University of sulaimani
College of science
Department of Computer Science

Compiler

Semantic Analysis

2024-2025

Mzhda Hiwa

Introduction

- We have learnt how a parser constructs parse trees in the syntax analysis phase.
- This does not carry any information of how to evaluate the tree, as it only shows the syntax rules of the language (using CFG).
- Semantics of a language provide meaning to its constructs. Semantics help interpret symbols, their types, and their relations with each other.
- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

- An important part of semantic analysis is **type checking,** where the compiler checks that each operator has matching operands. For example, many programing language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

- The Semantic Analyzer also check matching if...else statements, A variable is used but hasn't been initialized , Every variable is declared before used, etc.

Semantic Analysis

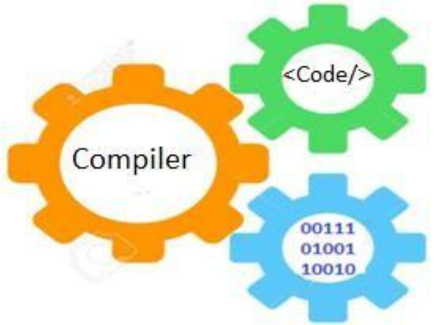
- Semantic analysis checks whether the syntax structure constructed in the source program derives any meaning or not.

For example:

- int x = “digit”; should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the variable differs.

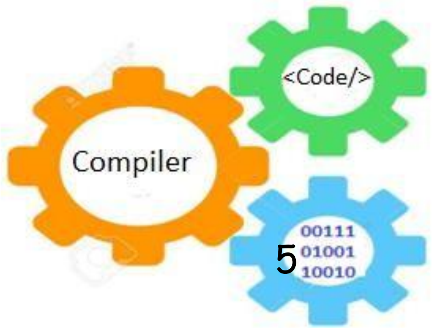
- string x;
  int y;
  y = x + 3;
  the use of x is type error

- int a, b;
  a = b + c;
  c is not declared



Compiler needs to know?

- Whether a variable has been declared?
- What is the type of the variable?
- Whether a variable is a scalar, an array, or a function?
- What declaration of the variable does each reference use?
- If an array use like A[i,j,k] is consistent with the declaration? Does it have three dimensions?

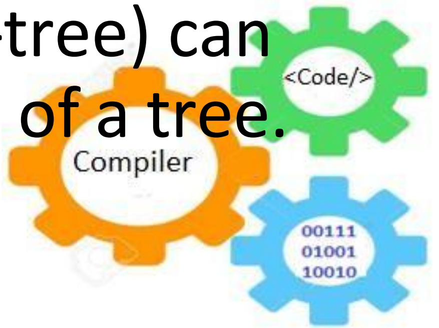


- How many arguments does a function take?
- Are all invocations of a function consistent with the declaration?
- If an operator/function is overloaded, which function is being invoked?
- Inheritance relationship
- Classes not multiply defined
- Methods in a class are not multiply defined
- The exact requirements depend upon the language

Attribute Grammar

- **Attribute grammar** is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.

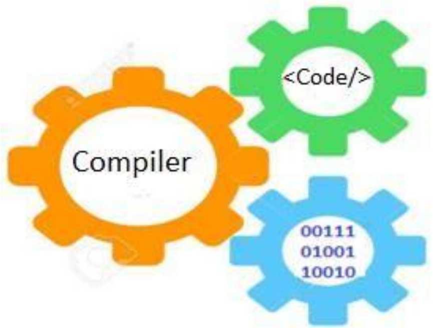
- Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.



Example

- E → E + T { E.value = E.value + T.value }
- The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

- Attributes may be of any kind: numbers, types, table references, or strings, for instance.



Types of Attribute grammar

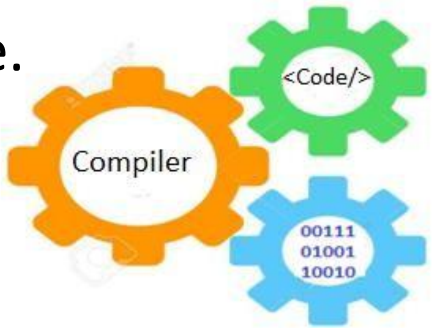
- Attribute grammar can be divided into two categories

1. **Synthesized attributes:**

- These attributes get values from the attribute values of their child nodes.
- Ex: S → ABC

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute.

