $b.value = 20$

Input: $i_1 + i_2$ \$

Rule: $T_1 \rightarrow F_1 \{R_1.i := F_1.s\} R_1 \{T_1.s := R_1.s\}$

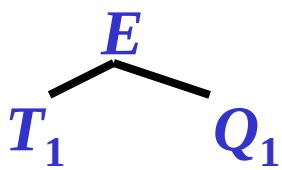
Parser pushdown:

T_1
$\{Q_1.i := T_1.s\}$
Q_1
$\{E.s := Q_1.s\}$
\$

Semantic pushdown:

10 | + | 20

Illustration:



Evaluation of Expressions: Example 3/16

Example for $a + b$, where $a.value = 10$, $b.value = 20$

Input: $i_1 + i_2$ \$

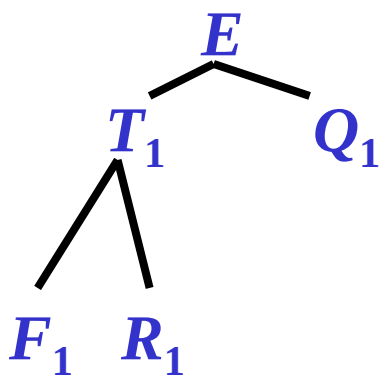
Rule: $F_1 \rightarrow i_1$ $\{F_1.s := i.value\}$

Parser pushdown:

F_1
$\{R_1.i := F_1.s\}$
R_1
$\{T_1.s := R_1.s\}$
$\{Q_1.i := T_1.s\}$
Q_1
$\{E.s := Q_1.s\}$
\$

Semantic pushdown:

Illustration:



10 + 20

Evaluation of Expressions: Example 4/16

Example for $a + b$, where $a.value = 10$, $b.value = 20$

Input: $i_1 + i_2$ \$

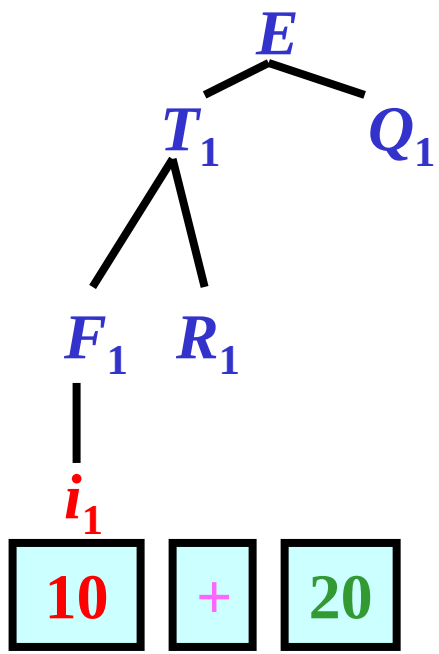
Rule:

Parser pushdown:

i_1
$\{F_1.s := i.value\}$
$\{R_1.i := F_1.s\}$
R_1
$\{T_1.s := R_1.s\}$
$\{Q_1.i := T_1.s\}$
Q_1
$\{E.s := Q_1.s\}$
\$

Semantic pushdown:

Illustration:



Evaluation of Expressions: Example 5/16

Example for *a* + *b*, where *a.value* = 10, *b.value* = 20

Input: $+ \; i_2 \; \$$

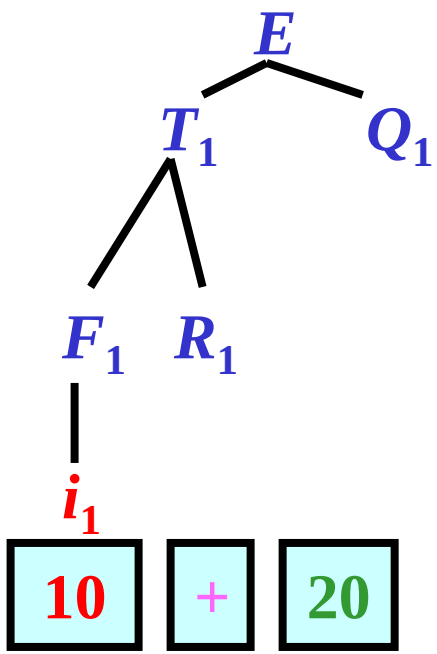
Rule:

Parser pushdown:

$\{F_1.s := i.value\}$
$\{R_1.i := F_1.s\}$
R_1
$\{T_1.s := R_1.s\}$
$\{Q_1.i := T_1.s\}$
Q_1
$\{E.s := Q_1.s\}$
$\$$

Semantic pushdown:

Illustration:



Evaluation of Expressions: Example 6/16

Example for $a + b$, where $a.value = 10$, $b.value = 20$

Input: $+ \; i_2 \; \$$

Rule: $R_1 \rightarrow \varepsilon \; \{R_1.s := R_1.i\}$

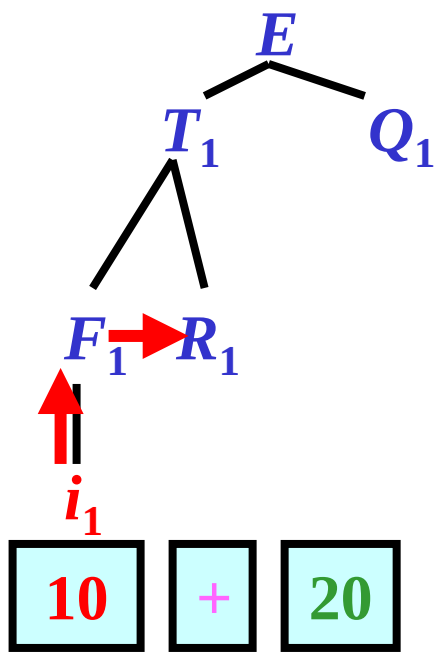
Parser pushdown:

R_1
$\{T_1.s := R_1.s\}$
$\{Q_1.i := T_1.s\}$
Q_1
$\{E.s := Q_1.s\}$
$\$$

Semantic pushdown:

$R_1.i = 10$

Illustration:



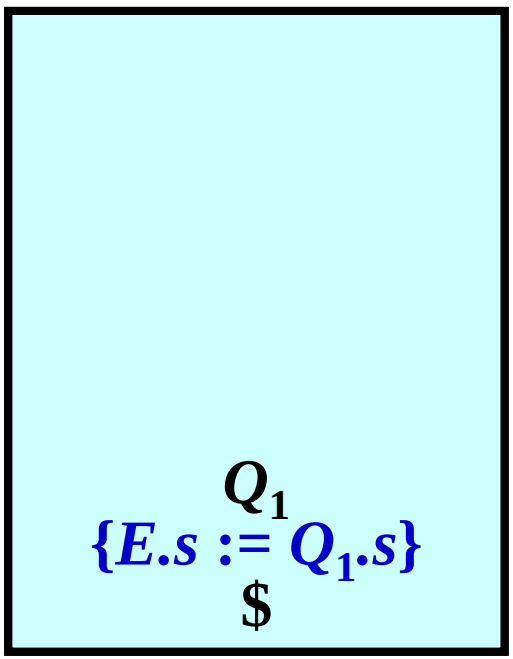
Evaluation of Expressions: Example 7/16

Example for $a + b$, where $a.value = 10$, $b.value = 20$

Input: $+ \; i_2 \; \$$

Rule: $Q_1 \rightarrow +T_2 \; \{Q_2.i := Q_1.i + T_2.s\} \; Q_2 \; \{Q_1.s := Q_2.s\}$

Parser pushdown:



Semantic pushdown:

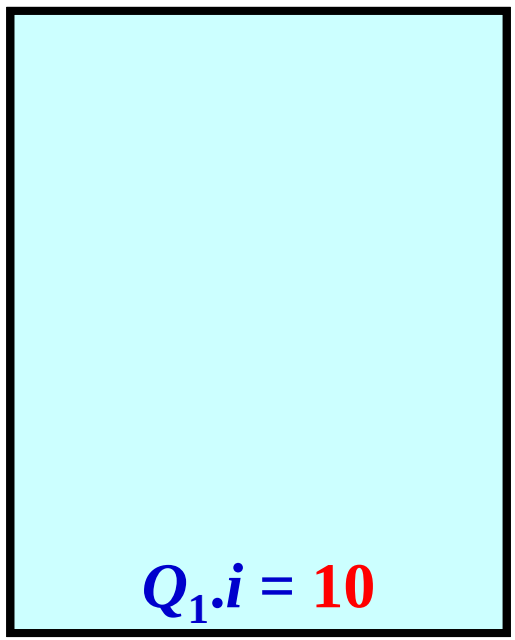
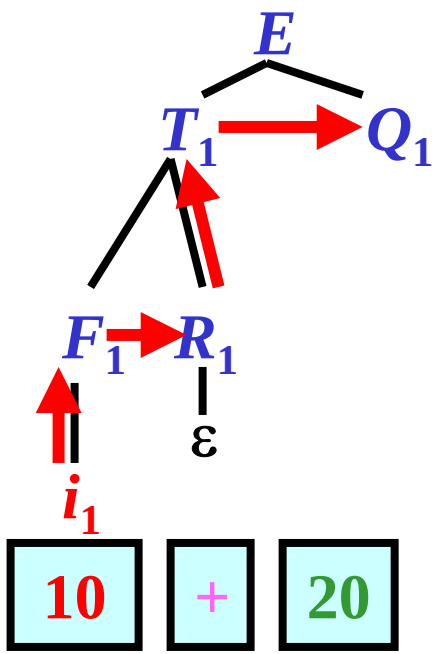


