PIE Tech

POLLACHI INSTITUTE OF ENGINEERING AND TECHNOLOGY

(Approved by AICTE and Affiliated to Anna University)

sky is the limit

Department of Computer Science and Engineering

Regulation 2021

III Year – V Semester

CS3501 – Compiler Design

CS3501 - COMPILER DESIGN

2021-Regulation Anna University

CS3501 COMPILER DESIGN LPTC

3 0 24

OBJECTIVES:

- To learn the various phases of compiler.
- To learn the various parsing techniques.
- To understand intermediate code generation and run-time environment.
- To learn to implement front-end of the compiler.
- To learn to implement code generator.
- To learn to implement code optimization.

UNIT I INTRODUCTION TO COMPILERS & LEXICAL ANALYSIS 8

Introduction- Translators- Compilation and 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 11

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 and Recovery in Syntax Analyzer-YACC tool - Design of a syntax Analyzer for a Sample Language

UNIT III SYNTAX DIRECTED TRANSLATION & INTERMEDIATE CODE GENERATION 9

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 and Declarations, Translation of Expressions, Type Checking, Back patching.

UNIT IV RUN-TIME ENVIRONMENT AND CODE GENERATION 9

Runtime Environments – source language issues – Storage organization – Storage Allocation Strategies: Static, Stack and Heap allocation - Parameter Passing-Symbol Tables - Dynamic Storage Allocation - Issues in the Design of a code generator – Basic Blocks and Flow graphs - Design of a simple Code Generator - Optimal Code Generation for Expressions– Dynamic Programming Code Generation.

UNIT V CODE OPTIMIZATION 8

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.

45 PERIODS

LIST OF EXPERIMENTS:

1. Using the LEX tool, Develop a lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc.). Create a symbol table, while recognizing identifiers.
2. Implement a Lexical Analyzer using LEX Tool
3. Generate YACC specification for a few syntactic categories.
   a. Program to recognize a valid arithmetic expression that uses operator +, -, * and /.
   b. Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.
   c. Program to recognize a valid control structures syntax of C language (For loop, while loop, if-else, if-else-if, switch-case, etc.).
   d. Implementation of calculator using LEX and YACC
4. Generate three address code for a simple program using LEX and YACC.
5. Implement type checking using Lex and Yacc.
6. Implement simple code optimization techniques (Constant folding, Strength reduction and Algebraic transformation)
7. Implement back-end of the compiler for which the three address code is given as input and the 8086 assembly language code is produced as output.

OUTCOMES: On Completion of the course, the students should be able to:

- Understand the different phases of compiler.
- Design a lexical analyzer for a sample language.
- Apply different parsing algorithms to develop the parsers for a given grammar.
- Understand syntax-directed translation and run-time environment.
- Learn to implement code optimization techniques and a simple code generator.
- Design and implement a scanner and a parser using LEX and YACC tools.

TEXT BOOK:

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques and Tools‖, Second Edition, Pearson Education, 2009.

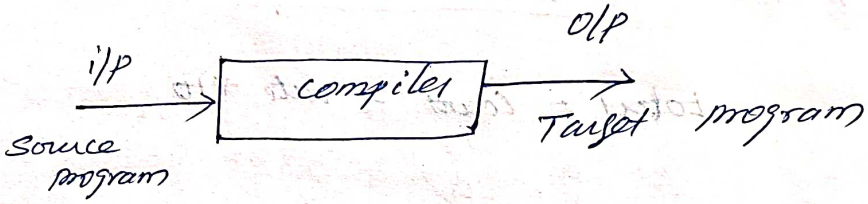
REFERENCES:

1. Randy Allen, Ken Kennedy, Optimizing Compilers for Modern Architectures: A Dependence based Approach, Morgan Kaufmann Publishers, 2002.
2. Steven S. Muchnick, Advanced Compiler Design and Implementation‖, Morgan Kaufmann Publishers - Elsevier Science, India, Indian Reprint 2003.
3. Keith D Cooper and Linda Torczon, Engineering a Compiler‖, Morgan Kaufmann Publishers Elsevier Science, 2004.
4. V. Raghavan, Principles of Compiler Design‖, Tata McGraw Hill Education Publishers, 2010.
5. Allen I. Holub, Compiler Design in C‖, Prentice-Hall Software Series, 1993.

UNIT-I INTRODUCTION TO COMPILERS

Structure of a Compiler

Compiler is a program which takes one language (source program) as input and translates it into an equivalent another language (target program).



⇒ during this process of translation if some errors are encountered then compiler displays them as error message.

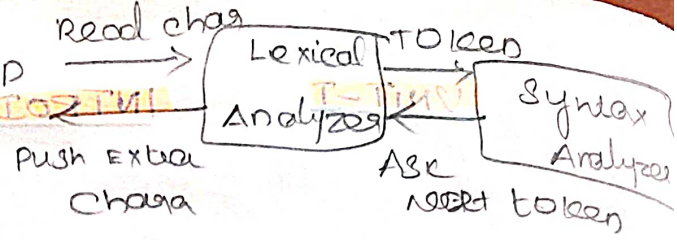
⇒ Compiler takes a source program as higher level language such as C, PASCAL, converts it into low level language or machine level language.

Analysis - synthesis model

- Analysis part the source program is read and broken down into consistuent pieces.

The syntax and the meaning of the source string is determined

⇒ synthesis part this intermediate form of the source language is taken and converted into an equivalent target program.



Phases of Compiler

1. Lexical Analysis

- The lexical Analysis is also called scanning. It converts the high level pgm into a seq. of Tokens
- This phase of compilation is which the complete source code is scanned and your source program is broken up into group of strings called token.

$$\text{total} = \text{count} + \text{rate} * 10$$

Id op Id op Id op const

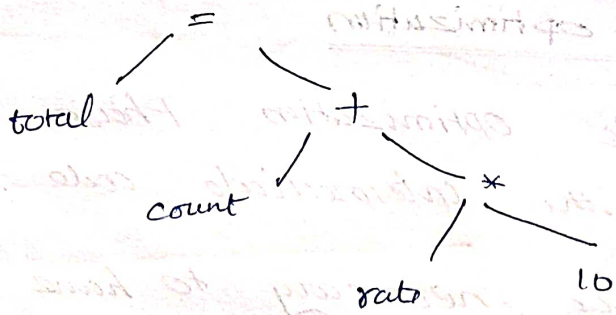
1. The identifier total 2. assignment symbol
3. identifier count 4. The plus sign
5. The identifier rate 6. The multiplication sign
7. The constant number 10.

token → I/P

2. Syntax Analysis (parser) (syntax tree)

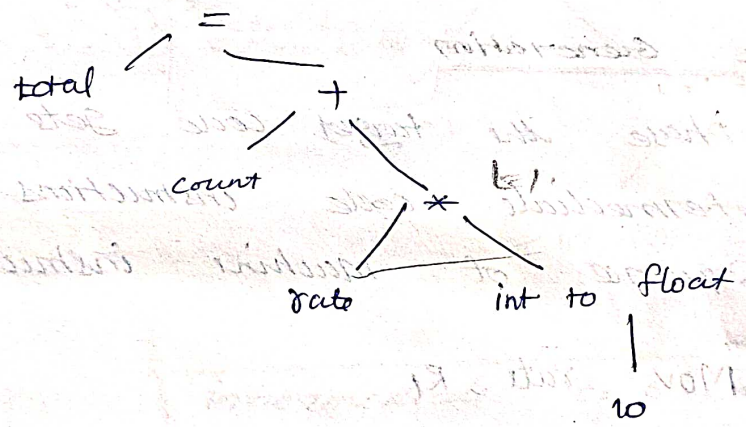
- The syntax analysis is also called parsing.
- The syntax analysis determines the structure of the source string by grouping the tokens together.
- expression $\text{total} = \text{count} + \text{rate} * 10$

$$\langle id,1 \rangle = \langle id,2 \rangle \langle + \rangle \langle id,3 \rangle \langle * \rangle \langle 10 \rangle$$



3. Semantic Analysis (Type checking)

- The Semantic Analysis determines the meaning of the source string.



4. Intermediate code Generation

- It is a kind of code which is easy to generate and this code can be easily converted to target code.
- This code is in variety of forms such as three address code, quadruple, triple.

$t_1 := \text{int to float}(10)$

$t_2 := \text{rate} \times t_1$

$t_3 := \text{count} + t_2$

$\text{total} := t_3$

5. code optimization

- The code optimization phase attempts to improve the intermediate code.
- This is necessary to have a faster execution code or less consumption of memory.
- Thus by optimizing the code the overall running time of the target program can be improved.

$t_1 := rate * 60$

6. code Generation

This phase the target code gets generated. The intermediate code instructions are translated into sequence of machine instructions.

```
MOV  rate , R1
MUL  #10.0 , R1
MOV  count , R2
ADD  R2 , R1
MOV  R1 , total
```

Write down the output of each phases for the expression

$c = a + b * 5$

↓

Lexical analyzer

↓

<id,1><=><id,2><+><id,3><*><5>

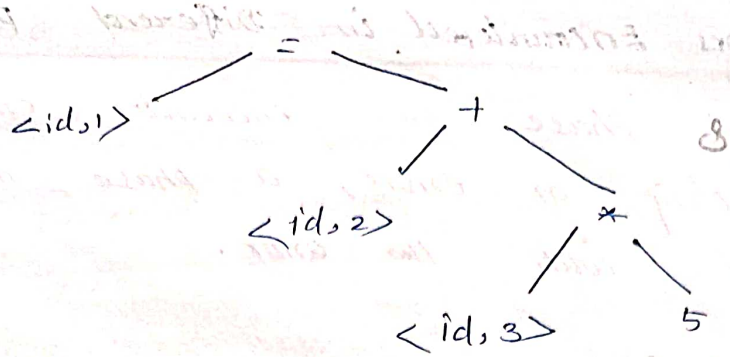
↓

Syntax analyzer

↓

a	...
b	...
c	...
:	...

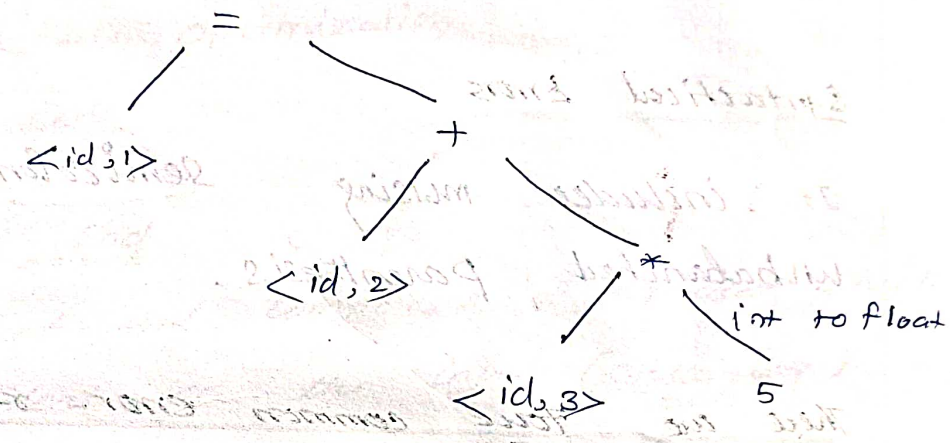
Symbol Table



Syntax Tree

Semantic analyzer

Semantic Tree



intermediate code generator

$t_1 = \text{int of float}(5)$

$t_2 = id_3 * t_1$

$t_3 = id_2 + t_2$

$id_1 = t_3$

Intermediate code

Code Optimizer

$t_1 = id_3 * (5)$

$id_1 = id_2 * t_1$

optimized code

Code generator

```
LDF   R2, id3
MULF  R2, # 5.0
LDF   R1, id2
ADDF  R1, R2
STF   id1, R1
```

Machine Code

(6)

Errors Encountered in Different Phases

Each phase can encounter errors. After detecting an errors, a phase must somehow deal with the error.

Lexical Errors

It includes incorrect or misspelled name of some identifier.

Syntactical Errors

It includes missing semicolon or unbalanced parenthesis.

There are four common error-recovery strategies

- Panic mode
- Statement level
- Error productions
- Global correction

Semantical Errors

- These errors are a result of incompatible value assignment.

- Type mismatch
- Undeclared variable
- Reserved identifier misuse
- Multiple declaration of variable in a scope

GROUPING OF PHASES

Front end

- → Lexical analysis
- → syntax analysis
- → Semantic analysis
- → intermediate code generation

Back end

- → code optimization
- → code generation

Front End

→ Front end comprises of phases which are dependent on the input (source lang) and independent on the target machine,

Lexical an

Passes

→ The phases of compiler can be implemented in a single pass by making the primary action.

* reading of input file
* writing to the output file

→ Several phases of compiler are grouped into one pass in such a way that the operations in each and every phases are incorporated during the pass.

Reducing the Number of Passes

→ Minimizing the number of passes improves the time efficiency as reading from and writing to intermediate files can be reduced.

Compiler Construction Tools

1. Parser generators
2. scanner generators
3. Syntax-directed translation engines
4. Automatic code generators
5. Data-flow analysis engines
6. Compiler construction toolkits

Parser generators

i/p — Grammatical description of a programming language

o/p — Syntax analyzers

Parser generator takes the grammatical description of a programming language and produces a Syntax analyzer.

Scanner Generators

i/p : Regular expression description of the tokens of a language.

o/p : Lexical analyzers.

Scanner generator generates lexical analyzers from a regular expression description of its tokens of a language.

Syntax-directed Translation Engines

i/p : Parse tree.

o/p : Intermediate code

Syntax directed translation engines produces collections of routines that walk a parse tree and generates intermediate code.

Automatic code Generators

i/p : Intermediate language

o/p : Machine language

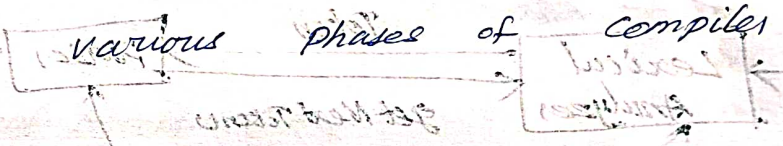
Code generator takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for a target machine.

Data Flow Analysis Engines

- Data Flow analysis engines gathers the information.
- Data Flow analysis is a key part of code optimization.

Compiler construction Toolkits

The toolkits provide integrated set of routines for various phases of compiler.



LEXICAL ANALYSIS

- Lexical analysis is the process of converting a sequence of characters from source program into a sequence of tokens.
- It consists of two stages.
  - → Scanning
  - → Tokenization

Token, pattern and Lexeme

Token

Token is a valid sequence of characters which are given by lexeme.

- → keywords (int, float, const) → Numbers
- → constant → operators (+, -, =)
- → identifiers → punctuation symbols (;)

Pattern

Pattern describes a rule that must be matched by sequence of characters to form a token.

It can be defined by regular expressions or Grammar rules.

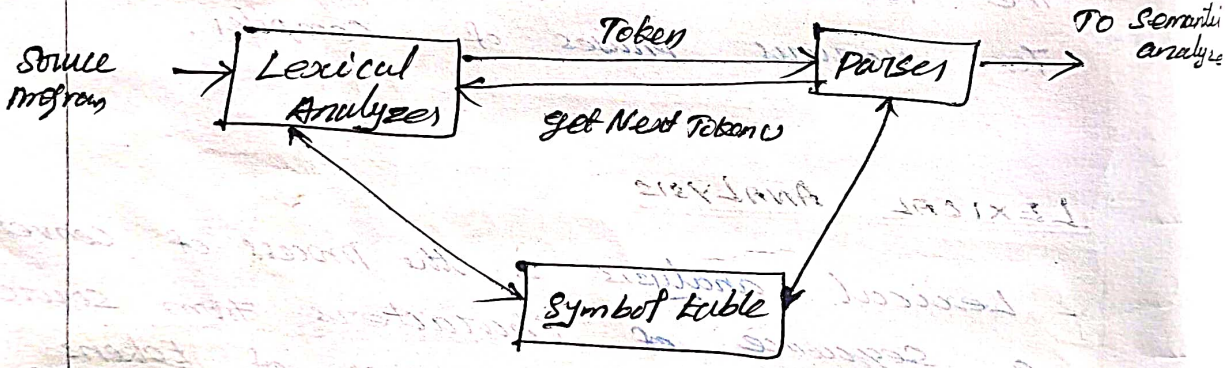
Lexeme

Lexeme is a sequence of characters that matches the pattern for a token.

identifier id id const

$C = a + b * 5$.

Role of Lexical Analyzer



Interaction b/w lexical analyzer and parser

→ Reads the source program, scans the input characters, group them into lexemes and produce the token as output.

→ Enters the identified token into the symbol table.

→ Strips out white spaces and comments from source program

→ Correlates error messages with the source program

→ Expands the macros if it is found in the source program.

Need of Lexical Analyzer

- Simplicity of design of compiler
- Compiler efficiency is improved
- Compiler portability is enhanced.

Issues in Lexical Analysis

- **Lookahead**

It is required to decide when one token will end and the next token will begin.

- **Ambiguities** (Rel°)

* The longest match is preferred
* Among rules which matched the same number of characters, the rule given first is preferred.

Lexical Errors (Refer)

- A character sequence that cannot be scanned into any valid token is a lexical error.
- Lexical errors are uncommon, but they still must be handled by a scanner.
- Misspelling of identifiers, keywords or operators are considered as lexical errors.

Error Recovery Schemes

- Panic mode recovery
- Local correction
- Global correction.

(12)

I/P → Buffer → Lexical Analyzer

Input Buffering → is a location that holds all input before it continues to op

- To ① ensure that a right lexeme is found or more characters have to be looked up beyond the next lexeme.

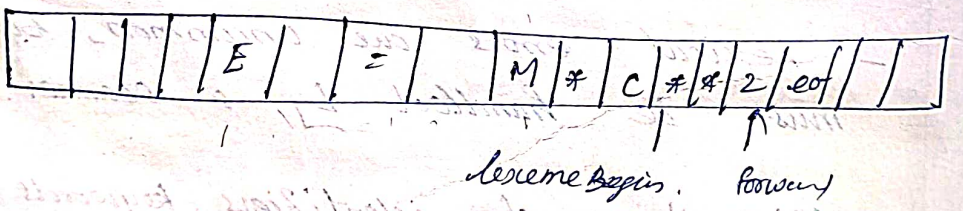
- A two-buffer scheme is introduced to handle large lookaheads safely.

- Techniques for speeding up the process of lexical analyzer such as the use of sentinels to mark the buffer end have been adopted

Buffer pairs

Because of large amount of time consumption in moving characters,

Specialized buffering techniques have been developed to reduce the amount of overhead required to process and input characters.



Buffer pairs.

Scheme

- consists of two buffers, each consists of N-character size which are reloaded alternately
- N - Number of characters on one disk block
- N characters are read from the input file to the buffer using one system read command
- eof is inserted at the end if the number of characters is less than N.

Pointers

Two pointers

- lexemeBegin
- forward

* **lexemeBegin**

Points to the beginning of the current lexeme which is yet to be found

* **Forward**

scans ahead until a match for a pattern is found.

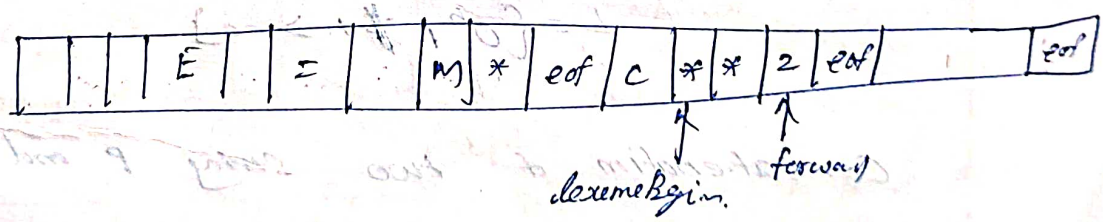
Sentinels → Special character - eof.

Test 1 : For end of buffer

Test 2 : To determine what character is read.

→ The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end

→ The sentinel is a special character that cannot be part of the source program.



Advantages

- Most of the times, it performs only one test to see whether forward pointer point to an eof
- More tests are performed only when it reaches the end of the buffers half or eof.

(1X)

Specification of Token – Reg. EXP.

- Regular expressions are a notation to represent lexeme patterns for a token
- They are used to represent the language for lexical analyzer
- They assist in finding the type of token that accounts for a particular lexeme

(1) Strings and languages

Alphabets are finite set of symbols Σ (sigma)

Σ = {0, 1} – binary alphabets

String represent: Σ = {a, b, c ... } lower case letters

Σ = {a, b} → i/p generate n number
w = {0, 1, 00, 01, 10, 11, 001, 010 ... }

length
empty
prefix
suffix
substring
proper subs

a, ba, ab, aab, ba.

to indicates the set of possible strings for the binary alphabet Σ.

Language (L) is the collection of strings which are generated from alp which are accepted by finite automata

$L = \{0^n \mid n >= 0\}$, Σ = {a, b} → a
L = {a, b, ab, ...

Concatenation of two string P and Q

P = 010

Q = 001

PQ = 010001

QP = 001010

PQ ≠ QP

3

Prefix

A prefix of any string S, is obtained by removing zero or more symbols from the end of S.

(S = balloon. ⇒ ball, balloon) S = abc ⇒ prefix = ϵ, abc, (ϵ, abc | abc · ϵ | ϵ . abc . ϵ) ↑ doesn't contain any chosen

Suffix: A suffix of any string S, is obtained by removing zero or more symbols from the beggining of S.

(S = balloon.

possible prefixes are: loon, balloon.)

2

Operations on Languages

- Union
- Concatenation
- Closure)

Union

Union of two language L and M produces the set of strings which may be either in language L or in language M or in both

$L \cup M = \{P \mid P \text{ is in } L \text{ or } P \text{ is in } M\}$ → pipe symbol

$L = \{0, 1\}$, $M = \{00, 11\}$, $L \cup M = \{0, 1, 00, 11\}$

Concatenation

The language L and M, produces a set of strings which are formed by merging the strings in L with strings in M.

$L \cup M = \{pq \mid p \text{ is in } L \text{ and } q \text{ is in } M\}$

x = abc, y = de, xy = abcde

(16)

Closure

(i) Kleene closure (L^*)

Kleene closure refers to zero or more occurrence of input symbols in a string and includes empty string ε

$$L^* = \bigcup_{i=0}^{\infty} L^i \quad \text{i.e. } L^0 \cup L^1 \cup L^2 \ldots$$

(ii) Positive closure (L^+)

It indicates one or more occurrence of input symbols in a string.

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

$L^* = \{\varepsilon, a, b, aa, ab, ba, bb, aab, aba, abba \ldots\}$

$L^+ = \{a, b, aa, ab, ba, bb, aab, abba\}$

$L^3 = \{aaa, aba, abb, bba, bab, bbb \ldots\}$

Precedence of operators

1. Unary operator (*)
2. Concatenation operator (.)
3. Union operator (| or ∪)

(17)

Regular Expressions

It is a combination of input symbols and language operators such as union, concatenation, and closure.

It can be used to describe the identities for a language.

- The identifier is a collection of letters, digits or underscore which must begin with a letter.

letter_ (letter_ | digit)*

S.NO	Regular expression	Languages
1	r	$L(r)$
2	a	$L(a)$
3	$r \mid s$	$L(r) \mid L(s)$
4	rs	$L(r)\, L(s)$
5	r^*	$(L(r))^*$

Language for regular expression

5

Regular Definition

Regular definition d, gives aliases to regular expression r and uses it for convenience

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$d_3 \rightarrow r_3$$
$$d_n \rightarrow r_n$$

Regular expressions for an identifier and number given as follows

letter → A | B | . . . | Z | a | b | . . . | z

digit → 0 | 1 | 2 . . . | 9

id → letter_ (letter | digit)*

num → digit (digit)*

Recognition of Tokens

It also generates code for examining the input string and to find the prefix (lexeme) that matches with any one of the patterns.

Rules for conditional statement

stmt → if expr then stmt
| if expr then stmt else stmt
| ε

expr → term relop term
| term

term → id | number

Conditions for branching statements

The terminals of the grammar which are if, then, else, relop, id and numbers are the names of tokens for lexical analyzer.

Lexical analyzer also performs stripping out of white spaces.

ws → (blank | tab | newline)+

Token names with their attribute value

Lexemes	Token names	Attribute value
Any ws	–	–
If	if	–
Then	then	–
Else	else	–
Any id	id	Pointer to symbol table entry
Any number	number	Pointer to symbol table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

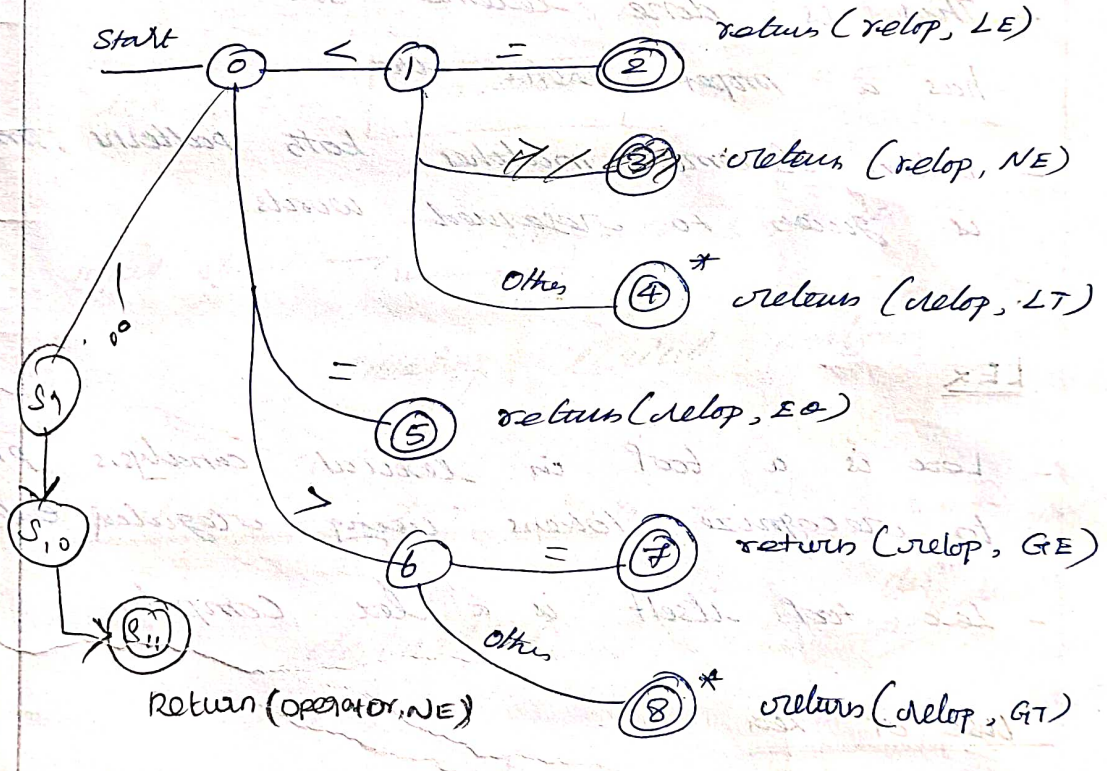
Transition Diagrams

→ Transition diagrams are pictorial representation of transition from one state to another one.

→ Start state

→ Final state

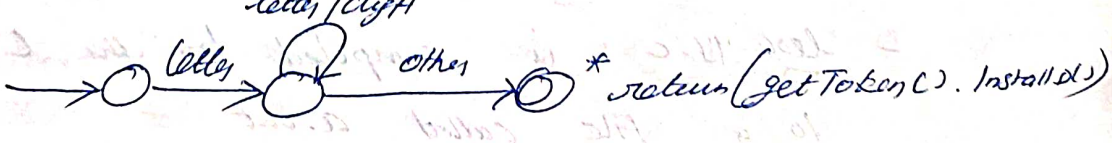
→ To indicate the retraction of forward pointer ('*')



Transition diagram for relational operations

Recognition of Reserved words and Keywords

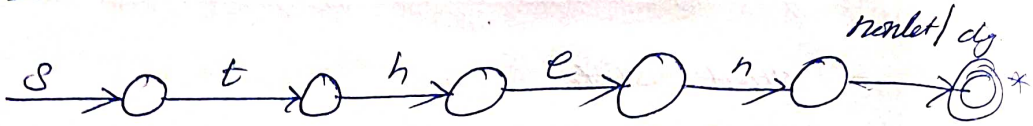
- Keyword patterns match that of identifiers.
- But they should be recognized differently



Transition diagram for identifier & keyword.

To handle reserved words different from identifi

- Create separate transition diagram for each keyword.



Transition diagram for keyword.

- A test for 'non-letter or digit to check the e for identifier
- if it reaches the accepting state it is recognised as keyword else identifier
- This is done lexeme such as the next val has a proper prefix then.
- when lexeme matches both patterns, priority is given to reserved words.

LEX

- Lex is a tool in lexical analysis phase to recognize tokens using regular expressio
- Lex tool itself is a lex compiler

Use of Lex

- lex.l - is an i/p file written in a language which describes the generation of lexical analyzer.

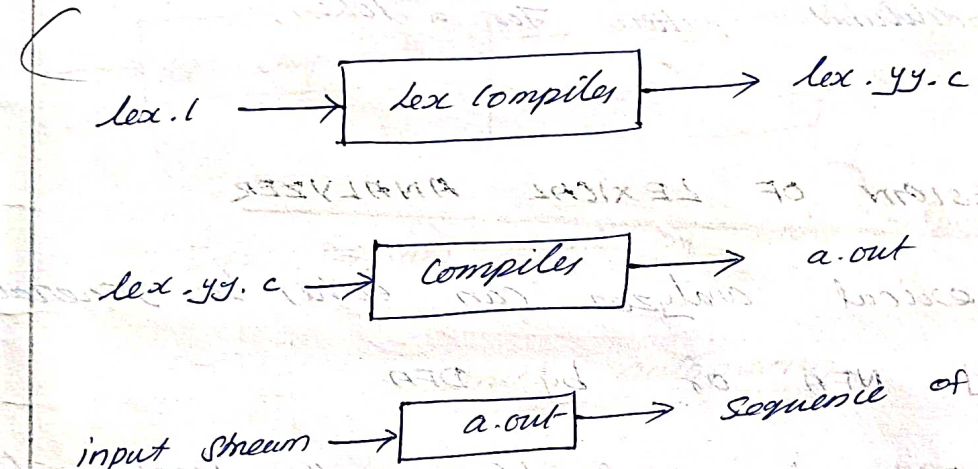
The lex compiler transforms lex.l to a C program known as lex.yy.c

- lex.yy.c is compiled by the C compiler to a file called a.out

- The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.

yylval is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.

- The attribute value can be numeric code, Pointer to symbol table or nothing.



creating lexical analyzer

Structure of Lex Programs

declarations
%%
translation rules
%%
auxiliary functions.

- declare variables, constants
- regular expression and code segment.
- hold the additional functions. These functions are compiled seperately and loaded with lexical analyzer.

Conflict Resolution in Lex

Conflict arises when several prefixes of input matches one or more patterns.

- Always prefer a longer prefix than a short prefix

- If two or more patterns are matched for the longest prefix, then the first pattern listed in lex program is preferred.

Lookahead operator

- Lookahead operator is the additional operator that is read by lex in order to distinguish additional pattern for a token.

DESIGN OF LEXICAL ANALYZER

- Lexical analyzer can either be generated by NFA or by DFA

- DFA is preferable in the implementation of lex.

Structure of Generated Analyzer

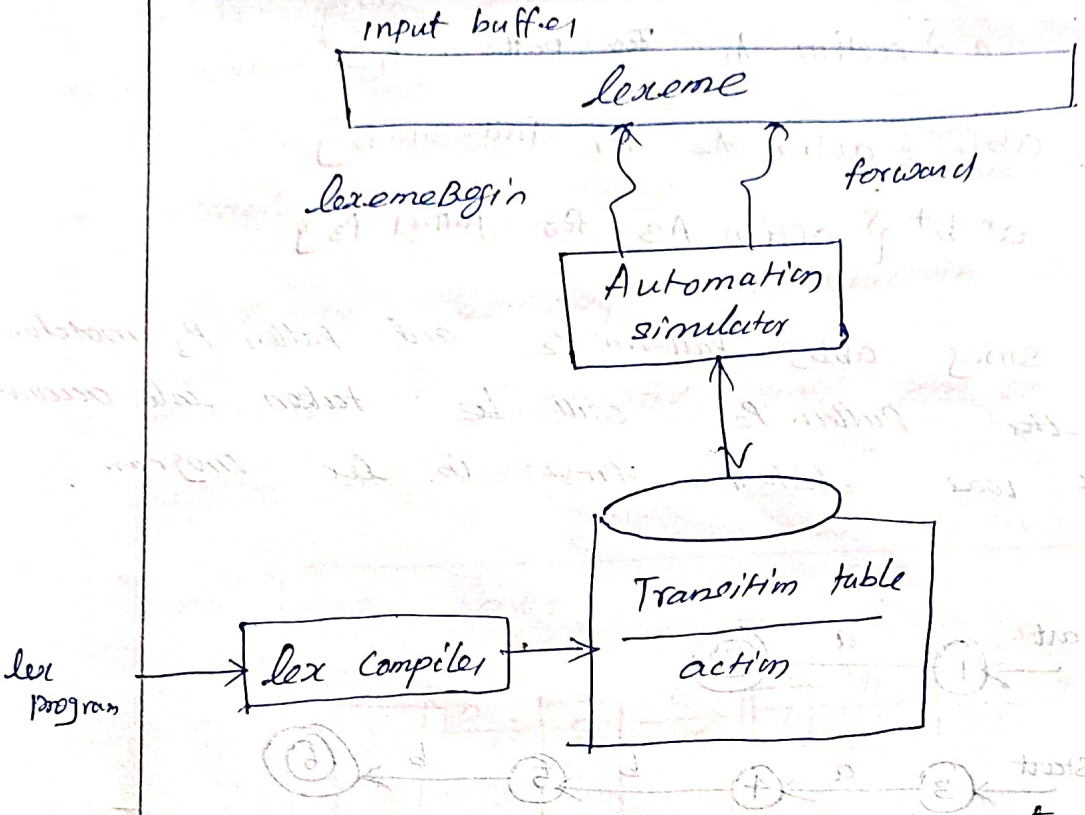
- Program to simulate automata

- Components created from lex program by lex itself which are listed as follows:

  * A transition table for automaton.

  * Functions that are passed directly through lex to the output.

  * Actions from input program (fragments of code) which are invoked by automaton simulator when needed.

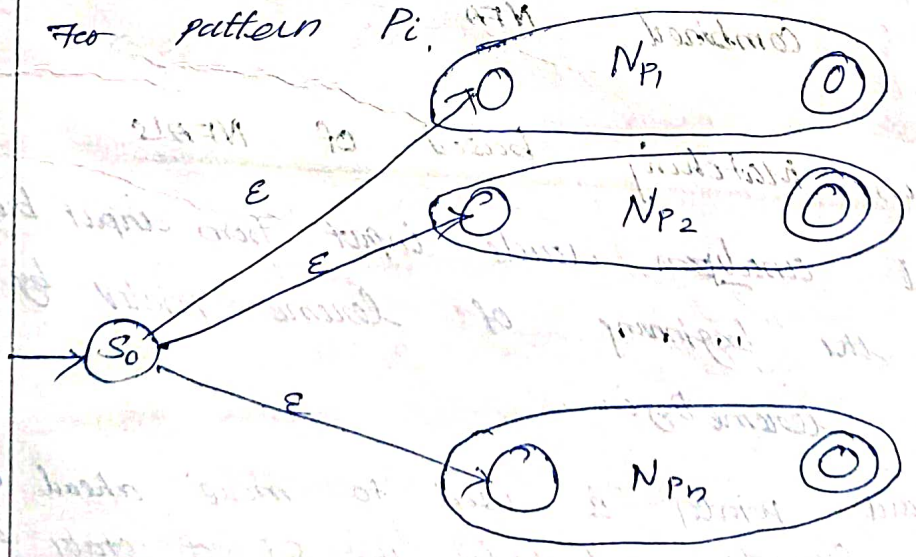


Lex program used by finite automation simulator.

Steps to construct automation

Step 1 : convert each regular expression into NFA either by Thompson's sub-set construction or direct method.

Step 2 : Combine all NFA into one by introducing new start state with ε-transitions to each of start states of NFA's N_i for pattern P_i.



