Attribute Grammars

Attribute Grammars were invented by Don Knuth as a way to unify all of the stages of compiling into one. They give a formal way to pass semantic information (types, values, etc.) around a parse tree.

We now allow any grammar symbol X to have attributes. The attribute a of symbol X is denoted X.a

If there is a grammar rule

$$P:\ X_0 ::= X_1X_2\ldots X_k$$

then a *semantic rule* for P computes the value of some attribute of one of the X_i in terms of other attributes of symbols in the rule.

If you think of the rule as forming the node of a tree, an attribute of a node gets its value from the attribute of its parent, siblings and children (but not from its grandparent, for example).

A *syntax-directed definition* (SDD) is a triple (G,A,R) where G is a context-free grammar, A is a set of attributes, and R is the set of semantic rules for G.

Example: Grammar symbols E, T, and F all have one attribute *val*. Where necessary we put subscripts on the grammar symbols to distinguish the child from the parent.

$E ::= E_1 + T \; \{E.val = E_1.val + T.val\}$

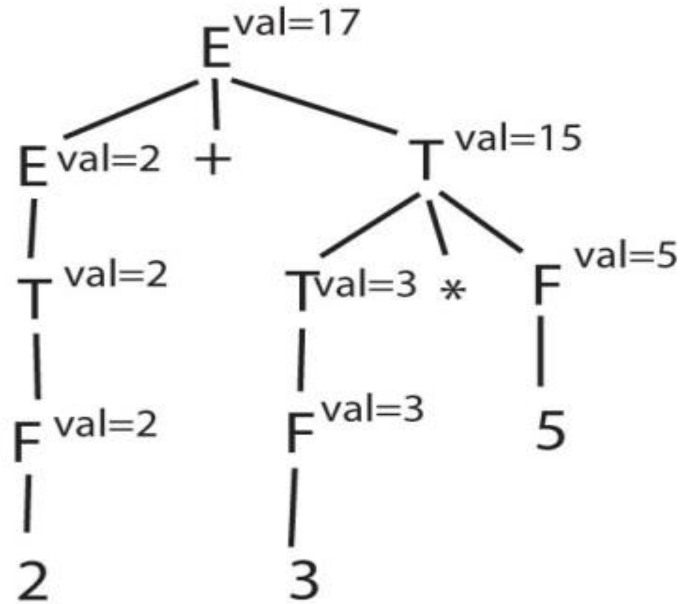
$E ::= T \; \{E.val = T.val\}$

$T ::= T_1 * F \; \{T.val = T_1.val * F.val\}$

$T ::= F \; \{T.val = F.val\}$

$F ::= num \; \{F.val = num\}$

With this grammar the expression 2+3*5 parses to



Note that the attributes implement the natural semantics of this simple language.

We say that an attribute X.a is *synthesized* if there is a grammar rule X ::= α and X.a is defined in terms of the attributes of the elements of α. We say that X.a is *inherited* if there is a rule Y ::= $\alpha X \beta$ and X.a is defined in terms of the attributes of Y, α, and β.

In other words, synthesized attributes get their values from their children while inherited attributes get their values from their parent and siblings

In the E ::= E+T example a from a few slides ago the attributes are all synthesized -- passed from the leaves up; evaluation of such attributes can be done easily in a bottom-up pass through the tree.

