Semantic Analysis

Semantic analysis, expressed, is the process of extracting meaning from text. Grammatical analysis and the recognition of links between specific words in a given context enable computers to comprehend and interpret phrases, paragraphs, or even entire manuscripts.

It is a crucial component of Natural Language Processing (NLP) and the inspiration for applications like chatbots, search engines, and text analysis using machine learning.

Attribute grammar

An **attribute grammar** is a formal way to supplement a formal grammar with semantic information processing. Semantic information is stored in attributes associated with terminal and nonterminal symbols of the grammar. The values of attributes are result of attribute evaluation rules associated with productions of the grammar. Attributes allow to transfer information from anywhere in the abstract syntax tree to anywhere else, in a controlled and formal way.[1]

Each semantic function deals with attributes of symbols occurring only in one production rule: both semantic function parameters and its result are attributes of symbols from one particular rule. When a semantic function defines the value of an attribute of the symbol on the left hand side of the rule, the attribute is called *synthesized*; otherwise it is called *inherited*.[2] Thus, synthesized attributes serve to pass semantic information up the parse tree, while inherited attributes allow values to be passed from the parent nodes down and across the syntax tree.

In simple applications, such as evaluation of arithmetic expressions, attribute grammar may be used to describe the entire task to be performed besides parsing in straightforward way; in complicated systems, for instance, when constructing a language translation tool, such as a compiler, it may be used to validate semantic checks associated with a grammar, representing the rules of a language not explicitly imparted by the syntax definition. It may be also used by parsers or compilers to translate the syntax tree directly into code for some specific machine, or into some intermediate language.

Example[edit]

The following is a simple context-free grammar which can describe a language made up of multiplication and addition of integers.

Expr → **Expr** + **Term**
Expr → **Term**
Term → **Term** * **Factor**
Term → **Factor**
Factor → "(" **Expr** ")"
Factor → *integer*

The following attribute grammar can be used to calculate the result of an expression written in the grammar. Note that this grammar only uses synthesized values, and is therefore an S-attributed grammar.

$\mathbf{Expr}_1$ → $\mathbf{Expr}_2$ + **Term** [$\mathbf{Expr}_1$.value = $\mathbf{Expr}_2$.value + **Term**.value]
Expr → **Term** [**Expr**.value = **Term**.value]

Term$_1$ → **Term**$_2$ * **Factor** [**Term**$_1$.value = **Term**$_2$.value * **Factor**.value]
Term → **Factor** [**Term**.value = **Factor**.value]
Factor → "(" **Expr** ")" [**Factor**.value = **Expr**.value]
Factor → *integer* [**Factor**.value = strToInt(*integer*.str)]

Special types of attribute grammars[edit]

- L-attributed grammar: *inherited attributes* can be evaluated in one left-to-right traversal of the abstract syntax tree
- LR-attributed grammar: an L-attributed grammar whose *inherited attributes* can also be evaluated in bottom-up parsing.
- ECLR-attributed grammar: a subset of LR-attributed grammars where equivalence classes can be used to optimize the evaluation of inherited attributes.
- S-attributed grammar: a simple type of attribute grammar, using only *synthesized attributes*, but no *inherited attributes*

Syntax Directed Definition

Syntax Directed Definition (SDD) is a kind of abstract specification. It is generalization of context free grammar in which each grammar production **X –> a** is associated with it a set of production rules of the form s = f(b_1, b_2, ……b_k) where s is the attribute obtained from function f. The attribute can be a string, number, type or a memory location. Semantic rules are fragments of code which are embedded usually at the end of production and enclosed in curly braces ({ }).

