CS3501- Compiler Design

Unit - I Introduction to Compilers & lexical Analysis

Introduction. Translators - Compilation & Interpretation - Language Processors - The Phases of Compiler - lexical Analysis - Role of Lexical Analyzer - Input Buffering - Specification of Tokens - Recognition of Tokens - Finite Automata - Regular Expressions to Automata NFA, DFA - Minimizing DFA - language for Specifying lexical Analyzers - lex tool.

Unit - II Syntax Analysis

Role of Parser - Grammars - Context-free grammars - Writing a grammar Top Down Parsing - General Strategies - Recursive Descent Parser Predictive Parser - LL(1) - Parser - Shift Reduce Parser - LR Parser - LR (0) item Construction of SLR Parsing Table - Introduction to LALR Parser - Error Handling & Recovery in Syntax Analyzer - YACC tool - Design of a Syntax Analyzer for a Sample language.

Unit - III Syntax Directed Translation & Intermediate code Generation.

Syntax Directed Definitions - Construction of Syntax Tree - Bottom-up Evaluation of S-Attribute Definitions - Design of Predictive translator - Type Systems - Specification of a Simple type checker - Equivalence of Type Expressions - Type conversions. Intermediate languages: Syntax Tree, Three Address code, Types & declarations, Translation of Expressions, Type checking, Back Patching.

Unit - IV Run-Time Environment & Code generation.

Runtime Environments - Source language issues - Storage organization - Storage Allocation strategies: Static, Stack & Heap allocation - Parameter Passing - Symbol tables - Dynamic storage Allocation - Issues in the Design of a Code generator - Basic Blocks & Flow graphs - Design of a Simple code Generator - Optimal code generation for Expressions - Dynamic Programming code Generation.

Unit - V Code Optimization

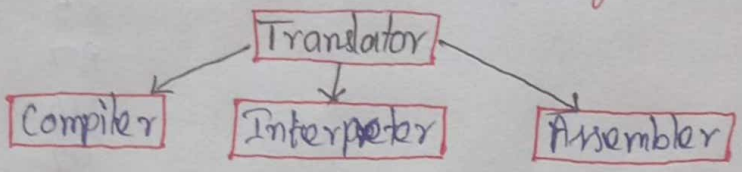
Principal Sources of Optimization - Peep-hole optimization - DAG - Optimization of Basic Blocks - Global Data Flow Analysis - Efficient Data Flow Algorithm - Recent trends in compiler Design.

Unit-I.

Translator

A translator translates one language into another language. Computers understand machine language, & users understand high-level language called source code. The translator takes input as the source code (high-level language). & gives output as the machine code. A translator is a program that converts instructions written in source code to object code or from high-level to machine language.

Translator in Compiler Design

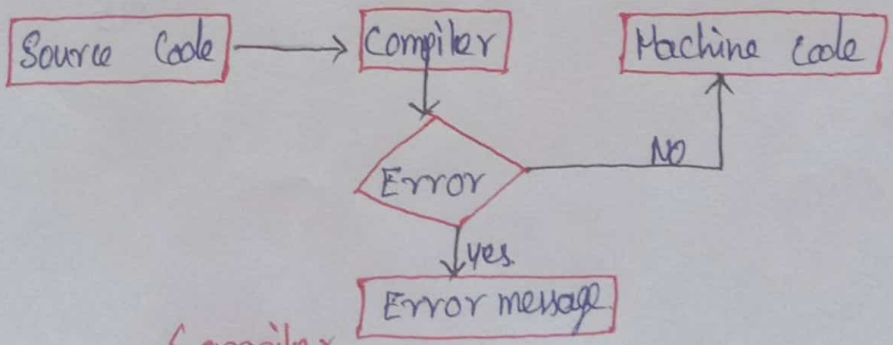


Types of translators that will do the same in a computer & help to communicate from user to computer.

* Compiler
* Interpreter
* Assembler

Compiler

The compiler is a translator that converts source code from high-level language to machine code. Compilers are much faster than interpreters, but error debugging is a little difficult process. languages that use compilers are c, c++, java. It works in three stages are as follows:

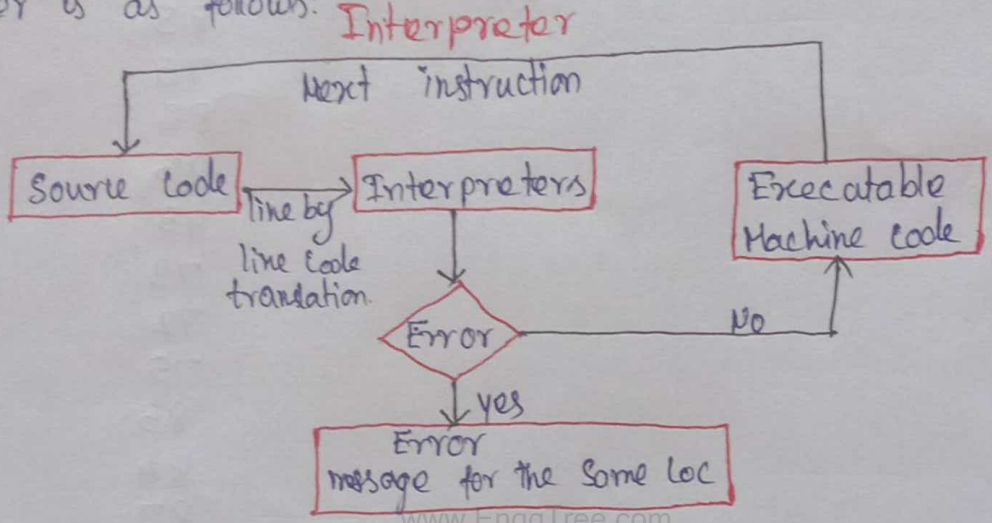


Compiler

1. It takes the Source code as input. 2. It checks the source code & validates it. If some error occurs (when the rules of the language is not followed), it will send an error message. 3. If no error occurs, it sends the machine code as output to the machine.

Interpreter

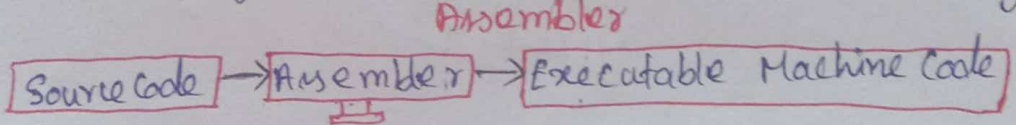
Interpreter changes the source code to machine code, but the process is a bit changed here than in compilers. An interpreter is a translator that effectively accepts a source program & executes it directly without producing any object code first. It does this by fetching the source code program instructions one by one, analyzing them one by one, & then executing them one by one. The process of translating the source code to machine code using an interpreter is as follows:



1. It takes the source code as input. 2. It checks the source code line-by-line & validates it. If some error occurs (when the rule of the language is not followed), it will stop the code execution & send an error. 3. If no error occurs it sends the machine code as output to the machine.

Assembler

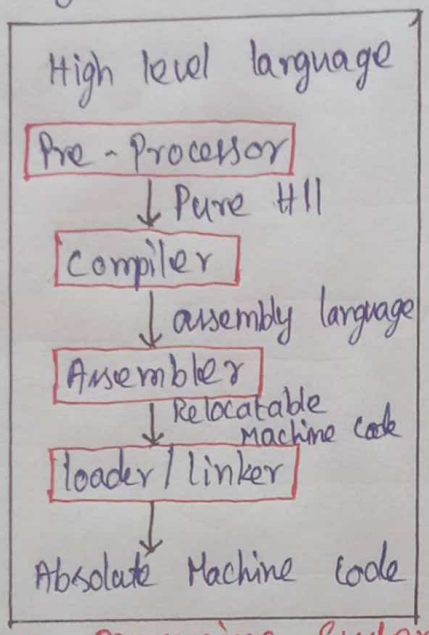
Assembler is said to be the fastest translator among the three, as the assembler translates the low-level assembly code to machine code. It translates by evaluating the symbols in the source code & generates the machine code accordingly. If an assembler does this in one scan, it is called a single-pass assembler. If it takes multiple scans, it is called a multi-pass assembler. The process of translating the source code to machine code using assemblers is as follows:



1. It defines symbols of the assembly code. 2. It processes some pseudocode operations. 3. Generates object code.

language Processors

language processing system, the source code is first preprocessed. The modified source program is processed by the compiler to form the target assembly program which is then translated by the assembler to create relocatable object codes that are processed by linker & loader to create the target program.



language processing system

High-level language

If a program includes #define or #include directives, including #include or #define, it is known as HLL.

Pre-Processor

The pre-processor terminates all the #include directives by containing the files named file inclusion and all the #define directives using macro expansion. A pre-processor can implement the following functions.

Macro Processing :-

A preprocessor can enable a user to define macros that are shorthands for higher constructs.

File Inclusion :-

A preprocessor can include header files into the program text.

Rational preprocessor :-

These preprocessors augment earlier languages with additional current flow-of-control & data structuring facilities.

language Extensions :-

These preprocessors try to insert capabilities to the language by

specific amounts to construct in macro.

Pure HII

It means that the program will not contain any # tags. Those # tags are also known as preprocessor directives.

Assembler

Assembler is a program that takes as input an assembly language program & changes it into its similar machine language code.

Assembly language

It is an intermediate state that is a sequence of machine instructions & some other beneficial record needed for implementation. It neither in the form of 0's & 1's.

Relocatable Machine Code

It means that can load that machine code at any point in the compu & it can run. The address inside the code will be so that it will maintain the code movement.

loader / linker

This is a code that takes as input a relocatable program & Compiles the library functions, relocatable object records, & creates its similar absolu machine program.

The Phases of Compiler

The Compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source progr & feeds its output to the next phase of the compiler. Basically have two phases of Compilers namely the Analysis phase & Synthesis phase.

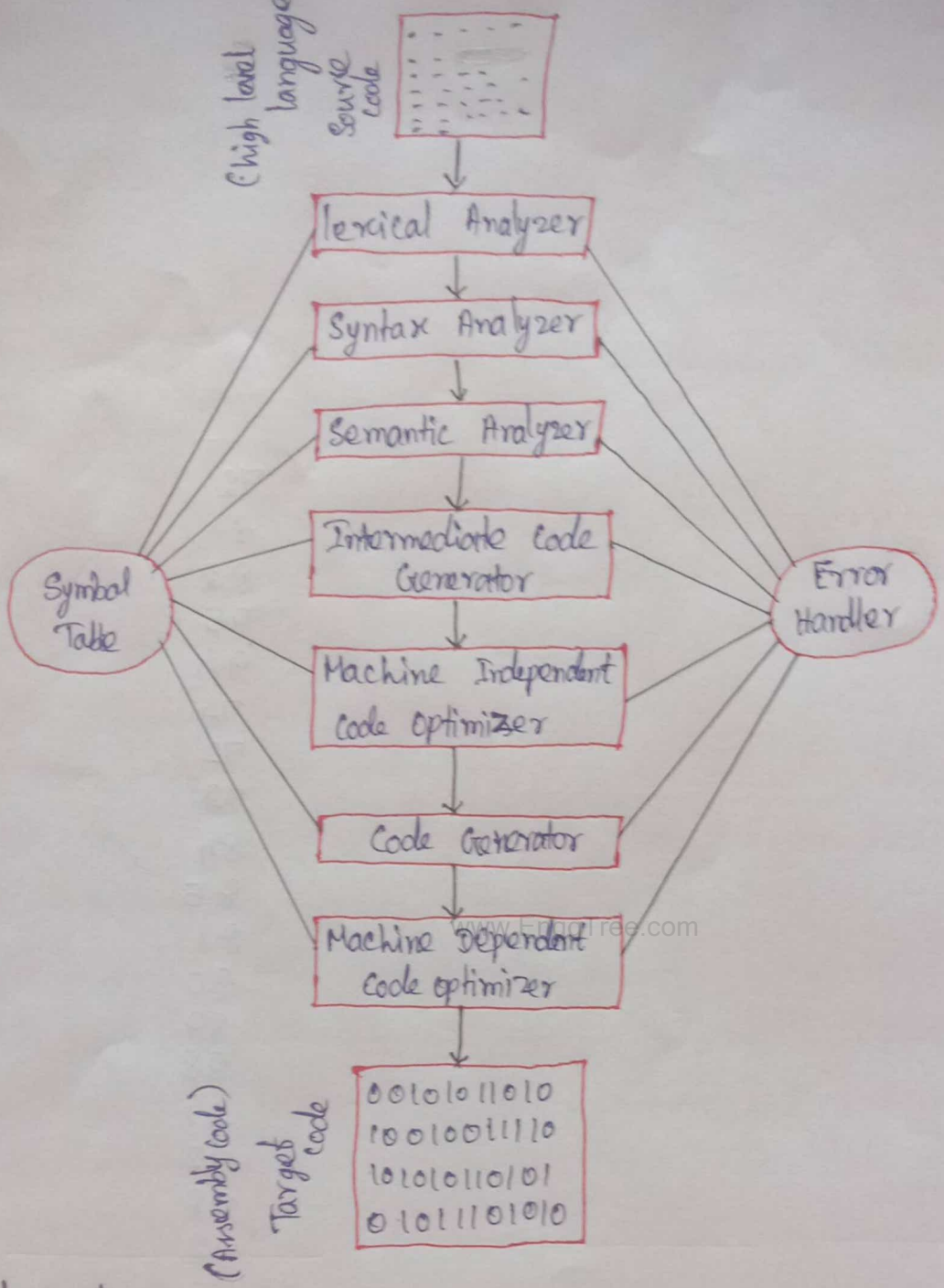
Analysis Phase

The analysis phase creates an intermediate representation from the given source code.

Synthesis Phase

The synthesis phase creates an equivalent target program from the intermediate represent

Phases of Compiler



lexical Analysis

The first phase of a compiler is lexical analysis, also known as scanning. This phase reads the source code & breaks it into a stream of tokens, which are the basic units of the programming language. The tokens are then passed on to the next phase for further processing.

Syntax Analysis

The second phase of a compiler is syntax analysis, also known as parsing. This phase takes the stream of tokens generated by the lexical analysis phase & checks whether they conform to the grammar of the programming language. The output of this phase is usually an Abstract Syntax Tree.

Semantic Analysis

The third phase of a compiler is semantic analysis. This phase checks whether the code is semantically correct, that is whether it conforms to the language type system & other semantic rules.

Intermediate Code Generation

The fourth phase of a compiler is intermediate code generation. This phase generates an intermediate representation of the source code that can be easily transl into machine code.

Code Optimization

The fifth phase of a compiler is optimization. This phase applies various optimizati techniques to the intermediate code to improve the performance of the generated machine code.

Code Generation

The final phase of a compiler is code generation. This phase takes the optimized intermediate code & generates the actual machine code that can be executed by the target hardware.

Symbol Table

It is a data structure being used & maintained by the compiler, consisting of all the identifier's names along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

The analysis of a source program is divided into mainly three phases. They are:

linear Analysis

This involves a scanning phases where the stream of characters is read from left to right. It is then grouped into various tokens having a collective meaning.

Hierarchical Analysis

This analysis phase, based on a collective meaning, the tokens are categorized hierarchically into nested groups.

Semantic Analysis

This phase is used to check whether the components of the source program are meaningful or n

The compiler has two modules namely the front end & the back end. Fron end constitutes the lexical analyzer, semantic analyzer, Syntax analyzer & intermediate cod [illegible] And the rest are [illegible] back end.

Error Handlers

It is invoked when a flaw error in the source program is detected. The output of LA is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as expression. Expression may further be combined to form statements. The Syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

The parser has two function. It checks if the tokens from lexical analyzer, occur in pattern that are permitted by the specification for the source language. It also imposes on tokens a tree like structure that is used by the sub-sequent phases of the compiler.

---

lexical Analysis

lexical Analysis in compiler is the first step in the analysis of the source program. The lexical analysis reads the input stream from the source program character by character & produces the sequence of tokens. These tokens are provided as an input to the parser for parsing.

Terminologies in lexical Analysis.

lexeme

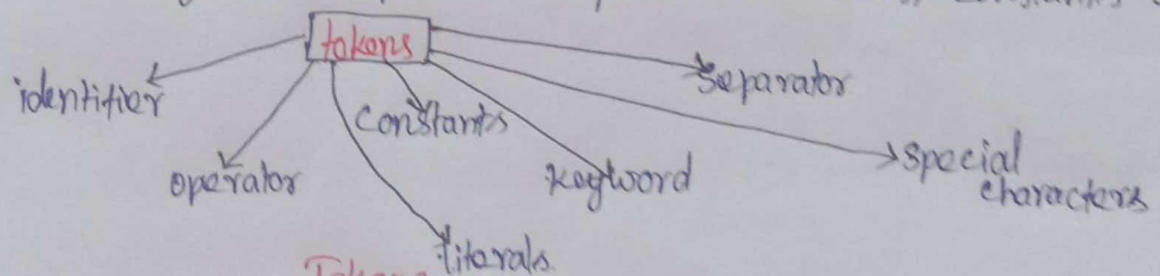
lexeme can be defined as a sequence of characters that forms a pattern & can be recognized as a token.

Pattern

After identifying the pattern of lexeme one can describe what kind of token can be formed. Such as the pattern of some lexeme forms a keyword, the pattern of some lexe forms an identifier.

Token

A lexeme with a valid pattern forms a token. In lexical analysis, a valid token can be identifiers, keywords, separators, special characters, constants & operators

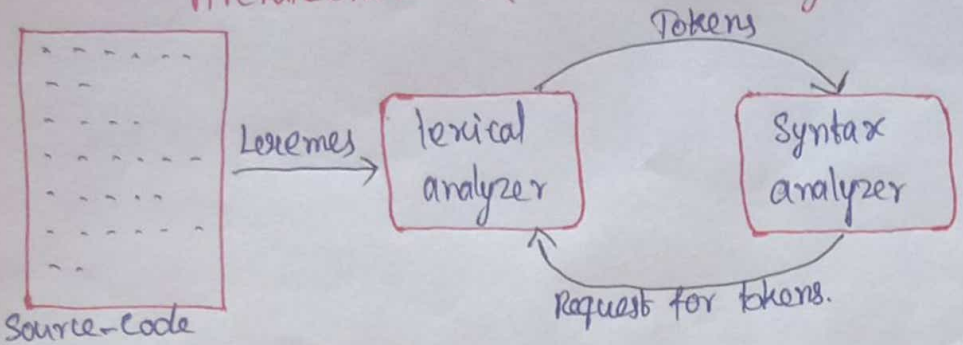


Tokens

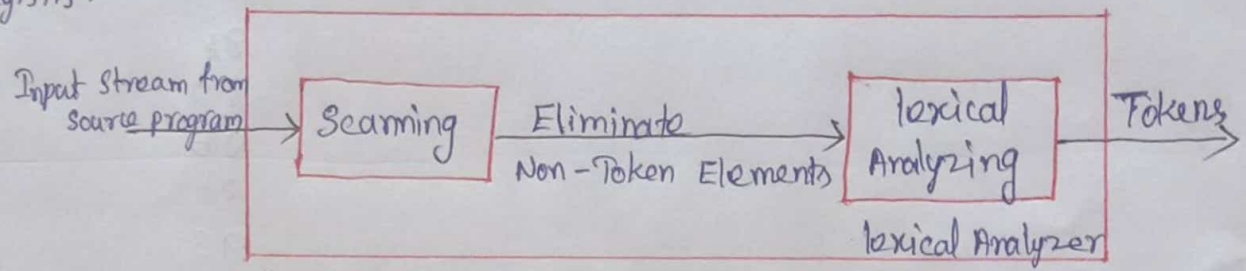
Architecture of lexical Analyzer

To read the input character in the source code & produce a token is the most impor task of a lexical analyzer. The lexical analyzer goes through with the entire source code & identifies each token one by one.

Architecture of lexical Analyzer



The lexical analyzer consists of two consecutive processes that include scanning & lexical analysis.



Two Cascading Processes of lexical Analyzer's.

Scanning

The scanning phase - only eliminates the non-token elements from the source program such as eliminating comments, compacting the consecutive white spaces etc.

lexical Analysis

lexical analysis phase performs the tokenization on the output provided by the scanner & thereby produces tokens.

Example of lexical Analysis.

```
Printf ("Value of i is %d", i);
```

Now let us try to find tokens out of this Input stream.

Keyword → printf

Special Character → (

Literal → "Value of i is %d"

Separator → ,

Identifier → i

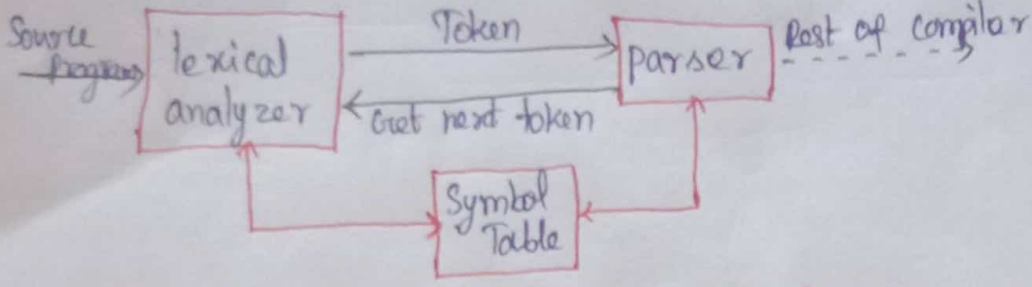
Special Character →)

Separator → ;

Roles of lexical Analyzer

The lexical analyzer is responsible for removing the white spaces & comments from the source program. It corresponds to the error messages with the saure program.

It helps to identify the tokens. The input characters are read by the lexical analyzer from the source code.



lexical Analyzer's Intraction with Parser.

Input preprocessing

This stage involves cleaning up the input text and preparing it for lexical analysis. This may include removing comments, whitespace, & other non-essential characters from the input text.

Tokenization

The process of breaking the input text into a sequence of tokens. this is usually done by matching the characters in the input text against a set of patterns or regular expressions that define the different types of tokens.

Token Classification

The lexer determines the type of each token. For example, in a programming language, the lexer might classify keywords, identifiers, operators, & punctuation symbols as separate token types

Token Validation

The lexer checks that each token is valid according to the rules of the programming language. For example, it might check that a variable name is a valid identifier, or that an operator has the correct syntax.

Output Generation

The lexer generates the output of the lexical analysis process, which is typically a list of tokens. This list of tokens can then be passed to the next stage of compilation or interpretation.

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

A statement through lexical analyzer -a=b+c; It will generate token sequence like this: id=id+id; where each id refers to it's variable in the symbol table referencing all details.

Input Buffering

lexical Analysis has to access secondary memory each time to identify tokens. It is time consuming & costly. So, the input strings are stored into a buffer & then scanned by lexical analysis.

lexical analysis scans input string from left to right one character at a time to identify tokens. It uses two pointers to scan tokens:

Begin Pointer (bptr).

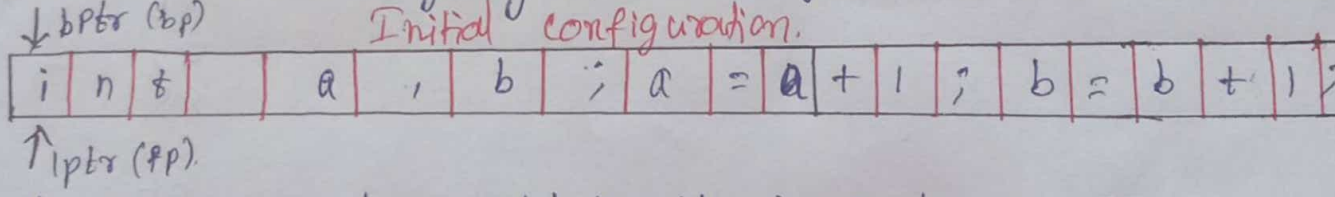
It points to the beginning of the string to be read.

look ahead Pointer (lptr).

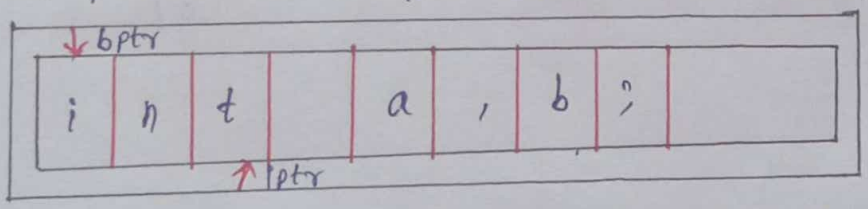
It moves ahead to search for the end of the token.

Example for statement int a, b;

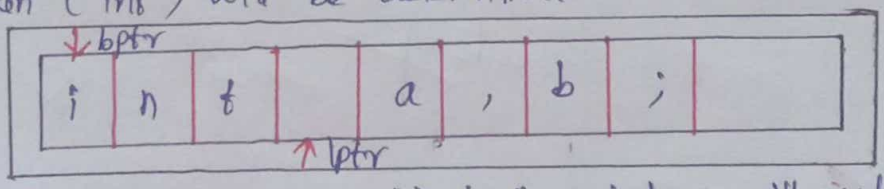
Both pointers start at the beginning of the string, which is stored in the buffer.



