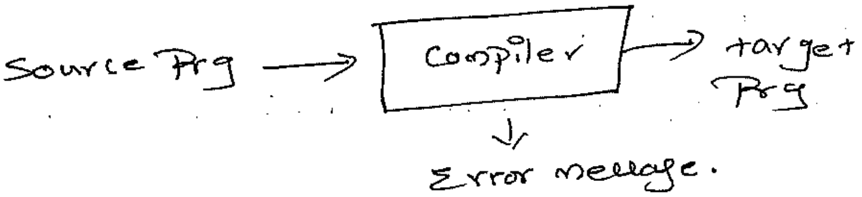
Compiler Design

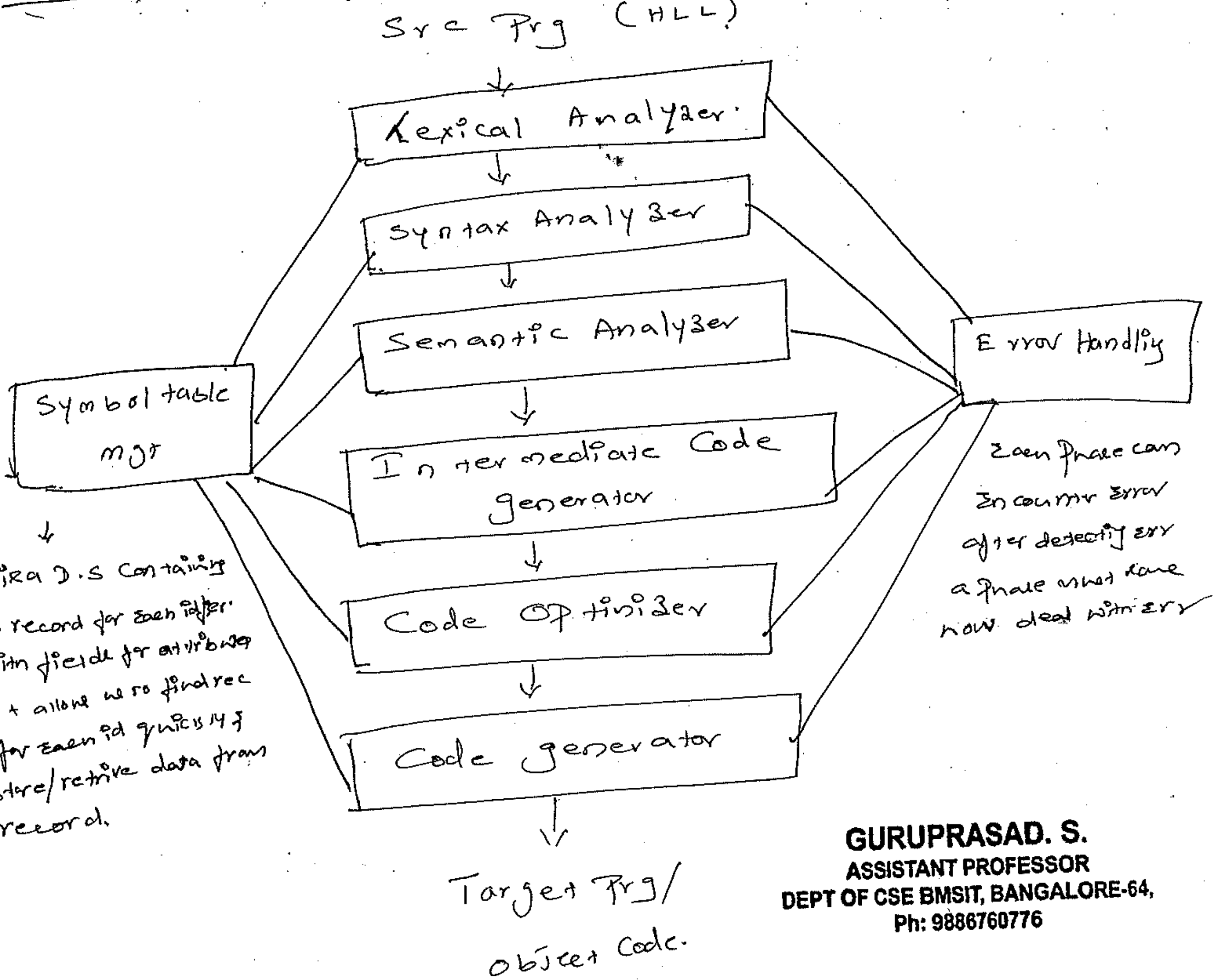
Compiler:

Compiler is a Program which accepts Source Program written in HLL and Produces object code/ target Program.



Since Compiler is a huge Program, & it is difficult to understand the entire Compilation Process, so the Process is divided into no of modules called Phases.

The Phases of Compiler / Structure of Compiler



it is a D.S containing a record for each idfer with fields for attributes. it allows us to find rec for each id quickly & for store/retrive data from record.

Each Phase can Encounter Error after detecting err a phase must some how deal with err

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

exical Analyser

- It Scans the src Prg from left to right & breaks the src Prg into maningfull tokens
- It removes Extra white spaces like tab, Space, new line
- → removes all comments.
- → Places/make entry for every Variable, Constants, label. into symbol table
- → An entry for a Variable/const is an orderd pair.

  < token name, attribute value>

  <id, 1>

yntax Analyser

- → It groups a set of tokens to identify a syntitacal structure defined by language.
- → It idfies the structure of the Prg from tokens obtained by Lexical Analyser.
- → It detects Syntax Errors & Produce Parsetree/ derivation tree

Semantic Analyser

- → It Checks for the Semantics of the identified Syntatical Structures.
- → checks for type checking, scope of Variables, Pointers & usage, Etc

Intermediate Code generation

- → Produces Explicit intermediate represention of Src Prg for abstract m/c
- → It should have two important Properties
  - → easy to Produce
  - → Easy to ~~understand~~. translate

Eg:- $D = A + B * C$

$T_1 = B * C$

$T_2 = A + T_1$

$D = T_2.$

Either three address repn or two addr repn is used.

Code optimization:

→ It attempts to improve intermediate code so that the target code runs faster & consume less memory space.

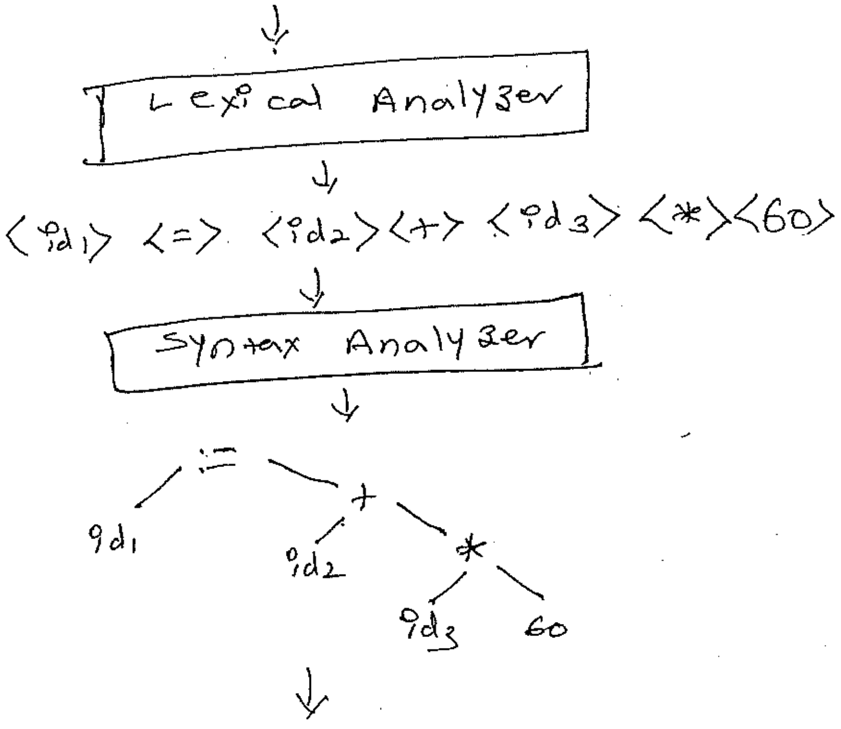
→ It uses techniques like local optimization, loop optzn, dead code elimination etc

Code generation:

→ It converts the optimized intermediate code into sequence of m/c instn,

→ This phase should know the m/c specific details & should utilize registers efficiently

Eg:- Position := initial + rate * 60



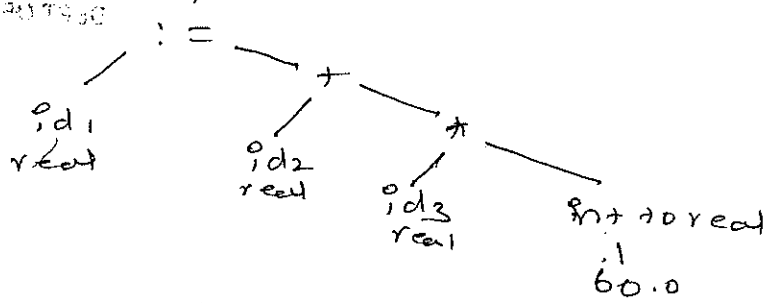
Symbol table

id	addr
Position	
initial	
rate	

↓

Semantic Analyzer

↓



↓

Intermediate code gen

↓

T_1 = int to real (60)

T_2 = id_3 * T_1

T_3 = id_2 + T_2

id_1 = T_3

↓

Code optimization

↓

T_1 = id_3 * 60.0

id_1 = id_2 + T_1

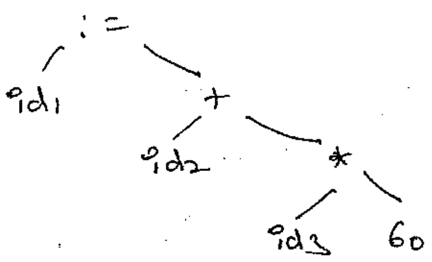
↓

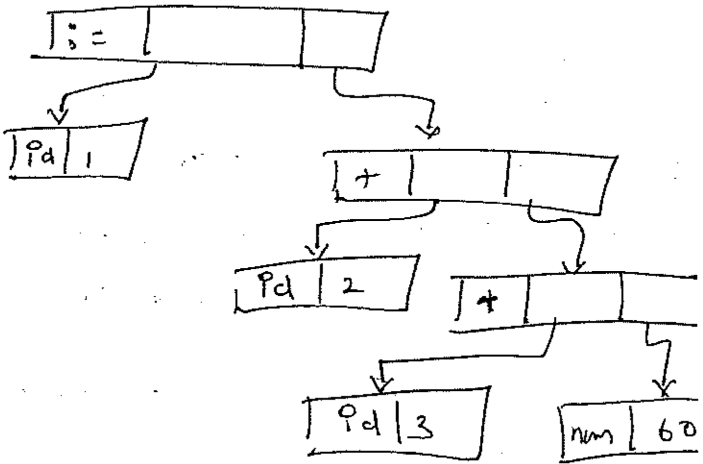
Code generation

↓

```
MOV F    id3, R2
MUL F    # 60.0 , R2
MOV F    id2, R1
ADD F    R2, R1
MOV F    R1, id1
```

Data Structure for Syntax Tree





→ Each operator is a **Node** with two Pointers to its left & right child.

→ A leaf is a record with two or more fields, one to idfy token & other to keep track of info about token.

The Analysis & Synthesis model of Compilation

The entire compilation Process is split into two Parts.

① Analysis

② Synthesis.

→ The Analysis Part breaks up the src Prg into Constituent Pieces, analyze it & Creates intermediate repn of src Prg.

→ The Synthesis Part Constructs the desired target Prg from the intermediate repn.

→ The Synthesis Part requires to know abt m/c dependencies & the language specific issues, so it uses a specialized technique.

Analysis of Src Prg

During analysis, the opn implied by src Prg are etermined & recorded in hierarchical structure called ree / Syntax tree where each node repn a opn & hildren repn arguments of opn

In compilation, analysis consists of three phases:

Linear / Lexical Analysis

Stream of characters of src Prg is read from left to right & grouped into tokens having collective meaning.

Eg:- Pos := initial + rate * 60

LA will generate following tokens.

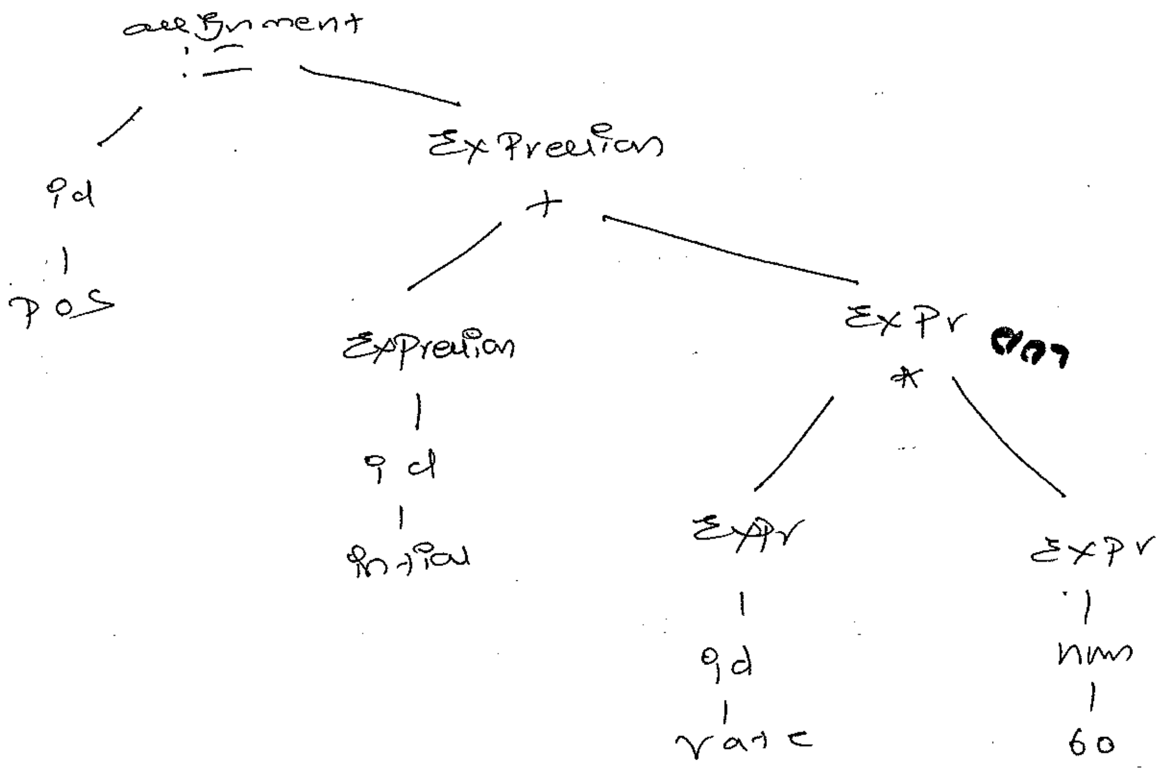
Pos — id
:= - assignment
initial - id
+
rate - id
*
60 - num

lank spaces & comments are eliminated.

