SDD, Attribute Grammar and SDT

Translation

- So far we have talked about parsing of a language. But our main goal is translation.
- Semantic actions to translate the source language to target language often go hand-in-hand with parsing. It is called syntax-directed translation.

Translation

- To perform semantic actions along with parsing (reduction for example), we may associate computation with the production rules. Computed values are propagated as attributes of non-terminals.
- Otherwise the parse tree may be built explicitly and semantic actions are performed while traversing the tree.

Example

Consider the following production rule of the classic expression grammar: $E \rightarrow E_1 + T$[a].

We consider three different translations:

- implementation of a simple calculator,
- conversion of an infix expression to a postfix expression,
- general purpose code generation.

[a] We have used subscript to differentiate between two instances of E.

Example: Calculator

We have already seen that the only attribute of E and T are values corresponding to the sub-trees of E and T. Let us call the attribute to be val.
The semantic action associated with the given production rule is,
$E \rightarrow E_1 + T \ \{E{\cdot}\text{val} = E_1{\cdot}\text{val} + T{\cdot}\text{val}\}$[a].

[a] In `bison` this gets translated to `$$ = $1 + $3`.

Note

The action may take place when $E_1 + T$ is reduced to E. The computed value is saved as the attribute of E. There is no other side-effect.

Note

But in the calculator if we want to keep a provision of storing a value as a named object (variable), we need a symbol table where the variable names and their values are stored.
In this case the semantic action of $ES \rightarrow id := E$ will changes the state of computation (side effect) by entering the $E \cdot \text{val}$ in the symbol table corresponding to $id.\text{name}$.

Example: Infix to Postfix Conversion

The problem is to convert an infix arithmetic expression to a postfix expression. Both input and output are strings of characters.
So the attribute of each non-terminal can be a string of characters (char *). Let the name of the field be exp. The semantic action associated with the production rule is as follows:

Example: Infix to Postfix

$E \rightarrow E_1 + T$

```
{
  E.exp=(char*)malloc(strlen(E1.exp)+
                      strlen(T.exp)+4);
  strcpy(E.exp, E1.exp); strcat(E.exp, " ");
  strcat(E.exp, T.exp);
  strcat(E.exp, " + ");
  free(E1.exp); free(T.exp);
}
```

Again there is no side-effect

Example: Code Generation

We assume that the computed values corresponding to the expressions E_1 and T are stored in temporary locations[a].
The main attribute of a non-terminal in this case is the address or index of the location[b] in the symbol table.

[a] Compiler defines temporary variables or virtual registers for this purpose and enters them in the symbol table.

[b] Note that the location has not yet been bound to memory or physical register. The address may be an index of the symbol table.

Example: Code Generation

$E \rightarrow E_1 + T$

```
{
 E.loc = newLoc();
 codeGen(assignPlus, E.loc, E1.loc, T.loc);
}
```

where assignPlus[a] means
`E.loc = E1.loc + T.loc`.

[a]This is an example of an intermediate code generated from the source language expression.

Note

This action has a side-effect as it makes an entry of the new location in the symbol table. It also adds the corresponding intermediate code in the code stream.

Associating Information

We associate information to language constructs by attaching attributes to the grammar symbols. Computation of these attributes are associated with the production rules in the form of semantic rules.
Initial attribute values are supplied by the scanner.

Definition

A syntax-directed definition is a context-free grammar where attributes are associated with the grammar symbols and semantic rules for computing the attributes are associated with the production rules. These grammars are also called attribute grammars when the definition does not have any side-effect. There is no strict order of evaluation of the attributes, but there should not be any circularity.

Definition

A syntax-directed translation is an executable specification of SDD. Fragments of programs are associated to different points in the production rules. The order of execution is important in this case.

Example

$$A \rightarrow \{\text{Action}_1\}\ B\ \{\text{Action}_2\}\ C\ \{\text{Action}_3\}$$

$\underline{\text{Action}_1}$: takes place before parsing of the input corresponding to the non-terminal B.
$\underline{\text{Action}_2}$: takes place after consuming the input for B, but before consuming the input for C.
$\underline{\text{Action}_3}$: takes place at the time of reduction of BC to A or after consuming the input corresponding to BC.

Note

- Embedded action may create some problem in parser generator like Bison.
- Bison replaces the embedded action in a production rule by an ε-production and associates the embedded action with the new rule.

Note

- But this may change the nature of the grammar. As an example, the grammar $S \rightarrow A|B, A \rightarrow aba, B \rightarrow abb$ is LALR. But if an embedded action is introduced as shown, $S \rightarrow A|B, A \rightarrow a$ {action} $ba, B \rightarrow abb$. Bison modifies the grammar to $S \rightarrow A|B, A \rightarrow aMba, B \rightarrow abb, M \rightarrow \varepsilon$ {action} , and the grammar is no longer LALR.

General Approach for SDT

Construct the complete parse tree. Compute the attributes of non-terminals by traversing the tree. Explicit construction of a tree is costly in terms of time and space.
There are SDDs that do not require explicit construction of the parse tree. We shall consider two of them - S-attributed and L-attributed definitions.

A Complete Example

Consider the following grammar (augmented) of binary strings:

$$
\begin{aligned}
0: &\quad S' \rightarrow N\$ \\
1: &\quad N \rightarrow S\ L \\
2: &\quad S \rightarrow + \\
3: &\quad S \rightarrow - \\
4: &\quad L \rightarrow L\ B \\
5: &\quad L \rightarrow B \\
6: &\quad B \rightarrow 0 \\
7: &\quad B \rightarrow 1
\end{aligned}
$$

Note

- We wish to translate a signed binary string to a signed decimal string.
- We first construct the $LR(0)$ automaton of the grammar and find that the grammar is SLR.
- We associate attributes to the non-terminals.

$LR(0)$ Automaton

q_0 :	$S' \to \bullet N\$ \quad N \to \bullet SL \quad S \to \bullet +$ $S \to \bullet -$
q_1 :	$S' \to N \bullet \$$
q_2 :	$N \to S \bullet L \quad L \to \bullet LB \quad L \to \bullet B$ $B \to \bullet 0 \qquad B \to \bullet 1$
q_3 :	$S \to + \bullet$
q_4 :	$S \to - \bullet$
q_5 :	$N \to SL \bullet \quad L \to L \bullet B \quad B \to \bullet 0$ $B \to \bullet 1$

$LR(0)$ Automaton

q_6 :	$L \rightarrow B\bullet$
q_7 :	$B \rightarrow 0\bullet$
q_8 :	$B \rightarrow 1\bullet$
q_9 :	$L \rightarrow LB\bullet$

SLR Parsing Table

S	*Action*					*Goto*			
	$+$	$-$	0	1	\$	N	S	L	B
0	s_3	s_4				1	2		
1					Acc				
2			s_7	s_8				5	6
3			r_6	r_6					
4			r_7	r_7					
5			s_7	s_8	r_1				9

SLR Parsing Table

S	*Action*					*Goto*			
	$+$	$-$	0	1	\$	N	S	L	B
6			r_5	r_5	r_5				
7			r_6	r_6	r_6				
8			r_7	r_7	r_7				

Attributes of Non-Terminals

Following are the attributes of different non-terminals:

Non-terminal	Attribute	Type
N	val	`int`
S	sign	`char`
L	val	`int`
B	val	`int`

Definition or Action

0 :	S'	$\rightarrow$	N	Accept
1 :	N	$\rightarrow$	SL	`if (S.sign == '-') N.val= - L.val;`
				`else N.val = L.val;`
2 :	S	$\rightarrow$	$+$	`S.sign = '+';`
3 :	S	$\rightarrow$	$-$	`S.sign = '-';`
4 :	L	$\rightarrow$	L_1B	`L.val = 2*L1.val+B.val;`
5 :	L	$\rightarrow$	B	`L.val = B.val;`
6 :	B	$\rightarrow$	0	`B.val = 0;`
7 :	B	$\rightarrow$	1	`B.val = 1;`

Example: Parsing & Value Stack

Type	Stack →	Input→	Action/Value
Parsing	$0	+101$	shift
Value	$		
Parsing	$03	101$	reduce
Value	$+		
Parsing	$02	101$	shift
Value	$S		S.sign='+'
Parsing	$028	01$	reduce
Value	$S1		