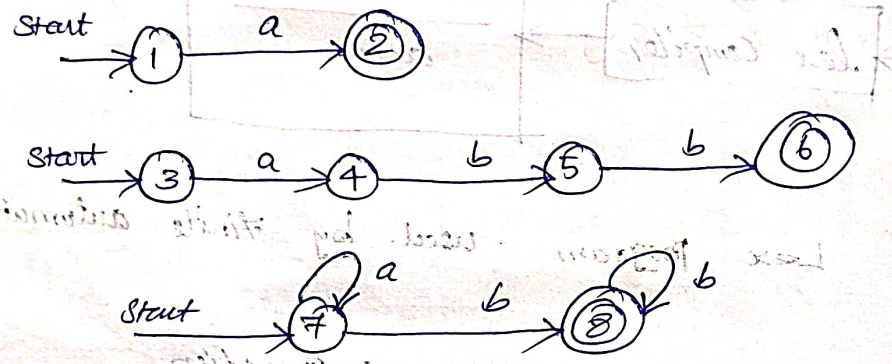
construction of NFA from Lex program

ex: a { action A_1 for pattern P_1 }

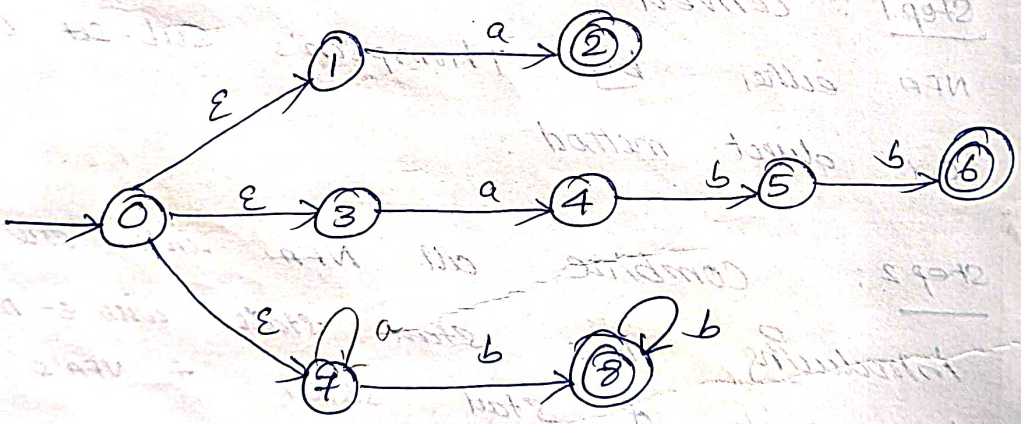
abb { action A_2 for pattern P_2 }

$a^* b^+$ { action A_3 for pattern P_3 }

For string abb, pattern P_2 and pattern P_3 match but the pattern P_2 will be taken into account as it was listed first in lex program.



NFA's for a, abb, $a^* b^+$



Combined NFA

Pattern Matching based on NFAs

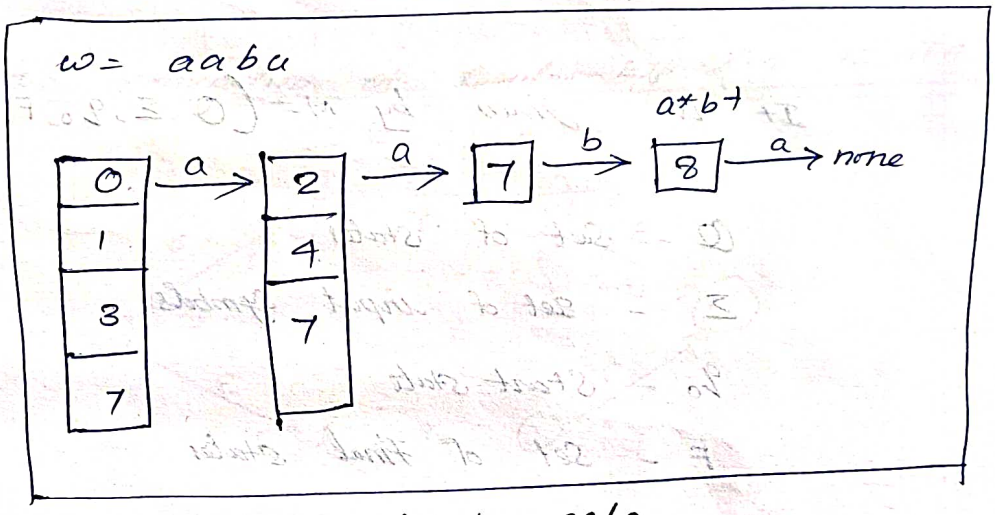
- Lexical analyzer reads input from input buffer from the beginning of lexeme pointer by the pointer lexeme Begin.
- Forward pointer is used to move ahead of input symbols, calculates the set of states it is in at each point.

- If NFA simulation has no next state for some input symbol, then there will be no longer prefix which reaches the accepting state exists.

- lexeme matching some pattern.

- process is repeated until one or more accepting states are reached.

ex: $w = aaba$



processing input aaba

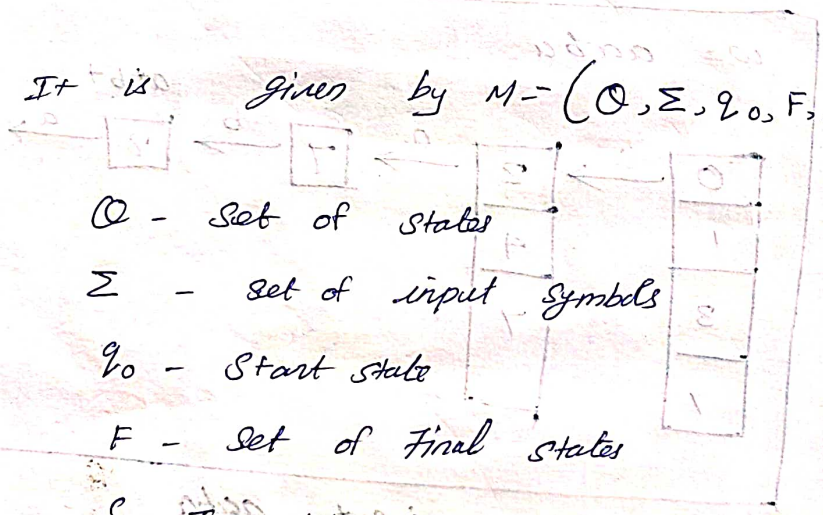
- process starts with ε closure of initial state 0.
- After processing all the input symbols, no state is found as there is no transition out of state 8 on input a.
- accepting is looked by retracting to previous state.
- From state 2 which is an accepting state is reached after reading i/p symbol a and therefore the pattern a has been matched.
- At state 8, string aab has been matched with pattern a^+b^+.
- By lex rule the longest matching prefix should be considered.
- Action A_3 corresponding to pattern P_3 will be executed for the string aab.

Finite Automata

A recognizer for a language is a program that takes as input a string x and answers yes if x is a sentence of the language and no otherwise.

⇒ A regular expression is compiled into a recognizer by constructing a generalized transition diagram called a Finite Automaton (FA)

It is given by $M = (Q, \Sigma, q_0, F, \delta)$



Q - Set of states

Σ - Set of input symbols

q_0 - Start state

F - Set of final states

δ - Transition function

Deterministic Finite Automata DFA

- For each state and for each input symbol exactly one transition occurs from the state.

Non-deterministic Finite Automata (NFA)

- More than one transition occurs for any input symbol from a state.
- Transition can occur even on empty string (ε)

Regular expression can be converted into DFA

(1) Thompson's Sub-set construction

- Given regular expression is converted into NFA
- Resultant NFA is converted into DFA

(ii) Direct method

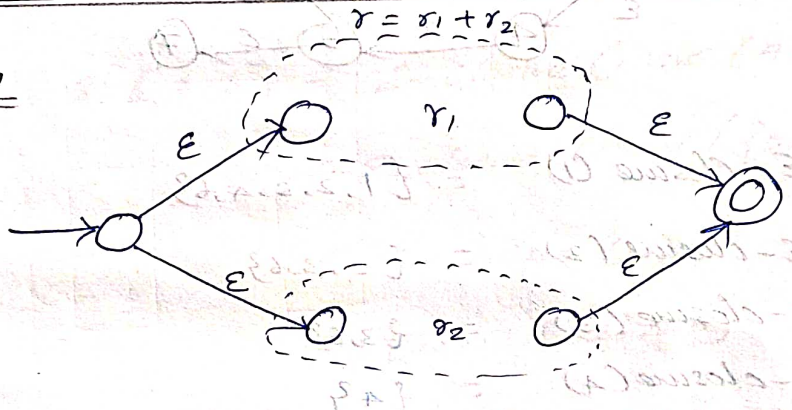
- In direct method, given regular expression is converted directly into DFA.

Regular Expressions to DFA

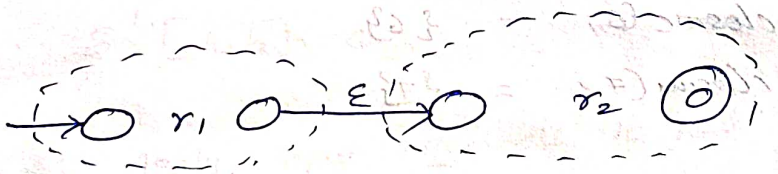
Regular expression is used to represent the language (lexeme) of finite automata (lexical analyzer)

Rules for conversion of Regular Expression to NFA

* Union

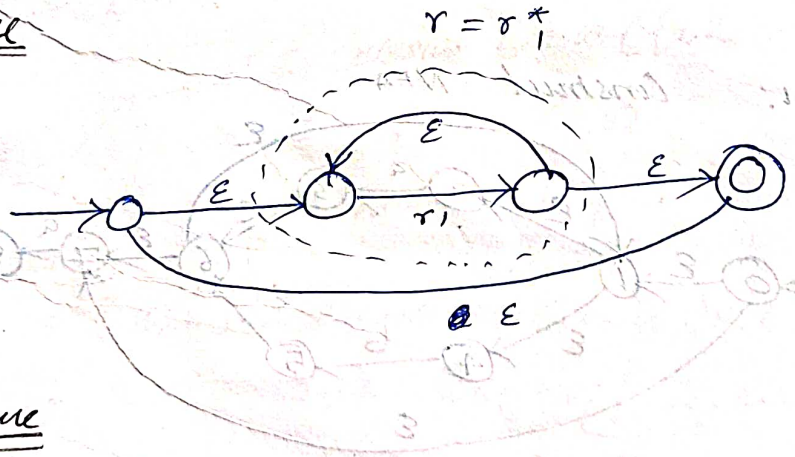


* concatenation



$r = r_1 r_2$

* closure



ε-closure

ε-closure is the set of states that are reachable from the state concerned on taking empty string as input.

It describes the path that consumes empty string (ε) to reach some state of NFA.

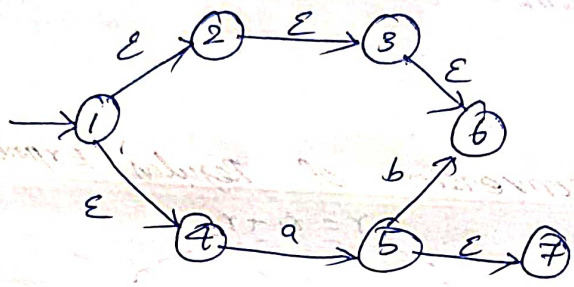
ex:

ε-closure $(q_0) = \{q_0, q_1, q_2\}$

ε-closure $(q_1) = \{q_1, q_2\}$

ε-closure $(q_2) = \{q_2\}$

ex:



ε-closure (1) = {1, 2, 3, 4, 6}

ε-closure (2) = {2, 3, 6}

ε-closure (3) = {3, 6}

ε-closure (4) = {4}

ε-closure (5) = {5, 7}

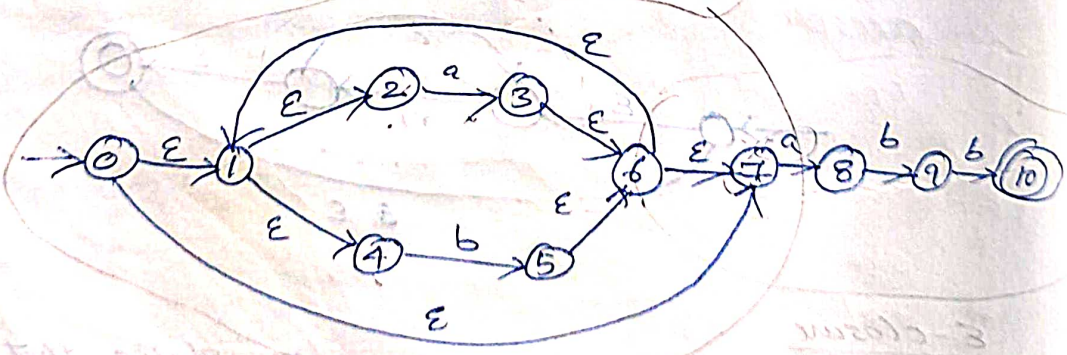
ε-closure (6) = {6}

ε-closure (7) = {7}

(4) ex: Construct Regular expression to DFA

RE = (a+b)* abb

Step 1: Construct NFA



Step 2: Start with finding ε-closure of state 0

ε-closure (0) = {0, 1, 2, 4, 7} = A

Step 3: Apply input symbol a, b to A

$\delta'[A, a] = \varepsilon\text{-closure}(\text{move}(A, a))$

$= \varepsilon\text{-closure}(\text{move}(\{0, 1, 2, 4, 7\}, a))$

$= \varepsilon\text{-closure}(3, 8)$ (0,a) (1,a) (2,a) (4,a) (7,a)

$= \{3, 6, 7, 1, 2, 4, 8\}$

$= \{1, 2, 3, 4, 6, 7, 8\} = B$

$\delta'[A, a] = B$

$\delta'[A, b] = \varepsilon\text{-closure}(\text{move}(A, b))$

$= \varepsilon\text{-closure}(\text{move}(\{0, 1, 2, 4, 7\}, b))$

$= \varepsilon\text{-closure}(5)$

$= \{5, 6, 7, 1, 2, 4, 8\}$

$= \{1, 2, 4, 5, 6, 7\} = C$

$\delta'[A, b] = C$

Step 4: Apply input symbols to new state B

$\delta'[B, a] = \varepsilon\text{-closure}(\text{move}(B, a))$

$= \varepsilon\text{-closure}(\text{move}(\{1, 2, 3, 4, 6, 7, 8\}, a))$

$= \varepsilon\text{-closure}(3, 8)$

$= \{1, 2, 3, 4, 6, 7, 8\} = B$

$\delta'[B, b] = \varepsilon\text{-closure}(\text{move}(B, b))$

$= \varepsilon\text{-closure}(\text{move}(\{1, 2, 3, 4, 6, 7, 8\}, b))$

$= \varepsilon\text{-closure}(5, 9)$ → 6, 1, 2, 4, 7

$= \{1, 2, 4, 5, 6, 7, 9\} = D$

$\delta'[B, b] = D$

Step 5: Apply input symbols to new state C

$\delta'[C,a]$ = ε-closure (move (C, a))

= ε-closure (move ({1,2,4,5,6,7}, a))

= ε-closure (3,8)

= {1,2,3,4,6,7,8} = B

$\delta'[C,a] = B$

$\delta'[C,b]$ = ε-closure (move (C, b))

= ε-closure (move ({1,2,4,5,6,7}, b))

= ε-closure (5)

= {1,2,4,5,6,7} = C

$\delta'[C,b] = C$

Step 6: Apply input symbols to new state D

$\delta'[D,a]$ = ε-closure (move (D, a))

= ε-closure (move ({1,2,4,5,6,7,9}, a))

= ε-closure (3,8)

= {1,2,3,4,6,7,8} = B

$\delta'[D,a] = B$

$\delta'[D,b]$ = ε-closure (move (D, b))

= ε-closure (move ({1,2,4,5,6,7,9}, b))

= ε-closure (5,10)

= {1,2,4,5,6,7,10} = E

$\delta'[D,b] = E$

Step 7: Apply input symbols to new State E

$$\delta'[E,a] = \varepsilon\text{-closure}(\text{move}(E,a))$$
$$= \varepsilon\text{-closure}(\text{move}(\{1,2,4,5,6,7,10\},a))$$
$$= \varepsilon\text{-closure}(3,8)$$
$$= \{1,2,3,4,6,7,8\} = B$$

$$\delta'[E,a] = B$$

$$\delta'[E,b] = \varepsilon\text{-closure}(\text{move}(E,b))$$
$$= \varepsilon\text{-closure}(\text{move}(\{1,2,4,5,6,7,10\},b))$$
$$= \varepsilon\text{-closure}(5)$$
$$= \{1,2,4,5,6,7\} = C$$

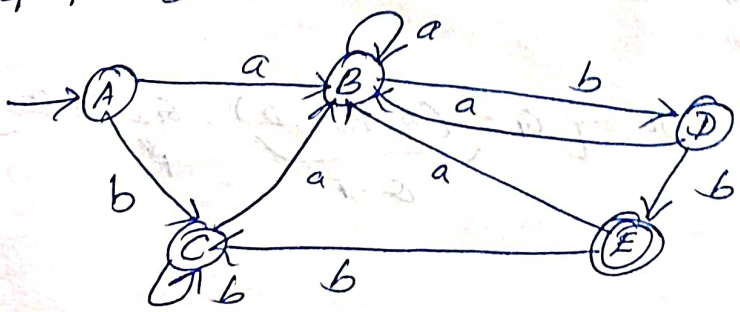
$$\delta'[E,b] = C$$

Step 8: Construct transition table

	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
* E	B	C

Note:
→ Start state is the ε-closure(0) i.e A.
→ Final state is the state that contains Final state of given NFA.

Step 9: Construct transition diagram



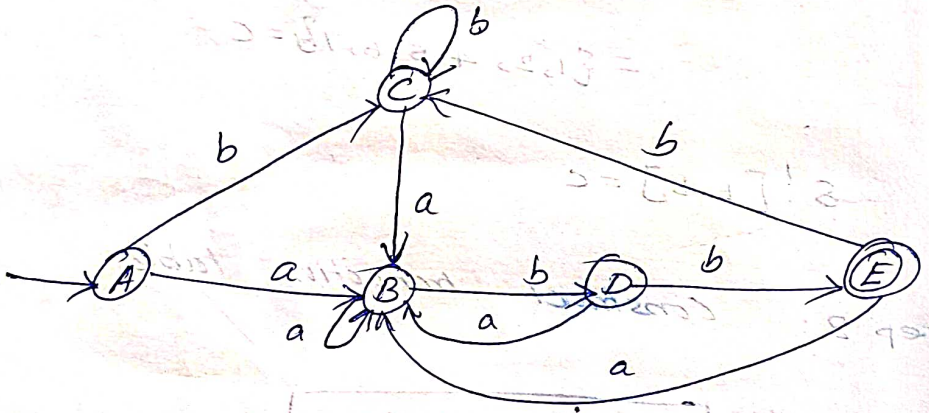
Minimization of DFA

→ Lexical analyzer can be implemented by DFA.

→ The number of states of constructed DFA must be minimized as for each state entry is made in a table which describes the lexical analyzer.

→ Minimization of DFA focuses on reducing the number of states in the given finite automata.

Ex: 1



Step 1: Construct transition table

	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C

Step 2: partition the states into two groups F and Q−F, final and non-final states respectively

$$\pi = \{ G_1 = \underbrace{(A, B, C, D)}_{Q-F}, \; G_2 = \underbrace{(E)}_{F} \}$$

Step 3: Construct π_1 by partitioning groups in π

Compare (A, B)

	a	b
A	B	C
B	B	D

G_1 G_1

Both are in same group

$\Rightarrow$ 1 - equivalent

Compare (A, C)

	a	b
A	B	C
C	B	C

G_1 G_1

Both are in same group

$\Rightarrow$ 1 - equivalent

Compare (A, D)

	a	b
A	B	C
D	B	E

↓

Not in same group

$\Rightarrow$ Not 1 - equivalent

$\pi_1 = \{(A, B, C), (D), (E)\}$

Step 4: Construct π_2 from π_1

For (A, B)

	a	b
A	B	C
B	B	D

↓

Not in same group

$\Rightarrow$ Not 2 - equivalent

For (A, C)

	a	b
A	B	C
C	B	C

G_1 G_2

$\Rightarrow$ 2 - equivalent

$\pi_2 = \{(A, C), (B), (D), (E)\}$

Step 5 : Construct π_3 from π_2

For (A, C)

	a	b
A	B	C
C	B	C

G_{i2} G_{i1}

Both are in same group

$\Rightarrow$ 3 - equivalent

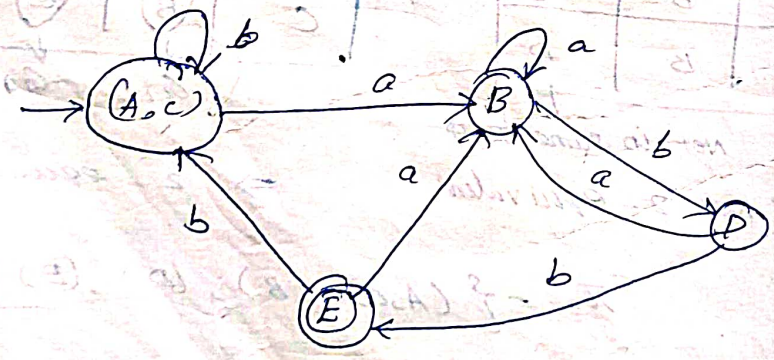
$$\pi_3 = \{(A, C), (B), (D), (E)\}$$

Step 6 : Stop partitioning since $\pi_3 = \pi_2$

Step 7 : Construct minimized transition table

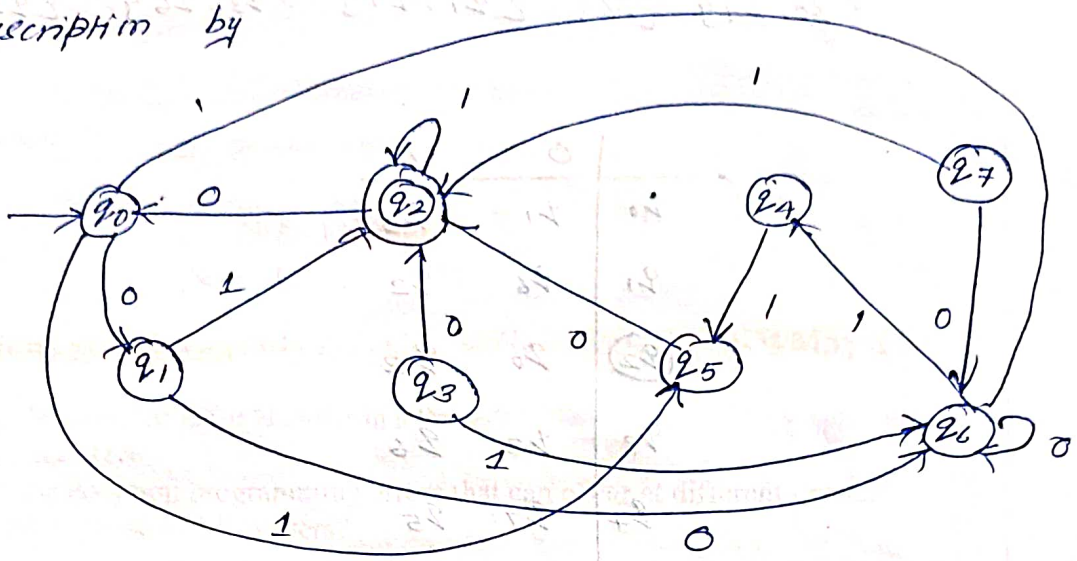
State \ Input	a	b
$\rightarrow$ (A, C)	B	(A, C)
B	B	D
D	B	E
* E	B	(A, C)

Step 8 : Draw minimized finite automata



Ex: 2

Construct a minimize DFA equivalent to the DFA description by



	0	1
→ q_0	q_1	q_5
q_1	q_6	q_2
(q_2)	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

0 — Equivalence

$\{q_0, q_1, q_3, q_4, q_5, q_6, q_7\}$ $\{q_2\}$

1 — Equivalence

$\{q_0, q_4, q_6\}$ $\{q_1, q_7\}$ $\{q_3, q_5\}$ $\{q_2\}$

2 — Equivalence

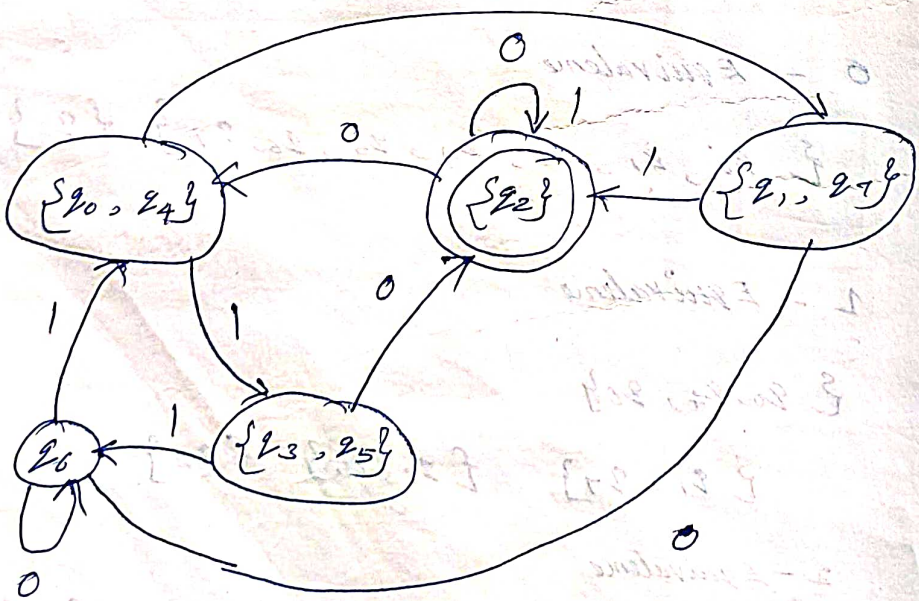
$\{q_0, q_4\}$ $\{q_6\}$ $\{q_1, q_7\}$ $\{q_3, q_5\}$ $\{q_2\}$

3- Equivalence

$\{q_0, q_4\}$ $\{q_6\}$ $\{q_1, q_7\}$ $\{q_3, q_5\}$ $\{q_2\}$

	0	1
q_0	q_1	q_5
q_1	q_6	q_2
(q_2)	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

	0	1
$\{q_0, q_4\}$	$\{q_1, q_7\}$	$\{q_3, q_5\}$
$\{q_6\}$	$\{q_6\}$	$\{q_0, q_4\}$
$\{q_1, q_7\}$	$\{q_6\}$	$\{q_2\}$
$\{q_3, q_5\}$	$\{q_2\}$	$\{q_6\}$
$(\{q_2\})$	$\{q_0, q_4\}$	$\{q_2\}$



Unit-I

Introduction to Compiler and Lexical Analysis

Introduction:

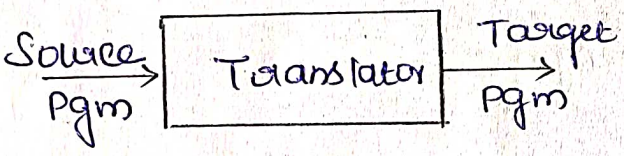
Compilers are basically translators. Designing a compiler for some language is a complex and time consuming process.

Translators:

A translator is one kind of pgm that takes source code as i/p and converts it into another form.

The i/p pgm is called source languages and the o/p pgm is called target language.

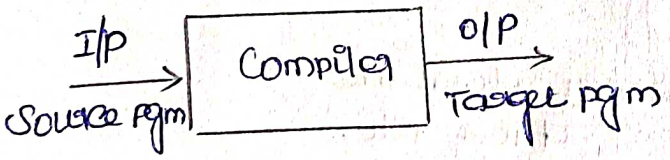
eg: C, C++, Fortran



Translator.

Process of compilation and Interpretation:

Compiler pgm which takes one language (source pgm) as i/p and translates it into an equivalent another language (target pgm)



Compiler

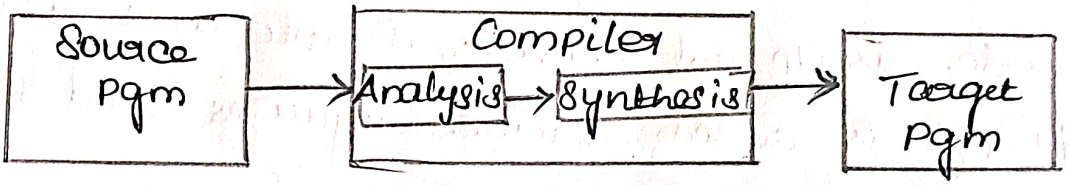
⇒ Source Prgms as higher level language such as C, PASCAL, FORTRAN and converts into low level language or a machine level language such as assembly language.

Analysis - Synthesis Model:

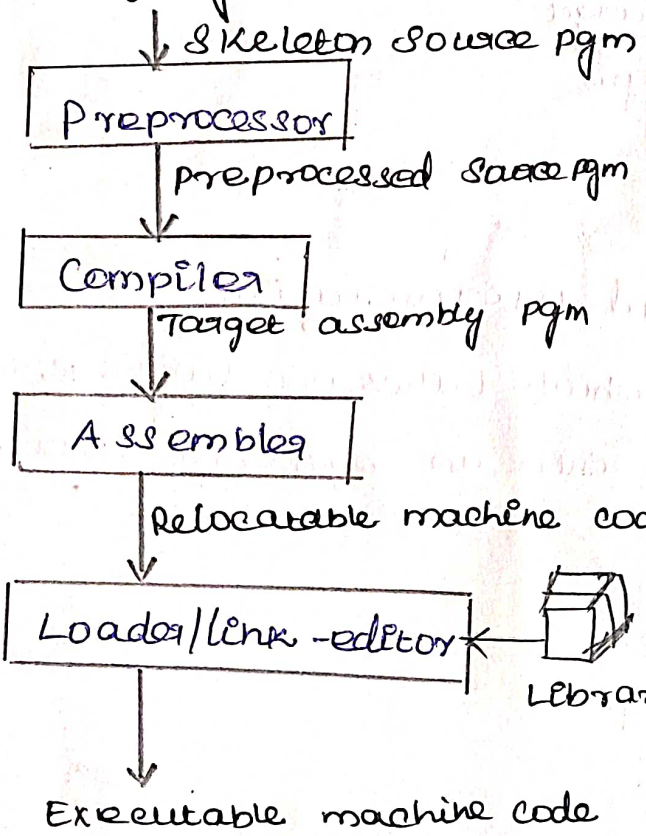
Compilation can be done in two parts:

① Analysis part: Source pgm is read and broken down into constituent piec

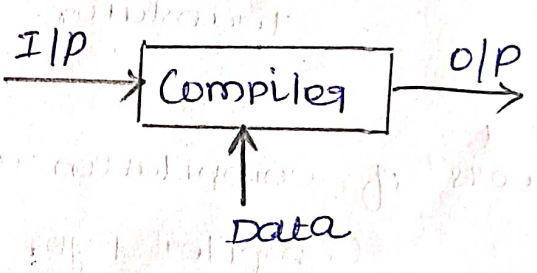
② Synthesis part: The intermediate form of the source language is taken and converted into an equivalent target pgm.



Execution of pgm:



(a) Process of Execution of Pgm.



(b) process of execution of pgm.

(pgm that can be loaded anywhere in the mm)

→ Generate executable file.

Linker - combines obj file

loader - loads exec. file to main mm

Properties of Compiler:

1) The compiler itself must be bug-free
2) It must generate correct machine code
3) The generated machine code must run fast
4) The compiler itself must run fast (Compilation time must be proportional to pgm size)
5) The compiler must be portable.
6) It must give good diagnostics and error msg.
7) The generated code must work well with existing debugg
8) It must have consistent optimization.

Why should we study compilers?

1. Performance can be measured.
2. The abilities of pgming languages can be understood
3. S/m building Tools such as LEX and can be developed by analytical study of compilers.

UNIT-II

SYNTAX ANALYSIS

PARSER

Parser is a program that obtains tokens from lexical analyzer and constructs the parse tree which is passed to the next phase of compiler for further processing.

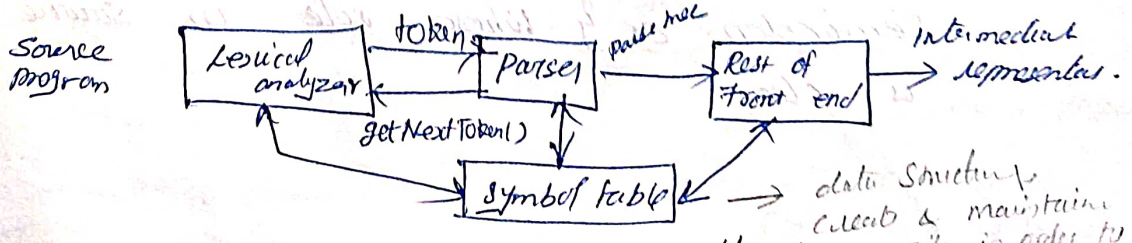
Parser implements CFG for performing error checks.

Types of Parser

- Top down parsers – construct parse tree from root to leaves
- Bottom up parsers – construct parse tree from leaves to root.

Role of Parser

* Once a token is generated by the lexical analyzer. It is passed to the parser.
* On receiving a token, the parser verifies the string of token names that can be generated by the grammar of source language.
* It calls the function getNextToken(), to notify the lexical analyzer to yield another token.
* It scans the token one at a time from left to right to construct the parse tree
* It also checks the syntactic construct of the grammar.



Position of parser in compiler

Need for parser

- Parser is needed to detect syntactic errors efficiently.
- Error is detected as soon as a prefix of the input cannot be completed to form a string in the language.
- This process of analyzing the prefix of input is called Viable-prefix property.

Syntax Analysis Error: ① Errors in structure
② Missing operators
③ unbalanced parenthesis

Error Recovery strategies

* It is used by the parser to recover from errors once it is detected.
* The simplest recovery strategy is to quit parsing with an error message for the itself.

1. Panic Mode Recovery

1. It prevents the parser from developing infinite loops.
2. When parser finds an error in the statements it ignores the rest of the statement by not processing the input.
3. The parser intends to find designated set of synchronizing tokens by discarding input symbols one at a time.
4. Synchronizing tokens may be delimiters semicolon or } whose role in source program is clear.

2. Phrase Level Recovery

- When a parser finds an error, it tries to take corrective measures so that the rest of inputs of statements allow the parser to parser ahead.
- one wrong correction will lead to an infinite loop.

3. Error production

- productions which generate erroneous constructs are augmented to the grammer by considering common errors that occur.
- These production detect the anticipated errors during parsing.
- Error diagnostic about the erroneous constructs are generated by the parser.

- Global correction

* There are algorithms which make changes to modify an incorrect string into a correct string.

GRAMMARS

It is used to describe the syntax of a programming language. It specifies the structure of expression and statement.

Stmt → if (expr) then Stmt

Types of Grammar

- Type 0 Grammar — Turing machine
- Type 1 Grammar — context sensitive language
- Type 2 Grammar — context free Grammar
- Type 3 grammar — Finite Automata

Context Free Grammar

A context free grammar G is defined by four tuples as

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

Where

- G - Grammar
- V - Set of variables
- T - Set of terminals
- P - Set of productions
- S - Start symbol

It produces context free Language (CFL) which is defined as

$$L(G) = \{ w \mid w \text{ is in } T^* \text{ and } S \xrightarrow[G]{*} w \}$$

- L - Language
- G - Grammer
- w - input string
- S - Start symbol
- T - Terminal

Conventions

Terminals

- Lowercase letters ie a, b, c
- Operators +, −, *
- Punctuation symbols, Comma, Parenthesis
- Digits 0, 1, 2 ... 9
- Boldface letters id, if

Non-terminals — A, B, C

Start symbols — → S

Production — LHS → RHS, (or) head → body

$E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T*F \mid T/F \mid F$

$F \rightarrow (E) \mid id$

Solution:

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

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

$S = \{E\}$

P:

$E \rightarrow E+T$

$E \rightarrow E-T$

$E \rightarrow T$

$T \rightarrow T*F$

Writing a Grammar

- Grammars are more powerful than regular expressions.

$RE = (a|b)^* abb$

$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \varepsilon$ (Set of string ending with abb

Derivations

Derivation is used to find whether the string belongs to a given grammar.

Types:

- Leftmost derivation
- Rightmost derivation

*Leftmost derivation

At each and every step the leftmost non-terminal is expanded by substituting its corresponding production to derive a string.

ex:

$E \rightarrow E+E \mid E*E \mid id$

let: $w = id+id*id$

$E \rightarrow E+E$

lm

$E \rightarrow id+E$ [$E \rightarrow id$]

lm

$E \rightarrow id+E*E$ [$E \rightarrow E*E$]

lm

$E \rightarrow id+id*E$ [$E \rightarrow id$]

lm

$E \rightarrow id+id*id$ [$E \rightarrow id$]

lm

ex:

$S \rightarrow SS+ \mid SS* \mid a$

$w = aa+a*$

H.W

$S \rightarrow SS*$

$S \rightarrow SS+S*$ [$S \rightarrow SS+$]

$S \rightarrow aS+S*$ [$S \rightarrow a$]

$S \rightarrow aa+S*$ [$S \rightarrow a$]

$S \rightarrow aa+a*$ [$S \rightarrow a$]

Rightmost derivation

ex:

$E \rightarrow E+E \mid E*E \mid id$

Let $w = id + id * id$

$E \rightarrow E+E$

$E \rightarrow E+E*E$ $(E \rightarrow E*E)$

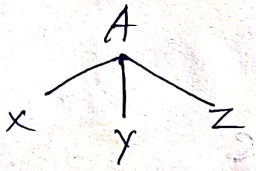
$E \rightarrow E+E*id$ $[E \rightarrow id]$

$E \rightarrow E+id*id$ $[E \rightarrow id]$

$E \rightarrow id+id*id$ $[E \rightarrow id]$

Parse Tree

- Parse tree is a hierarchical structure which represents the derivation of the grammar to yield input strings.
- Root node of parse tree has the start symbol of the given grammar from where the derivation proceeds.

