LECTURE 10

Semantic Analysis

REVIEW

So far, we've covered the following:

- Compilation methods: compilation vs. interpretation.
- The overall compilation process.
  - Front-end analysis
    - Lexical Analysis
    - Syntax Analysis
    - Semantic Analysis
  - Back-end synthesis

REVIEW

- Lexical Analysis
  - Specify valid tokens with regular expressions.
  - Convert regular expression to a scanner which can recognize the valid tokens.
    - Hand-written or table-driven.
  - Use of automatic scanner generators like Lex.
- Syntax Analysis
  - Specify valid patterns of tokens with context-free grammars.
  - Convert context-free grammar to a parser which can recognize valid strings in the language.
    - Hand-written or table-driven.
  - Next lecture: Use of automatic parser generators like Yacc.

SEMANTIC ANALYSIS

Now that we've covered the process of validating the *structure* of a program, now we can discuss the process of validating its *meaning*.

In general, semantic analysis verifies the portion of the language definition which cannot feasibly be described by a context-free grammar. This includes meaning but also more complicated syntax checks. For example,

- Matching the number of arguments of a subroutine to its number of formal parameters.
  Not nested → cannot be counted by context-free grammars.
- Making sure every function has a return statement. Too complex to be feasibly represented by grammar.

SEMANTIC RULES

We can divide semantic rules into two categories.

- Static semantic rules.
  - Enforced at compile-time.
  - Implemented in semantic analysis phase of the compiler.
  - Examples include:
    - Type checking.
    - Identifiers are used in appropriate context.
    - Check subroutine call arguments.
    - Check labels.

SEMANTIC RULES

- Dynamic semantic rules.
  - Code is generated to enforce at run-time.
  - Examples include:
    - Array subscript values are within bounds.
    - Arithmetic errors.
    - Pointers are not dereferenced unless pointing to valid object.
    - A variable is used but hasn't been initialized.
  - Some languages allow programmers to add explicit dynamic semantic checks in the form of assertions.

```
assert denominator != 0;
```

  - When a check fails at run time, an exception is raised.

SEMANTIC CHECKS

Most compilers perform some amount of static analysis at compile-time and generate code to check the rest dynamically.

Dynamic checks can be costly at run-time, but they allow for greater flexibility in the language.

Some languages take a conservative approach by tightening semantic rules so that dynamic checks are minimized.

ATTRIBUTE GRAMMARS

We've discussed in previous lectures how the syntax analysis phase of compilation results in the creation of a parse tree.

Semantic analysis is performed by annotating, or *decorating*, the parse tree.

These annotations are known as *attributes*.

An attribute grammar "connects" syntax with semantics.

ATTRIBUTE GRAMMARS

- Each grammar production has a semantic rule with actions (e.g. assignments) to modify values of attributes of (non)terminals.
- A (non)terminal may have any number of attributes.
- Attributes have values that hold information related to the (non)terminal.

- General form:

production	**semantic rule**
<*A*> → <*B*> <*C*>	*A*.a := ...; *B*.a := ...; *C*.a := ...

ATTRIBUTE GRAMMARS

First, let's consider the following LR grammar. It derives strings of arithmetic expressions composed of const values.

$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow T / F$
$T \rightarrow F$
$F \rightarrow - F$
$F \rightarrow (E)$
$F \rightarrow \text{const}$

ATTRIBUTE GRAMMARS

Now, let's add meaning to the productions. We'll associate a *val* attribute with E, T, F, and const. The *val* of const is provided by the scanner as the image of the token.

$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow T / F$
$T \rightarrow F$
$F \rightarrow - F$
$F \rightarrow (E)$
$F \rightarrow \text{const}$ $\quad F.\text{val} := \text{const}.\text{val}$

Every production will only need a single rule in this grammar. We'll start with the *copy* rules. These allow us to specify that one attribute should copy the value of another.

ATTRIBUTE GRAMMARS

Now, let's add meaning to the productions. We'll associate a *val* attribute with E, T, F, and const. The *val* of const is provided by the scanner as the image of the token.

$E \rightarrow E + T$	
$E \rightarrow E - T$	
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T * F$	
$T \rightarrow T / F$	
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow - F$	
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow const$	$F.val := const.val$

ATTRIBUTE GRAMMARS

Now, let’s add meaning to the productions. We’ll associate a *val* attribute with E, T, F, and const. The *val* of const is provided by the scanner as the image of the token.

$E_1 \rightarrow E_2 + T$	$E_1.\mathrm{val} := \mathrm{sum}(E_2.\mathrm{val}, T.\mathrm{val})$
$E_1 \rightarrow E_2 - T$	
$E \rightarrow T$	$E.\mathrm{val} := T.\mathrm{val}$
$T_1 \rightarrow T_2 * F$	
$T_1 \rightarrow T_2 / F$	
$T \rightarrow F$	$T.\mathrm{val} := F.\mathrm{val}$
$F_1 \rightarrow - F_2$	
$F \rightarrow (E)$	$F.\mathrm{val} := E.\mathrm{val}$
$F \rightarrow \mathrm{const}$	$F.\mathrm{val} := \mathrm{const}.\mathrm{val}$

Now, we’ll invoke *semantic functions* which act on attributes and return a value to be stored in another attribute. To avoid ambiguity, we add subscripts to non-terminals.

ATTRIBUTE GRAMMARS

Now, let’s add meaning to the productions. We’ll associate a *val* attribute with E, T, F, and const. The *val* of const is provided by the scanner as the image of the token.

$E_1 \rightarrow E_2 + T$	$E_1.\text{val} := \text{sum}(E_2.\text{val}, T.\text{val})$
$E_1 \rightarrow E_2 - T$	$E_1.\text{val} := \text{diff}(E_2.\text{val}, T.\text{val})$
$E \rightarrow T$	$E.\text{val} := T.\text{val}$
$T_1 \rightarrow T_2 * F$	$T_1.\text{val} := \text{product}(T_2.\text{val}, F.\text{val})$
$T_1 \rightarrow T_2 \ / \ F$	$T_1.\text{val} := \text{quotient}(T_2.\text{val}, F.\text{val})$
$T \rightarrow F$	$T.\text{val} := F.\text{val}$
$F_1 \rightarrow - F_2$	$F_1.\text{val} := \text{negation}(F_2.\text{val})$
$F \rightarrow (E)$	$F.\text{val} := E.\text{val}$
$F \rightarrow \text{const}$	$F.\text{val} := \text{const}.\text{val}$

ATTRIBUTE GRAMMARS

Strictly speaking, attribute grammars only consist of copy rules and calls to semantic functions. But in practice, we can specify well-defined notation to make the semantic rules look more code-like.

$E_1 \rightarrow E_2 + T$	$E_1.\mathrm{val} := E_2.\mathrm{val} + T.\mathrm{val}$
$E_1 \rightarrow E_2 - T$	$E_1.\mathrm{val} := E_2.\mathrm{val} - T.\mathrm{val}$
$E \rightarrow T$	$E.\mathrm{val} := T.\mathrm{val}$
$T_1 \rightarrow T_2 * F$	$T_1.\mathrm{val} := T_2.\mathrm{val} * F.\mathrm{val}$
$T_1 \rightarrow T_2 / F$	$T_1.\mathrm{val} := T_2.\mathrm{val} / F.\mathrm{val}$
$T \rightarrow F$	$T.\mathrm{val} := F.\mathrm{val}$
$F_1 \rightarrow - F_2$	$F_1.\mathrm{val} := -F_2.\mathrm{val}$
$F \rightarrow (E)$	$F.\mathrm{val} := E.\mathrm{val}$
$F \rightarrow \mathrm{const}$	$F.\mathrm{val} := \mathrm{const.val}$

