Compiler Deisgn Unit 1,2

- **Cross Compiler** that runs on a machine ‘A’ and produces a code for another machine ‘B’. It is capable of creating code for a platform other than the one on which the compiler is running.
- **Source-to-source Compiler** or transcompiler or transpiler is a compiler that translates source code written in one programming language into source code of another programming language.
- **Assembler –** For every platform (Hardware + OS) we will have a assembler. They are not universal since for each platform we have one. The output of assembler is called object file. Its translates assembly language to machine code.
- **Relocatable Machine Code –** It can be loaded at any point and can be run. The address within the program will be in such a way that it will cooperate for the program movement.
- **Loader/Linker –** It converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message (or sometimes both can happen). Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.

Analysis Phase – An intermediate representation is created from the give source code :

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer

- Lexical analyzer divides the program into “tokens”, Syntax analyzer recognizes “sentences” in the program using syntax of language and Semantic analyzer checks static semantics of each construct.

Synthesis Phase – Equivalent target program is created from the intermediate representation. It has three parts :

1. Intermediate Code Generator
2. Code Optimizer
3. Code Generator

Intermediate Code Generator generates “abstract” code, Code Optimizer optimizes the abstract code, and final Code Generator translates abstract intermediate code into specific machine instructions.

Compiler construction tools

The compiler writer can use some specialized tools that help in implementing various phases of a compiler. These tools assist in the creation of an entire compiler or its parts. Some commonly used compiler construction tools include:

1. **Parser Generator –**
   It produces syntax analyzers (parsers) from the input that is based on a grammatical description of programming language or on a context-free grammar. It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time.
   Example:PIC, EQM

2. **Scanner Generator –**
   It generates lexical analyzers from the input that consists of regular expression description based on tokens of a language. It generates a finite automation to recognize the regular expression.

Example: Lex

3.Syntax directed translation engines –

It generates intermediate code with three address format from the input that consists of a parse tree. These engines have routines to traverse the parse tree and then produces the intermediate code. In this, each node of the parse tree is associated with one or more translations.

4.Automatic code generators –

It generates the machine language for a target machine. Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator. Template matching process is used. An intermediate language statement is replaced by its equivalent machine language statement using templates.

5.Data-flow analysis engines –

It is used in code optimization.Data flow analysis is a key part of the code optimization that gathers the information, that is the values that flow from one part of a program to another. Refer – data flow analysis in Compiler

6.Compiler construction toolkits –

It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compiler.

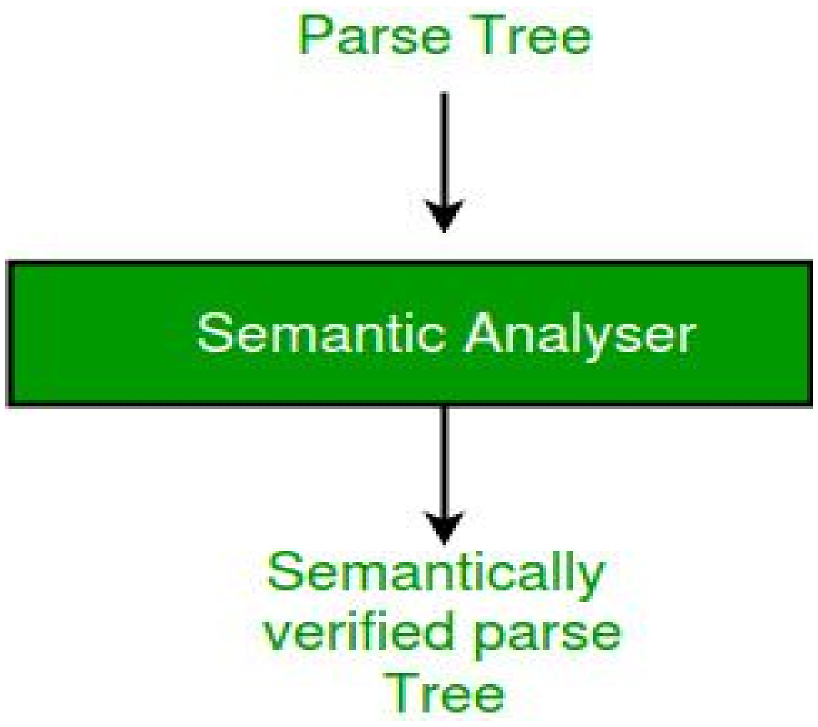
We basically have two phases of compilers, namely Analysis phase and Synthesis phase. Analysis phase creates an intermediate representation from the given source code. Synthesis phase creates an equivalent target program from the intermediate representation.

Symbol Table – It is a data structure being used and maintained by the compiler, consists all the identifier's name along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

The compiler has two modules namely front end and back end. Front-end constitutes of the Lexical analyzer, semantic analyzer, syntax analyzer and intermediate code generator. And the rest are assembled to form the back end.