- Attribute value for the LHS nonterminal is computed from the attribute values of the symbols at the RHS of the rule.



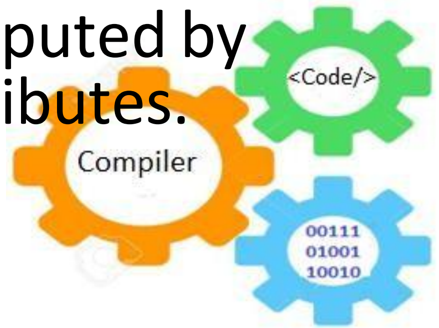
Types of Attribute grammar… cont’d

2. Inherited attributes:

- inherited attributes can take values from parent and/or siblings.
- Attribute value of a RHS nonterminal is computed from the attribute values of the LHS nonterminal and some other RHS nonterminals.
- Ex: S → ABC

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

- Terminals can have synthesized attributes, computed by the lexer (e.g., id.lexeme), but no inherited attributes.



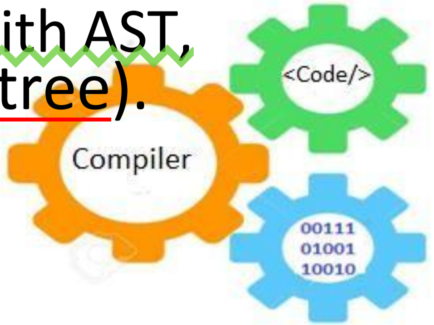
Syntax Directed Translations

- A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. If X is a symbol and a is one of its attributes, then we write X.a to denote the value of a at a particular parse-tree node labeled X.

- CFG + semantic rules = Syntax Directed Definitions

- Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).

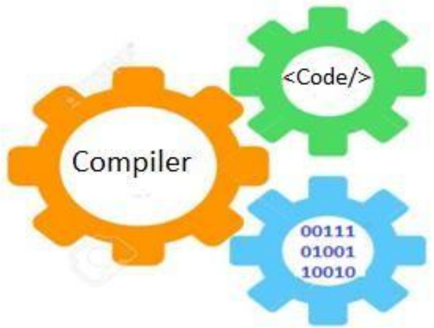
- Semantic analyzer attaches attribute information with AST, which are called Attributed AST (annotated parse tree).



Syntax Directed Translations

CFG + semantic rules = Syntax Directed Definitions

CFG	semantic rules
E → E1+T	{E.value=E1.value+T.value}
E → T	{E.value=T.value}
T → T1*F	{T.value=T1.value*F.value}
T → F	{T.value=F.value}
F → num	{F.value=num.lexvalue}



Parse tree

Input for Semantic phase is (parse tree)

$$\begin{tikzcd}
 & & E \arrow[dll] \arrow[d] \arrow[drr] & & & \\
E \arrow[d] & & + & & T \arrow[dl] \arrow[d] \arrow[dr] & \\
T \arrow[d] & & & T \arrow[d] & * & F \arrow[d] \\
F \arrow[d] & & & F \arrow[d] & & num \\
num & & & num & &
\end{tikzcd}$$

