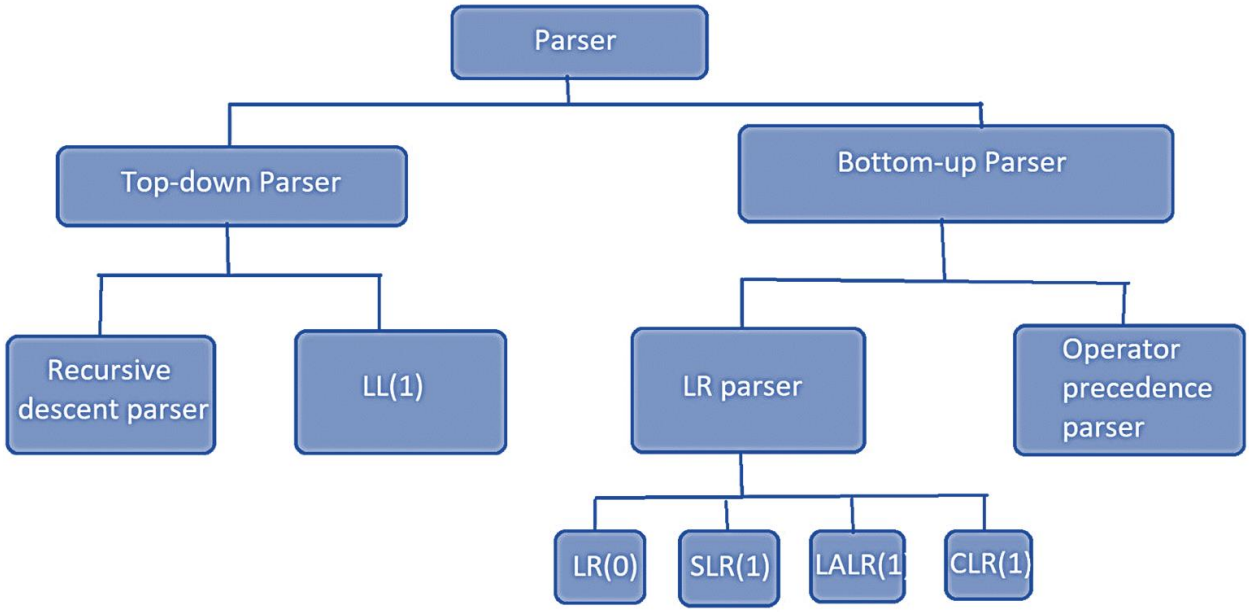
LECTURE -1- : TYPES OF PARSERS IN COMPILER DESIGN

The **parser** is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation. The parser is also known as *Syntax Analyzer*.



Types of Parser:

The parser is mainly classified into two categories, i.e. Top-down Parser, and Bottom-up Parser. These are explained below:

1- Top-Down Parser:

The top-down parser is the parser that generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation.

Further Top-down parser is classified into **2 types**: *Recursive descent parser*, and *Non-recursive descent parser*.

Recursive descent parser is also known as the Brute force parser or the backtracking parser. It basically generates the parse tree by using brute force and backtracking.

Non-recursive descent parser is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses a parsing table to generate the parse tree instead of backtracking.

2- Bottom-up Parser:

Bottom-up Parser is the parser that generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the start symbol. It uses the reverse of the rightmost derivation.

Further Bottom-up parser is classified into two types: LR parser, and Operator precedence parser.

LR parser is the bottom-up parser that generates the parse tree for the given string by using unambiguous grammar. It follows the reverse of the rightmost derivation.

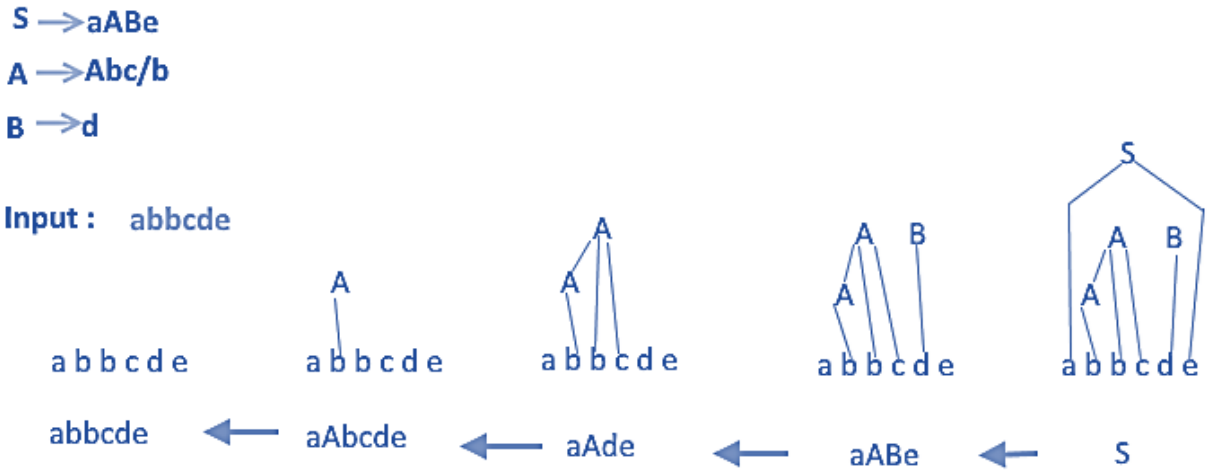
LR parser is of four types:

a- LR(0) b- SLR(1) c-LALR(1) d-CLR(1)

