Lesson 12: Semantic Analysis and Syntax Directed Definitions

# Introduction to Semantic Analysis

By now we know the front-end analysis pipeline of a compiler consists of three stages. The first, lexical analysis, stage returns the tokens (identifiers, numbers, keywords, operators, delimiters, etc.) from sequence of characters under the command of the second, syntax analysis, stage. This second stage uses a context free grammar to describe the syntax of the programming language and construct a parsing table from that grammar. The syntax analysis is done by looking up that table and using a parsing stack. The final analysis stage is the semantic analysis. Semantic means ‘meaning.’

To understand what role the semantic analysis stage plays, consider interpretation of a natural language essay. If someone does the lexical analysis and syntax analysis of the essay then the outcome, he/she deduces is basically a statement that says whether the essay has spelling and grammar errors. However, if we ask the reader ‘what does the essay says?’ then he/she she is clueless. Semantic analysis helps him/her to answer such questions.

However, we should be careful about the distinction of semantic analysis of a human speech/text by another human and that of a high-level program by the compiler. In real-world context, a reader understands the semantics means he/she can give an elevator speech about what he/she just read. In the computer program interpreting context, semantic analysis only means that the compiler understands the interpretation of the elements of the program to sufficient extent so that it can generate an executable code. How to code is going to behave when it runs and what output it will produce is beyond the scope of the compiler.

So, what the compiler needs to understand about the program that would allow it to generate an executable code? Remember that the power of the computer as a machine is quite limited. A computer can do basic arithmetic logic operations on CPU registers, load/store data between CPU and memory, and jump from one elementary instruction to another instruction in the executable. That is all of it. However, the code we write using a high-level language includes many and different types of variables, program structuring blocks such as branching and loop, function calls, etc. To translate these high-level constructs to elementary operations and data that the computer can execute, the compiler needs to determine how to store the various variables, what are their sizes, what statements should be skipped/executed in case of branching and iterations, how to and what parameters to pass to a function call and how to extract function results and return to the caller functions, among many other things.

Semantic analysis helps the compiler to discover necessary information and decide on the above matters. Now this information and decisions must be stored in an intermediate form so that the synthesis phase of the compiler that ultimately generates the code can use it. Typically, the information is stored using a combination of two data structures. The first is called a symbol table. A symbol table contains information about the variables, functions, and user defined types (i.e., classes) found in the program. We will learn in detail about symbol table and their constructions later. The second data structure is called the parse tree. We already learned during syntax analysis discussion that a parse tree shows how an input sentence can be derived from the start symbol of the grammar. However, just syntax analysis on its own does not create the parse tree, it only constructs the parsing table and validates the sentence using the parsing stack. For the sake of semantic analysis, a parse tree is constructed then additional information get stored in the nodes of the parse tree. To give a concrete example, consider the partial arithmetic expression grammar as follows:

E -> E + T | T

T -> T \* F | F

F -> id | number | (E)

Figure 1: Partial Arithmetic Expression Grammar

Then the parse tree for input sentence x + y \* 10.5 would be as follows:

**E**

E + **T**

T T \* F

F F number

id id

Figure 2: The parse tree for x + y \* 10.5

The semantic analysis phase may store several information associated with this parse tree. For example, for both bolded E and T, it needs to store that the result of arithmetic will be fractional numbers. So, it will choose an addition and multiplication operations that work with fractional numbers, not with integers. Then for the bolded T, it will note that a temporary variable needs to be created that will hold the result of the multiplication to be used later in the addition. For more complex constructs such as branching, loops, and functions, many more information is stored in the parse tree nodes. As after syntax analysis, the parse tree is not a mere representation of derivation of the sentence from the start symbol, rather a heavy information container, it is rather called an **abstract syntax tree (ADT)**, as opposed to the parse tree.

In both cases of symbol table and abstract syntax tree, information is stored as attributes of the symbol table entries and syntax tree nodes. Typically, there is a complex object class so that different types of attributes for different features can all be accommodated. Another common solution is to store each attribute as a key value pair. That is the symbol table entries and abstract syntax tree nodes hold a map/dictionary of attributes.

# Introduction to Syntax Directed Translation

Now let us go back to the analogy of interpreting a human speech/essay to understand the notion of syntax directed translation. When we read an essay, we do not read it entirely for grammatical accuracy, devoid of meaning, then read it again now confident that there is no syntax error. Rather, as we read each sentence in the essay, or even sentence parts (i.e., a clause or phrase) we accumulate meaning. Syntax directed translation has the same philosophy. The idea is, collect information about the program components as we do syntax analysis. Now the question is, how it is done. To understand that, we first need to understand the nature of the attributes that we add as information in the symbol table and parse tree nodes.