Example:
E --> E_1 + T { E.val = E_1.val + T.val}
Annotated Parse Tree – The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.
Features –

- High level specification
- Hides implementation details
- Explicit order of evaluation is not specified

Types of attributes – There are two types of attributes:
1. Synthesized Attributes – These are those attributes which derive their values from their children nodes i.e. value of synthesized attribute at node is computed from the values of attributes at children nodes in parse tree.
Example:
E --> E_1 + T { E.val = E_1.val + T.val}
In this, E.val derive its values from E_1.val and T.val
Computation of Synthesized Attributes –

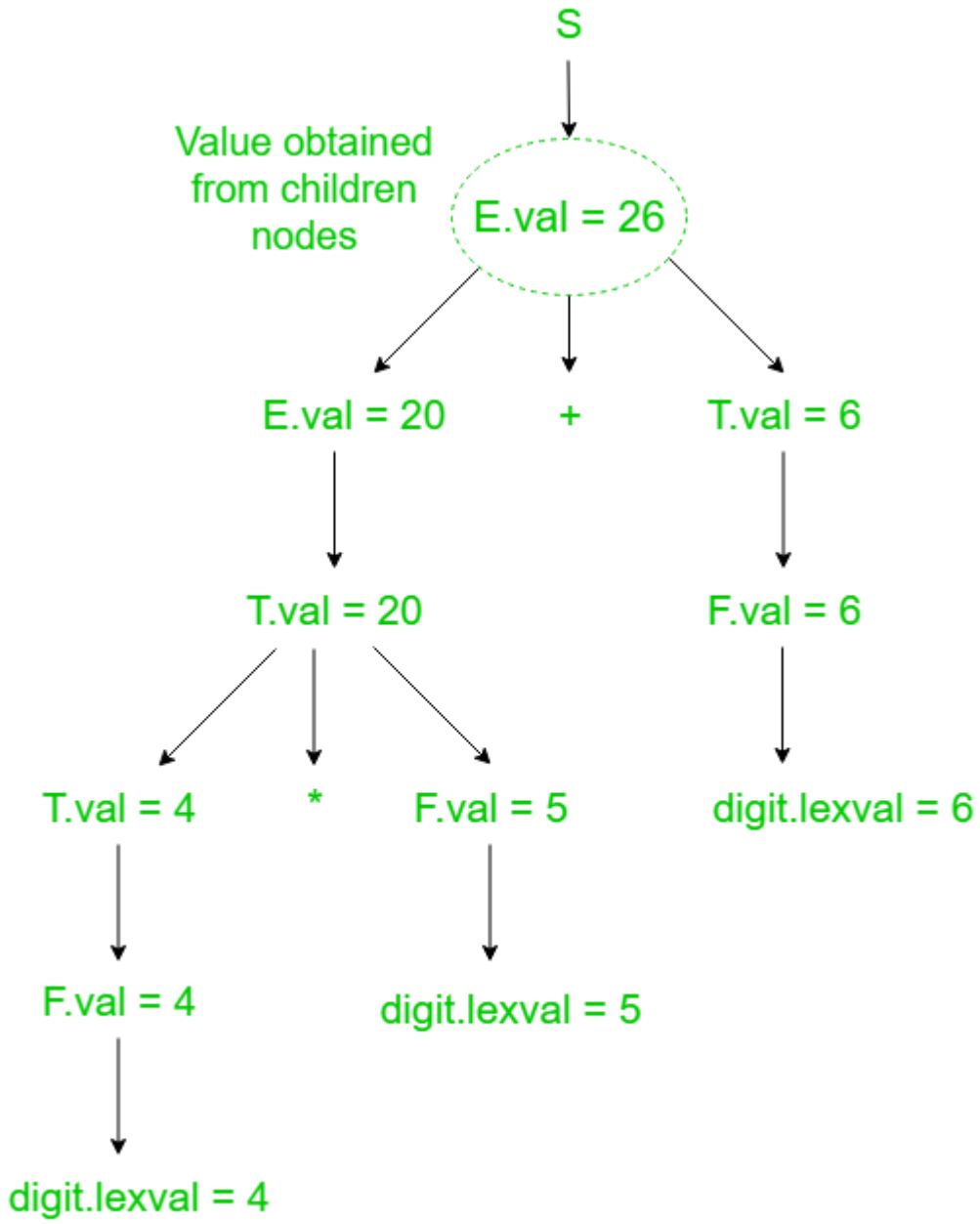
- Write the SDD using appropriate semantic rules for each production in given grammar.
- The annotated parse tree is generated and attribute values are computed in bottom up manner.
- The value obtained at root node is the final output.