Illustration:



Evaluation of Expressions: Example 8/16

Example for $a + b$, where $a.value = 10$, $b.value = 20$

Input: $+ \; i_2 \; \$$

Rule:

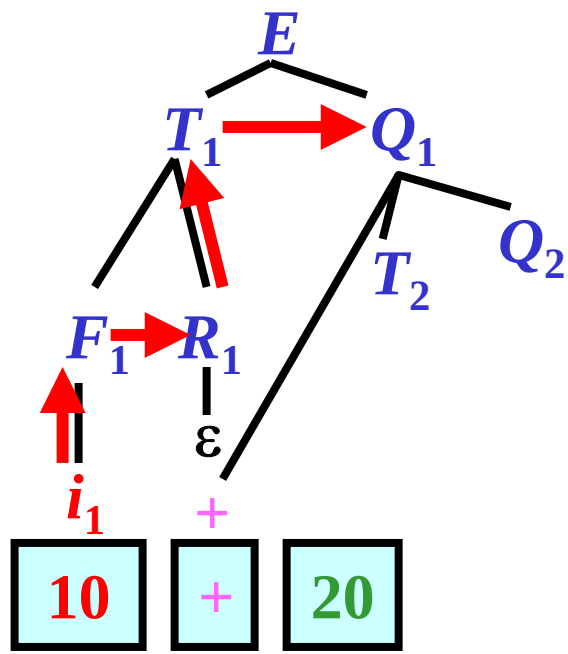
Parser pushdown:

$+$
T_2
$\{Q_2.i := Q_1.i + T_2.s\}$
Q_2
$\{Q_1.s := Q_2.s\}$
$\{E.s := Q_1.s\}$
$\$$

Semantic pushdown:

$Q_1.i = 10$

Illustration:



Evaluation of Expressions: Example 9/16

Example for $a + b$, where $a.value = 10$, $b.value = 20$

Input: i_2 \$

Rule: $T_2 \rightarrow F_2 \{R_2.i := F_2.s\}\ R_2\ \{T_2.s := R_2.s\}$

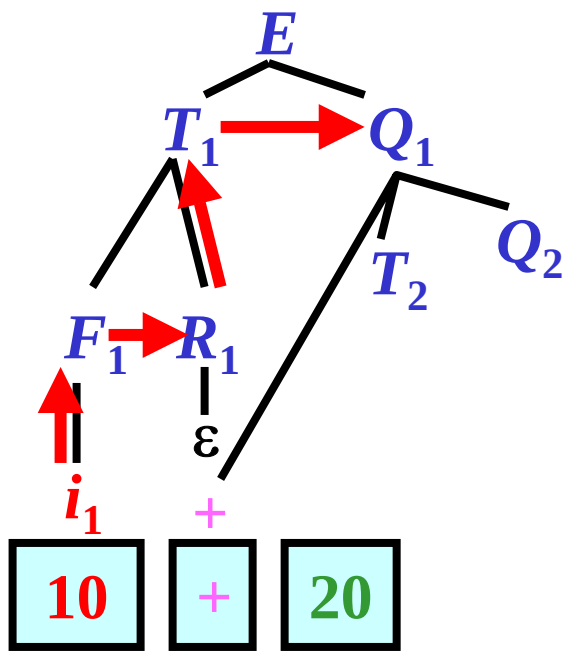
Parser pushdown:

T_2
$\{Q_2.i := Q_1.i + T_2.s\}$
Q_2
$\{Q_1.s := Q_2.s\}$
$\{E.s := Q_1.s\}$
\$

Semantic pushdown:

$Q_1.i = 10$

Illustration:



Evaluation of Expressions: Example 10/16

Example for $a + b$, where $a.value = 10$, $b.value = 20$

Input: i_2 \$

Rule: $F_2 \rightarrow i_2$ $\{F_2.s := i.value\}$

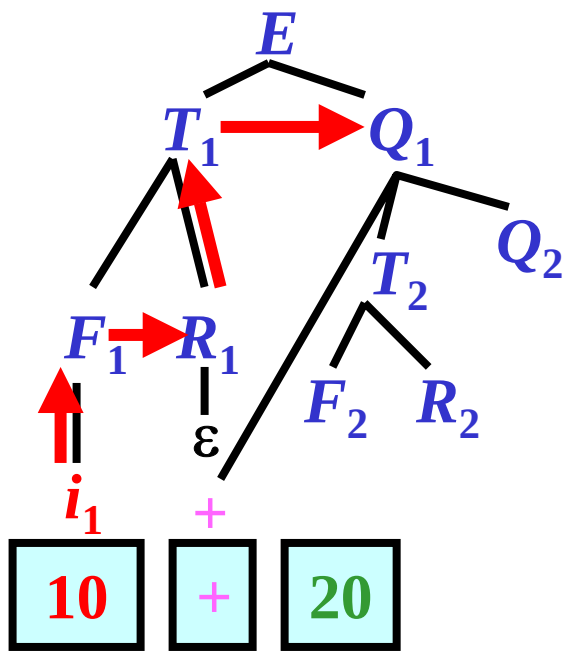
Parser pushdown:

F_2
$\{R_2.i := F_2.s\}$
R_2
$\{T_2.s := R_2.s\}$
$\{Q_2.i := Q_1.i + T_2.s\}$
Q_2
$\{Q_1.s := Q_2.s\}$
$\{E.s := Q_1.s\}$
\$

Semantic pushdown:

$Q_1.i = 10$

Illustration:



Evaluation of Expressions: Example 11/16

Example for $a + b$, where $a.value = 10$, $b.value = 20$

Input: i_2 \$

Rule:

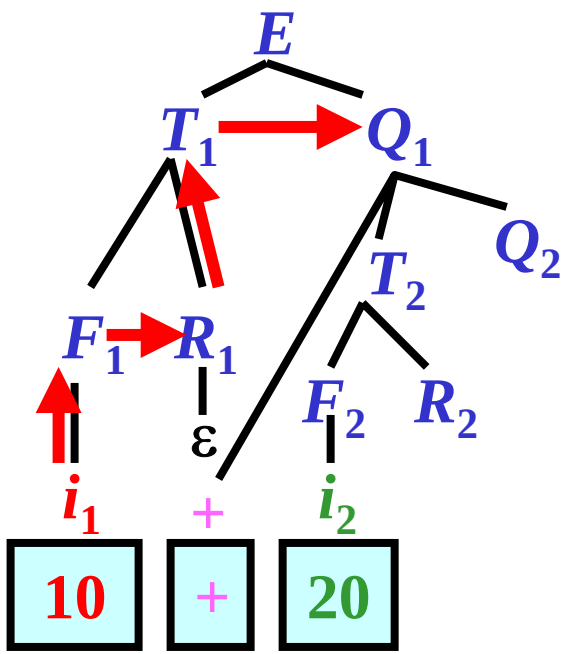
Parser pushdown:

i_2
$\{F_2.s := i.value\}$
$\{R_2.i := F_2.s\}$
R_2
$\{T_2.s := R_2.s\}$
$\{Q_2.i := Q_1.i + T_2.s\}$
Q_2
$\{Q_1.s := Q_2.s\}$
$\{E.s := Q_1.s\}$
\$

Semantic pushdown:

$Q_1.i = 10$

Illustration:



Evaluation of Expressions: Example 12/16

Example for *a* + *b*, where *a.value* = 10, *b.value* = 20

Input: $

Rule:

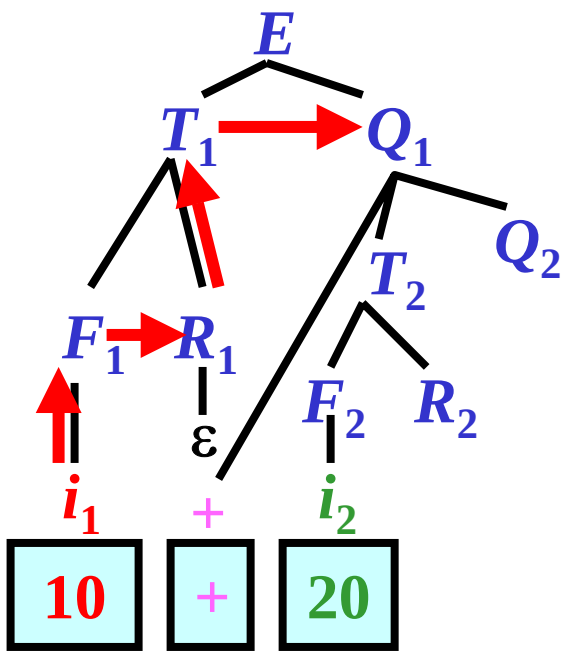
Parser pushdown:

$\{F_2.s := i.value\}$
$\{R_2.i := F_2.s\}$
R_2
$\{T_2.s := R_2.s\}$
$\{Q_2.i := Q_1.i + T_2.s\}$
Q_2
$\{Q_1.s := Q_2.s\}$
$\{E.s := Q_1.s\}$
$

Semantic pushdown:

$Q_1.i = 10$

Illustration:



Evaluation of Expressions: Example 13/16

Example for $a + b$, where $a.value = 10$, $b.value = 20$

Input: $

Rule: $R_2 \rightarrow \varepsilon \; \{R_2.s := R_2.i\}$

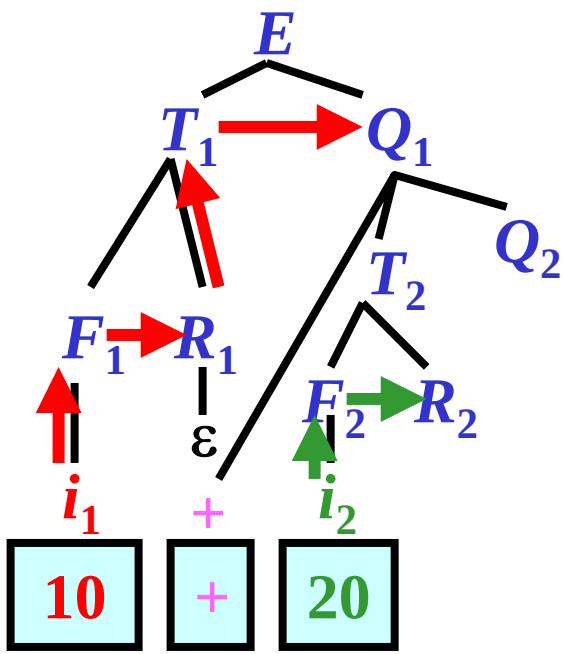
Parser pushdown:

R_2
$\{T_2.s := R_2.s\}$
$\{Q_2.i := Q_1.i + T_2.s\}$
Q_2
$\{Q_1.s := Q_2.s\}$
$\{E.s := Q_1.s\}$
$

Semantic pushdown:

$R_2.i = 20$
$Q_1.i = 10$

Illustration:



Evaluation of Expressions: Example 14/16

Example for $a + b$, where $a.value = 10$, $b.value = 20$

Input: $

Rule:

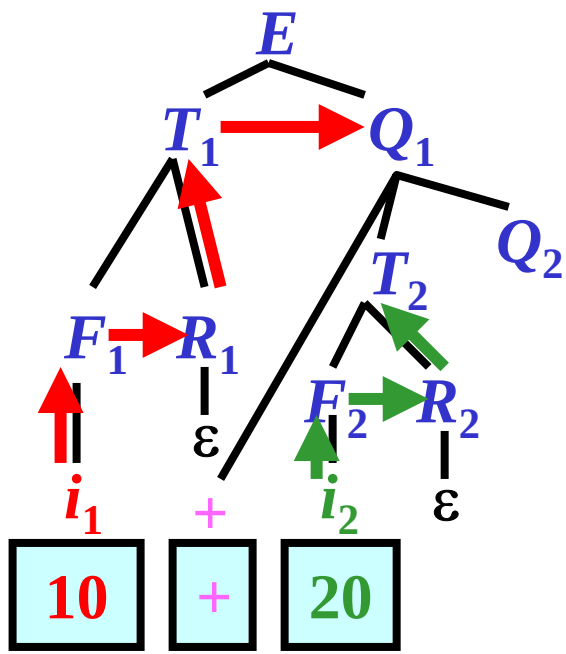
Parser pushdown:

$\{Q_2.i := Q_1.i + T_2.s\}$
Q_2
$\{Q_1.s := Q_2.s\}$
$\{E.s := Q_1.s\}$
$

Semantic pushdown:

$T_2.s = 20$
$Q_1.i = 10$

Illustration:



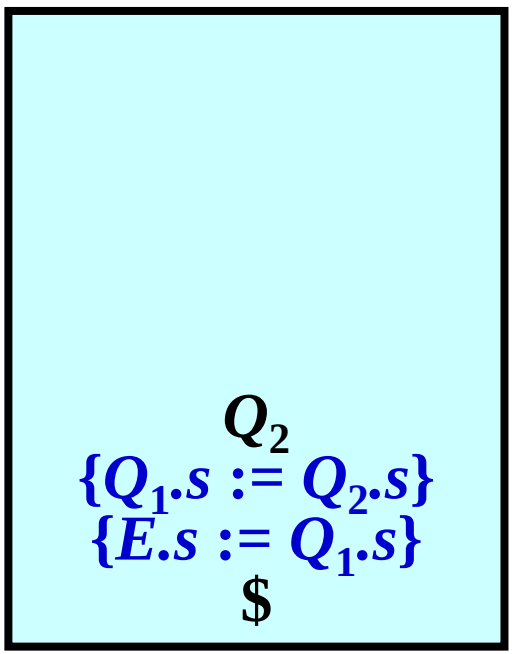
Evaluation of Expressions: Example 15/16

Example for *a* + *b*, where *a.value* = 10, *b.value* = 20

Input: $

Rule: $Q_2 \rightarrow \varepsilon \; \{Q_2.s := Q_2.i\}$

Parser pushdown:



Semantic pushdown:

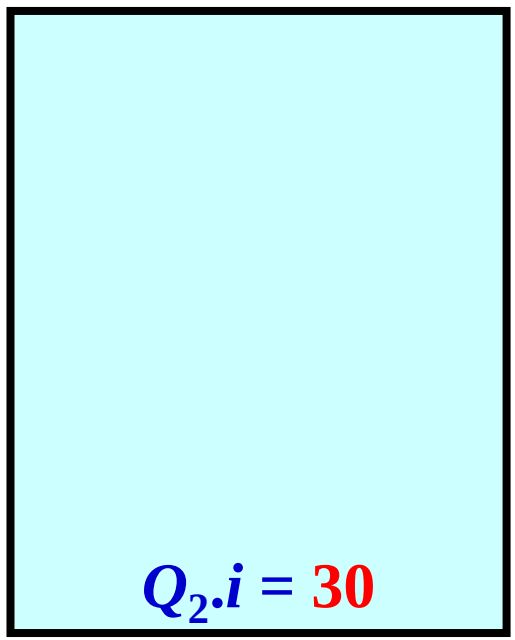
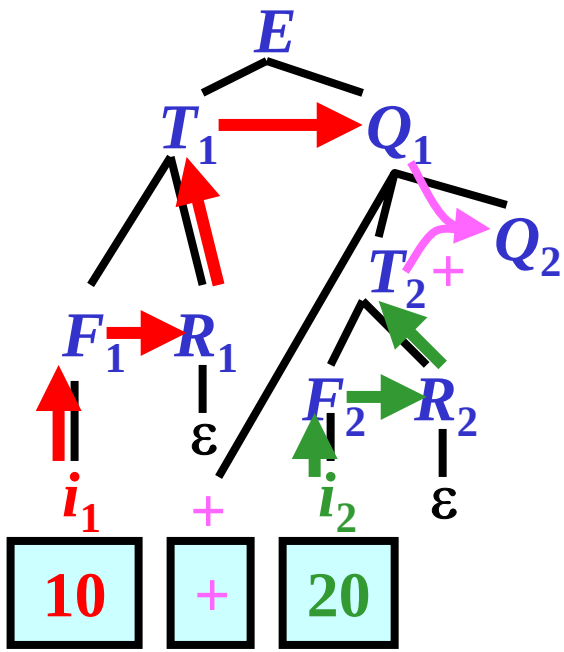


Illustration:



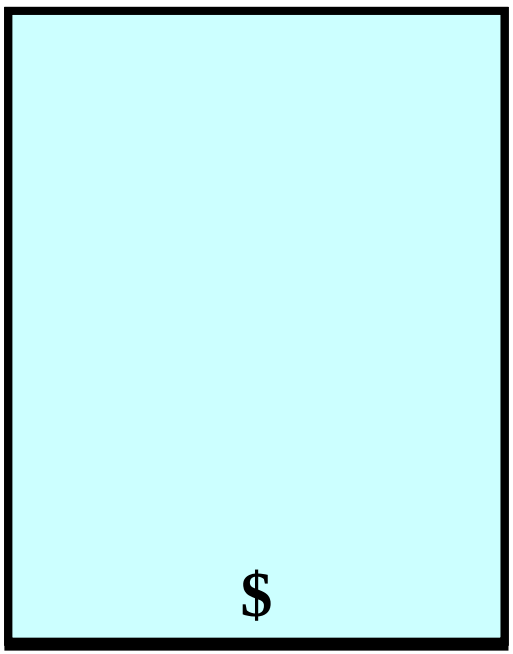
Evaluation of Expressions: Example 16/16

Example for *a* + *b*, where *a.value* = 10, *b.value* = 20

Input: $

Rule:

Parser pushdown:



Semantic pushdown:

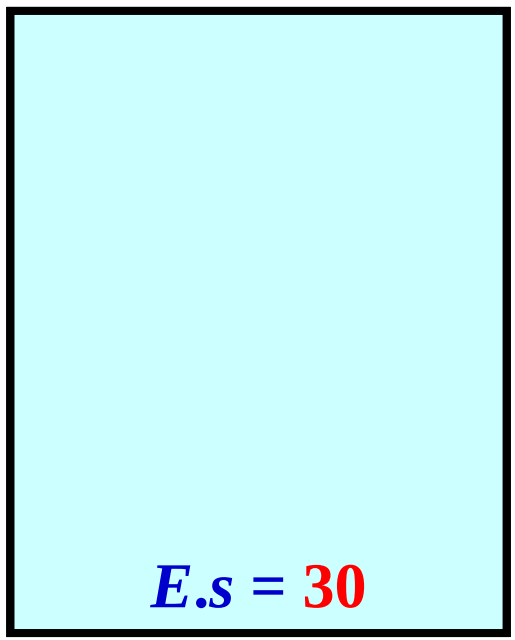
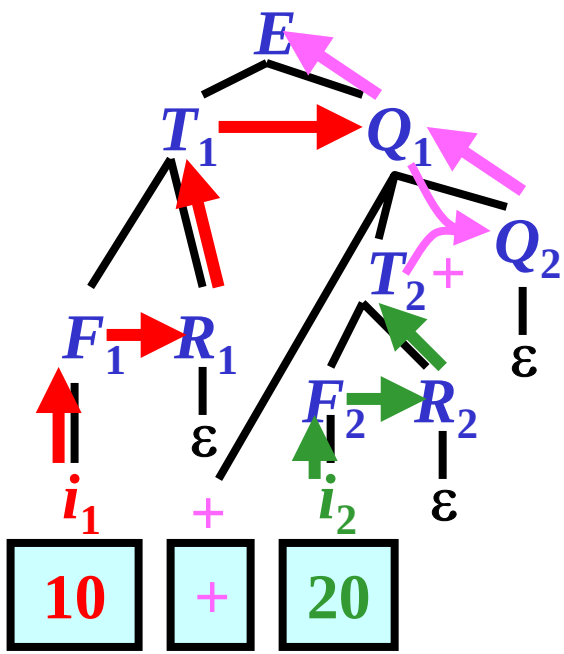


Illustration:



Semantic Analysis: Type Checking

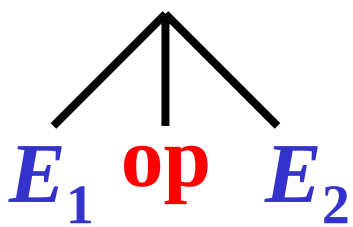
1) **Rule:**

Action:

***E*.type := *id*.type**

---

2) **Rule:**



Operation op is defined over types:

$$t_1 \text{ op } t_2 \rightarrow t_3$$

Action:

if (*E*$_1$.type = t_1 **or**
E$_1$.type is convertable to t_1)
and
(*E*$_2$.type = t_2 **or**
E$_2$.type is convertable to t_2)
then
E.type := t_3
else
Semantic Error.

Type Checking: Example 1/3

- **Make a type-checking for a grammar:**
- $G_{expr1} = (N, T, P, E)$, where $N = \{E, F, T\}$, $T = \{i, +, *, (,)\}$,
  $P = \{ E \to E+T, E \to T, T \to T*F, T \to F, F \to (E), F \to i \}$

- **Operators *, + are defined as:**
  - **int * int → int**
  - **int + int → int**
  - **real * real → real**
  - **real + real → real**

Possible Conversion:

- **From int to real**

Rule: $F \to i$	{*F.type* := *i.type*; generate(:=, *i.loc*, ,*F.loc*) }
Rule: $F_i \to (E_j)$	{F_i.*type* := E_j .*type*}
Rule: $T_i \to F_j$	{T_i.*type* := F_j .*type*}
Rule: $E_i \to T_j$	{E_i.*type* := T_j .*type*}

Type Checking: Example 2/3

Rule: $E_i \rightarrow E_j + T_k$ { **if** E_j**.*type*** = T_k**.*type* then begin**
E_i**.*type*** := E_j**.*type***
generate(+, E_j**.*loc***, T_k**.*loc***, E_i**.*loc*)**
end
else begin
generate(***new.loc***, *h*, ,)
if E_j**.*type* = *int* then begin**
generate(***int-to-real***, E_j**.*loc*** , , *h*)
generate(+, *h*, T_k**.*loc***, E_i**.*loc*)**
end
else begin
generate(***int-to-real***, T_k**.*loc*** , , *h*)
generate(+, E_j**.*loc***, *h*, E_i**.*loc*)**
end
E_i**.*type*** := ***real***
end
}

Type Checking: Example 3/3

Rule: $T_i \rightarrow T_j * F_k$ { **if** T_j**.*type* = F_k.*type* then begin**
T_i**.*type* :=** T_j**.*type***
generate(*, T_j.*loc*, F_k.*loc*, T_i.*loc*)
end
else begin
generate(***new.loc*, *h*, ,)**
if T_j **.*type* = *int* then begin**
generate(***int-to-real*,** T_j **.*loc* , , *h*)**
generate(*, *h*, F_k.*loc*, T_i.*loc*)
end
else begin
generate(***int-to-real*,** F_k**.*loc* , , *h*)**
generate(*, T_j.*loc*, *h*, T_i.*loc*)
end
T_i**.*type* := *real***
end
}

Short Evaluation (Jumping Code)

Idea:

- ***a*** **=** ***true*** **implies** ***a*** **or (… ? …) =** ***true***
- ***a*** **=** ***false*** **implies** ***a*** **and (… ? …) =** ***false***

Note: (… ? …) is <u>not evaluated</u>.

---

1) (*a* and *b*) = *p*:

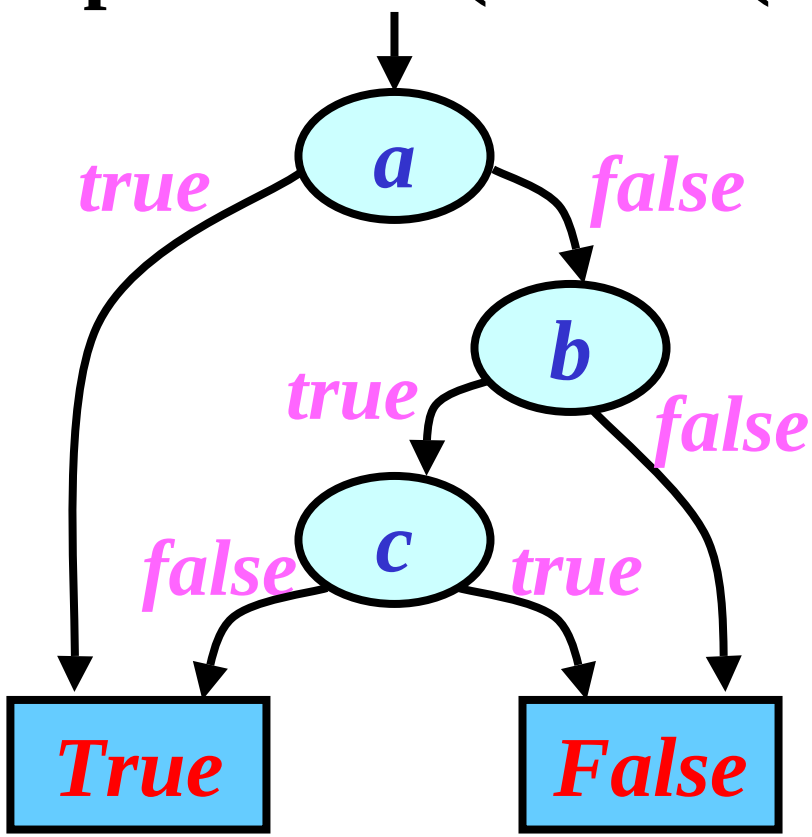
if ***a*** **=** ***false*** **then** ***p*** **=** ***false***
else ***p*** **=** ***b***

2) (*a* or *b*) = *p*:

if ***a*** **=** ***true*** **then** ***p*** **=** ***true***
else ***p*** **=** ***b***

Short Evaluation: Graphic Representation

Example: *a* or (*b* and (not *c*)):