Example: Parsing & Value Stack

Type	Stack →	Input→	Action/Value
Parsing	$026	01$	reduce
Value	$SB		B.val = 1
Parsing	$025	01$	shift
Value	$SL		L.val = B.val
Parsing	$0257	1$	reduce
Value	$SL0		
Parsing	$0259	1$	reduce
Value	$SLB		B.val = 0

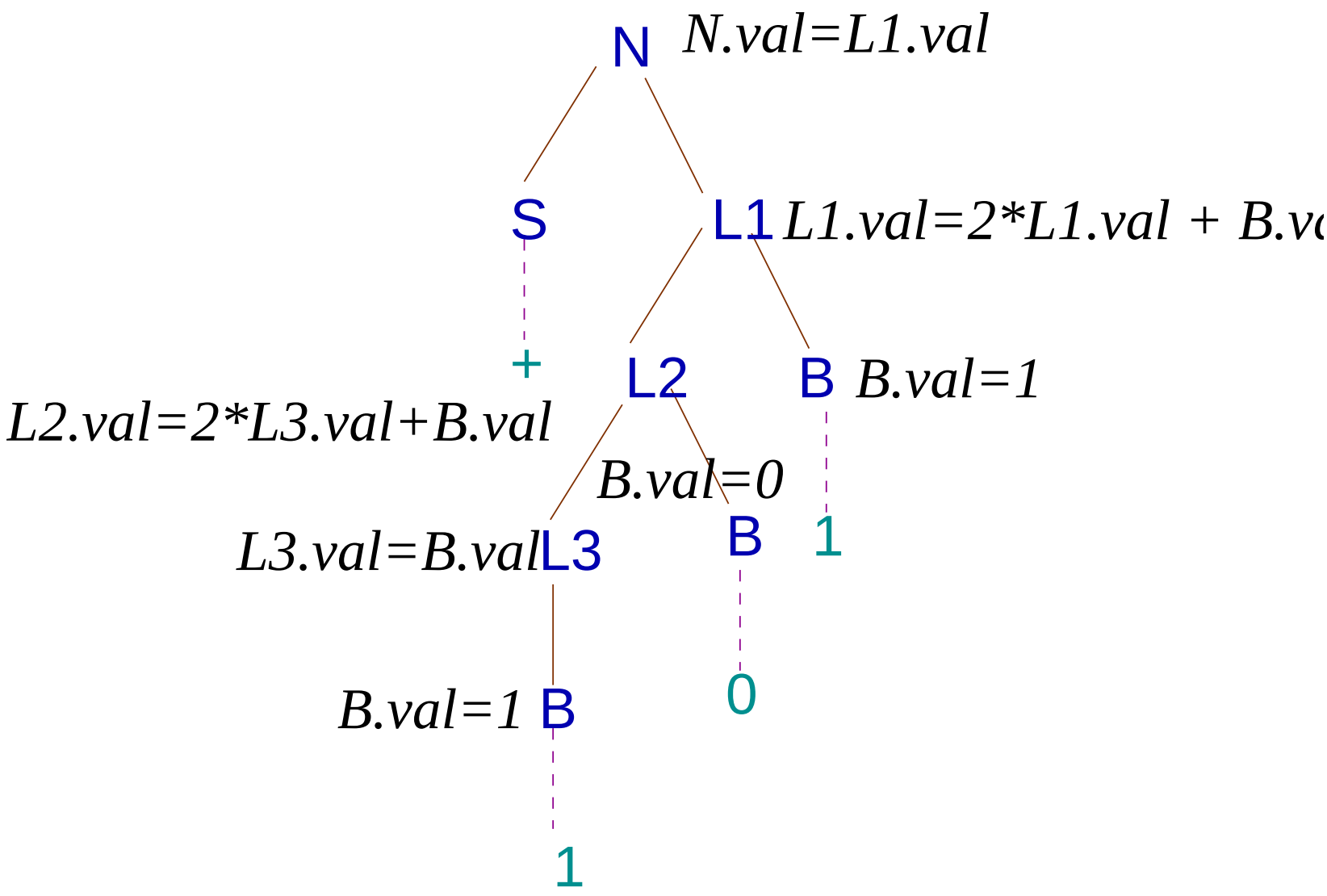
Example: Parsing & Value Stack

Type	Stack →	Input→	Action/Value
Parsing	$025	1$	shift
Value	$SL		L.val = 2*L1.val + B.val
Parsing	$0258	$	reduce
Value	$SL1		
Parsing	$0259	$	reduce
Value	$SLB		B.val=1
Parsing	$025	$	reduce
Value	$SL		L.val = 2*L1.val + B.val

Example: Parsing & Value Stack

Type	Stack →	Input→	Action/Value
Parsing	`$01`	`$`	Accept
Value	`$N`		`N.val = +L.val`

Decorated Parse Tree



Synthesized Attribute

- In this example the value of an attribute of a non-terminal is either coming from the scanner[a] or it is computed from the attributes of its children.
- This type of attribute is known as a synthesized attribute.

[a] Attribute of a terminal comes from the scanner.

S-Attributed

- An attributed grammar is called S-attributed if every attribute is synthesized.
- Attributes in such a grammar can be easily computed during a bottom-up parsing.

Note

Attribute of a non-terminal depends on the nature of translation. But it may also depend on the nature of the grammar.

Exercise

Following grammar of 2's complement numerals is to be translated to a signed decimal integer.

$$1: \quad N \rightarrow L$$

$$2: \quad L \rightarrow L\ B$$

$$3: \quad L \rightarrow B$$

$$4: \quad B \rightarrow 0$$

$$5: \quad B \rightarrow 1$$

Exercise

Associate attributes to the non-terminals and give rules for semantic actions. Write bison specification for the grammar.

Example

Consider a right-recursive grammar of signed binary strings:

$$
\begin{aligned}
0: &\quad S' \rightarrow N \\
1: &\quad N \rightarrow S\ L \\
2: &\quad S \rightarrow + \\
3: &\quad S \rightarrow - \\
4: &\quad L \rightarrow B\ L \\
5: &\quad L \rightarrow B \\
6: &\quad B \rightarrow 0 \\
7: &\quad B \rightarrow 1
\end{aligned}
$$

Attributes of Non-Terminals

We need a new attribute of L to remember the bit position:

Non-terminal	Attribute	Type
N	val	`int`
S	sign	`char`
L	val	`int`
	pos	`int`
B	val	`int`

Action for Rules

$0:$	S'	$\rightarrow$	N	Accept
$1:$	N	$\rightarrow$	SL	`if (S.sign == '-') N.val=- L.val;`
				`else N.val = L.val;`
$2:$	S	$\rightarrow$	$+$	`S.sign = '+';`
$3:$	S	$\rightarrow$	$-$	`S.sign = '-';`
$4:$	L	$\rightarrow$	BL_1	`if(B.val)`
				`  L.val=pow(2,L1.pos)+L1.val;`
				`else L.val=L1.val;`
				`L.pos=L1.pos+1;`

Actions for Production Rules

$5: \quad L \rightarrow B$ `L.val = B.val; L.pos = 1`

$6: \quad B \rightarrow 0$ `B.val = 0;`

$7: \quad B \rightarrow 1$ `B.val = 1;`

Example

Consider the following grammar for variable declaration:

$$1: \quad D \rightarrow T\ L\ ;$$

$$2: \quad T \rightarrow \texttt{int}$$

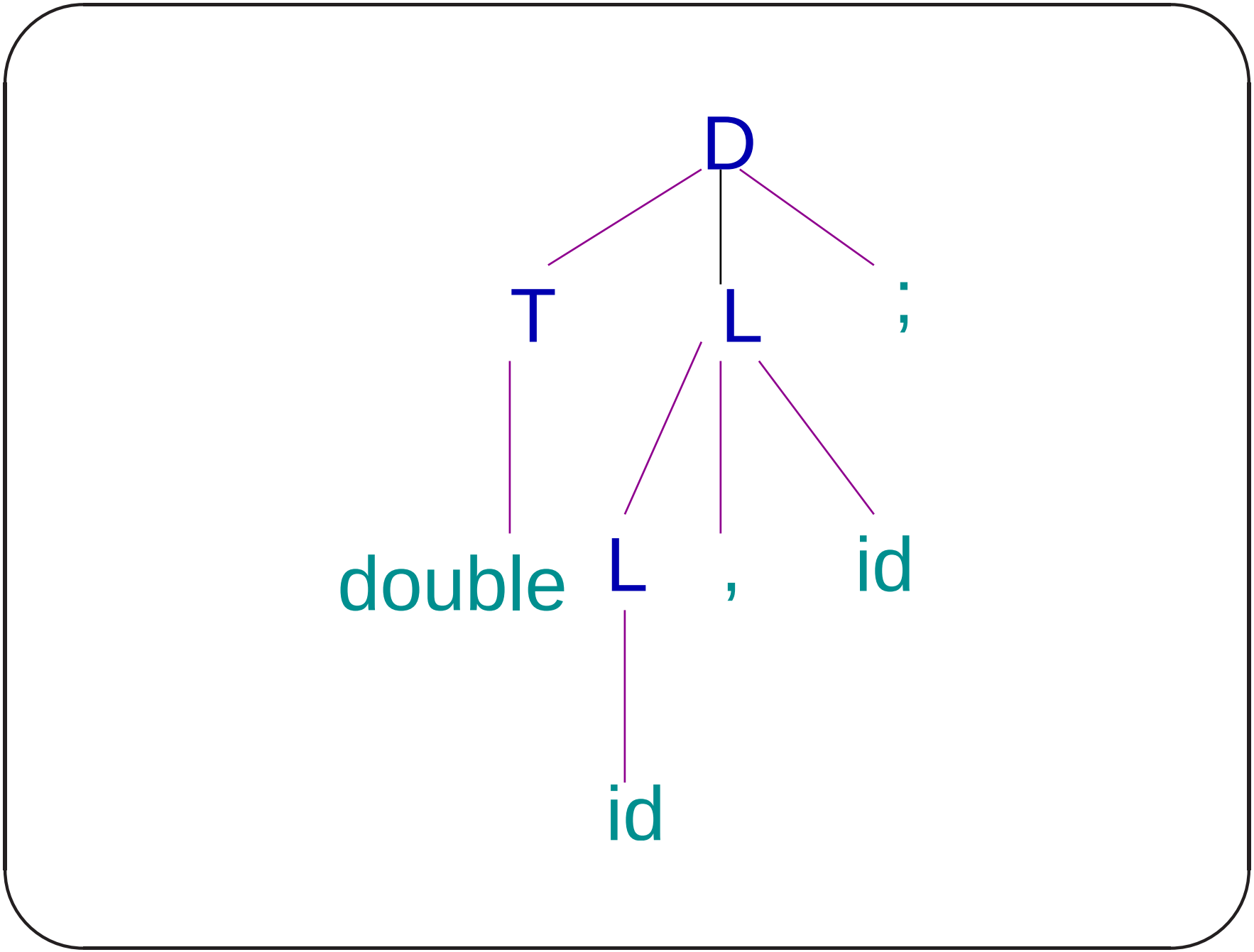
$$3: \quad T \rightarrow \texttt{double}$$

$$4: \quad L \rightarrow L\ ,\ \texttt{id}$$

$$5: \quad L \rightarrow \texttt{id}$$

Parse Tree

The parse tree for the string `double id, id;` is as follows:


D
T
L
;
double
L
,
id
id

Note

When an `id` is reduced to the non-terminal L, it is inserted in the symbol table along with its type information[a]. The type information is not available from any subtree rooted at L. It has to be inherited from T via the root D.

[a]The type information is important for space allocation, representation, operations, correctness and other purposes.

SDDefinition

1 :	D	$\rightarrow$	TL;	L.type = T.type
2 :	T	$\rightarrow$	int	T.type = INT
3 :	T	$\rightarrow$	double	T.type = DOUBLE
4 :	L	$\rightarrow$	L_1, id	L1.type = L.type
				addSym(id.name, L.type)
5 :	L	$\rightarrow$	id	addSym(id.name, L.type)

Inherited Attribute

Let B be a non-terminal at a parse tree node N. Let M be the parent of N. An inherited attribute $B.i$ is defined by the semantic rule associated with the production rule of M (parent).
Inherited attribute at the node N is defined in terms of M, N and N's siblings.
In the previous example the non-terminal L gets the attribute from T as an inherited attribute.

S-Attributed Definitions

An SDD is S-attributed if every attribute is synthesized. The attribute grammar may be called S-attributed grammar.

This definition can be implemented in a LR-parser as the reduction traverse the parse-tree in postorder.

L-Attributed Definitions

An SDD is called L-attributed ('L' for left) if each attribute is either

- synthesized, or
- inherited with the following restrictions: if $A \to \alpha_1 \alpha_2 \cdots \alpha_n$ be a production rule, and α_k has an inherited attribute 'a' computed in this rule, then the computation may involve

L-Attributed Grammar

- inherited attribute of A (parent), or
- attributes (inherited or synthesized) of $\alpha_1, \alpha_2, \cdots, \alpha_{k-1}$ (symbols to the left of α_k),
- attributes of α_k, provided no dependency cycle[a] is formed.

[a] $A \rightarrow B \ \{ A.s = B.i; B.i = A.s + k \}$.

Rules

The type definition mentioned earlier is L-attributed.

1 :	D	$\rightarrow$	TL;	`L.type = T.type`
2 :	T	$\rightarrow$	`int`	`T.type = INT`
3 :	T	$\rightarrow$	`double`	`T.type = DOUBLE`
4 :	L	$\rightarrow$	L_1, `id`	`L1.type = L.type`
				`addSym(id.name, L.type)`
5 :	L	$\rightarrow$	`id`	`addSym(id.name, L.type)`

Note

The question is how to propagate the type information in a parser generated by `bison`?
The non-terminal T gets the value of synthesized type attribute when a T-production rule is reduced.
But that cannot be propagated as an attribute of the non-terminal L as this non-terminal is not present in the stack.

Solution I

A very ad hoc solution is to use a global variable to hold the type value.

T	$\rightarrow$	`int`	`type = INT`
T	$\rightarrow$	`double`	`type = DOUBLE`
L	$\rightarrow$	L_1, `id`	`addSym(id.name, type)`
L	$\rightarrow$	`id`	`addSym(id.name, type)`

Solution II

We introduce a different attribute of L, a list of symbol table entries corresponding to different identifiers, and initialize their types at the end.

1 :	$D \rightarrow TL;$		`initType(L.list, T.type)`
2 :	$T \rightarrow$	`int`	`T.type = INT`
3 :	$T \rightarrow$	`double`	`T.type = DOUBLE`
4 :	$L \rightarrow L_1,$	`id`	`L.list = L_1.list +`
			`mklist(addSym(id.name))`
5 :	$L \rightarrow$	`id`	`L.list =`
			`mklist(addSym(id.name))`

Read '+' as append in the list.

Solution III

We can device another solution from the value stack. For that we consider the states of $LR(0)$ automaton of the grammar.

$LR(0)$ Automaton

q_0 :	$S \to \bullet D \quad D \to \bullet T\,L; \quad T \to \bullet\texttt{int}$ $T \to \bullet\texttt{double}$
q_1 :	$S \to D\bullet$
q_2 :	$D \to T \bullet L \quad L \to \bullet L, \texttt{id} \quad L \to \bullet\texttt{id}$
q_3 :	$T \to \texttt{int}\bullet$
q_4 :	$S \to \texttt{double}\bullet$
q_5 :	$D \to T\,L\bullet; \quad L \to L\bullet, \texttt{id}$
q_6 :	$L \to \texttt{id}\bullet$

$LR(0)$ Automaton

q_7 :	$L \rightarrow L, \bullet \texttt{id}$
q_8 :	$L \rightarrow L, \texttt{id} \bullet$
q_9 :	$D \rightarrow TL; \bullet$

Example: Parsing & Value Stack

Type	Stack →	Input→	Action/Value
Par	$0	int id, id;$	shift
Val	$		
Par	$03	id, id;$	reduce
Val	$int		
Par	$02	id, id;$	shift
Val	$T		T.type=INT
Par	$026	, id;$	reduce
Val	$T id		

Example: Parsing & Value Stack

Type	Stack →	Input→	Action/Value
Par	$025	, id;$	reduce
Val	$T L		addSym(id.name,L.type)

How does L gets the type information. Note that in bison $L \equiv \$\$$ and `id` $\equiv \$1$. But the type information is available in T in the stack, below the handle.