Operator precedence parser generates the parse tree form given grammar and string but the only condition is two consecutive non-terminals and epsilon never appear on the right-hand side of any production.

Bottom Up Parsers / Shift Reduce Parsers

Bottom up parsers start from the sequence of terminal symbols and work their way back up to the start symbol by repeatedly replacing grammar rules' right hand sides by the corresponding non-terminal. This is the reverse of the derivation process, and is called "reduction".



Example:1 consider the grammar

S→ aABe

A→ Abc|b

B→ d

The sentence **abbcde** can be reduced to S by the following steps:

Sol:

abbcde

aAbcde

aAde

aABe

S

Example:2 consider the grammar

S→ aABe

A→ Abc|bc

B→ dd

The sentence **abcbcdde** can be reduced to S by the following steps:

Sol:

abcbcdde

aAbcdde

aAdde

aABe

S

Example:3 Using the following arithmetic grammar

E → E+T | T
T → T*F | F
F → (E) | id

Illustrates the bottom-up parse for string w= id * id
The reductions will be discussed in terms of the sequence of strings

id * id
F * id
T * id
T * F **handle**
T **handle pruning start**
E **start**

The following *derivation* corresponds to the parse

E → T
→T * F
→ T * id
→ F * id
→ id * id

This derivation is in fact a ***RightMost Derivation*** (RMD):

1	2	3	4	5	6
id * id	F * id ↑ id	T * id ↑ F ↑ id	T * F ↑ ↑ F id ↑ id	T ↗ ↖ T * F ↑ ↑ F id ↑ id	E ↑ T ↗ ↖ T * F ↑ ↑ F id ↑ id

We can think of bottom-up parsing as the process of "reducing" a string w to the start symbol of the grammar.

At each reduction step, a specific substring matching the body of a production is replaced by the *nonterminal* at the head of that production.

The key decisions during bottom-up parsing are *about when to reduce* and about *what production to apply*, as the parse proceeds.

The grammar is the expression grammar in example 3:

The reductions will be discussed in terms of the sequence of strings

id * id → **F * id** → **T * id** → **T * F** → **T** → E ⋯ (reductions)

By definition, a ***Reduction*** is the reverse of a step in a derivation (recall that in a derivation, a nonterminal in a sentential form is replaced by the body of one of its productions).

The goal of Bottom-Up Parsing is therefore to construct a derivation in reverse. The following derivation corresponds to the parse in example 3:

E → **T** → **T * F** → **T * id** → **F * id** → **id * id** ⋯ **(**RMD derivation**)**

Handle Pruning:

Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse. Informally, a "**handle**" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

For example, adding subscripts to the tokens **id** for clarity, the handles during the parse of $\mathbf{id_1 * id_2}$ according to the expression grammar

E → E+T | T
T → T*F | F
F → (E) | id

Although **T** is the body of the production **E → T**, the symbol **T** is not a handle in the sentential form **T * id2**.
If **T** were indeed replaced by **E**, we would get the string **E * id2**, which cannot be derived from the start symbol **E**.
Thus, the leftmost substring that matches the body of some production need not be a handle.

E → T → T * F → F * F → F * id1 → id1 * id2 ⋯ (derivation)

Right Sentential Form	Handle	Reducing Production
id * id	id	F → id
F * id	F	T → F
T * id	id	F → id
T * F	T * F	T → T * F
T	T * F	T → T * F
E	T	E → T

Example: consider the grammar:
E' → E
E → E+T | E-T | T
T → (E) | id
By using RMD derivation **derive id + (id - id)**
Solution:
E' → E
→ E + T
→ E + (E)
→ E + (E - T)
→ E + (E - id)
→ E + (T - id)
→ E + (id - id)
→ T + (id - id)
→ id + (id - id)

Right Sentential Form	Handle	Reducing Production
id + (id – id)		
T + (id – id)	id	
E + (id – id)	T	T → id
E + (T – id)	id	E → T
E + (E – id)	T	E → T
E + (E – T)	E	T → id
E + (E)	(E – T)	E → E – T
E + T	(E)	T → E
E	E + T	E → E + T
E'	E	E' → E

H.W.