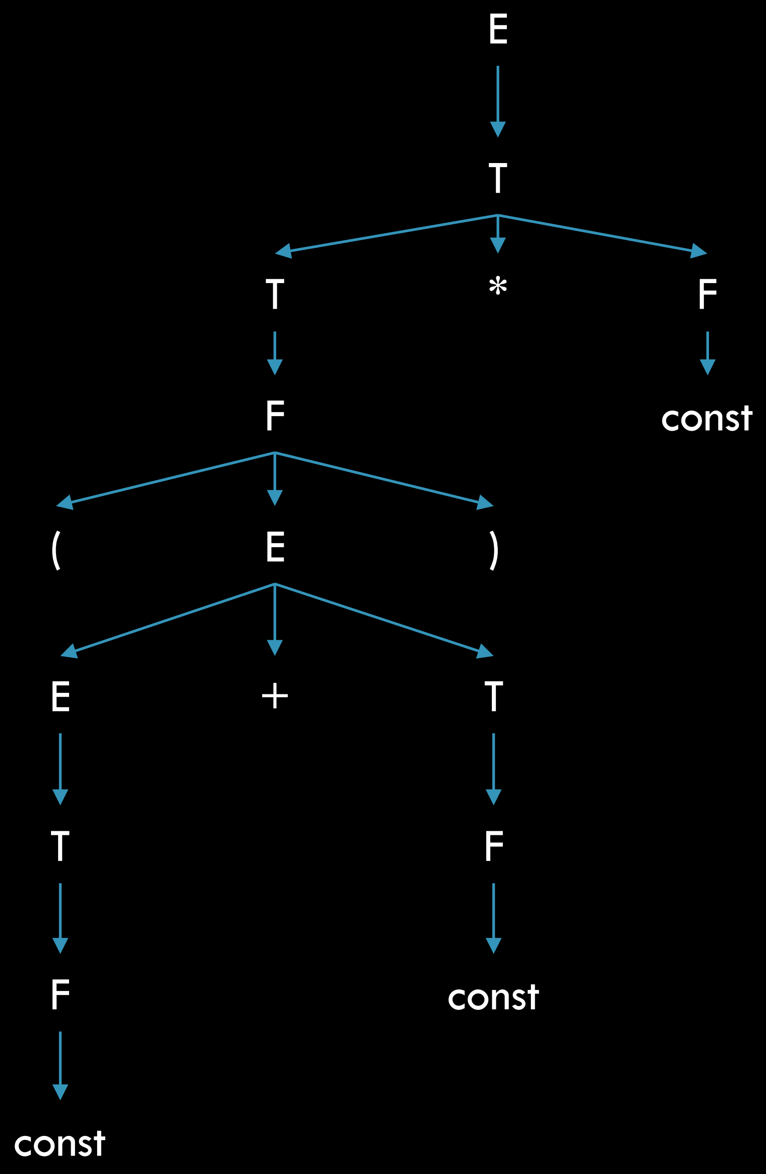
ATTRIBUTE GRAMMARS

Some points to remember:

- A (non)terminal may have any number of attributes.
- The *val* attribute of a (non)terminal holds the subtotal value of the subexpression.
- Nonterminals are indexed in the attribute grammar to distinguish multiple occurrences of the nonterminal in a production – this has no bearing on the grammar itself.
- Strictly speaking, attribute grammars only contain copy rules and semantic functions.
- Semantic functions may only refer to attributes in the current production.

DECORATED PARSE TREES

Evaluation of the attributes is called the decoration of the parse tree. Imagine we have the string (1+3)*2. The parse tree is shown here.



DECORATED PARSE TREES

Evaluation of the attributes is called the decoration of the parse tree. Imagine we have the string (1+3)*2. The parse tree is shown here.

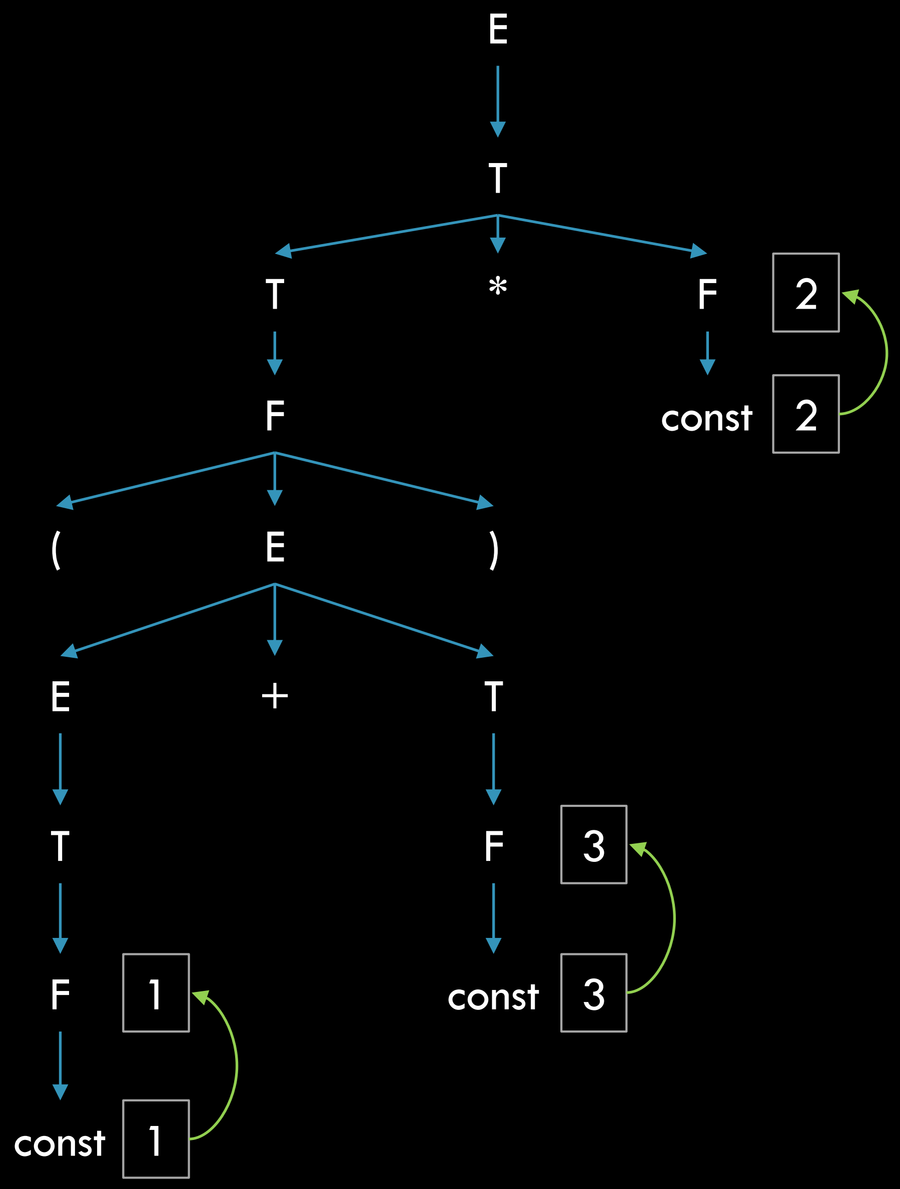
The val attribute of each symbol is shown beside it. Attribute flow is upward in this case.

$$\begin{tikzcd}
 & & E \arrow[d] & \\
 & & T \arrow[dl] \arrow[d] \arrow[dr] & \\
 & T \arrow[d] & * & F \arrow[d] \\
 & F \arrow[dl] \arrow[d] \arrow[dr] & & \text{const } \boxed{2} \\
(& E \arrow[dl] \arrow[d] \arrow[dr] &) & \\
E \arrow[d] & + & T \arrow[d] & \\
T \arrow[d] & & F \arrow[d] & \\
F \arrow[d] & & \text{const } \boxed{3} & \\
\text{const } \boxed{1} & & &
\end{tikzcd}$$

DECORATED PARSE TREES

Evaluation of the attributes is called the decoration of the parse tree. Imagine we have the string (1+3)*2. The parse tree is shown here.