1. **Lexical Analyzer –** It reads the program and converts it into tokens. It converts a stream of lexemes into a stream of tokens. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes white-spaces and comments.
2. **Syntax Analyzer –** It is sometimes called as parser. It constructs the parse tree. It takes all the tokens one by one and uses Context Free Grammar to construct the parse tree.
   Why Grammar ?
   The rules of programming can be entirely represented in some few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.
   Syntax error can be detected at this level if the input is not in accordance

with the grammar.



3. **Semantic Analyzer –** It verifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree.It also does type checking, Label checking and Flow control checking.
4. Intermediate Code Generator **–** It generates intermediate code, that is a form which can be readily executed by machine We have many popular intermediate codes. Example – Three address code etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.
   Till intermediate code, it is same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don't need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.
5. Code Optimizer **–** It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being

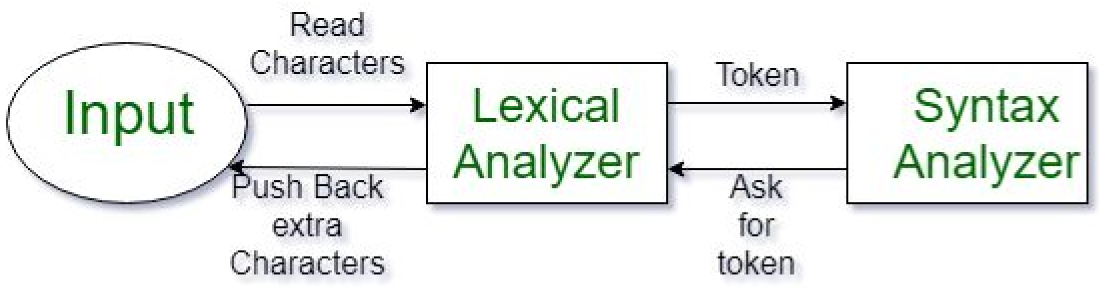
transformed is not altered. Optimisation can be categorized into two types: machine dependent and machine independent.

6. **Target Code Generator –** The main purpose of Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection etc. The output is dependent on the type of assembler. This is the final stage of compilation.

Symbol table is build in syntax and semantic analysis phase.
It is used by compiler to achieve compile time efficiency.

Lexical Analysis is the first phase of compiler also known as scanner. It converts the High level input program into a sequence of **Tokens**.

- Lexical Analysis can be implemented with the Deterministic finite Automata.
- The output is a sequence of tokens that is sent to the parser for syntax analysis



What is a token?

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

```
Keywords; Examples-for, while, if etc.
```

```
Identifier; Examples-Variable name, function name etc.
```

```
Operators; Examples '+', '++', '-' etc.
```

```
Separators; Examples ',' ';' etc
```

Example of Non-Tokens:

- Comments, preprocessor directive, macros, blanks, tabs, newline etc

Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

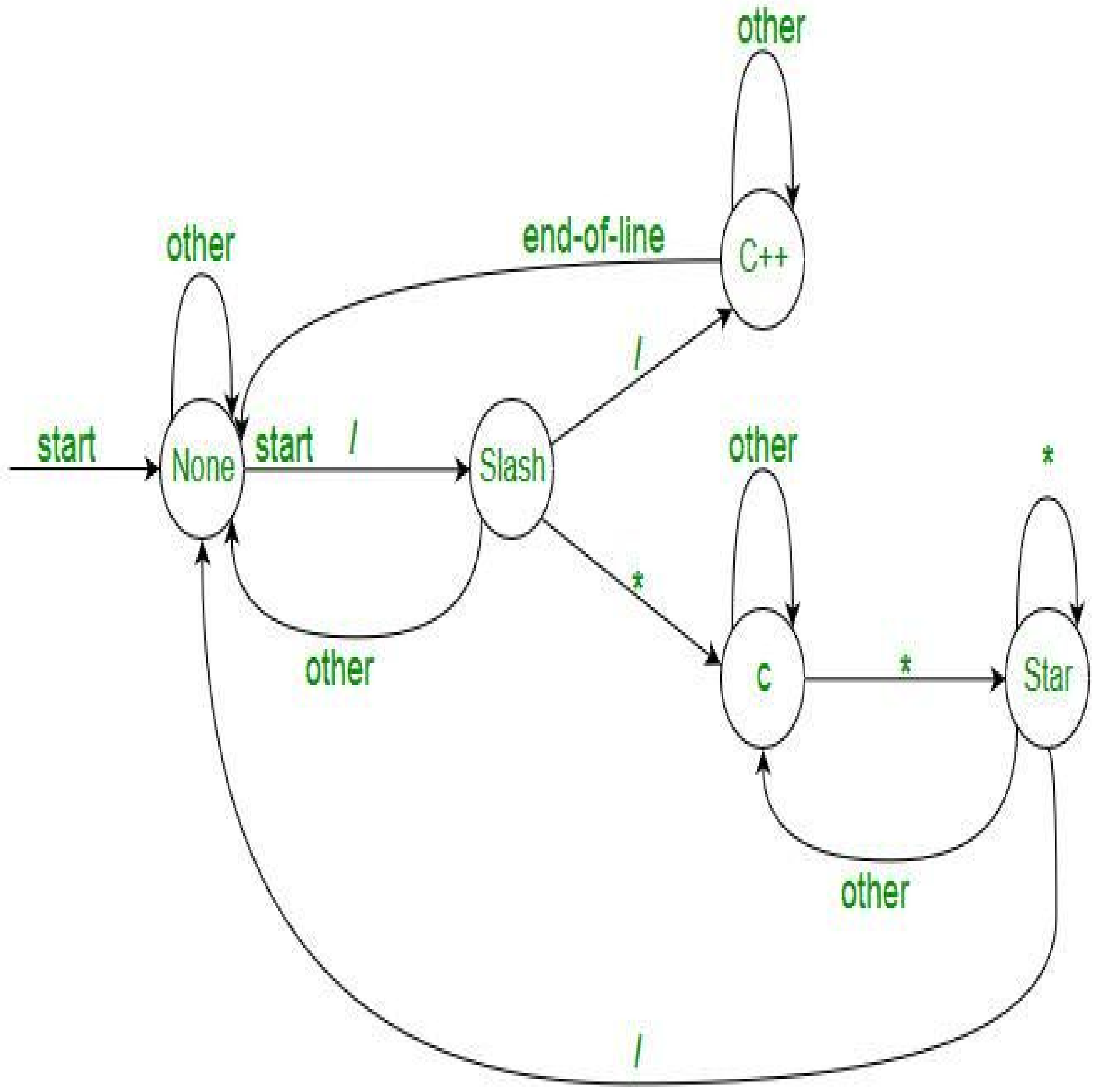
How Lexical Analyzer functions

1. Tokenization .i.e Dividing the program into valid tokens.

2. Remove white space characters.

3. Remove comments.

4. It also provides help in generating error message by providing row number and column number.



The lexical analyzer identifies the error with the help of automation machine and the grammar of the given language on which it is based like C , C++ and gives row number and column number of the error.

Suppose we pass a statement through lexical analyzer –

a = b + c ;   It will generate token sequence like this:

id=id+id;   Where each id reference to it’s variable in the symbol table referencing all details

For example, consider the program

```
int main()

{

  // 2 variables

  int a, b;

  a = 10;

 return 0;

}
```

All the valid tokens are:

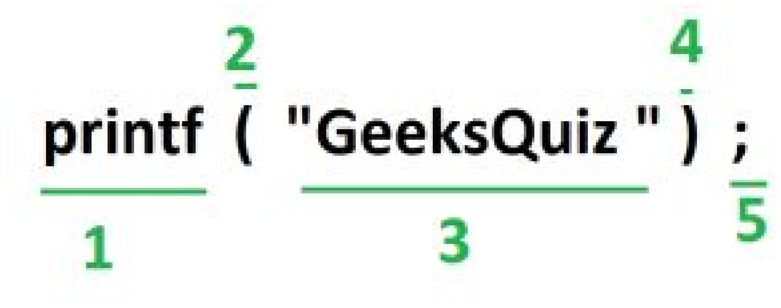
```
'int'  'main'  '('  ')'  '{'  '}'  'int'  'a' ','  'b'  ';'

 'a'  '='  '10'  ';' 'return'  '0'  ';'  '}'
```

Above are the valid tokens.

You can observe that we have omitted comments.

As another example, consider below printf statement.



Flex (Fast Lexical Analyzer Generator)

FLEX (fast lexical analyzer generator) is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. It is used together with Berkeley Yacc parser generator or GNU Bison parser generator. Flex and Bison both are more flexible than Lex and Yacc and produces faster code.

Bison produces parser from the input file provided by the user. The function **yylex()** is automatically generated by the flex when it is provided with a **.l file** and this yylex() function is expected by parser to call to retrieve tokens from current/this token stream.

Introduction to Syntax Analysis

When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scans the input and divides it into tokens) to target code generation.

Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a Parse tree or Syntax tree. The parse tree is constructed by using the pre-defined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. if not, error is reported by syntax analyzer.

The Grammar for a Language consists of Production rules.

Example:

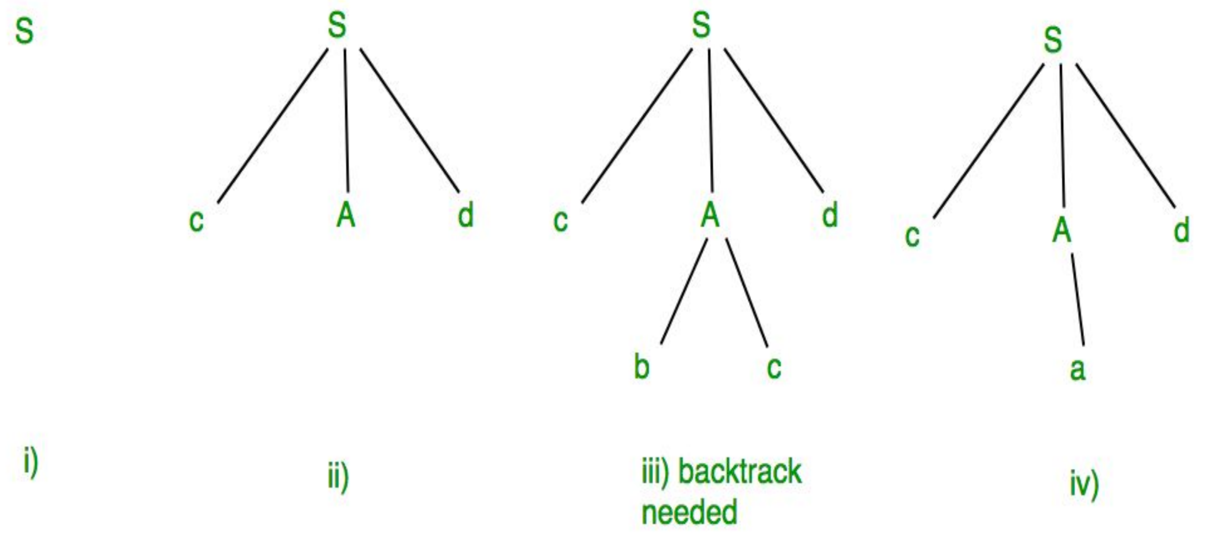
Suppose Production rules for the Grammar of a language are:

```
 S -> cAd
```

```
  A -> bc|a
```

```
  And the input string is “cad”.
```

Now the parser attempts to construct syntax tree from this grammar for the given input string. It uses the given production rules and applies those as needed to generate the string. To generate string “cad” it uses the rules as shown in the given diagram:



In the step iii above, the production rule A->bc was not a suitable one to apply (because the string produced is “cbcd” not “cad”), here the parser needs to backtrack, and apply the next production rule available with A which is shown in the step iv, and the string “cad” is produced.

Thus, the given input can be produced by the given grammar, therefore the input is correct in syntax.

But backtrack was needed to get the correct syntax tree, which is really a complex process to implement.

There can be an easier way to solve this, which we shall see in the next article “Concepts of FIRST and FOLLOW sets in Compiler Design”.

Context **F**ree **G**rammars (CFG) can be classified on the basis of following two properties:

1) Based on number of strings it generates.

- If CFG is generating finite number of strings, then CFG is **Non-Recursive**
- If CFG can generate infinite number of strings then the grammar is said to be **Recursive** grammar

During Compilation, the parser uses the grammar of the language to make a parse tree(or derivation tree) out of the source code. The grammar used must be unambiguous. An ambiguous grammar must not be used for parsing.

2) Based on number of derivation trees.

- If there is only 1 derivation tree then the CFG is unambiguous.
- If there are more than 1 derivation tree, then the CFG is ambiguous.

Types of Recursive Grammars

Based on the nature of the recursion in a recursive grammar, a recursive CFG can be again divided into the following:

- Left Recursive Grammar (having left Recursion)
- Right Recursive Grammar (having right Recursion)
- General Recursive Grammar(having general Recursion)

Note:A linear grammar is a context-free grammar that has at most one non-terminal in the right hand side of each of its productions.

Inherently ambiguous Language:

Let L be a Context Free Language (CFL). If every Context Free Grammar G with Language L = L(G) is ambiguous, then L is said to be inherently ambiguous Language. Ambiguity is a property of grammar not languages. Ambiguous grammar is unlikely to be useful for a programming language, because two parse trees structures(or more) for the same string(program) implies two different meanings (executable programs) for the program.

An inherently ambiguous language would be absolutely unsuitable as a programming language, because we would not have any way of fixing a unique structure for all its programs.

> **Disambiguate the grammar** i.e., rewriting the grammar such that there is only one derivation or parse tree possible for a string of the language which the grammar represents.

Role of the parser :

In the syntax analysis phase, a compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language. This is done by a parser. The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be the grammar for the source language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.

The process of deriving the string from the given grammar is known as derivation (parsing).

Depending upon how derivation is done we have two kinds of parsers :-

1. Top Down Parser
2. Bottom Up Parser

Top Down Parser

Top down parsing attempts to build the parse tree from root to leave. Top down parser will start from start symbol and proceeds to string. It follows leftmost derivation. In leftmost derivation, the leftmost non-terminal in each sentential is always chosen.

Parsing is classified into two categories, i.e. Top Down Parsing and Bottom-Up Parsing. Top-Down Parsing is based on Left Most Derivation whereas Bottom Up Parsing is dependent on Reverse Right Most Derivation.

The process of constructing the parse tree which starts from the root and goes down to the leaf is Top-Down Parsing.

- Top-Down Parsers constructs from the Grammar which is free from ambiguity and left recursion.
- Top Down Parsers uses leftmost derivation to construct a parse tree.

- It allows a grammar which is free from Left Factoring.

Classification of Top-Down Parsing –

1. **With Backtracking:** Brute Force Technique or Recursive Descent Parsing
2. **Without Backtracking:** Predictive Parsing or Non-Recursive Parsing or LL(1) Parsing or Table Driver Parsing

Brute Force Technique or Recursive Descent Parsing –

1. Whenever a Non-terminal spend first time then go with the first alternative and compare with the given I/P String
2. If matching doesn't occur then go with the second alternative and compare with the given I/P String.
3. If matching again not found then go with the alternative and so on.
4. Moreover, If matching occurs for at least one alternative, then the I/P string is parsed successfully.

