Compiler Design

Phases of Compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation.

There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)

b. Synthesis (Machine Dependent/Language independent)

Above two phases are further classified as follows:

- Lexical Analysis: Converts a sequence of characters into words, or tokens
- Syntax Analysis: Converts a sequence of tokens into a parse tree
- Semantic Analysis: Manipulates parse tree to verify symbol and type information
- Intermediate Code Generation: Converts parse tree into a sequence of intermediate code instructions
- Optimization: Manipulates intermediate code to produce a more efficient program
- Final Code Generation: Translates intermediate code into final (machine/assembly) code.

Lexical Analysis:

- Convert from a sequence of characters into a (shorter) sequence of tokens
- Identify and categorize specific character sequences into tokens
- Skip comments & whitespace
- Handle lexical errors (illegal characters, malformed tokens)
- Efficiency is crucial; scanner may perform elaborate input buffering
- Tokens are specified as regular expressions.
- Lexical Analyzers are implemented by DFAs.

Symbol Table: It is the data structure used to store the attribute information of tokens. Symbol table can be accessed in all phases of compilers.

- Scanning: Insertion of new identifiers.
- Parsing: Access to the symbol table to ensure that an operand exists (declaration before use).
- Semantic analysis: Determination of types of identifiers from declarations, type checking to ensure that operands are used in type-valid contexts, Checking scope, visibility violations.
- Intermediate code generation: Memory allocation and relative address calculation.
- Optimization: All memory accesses through symbol table.
- Target code: Translation of relative addresses to absolute addresses in terms of word length, word boundary etc.

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are, 1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

LEXICAL ERRORS: Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there's no way to recognise a lexeme as a valid token for you lexer.

The Lexical Analyzer does the following:

- Reads input characters.
- Groups characters into meaningful units or ``words", producing data structures called tokens.
- Converts units to internal form, e.g. converts numbers to machine binary form.
- Serves as a front end for and provides input to the Parser.

Syntactic Analyzer

The Syntactic Analyzer (or Parser) will analyze groups of related tokens (``words") that form larger constructs (``sentences"). It will convert the linear string of tokens
into structured representations such as expression trees and program flow graphs

Semantic Analysis

This phase is concerned with the semantics, or meaning, of the program. Semantic processing is often performed along with syntactic analysis. It may include:

- Semantic error checking, such as checking for type errors.
- Insertion of extra operations, such as type coercion or code for array references.

	Scanning	Parsing
Task	determining the structure of tokens	determining the syntax or structure of a program
Describing Tools	regular expression	context-free grammar
Algorithmic Method	represent by DFA	top-down parsing bottom-up parsing
Result Data Structure	liner structure	parser tree or syntax tree, they are recursive

Ambiguity: Many derivations could correspond to a single parse tree.

- A grammar for which some sentence has more than one parse tree is ambiguous. This means also more than one leftmost derivation for a given string Also, more than one rightmost derivation for same string.

Left Recursion: A grammar is left recursive iff A $+\Rightarrow$ A α for some nonterminal A. A left recursive grammar can cause a top-down recursive descent parser to go into an infinite loop.

Non-left factored grammar: If more than one RHS of same non-terminal productions has common prefix then such grammar called as Non-left factored grammar. Due to common prefixes parsers may suffer with backtracking problem for selection of right production.

LL(k): k tokens of lookahead are used in LL parsing. The first L means that token sequence is read from left to right. The second L means a leftmost derivation is applied at each step.

An LL parser consists of:

- Parser stack that holds grammar symbols: non-terminals and tokens.
- Parsing table that specifies the parser action.
- Driver function that interacts with parser stack, parsing table and scanner.

Actions of LL(1) Parser:

- Match: to match top of parser stack with next input token
- Predict: to predict a production and apply it in a derivation step
- Accept: to accept and terminate the parsing of a sequence of tokens
- Error: to report an error message when matching or prediction fails

LL(1) Grammar: A grammar is LL(1) if for all production pairs $A \rightarrow \alpha \mid \beta$

1. $FIRST(\alpha) \cap FIRST(\beta) = \varphi$.
2. If $\beta \Rightarrow^* \varepsilon$, then no string derived from α begins with a terminal in FOLLOW(A). Similarly, if $\alpha \Rightarrow^* \varepsilon$.