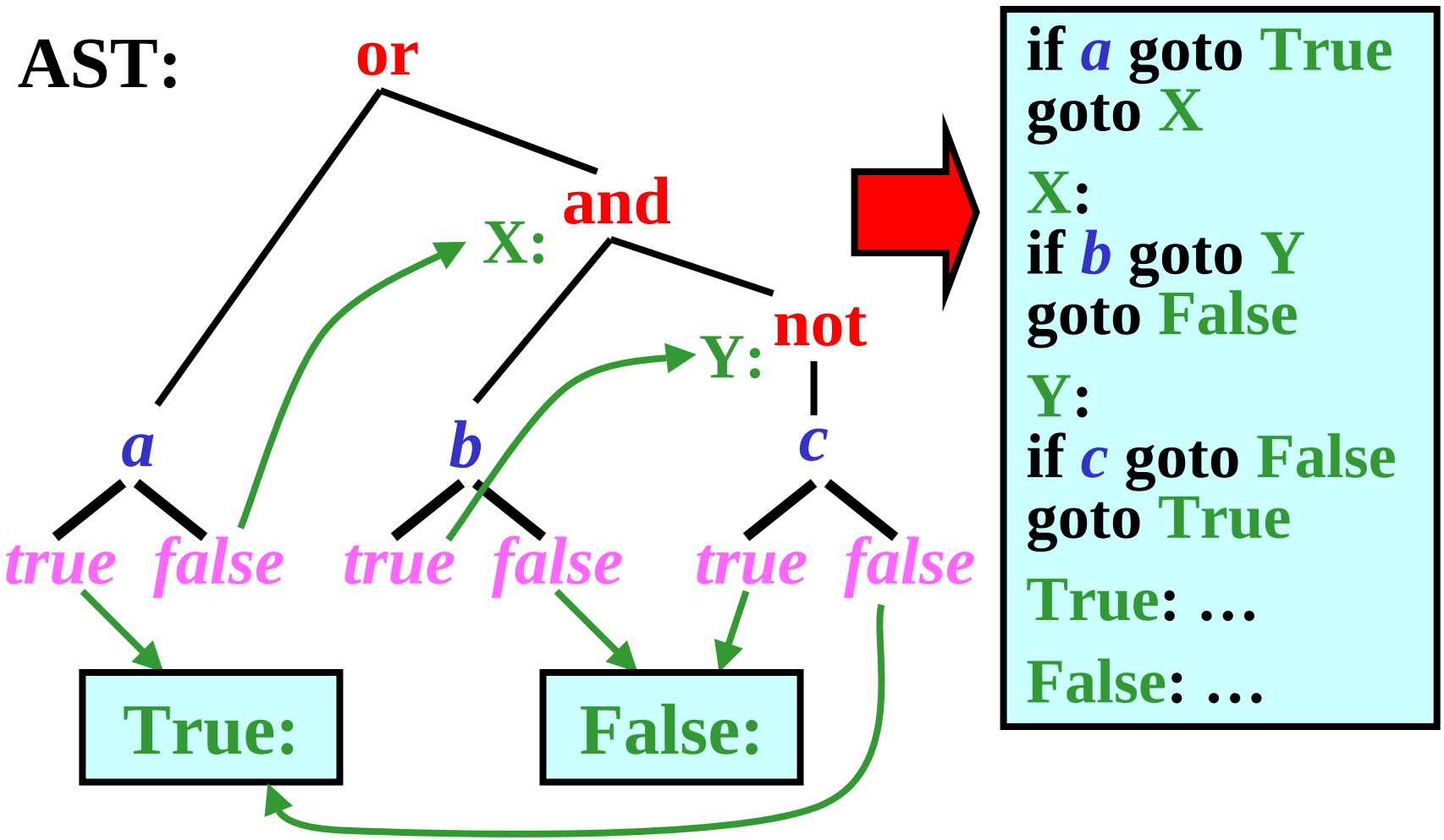



```
if a goto True
goto X
X:
if b goto Y
goto False
Y:
if c goto False
goto True
True: …
False: …
```

- Simulation of this graphic representation by 3AC jumps

Short Evaluation Using AST: Introduction

Example: ***a*** **or (*b* and (not *c*)):**

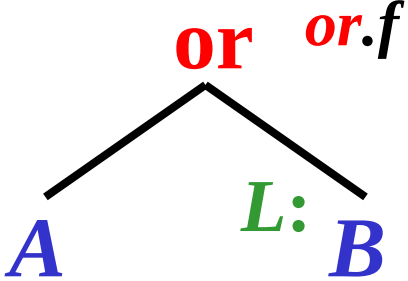


Short Evaluation Using AST: Implementation

• Every AST node, ***X***, has assigned two attributes ***X.t***, ***X.f***

Elementary ASTs:

1)

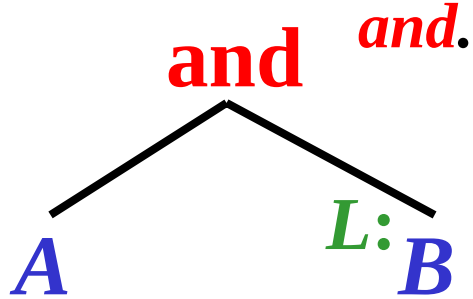


or.t
or.f

A.t := *or.t* *B.t* := *or.t*
A.f := *L* *B.f* := *or.f*

• **Note:** *L* = a new label

2)



and.t
and.f

A.t := *L* *B.t* := *and.t*
A.f := *and.f* *B.f* := *and.f*

• **Note:** *L* = a new label

3)

not.t
not.f

A.t := *not.f*
A.f := *not.t*

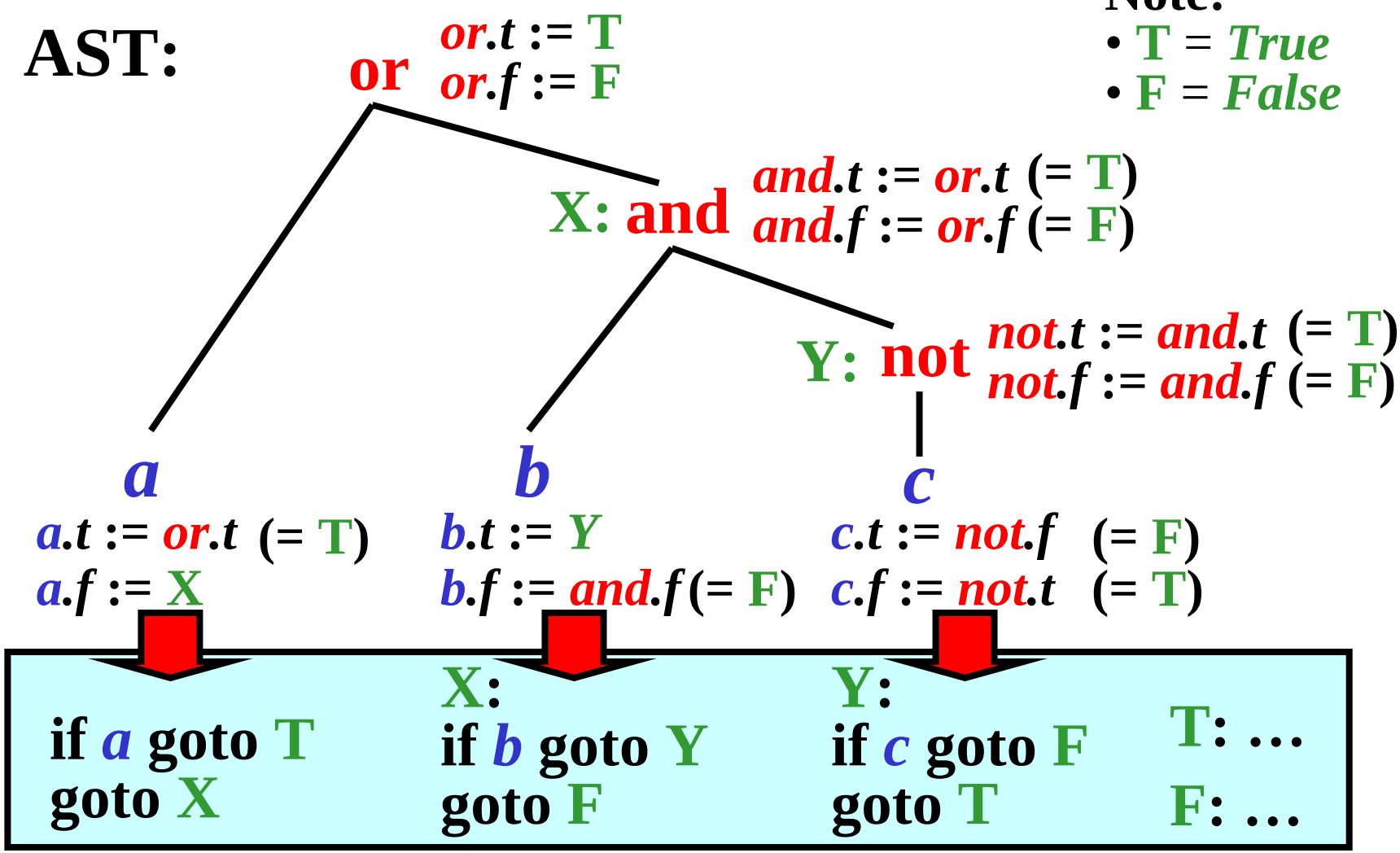
• **Initialization:** Let ***R*** is the root of AST, then:
R.t := **True**, ***R.f*** := **False** (**True** & **False** are labels)

• **Propagation:** Attributes are propagated from root to leaves in AST using rules **1)**, **2)** and **3)**.