Hierarchical Analysis / Syntax Analysis

> The tokens are grouped together hierarchically into nested collection to idfy a syntactical structure of Prg

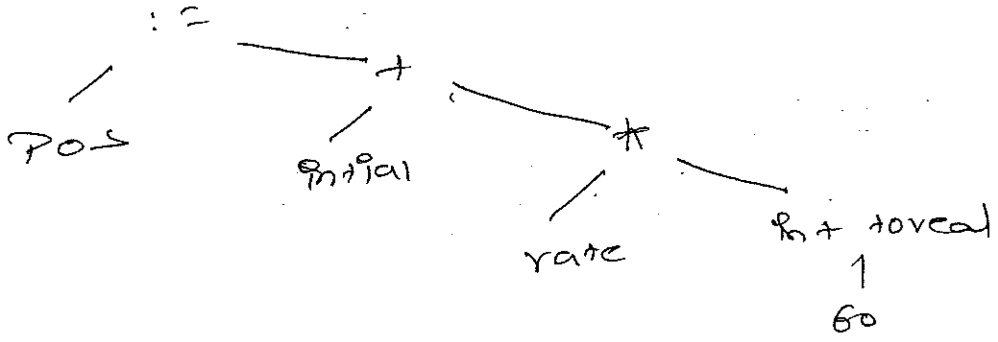
Eg:-



③ Semantic analysis

→ It check the semantics of the identified syntatical structure by Preforming type checking, scope resolution etc.

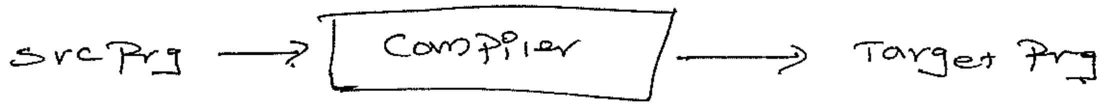
Eg:-



The Language Processors

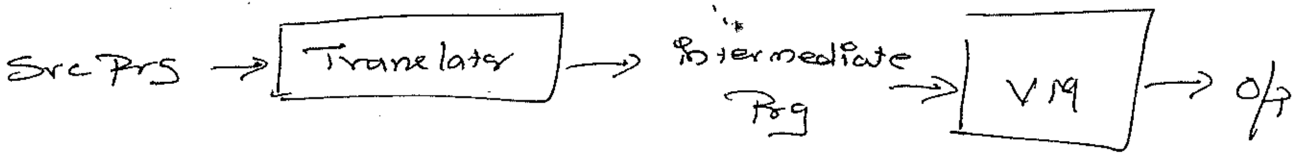
In addition to Compilers several other Programs may be required to create Executable target Program, like Preprocessors, assemblers, Loader & linker editors, Virtual machine etc.

Compiler:

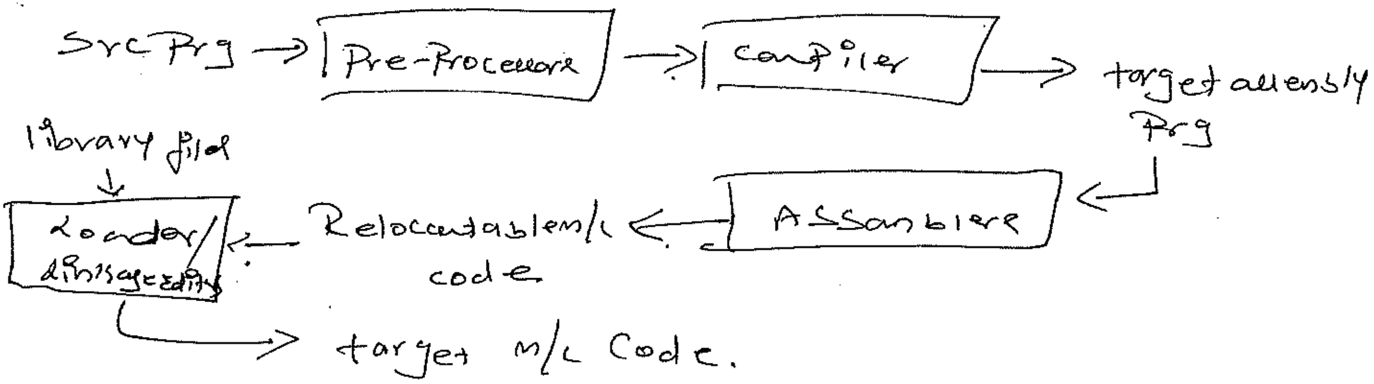


Interpreter:

VM:



Loader/Linkers:



Tools used by Compilers

① **Structure Editor:**

It takes sequence of commands as i/p to build a src Prg, it not only performs text creation & modfn fn but also analyzes Prg text, Putting an appropriate hierarchical structure on the src Prg.

It does additional tasks such as checking the i/p are correctly formed, can supply keyword automatically.

② **Pretty Printers:-**

It analyzes the Prg & Prints it in such a way that the structure of Prg becomes clearly visible. by indendation, changing font of comments etc.

③ **Static Checkers:-**

It attempts to detect errors/bugs with out actually running the Prg.

④ **Interpreters::**

It converts each line of src Prg to target Prg & executes it immediatly.

The Evolution of Programming Languages

The first electronic computers appeared in 1940s & were programmed in m/c language by sequence of 0's & 1's. The programming was slow, tedious & error prone.

he move to Higher Level Languages

The first step towards userfriendly Prg lang was development menomics & Assembly level languages in early 1950s, which ued Menomics to repn diff opns & Assemblers converted enomics to target m/c code.

It was Easier compared to m/c lang Prg but was m/c dependent enomics & instns.

The major step towards HLL was made in later half of 1950s with the development of languages like FORTRAN, COBOL

In the following decades many more. HLL were created with innovative features to help Programming Easier & robust.

Today there are thousands of Prg langs. they can be classified in variety of ways one classn is by generation.

I generation — m/c language

II generation — Assembly language

III generation — HLL like FORTRAN, COBOL, C, etc

IV generation — Lang designed for specific appn like NOMAD SQL

V generation — Language applied to Logic & Constraints like Prolog & OPS5.

OO — is a lang that supports object oriented Programming.

Impact on Compilers

Since the desgn of Prg lang & Compiler are intimately related, the advances in Prg lang placed new demands on compiler writers, compilers helped to promote use of HLL by minimizing the exn overhead of Prgs, A compiler must translate correctly the potentially infinite set of Prgs.

The science of building Compilers

① Modelling in compiler design & Implementation

math models & algorithm

The study of compilers is mainly a study of how we design the right mathematical models, & choose the right algorithms, while balancing the need of generality & power against simplicity & efficiancy.

Some of the fundamental models are finite state m/c & Regular Expression which are useful for describing the lexical units of Program.

CFG is used to describe the syntactic structure of Programming Lang.

② The science of Code optimization

Optimization refers to attempts that a compiler makes to produce code that is more efficient than the obvious code.

Compiler optimization must meet the following design objectives.

- The optimization must be correct, i.e it should Preserve the meaning of compiled Program.

- The optimization must improve the Performance of many Prgs.

- The compilation time must be kept resonable

- The effort required for opt^n must be manageable or should be eqt to the improve in performance.

Applications of Compiler Technology.

- Implementation of HLL Prg lage.
- Optimization for Computer architecture
- Design of new computer architecture & Inst^n sets
- Program translation
- S/W Productivity tools.

Compiler Construction Tools

1) Parser generators - automatically Produce syntax analyzer from a grammatical description of Prg Lang.

2) Scanner generators - Produce Lexical Analysis from Reg Exp^n description of tokens of a language.

3) Syntax-directed translation engines - Produce collections of subroutine & generates intermediate code.

4) Code generators - generates the target codes from a collection of rules & m/c registers

5) Data flow analysis Engine - facilitates the gathering of instructions flow & how values are transmitted from one part to another.

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64.
Ph: 9886760776

Programming Language Basics

1) Static / dynamic distinction

The most important issue that we face when designing a compiler for a language is what decisions can compiler make about a program.

If a language uses a policy that allows compiler to decide an issue, then we say that the lang uses Static policy. where the decisions are made during compile time.

If the compiler makes decisions during the Execution the then it is called Dynamic Policy.

```
fn ()        class A
{ int i      int i
  i++;       }
}            A  a1,a2,a3
```

2) Environments & States,

Another important distinction we must make when discussing programming language is whether changes occuring while the Prg runs will affect the value of data element or affect the interpretation of name for the data.

The association of names with locations in memory & then with values can be described by two mapping

$$\begin{tikzcd} \text{Names} \arrow[r, bend left, "\text{Environment}"] \arrow[r, phantom, "\text{dynamic}"', shift right=4] & \text{Locations} \arrow[r, bend left, "\text{State}"] \arrow[r, phantom, "\text{dynamic}"', shift right=4] & \text{Values} \end{tikzcd}$$

```
main ()
{ int i
  fn ()
  { int i
    i = 4
  }
  x = i+4
}
```

) Static Scope & Block Structure

The Static Scope are Public, Private & Protected.

Block is a grouping of declarations & statements in C

) Explicit Acess Control

Public, Private & Protected.

) Dynamic Scope ⇗

```
# define a (x+1)
  int x = 2;
void b() {int x=1; pf("%d",a);}
 void c() {pf("%d",a);}
   void main() {b(); c();}
```

It refers to following policy: A use of name x refers to the declration of x in the most recently called procedures with such a declration.

Eg:- Macro Expansion in the 'C' Pre-Processore.

) Parameter Passing machenisms

Call by Value - actual Parameters are Evaluated & copied into formal Parameters changes will not affect original value.

Call by reference - The address of actual parameters are passed to the called procedure the changes made will affect original value

Call by name - used in Early prg lg like Algol 60 ||r to Call by value

Copy Restore - ||r to call by value but the values are copied back while return

—x—

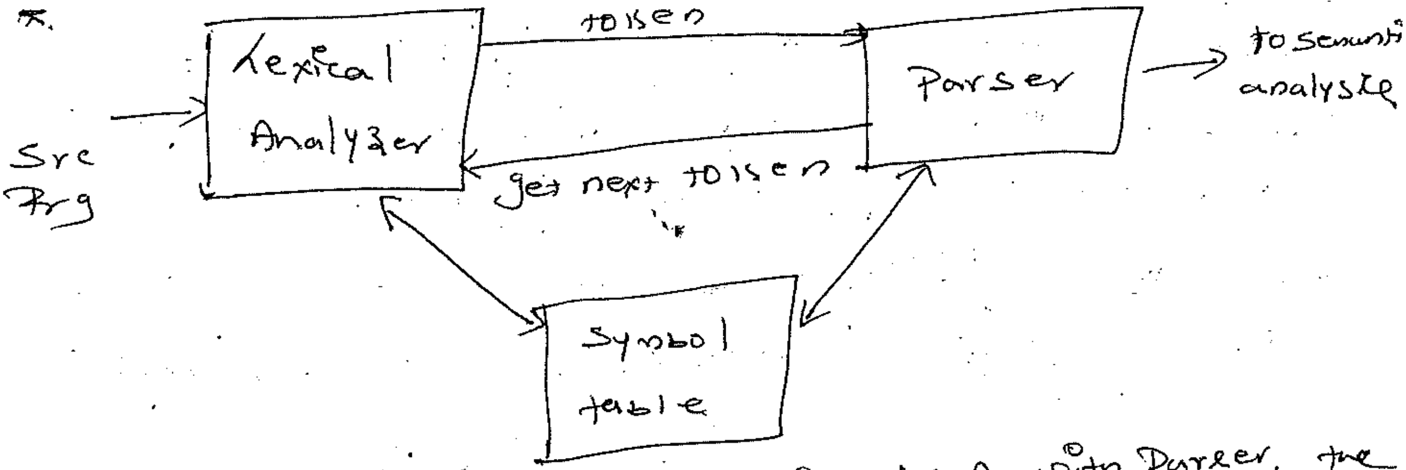
Lexical Analyser

The Role of Lexical Analyzer

* As the first Phase of compiler the main task of LA is to read the i/p char from Src Prg, group them into Lexemes, and Produce o/p as a sequence of tokens for each lexeme.

* It interacts with symbol Table to Enter the lexeme for identifiers constants Etc. & read the info of certain lexeme.

*



The fig depicts the interaction of LA with Parser, the Parser issues get next token cmd & the LA will read the chars from Src Prg until next token is recognized & is given to Parser.

* The LA Performs stripping out of Comments, Whitespace

Some times LA is devided into Cascade of two Processes:

① Scanning consists of simple Process that do not require tokenization of i/p such as deln of comments, whitespace etc

② LA Proper is more complex Portion, where scanner Produces the sequence of tokens as output.

Lexical Analyse v/s Parsing

There are no of reasons why analysis Portion is seperated into LA & SA (Parsing) Process.

- * simplicity of design
- * Compiler efficiancy is improved
- *. compiler Portabality is enhanced

Tokens Patterns & Lexemes

Token:

* It is the basic lexical unit of the Prg languge,
* It is a sequence of characters. that can be treated as a unit in the grammar of languge.
* Prg lang classifies tokens into finite set of tokentypes. like, keywords, identifiers, i/o statements, Punctuations Etc
* tokens are also known as lexemes.
* A token is a pair consisting of token name & an optional attribute value

Pattern:

* It is a description of the form that the lexemes of a token may take.

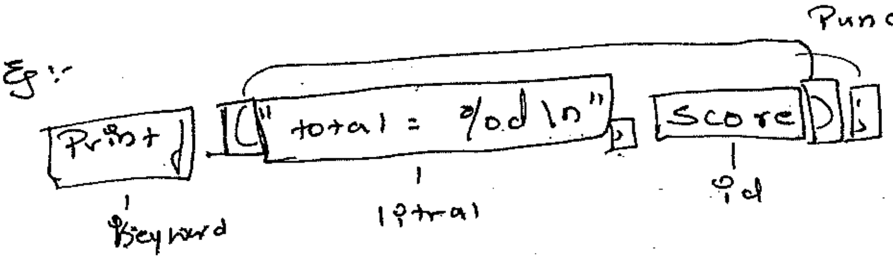
Eg:- ~~Profit display~~ int Emp_HRA

the identifier is a combination of characters.

Lexeme:

Is a sequence of characters in the src Prg that matches the Pattern for a token & is identified by LA as an instance of that token

Eg:-



Each Prg lang will have finite set of token type like

① keyword

② Operators

③ Identifiers.

④ constants, numbers, litrals.

⑤ Punctiation Symbol like , ; { ( ) Etc

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Attributes of Tokens

When more than one lexeme can match a Pattern the LA must Provide additional info about lexeme for subsequent Phases.

Eg: token type id diff info is associated with it like, type, locn first found, initial Value Etc. which is stored in ST & a Ptr to ST will be Provided with each id.

Eg:- E = M * C * * 2 the token & attributes are

<id, Ptr to ST Entry for E>
<assign_op>
<id, Ptr to ST Entry for M>
<Mul_op>
<id, Ptr to ST Entry of C>
<Exp_op>
<num, int val 2>

Lexical Errors

It is hard for LA to idfy errors in src code w/o aid of other components.

Eg:- fi (a == f(x)) ...

the LA cant tell wheather fi is misspelling of 'if' or undeclared fn name., in this case LA return fi as id & next phase detects it as error.

The simplest recovery strategy followed by LA is "Panic mode" recovery, where compiler deletes successive characters from i/p till it finds a well formed token.

Other possible error recovery actions are:

① Delete one character from remaining i/p
② Insert a missing character into the remaining i/p
③ Replace a character by another character.
④ Transpose two adjacent characters.

Input Buffering

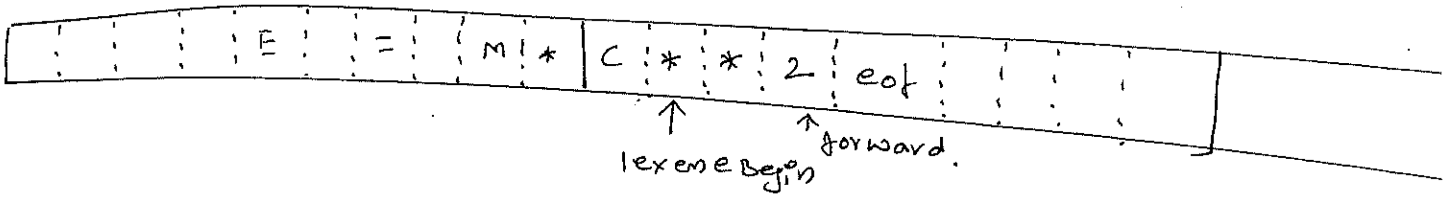
The task of reading the src prg has to be speeded up. but this task is made difficult because the LA must look one or more character beyond the next lexeme before it recognize the right lexeme.