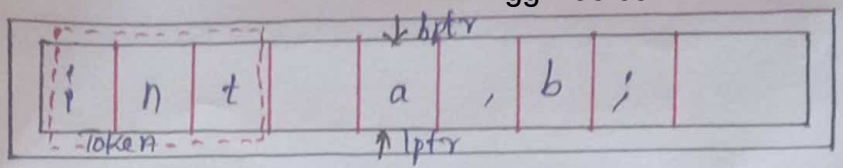
look ahead pointer scans buffer until the token is found.



The character ("blank space") beyond the token ("int") have to be examined before the token ("int") will be determined.

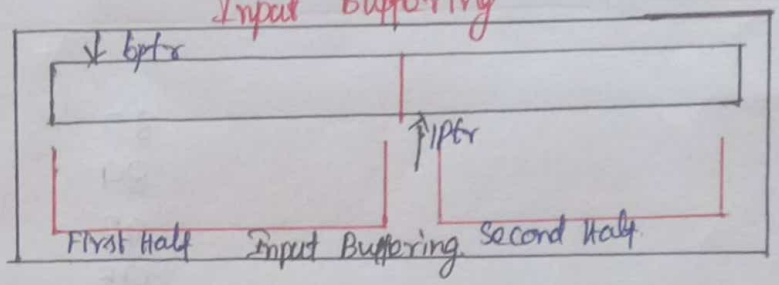


After processing token ("int") both pointers will set to the next token ('a') & this process will be repeated for the whole program.



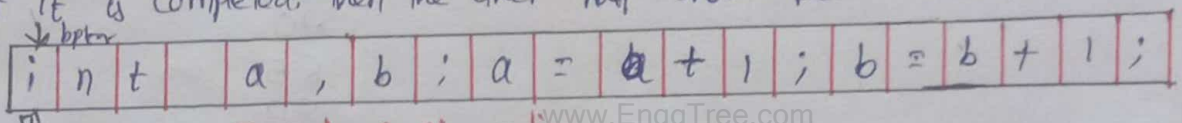
A buffer can be divided into two halves. If the look ahead pointer moves towards halfway in first half, the second half is filled with new characters to be read. If the look ahead pointer moves towards the right end of the buffer of the second half, the first half will be filled with new characters, & it goes on.

Input Buffering

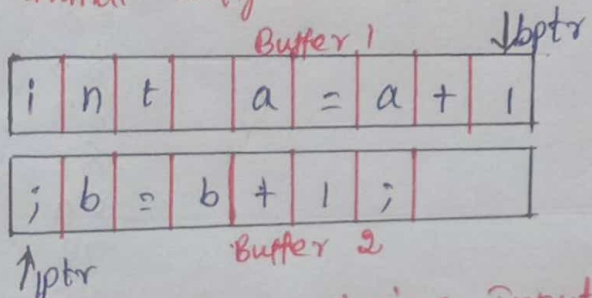


Sentinels

Sentinels are used to making a check, each time when the forward pointer is converted, a check is completed to provide that one half of the buffer has not converted off if it is completed, then the other half should be reloaded.



Initial Configuration



Two buffer scheme storing Input Buffering

Buffer Pairs

A specialized buffering technique can decrease the amount of overhead, which is needed to process an input character in transferring characters. It includes two buffers, each includes N- character size which is reloaded alternatively.

There are two pointers such as the lexeme Begin & forward are supported. lexeme Begin points to the starting of the current lexeme which is discovered. Forward scans ahead before a match for a patterne are discovered. Before a lexeme is initiated, lexeme begin is set to the character directly after the lexeme which is only constructed, & forward is set to the character at its right end.

Specification of Tokens

Specification of tokens depends on the pattern of the lexeme. Here using regular expressions to specify the different types of patterns that can actually form tokens. Although the regular expressions are inefficient in specifying all the patterns forming tokens. yet it reveals almost all types of pattern that forms a token.

There are three specifications of token.

1. String
2. language
3. Regular Expressions.

Strings.

Strings are a finite set of symbols or characters. These symbols can be a digit or an alphabet. There is also an empty string which is denoted by ε.

Operations on Strings.

The following string-related terms are commonly used.

Prefix

A prefix of string s is any string obtained by removing zero or more symbols from the end of string s. For example, ban is a prefix of banana.

Suffix

A suffix of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, nana is a suffix of banana.

Substring

A substring of s is obtained by deleting any prefix & any suffix from s. For example, nan is a substring of banana.

The proper prefixes, suffixes & substrings

The proper prefixes, suffixes & substrings of a string s are those prefixes, suffixes, & substrings, respectively of s that are not ε or not equal to s itself.

A <u>subsequence</u> of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, baan is a subsequence of banana.

language

A language can be defined as a finite set of strings over some symbols or alphabets.

Operations on language

The following operations are performed on a language in the lexical analysis phase

Union

Union is one of the most common operations perform on a set. In terms of languages also, it will hold a similar meaning. Suppose there are two languages, L & S. Then the union of these two languages will be

L ∪ S will be equal to {x | x belongs to either L or S}.

ex: If L = {a, b} & S = {c, d} Then L ∪ S = {a, b, c, d}.

Concatenation

Concatenation links two languages by linking the strings from one language to all the strings of the other language. If there are two languages, L & S, then the concatenation of L & S will be LS equal to {ls | where l belongs to L & S belongs to S}.

ex: There are two languages, L & S, such that {L', L''} is the set of strings belonging to language L & S {S, S', "S'"} is the set of strings belonging to language Then the concatenation of L & S will be LS will be {L'S', L'S'', L''S', L''S''}

Kleene closure

Kleene closure of a language L is denoted by L^* provides a set of all the strings that can be obtained by concatenating L zero or more times.

if L = {a, b}.

then L^* = {ε, a, b, aa, bb, aaa, bbb, ...}

Positive closure

L^+ denotes the positive closure of a language L & provides a set of all the strings that can be obtained by concatenating L one or more times.

if L = {a, b}

then L^+ = {a, b, aa, bb, aaa, bbb, ...}

Regular Expression

Regular expressions are strings of characters that define a searching pattern with the help of which can form a language & each regular expression represents a language. A regular expression r can denote a language L(r) which can be built recursively over the smaller regular expression by rules.

Writing Regular Expressions.

Following symbols are used very frequently to write regular expressions.

The astarisk symbol (*)

It is used in our regular expression to instruct the compiler that the symbol that preceded the * symbol can be repeated any number of times in the pattern.

ex:

if the expression is ab^*c, then it gives the following string - ac, abc, abbc, abbbc, abbbbbc, ... & so on.

The plus Symbol (+)

It is used in our regular expression to tell the compiler that the symbol that preceded + can be repeated one or more times in the pattern.

ex:

if the expression is ab+c, then it gives the following string - abc, abbc, abbbc, abbbbbc, ... & so on.

Wildcard Character (.)

The . A symbol, also known as the wildcard character, is a character in our regular expression that can be replaced by another character.

Character Class

It is a way of representing multiple characters.

ex:

[a-z] denotes the regular expression a|b|c|d|.....|z.

The following rules are used to define a regular expression r over some alphabet Σ and the languages denoted by these regular expressions.

If ε a regular expression that denotes a language $L(\varepsilon)$. The language $L(\varepsilon)$ has a set of strings $\{\varepsilon\}$ which means that this language has a single string which is the empty string.

If there is a symbol 'a' in Σ, then 'a' is a regular expression that denotes a language $L(a)$. The language $L(a) = \{a\}$, that is the language has only one string of length, & the string holds 'a' in the first position.

Consider the two regular expressions r & s then

r|s denotes the language $L(r) \cup L(s)$.

(r)(s) denotes the language $L(r) \cdot L(s)$.

$(r)^*$ denotes the language $(L(r))^*$. & $(r)^+$ denotes the language $L(r)$.

Regular Set

A language that can be defined by a regular expression is called a regular set. If two regular expressions r & s denote the same regular set, they ar equivalent & write r = s. There are a number of algebric laws for regular expressions that can be used to manipulate into equivalent forms. For instance, r|s = s|r is commutative; r|(s|t) = (r|s)|t is associative.

Regular Definitions

Giving names to regular expression referred to as a regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

$\cdots$

$d_n \rightarrow r_n$

1. Each d_i is a distinct name
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \cdots d_{i-1}\}$.

ex:

Identifiers is the set of strings of letters & digits beginning with a letter. Regular definition for this set:

letter → A|B...|Z|a|b|...|z| digit → 0|1|...|9|

id → letter (letter|digit)*

Recognition of Tokens

Recognition of tokens in compiler design is a crucial step. It breaks down the source code into understandable parts, like words & symbols. This helps the compiler understand the code's structure & meaning. For this language fragment the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id & num.

Consider the following grammar fragment:

stmt → if expr then stmt

| if expr then stmt else stmt | ε

expr → term relop term

| term

term → id | num

if → if
then → then
else → else
relop → <|<=|=|<>|>|>=
id → letter (letter|digit)*
num → digit + (.digit +) ? (E (+|-) ? digit +) ?

Tokens obtained during lexical analysis are recognized by Finite Automata. Finite Automata is a simple idealized machine that can be used to recognize patterns within input taken from a character set or alphabet (denoted as c). The primary task of an finite automata is to accept or reject an input based on whether to defined pattern occurs within the input.

There are two notations for representing finite automata. They are.

1. Transition table
2. Transition Diagram

Transition Table

It is a tabular representation that lists all possible transitions for each state E& input symbol combination. It is the following token that gets returned to the Pars

lexemes	Token Name	Attribute value
Any ws	-	-
if	if	-
then	then	-
else	else	-
Any id	id	Pointer to table entry
Any number	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE

Assume lexemes are separated by white space, consisting of non null sequences of blanks, tabs, & newline. Our lexical analyzer will strip out whi

Space. It will do so by comparing a string against the regular definition ws, below.

Delim → blank | tab | newline

ws → delim

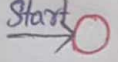
If a match for ws is found, the lexical analyzer does not return a token to the parser.

Transition Diagram

It is a directed labeled graph consisting of nodes & edges. Nodes represent states, while edges represent state transitions.

Components of Transition Diagram

1. One state is labelled the start state. It is the initial state of transition diagram where control resides when we begin to recognize a token.

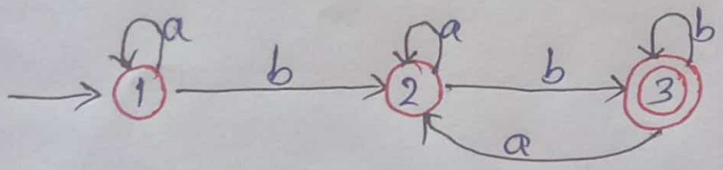


2. Position is a transition diagram are drawn circles & are called states.

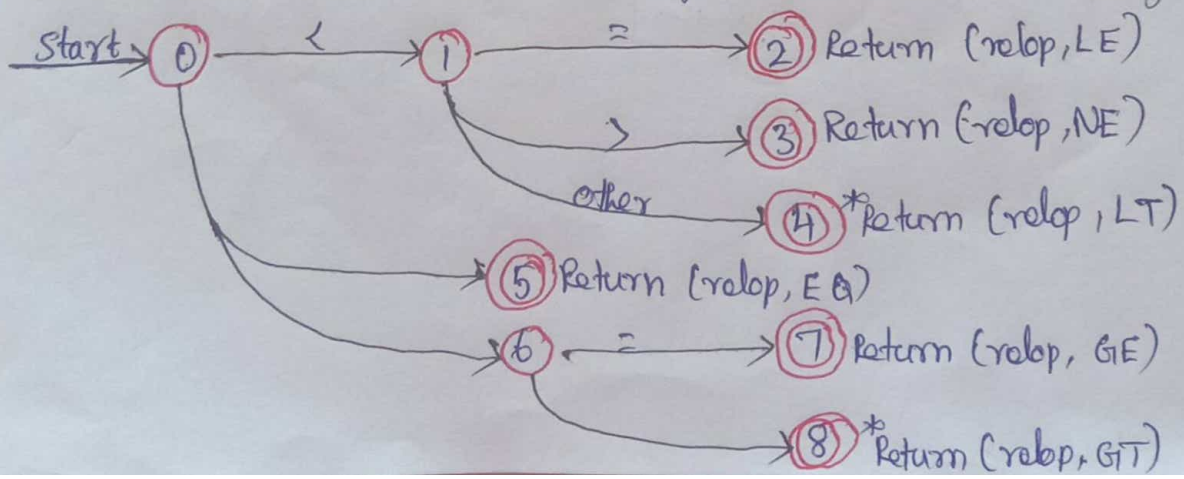
3. The States are connected by Arrows called edges. Labels on edges are indicating the input characters. →

4. zero or more final states or Accepting states are represented by double circle in which the tokens has been found.

ex

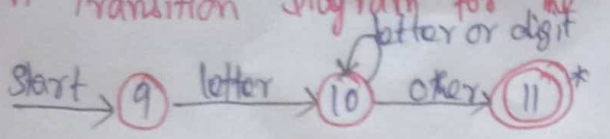


where state '1' is initial state & state 3 is final state. Here is the transition diagram of finite automata that recognizes the lexemes matching the token relop.

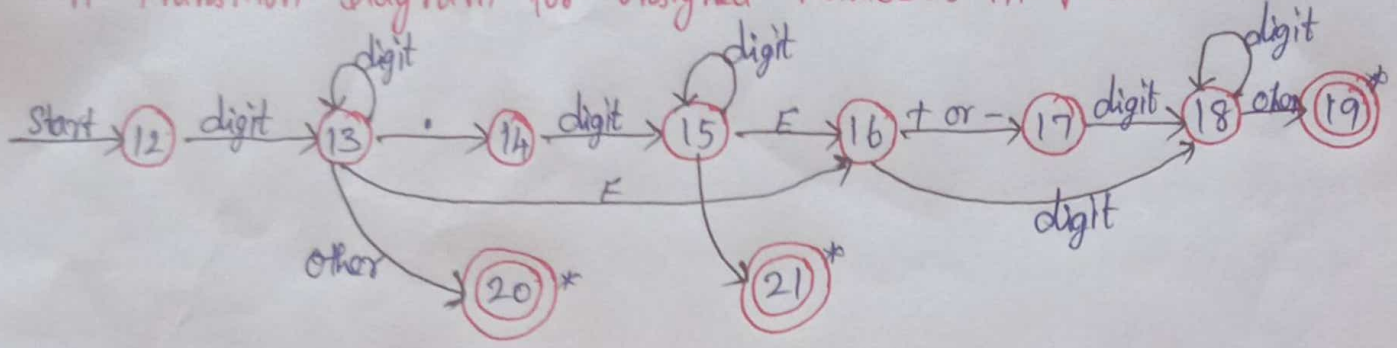


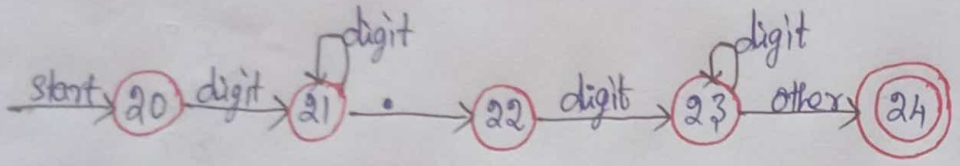
Here is the finite automata transition diagram for the identifiers & keywords

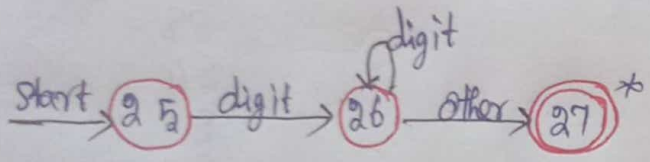
A Transition Diagram for the identifiers & keywords



A Transition Diagram for unsigned Numbers in Pascal.

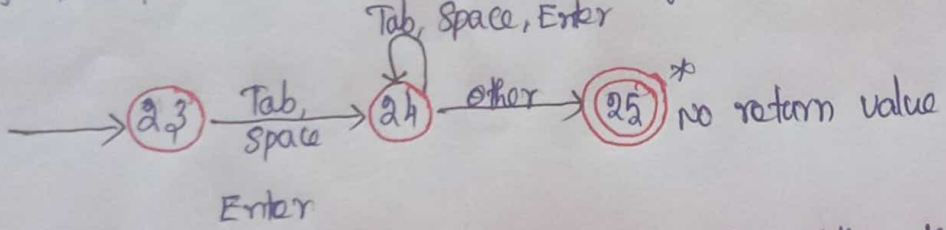






num → digit * (. digit + | ∈) (E (+|-|∈) digit + | ∈)

Here is the finite automata transition diagram for recognizing white spaces.



These finite automata can be constructed using either the transition diagram or the transition table representation. Both transition diagrams & transition tables serve the same purpose of defining & representing the behavior of an finite automata. They provide different visual & structural representations, allowing designers to choose the format that best suits their preferences or requirements.

Finite Automata

Finite automata can be defined as a recognizer that identifies whether the input string represents the regular language. The finite automata accept the input string if the input string is a regular expression representing the regular language else it rejects it. When finite automata accept the regular expression, it compiles

it to form a transition diagram.

Types of Finite Automata

There are two kind of finite automata.

1. Deterministic Finite Automata (DFA)
2. Non-Deterministic Finite Automata (NFA)

Deterministic Finite Automata

In deterministic finite automata in response to a single input alphabet, the sta transits to only one state. In DFA, no null moves are acepted.

The DFA have five tuples as discussed below

$M = \{Q, \Sigma, q_0, F, \delta\}$

Q = Set of all states

Σ = Set of all input alphabets

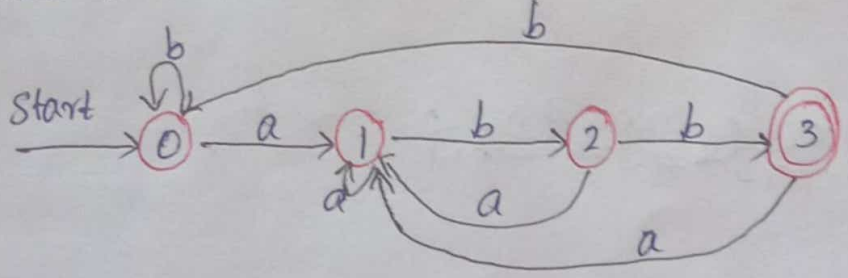
q_0 = initial state

F = Set of Final State

δ = Transition function.

The DFA can be represented by the transition graph. In the transition graph the nodes of the graph represent the states & the edges represent the transition of one state to another.

ex



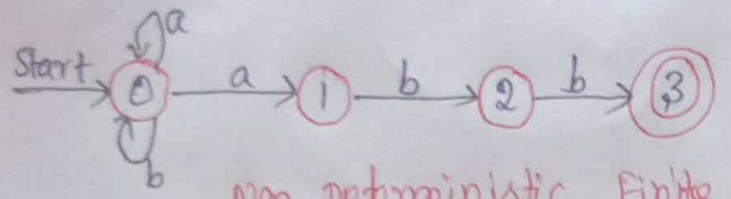
Deterministic finite automata.

State	a	b
0	{1}	{0}
1	{1}	{2}
2	{1}	{3}
3	{1}	{0}

Transition table of DFA.

Non-Deterministic Finite Automata

In non-deterministic finite automata, in response to a single input alphabet, a state may transit to more than one state. The NFA also accept the null move.



Non-Deterministic Finite Automata.

State	a	b	∈
0	{0,1}	{0}	∅
1	∅	{2}	∅
2	∅	{3}	∅
3	∅	∅	∅

Transition table of NFA.

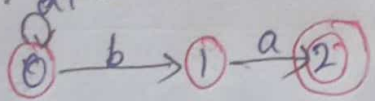
With the transition table, it becomes easy to identify the transition on a given state with the given input. Although it takes a lot of space if there are a lot of input alphabets as most of the state do not perform any transition on most of the input symbols.

Regular Expression to finite automata.

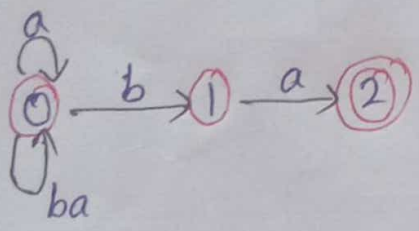
A regular expression (a+ba)* ba & to convert it to finite automata.

ex

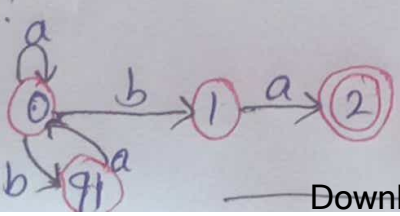
Step : 1:



Step 2:



Step 3:



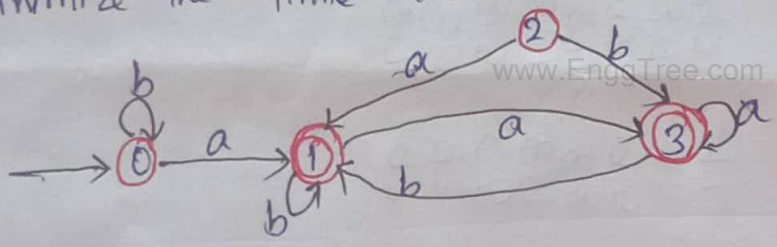
Minimization of DFA

Minimization of DFA is reducing the DFA to minimum states & keeping it equivalent to the provided DFA. The finite automata must be containing some redundant states which unnecessarily increases the size of automata.

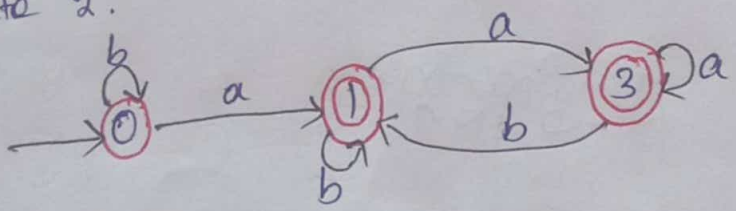
The steps of minimizing of Finite Automata.

1. Remove all unreachable states.
2. Draw a transition table for all pairs of states.
3. Split the set of all states into two T_1 & T_2. T_1 - Set of final states. T_2 - Set of non-final states.
4. Identify & merge equivalent state from set T_1 & T_2 repeat the same for T_2.
5. Now after merging the equivalent states in T_1 & T_2 combine the reduced sets T_1 & T_2.

ex: Minimize the finite automata shown in the diagram below.



Step 1:
In the automata that the state '2' is unreachable, so first remove state '2'.



Step 2:
To create a transition diagram for the automata.

State	a	b
0	1*	0
1*	3*	1*
3*	3*	1*

Step 3:

To make two sets one consisting of the final states & the other consist the non-final states.

State	a	b
0	1	0

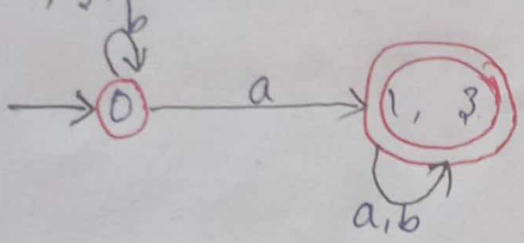
T_1 set of non final states

State	a	b
1*	3*	1*
3*	3*	1*

T_2 Set of Final States.

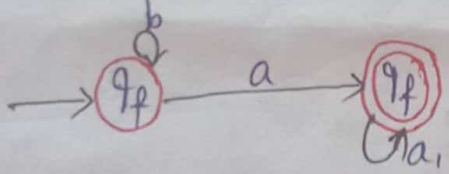
Step 4:

As the set with final states has two equivalent rows leading to this merge state '1, 3'.



Step 5:

The final minimized finite automata are as follows.

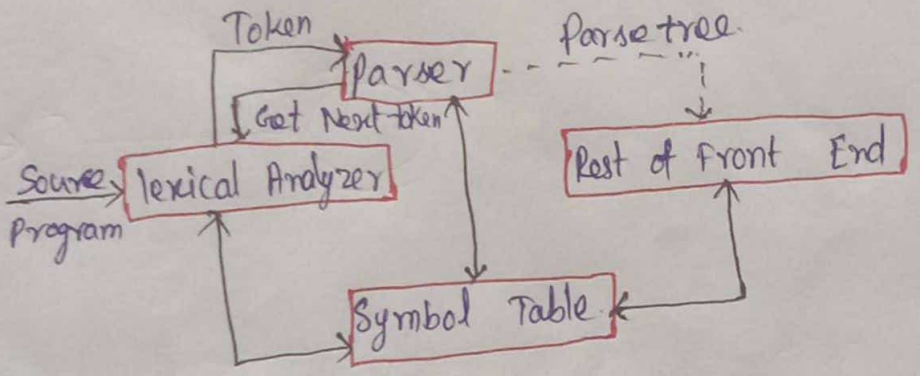


Unit - II

Role of Parser

~~Parsing~~

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 & verifies that the string can be the grammar for the source language. It detects & reports any syntax errors & produces a parse tree from which intermediate code can be generated.