Short Evaluation Using AST: Example

Example: ***a*** **or (** ***b*** **and (not** ***c*****)):**

AST:



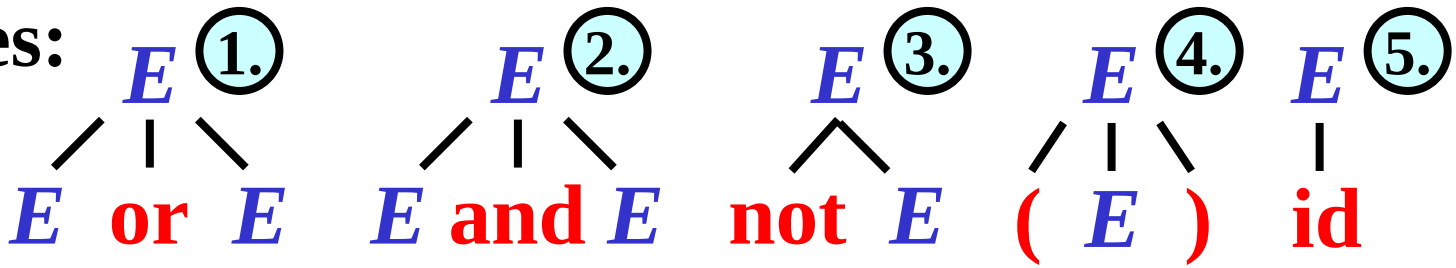
Note:

- **T** = ***True***
- **F** = ***False***

Short Evaluation: Direct Code Generation 1/5

- **Grammar for boolean expressions:**

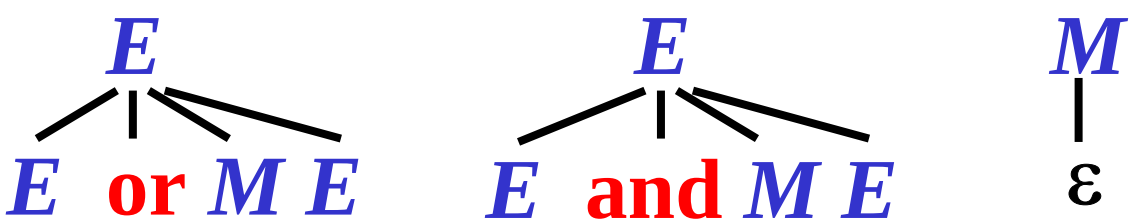
Rules:



Note: **Ambiguity!**

- **Modification of grammar:**

1) Replace rules ① & ② with:



2) Assign to each rule the following semantic action

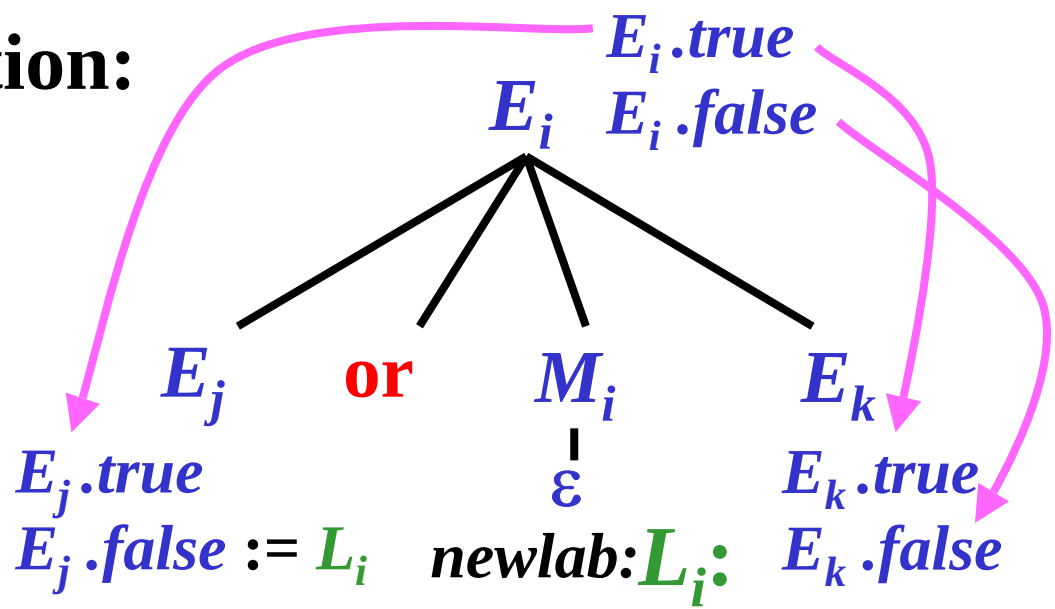
Short Evaluation: Direct Code Generation 2/5

$M_i \rightarrow \varepsilon$ {generate "M_i . ***lab:***"} // Generation of a new label

---

$E_i \rightarrow E_j$ **or** $M_i\ E_k$ {$M_i.lab$:= *GenerateNewLab*;
$E_j.true := E_i.true$; $E_j.false := M_i.lab$
$E_k.true := E_i.true$; $E_k.false := E_i.false$ }

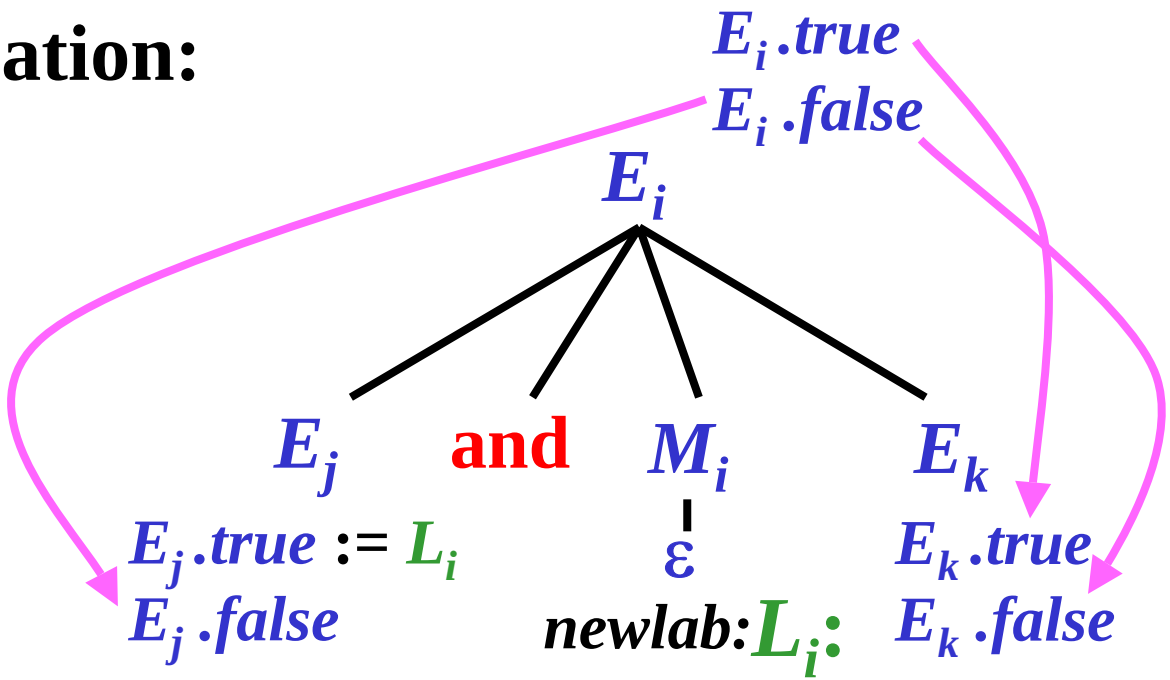
Illustration:



Short Evaluation: Direct Code Generation 3/5

$E_i \rightarrow E_j$ **and** $M_i\ E_k$ {$M_i.lab$:= *GenerateNewLab*;
$E_j.true := M_i.lab$; $E_j.false := E_i.false$
$E_k.true := E_i.true$; $E_k.false := E_i.false$ }

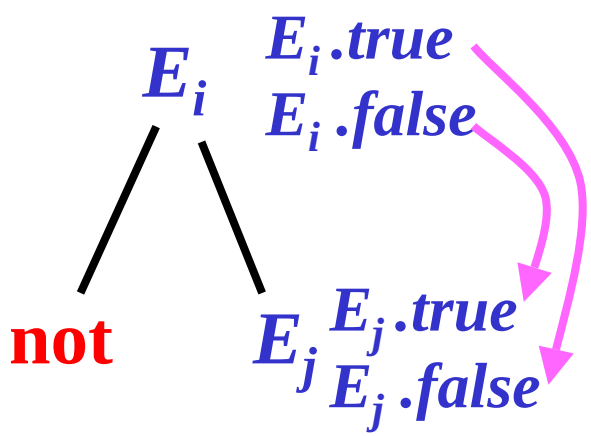
Illustration:



Short Evaluation: Direct Code Generation 4/5

$E_i \rightarrow$ **not** E_j { $E_j.true$:= $E_i.false$;
$E_j.false$:= $E_i.true$ }