Eg:- to idfy an id it should scan till it gets a sym which is not a dig/letter/ _

So to accomplish this task two - buffer schemes are used to handel large lookahead safely.

① Buffer Pairs

To avoid overhead required to Process single i/p char at a time this scheme uses two buffers, that are alternately reloaded.



Each buffer is of the same size N, usually (4096 bytes) using one System read command we can read N characters into a buffer, rather than using one system call per character.

If fewer than N characters remain in the i/p file, then spl char. repⁿ by eof.

two Pointers to the i/p is maintained.

① Pointer lexemeBegin, marks the beginning of the current lexeme,

② Pointer foreward scans ahead until a Pattern match found.

once a Pattern is matched it is returned to parser & beginPtr is set to the next char after Pattern & foreward Ptr is advanced till next

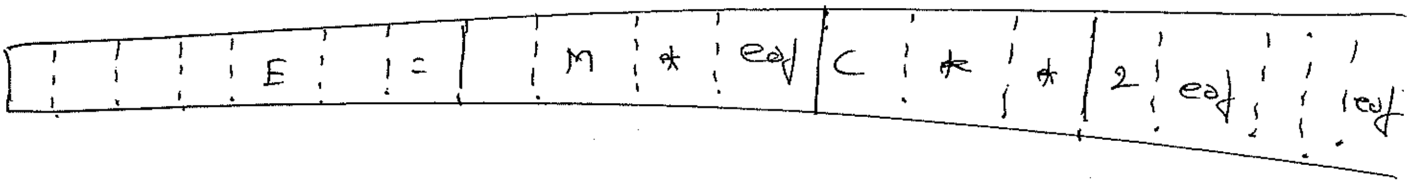
Sentinels

In i/p buffer pair, we must check each time we advanced foreward, that we have not moved off one of the buffers, if we do then we must also reload the other buffer, thus for each character read, we make two tests

- one for the end of the buffer.
- one to determine what char is read.

We can combine the buffer-end test with the test for the current char if we extend each buffer to hold a sentinel char at the end.

The sentinel is a special char that cannot be a part of the src Prg like eof



i.e one eof for each buffer. to repn End of buffer & one eof for end of Pattern

Specification of Tokens

Regular Expressions are an important notation for specifying lexeme patterns.

Strings & Languages

Alphabet: is a finite set of symbols 'Σ'

Eg:- {0, 1} binary alphabet {a-z A-Z} - letter.

{0-9} - digit

String:- is a finite sequence of symbols drawn from alphabet. length of String is repn as |s|

Empty string is denoted by ε or λ

String - Related terms:

① Prefix of String S: is any string obtained by removing zero or more symbols from end of S

Eg: S = banana

∴ Prefix ban bana, ~~ban~~ ba, ε

② Suffix of String S: is any string obtained by removing zero or more symbols from the beginning of S

Eg:- S = banana.

Suffix: nana, na, ε

③ Sub String of S: is obtained by deleting any Prefix & any suffix of String S.

④ Proper Prefixes, Suffixes & Substrings of S: are those of S that are not ε or S itself.

⑤ Sub Sequence of S: is any String formed by deleting zero or more not necessarily Consecutive Position of S

Eg:- S = banana

Subsequence bnna aaa nn

⑥ Concatination of String: if x & y are two strings then concatination is repn by xy

Eg x = dog y = house xy = doghouse

Operations on Languages

Language L - is set of finite strings formed by the fixed alphabet

Eg:- $\Sigma = \{0, 1\}$

$L = \{0, 0, 1, 11, 10, 1010, 0101, \ldots\}$

$\Sigma = \{a-z\}$

$L = \{$ alpha, beta, gama ... a, b, aa, bb ....$\}$

~~Clo~~

The Kleene closure of a Language L is denoted by L^*. is the set of strings got by concatinating L zero or more times.

$$L^* = \bigcup_{i=0}^{\infty} L^i \quad \boxed{or} \quad L = \Sigma^*$$

i.e it includes all possiable strings of language L and also null string ϵ

Positive Clasure of L is denoted by L^+ includes all strings of alphabets except nullstring

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

Union of L & M

$L \cup M = \{s \mid s$ is in L or is in M$\}$

concatination of L & M

$LM = \{st \mid s$ in L and t in M$\}$

Regular Expression

The RE are built recursively out of smaller RE. using the following rules.

Each RE r denotes a language $L(r)$

the rules that define RE over some alphabet Σ. & languages those expr^n denote.

Rule1: r $L(r)$

ϵ is a RE. and $L(\epsilon)$ is $\{\epsilon\}$

Rule2:

if a is a symbol of Σ then a is an RE where

$L(a) = \{a\}$

Rule3:

if r & s are two REs denoting $L(r)$ & $L(s)$ then

$(r)|(s)$ is a RE denoting $L(r) \cup L(s)$

Rule4:

if r & s are two REs denoting $L(r)$ & $L(s)$ then

$(r)(s)$ is a RE denoting $L(r)\,L(s)$

Rule5:

r^* is a RE denoting $(L(r))^*$

Eg:- $\Sigma = \{a, b\}$

$a|b$ denotes $\{a, b\}$

$(a|b)(a|b)$ denotes $\{aa, bb, ab, ba, bb\}$

a^* denotes $\{\epsilon, a, aa, aaa \ldots\}$

$(a|b)^*$

$L = \{\epsilon, a, b, ab, bb, aaaa\} \ldots$

$a|a^*b$

$\{a, ab, aaab \ldots\}$

Algebric laws of RE

LAW	DESCRIPTION
$r\|s = s\|r$	| is commutative.
$r(st) = (rs)t$	concatenation is associative
$r\|(s\|t) = (r\|s)\|t$	| is associative.

a) The unary operator * has highest Precedance & is left associative

b) concatination has second highest Precedance & left associative

c) | has lowest Precedance & left associative.

Regular Definations

To give names to certain RE & we use those names in subsequent expressions.

If Σ is alphabet of basic symbols then regular defination is a sequence of defination of the form.

$d_1 \rightarrow r_1$ where d_i is new symbol &

$d_2 \rightarrow r_2$ r_i is RE over alphabet Σ

....

$d_n \rightarrow r_n$

Eg:- C identifiers are strings of letters digits & underscore.

letter → A|B|.....|Z|a|b| ... |z|_

digit → 0|1| .....|9

id → letter_ (letter_ | digit)*

Eg2: unsigned numbers. (int or float) strings.

digit $\rightarrow$ $0|1|\cdots|9$

digits $\rightarrow$ digit digit*

optional fraction $\rightarrow$. digits $|\ \epsilon$

optional exponent $(E\ (+|-|\epsilon)\ \text{digits})\ |\ \epsilon$

number $\rightarrow$ digits optional fraction optional exponent

6.336E4 1.89E-4

Extensions of Regular Expression

Kleene introduced RE with basic operators of union & concitanation i.e | and . but many extensions are added.

① one or more instance.

if r is an RE then $(r)^+$ denotes $(L(r))^+$

$r^* = r^+ | \epsilon$ and $r^+ = rr^* = r^*r$

② zero or one instance.

if r is an RE then $r?$ denotes $L(r?) = L(r) \cup \{\epsilon$

i.e. $r? = r|\epsilon$

③ Character classes

$a|b|c$ is denoted by [abc]

and $a|b|c\ldots|z$ is denoted by [a-z]

Summary of RE

RE is specified on alphabet Σ.
following are ~~the~~ REs

ϵ	repn	$L(\epsilon) = \epsilon$
a	repn	$L(a) = \{a\}$
$(r)\|(s)$	repn	$L(r) \| L(s)$
$(r)(s)$	repn	$L(r)\,L(s)$
r^*	repn	$(L(r))^*$
r^+	repn	$(L(r))^+$.
$r?$	repn	$L(r) \cup \{\epsilon\}$
$[a-z]$	repn	$L = \{a\|b\|\ldots\ldots z\}$

Eg1:- letter → [a-zA-Z_]

dig → [0-9]

id → letter_ (letter_ | digit)*.

Eg2: dig - [0-9]

digs - $[0-9]^+$.

~~num = digs (.digs)? (E [+-~~

$$num = \underbrace{digs}_{dig}\ \underbrace{(.digs)?}_{fraction}\ \underbrace{(E\,[+\,-]?\ digs)?}_{exponent}$$

optional

Recognition of Tokens

Eg:- stmt → if expr then stmt
| if expr then stmt else stmt
| ε

expr → term relop term
| term

term → id | number

id:

letter → [a-z A-Z_]
dig → [0-9]
id → letter_ (letter|dig)*

number:
digs → [0-9]+
number = digs (.digs)? (E [+-]? digs)?

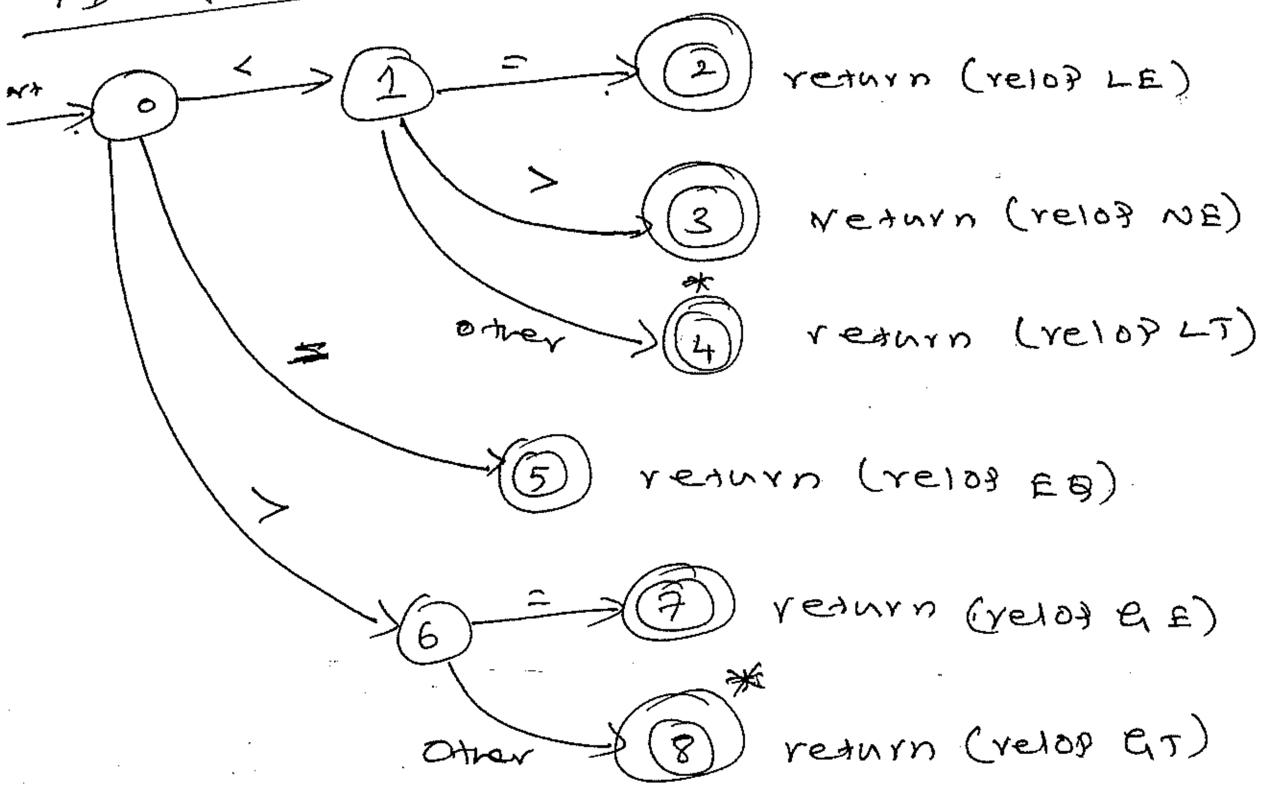
if → if then → then else → else.

relop → < | > | <= | >= | = | <>
LT GT LE GE EQ NE

An intermediate step in construction of LA is to convert RE to Transition Diagram or DFA

* TD have colln of nodes & edges, called states.
* Edges are directed from one state to another.
* Each edge is labeled by a symbol.
* certain states are said to be accepting or final states. which repn that lexeme has been found.

TD for Rel op



* indicates retract the foreward pointer one position means move back.

he next step is to implementation of TD

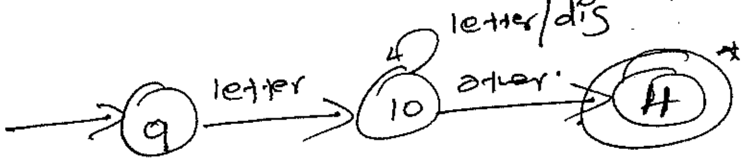
```
TOKEN getRelop ()
 {
    state = 0;
     for (;;)
     {
     switch (state)
      {
      case 0 :   c = getch();
                 if (c=='<')       state=1;
                else if (c=='=')   state=5;
                 else if (c=='>')  state=6;

                 else.  fail();
                 break;
    case 1:  c=getch();
              if (c=='=')      state=2;
            else if (c=='>')   state=3;
             else    state=4;        break;
```

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

```
Case 2:    return LE;  break;
Case 3:    return NE;  break;
case 4:    retract ();
             return LT;  break;
case 5:    return EQ;  break;
case 6:    c = getch ();
             if (c == '=') state = 7;
             else state = 8;
                 break
Case 7:    return GE;  break;
case 8:    retract ();
             return GT;
               break;
```

Recognization of Reserved words & Identifiers



return(gettoken(), install ID());

These TD recognizes id's & key words, there are two way that we can handle reserve words that looks like id

① Install the reserved words in the symbol Table initially.

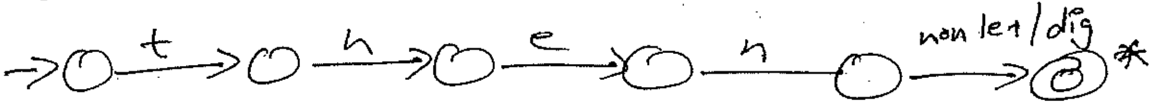
A field of the symbol Table Entry indicates that these strings (keywords) are never ordinary identifiers & tells which tokens they represent.

when we find an identifier. a call to Install ID Places it into ST & returns a pointer to symbol table Entry for lexeme found.