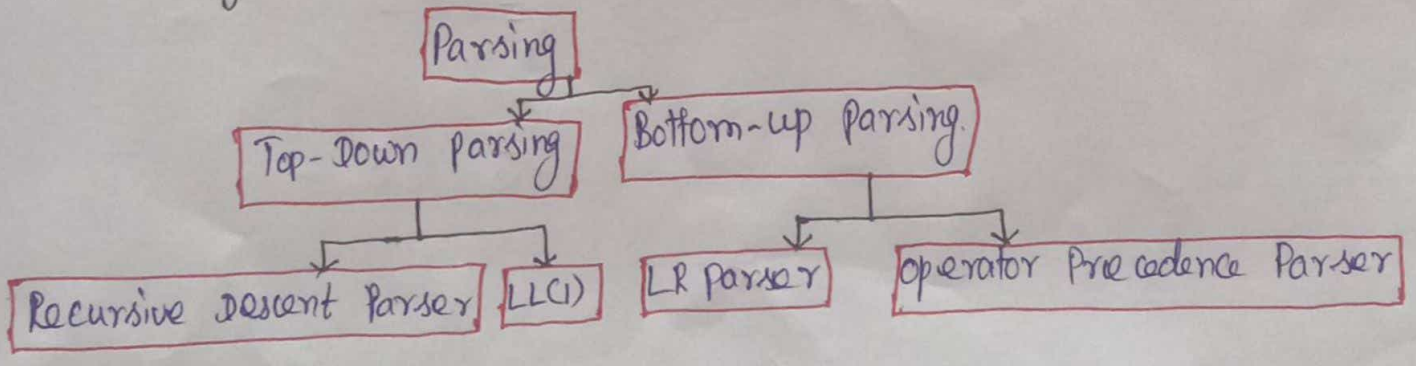


Role of Parsing

Parsing

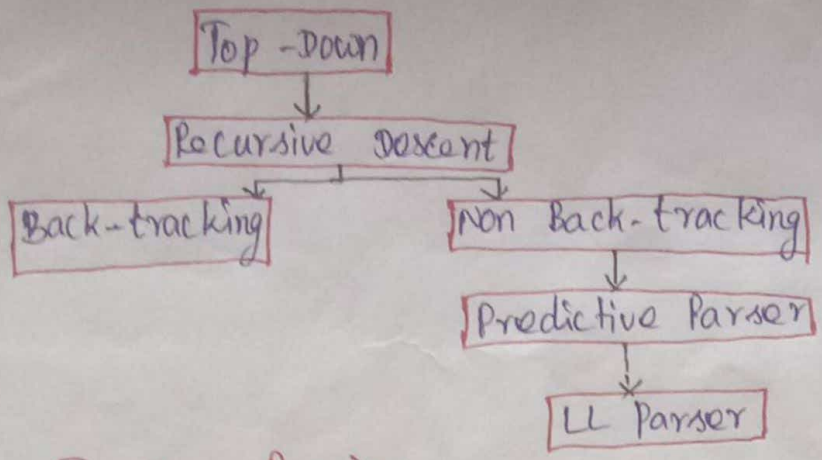
Parsing is performed at the syntax analysis phase where a stream of tokens is taken as input from the lexical analyzer & the parser produces the parse tree for the tokens while checking the stream of tokens against the Syntax errors.

Types of Parsing



Top-Down Parsing

When the parser generates a parse with top-down expansion to the first trace, the left-most derivation of input is called top-down parsing. The top-down parsing initiates with the start symbol & ends on the terminals. Such parsing is also known as predictive parsing.



Top-Down Parsing

Recursive Descent Parsing

Recursive descent parsing is a type of top-down parsing technique. This technique follows the process for every terminal & non-terminal entity. It reads the input from left to right & constructs the parse tree from right to left. As the technique works recursively, it is called recursive descent parsing.

Back-tracking

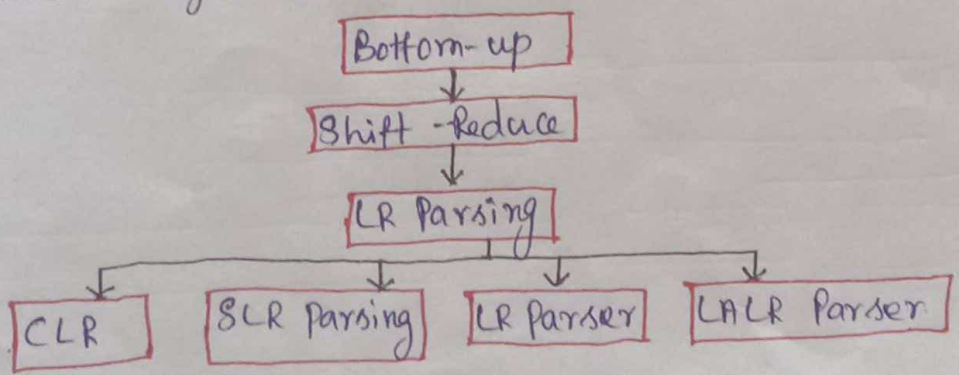
The parsing technique that starts from the initial pointer, the root node. If the derivation fails, then it restarts the process with different rules. Brute Force technique.

Non-Backtracking

Recursive descent parsing, Predictive parsing or non-recursive parsing or LL(1) parsing or table driver parsing.

Bottom-up parsing

The bottom-up parsing works just the reverse of the top-down parsing. It first traces the right most derivation of the input until it reaches the start symbol.



Shift-Reduce Parsing

Shift reduce parsing works on two steps: Shift Step & Reduce Step.

Shift Step

The shift step indicates the increment of the input pointer to the next input

Symbol that is shifted.

Reduce Step

When the parser has a complete grammar rule on the right-hand side & replaces it with RHS

LR Parsing

LR parser is one of the most efficient syntax analysis techniques as it works with context-free grammar. In LR parsing L stands for the left to Right tracing & R stands for the right to left tracing.

Operator Precedence Parsing

The grammar defined using operator grammar is known as operator precedence parsing. In operator precedence parsing there should be no null production & two non-terminals should not be adjacent to each other.

Grammars

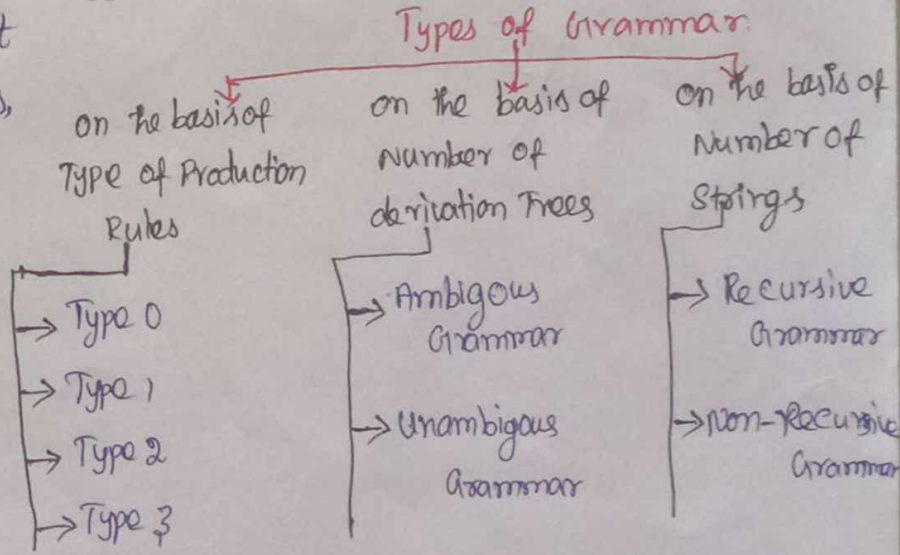
Grammars are used to describe the syntax of a programming language. It specifies the structure of expression & statements.

Stmt → if (expr) then Stmt

where stmt denotes statements, expr denotes expressions.

Types of Grammar

1. Type 0 grammar
2. Type 1 grammar
3. Type 2 grammar
4. Type 3 grammar



Content free Grammar

Content free grammar is also called as Type 2 grammar. A content free grammar G is defined by four tuples as,

$$G = (V, T, P, S)$$

where, G - Grammar
P - Set of Productions
V - Set of variables
S - Start Symbol.
T - Set of Terminals

It Produces context free language (CFL) which is defined as.

$$L(G) = \{ w | w \text{ is in } T^* | S \xrightarrow[G]{*} w, y.$$

where,

l - Language

G - Grammar

w - Input String

S - Start Symbol

T - Terminal

Hence, CFL is a collection of input strings which are terminals, derived from the start symbol of grammar on multiple steps.

A Context-free grammar has four components

Terminals

Terminals are symbols from which strings are formed.

* lowercase letters (a, b, c)
* operators (+, −, *)
* Puctuation symbols (, , ( ))
* Digits (0, 1, .. 9)
* Boldface letters (id, if)

Non-terminals

non-terminals are syntactic variables that denote a set of strings.

* uppercase letters (A, B, C)
* Lowercase italic names (expr, stmt)

Start Symbol

Start symbol is the head of the production stated first in the grammar.

Production

Production is of the form LHS → RHS (or) head → body, where head contains only one non-terminal & body contains a collection of terminals & non-terminals.

ex let G be,

$$E \longrightarrow E+T \mid E-T \mid T$$
$$T \longrightarrow T*F \mid T/F \mid F$$
$$F \longrightarrow (E) \mid id$$

Sln:

$V = \{ E, T, F \}$

$T = \{ =, -, *, /, (,), id \}$

$S = \{ E \}$.

P:

$E \longrightarrow E + T$ $\quad T \longrightarrow T / F$

$E \longrightarrow E - T$ $\quad T \longrightarrow F$

$E \longrightarrow T$ $\quad F \longrightarrow (E)$

$T \longrightarrow T * F$ $\quad F \longrightarrow id$

Writing a Grammar.

A grammar consists of a number of productions. Each production has an abstract symbol called a nonterminal as its left-hand side, & a sequence of one or more nonterminals & terminals symbols as its right-hand side. For each grammar, the terminal symbols are drawn from a specified alphabet.

Regular Expression

It is used to describe the tokens of programming languages. It is used to check whether the given input is valid or not using transition d. The transition diagram has set of states & edges. It has no start symbol. It is useful for describing the structure of lexical constructs such asidentifiers, constants, keywords, & so forth.

Content-free Grammar

It is used to check whether the given input is valid or not using derivation. The content-free grammar has set of productions. It has start symbol.

There are four categories in writing a grammar.

1. Regular Expression vs Content free grammar.
2. Eliminating ambiguous grammar.
3. Eliminating left-recursion.
4. left-factoring.

Regular- Expression	Context-free grammar
It is used to describe the tokens of Programming languages.	It consists of a quadruple where S→ Start symbol, P→ production, T→ terminal, V→ variable or non-terminal.
It is used to check whether the given input is valid or not using transition diagram.	It is used to check whether the given input is valid or not using derivation.
The transition diagram has set of states & edges.	The context-free grammar has set of productions.
It has no start Symbol.	It has start symbol.
It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, & so forth.	It is useful in describing nested structures such as balanced parenthesis matching begin-end's & so on.

Eliminating ambiguity

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

Consider this example, G1: stmt → if expr then stmt | if expr then stmt else stmt | other. This grammar is ambiguous since the string if E_1 then if E_2 then S_1 else S_2 has the following two parse trees for leftmost derivation.

To eliminate ambiguity, the following grammar may be used:

stmt → matched | unmatched stmt_ Stmt

matched → if expr stmt then matched else matched stmt_ Stmt | other

unmatched → if expr then stmt | if expr then matched else unmatched stmt.

Eliminating left Recursion

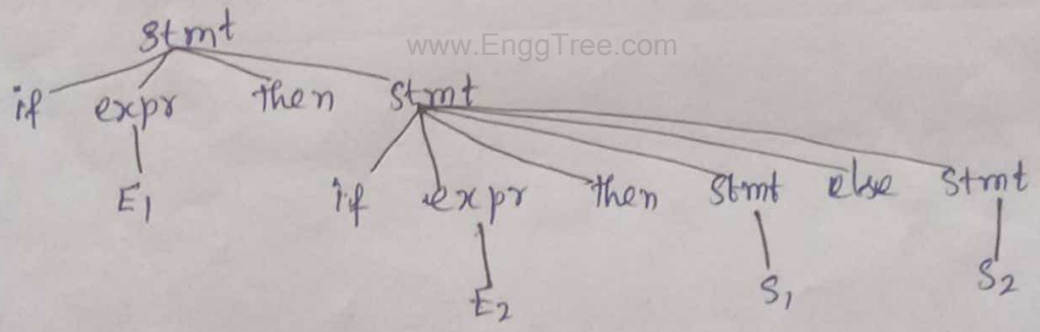
A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow Aa$ for some string a. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

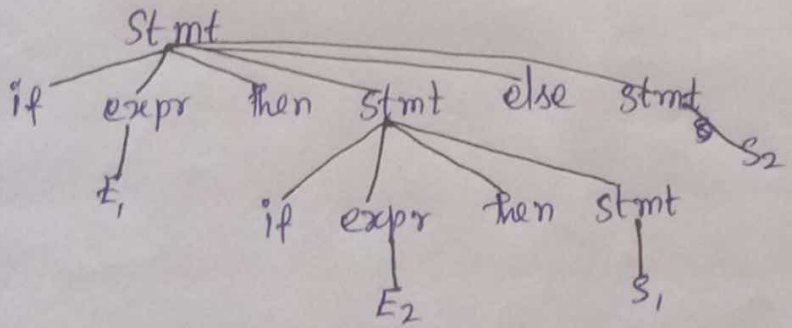
If there is $A \to \alpha$ a production can A be replaced with a s $A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$
without changing the set of strings derivable from A.

Algorithm to eliminate left recursion

Arrange the non-terminals in some order $A_1, A_2 \cdots A_n$.

```
for i := 1 to n do begin
  for j := 1 to i-1 do begin
    replace each production of the form Ai → Aj γ by the
    productions Ai → δ1 γ | δ2 γ | ... | δk γ
    where Aj → δ1 | δ2 | ... | δk are all the current Aj-Productions;
  end
  eliminate the immediate left recursion among the Ai - Productions
end
```





Two parse trees for an ambiguous sentence

left factoring

left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, rewrite the A-Productions to defer the decision until have seen enough of the input to make the right choice.

If there is any $\rightarrow \alpha\beta_1 | \beta_2$ production $\alpha\beta$, it can be A rewritten as

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 | \beta_2$

Consider the grammar, G: $S \rightarrow iEts | iEtSeS | a$

$E \rightarrow b$

left factored, this grammar becomes

$S \rightarrow iEtSS' | a$

$S' \rightarrow eS | \varepsilon$ $E \rightarrow b$

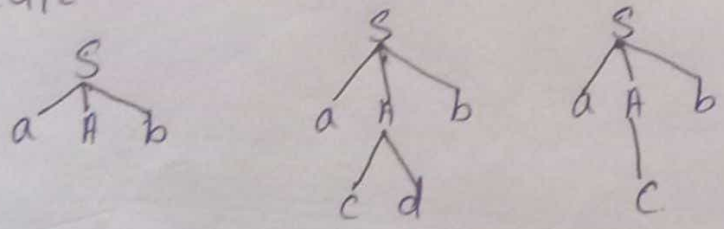
Top Down Parsing

Top-down parsing is a method of parsing the input string provided by the lexical analyzer. The top-down parser parses the input string & then generates the parse tree for it. The top-down approach construction of the parse tree starts from the root node & end up creating the leaf nodes. Here the leaf node presents the terminals that match the terminals of the input string.

Consider the lexical analyzer's input string 'acb' for the following grammar by using left most derivation.

$S \rightarrow aAb$

$A \rightarrow cd | c$



Top-Down Parsing

Recursive Descent parsers

Recursive descent parsers are a type of top-down parser that uses a set of recursive procedures to parse the input. Each non-terminal symbol in the grammar corresponds to a procedure that parses input for that symbol.

Backtracking Parsers

Backtracking parsers are a type of top-down parser that can handle non-deterministic grammar. When a parsing decision leads to a dead end, the parser can backtrack & try another alternative. Backtracking parsers are not as efficient as other top-down parsers because they can potentially explore many parsing paths.

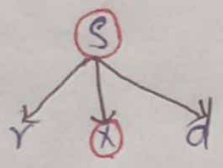
S → rXd | rzd

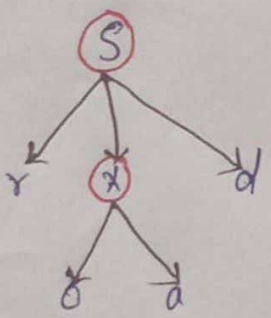
X → oa | ea

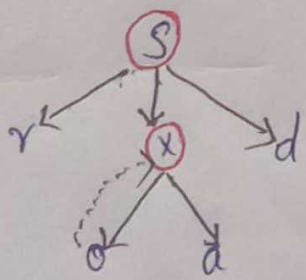
z → ai

For an input string: read, a top-down parser, will behave like this. It will start with S from the production rules Σ_p will match its yield to the left-most letter of the input that is 'r'. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter that is 'e'. The parser tries to expand non-terminal 'X' Σ_q checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea). The parser matches all the input letters in an ordered manner. The string is accepted.

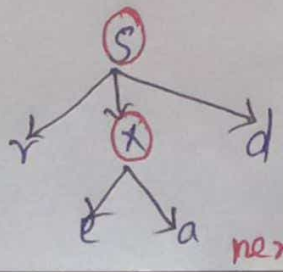
Back tracking







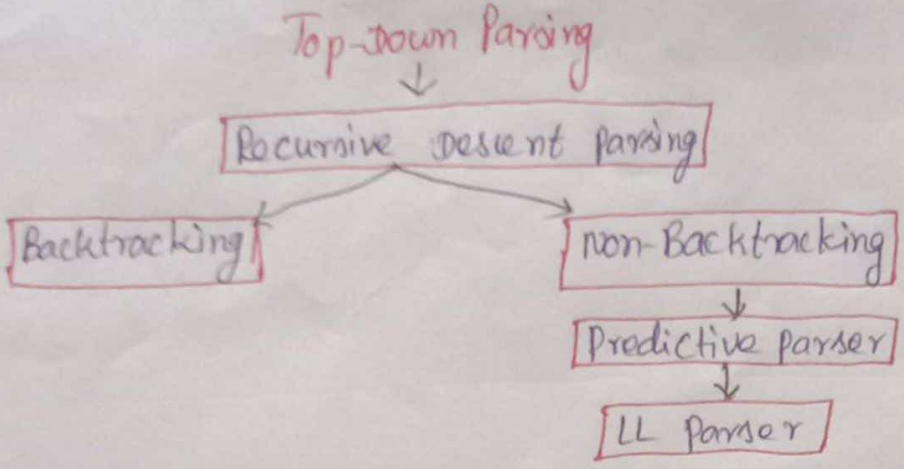
Back-tracking



next - Production.

Types of Top-Down Parser

The most general form of top-down parsing is recursive descent parsing.



Non-Backtracking Parsers

Non-backtracking is a technique used in top-down parsing to ensure that the parser doesn't revisit already-explored paths in the parse tree during the parsing process. This is achieved by using a predictive parsing table that is constructed in advance & selecting the appropriate production rule based on the top non-terminal symbol on the parser's stack & the next input symbol. By not backtracking, predictive parsers are more efficient than other types of top-down parsers, although they may not be able to handle all grammar.

Predictive Parsing

Predictive parsing is simple form of recursive descent parsing. & it requires non backtracking. Instead it can determine which A-Production must be chosen to derive the input string. It allows looking ahead a fixed number of input symbols from the input string.

Components of Predictive top-down Parser

Predictive top-down pasing program maintains three components

Stack

A predictive parser maintains a stack containing a sequence of grammar symbols.

Input Buffer

It contains the input string that the predictive parser has to parse.

Parsing Table

The entries of this table it becomes easy for the top-down parser to choose the production to be applied.

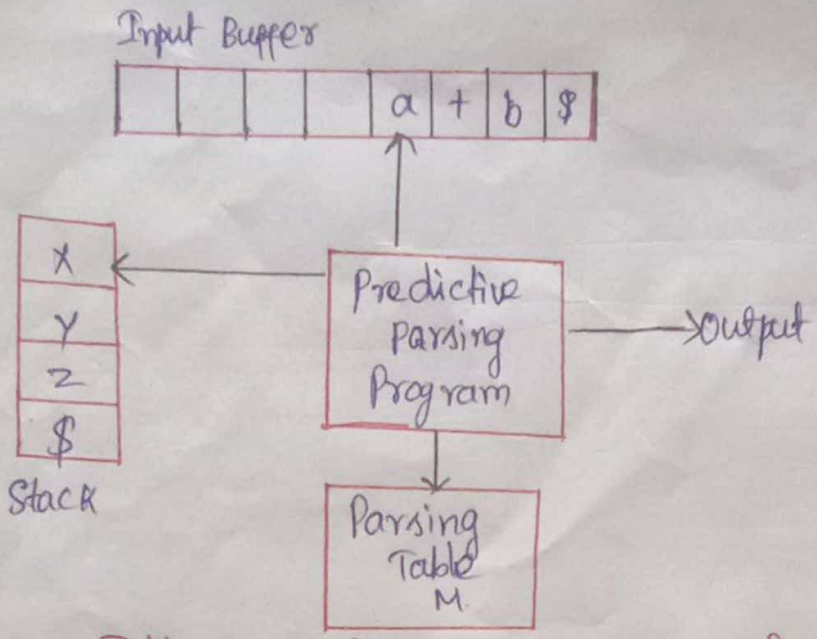


Table Driven Predictive Recursive Parsing

Input buffer & stack both contain the end marker '$'. It indicates the bottom of the stack & the end of the input string in the input buffer.

Steps to perform predictive parsing.

Top-Bottom Parser

↓ Remove Left Recursion
Left Factored Grammar

Recursive Desent

↓ Remove Back-tracking

Predictive Parser

↓ Use Table
Remove Recursion.

Non-recursive Predictive Parser

In recursive desent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

LL Parsing

The LL parser is a predictive parser that doesn't need backtracking. LL(1) parser accepts only LL(1) grammar.

*First L in LL(1) indicates that the parser scans the input string from left to right.

*Second L determines the left most derivation for the input string.

*The '1' in LL(1) indicates that the parser lookahead only one input symbol from the input string.

LL(1) grammar doesn't include left recursion & there is no ambiguity in the LL(1) grammar.

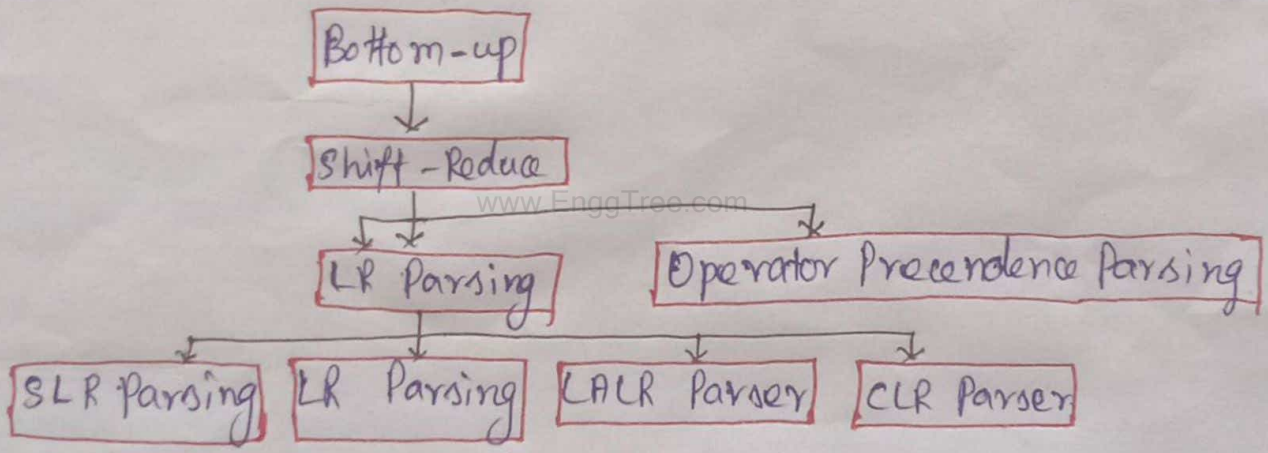
left to right → LL(k) → k lookahead symbol.
left most derivation

ex
A grammar G is LL(1) if $A \to \alpha | \beta$ are two distinct productions of G

* for no terminal, both α & β derive strings beginning with a.
* at most one of α & β can derive empty string.
* if $\beta \to t$, then α does not derive any string beginning with a terminal in FOLLOW(A).

Bottom-up Parsing

Bottom-up parsing starts from the leaf nodes of a tree & works in upward direction till it reaches the root node. Here, start from a sentence & then apply production rules in reverse manner in order to reach the start symbol.