LL(1) or Table Driver or Predictive Parser –

1. In LL1, first L stands for Left to Right and second L stands for Left-most Derivation. 1 stands for number of Look Aheads token used by parser while parsing a sentence.
2. LL(1) parsing is constructed from the grammar which is free from left recursion, common prefix, and ambiguity.
3. LL(1) parser depends on 1 look ahead symbol to predict the production to expand the parse tree.

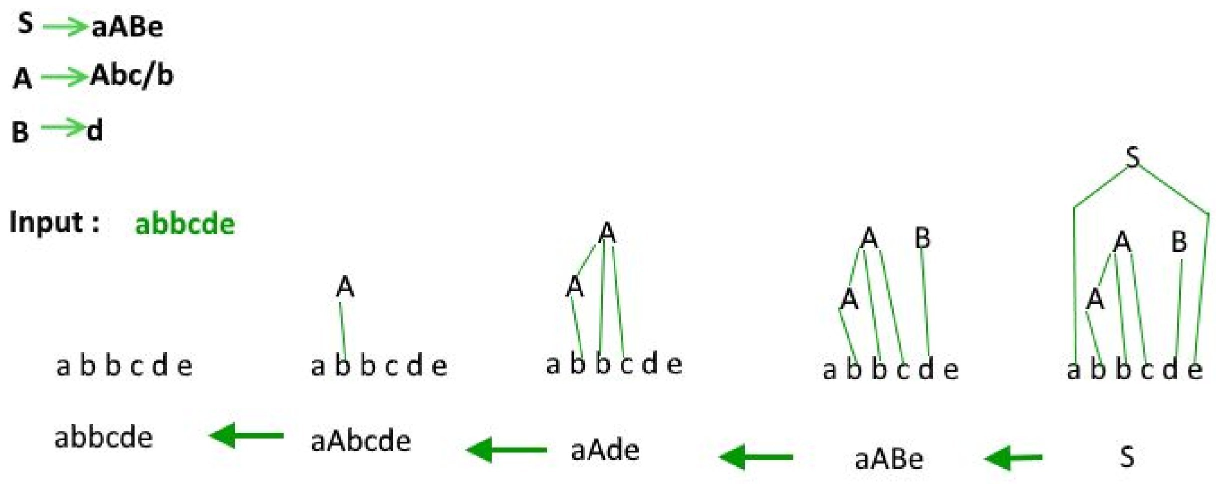
4. This parser is Non-Recursive.

In this article, we are discussing the Bottom Up parser.

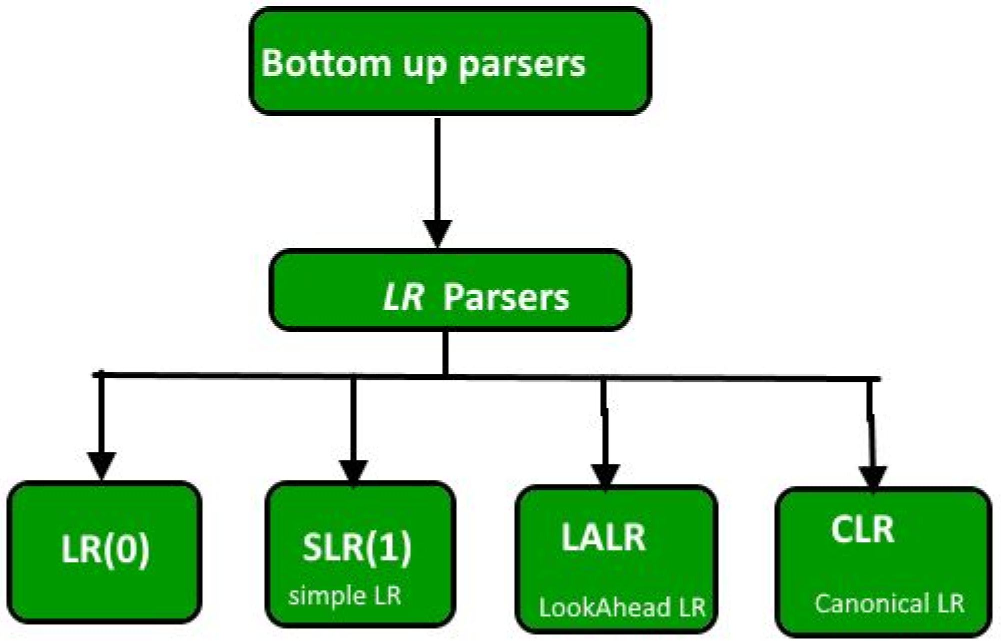
Bottom Up Parsers / Shift Reduce Parsers

Build the parse tree from leaves to root. Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of grammar by tracing out the rightmost derivations of w in reverse.

Eg.



Classification of bottom up parsers



A general shift reduce parsing is LR parsing. The L stands for scanning the input from left to right and R stands for constructing a rightmost derivation in reverse.