Annotated parse tree for 1+3*2

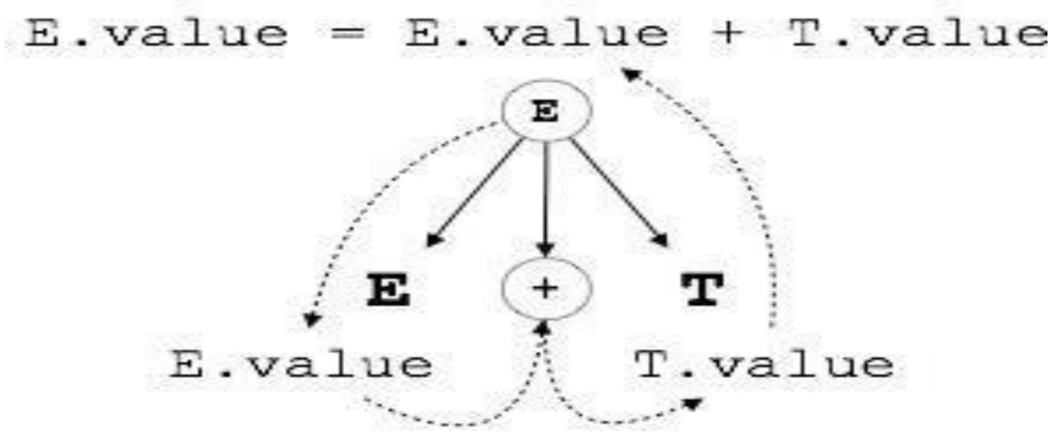
Output Semantic phase is annotated parse tree

$$\begin{tikzcd}
 & & \text{E.value=7} \arrow[dll] \arrow[dl] \arrow[dr] & & \\
\text{E.value=1} \arrow[d] & + & & \text{T.value=6} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{T.value=1} \arrow[d] & & \text{T.value=3} \arrow[d] & * & \text{F.value=2} \arrow[d] \\
\text{F.value=1} \arrow[d] & & \text{F.value=3} \arrow[d] & & \text{Num.lexvalue=2} \\
\text{num.lexvalue=1} & & \text{Num.lexvalue=3} & &
\end{tikzcd}$$

Evaluation of Attributes

- **S-attributed SDT:**

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).

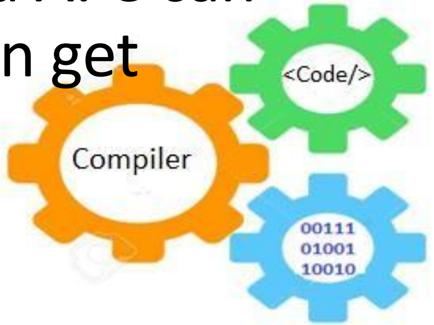


S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

Evaluation of Attributes … cont'd

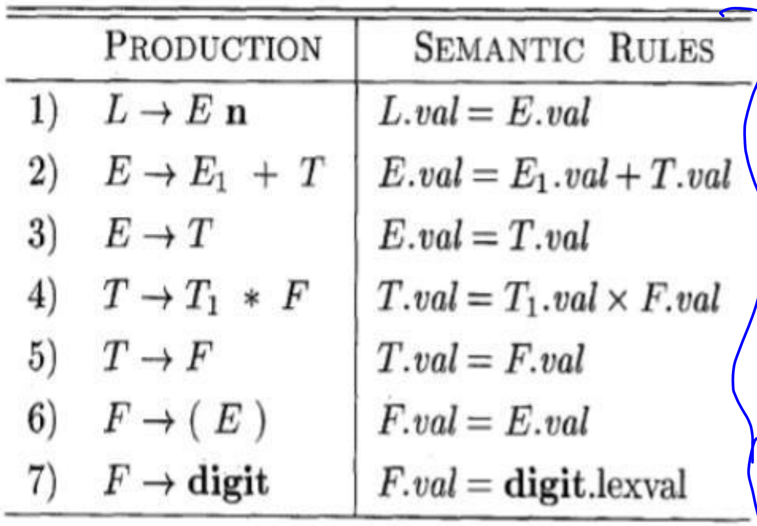
- **L-attributed SDT:**
- This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.
- In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production
- S → ABC
- S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right,

