Syntax Directed Attribute Synthesis

Context-Free Processing

- The scanner and Parser together performs the context-free analysis of the program text.
- The scanner supplies the token and its attributes.
- The parser forms the parse tree or abstract syntax tree.
- But this analysis does not go beyond the local and nested structure of the grammar.

Non-Context-Free Structural Features

- Context-dependent language constraints cannot be checked during pure context-free analysis.
- Typical examples are deceleration and initialization of variables before use.
- Correspondence between the formal and actual parameters of functions etc..

Translation

- Our main goal is the translation of source language to a target language.
- This requires collection and synthesis of attributes over the entire syntax tree.
- Attribute synthesis is often done hand-in-hand with parsing.

Translation

If the complete parse tree is available and the dependence of attributes of different non-terminals are known. The parse tree can be traversed to compute the attributes of nonterminals[a] and necessary semantic actions can be performed.

[a]Dependence should not form a cycle.

Translation

- But often Computation rules can be associated with the production rules to perform semantic actions along with parsing.
- Computed information is stored as attributes of non-terminals in some data structure e.g. symbol table.

Example

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

We consider three different translations:

- implementation of a simple calculator,
- conversion of an infix expression to a postfix expression,
- Semantically equivalent intermediate Code generation for this rule.

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

Note

- In first two cases the back-end is like an interpreter.
- In the third case it translates the high-level code to an intermediate code of a virtual machine.

Example: Calculator

- The attributes of E and T are the values of the expression corresponding to the sub-tree of E and T.
- Let the name of the attribute be val.
- The semantic action associated with the production rule is,

  $E \rightarrow E_1 + T$ $\{E{\cdot}\mathrm{val} = E_1{\cdot}\mathrm{val} + T{\cdot}\mathrm{val}\}$[a].

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

Note

- The action takes place when $E_1 + T$ is reduced to E. The value is computed from the attributes of E_1 and T, and is saved as the attribute of E.
- Alternatively, evaluation may take place during the postorder traversal of the syntax tree.
- There is no other side-effect of the semantic action. It is local.

Note

- An obvious question is where do we store the attributes of the non-terminals?
- A non-terminal and its attributes may from a structure or record.

Note

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

Example: Infix to Postfix Conversion

- Here the problem is to convert an infix arithmetic expression to an equivalent postfix expression.
- Both the input and output are strings of characters, postfix expressions.
- Let the attribute `exp` be associated with non-terminals E and T. Its type is `char *`.