Bottom-up Parsing.

Shift-Reduce Parsing

Shift-Reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step & reduce-step.

Shift-Step

The Shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.

Reduce-Step

When the parser finds a complete grammar rule (RHS) & replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a

handle. To reduce, a POP function is performed on the stack which pops off the handle & replaces it with LHS non-terminal symbol.

Four Functions of Shift-Reduce Parsing.

1. Shift

This action shifts the next input symbol present on the input buffer onto the top of the stack.

2. Reduce

This action is performed if the top of the stack has an input symbol that denotes a right end of a substring & within the stack there exist the left end of the substring. The reduce action replaces the entire substring with the appropriate non-terminal. The production body of this non-terminal matches the replaced substring.

3. Accept

When at the end of parsing the input buffer becomes empty & the stack has left with the start symbol of the grammar. The parser announces the successful completion of parsing.

4. Error

This action identifies the error & performs an error recovery routine.

Ex: String id * id + id $.

Grammar

~~S → S+S~~ E → E+T | T

~~S → S*S~~ T → T * F | F

F → (E) | id

Shift-Reduce Parsing on Input String id + id * id.

Stack	Input Buffer	Action
$	id+id*id $	
$id	+id*id $	Shift id
$F	+id*id $	Reduce F→id
$T	+id*id $	Reduce T→F
$E	+id*id $	Reduce E→T
$E+	id*id $	Shift +
$E+id	*id $	Shift id
$E+F	*id $	Reduce F→id
$E+T	*id $	Reduce T→F
$E+T*	id $	Shift*
$E+T*id	$	Shift id
$E+T*F	$	Reduce F→id
$E+T	$	Reduce T→F
$E	$	Reduce E→E+T

There are two main categories in shift-reduce parsing as follows:

1. Operator-Precedence Parsing
2. LR-Parser

Operator Precedence Parsing

Operator precedence grammar is a category in the shift-reduce method of parsing. It is a applied to a class of grammars operators. Operator precedence grammar must have following two properties:

No RHS of any product has a ϵ

Two non-terminals must not be adjacent.

Operator precedence can only be fixed between the terminals of the grammar. It generally ignores the non-terminal.

There are the three operator precedence relations:

* $a \gtrdot b$ means that terminal "a" has higher precedence than terminal "b".
* $a \lessdot b$ means that terminal "a" has lower priority than terminal "b".
* $a \doteq b$ means that the terminal "a" & "b" both have the same precedence.

LR Parsing

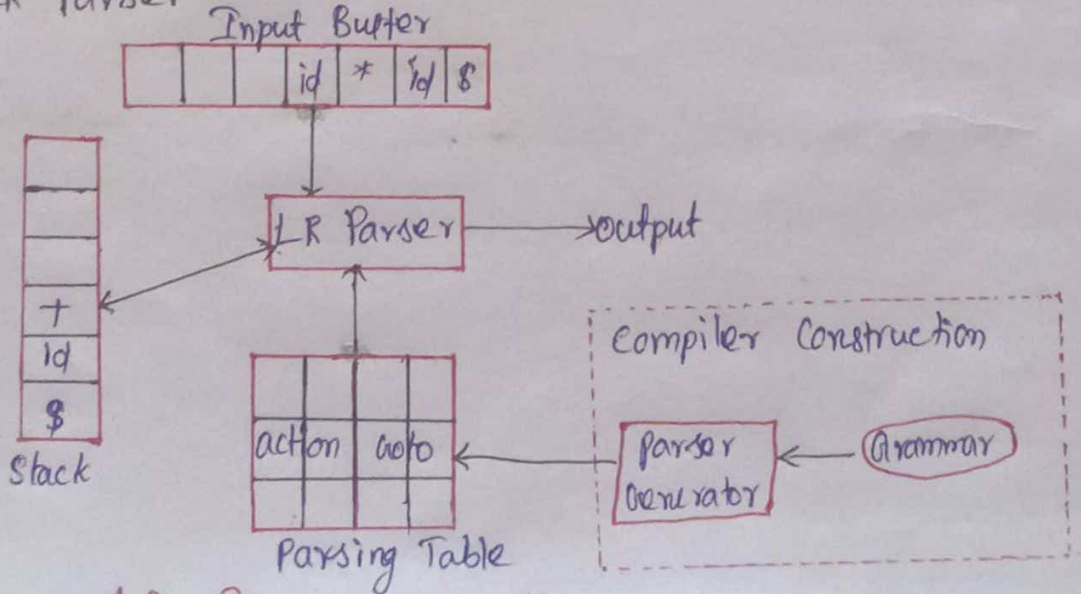
It is generally used to parse the class of grammars whose size is large.

L: L stands for scanning of the input left-to-right.

R: R stands for constructing a rightmost derivation in a reverse way.

K: K stands for the count of input symbols of the look-ahead that are used to make many parsing decisions.

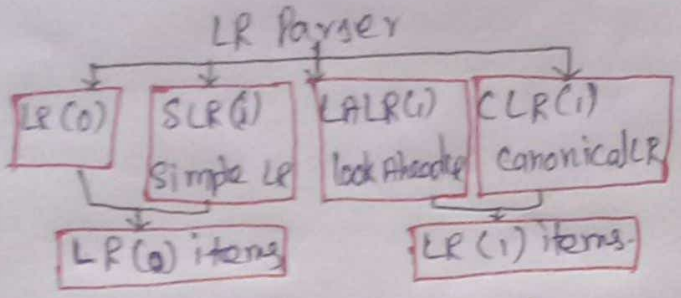
Structure of LR Parser



LR Parsing Algorithm

Similar to predictive parsing the end of the input buffer & end of stack has

Types of LR Parsing



LR(0) Parser

The LR Parser is a Shift-reduce parser that makes use of a deterministic finite automata, recognizing the set of all variable prefixes by reading the stack from Bottom to Top. If a finite-state machine that recognizes variable prefixes of the right sentential forms is constructed, it can be used to guide the handle selection in the shift-reduce parser.

Handle

Handle is a substring that matches the body of a production. Handle is a Right Sentential form + Position where reduction can be performed + Production used for reduction.

ex

(S) S. with $S \to \varepsilon$

(S) .S with $S \to S$

((S) S.) with $S \to (S)S$

Handle Pruning

Handle pruning specifies that the reduction represents one step along the reverse of a right most derivation.

Right Sentential form	Handle	Reducing Production
id * id	id	$F \to id$
F * id	F	$T \to F$
T * id	id	$F \to id$
T * F	T * F	$E \to T * F$

Augmented Grammar

If G is Grammar with start Symbol S, the augmented grammar G' is G with a new start Symbol S' & new Production $S' \to S\ \$$. The purpose of the Augmented Grammar is to indicate to the parser when it should stop parsing & announce acceptance of the input.

LR (0) Items

An LR(0) item of a Grammar is a production of G with a Dot (·) at some Position of the right side.

Production $A \to XYZ$ yields the four items:

1. $A \to \cdot XYZ$ a string derivable from XYZ next on the input.
2. $A \to X \cdot YZ$ the input a string derivable from X & that next to see a string derivable from YZ next on the input.
3. $A \to XY \cdot Z$
4. $A \to XYZ \cdot$

The production $A \to \varepsilon$ generates only one item, $A \to \cdot$. Each of this item is a variable prefixes.

Closure Item

An Item created by the closure operation on a state.

Complete item

An Item where the Item Dot is at the end of the RHS.

Ex

Context free Grammar

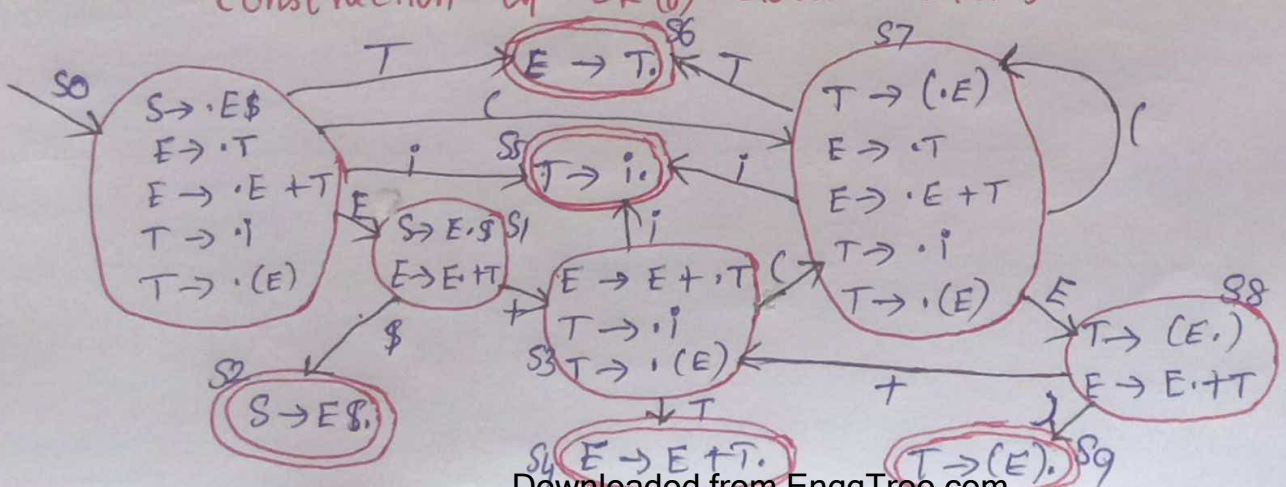
$S \to E$
$E \to T$
$E \to E + T$
$T \to i$
$T \to (E)$

rule 1: $(S \to S')$

Augmented grammar

$S \to \cdot E$
$E \to \cdot T$
$E \to \cdot E + T$
$T \to \cdot i$
$T \to \cdot (E)$

Construction of LR(0) Closure Items



Construction of LR(0) Items

I_0:
$S \to \cdot E\$$
$E \to \cdot T$
$E \to \cdot E+T$
$T \to \cdot i$
$T \to \cdot (E)$

I_1: Goto (I_0, E)
$S \to E \cdot \$$
$E \to E \cdot +T$

I_2: Goto $(I_1, \$)$
$S \to E\$ \cdot$

I_3: Goto $(I_1, +)$
$E \to E+\cdot T$
$T \to \cdot i$
$T \to \cdot (E)$

I_4: Goto (I_3, T)
$E \to E+T \cdot$

I_5: Goto (I_0, i)
$T \to i \cdot$

I_5: Goto (I_3, i)
$T \to i \cdot$

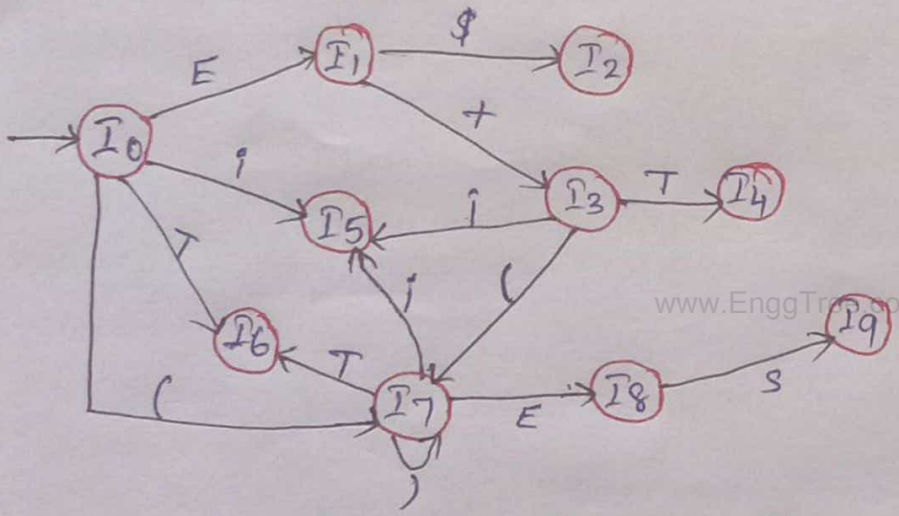
I_6: Goto (I_0, T)
$E \to T \cdot$

I_7: Goto $(I_0, ()$
$T \to (\cdot E)$
$E \to \cdot T$
$E \to \cdot E+T$
$T \to \cdot i$
$T \to \cdot (E)$

I_8: Goto (I_7, E)
$T \to (E \cdot)$
$T \to E \cdot +T$

I_9: Goto $(I_7,))$
$T \to (E) \cdot$

Construction of DFA for LR(0) Items



Construction of LR(0) Parsing Table

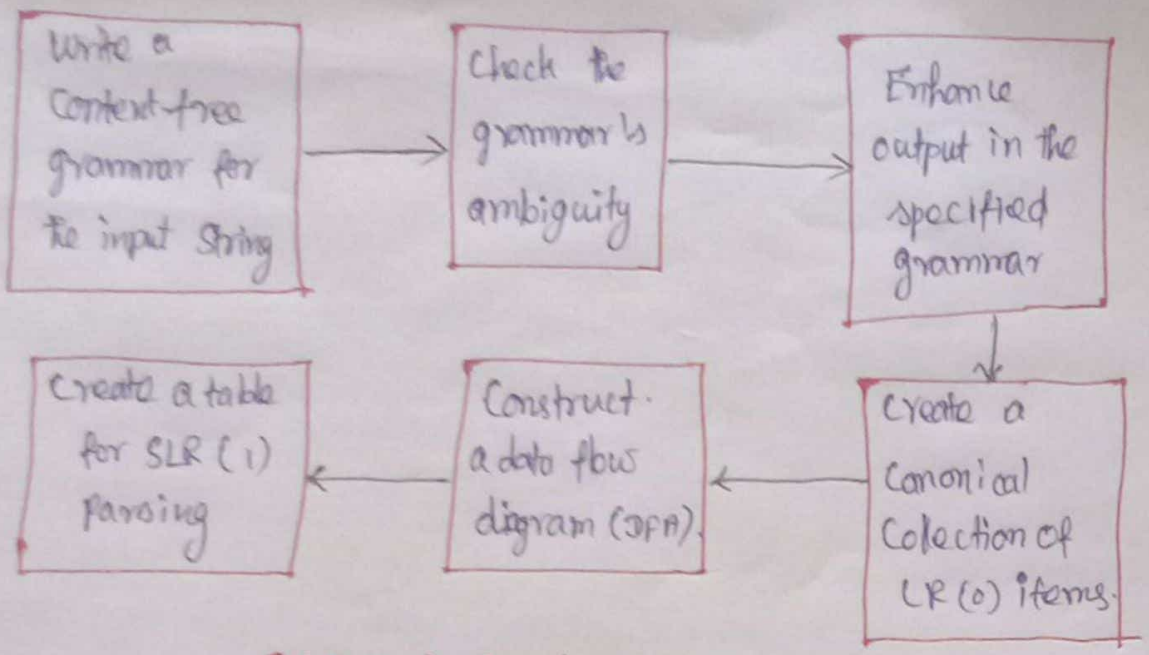
State	Input: Action Part (Terminals) i	+	(	)	$	Goto Part (non-terminals) E	T	
0	5		7			1	6	Shift
1		3			2			Shift
2	$S \to E\$$							Reduce
3	5		7				4	Shift
4	$E \to E+T$							Reduce
5	$T \to i$							Reduce
6	$E \to T$							Reduce
7	5		7			8	6	Shift
8		3		9				Shift
9	$T \to (E)$							Reduce

String Acceptance

Stack	Input	Action
S_0	i + (i + i) $	Shift
S_0 i S_5	+ (i + i) $	Reduce by T → i
S_0 T S_6	+ (i + i) $	Reduce by E → T
S_0 E S_1	+ (i + i) $	Shift
S_0 E S_1 + S_3	(i + i) $	Shift
S_0 E S_1 + S_3 (S_7	i + i) $	Shift
S_0 E S_1 + S_3 (S_7 i S_5	+ i) $	Reduce by T → i
S_0 E S_1 + S_3 (S_7 T S_6	+ i) $	Reduce by E → T
S_0 E S_1 + S_3 (S_7 E S_8	+ i) $	Shift
S_0 E S_1 + S_3 (S_7 E S_8 + S_3	i) $	Shift
S_0 E S_1 + S_3 (S_7 E S_8 + S_3 i S_5	) $	Reduce by T → i
S_0 E S_1 + S_3 (S_7 E S_8 + S_3 T S_4	) $	Reduce by E → E + T
S_0 E S_1 + S_3 (S_7 E S_3 (S_7 E S_8	) $	Shift
S_0 E S_1 + S_3 (S_7 E S_8) S_9	$	Reduce by T → (E)
S_0 E S_1 + S_3 T S_4	$	Reduce by E → E + T
S_0 E S_1	$	Shift
S_0 E S_1 $ S_2		Reduce by S → E $
S_0 S		accept

Simple LR (SLR) Parser

Execution is relatively simple & inexpensive. But for particular classes of grammar, it is unable to create a parsing table. It creates parsing tables that aid in performing input string parsing. SLR parsing is possible with the provision of Context-free grammar.



SLR(1) Parsing.

SLR(1) Parser

It is similar to LR(0) parsing because LR(0) parsing requires just the canoncical items for the construction of a parsing table. A collection of rules is used to create the table using the provided grammar in order to determine the closure & goto operations for each state. The SLR parsing algorithm is reasonably easy to use & effective.

Working of SLR(1) Parser

An SLR parser is a bottom-up parsing technique that uses a deterministic finite automata (DFA) to recognize the valid syntax of a language. A parse table is produced by this parser from a collection of grammar rules & the DFA. The grammar rule to apply at each stage of the parsing process is subsequently determined using the parse table.

ex

Implement SLR Parser for the given grammar

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id.$

Parsing table

To fill the reduction action in the ACTION section of the table, computation of FOLLOW is necessary.

Follow (E) : { $,), + }

FOLLOW (T) : { $, +,), * }

Follow (F) : { *, +,), $ }

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				acc			
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			

Si

Si means to shift & stack state i.

r_j

r_j means reduce by production numbered j.

acc

acc means accept.

Blank

Blank means error.

Parse the given input. Parse the string id * id + id using stack implementation.

Stack.	input	Action
0	id * id + id $	S5 Shift 5
0 id 5	* id + id $	r6 Reduce by F → id
0 F 3	* id + id $	r4 Reduce by T → F
0 T 2	* id + id $	S7 Shift 7
0 T 2 * 7	id + id $	S5 Shift 5
0 T 2 * 7 id 5	+ id $	r6 Reduce by F → id
0 T 2 * 7 F 10	+ id $	r3 Reduce by T → T * F
0 T 2	+ id $	r2 Reduce by E → T
0 E 1	+ id $	S6 Shift 6
0 E 1 + 6	id $	S5 Shift 5
0 E 1 + 6 id 5	$	r6 Reduce by F → id
0 E 1 + 6 F 3	$	r4 Reduce by T → F
0 E 1 + 6 T 9	$	r1 Reduce by E → E + T
0 E 1	$	Accept

lookahead LR Parser (LALR)

LALR is the most powerful parser which can handle large classes of grammar. The size of CLR parsing table is quite large as compared to other parsing table. LALR reduces the size of this table. LALR works similar to CLR. The only difference is, it combines the similar states of CLR parsing table into one single state.

The general syntax becomes [A → α.B, a]

where A → α.B is production & a is a terminal or right end marker $.

LR(1) items = LR(0) items + look ahead.

Types of LALR parser

1. Simple LALR Parser.
2. Canonical LALR parser

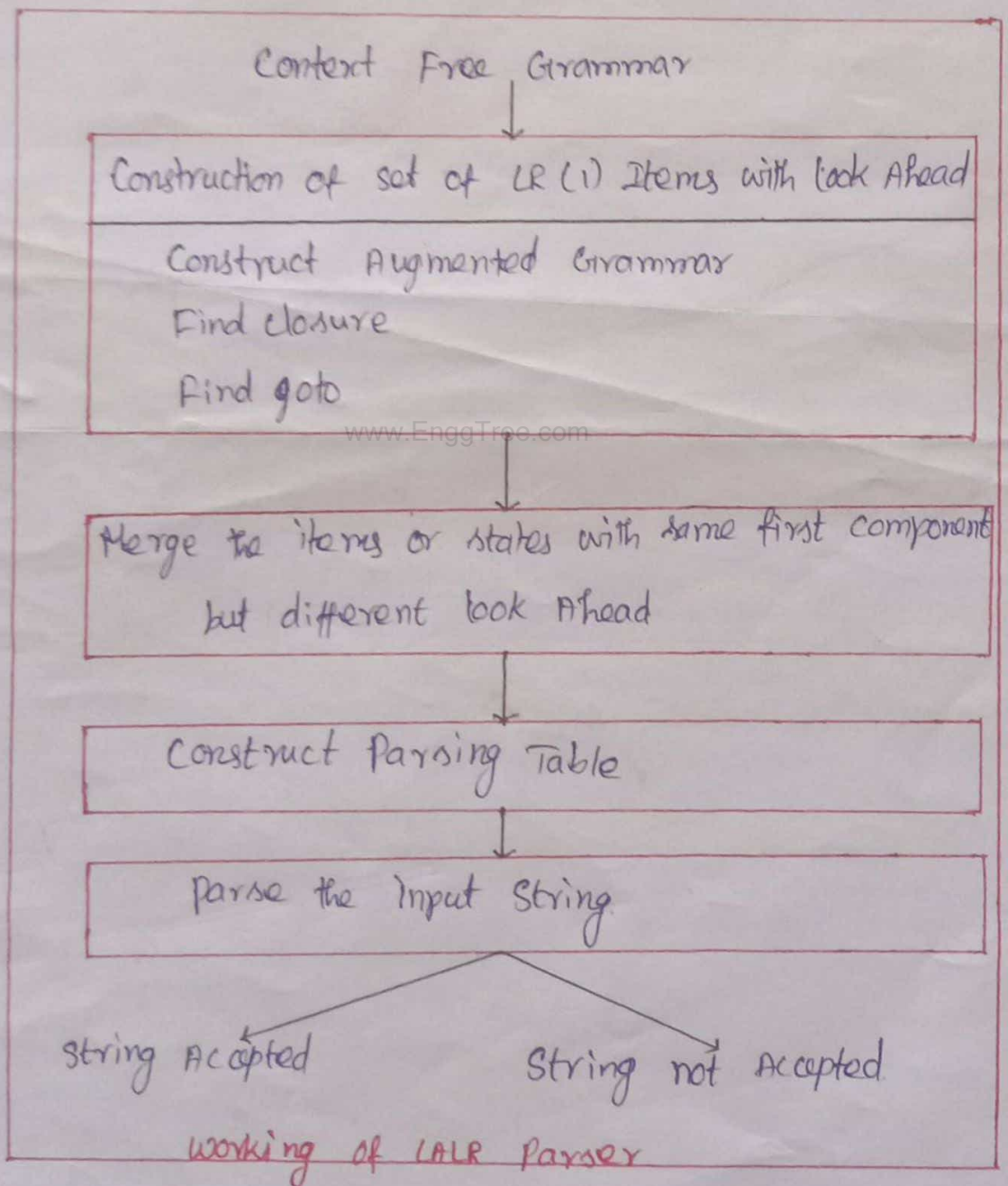
Simple LALR Parser

It is also known as SLR parser. Simple LALR uses a simple parse table construction algorithm. It only handles a limited class of grammar. Class Power

Canonical LALR Parser

It is also known as CLR parser. Canonical LALR uses a complex parse table construction algorithm. It can handle a large range of class grammar once efficiently.

Working of LALR Parser



Working of LALR Parser

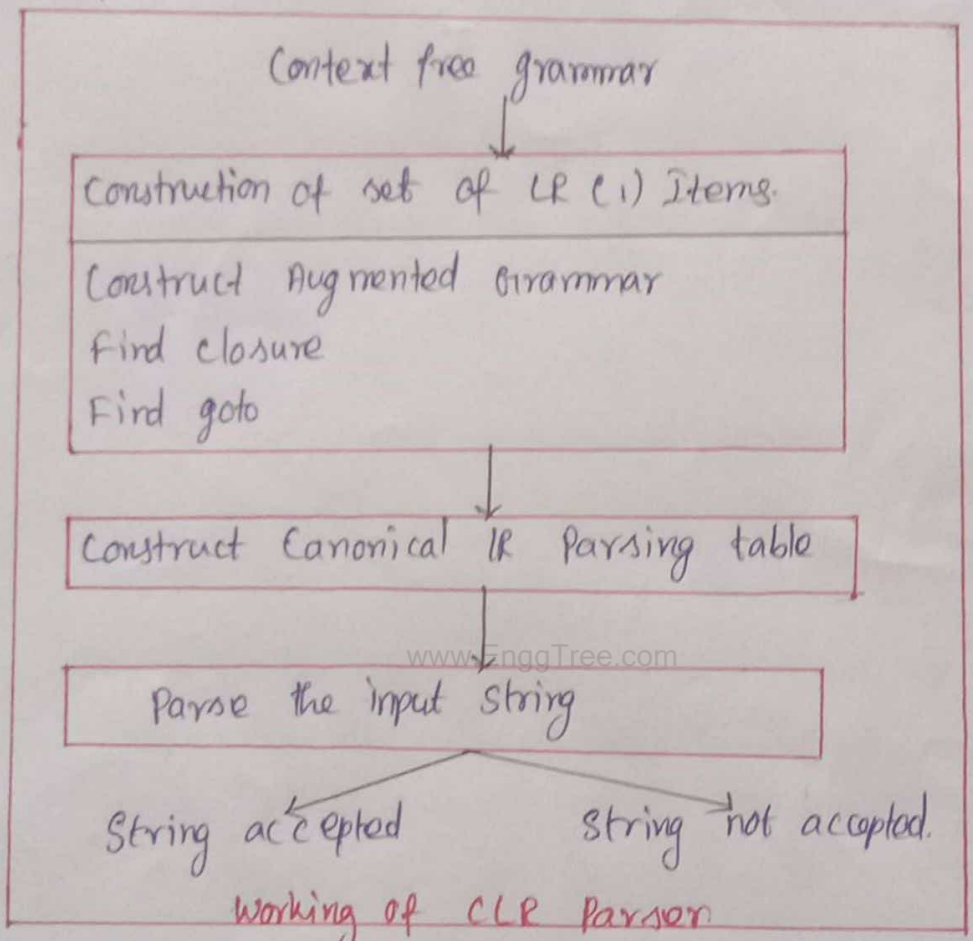
LALR Grammar

A grammar where the LALR parsing table has no multiply represented entries are said to be LALR(1) or LALR grammar.