For this grammar

E → E+T | T

T → T*F | F

F → id | (E)

Parse the input id * id + id

LECTURE -2- : LR PARSER FAMILY

The ***LR(k)*** parsing technique was introduced by Knuth in 1965

L is for Left-to-right scanning of input, ***R*** corresponds to a Rightmost derivation done in reverse, and ***k*** is the number of lookahead symbols used to make parsing decisions.

There are three widely used Algorithms available for constructing an LR parser:

- SLR (1) – Simple LR Parser.
- LR (1) – LR Parser.
- LALR (1) – Look-Ahead LR Parser.

Rules for LR parser:

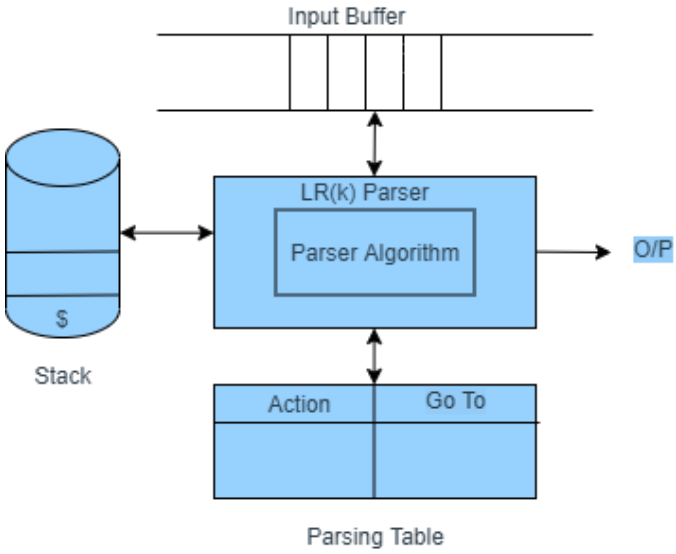
The rules of LR parser as follows:

- The first item from the given grammar rules adds itself as the first closed set.
- If an object is present in the closure of the form A⟶ α. β. γ, where the next symbol after the symbol is non-terminal, add the symbol's production rules where the dot precedes the first item.
- Repeat steps (B) and (C) for new items added under (B).

The LR-Parsing Algorithm

A schematic of an LR parser consists of an ***input, an output, a stack, a driver program, and a parsing table*** that has two pasts (ACTION and GOTO).

- The driver program is the same for all LR parsers; only the **parsing table** changes from one parser to another.
- The parsing program reads characters from an **input buffer** one at a time.
- Where a shift-reduce parser would **shift a symbol**, an LR parser shifts a state.
- Each state summarizes the information contained in the **stack** below it.



Parsing Table:

Parsing table is divided into two parts- ***Action table and Go-To table***. The action table gives a grammar rule to implement the given current state and current terminal in the input stream.

1. The ACTION function takes as arguments a state i and a terminal a (or $, the input endmarker).

   The value of ACTION [i, a] can have one of four forms:

   a. Shift j, where j is a state : The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a.

   b. Reduce A → β. The action of the parser effectively reduces β on the top of the stack to head A.

   c. Accept. The parser accepts the input and finishes parsing.

   d. Error. The parser discovers an error in its input and takes some corrective action.

2. We extend the **GOTO function**, defined on sets of items, to states: if GOTO [Ii, A] = Ij, then GOTO also maps a state i and a nonterminal A to state j.

Example: The **ACTION** and **GOTO** functions of an **LR**-parsing table for the expression the following grammar,

E → E+T | T
T → T*F | F
F → (E) | id

Repeated with the productions numbered

1. **E → E + T**
2. **E → T**
3. **T → T * F**
4. **T → F**
5. **F → (E)**
6. **F → id**

The codes for the actions are:

1. ***si*** means **shift** and stack state ***i***,
2. **rj** means **reduce** by the production numbered **j**,
3. **acc** means **accept**,
4. **blank** means **error**.

First construct the set of items I0:

I0:

E → ·E + T r1
E → ·T r2
T → ·T * F r3
T → ·F r4
F → ·(E) r5
F → ·id r6

I1: Goto [I0, E]

E → E· + T ⋯ Accept

I2: Goto [I0, T]

E → T· ⋯ Complete
T → T· * F

I3: Goto [I0, F]

T → F·

I4: Goto [I0, ( ]

F → (·E)
E → ·E + T
E → ·T

T → ·T * F
T → ·F
F → ·(E)
F → ·id

I5: Goto [I0, id]
F → id· ⋯ Complete

I6: Goto [I1, +]
E → E +· T
T → ·T * F
T → ·F
F → ·(E)
F → ·id

I7: Goto [I2, *]
T → T *· F
F → ·(E)
F → ·id

I8: Goto [I4, E]
F → (E·)
E → E· + T
Goto [I4, T] = I2 حالات مكررة
Goto [I4, F] = I3
Goto [I4, (] = I4
Goto [I4, id] = I5