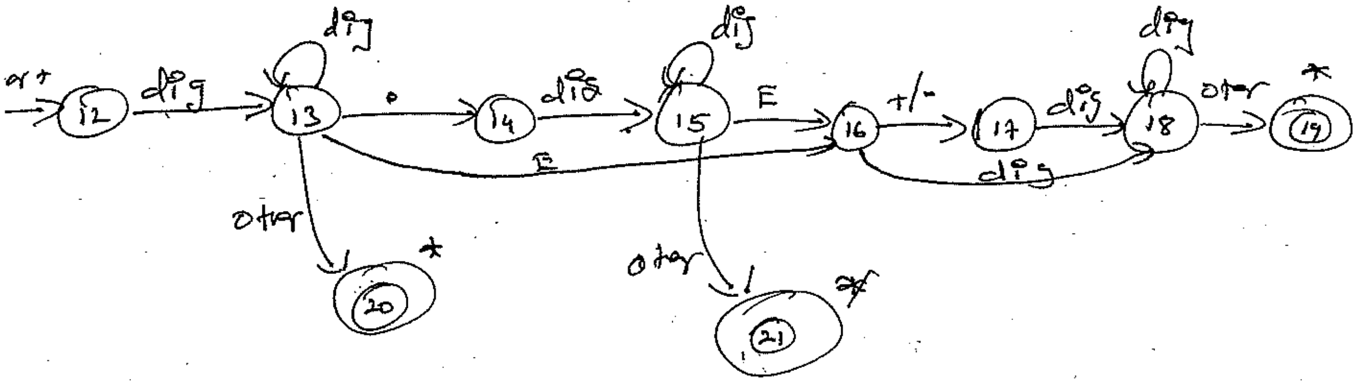
The function get Token examines the ST Entry for lexeme found & returns whatever token - Either id or key word.

) Create Seperate Transition Diagram for Each key word.

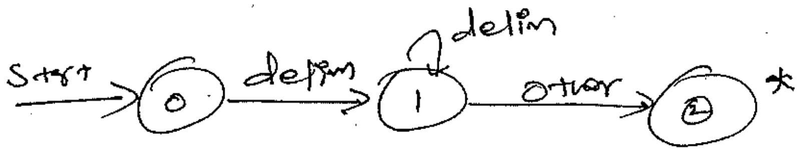
Eg :-



TD for unsigned numbers



TD for white space



delim = (blank| tab| newline)+

X

End of unit I

SYNTAX ANALYSIS

objectives:
we learn Parsing methods.
- basic concepts
- techniques for hand impl^n
- algorithms used by automated tools
- Error recovery techniques.

Preamble:

Every Prg lang has Precise rules that Prescribe the syntatic structure of Prg.

Eg:- C Prg is made up of functions, functions out of declr^n & stmts & stmts out of expressions & so on.

The syntax of Prg lang can be specified using Context Free Grammar CFG or BNF Backus Naur Form. Notation.

Grammar offer significant benefits for both language designers & Compiler writers.

→ Grammar gives precise yet easy to understand syntatic spf^n of Prg lang

→ From certain classes of grammar we can construct automatically an efficient parser that determines the syntatic structure of src Prg.

150

→ The structure of Prglang designed by grammar will help in translating src Prg to correct obj code & detect errors.

→ The grammar allows language to be evolved or developed iteratively by adding new constructs to perform new tasks.

10

Introduction:

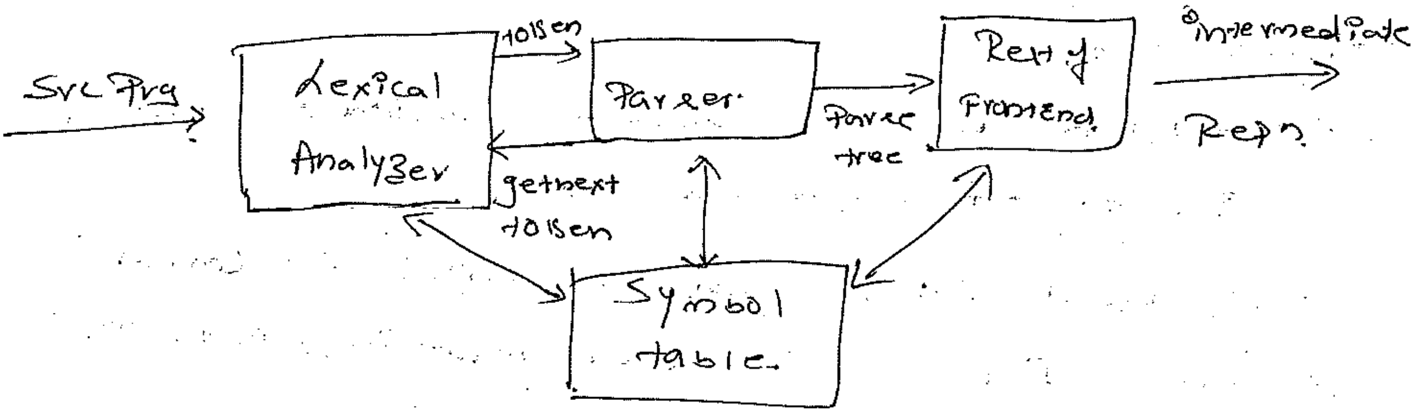
⇒ How Parser fit in Compiler.

⇒ typical grammar for arithmatic Exprn

⇒ Parsing technique for Exprn

⇒ Error handling.

The Role of Parser:



The Parser obtains strings of tokens from LA, and verifies that String of tokens can be generated by the grammar of src. language.

The Parser should report any Syntatical Errors in an intelligiable fashion & to recover from commonly occurring Errors to continue processing remainder of Program.

The Parser should generate Parse tree for well-formed Programs & pass it to rest of Compiler.

There are three general types of Parsers:

① Universal Parsers.

② Top down Parsers

③ Botten up Parsers

① Universal Parsing:

These methods such as Cocise - Yonger - Kasami algorithms & ~~also~~ Earley's algo can parse any grammar but are too in-efficiant to use in production compilers.

② Top down Parsing

They build the parse tree from Top (root) to the Bottom (leaves). the i/p is scanned from left to right one symbol at a time. they work for a sub class of grammar,

Eg:- LL Parser.

③ Bottom Up Parsing :-

They build parse tree from Bottom (leaves) towards Top (root) the i/p is scanned from left to Right one symbol at a time, works for sub class of grammars.

Eg:- operator Precedence, LR.

Many of BU Parsing is constructed using automated tools.

Representative Grammars

The parsing of Prg lang constructs like while int etc are easy as keyword guides the choice of grammar. So we concentrate on Expression because of associativity & Precedence of operators

$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to (E) \mid id.$$

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

This grammar belongs to the class of LR grammars that are suitable for BU Parsing.

The grammar is Left Recursive so cannot be used for Top Down Parsing, but after eliminating Left Recursion it can be used.

$E \Rightarrow TE'$

$E' \Rightarrow +TE' | \varepsilon$

$T \Rightarrow FT'$

$T' \Rightarrow *FT' | \varepsilon$

$F \Rightarrow (E) | id.$

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64
Ph: 9886760776

Syntax Error Handling

If compiler had to process only correct prgms the design & impln would be very simple. but practically it is not possible.

Planning the error handling right from start can both simplify compiler structure. & improve handling of errors.

Common Programming Errors that can occur at different levels:

* Lexical Errors - include misspellings of id, keywords, operators

* Syntatic Errors - misplaced semicolns, braces, etc.

* Semantic Errors - type mismatches improper use of pointers etc.

* Logical Errors - like use of assignment op = in place of comparison op ==

Several parsing techniques like LL & LR methods detect an error soon as possible

The goals of Error handler in Parser :

- Report the Presence of Error clearly & accurately
- Recover from each error quickly enough to detect subsequent Errors.
- Add minimal overhead to Processing of correct Programs.

Error Recovery Strategies.

Once the Error is detected, the simplest approach for the Parser is to quit with an informative Error message when it detects first Error. additional Errors are detected if it can restore from first error.

The various Error recovery Strategies are

Panic mode Recovery :-

On discovering an Error, the Parser discards i/p symbols one at a time until one of a designated set of tokens are Synchronizing tokens are found. like (; }) etc.

Panic mode correction often skips a considerable amount of i/p without checking it for additional Errors.

Phrase - level - recovery

On discovering an error the Parser performs local Correction on the remaining i/p by replacing Prefix by some string that allows Parser to continue.

Typical local correction is to replace comma by semicolon deleting extra semicolon Etc

But it is difficult in the situation where Error has occured before the Point of detection.

The goals of Error handler in Parser :

* Report the presence of error clearly & accurately
* Recover from each error quickly enough to detect subsequent errors.
* Add minimal overhead to processing of correct programs.

Error Recovery Strategies.

Once the Error is detected, the simplest approach for the Parser is to quit with an informative. Error message. when it detects first Error. additional errors are detected if it can restore form first error

The various Error recovery Strategies are

1) Panic mode Recovery :-

On discovering an Error, the Parser discards i/p symbols one at a time. until one of a designated set of tokens are Synchronizing tokens are found. like. (; }) etc.

Panic mode correction often skips a considerable amount of i/p without checking it for additional errors.

2) Phrase - level - recovery

On discovering an error the Parser performs local correction on the remaining i/p, by replacing prefix by some string that allows Parser to continue.

Typical local correction is to replace. comma by semicolon deleting extra semicolon Etc

But it is difficult in the situation where Error has occured before the point of detection.

rror Productions

By Considering Common Errors, we can argument e gramar with Productions that generate Erroneous onstructs.

When error occurs the parser can generate a iagnostics of the Erroneous construct that has been recognized.

lobal Correction

They are the algorithms for choosing minimal equence of changes to obtain a globally least cost orrection.

Given an incorrect i/p string x and grammar G. ese alg will find a parse tree for a related string y ch that no of insertions, deletions & changes of tokens q'd to transform x into y is as small as possible.

These methods are in generally too costly to implement in terms of time & Space.

ontext Free Grammars

CFG is used for formal description of structure of prg lang

set of strings constructed using tokens

Consists of terminals, non terminals, Start symbol & Productions.

formally defined as a quadruple $G = (NTPS)$

N - set of non terminals

T - set of terminals

P - set of Productions

S - start state

Productions is of the form $A \rightarrow \alpha$. where $A \in N$ & $\alpha \in (N \cup T)^*$

Notational convention used:

a, b, c ... 1,2,3, id, if ... used for terminals

+, -, *, => ...... used for operators

A, B, C, Q, T ... used for non terminals.

S - is start state

α, β, γ repn grammar symbols like $A \rightarrow \alpha_1 | \alpha_2 \ldots$

Eg:-

$$G: \begin{aligned} E &\rightarrow E+T \mid E-T \mid T \\ T &\rightarrow T*F \mid T/F \mid F \\ F &\rightarrow (E) \mid id \mid num \end{aligned}$$

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64
Ph: 9886760776

Derivation:

Construction of Parse tree is done by deriving the required string from the grammar rule, in every step the non terminal is replaced by its productions.

Eg:- derive string ctr + (a-b)/2.

$$\left.\begin{aligned} E &\rightarrow E+T \\ &\rightarrow T+T/F. \\ &\rightarrow F+F/id \\ &\rightarrow id+(E)/id \\ &\rightarrow ctr+(E-T)/2 \\ &\rightarrow ctr+(T-F)/2 \\ &\rightarrow ctr+(F-b)/2 \end{aligned}\right\} \text{Sen final form of } G$$

$$\rightarrow ctr+(a-b)/2 \;\}\; \text{sentence of } G.$$

$A \Rightarrow \alpha\beta$ represents that $\alpha\beta$ is derived from A in single step

$A \xRightarrow{*} \alpha\beta$ rep^n that $\alpha\beta$ is derived from A in zero or more steps.

$A \xRightarrow{+} \alpha\beta$ rep^n that $\alpha\beta$ is derived from A in one or more steps.

if $A \xRightarrow{*} \alpha$ and

α contains only terminals then it is <u>sentence of gramar</u>

α contains both terminals & Non terminals then it is sentinal form of G.

<u>Left most Derivation</u> (LMD)

while deriving a string from a grammar G if we start replacing left most non terminal every time then it is called LMD

rep^n by $S \xRightarrow{L} \alpha$.

Each step in the derivation is called <u>Vaiable Prefix</u> & the portion (non terminal) replaced is called <u>HANDLE</u>.

Eg:-

$S \rightarrow S ; S$

$S \rightarrow id = E$

$S \rightarrow Print (L)$

$E \rightarrow id \mid num$

$E \rightarrow E + E$

$E \rightarrow (S, E)$

$L \rightarrow E$

$L \rightarrow L , E$

Using LMD derive the string:

X := (Y := 1, Y + 2); Print (X + 10, Y)

S → S ; S

S → id = E ; S

S → id = (S, E) ; S

S → id = (id = E, E) ; S

S → id = (id = num, E) ; S

S → id = (id = num, E + E) ; S

S → id = (id = num, id + num) ; S

S → ———— 〃 ————; Print (L)

S → ———— 〃 ————; Print (L, E)

S → ———— 〃 ————; Print (E, E)

S → ———— 〃 ————; Print (E + E, E)

S → ———— 〃 ————; Print (id + num, id)

S → id = (id = num, id + num); Print (id + num, id)

$$\boxed{S \xRightarrow{+L} X = (Y = 1, Y + 2);\ \text{Print}\,(X + 10, Y)}$$

Right most Derivation (RMD)

While deriving string from grammar G if we replace the right most non terminal every time then it is RMD

repn by $S \xRightarrow{R} \alpha$.

Eg!-

S → S; S

S ⇒ S; Print (L)

S ⇒ S; Print (L, E)

S ⇒ S; Print (L, id)

S ⇒ S; Print (E, id)

S ⇒ S; Print (E + E, id)

S ⇒ S; Print (id + num, id)

S ⇒ id = E ; ——"——.

S ⇒ id = (S, E); ——"——.

S ⇒ id = (S, E + E); ——"——.

S ⇒ id = (S, id + num); ——"——

S ⇒ id = (id = E, id + num); —"——

S ⇒ id = (id = num, id + num); Print (id + num, id)

$$\boxed{S \overset{R}{\Longrightarrow} x = (y = 1,\ y + 2);\ \text{Print}\ (x + 10,\ y)}$$

ree Tree:

> A parse tree is a graphical representation of derivation quence. showing how the string is derived from grammar m start state.

> Each interior node of parse tree represents the application Productions

> Nodes repn Nonterminals & leaves repn terminals.

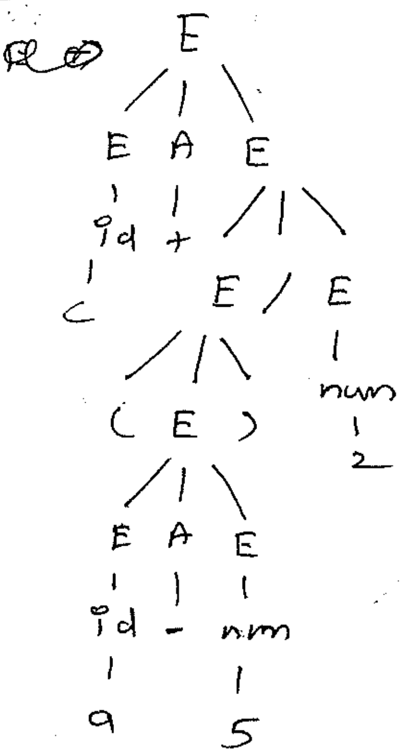
> The string derived from parse tree is called $\boxed{\text{Yield}}$.