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); // you may or may not
}
```

Again there is no side-effect

Example: Code Generation

- The main difference of translation for code generation with two previous translations is that often no data value corresponding to E_1 or T is available during compilation.
- The attributes E_1 and T are two sequences of translated codes. They will compute values of expressions corresponding to E_1 and T during a program execution.

Example: Code Generation

- The translation for to the rule $E \rightarrow E_1 + T$ generates code so that at run time the computed values of E_1 and T are added to generate and store the value of expression E.
- The computed values of the expressions E_1 and T are possibly stored in compiler defined temporary variables/virtual registers.

Example: Code Generation

- The compiler creates temporary variables where the intermediate values of sub-expressions are stored. These variable names may also be entered in the symbol table.
- The attributes of a non-terminal like E or T may be a code sequence and information about the corresponding temporary variable.

Example: Code Generation

The code corresponding to $E \rightarrow E_1 + T$ may look like,

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

where assignPlus means
`E.loc = E1.loc + T.loc`.

Note

- This action has side-effects, it may make an entry of the new location in the symbol table. And the generated code is added in a data structure of code sequence.
- As an alternative E and T may store their code sequences as their second attribute.

Associating Information with CFG

- A context-free grammar can be extended by associating two features with it: data and computation.
- Data is associated to a syntactic category by attaching attributes to the non-terminals.
- Computation is associated with the production rules.

Syntax Directed Definition

- Initial attribute values are supplied by the scanner.
- A context-free grammar augmented with attributes and rules for computing the attributes, is a syntax directed definition of semantics. It is called an attribute grammar.
- There should not be any circularity in the definition of attributes.

Syntax Directed Translation

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

Example

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

Action$_1$: takes place before parsing of the input corresponding to the non-terminal B.
Action$_2$: takes place after consuming the input for B, but before consuming the input for C.
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 a 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 $S \to A \mid B,\ A \to aba, B \to abb$ is LALR.
- An embedded action is introduced as shown, $S \to A \mid B,\ A \to a\ \{\text{action}\}\ ba, B \to abb.$

Note

- Bison modifies the grammar to
  $S \rightarrow A|B, A \rightarrow aMba, B \rightarrow abb,$
  $M \rightarrow \varepsilon$ {action} .

- The modified grammar is no longer LALR. The state
  $\{A \rightarrow a \bullet Mba,\ \$,\ B \rightarrow a \bullet bb,\ \$, M \rightarrow \bullet,\ b\}$
  has a shift-reduct conflict.

Attribute Computation: a General Approach

- Construct the parse tree. Compute the attributes of the non-terminals following the data-flow in attribute dependence graph. But construction of complete parse tree is costly.