CLR Parser

CLR defines canonical lookahead. CLR parsing use the canonical collection of LR(1) items to construct the CLR(1) parsing table. CLR(1) parsing table makes the more number of states as compare to the SLR(1) parsing. In the CLR(1), we can locate the reduce node only in the lookahead symbols.

working of CLR Parser



Working of CLR Parser

Error handling & Recovery in Syntax Analyzer

A compiler error occurs whenever it fails to compile a line of code or an algorithm due to a bug in the code or a bug in the compiler itself. The goals of the error handling process are to identify each error, display it to the user, & then develop & apply a recovery strategy to handle the error.

Features of an Error Handler

1. Error Detection
2. Error Reporting
3. Error Recovery

Types of Errors

1. Compile time errors
2. Runtime errors
3. logical errors

Compile time errors

Compile time errors appear during the compilation process before the program is executed. This can be due to a syntax error or a missing file reference that stops the application from compiling properly.

Types of compile time errors

1. lexical phase errors
2. Syntactic phase errors
3. Semantic errors

lexical phase errors

Misspellings of identifiers, keywords, or operators fall into this category. These errors occur both at the lexical phase & during program execution.

Ex

```
class Factorial {
public static void main(String args[]){
int i, fact=1; int number=5;
for(i=1; i<=number; i++){
fact=fact*i;
}System.out.println("Factorial of "+number+" is: "+fact);}}
```

error: illegal start of expression

Syntactic Phase errors

These problems arise during the syntactic phase & execution. These issues occur when there is an imbalance in the parenthesis or when some operators are missing.

Typically syntactic errors look like this:

* Discrepancies in the structure
* The operator is absent
* Misspelled keywords
* Parenthesis that are not balanced

ex

```
class Factorial {
public static void main (String args[]) {
int i, fact = 1; int number = = 5;
for (i=1; i ≤ number ;i++) {
fact = fact * i;
} system.out. println ("Factorial of "+ number +" is : " + fact);
}}
```

error: invalid expression (= =) instead.

Semantic Errors

These errors are detected during the compilation time of a program when they occur during the semantic analysis step.

Some of the semantic errors are

* Inappropriate types of operands
* variables that were not declared
* Actual arguments do not match a formal argument.

ex:

```
class Factorial{
public static void main (String args [])
int i, number = 5;
for (i=1; i ≤ number; i++){
fact = fact * 1;
} System.out. println (" Factorial of "+ number +" is ; " + fact);
}}
```

error: cannot find symbol.

Runtime errors

A run-time error occurs during the execution of a program & is most commonly caused by incorrect system parameters or improper input data. This can include a lack of memory to run an application, a memory conflict with another software, or a logical error, which is an example of run-time error. These are errors that occur when a user enters improper syntax into a code or enters code that a typical compiler cannot run.

logical errors

When programs execute poorly & yet don't terminate abnormally, logic errors occur. A logic error can result in unexpected or unwanted outputs or other behavior even if it is not immediately identified as such. These are errors that occur when the specified code is unreachable or when an infinite loop is present.

Modes of error Recovery

The compiler's simplest requirement is to simply stop, issue a message, & halt compiling. To cope with problems in the code, the parser can implement one of five typical error-recovery mechanisms in the following which are some of the most prevalent recovery strategies.

1. Panic Mode Recovery

This method involves removing successive characters from the input one by one until a set of synchronized tokens are found.

ex: int a, $b, sum, 52;

2. Statement Mode Recovery

When a parser detects an error, it attempts to rectify the problem so that the rest of the statement's inputs allow the parser to continue parsing. Inserting a missing semicolon, replacing a comma with a semicolon, & so forth. Designers of parsers must exercise caution here, as one incorrect correction could result in an unending cycle.

3. Error Productions

If a compiler designer is aware of prevalent errors, these types of faults can be recovered by enhancing the grammar with error production that results in improper constructs.

4. Global corrections.

The parser looks over the entire program & tries to figure out what it's supposed to accomplish, then finds the closest match that's error-free. When given an incorrect input x, it generates a parse tree for the closest error-free y.

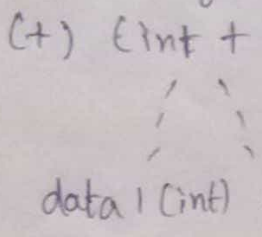
5. Using Symbol Table

The errors are retrieved by using a symbol table for the relevant identifier. & the compiler does automated type conversion if the data types of two operands are

incompatible.

ex: int data1 = "10";
string data2 = "12";
int data3 = data1 + data2;

after compiling this statement, it will result in the following parse tree:



The diagram given below depicts in which phases the recovery methods can be applied.

Recovery Methods \ Error	lexical Phase Error	Syntactic Phase Error	Semantic Phase Error
Panic Mode	✓	✓	X
Statement Mode	X	✓	X
error production	X	✓	X
Global Production	X	✓	X
Using symbol table	X	X	✓

Syntax Directed Definitions.

A Context free grammar with attributes & rules is called a Syntax-directed definition. The properties are linked to the grammar symbols in an extended CFG. The rules are associated with grammar production. A grammar symbol's attribut can now be integers, types, table references, or a string.

ex:

$E \rightarrow E_1 + T \; \{ E.val = E_1.val + T.val \}$

Annotated Parse Tree.

The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

Types of attributes

1. Synthesized Attributes
2. Inherited Attributes

Bottom up evaluation.

a bottom up evaluation of an SDD, the attributes are evaluated in the order in which the parse tree is constructed Followed by the attributes of the internal nodes in a bottom up fashion. Types of attributes.

Synthesized Attributes

These are those attributes which derive their values from their child nodes. That is value of synthesized attribute at node is computed from the values of attributes at child nodes in parse tree.

ex $E \rightarrow E_1 + T \; \{ E.val = E_1.val + T.val \}$

where, E.val derive its values from $E_1.val$ & T.val.

Computation of Synthesized attributes

write the SDD using appropriated semantic rules for each production in given gramm

The annotated parse tree is generated & attribute values are computed in bottom up manner.

The value obtained at root node is the final output.

ex. Consider the following grammar

$S \rightarrow E$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

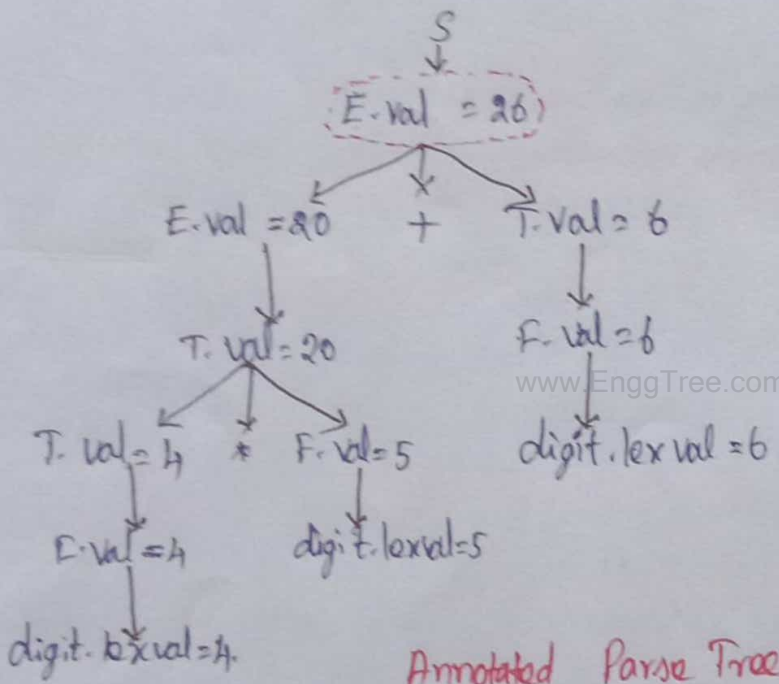
$T \rightarrow F$

$F \rightarrow digit$

The SDD for the above grammar can be written as follow

Production	Semantic Actions
$S \to E$	Print (E.val)
$E \to E_1 + T$	E.val = E_1.val + T.val
$E \to T$	E.val = T.val
$T \to T_1 * F$	T.val = T_1.val * F.val
$T \to F$	T.val = F.val
$F \to$ digit	F.val = digit.lexval

An input string 4 * 5 + 6 for computing synthesized attributes. The annotated parse tree for the input string is



Annotated Parse Tree

Inherited Attributes

These are the attributes which derive their values from their parent or sibling nodes. That is value of inherited attributes are computed by value of parent or sibling nodes

ex: $A \to BCD$ { C.in = A.in, C.type = B.type }

Computation of inherited Attributes

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

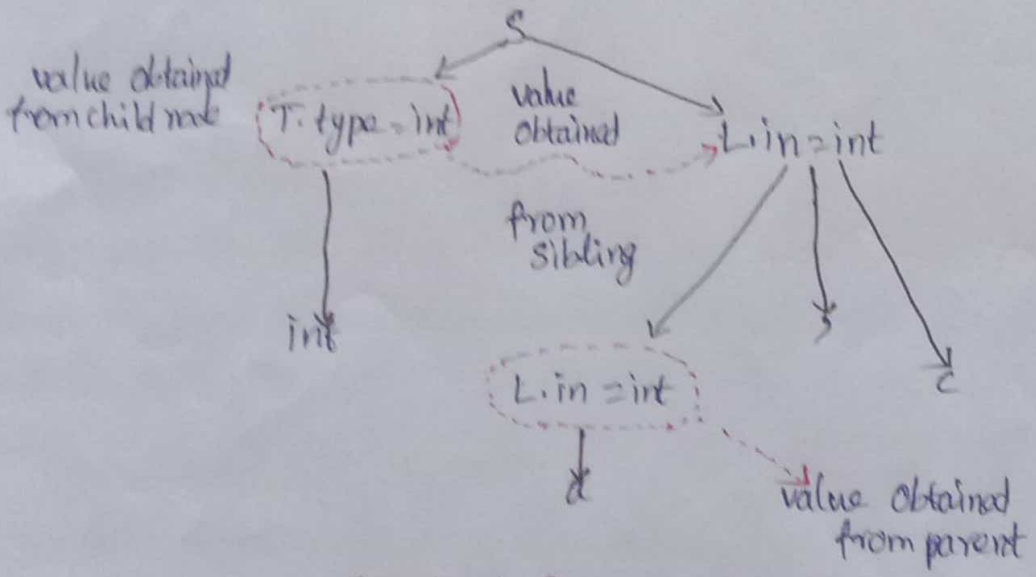
ex: consider the following grammar

S → TL
T → int
T → float
T → double
L → L1, id
L → id

The SDD for the above grammar can be written as follow.

Production	Semantic Actions
S → TL	L.in = T.type
T → int	T.type = int
T → float	T.type = float
T → double	T.type = double
L → L1, id	L1.in = L.in Entry_type(id.entry, L.in)
L → id	Entry_type(id.entry, L.in)

An input string int a, c for computing inherited attributes. The annotated parse tree for the input string is



Annotated Parse Tree

Types of Syntax Directed Definitions - Predictive Translator

1. S-attributed translation
2. L-attributed Translation.

S-attributed translation

If the nodes attributes are synthesised attributes, the SDD is S-attributed. For evaluation of an S-attributed SDD, the nodes of the parse tree can be traversed in any bottom-up sequence.

L-attributed Translation

If the attributes of nodes are synthesized or inherited, an SDD is L-attributed. The parse tree can now be traversed exclusively from left to right. This is because L stands for left to right traversal in 'L-attributed translation'.

Start analyzing the L-attributed SDD in the following order:

1. The inherited attribute's value.
2. The value of synthesizing attributes.

Syntax Directed Translation

Syntax directed translation adds 'semantics rules' or actions to CFG Products. As a result, refer to the grammar that has been augmented as attributed grammar.

Construction of Syntax Tree

The Syntax tree nodes can all be treated as data with several fields. The operator is identified by one node element, where as the remaining areas include a pointer to the operand nodes. The node's label as also known as the operator.

The following functions are used to generate the syntax tree nodes for expressions using binary operators.

1. mknode (op, left, right):

It creates an operator node with the name op & two fields, containing left & right pointers

2. mkleaf (id, entry):

It creates an identifier node with the labled & the entry field, which contains a reference to the identifier's symbol table entry.

3. mkleaf (num, val):

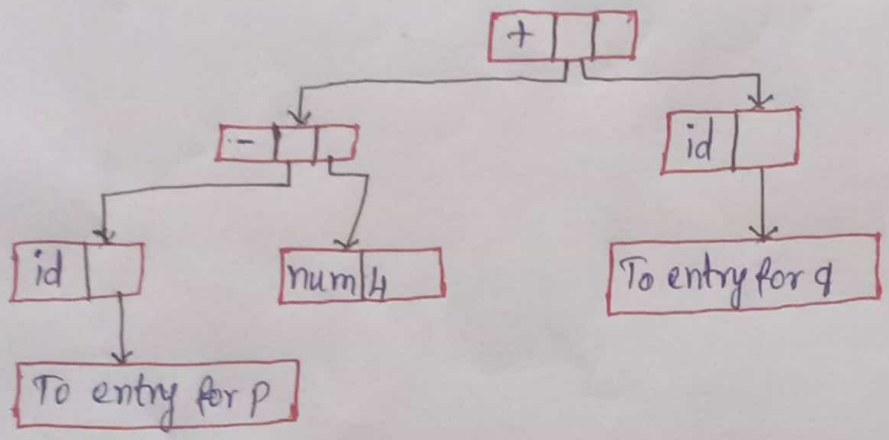
It creates a number node with the name num & a field containing the number's value, val.

ex:

Construct a syntax tree for an expression p-4+q. In this sequence a1, a2, ... a5 are pointers to the symbol table entries for identifier 'p' & '4' respectively.

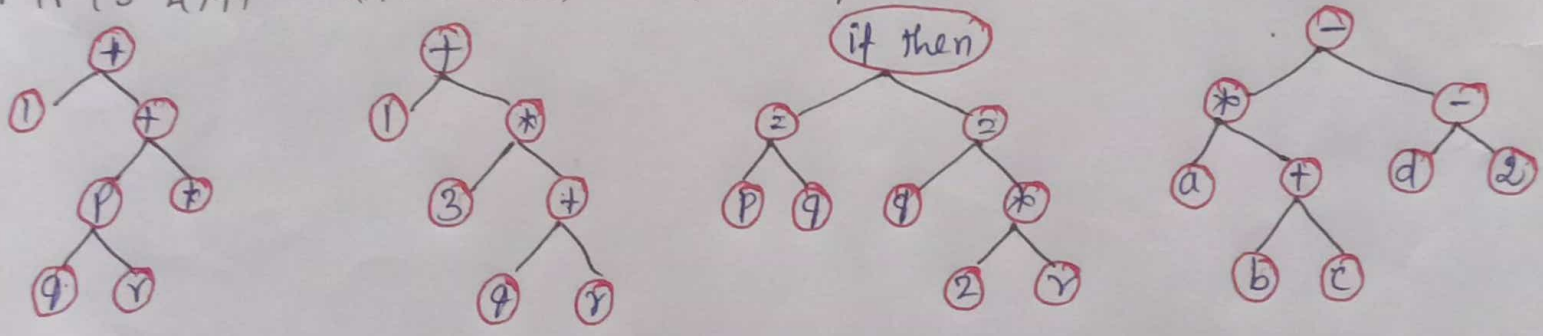
a1 - mkleaf (id, entry p);
a2 - mkleaf (num, 4);
a3 - mknode ('-', a1, a2)
a4 - mkleaf (id, entry q)
a5 - mknode ('+', a3, a4);

The tree is contructed from the ground up. The leaves for p & 4 are created using mkleaf (id, entry p) & mkleaf (num 4), respectively.



Syntax Tree for p-4+q

Ex
1. 1+(3*4)+1
2. 1+3*(4+1)
3. if p=q then q=2*r
4. a*(b+c)-d/2



Type Systems

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, & the rules for assigning types of language constructs.

ex: if both operands of the arithmetic operators of +, - & * are of type integer then the result is of type integer.

A type system is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a type system. It is specified in a syntax-directed manner. Different type systems may be used by different compilers or processors of the same language.

Specification of a Simple type checker

A type checker for a simple language checks the type of each identifier. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements & functions.

A Simple language

Consider the following grammar:

$P \rightarrow D ; E$

$D \rightarrow D ; D \mid id : T$

$T \rightarrow char \mid integer \mid array [num] of T \mid \uparrow T$

$E \rightarrow literal \mid num \mid id \mid E \text{ mode } E \mid E [E] \mid E \uparrow$

Translation scheme

$P \rightarrow D ; E$

$D \rightarrow D ; D$

$D \rightarrow id : T \{ addtype (id.entry, T.type) \}$

$T \rightarrow char \{ T.type := char \}$

$T \rightarrow integer \{ T.type := integer \}$

$T \rightarrow \uparrow T_1 \{ T.type := pointer (T_1.type) \}$

$T \rightarrow array [num] of T_1 \{ T.type := array (1 \ldots num.val, T_1.type) \}$

There are two basic types : Char & integer ; → type_error is used to signal errors;

The prefix operator ↑ builds a pointer type. Example, ↑integer leads to the type expression pointer (integer).

Equivalence of Type Expressions

In programming languages, type checking is the process of verifying the compatibility of data types used in expressions. When comparing expressions for equivalence, type checking ensures that the types of operands on both sides of the comparison operator are compatible. Equivalence of expressions is checked through type checking.

Basic Types

In many programming languages, basic types such as integers, floating, characters & booleans are compared for equivalence using the appropriate comparison operators.

for example, in C, the equivalence of two integers is checked by comparing their values using the == operator.

Object types

In languages that support object oriented programming, equivalence of expressions involving objects may have different rules based on the languages type system. For content based equivalence, such as comparing the values of two objects, the equals() method is often used.

Type Conversion & Promotion

Type checking also involves implicit or explicit type conversion or promotion to ensure compatibility when comparing different types.

For example, in C, when comparing an integer with a floating no the integer may be promoted to a floating-point type before the comparison.

Type errors

Type errors typically result in a compilation error or runtime exception, depending on when the type checking occurs in the languages execution model.

Two categories of type equivalence:

1. Structural equivalence
2. Name equivalence.

1. Structural Equivalence

Replace the named types by their definitions & recursively check the substituted trees. If type expressions are built from basic types & constructors then those expressions are called structurally equivalent.

Ex:

S1	S2	Equivalence	Reason
Char	Char	S1 equivalent to S2	Similar basic types
Pointer (char)	Pointer (char)	S1 equivalent to S2	Similar constructor ptr to the char type

2. Name Equivalence

Two type expressions are name equivalent if & only if they are identical, that

is if they can be represented by the same syntax tree, with the same labels.

ex:

```
typedef struct Node
{
  int x;
} Node;
Node *first, *second;
struct Node *last1, *last2;
```

The variables first & second are name equivalent similarly last1 & last2 are name equivalent.

Type Conversions.

The type conversion is an operation that takes a data object of one type & creates the equivalent data objects of multiple types. The signature of a type conversion operation is given as

conversion_op : type1 → type2.

There are two types of type conversions which are as follows:-

1. Implicit type conversion
2. Explicit type conversion

Implicit type conversion

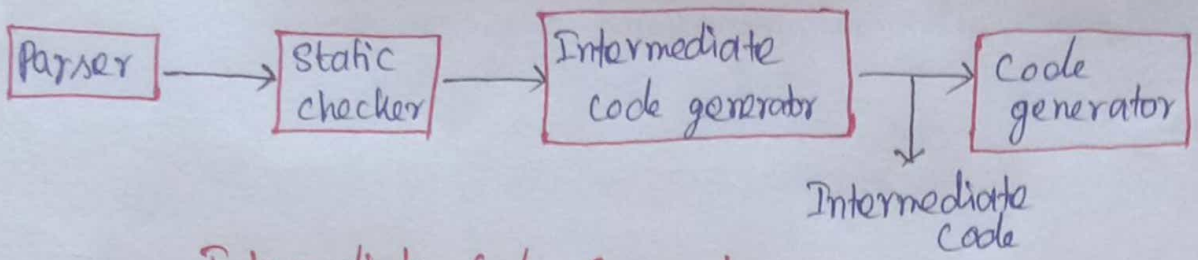
The programming languages that enable mixed-mode expressions should describe conventions for implicit operand type conversions.

Explicit type conversion

Some languages support few efficiencies for doing explicit conversions, both widening & narrowing. In some cases, warning messages are created when an explicit narrowing conversion results in a meaningful change to the value of the object being modified.

Intermediate languages

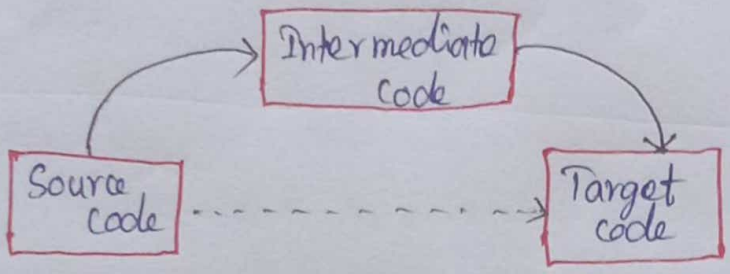
If can directly translate a source code into its target machine code, why do need to translate it into intermediate code that is then translated to its target code.



Intermediate Code Generator

Intermediate Code Generation

The source code is translated into machine code using intermediate code. Between the high-level language & the machine language is intermediate code.

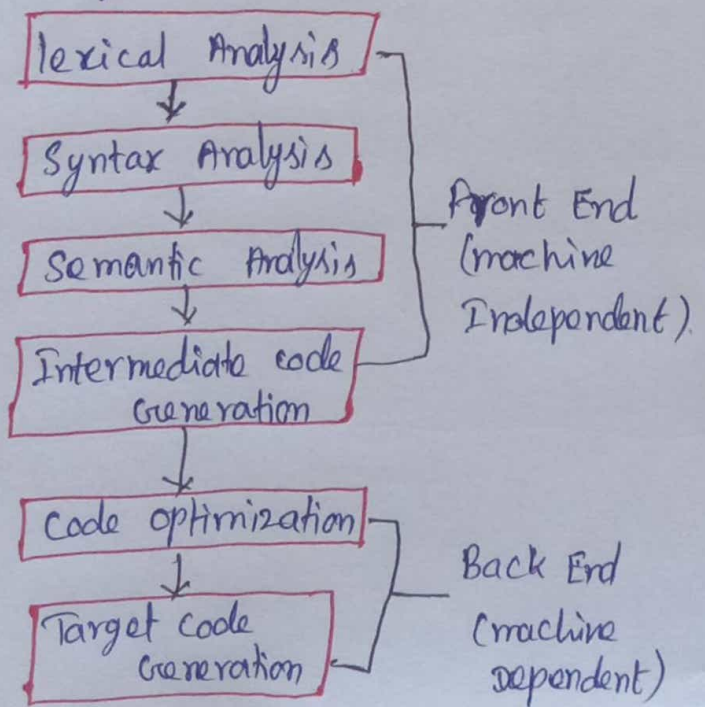


*A compiler translates the source language to its target machine language without having the option of generating intermediate code then for each new machine. In that case require a full native compiler.

*Intermediate code eliminates the need for a new full compiler for every unique machine by keeping the analysis portion the same for all the compilers.

*The second part of the compiler, synthesis, changes according to the target machine.

*The source code modifications to improve code performance by using code optimization techniques on the intermediate code.



*No actual back-end is required to be written for each new machine.

*The intermediate form may be more compact than machine code. It saves space in distribution & on the machine that executes the programs.

Intermediate Representation.

Represent intermediate code in two ways:.