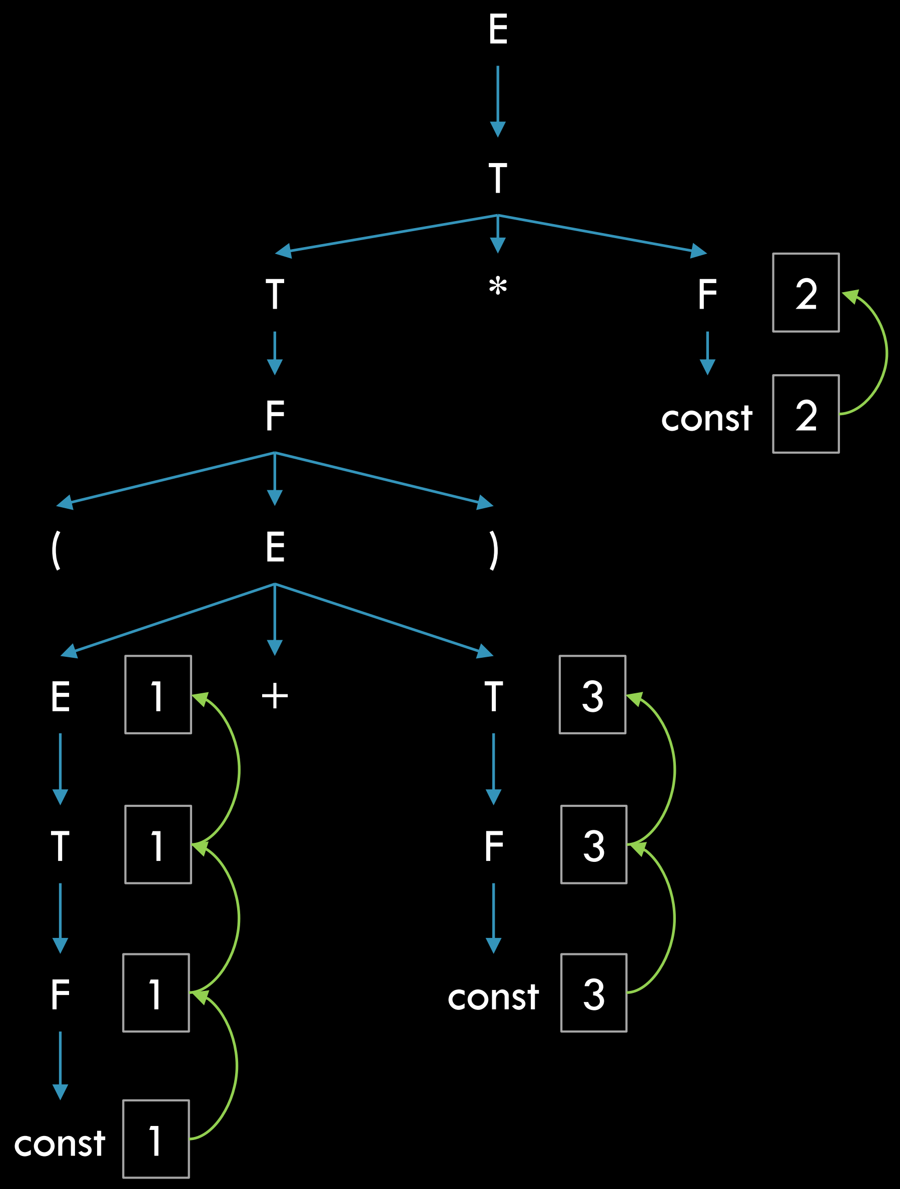
The val attribute of each symbol is shown beside it. Attribute flow is upward in this case.



DECORATED PARSE TREES

Evaluation of the attributes is called the decoration of the parse tree. Imagine we have the string (1+3)*2. The parse tree is shown here.

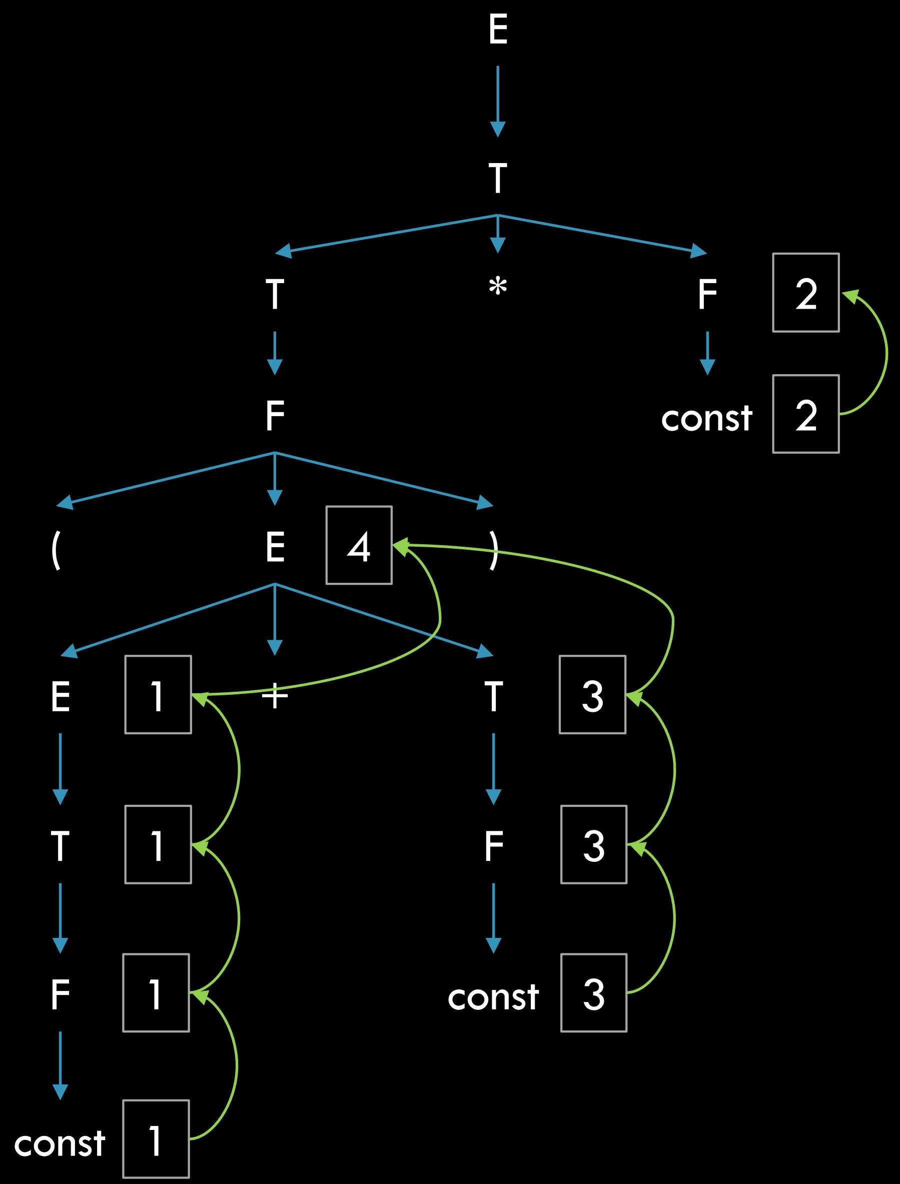
The val attribute of each symbol is shown beside it. Attribute flow is upward in this case.



DECORATED PARSE TREES

Evaluation of the attributes is called the decoration of the parse tree. Imagine we have the string (1+3)*2. The parse tree is shown here.