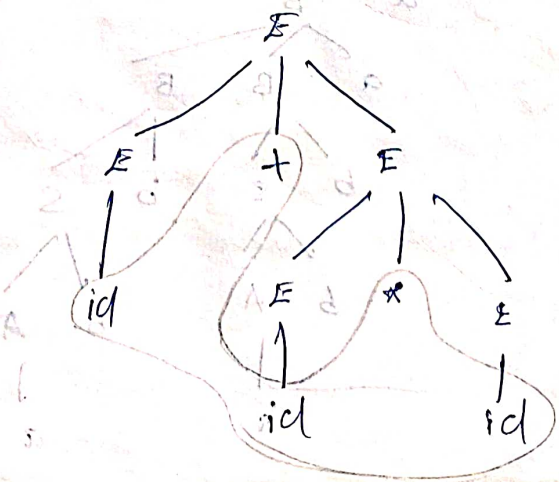


ex: Construct parse tree

$E \rightarrow E+E \mid E*E \mid id$

Let

$w = id + id * id$



ex: Let G be the grammar

$S \to aB \mid bA$

$A \to a \mid aS \mid bAA$

$B \to b \mid bS \mid aBB$

$S \to aABe$

$A \to Abc \mid b$

$B \to d$

$w = abbcde$

H.W

For the string $baaabbabba$. Find leftmost derivation, rightmost derivation and parse tree.

Leftmost

S

bA

baS

$baaB$

$baaaBB$

$baaabSB$

$baaabbAB$

$baaabbaB$

$baaabbabS$

$baaabbabbA$

$baaabbabba$

$S \to bA$

Rightmost

S

bA

baS

$baaB$

$baaaBB$

$baaaBbS$

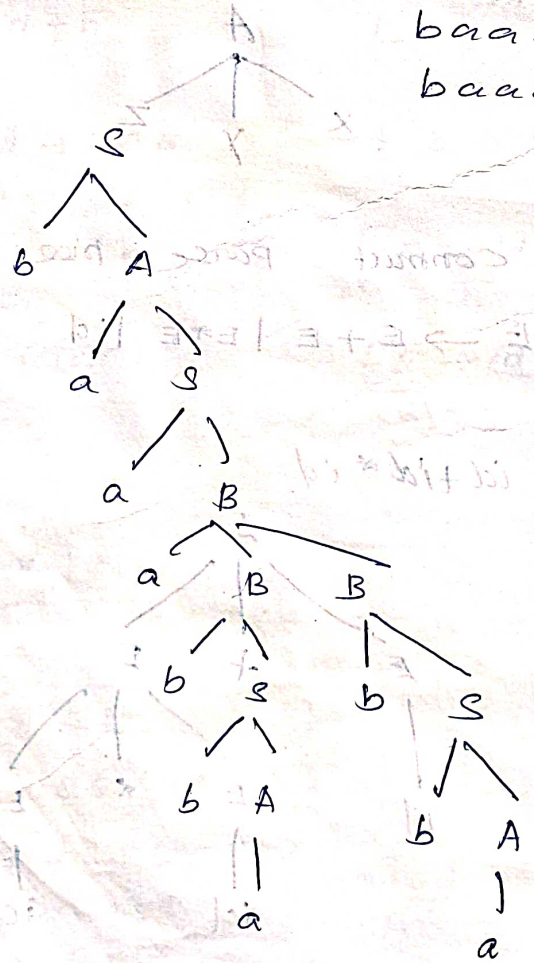
$baaaBbbA$

$baaaBbba$

$baaabSbba$

$baaabbAbba$

$baaabbabba$



AMBIGUITY

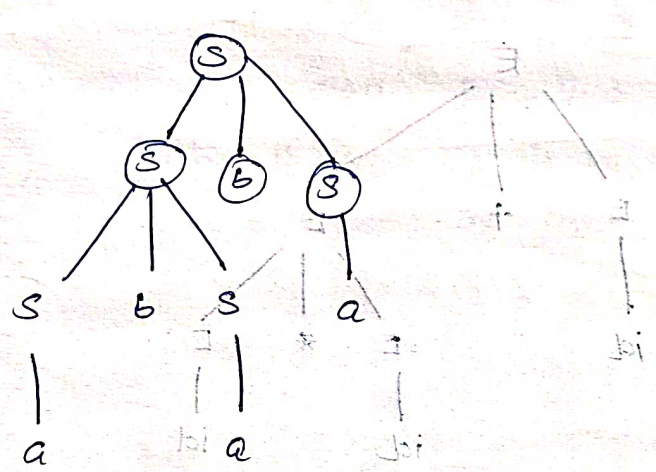
Grammar G is said to be ambiguous if it produces different parse trees for the same sentence or yield w which has been derived from the same start symbol.

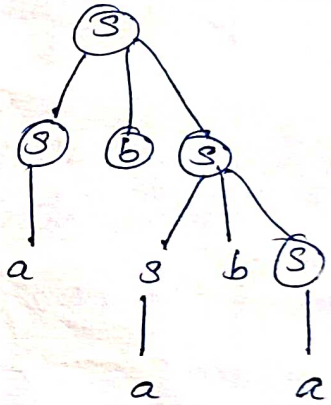
ex: 1 :

if G is the grammar $S \to SbS \mid a$. Show that G is ambigous

W = ababa be the string

$S \to SbS$
$S \to a$





LM ⇒ ababa ⇐ RM

---

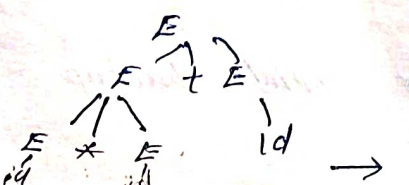
ex 2 :

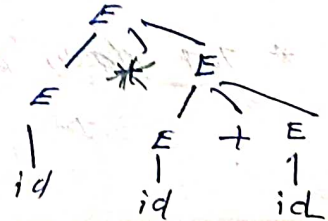
The CFG given by $G = (V, T, P, S)$ where

(H.W)

$V = \{E\}$
$T = \{id\}$
$P = \{E \to E+E,\ E \to E*E,\ E \to id\}$
$S = \{E\}$

Is the grammar ambigous ? G = id*id+id





Top- down Parsing

Top- down parsing constructs parse tree for the input string,

Starting from root node and creating the nodes of parse tree in pre-order.

ex: $E \rightarrow E+E \mid E*E \mid id$

$w = id+id*id$

$S \rightarrow xpz$

$p \rightarrow yw \mid y$

w =	x	y	z

$E \rightarrow E+E$

lm

$E \rightarrow id+E$

$E \rightarrow id+E*E$

$E \rightarrow id+id*E$

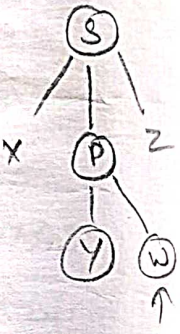
$E \rightarrow id+id*id$

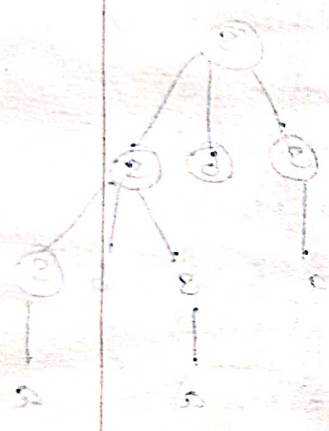
$E \rightarrow id$

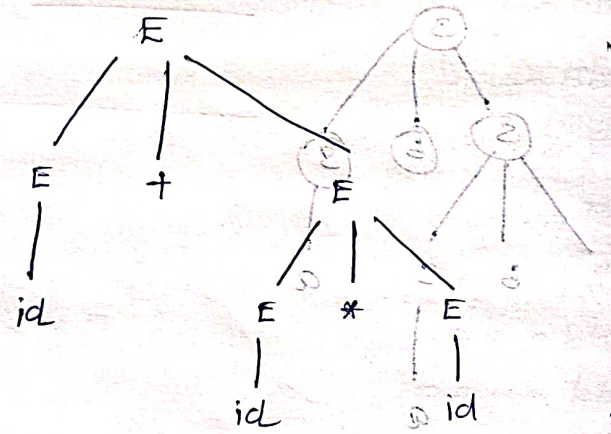
$E \rightarrow E*E$

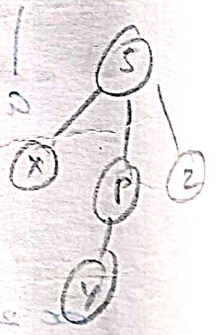
$E \rightarrow id$

$E \rightarrow id$









General strategies

* **Brute - force method**

All possible combinations are attempted before the failure to parse is recognized.

* **Recursive descent**

It is parsing techniques which does not allow backup.

* Involves backtracking and left recursion

* **Top down parsing**

It with limited or partial backup.

Recursive descent parser

* Recursive descent parser is a top-down parser.
* It requires backtracking to find the correct productions to be applied.
* The parsing program consists of a set of procedures, one for each non-terminal.
* Process begins with the procedure for start symbol.

```
Void A() {
    Choose an A-production, A → X1 X2 X3 ... Xk;
    For (i=1 to k)
        if (Xi is a non-terminal)
            call procedure Xi();
        else if (Xi equals the current input symbol a)
            advance the input to the next symbol;
        else
            error;
}
```

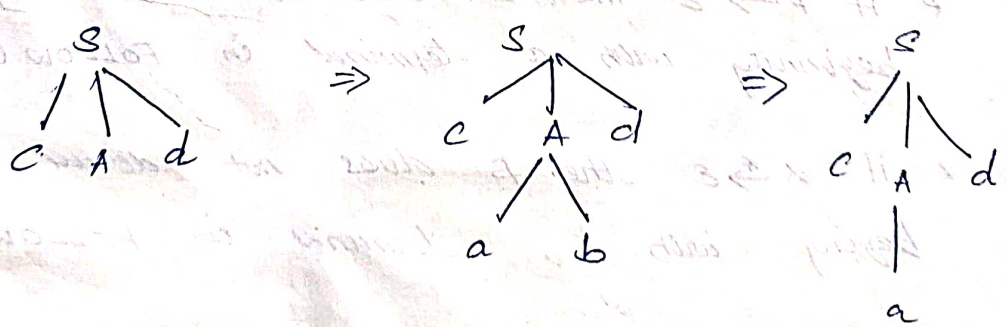
Procedure for a non-terminal in top-down parser

ex: Let Grammar G be

$S \rightarrow cAd$

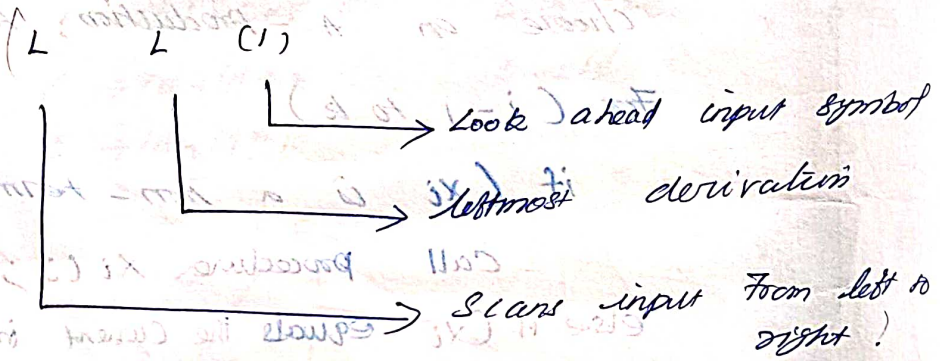
$A \rightarrow ab \mid a$

$w = cad$



PREDICTIVE PARSER / LL(1) PARSER

- Predictive parser are top-down parser.
- It is a type of recursive descent parser but with no backtracking
- It can be implemented non-recursively by using stack data structure.
- They can also be termed as LL(1) parser, as it is constructed for a class of grammars called LL(1)



A Grammar G is LL(1) if there are two distinct productions $A \to \alpha | \beta$ with the following conditions hold:

* For no terminal α and β derive strings beginning with a
* At most one of α and β can derive empty string
* if $\beta \Rightarrow^{*} \varepsilon$ then α does not derive any string beginning with a terminal in FOLLOW(A).
* if $\alpha \Rightarrow^{*} \varepsilon$ then β does not derive any string beging with a terminal in FOLLOW(A).

(LL(1)) Parser is designed using stack data structure explicitly to hold Grammar symbols.

- Left-recursion is eliminated
- Common prefixes are also eliminated (Left-factoring)

Eliminating left-recursion

A Grammar is left recursion if it has a production of the from $A \to A\alpha$, for some string α.

To eliminate left-recursion for the production

$$A \to A\alpha \mid \beta$$

Rule:

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \varepsilon$$

$A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$

ex:

$$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid \beta_1 \mid \beta_2 \mid \dots \beta_m$$

Solution

$$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_m A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \varepsilon$$

ex:

$$S \to Aa \mid b$$
$$A \to Ac \mid Sd \mid \varepsilon$$

$A \to \beta A'$
$A \to bA'$
$A' \to aA' \mid \varepsilon$

Sol:

Substitute the productions of S in A.

$$A \to Ac \mid Aad \mid bd \mid \varepsilon$$

After eliminating left-recursion

$$S \to Aa \mid b$$
$$A \to bdA' \mid A'$$
$$A' \to cA' \mid adA' \mid \varepsilon$$

$S \to bdA'$
$A' \to cA' \mid adA' \mid \varepsilon$

Left factoring

This can be done through rewriting the productions until enough of the input has been seen

Rule

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2$$

ex:

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_m \mid \gamma$$

Sol:

$$A \to \alpha A' \mid \gamma$$
$$A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

ex:

$$S \to \underbrace{iEtS}_{\alpha\ \beta_1} \mid \underbrace{iEtSeS}_{\alpha_2\ \beta_2} \mid a$$
$$E \to b$$

Sol:

$$S \to iEtSS' \mid a$$
$$S' \to eS \mid \varepsilon$$
$$E \to b$$

Computation of First

Rules:

To compute FIRST(X), where X is a grammar symbol.

- If X is a terminal, then FIRST(X) = {X}
- If $X \to \varepsilon$ is a production, then add ε to FIRST(X)
- If X is a non-terminal and $X \to Y_1 Y_2 \ldots Y_k$ is a production, then add FIRST(Y_1) to FIRST(X). If Y_1 derives ε, then add FIRST(Y_2) to FIRST(X).

Computation of FOLLOW

- For the FOLLOW (Start Symbol) Place $, where $ is the input end marker.
- If there is a production $A \to \alpha B\beta$, then everything in FIRST(β) except ε is in FOLLOW(B)
- if there is a production $A \to \alpha B$, or a production $A \to \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Construction of parsing table

Input Grammar G
Output parsing table M
Method For each production $A \to \alpha$, do the following:

1. For each terminal a in FIRST(α), add $A \to \alpha$ to M[A, a]
2. If ε is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to M[A, b].
3. If ε is in FIRST(α) and $ is in FOLLOW(A), add $A \to \alpha$ to M[A, $].
4. If no production is found in M[A, a] then set error to M[A, a].

Parsing of input

* Stack – holds sequence of grammar symbol with $ on the bottom of stack
* Input buffer – contains the input to be parsed with $ as end marker for the string.
* parsing table.

Alg:

Input: A string w and parsing table M for grammar G

Output: If w is in L(G) then success: otherwise error

Method

Let a be the first symbol of w; Let x be the top of stack symbol; while (x ≠ $)

if (x = a) pop the stack and let a be the next symbol of w;

else if (x is a terminal) error();

else if (M[x,a] is an error entry) error();

else if (M[x,a] = x → Y1 Y2 .. Yk){

Output the production x → Y1 Y2 .. Yk;

Pop the stack;

Push Yk, Yk-1 .. Y1 onto stack with Y1 on the top;}

Let x be the top stack symbol;}

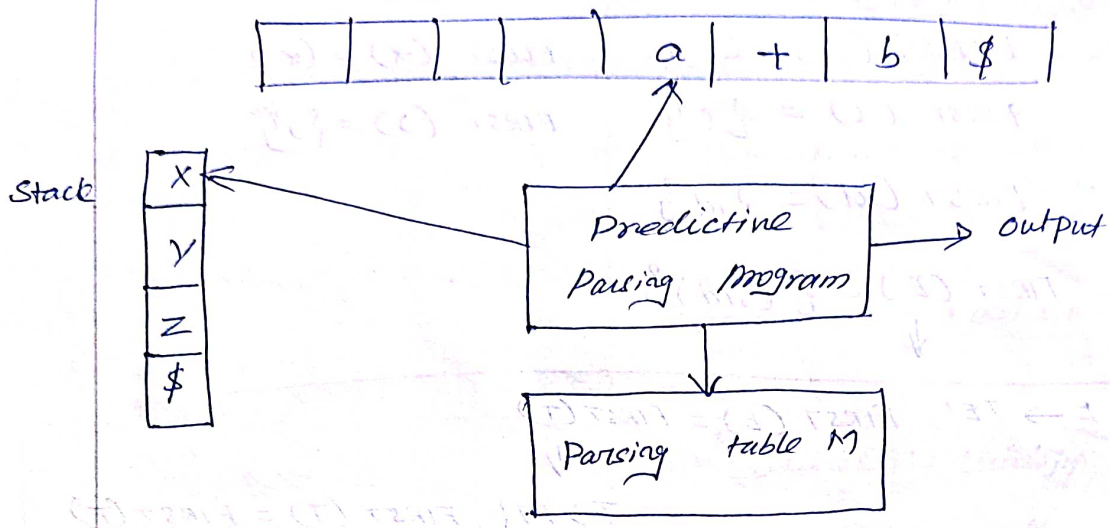
Non - recursive predictive parser

- This prevent implicit recursive calls.
- It can also be termed as table - driven predictive parser.

Components

* Input buffer – holds input string to be parsed.
* Stack – holds sequence of grammar symbols.
* predictive parsing algorithm – contains steps to parse the input string, controls the

Parsing table - contains entries based on which parsing actions has to be carried out.



Model of a table-driven predictive Parser

LL(1)

ex: construct predictive parsing table for the grammar.

Steps: 1. Eliminating Left Recursion
2. left factoring
3. FIRST & FOLLOW functions
4. Predictive Parsing table
5. Parse input string and parse the input id + id * id

Same means occurs

$E \rightarrow E+T \mid T$ — Left Recursion

$T \rightarrow T*F \mid F$ — Same, so Left Recursion

$F \rightarrow (E) \mid id$ — not

Solution:

Step 1: Eliminate left-recursion

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

$V = \{E, T, F, E', T'\}$

$T = \{+, *, (,), id\}$

Step 2: Left-factoring

No common prefixes for any production with same head. (ie) no need of left-factoring.

Step 3: Compute FIRST

First (terminal) = {terminal}

FIRST (+) = (+) FIRST (*) = (*)

FIRST (() = {(} FIRST ()) = {)}

FIRST (id) = {id}

FIRST (E) = {(, id)}

↓

E → TE', FIRST (E) = FIRST (T)

↓

T → FT', FIRST (T) = FIRST (F)

→ F → (E), FIRST (F)

→ F → id, FIRST (F) = {id}

E → E+T | T

T → T*F | F

F → (E) | id

→ FIRST (F) = {(, id}

Hence, FIRST (E) = FIRST (T) = FIRST (F) = {(, id}

Rule if a non-terminal derives ε, then ε is in FIRST (non-terminal)

If A → ε, then FIRST (A) = {ε}

FIRST (E') = {+, ε} → [E' → +TE' is a production start with a terminal '+', FIRST (E') include '+']

FIRST (T') = {*, ε} [T' → *FT' is a production start with terminal '*' FIRST (T') include '*']

Step 4: Compute FOLLOW

Rule Place $ in Follow (start symbol)

FOLLOW (E) = {$,)}

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

Since, F → (E) is a production where E is followed by a terminal ')' FOLLOW [E] contain ')']

Production F → (E)

A α B β

4 (c)

FOLLOW (E) = FIRST (β) = {)}

FOLLOW(E') = {$,)}

Production $E \rightarrow T\ E'$

$E \rightarrow A$, $T \rightarrow \alpha$, $E' \rightarrow B$

(i.e)
FOLLOW(E') = FOLLOW(E) = {), $}

FOLLOW(T) = {+,), $}

Production $E' \rightarrow +\ T\ E'$

$E' \rightarrow A$, $+ \rightarrow \alpha$, $T \rightarrow B$, $E' \rightarrow \beta$

(i.e)
FOLLOW(T) = FIRST(B) = FIRST(E')
= {+, ε}

Rules:
1) $
2) if $A \rightarrow \alpha B \beta$
3) $A \rightarrow \alpha B$

(i.e)
FOLLOW(B) = FOLLOW(A) ∪ {FIRST(β) − ε}

FOLLOW(T) = FOLLOW(E') ∪ {FIRST(E') − ε}
= {$,)} ∪ [{+, ε} − ε]
= {$,), +}

FOLLOW(T') = {+,), $}

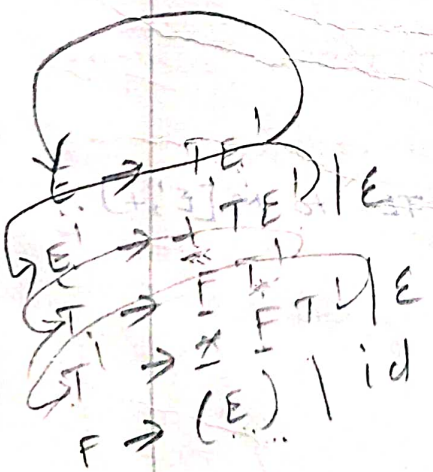
Production $T \rightarrow F\ T'$

$T \rightarrow A$, $F \rightarrow \alpha$, $T' \rightarrow B$

ie FOLLOW(B) = FOLLOW(A)

FOLLOW(T') = FOLLOW(T) = {+,), $}

FOLLOW(F) = {+, *,), $},



Production $T' \rightarrow *\ F\ T'$

$T' \rightarrow A$, $* \rightarrow \alpha$, $F \rightarrow B$, $T' \rightarrow \beta$

ie. FOLLOW(F) = FIRST(B) = FIRST(T') = {*, ε}

Since ε is in FIRST(T') everything in FOLLOW(T') is in FOLLOW(F).

ie. FOLLOW(B) = FOLLOW(A) ∪ {FIRST(β) − ε}

FOLLOW(F) = FOLLOW(T') ∪ {FIRST(T') − ε}
= {$,), +} ∪ [{*, ε} − ε]
= {$,), +, *}

Step 5. construct parsing table.

Non-terminal	input symbol				
	id	+	*	(	)
E	$E \to TE'$			$E \to TE'$	
E'		$E' \to +TE'$			$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$	
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$	

To make over parsing table entry, analyze each and every production.

For the production

$$E \to T \quad E'$$
$$\downarrow \quad \downarrow$$
$$A \quad \alpha$$

FIRST (TE') = FIRST (T) = {(, id)}

add the production $E \to TE'$ to M[E, (] and M[E, id]

For the production,

$$E' \to + \quad T \quad E'$$
$$\downarrow \quad \downarrow$$
$$A \quad \alpha$$

FIRST (+ TE') = FIRST(+) = {+}

So, add the production $E' \to +TE'$ to M[E', +]

For the production

$$E' \to \varepsilon$$
$$\downarrow \quad \downarrow$$
$$A \quad \alpha$$

FIRST (ε) = {ε}

Since, FIRST (α), ie FIRST (ε) contains ε, add the production to M[A, FOLLOW(A)], FOLLOW(E') = {), $}

So, add the production $E' \to \varepsilon$ to M[E',)] and M[E', $]

For the production,

$$T \rightarrow F\ T'$$
$$\downarrow \quad \downarrow$$
$$A \quad \alpha$$

$$FIRST(FT') = FIRST(F) = \{(, id)\}$$

add the production $T \rightarrow FT'$ to $M[T, (]$ and $M[T, id]$.

For the production

$$T' \rightarrow *\ F\ T'$$
$$\downarrow \quad \downarrow$$
$$A \quad \alpha$$

$$FIRST(FT') = FIRST(*) = \{*\}$$ ~~$= \{(, id)\}$~~

add the production ~~$T \rightarrow FT'$ to $M[T, (]$ and $M[T, id]$~~

$T' \rightarrow *FT'$ to $M[T', *]$

For the production

$$T' \rightarrow \varepsilon$$
$$\downarrow \quad \downarrow$$
$$A \quad \alpha$$

$$FIRST(\varepsilon) = \{\varepsilon\}$$

add the production $T' \rightarrow \varepsilon$ to $M[T', +]$, $M[T',)]$ and $M[T', \$]$.

For the production

$$F \rightarrow (\ E\ )$$
$$\downarrow \quad \downarrow$$
$$A \quad \alpha$$

$$FIRST((E)) = FIRST(() = \{(\}$$

add the production $F \rightarrow (E)$ to $M[F, (]$.

For the production

$$F \rightarrow id$$

$A \quad \alpha$

FIRST (id) = {id}

add the production $F \rightarrow id$ to M[F, id].

Step 6 : Parse the given input

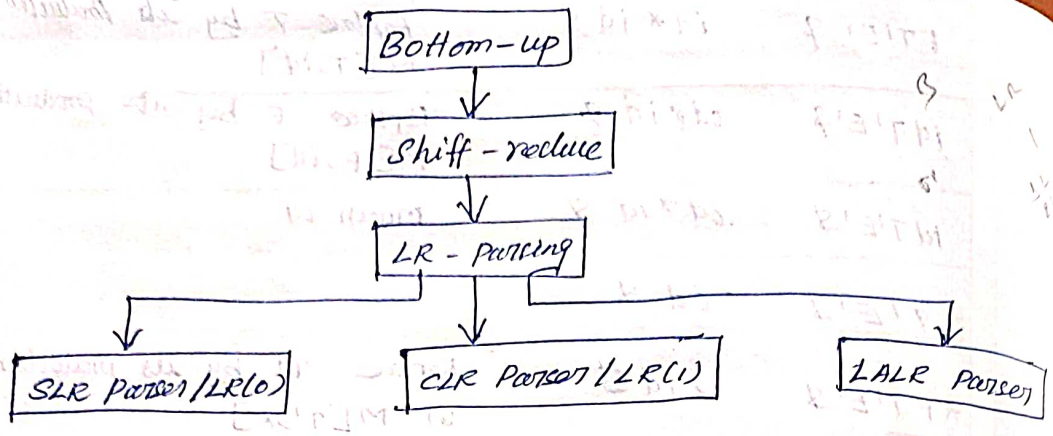
$w = id + id * id$

Stack	Input	Action
E $	id + id * id $	
T E' $	id + id * id $	Replace E by its production $E \rightarrow TE'$ in M[E, id]
F T' E' $	id + id * id $	Replace T by its production $T \rightarrow FT'$ in M[T, id]
id T' E' $	id + id * id $	Replace F by its production $F \rightarrow id$ in M[F, id]
id T' E' $	id + id * id $	Match id
T' E' $	+ id * id $	
ε E' $	+ id * id $	Replace T' by its production $T' \rightarrow \varepsilon$ in M[T', +]
E' $	+ id * id $	
+ T E' $	+ id * id $	Replace E' by its production $E' \rightarrow +TE'$ in M[E', +]
+ T E' $	+ id * id $	Match +
T E' $	id * id $	

FT'E'$	id * id $	Replace T by its production T → FT' in M[T, id]
id T'E'$	id * id $	Replace F by its production F → id in M[F, id]
id T'E'$	id * id $	Match id
T'E'$	* id $	
* FT'E'$	* id $	Replace T' by its production T' → *FT' in M[T', *]
* FT E'$	* id $	Match *
FT' E' $	id $	
id T' E' $	id $	Replace F by its production F → id in M[F, id]
id T' E' $	id $	Match id
T' E' $	$	
ε E' $	$	Replace F by its production T → ε in M[T, $]
E' $	$	
ε $	$	Replace E' by its production E' → ε in M[E', $]
$	$	Successful parsing i.e accept the string.

BOTTOM - UP PARSING

* Bottom - up parsers construct parse trees starting from the leaves and work up to the root.
* Bottom - up syntax analysis is also termed as shift reduce parsing
* The common method of shift - reduce parsing is called LR Parsing.
* Operator precedence parsing is an easy - to - implement shift - reduce parser.



Classification of bottom-up parsing

Handles

A handle of string is a substring that matches the right side of a production and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

ex: Grammar

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

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

abbcde
aAbcde
aAde
aABe
S

These reductions trace out the following rm derivation in reverse.

$S \rightarrow aABe \rightarrow aAde \rightarrow aAbcde \rightarrow abbcde$
(rm, rm, rm)

Handle Pruning

* If $A \rightarrow B$ is a production then reducing B to A by the given production is called handle pruning i.e. removing the children of A from the parse tree.

* A rightmost derivation in reverse can be obtained by handle pruning.

ex: $E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

Right Sentential form	Handle	Reduction production
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Shift - reduce parsing

(i) shift-reduce parsing is a bottom-up parsing that reduces a string w to the start symbol of grammar

(ii) It scans and parses the input text in one forward pass without backtracking

* Stack implementation of shift - reduce parsing
  * Locating the substring to be reduced in a right sentential form.
  * Determining what production to choose in case there is more than one productions with that substring on the right side.

(2)

* Implementation of shift reduce parser

- stack is used to hold grammar symbols
- An input buffer is used to hold the string to be parsed.

ex: consider grammar

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

① Shift
② Reduce
③ Accept
④ Reject

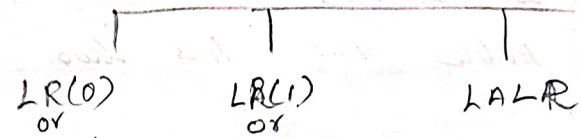
Stack	input	Action
$	$id_1 + id_2 * id_3$ $	shift id
$id_1	$+ id_2 * id_3$ $	reduce by $E \rightarrow id$
$E	$+ id_2 * id_3$ $	shift +
$E+	$id_2 * id_3$ $	shift id
$E+E	$* id_3$ $	shift
$E+E*	id_3 $	shift
$E+E*id_3	$	reduce by $E \rightarrow id$
$E+E*E	$	reduce by $E \rightarrow E*E$
$E+E	$	reduce by $E \rightarrow E+E$
$E	$	accept

* Viable Prefixes

The set of prefixes of right sentential forms that can be appears on the stack of a shift - reduce parser are called Viable prefixes

* Conflicts during shift - reduces parsing

Shift - reduce parsing cannot be used in Context Free Grammar. bottom up Parsers



LR PARSERS

LR Parsers are used to parse the large class of Context Free Grammars.

This technique is called LR(k) parsing

* L is left - to - right scanning of the input
* R is for constructing a rightmost derivation in reverse
* K is the number of input symbols of lookahead that are used in making parsing decisions.

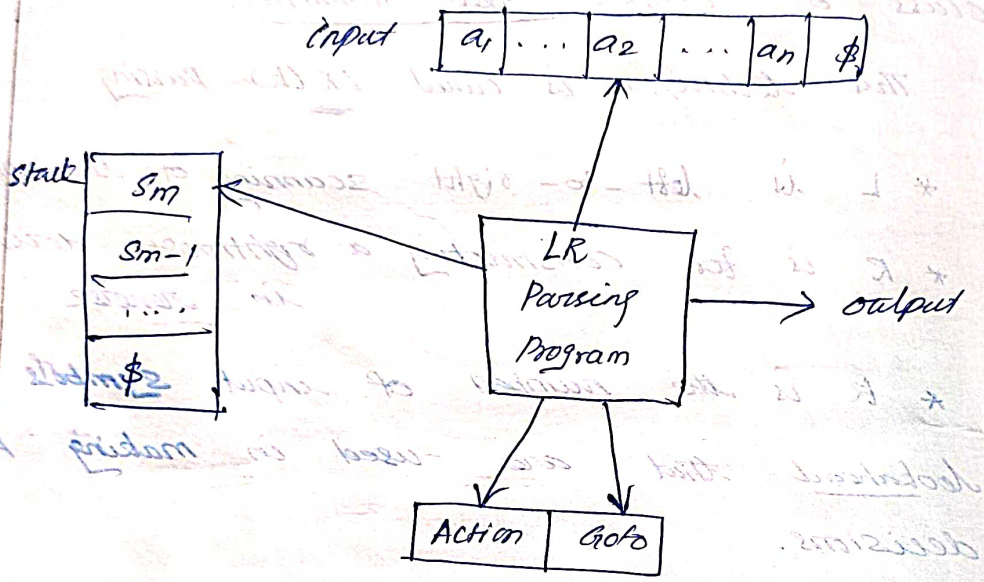
Algorithm an LR Parser

* SLR(1) - Simple LR
  * works on smallest class of grammar.
  * Few number of states hence very small table
  * Simple and Fast Construction
* LR(1) - LR Parser
  * It called Canonical LR Parser
  * works on Complete set of LR(1) Grammar.
  * Generates large table and large number of states
  * Slow construction
* LALR(1) - Lookahead LR Parser
  * Works on intermediate size of grammar
  * Number of states are same as in SLR(1).

Model of LR Parser

LR parser consists of an input, an output, a stack, a driver program and a parsing table that has two functions

1. ACTION
2. GOTO



Model of an LR Parser

ACTION

This function takes as arguments a state i and a terminal a (or $, the input end marker). The value of ACTION[i, a] can have one of the four forms:

(i) shift j, where j is a state
(ii) reduce by a grammar production $A \rightarrow \beta$
(iii) accept
(iv) error

GOTO

This function takes a state and grammar symbol as arguments and produces a state. If GOTO[I_i, A] = I_j, the GOTO also maps a state i and non-terminal A to state j.

* LR Parsing Algorithm

(Refer book 164 Yescee publication)

* LR(0) Items

An LR(0) item of a Grammar G is a production of G with a dot at some position of the body.

ex. $A \rightarrow .XYZ$

$A \rightarrow X.YZ$

$A \rightarrow XY.Z$

$A \rightarrow XYZ.$

One collection of set of LR(0) items. Called Canonical LR(0) collection, provides finite automaton that is used to make parsing decisions.

Such an automation is called an LR(0) automaton.

* LR(0) parser / SLR(1) parser

An LR(0) parser is a shift-reduce parser that uses zero tokens of lookahead to determine what action to take (hence the 0.)

This means that in any configuration of the parser,

The parser must have an unambiguous action to choose - either it shifts a specific symbol or applies a specific reduction.

If there are ever two or more choices to make, the parser fails and the Grammar is not LR(0).

Example:

Construct LR(0) item for the grammar

LR(0) or SLR(1)

$E \to E + T$

$E \to T$

$T \to id$

$T \to (E)$

Follow(E) = {+, $}

Follow(T) = {+, $}

Solution:

Step 1: Number the productions

1. $E \to E + T$
2. $E \to T$
3. $T \to id$
4. $T \to (E)$

Step 2: construct augmented grammar by introducing a new start symbol E' and a new start production $E' \to E$

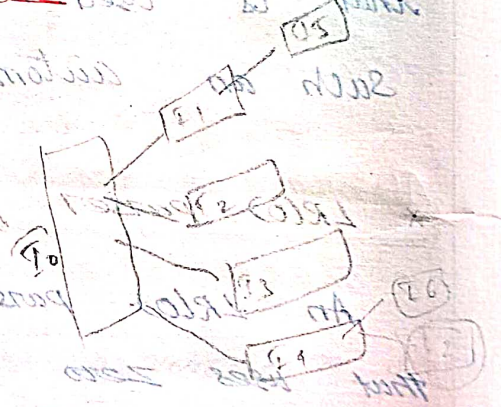
$E' \to E$

$E \to E + T$

$E \to T$

$T \to id$

$T \to (E)$



Step 3: Find closure $(E' \to \bullet E)$

I_0	$E' \to \bullet E$ $E \to \bullet E + T$ $E \to \bullet T$ $T \to \bullet id$ $T \to \bullet (E)$	List down symbol next to dot in each production $V = \{E, T\}$ $T = \{id, (\}$

Step 4 : Find goto for itemset I_0

goto (I_0, E)	$E' \to E\bullet$ $E \to E\bullet + T$	I_1
goto (I_0, T)	$E \to T\bullet$	I_2
goto (I_0, id)	$T \to id\bullet$	I_3
goto $(I_0, ()$	$T \to (\bullet E)$ $E \to \bullet E + T$ $E \to \bullet T$ $T \to \bullet id$ $T \to \bullet (E)$	I_4

Step 5 : Find goto for Itemset I_1

goto $(I_1, +)$	$E \to E + \bullet T$ $T \to \bullet id$ $T \to \bullet (E)$	I_5

Step 6 Find goto for itemset I_4

goto (I_4, E)	$T \to (E\bullet)$ $E \to E\bullet + T$	I_6
goto (I_4, T)	$E \to T\bullet$	I_2
goto (I_4, id)	$T \to id\bullet$	I_3
goto $(I_4, ()$	$T \to (\bullet E)$ $E \to \bullet E + T$ $E \to \bullet T$ $T \to \bullet id$ $T \to \bullet (E)$	I_4

Step 7 Find goto for itemset I_5

goto (I_5, T)	$E \to E+T\bullet$	I_7
goto (I_5, id)	$T \to id\bullet$	I_3
goto $(I_5, ()$	$T \to (\bullet E)$ $E \to \bullet E+T$ $E \to \bullet T$ $T \to \bullet id$ $T \to \bullet (E)$	I_4

Step 8: Find goto for itemset I_6

goto $(I_6,))$	$T \to (E)\bullet$	I_8
goto $(I_6, +)$	$E \to E+\bullet T$ $T \to \bullet id$ $T \to \bullet (E)$	I_5

Step 9: Find goto for itemset I_6

goto $(I_6,))$	$T \to (E)\bullet$	I_8
goto $(I_6, +)$	$E \to E+\bullet T$ $T \to \bullet id$ $T \to \bullet (E)$	I_5

Step 9: The LR(0) table of Grammar G1

* For a shift, the token to be shifted determines the next state.
* For a reduce, the state on top of stack specifies the production to be used.