E → E A E | (E) | id | num

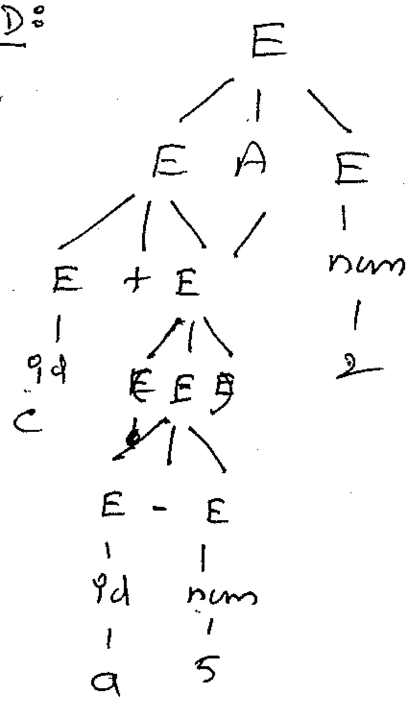
A → + | - | * | /

Derive Parse tree for c + (a-5)/2 using RMD & LMD

MD:

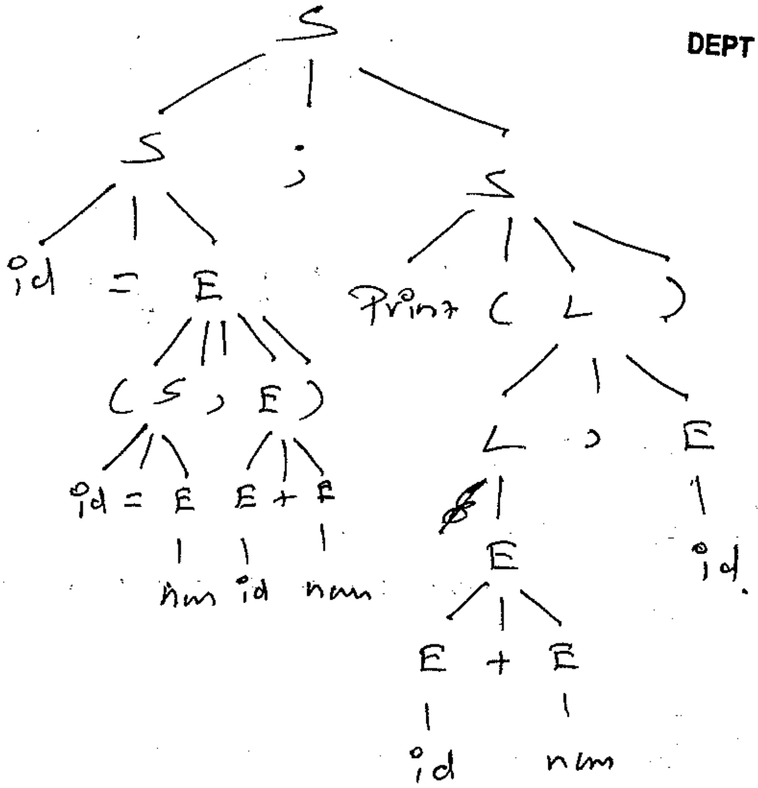


RMD:



GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

2:



See another 900b in page and

and | String be obtained by reading leaves from left to right

Ambigaity of Grammar

A grammar that Produces more than one Parse tree for same sentence is said to be ambigious.

or.

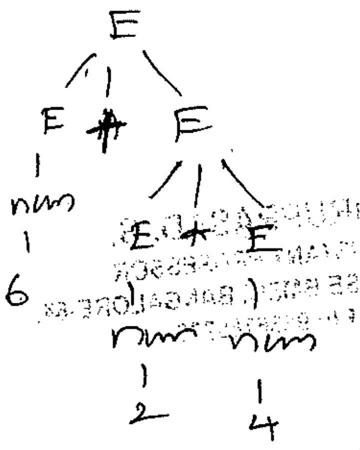
An ambigious grammar Produces more than one LMD or more than one RMD for same sentance.

Eg:- G: $E \rightarrow EAE \mid (E) \mid id \mid num$

$A \rightarrow + \mid - \mid * \mid /$

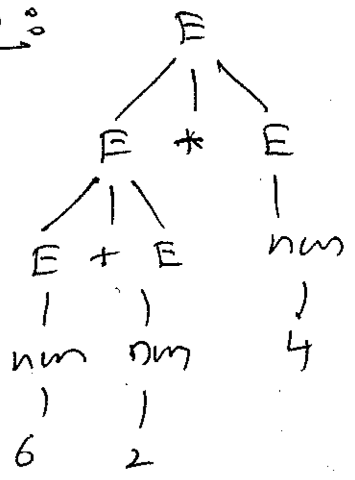
derive string $6+2*4$ using LMD & RMD

LMD:



$6 + (2*4) = \boxed{14}$

RMD:



$(6+2) * 4 = \boxed{32}$

Ambigious IF:

Context free

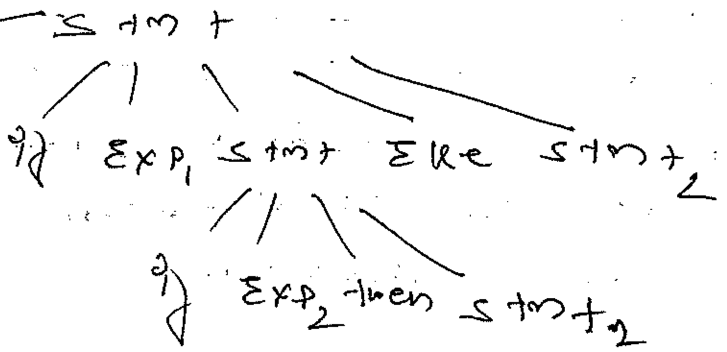
G: Stmt → if Exp then Stmt |
if Exp then Stmt Else Stmt | other Stmt

Derive String : if Exp1 then if Exp2 then Stmt1 Else Stmt2

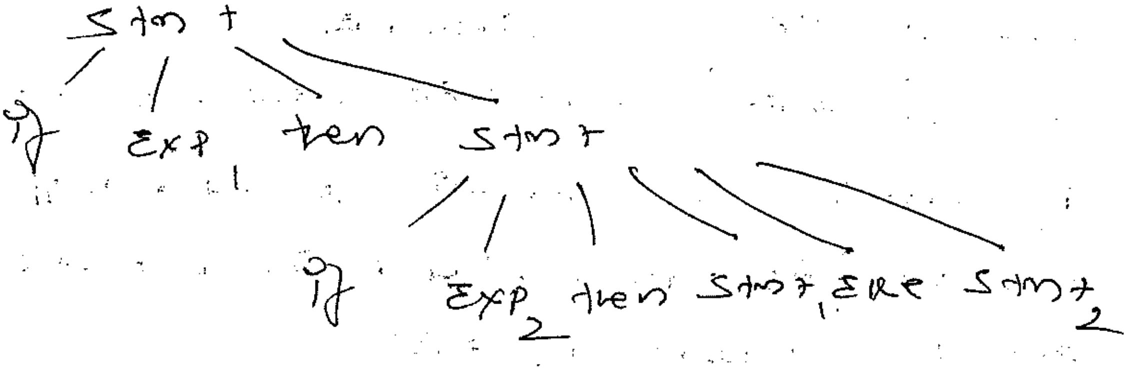
2 derivations:

~~Stmt if Exp then~~

Prodn 1:



Prodn 2:



Eg:- if (x != 0) then
if (y = 1/x) then ok := true
else
z = 1/x;
end if

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64
Ph: 9886760776

if the Else part is associated with first if stmt then we get divide by zero Error so the Else Part has tobe attached to the second if stmt (there is ambiguity)

so Prodn2 will yield correct string.

Ambiguity of grammar is because of 2 reasons

① LMD & RMD Exp

② the way grammar is designed. if stmt

CFG vs RE

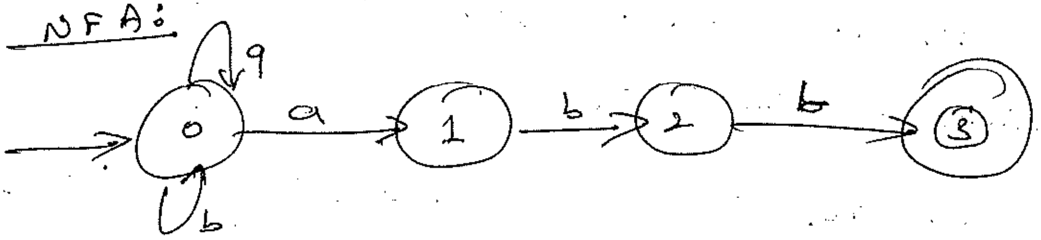
Grammars are more powerful notation than R.E, Every onstruct that can be described by R.E can be described y the grammar but not vice versa, Every Regular Language s a CFL, but not vice versa.

To convert RE to CFG :-

* Construct NFA for the given RE.
* for each state i of the NFA create a non terminal A_i
* if state i has a transition to state j on i/p a, add the Production $A_i \rightarrow a A_j$. If state i goes to state j on i/p ϵ add the Production $A_i \rightarrow A_j$
* if i is the an accepting state add $A_i \rightarrow \epsilon$.
* if i is the start state make A_i as start symbol of grammar

Eg :- ① $(a|b)^* abb$

NFA:



CFG:

$A_0 \rightarrow a A_0 | b A_0 | a A_1$

$A_1 \rightarrow b A_2$

$A_2 \rightarrow b A_3$

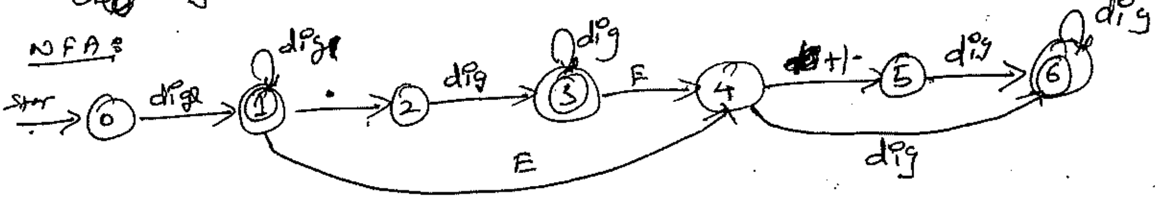
$A_3 \rightarrow \epsilon$

Convert the RE to CFG

Eg ②: ~~dig+ (.dig~~

~~dig~~ digs (.digs)? (E [+-]? digs)?

NFA:



CFG:

$A_0 \rightarrow dig\, A_1$

$A_1 \rightarrow dig\, A_1 \mid .\, A_2 \mid E\, A_4$

$A_2 \rightarrow dig\, A_3$

$A_3 \rightarrow dig\, A_3 \mid E\, A_4$

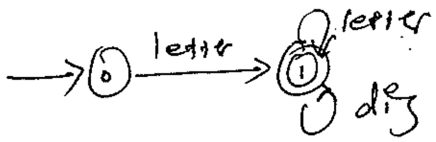
$A_4 \rightarrow dig\, A_6 \mid +A_5 \mid -A_5$

$A_5 \rightarrow dig\, A_6$

$A_6 \rightarrow dig\, A_6 \mid \epsilon$

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

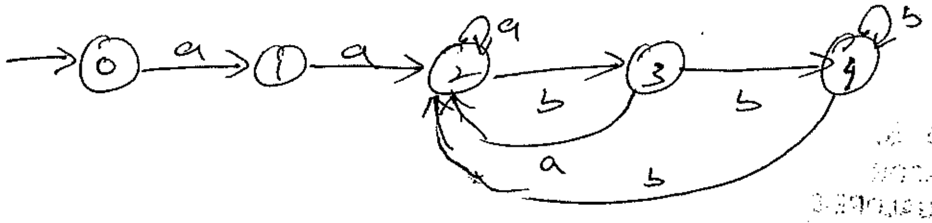
Eg ③: id → letter (letter | dig)*



$A_0 \rightarrow letter\, A_1$

$A_1 \rightarrow letter\, A_1 \mid dig\, A_1 \mid \epsilon$

Eg ④:
atleast 2 a's Ending with atleast 2 b's



$A_0 \rightarrow a A_1$

$A_1 \rightarrow a A_2$

$A_2 \rightarrow a A_2 \mid b A_3$

$A_3 \rightarrow a A_2 \mid b A_4$

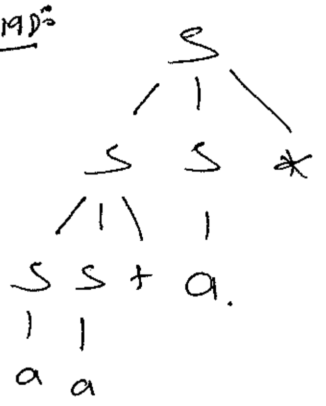
$A_4 \rightarrow a A_2 \mid b A_4 \mid \epsilon$

For the grammar construct LMD RMD Parse tree for ~~given~~ string and check whether grammar is ambigious.

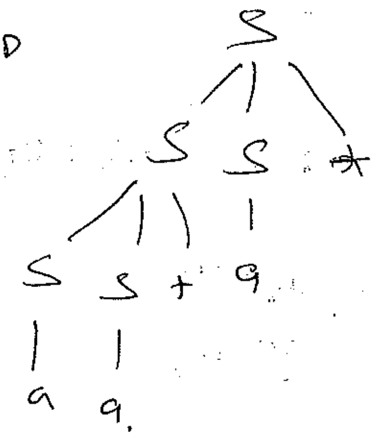
Eg: $S \rightarrow SS+ \mid SS* \mid a$

String aa+a*

LMD:



RMD



Both LMD & RMD yielde the same Parse tree so the grammar is un ambigiare.

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64.
Ph: 9886760776

Writing a Grammar:

-> Grammars are capable of describing most, but not all of the syntax of Prg lang

Eg:- id must be declared before its use cant be expressing grammars

-> such things are checked by next phases of compiler.

Lexical v/s Syntax Anlyses

-> LA & SA are seperated due to modularizing compiler

-> LA ~~rules~~ rules are quite simple so we use RE not CFG.

-> RE provides concise & easier to understand notation for tokens.

-> Efficent LA can be constructed automatically from RE than with CFG.

Note:

There are several transformations have to be applied to grammar to make it suitable for parsing like.

-> Eliminating Ambiguity

-> Eliminating left recursion

-> left factoring.

Eliminating Ambiguity

There are two solutions to Eliminate Ambiguity

① Re-write Grammar - for context sensetive Ambiguity

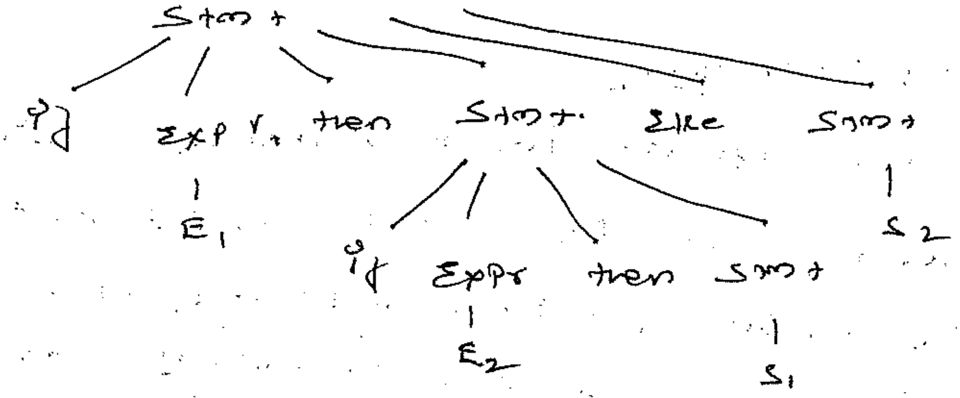
② ADD Precedance - for Context free Ambiguity.