1. High-level intermediate representation.
2. low-level intermediate representation.

High-level intermediate representation

High-level intermediate code representation is very close to the source language itself. It can quickly generate from the source code & can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.

low-level Intermediate representation

This one is close to the target machine, making it suitable for register & memory allocation, instruction set selection, etc. It is ideal for machine-dependent optimization.

Postfix Notation

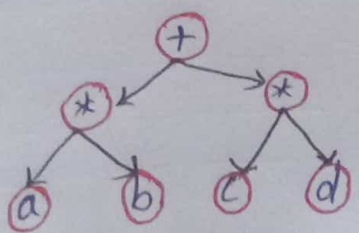
The standard (infix) way of writing the sum of a & b is with the operator in the middle: a+b. The operator at the right end of the phrase in postfix notation as ab+. Because the operators location & arity (number of arguments) allow only one way to decode a postfix expression, it requires no parentheses in postfix notation. The operator comes after the operand in postfix notation.

ex:
The postfix representation of the expression (A/(B-C)*D+) is: ABC-/D*E+.

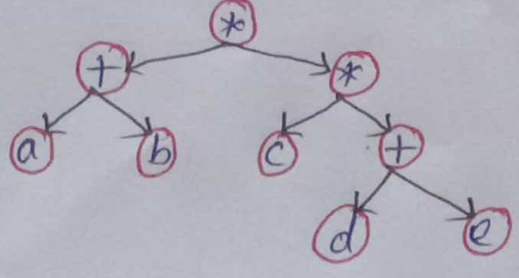
Syntax tree

The Syntax tree is nothing other than a reduced form of a parse tree. The operator & keyword nodes of the parse tree to their parents, & the single link in the syntax tree replaces a chain of single productions.

ex: (a*b)+(d*e)



(a+b)*(c*(d+e))



Three Address Code

Three address code is a sort of intermediate code that is simple to create & convert to machine code. It can only define an expression with three addresses & one operator. The three address codes help in determining the sequence in which operations are actioned by the compiler.

Implementation of Three Address Code

There are three different ways to express three address codes:

1. Quadruple
2. Triples
3. Indirect Triples

Quadruple

It is a structure that has four fields: op, arg1, arg2, & result. The operator is denoted by op, arg1 & arg2 denote the operands, & the result is used to record the outcome of the expression. These quadruples play a crucial role in breaking down high-level digestible parts, facilitating compilation-stage analysis & optimization procedures.

ex:

Convert $a = -b * c + d$ into three address codes.

The following is the three-address code:

$t_1 = -b$

$t_2 = c + d$

$t_3 = t_1 * t_2$

$a = t_3$

Quadruples are used to symbolize these statements:

#	Op	Arg1	Arg2	Result
0	Unimus	b	-	t_1
1	+	c	d	t_2
2	*	t_1	t_2	t_3
3	=	t_3	-	a

Triples

Instead of using an additional temporary variable to represent a single action.

a pointer to the triple is utilized when a reference to another triple's value is required. As a result, it only has three fields: op, arg1, & arg2. 62

ex:

Convert $a = -b * c + d$ into three address codes.

The following is the three-address code:

$t_1 = -b$

$t_2 = c + d$

$t_3 = t_1 * t_2$

$a = t_3$

The following triples represent these statements:

#	op	Arg1	Arg2
0	uminus	b	-
1	+	c	d
2	*	0	1
3	=	2	-

Indirect Triples

This approach employs a pointer to a list of all references to computations that are created & kept separately. Its usefulness is comparable to quadruple representation, however, it takes up less space. Temporaries are easy to rearra since they are implicit.

ex:

Convert $a = b * -c + b * -c$ into three address codes.

The following is the three-address code:

$t_1 = \text{uminus } c$

$t_2 = b * t_1$

$t_3 = \text{uminus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

#	op	Arg1	Arg2
14	uminus	c	-
15	*	14	b
16	uminus	c	-
17	*	16	b
18	+	15	17
19	=	a	18

list of pointer to the table

#	Statement
0	14
1	15
2	16
3	17
4	18
5	19

Types & Declarations

Typical basic types & declarations for a language contain boolean, char, integer, float, & void. void denotes the absence of a value. A type name is a type expression. Declaration involves allocating space in memory & entering type & name in the symbol table.

Applications of Types & Declarations

Type checking

Translation Applications.

Type checking

It uses logical rules to reason regarding the behavior of a program at run time. It ensure that the operands match the type that an operation expects.

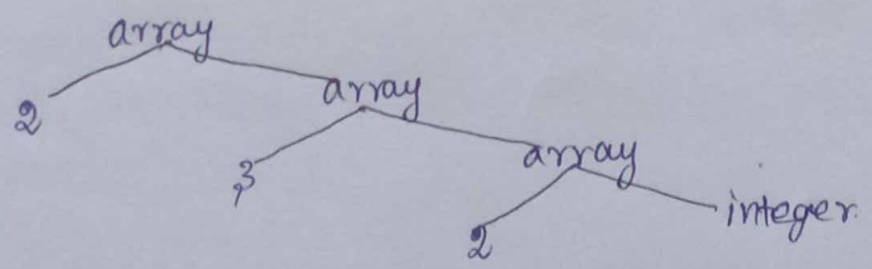
Translation Applications

A compiler can determine the storage that need for that name (declared within a procedure or a class). at run time from the type of name. Type information is also required to calculate the address denoted by an array reference to insert explicit type conversions & to choose the correct version of an arithmetic operator, among other things.

Type expressions

A primary type is a type of expression. Typical basic types for a language contain boolean, char, integer, float, & void. A type name or type expression by applying the array type constructor to a number & a type expression.

ex:



The array type int [2][3][2] can be read as an array of 2 arrays each of 3 arrays of 2 integers each & written as a type expression array (2, array (3, array (2, integer))).

Basic types

char, int, double, float are type expressions.

Type names

It is convenient to consider that names of types are type expressions.

Arrays

If E is a type expression & n is an int, then

array (n, E)

is a type expression indicating the type of an array with elements in T & indices in the range 0 ... n-1.

Products

If E_1 & E_2 are type expressions then

$E_1 * E_2$

is a type expression indicating the type of an element of the cartesian produ of E_1 & E_2. The products of more than two types are defined similarly & this product operation is left-associative. Hence,

$(E_1 * E_2) * E_3$ & $E_1 * E_2 * E_3$ are equivalent.

Records

The only difference between a record & a product is that the fields of a record have names. If abc & xyz are two type names, if E_1 & E_2 are type expressions then

record (abc : E_1, xyz : E_2)

is a type expression indicating the type of an element of the cartesian product of T_1 & T_2.

Pointers

let E is a type expression, then

pointer (E)

is a type expression denoting the type pointer to an object of type T.

Function types

If E_1 & E_2 are type expressions, then

$E_1 \rightarrow E_2$

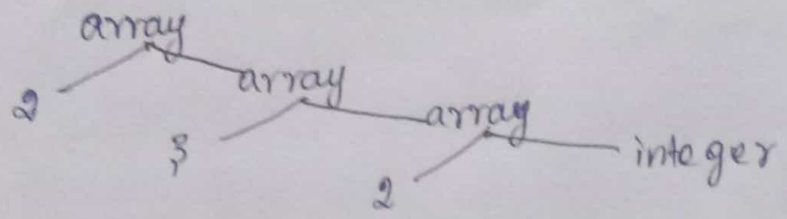
is a type expression denoting the type of functions associating an element from E_1 with an element from E_2.

Type Equivalence

when graphs represent type expressions, two types are structurally equivalent if & only if one of the following conditions is true:

* They are of the same basic type.
* They are formed by applying the identical constructor to structurally equivalent types.
* One is a type name that refers to the other.

example of a non cyclic graph:



Declarations

When encounter declarations, need to layout storage for the declared variable For every local name in a procedure, create a Symbol Table (ST) entry including

1. The type of the name
2. How much storage the name requires

Grammar

D → real, id
D → integer, id
D → D1, id

let use ENTER to enter the symbol table & use ATTR to trace the data type

Production Rule	Semantic Action
D → real, id	ENTER (id.PLACE, real) D.ATTR = real
D → integer, id	ENTER (id.PLACE, integer) D.ATTR = integer
D → D1, id	ENTER (id.PLACE, D1.ATTR) D.ATTR = D1.ATTR

Translation of Expressions

The assignment statement is used to deal with expressions in syntax-directed translation. The type of the expression can be real, integer, array or record. A Multi-operator expression, such as $a+b*c$ will be translated into instructions with only one operator per instruction. There can only be one operator on the right side of instruction in a three-address code, no built-up arithmetic expressions are allowed. As a result a source language statement like $x+y*z$ might be translated into a three-address instruction sequence.

$t_1 = y * z$

$t_2 = x + t_1$

where t_1 & t_2 are temporary names produced by the compiler.

Operations within Expressions

Using attribute code for S & attributes addr & code for an expression E, the syntax-directed definition in the below figure constructs the three-address code for an assignment statement S. S.code & E.code are attributes that represent the three-address code for S & E, respectively.

Production	Semantic Rules
$S \rightarrow id = E;$	S.code = E.code || gen(top.get(id.lexeme) '=' E.addr)
$E \rightarrow E_1 + E_2$	E.addr = new Temp() E.code = E_1.code || E_2.code || gen(E.addr '=' E_1.addr '+' E_2.addr)
$\mid -E_1$	E.addr = new Temp() E.code = E_1.code || gen(E.addr '=' 'minus' E_1.addr)
$\mid (E_1)$	E.addr = E_1.addr E.code = E_1.code
$\mid$ id	E.addr = top.get(id.lexeme) E.code = ''

Three-address code for expressions

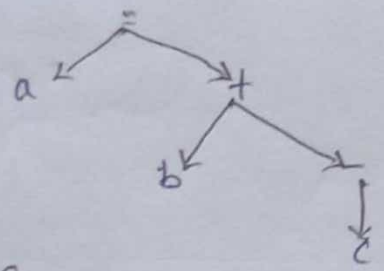
This is the translation schema.

Consider the last production, E → . id, in the syntax-directed definition in

* top denote the current symbol table.
* Function top.get retrieves the entry
* Attribute E.addr denotes the address that will hold the value of E.
* new Temp() creates t1, t2 ...
* gen ('x' = 'y' + 'z') represents the three address instruction x = y + z.

Expression appearing in place of variables like x, y, z & z are evaluated when passed to gen.

The Syntax-directed definition converts the assignment phrase a = b + c, into a three-address code sequence



a = b + - c

⇒ t2 = minus c

t1 = b + t1

a = t2.

Incremental Translation.

The code property is not needed in the incremental technique since a single sequence of instructions is formed by successive calls to gen. The semmantic rule in the generating three-address code for expressions incrementally for E → . E1 + E2 merely calls gen to construct an add instruction. the instructions to calculate E1 into E1. addr & E2 into E2. addr have already been generated.

E. addr = new Node ('+', E1. addr, E2. addr);

E → E1 + E2

E. addr = new Node ('+', E1. addr, E2. addr);

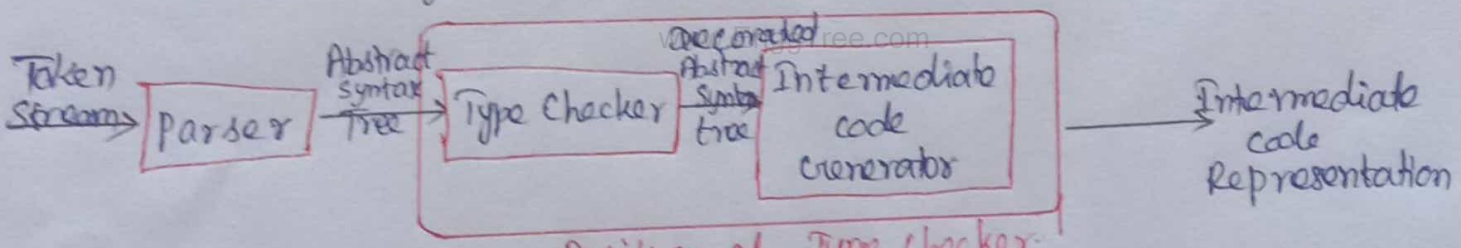
A syntax tree can also be built using the method generating three-address code for expressions incrementally. The new semantic action for E → E1 + E2 uses a function object e) & [native code]} to create a node.

$S \rightarrow id = E;$	{gen (top.get (id.lexeme) '=' E.addr); }
$E \rightarrow E_1 + E_2$	{ E.addr = new Temp (); gen (E.addr '=' E_1.addr '+' E_2.addr); }
$\mid -E_1$	{ E.addr = new Temp (); gen (E.addr '=' 'minus' E_1.addr); }
$\mid (E_1)$	{ E.addr = E_1.addr; }
$\mid id$	{ E.addr = top.get (id.lexeme); }

Generating three-address code for expressions incrementally

Type checking

The technique of verifying & enforcing type restrictions in values is type checking. The type checking phases of compiler design is interleaved with the syntax analysis phase, so it is done before a program's execution or translation (static typing). & the information is gathered for used by subsequent phases.



Position of Type checker

Types of Type checking

There are two kinds of type checking:

1. Static type checking
2. Dynamic type checking

Static type checking

Static type checking is defined as type checking performed at compile time. It checks the type variables at compile-time, which means the type of the variables is known at the compile time.

Examples of static checks include

Type checks

A compiler should report an error if an operator is applied to an incompatible operand.

The flow of control checks

Statements that cause the flow of control to leave a construct must have someplace to which to transfer the flow of control.

Uniqueness checks

There are situations in which an object must be defined only once.

Name-related checks

Sometimes the same name may appear two or more times.

Dynamic Type checking

Dynamic type checking is defined as the type checking being done at run time. In dynamic type checking, types are associated with values, not variables. Implementation of dynamically type-checked languages runtime objects are generally associated with each other through a type tag, which is a reference to a type containing its type information. Dynamic typing is more flexible. A static type system always restricts what can be conveniently expressed.

Overloading

An overloading symbol is one that has different operations depending on its context.

overloading is of two types

1. operator overloading
2. Function overloading

Operator overloading

The arithmetic expression "x+y" has the addition operator '+' is overloaded because '+' in "x+y" have different operators when 'x' & 'y' are integers complex numbers, reals, & matrices.

Function overloading

The type checker resolves the function overloading based on types of arguments & numbers.

ex: $E \rightarrow E_1 (E_2)$

{

$E.type := if\ E_2.type = S$

$E_1.type = S \rightarrow t$ then t

else type_error }

Back Patching

Backpatching is a technique used in compiler design to delay the assignment of addresses to code or data structures until a later stage of the compilation process. This allows the compiler to generate code with placeholder addresses that are later updated or backpatched with the correct addresses once they are known. Back patching is commonly used in compilers for languages that support complex control structures or dynamic memory allocation.

One-Pass code generation using backpatching

In a single pass, backpatching may be used to create a boolean expressions program as well as the flow of control statements. The synthesized properties truelist & falselist of non-terminal B are used to handle labels in jumping code for Boolean statements.

There are three functions to modify the list of jumps:

Make list (i):

Create a new list including only i, an index into the array of instructions & the makelist also returns a pointer to the newly generated list.

Merge (P1, P2);

Concatenates the lists pointed to by P1 & P2 & returns a pointer to the concatenated list.

Backpatch (P, i):

Inserts i as the target label for each of the instructions on the record pointed to by P.

Need for Backpatching

Backpatching is mainly used for two purposes:

1. Boolean expression
2. Flow of control statements
3. Labels & Gotos

Boolean expression

using a translation technique, it can create code for Boolean expressions during bottom-up parsing. In grammar, a non-terminal marker M creates

a semantic action that picks up the index of the next instruction to be created at the proper time.

Ex:

Backpatching using boolean expressions production rules table

Step 1: Generation of the production table

Production rule	semantic action
E E_1 OR M E_2	{backpatch (E_1.flist, M.quad); E.Tlist = merge (E_1.Tlist, E_2.Tlist) E.Flist = E_2.plist}
E E_1 AND M E_2	{backpatch (E_1.Tlist, M.quad); E.Tlist = E_2.Tlist; E.Flist = merge (E_1.Flist. E_2.Plist); }
E NOT E_1	{ E.Tlist := E_1.Plist; E.Flist := E1.Tlist; }
E (E_1)	{ E.Tlist := E_1.Tlist; E.Plist := E1.Plist; }
E id1 relop id2	{ E.Tlist := mklist (nextstate); E.Flist := mklist (next state +); Append ('if' id1.place relop.op id2.place 'goto-'); Append('goto-')}
E true	{ E.Tlist := mklist (next state); Append ('goto-'); }
E false	{ E.Flist := mklist (next state); Append ('goto-'); }
M E	{ m.quad := next quad; }

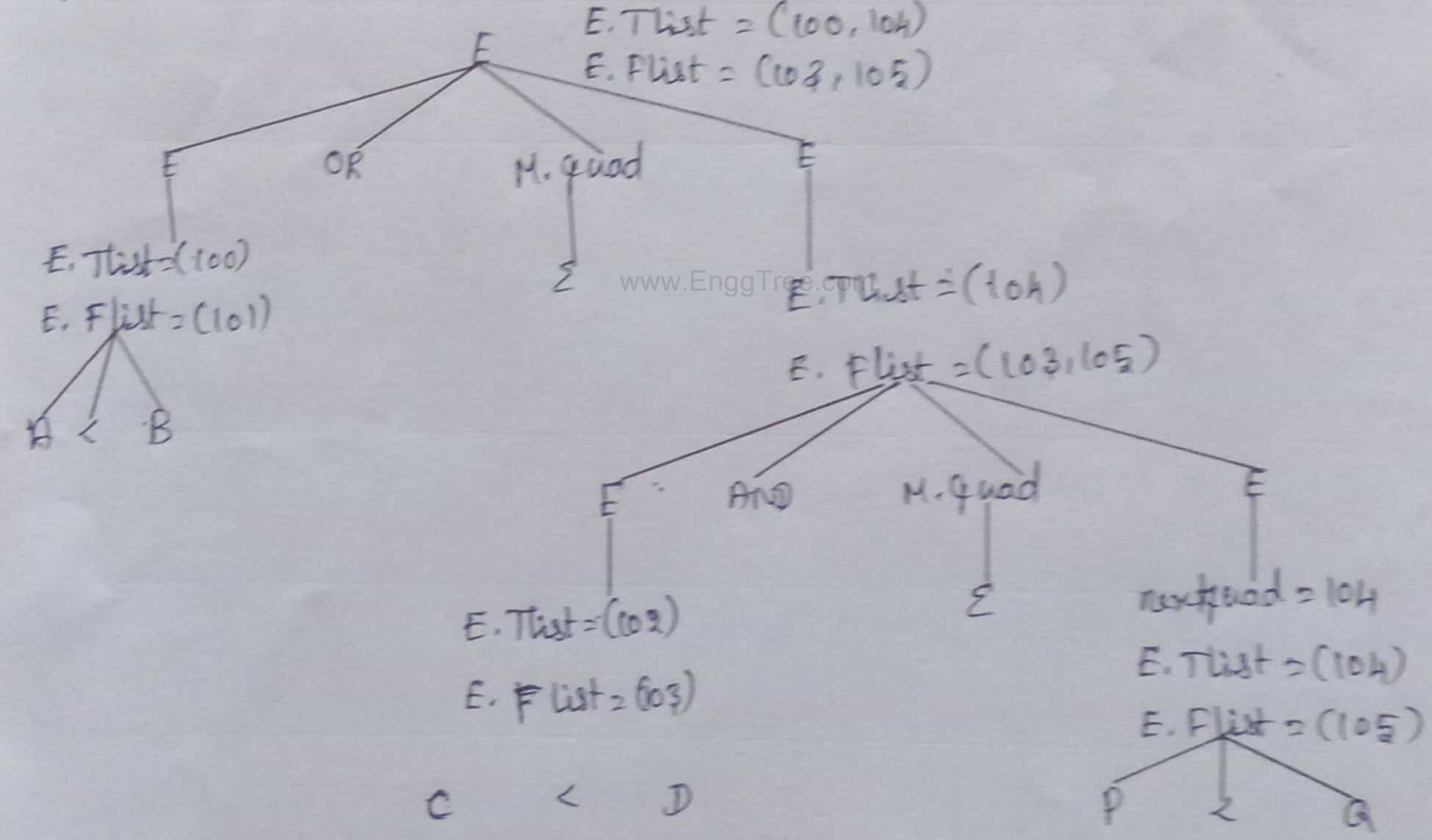
Production Table for Backpatching

Step 2: To find the TAC (Three address code) for the given expression using backpatching:

100 | if A<B goto - Backpat (E1.Flist, 102)
101 | goto 102 E.Tlist = {100, 104}
102 | if C<D goto 104 E.Flist = {103, 105}
103 | goto -
104 | OR | if P<Q goto - Backpatch (E1.Tlist, 104)
105 | goto - M.quad = nextquad
AND E.Tlist := {104}
E.Flist := {103, 105}

Three address codes for the given example

Step 3: Now will make the parse tree for the expression



Parse tree for the example

The flow of control statement

Control statements are those that alter the order in which statements are executed. If, If-else, Switch-case, & while-do statements are examples. Boolean expressions are often used in computer languages to alter the flow of control.

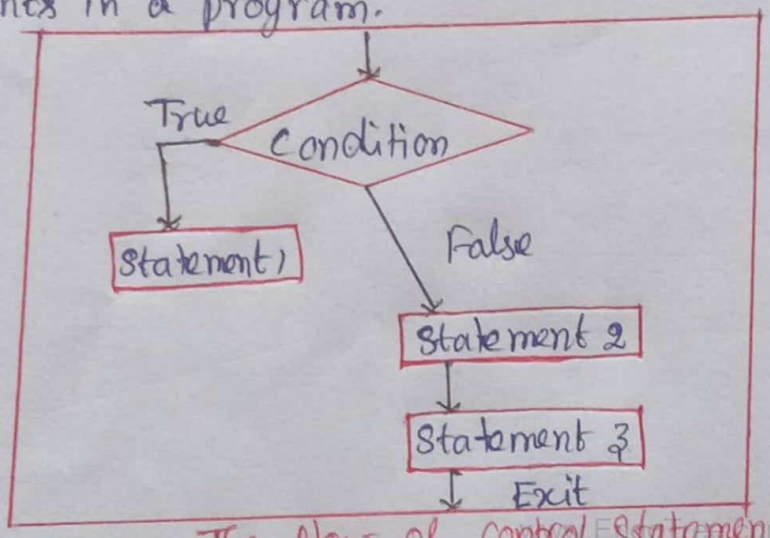
Boolean expressions are conditional expressions that change the flow of

control in a statement. The value of such a boolean statement is implicit
n the program's position.

Compute logical values

During bottom-up parsing, it may generate code for boolean statements vi
a translation mechanism. A non-terminal marker M in the grammar establish
a semantic action that takes the index of the following instruction to be forme
at the appropriate moment.

The flow of control statements needs to be controlled during the execution of
statements in a program.
ple:



The flow of control Statements

Labels & Gotos

labels & Gotos are programming constructs that allow programmers to contr
he flow of execution in their code. A label is a named location in code
hat a goto statement can reference. A goto statement allows a programmer to
ransfer control to the labelled location in code, by passing any statements in
etween, while powerful, goto statements are usually discouraged because
hey make code more difficult to understand & maintain. Goto statements are
ot supported by many computer languages, including Java & Python.

Unit - IV

Runtime Environment

A translation needs to relate the static source text of a program to the dynamic actions that must occur at runtime to implement the program. The program consists of names for procedures, identifiers, etc. that require mapping with the actual memory location at runtime.

Source language issues

Activation tree

A program consist of procedure a procedure definition is a declaration that, in its simplest form, associates an identifier with a statement. Each execution of the procedure is referred to as an activation of the procedure. A procedure is recursive if a new activation begins before an earlier activation of the same procedure has ended.