State	Action					Goto	
	+	id	(	)	$	E	T
0		s_2	s_4			1	2
1	s_5				accept		
2	r_2			r_2	r_2		
3	r_3			r_3	r_3		
4		s_3	s_4			6	2
5		s_3	s_4				7
6	s_5			s_8			
7	r_1			r_1	r_1		
8	r_4			r_4	r_4		

Non-terminals

goto $(I_0, E) = I_1$ ∴ goto $(0, E) = 1$

goto $(I_0, T) = I_2$ ∴ goto $(0, T) = 2$

goto $(I_4, E) = I_6$ ∴ goto $(4, E) = 6$

goto $(I_4, T) = I_2$ ∴ goto $(4, T) = 2$

goto $(I_5, T) = I_7$ ∴ goto $(5, T) = 7$

$(I_1, E) = E \to E. \to I_1 \to \$$

$(I_2, T) = E \to T. \to I_2 \to$ FOLLOW $(E) = \{\$, +,)\}$

reduce r_1 = follow (E) = follow (T)

Terminals

goto $(I_0, id) = I_3$ ∴ action $(0, id) = s_3$

goto $(I_0, ( ) = I_4$ ∴ action $(0, ( ) = s_4$

goto $(I_1, +) = I_5$ ∴ action $(1, +) = s_5$

goto $(I_4, id) = I_3$ ∴ action $(4, id) = s_3$

goto $(I_4, ( ) = I_4$ ∴ action $(4, ( ) = s_4$

goto $(I_5, id) = I_3$ ∴ action $(5, id) = s_3$

goto $(I_5, ( ) = I_4$ ∴ action $(5, ( ) = s_4$

goto $(I_6, +) = I_5$ ∴ action $(6, +) = s_5$

goto $(I_6,)) = I_8$ ∴ action $(6,)) = s_8$

Computation of FOLLOW

For the production

$$E \rightarrow \varepsilon\ T\ \varepsilon$$
$$\downarrow \quad\ \downarrow\ \downarrow\ \downarrow$$
$$A \quad\ \alpha\ B\ \beta$$

Since FIRST(B) contains ε, FOLLOW(B) = FOLLOW(A)

i.e, FOLLOW(T) = FOLLOW(E) = {$, +,)}

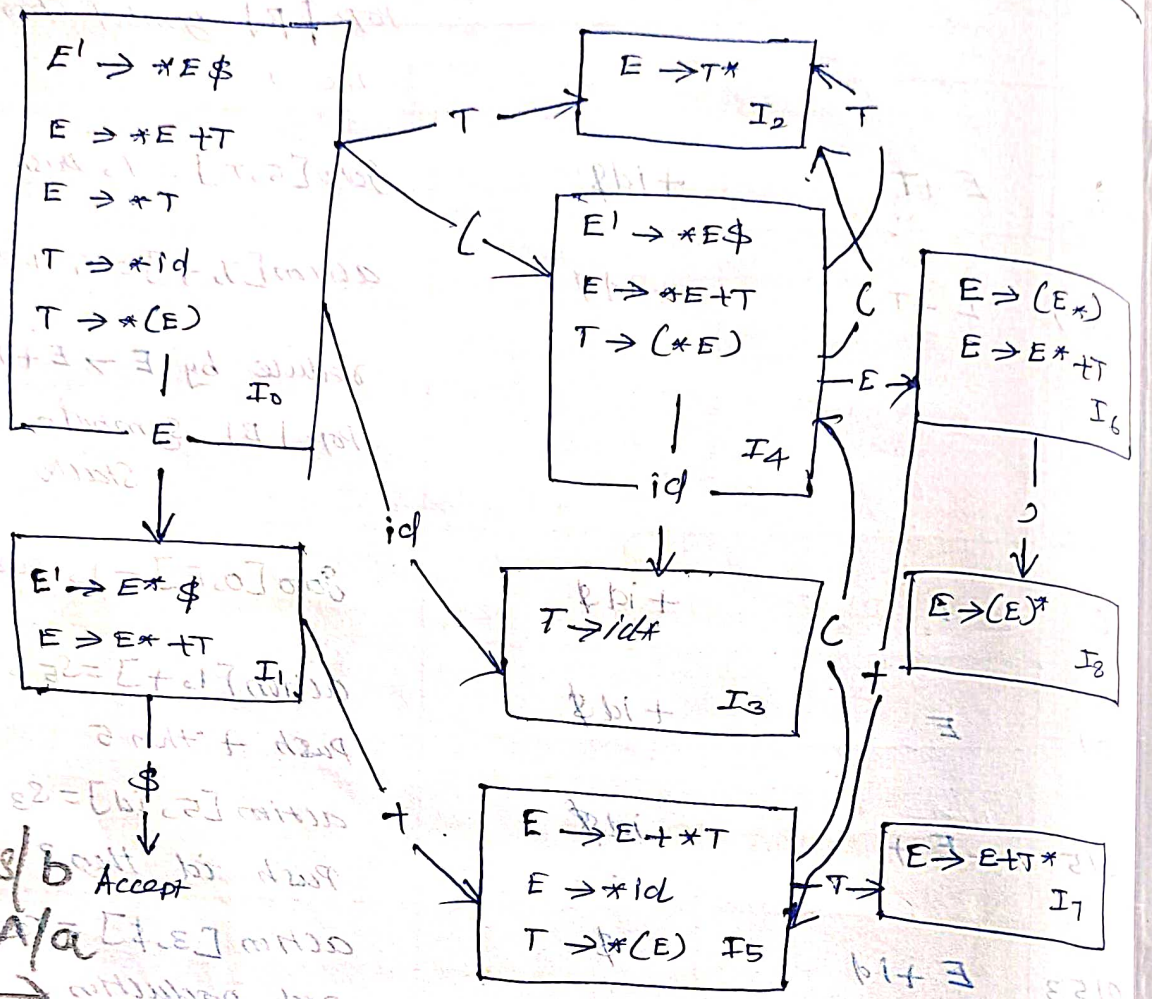
Computing FOLLOW(T)

Step 1:- parser table on input id+id+id

Stack	symbols	input	Action
0	0	id+id+id$	action[0,id] = S3, shift 3, Push id then 3
03	0id3	+id+id$	action[3,+] = r3, ie 3rd production reduce by T→id, Pop |β| symbol from stack i.e 1.
0	0T	+id+id$	goto[0,T] = 2
02	0T2	+id+id$	action[2,+] = r2, ie 2nd production reduce by E→T, Pop |β| symbol from stack i.e 1.
0	0E	+id+id$	goto[0,E] = 1, push 1
01	0E1	+id+id$	action[1,+] = S5, shift 5, Push + then 5
015	0E1+5	id+id$	action[5,id] = S3 shift 3, Push id then 3

0153	E+id 0E+1015 3	+id$	action[3, +] = r_3, i.e, 3rd production reduce by T → id Pop |B| symbol from stack i.e 1
015	E+T 0E+1 15	+id$	goto[5, T] = 7, push 7
0157	E+T 0E+T157	+id$	action[7, +] = r_1 i.e 1st production reduce by E → E+T Pop |B| symbols from stack i.e 3
0	E 0E1	+id$	goto[0, E] = 1, push 1
01	E 0E1	+id$	action[1, +] = S_5, shift 5 push + then 5
015	E+ 0E+1	id$	action[5, id] = S_3, shift 3, Push id then 3
0153	E+id	$	action[3, $] = r_3, i.e 3rd production, reduce by T → id Pop |B| symbol from stack i.e 1
015	E+T	$	goto[5, T] = 7, push 7
0157	E+T	$	action[7, $] = r_1, i.e. 1st production reduce by E → E+T Pop |B| symbols from stack i.e., 3
0	E	$	goto[0, E] = 1, push 1
01	E	$	action[1, $] = accept Parser accept the input and halts.

Step II: Transition diagram of DFA for the Grammar



Hw

S → A$ / b

A → SA / a

abab = ω

LR(1) Parser / canonical LR (CLR)

* Even more powerful than SLR(1) is the LR(1) Parsing method.
* LR(1) includes LR(0) items and a look ahead token is itemsets
* An LR(1) item consists
  - → Grammar production rule
  - → Right – hand position represented by the dot
  - → Lookahead token.
* An LR(1) state is a set of LR(1) items.

Example

construct LR(1) items for the Grammar

$$S \to CC$$
$$C \to cC \mid d$$

Unit - 2
→ LR(0)
→ SLR
→ CLR
→ LALR

Solution:

Step 1: Number the productions

1. $S \to CC$
2. $C \to cC$
3. $C \to d$

Step 2: Construct augmented Grammar by introducing a production of the form

$S' \to S$ where S is the start symbol

$S' \to S$
$S \to CC$
$C \to cC$
$C \to d$

Step 3: Find Closure $(S' \to \bullet S, \$)$, $S, \$$

I_0

$S' \to \bullet S, \$$
$S \to \bullet CC, \$$
$C \to \bullet cC, c/d$
$C \to \bullet d, c/d$

List down symbol next to dot in each production

$V = \{S, C\}$
$T = \{c, d\}$

Note:

If a production is of the form $A \to \alpha \bullet B\beta, a$ then its lookahead is computed by FIRST (βa).

$S' \to \varepsilon \bullet S, \varepsilon, \$$

$A \quad \alpha \quad B \quad \beta \quad a$

FIRST (βa) = FIRST $(\varepsilon \$) = \{\$\}$

(14)

Production $C \to \cdot cC, \$$

$S \to \cdot CC, \$$

$A \quad \alpha \quad B \quad B \quad a$

FIRST (Ba) = FIRST $(C\$)$ = FIRST (C) = $\{c, d\}$

Production $C \to \cdot d$

$S \to \cdot CC, \$$

$A \quad \alpha \quad B \quad B \quad a$

FIRST (Ba) = FIRST $(C\$)$ = FIRST (C) = $\{c, d\}$

Step 4: Find goto for itemset I_0

goto (I_0, S)	$S' \to S \cdot, \$$	I_1
goto (I_0, C)	$S \to C \cdot C, \$$ $C \to \cdot cC, \$$ $C \to \cdot d, \$$	I_2 List down symbols next to dot in each production $V = \{c\}$ $T = \{c, d\}$
goto (I_0, c)	$C \to c \cdot C, c/d$ $C \to \cdot cC, c/d$ $C \to \cdot d, c/d$	I_3 List down symbols next to dot in each production $V = \{c\}$ $T = \{c, d\}$
goto (I_0, d)	$C \to d \cdot, c/d$	I_4

Step 5: Find goto for itemset I_2

goto (I_2, C)	$S \to CC \cdot, \$$	I_5
goto (I_2, c)	$C \to c \cdot C, \$$ $C \to \cdot cC, \$$ $C \to \cdot d, \$$	I_6
goto (I_2, d)	$C \to d \cdot, \$$	I_7

Step 6: Find goto for itemSets I_3

goto (I_3, C)	$C \to cC_{\bullet}$, c/d	I_8
goto (I_3, c)	$C \to c_{\bullet}C$, c/d $C \to \bullet cC$ c/d $C \to \bullet d$, c/d	I_3
goto (I_3, d)	$C \to d_{\bullet}$, c/d	I_4

Step 7: Find goto for item set I_6

goto(I_6, c)	$C \to cC_{\bullet}$, $	I_9
goto(I_6, c)	$C \to c \bullet C$, $ $C \to \bullet cC$, $ $C \to \bullet d$, $	I_6
goto(I_6, d)	$C \to d_{\bullet}$, $	I_7

Step 8: All itemsets are processed, i.e, no more new item set.

Step 9: Construct parsing table

States	Action			Goto	
	c	d	$	S	C
0	S_3	S_4		1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

Non - terminals

goto $(I_0, S) = I_1$ ∴ goto (0, S) = 1

goto $(I_0, C) = I_2$ ∴ goto (0, C) = 2

goto $(I_2, C) = I_5$ ∴ goto (2, C) = 5

goto $(I_3, C) = I_8$ ∴ goto (3, C) = 8

goto $(I_6, C) = I_9$ ∴ goto (6, C) = 9

Terminals

goto $(I_0, C) = I_3$ ∴ action (0, C) = S_3

goto $(I_0, d) = I_4$ ∴ action (0, d) = S_4

goto $(I_2, C) = I_6$ ∴ action (2, C) = S_6

goto $(I_2, d) = I_7$ ∴ action (2, d) = S_7

goto $(I_3, C) = I_3$ ∴ action (3, C) = S_3

goto $(I_3, d) = I_4$ ∴ action (3, d) = S_4

goto $(I_6, C) = I_6$ ∴ action (6, C) = S_6

goto $(I_6, d) = I_7$ ∴ action (6, d) = S_7

S → AS/b
A → SA/a

HW ①
S → SS
S → a
S → ε

②
S → AA
A → Aa/b

S → OS1/01

③

INTRODUCTION TO LALR PARSER

* LALR stands for lookahead LR parsers
* This is the extension of LR(0) items by introducing the one symbols of lookahead on the input.
* It supports large class of grammars.
* The number of states in LALR parsers is lesser than that of LR(1) parsers.
* LALR is preferable as it can be used with reduce memory
* Most syntactic constructs of programming language

Steps to construct LALR parsing table

* Generates LR(1) items
* Find the items that have same set first component (core) and merge those sets into one.
* Merge the goto's of combined itemset
* Revise the parsing table of LR(1) parser by replacing states and goto's with combined states and combined goto's respectively.

ex: For the grammar

1. $S \to CC$
2. $C \to cC \mid d$

Solution:

Step 1: Number the productions

1. $S \to CC$
2. $C \to cC$
3. $C \to d$

Step 2: Construct augmented grammar by introducing a production of the form $S' \to S$ where S' is the start symbol

$S' \to S$
$S \to CC$
$C \to cC$
$C \to d$

Step 3: Find closure $(S' \to \cdot S, \$)$

I_0	$S' \to \cdot S, \$$ $S \to \cdot CC, \$$ $C \to \cdot cC, c\|d$ $C \to \cdot d, c\|d$	List closure symbols next to dot in each production $V = \{S, C\}$ $T = \{c, d\}$

Production $S \rightarrow \cdot CC$

$S' \rightarrow \cdot S \;,\; \$$

$A \;\; \alpha \;\; B \;\; \beta \;\; a$

$FIRST(B a) = FIRST(C \$) = \{\$\}$

Production $C \rightarrow \cdot cC$

$S \rightarrow \cdot C \;\; C \;,\; \$$

$A \;\; \alpha \;\; B \;\; \beta \;\; a$

$FIRST(B a) = FIRST(C \$) = FIRST(c) = \{c, d\}$

Production $C \rightarrow \cdot d$

$S \rightarrow \cdot C \;\; C \;,\; \$$

$A \;\; \alpha \;\; B \;\; \beta \;\; a$

$FIRST(B a) = FIRST(C \$) = FIRST(c) = \{c, d\}$

Step 4: Find goto for itemsets I_0

goto (I_0, S)	$S' \rightarrow S \cdot , \$$	I_1
goto (I_0, C)	$S \rightarrow C \cdot C , \$$ $C \rightarrow \cdot cC , \$$ $C \rightarrow \cdot d , \$$	I_2 List down symbols next to dot in each production $V = \{C\}$ $T = \{c, d\}$
goto (I_0, c)	$C \rightarrow c \cdot C , c \mid d$ $C \rightarrow \cdot cC , c \mid d$ $C \rightarrow \cdot d , c \mid d$	I_3 " $V = \{C\}$ $T = \{c, d\}$
goto (I_0, d)	$C \rightarrow d \cdot , c \mid d$	I_4

Step 5 : Find goto for itemSet I_2

goto (I_2, c)	$C \to cC\bullet, \$$	I_5
goto (I_2, c)	$C \to c\bullet C, \$$ $C \to \bullet cC, \$$ $C \to \bullet d, \$$	I_6
goto (I_2, d)	$C \to d\bullet, \$$	I_7

Step 6 : Find goto for itemset I_3

goto (I_3, c)	$C \to cC\bullet, cld$	I_8
goto (I_3, c)	$C \to c\bullet C, cld$ $C \to \bullet cC, cld$ $C \to \bullet d, cld$	I_3
goto (I_3, d)	$C \to d\bullet, cld$	I_4

Step 7 : Find goto for itemsets I_6

goto (I_6, c)	$C \to cC\bullet, \$$	I_9
goto (I_6, c)	$C \to c\bullet C, \$$ $C \to \bullet cC, \$$ $C \to \bullet d, \$$	I_6
goto (I_6, d)	$C \to d\bullet, \$$	I_7

Step 8 : All itemsets are processed, no more new itemsets

Step 9: Construct parsing table

States	Action			Goto	
	c	d	$	S	C
0	S_3	S_4		1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

Non - terminals

goto $(I_0, S) = I_1$ ∴ goto $(0, S) = 1$

goto $(I_0, C) = I_2$ ∴ goto $(0, C) = 2$

goto $(I_2, C) = I_5$ ∴ goto $(2, C) = 5$

goto $(I_3, C) = I_8$ ∴ goto $(3, C) = 8$

goto $(I_6, C) = I_9$ ∴ goto $(6, C) = 9$

Terminals

goto $(I_0, c) = I_3$ ∴ action $(0, c) = S_3$

goto $(I_0, d) = I_4$ ∴ action $(0, d) = S_4$

goto $(I_2, c) = I_6$ ∴ action $(2, c) = S_6$

goto $(I_2, d) = I_7$ ∴ action $(2, d) = S_7$

goto $(I_3, c) = I_3$ ∴ action $(3, c) = S_3$

goto $(I_3, d) = I_4$ ∴ action $(3, d) = S_4$

goto $(I_6, c) = I_6$ ∴ action $(6, c) = S_6$

goto $(I_6, d) = I_7$ ∴ action $(6, d) = S_7$

In LR(1) items

ItemSets I_3 and I_6 have same core with different lookaheads, so merge I_3 and I_6, to form I_{36}.

ItemSets I_4 and I_7 have same core with different lookaheads. So merge I_4 and I_7, to form I_{47}.

ItemSet I_8 and I_9 have same core with different lookaheads so merge I_8 and I_9. to form I_{89}.

same core, different lookaheads

I_8: core $C \to c \cdot C, cld$ lookaheads $C \to \cdot cC, cld$ $C \to \cdot d, cld$	I_6: $C \to c \cdot C, \$$ $C \to \cdot cC, \$$ $C \to \cdot d, \$$	I_{36}: $C \to c \cdot C, cld \mid \$$ $C \to \cdot cC, cld \mid \$$ $C \to \cdot d, cld \mid \$$
I_4: $C \to d \cdot, cld$	I_7: $C \to d \cdot, \$$	I_{47}: $C \to d \cdot, cld \mid \$$
I_8: $C \to cC \cdot, cld$	I_9: $C \to cC \cdot, \$$	I_{89}: $C \to cC \cdot, cld \mid \$$

Parsing table

States	Action			Goto	
	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		

OPERATOR PRECEDENCE PARSER

Operator Grammar

- No ε in its right side of production
- No two adjacent non-terminals.

Operator precedence parsers can parse only small class of grammars.

Operator precedence relations

Operator precedence relation describes relation between pair of terminals which guides the selection of handles.

(i) Associativity and precedences of operators are found

(ii) unambiguous grammar is constructed for the language that reflects correct associativity and precedence in its parse tree.

operator precedence relation and its meaning

Relation	Meaning
$<.$	a yields precedence to b
$>.$	a takes precedence over b
$=$	a has same precedence as b

Algorithm as follows page no 123:

Precedence Functions.

Precedence functions are used to map terminals to integers.

It eliminates the need for storing the table of precedence relations by encoding into functions.

For symbols a and b

* $f(a) < g(b)$ whenever $a <. b$
* $f(a) > g(b)$ whenever $a .> b$
* $f(a) = g(b)$ whenever $a = b$

ex:

$E \rightarrow E+E \mid E*E \mid id$

With input string id+id*id

Step 1:

Construct operator precedence relation table

	id	+	*	$
id	–	.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	–

Note:

- Always id takes precedence over other terminals
- $ yields precedence to other terminals
- Other terminals have precedence relations based on associativity and precedence.

Step 2: Parsing table

Stack	input	Action
$	id+id*id$	a=$, b=id => a<.b so Push b(id)
id$	+id*id$	a=id, b=+ => a.> b so Pop from stack

$	+id*id$	Compare element in Stack and the most recently Popped element => $ <. id , so Stop Pop
$	+id*id$	a = $, b = + => a <. b so Push b (+)
+$	id*id$	a = +, b = id => a <. b so Push b (id)
id+$	*id$	a = id, b = * => a .> b so Pop from Stack (id)
+$	*id$	a = +, b = id => a <. b Stop Pop
+$	*id$	a = +, b = * => a <. b Push b (*)
*+$	id$	a = *, b = id => a <. b Push b (id)
id*+$	$	a = id, b = $ => a .> b Pop from Stack (id)
*+$	$	a = *, b = id => a <. b Stop Pop
*+$	$	a = *, b = $ => a .> b Pop from Stack (*)
+$	$	a = +, b = * => a <. b Stop Pop
+$	$	a = +, b = $ => a .> b Pop From Stack (+)
$	$	a = $, b = + => a <. b Stop Pop
$	$	a = $, b = $ => HALT

For Precedence Function.

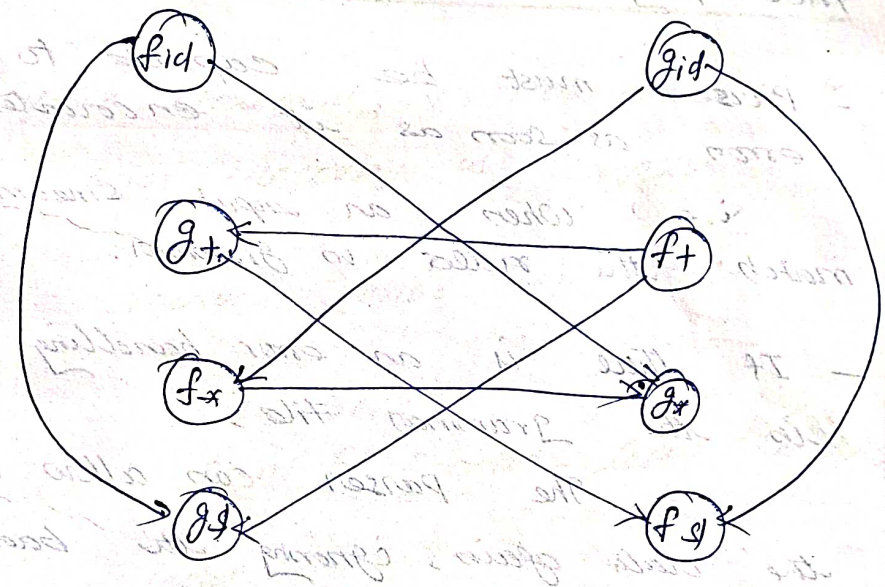
1. Create symbol f_a and g_a for each terminal a, i.e.. f_{id}, f_+, f_*, $f_\$$, g_{id}, g_+, g_* and $g_\$$.

2. Create director graph

3. If there are no cycles So precedence Function exists

4. Precedence functions are given as Follow.

The Path of f_a is given by the Grammer of edges from f_a to either $f_\$$ or $g_\$$.



Path $f_+ = 2 \Rightarrow f_+ \rightarrow g_+ \rightarrow g\$$

Path of $f_* = 4 \Rightarrow f_* \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f\$$

Path of $f_{id} = 4 \Rightarrow f_{id} \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f\$$

Path of $f_\$ = 0 \Rightarrow f\$$

Path of $g_+ = 1 \Rightarrow g_+ \rightarrow f\$$

Path of $g_* = 3 \Rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f\$$

Path of $g_{id} = 5 \Rightarrow g_{id} \rightarrow f_* \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f\$$

Path of $g_\$ = 0 \Rightarrow g\$$

	+	*	id	$
f	2	4	4	0
g	1	3	5	0

Error Handling and Error Recovery in Syntax Analyzer

→ An efficient program should not terminate on an parse error

→ It must recover to parse the rest of the input and check the subsequent errors.

→ For one line input, the routine YYparse() can be made to return 1 on error and then calls YYparse() again.

YACC program error handling

- Parser must be capable to detecting the error as soon as it encounters.

  i.e) when an input stream does not match the rules in grammar.

- If there is an error handling subroutine in the grammar file,

  The parser can allow for entering the data again, ignoring the bad data or initiating a cleanup and recovery action.

- when the parser finds an error, it may need to reclaim parse tree storage, delete or alter symbol table entries and set switches to avoid generating further output.

- Error handling routines are used to restart the parser to continue its process even after the occurrence of error.

- Tokens following the error get discarded to restart the parser.

e.g. stat : error ';'

Providing 80 error correction

The input errors can be corrected by entering a line in the data stream again.

```
input : error '\n'
  {
    printf ("Reenter last line :");
  }
input
  {
  $$ = $4;
  };
```

The YACC statement, yyerrok is used to indicate that error recovery is complete.

```
input : error '\n'
  {
    yyerrok;
    printf ("Reenter last line :");
  }
input
  {
  $$ = $4;
  };
```

Clearing the lookahead token

- When an error occurs, the lookahead token becomes the token at which the error was detected.
- The lookahead token must be changed if the error recovery action includes code to find the correct place to start processing again.

- To clear the lookahead token, the error recovery action discuss the following statement.

YY clearin

To assist in error handling, macros can be placed in YACC action.

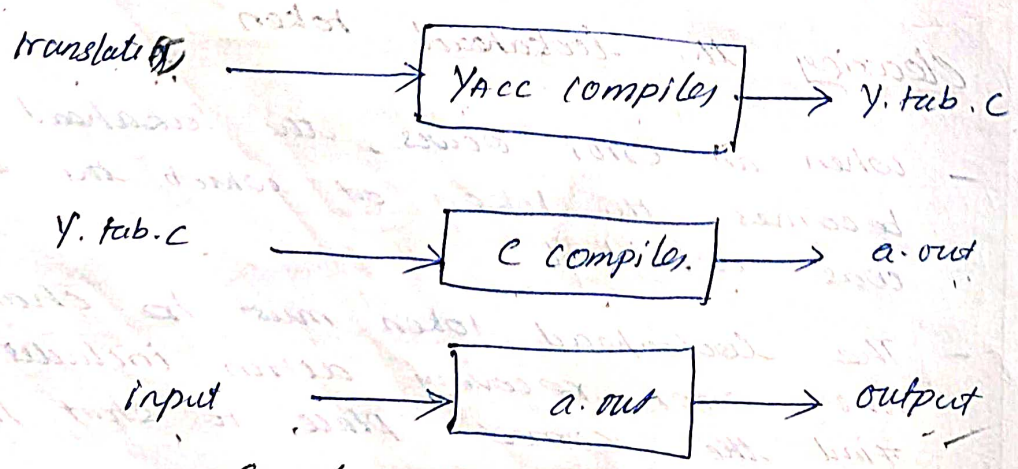
YYERROR	causes the parser to initiate error handling.
YYABORT	Causes the parser to return with a value of 1.
YYACCEPT	causes the parser to return with a value of 0.
YYRECOVERING()	Returns a value of 1 if a syntax error has been detected and the parser has not yet fully recovered.

Macros for error handling.

YACC

* YACC stands for Yet Another Compiler - Compiler
* It is a LALR Parser Generator.

Design of Syntax analyzer / Parser Generator YACC



Creating translator using YACC

- YACC specification of the translator (translate .y) is given to YACC compiler which transform it into y.tab.c that represent LALR parser in C program.

- The transformation of YACC specification into C program is done by the UNIX command "yacc yaccfile",

  ie. yacc translate.y which uses the LALR method.

- It contains a routine yyparse() which returns 0 if the program is correct, non-zero otherwise.

- The object program a.out is obtained by the compilation of y.tab.c with ly library using the command cc y.tab.c -ly.

- It also works with lex. YACC calls yylex() to get next token.

- YACC and lex must agree on the values for each token.

e.g.

```
yyerror(str)
char *str;
{ printf("yyerror : %s at line %d\n", str,
                                  yyline);
}
main()
{
    if (!yyparse())
    {
        printf("accept\n");
    }
    else printf("reject\n");
}
```

YACC file format

declarations

%%

translation rules

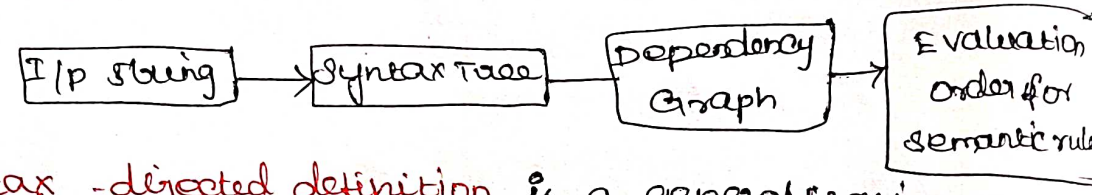
%%

supporting C-routines.

Unit-III Syntax Directed Translation and Intermediate Code Generation

(1) Syntax directed Definitions: -augmented CFG is gra

It is a kind of abstract specification. The conceptual view of syntax-directed translation can be,



Syntax-directed definition is a generalization of context free grammar in which each grammar production $X \to \alpha$ is associated with it a set of semantic rules of the form $a := f(b_1, b_2, \ldots b_k)$ where a is an attribute obtained from the function f.

attribute: can be string, number, type,

$X \to \alpha$ be a CFG

$a := f(b_1, b_2, \ldots b_k)$ where a is attribute,

Two types attribute:

① Synthesized attribute: 'a' is called synthesized attribute of X and $b_1, b_2, \ldots b_k$ are attributes belonging to the production symbols.

The value of synthesized attribute at a node is computed from the values of attributes at the children of that node.

② Inherited attribute: 'a' is called inherited attribute of one of the grammar symbol on the right side of the production (α) & $b_1, b_2, \ldots b_k$ are belonging to either X or α.

value is computed siblings & parent of the

Synthesized attribute:

Eg: $S \to EN$
$E \to E+T$
$E \to E-T$
$E \to T$
$T \to T*F$
$T \to T/F$
$T \to F$
$F \to (E)$
$F \to digit$
$N \to ;$

Sol:

Production rule	Semantic actions
$S \to EN$	print(E.val)
$E \to E_1 + T$	E.val = E_1.val + T.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
$T \to T_1 / F$	T.val := T_1.val / F.val
$T \to F$	T.val = F.val
$F \to (E)$	F.val := E.val
$F \to digit$	F.val := digit.lexval
$N \to ;$	Can be ignored by lexical analyzer as; is terminating symbol.

To compute S-attribute definition:

① Write the syntax directed def. using the appropriate semantic actions for corresponding production rule.

② Annotated parse tree is generated & attribute values are computed. (bottom up manner)

③ The value is obtained at the root node is supposed to be the final o/p.

Eg: Construct parse tree, syntax tree & annotated parse tree for the i/p string is 5*6+7;

Sol:

$F \to digit$

F.val = digit.lexval.

F.val = 5

T.val = T_1.val * F.val

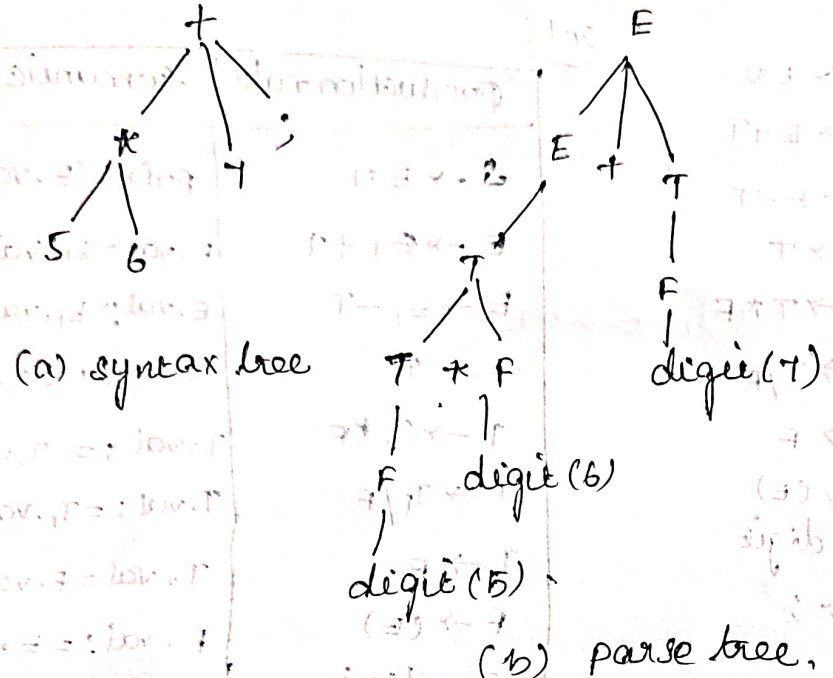
= 5 * 6

T.val = 30

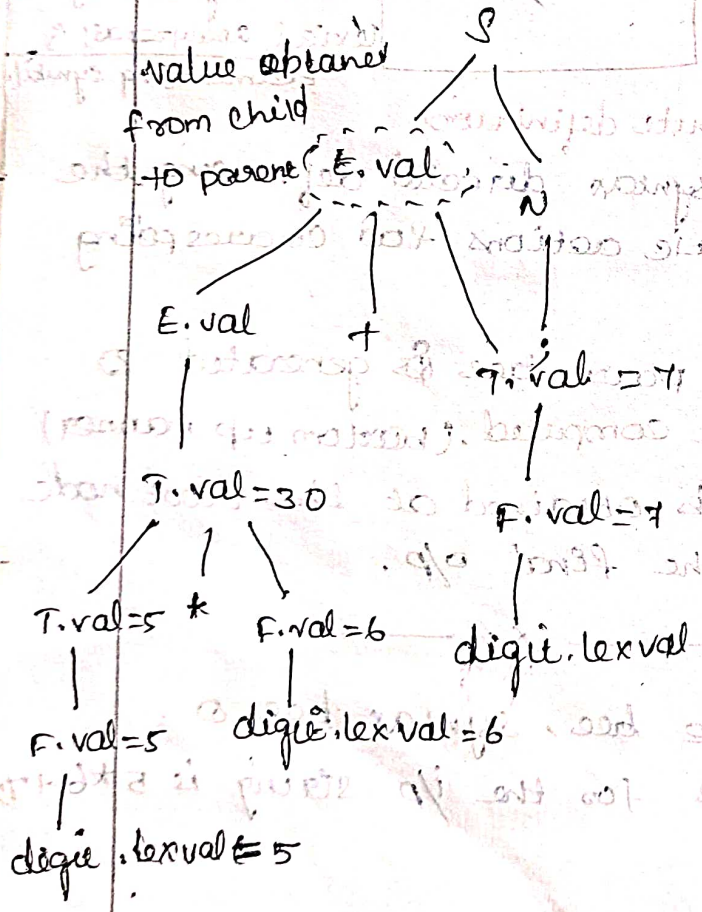
E.val = E_1.val + T.val

= 30 + 7

E.val = 37



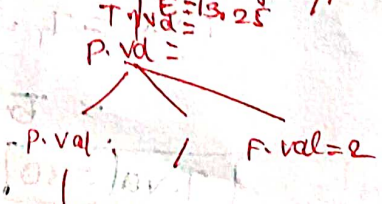
(a) syntax tree (b) parse tree.



(c) Annotated parse tree.