- There are restricted SDDs that do not require explicit construction of parse tree. They are S-attributed and L-attributed definitions.

A Simple Example

Consider the following grammar of signed binary numerals, a character string of 1's and 0's. We wish to translate it to integer.

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

Note

- We first construct the $LR(0)$ automaton of the grammar and find that the grammar is SLR.
- We associate attributes to the non-terminals.
- We also associate SDDs to the production rules.

$LR(0)$ Automaton

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

$LR(0)$ Automaton

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

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_2	r_2					
4			r_3	r_3					
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`

SDD

$0:$	S'	$\rightarrow$	N	print N.val
$1:$	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 $\rightarrow$	Input$^{\rightarrow}$	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`

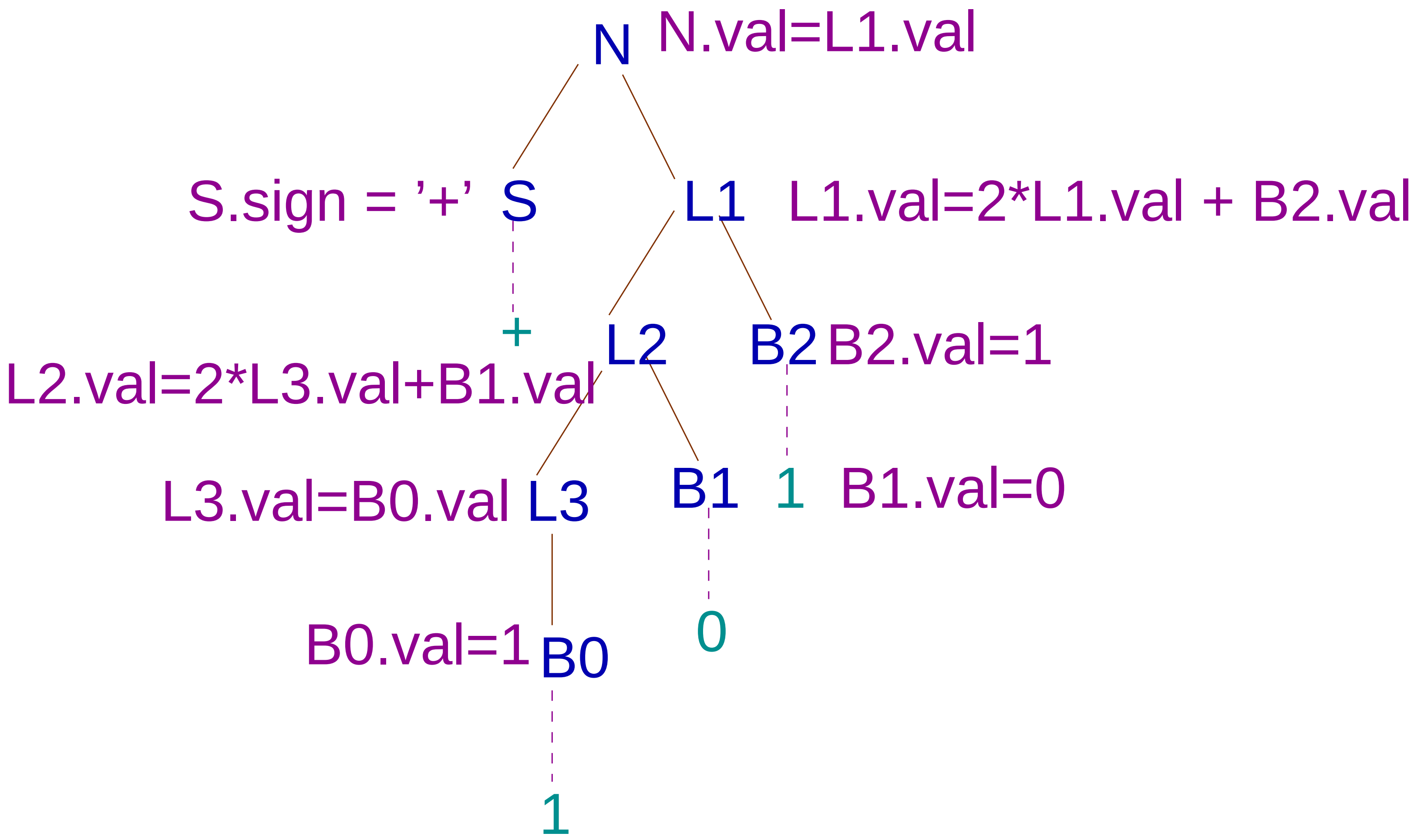
Example: Parsing & Value Stack

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.

Another Set of Attributes

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

SDD

$0:$	S'	$\rightarrow$	N	print N.val
$1:$	N	$\rightarrow$	SL	`L.pos = 0`
				`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	$\texttt{L}_1\texttt{.pos = L.pos+1;}$
				`B.pos = L.pos;`
				$\texttt{L.val = L}_1\texttt{.val+B.val;}$

SDD

$5:$	L	$\rightarrow$	B	`B.pos = L.pos;`
				`L.val = B.val;`
$6:$	B	$\rightarrow$	0	`B.val = 0;`