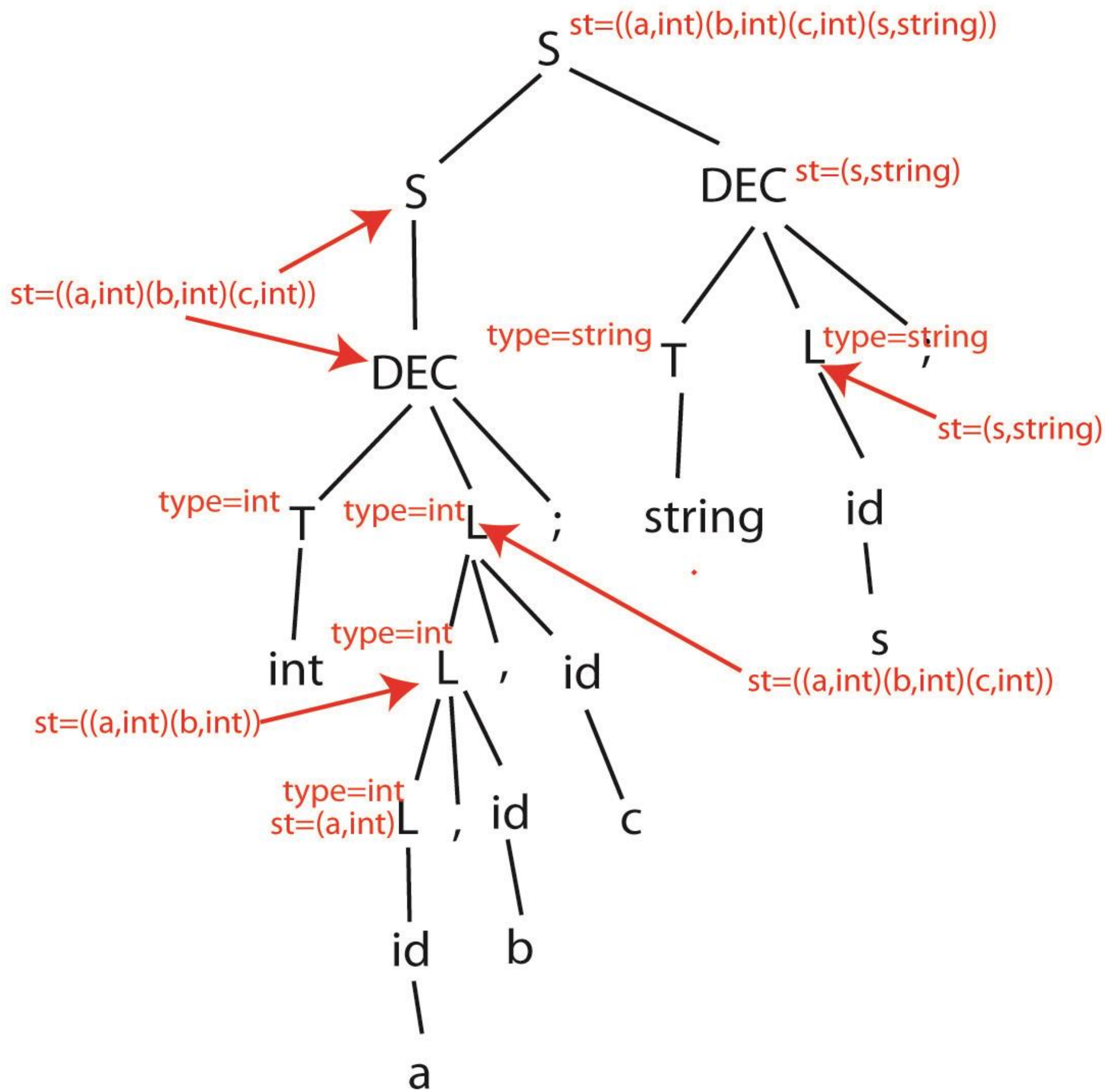
Here is an example that uses attributes for automatic type evaluation. The st attribute is a symbol table -- a list of (id,type) pairs.

S ::= DEC {S.st = DEC.st}
S ::= S_1 DEC {S.st = S_1.st || DEC.st} (||=concatenate)
DEC ::= T L ; {L.type = T.type; DEC.st = L.st}
T ::= int {T.type = int}
T ::= string {T.type = string}
L ::= id {L.st = (id.name, L.type)}
L ::= L_1, id {L1.type = L.type; L.st = L_1.st || (id.name, L.type)}

Note that L.type is inherited, but the st attribute is synthesized.

Here is the attributed tree this grammar generates for

int a, b, c;
string s;



A grammar is *L-attributed* if each attribute defined in a rule $A ::= X_1 \ldots X_k$ is either

a) Synthesized (i.e., an attribute of A)
b) An inherited attribute of some Xi that depends only on the inherited attributes of A and the attributes of X_j for $j < i$

We can evaluate the attributes in an L-attributed grammar in a bottom-up, left-to-right pass using the following invariant:

> When we get to a node during parsing, we must have all of the information we need to evaluate its inherited attributes. Before we leave the node we must have all of the information we need to evaluate its synthesized attributes.

Here is an example that evaluates fractional binary strings, such as .101 (which is 1/2+1/8, or 5/8)