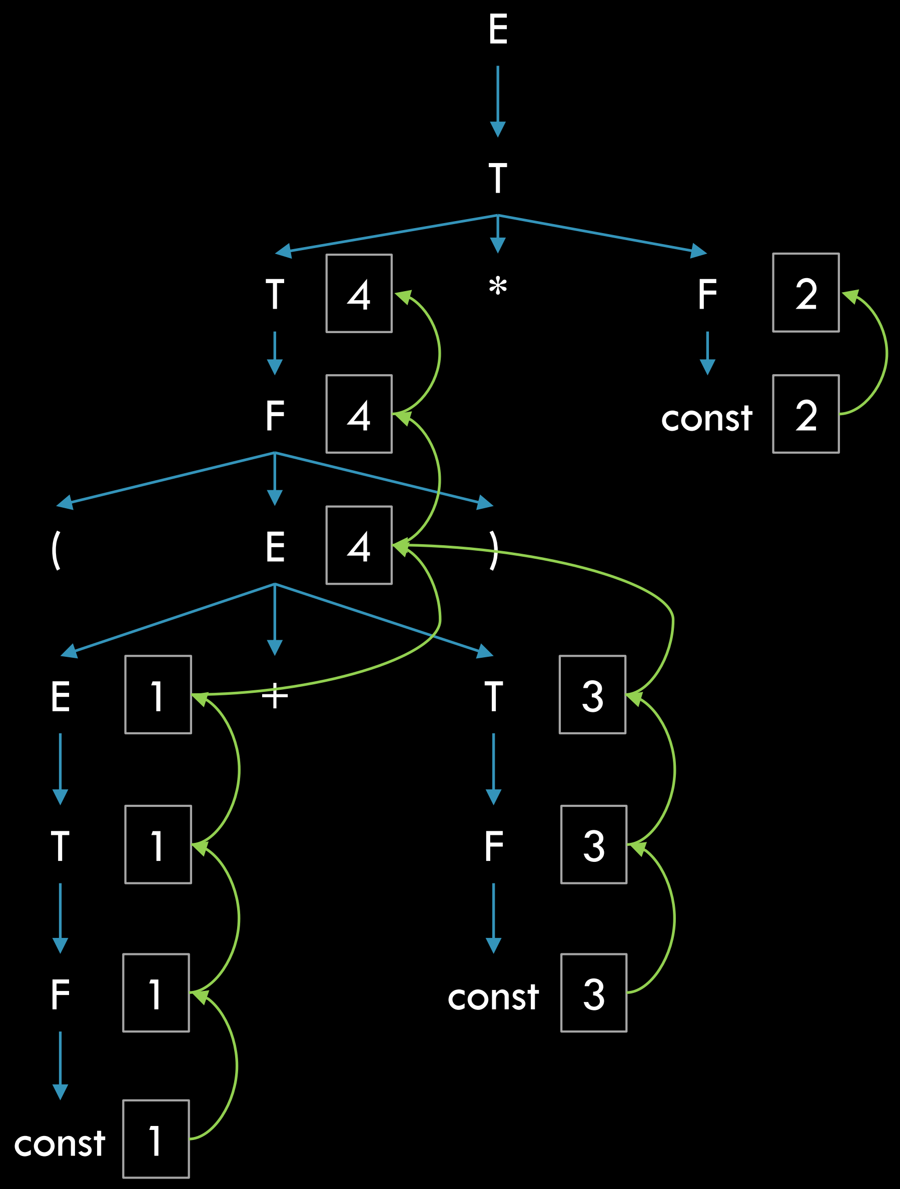
The val attribute of each symbol is shown beside it. Attribute flow is upward in this case.



DECORATED PARSE TREES

Evaluation of the attributes is called the decoration of the parse tree. Imagine we have the string (1+3)*2. The parse tree is shown here.

The val attribute of each symbol is shown beside it. Attribute flow is upward in this case.

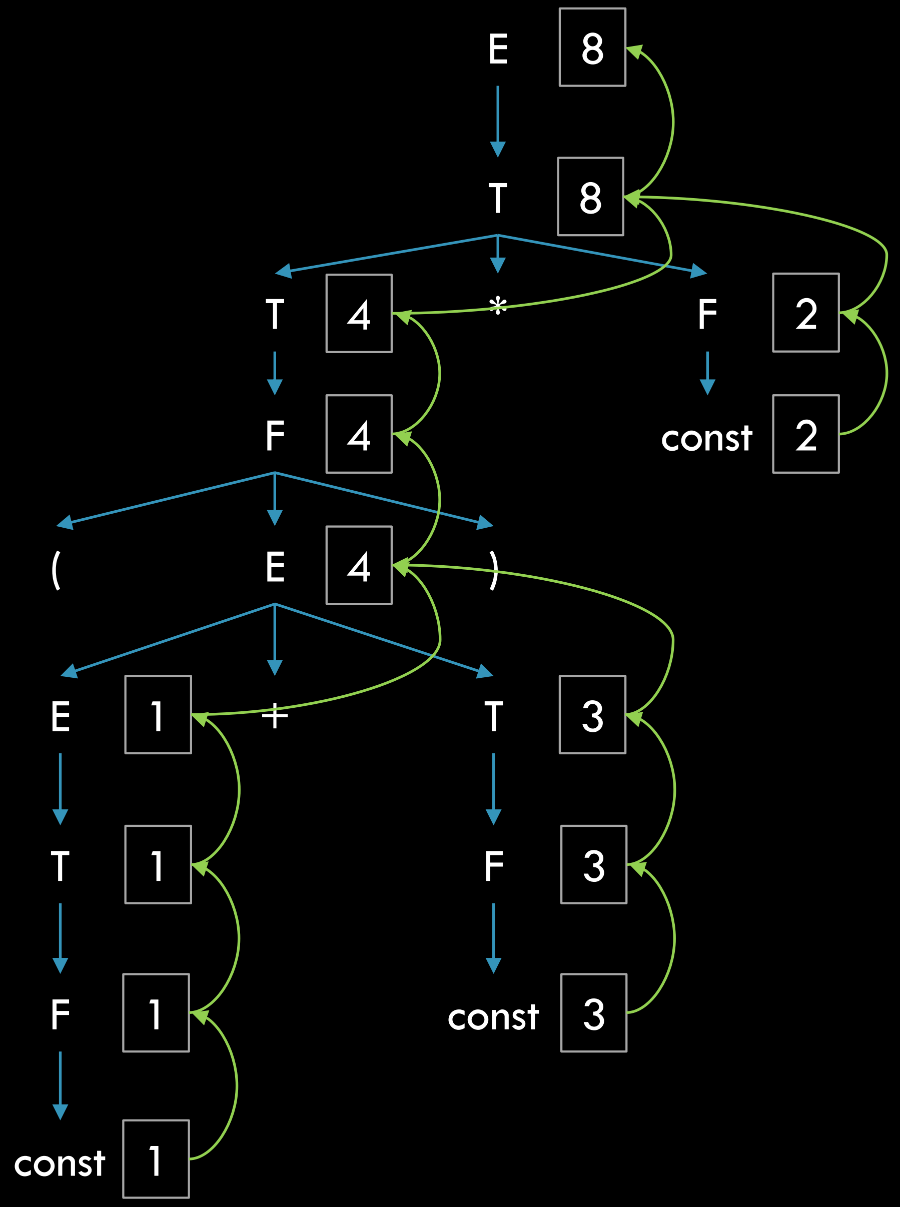


DECORATED PARSE TREES

Evaluation of the attributes is called the decoration of the parse tree. Imagine we have the string (1+3)*2. The parse tree is shown here.

The val attribute of each symbol is shown beside it. Attribute flow is upward in this case.