Type	Stack →	Input→	Action/Value
Par	$0257	id;$	shift
Val	$T L ,		

Example: Parsing & Value Stack

Type	Stack →	Input→	Action/Value
Par	$02578	;$	reduce
Val	$T L , id		
Par	$025	;$	
Val	$T L		addSym(id.name,L.type)

Again the type information is available just below the handle.

Note

In Bison the attribute below the handle can be accessed. In this case the non-terminal T corresponds to `$0` and its type attribute is `$0.type`.

Note

Often a natural grammar is transformed to facilitate some type of parsing, and the parse tree does not match with the abstract syntax tree of the language.
As an example the left recursion is removed for $LL(1)$ parsing. How does the original S-attributed grammar gets modified after the removal of left-recursion? We consider the following example.

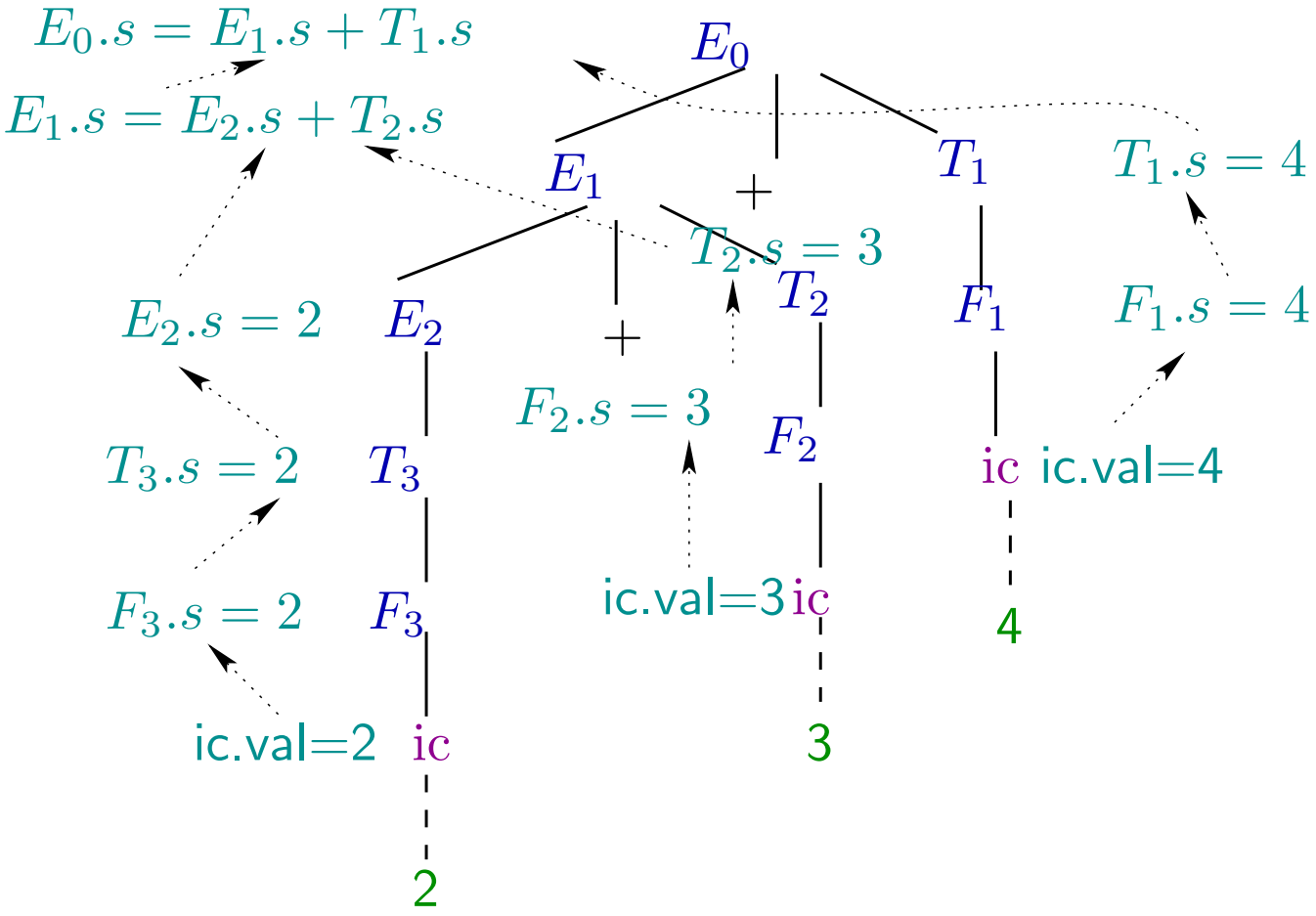
S-Attributed Expression Grammar

S	$\rightarrow$	$E\$$	`{ print E.val }`
E	$\rightarrow$	$E_1 + T$	`{ E.val = E1.val + T.val}`
E	$\rightarrow$	T	`{ E.val = T.val}`
T	$\rightarrow$	$T_1 * F$	`{ T.val = T1.val * F.val}`
T	$\rightarrow$	F	`{ T.val = F.val}`
F	$\rightarrow$	(F)	`{ F.val = E.val}`
F	$\rightarrow$	`ic`	`{ F.val = ic.val}`

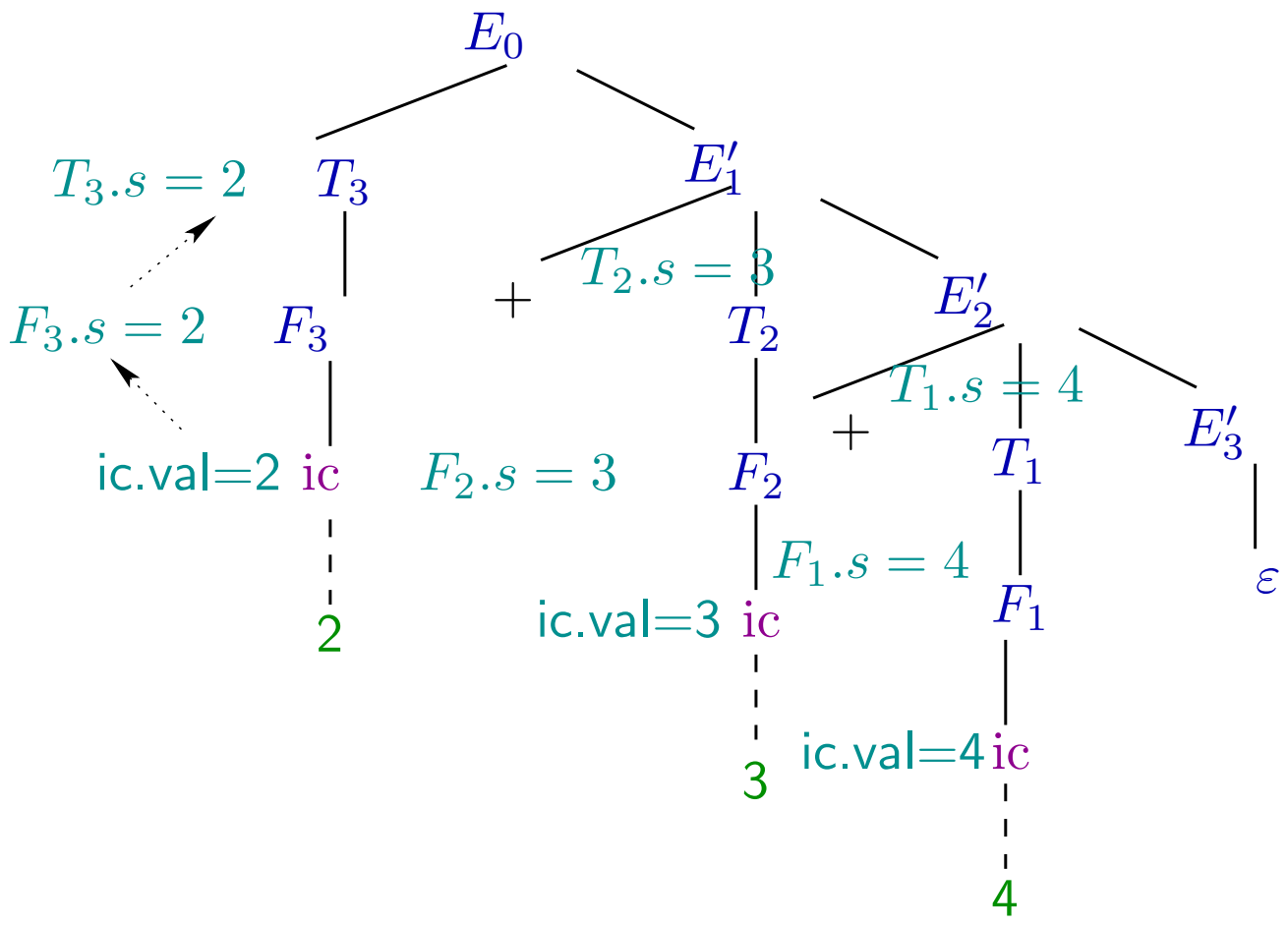
Equivalent $LL(1)$ Grammar

$$
\begin{aligned}
S &\rightarrow E\$ \\
E &\rightarrow TE' \\
E' &\rightarrow +TE' \\
E' &\rightarrow \varepsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \\
T' &\rightarrow \varepsilon \\
F &\rightarrow (E) \\
F &\rightarrow \texttt{ic}
\end{aligned}
$$