Rewrite Grammar:

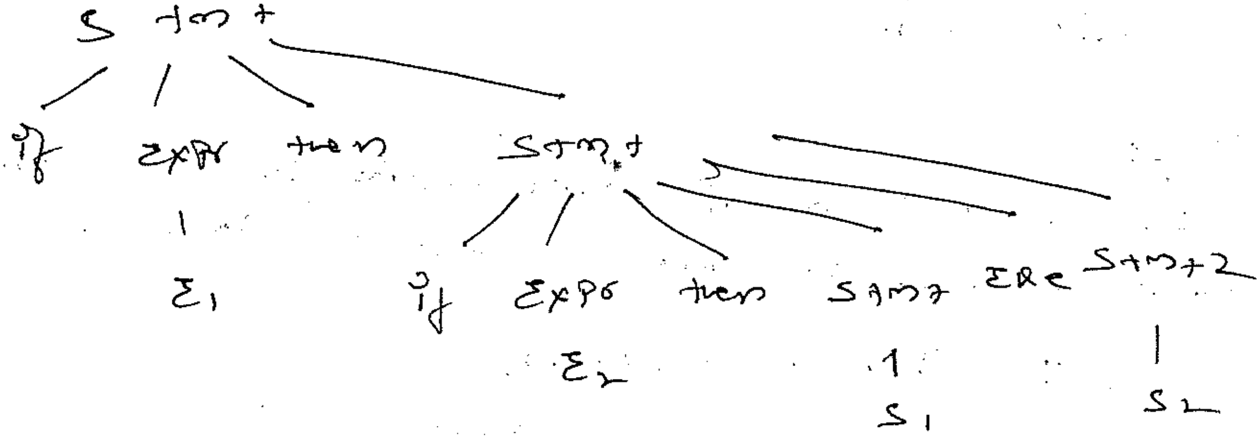
eg:- Stmt → if Expr then Stmt |
if Expr then Stmt else Stmt | other

So String: if E_1 then if E_2 then S_1 else S_2

D_1



D_2



Deryn 2 yields correct result the gramar is ambigiors as it has 2 diff parse trees.

So to Eliminate ambiguity we re-write the Grammar

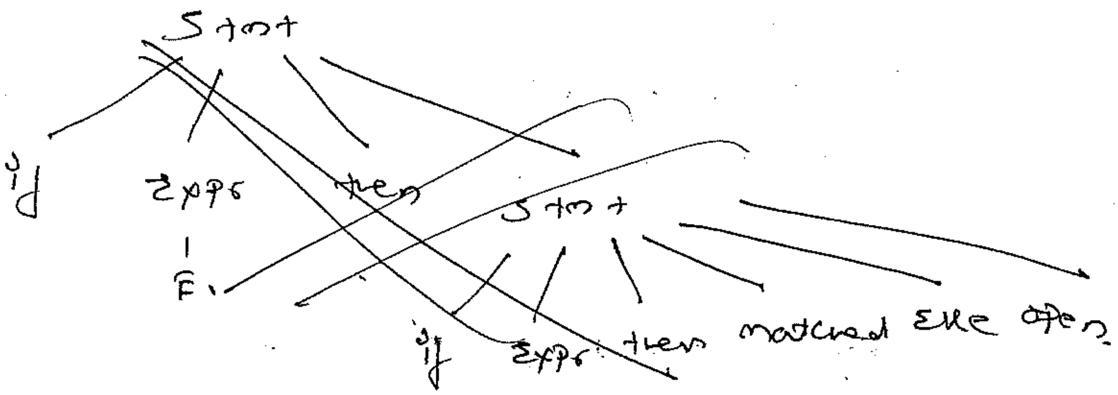
The idea is that a stmt appearing b/w then & else must be 'matched'

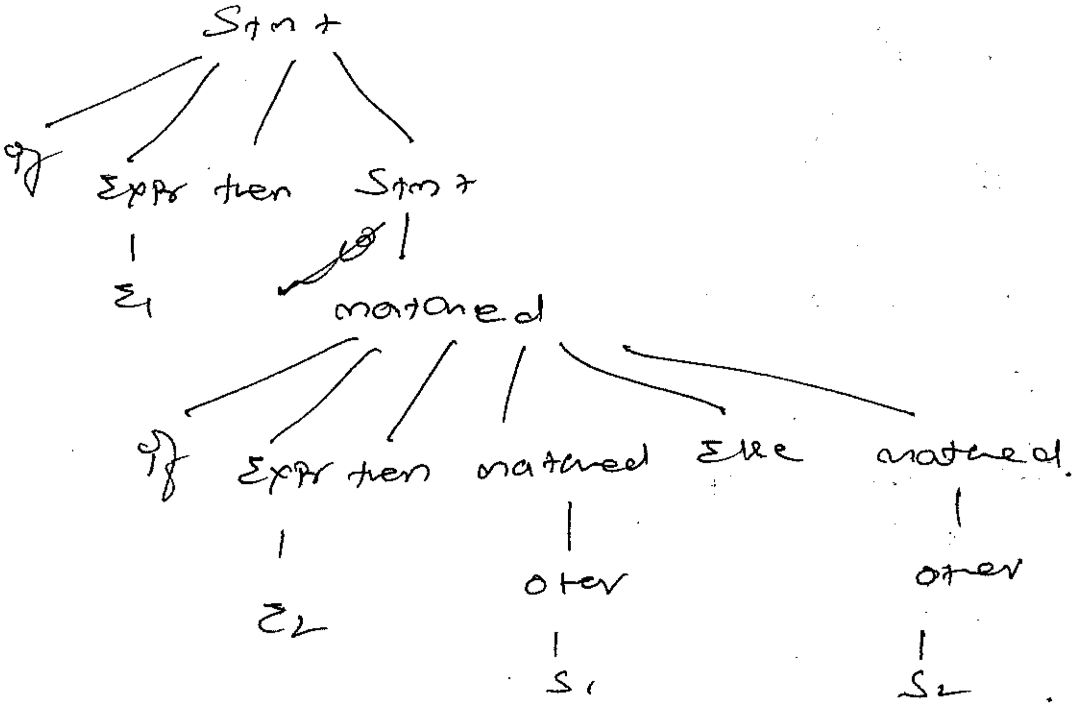
Stmt → matched | open

matched → if Expr then matched else matched |
oter.

open Stmt → if Expr then Stmt |
if Expr then matched else open

if E1 then if E2 then S1 else S2.





① Add Precedance

Eg :- $E \rightarrow EAE \mid (E) \mid id \mid num$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

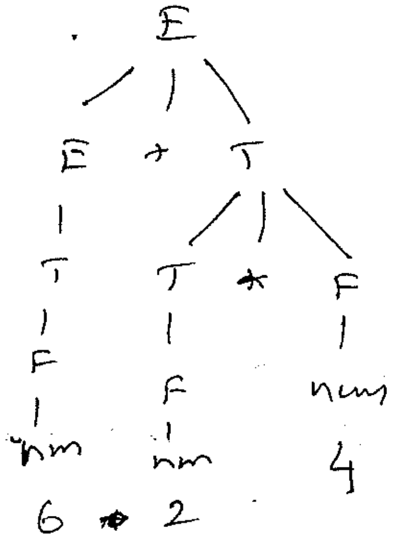
If we rewrite the grammar by adding precedance such that the lower precedance operators appear towards root & higher precedance operators towards leaf

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

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

$F \rightarrow (E) \mid id \mid num.$

String $6+2*4$



$(6+2)*4$ $\boxed{=32}$

(ii) Eliminating Left Recursion

A grammar is left recursive if it is of the form

$$A \rightarrow A\alpha \mid \beta$$

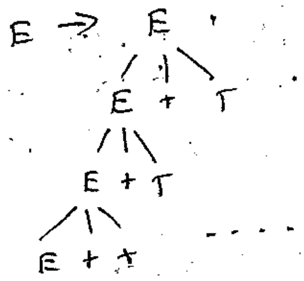
Eg:-

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

because,



So to Eliminate left recursion we rewrite the grammar as.

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

i.e

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \; A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

$\Downarrow$

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \epsilon$

Eg:-

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Eg2 :- $S \Rightarrow Aa \mid b$

$A \Rightarrow Ac \mid Sd \mid \epsilon$

Substitute S by Aa|b

~~$S \Rightarrow Aa \mid b$~~

~~$A \Rightarrow Ac \mid Aad \mid bd \mid \epsilon$~~

~~$A\alpha_1 \quad A\alpha_2 \quad \beta$~~

~~Eliminate left recursion.~~

~~$S \Rightarrow Aa \mid b$~~

~~$A \Rightarrow bdA' \mid A'$~~

$A \Rightarrow A\alpha \mid \beta$

$\Downarrow$

$A \Rightarrow \beta A'$

$A' \Rightarrow \alpha A' \mid \epsilon$

$S \Rightarrow Aa \mid b$

$A \Rightarrow Ac. \quad \Big\{ \longrightarrow \quad A \rightarrow A'$

$\qquad\qquad\qquad\qquad A' \rightarrow cA' \mid \epsilon$

$A \Rightarrow Aad \mid bd \Big\} \longrightarrow \quad A \Rightarrow bdA'$

$A \Rightarrow \epsilon \qquad\qquad\qquad A' \Rightarrow adA' \mid \epsilon$

$\Downarrow$

$$\boxed{\begin{aligned} &S \Rightarrow Aa \mid b \\ &A \rightarrow A' \mid bdA' \mid \epsilon \\ &A' \Rightarrow cA' \mid adA' \mid \epsilon \end{aligned}}$$

Left factoring

Even though there is no left recursion the grammar it may not be parsable for we+O we it is Parser. if it is of the form

$$\boxed{A \to \alpha\beta \mid \alpha\gamma}$$

if α is returned from LA it is difficult to tell weather to choose $\alpha\beta$ or $\alpha\gamma$

So to Eliminate this we do left factoring using

$$\boxed{\begin{array}{l} A \to \alpha A' \\ A' \to \beta \mid \gamma \end{array}}$$

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Eg:- $S \to aA \mid aB$

$\Downarrow$

$S \to aS'$

$S' \to A \mid B$

Eg2:-

$$S \to \underbrace{ictS}_{\alpha} \mid \underbrace{ictS}_{\alpha}\underbrace{eS}_{\beta} \mid \underbrace{a}_{\gamma}$$

$$C \to b$$

$\Downarrow$

$S \to ictSS' \mid a$

$S' \to eS \mid \epsilon$

$C \to b$

Top down Parsing

Left factoring Examples

① $Stmt\ seq \rightarrow \underline{Stmt ;\ Stmt seq} \mid \underline{Stmt}$

$A \rightarrow \alpha \quad \beta \quad \alpha.$

$A \rightarrow \alpha\beta \mid \alpha\gamma$
$\Downarrow$
$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1 \mid \gamma.$

~~Stmtseq → Stmtseq~~

$Stseq \rightarrow Stmt\ Stseq'$

$Stseq' \rightarrow ;\ Stseq.\mid \epsilon$

---

② $\underline{E} \rightarrow \underline{T} + \underline{E} \mid \underline{T}$
$A \rightarrow \alpha \quad \beta \quad \alpha$

$E \rightarrow T E'$

$E' \rightarrow + E \mid \epsilon$

replaces TE'

$E' \rightarrow + T E' \mid \epsilon$

---

③ $St \rightarrow \underline{id} = \underline{E} \mid \underline{id}\ \underline{(E)} \mid other.$
$A \rightarrow \alpha \quad \beta \quad \alpha \quad \gamma$

$St \rightarrow id.\ St' \mid other.$

$St' \rightarrow = E \mid (E)$

Top down Parsing

It can be viewed as Problem of Constructing a Parse tree starting from root in Pre-order.

or

Equivalent to finding the LMD for all i/p string

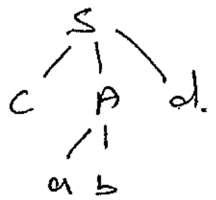
- → Start at root
- → Repeat untill Parse tree matches i/p string
- → determine the correct Production to be applied for a non terminal (Key Point)
- → Proceed constructing left tree by Expanding left-most "NT" till terminal is reached.

Recursive Decendent Parsing LL(0)

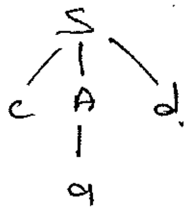
A RD Parsing Program consists of a set of Procedures one for each non-terminal, Ex^n begins with Procedure of Start symbol, which halts and announces success if its Procedure body scans the Entire string.

RD Parser may require back tracking to find correct Prod^n to be applied.

Eg:- $S \Rightarrow cAd$ String: cad
$A \Rightarrow ab|a$



we need to backtrack



Predective Parsing dont need back tracking.

Eg:- Recursive decendent Parser:

$E \Rightarrow TE'$

$E' \Rightarrow +TE' | \epsilon$

$T \Rightarrow FT'$

$T' \Rightarrow *FT' | \epsilon$

$F \Rightarrow (E) | id$

the Procedure for RD Parser is.

```
Procedure E()
 {
   T();
   EPRIME();
 }
Procedure EPRIME()
{
ch = getch();
  if (ch=='+')
   {
    T();
    EPRIME();
   }
}
Procedure T()
  {
    F();
   TPRIME();
  }
Procedure TPRIME()
{ ch = getch();
 if (ch=='*')
 {F();
 TPRIME();
 }
}
```

```
Procedure F()
  {
   ch=getch();
    if (ch=='(')
    {
     E();
    ch=getch();
    if (ch==')')
    return TRUE;
    Else
     return FALSE;
    }
Else
ch = getch();
   if (ch=='id')
    return TRUE;
    Else
     Return FALSE;
  }
```

Predective Parsing LL(1)

The Parser Predicts the next alternative to choose so the grammar need to be unambiguous. i.e need to define one & only alternative at any point

(S1) So the grammer need to be eleminated with
(i) Ambiguity ✓ (ii) left factoring ✓ (iii) left recursion.

(S2) Two functions are used for finding Parse table. so find
(i) first symbols (ii) follow symbols.

(S3) Construct Parse table

(S4) Parse the given string using Stacks.

First & Follow Set

They are the set terminal symbol that help for filling Parse table with vallid Entry & used in Error recovery.

→ **First:** if α is the string of grammar symbols then First(α) is the set of terminals that begins the derivation from α. if $\alpha \overset{*}{\Rightarrow} \varepsilon$ then ε is in first(α)

→ **Follow(A)** is the set of terminals 'a' that can appear immediatly to the right of A such that there Exists a derivation of the form $S \overset{*}{\Rightarrow} \alpha A a \beta$

→ The set of terminals (including ε) that can appear at far left of any Parse tree derived from a NT is the first of that NT.

→ The set of terminals (Excluding ε) that follow a NT in any dervn is called FOLLOW set of that NT.

Rules to Create FIRST:

① If X is a terminal then $first(X) = \{x\}$

2) If $X \rightarrow \varepsilon$ then ε is in $first(X)$

3) if $X \rightarrow Y_1 Y_2 \cdots Y_k$ and $Y_1 \cdots Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$ and $a \in first(Y_i)$ then $a \in first(X)$

i.e if $X \rightarrow Y$ & $Y \overset{*}{\Rightarrow} \varepsilon$ and if a is in $first(Y)$ then a is in $first(X)$

~~if X → YZ & Z *⇒ ε then first(X) = first(Y) + first(Z)~~ ~~if all RHS derive ε then default add ε~~

Rules to Create Follow

1) If S is a start symbol then place \$ in $Follow(S)$

2) If there is a prodn of the form $A \rightarrow \alpha B \beta$ then everything in $first(\beta)$ except ε is placed in $Follow(B)$ $\beta \neq \varepsilon$

④ If there is a prodn of the form $A \rightarrow \alpha B$ then everything in $Follow(A)$ is in $Follow(B)$ $\beta \Rightarrow \varepsilon$

③ If there is a prodn of the form $A \rightarrow \alpha B \beta$ where $\beta \overset{*}{\Rightarrow} \varepsilon$ then everything in $Follow(A) = Follow(B)$

---

first rule.

4) if $X \rightarrow Y Z a$ and $Y \overset{*}{\Rightarrow} \varepsilon$ then

$first(X) = first(Y) + first(Z)$

and

if $Z \overset{*}{\Rightarrow} \varepsilon$ then

$first(X) = first(Y) + first(Z) + first(a)$

if all RHS produces ε include ε by default in $first(X)$.