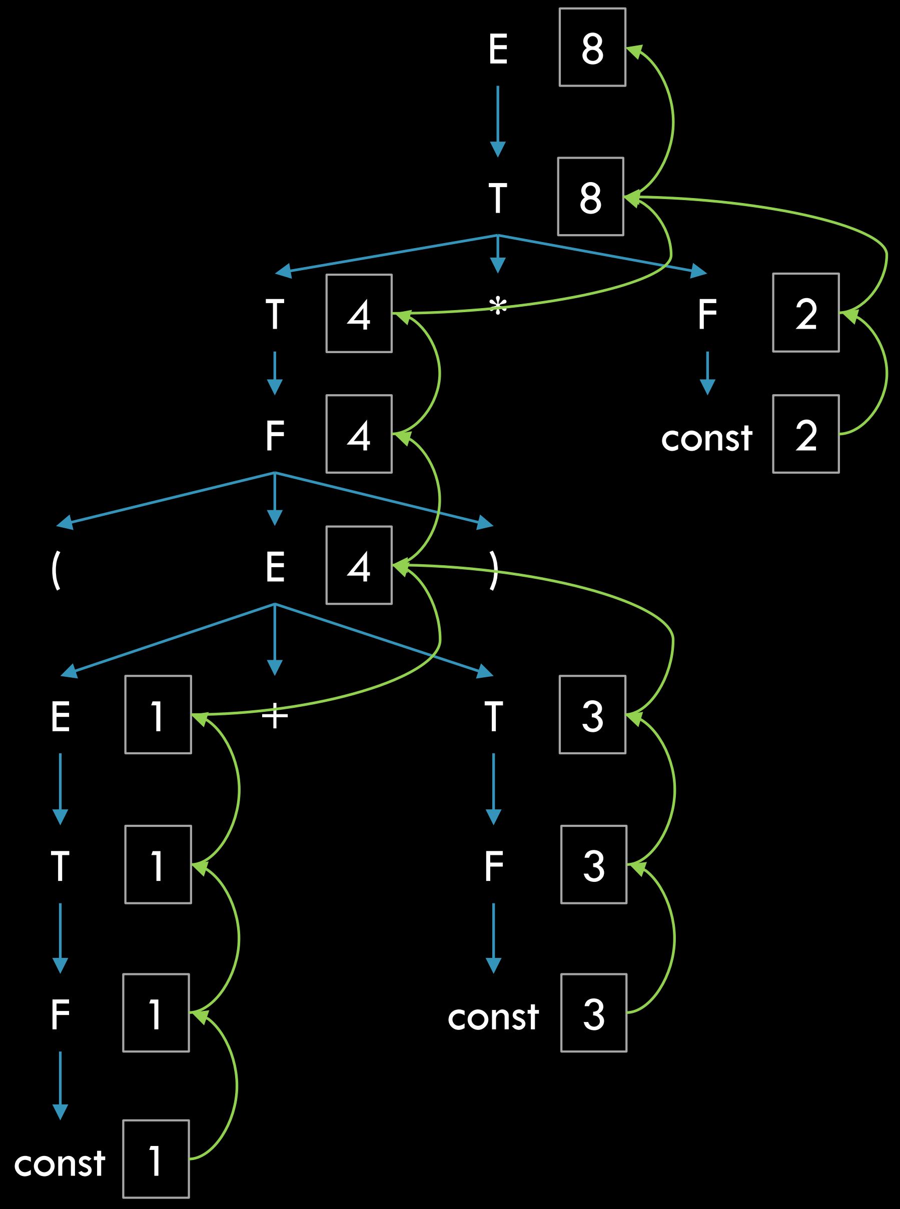
The val of the overall expression is the val of the root.



DECORATED PARSE TREES

The attribute grammar represented is very simple.

- All the symbols have at most one attribute.
- All the attributes are *synthesized* -- values calculated only in productions where their symbol is on the left-hand side.



SYNTHESIZED AND INHERITED ATTRIBUTES

Each grammar production A → ω is associated with a set of semantic rules of the form

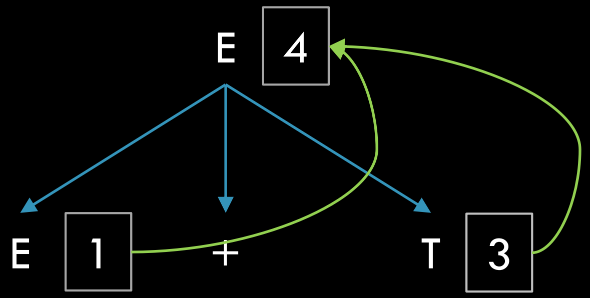
b := f(c1, c2, …, ck)

- If b is an attribute associated with A, it is called a synthesized attribute.
- If b is an attribute associated with a grammar symbol on the right side of the production (that is, in ω) then b is called an inherited attribute.

SYNTHESIZED ATTRIBUTES

Synthesized attributes of a node hold values that are computed from attribute values of the child nodes in the parse tree and therefore information flows upwards.

production	**semantic rule**
$E_1 \rightarrow E_2 + T$	$E_1.\text{val} := E_2.\text{val} + T.\text{val}$



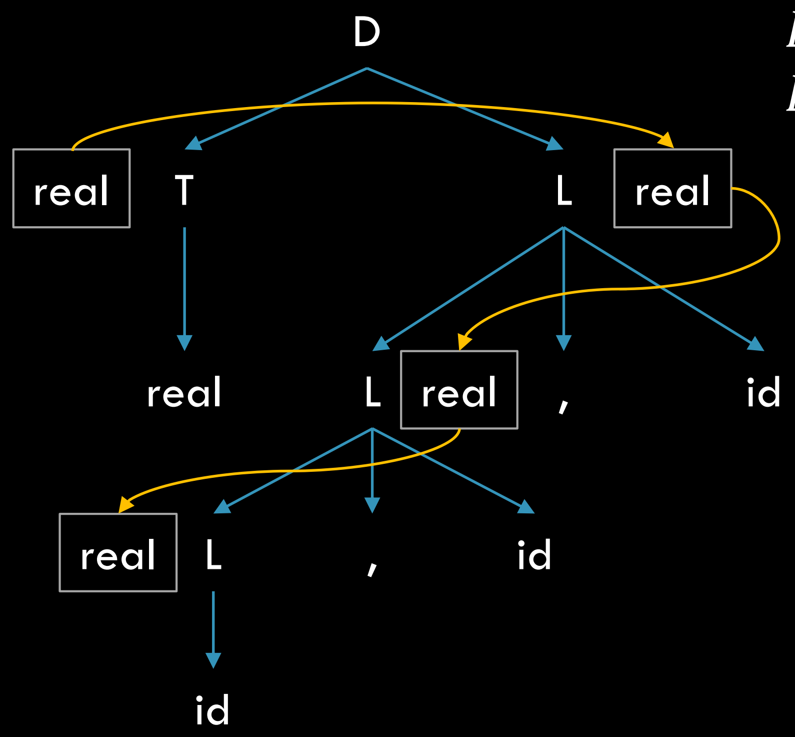
INHERITED ATTRIBUTES

Inherited attributes of child nodes are set by the parent node or sibling nodes and therefore information flows downwards. Consider the following attribute grammar.

$D \rightarrow T\ L$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in,\ \ addtype(id.entry, L.in)$
$L \rightarrow id$	$addtype(id.entry, L.in)$

real id1, id2, id3

INHERITED ATTRIBUTES



$D \rightarrow T\ L$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in, addtype(id.entry, L.in)$
$L \rightarrow id$	$addtype(id.entry, L.in)$