$7:$	B	$\rightarrow$	1	B.val = $2^{\texttt{B.pos}}$;

Exercise

Draw the parse tree for -101 and show the flow of information.

Note

- Attributes of a non-terminal depends on the nature of translation. But it may also depend on the nature of the grammar.
- Following is a grammar of integers in 2's complement numerals. It is to be translated to a signed decimal numeral.

Exercise

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

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

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

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

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

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

Example

Consider a right-recursive grammar of signed binary strings:

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

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

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

Actions for Production Rules

$5: \; L \; \rightarrow \; B$ `L.val = B.val; L.pos = 0`

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

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

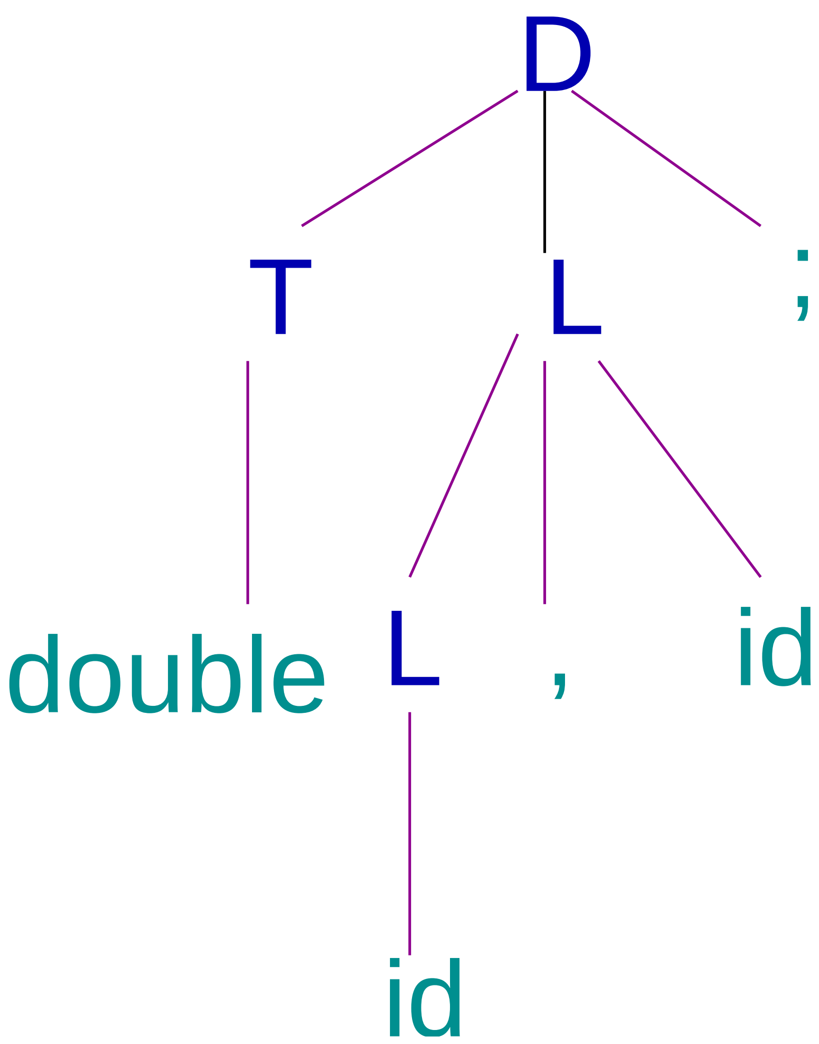
Example

Consider the following grammar for variable declaration:

$$
\begin{aligned}
1:\quad & D && \rightarrow && T\ L\ ; \\
2:\quad & T && \rightarrow && \texttt{int} \\
3:\quad & T && \rightarrow && \texttt{double} \\
4:\quad & L && \rightarrow && L\ ,\ \texttt{id} \\
5:\quad & L && \rightarrow && \texttt{id}
\end{aligned}
$$

Parse Tree

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


D
T
L
;
double
L
,
id
id

Note

