Semantic Analysis

- Ultimate goal: generate machine code.
- Before we generate code, we must collect information about the program:
- Front end
  - scanning (recognizing words) CHECK
  - parsing (recognizing syntax) CHECK
  - semantic analysis (recognizing meaning)
    - There are issues deeper than structure. Consider:

```
int func (int x, int y);
int main () {
        int list[5], i, j;
        char *str;
        j = 10 + 'b';
        str = 8;
        m  = func("aa", j, list[12]);
        return 0;
}
```

This code is syntactically correct, but will not work. What problems are there?

Semantic analysis

- Collecting type information may involve "computations"
  - What is the type of x+y given the types of x and y?
- Tool: attribute grammars
  - CFG
  - Each grammar symbol has associated attributes
  - The grammar is augmented by rules (semantic actions) that specify how the values of attributes are computed from other attributes.
  - The process of using semantic actions to evaluate attributes is called syntax-directed translation.
  - Examples:
    - Grammar of declarations.
    - Grammar of signed binary numbers.

Attribute grammars

Example 1: Grammar of declarations

Production	Semantic rule
$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ char	T.type = character
$L \rightarrow L_1$, id	L_1.in = L.in addtype (id.index, L.in)
$L \rightarrow$ id	addtype (id.index, L.in)

Attribute grammars

Example 2: Grammar of signed binary numbers

Production	Semantic rule
$N \rightarrow S\ L$	if (S.neg) print('-'); else print('+'); print(L.val);
$S \rightarrow +$	S.neg = 0
$S \rightarrow -$	S.neg = 1
$L \rightarrow L_1, B$	$L.val = 2*L_1.val+B.val$
$L \rightarrow B$	L.val = B.val
$B \rightarrow 0$	$B.val = 0*2^0$
$B \rightarrow 1$	$B.val = 1*2^0$

Attribute grammars

Example 3: Grammar of expressions
Creating an AST

The attribute for each non-terminal is a node of the tree.

Production	Semantic rule
$E \rightarrow E_1 + E_2$	E.node = new PlusNode(E_1.node,E_2.node)
$E \rightarrow$ num	E.node = num.yylval
$E \rightarrow (E_1)$	E.node = E_1.node

Syntax-Directed Definitions and Translation Schemes

- When we associate semantic rules with productions, we use two notations:
  - **Syntax-Directed Definitions**
  - **Translation Schemes**
- **Syntax-Directed Definitions:**
  - give high-level specifications for translations
  - hide many implementation details such as order of evaluation of semantic actions.
  - We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.
- **Translation Schemes:**
  - indicate the order of evaluation of semantic actions associated with a production rule.
  - In other words, translation schemes give a little bit information about implementation details.

Syntax-Directed Definitions and Translation Schemes

- With each production in a grammar, we give semantic rules or *actions, which describe* how to compute the attribute values associated with each grammar symbol in a production. The attribute value for a parse node may depend on information from its children nodes below or its siblings and parent node above.
- Evaluation of these semantic rules (using SDT one can perform following with parser):
  - may generate intermediate codes
  - may put information into the symbol table
  - may perform consistency check like type checking, parameter checking etc…
  - may issue error messages
  - may build syntax tree
  - in fact, they may perform almost any activities.
- Procedure :

1) Input – Grammer
2) Output – Attached semantic rules

Syntax-Directed Definitions

- A syntax-directed definition is a generalization of a context-free grammar in which:
  - Each grammar symbol is associated with a set of attributes.
  - This set of attributes for a grammar symbol is partitioned into two subsets called **synthesized** and **inherited** attributes of that grammar symbol.
  - Each production rule is associated with a set of semantic rules.
- *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*.
- This *dependency graph* determines the evaluation order of these semantic rules.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.
- The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.
- The order of these computations depends on the dependency graph induced by the semantic rules.

- An attribute is said to be **synthesized** if its value at a parse tree node is determined by the attribute values at the child nodes.
- An attribute is said to be **inherited** if its value at a parse tree node is determined by the attribute values of the parent and/or siblings of that node.