I9: Goto [I6, T]
E → E + T· ⋯ Complete
Goto [I6, T] = I2
Goto [I6, F] = I3
Goto [I6, (] = I4
Goto [I6, id] = I5

I10: Goto [I7, F]
T → T * F· ⋯ Complete
Goto [I7, (] = I4
Goto [I7, id] = I5

I11: Goto [I8,)]
F → (E) · ⋯ Complete
Follow(E) = {$, +,)}
Follow(T) = {$, +,), *}
Follow(F) = {$, +,), *}

LR Parser Table:

Stack	Input Symbols	Output
0	id * id + id $	Shift
0 id 5	* id + id $	Reduce F → id
0 F 3	* id + id $	Reduce T → F
0 T 2	* id + id $	Shift
0 T 2 * 7	id + id $	Shift
0 T 2 * 7 id 5	+ id $	Reduce F → id
0 T 2 * 7 F 10	+ id $	Reduce T → T * F
0 T 2	+ id $	Reduce E → T
0 E 1	+ id $	Shift
0 E 1 + 6	id $	Shift
0 E 1 + 6 id 5	$	Reduce F → id
0 E 1 + 6 F 3	$	Reduce T → F
0 E 1 + 6 T 9	$	Reduce E → E + T
0 E 1	$	Accept

Constructing SLR-Parsing Tables

We shall refer to the parsing table constructed by this method as an SLR table, and to an LR parser using an SLR-parsing table as an SLR parser.
The other two methods augment the SLR method with lookahead information.
The ***action*** and ***goto*** entries in the parsing table are then constructed using the following algorithm. It requires us to know FOLLOW(A) for each nonterminal A of a grammar.

Constructing an SLR-parsing table Algorithm:

INPUT: An augmented grammar G'.
OUTPUT: The SLR-parsing table functions ACTION and COTO for G'.
METHOD:
1. Construct C = {I_0, I_1, …., I_n}, the collection of sets of LR(0) items for G'.
2. State ***i*** is constructed from I***i***.

The parsing actions for state ***i*** are determined as follows:

a. If [**A** → α **.** a β] is in I***i*** and GOTO (I***i***, α) = I**j**, then set ACTION [***i***, a] to "**shift j**." Here a must be a terminal.

b. If [**A** → α **.**] is in I***i***, then set ACTION [***i***, a] to "**reduce A** → α " for all a in FOLLOW(A); here **A** may not be **S'**.

c. if [**S'** → **S.**] is in I***i***, then set ACTION [***i***, **S**] to "**accept**." If any conflicting actions result from the above rules, we say the grammar is not SLR (1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state ***i*** are constructed for all nonterrninals **A** using the rule: If GOTO (I***i***, A) = I**j**, then GOTO [***i***, A] = **j**.

4. All entries not defined by rules (2) and (3) are made "**error**."

5. The initial state of the parser is the one constructed from the set of items containing [**S'** → **.S**].

Example:

Let us construct the SLR table for the **augmented** expression grammar.
The canonical collection of sets of LR(0) items for the grammar.

I0:

E' →	**·E**	**r1**
E →	**·E + T**	**r2**
E →	**·T**	**r3**
T →	**·T * F**	**r4**
T →	**·F**	**r5**
F →	**·(E)**	**r6**
F →	**·id**	**r7**

I1: Goto [I0, E]

E' → **E·** ⋯ Accept
E → **E·** **+ T**

I2: Goto [I0, T]

E → **T·** ⋯ **Complete**
T → **T·** *** F**

I3: Goto [I0, F]

T → **F·** ⋯ **Complete**

I4: Goto [I0, (]

F → (·E)
E → ·E + T
E → ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id

I5: Goto [I0, id]

F → id· ⋯ Complete

I6: Goto [I1, +]

E → E +· T
T → ·T * F
T → ·F
F → ·(E)
F → ·id

I7: Goto [I2, *]

T → T *· F
F → ·(E)
F → ·id

I8: Goto [I4, *]

F → (E·)

Goto [I4, E] = I1
Goto [I4, T] = I2
Goto [I4, F] = I3
Goto [I4, (] = I4
Goto [I4, id] = I5

I9: Goto [I6, T]

E → E + T· ⋯ Complete

Goto [I6, T] = I2
Goto [I6, F] = I3
Goto [I6, (] = I4
Goto [I6, id] = I5

I10: Goto [I7, F]
T → T * F· ⋯ Complete
Goto [I7, (] = I4
Goto [I7, id] = I5

I11: Goto [I8,)]
F → (E)· ⋯ Complete

Follow(E) = {$, +,)}
Follow(T) = {$, +,), *}
Follow(F) = {$, +,), *}

SLR-Parsing Tables:

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				Accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

LECTURE -3- : SYNTAX DIRECTED TRANSLATION

Syntax Directed Translation has 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 with 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.