Example: Consider the following grammar
S --> E

E --> E_1 + T
E --> T
T --> T_1 * F
T --> F
F --> digit
The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> E	Print(E.val)
E --> E_1 + T	E.val = E_1.val + T.val
E --> T	E.val = T.val
T --> T_1 * F	T.val = T_1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval

Let us assume an input string **4 * 5 + 6** for computing synthesized attributes. The annotated parse tree for the input string is



Annotated Parse Tree

For computation of attributes we start from leftmost bottom node. The rule F –> digit is used to reduce digit to F and the value of digit is obtained from lexical analyzer which becomes value of F i.e. from semantic action F.val = digit.lexval. Hence, F.val = 4 and since T is parent node of F so, we get T.val = 4 from semantic action T.val = F.val. Then, for T –> T_1 * F production, the corresponding semantic action is T.val = T_1.val * F.val . Hence, T.val = 4 * 5 = 20

Similarly, combination of E_1.val + T.val becomes E.val i.e. E.val = E_1.val + T.val = 26. Then, the production S –> E is applied to reduce E.val = 26 and semantic action associated with it prints the result E.val . Hence, the output will be 26.

2. Inherited Attributes – These are the attributes which derive their values from their parent or sibling nodes i.e. value of inherited attributes are computed by value of parent or sibling nodes.

Example:

A --> BCD { C.in = A.in, C.type = B.type }

Computation of Inherited Attributes –

- Construct the SDD using semantic actions.
- The annotated parse tree is generated and attribute values are computed in top down manner.

