Semantic Analysis -1

Semantic Analysis

- **Lexical analysis**: Source code to tokens
- **Parsing**: Validate token conformity with grammar

- Now we need to check if the code makes sense (have correct meaning)
    - Lexically and syntactically correct programs may still contain other errors – correct usage of variables, objects, functions, ...

- **Semantic analysis:** Ensure that the program satisfies a set of rules regarding the usage of programming constructs (variables, objects, expressions, statements)

Static Vs. Dynamic Semantics

- **Static Semantics**
  - Variables are declared before use
  - Types match on both sides of an argument
  - Parameter types and number match in declaration and use

- **Dynamic Semantics**
  - Whether overflow will occur during an arithmetic operation
  - Whether array limits will be crossed during execution
  - Whether recursion will cross stack limits
  - Whether heap memory will be insufficient

Static Vs. Dynamic Semantics: Example

```
int dot_prod (int x[], int y []) {
    int d, i;  d= 0;
    for ( i = 0; i < 10; i++) d += x[i]*y[i];
    return d;
}

main () {
    int p; int a[10], b[10];
    p = dot_prod (a, b);
}
```

- Static Checks
  - Types of *p* and return type of dot_prod() match
  - Type of *d* matches with the result type of ‘*’
  - Elements of arrays ‘*x*’ and ‘*y*’ are compatible with ‘*’
  - Number and type of parameters in dot_prod() match in declaration and use
  - *p*, *a*, *b* are declared before use
- Dynamic Checks
  - Value of *i* do not exceed the declared range of arrays *x* and *y*
  - There are no overflows during operations of ‘*’ and ‘+’ in d+=x[i]*y[i];

Static Semantics: Errors Produced by GCC

```
int dot_product(int a[], int b[]) {...}

1 main(){int a[10]={1,2,3,4,5,6,7,8,9,10};
2 int b[10]={1,2,3,4,5,6,7,8,9,10};
3 printf("%d", dot_product(b));
4 printf("%d", dot_product(a,b,a));
5 int p[10]; p=dotproduct(a,b); printf("%d",p);}

In function 'main':
error in 3: too few arguments to fn 'dot_product'
error in 4: too many arguments to fn 'dot_product'
error in 5: incompatible types in assignment
warning in 5: format '%d' expects type 'int', but
              argument 2 has type 'int *'
```

Semantic Analysis

- Type information is stored in the symbol table or the syntax-tree itself
  - Types of variables, function parameters, array dimensions etc.
  - Used for semantic validation but also for subsequent phases of compilation

- Static semantics of programming languages can be specified using attribute grammars

- Semantic analyzers can be generated semi-automatically from attribute grammars.

- Attribute grammars are extensions of context free grammars

Syntax-Directed Translation

1. Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
2. Values of these attributes are evaluated by the **semantic rules** associated with the production rules.

3. Evaluation of these semantic rules:
   – may generate intermediate codes
   – may put information into the symbol table
   – may perform type checking
   – may issue error messages
   – may perform some other activities
   – in fact, they may perform almost any activity.

4. An attribute may hold almost any thing.
   – a string, a number, a memory location, a complex record.

Syntax-Directed Definitions

1. 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.

2. *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*.

3. This *dependency graph* determines the evaluation order of these semantic rules.

4. 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

1. A parse tree showing the values of attributes at each node is called an **annotated parse tree**.

2. The process of computing the attributes' values at the nodes is called **annotating** (or **decorating**) of the parse tree.

3. The order of these computations depends on the dependency graph induced by the semantic rules.

Syntax-Directed Definition

In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

$$b=f(c_1,c_2,\ldots,c_n)$$

where f is a function and b can be one of the following:

➔ b is a synthesized attribute of A and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

OR

➔ b is an inherited attribute of one of the grammar symbols in α (on the RHS of the production), and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

Attribute Grammar

- So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends* o*n* attributes $c_1,c_2,\ldots,c_n$.

- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.

- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

Synthesized and Inherited Attributes

An attribute cannot be both synthesized and inherited, but a symbol can have both types of attributes

Attributes of symbols are evaluated over a parse tree by making passes over the parse tree

Synthesized attributes are computed in a bottom-up fashion from the leaves upwards

- Always synthesized from the attribute values of the children of the node
- Leaf nodes (terminals) have synthesized attributes initialized by the lexical analyzer and cannot be modified
- An AG with only synthesized attributes is an *S-attributed grammar (SAG)*
- YACC permits only SAGs

Inherited attributes flow down from the parent or siblings to the node in question

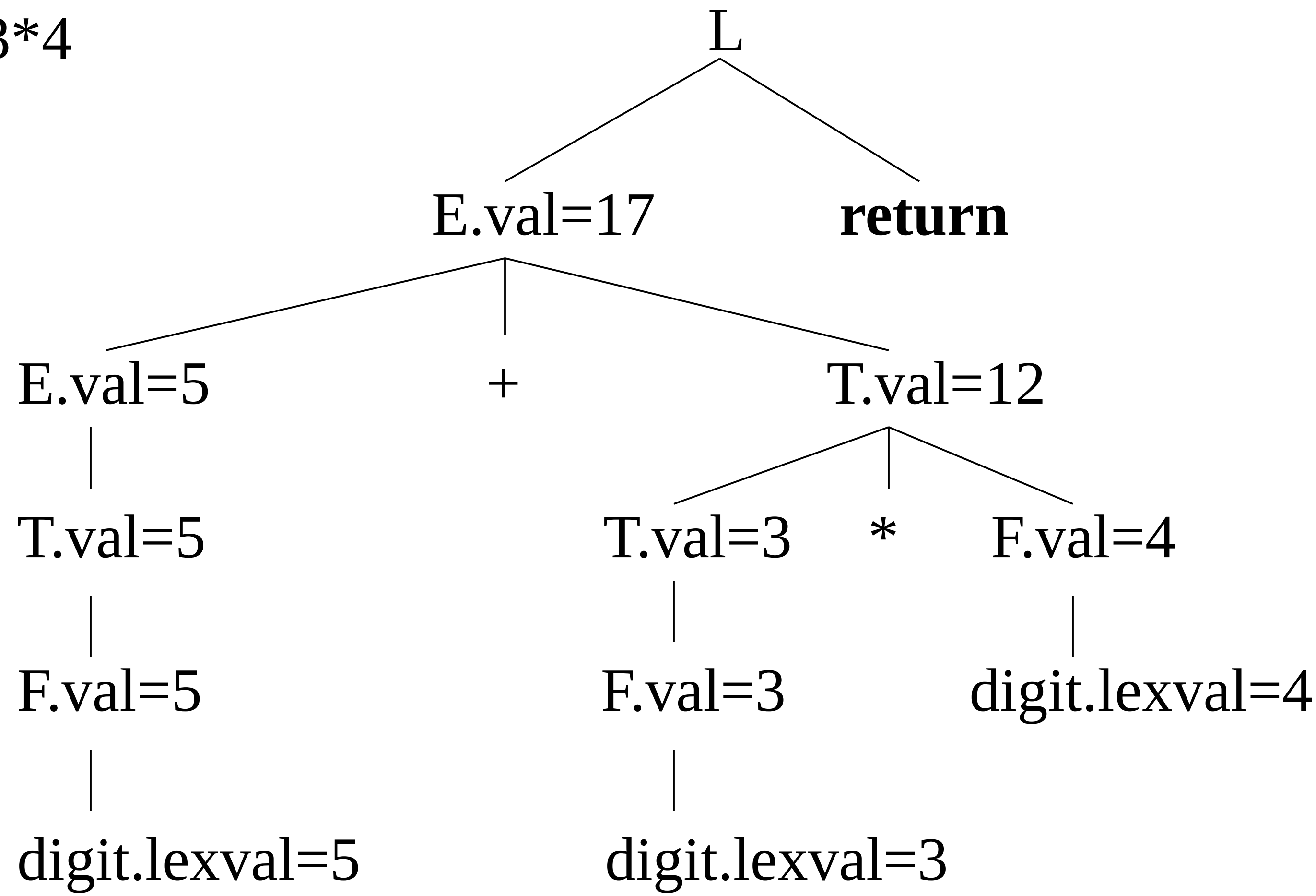
Syntax-Directed Definition -- Example

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

1. Symbols E, T, and F are associated with a synthesized attribute *val*.
2. The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