Summary of first & follow rules

first rules :

① X is term then first(X) = {X}

② if $X \rightarrow \varepsilon$ first(X) = {ε}

③ $X \rightarrow Y$ first(X) = first(Y)

④ $X \rightarrow YZa$ $Y \overset{*}{\Rightarrow} \varepsilon$ $Z \overset{*}{\Rightarrow} \varepsilon$ then
first(X) = first(Y) + first(Z) + first(a)

---

follow rules:

Note: all rules are recursively applied on Every Prodn

① $S \rightarrow \$$

② $A \rightarrow \alpha B \beta$ $\beta \neq \varepsilon$
first

③ $A \rightarrow \alpha B \beta$ $\beta \overset{*}{\Rightarrow} \varepsilon$.
foll

④ $A \rightarrow \alpha B$ $\beta = \varepsilon$
foll

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Prob: G : ~~E → E A E | (E) | id~~
~~A → + | *~~

G: E → E+E | E*E | (E) | id.
(S1) is ambigious so rewrite with precedance

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

} (S2) Eliminate left recursion

E → TE'
E' → +TE'
T → FT'
T' → *FT'
F → (E) | id

(S3) first & follow

)Compute the first & follow for the grammar

23) $E \to TE'$ $E' \to +TE'|\varepsilon$ $T \to FT'$ $T' \to *FT'|\varepsilon$ $F \to (E)|id$

	E	E'	T	T'	F
first	(. id	+ ε	(id	* ε	(id.
follow	$)	$)	+ $)	+ $)	* + $)

to compute first:

Non Terminal	Production	First
E	$E \to TE'$ ③ $T \not\Rightarrow \varepsilon$ 4x	first(E) = first(T) {(, id}
E'	$E' \to +TE'\|\varepsilon$	first(E') = {+, ε}
T	$T \to FT'$ ③ $F \not\Rightarrow \varepsilon$ no ④x	first(T) = first(F) {(, id}
T'	$T' \to *FT'\|\varepsilon$	first(T') = {*, ε}
F	$F \to (E)$ $F \to id$	first(F) = {(, id}

no rule 4 is applied.

28

59

60

to compute follow :

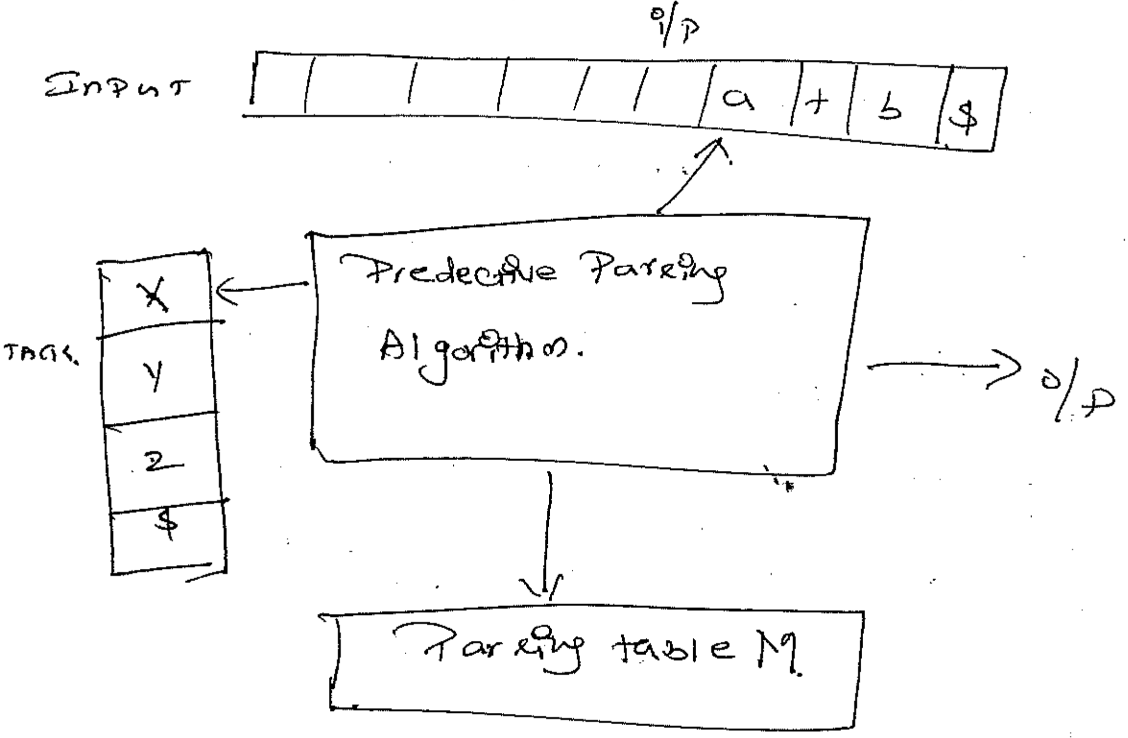
NT	Prodn	Follow
E	$E \rightarrow TE'$ ① $S \rightarrow \$$ ② $A \rightarrow \alpha B \beta$ $\beta \rightarrow \varepsilon$ ③ $\rightarrow$ ④ $A \rightarrow \alpha B$	foll(E) = \$ foll(T) = first(E') = + foll(T) = foll(E) = \$ foll(E') = foll(E) = \$
E'	$E' \rightarrow +TE' \mid \varepsilon$ 1× ② A $\alpha B \beta$ $\beta \neq \varepsilon$ $E' \rightarrow +TE'$ ③ $A \rightarrow \alpha B \beta$ $\beta \rightarrow \varepsilon$ foll ④ $E' \rightarrow \underline{+T}\,\underline{E'}$ $A \rightarrow \alpha \; B$ $\beta = \varepsilon$	foll(T) = first(E') = + foll(T) = foll(E') = \$ foll(E') = foll(E')
T	$T \rightarrow FT'$ ② $A \rightarrow \alpha B \beta$ $\beta \neq \varepsilon$ $T \rightarrow FT'$ ③ $A \rightarrow \alpha B \beta$ $\beta \rightarrow \varepsilon$ ④ $T \rightarrow FT'$ $A \rightarrow \alpha B$ $\beta = \varepsilon$	foll(F) = first(T') = * foll(F) = foll(T) = + \$ foll(T') = foll(T) = + \$
T'	$T' \rightarrow *FT' \mid \varepsilon$ ② $A \rightarrow \alpha B \beta$ $\beta \neq \varepsilon$ ③ $T \rightarrow *FT'$ $A \rightarrow \alpha B \beta$ $\beta \rightarrow \varepsilon$ ④ $T \rightarrow \underline{*F}\,\underline{T'}$ $A \rightarrow \alpha \; B$ $\beta = \varepsilon$	foll(F) = first(T') = * foll(F) = foll(T) = + \$ foll(T') = foll(T) = + \$
F	$F \rightarrow (E)$ ② $A \rightarrow \alpha B \beta$ ③ x ④ x	foll(E) = first$( ))$ =)

Non-Recursive Predicting Parser LL(1)

It is built by maintaining a stack explicitly rather than implicitly via recursive calls.

The Parser Produce LMD, if w is the i/p that has been matched so far Then stack holds a sequence of grammar symbols α such that

$$S \xRightarrow[LMD]{*} w\alpha$$



1) Input : The string to be parsed followed by $

2) Stack : Initially $ is placed on top of Stack, later it may contain grammar Symbol

3) Parsing table : It contains 2-D array $M[\underset{NT}{A}, \underset{T \text{ or } \$}{a}]$

Every Entry in table is Either Prod^{n} or Error.

Predective Parsing Algorithm

The P.P Program determines X which is the symbol on top of the stack & the current i/p symbol 'a'. these 2 symbols determine the action of Parser.

There are 3 possibilities of action

① If X = a = \$, the Parser halts & and announces successful completion of Parsing.

② If X = a ≠ \$ then Pop X from the stack & advance the i/p ptr to point next symbol.

③ If X is NT, the Program consults the Parsing table which may contain X Prodn or Error entry.
If $M[X, a] = X \rightarrow UVW$, the Parser replaces X on top of the stack by UVW in reverse order.

Algorithm to Construct Predective parsing Table

I/p : Grammar G.
o/p : Parsing table M.

Procedure: For each Prodn $A \rightarrow \alpha$ of the grammar, do the following

(i) for each terminal a in first(α) add $A \rightarrow \alpha$ to $M[A, a]$

(ii) If ϵ is in first(α) then for each terminal 'b' in follow(A) add $A \rightarrow \alpha$ to $M[A, b]$ if ϵ is in first(α) & \$ is in follow(A) add $A \rightarrow \alpha$ to $M[A, \$]$ as well

(iii) Make undefined entries in table as Errors.

b) Construct Predective Parsing table

M [N T]

N \ T	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

(35) Parse the String id + id * id

Parse stack	i/p	action
$ E	id + id * id $	$E \to TE'$
$ E' T	id + id * id $	$T \to FT'$
$ E' T' F	id + id * id $	$F \to id$
$ E' T' id	id + id * id $	match
$ E' T'	+ id * id $	$T' \to \varepsilon$
$ E'	+ id * id $	$E' \to +TE'$
$ E' T +	+ id * id $	match
$ E' T	id * id $	$T \to FT'$
$ E' T' F	id * id $	$F \to id$

$ E'T'id	id * id $	match
$ E'T'	* id $	T' → *FT'
$ E'T'F*	* id $	match
$ E'T'F	id $	F → id
$ E'T'id.	id $	match
$ E'T'	$	T' → ε
$ E'	$	E' → ε
$	$	accept

② Compute first & follow & obtain Predective parsing table for the following grammar:

S → AB | PQx.
A → xy | m
B → bc
P → pP | ε
Q → qQ | ε
C → e.

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

* The grammar is not left recursive nor require left factoring

* Then we need to compute first & follow

First sets.

NT	Production	first
S	$S \rightarrow AB$	first(S) = first(A) {x, m}
{x p q m}	$S \rightarrow PQx$	first(S) = first(P) + first(Q) + first(x) none ε none ε {x, p, q}
A {x m}	$A \rightarrow xy \mid m$	first(A) = {x, m}
B {b}	$B \rightarrow bC$	first(B) = {b}
P {p ε}	$P \rightarrow pP \mid \varepsilon$	first(P) = {p, ε}
Q {q ε}	$Q \rightarrow qQ \mid \varepsilon$	first(Q) = {q, ε}
C {e}	$C \rightarrow e$	first(C) = {e}

	S	A	B	P	Q	C
first	x p q m	x m	b	p ε	q ε	e
follow	$	$ b	$	q x	x	$

Notes :- if there are more than one symbol in (β)

dont check $\beta \xRightarrow{*} \varepsilon$

Followsets.

NT	Productions	follow
S	$S \rightarrow AB$ ① $A\ \ \alpha B\beta$ $\beta \neq \varepsilon$ ② $S \rightarrow AB$ $\beta \Rightarrow \varepsilon$ $A \rightarrow \alpha B$ $\beta = \varepsilon$	$follow(S) = \{\$\}$ $foll(A) = first(B) - \$$. $foll(B) = foll(S)$
	$S \rightarrow PQx$ $A \rightarrow \alpha B\beta$ $\beta \neq \varepsilon$ $\beta \Rightarrow \varepsilon_x$ $S \rightarrow PQx$ $A \rightarrow \alpha B\ \beta$ $\beta \neq \varepsilon$ $\beta \not\Rightarrow \varepsilon$	$foll(Q) = first(x) = x$ $foll(P) = first(Q) + first(x)$ $q\ x$ ~~foll(Q) = foll(S) = \$.~~
A	$A \rightarrow x y \mid m$	x
B	$B \rightarrow bC$ $A \rightarrow \alpha B$	$foll(C) = foll(B)$
P	$P \rightarrow pP$ $A \rightarrow \alpha B$	$foll(P) = foll(P)$
Q	$Q \rightarrow qQ$ $A \rightarrow \alpha B$	$foll(Q) = foll(Q)$
C	$C \rightarrow e$	x

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Parse table :

NT	I/P Symbol								
	x	y	m	p	q	e	b	$	s
S	S → PQx		S → AB	S → PQx	S → PQx				
A	A → xy		A → m						
B							B → bC		
P	P → ε			P → pP	P → ε				
Q	Q → ε				Q → qQ.				
C						C → e.			

Prob ③ :

$S \rightarrow iCtSS' \mid a.$

$S' \rightarrow eS \mid \varepsilon.$

$C \rightarrow b$

	S	S'	C
first	i a	e ε	b
follow	$ e	$ e	t

first symbols

N.T	Prodn	first set
S	$S \rightarrow iCtSS' \mid a.$	first(S) = {i, a}
S'	$S' \rightarrow eS \mid \varepsilon$	first(S') = {e ε}
C	$C \rightarrow b$	first(C) = {b}

followsets

NT	Prodn	followset
S	$S \to i(tSS'$ $A \to \alpha B \beta$ $S \to \underbrace{iCtS}_{\alpha}\underbrace{S'}_{B}$ $S \to \underbrace{i}_{\alpha}\underbrace{C}_{B}\underbrace{tSS'}_{\beta}$ $\beta \not\Rightarrow \varepsilon$	① follow(S) = {\$} ② foll(S) = first(S') foll(S) = foll(S) foll(S') = foll(S) foll(C) = first(tSS')

*Note: if there be more than one symbol in 'β' don't check $\beta \overset{*}{\Rightarrow} \varepsilon$

NT	Prodn	followset
S'	$S' \to eS$ $A \to \alpha B$	foll(S) = foll(S')

Note: if there are multiple Entries in Parse table (PP) then it be not LL(1) gramar

Parse table

NT	i	t	e	a	b	\$
S	$S \to i(tSS'$			$i \to a$		
S'			$S' \to \varepsilon$ $S' \to eS$			$S' \to \varepsilon$
C					$C \to b$	

grammar:

$Stmt \rightarrow Ifstmt$

$Stmt \rightarrow other$

$fstmt \rightarrow if\ (Exp)\ Stmt\ Else Part$

$lsePart \rightarrow else\ Stmt \mid \varepsilon$

$xP \rightarrow 0 \mid 1$

Note again:

If $A \rightarrow \alpha B \beta$ & β is having more than one symbol then only consider rule ② i.e

$A \rightarrow \alpha B \beta$ $\beta \neq \varepsilon$ $foll(B) = first(\beta)$

don't check rule ③ or rule ④

If there are multiple entries in parse table then it is not LL(1) grammar.

rst symbols

NT	Productions	First
tmt	$Stmt \rightarrow Ifstmt \mid other$	$first(Stmt) = \{first(Ifstmt) +$ $\{if\ other\}$ $first(other)$
fstmt	$Ifstmt \rightarrow if\ (Exp)\ Stmt\ Else part$	$first(Ifstmt) = \{if\}$
epart	$ElsePart \rightarrow else\ Stmt \mid \varepsilon$	$first(ElsePart) = \{else, \varepsilon\}$
xp	$Exp \rightarrow 0 \mid 1$	$first(Exp) = \{0\ 1\}$

	Stmt	Ifstmt	ElsePart	Exp
first	if other	if	else ε	0 1
follow	$ else	$ else	$ else	)

follow sets

NT	Production	follow.
Stmt	Stmt → Ifstmt A → α B	① follow(Stmt) = {$} ④ follow(Ifstmt) = foll(Stmt)
Ifstmt	Ifstmt → if (Exp) Stmt Elsepart A → α B B Ifstmt → if (Exp) Stmt Elsepart A → α B B B ⇒ ε Ifstmt → if (Exp) Stmt Elsepart A → α B	foll(Exp) = first() Stmt Elsepart) =) foll(Stmt) = first(Elsepart) foll(Stmt) = foll(Ifstmt) foll(Elsepart) = foll(Ifstmt)
else	Elsepart → else Stmt A → α B	foll(Stmt) = foll(Elsepart)

Parse table

	other	if	(	)	else	0	1	$
Stmt	Stmt → other	Stmt → Ifstmt						
Ifstmt		Ifstmt → if (Exp) Stmt Elsepart						
Elsepart					Elsepart → else Stmt Else → ε			Else → ε
Exp						Exp → 0	Exp → 1	

Error recovery in Predective Parsing.