Properties of activation trees are:

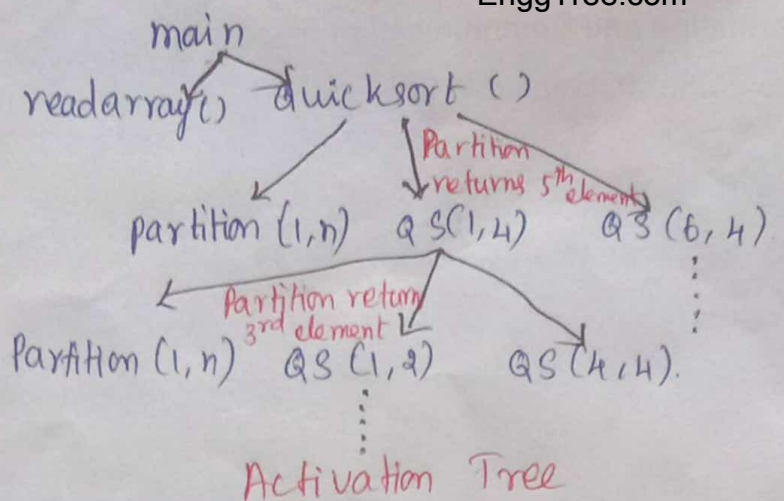
* Each node represents an activation of a procedure.
* The root shows the activation of the main function.
* The node for procedure 'x' is the parent of node for procedure 'y' if & only if the control flows from procedure x to procedure y.

ex: Consider the following program of Quicksort

```
main()
{
   int n;
   readarray();
   quicksort(1,n);
}
quicksort(int m, int n)
{
   int i=partition(m,n);
   quicksort(m,i-1);
   quicksort(i+1,n);
}
```

Activation tree for this program will be:

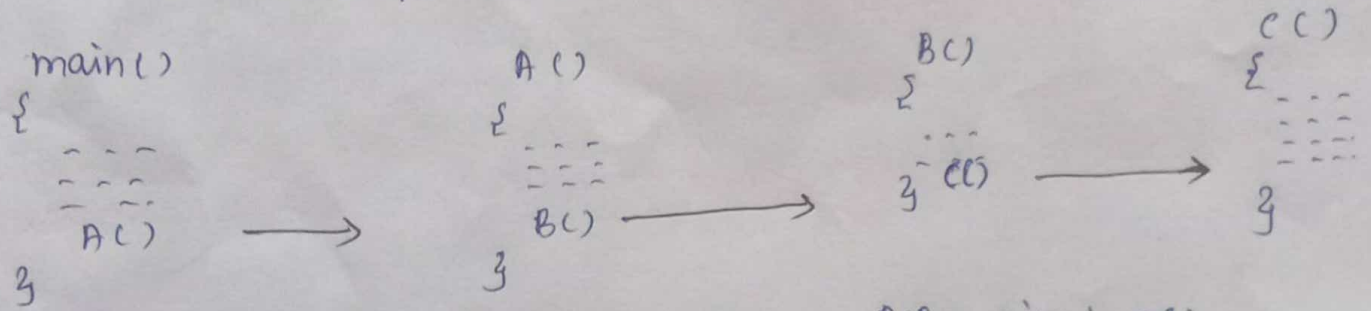
First main function as the root then main calls readarray & quicksort.



Activation Tree

Control Stack & Activation Records

Control Stack or runtime stack is used to keep track of the live procedure activations that is the procedures whose execution have not been completed.



Being going to A() activation record of main is created when A() is called activation record of A() is created.

Before going to B() activation record of main & A() in Stack.

Before going to C() activation record of main & B() is pushed in stack.

A General activation record consists of the following things:

Local variables: Hold the data that is local to the execution of the procedure.

Temporary values: Stores the values that arise in the evaluation of an expression

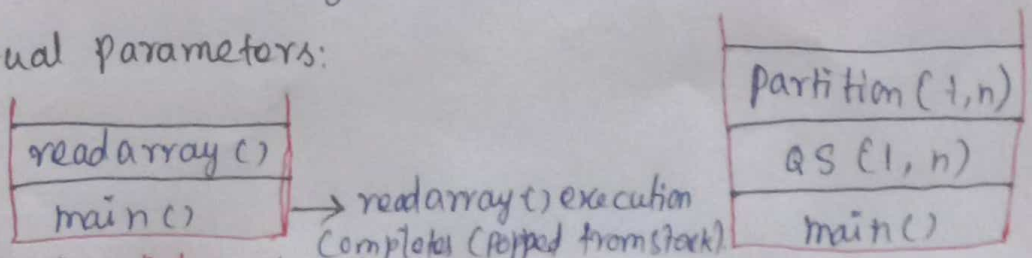
Machine Status: holds the information about the status of the machine just before the function call.

Access link: refers to non-local data held in other activation records.

Control link: Points to activation record of caller.

Return value: used by the called procedure to return a value to calling procedure

Actual Parameters:

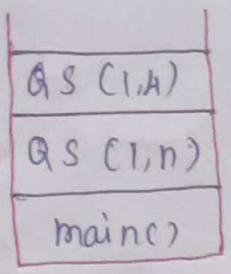


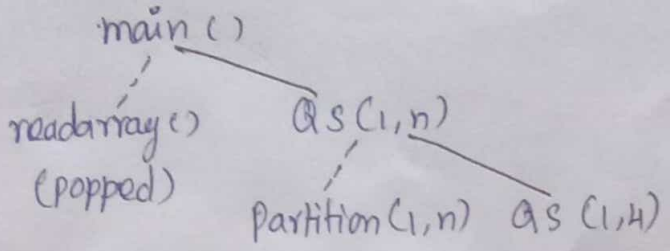
QS is called so it Enters

execution completed (Popped out

Control Stack for the above quicksort example:

Now QS is called again so it enters the stack.





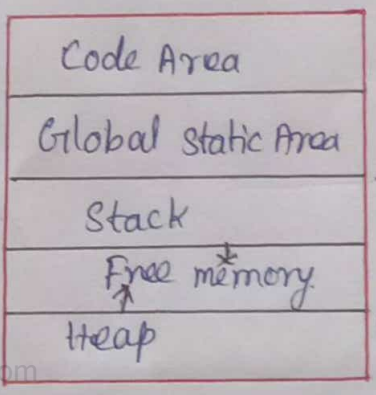
Storage organization

Subdivision of Runtime Memory

Runtime storage can be subdivided to hold:

Target code the program code, is static as its size can be determined at compile time: different components of an executing program:

* Static data objects
* Dynamic data objects - heap
* Automatic data objects - stack.
* Generated executable code.



Subdivision of Runtime memory.

Storage organization

When the target program executes then it runs in its own logical addres space in which the value of each program has a location. The logical address space is shared among the compiler, operating system & target machine for management & organization. The operating system is used to map the logical addre into physical address which is usually spread throughout the memory.

Subdivision of Run-time memory

Code:
Memory locations for code are determined at compile time.

Static data:
locations of static data can also be determined at compile time.

Stack:
Data objects Allocated at Run-time. (Activation Records)

Heap:
other Dynamically Allocated data objects at Run-time. (for example Malloc Area in C

Storage Allocation strategies

A compiler is a program that converts high-level language to low-level language like machine language. There is a need for storage allocation strategies in compiler design.

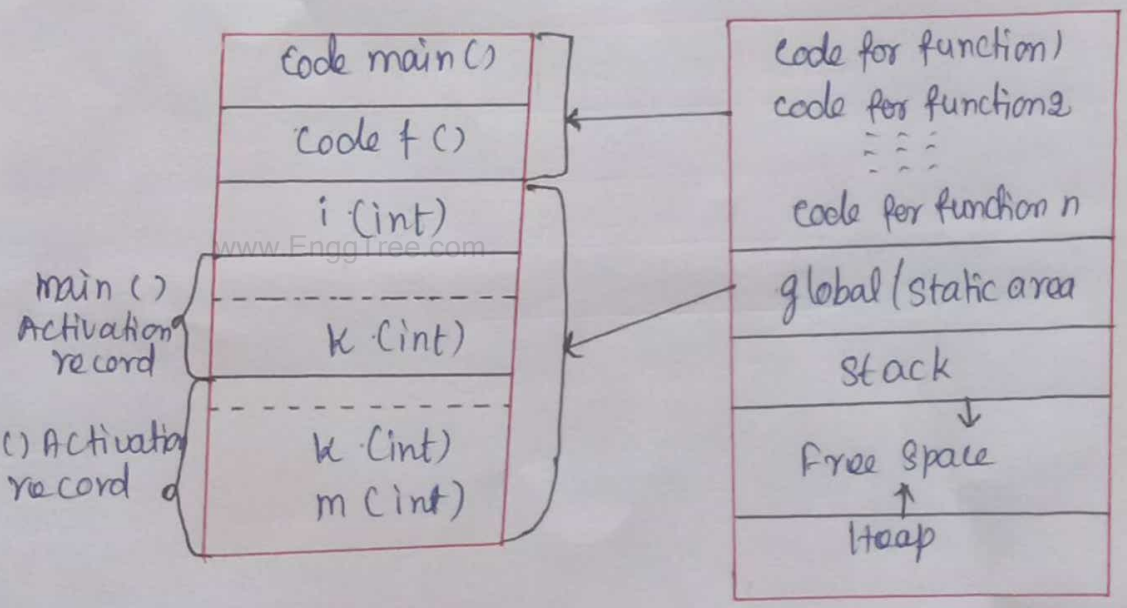
There are mainly three types of storage allocation strategies.

1. Static Allocation
2. Heap Allocation
3. Stack Allocation

Static storage allocation

If memory created at compile time then the memory will be created in static area & only once. Static allocation supports the dynamic data structure that means memory is created only at compile time & deallocated after program completion.

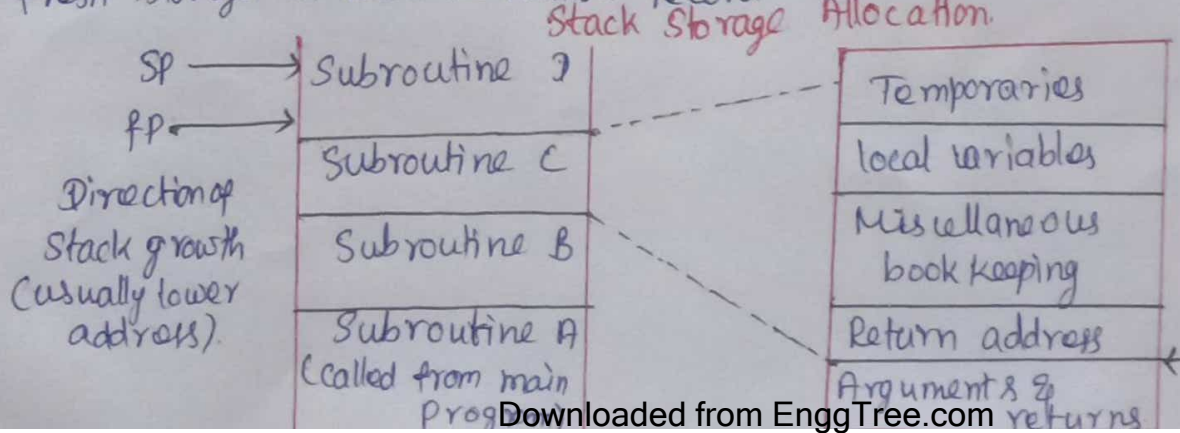
```
int i = 10;
int f (int i)
{
  int k;
  int m;
  ...
}
main()
{
  int k;
  f(k);
}
```



Static Allocation.

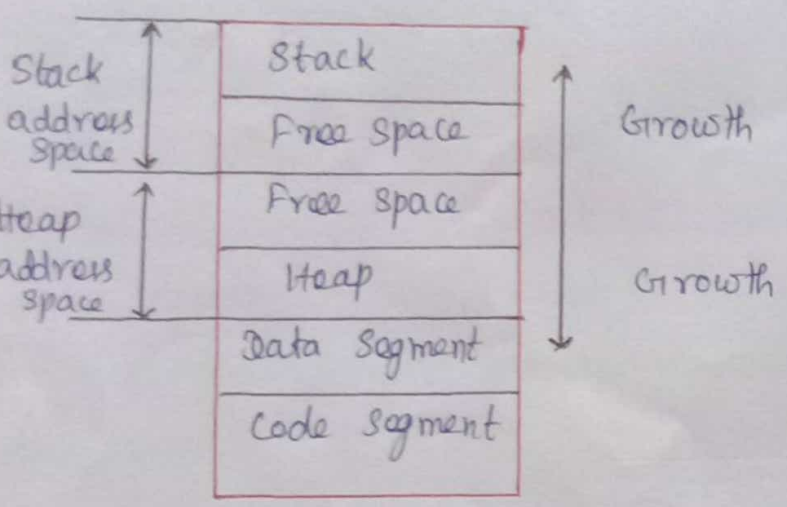
Stack Storage Allocation

An activation record is pushed into the stack when activation begins & it is popped when the activation end. Activation record contains the locals so that they are bound to fresh storage in each activation record.



Heap Storage allocation

Heap allocation is the most flexible allocation scheme. Allocation & de-allocation of memory can be done at any time & at any place depending upon the user's requirement. Heap allocation is used to allocate memory to the variables dynamically & when the variables are no more used then claim it back. Heap allocation supports the recursion process.



Heap Allocation.

Parameter Passing

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism.

Parameter Passing of two types:

Parameter
- Actual Parameter
- Formal Parameter

Actual Parameter

Actual Parameters are variables that accept data given by the calling process. These variables are listed in the called function's definition. They accept the calling process's data. The called functions definition contains a list of these variables.

Formal Parameter

Formal Parameters are variables whose values & functions are given to the called function. These variables are supplied as parameters in the function call. They must include the data type.

ex:

```
#include <bits/stdc++.h>
```

```
using namespace std;
// function declaration
int add(int x, int y)
int main()
{
   int x=50; int y=50;
   // Actual parameters
   int sum=add(x,y);
   cout<<"Actual parameters are:"<<x<<"and"<<y<<endl;
   cout<<"Result is:"<<sum;
}
// Function definition
// Formal parameter
int add(int a, int b)
{
   cout<<"Formal parameters are:"<<a<<"and"<<b<<endl);
   int c=a+b;
   return c;
}
```

Output:

Formal parameters are: 50 and 5
Actual Parameters are: 50 and 5
Result is: 55

Basic Terminology in Parameter Passing

Some basic terminologies in parameter passing are the following:

R-value

The value of an expression is its r-value. An r-value is a temporary object or literal without a permanent place in memory. They are basically placed on the right-hand side of the assignment operator.

ex:

```
a=1;
b=a+7;
c=b+12;
```

all variables including a, b, & c, have R-value.

L-value

The L-value of the expression refers to the location in memory where the expression is kept. Here the expression has a permanent memory location assigned. L-values are placed on the left side of the assignment operator.

Ex:

```
x = 10;
a = 2;
```

x and a are l-values & the R-values are 10 & 20.

Modes:

IN: Passes info from caller to the callee.

OUT: callee writes values in the caller.

INOUT: The caller tells the callee the value of the variable, which may be updated by the callee.

Methods for parameter passing.

* Call by values
* call by Reference
* call by copy Restore
* call by Name.

Call by values

The compiler adds to r-value of the actual parameters that were passed to the calling procedure to the called procedures activation record. Any modifications made to the formal parameters do not affect to actual parameters because they include the values given by the calling procedure.

Call by Reference

The formal & actual parameters in a call by reference relate to the same memory address. The activation record of the called function receives a copy of the L-value of the actual arguments. As a result, the address of actual parameters is passed to the called func

Call by copy Restore

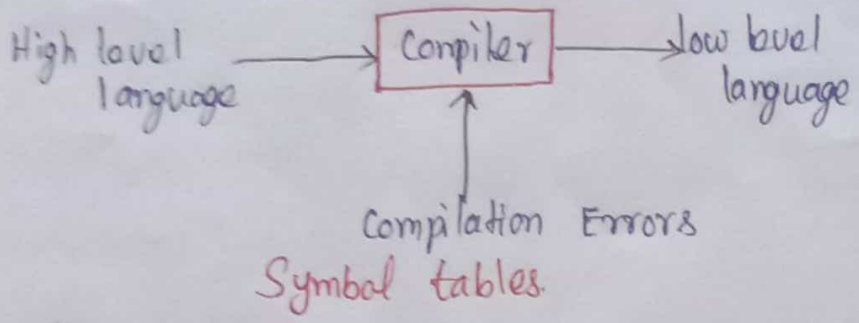
In contrast to call by reference, changes to actual parameters are made whe the called procedure completes. The actual parameter values are stored in the called procedures activation record during the function call. The manipulation of formal parameters has no immediate impact on the actual parameters.

Call by Name

A new form of preprocessor like argument parsing mechanism is offered by languages like Algol.

Symbol tables

A Symbol table is an important data structure used by compilers to manage identifiers in a program. An identifier is a name given to a variable, function, class or other programming construct that is used to represent some data or functionality in a program.



Symbol table entries

Each item in the symbol table is linked to a set of properties that assist the compiler at various stages.

Items stored in Symbol table

* labels in source languages
* Procedure & function names
* variable names & Constants
* Compiler generatored temporaries
* literal Constants & strings

Information used by the compiler from the Symbol table

* Datatype & name
* Offset in storage
* Declaring procedures
* For parameters, whether parameter passing by value or by reference.
* Number & type of arguments passed to function.
* Base address.

Basic Operations

Operations	Functions
allocate	To allocate a new empty Symbol table
free	To remove all entries & free storage of Symbol table
lookup	To Search for a name & return a pointer to its entry
insert	To insert a name in a symbol table & return a pointer to its entry
set_attribute	To associate an attribute with a given array
get_attribute	To get an attribute associated with a given array.

Information used by the compiler from Symbol table

Symbol identification

The Symbol table enables the compiler to apprehend & keep track of variables, functions, & different symbols in your application.

Data type information

It stores data about the data types of variables, like whether they're numbers, text or something else.

Variable Scope

It is information where in variables are declared & where they may be used within your code

Function details

For functions, it stores their names, parameters, & what they do.

Error Detection

The symbol desk aids in detecting mistakes for your code by way of making sure that variables & functions are used correctly & consistently.

Code Generation

It assists in generating machine code or executable programs primarily based on the statistics saved, making your code run effectively.

Implementation of Symbol table

The Symbol table can be implemented as an unordered list, which is simple to code but only works for small tables.

The following methods can be used to create a symbol table

List

A List is a collection of elements in which each element has a position or an index.

Linked list

A Linked list is a data structure in which each element contains a reference to the next element in the list.

Binary Search tree

A Binary search tree is a data structure in which each node has at most two children, & the values in the left subtree are less than the value in the node, & the values in the right subtree are greater than the value in the node.

Hash table

A hash table is a data structure that uses a hash function to map keys to values

Dynamic Storage Allocation

Dynamic Storage allocation is a process in which allocate or deallocate a block of memory during the run-time of a program. It can also be referred to as a procedure to use heap memory in which vary the size of a variable or data structure during the lifetime of a program using the library functions.

Functions of Dynamic Storage Allocation

1. malloc()
2. calloc()
3. realloc()
4. free()

} Present in <stdlib.h>

malloc() method

malloc() is a method in C which is used to allocate a memory block in the heap section of the memory of some specified size (in bytes) during the run-time of a C program.

Syntax of malloc()

```
(cast-data-type *) malloc (size-in-bytes);
```


Calloc() Method

Calloc() is a method in C which is also used to allocate memory blocks in the heap section, but it is generally used to allocate a sequence of memory blocks (contiguous memory) like an array of elements.

Syntax of calloc()

```
(cast-data-type *) calloc (num, size-in-bytes);
```

realloc() Method

realloc() is also a method in C that is generally used to reallocate a memory block, reallocate means to increase or decrease the size of a memory block previously allocated using malloc() or calloc() methods.

Syntax of realloc()

```
(cast-data-type *) realloc (ptr, new-size-in-bytes);
```

free() Method

free() as the name suggests is used to free or deallocate a memory block previously allocated using malloc() & calloc() functions during run-time of our program.

Syntax of free()

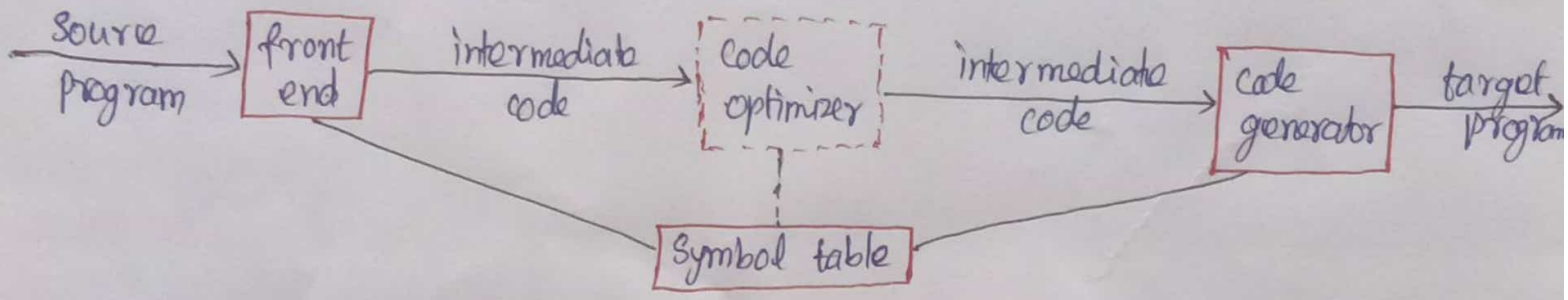
```
free (pointer);
```

Issues in the Design of a code generator

The code generator transforms the intermediate representation of source code into a machine-readable format.

Code generation Process

The final phase in the compiler model is the code generator. It carries an intermediate replica of the source program as input & delivers an equivalent target program as output.



Position of code Generator

Issues in the design of a code generator

Main issues in the design of a code generator are:

* input to the code generator
* target program
* memory management
* instruction selection
* register allocation
* evaluation order

Input to the code generator

Input to the code generator design issues in the code generator intermediate code created by the frontend & information from the symbol table that defines the run-time addresses of the data objects signified by the names in the intermediate representation are fed into the code generator. Intermediate codes may be represented mainly in quadruples, triples, indirect triples, Directed Acyclic Graph, postfix notation, syntax trees, etc.

Target program

The code generators output is the target program. The result could be:

Assembly language

It allows subprograms to be separately compiled.

Relocatable machine language

It simplifies the code generating process.

Absolute machine language

It can be stored in a specific position in memory & run immediately.

Memory management

The memory management design, the source program frontend and code generator map names address data items in run-time memory. It utilizes a symbol table. In a three-address statement, a name repers to the names symbol table entry.

ex:

j: goto i generates the following jump instruction:

if i < j, a backward jump instruction is generated with a target address equal to the quadruple i code location.

if i > j, a forward jump. The position of the first quadruple j machine instruction must be saved on a list for quadruple i. when i is processed, the machine locations for all instructions that forward hop to i are populated.

Instruction Selection

The instruction selection the design issues in the code generator program's efficiency will be improved by selecting the optimum instructions. It contains all of the instruction, which should be thorough & consistent.

The relevant three-address statements, ex would be translated into the following code sequence:

P: = Q + R

S: = P + T

```
MOV  Q, R0
ADD  R, R0
MOV  R0, P
MOV  P, P0
ADD  T, P0
MOV  R0, S
```

Register allocation

The register allocation design issues in the code generator can be accessed faster than memory. The instructions involving operands in the register are shorter & faster than those involved in memory operands.

The following sub-problems arise when use registers:

Register Allocation

Register allocation, select the set of variables that will reside in the regist

Register assignment

The register assignment, pick the register that contains a variable.

Certain machines require even-odd pairs of register for some operands & results.

ex

Consider the following division instruction of the form:

D x, y

where,

x is the dividend even register in even/odd register pair.

y is the divisor.

An old register is used to hold the quotient.

Evaluation order

The code generator determines the order in which the instructions are executed The target code efficiency is influenced by order of computations. Many computational orders will only require a few registers to store interim results however, choosing the best order is a completely challenging task in the general case.

Design of a Simple code generator

Code generation is the last scenario of compiler design. Code generation is the activity of generating assembly code / machine-readable code. Some significant properties of this activity are given as

* Absolute quality code
* Efficient use of resources
* Quick code

Ex

let us take the three address statements a := b + c.

It consists of the following sequence of codes.

MOV a, R0

ADD b, R0

Address Descriptor

An address descriptor is a container used for storing the location of the name of the current value so that it can be accessed during run time.

Register Descriptor

A register descriptor is a container depicting that all registers are empty in their initial states. It contains complete information about each track of the registe

Algorithm

The algorithm follows a queue of three address statements. The address statements are of form x : y op z simulates specific tasks.

The task description is given with the help of the following algorithm:

The initial step is to invoke a function getreg to capture the location L where the optimization of y op z is stored.

Manage the descriptive address of b to determine b'. If the value of b is present in memory & registers both, then the value of b' will be considered. If there is the absence of a value of b in L, then there is a need to generate the instruction MOV b', to copy a value of b in L.

A new instruction op c' is to be generated. Updation of address descriptor of a takes place to depict that a is stored in L. If a is already present in L, then update the descriptor & remove a from all other descriptors.

After reaching the stage where b or c have no further uses, alter the register description. Now, the register will not store the values of b, & c.

Generating Code

The given assignment statement w := (x - y) + (x - z) + (x - z) can be depicted into the queue of three address codes:

a := x - y @ b : x - z @ c := a + b @ w := b + c

Code sequence for the above example is given as:

Statement	code generation	Register descriptor	Address descriptor
a := x-y	MOV x, R0 SUB y, R0	R0 Contains a	a in R0
b := x-z	MOV x, R1 SUB z, R1	R0 Contains a R1 Contains b	a in R0 b in R1
c := a+b	ADD R1, R0	R0 Contains c R1 Contains b	c in R1 b in R1
w := b+c	ADD R1, R0 MOV R0, w	R0 Contains w	w in R0 w in R0 & memory

Optimal Code generation for Expressions

When a basic block consists of a single expression in which each operand appears only once, we can generate "optimal" code. We will label each node v of the expression tree with the smallest number of registers required to evaluate v's subtree without using temporary variables.