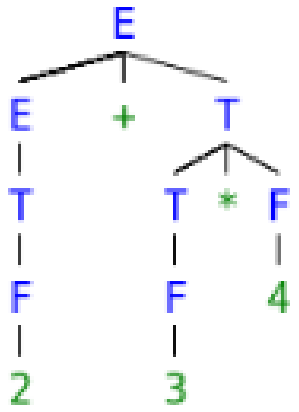
Example

E → E+T | T
T → T*F | F
F → id

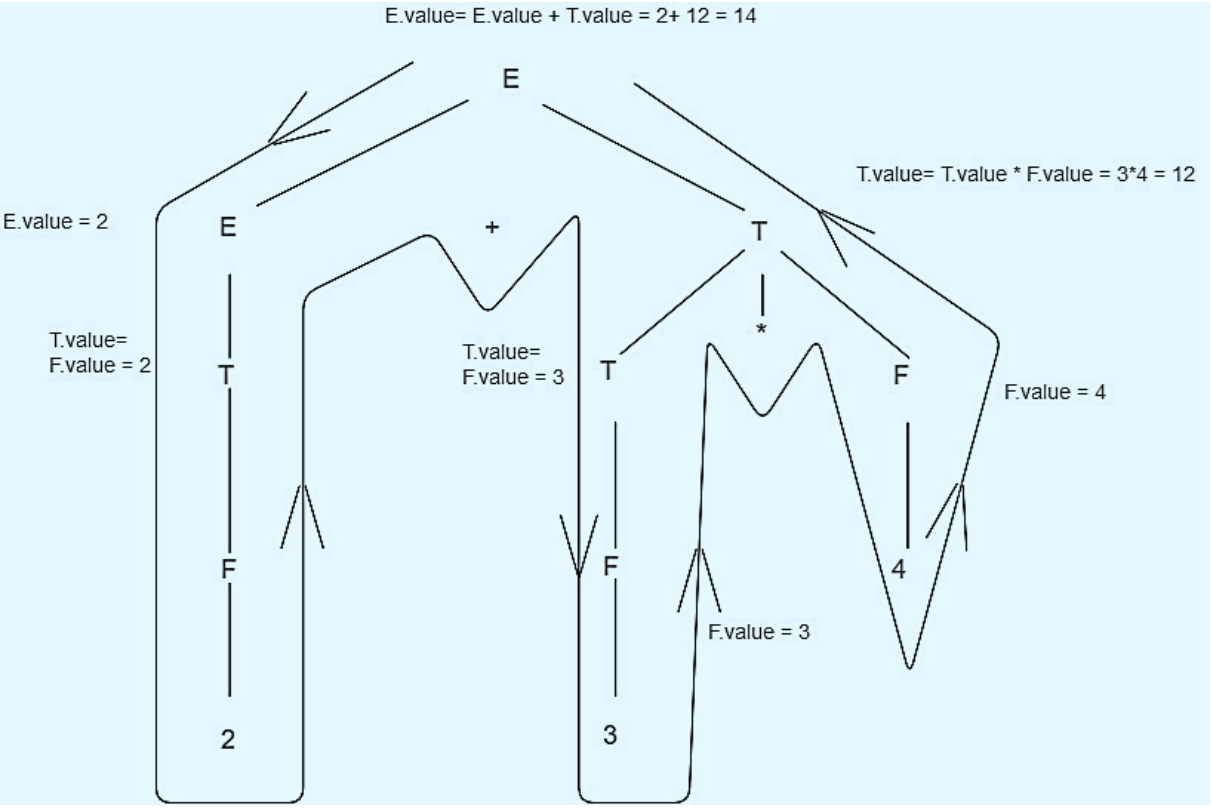
This is a grammar to syntactically validate an expression having additions and multiplications in it. Now, to carry out semantic analysis we will augment SDT rules to this grammar, in order to pass some information up the parse tree and check for semantic errors, if any. In this example, we will focus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

1. **E → E + T** { E.val = E.val + T.val }
2. **E → T** { E.val = T.val }
3. **T → T * F** { T.val = T.val * F.val }
4. **T → F** { T.val = F.val }
5. **F → id** { F.val = id.lexval }

Semantic analysis for (S = 2+3*4)



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's 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 the left to right fashion for computing the translation rules of our example.



S–attributed and L–attributed SDTs in Syntax directed translation

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 then 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).

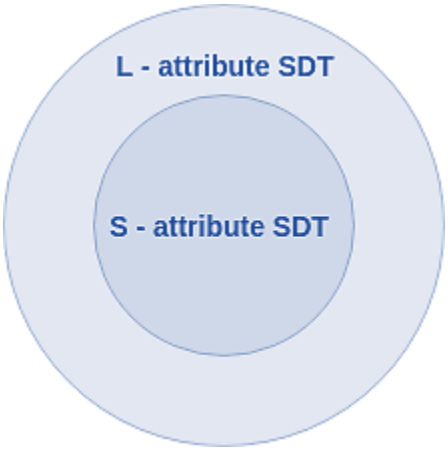
For example, let's say A -> BC is a production of a grammar and B's attribute is dependent on A's attributes or C's attributes then it will be inherited attribute.