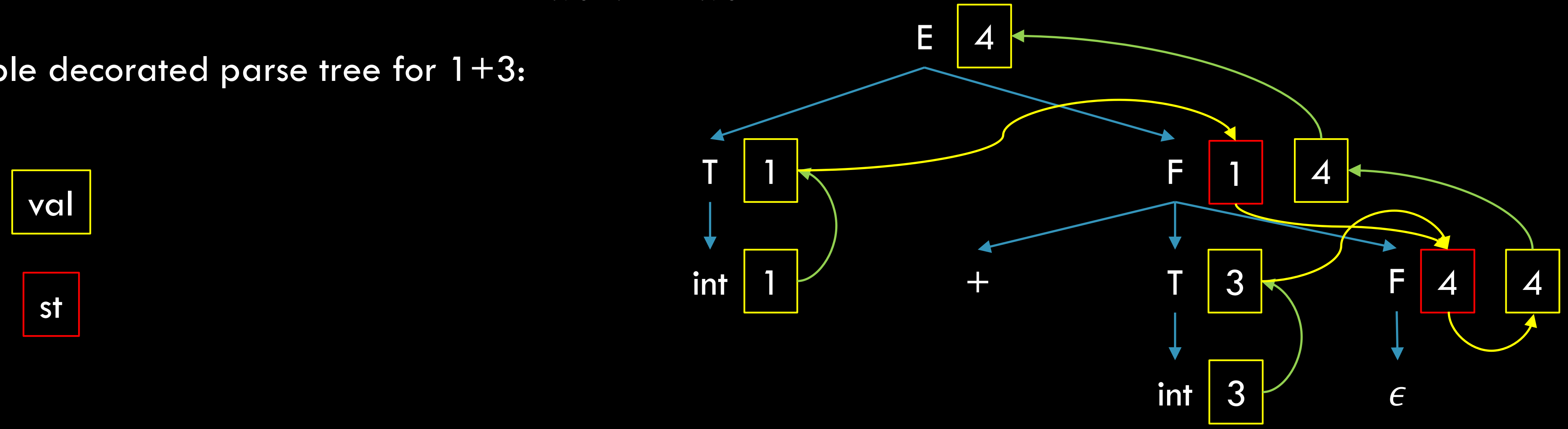
Notice how *L.in* is inherited from sibling *T.type* or from parent *L.in*.

SYNTHESIZED AND INHERITED ATTRIBUTES

Here's an example that exhibits both types of attributes.

production	**semantic rule**
$E \rightarrow T\ F$	F.st := T.val; E.val := F.val
$F_1 \rightarrow +\ T\ F_2$	F_2.st := F_1.st + T.val; F_1.val := F_2.val
$F \rightarrow \epsilon$	F.val := F.st
$T \rightarrow int$	T.val := int.val

Example decorated parse tree for 1+3:



ATTRIBUTE FLOW

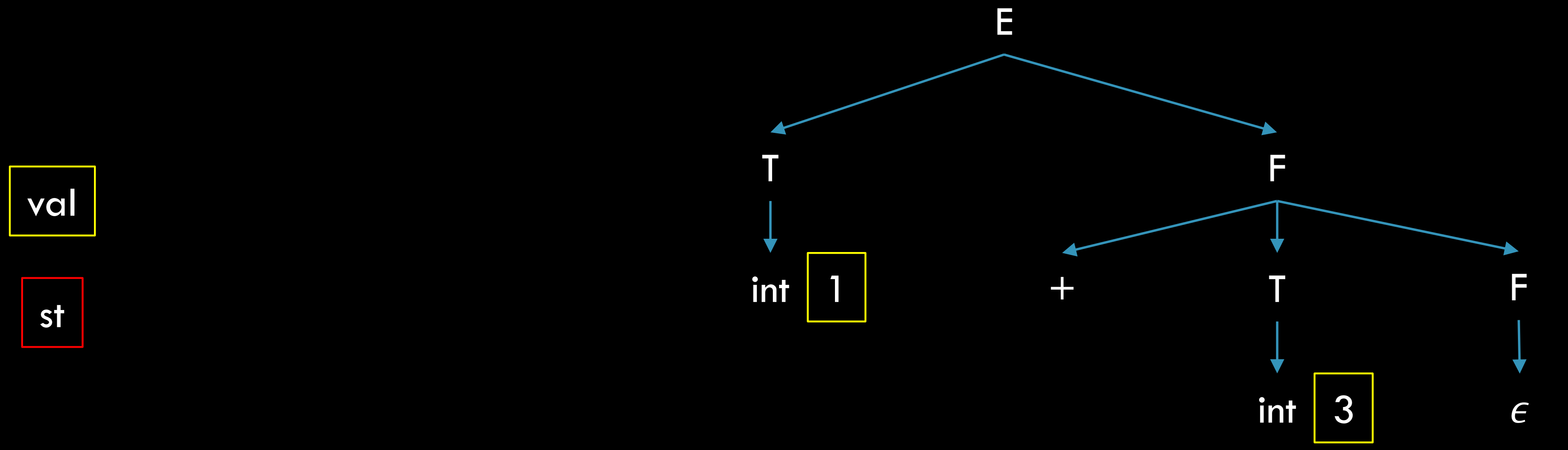
In the same way that a context-free grammar does not indicate how a string should be parsed, an attribute grammar does not specify how the attribute rules should be applied. It merely defines the set of valid decorated parse trees, not how they are constructed.

An *attribute flow algorithm* propagates attribute values through the parse tree by traversing the tree according to the *set* (write) and *use* (read) dependencies (an attribute must be set before it is used).

ATTRIBUTE FLOW

Consider the previous decorated parse tree.

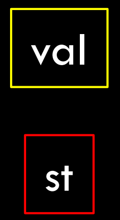
production	**semantic rule**
$E \rightarrow T\ F$	F.st := T.val; E.val := F.val
$F_1 \rightarrow +\ T\ F_2$	F_2.st := F_1.st + T.val; F_1.val := F_2.val
$F \rightarrow \epsilon$	F.val := F.st
$T \rightarrow int$	T.val := int.val

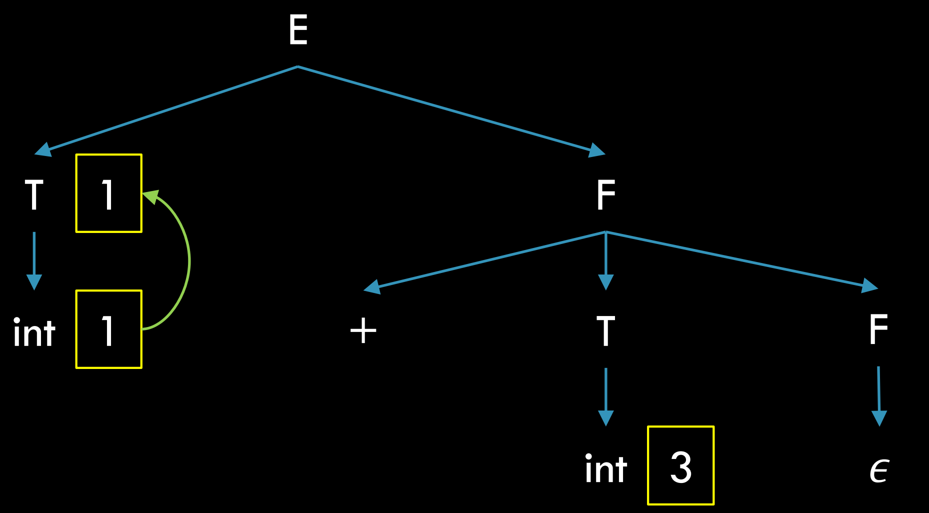


ATTRIBUTE FLOW

Consider the previous decorated parse tree.

production	**semantic rule**
$E \rightarrow T\ F$	$F.st := T.val;\quad E.val := F.val$
$F_1 \rightarrow +\ T\ F_2$	$F_2.st := F_1.st + T.val;\quad F_1.val := F_2.val$
$F \rightarrow \epsilon$	$F.val := F.st$
$T \rightarrow int$	$T.val := int.val$