Decorated Parse Tree of $2 + 3 + 4$



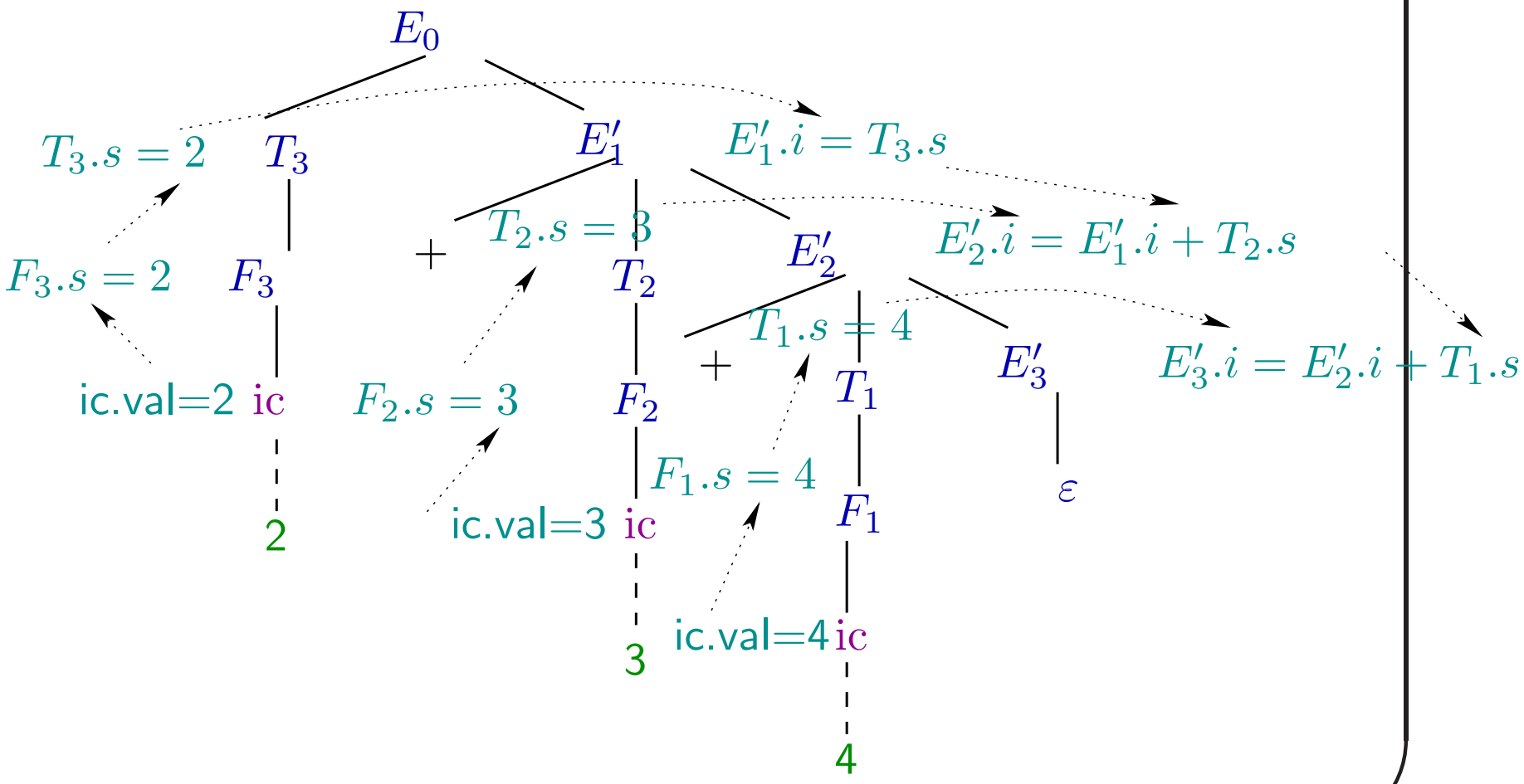
Parse Tree of $LL(1)$ Grammar



Note

- Two arguments of '+' are in different subtrees. It is necessary to pass the value of $T_3.s$ to the subtree of E'_1.
- It is also necessary for left-associativity of '+', to propagate the computed value down the tree say from E'_1 to E'_2.
- We achieve this by inherited attributes $E'.i$ and $T'.i$ of the non-terminals E' and T'.

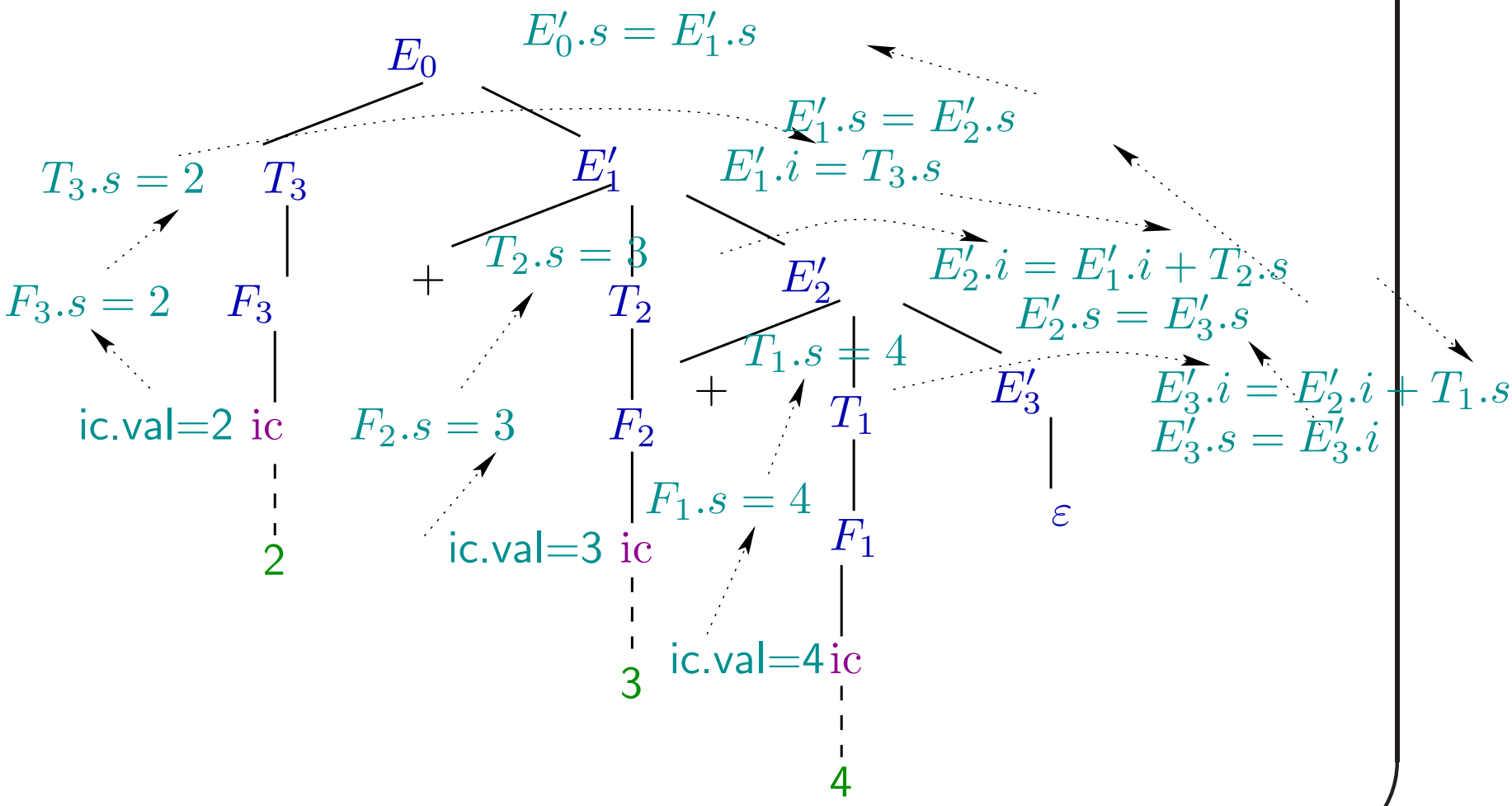
Parse Tree of $LL(1)$ Grammar



Note

But it is also necessary to propagate the computed value towards the root. This is done through the synthesized attributes of E' and T' - $E'.s, T'.s$.