2) Construct a decorated parse tree according to the syntax directed definition for following i/p

Stmt : (4 + 7.5 * 3)/2



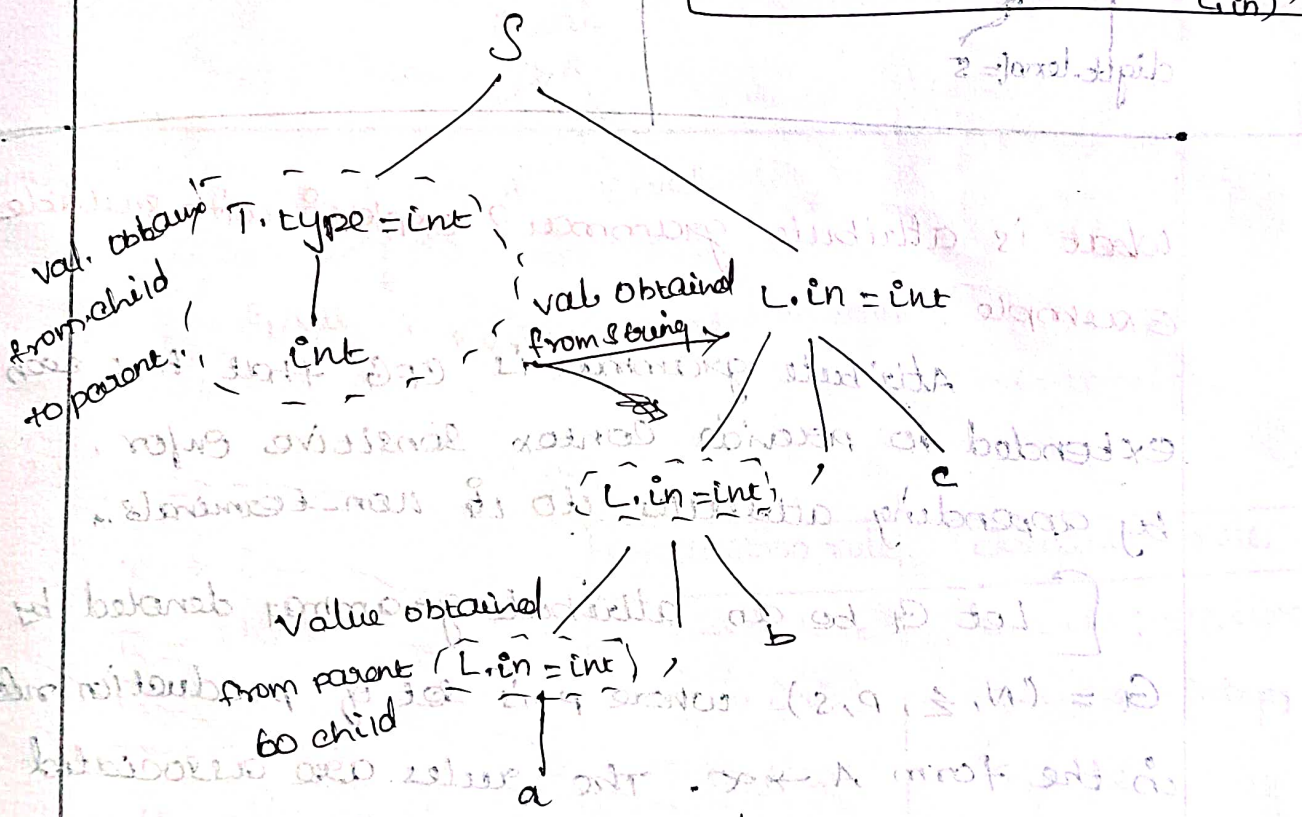
(2) Inherited attribute.

The value of inherited attribute at a node in a parse tree is defined using the attribute values at the parent or siblings.

$S \rightarrow TL$
$T \rightarrow int$
$T \rightarrow float$
$T \rightarrow char$
$T \rightarrow double$
$L \rightarrow L_1, id$
$L \rightarrow id$

Production Rule	Semantic action
$S \rightarrow TL$	L.in = T.type
$T \rightarrow int$	T.type := integer
$T \rightarrow float$	T.type := float
$T \rightarrow char$	T.type := char
$T \rightarrow double$	T.type := double
$L \rightarrow L_1, id$	L_1.in = L.in Enter_type(id.Entry, L.in)
$L \rightarrow id$	Enter_type(id.Entry, L.in)

Annotated parse tree.

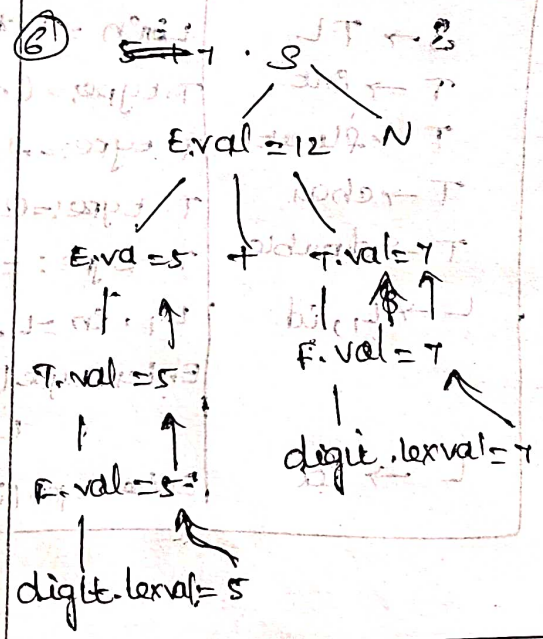


Synthesized translation	Inherited Translation
① Definition	① Definition.
② Syn. attributed are computed	② Inherited attribute are Computed.
③ using values of children	③ " using values of parent & siblings node

(4) Info. is passed up in syntax tree

(5) Simple to compute the value

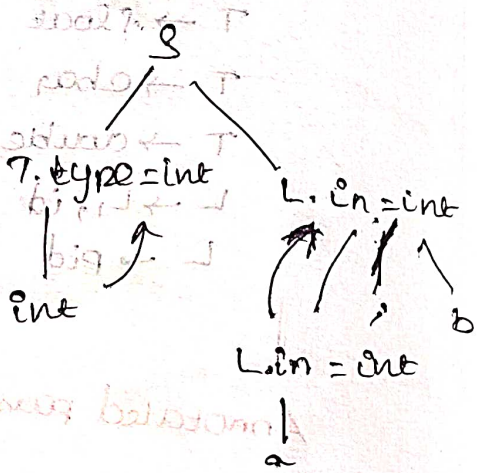
(6)



(4) Info is passed down in syntax tree.

(5) Complex to Compute the value.

(6) eg:



What is attribute grammar? Explain with suitable Example?

Attribute grammar is CFG that has been extended to provide context sensitive info. by appending attributes to its non-terminals.

[Let G be an attribute grammar denoted by $G = (N, \Sigma, P, S)$ where P is set of production rules in the form $A \to \alpha$. The rules are associated with set of semantic rules of the form $b := f(a_1, a_2, a_3 \ldots a_n)$ where f denotes] transitionality.

Two types:

(1) synthesized attribute

(2) Inherited attribute.

3) Dependency graph:

A directed graph that represents the interdependencies b/w synthesized & inherited attributes at nodes in the parse tree is called dependency graph.

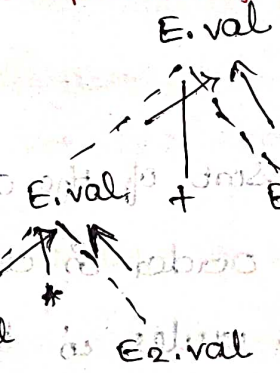
$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

Sol:

production rule	semantic rule
$E \rightarrow E_1 + E_2$	$E.val := E_1.val + E_2.val$
$E \rightarrow E_1 * E_2$	$E.val := E_1.val \times E_2.val$

Dependency Graph.



2)

$S \rightarrow T\ List$

$T \rightarrow int$

$T \rightarrow float$

$T \rightarrow char$

$T \rightarrow double$

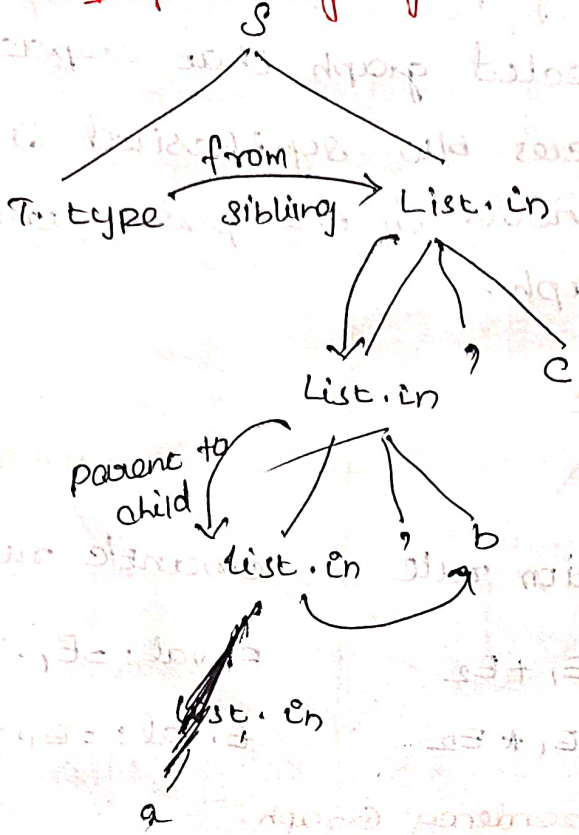
$List \rightarrow List_1, id$

$List \rightarrow id$

Sol:

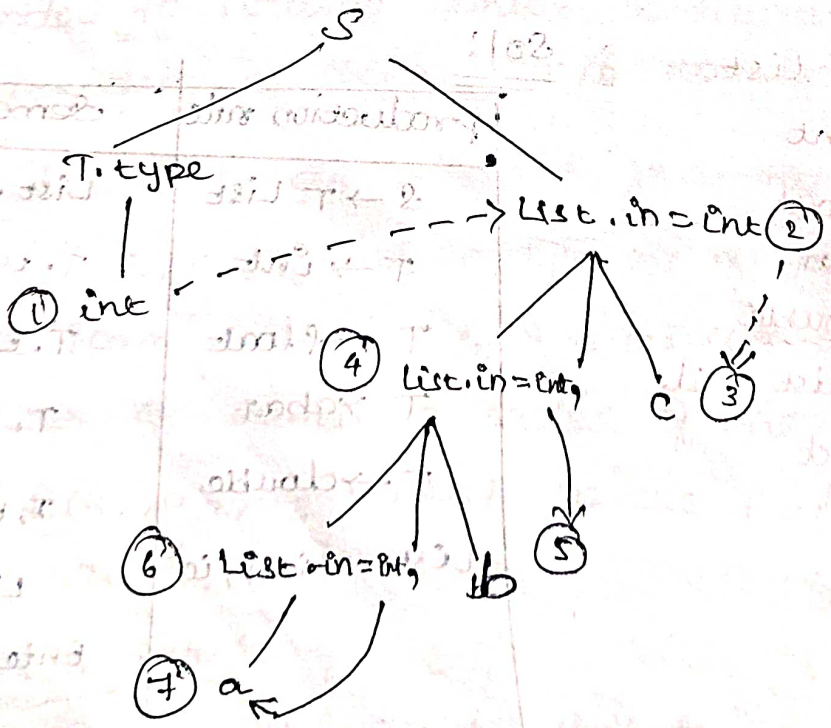
production rule	semantic rule
$S \rightarrow T\ List$	List.in := T.type
$T \rightarrow int$	T.type := integer
$T \rightarrow float$	T.type := float
$T \rightarrow char$	T.type := char
$T \rightarrow double$	T.type := double
$List \rightarrow List_1, id$	$List_1.in := List.in$ Enter_type(id.entry, List.in)
$List \rightarrow id$	Enter_type(id.entry, List.in)

Dependency graph.



4) Evaluation Order:

The topological sort of the dependency graph decides the Evaluation order in a parse tree, ~~synt~~ semantic rules in the syntax-directed defi. are used.



Evaluation order.

(2) Construction of Syntax Tree Expression:

Grammar:

$E \rightarrow E_1 + T$

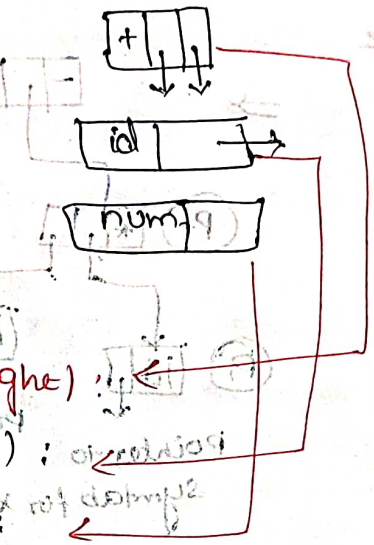
$E \rightarrow E_1 - T$

$E \rightarrow E_1 * T$

$E \rightarrow T$

$T \rightarrow id$

$T \rightarrow num.$



(1) mknode (op, left, right) :

(2) mkleaf (id, entry) :

(3) mkleaf (num, val) :

---

(1) Construct the syntax tree for expression x*y-5+z

Sol:

Step 1: Convert the expression from infix to postfix

x*y-5+z

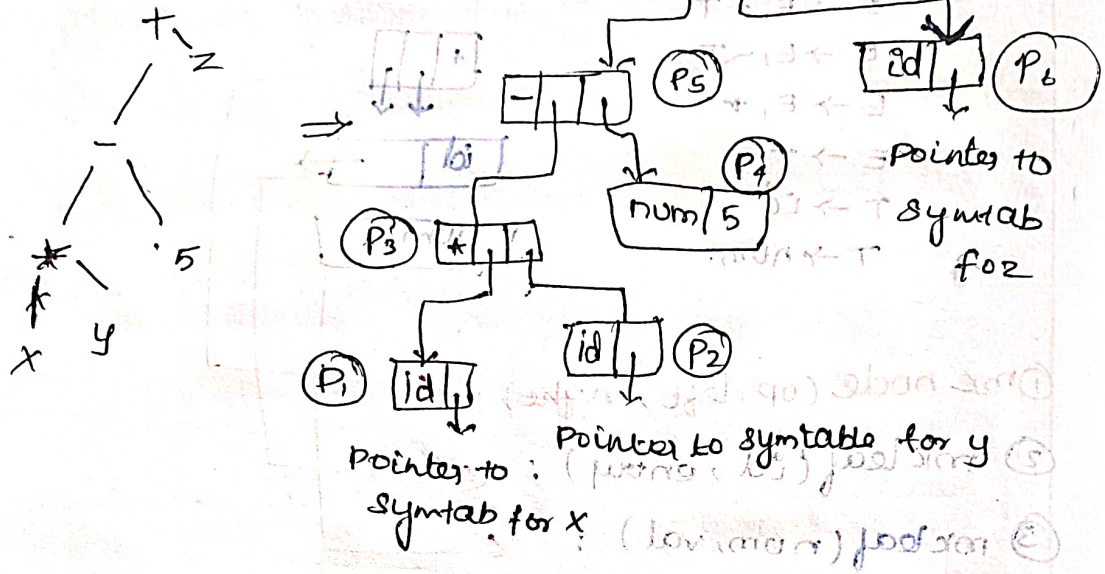
Step 2: Make use of the fun. mknode (), mkleaf (id, ptr) & mkleaf (num, val).

Step 3: The seq. of fun calls is given.

postfix expression xy*5-z+

symbol	operation
x	P_1 = mkleaf (id, ptr to entry x
y	P_2 = mkleaf (id, ptr to entry y)
*	P_3 = mknode (*, P_1, P_2)
5	P_4 = mkleaf (num, 5)
-	P_5 = mknode (-, P_3, P_4)
z	P_6 = mkleaf (id, ptr to entry z)
+	P_7 = mknode (+, P_5, P_6)

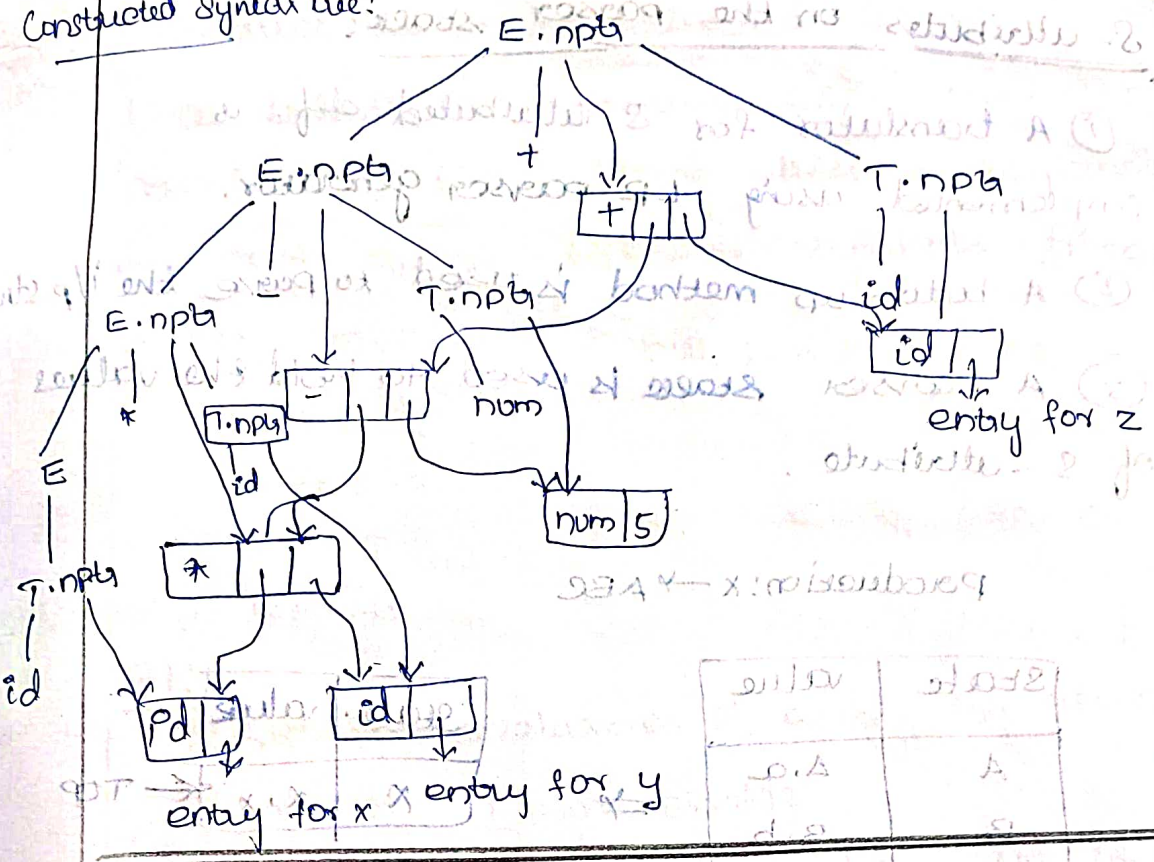
Syntax tree:



Syntax directed definition for grammar

production rule	semantic operation
$E \rightarrow E_1 + T$	E.nptr := mknode('+', E_1.nptr, T.nptr)
$E \rightarrow E_1 - T$	E.nptr = mknode('-', E_1.nptr, T.nptr)
$E \rightarrow E_1 * T$	E.nptr := mknode('*', E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow id$	E.nptr := mkleaf(id, id.ptr_symtab)
$T \rightarrow num$	T.nptr := mkleaf(num, num.val)

Constructed syntax tree:



(3). Bottom up Evaluation of S- Attributed definition:

⇒ Syntax directed translation scheme is implemented for the syntax directed definitions.

⇒ Translator is built.

⇒ S.D.D is very difficult to construct the translator.

(*) S-attributed def. is one such class of syntax directed def. with synthesized attribute only.

(*) Synthesized attributes can be evaluated using bottom up parser.

(*) Synthesized attribute associated with the grammar symbol on its stack. Stack is known as parser stack.

S-attributes on the parser stack:

① A translator for S-attributed def. is implemented using LR parser generator.

② A bottom up method is used to parse the i/p string.

③ A parser stack is used to hold the values of S-attribute.

Production: $X \rightarrow ABC$

state	value
A	A.a
B	B.b
TOP → C	C.c

state	value
X	X.x ← TOP

Before reduction

After reduction

The TOP symbol on the stack is pointed by pointer top

Prod. rule	Sem. action
$X \rightarrow ABC$	$X.x = f(A.a, B.b, C.c)$

④ After reduction top is decremented by 2 the state covering X is placed at the top of state[TOP] & value of S.attribute X.x is put in value[TOP]

⑤ If the symbol has no attribute then the corresponding entry in the value array will be kept undefined.

Eg: Construct a syntax directed translation scheme that translates arithmetic exp. from infix to postfix notation. Using ~~arithmo~~ semantic attrb for each of the grammar symbols & semantic rules, evaluate input: 3 * 4 + 5 * 2.

$E \to E+T$

$E \to E-T$

$E \to T$

$T \to id$

Sol:

SDT for evaluation of expr. for given gramma

$L \to E$ { print (E.val) }

$E \to E+T$ { E.val = E.val + T.val }

$E \to T$ { E.val = T.val }

$T \to T*F$ { T.val = T.val * F.val }

$T \to F$ { T.val = F.val }

$F \to id$ { F.val = id.lexval }

Symbol stack	value stack	I/P string	Syntax action	Semantic rule.
$	$	3*4+5*2$	shift	
$id	$3	*4+5*2$	Reduce F→id	F.val = id.val
$F	$3	*4+5*2$	Reduce T→F	T.val = F.val
$T	$3	*4+5*2$	shift	
$T*	$3*	4+5*2$	shift	
$T*id	$3*4	+5*2$	Reduce F→id	F.val = id.val

$T*F	$12*4	+5*2$	Reduce E→T*F	T.val = T.val * F.val
$T	$12	+5*2$	Reduce E→T	E.val = T.val
$E	$12	+5*2$	Shift	
$E+	$12+	5*2$	Shift	
$E+id	$12+5	*2$	Reduce F→id	F.val = id.val
$E+F	$12+5	*2$	Reduce T→F	T.val = F.val
$E+T	$12+5	*2$	Shift	
$E+T*	$12+5*	2$	Shift	
$E+T*id	$12+5*2	$	Reduce F→id	F.val = id.val
$E+T*F	$12+5*2	$	Reduce T→T*F	T.val = T.val * F.val
$E+T	$12+10	$	Reduce E→E+T	E.val = E.val + T.val
$E	$22	$	Reduce S→E	Print (E.val)

L-attributed definition

SDD can be defined as the L-attributed for the production rule $A \to X_1 X_2, \ldots X_n$ where the inherited attribute X_k is such that $1 \le k \le n$. The production $A \to X_1 X_2 \ldots X_n$ is such that.

① It depends upon the attributes of the symbol $X_1, X_2, \ldots X_{j-1}$ to left of X_j.

② It also depends upon the inherited attribute A.

check whether the given SDD is L-attributed or not.

$A \to PQ$	$P.in := p(A.in)$
	$Q.in := q(P.sy)$
	$A.sy := f(Q.sy)$
$A \to XY$	$Y.in := y(A.in)$
	$X.in := x(Y.sy)$
	$A.sy := f(X.sy)$

Sol:

The attri. in and sy represent the inherited and S-attri respectively. The given SDD is not L-attri.

Prod. Rule	Sem. action	class of attri
$A \to PQ$	$P.in = p(A.in)$	L-attribute
	$Q.in := q(P.sy)$	
	$A.sy := f(Q.sy)$	
$A \to XY$	$Y.in := y(A.in)$	L-attribute
	$X.in := x(Y.sy)$	not-L-attri
	$A.sy := f(X.sy)$	L-attribute.

Definition $X.in := x(Y.sy)$.

Sem. action suggests the value X.in depends upon value Y.sy. violets the 1st rule of L-attri defn.

Thus $X.in := x(Y.sy)$ is not L-attribute.

(10b)

S-attributed grammar	L-attributed grammar.
(1) Grammar having no inherited attri, but only synthesized attri	attri to be evaluated in one left-to-right traversal of abstract syntax tree. Grammar in which inherited attri can be evaluated.
(2) Both Bottomup & Top-down parsing	Incorporated in using top-down parsing
(3) The YACC family can be broadly considered as S-attributed grammar	special type of compiler narrow compilers are based on some of the L-attri. grammar.
(4) S-attri grammar can be L-attri grammar	L-attri grammar can not be S-attri
(5) Eg:	Eg:

* Translation Scheme:

⇒ During the process of parsing the evaluation of attribute takes place by consulting the semantic action enclosed in { } at the right of the grammar symbol. This process of removing code fragement semantic actions from the syntax directed def is called Syntax directed translation. Thus the SDD can be done by syntax directed translation scheme.

⇒ A translation scheme generates the O/P by executing the sem. actions in order naming

⇒ This processing is using depth first traversal

Give the translation scheme that converts infix to postfix form for the following grammar. Also generates the annotated parse tree i/p string 2+6+1.

$E \to E+T$ {print('+')}

$E \to T$

$T \to 0$ {print('0')}

$T \to 1$ {print('1')}

$T \to 2$ {print('2')}

$T \to 3$ {print('3')}

$T \to 4$ {print('4')}

$T \to 5$ {print('5')}

$T \to 6$ {print('6')}

$T \to 7$ {print('7')}

$T \to 8$ {print('8')}

$T \to 9$ {print('9')}

Sol: Let us convert this grammar into non left recursive form.

$E \to TP$

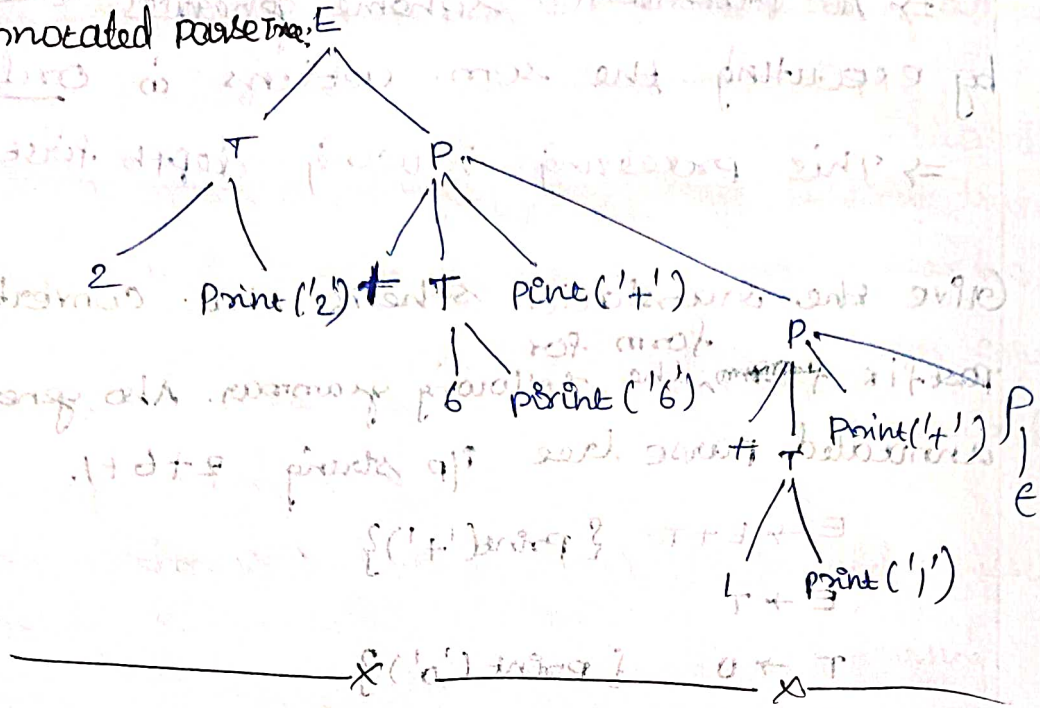
$T \to 0|1|2|3|4|5|6|7|8|9$

$P \to +TP|\epsilon$

Translation scheme for this grammar:

productions	semantic actions.
$E \to TP$	
$T \to 0$	{print('0')}
$T \to 1$	{print('1')}
$T \to 2$	
$T \to 3$	
⋮	⋮
$T \to 9$	{print('9')}
$P \to +TP/\epsilon$	{print('+')} P/ε

Annotated Parse tree:

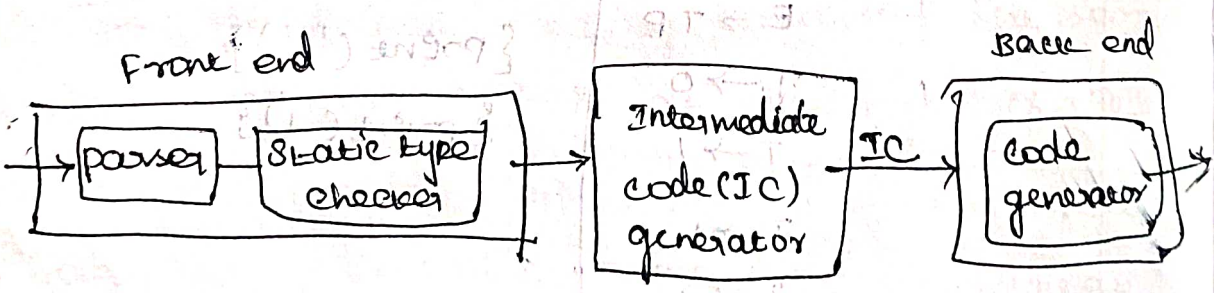


Intermediate languages:

Introduction:

Compiler converts the source pgm into machine pgm. This activity can be done directly, but it is not always possible to generate such machine code directly in one pass.

Then typically compiler generates an easy to represent form a source pg. language which is called Intermediate language the generation of an intermediate language leads to efficient code generation.



Properties:

① The IC is an easy form of source language which can be generated eff. by the compiler

② The gener. of IC should lead to eff. code gener.

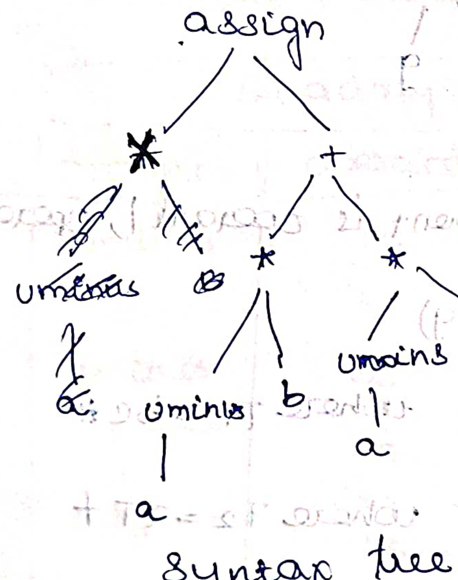
③ The Intermediate language should act as "effective mediator" b/w front & back end.

④ I.L should be flexible enough so that optimized code can be generated.

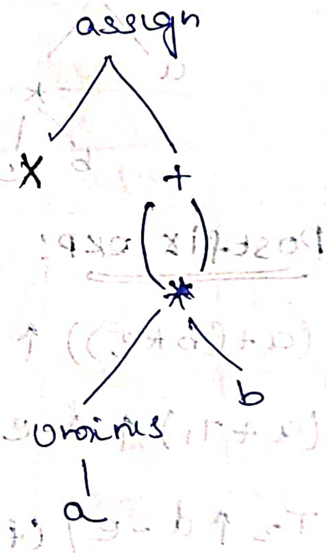
Forms of IC:

Three types:

① Abstract syntax tree: $X := a*b + -a*b$



Syntax tree



DAG

② Polish notation: The polish notation is also called as Prefix notation in which the operator occurs first then operands are arranged.

(a+b) * (c-d)

* +ab -cd

+ ↑ oper; ab → Two operand; cd → Two operands; - → operator.

(10) There is a reverse polish notation which can be using postfix or posix representation.

$X := -a * b + -a * b$

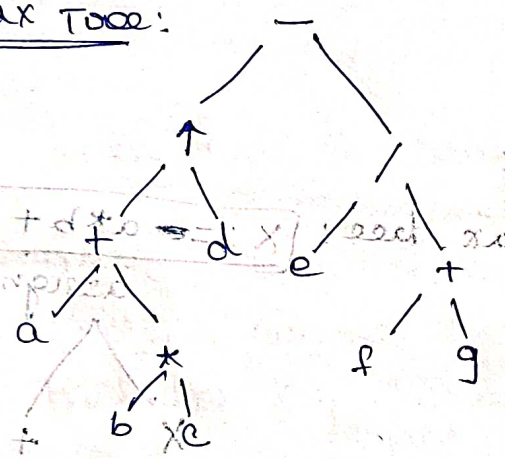
Postfix:

$X a - b * a - b * + ; =$

(1) Construct syntax tree & postfix notation for the following exp.

$(a + (b * c)) \uparrow d - e / (f + g))$

Syntax Tree:



Postfix exp: Arrangement is operand 1, operand 2, operat

$(a + (b * c)) \uparrow d - e / (f + g)$

$(a + T_1) \uparrow d - e / (f + g)$ where $T_1 = bc*$

$T_2 \uparrow d - e / (f + g)$ where $T_2 = aT+$

$T_3 - e / (f + g)$ where $T_3 = T_2 d \uparrow$

$T_3 - e / T_4$ where $T_4 = fg+$

$T_3 - T_5$ where $T_5 = eT_4/$

T_6 where $T_6 = T_3 T_5 -$

backward substituting the values:

T_6

$T_3 T_5 -$

$T_3 e T_4 / -$

$T_3 e f g + / -$

$T_3 = T_2 d \uparrow$

$T_2 d \uparrow e f g + / -$

$a T_1 + d \uparrow e f g + / -$

$a b c * + d \uparrow e f g + / -$

(3) Three address code:

Abstract form of IC that can be implemented as a record with the address fields.

Representations used three address code such as

① quadruples ② Triples ③ Indirect triples.

Quadruple representation:

Four fields - op, arg1, arg2, result.

op - fields is represent internal code for operator.

arg1, arg2 - Two operands used

Result - Result to store of an expr.

I/p statement:

$X := -a * b + -a * b$

Quadruples

	Operator	operand1	operand2	Result	
(0)	uminus	a		t_1	$t_1 := \text{uminus } a$
(1)	*	t_1	b	t_2	$t_2 := t_1 * b$
(2)	uminus	a		t_3	$t_3 := -a$
(3)	*	t_3	b	t_4	$t_4 := t_3 * b$
(4)	+	t_2	t_4	t_5	$t_5 := t_2 + t_4$
(5)	:=	t_5		X	$X := t_5$

Triples:

Temporary variable is avoided by referring the pointers in the symbol table.

$X := -a * b + -a * b.$

Number	Operator	operand1	operand2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	X	(4)

(10)

Indirect triples: listing pointers are used instead of using statements.

Num	operator	operand1	operand2
(0)	uminus	a	
(1)	*	(11)	b
(2)	uminus	a	
(3)	*	(13)	b
(4)	+	(12)	(4)
(5)	:=	x	(15)

Statement	
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)

Quadruple	Triple	Indirect triple
① Format of quadruple is (OP, op1, operand2, result)	(OP, operand1, operand2)	but is makes use of two tables.
② Temporary var. is used from access of symbol table.	pointer is used. To access from symbol table.	Computations is made separately & stored saves some amt of space compared with quadruples.
③ This representation is beneficial for code optimization.	the use of pointer allows to access the symbol table entries quickly.	During the code optimization task of interchanging the instrs. around and deleting the instrs. is involved. Easy for quadruples & difficult for triples.

Eg: $-(a*b)+(c+d)-(a+b+c+d)$

Three address Code:

$t_1 := a * b$
$t_2 := \text{uminus } t_1$
$t_3 := c + d$
$t_4 := t_2 + t_3$
$t_5 := a + b$
$t_6 := t_5 + t_3$
$t_7 := t_4 - t_6$.

Quadruple:

Location	operator	operand1	operand2	Result
(1)	*	a	b	t1
(2)	uminus	t1		t2
(3)	+	c	d	t3
(4)	+	t2	t3	t4
(5)	+	a	b	t5
(6)	+	t5	t3	t6
(7)	-	t4	t6	t7

Triple:

Location	operator	operand1	operand2
(1)	*	a	b
(2)	uminus	(1)	
(3)	+	c	d
(4)	+	(2)	(3)
(5)	+	a	b
(6)	+	(5)	(3)
(7)	-	(4)	(6)

Indirect triple:

Loc.	operator	operand1	operand2
(1)	*	a	b
(2)	uminus	(11)	
(3)	+	c	d
(4)	+	(12)	(13)
(5)	+	a	b
(6)	+	(15)	(13)
(7)	-	(14)	(16)

	Stmt
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)
(7)	(17)

(121)

Types of Three address Code:

Language Constructs	IC form	Meaning
Assig. stmt	x := y op z	Binary operation is performed using operator 'op'
Assig. stmt	x := op y	Unary operation is performed. op is unary operator.
Copy stmt	x := y	value of y is assigned to x.
Unconditional Jump	goto L	The control flow goes to stmt labeled by L.
Conditional Jump	if x relop y goto L	relational operators such as <, =, >=. if x relop is true then it executes goto L stmt.
procedure calls	param x1 param x2 ... param xn call p,n return y	x1, x2, ... xn are used as parameter to the procedure p. return value of y.
Array stmt	x := y[i] x[i] := y	ith index of array y is assigned to x. value of y is assigned to index i of array x
Address & pointer assignment	x := &y x := *y *x := y	address of y stores to x pointer of x stores to x

Declaration:

In declarative stmt the data items along with their data types are declared.

$S \rightarrow D$	{offset := 0}
$D \rightarrow id : T$	{enter_tab(id.name, T.type, offset); {offset := offset + T.width)}
$T \rightarrow$ Integer	{T.type := integer; T.width := 4}
$T \rightarrow$ real	{T.type := real; T.width := 8}
$T \rightarrow$ array[num] of T_1	{T.type := array(num.val, T_1.type) T.width := num.val × T_1.width}
$T \rightarrow *T_1$	{T.type := pointer(T.type) T.width := 4}

$D \rightarrow id.T$ is a declarative stmt for id declaration

---

Assignment Stmt:

Mainly deals with expressions. The exp. can be type of integer, real, array & record.