$N ::= .L \{N.v = L.v;\ L.c = -1\}$

$L ::= B\ L_1 \{L.v=B.v+L_1.v;\ L_1.c=L.c-1;\ B.c=L.c\}$

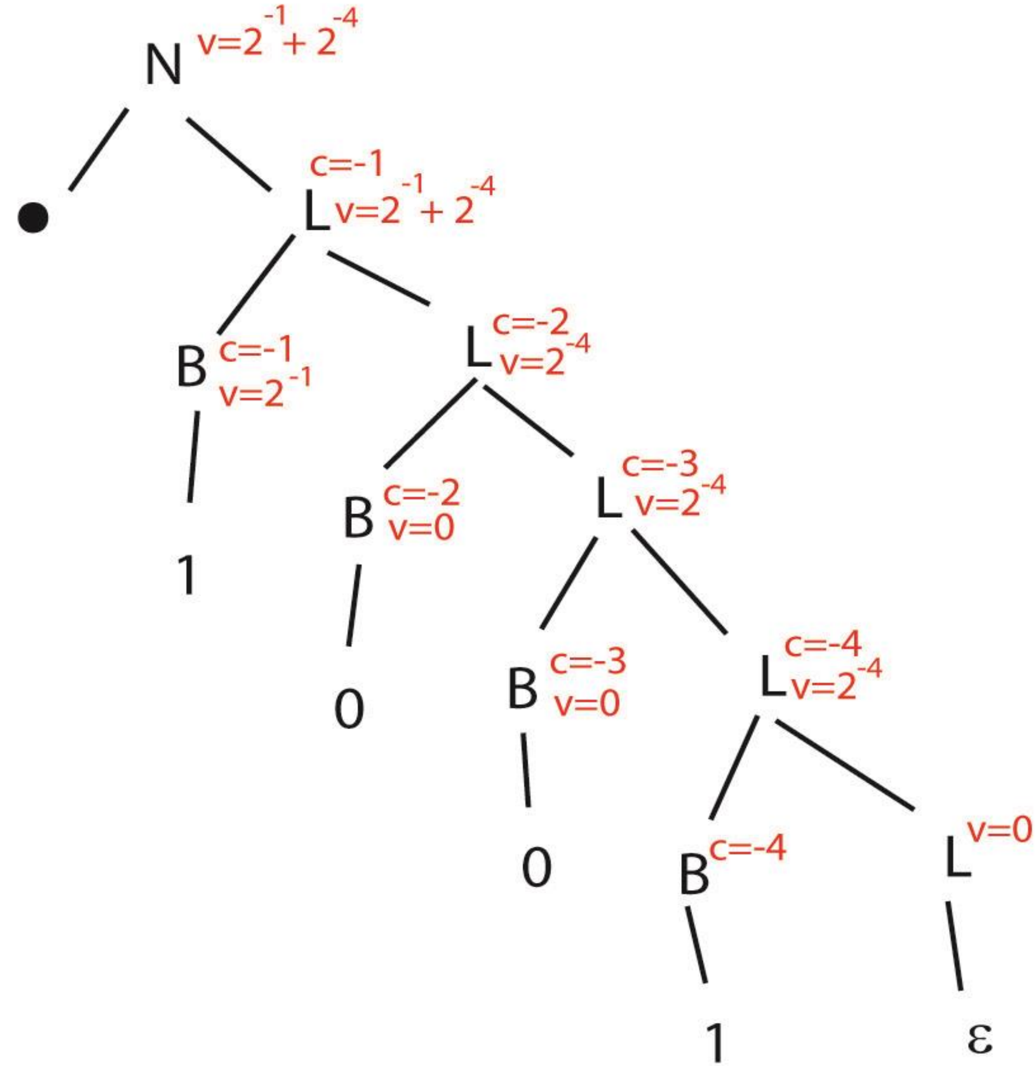
$L ::= \varepsilon \{L.v=0\}$

$B ::= 0 \{B.v=0\}$

$B ::= 1 \{B.v=2^{B.c}\}$

Note that L.c is inherited, while L.v is synthesized.

Here we use this grammar to parse and evaluate the string .1001



To write a recursive descent parser for an L-attributed grammar apply the following pattern.

For rule A ::= $X_1 \ldots X_k$ the function A() that parses this rule should have as arguments all of the inherited attributes for A; before it returns it should evaluate all of the synthesized attributes of A.

A *translation scheme* has the same information as an L-attributed grammar but provides an ordering for the parsing and attribute evaluation.

For example, the previous grammar could be written

$N ::= .\ \{L.c = -1\}\ L\ \{N.v = L.v\}$

$L ::= \{B.c = L.c\}\ B\ \{L_1.c=L.c-1\}\ L_1\ \{L.v=B.v+L_1.\ v\}$

$L ::= \varepsilon\ \{L.v=0\}$

$B ::= 0\ \{B.v=0\}$

$B ::= 1\ \{B.v=2^{B.c}\}$

Here is a more realistic example of attribute grammars. This produces "assembly code" for an if-then-else statement.