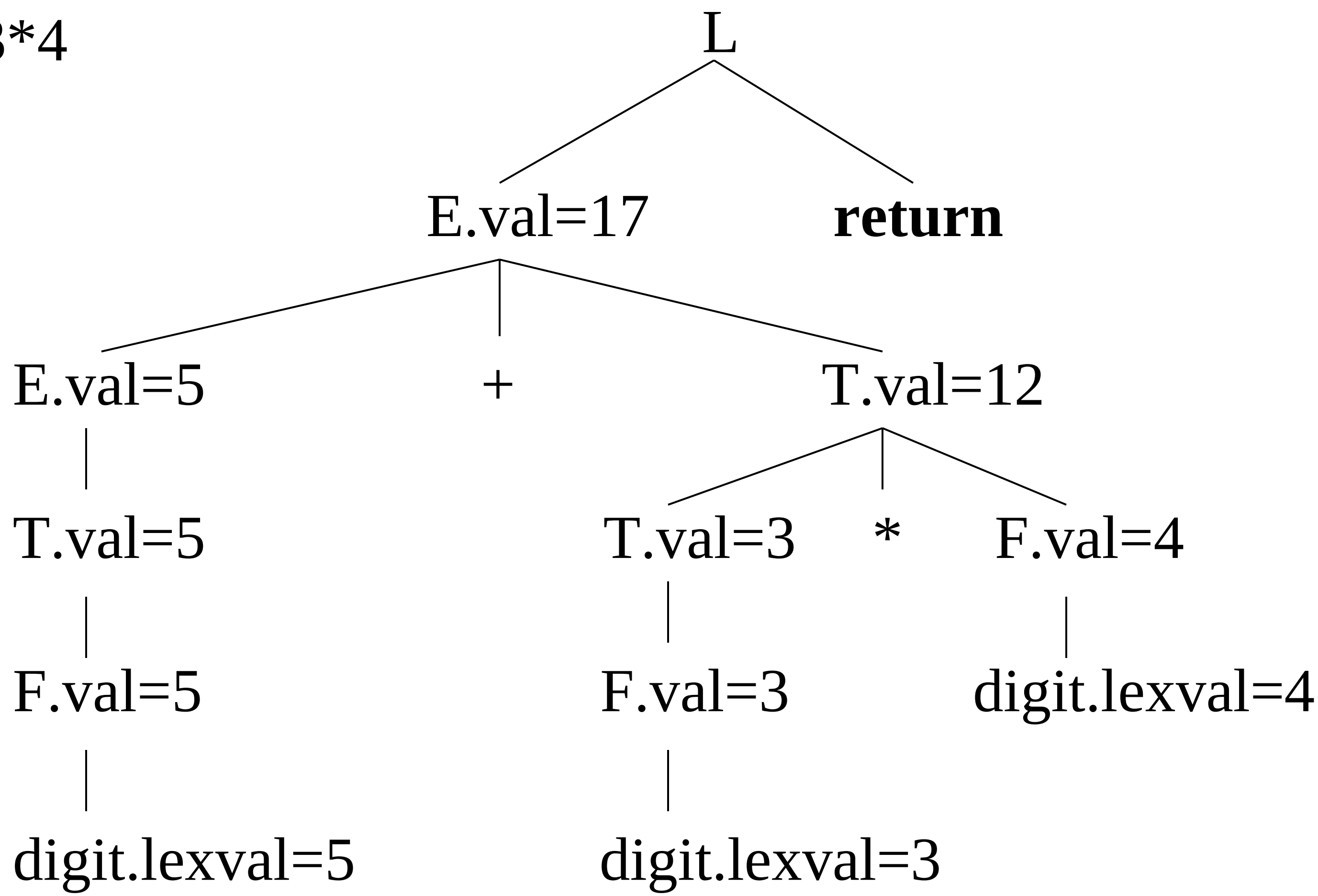
Example

Production	**Semantic Rules**
$L \rightarrow E$ **return**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	F.val = **digit**.lexval

- Symbols E, T, and F are associated with a synthesized attribute *val*.
- The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).
- Terminals attributes calculated at the time of lexical analysis phase

Annotated Parse Tree -- Example

Input: 5+3*4



Example - Inherited Attributes

<u>**Production**</u>	<u>**Semantic Rules**</u>
$D \rightarrow T\ L$	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)

- Symbol T is associated with a synthesized attribute *type*.
- Symbol L is associated with an inherited attribute *in*.

Translation Schemes

- In a syntax-directed definition, we do not say anything about the evaluation times of the semantic rules (when the semantic rules associated with a production should be evaluated?).

- A **translation scheme** is a context-free grammar in which:
  - attributes are associated with the grammar symbols and
  - semantic actions enclosed between braces {} are inserted within the right sides of productions.

- *Ex:* $A \rightarrow \{ \dots \}\ X\ \{ \dots \}\ Y\ \{ \dots \}$

Semantic Actions

Translation Schemes

- When designing a translation scheme, some restrictions should be observed to ensure that an attribute value is available when a semantic action refers to that attribute.
- These restrictions (motivated by L-attributed definitions) ensure that a semantic action does not refer to an attribute that has not yet computed.
- In translation schemes, we use *semantic action* terminology instead of *semantic rule* terminology used in syntax-directed definitions.
- The position of the semantic action on the right side indicates when that semantic action will be evaluated.