Decorated Parse Tree of $LL(1)$ Grammar



L-Attributed $LL(1)$ Expression Grammar

$E \rightarrow T$ { E'.ival = T.sval } E'
{ E.sval = E'.sval }

$E' \rightarrow +T$ { E1'.ival = E'.ival + T.sval } E'_1
{ E'.sval = E1'.sval }

$E' \rightarrow \varepsilon$ { E'.sval = E'.ival }

L-Attributed $LL(1)$ Expression Grammar

$$
\begin{aligned}
T &\rightarrow F \{ \texttt{T'.ival = F.sval} \}\, T' \\
&\quad \{ \texttt{T.sval = T'.sval} \} \\
T' &\rightarrow *F \{ \texttt{T1'.ival = T'.ival * F.sval} \}\, T'_1 \\
&\quad \{ \texttt{T'.sval = T1'.sval} \} \\
T' &\rightarrow \varepsilon \{ \texttt{T'.sval = T'.ival} \}
\end{aligned}
$$

L-Attributed $LL(1)$ Expression Grammar

$$F \rightarrow (E) \{ \texttt{F.sval = E.sval} \}$$

$$F \rightarrow ic \{ \texttt{F.sval = ic.val} \}$$

$LL(1)$ Parsing Table

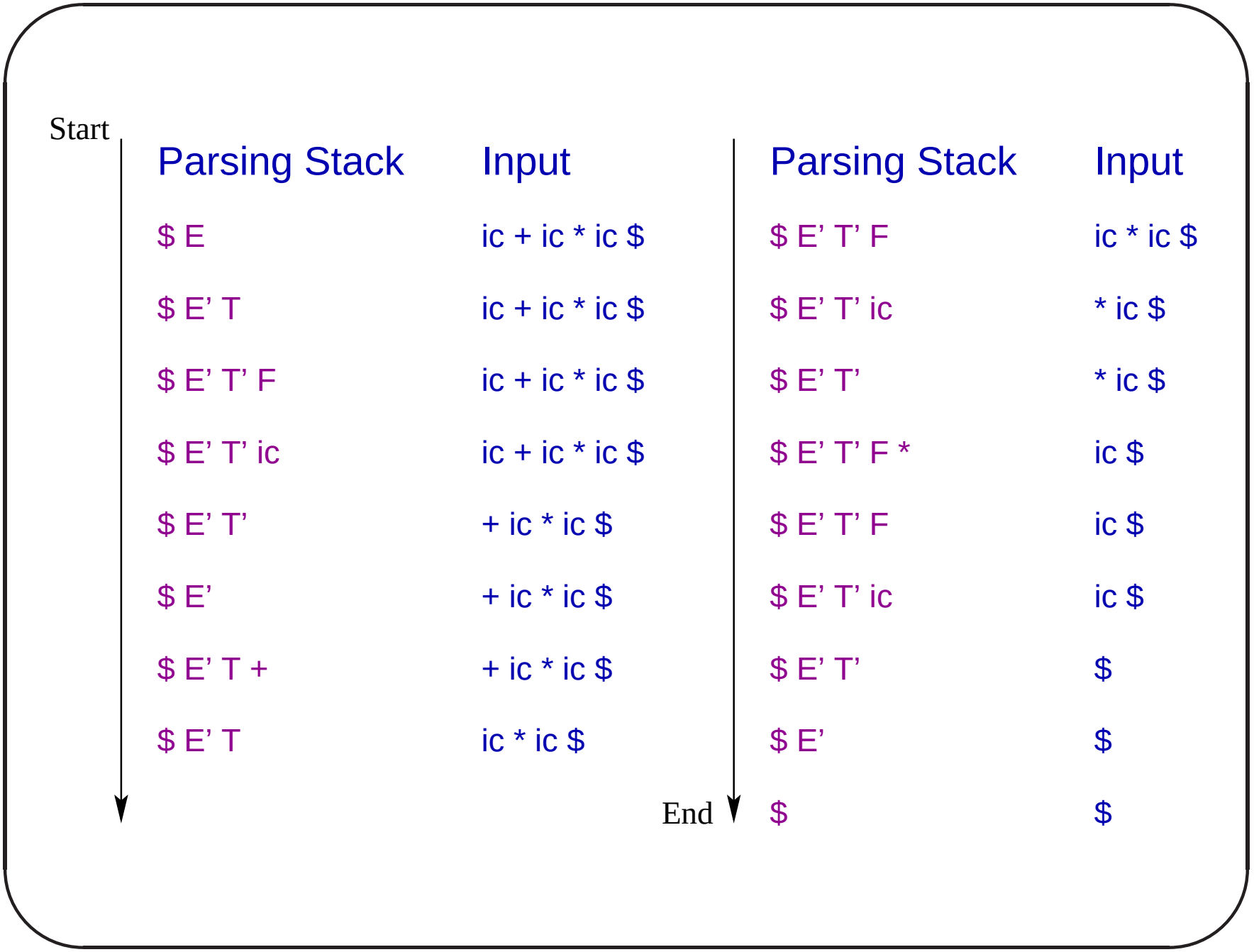
Non Term.	Terminal					
	$+$	$*$	$($	$)$	ic	\$
E			$E \to TE'$		$E \to TE'$	
T			$T \to FT'$		$T \to FT'$	
F			$F \to (E)$		$F \to$ ic	
E'	$E' \to +TE'$			$E' \to \varepsilon$		$E' \to \varepsilon$
T'	$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$		$T' \to \varepsilon$

Example

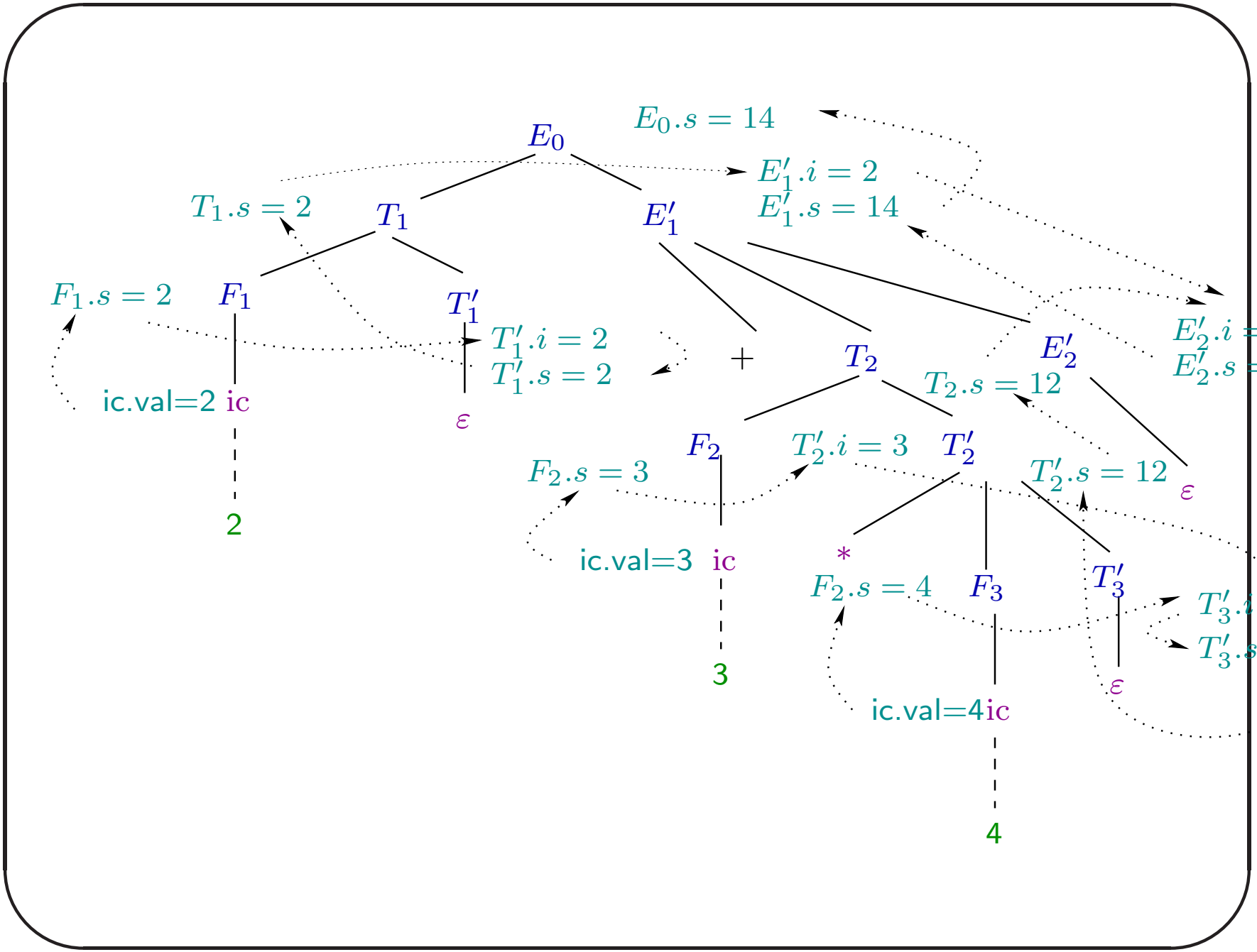
Consider the leftmost derivation and the parse tree decorated with attributes. corresponding to the input `2 + 3 * 4`.