eg



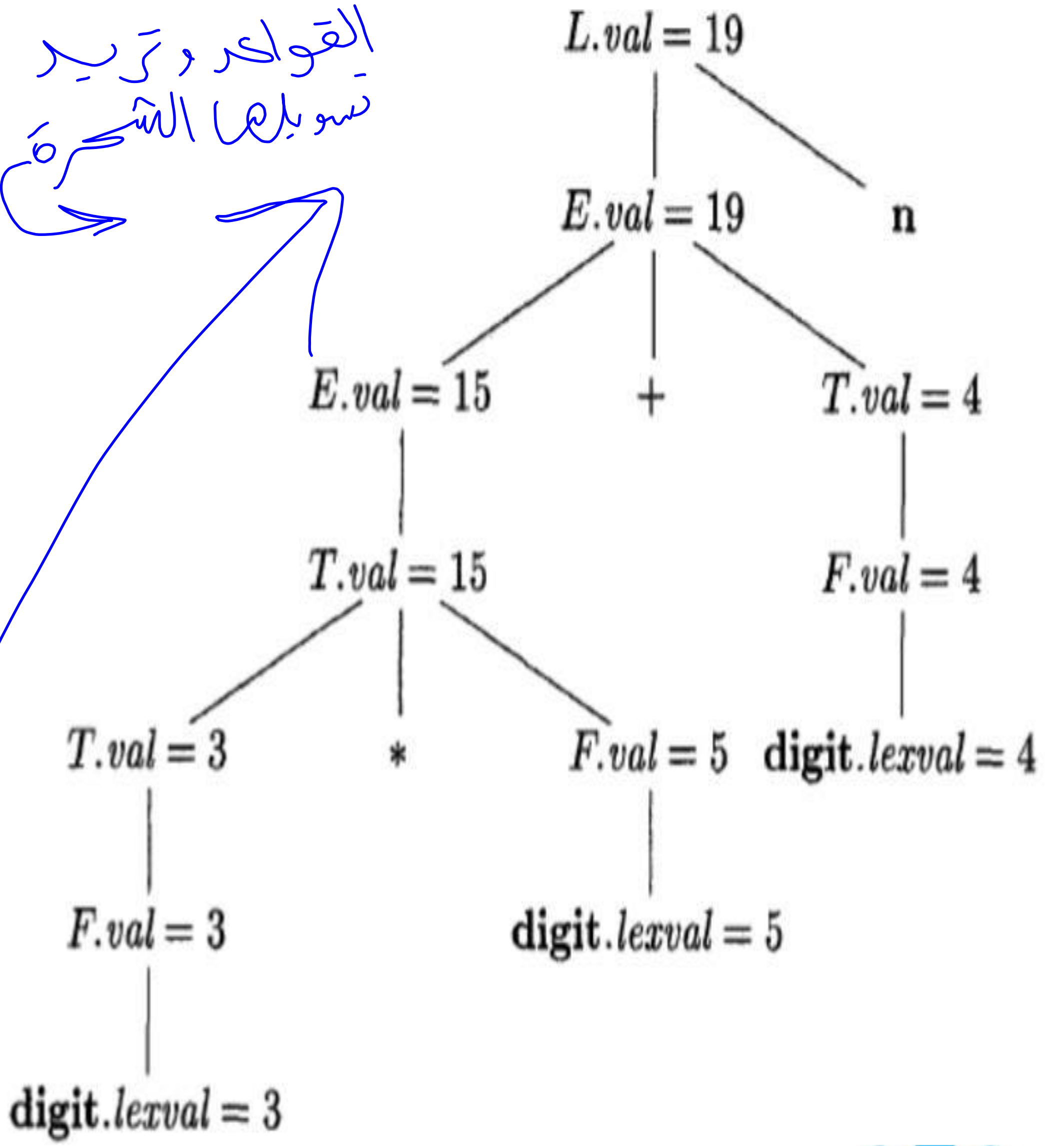
Evaluation of S-attribute (Example)

من حيث هذه القواعد وتريد تمثيلها الشجرة



	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

3*5+4



Evaluation of L-attributed (example)

Productions with semantic rules :