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.



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.

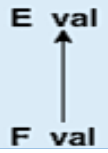
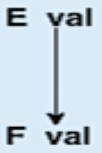
Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

Semantic actions are placed anywhere in RHS.

For example: A -> XYZ {Y.S = A.S, Y.S = X.S, Y.S = Z.S}
is not an L-attributed grammar since Y.S = A.S and Y.S = X.S are allowed but Y.S = Z.S violates the L-attributed SDT definition as attributed is inheriting the value from its right sibling.

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

The comparison between these two attributes are given below:

S.NO	Synthesized Attributes	Inherited Attributes
1.	An attribute is said to be Synthesized attribute if its parse tree node value is determined by the attribute value at child nodes.	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.
2.	The production must have non-terminal as its head.	The production must have non-terminal as a symbol in its body.
3.	A synthesized attribute at node n is defined only in terms of attribute values at the children of n itself.	A Inherited attribute at node n is defined only in terms of attribute values of n's parent, n itself, and n's siblings.
4.	It can be evaluated during a single bottom-up traversal of parse tree.	It can be evaluated during a single top-down and sideways traversal of parse tree.
5.	Synthesized attributes can be contained by both the terminals and non-terminals.	Inherited attributes can't be contained by both, It is only contained by non-terminals.
6.	Synthesized attribute is used by both S-attributed SDT and L-attributed STD.	Inherited attribute is used by only L-attributed SDT.
7.	**EX:-** **E.val -> F.val**  	**EX:-** **E.val = F.val** 

LECTURE -4- : SEMANTIC ANALYSIS IN COMPILER DESIGN

Semantic Analysis is the third phase of Compiler. Semantic Analysis makes sure that declarations and statements of program are semantically correct. It is a collection of procedures which is called by parser as and when required by grammar. Both syntax tree of previous phase and symbol table are used to check the consistency of the given code. Type checking is an important part of semantic analysis where compiler makes sure that each operator has matching operands.

Semantic Analyzer:

It uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition. It gathers type information and stores it in either syntax tree or symbol table. This type information is subsequently used by compiler during intermediate-code generation.

Semantic Errors:

Errors recognized by semantic analyzer are as follows:

1. Type mismatch
2. Undeclared variables
3. Reserved identifier misuse
4. Multiple declaration of variable in a scope.
5. Accessing an out-of-scope variable.
6. Actual and formal parameter mismatch.

Functions of Semantic Analysis:

1- Type Checking –

Ensures that data types are used in a way consistent with their definition.

2- Label Checking –

A program should contain labels references.

3- Flow Control Check –

Keeps a check that control structures are used in a proper manner.(example: no break statement outside a loop)

Example:

float x = 10.1;

float y = x*30;

In the above example integer 30 will be type casted to float 30.0 before multiplication, by semantic analyzer.

Static and Dynamic Semantics:

In many compilers, the work of the semantic analyzer takes the form of semantic action routines, invoked by the parser when it realizes that it has reached a particular point within a grammar rule.

Of course, not all semantic rules can be checked at compile time. Those that can are referred to as the static semantics of the language. Those that must be checked at run time are referred to as the dynamic semantics of the language. C has very little in the way of dynamic checks.

Examples of rules that other languages enforce at run time include the following:

- Variables are never used in an expression unless they have been given a value.
- Pointers are never dereferenced unless they refer to a valid object.
- Array subscript expressions lie within the bounds of the array.
- Arithmetic operations do not overflow.

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

If a semantic analyzer has a symbol table for each separate procedure, it can find semantic errors that occur because of the following mistakes:

- Names that aren't declared
- Operands of the wrong type for the operator they're used with
- Values that have the wrong type for the name to which they're assigned

If a semantic analyzer has a symbol table for the program as a whole, it can find semantic errors that occur because of the following mistakes:

- Procedures that are invoked with the wrong number of arguments
- Procedures that are invoked with the wrong type of arguments
- Function return values that are the wrong type for the context in which they're used

If a semantic analyzer has control-flow and data-flow information for each separate procedure, it can find semantic errors that occur because of the following mistakes:

- Code blocks that are unreachable
- Code blocks that have no effect
- Local variables that are used before being initialized or assigned
- Local variables that are initialized or assigned but not used

If a semantic analyzer has control-flow and data-flow information for the program as a whole, it can find semantic errors that occur because of the following mistakes:

- Procedures that are never invoked
- Procedures that have no effect

- Global variables that are used before being initialized or assigned
- Global variables that are initialized or assigned, but not used

Examples

1- the following code is correct

```
while (x <= 5)
   writeOut "OK";
   break;
;
```

Whereas the following one isn't, and should be rejected.

```
while (x <= 5)
   writeOut "OK";
;
break;
```

2-

```
x = 3;
z = "abc";
y = x + z;
```

The three lines above should also generate a compilation error. The reason is that the operator + is used with a int type (x) and a string type (z). Even though this kind of operation may be allowed in some languages.