- A scanner/lexical analyzer returns the token corresponding to an identifier.
- If it returns the identifier name as an attribute. The parser takes the necessary action to search and enter it in the symbol table.

Note

- Otherwise the scanner may search and make an entry in the symbol table if it is new. It sends the pointer to the symbol table as an attribute to the parser.
- Then parser reduces the id to the non-terminal L. It is also necessary to update the type information[a] of the identifier in the symbol table.

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

Note

But the type information is not available from any subtree rooted at L. It has to be inherited from T via the root D.

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 of a parse tree node.
- An inherited attribute $B.i$ is defined in terms of the attributes of the parent and sibling nodes of B.
- In the previous example the non-terminal L gets the attribute from T as an inherited attribute.

Synthesized Attribute

- The synthesized attribute $B.s$ of a non-terminal B is defined by the attributes of its children.
- The attribute of a leaf-node comes from the scanner.

S-Attributed Definitions

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

This definition can be implemented in a LR-parser during a reduction as the traversal on the parse-tree is 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.
Let $A \rightarrow \alpha_1\alpha_2 \cdots \alpha_n$ be a production rule, and α_k has an inherited attribute 'a'.

L-Attributed Definition

The value of $\alpha_k.a$ is computed using

- the inherited attribute of A (parent),
- the inherited or synthesized attributes 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 directly as this non-terminal is not present in the stack.

Solution I

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

$LR(0)$ Automaton

q_7 :	$L \rightarrow L, \bullet\mathtt{id}$
q_8 :	$L \rightarrow L, \mathtt{id}\bullet$
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`.

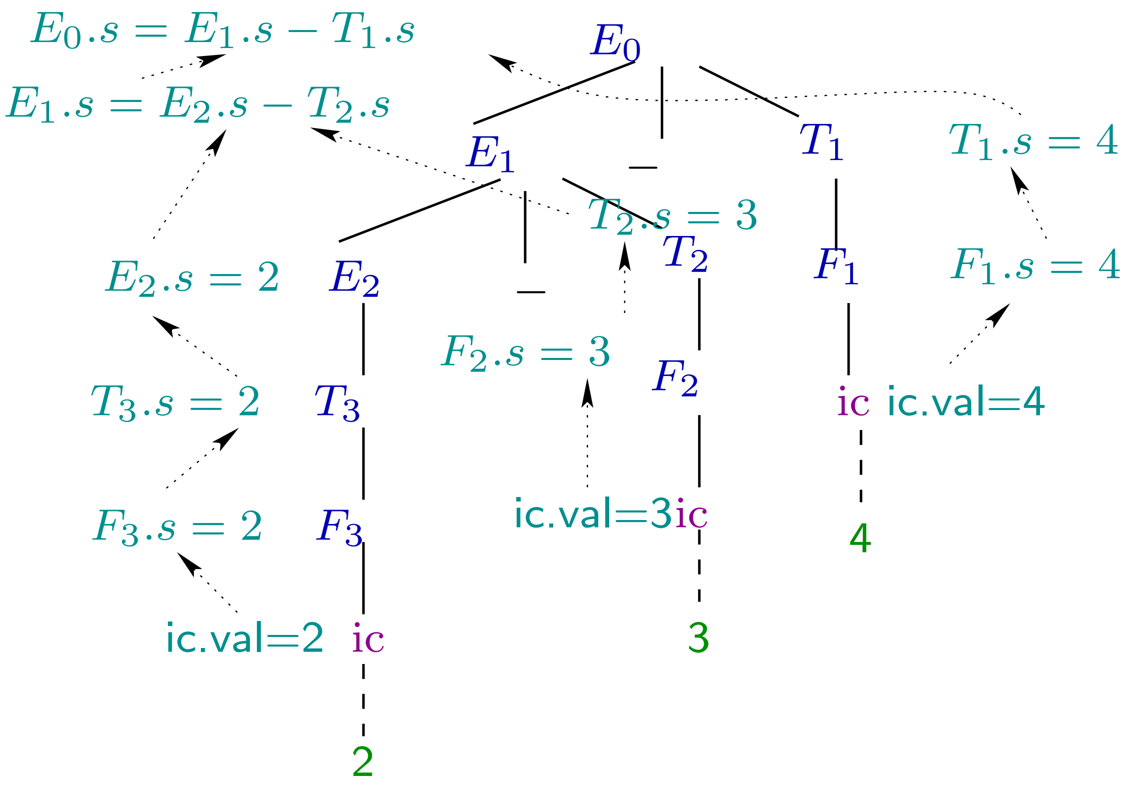
Note

- Often a natural grammar is transformed to make it suitable for parsing.
- But the new parse tree no longer match with the abstract syntax tree of the language.
- As an example the left-recursion is removed from the grammar for $LL(1)$ parsing.
- The original S-attributed grammar gets modified after this transformation.

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}`

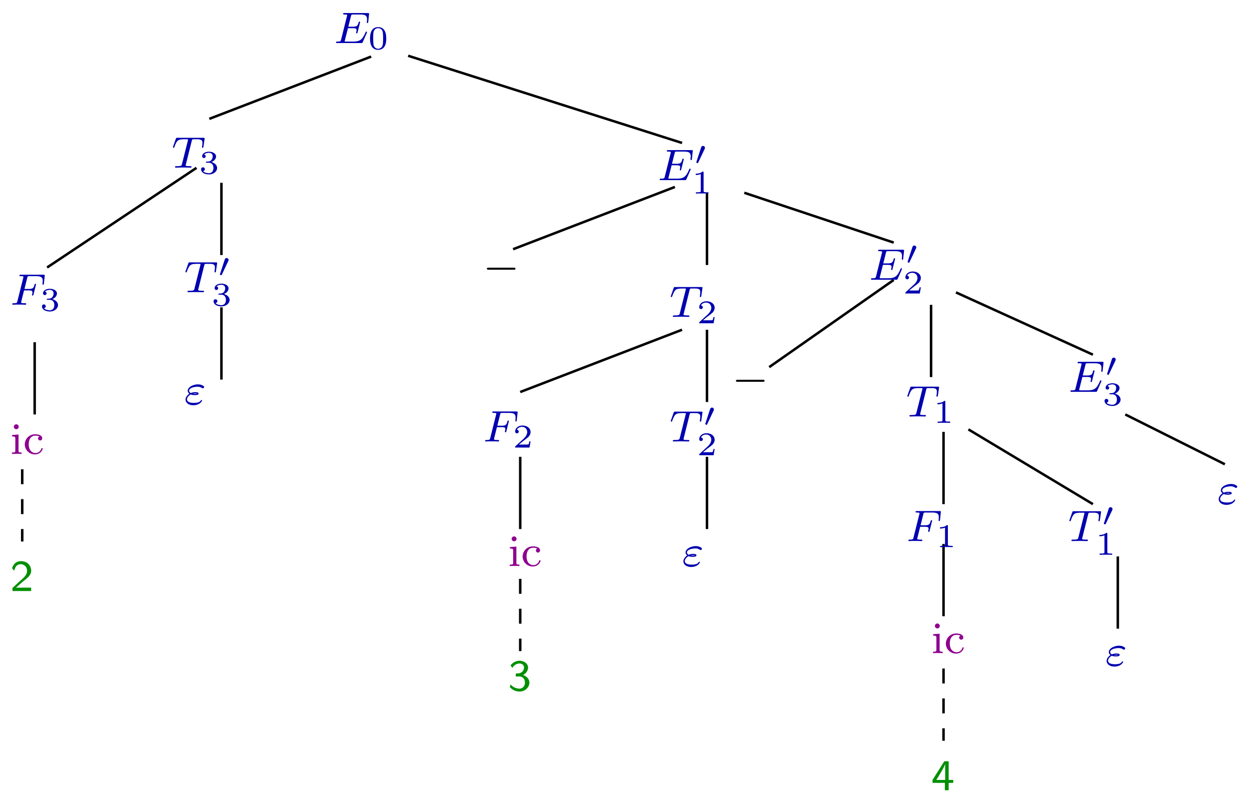
Decorated Parse Tree of $2 - 3 - 4$



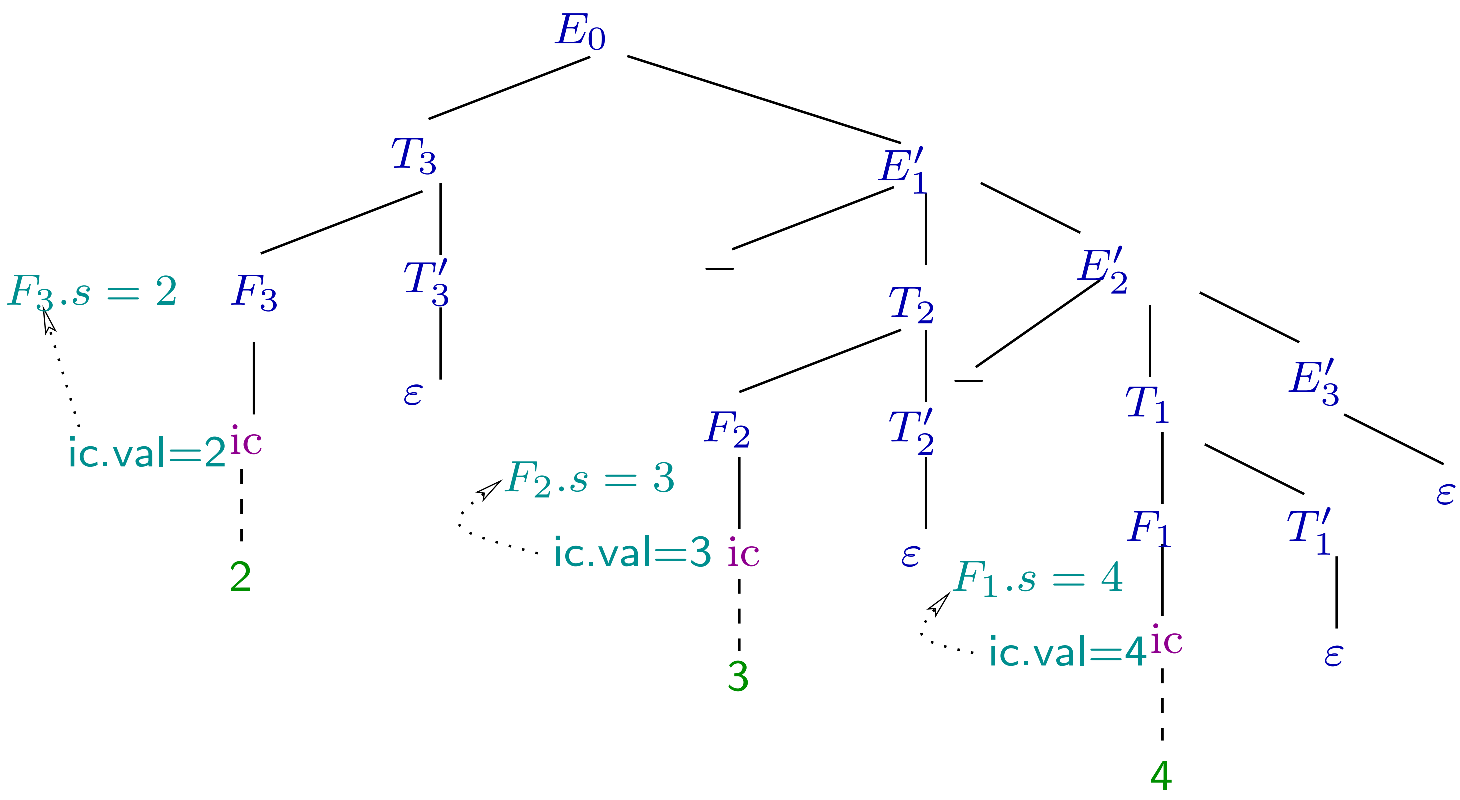
Equivalent $LL(1)$ Grammar

$$
\begin{array}{rcl}
S & \rightarrow & E\$ \\
E & \rightarrow & TE' \\
E' & \rightarrow & -TE' \\
E' & \rightarrow & \varepsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & /FT' \\
T' & \rightarrow & \varepsilon \\
F & \rightarrow & (E) \\
F & \rightarrow & \mathtt{ic}
\end{array}
$$

Parse Tree of $LL(1)$ Grammar



Partially Decorated 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'.

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' i.e. $E'.s, T'.s$.

L-Attributed $LL(1)$ Expression Grammar

$$
\begin{array}{lll}
E & \rightarrow & T \; \{ \; \texttt{E'.ival = T.sval} \; \} \; E' \\
 & & \{ \; \texttt{E.sval = E'.sval} \; \} \\
E' & \rightarrow & -T \; \{ \; \texttt{E1'.ival = E'.ival - T.sval} \; \} \; E'_1 \\