(1) Obtain the translation scheme for obtaining the three address code for the grammar

$S \rightarrow id := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow -E_1$

$E \rightarrow (E_1)$

$E \rightarrow id$

(11)

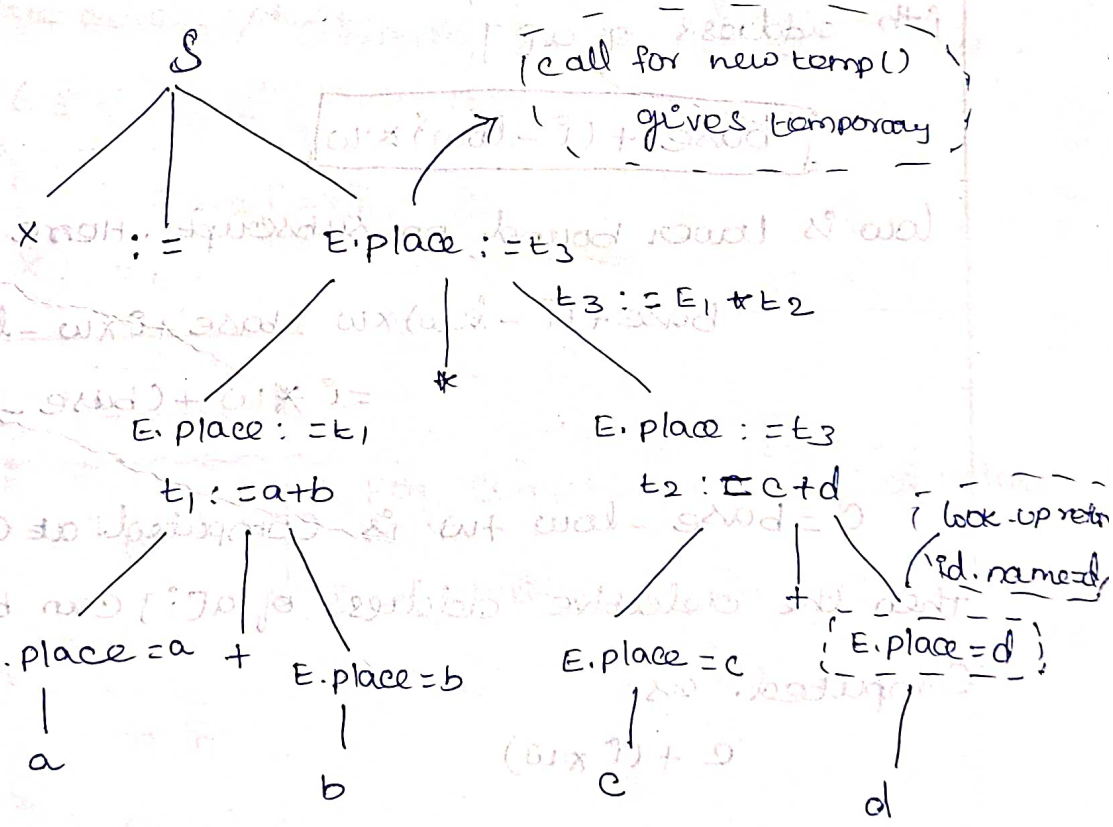
Sol:

Prod. Rule	Semantic Actions.
$S \rightarrow id := E$	{ id_entry := look_up (id.name). if id_entry ≠ nil then append (id_entry := 'E.place) else error; /* id not decl */ }
$E \rightarrow E_1 + E_2$	{ E.place := newtemp(); append (E.place := 'E_1.place' + 'E_2.place) }
$E \rightarrow E_1 * E_2$	{ E.place := newtemp(); append (E.place := 'E_1.place '*' E_2.place)
$E \rightarrow -E_1$	{ E.place := newtemp(); append (E.place := 'uminus' E_1.place) }
$E \rightarrow (E_1)$	{ E.place := E_1.place }
$E \rightarrow id$	{ id_entry := look(id.name); if id_entry ≠ nil then append (id_entry := 'E.place) else error; }

Consider the assignment stmt: $x := (a+b) * (c+d)$

Produ. Rule	Sem. Action	O/p
$E \rightarrow id$	E.place := a	
$E \rightarrow id$	E.place := b	
$E \rightarrow E_1 + E_2$	E.place := t_1	$t_1 := a+b$
$E \rightarrow id$	E.place := c	
$E \rightarrow id$	E.place := d	
$E \rightarrow E_1 + E_2$	E.place := t_2	$t_2 := c+d$
$E \rightarrow E_1 * E_2$	E.place := t_3	$t_3 := (a+b) * (c+d)$
$S \rightarrow id := E$		$x := t_3$

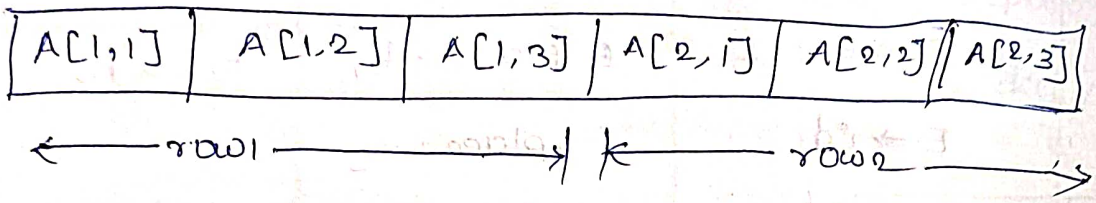
Annotated parse tree: for generation of 3 address code

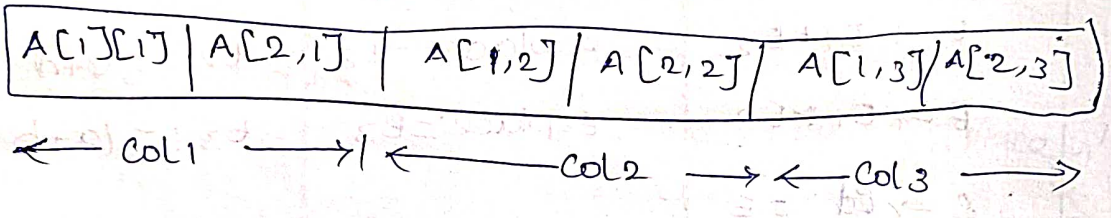


Array: Contiguous storage of elements.

2 Representations:

1. Row major repres.
2. Column major repres.





Let base is the address at a[] and w is the width of element (req. mry units) then to compute ith address of a[]

$$\boxed{base + (i - low) \times w}$$

low is lower bound on subscript. Here a[low]=base

$$base + (i - low) \times w = base + i \times w - low \times w$$

$$= i * w + (base - low * w)$$

$c = base - low * w$ is computed at compile time.

Then the relative address of a[i] can be computed as

$$c + (i \times w)$$

Generate the three address code for expression $x := A[i,j]$ for an array 10×20. Assume $low_1 = 1$ and $low_2 = 1$

Sol: Given that

$low_1 = 1$ and $low_2 = 1$

$n_1 = 10$, $n_2 = 20$.

$A[i,j] = ((i \times n_2) + j) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$

$A[i,j] = ((i \times 20) + j) \times 4 + (base - ((1 \times 20) + 1) \times 4$

$A[i,j] = 4 \times (20i + j) + (base - 84)$

Three address code for expr.

$t_1 := i * 20$

$t_1 := t_1 + j$

$t_2 := c$ /* Computation of $c = base - 84$ */

$t_3 := 4 * t_1$

$t_4 := t_2[t_3]$

$x := t_4$

Boolean Expression:

2 Types —
- For computing the logical values
- In conditional exp. using if-then-else or while-do.

Grammar:

$E \rightarrow E$ OR E

$E \rightarrow E$ AND E

$E \rightarrow$ NOT E

$E \rightarrow (E)$

$E \rightarrow$ id relop id

$E \rightarrow$ TRUE

$E \rightarrow$ FALSE

(*) Relop denoted by $\leq, =, \neq, <, >$

(*) OR, AND – left associate.

(*) Highest precedence: NOT, AND, OR

Numerical Representation:

Translation scheme for generation of 3 address code for Boolean Expr.

Production Rule	Semantic Rule.
$E \rightarrow E_1$ OR E_2	{ E.place := newtemp() append(E.place ':=' E_1.place 'OR' E_2.place) }
$E \rightarrow E_1$ AND E_2	{ E.place := newtemp() append(E.place ':=' E_1.place 'AND' E_2.place) }
$E \rightarrow$ NOT E_1	{ E.place := newtemp() append(E.place ':=' 'NOT' E_1.place }
$E \rightarrow (E_1)$	{ E.place := E_1.place }
$E \rightarrow id_1$ relop id_2	{ E.place := newtemp() append('if' id_1.place relop.op id_2.place 'goto' next_state+3); append(E.place ':=' '0'); append('goto' next_state+2); append(E.place ':=' '1'); }
E → TRUE	{ E.Place := newtemp(); append(E.place ':=' '1') }
E → FALSE	{ E.place := newtemp()

P > q AND r < s OR u > v

```
100: if P>q goto 103        5>4
101: t1 := 0
102: goto 104
103: t1 := 1
104: if r<s goto 107        7<10
105: t2 := 0
106: goto 108
107: t2 := 1
108: if u>v goto 111        15>3
109: t3 := 0
110: goto 112
111: t3 := 1
112: t4 := t1 AND t2
113: t5 := t4 OR t3
```

(*) This method of evaluation is called "Short-circuit".

Flow of Control Stmt:

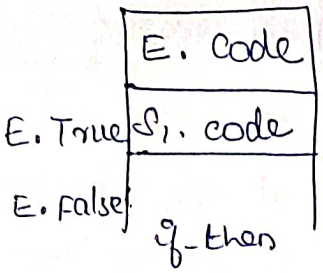
Stmts are ① If-then-else
② while-do.

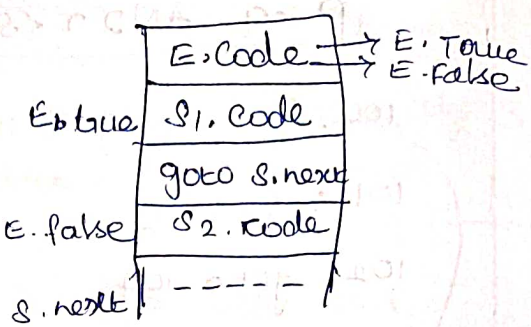
$S \rightarrow$ if E then S_1
| if E then S_1 else S_2
| while E do S_1

generating three address code -

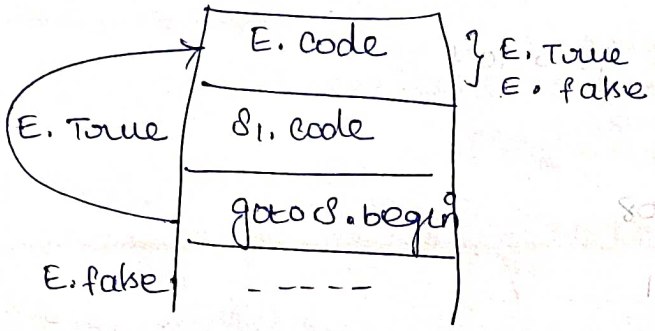
→ To generate new symbolic label the function new-label() is used.

→ With the Expr: E.true & E.false are the labels associated.





if-then



while-do

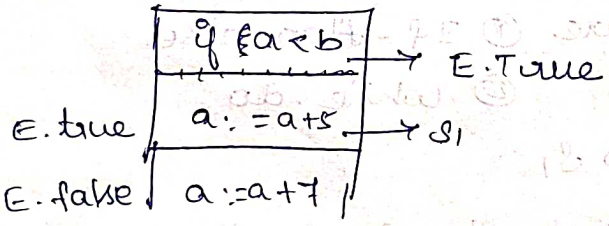
Short circuit code is:

$S \rightarrow$ if E then S_1

E.true := new_label()
E.false := S.next
S_1.next := S.next
S.code := E.code || gen_code(E.true : || S_1.code

|| - Concatenate string

if a<b then a=a+5 else a=a+7



Three address code

100 : if a<b goto L_1
101 : goto 103
102 : L_1 : a = a+5 /* E.True */
103 : a := a+7

*: Construct 3 address code for:

if [(a<b) and ((c>d) or (a>d))] then

z = x+y*z

else z = z+1

Sol: Three address code is

100: if a<b goto 102
101: goto 110
102: if c<d goto 106
103: goto 104
104: if a>d goto 106
105: goto 110
106: t1 := x+y
107: t2 := t1 * z
108: z := t2
109: goto 112
110: t3 := z+1
111: z := t3
112: STop.

Concept of Backpatching:

Backpatching is the activity of filling up unspecified info. of labels using appropriate semantic actions in during the code generation process.

will use
- → Boolean Expression
- → Flow of control stmts.

following fun are used:

- → mklist(i)
- → merge-list(P1, P2) | Concatenates P1, P2 |
- → backpatch(P, i) | insert i as target label for the stmt pointed by pointer P.

Backpatching using Boolean Expression:

grammar for BE:

E → E1 OR M E2
E → E1 AND M E2
E → NOT E1
E → (E1)
E → id1 relop id2
E → TRUE
E → FALSE
M → ε

M is nonterminal inserted

(121)

* Tlist & Flist are used to generate jumping code.
(*) attribute 'state' will associated with the M and that is used to record the number (address) of stmt. denoted as M.state. The 'nextstate' will point to next stmt.

Production Rule	Semantic action.
$E \rightarrow E_1$ OR M E_2	{ backpatch(E_1.Flist, M.state); E.Tlist := merge(E_1.Tlist, E_2.Tlist); E.Flist := E_2.Flist }
$E \rightarrow E_1$ AND M E_2	{ backpatch(E_1.Tlist, M.state); E.Tlist := E_2.Tlist; E.Flist := merge(E_1.Flist, E_2.Flist); }
$E \rightarrow$ NOT E_1	{ E.Tlist := E_1.Flist; E.Flist := E_1.Tlist; }
$E \rightarrow (E_1)$	{ E.Tlist := E_1.Tlist; E.Flist := E_1.Flist; }
$E \rightarrow id_1$ relop id_2	{ E.Tlist := mklist(nextstate); E.Flist := mklist(nextstate+1); append('if' id1.place relop 'opt' id_2.place 'goto _'); append('goto _'); }
E → True	{ E.Tlist := mklist(nextstate); append('goto _');

E-FALSE	{ E.Flist := mklist(nextstate); append('goto -'); }
~~M → ε~~	{ M.state := nextstate; }

(*) Using Backpatching, generate an IC for following EXP.

A<B OR C<D AND P<Q

Sol: scan it from left to right:

A<B matches with rule E → id$_1$ relop id$_2$.

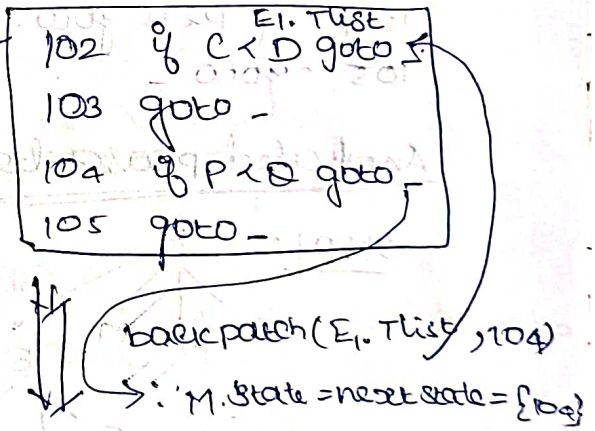
Three address code generated as:

100 if A<B goto }
1001 goto - }

similarly, for remaining part of exp.

102 if C<D goto -
103 goto -
104 if P<Q goto -
105 goto -

Hence



102 if C<D goto 104
103 goto -
104 if P<Q goto -
105 goto -

~~xxxx~~

Now Consider,

A<B OR C<D AND P<Q

E1.Tlist

100 if A<B goto-

101 goto-

backpatch(E1.Flist, 102)
E.Tlist = {100, 104}

E.Flist

102 if C<D goto- nextstate

103 goto-

104 if P<Q goto- ← E2.Tlist

105 goto-

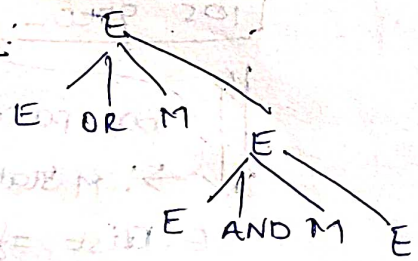
At this line C<D AND P<Q decided to be true.
That means if C<D is true and if P<Q is also true then C<D AND P<Q = TRUE.
Hence E2.Tlist = {104}

Finally 3 address code with backpatching:

100 if A<B goto -
101 goto 102
102 if C<D goto 104
103 goto -
104 if P<Q goto -
105 goto -

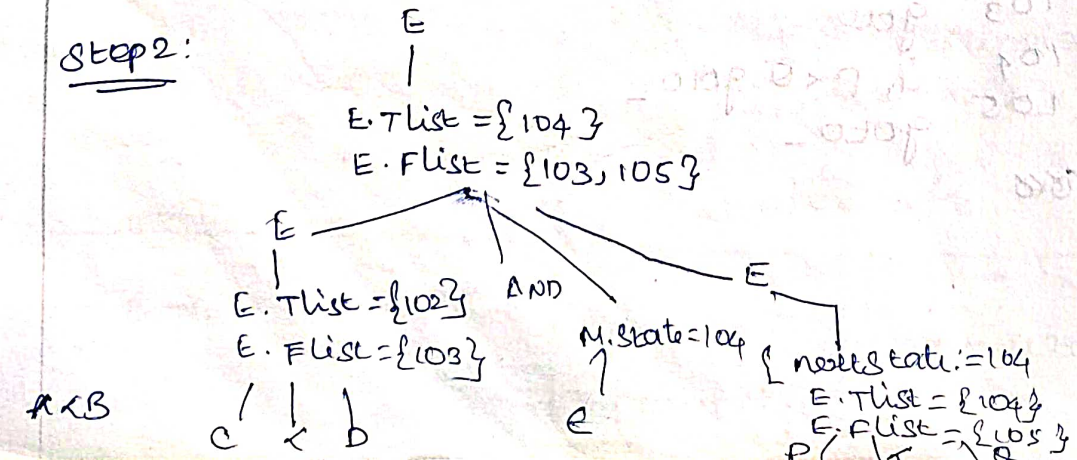
Annotated parse tree:

Step 1:

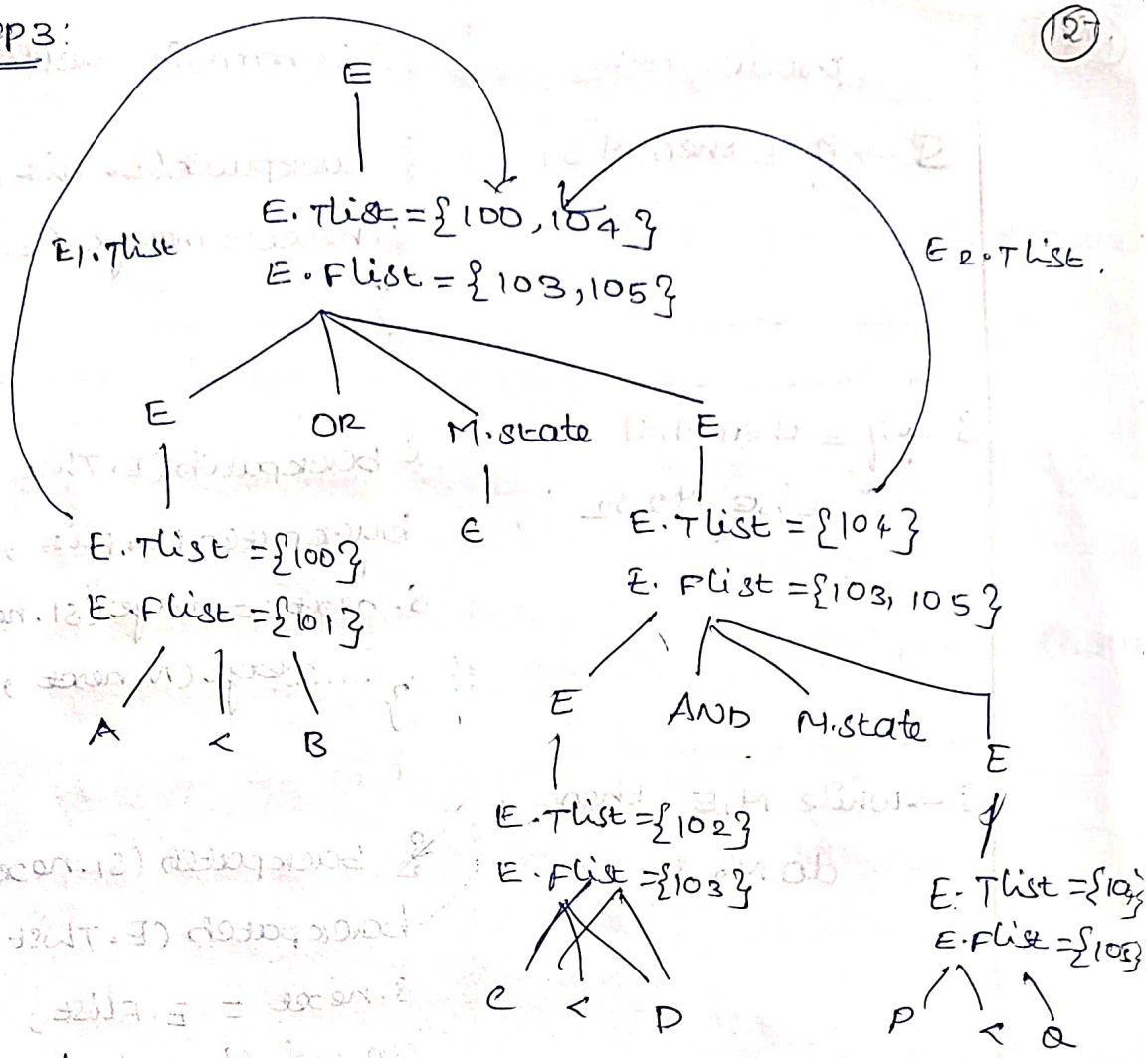


A<B C<D P<Q

Step 2:



Step 3:



Finally the annotated parse tree will be:

Remove all E from above diagram

Backpatching using Flow of Control stmts:

S → if E then S
S → if E then S else S
S → while E then do S
S → begin L end
S → A
L → L; S
L → S

S - stmt
E - Boolean exp
L - stmt list
A → Assignment stmt

Marker non terminals:

S → if E then M S1
S → if E then M1 S1 N else M2 S2
S → while M1 E then do M2 S
S → begin L end
S → A

L → L1; M S
L → S
M → ε
N → ε

prodc. rule	semantic action
$S \rightarrow$ if E then M S_1	{ backpatch(E.Tlist, M.State); S.next := merge (E.Flist, S1.next) }
$S \rightarrow$ if E then $M_1 S_1$ N else $M_2 S_2$	{ backpatch (E.Tlist, M_1.state); backpatch (E.Flist, M_2.state); S.next := merge (S1.next, merge (N.next, S_2.next)) }
$S \rightarrow$ while M_1 E then do M_2 S	{ backpatch (S1.next, M_1.state); backpatch (E.Tlist, M_2.state); S.next = E.Flist; append ('goto', M1.state); }
S → begin L end	{ S.next := L.next }
S → A	{ S.next := NULL }
L → L; M S	{ backpatch (L1.next, M.state); L.next = S.next; }
L → S	{ L.next := S.next; }
M → ∈	{ M.state := next state; }
N → ∈	{ N.next := mklist (nextstat); append ('goto _'); }

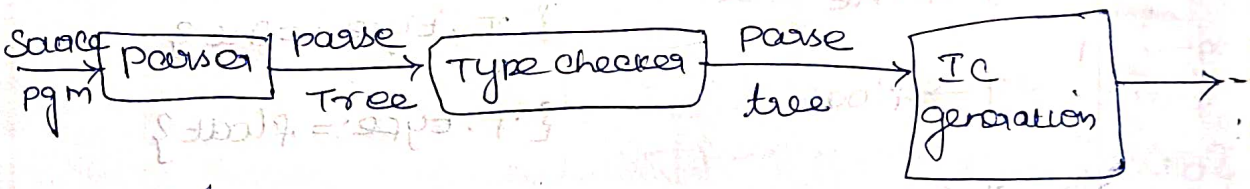
Type Checking:

Type system:

Type analysis and type checking is important activity done in sem. analysis phase. Need for type checking is -

① To detect the errors arising in the exp due to incompatible operand.

② To generate intermediate code for exp. & stmt



Role of type checker

Type Expression: - int, char, float, double, enum are type expr.

typedef int *INT_PTR

① Array : array(1,T) Eg: int arr[20];

② Product :

③ Struct : struct stud { char name[10]; float marks; } struct stud student[10];

④ Pointer; float *ptr | Pointer(float)

⑤ Function - int sum(int a, int b)

Unit-III

Syntax directed Translation and Intermediate code generation.

Design of predictive Translator:

⇒ During the process of parsing the evaluation of attribute takes place by consulting the semantic action enclosed in { } at the right of the grammar symbol. This process of execution of code fragement semantic actions from the Syntax directed definition is called Syntax-directed translation

⇒ A Translator scheme generates the o/p by executing the semantic actions in an ordered manner.

⇒ This processing is using Depth first traversal.

Give the translation scheme that converts infix to postfix form for the following grammar. also generate the annotated parse tree for string 2+6+1.

$E \rightarrow E+1$ { print('+') }
$E \rightarrow T$
$T \rightarrow 0$ { print('0') }
$T \rightarrow 1$ { print('1') }
$T \rightarrow 2$ { print('2') }
$T \rightarrow 3$ { print('3') }
$T \rightarrow 4$:
$T \rightarrow 5$:
$T \rightarrow 6$:
$T \rightarrow 7$:
$T \rightarrow 8$:
$T \rightarrow 9$ { print('9') }

Solution: Let us convert this grammar onto non left recursive

$E \rightarrow TP$

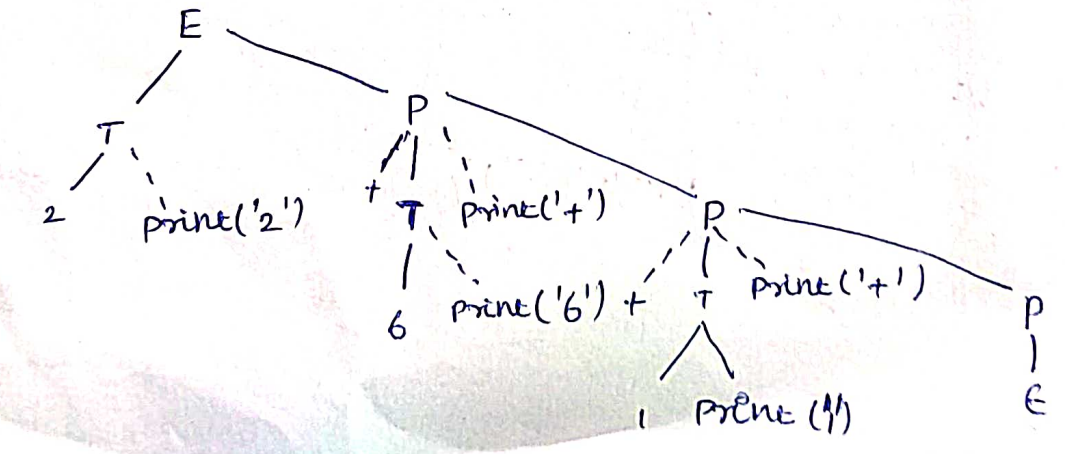
$T \rightarrow 0|1|2|3|4|5|6|7|8|9$

$P \rightarrow +TP|\epsilon$

Translation scheme:

Productions	Semantic Actions.
$E \rightarrow TP$	
$T \rightarrow 0$	{print('0')}
$T \rightarrow 1$	{print('1')}
$T \rightarrow 2$	{print('2')}
$T \rightarrow 3$	{print('3')}
$T \rightarrow 4$	{print('4')}
$T \rightarrow 5$	{print('5')}
$T \rightarrow 6$	{print('6')}
$T \rightarrow 7$	{print('7')}
$T \rightarrow 8$	{print('8')}
$T \rightarrow 9$	{print('9')}
$P \rightarrow +TP/\epsilon$	{print('+')} P/ϵ

Annotated parse tree



Guideline for Designing the translation scheme:

Production Rule	Semantic Action
$E \rightarrow E_1 * T$	$\{ E.val := E_1.val * T.val \}$

val - synthesized attribute.

① $A \rightarrow X\ Y$ then computation of X.in must be done alrg -dy

② $A \rightarrow B_1\ B_2$ the semantic action $\{ B_1.s := B_2.s \}$ is Invalid as computation of attribute for B_1 is based on Computation of attribute of B_2

③ Computation of synthesized attribute for non-termin On the left, first do all the computations of the attributes that refer this non terminal.

Type Systems: - Important activity done in the semantic Analysis Phase

Need:

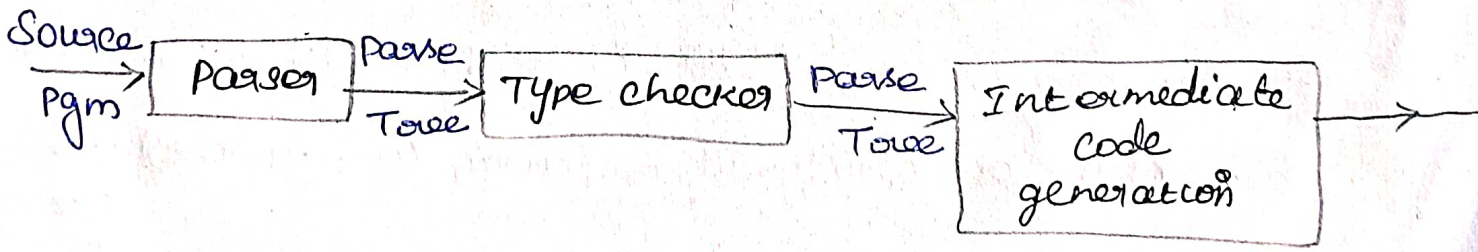
① To detect the errors arising in the expression due to Incompatible operand.

② To generate intermediate code for expressions and stmts.

Supports two types of data types.

① Basic data types - Integer, character, real Boolean, ...

② Constructed data types - Arrays, record, set, Pointers...



Role of type checker.

Type Expression:

① Basic type is called type expression. Hence int, float, double, enum are type expressions.

② While performing type checking two special basic types are needed such as type_error and void

③ The type name is also type expression.

typedef int *INT_PTR

④ Type constructors are also type expressions.

eg: array, product, struct, product.

i) Array – array(I, T) I - Index set, usually Integer

2) Product – $T_1 \times T_2$

3) Struct – struct stud {
char name[10];
float marks;
};
struct stud student[10];

4) Pointers – float *xyz;
pointer(float)

5) Function – int sum (int a, int b)

LIST → id LIST.type = T.type

Here T.type = int hence LIST.type = int.

∴ LIST → L_1[num] – L_1.type := { array (0,..., num.val − 1), LIST.type }

L_1.type := { array (0, ... 9), int } as num.val=10

the entry for A[10] will be array (0 ... 9, int) as type expressions

Specification of simple type checker:

Type checking of Expressions:

E → literal { E.type := char }

E → num { E.type := int }

Here E can be literal or num, the data types associated with them can be char or int respectively.

E → id { E.type = look_up(id.entry)

look_up function reads the symbol table for id entry and thereby it obtains the type of identifier.

E → E1 mod E2 { E.type := if E1.type = int and E2.type = int
then int
else
type_error
}

Type checking of stmts:

Special type void is used.

S → id := E

S → if (E) S1

S → while (E) S1

S → S1 ; S2

production Rule	Semantic rule.
S → id := E	{ S.type := if id.type = E.type then void else type_error }
S → if (E) S1	{ S.type := if E.type := boolean(True) then S1.type else type_error }

$S \rightarrow \text{while}(E) S_1$	{S.type := if E.type := boolean (True) then S_1.type else type-error }
$S \rightarrow S_1 ; S_2$	{ S.type := if S_1.type := void and S_2.type = void then void else type-error }

Equivalence of Type Expressions:

The Job of type checker is to find whether two type expressions are equivalent or not.

Two categories:

⇒ Name equivalence
⇒ Structural equivalence.

① Structural Equivalence of Type expressions:

When two expressions are the same basic type or formed by applying the same constructor to structually equivalent types then those expressions are called structurally equivalent.

S_1	S_2	Equivalence	Reason.
char	char	S_1 is Equivalent to S_2	Similar basic types
pointer (char)	pointer (char)	S_1 is equivalent to S_2	Similar constructor pointer to the char type.

Four bits encoding

Basic type	Encoding
boolean	0000
char	0001
Integer	0010
real	0011

Two bits encoding

Type constructor	Encoding
pointer	01
array	10
function	11

Name Equivalence of Type Expressions:

In the name equivalence the type expressions are given the names. The two types expressions are said to be name equivalent if and only if they are identical.

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

First, second are name equivalent.
last1, last2 are name equivalent.

Type conversions:

⇒ The process of converting one type to another.

Eg: Stmt f+i where f is a float type identifier and i is an integer type identifier and addition of one has to be done.

⇒ Computer wants to convert one type to another type.

Two types of conversions:

① Explicit Conversion:

```
int xyz, p;
p = (float) xyz;
```

Identifier xyz is type-casted and this is how explicit conversion from int to float takes place.

② Implicit type conversion:

```
for (i=0; i<n; i++)
A[i] = 1;
```

} takes 4.8 microseconds to execute

Explicit type conversion.

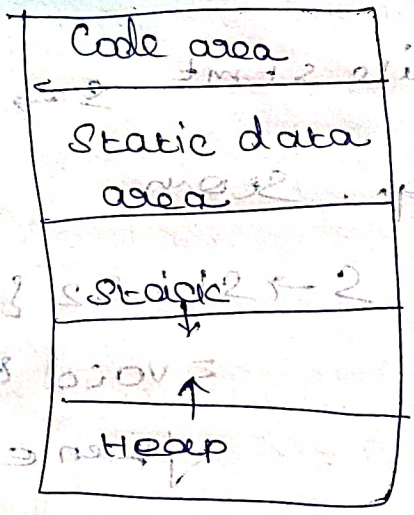
```
for (i=0; i<n; i++)
A[i] = 1.0
```

} 5.4 ms to execute.

Unit - IV

Run Time Environment and Code Generation

Storage Organization: - N/D - 2022 A/P - 2023

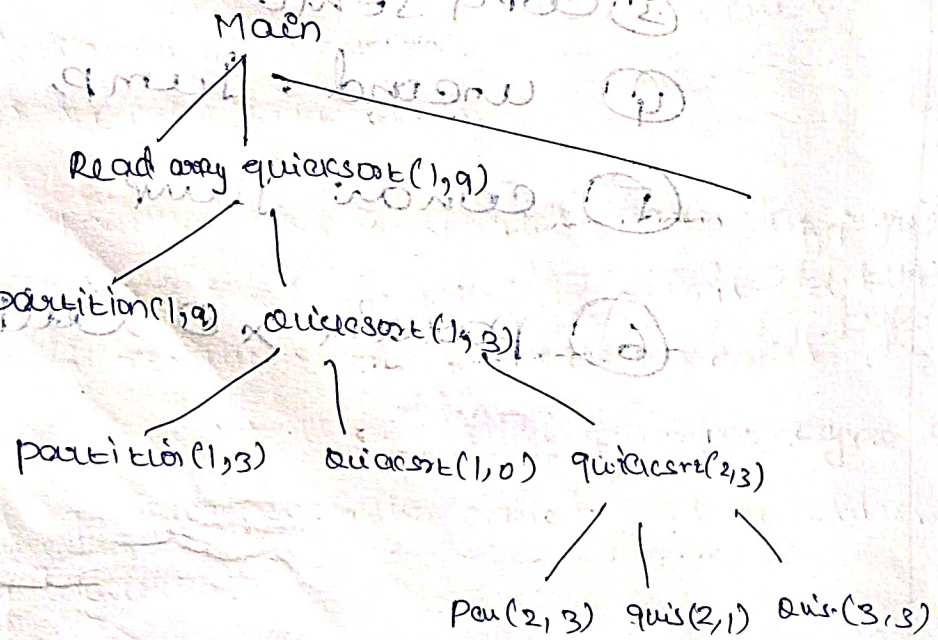


Strategies: April/May, -2022

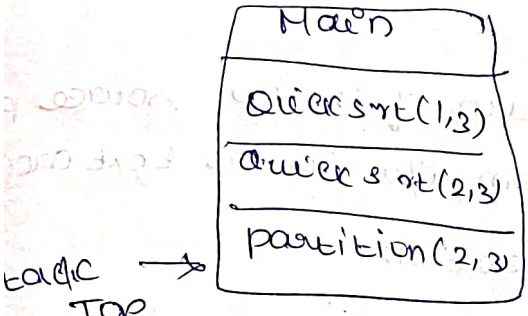
1. Static Allocation
2. Stack Allocation
3. Heap Allocation.