LECTURE -5- : SEMANTIC ANALYSIS (TYPE CHECKING)

A semantic analyzer checks the source program for semantic errors. Type-checking is an important part of semantic analyzer. ***Type checking*** is the process of verifying and enforcing constraints of types in values and attempts to catch programming errors based on the theory of types.

Two types of semantic Checks are performed within this phase these are:-

1. Static Semantic Checks are performed at compile time like:-
   - Type checking.
   - Every variable is declared before used.
   - Identifiers are used in appropriate contexts.
   - Check labels
2. Dynamic Semantic Checks are performed at run time, and the compiler produces code that performs these checks:-
   - Array subscript values are within bounds.
   - Arithmetic errors, e.g. division by zero.
   - A variable is used but hasn't been initialized.

Three kinds of languages:

1- Statically(typed: All or almost all checking of types is done as part of compilation (C, Java)
2- Dynamically(typed: Almost all checking of types is done as part of program execution (Scheme)
3- Un-typed: No type checking (machine code).

NOTE: Some programming languages such as C will combine both static and dynamic typing i.e, some types are checked before execution while others during execution.

The design of type checker depends on:

1- Syntactic structure of language constructor.
2- The type expression of language.
3- The rules of the assigning types to construct.

Type Expression and Type Systems

Type Expression

The type of a language construct will be denoted by a *type expression*. A type expression is either a **basic type** or is formed by applying an operator called a **type constructor** to other type expressions.

1- Basic type
   • Integer: 7, 34, 909.
   • Floating point: 5.34, 123, 87.
   • Character: a, A.
   • Boolean: not, and, or, xor.
2- Type constructor
   - Arrays: If T is a type expression, then array (I, T) is a type expression denoting the type of an array with elements of type T and index set I.
   - Products: If T1 and T2 are type expressions, then their Cartesian product T1×T2 is a type expression.
   - Records: The type of a record is in a sense the product of the types of its fields. The difference between a record and a product is that the fields of a record have names.
   - Pointers: If T is a type expression, then pointer (T) is a type expression denoting the type pointer to an object of type T.
   - Functions: Functions take values in some domain and map them into value in some range.

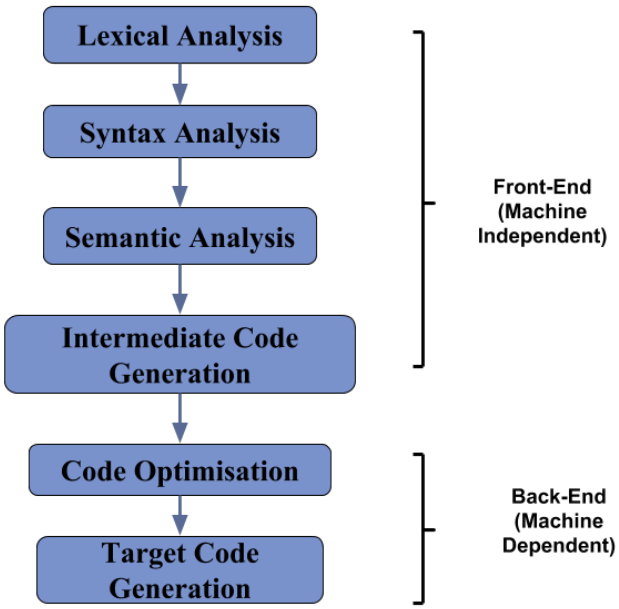
Type System

Collection of rules for assigning types expression. In most languages, types system are:

1- **Basic types** are the atomic types with no internal structure as far as the programmer is concerned (int, char, float,….).
2- **Constructed types** are arrays, records, and sets. In addition, pointers and functions can also be treated as constructed types.
3- **Type Equivalence:**
    - Name equivalence: Types are equivalence only when they have the same name.
    - Structural equivalence: Types are equivalence when they have the same structure.
    - Example: In C uses structural equivalence for structs and name equivalence for arrays/pointers.

LECTURE -6-: INTERMEDIATE CODE GENERATION

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).



The benefits of using machine independent intermediate code are:

- If a compiler translates the source language to its target machine language without having the ability to generate intermediate code, so for each new machine a full native compiler is required.
- The intermediate code eliminates the need for a complete new compiler for every single machine by keeping the parsing part the same for all compilers.
- The second part of the compiler, the synthesis, is modified depending on the target machine.
- It becomes easier to apply source code changes to improve code performance by applying code optimization techniques on intermediate code.