 & & \{ \; \texttt{E'.sval = E1'.sval} \; \} \\
E' & \rightarrow & \varepsilon \; \{ \; \texttt{E'.sval = E'.ival} \; \}
\end{array}
$$

L-Attributed $LL(1)$ Expression Grammar

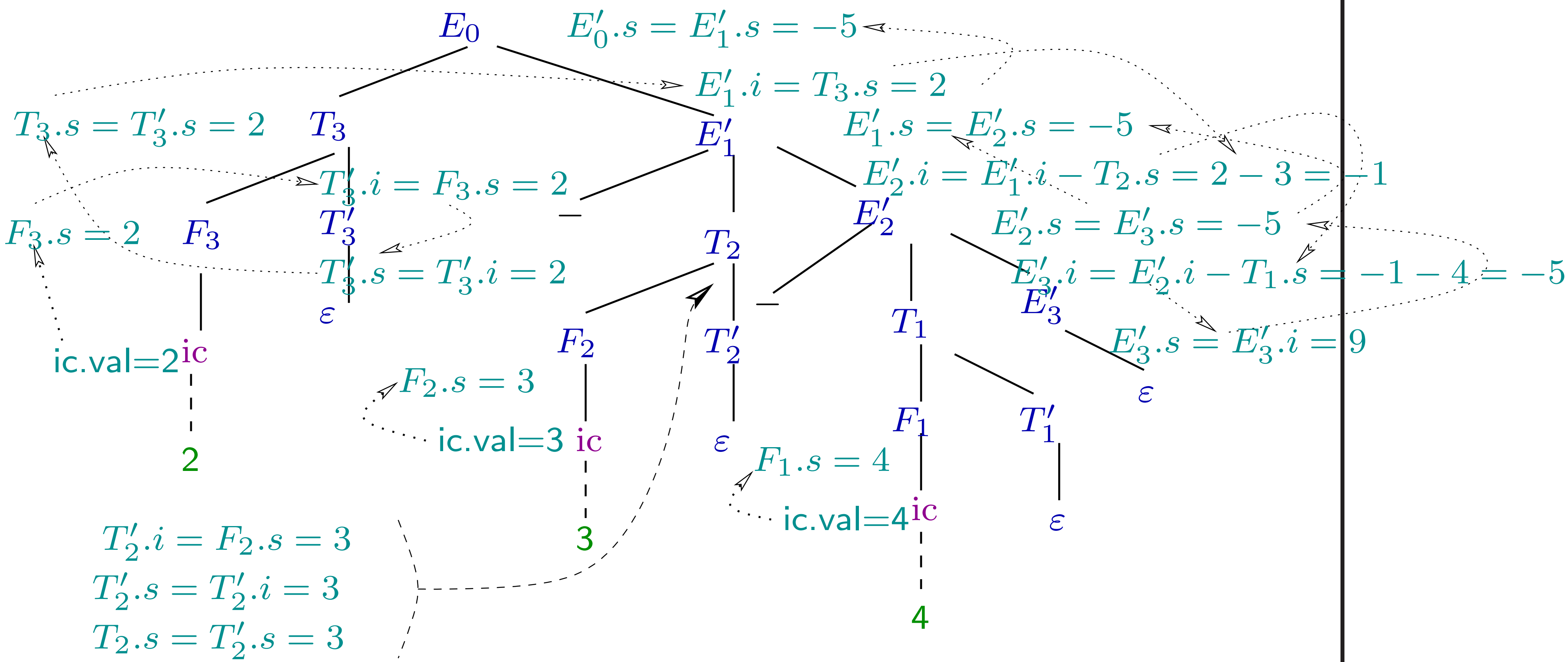
$$
\begin{array}{lll}
T & \rightarrow & F \{ \texttt{ T'.ival = F.sval } \} T' \\
 & & \{ \texttt{ T.sval = T'.sval } \} \\
T' & \rightarrow & /F \{ \texttt{ T1'.ival = T'.ival / F.sval } \} T'_1 \\
 & & \{ \texttt{ T'.sval = T1'.sval } \} \\
T' & \rightarrow & \varepsilon \{ \texttt{ T'.sval = T'.ival } \}
\end{array}
$$

L-Attributed $LL(1)$ Expression Grammar

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

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

Decorated Parse Tree of $LL(1)$ Grammar



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 transferred 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.	$\mathrm{S_1}$.next = $\mathrm{S_2}$.next = IS.next
	IS.code = BE.code + l1':' +
	$\mathrm{S_1}$.code + l2':' + $\mathrm{S_2}$.code

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

IS → `if` {l1=newLabel(),l2=newLabel()
BE.true = l1, BE.false=l2}
`BE` {S_1.next = IS.next }
`then` S_1 {S_2.next = IS.next }
`else` S_2.
{IS.code = BE.code + l1':' +
S_1.code + l2':' + S_2.code}

Note

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

SDD for Boolean Expression and

$\text{BE} \rightarrow \text{BE}_1$ 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`

BE $\rightarrow$ { BE_1.true=l=newLabel()

BE_1.false = BE.false }

BE_1 `and`

{ BE_2.true = BE.true

BE_2.false = BE.false }

BE_2

{ BE.code = BE_1.code +

l':' + BE_2.code }