The attributes compiler collects for semantic analysis falls in two categories:

1. **Synthesized attribute:** if an attribute of a parse tree node is determined from the attributes of its children, then it is a synthesized attribute.
2. **Inherited Attribute:** if an attribute of a parse tree node is determined from its parent, then it is an inherited attribute.

Now if, we only generate synthesized attribute values during syntax analysis then collecting them during bottom-up parsing process is very easy. Remember that, in bottom-up parsing, we convert a production body to its head when applying a reduction. The synthesized attribute for the head, which is the parent of the nodes representing the production body in the parse tree can be generated from the attributes of the terminals and non-terminals forming the production body. How do we get the attributes of the terminals and non-terminals in the production body in this case? Well, the non-terminals in the production body have been generated by some earlier reductions, where we applied the same synthesis process. For terminal, attributes of terminals are already available from lexical analysis (for example, the actual variable name corresponding to an ID and the actual value of a number) and need not be derived.

In fact, the parse tree construction process itself, is an exercise of generating synthesized attributes. Consider using the following reduction during an LR parsing:

**T** -> T \* F.

When we apply the reduction, we already have a node for the F and T in the production body. We create a new node for the T on the head by simply as follows.

**new** Node(**T**)

Node(T) **new** Leaf-Node(\*) Node(F)

Figure 3: Node arrangement for the part of the parse tree for production T -> T \* F

That is we create a new leaf node for the terminal ‘\*’ and a new node for the non-terminal at the head of the production, then add the children node to the parent in the order they appear left-to-right in the production body. In this manner, the whole parse tree will be constructed in a bottom-up fashion and eventually the node for the start symbol will represent the root of that tree.

Once we have the parse tree available, inherited attributes for the parse tree nodes and symbol table entries can be easily collected by traversing the parse tree in pre-order and passing necessary information from parents to their children node. This is the most generic approach and it always work. However, there are ways of avoiding parse tree construction and still being able to generate all synthesized and inherited attributes that we will learn later.

# Specification of Syntax Directed Translation

The way typical parser generator such as BISON/YACC work is that you can specify one or more semantic rules for attribute generation next to the grammar production. This style is called syntax directed definition. For example, consider the syntax directed definition for parse tree construction for the partial arithmetic expression of Figure 1. Here color coding is used to distinguish between the non-terminals appeared on both side of a production.

|  |  |
| --- | --- |
| **Grammar Rule** | **Semantic Rule** |
| E -> E + T | E.node = new Node()  E.node.children = [E.node, new LeafNode(‘+’), T.node] |
| E -> T | E.node = T.node |
| T -> T \* F | T.node = new Node()  T.node.children = [T.node, new LeafNode(‘\*’), F.node] |
| T -> F | T.node = F.node |
| F -> id | F.node = new LeafNode(name: id.lexeme) |
| F -> number | F.node = new LeafNode(value: number.lexical\_value) |
| F -> (E) | F.node = E.node |

When a production is used for a reduction operation then corresponding semantic rule gets executed and a node is created for or assigned to a non-terminal. In the next reduction, the earlier nodes associated with the non-terminals get used to assign it to the parent from a child (as in case of F -> (E)) or to create new node for the parent (as in case of E -> E + T). Try to convince yourself that you can generate any parse tree for any arithmetic expression using these semantic rules.

The syntax directed translation is a more elegant process that does not separate the semantic rules from the grammar rules, rather mixes them together in the following format:

E -> E + T { E.node = new Node(); E.node.children = [E.node, new LeafNode(‘+’), T.node]}

E -> T { E.node = T.node }

T -> T \* F { T.node = new Node(); T.node.children = [T.node, new LeafNode(‘\*’), F.node]}

T -> F {T.node = F.node}

F -> id { F.node = new LeafNode(name: id.lexeme)}

F -> number { F.node = new LeafNode(value: number.lexical\_value)}

F -> (E) { F.node = E.node }

You may not see any different between syntax directed definitions (SDDs) and syntax directed translations (SDTs) in this simple case. However, the advantage of SDT is that the semantic rule inside the curly braces is considered part of the production rule and goes inside the LR automaton. As a result, it is possible to evaluate attributes not only during the reduction operations but also during a shift operation in LR parsing. The general idea is that, if we have an LR(0) item in a state as follows:

A -> α **.**B {semantic rule 1} β {semantic rule 2}

And make a transition to another state on B and having the item below:

A -> α B**.** {semantic rule 1} β {semantic rule 2}

Then the parser immediately applies semantic rule # 1. Semantic rule #2 is applied as usual when we apply a reduction from a state having the item.

A -> α B {semantic rule 1} β**.** {semantic rule 2}

Therefore, SDTs allow a faster processing of attributes than SDDs.