If we generate machine code directly from source code then for n target machine we will have n optimizers and n code generators but if we will have a machine independent intermediate code, we will have only one optimizer. Intermediate code can be either language specific (e.g., Bytecode for Java) or language independent (three-address code).

The following are commonly used intermediate code representation:

1- Postfix Notation –

The ordinary (infix) way of writing the sum of a and b is with operator in the middle: a + b

The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.

Example –

The postfix representation of the expression (a – b) * (c + d) + (a – b) is:

ab – cd + *ab -+.

2- Three-Address Code –

A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form

x = y op z where x, y, z will have address (memory location).

Sometimes a statement might contain less than three references but it is still called three address statement.

Example – The three address code for the expression a + b * c + d :

T 1 = b * c

T 2 = a + T 1

T 3 = T 2 + d

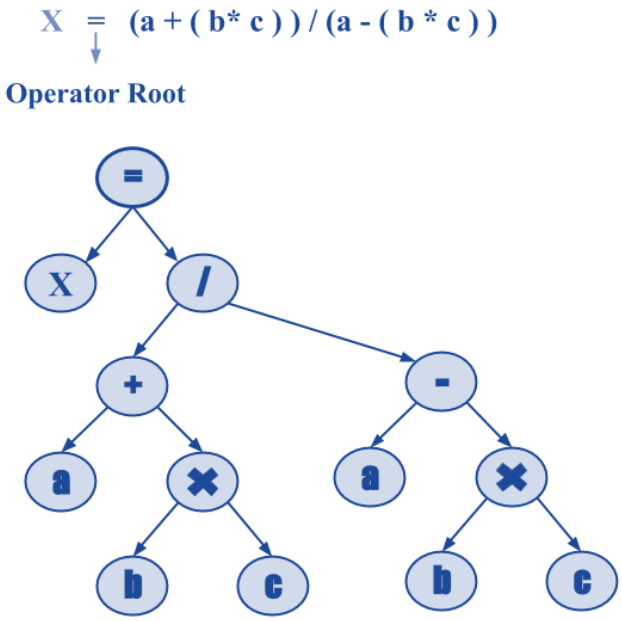
T 1 , T 2 , T 3 are temporary variables.

3- Syntax Tree –

Syntax tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Example –

x = (a + b * c) / (a – b * c)



Some of the basic operations which in the so program, to change in the assembly language

Operations	H.L.L	Assembly language
Math. operation	+, -, *, /	Add, sub, mult, div
Boolean operation	&, |, ~	And, or, not
Assignment	:=	Mov, LD, Store
Jump	Go to	JP, JN, JC
Conditional	If, Case	CMP
Loop instruction	For, Do, Repeat, While	These must have I.C

The operation which change H.L.L to Assembly language, is called the Intermediate code generation and there is the division operation come it, which mean every statement have a sing operation.

Example: X=A+B*C/D-Y*N
T1= B*C
T2=T1/D
T3=Y*N
T4=A+T2
T5=T4-T3

Example: Y= Cos(A*B)+C/N-X*P
T1=A*B
T2=Cos(T1)
T3=X*p
T4=C/N
T5=T2+T4
T6=T5-T3

If Condition Statement:

Example:
X=1;
If (X>Y)
{ A=A+1;
B=B-A+2;
}
P=P+1;

```
10 X=1

20 If X<=Y go to 60

30 A=A+1

40 T1= B-A

50 B= T1+2
60  P=P+1
```

Example:

```
X=1
If ((X>Y) && (Y>=2))
{
A=A+1
B=B-A+2
}
Else X=X+1;
P=P+2+X;
```

```
10 X=1
20 If X>Y go to 50
30 X= X+1
40 go to 100
50 If Y>=2 go to 70
60 go to 30
70 A=A+1
80 T1=B-A
90 B=T1+2
100 T2=P+2
110 P=T2+X
```

For - Loop

Example:

```
For (i=1; i<=10;i++)
X = X+ (i*Y);
```

```
10 i= 1
20 If i> 10 go to 70
30 T1= i* Y
40 X= X+T1
50 i= i+1
60 go to 20
70 end
```

Issues in the design of a code generator

Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.

1. **Input to code generator**

   The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

2. **Target program**

   The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.

   1. Absolute machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.
   2. Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.
   3. Assembly language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.

3. **Memory Management:**

   Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

4. **Instruction selection:**

Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

For example, the respective three-address statements would be translated into the latter code sequence as shown below:

```
P:=Q+R
S:=P+T

MOV Q, R0
ADD R, R0
MOV R0, P
MOV P, R0
ADD T, R0
MOV R0, S
```

Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

5. **Register allocation issues:**

Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two sub-problems:

1. During **Register allocation –** we select only those set of variables that will reside in the registers at each point in the program.
2. During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example

M a, b

These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

6. **Evaluation order:**
   The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.
7. **Approaches to code generation issues:**
   Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:
   - Correct
   - Easily maintainable
   - Testable
   - Efficient