Benefits of LR parsing:

1. Many programming languages using some variations of an LR parser. It should be noted that C++ and Perl are exceptions to it.
2. LR Parser can be implemented very efficiently.
3. Of all the Parsers that scan their symbols from left to right, LR Parsers detect syntactic errors, as soon as possible.

Important Notes

1. Even though CLR parser does not have RR conflict but LALR may contain RR conflict.

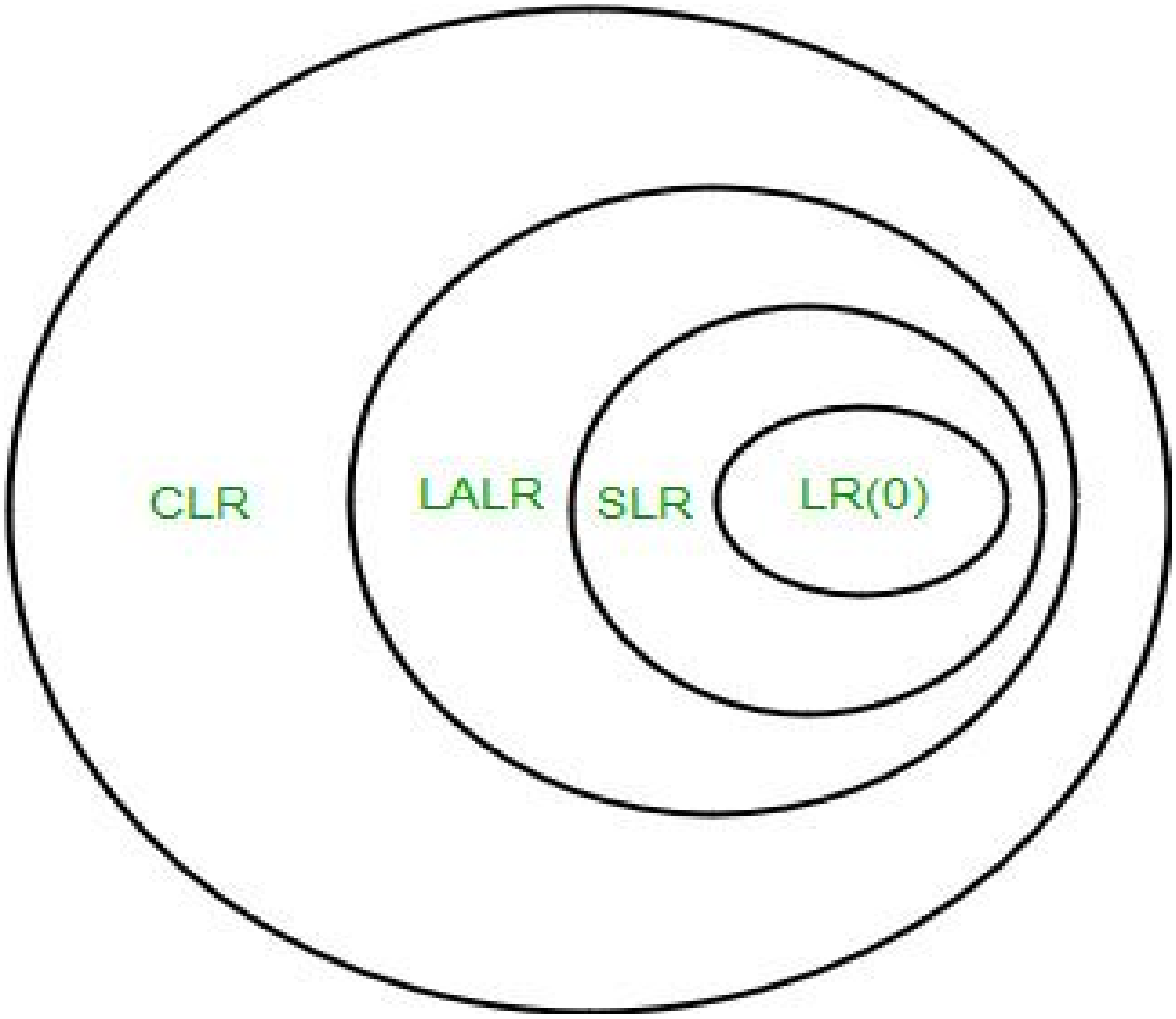
2. If number of states LR(0) = n1,

number of states SLR = n2,

number of states LALR = n3,

number of states CLR = n4 then,

n1 = n2 = n3 <= n4



Operator grammar and precedence parser

A grammar that is generated to define the mathematical operators is called **operator grammar** with some restrictions on grammar. An **operator precedence grammar** is a context-free grammar that has the property that no production has either an empty right-hand side (null productions) or two adjacent non-terminals in its right-hand side.

Examples –

This is the example of operator grammar:

```
E->E+E/E*E/id
```

However, grammar that is given below is not an operator grammar because two non-terminals are adjacent to each other:

```
S->SAS/a
```

```
A->bSb/b
```

Although, we can convert it into an operator grammar:

```
S->SbSbS/SbS/a
```

`A->bSb/b`

Operator precedence parser –

An operator precedence parser is a one of the bottom-up parser that interprets an operator-precedence grammar. This parser is only used for operator grammars. *Ambiguous grammars are not allowed* in case of any parser except operator precedence parser.