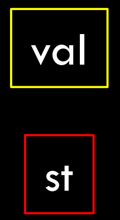


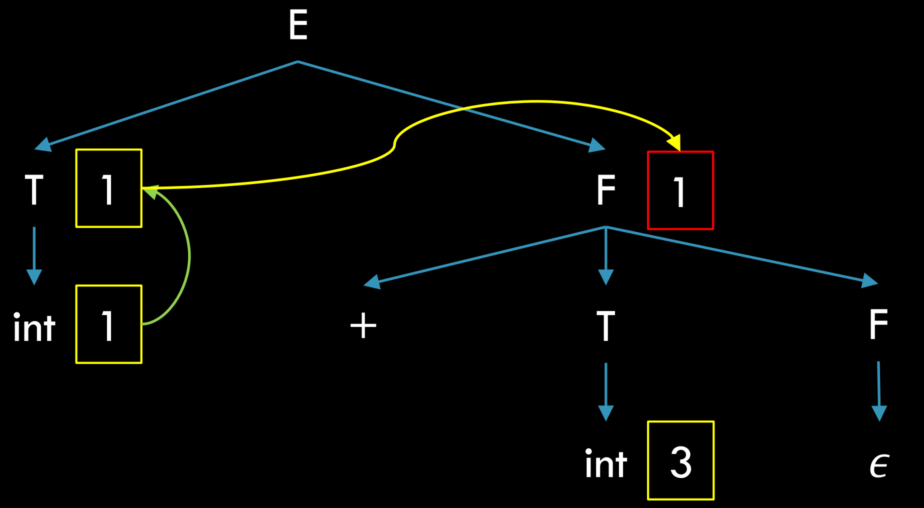


ATTRIBUTE FLOW

Consider the previous decorated parse tree.

production	**semantic rule**
$E \rightarrow T\ F$	F.st := T.val; E.val := F.val
$F_1 \rightarrow +\ T\ F_2$	F_2.st := F_1.st + T.val; F_1.val := F_2.val
$F \rightarrow \epsilon$	F.val := F.st
$T \rightarrow int$	T.val := int.val

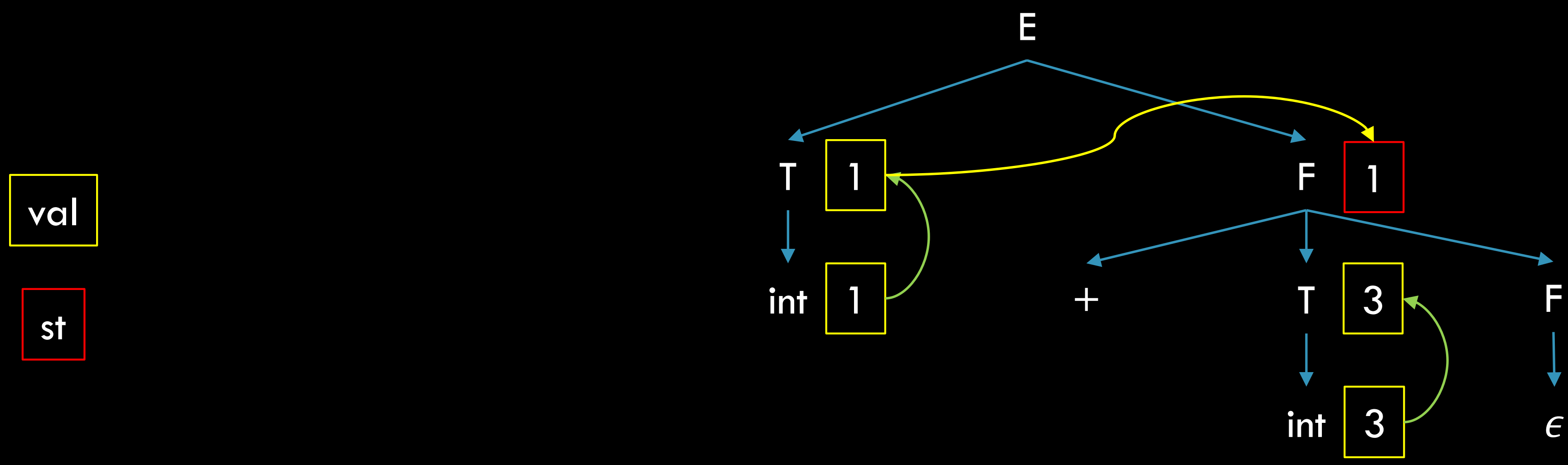




ATTRIBUTE FLOW

Consider the previous decorated parse tree.

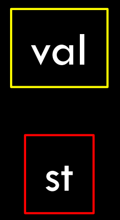
production	**semantic rule**
$E \rightarrow T\ F$	$F.\mathrm{st} := T.\mathrm{val}; \quad E.\mathrm{val} := F.\mathrm{val}$
$F_1 \rightarrow +\ T\ F_2$	$F_2.\mathrm{st} := F_1.\mathrm{st} + T.\mathrm{val}; \quad F_1.\mathrm{val} := F_2.\mathrm{val}$
$F \rightarrow \epsilon$	$F.\mathrm{val} := F.\mathrm{st}$
$T \rightarrow int$	$T.val := int.val$

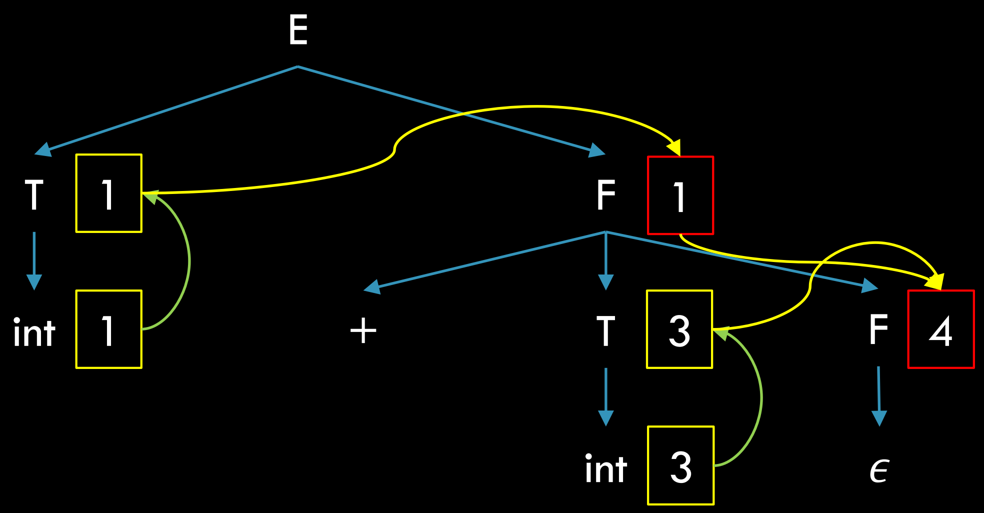


ATTRIBUTE FLOW

Consider the previous decorated parse tree.

production	**semantic rule**
$E \rightarrow T\ F$	$F.st := T.val;\quad E.val := F.val$
$F_1 \rightarrow +\ T\ F_2$	$F_2.st := F_1.st + T.val;\quad F_1.val := F_2.val$
$F \rightarrow \epsilon$	$F.val := F.st$
$T \rightarrow int$	$T.val := int.val$

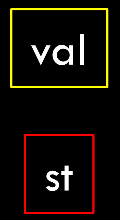


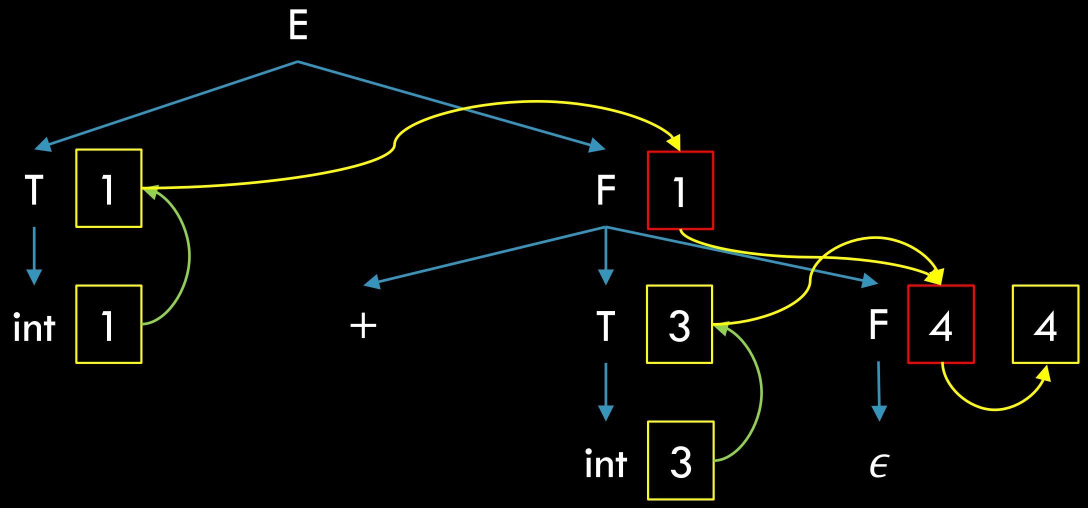


ATTRIBUTE FLOW

Consider the previous decorated parse tree.