Storage Allocation space:

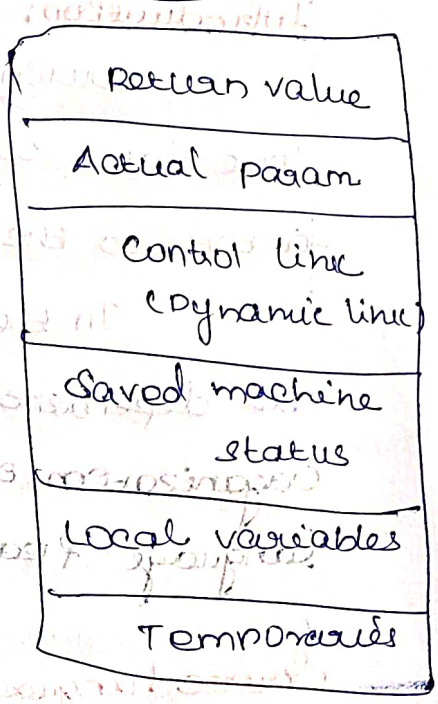
Activation Tree: - graphical representation.



Control stack:



Activation Record - 2022



Calling sequences:

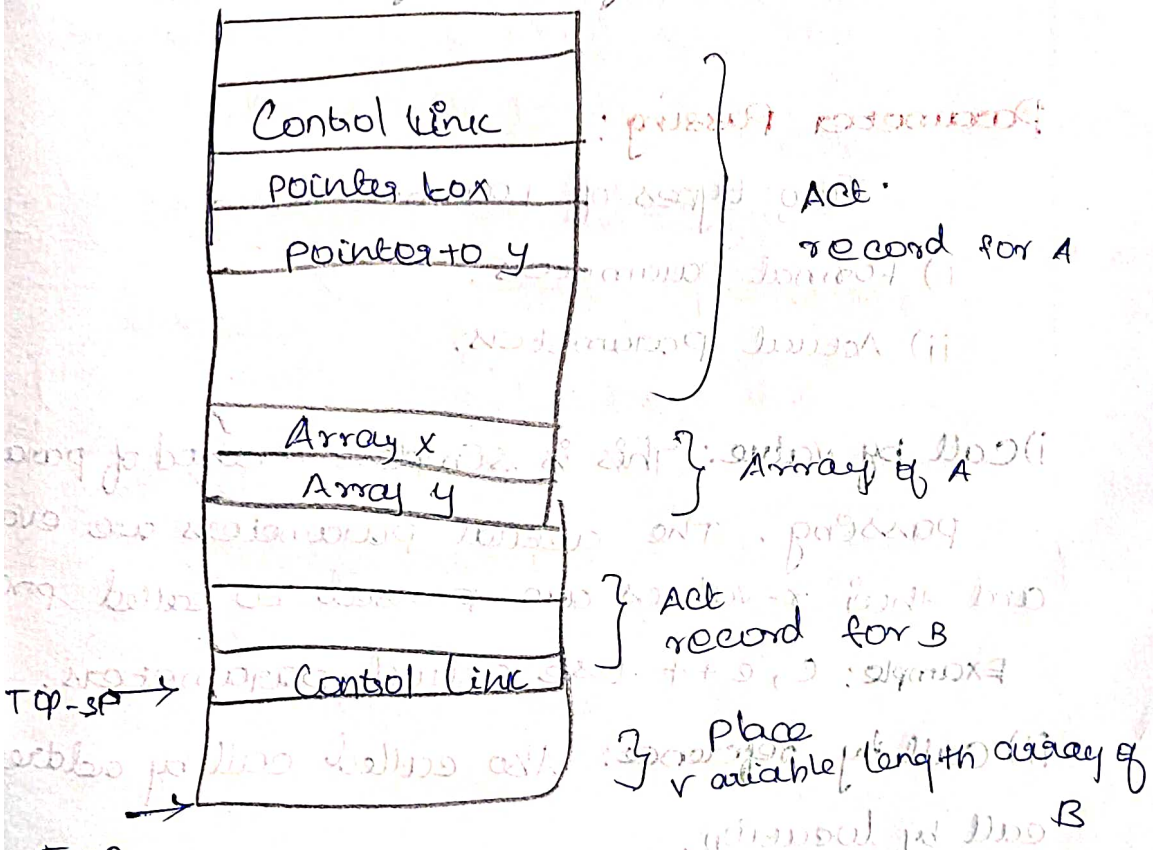
⇒ Series of calls to the procedures

⇒ Allocate Activation Records on the stack.

⇒ Return sequence is used to restore the state of machine.

⇒ Caller is procedure that calls Callee

Variable Length data:



Runtime Environments:

Introduction:

During execution of the i/p source pgm some data objects from i/p source text are important factors to the code generation.

In the code generation these data objects are dependent upon the org. of my. And the Organization of memory is driven by the source language features.

Source Language Issues:

① Does the source language allow recursion?

② How the parameters are passed to the Procedure?

③ Does the procedure refer nonlocal names? How

④ Does the language support the my allocation and deallocation dynamically?

Parameter Passing:

Two types of parameters:

i) Formal parameters.

ii) Actual parameters.

i) Call by value: This is simplest method of parameter passing. The actual parameters are evaluated and their r-values are passed to called procedure

Example: C, C++ use actual parameters.

ii) Call by reference: Also called call by address or call by location.

The values of actual parameters can be changed

Eg: Reference parameters in C++, pascal's var param

3) Copy restore: This method is hybrid b/w call by value and call by reference.

Eg: Also called copy-in-copy-out or values result.

4) Call by name: This is less popular method of Parameter passing.

⇒ The locals names of called procedure and names of calling procedure are distinct.

Eg: ALGOL uses call by name method.

call by value	call by reference	copy restore	call by name.
i 50	i 50	i 50	i 50
7 50	50 50	50 i	Error.

Symbol Table: - A|M-2022

(*) The symbol table is a DS used by compiler to keep track of semantics of variable.

(*) The symbol table is built in lexical and Syntax analysis phase.

I-value and r-value:

The I and r prefixes come from left and right side assignment.

```
a := i+1
↑    ↑
I-value  r-value
```

Symbol Table Entries:

The items to be stored in symbol table a

1) variable names
2) constants.
3) procedure names
4) function names
5) Literal constants & strin
6) Compiler generated tem
7) Labels in source langua

Compiler uses following types of information fro Symbol table:

1) Data type
2) Name
3) Declaring procedures
4) Offset in storage
5) If structure or record
6) No. and type of arguments passed,
7) Base address.

How to store names in Symbol Table?

Two types of name representation.

1) Fixed - length Name:

Fixed space for each name is allocated.

Name	Attributes
c a l c u l a t e	
S U m	
a	
b	

2) Variable Length name:

The amount of space required by string is used to store the names.

Name		Attribute
Starting Index	Length	
0	10	
10	4	
14	2	
16	2	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
c	a	l	c	u	l	a	t	e	$	s	u	m	$	a	$	b	$

Symbol table Management:

Requirements:

① For quick insertion of identifies and related infor.

② For quick searching of identifies.

i) List Data structure for symbol Table:

⇒ Linear Data structure list is a simplest kind of mechanism to implement the symbol table.

⇒ In this method an array is used to store names and associated information.

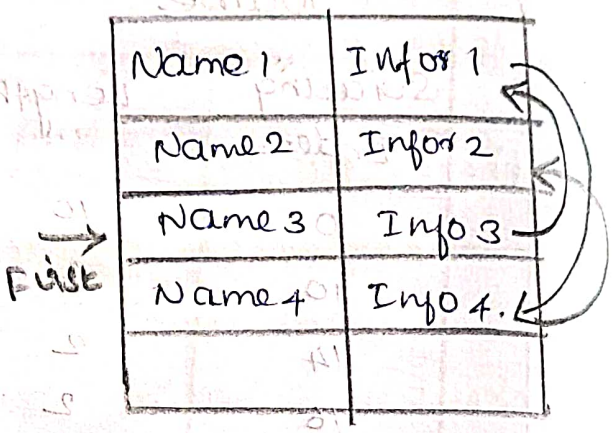
Name 1	Info 1
Name 2	Info 2
Name 3	Info 3
.....	.....
Name n	Info n

⇒ The advantages of list organization is that it takes minimum amount of space.

2) Self organizing List:

The symbol table implementation is using linked list. A link field is added to each record.

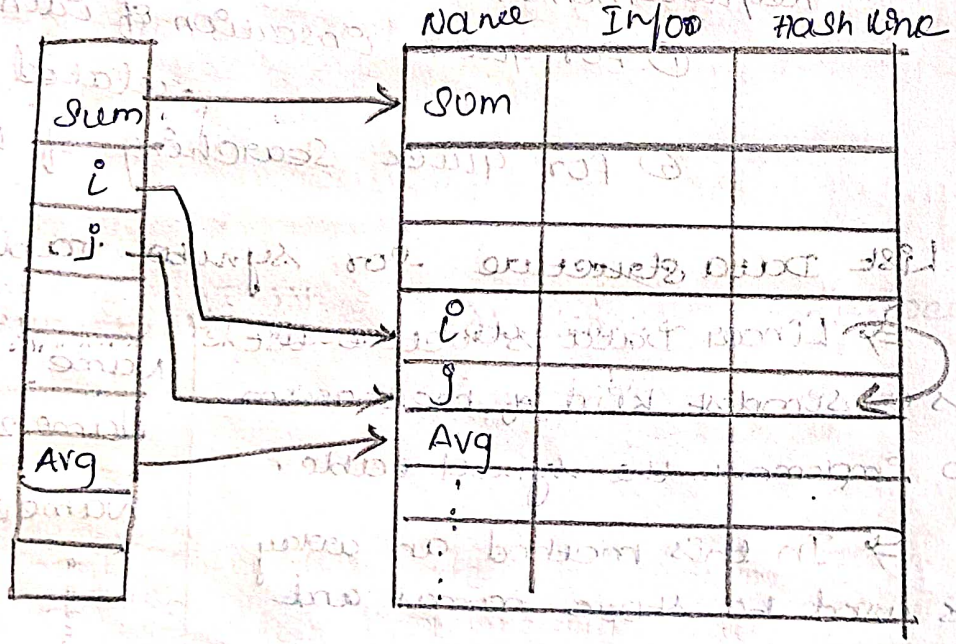
A pointer "First" is maintained to point to first record of the symbol table.



3) Hash Table:

Hashing is an important techniques used to search the records of symbol table. This method is superior to list organization.

position = h (name)



Advantages of hashing is quick search is possible and the disadvantage is that hashing is complicated to implement.

Dynamic Storage Allocation:

Two techniques.

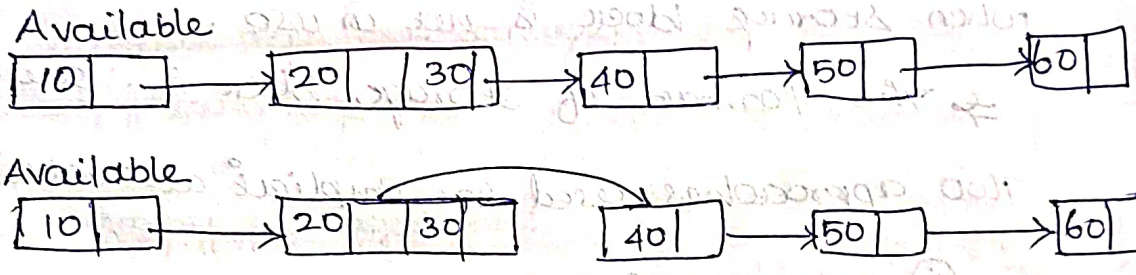
1. Explicit allocation.
2. Implicit allocation.

Explicit Allocation:

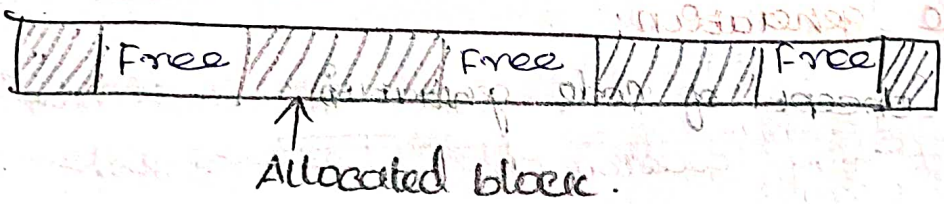
It can be done for fixed size and variable sized blocks.

In this techniques free list is used. Free list is a set of free blocks. This list can observed when we want to allocate memory.

The pointer which points to first block of memory is called "Available".



Explicit Allocation of variable sized blocks:

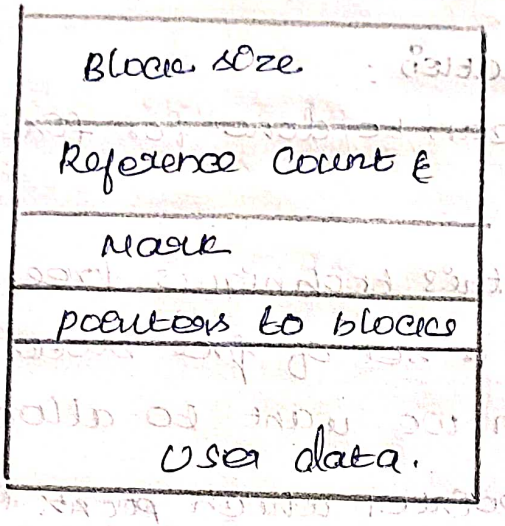


List of 4 blocks gets allocated and second, forth and sixth block is deallocated then fragmentation occurs. Thus we get variable sized blocks that are available free.

(AA)

Implicit Allocation:

The implicit allocation is performed using user pgm and runtime packages.



⇒ The run time package is required to know when storage block is not in use.

⇒ The format of storage block is shown.

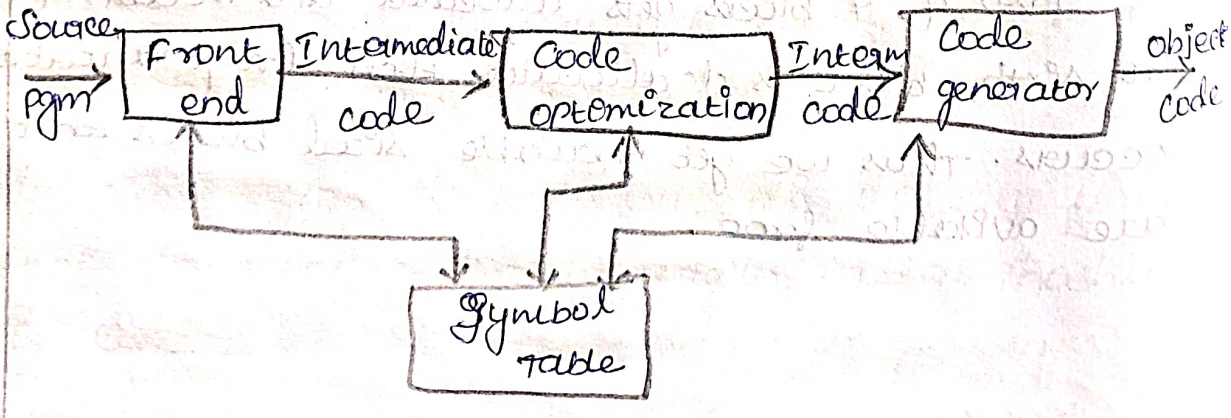
Two approaches used for implicit allocation.

① Reference Count

② Marking techniques.

Code generation:

Concept of code generation:



(A4) Issues in the Design of a code generator:- A/M N/D 2022

① I/P to the code generator:

(*) The code generator phase take IC as I/P. IC along with the symbol table info is used to determine the runtime addresses of the data objects

(*) The IC may be represented using graphical representations such as syntax trees, or DAGs.

Target programs:

The O/P of code generator is target code. The target code comes in three forms such as;

1) absolute machine language,
2) relocatable machine lanage,
3) assembly language.

Memory Management:

(*) Both the front end and code generator performs the task of mapping the names in the source pgm to addresses to the data objects in run time mcy.

(*) The names in source pgm, in three address code refer to the entries in the symbol table.

Instruction Selection:

The uniformity and completeness of instruction set is an important factor for the code generator. The selection of instruction depends upon the instruction set of target machine.

eg: X := a + b

```
MOV a, R0
ADD b, R0
MOV R0, X
```

Register allocation :-

During register allocation, select appropriate set of variables that will reside in registers.

$t_1 = a + b$	MOV a, R0
$t_2 = t_1 * c$	ADD b, R0
$t_1 = t_1 / d$	MUL c, R0
	DIV d, R0
	MOV R0, t_1

Target Machine Description :

For designing the good code generator it is necessary to have prior knowledge of target machine and instruction set used for this target machine.

MOV - moves from source to destination

ADD - Add source to des.

SUB - subtracts source from des.

Eg:

MOV R1, M stores the contents of register R1 into my location M

→ Indexed addressing mode the address offset c from the value of register R0 can be written as, MOV c(R1), M

MOV #5, R0 this instruction we can store the constant 5 into register R0

(8) Cost of Instruction:

Instruction	Cost	Interpretation.
MOV R0, R1	1	Cost of register mode +1 = 0+1 = 1
MOV R1, M	2	use of mem variable +1 = 1+1 = 2
SUB 5(R0) * 10(R1)	3	use of first constant + use of 2nd constant +1 = 3

Design of a simple code Generator: - A/M -2018 N/D -2022

Generating target code from three address statement.

```
x := a+b;
```

Corresponding target code:

```
ADD b, Ri
```

. Here Ri holds value of a

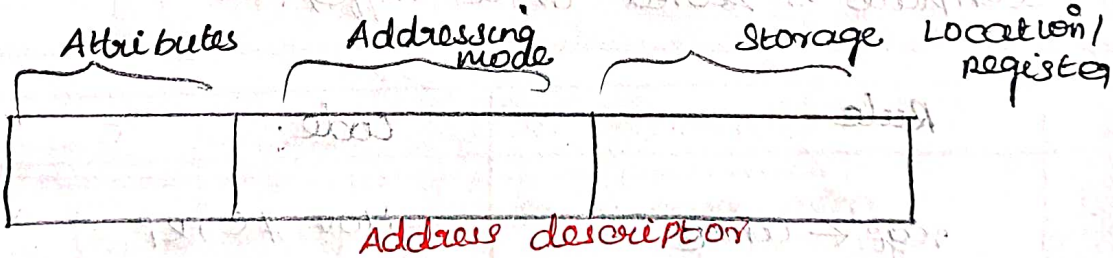
Here cost = 2

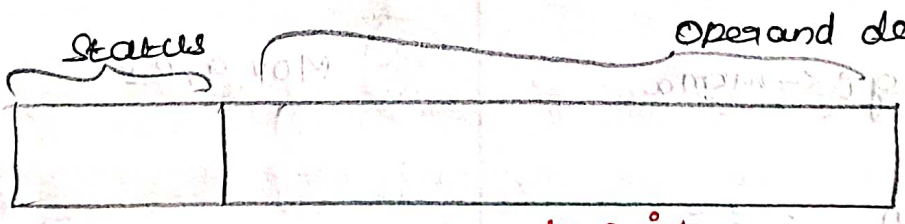
(or)

```
MOV b, R1
ADD R1, R0
```

Here R0 holds value of a

Here cost = 2





Three address code:	Target code:
$t_1 = a + b$	MOV a, R0
$t_2 = b - c$	ADD b, R0
$t_3 = t_1 / t_2$	MOV b, R1
$t_4 = t_1 * t_2$	SUB c, R1
$t_5 = t_3 - t_4$	MOV R1, R2
$t_6 = t_5 + f$	DIV R0, R2
$x = t_6$	SUB R1, R0
	ADD f, R0
	MOV R0, x

Optimal Code Generation for Expressions:

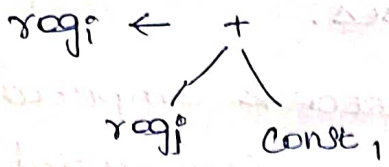
The code generator is a tree rewriting techniques which the instruction selection can be done automatically from a high-level specification of the target machine.

Draw the tree structure for complete expressi

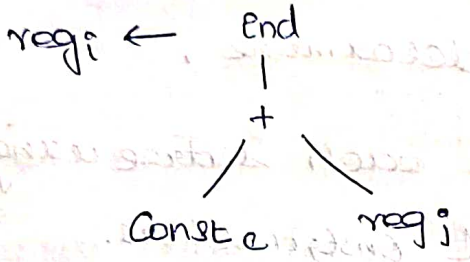
Traverse the tree in bottom up fashion and match the subtrees

The corresponding code for the matched template is stored in a buffer.

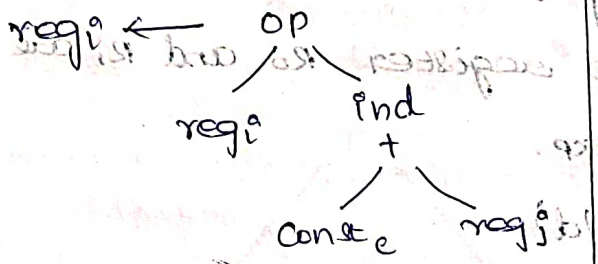
Rule	code.
reg i ← const c	MOV #c, Ri
reg i ← mem a	MOV a, Ri
mem ← := (mem a, reg i)	MOV Ri, a



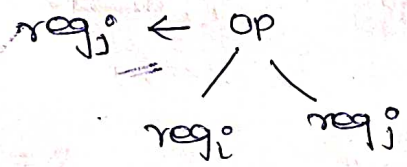
MOV Rj * Ri



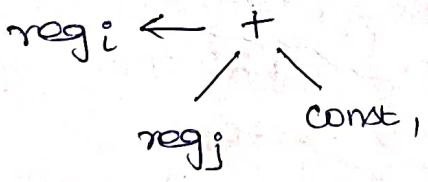
MOV c(Rj), Ri



OP c(Rj), Ri



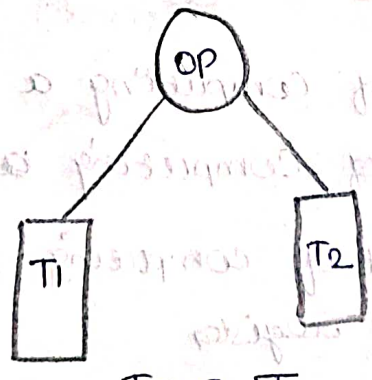
OP Rj, Ri



MOV Const, Rj
OP Rj, Ri

Dynamic prgmng Code Generation:

Principle of Dynamic programming



Tree T

Its working in three phases.

1st phase the cost vector is computed in the bottom up fashion for the constructed exp. T

2nd phase of alg, the Tree T is computed using cost vector to determine.

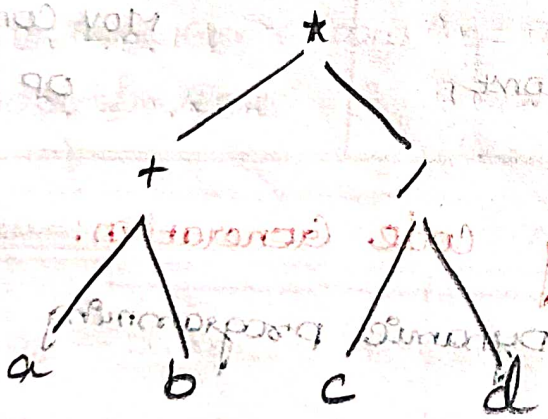
3rd phase traverse each subtree using cost vectors and associated instructions.

Eg: By considering the instruction set each of unit cost and two register R_0 and R_1 are available the cost vector for exp.

$$(a+b) * (c/d)$$

$R_i = M$
$R_i = R_i$ op R_j
$R_i = R_i$ op M
$R_i = R_j$
$M = R_i$.

Solution: For node labelled as 'a'



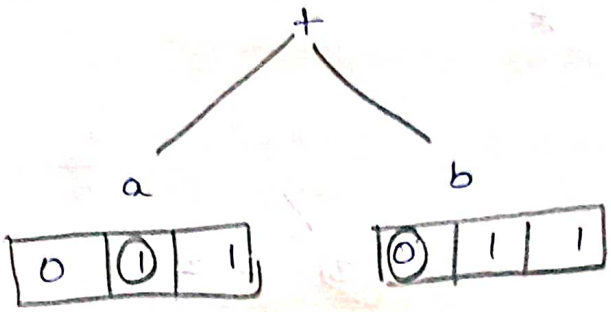
C[0] = 0 The cost of computing a in mem.

C[1] = 1 The cost of computing a in register

C[2] = 1 The cost of computing a into two available register

C[0]=3 This cost of Computing + one o mxy

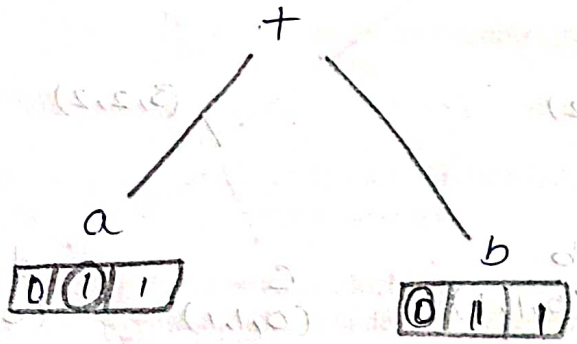
$R_0 = R_0 + M$



Total cost = 1 = 2+1=3

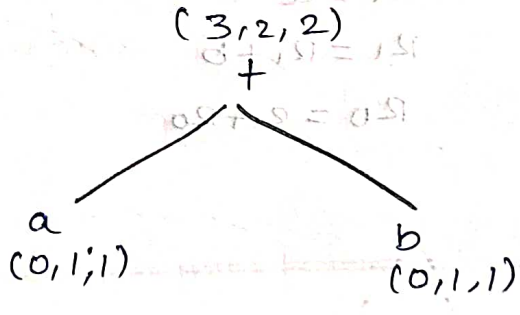
C[1]=2

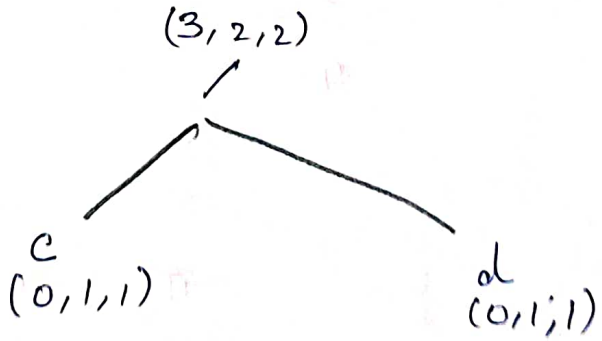
$R_0 = R_0 + M$
$\Rightarrow 1 + 0 + 1 = 2$



C[2]=2

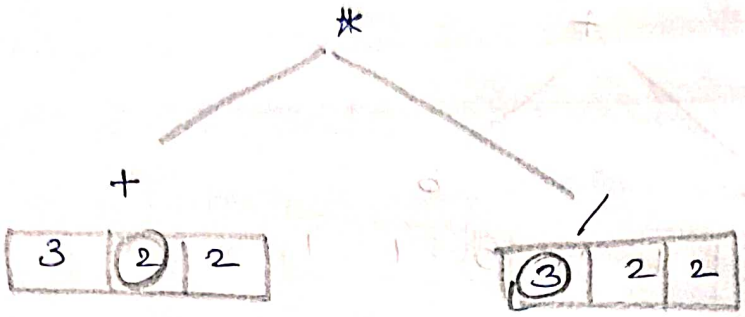
c[2] = 1+0+1=2

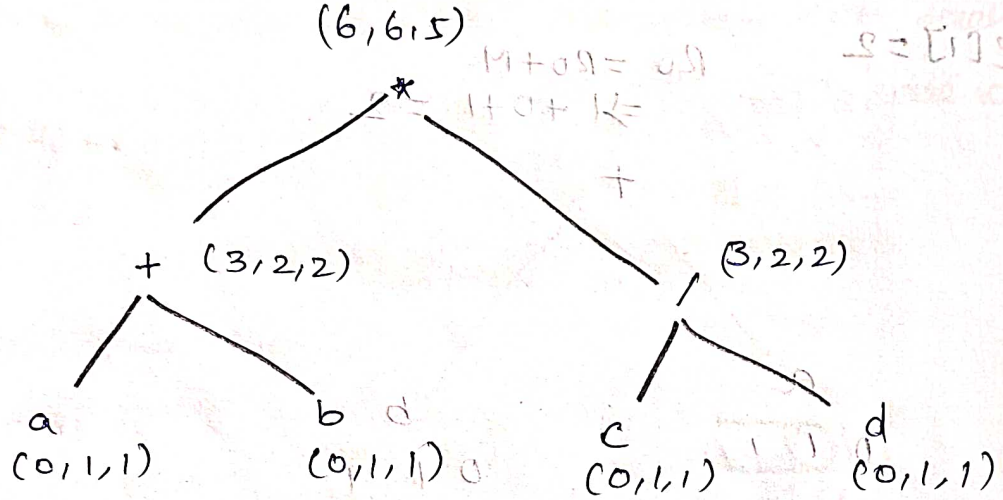




c[1]=6

c[1] = 2+3+1 = 6





R0 = a or

R0 = R0 + b

R1 = c

R1 = R1 / d

R1 = R1 * R0

R0 = c

R0 = R0 / d

R1 = a

R1 = R1 + b

R0 = R1 + R0

Unit-5

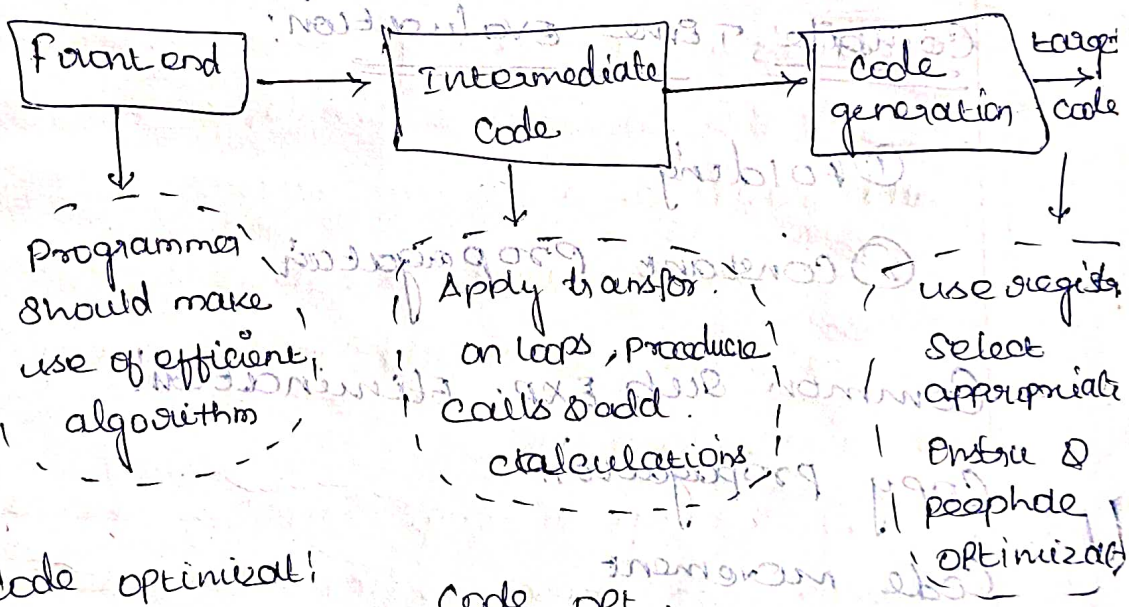
Code Optimization

The code optimization is a technique required to produce an efficient target code.

Two important issues:

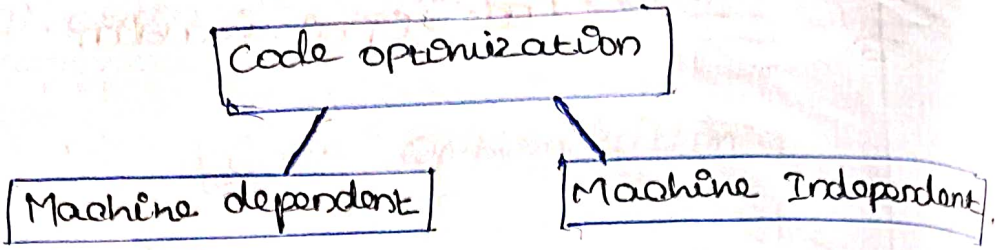
① The semantic equivalence of the source pgm must not be changed.

② Pgm. efficiency must be achieved without changing the algorithm of the pgm.



9342-112911

classification of optimization:



Machine dependent optimization is based on characteristics of the target machine for the instruction set used and addressing modes used for the instructions to produce the efficient target code.

Machine Independent optimization is based on the characteristics of the programming languages for appropriate prgming structures.

Properties of optimizing compilers:

(1) The source code should be such that it should produce minimum amount of target code

(2) There should not be any unreachable code

(3) Dead code should be completely removed from source language.

(4) The optimizing compilers should apply following code improving on source languages.

i) Common subexp. elimination

ii) Dead code elimination

iii) Code movement

iv) Strength reduction.

Principle Source of optimization:

Optimization can be done locally or globally. If the transformation is applied on the same basic block then that kind of transformation is done locally otherwise transformation is done globally.

Function preserving transformations:

1) Common sub expression elimination.
2) Copy propagation
3) dead-code elimination
4) Constant folding

Compile Time Evaluation:

1) Folding:

In the folding technique the computation of constant is done at compile time instead of execution time.

length = (22/7) * d

2) Constant propagation:

In this technique of the value of variable is replaced and computation of an expression is done at the compilation time.

```
PI = 3.14
r = 5
Area = P * r * r.
```

Area = 3.14 * 5 * 5

2) Common sub expression Elimination:

An expression appearing repeatedly in the pgm which is computed.

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

sub expression eliminated:

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

Copy propagation:

variable propagation means use of one variable instead of another.

Eg:
```
x = pi;
...
area = x*r*r;
```

Optimization using variable propagation:

```
area = pi*r*r;
```

Code movement:

① To reduce the size of the code.

② To reduce the frequency of exe. code.

```
for (i=0; i<=10; i++)
{
  x = y*5;
  k = (y*5)+50;
}
```

Optimized:
```
z = y*5
for (i=0; i<=10; i++)
{
  x = z;
  k = z+50;
}
```

Strength Reduction:

The strength of certain operators is higher than others.

```
for(i=1; i<=50; i++)
{
...
count = i * 7;
...
}
```

The code can be replaced by using strength reduction:

```
temp = 7
for (i=1; i<=50; i++)
{
...
count = temp;
temp = temp + 7;
}
```

Dead code elimination:

The variable is said to be live in a pgm if the value contained into it is used subsequently.

For eg:

```
j = i;
...
x = i + 10;
...
```

The optimization can be performed by eliminating the assignment stmt j = i. The assignment stmt is called dead assignment.

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

Peep-hole - Optimization:

Peep-hole optimization is a simple and effective technique for locally improving target code.

Characteristics of peephole optimization:

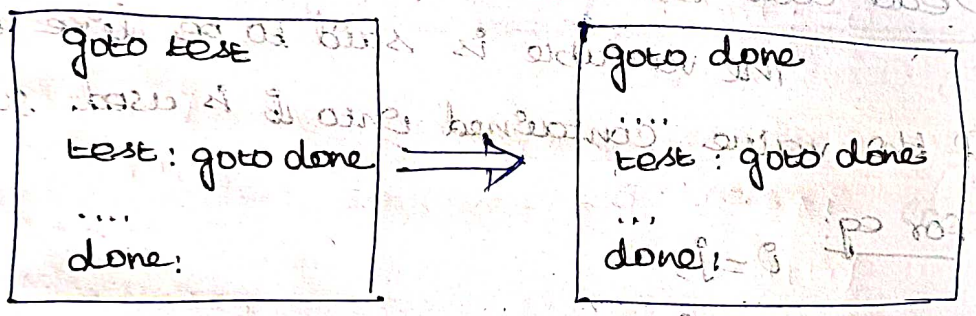
① Redundant Instruction elimination:

Redundant loads and stores can be eliminated in this type of transformations.

Eg: MOV R0, x
MOV x, R0

② Flow of control optimization:

Unnecessary jumps on jumps can be eliminated.



③ Algebraic simplification:

Peephole optimization is an effective technique for algebraic simplification.

x := x + 0

(or)

x := x * 1

④ Reduction in strength:

If we can replace these instructions by cheaper instruction.

Eg: x^2 is cheaper than $x * x$.

⑤ Machine idioms:

The target instructions have equivalent machine instructions for performing some operations. Hence we can replace these target instructions by equivalent machine instructions.

---

5) Basic blocks and flow graphs:

The basic block is a sequence of consecutive stmts in which flow of control enters at the beginning and leaves at the end.

```
t1 = a*5
t2 = t1+7
t3 = t2-5
t4 = t1+t3
t5 = t2+b
```

Algorithms for partitioning into Blocks:

(1) First determine the leaders by using following rules

⇒ The 1st stmt is leader.