Example: Consider the following grammar

S --> T L

T --> int

T --> float

T --> double

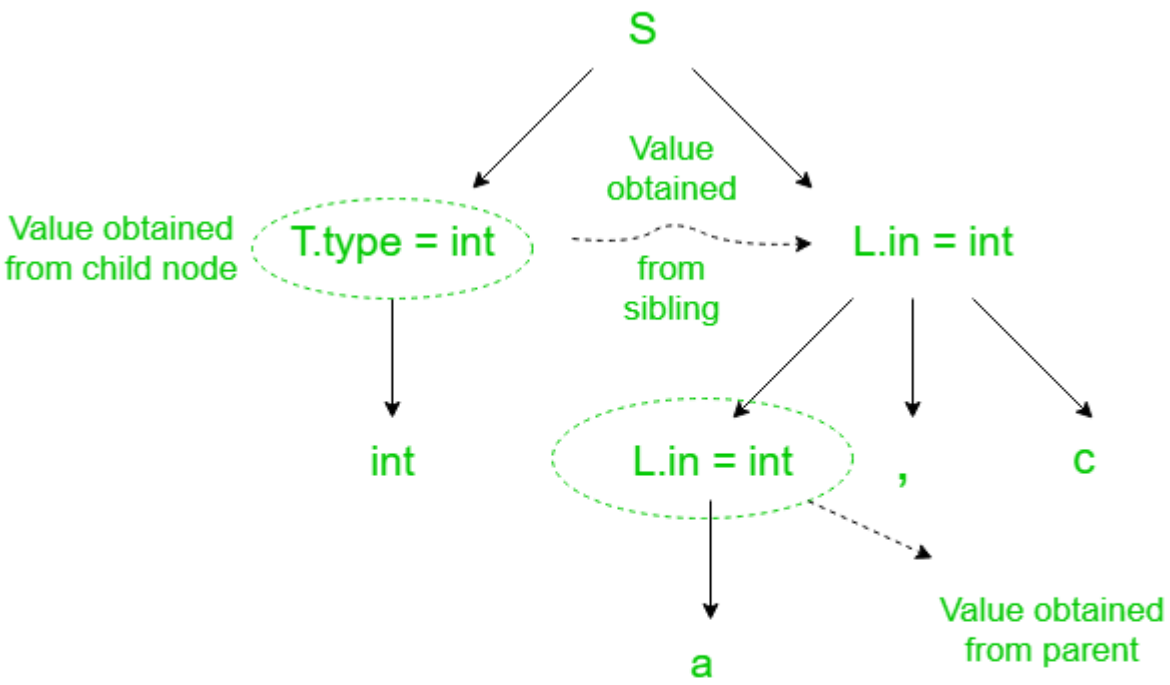
L --> L_1, id

L --> id

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> T L	L.in = T.type
T --> int	T.type = int
T --> float	T.type = float
T --> double	T.type = double
L --> L_1 , id	L_1.in = L.in Enter_type(id.entry , L.in)
L --> id	Entry_type(id.entry , L.in)

Let us assume an input string **int a, c** for computing inherited attributes. The annotated parse tree for the input string is



Annotated Parse Tree

The value of L nodes is obtained from T.type (sibling) which is basically lexical value obtained as int, float or double. Then L node gives type of identifiers a and c. The computation of type is done in top down manner or preorder traversal. Using function Enter_type the type of identifiers a and c is inserted in symbol table at corresponding id.entry.

Synthesized and Inherited Attributes

In Syntax Directed Definition, two attributes are used one is Synthesized attribute and another is inherited attribute. An attribute is said to be **Synthesized attribute** if its parse tree node value is determined by the attribute value at child nodes whereas An attribute is said to be **Inherited attribute** if its parse tree node value is determined by the attribute value at parent and/or siblings node. Now, we shall see the comparison between Synthesized Attributes and Inherited Attributes.

Synthesized Attributes

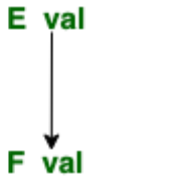
EX:-

E.val -> F.val

Inherited Attributes

EX:-

E.val = F.val



Dependency graph

In mathematics, computer science and digital electronics, a **dependency graph** is a directed graph representing dependencies of several objects towards each other. It is possible to derive an evaluation order or the absence of an evaluation order that respects the given dependencies from the dependency graph

Given a set of objects S and a transitive relation R ⊆ S x S with (a,b) ∈ R modeling a dependency "*a* depends on *b*" ("*a* needs *b* evaluated first"), the dependency graph is a graph G= (S,T) with T ⊆ R the transitive reduction of *R*.

For example, assume a simple calculator. This calculator supports assignment of constant values to variables and assigning the sum of exactly two variables to a third variable. Given several equations like "*A* = *B*+*C*; *B* = 5+*D*; *C*=4; *D*=2;", then S={A,B,C,D} and R={(A,B), (A,C), (B,D)}. You can derive this relation directly: *A* depends on *B* and *C*, because you can add two variables if and only if you know the values of both variables. Thus, *B* must be calculated before *A* can be calculated. However, the values of *C* and *D* are known immediately, because they are number literals.

A dependency graph is used to represent the flow of information among the attributes in a parse tree. In a parse tree, a dependency graph basically helps to determine the evaluation order for the attributes. The main aim of the dependency graphs is to help the compiler to check for various types of dependencies between statements in order to prevent them from being executed in the incorrect sequence, i.e. in a way that affects the program's meaning. This is the main aspect that helps in identifying the program's numerous parallelizable components.