Left-most Derivation

$$
\begin{array}{lcl}
1 & & E \\
2 & \rightarrow & T\,E' \\
3 & \rightarrow & F\,T'\,E' \\
4 & \rightarrow & ic\,T'\,E' \\
5 & \rightarrow & ic\,\varepsilon\,E' \\
6 & \rightarrow & ic\,+\,T\,E' \\
7 & \rightarrow & ic\,+\,F\,T'\,E' \\
8 & \rightarrow & ic\,+\,ic\,T'\,E' \\
9 & \rightarrow & ic\,+\,ic\,*\,F\,T'\,E'
\end{array}
\qquad
\begin{array}{lcl}
10 & \rightarrow & ic\,+\,ic\,*\,ic\,T'\,E' \\
11 & \rightarrow & ic\,+\,ic\,*\,ic\,\varepsilon\,E' \\
12 & \rightarrow & ic\,+\,ic\,*\,ic
\end{array}
$$


Start
Parsing Stack
Input
$ E
ic + ic * ic $
$ E' T
ic + ic * ic $
$ E' T' F
ic + ic * ic $
$ E' T' ic
ic + ic * ic $
$ E' T'
+ ic * ic $
$ E'
+ ic * ic $
$ E' T +
+ ic * ic $
$ E' T
ic * ic $
Parsing Stack
Input
$ E' T' F
ic * ic $
$ E' T' ic
* ic $
$ E' T'
* ic $
$ E' T' F *
ic $
$ E' T' F
ic $
$ E' T' ic
ic $
$ E' T'
$
$ E'
$
End
$
$

Decorated Parse Tree


E_0
$E_0.s = 14$
$E_1'.i = 2$
$E_1'.s = 14$
$T_1.s = 2$
T_1
E_1'
$F_1.s = 2$
F_1
T_1'
$T_1'.i = 2$
$T_1'.s = 2$
ic.val=2 ic
2
ε
+
T_2
E_2'
$E_2'.i = 14$
$E_2'.s = 14$
$T_2.s = 12$
F_2
$T_2'.i = 3$
T_2'
$F_2.s = 3$
$T_2'.s = 12$
ε
ic.val=3 ic
3
*
$F_2.s = 4$
F_3
T_3'
$T_3'.i = 12$
$T_3'.s = 12$
ε
ic.val=4ic
4

Implementation of L-Attributed Translation

The important question is how to implement L-attributed translation with the top-down parsing. Following is a general scheme.

- A simple parser stack holds records corresponding to terminals and non-terminals.

Implementation of L-Attributed Translation

- But to implement L-attributed translations, it should also hold records containing pointer to action (code), synthesize attributes and inherited attributes.
- For a non-terminal A, the inherited attributes are kept in the record corresponding to A itself.

Implementation of L-Attributed Translation

- The pointer to the code to evaluate the inherited attributes of A are naturally kept above the record of A.
- Synthesized attributes of A are kept below the record of A and its inherited attribute. This record may also contain pointer to some code to copy the synthesized attributes in records down the stack (below a fixed depth).

An Example

Consider the following rule of the L-attributed expression grammar:

$$E \rightarrow T \{ \texttt{E'.ival = T.sval} \} E'$$

$$\{ \texttt{E.sval = E'.sval} \}$$

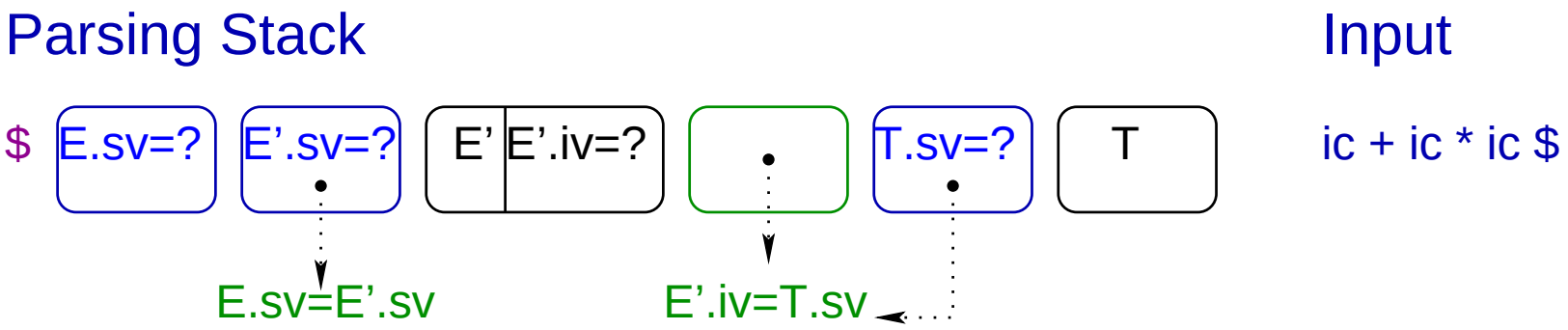
The stack contains following records before the E is expanded.