LL(1) grammar properties:

- A left recursive grammar cannot be a LL(1) grammar.
- A grammar is not left factored, it cannot be a LL(1) grammar
- An ambiguous grammar cannot be a LL(1) grammar.

First(X):

- If X is terminal, $FIRST(X) = \{X\}$.
- If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).
- If X is a non-terminal, and $X \rightarrow Y_1\ Y_2 \ldots Y_k$ is a production, and ε is in all of $FIRST(Y_1), \ldots, FIRST(Y_k)$, then add ε to FIRST(X).
- If X is a non-terminal, and $X \rightarrow Y_1\ Y_2 \ldots Y_k$ is a production, then add a to FIRST(X) if for some i, a is in $FIRST(Y_i)$, and ε is in all of $FIRST(Y_1), \ldots, FIRST(Y_{i-1})$.

Follow(X):

- If \$ is the input end-marker, and S is the start symbol, $\$ \in FOLLOW(S)$.
- If there is a production, $A \rightarrow \alpha B \beta$, then $(FIRST(\beta) - \varepsilon) \subseteq FOLLOW(B)$.
- If there is a production, $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where $\varepsilon \in FIRST(\beta)$, then $FOLLOW(A) \subseteq FOLLOW(B)$.

Bottom Up Parsing: Bottom-up parsing is also known as shift-reduce parsing because its two main actions are shift and reduce.

A bottom-up parser tries to find the right-most derivation of the given input in the reverse order.

- A shift-reduce parser tries to reduce the given input string into the starting symbol.
- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

Handle: A "handle" of a string is a substring that matches the RHS of a production and whose reduction to the non-terminal (on the LHS of the production) represents one step along the reverse of a rightmost derivation toward reducing to the start symbol.

There are four possible actions of a shift-parser action:

- Shift: The next input symbol is shifted onto the top of the stack.
- Reduce: Replace the handle on the top of the stack by the non-terminal.
- Accept: Successful completion of parsing.
- Error: Parser discovers a syntax error, and calls an error recovery routine.

LR(k): 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

- LR(0) : LR parsing with no lookahead token to make parsing decisions.
- SLR(1) : Simple LR, with one token of lookahead.
- LR(1) : Canonical LR, with one token of lookahead.
- LALR(1) : Lookahead LR, with one token of lookahead

LR(0) Conflicts:

- Shift-reduce conflict: Both Shift and Reduced items are present in the same state. It means parser can shift and can reduce.
- Reduce-reduce conflict: Two or more reduced items are present in the same item.

SLR(1): A grammar is SLR(1) if two conditions are met in every state.

- If $A \rightarrow \alpha \bullet x \gamma$ and $B \rightarrow \beta \bullet$ then token $x \notin FOLLOW(B)$, otherwise shift-reduce conflict occurs in that state.
- If $A \rightarrow \alpha \bullet$ and $B \rightarrow \beta \bullet$ then $FOLLOW(A) \cap FOLLOW(B) = \emptyset$, otherwise reduce-reduce conflict occurs in that state.

CLR(1): A grammar is LR(1) if the following two conditions are met.

- If a state contains $A \rightarrow \alpha \bullet x \gamma$, { a } and $B \rightarrow \beta \bullet$, { b } then x is not present in {b}, Otherwise produces SR conflict in LR(1).
- If a state contains $A \rightarrow \alpha \bullet$, { a } and $B \rightarrow \beta \bullet$, { b } then {a} and {b} has no common element, otherwise produces RR conflict.

LALR(1): LALR(1) can be obtained from LR(1) by Merging LR(1) states that have same LR(0) items, obtaining the union of the LR(1) lookahead tokens**.**

Number of States:

- LR(0), SLR(1), LALR(1) contains same number of states for any given grammar.
- CLR(1) can contain either equal or greater number of states than LALAR(1).