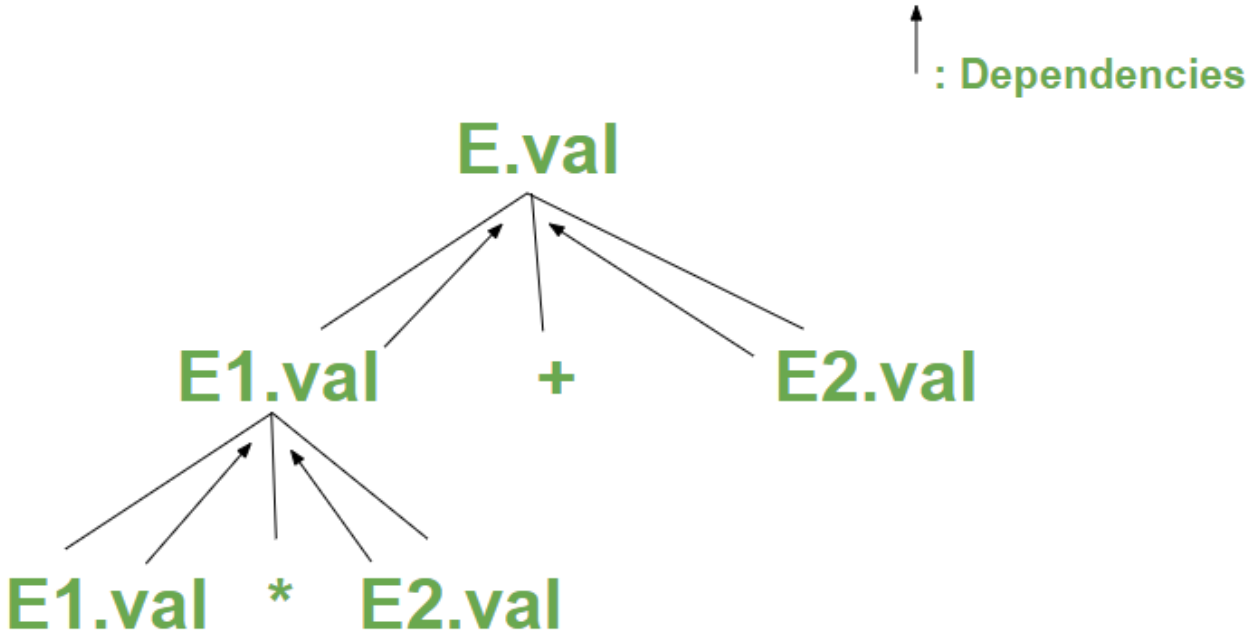
Example of Dependency Graph:

Design dependency graph for the following grammar:

```
E -> E1 + E2
E -> E1 * E2
```

PRODUCTIONS	SEMANTIC RULES
E -> E1 + E2	E.val -> E1.val + E2.val
E -> E1 * E2	E.val -> E1.val * E2.val

Required dependency graph for the above grammar is represented as –



Dependency Graph for the above example

1. Synthesized attributes are represented by **.val**.
2. Hence, **E.val**, **E1.val**, and **E2.val** have synthesized attributes.
3. Dependencies are shown by black arrows.
4. Arrows from E1 and E2 show that the value of E depends upon E1 and E2.