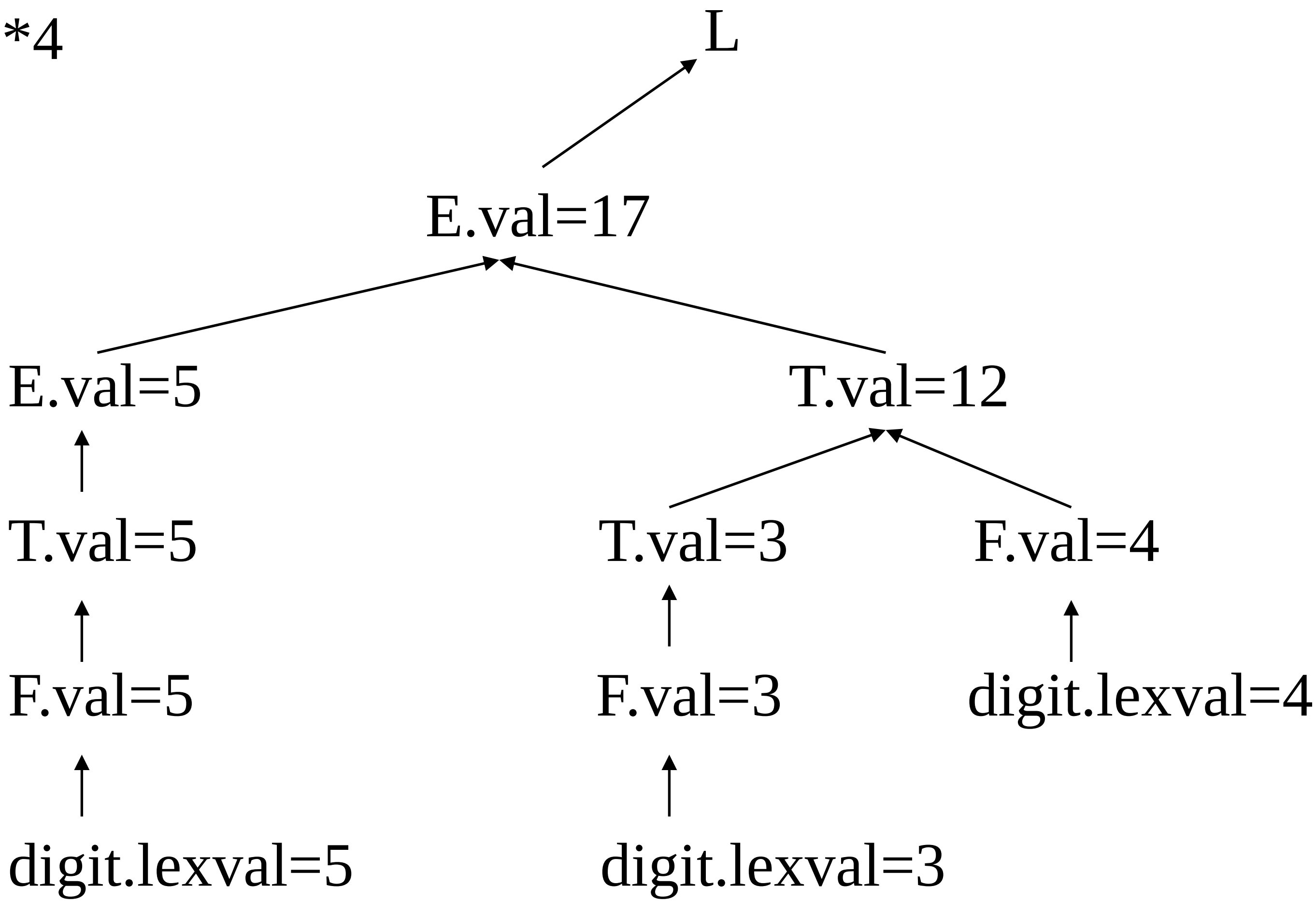
Annotated Parse Tree -- Example

Input: 5+3*4



Dependency Graph

Input: 5+3*4



Dependence Graph

- Let T be a parse tree generated by the CFG of an AG, G.

- The attribute dependence graph for T is the directed graph, $DG(T) = (V,E)$, where
  - $V = \{b \mid b$ is an attribute instance of some tree node$\}$, and
  - $E = \{(b, c) \mid b, c \in V$, b and c are attributes of grammar symbols in the same production p of T and the value of b is used for computing the value of c in an attribute computation rule associated with production $p\}$

Attribute Evaluation Algorithm

Input: A parse tree T with unevaluated attribute instances
Output: T with consistent attribute values

$\{$ Let $(V, E) = DG(T)$;
  Let $W = \{b \mid b \in V \;\&\; indegree(b) = 0\}$;
  while $W \neq \phi$ do
    $\{$ remove some b from W;
      $value(b)$:= value defined by appropriate attribute
          computation rule;
      for all $(b, c) \in E$ do
        $\{$ $indegree(c) := indegree(c) - 1$;
          if $indegree(c) = 0$ then $W := W \cup \{c\}$;
        $\}$
    $\}$
$\}$