Illustration:

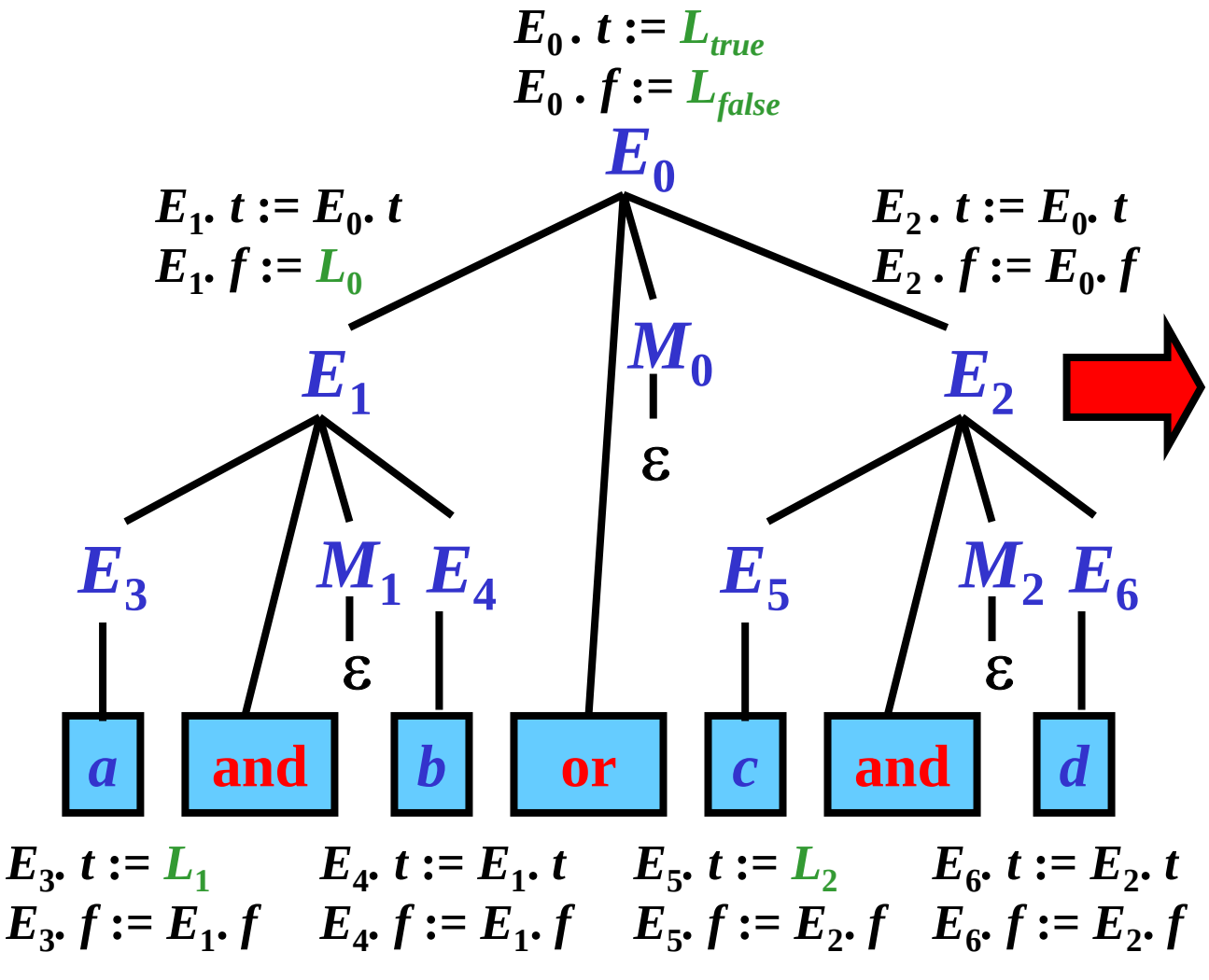


$E_i \rightarrow (E_j)$ {$E_j.true$:= $E_i.true$;
$E_j.false$:= $E_i.false$ }

$E_i \rightarrow id_j$ { generate "**if** $id_j.val$ **goto** $E_i.true$";
generate "**goto** $E_i.false$" }

Short Evaluation: Direct Code Generation 5/5

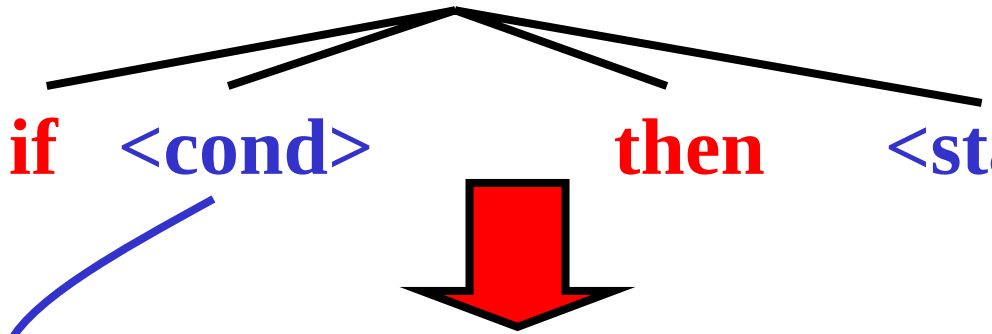
Example: *a* and *b* or *c* and *d*:



if ***a*** *goto* L_1
goto L_0
L_1:
if ***b*** *goto* L_{true}
goto L_0
L_0:
if ***c*** *goto* L_2
goto L_{false}
L_2:
if ***d*** *goto* L_{true}
goto L_{false}
L_{true}: …
L_{false}: …

Branching: If-Then

Rule: <if-then>



Semantic action:

```
{ // evaluation of cond
  // to c.val
(not , c.val, , c.val)
(goto, c.val, , L1   )
 // code of stat1      }
(lab , L1   , ,       )
```

Branching: If-Then-Else

Rule: <if-then-else>

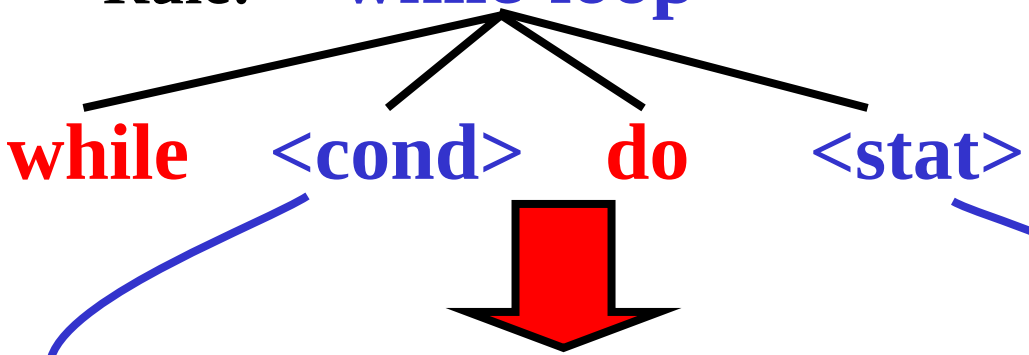
if <cond> then <stat$_1$> else <stat$_2$>

Semantic action:

```
{ // evaluation of cond
  // to c.val
(not , c.val, , c.val)
(goto, c.val, , L1   )
 // code of stat1      }
(goto,       , , L2   )
(lab , L1    , ,      )
 // code of stat2      }
(lab , L2    , ,      )
```

While Loop

Rule: <while-loop>



Semantic action:

```
(lab , L1   , ,        )
// evaluation of cond
// to c.val
(not , c.val, , c.val)
(goto, c.val, , L2    )
 // code of stat
(goto,       , , L1    )
(lab ,  L2   , ,        )
```

Repeat Loop

Rule: **<repeat-loop>**

repeat **<stat>** **until** **<cond>**

Semantic action:

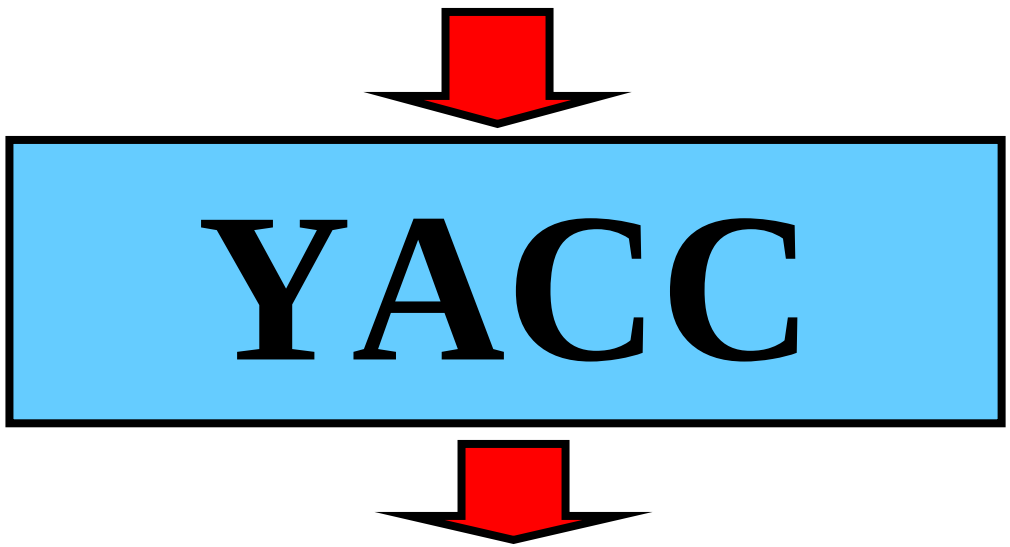
```
(lab , L1   , ,        )
{// code of stat

 // evaluation of cond
 // to c.val
(not , c.val, , c.val)
(goto, c.val, , L1   )
```

