Semantic Analysis

Role of Semantic Analysis in Compilation

- Semantic Analysis go hand-in-hand with syntax analysis.
- It checks for logical errors that the grammar cannot detect.
- Ensures type correctness, scope resolution, and semantic consistency.
- Uses Attribute Grammars to propagate and verify information.

Type Checking in Expressions

- Type Checking ensures that operations in an expression are performed on compatible data types.
- It helps catch type mismatches before code generation.
- There are two types of type checking:
  - Static Type Checking: Performed at compile time.
  - Dynamic Type Checking: Performed at runtime.

Attribute Grammar for Type Checking

- Attribute Grammar defines rules for assigning and checking types during parsing.
- Uses Synthesized Attributes to propagate type information in the parse tree.
- Ensures type consistency for expressions.

E → E1 + T { E.type = typeCheck(E1.type, T.type) }

T → int { T.type = int }

T → float { T.type = float }

Syntax-Directed Definitions

- Syntax-Directed Definitions (SDD) associate semantic rules with grammar production rules.
- These rules define how attributes (e.g., types, values, scope) are computed.
- Attributes can be:
  - Synthesized Attributes (computed from children).
  - Inherited Attributes (passed from parents or siblings).

Attribute Grammars for Semantic Analysis

- Attribute Grammars define how attributes are propagated in the parse tree.
- There are two types:
- S-attributed grammars (only synthesized attributes, works with bottom-up parsing).
- L-attributed grammars (uses both inherited and synthesized attributes, works with top-down parsing).

Example

Consider the following grammar of signed binary numerals. We wish to translate it to decimal number

$$
\begin{array}{rlll}
0: & S' & \rightarrow & N\$ \\
1: & N & \rightarrow & S\ L \\
2: & S & \rightarrow & + \\
3: & S & \rightarrow & - \\
4: & L & \rightarrow & L\ B \\
5: & L & \rightarrow & B \\
6: & B & \rightarrow & 0 \\
7: & B & \rightarrow & 1
\end{array}
$$

Syntax-Directed Definition (SDD)

0 : S′ → N	print N.val
1 : N → S L	if (S.sign == ‘-’) N.val= - L.val; else N.val = L.val;
2 : S → +	S.sign = ‘+’;
3 : S → −	S.sign = ‘-’;
4 : L → L1 B	L.val = 2 * L1.val + B.val;
5 : L → B	L.val = B.val;
6 : B → 0	B.val = 0;
7 : B → 1	B.val = 1;

Synthesized Attribute

- In this example the value of an attribute of a non-terminal is either coming from the attributes of its children.
- This type of attribute is known as a synthesized attribute.
- An attributed grammar is called S-attributed if every attribute is synthesized.

$0 : S' \rightarrow N$	print N.val	
$1 : N \rightarrow S\ L$	L.pos = 0	if (S.sign == '-') N.val = -L.val; else N.val = L.val;
$2 : S \rightarrow +$	S.sign = '+';	
$3 : S \rightarrow -$	S.sign = '-';	
$4 : L \rightarrow L1\ B$	L1.pos = L.pos + 1; L.val = L1.val + B.val;	B.pos = L.pos;
$5 : L \rightarrow B$	B.pos = L.pos;	L.val = B.val;
$6 : B \rightarrow 0$	B.val = 0;	
$7 : B \rightarrow 1$	B.val = $2^{B.pos}$;	

Inherited Attribute

- Let B be a non-terminal of a parse tree node.
- An inherited attribute B.i is defined in terms of the attributes of the parent and sibling nodes of B.
- In the previous example the non-terminal L gets the attribute from T as an inherited attribute.

L-Attributed Definitions

An SDD is called L-attributed (‘L’ for left) if each attribute is either synthesized, or inherited with the following restrictions

S-Attributed Expression Grammar

S → E$ { print E.val }

E → E1 + T { E.val = E1.val + T.val }

E → T { E.val = T.val }

T → T1 * F { T.val = T1.val * F.val }

T → F { T.val = F.val }

F → (E) { F.val = E.val }

F → id { F.val = id.val }