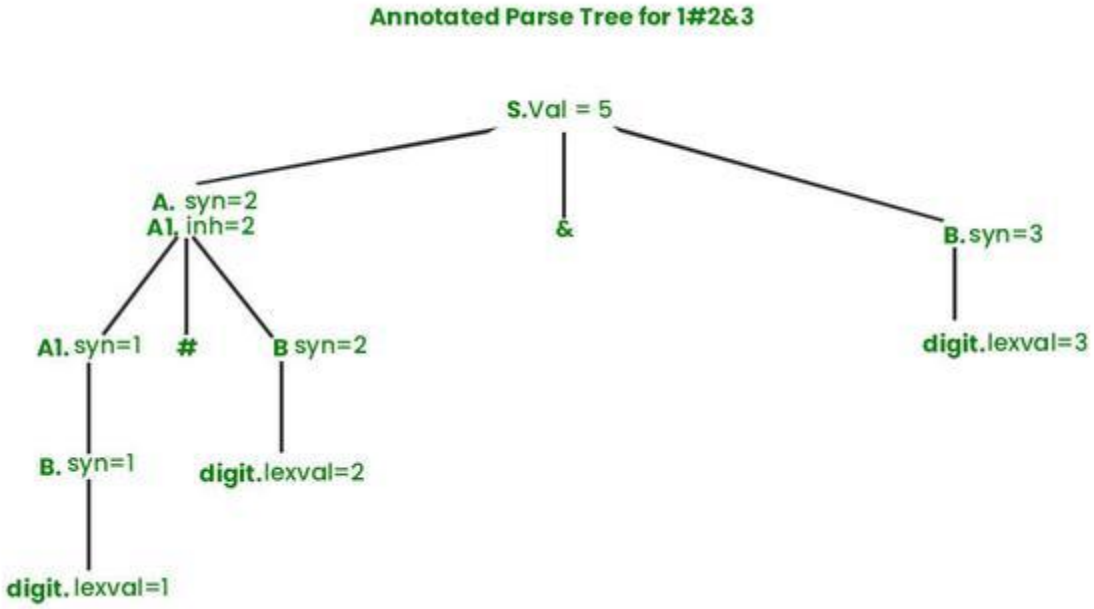
Ordering the Evaluation of Attributes

The dependency graph describes the numerous ways in which the properties at the various nodes of a parse tree can be evaluated. If there happens to be an edge from node M to node N in the dependency graph, the attribute corresponding to M must be assessed before the attribute corresponding to N. As a result, the only valid evaluation orders are those with nodes N_1, N_2, and N_3,...N_k is such that if there is a dependency graph edge from N_i to N_j, then i<j. A topological sort of graph is an ordering that embeds a directed graph into a linear order.
There are no topological sorts if the graph contains a cycle; that is, there is no way to assess the SDD on this parse tree. However, even if there are no cycles, there is always at least one topological sort. To illustrate why we can quickly discover a node with no edge entering because there are no cycles. If there was no such node, we might go from predecessor to predecessor until we reached a node we'd seen before, resulting in a cycle. Remove the node from the dependency network, make it the first in the topological order, and continue with the other nodes.

topological order.
There is no way to evaluate SDD on a parse tree when there is a cycle present in the graph and due to the cycle, no topological order exists.

Production Table

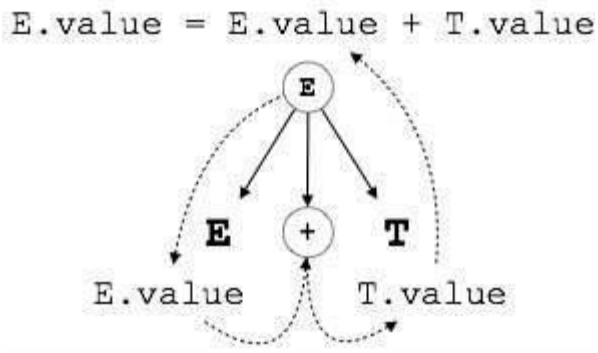
S.No.	Productions	Semantic Rules
1.	S ⇢ A & B	S.val = A.syn + B.syn
2.	A ⇢ A_1 # B	A.syn = A1.syn * B.syn A1.inh = A.syn
3.	A1 ⇢ B	A1.syn = B.syn
4.	B ⇢ digit	B.syn = digit.lexval



Annotated Parse Tree For 1#2&3

S-attributed and L-attributed Definitions

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).

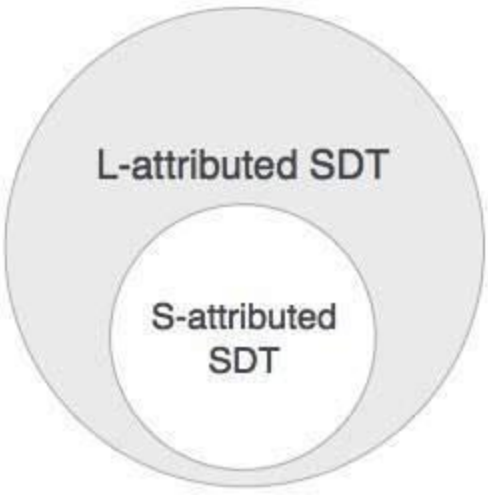


S-attributed SDT As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

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.



Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

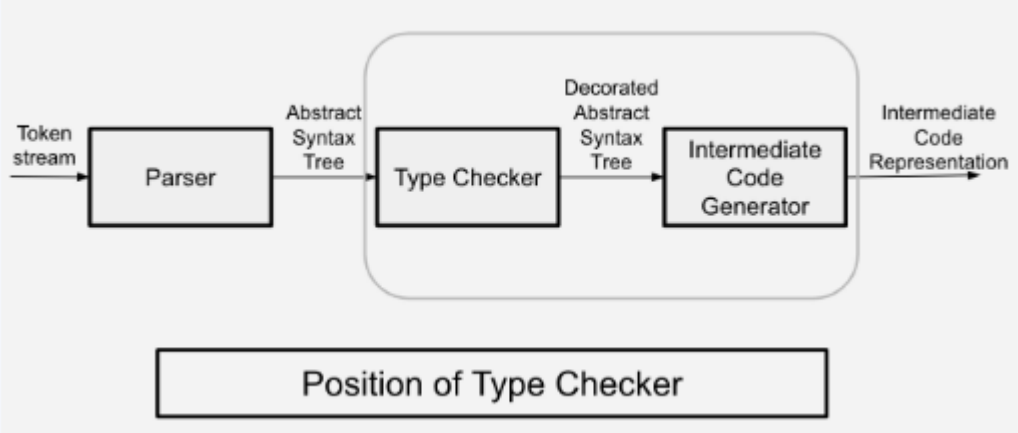
L-attributed SDT We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

Type Checking

Type checking or static checking is performed by the compiler (checking is done at the compiler time). Specific forms of programming faults will be recognized and reported as a result of this.

The technique of verifying and enforcing type restrictions in values is type checking. Many texts are filtered out by the compiler's Lexical Analysis and parsing phases. Still, these techniques cannot handle many programming languages with well-formed needs since they are frequently not context-free and cannot be checked by the membership of a context-free grammar.

The type-checking Phases of Compiler design is interleaved with the syntax analysis phase, so it is done before a program's execution or translation (static typing), and the information is gathered for use by subsequent phases, such as the translator, which will naturally combine type calculation with the actual translation.



A type checker ensures that a construct's type matches the type expected by its context.
For example, in Pascal, the arithmetic operator mod requires integer operands; hence a type checker ensures that the operands of the mod are of type integer.
When the code is generated, type information acquired by a type checker may be required.

Dynamic typing vs. static typing

This topic is provided for reverence only as it explains the differences between dynamic and static typing. Understanding the differences between dynamic and static typing is key to understanding the way in which transformation script errors are handled, and

how it is different from the way Groovy handles errors. This will also help you interpret errors created by your transformation script.

Note: It is important to know that the Groovy implementation within Big Data Discovery enforces static typing. For information on exception handling in **Transform**, which uses a static parser overriding Groovy's dynamic typing behavior, see Exception handling and troubleshooting your scripts.

There are two main differences between dynamic typing and static typing that you should be aware of when writing transformation scripts.

First, dynamically-typed languages perform type checking at runtime, while statically typed languages perform type checking at compile time. This means that scripts written in dynamically-typed languages (like Groovy) can compile even if they contain errors that will prevent the script from running properly (if at all). If a script written in a statically-typed language (such as Java) contains errors, it will fail to compile until the errors have been fixed.

Second, statically-typed languages require you to declare the data types of your variables before you use them, while dynamically-typed languages do not. Consider the two following code examples:

```
// Java example

int num;

num = 5;

// Groovy example

num = 5
```

Both examples do the same thing: create a variable called num and assign it the value 5. The difference lies in the first line of the Java example, int num;, which defines num's data type as int. Java is statically-typed, so it expects its variables to be declared before they can be assigned values. Groovy is dynamically-typed and determines its variables' data types based on their values, so this line is not required.

Dynamically-typed languages are more flexible and can save you time and space when writing scripts. However, this can lead to issues at runtime. For example:

```
// Groovy example

number = 5

numbr = (number + 15) / 2  // note the typo
```

The code above should create the variable number with a value of 5, then change its value to 10 by adding 15 to it and dividing it by 2. However, number is misspelled at the beginning of the second line. Because Groovy does not require you to declare your variables, it creates a new variable called numbr and assigns it the value number should have. This code will compile just fine, but may produce an error later on when the script tries to do something with number assuming its value is 10.