Syntax Directed Translation

Background : Parser uses a CFG(Context-free-Grammer) to validate the input string and produce output for next phase of the compiler. Output could be either a parse tree or abstract syntax tree. Now to interleave semantic analysis with syntax analysis phase of the compiler, we use Syntax Directed Translation.

Syntax Directed Translation are augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down the parse tree in form of attributes attached to the nodes.

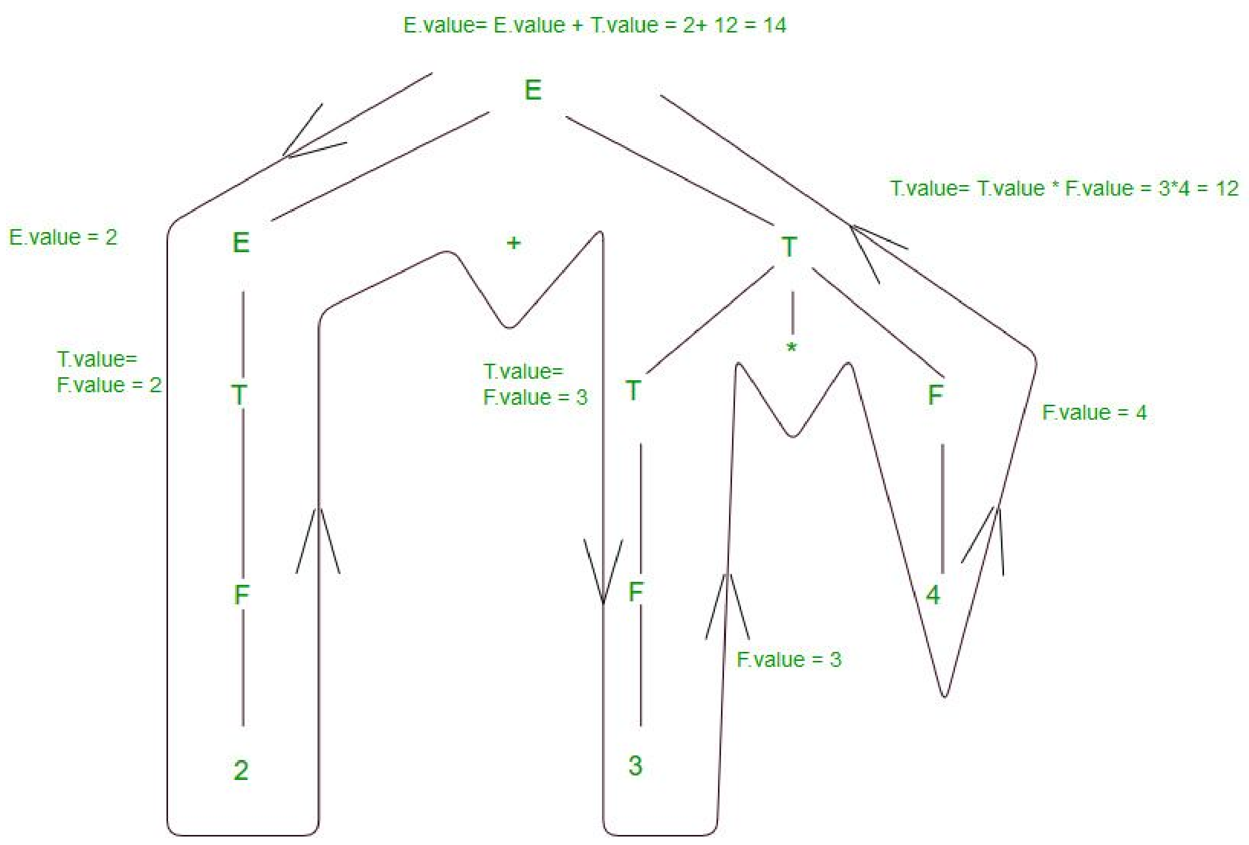
Syntax directed translation rules use 1) lexical values of nodes, 2) constants & 3) attributes associated to the non-terminals in their definitions.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

This is a grammar to syntactically validate an expression having additions and multiplications in it. Now, to carry out semantic analysis we will augment SDT rules to this grammar, in order to pass some information up the parse tree and check for semantic errors, if any. In this example we will focus on evaluation of the given expression, as we don't have any semantic assertions to check in this very basic example.

For understanding translation rules further, we take the first SDT augmented to [E -> E+T] production rule. The translation rule in consideration has val as attribute for both the non-terminals – E & T. Right hand side of the translation rule corresponds to attribute values of right side nodes of the production rule and vice-versa. Generalizing, SDT are augmented rules to a CFG that associate 1) set of attributes to every node of the grammar and 2) set of translation rules to every production rule using attributes, constants and lexical values.

To evaluate translation rules, we can employ one depth first search traversal on the parse tree. This is possible only because SDT rules don't impose any specific order on evaluation until children attributes are computed before parents for a grammar having all synthesized attributes. Otherwise, we would have to figure out the best suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals. For better understanding, we will move bottom up in left to right fashion for computing translation rules of our example.