production	**semantic rule**
$E \rightarrow T\ F$	$F.st := T.val; \quad E.val := F.val$
$F_1 \rightarrow +\ T\ F_2$	$F_2.st := F_1.st + T.val; \quad F_1.val := F_2.val$
$F \rightarrow \epsilon$	$F.val := F.st$
$T \rightarrow int$	$T.val := int.val$

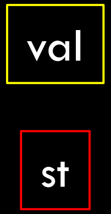


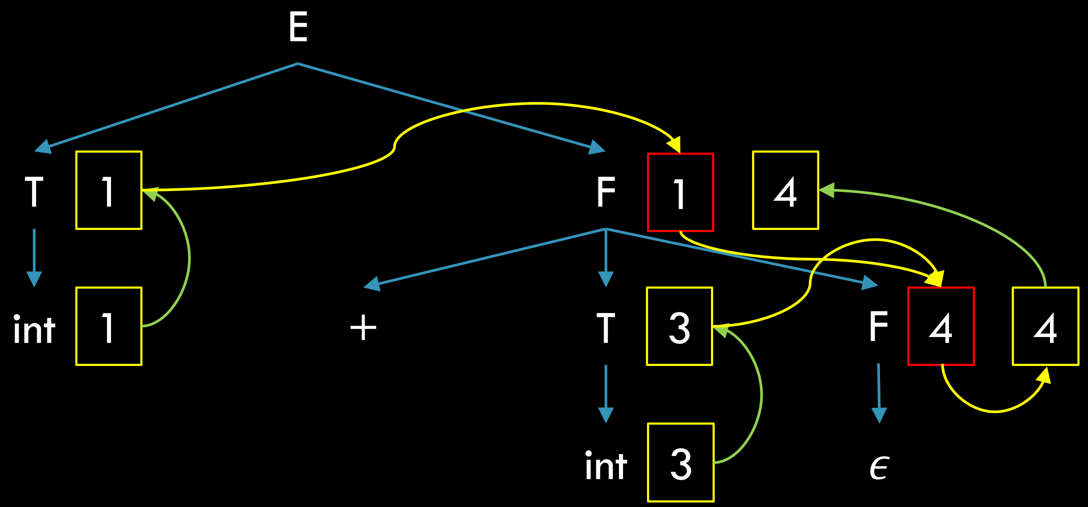


ATTRIBUTE FLOW

Consider the previous decorated parse tree.

production	**semantic rule**
$E \rightarrow T\ F$	F.st := T.val; E.val := F.val
$F_1 \rightarrow +\ T\ F_2$	F_2.st := F_1.st + T.val; F_1.val := F_2.val
$F \rightarrow \epsilon$	F.val := F.st
$T \rightarrow int$	T.val := int.val

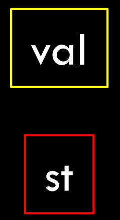


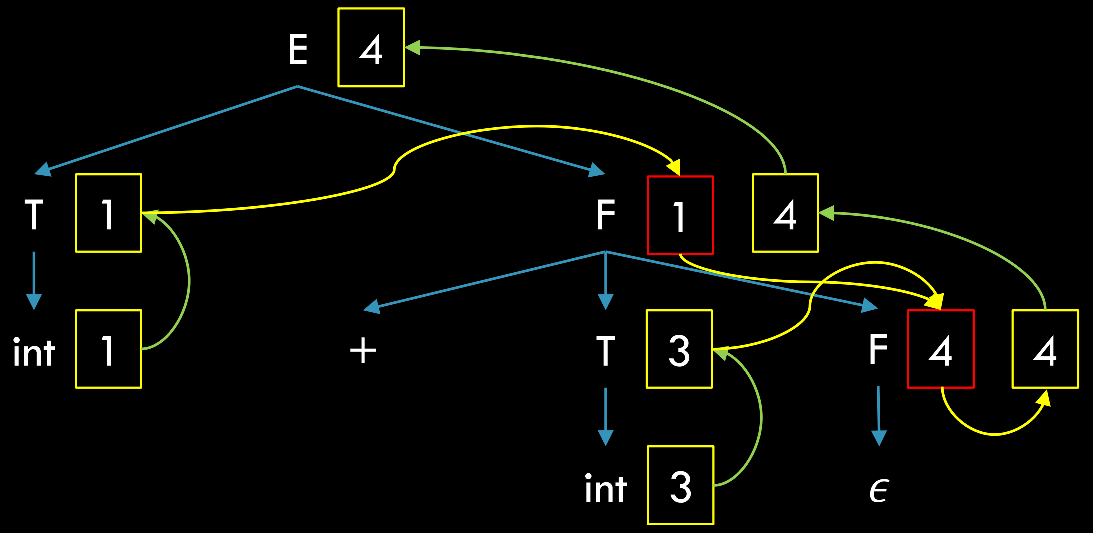


ATTRIBUTE FLOW

Consider the previous decorated parse tree.

production	**semantic rule**
$E \rightarrow T\ F$	F.st := T.val; E.val := F.val
$F_1 \rightarrow +\ T\ F_2$	F_2.st := F_1.st + T.val; F_1.val := F_2.val
$F \rightarrow \epsilon$	F.val := F.st
$T \rightarrow int$	T.val := int.val





S- AND L- ATTRIBUTED GRAMMARS

A grammar is called S-attributed if all attributes are synthesized.

$E_1 \rightarrow E_2 + T$	$E_1.\mathrm{val} := E_2.\mathrm{val} + T.\mathrm{val}$
$E_1 \rightarrow E_2 - T$	$E_1.\mathrm{val} := E_2.\mathrm{val} - T.\mathrm{val}$
$E \rightarrow T$	$E.\mathrm{val} := T.\mathrm{val}$
$T_1 \rightarrow T_2 * F$	$T_1.\mathrm{val} := T_2.\mathrm{val} * F.\mathrm{val}$
$T_1 \rightarrow T_2 \; / \; F$	$T_1.\mathrm{val} := T_2.\mathrm{val} / F.\mathrm{val}$
$T \rightarrow F$	$T.\mathrm{val} := F.\mathrm{val}$
$F_1 \rightarrow - F_2$	$F_1.\mathrm{val} := -F_2.\mathrm{val}$
$F \rightarrow (E)$	$F.\mathrm{val} := E.\mathrm{val}$
$F \rightarrow \mathrm{const}$	$F.\mathrm{val} := \mathrm{const}.\mathrm{val}$

S- AND L- ATTRIBUTED GRAMMARS

A grammar is called L-attributed if the parse tree traversal to update attribute values is always left-to-right and depth-first.

- For a production A → $X_1 \ X_2 \ X_3 \ \ldots X_n$
  - The attributes of X_j (1<= j <= n) only depend on:
    - The attributes of $X_1 \ X_2 \ X_3 \ \ldots X_{j-1}$.
    - The inherited attributes of A.

Values of inherited attributes must be passed down to children from left to right.

Semantic rules can be applied immediately during parsing and parse trees do not need to be kept in memory. This is an essential grammar property for a one-pass compiler.

An S-attributed grammar is a special case of an L-attributed grammar.

NEXT LECTURE

Semantic Analysis and Yacc