Parsing Stack with Attribute Records

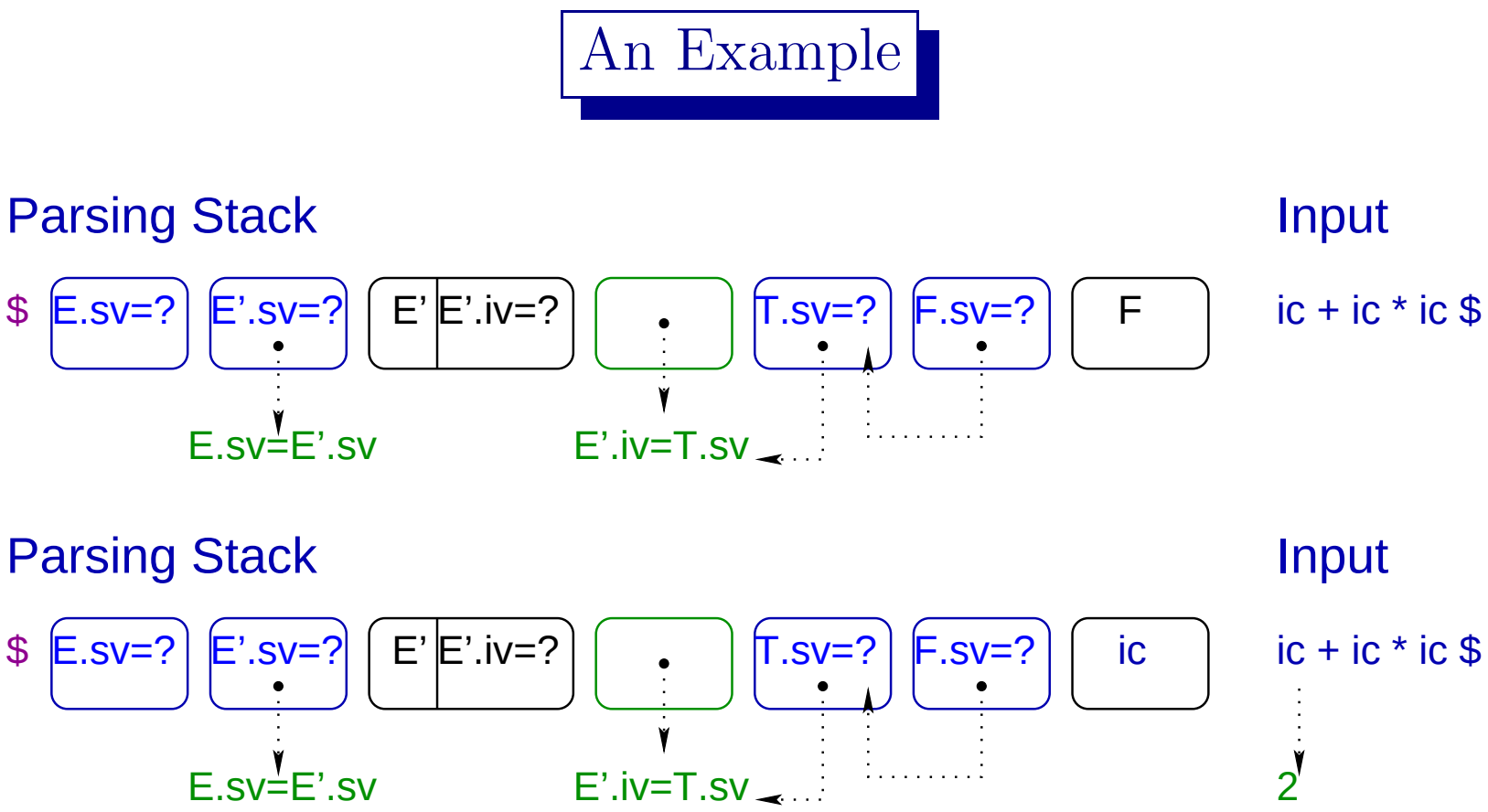
Parsing Stack		Input
$	E.sv=? E	ic + ic * ic $

An Example

After the replacement of the non-terminal E, the stack looks like:

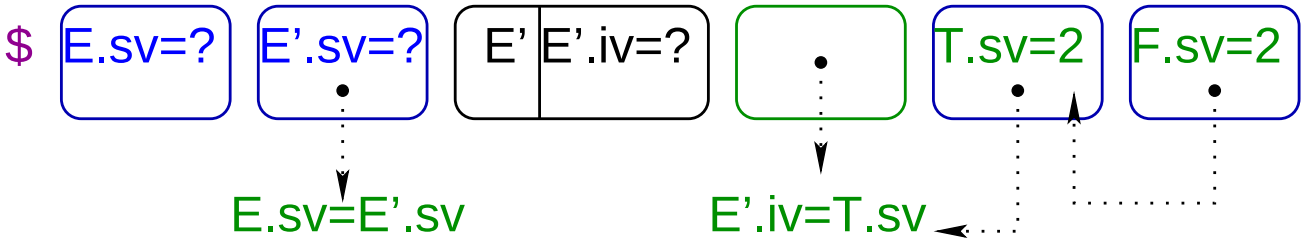


An Example


Parsing Stack
Input
$
E.sv=?
E'.sv=?
E'
E'.iv=?
T.sv=?
F.sv=?
F
ic + ic * ic $
E.sv=E'.sv
E'.iv=T.sv
Parsing Stack
Input
$
E.sv=?
E'.sv=?
E'
E'.iv=?
T.sv=?
F.sv=?
ic
ic + ic * ic $
E.sv=E'.sv
E'.iv=T.sv
2

An Example

Parsing Stack

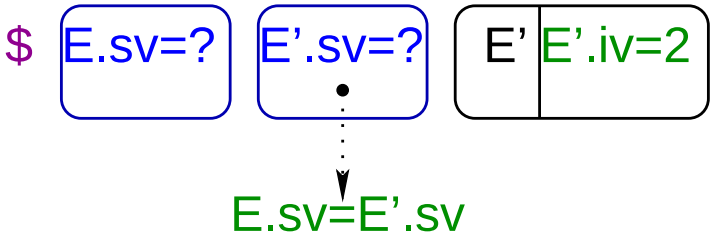


Input

+ ic * ic $

After three pops

Parsing Stack



Input

+ ic * ic $

Another Example

L-attributed grammars come naturally with flow-control statements. Following is an example with `if-then-else` statement.

$$\text{IS} \rightarrow \texttt{if}\ \text{BE}\ \texttt{then}\ S_1\ \texttt{else}\ S_2.$$

Attributes of Statement

- Every statement has a natural synthesized attribute, S.code, holding the code corresponding to S.
- Also a statement S has a continuation, the next instruction to be executed after execution of S. This may be handled as a jump target (label). But this label is an inherited attribute of S, S.next, propagated in the subtree of S.

Attributes of Boolean Expression

- The boolean expression also has a synthesized attribute BE.code.
- But it has two inherited attributes, BE.true, a jump target (label) where the control is transfered if the boolean expression is evaluated to true. This is the beginning of S_1.
  Similarly there is BE.false, a label at the beginning of S_2.

SDD for `if-then-else`

IS $\rightarrow$	`if` BE	l1=newLabel(), l2=newLabel()
	`then` S_1	BE.true = l1, BE.false=l2
	`else` S_2.	S_1.next = S_2.next = IS.next
		IS.code = BE.code + l1':' +
		S_1.code + l2':' + S_2.code

L-Attributed SDT for `if-then-else`

$$
\begin{aligned}
\text{IS} \rightarrow\ & \texttt{if} && \{\text{l1=newLabel(),l2=newLabel()} \\
& && \text{BE.true} = \text{l1},\ \text{BE.false=l2}\} \\
& \texttt{BE} && \{\text{S}_1\text{.next} = \text{IS.next}\ \} \\
& \texttt{then}\ S_1 && \{\text{S}_2\text{.next} = \text{IS.next}\ \} \\
& \texttt{else}\ S_2. && \\
& && \{\text{IS.code} = \text{BE.code} + \text{l1':'} + \\
& && \text{S}_1\text{.code} + \text{l2':'} + \text{S}_2\text{.code}\}
\end{aligned}
$$

Note

Afterward we shall see how this is managed in an actual implementation using back-patching.

SDD for Boolean Expression and

$BE \rightarrow BE_1$ and BE_2	BE_1.true=l=newLabel()
	BE_1.false = BE.false
	BE_2.true = BE.true
	BE_2.false = BE.false
	BE.code = BE_1.code +
	l':' + BE_2.code

L-Attributed SDT for Boolean Expression `and`

$$
\begin{aligned}
BE \rightarrow\ & \{\ BE_1.true = l = newLabel() \\
& BE_1.false = BE.false\ \} \\
& BE_1\ \texttt{and} \\
& \{\ BE_2.true = BE.true \\
& BE_2.false = BE.false\ \} \\
& BE_2 \\
& \{\ BE.code = BE_1.code + \\
& l\text{':'} + BE_2.code\ \}
\end{aligned}
$$

Note

Not all definitions can be implemented during parsing. Consider the following definition to convert infix-expression to prefix expression.

Rules

$1: \quad E \rightarrow$ `{ putchar('+'); }` $E_1 + T$

$2: \quad E \rightarrow T$

$3: \quad T \rightarrow$ `{ putchar('*'); }` $T_1 * F$

$4: \quad T \rightarrow F$

$5: \quad F \rightarrow (E)$

$6: \quad F \rightarrow ic$ `{printf(" \%d ", ic.val);}`

Note

It is not possible to activate any one of the `putchar()` operations while parsing without seeing '`+`' or '`*`'. But if the whole parse tree is available[a] New leaf nodes corresponding to actions can be added to internal nodes. Finally a preorder traversal in the modified tree will do the job.

[a] The whole parse tree is available only after the parsing is complete.