Above diagram shows how semantic analysis could happen. The flow of information happens bottom-up and all the children attributes are computed before parents, as discussed above. Right hand side nodes are sometimes annotated with subscript 1 to distinguish between children and parent.

Additional Information

Synthesized Attributes are such attributes that depend only on the attribute values of children nodes.

Thus [E -> E+T { E.val = E.val + T.val }] has a synthesized attribute val corresponding to node E. If all the semantic attributes in an augmented grammar are synthesized, one depth first search traversal in any order is sufficient for semantic analysis phase.

Inherited Attributes are such attributes that depend on parent and/or siblings attributes.

Thus [Ep -> E+T { Ep.val = E.val + T.val, T.val = Ep.val }], where E & Ep are same production symbols annotated to differentiate between parent and child, has an inherited attribute val corresponding to node T.

Types of attributes –

Attributes may be of two types – Synthesized or Inherited.

1. **Synthesized attributes –**
   A Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production. Synthesized attributes represent information that is being passed up the parse tree. The attribute can take value only from its children (Variables in the RHS of the production).

   For eg. let's say A -> BC is a production of a grammar, and A's attribute is dependent on B's attributes or C's attributes than it will be synthesized attribute.

2. **Inherited attributes –**

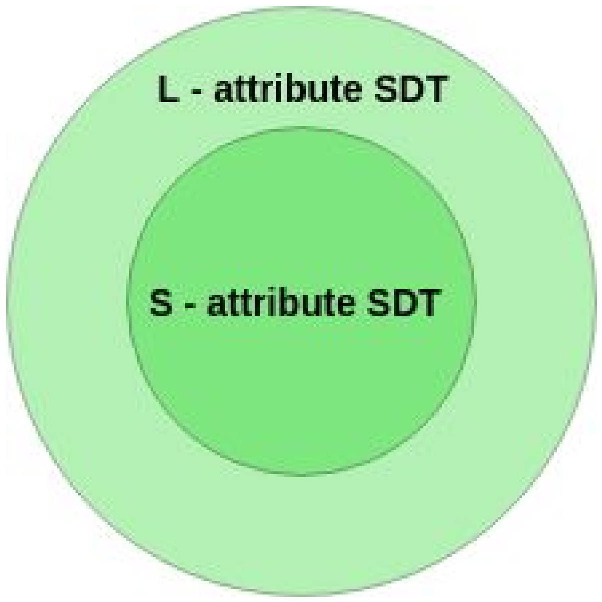
An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute. The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production).

For example, let’s say A -> BC is a production of a grammar and B’s attribute is dependent on A’s attributes or C’s attributes than it will be inherited attribute.

Now, let’s discuss about S-attributed and L-attributed SDT.

1. **S-attributed SDT :**
    - If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
    - S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
    - Semantic actions are placed in rightmost place of RHS.
2. **L-attributed SDT:**
    - If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
    - Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
    - Semantic actions are placed anywhere in RHS.

Note – If a definition is S-attributed, then it is also L-attributed but **NOT** vice-versa.



Semantics

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

```
CFG + semantic rules = Syntax Directed Definitions
```

For example:

```
int a = “value”;
```

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 assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking

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. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

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.

Reduction : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

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.

Attributes are two tuple value, <attribute name, attribute value>

For example:

```
int value  = 5;
```

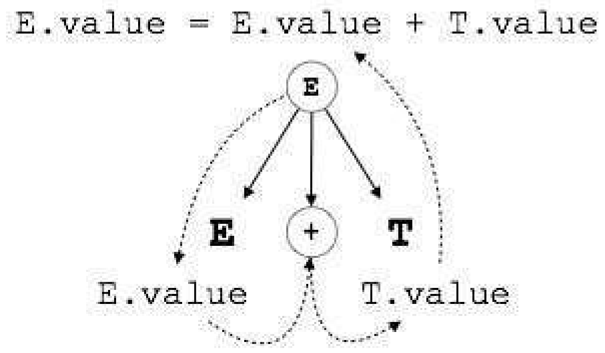
```
<type, “integer”>
```

```
<presentvalue, “5”>
```

For every production, we attach a semantic rule.

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).



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

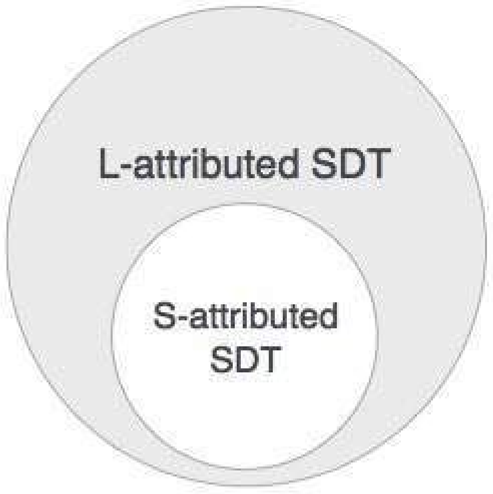
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

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.



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