Size of Tables:

- Size of LR(0) table = Size of SLR(1) table = Size of LALR(1) table

Number of Conflicts:

- LR(0) conflicts >= SLR(1) Conflicts >= LALR(1) Conflicts >= LR(1) Conflicts

Set of LR grammars:

- LR(0) ⊂ SLR(1) ⊂ LALR(1) ⊂ LR(1)

Expressive Powers:

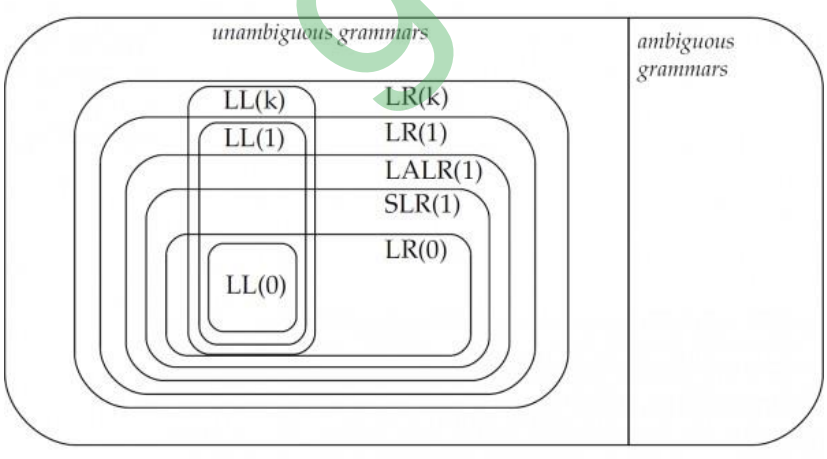
- LR(0) < SLR(1) < LALR(1) < LR(1)

Operator Grammar: In an operator grammar, no production rule can have ε at the right side, and two adjacent non-terminals at the right side.

Precedence Relations: In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

- a <. b a has lower precedence than b
- a .> b a has lower precedence than b
- a = b a has lower precedence than b

Relationship of LL and LR grammars:



Intermediate Code: Intermediate codes are machine independent codes, but they are close to machine instructions.

- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.

- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.

1. Syntax trees can be used as an intermediate language.
2. Postfix notation can be used as an intermediate language.
3. Three-address code (Quadruples) can be used as an intermediate language

Example:

A = -B * (C + D)

Three-Address code is as follows:

T1 = -B
T2 = C + D
T3 = T1 * T2
A = T3

Quadruple:

	Operator	Operand 1	Operand 2	Result
(1)	-	B		T1
(2)	+	C	D	T2
(3)	*	T1	T2	T3
(4)	=	A	T3	

Triple:

	Operator	Operand 1	Operand 2
(1)	-	B	
(2)	+	C	D
(3)	*	(1)	(2)
(4)	=	A	(3)

Indirect Triple:

	Statement
(0)	(56)
(1)	(57)
(2)	(58)
(3)	(59)

	Operator	Operand 1	Operand 2
(56)	-	B	
(57)	+	C	D
(58)	*	(56)	(57)
(59)	=	A	(58)

Three Address Code for A<B:

(1) if A < B goto (4)
(2) T := 0
(3) goto (5)
(4) T := 1
(5)

Syntax Directed Definition: Specifies the values of attributes by associating semantic rules with the productions. SDD is easier to read;

Syntax Directed Translation scheme: embeds program fragments (also called semantic actions) within production bodies. The position of the action defines the order in which the action is executed (in the middle of production or end). easy for specification. SDT scheme can be more efficient, and easy for implementation.

Attribute: Attribute is any quantity associated with a programming construct. Example: data types, line numbers, instruction details.

Two kinds of attributes: for a non-terminal A, at a parse tree node N.

- A synthesized attribute: defined by a semantic rule associated with the production at N. defined only in terms of attribute values at the children of N and at N itself.
- An inherited attribute: defined by a semantic rule associated with the parent production of N. defined only in terms of attribute values at the parent of N siblings of N and at N itself.