⇒ Any target stmt of conditional or unconditional goto is a leader

⇒ Any stmt that immediately follow a goto or unconditional goto is a leader.

2) The basic block is formed starting at the leader stmt & ending just before the next leader stmt.

Eg: Two vectors a and b length 10 and partition it into basic blocks.

Solution:

```
Prod = 0;
i = 1;
do
{
prod = prod + a[i] * b[i]
i = i+1
}while (i <= 10);
```

Solution: Equivalent three address code:

1. Prod := 0
2. i := 1
3. t1 := 4 * i
4. t2 = a[t1]
5. t3 = 4 * i
6. t4 = b[t3]
7. t5 = t2 * t4
8. t6 = Prod + t5
9. Prod = t6
10. t7 = i + 1
11. i = t7
12. if i <= 10 goto (3)

Block:1

1. Prod := 0
2. i = 1

Block:2

3. t1 = 4 * i
4. t2 = a[t1]
5. t3 = 4 * i
6. t4 = b[t3]
7. t5 = t2 * t4
8. t6 = prod + t5
9. prod = t6
10. t7 = i + 1
11. i = t7
12. if i <= 10 goto (3)

DAG: Representation:

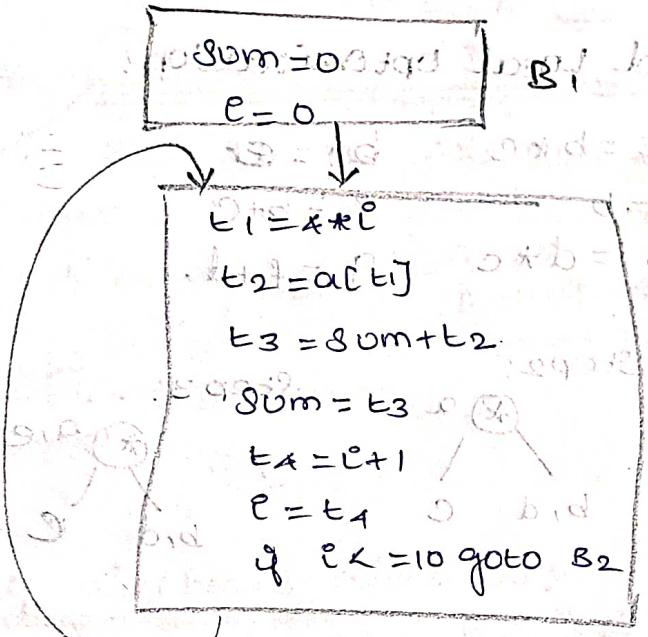
Construct DAG for

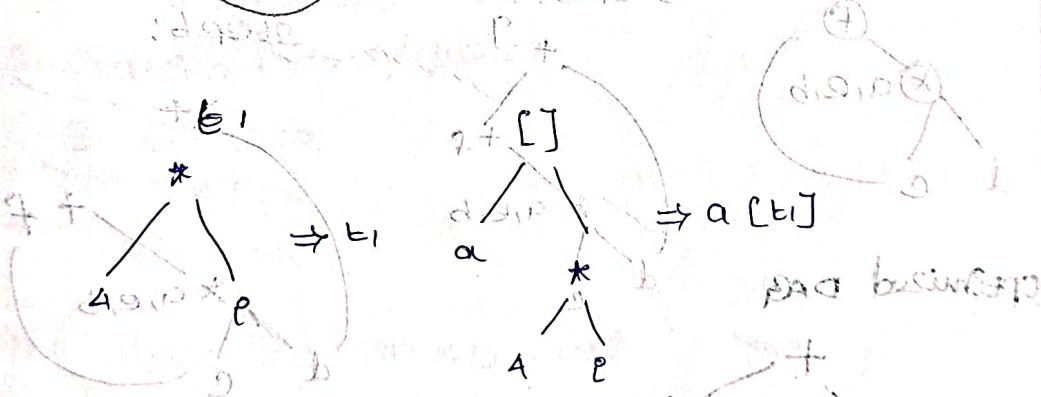
```
sum=0;
for (i=0; i<=10; i++)
sum=sum+a[i]
```

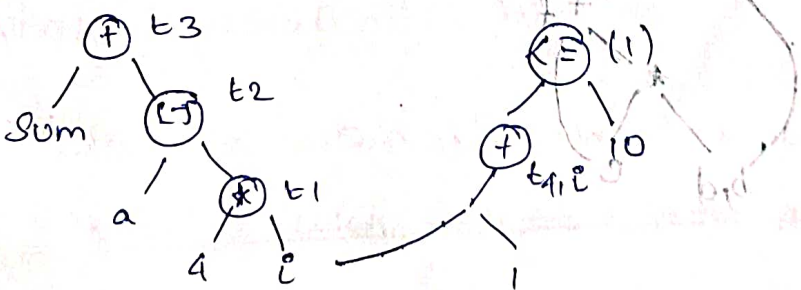
Solution:

1) sum = 0
2) i = 0
3) $t_1 = A * i$
4) $t_2 = a[t_1]$
5) $t_3 = sum + t_2$
6) $sum = t_3$
7) $t_4 = i + 1$
8) $i = t_4$
9) if $i <= 10$ goto (3)

Block B_2 for construction of DAG:







Algorithm for Construction of DAG:

case (i) $x = y \text{ op } z$

Case ii) $x := \text{op } y$

case iii) $x := y$;

Applications of DAG:

① determining the common sub-expressions.

② determining which names are used inside the block and computed outside the block.

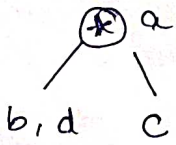
③ determining which stmts of the block could have their computed value outside the block.

DAG based Local Optimization:

$a := b * c$
$d := b$
$e := d * c$
$b := b$
$f = b + c$
$g = f + d$.

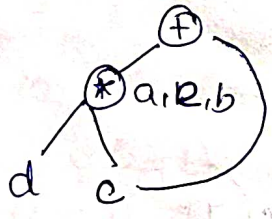
Step 1:

Step 2:

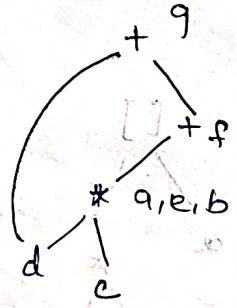


Step 3:

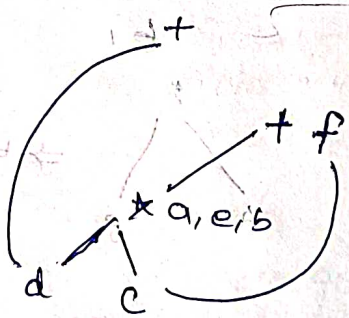
Step 4:



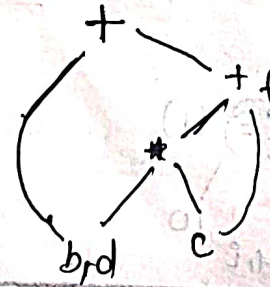
Step 5:



Step 6:



Optimized DAG



7) DAG - Optimization of basic block:

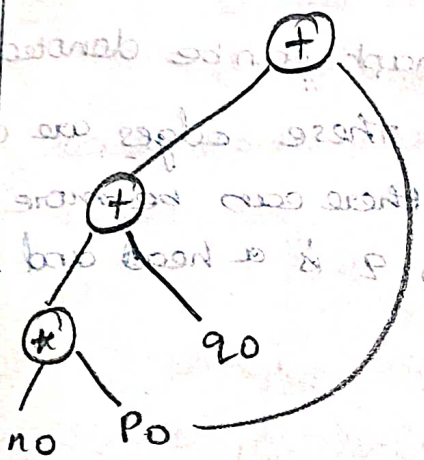
Two types of optimizations that can be done on basic blocks.

① Structure preserving transformation:

Structure preserving transf. can be applied by applying some principle techniques such as common sub expression elimination, variable and constant propagation, code movement, dead code elim.

Eg: $m := n * p$
$n := m + q$
$p := n * p$
$q := m + q$

If we assume the values of $n = 1$, $p = 2$ and $q = 3$ then expressions becomes



$m = 0 * p$ ←
$: 1 * 2$
$\therefore m := 2$
$n := m + q$ ←
$:= 2 + 3$
$\therefore n := 5$
$p := n * p$ ←
$:= 5 * 2$
$\therefore p := 10$
$q := m + q$ ←
$: 2 + 3$
$\therefore q := 5$

2) Use of algebraic identities:

Algebraic identities are used on peephole optimization techniques.

Eg: $a + 0 = a$
$a * 1 = a$
$a / 1 = a$

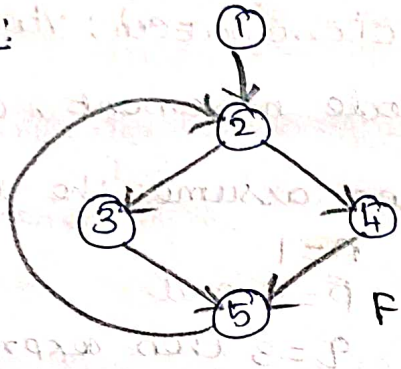
The algebraic transformation can be obtained using the strength reduction technique.

Eg:
Instead of using $2 * a$ we can use $a + a$
Instead of using $a / 2$ we can use $a * 0.5$

8) Loops in flow Graph:

① Dominators: In a flow graph, a node d dominates n if every path to node n from initial node goes through d only. This can be denoted as 'd dom n'.

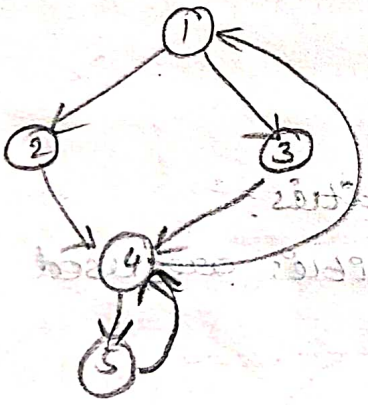
Eg:



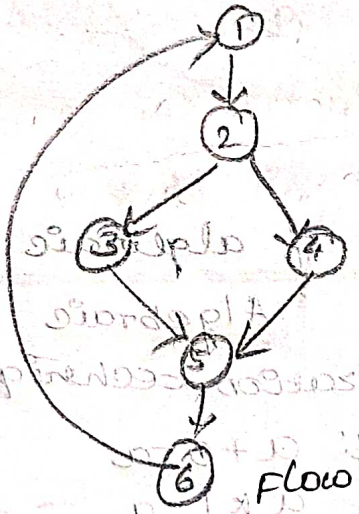
Flow graph.

② Natural loops:

Loops in a flow graph can be denoted by $n \to d$ such that d dom n. These edges are called back edges and for a loop there can be more than one back edges. $p \to q$ then q is a head and p is a tail.



Flow graph with loops

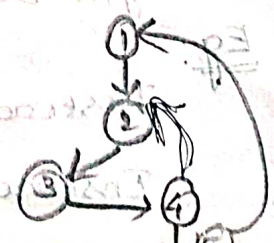


Flow graph with natural loop.

3) Inner loops:

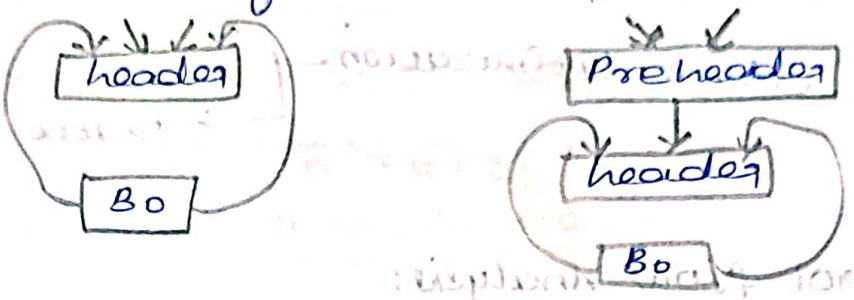
The inner loop is a loop that contains no other loop.

Here the inner loop is $4 \to 2$ that means edge given by 2-3-4



4) Pre-header:

The pre-header is a new block created such that successor of this block is the header block.



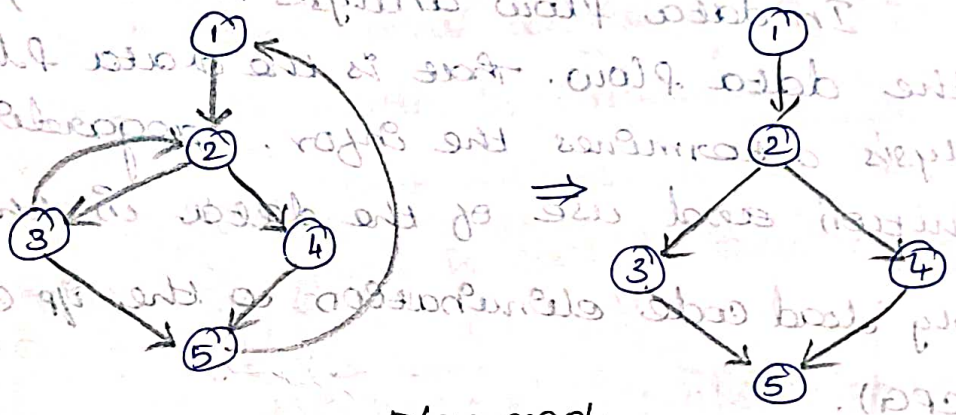
5) Reducible flow graph:

The reducible flow graph is a flow graph in which there are two types of edges forward edges and backward edges.

properties:

① The flw edge from an acyclic graph.

② The back edge are such edge whose head dominates their tail.



Flow graph.

9) Global data flow Analysis:-

The local optimization has a very restricted scope on the other hand the global optimization is applied over a broad scope such as procedure or function body.

There are two types of analysis performed for global optimization:

- Global optimization
  - Control flow analysis
  - Data flow analysis

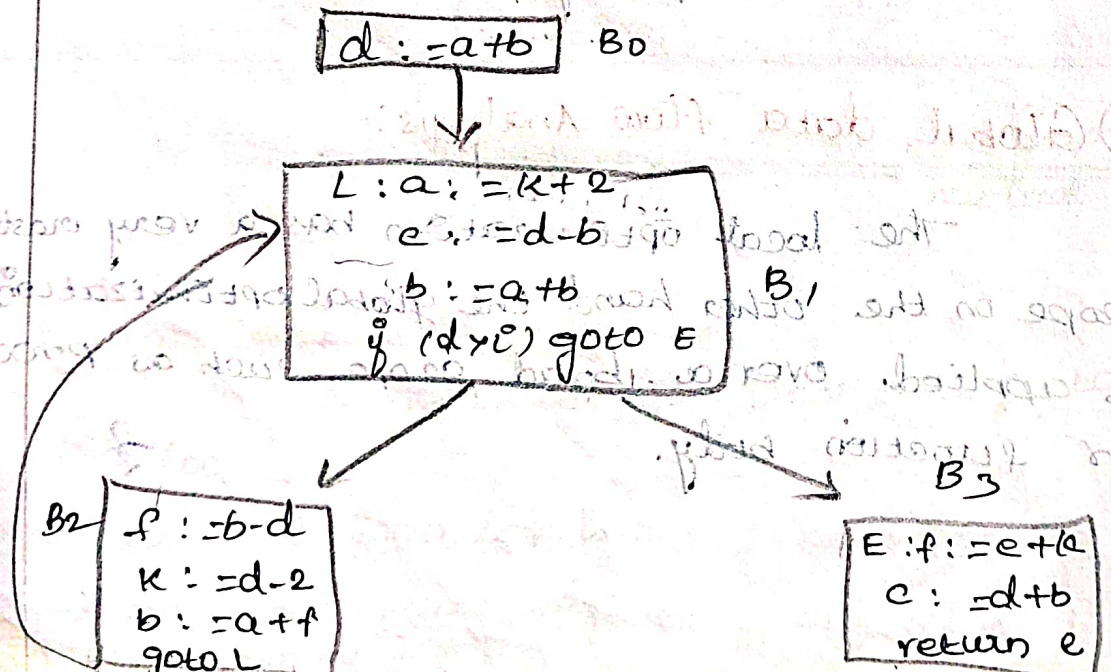
Control flow Analysis:

The control flow analysis determines, the info. regarding arrangement of graph nodes, presence of loops, nesting of loops and nodes visited before execution of a specific node.

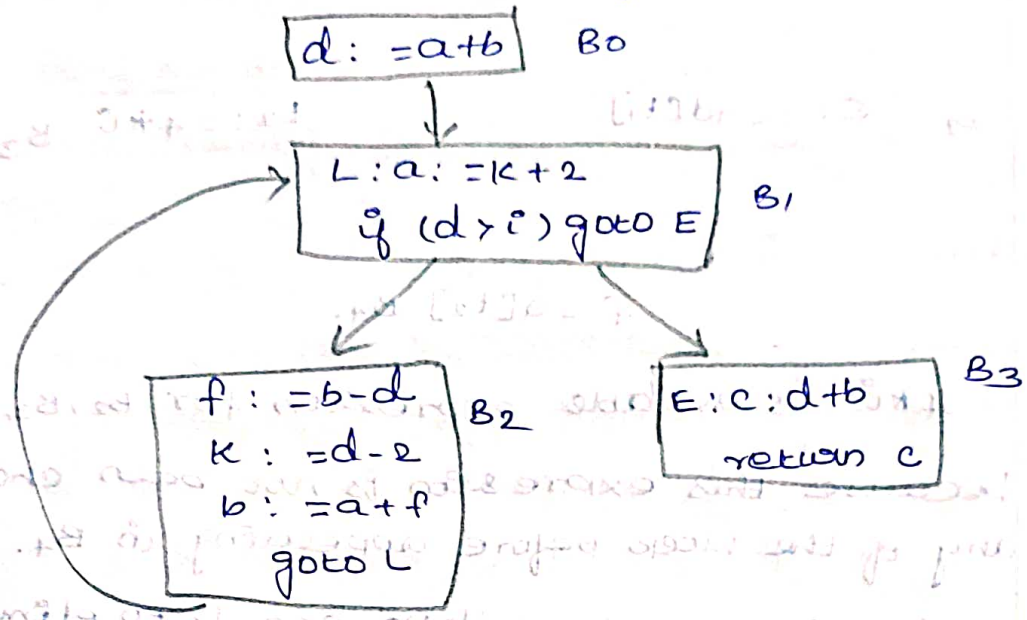
Thus control flow analysis is made on the flow of control by carefully examining the program flow graph.

In data flow analysis the analysis is made on the data flow. That is the data flow analysis determines the info. regarding the definition and use of the data in the pgm.

Apply dead code elimination to the i/p code given in (CFG).



dead code c: =d-b, similarly B_3 block f: =e+k has no meaning:

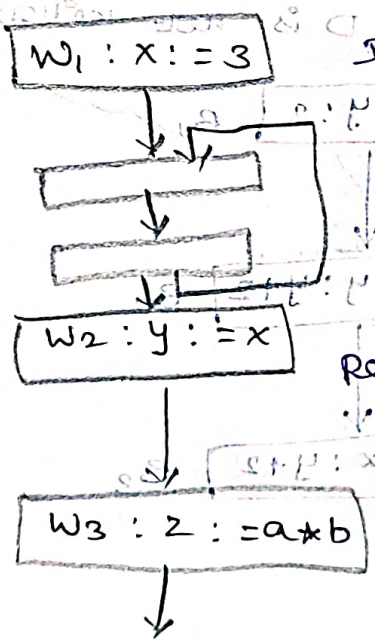


Data flow properties:

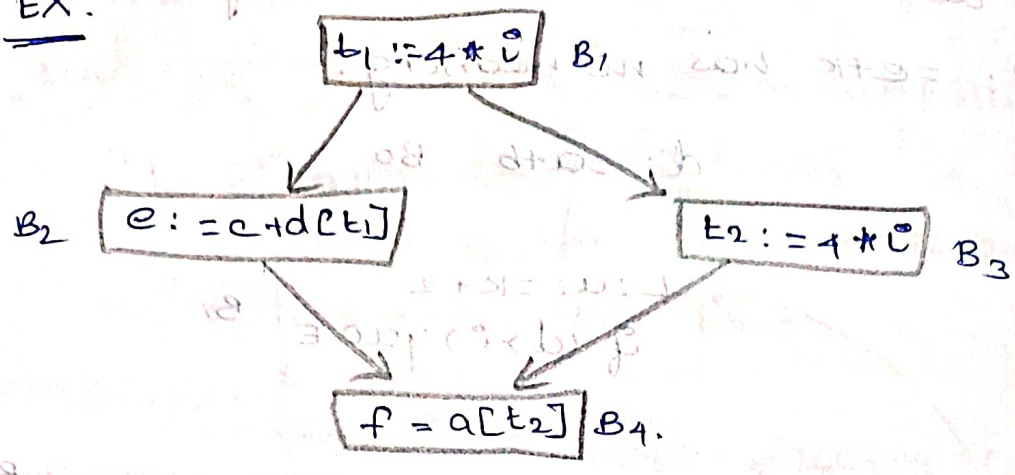
A pgm point containing the definition is called "definition point".

A pgm point at which a reference to a data item is made is called "reference point".

A pgm point at which some evaluating expression is given is called "evaluation point".



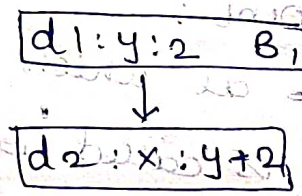
EX:



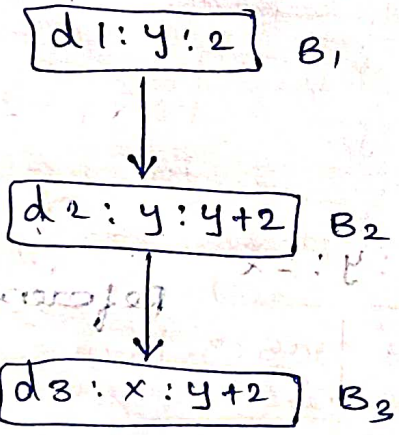
$4 * i$ is available expression for B_2, B_3, B_4, because this expression is not been changed by any of the block before appearing in B_4.

Adv: The use of available exp. is to eliminate common sub expressions.

2) Reaching definitions:



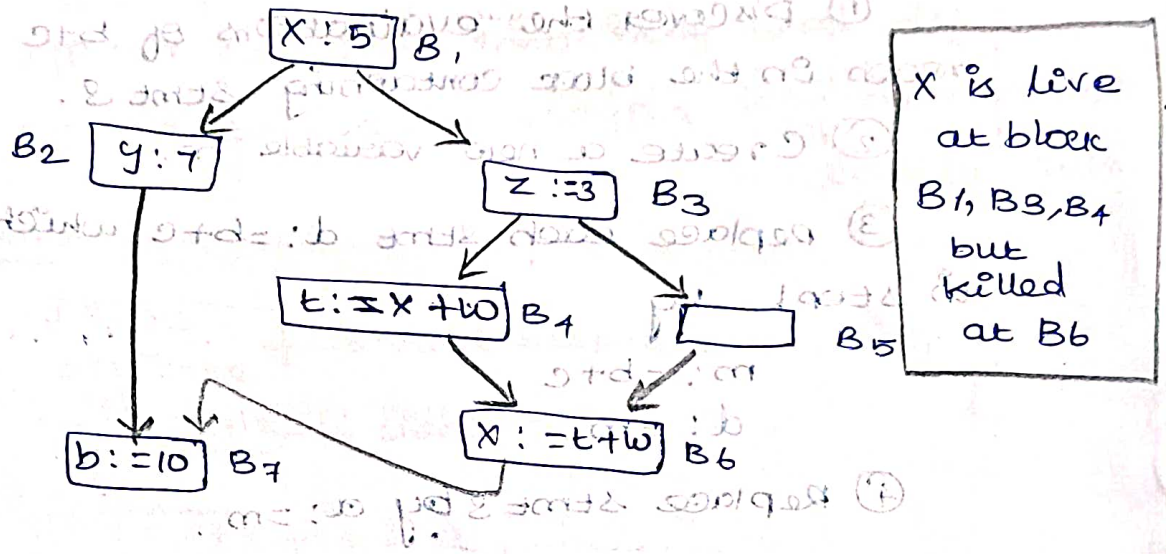
A definition D reaches at point P if there is a path from D to P if there is a path from D to P along which D is not killed.



Adv: Reaching definitions are used in constant and variable propagation.

3) Live variable:

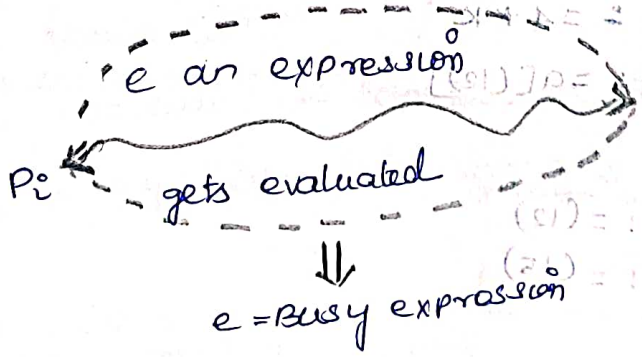
A variable x is live at some point p, if there is a path from p, to the exit, along which the value of x is used before it is redefined.



Adv: ① Live variables are useful in reg. allocation.
② Live variables are useful for dead code elimination.

4) Busy expression:

An expression e is said to be a busy expression along some path $P_i \dots P_j$ if and only if an evaluation of e exists along some path $P_i \dots P_j$ and no definition of any operand exists before its evaluation along the path.



11) Efficient Data flow Algorithms:

Redundant Common Subexpression Elimination:

Alg: Algorithm for global common exp. elimin

I/P: A flow graph with available expression.

O/P: A flow graph after eliminating common subexpr

① Discover the evaluations of b+c that reach an the block containing stmt s.

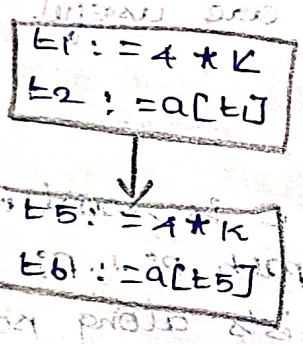
② Create a new variable m.

③ Replace each stmt d:=b+c which obtain in step 1. by

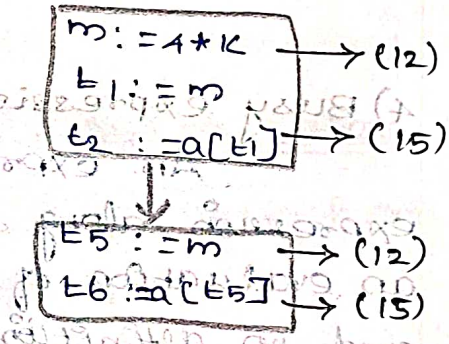
```
m := b+c
d := m
```

④ Replace stmt s by a:=m.

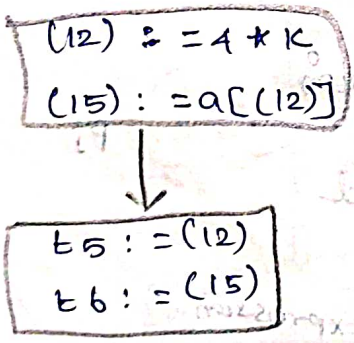
Step 1:



Step 2 & 3:



Step 4:



Copy propagation:

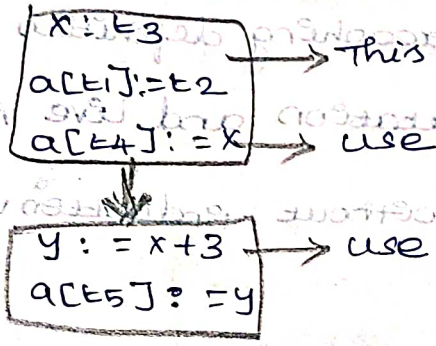
The assignment of the form a := b is called copy stmt. The idea behind the copy propg is to use b for a wherever possible after copy stmt a := b. Let us see the alg. for copy propagation.

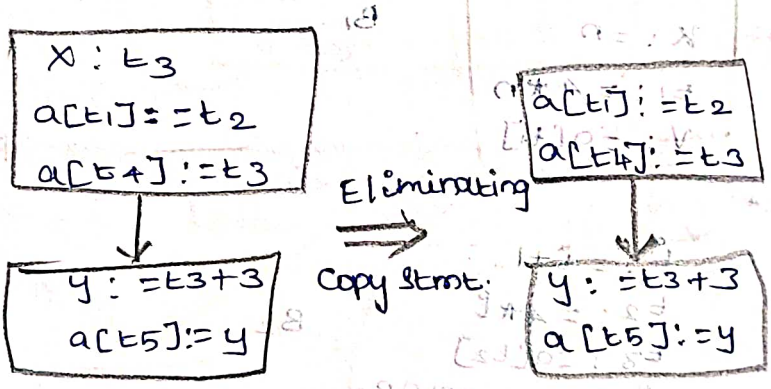
Alg: copy propagation.

I/p : A flow graph containing used - def.

o/p : A graph after applying copy propagation transformation.

Step1:





Induction variable:

A variable i is called an induction variable of loop L if every time the variable i changes values.

a := i + b
a := b * i
a := i - b
a := b - i.

a := i * b then the triple for a is (i, b, 0). We will understand this concept of Waiting triple with the help of block.

```
i := i+1
t2 := 4*i
t3 := a[t2]
if t3 < 10 goto B
```

(i , 4 , 0)

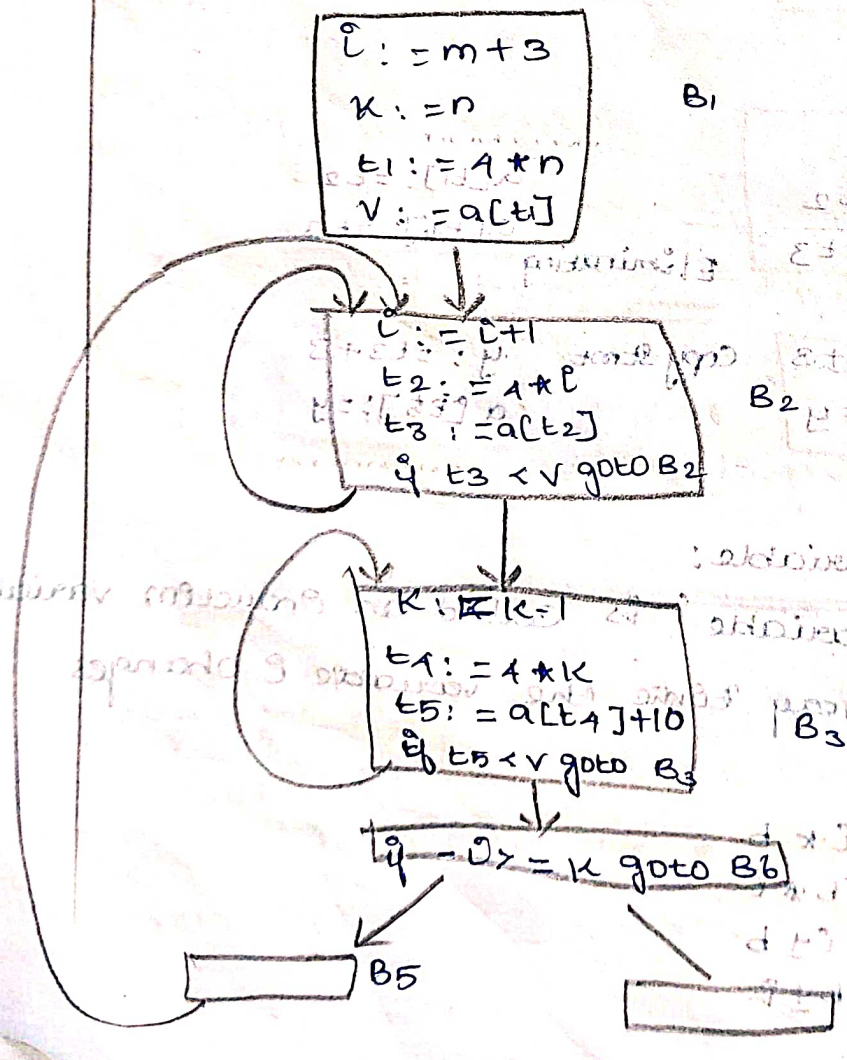
↓ Induction variable

↓ Matching with Constant b.

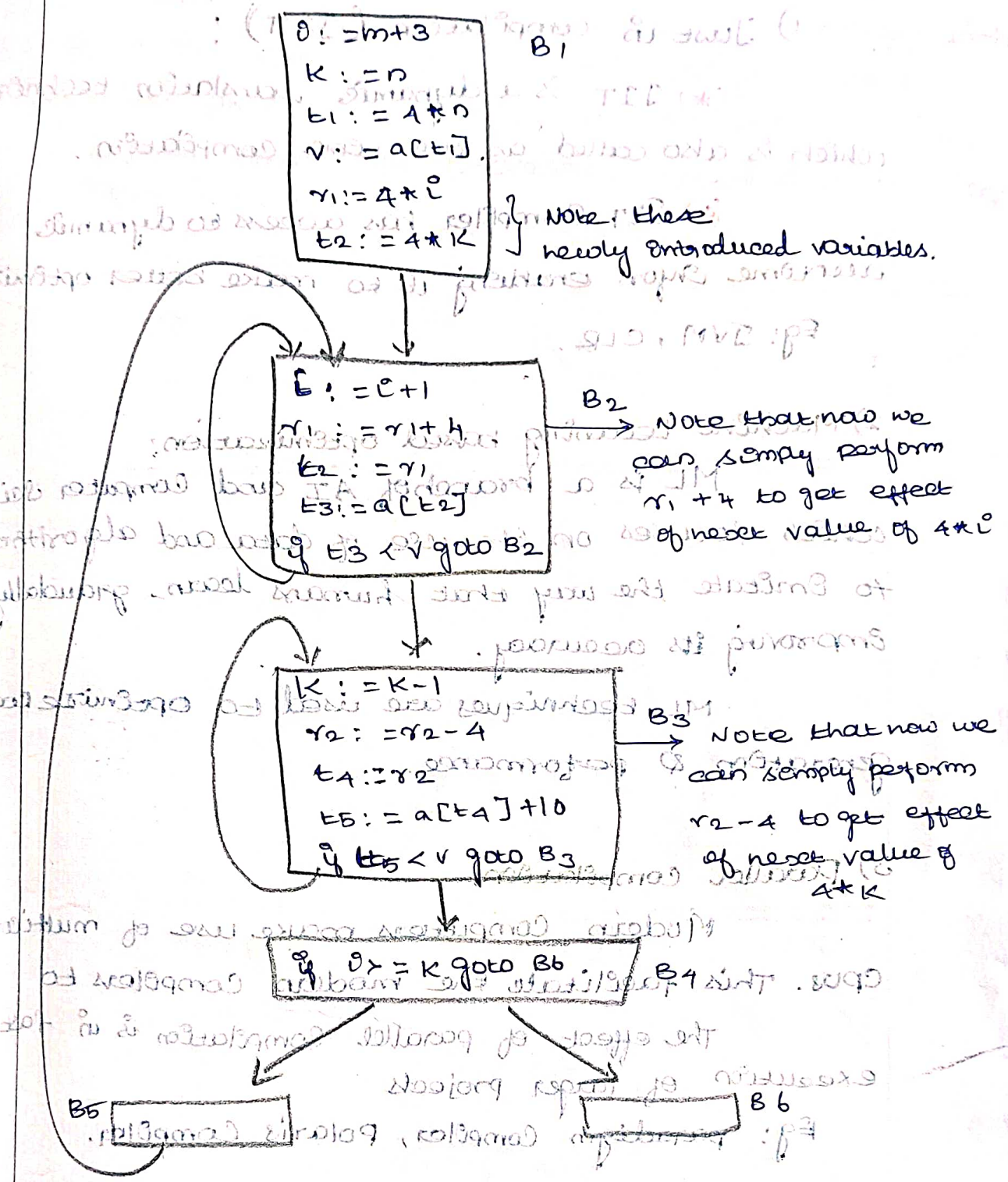
Alg: Elimination of induction variables.

I/P : A loop L with reaching definition info, loop invariant computation and live vari. info.

O/P: A flow graph without induction variables.



The flow can be rewritten as:



(14)

12) Recent Trends in Compiler Design:

1) Just in Compilation (JIT):

(*) JIT is a dynamic translation technique which is also called as run time compilation.

(*) JIT Compiler has access to dynamic runtime info: enabling it to make better optimiza

Eg: JVM, CLR.

2) Machine Learning based optimization:

ML is a branch of AI and Computer scien which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy.

ML techniques are used to optimize code generation & performance.

3) Parallel Compilation:

Modern Computers make use of multicore CPUs. This facilitate the modern Compilers to

The effect of parallel Compilation is in fast execution of larger projects

Eg: paradigm Compiler, Polaris Compiler.

4) Language - Independent Compiler Infrastructure:

The support of for multiple pgming langs make the modern compilers language Onderdent.

This allows the developers to use the same compilation tools to Compile and optimize the code generation across diff. languages

5) Advanced compiler analysis and optimization.

Today, we embrace new technology with great changes in architectural design, highly parallel processing, registers allocations, cache hierachy and high processing speed demands for better compilers.