Starting grammar:

S ::= if (E) S | if (E) S else S | <other stuff>

We will use 3 "assembly language" instructions:

JMPF label	conditional branch
JMP label	unconditional branch
LABEL lab	place a label

We want to produce something like this:

if (e) s	if (e) s_1 else s_2
--------------	----------------------
code for e	code for e
JMPF L1	JMPF L1
code for s	code for s_1
LABEL L1	JMP L2
	LABEL L1
	code for s_2
	LABEL L2

First, left-factor the grammar so we can parse it:

S ::= if (E) S TAIL

S ::= <other stuff>

TAIL ::= else S

TAIL ::= ε

We will give S two inherited attributes:

S.temp and S.label

We give TAIL one synthesized attribute:

TAIL.label

Here is the translation scheme:

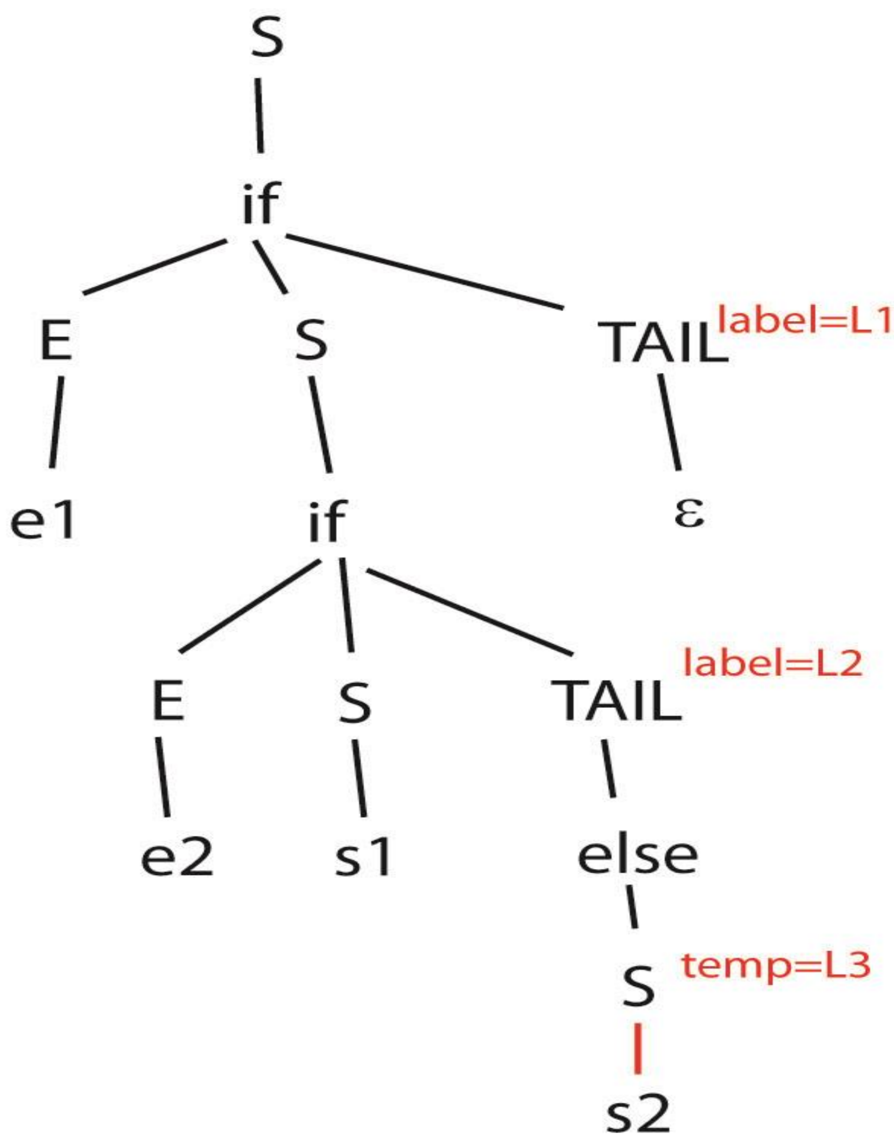
```
S ::= if (E) {TAIL.label = new label();
              emit( "JMPF", TAIL.label)} S1 TAIL
S ::= <other stuff>
TAIL ::=  else {S.temp=new label();
               emit( "JMP", s.temp);
               emit( "LABEL", TAIL.label); }
               S {emit("LABEL", s.temp);}
TAIL ::= ε {emit( "LABEL", TAIL.label);}
```

The expression

```
if (e1) {
    if (e2)
        s1
    else
        s2
}
```

generates the following:

```
code for e1
JMPF L1
code for e2
JMPF L2
code for s1
JMP L3
LABEL L2
code for s2
LABEL L3
LABEL L1
```



The expression

```
if (e1) {
    if (e2)
        s1
}
else
    s2
```

generates the following:

```
code for e1
JMPF L1
code for e2
JMPF L2
code for s1
LABEL L2
JMP L3
LABEL L1
code for s2
LABEL L3
```