L-Attributed SDD: It contain both synthesized and restricted inherited attributes. SDT can contain translations anywhere in RHS of production.

S-Attributed SDD: It contain only synthesized. SDT can contain translations only at the end of production. Evaluation order depends on LR parser evaluation.

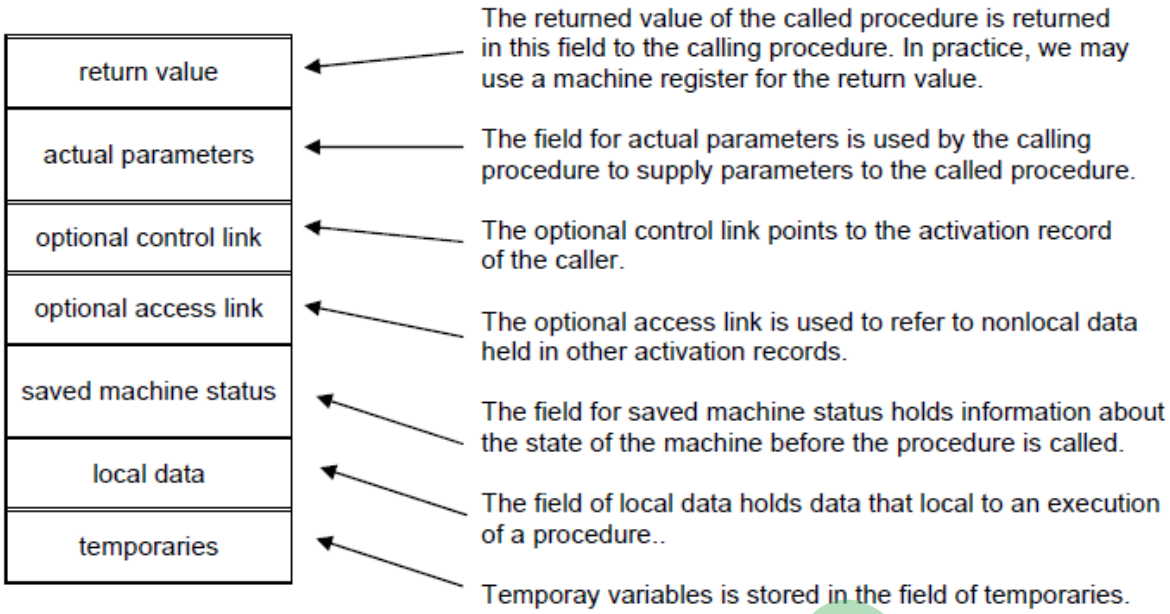
SDT to count the number of balanced parenthesis and bracket pairs in the expression:

exp -> epsilon { exp.trans = 0 }
-> (exp) { exp_1.trans = exp_2.trans + 1 }
-> [exp] { exp_1.trans = exp_2.trans + 1 }

Activation Record:

Information needed by a single execution of a procedure is managed using a contiguous block of storage called **activation record**.

An activation record is allocated when a procedure is entered, and it is de-allocated when that procedure exited.



Control Stack:

The flow of the control in a program corresponds to a depth-first traversal of the activation tree that:
– starts at the root, visits a node before its children, and recursively visits children at each node an a left-to-right order.
A stack (called **control stack**) can be used to keep track of live procedure activations.
– An activation record is pushed onto the control stack as the activation starts.
– That activation record is popped when that activation ends.
When node n is at the top of the control stack, the stack contains the nodes along the path from n to the root.

Storage Allocation Strategies: Static allocation lays out storage for all data objects at compile time.

- Size of object must be known and alignment requirements must be known at compile time.
- No recursion.
- No dynamic data structure

Stack Allocation Strategies: Stack allocation manages the run time storage as a stack. The activation record is pushed on as a function is entered. The activation record is popped off as a function exits.
Heap allocation: Allocates and deallocates storage as needed at runtime from a data area known as heap.

- Most flexible: no longer requires the activation of procedures to be LIFO.
- Most inefficient: need true dynamic memory management.