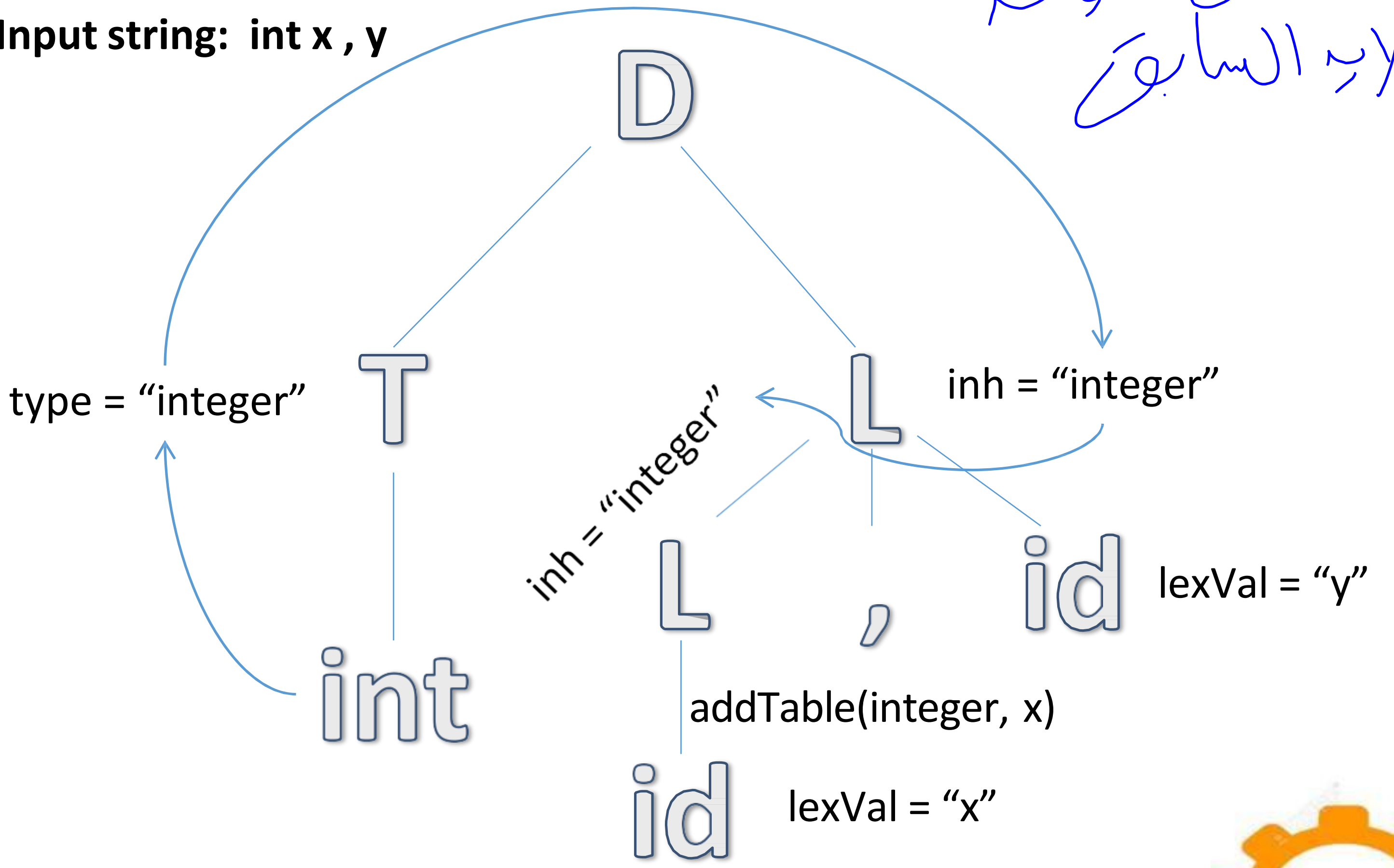
D → T L	L.inh = T.type	
T → int	T.type = integer	
T → float	T.type = float	
L → L1 , id	L_1.inh = L.inh	addTable(L.inh, id.lexVal)
L → id	addTable(L.inh, id.lexVal)	

addTable() function adds the datatype to the identifier

Evaluation of L-attributed (example)

هذا المخطط تم عمله بالاعتماد على القواعد في السلايد السابق

Input string: int x , y



D
T
L
inh = "integer"
type = "integer"
inh = "integer"
int
L
,
id
lexVal = "y"
addTable(integer, x)
id
lexVal = "x"