1. Ershov Numbers

The nodes of an expression tree a number that tells how many registers are needed to evaluate that node without storing any temporaries. These numbers are sometimes called Ershov numbers, after A. Ershov, who used a similar scheme for machines with a single arithmetic register.

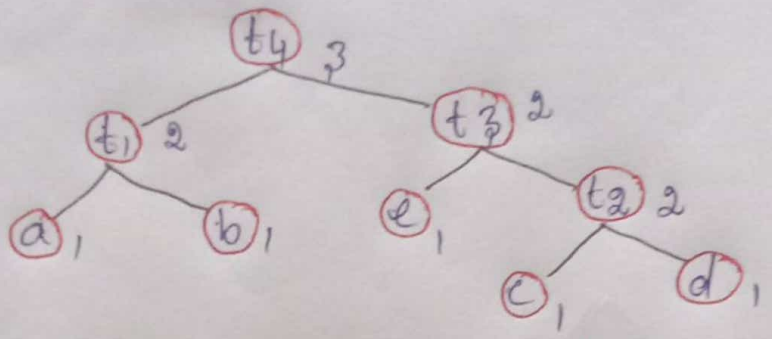
The rules are

* label any leaf 1.
* The label of an interior node with one child is the label of its child.
* The label of an interior node with two children is:

The larger of the labels of its children, if those labels are different one plus the label of its children if the labels are the same. (89)

An expression tree (with operators omitted) that might be the tree for expression (a-b) + e x (c+d) or the three-address code:

$t_1 = a - b$
$t_2 = c + d$
$t_3 = e * t_2$
$t_4 = t_1 + t_3$



A tree labeled with Ershov numbers

2. Generating code from labeled expression trees

It can be proved that, in our machine model, where all operands must be in registers, & registers can be used by both an operand & the result of an operation, the label of a node is the fewest registers with which the expression can be evaluated using no stores of temporary results.

Algorithm

Input: A labeled tree with each operand appearing once (that is, no common subexpressions).

Output: An optimal sequence of machine instructions to evaluate the root into a register.

Method: The following is a recursive algorithm to generate the machine code. The steps below are applied, starting at the root of the tree. If the algorithm is applied to a node with label k, then only k registers will be used. However, there is a "base" $b > 1$ for the registers used so that the actual registers used are $R_b, R_{b+1}, \ldots R_{b+k-1}$. The result always appears in R_{b+k-1}.

To generate machine code for an interior node with label k & two children with equal labels (which must be k-1) do the following:

Recursively generate code for the right child, using base b+1. The result of the right child appears in register R_{b+k}.

Recursively generate code for the left child, using base b; the result

appears in Rb+k-i.

Generate the instruction op Rb+k, Rb+k-i, Rb+k, where op is the appropriate operation for the interior node in question.

3. Evaluating expressions with an insufficient supply of Registers.

When there are fewer registers available than the label of the root of the tree, we cannot apply Algorithm directly. We need to introduce some store instructions that spill values of subtrees into memory. Here is the modified algorithm that takes into account a limitation on the number of register.

Algorithm

Input: A labeled tree with each operand appearing once (that is no common subexpressions) & a number of registers $r \geq 2$.

Output: An optimal sequence of machine instructions to evaluate the root into a register, using no more than r registers, which we assume are $R_1, R_2 \cdots R_r$

Method: Apply the following recursive algorithm, starting at the root of the tree, with base $b = 1$. The

The modifications to the basic algorithm are as follows:

1. Node N has at least one child with label r or greater. Pick the larger child (or either if their labels are the same) to be the "big" child & let the other child be the "little" child.

2. Recursively generate code for the big child, using base $b = 1$. The result of this evaluation will appear in register R_r.

3. Generate the machine instruction ST t-k, Rr, where is a temporary variable used for temporary results used to help evaluate nodes with label k.

4. Generate code for the little child as follows. If the little child has label r or greater, pick base $b = 1$. If the label of the little child is $j \geq r$, then pick $b = r - j$. Then recursively apply this algorithm to the little child, the result appears in Rr.

5. Generate the instruction LD R_r, t-k.

6. If the big child is the right child of N, then generate the instruction op R_r, R_r, R_{r-1}. If the big child is the left child, generate op R_r, R_{r-1}, R_r.

2. The following code for the node labeled i:

```
LD  R3, d
LD  R2, e
ADD R3, R2, R3
```

next, we generate code for the left child of the right child of the root. this node is the leaf labeled e. Since b-2, the proper instruction is

```
LD  R2, e
```

Now, we can complete the code for the right child of the root by adding the instruction

```
MUL R3, R2, R3.
```

The complete sequence of instructions is:

```
LD  R3, d
LD  R2, c
ADD R3, R2, R3
LD  R2, e
MUL R3, R2, R3
LD  R2, b
LD  R1, a
SUB R2, R1, R2
ADD R3, R2, R3
```

Optimal tree-register code for the tree of labeled with Ershov

3. Generate in optimal tree-register code for the tree of labeled with Ershov numbers, with registers R_1 & R_2 in place of R_2 & R_3.

This code is:

```
LD  R2, d
LD  R1, c
ADD R2, R1, R2
LD  R1, e
MUL R2, R1, R2
```

to generate the instruction

```
ST  t3, R2
```

Next, the left child of the root is loaded. Again the number of registers is sufficient for this child, & the code is


```
LD  R2, b
LD  R1, a
SUB R2, R1, R2
```

Finally, we reload the temporary that holds the right child of the root with the instruction

```
LD  R1, t3
```

& execute the operation at the root of the tree with the instruction

```
ADD R2, R2, R1
```

The complete sequence of instructions is

```
LD  R2  d
LD  R1  c
ADD R2  R1, R2
LD  R1  e
MUL R2  R1, R2
ST  t3  R2
LD  R2  b
LD  R1  a
SUB R2  R1, R2
LD  R1  t3
ADD R2  R2, R1
```

Optimal three-register code for the tree of labeled with Ershov numbers using only two registers.

Unit - V

Principal Sources of optimization

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block. otherwise it is called global. Many transformations can be performed at both the local & global levels. local transformations are usually performed first.

Function-Preserving transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes.

Function Preserving transformations examples:

* Common Sub expression elimination
* copy Propagation
* Dead-code elimination
* Constant folding

Common Sub expression elimination

An occurence of an expression E is called a common Sub-expression if E was previously computed, & the values of variables in E have not changed since the previous computation.

ex:

Initial code

```
t1 := 4*i
t2 := a[t1]
t3 := 4*j
t4 := 4*i
t5 := n
t6 := b[t4]+t5
```

optimized code

```
t1 := 4*i
t2 := a[t1]
t3 := 4*j
t5 := n
t6 := b[t1]+t5
```

Copy Propagation

Assignments of the form f := g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f := g. Copy propagation means use of one variable instead of another.

ex

Initial code

```
x = Pi;
A = x * r * r;
```

Optimized code

```
A = Pi * r * r;
```

Dead-Code Eliminations

A variable is live at a point in a program if its value can be used subsequently. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

ex:

```
i=0;
if (i=1)
{
   a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

Constant Folding

Deducing at compile time that the value of an expression is a constant & using the constant instead is known as constant folding.

ex:

Consider an expression: a = b op c & the values b & c are constants, then the value of a can be computed at compile time.

```
#define k 5
x = 2 * k
y = k + 5
```

o/p

```
x = 6
y = 10
```

loop Optimizations

The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization.

* Code motion
* Induction-variable elimination
* Reduction in strength.

Code motion

An important modification that decrease the amount of code in a loop is Code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) & places the expression before the loop.

ex: Initial code

```
i = 10
x = 15
j = x + 10
while (i < (x + 10))
```

Optimized code

```
i = 10
x = 15
while (i < j)
```

Induction variables

loops are usually processed inside out. For example consider the loop around B_3. That the values of j & t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because $4*j$ is assigned to t_4. Such identifiers are called induction variables.

ex

As the relationship $t_4 := 4*j$ surely holds after such an assignment to t_4. & t_4 is not changed elsewhere in the inner loop around B_3, it follows that just after the statement $j := j-1$ the relationship $t_4 := 4*j - 4$ must hold. We may therefore replace the assignment $t_4 := 4*j$ by $t_4 := t_4 - 4$.

```
j := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3
```

Reduction in Strength

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others & can often be used as special cases of more expensive operators.

ex `j = j * t4 ....`

4, 8, 12, 16, 20 ....

Initial code

```
for (i=1; i<10; i++)
{
    int j = 4 * i;
    print j
}
```

Optimized code

```
for (int j=1; j<10; j++)
{
    int j = j + 4;
    print j;
}
```

Peep-hole optimization

A statement by statement code generations strategy often produces target code that contains redundant instructions & suboptimal constructs. The quality of such target code can be improved by applying optimizing transformations to the target Program. The peephole is a small, moving window on the target program. The code in the peephole need not be contiguous, although some implementations do require this.

Objectives of Peephole Optimization

- To improve performance
- To reduce memory footprint
- To reduce code size

Increasing code speed

Peephole optimization in compiler design seeks to improve the execution speed of generated code by removing redundant instructions or unnecessary instructions.

Reduced code size

Peephole optimization in compiler design seeks to reduce generated code size by replacing the long sequence of instructions with shorter ones.

Getting rid of dead code

Such as unreachable code, redundant assignments or constant expressions that have no effect on the output of the program.

Simplifying code

To make generated code more understandable & manageable by removing unnecessary complexities.

Working of Peephole optimization

There are mainly four steps in peephole optimization. The steps are as follows:

* Identification
* Optimization
* Analysis
* Iteration.

Identification

Identify the code section where need the peephole optimization. Peephole is an instruction with a fixed window size, so the window size depends on the specific optimization being performed. The compiler helps to define the instructions within the window.

Optimization

The rules of optimizations pre-defined in the peephole. The compiler will search for the specific pattern of instructions in the window. There can be many types of patterns, such as insufficient code, series of loads & stores or complex patterns like branches.

Analysis

After the pattern is identified, the compiler will make the changes in the instructions The compiler will cross-check the code to determine whether the changes improved the code. It will check the improvement based on size, speed & memory usage.

Iteration

The above steps will go on a loop by finding the peephole repeatedly until no more optimization is left in the code. The compiler will go to each instruction one at a time & make the changes & reanalyse it for the best result.

Peephole Optimization Techniques

There are various peephole optimization techniques.

Redundant Load & Store Elimination.

Redundant load & store elimination is also one of the peephole optimization works by finding code that performs the same memory access many times & removes the redundant accesses.

ex

Initial code	Optimized code
r2 = r1 + 5;	r2 = r1 + 5;
i = r2;	i = r2;
r3 = i;	r4 = r2 * 3;
r4 = r3 * 3;	

Flow of Control Optimizations

The unnecessary jumps can be eliminated in either the intermediate code or

the target code by the following types of peephole optimizations.

ex:

Initial code	optimized code
goto L_1	goto L_3
L_1 : goto L_2	L_1 : goto L_3
L_2 : goto L_3	L_3 : goto L_3
L_3 :	L_3 : ....

Algebraic Simplifications

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them.

ex:

$x^2 = x * x$ $x = x + 0$

$2x = x + x$ $x = x * 1$ are often produced by straightforward intermediate code generation algorithms, & they can be eliminated easily through peephole optimization.

Use of machine idioms

The target machine may have hardware instructions to implement certain specific operations efficiently. Some machine have auto-increment & decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing.

ex:

$x = x + 1 \rightarrow x++$

$x = x - 1 \rightarrow x--$

Unreachable code elimination

Peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions.

ex: Initial code

```
int
{
...
return
a = a+b; }
```

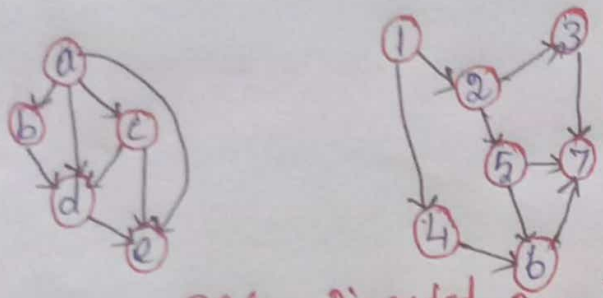
optimized code

```
int
{
...
return
}
```

Directed Acyclic Graph

The directed acyclic graph is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks & to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation.

ex



DAG - Directed Acyclic Graph

Characteristics of DAG

The following are some characteristics of DAG.

* DAG is a type of data structure used to represent the structure of basic blocks
* Its main aim is to perform the transformation on basic blocks.
* The leaf nodes of the DAG represent a unique identifier that can be a variable or a constant.
* The non-leaf nodes represent an operator symbol.
* The nodes are also given a string of identifiers to use as labels for the computed value.
* Transitive closure & transitive reduction are defined differently in DAG.
* DAG has defined topological ordering.

Algorithm for Construction of DAG

For constructing a DAG, the input & output are as follows:

input: The input will contain a basic block.

output: The output will contain the following information.

* Each node of the graph represents a label.
    * If the node is a leaf node, the label represents an identifier.
    * If the node is a non-leaf node, the label represents an operator.
* Each node contains a list of attached identifiers to hold the computed value.

There are three possible scenarios in which can construct a DAG.

* Case 1: $x = y$ op z

where x, y, & z are operands & op is an operator.

* Case 2: $x =$ op y

where x & y are operands & op is an operator.

* Case 3: $x = y$

where x & y are operands

The steps to draw a DAG following the above three cases.

Steps:

To draw a DAG, follow these three steps.

Step 1:

If, in any of the three cases, the y operand is not defined, then create a node (y).

If, in case 1, the z operand is not defined, then create a node (z).

Step 2:

For case 1, create a node (op) with node (y) as its left child & node (z) as its right child. let the name of this node be n.

For case 2, check whether there is a node (op) with one child node as node (y). If there is no such node, then create a node.

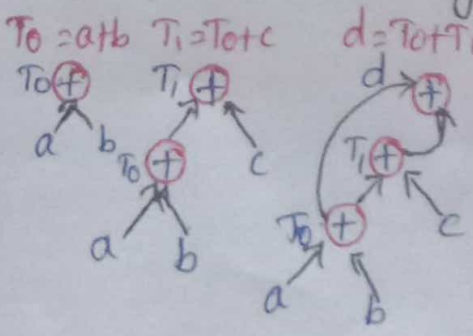
For case 3, node n will be node (y).

Step 3:

For a node (x), delete x from the list of identifiers. Add x to the list of attached identifiers list found in step 2. At last, set node (x) to n.

Consider an example of drawing a DAG.

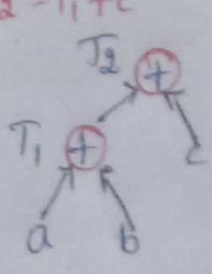
1. $T_0 = a + b$
   $T_1 = T_0 + c$
   $d = T_0 + T_1$

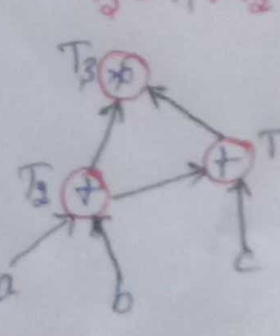


Final DAG

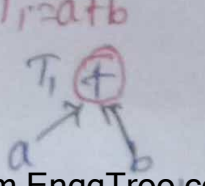
DAG - Directed Acyclic Graph

2. $T_1 = a + b$
   $T_2 = T_1 + c$
   $T_3 = T_1 * T_2$





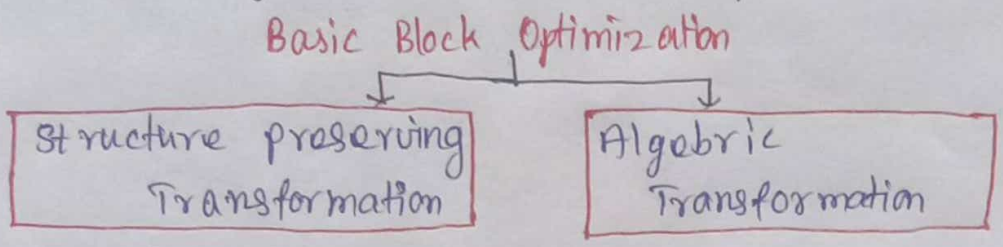
Final Directed Acyclic Graph



Optimization of Basic Blocks

Optimization of applied to the basic blocks after the intermediate code generation phase of the compiler. Optimization is the process of transforming a program that improves the code by consuming fewer resources & delivering high speed. In optimization, high-level codes are replaced by their equivalent efficient low-level codes.

There are two types of basic block optimizations.



Structure Preserving Transformations

The structure preserving transformation on basic blocks includes:

* Dead code Elimination
* Common subexpression elimination
* Renaming of Temporary variables
* Interchange of two independent adjacent statements.

Dead code elimination

Dead code is defined as that part of the code that never executes during the program execution. The code which is never executed during the program (dead code) takes time so, for optimization & speed, it is eliminated from the code. Eliminating the dead code increases the speed of the program as the compiler does not have to translate the dead code.

ex: Initial code

```
int main()
{
  x = 2
  if (x > 2)
    cout << "code";
  else
    cout << "optimization";
  return 0;
}
```

optimized code.

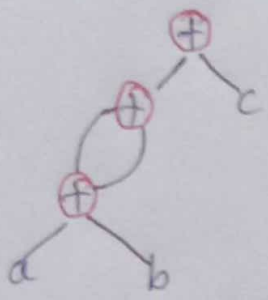
```
int main()
{
  x = 2;
  cout << "optimization";
  return 0;
}
```

Common Subexpression Elimination

The sub-expression which are common are used frequently are calculated only once & reused when needed. DAG is used to eliminate common subexpression

ex:

$x = (a+b) + (a+b) + c$



DAG Representation.

Renaming of temporary variables

Statements containing instances of a temporary variable can be changed to instances of a new temporary variable without changing the basic block value

ex:

Statement $t = a+b$ can be changed to $x = a+b$ where t is a temporary variable & x is a new temporary variable without changing the value of the basic block.

Interchange of two independent adjacent statements:

If a block has two adjacent statements which are independent can be interchanged without affecting the basic block value.

ex:

$t1 = a+b$

$t2 = c+d$.

These two independent statements of a block can be interchanged without affecting the value of the block.

Algebraic Transformation

Countless algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Some of the algebraic transformation on basic blocks includes:

constant folding

copy propagation

strength reduction.

Constant Folding

Solve the constant terms which are continuous so that compiler does not need to solve this expression.

ex:

$x = 2 * 3 + y \qquad x = 6 + y$

Copy Propagation

It is two types, variable propagation & constant propagation.

Variable propagation

$x = y \Rightarrow z = y + 2 \qquad z = x + 2$

Constant propagation

$x = 3 \Rightarrow z = 3 + a \qquad z = x + a$

Strength Reduction

Replace expensive statement or instruction with cheaper ones.

$x = 2 * y$ (costly) $\Rightarrow x = y + y$ (cheaper)

$x = 2 * y$ (costly) $\Rightarrow x = y << 1$ (cheaper)

Data Flow Analysis

Code optimization is a program transformation technique that tries to improve the intermediate code to consume fewer resources such as CPU, memory, etc., resulting in faster machine code.

There are two types of code optimization techniques.

loop optimization

This code optimization applies to a small block of statements.

ex:

local optimization are variable or constant propagation & common subexpression elimination.

Global optimization

This code optimization applies to large program segments like functions, loops, procedures, etc, ...

ex:

Global optimization is data flow analysis.

The compiler performs code optimization efficiently by collecting all the information about a program & distributing it to each block of its control flow graph. This process is known as data flow analysis.

Basic Technologies of data flow Analysis

Below are some basic terminologies related to data flow analysis.

Definition Point

A definition point is a point in a program that defines a data item.

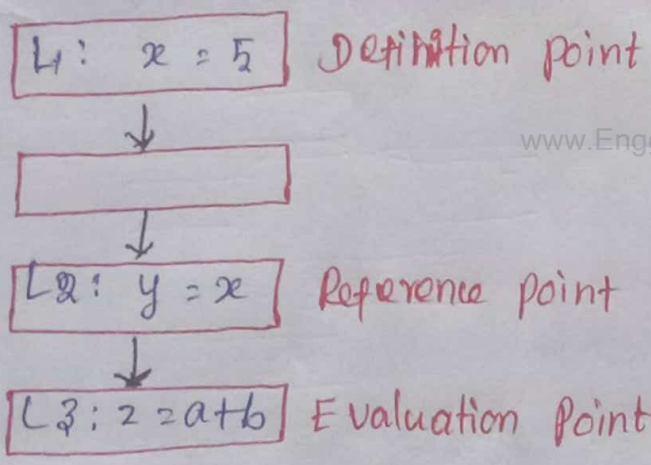
Reference Point

A reference points is a point in a program that contains a reference to a data item.

Evaluation Point

An evaluation point is a point in a program that contains an expression to be evaluated.

ex:



Data Flow analysis equation

The data flow analysis equation is used to collect information about a program block. The following is the data flow analysis equation for a statement S-

$$Out[S] = gen[S] \cup In[S] - kill[S]$$

where

* Out [S] is the information at the end of the statement S.
* gen [S] is the information generated by the statement S.
* In [S] is the information at the beginning of the statement S.
* kill [S] is the information killed or removed by the statement S.

The main aim of the data flow analysis is to find a set of constraints on the In [S]'s & out [S]'s for the statement S.

The constraints include two types of constraints:

Transfer function & control flow constraint

Transfer function

The semantics of the statement are the constraints for the data flow values before & after a statement.

ex:

Consider two statements $x = y$ & $z = x$. Both these statements are executed

There are two types of transfer functions:

Forward propagation & Backward propagation

Forward Propagation

The transfer function is represented by Fs for any statement S. This statement transfer function accepts the data flow values before the statement & outputs the new data flow value after the statement. Thus the new data flow or the output after the statement will be $out[S] = FS[In[s]]$.

Backward Propagation

The backward propagation is the converse of the forward propagation. After the statement, a data flow value is converted to a new data flow value before the statement using this transfer function. Thus the new data flow or the output will be $In[S] = FS(Out[S])$.

Control Flow Constraints

The second set of constraints comes from the control flow. The control flow value of S_i will be equal to the control flow values into S_{i+1} if block B contains statements $S_1, S_2, \ldots S_n$.

$$IN[S_{i+1}] = OUT[S_i], \text{ for all } i = 1, 2, \ldots, n-1.$$

Data flow Properties

Some properties of the data flow analysis are:-

* Available expression
* Reaching Definition
* Live variable
* Busy expression

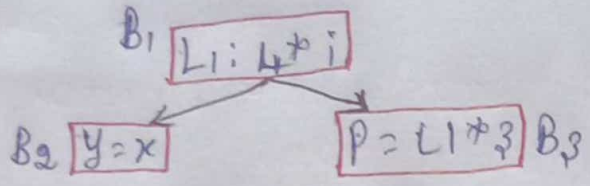
Available expression

An expression a+b is said to be available at a program point x if none of

its operands gets modified before their use. It is used to eliminate common subexpressions.

An expression is available at its evaluation point.

ex:

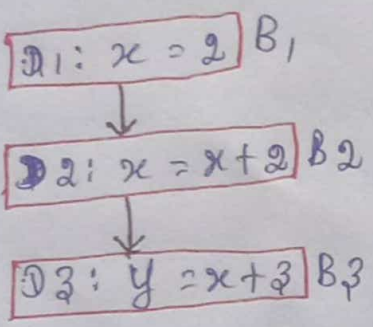


The expression L1: 4* i is an available expression since this expression is available for block B2 & B3, & no operand is getting modified.

Reaching Definition

A definition D is reaching a point x if D is not killed or redefined before that point. It is generally used in variable or constant propagation.

ex:

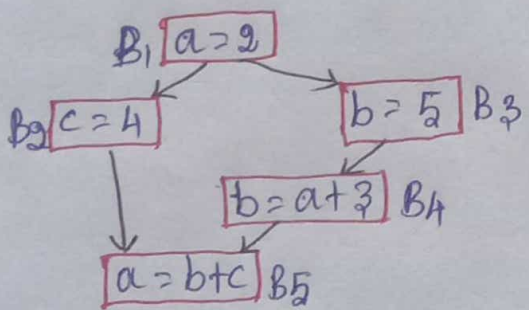


D1 is a reaching definition for block B2 since the value of x is not changed (it is two only) but D1 is not a reaching definition for block B3 because the value of x is changed to x+2. This means D1 is killed or redefined by D2.

live variable

A variable x is said to be live at a point p if the variable's value is not killed or redefined by some block. If the variable is killed or redefined, it is said to be dead. It is generally used in register allocation & dead code elimination.

ex:



The variable a is live at blocks B1, B2, B3, & B4 but is killed at block B5 since its value is changed from 2 to b+c. Similarly, variable b is live at block B3 but is killed at block B4.

Busy Expression

An expression is said to be busy along a path if its evaluation occurs along that path, but none of its operand definitions appears before it. It is used for performing code movement optimization.