Yacc: Basic Idea

- Automatic construction of **parser** from **CFG**
- Yacc compiler × Yacc language
- *Yacc* from *Y*et *a*nother *c*ompiler *c*ompiler

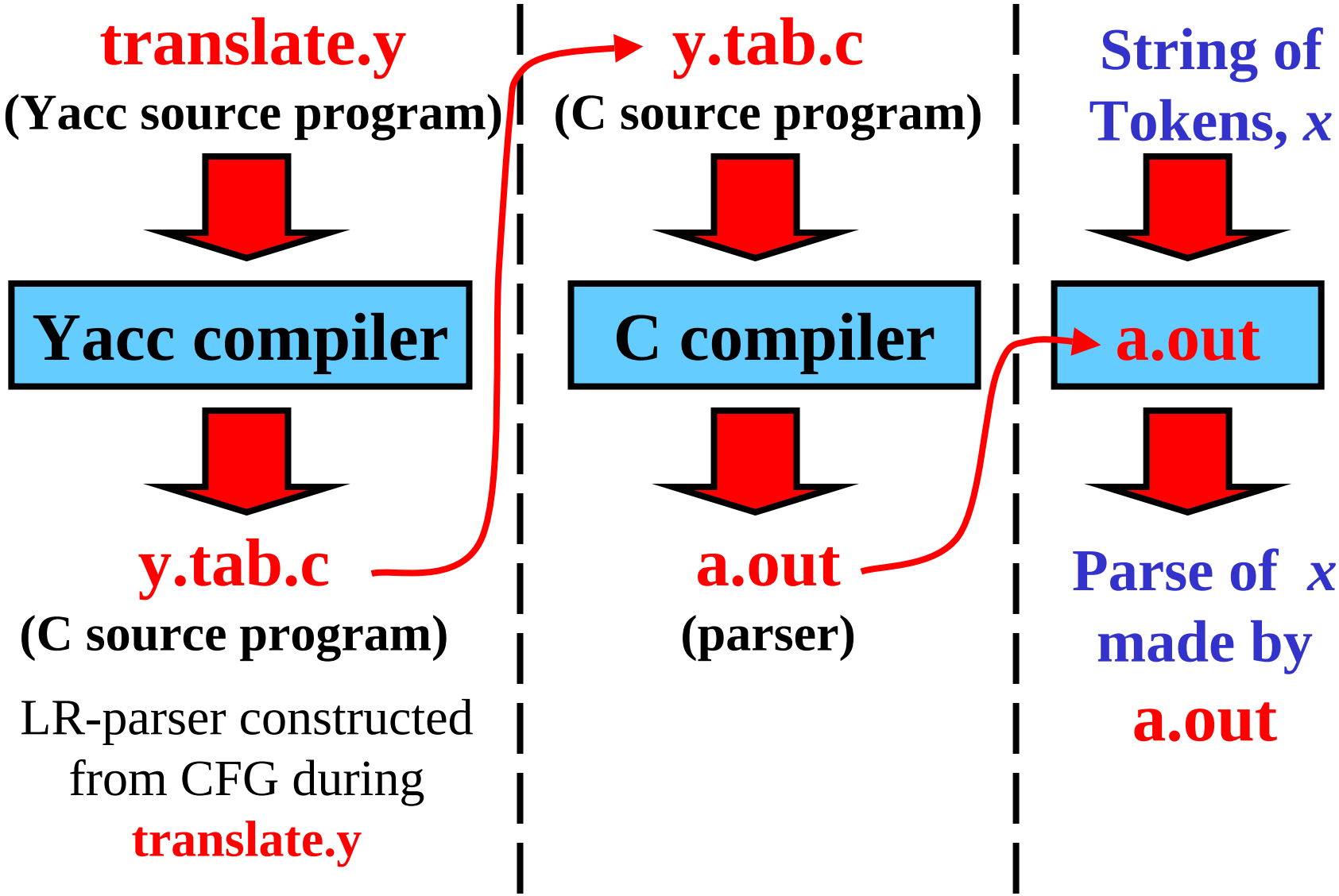
Ilustrace:

Context-free grammar, *G*



Parser for *G*

Yacc: Phases of Compilation



Structure of Yacc Source Program

/* Section I: Declaration */

$d_1, d_2, \ldots, d_i$

%% **/* End of Section I*/**

/* Section II: Translation rules */

$r_1, r_2, \ldots, r_j$

%% **/* End of Section II*/**

/* Section III: Auxiliary procedures*/

$p_1, p_2, \ldots, p_k$

Description of Grammar in Yacc

- **Nonterminals:** names (= strings)
- **Example:** `prog`, `stat`, `expr`, …

- **Terminals:** Characters in quotes or declared tokens
- **Example:** `'+'`, `'*'`, `'('`, `')'`, `ID`, `INTEGER`

- **Rules:** Set of *A*-rules { $A \to x_1$, $A \to x_2$, … $A \to x_n$ }
  is written as

```
A : x1
  | x2
  …
  | xn
```

- **Example:**

```
expr : expr '+' expr
     | ID
```

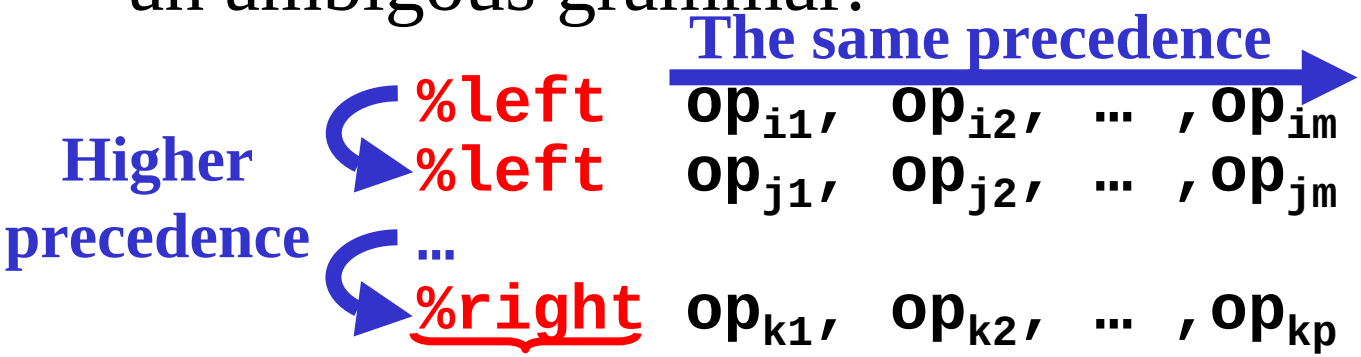
- **Start Nonterminal:** A left side of the first rule.

Section I: Declaration

1) Declaration of tokens

```
%token TYPE_OF_TOKEN
```

2) Specification of asociativity & precedence in an ambigous grammar.



Example:

```
%token INTEGER
%token ID
%left ‘+’
%left ‘*’
```

Section II: Translation Rules

- Translation rules are in the form:

```
Rule    Semantic_Action
```

- **`Semantic_Action`** is a program routine that specifies what to do if **`Rule`** is used.

Special symbols for a rule, *r*:

$$ = attribute of *r*'s left-hand side

$*i* = attribute of the *i*-th symbols on *r*'s right-hand side

---

Example:

```
expr : expr '+' expr {$$ = $1 + $3}
     | expr '*' expr {$$ = $1 * $3}
     | '(' expr ')'  {$$ = $2}
     | INTEGER
     | ID
```

Section III: Auxiliary Procedures

- Auxiliary procedures used by translation rules

Note: If the Yacc-parser do not cooperate with a scanner (e.g. Lex), then there is **yylex()** implemented in this section.

Example:

```
int yylex() {
      /* Get the next token */
      &yylval = attribute;
      return TYPE_OF_TOKEN;
}
```

Complete Source Program in Yacc

```
%token INTEGER
%token ID
%left ‘+’
%left ‘*’

%%

expr : expr ‘+’ expr {$$ = $1 + $3}
     | expr ‘*’ expr {$$ = $1 * $3}
     | ‘(’ expr ‘)’  {$$ = $2}
     | INTEGER
     | ID

%%

int yylex () { …}
```