An Error is detected during P.P when terminal symbol ~~terminal~~ on top of parsing stack does not match the next i/p symbol, or when non terminal A is on top of the stack & a is on i/p symbol & M[A,a] is Error in parsing table

Panic mode Recovery

Panic mode recovery is based on idea of skipping symbols on the i/p until a token in selected set of synchronizing tokens appear. there are two ways to find the synchronizing tokens.

(i) As a starting point, place all symbols in follow(A) into the synchronizing set of NT A.

If we skip tokens until an element of follow(A) is seen & pop A from stack it is likely that parsing can continue.

(ii). If we add symbols in First(A) to the synchronizing set of NT A then it may be possible to resume parsing

Eg:-

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	~~[illegible]~~ ST	ST
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$	ST		$T \to FT'$	ST	ST
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$	ST	ST	$F \to (E)$	ST	ST

	E	E'	T	T'	F
first	(id	+ ε	(id	* ε	(id
follow	$)	$)	+ $)	+ $)	* + $)

Parse String) id * + id

Parse stack	I/P	action
$ E	) id * + id $	Error skip)
$ E	id * + id $	E → TE'
$ E'T	id * + id $	T → FT'
$ E'T'F	id * + id $	F → id
$ E'T' id	id * + id $	match
$ E'T'	* + id $	T' → *FT'
$ E'T'F *	* + id $	match
$ E'T'F	+ id $	ST POP F
$ E'T'	+ id $	T' → ε
$ E'	+ id $	E' → +TE'
$ E'T +	+ id $	match
$ E'T	id $	T → FT'
$ E'T'F	id $	F → id
$ E'T' id	id $	match
$ E'T'	$	T' → ε
$ E'	$	E' → ε
$	$	

	E	E'	T	T'	F
first	(id	+ ε	(id	* ε	(id.
follow	$)	$).	+ $)	+ $)	* + $).

Parse String) id * + id

Parse stack	I/P	action
$ E	) id * + id $	Error skip)
$ E	id * + id. $	E → T E'
$ E' T	id * + id $	T → F T'
$ E' T' F	id * + id $	F → id
$ E' T' id	id * +. id. $	match
$ E' T'	* + id $	T' → * F T'
$ E' T' F *	* + id $	match
$ E' T' F	+ id $	ST POP F
$ E' T'	+ id. $	T' → ε
$ E'	+ id. $	E' → + T E'
$ E' T +	+ id $	match
$ E' T	id. $	T → F T'
$ E' T'F	id $	F → id
$ E' T' id	id $	match
$ E' T'	$	T' → ε
$ E'	$	E' → ε
$	$	Successful !!

Phrase level Recovery

This recovery technique are implemented by filling in the blank Entries in the Predictive parsing table with pointers to Error routines. these routines may change or insert or delete symbols on the i/p & issue appropriate Error message they may also pop from stacks.

X

End of unit II

Unit-③. Bottom up Parsing

A bottom-up Parse corresponds to the construction of a Parse tree for the i/p string beginning at the leaves (bottom) and working up towards the root (top).

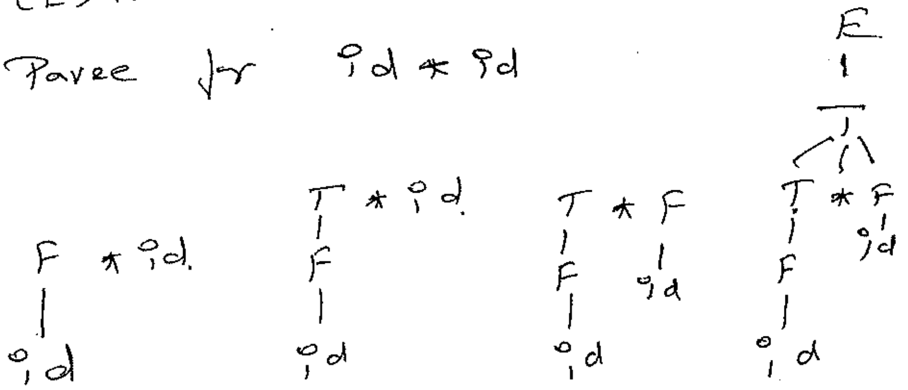
Eg:- G: $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Show the BU Parse for id * id

id * id.



id + id

Reductions:

Bottom up Parsing is a Process of reducing a string w to the start symbol of the grammar. At each reductio step a specific sub-string matching the body of the Prodn is replaced by the Non Terminal so the key decision in the BU Parsing is about when to reduce & about what Prodn to apply as the Parsing Proceeds.

Handle Pruning

Bottom up Parser scans symbols from left to right & constructs a RMD in reverse order:

A "HANDLE" is defined as substring that matches the body of the Production and whose reduction represents one step along the reverse of RMD...

Eg:- $id_1 * id_2$

String	Handle	reducing Prodn
$id_1 * id_2$	id_1	$F \rightarrow id_1$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id_2$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$
E	Start symbol	so string accepted.

formally if $S \overset{*}{\Rightarrow} \alpha A w \Rightarrow \alpha \beta w$ then the prodn $A \to \beta$ in the position follow α is a handle of $\alpha \beta w$

Shift Reduce Parsing

This is one of the technique used in BU Parsing.

In S-R-Parser we use stack, i/p buffer, where \$ is used to mark the bottom of Stack & right End of the i/p.

Intial Con-figuration

Stack	Input
\$	w \$ string

Parser operates by shifting zero or more i/p symbols on to the stack untill an handle is on top of the stack. The Parser then reduce β to the LHS of appropriate Prodn

The Parser repeates this cycle untill it has detected an Error or stack contains Start Symbol & i/p is Empty. then Parser stops & announces succesfull completion of Parsing / string accepted.

Final Configuration

Stack	i/p
\$ S	\$

The action made by Shift reduce Parser are:

① Shift : The next i/p symbol is shifted onto the top of stack.

② Reduce - The Parser knows that handle is on top of stack & should be replaced with appropriate LHS of Prodn.

③ Accept - Start symbol on top of stack & i/p buffer is empty.

④ Error - the Parser discovers the error & calls error recovery routine.

Eg:-

$E \Rightarrow E+E \mid E*E \mid (E) \mid id$

Stack	i/p	action
\$	id * id \$	shift id
\$ id.	* id \$	E → id.
\$ E	* id \$	shift *
\$ E*	id \$	shift id
\$ E * id	\$	E → id,
\$ E * E	\$	E → E * E
\$ E	\$	successfull

Eg2:- id + id * id

Stacks	i/p	actions
$	id+id*id$	Shift id
$ id	+ id*id$	E → id reduce
$ E	+ id * id$	Shift +
$ E +	id * id $	Shift id
$ E + id	* id $	E → id reduce
$ E + E	* id $	E → E+E reduce
$ E	* id $	Shift *
$ E *	id $	Shift id
$ E * id	$	E → id reduce
$ E * E	$	E → E*E reduce
$ E	$	accept

Eg③ id * id + id

Conflict during Shift reduce Parser

There are few class of CFG for which SR Parsing can't be used.

Every SR Parser for such a grammar can reach a configuration in which the parser cannot decide whether to Shift or to Reduce (Shift - Reduce Conflict) inspite of knowing the entire stack contents & next i/p symbol.

The other conflict is the parser cannot decide which of the several reductions to make. known as Reduce - Reduce Conflict

Technically these grammars are not in the LR(k), where k is look ahead operator & any ambiguous grammar cannot be LR grammar

~~LR Parsers~~

~~There Pa~~

Eg:- $S \rightarrow aABe$ $A \rightarrow Abc$ $A \rightarrow b$ $B \rightarrow d$ string abbcde

Parse stack		i/p	action
\$		abbcde \$	Shift
\$a		bbcde\$	shift
\$a<u>b</u>		bcde\$	$A \rightarrow b$
\$aA	S|R $A \rightarrow b$ $A \rightarrow Abc$?	bcde\$	shift
\$aA<u>b</u>		cde\$	shift
\$a<u>Abc</u>		de\$	$A \rightarrow Abc$
\$aA		de \$	....

LR-Parsers.

These Parsers are more general than SR parsers

L - Scans I/P from left to right

R - Obtains RMD in reverse.

Advantages:

→ LR Parsers can be constructed to synchronize virtually all parsing language constructs for which CFG Exists.

→ it is most general non-backtracking SR parser.

→ The class of grammars that can be used using LR methods is proper subset of grammars that can be parsed with Predictive Parsers

→ An LR parser can detect error as soon as possible..

Disadvantages:

→ It needs lot of Effort to construct LR parsers by hand for typical Programming lang grammar.

→ So we need specilized tool for LR parser generators

→ If the grammar contains ambiguities or other constructs. it will fail.

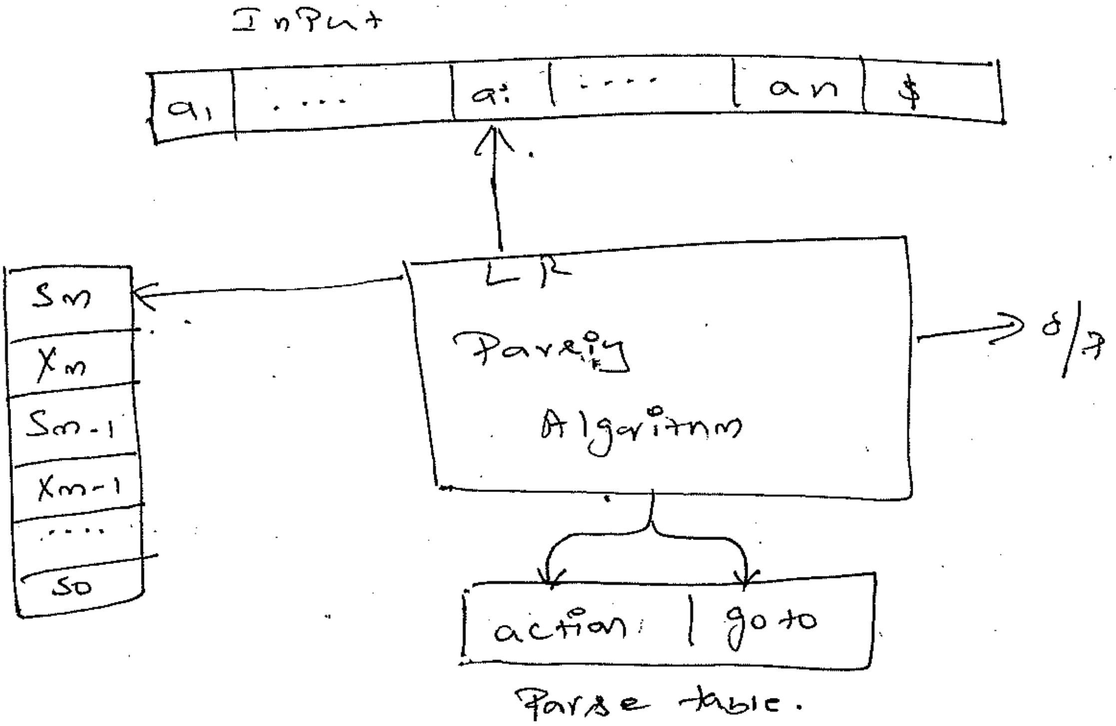
There are 3 techniques for constructing LR Parsing table

① Simple LR (SLR) Easy to impl^n & least powerfull

② Canonical LR - Most powerfull & Expensive

③ Look Ahead LR (LALR) - is intermediate in power & cost & will work on most Prglang grammars & can be impl^n at moderate effort.

LR Parsing Algo



- It contains of an i/p, o/p, stack, driver program & Parsing table that has two parts. (Action & goto)

- The driver Prg is same for all LR Parsers only Parsing Table will change.

- I/P - I/p is read from Left to Right one symbol at a time till it ends with $

- Stack - Contains string of the form $S_0 X_1 S_1 X_2 S_2 \dots X_m S_m$ where X_i is a grammar symbol & S_i is state.

Parsing Table: Consists of two parts, Parsing action function & a goto function.

Driver Prg: The driver Prg behaves as follows.

It determines Sm the state currently on top of the stack & ai the current i/p symbol, then consults the action [Sm, ai] from parse table which can have one of the four values

(i) Shift S where S is a state

(ii) Reduce by a grammar A → α.

(iii) Accept

(iv) Error:

The function goto takes a state & grammar symbol as arguments & produce a state.

Configuration of LR parser is a pair whose first component is the stack & whose second component is i/p

$(S_0 X_1 S_1 X_2 \cancel{S}_2 \ldots\ldots X_m S_m, a_i a_{i+1} \ldots a_n \$)$

The next move of the parser is determined by reading ai, the current i/p symbol & Sm the state on top of the stack & then consulting parsing action table. Entry action [Sm, ai]

4 types of moves are as follows

(i) If action [Sm ai] = Shift S the parser executes a shift move.

(ii) If action [Sm ai] = reduce A → β then POP 'r' grammar symbols & 'r' state from the stack where 'r' is length of 'β'

(iii) If action $[S_m\ a_i]$ = accept Parsing is Complete.

(iv) If action $[S_m\ a_i]$ = Error the Parser has discovered an Error & Calls an Error recovery routine.

Construction of SLR Parsing Table

Item: An LR(0) item or item of a grammar G is a Prodn of G with a dot at some Posn on the RHS of Prodn

eg:- $A \rightarrow XYZ$

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$

If $A \rightarrow \varepsilon$ then $A \rightarrow \cdot$

Item indicates how much of the Prodn we have seen at a given pt in Parsing Process.

eg:- $A \rightarrow X \cdot YZ$ means we have seen i/p string derivable from X & we have to see the string derivable from YZ

The colln of LR(0) items is called canonical LR(0) colln for grammar.

The grammar G is argumented before LR(0) items by giving new start state S' where $S' \rightarrow S$ which indicate Parser when it should stop Parsing and announces acceptance of i/p.

Algorithm to construct LR(0) items

```
Procedure items (G').
begin
    C = closure ($S' \rightarrow .S$)
    repeat
        for each set of items I in C & each grammar symbol X
    such that GOTO(I, X) is not empty.
        {
            add GOTO (I, X) to C
        }
    until no more sets of items can be added to it.
End.
```

The Closure opn

If I is a set of items for grammar G then closure (I) is set of items constructed from I by the two rules.

(i) Initially every item in I is added to closure

(ii) If $A \rightarrow \alpha . B\beta$ is closure (I) & $B \rightarrow \gamma$ is Prodn then add the item $B \rightarrow . \gamma$ to I if it is not already there. Apply this rule until no more new items can be added to closure (I).

The function

```
Begin
    J = I;
    repeat
        for each items $A \rightarrow \alpha . B\beta$ in J & each Prodn $B \rightarrow \gamma$
        of G such that $\gamma$ is not in J do
        {
            add $B \rightarrow . \gamma$ to I
        }
```

until no more items can be added to I

return J

End

Goto operation

Goto (I, X) where I is set of Prodn & X is grammar symbol Goto (I, X) is defined to be the closure of set of all items [A → α X . β) such that [A → α . X β] is in I.

$$Goto\ (I, x) = Closure\ (A \to \alpha X \cdot \beta)\ \text{if}\ A \to \alpha . X \beta\ \text{is in}\ I.$$

Algorithm to Construct SLR Parsing table

I/P : canonical colln. of set of items LR(0) items

O/P : SLR Parsing table consisting of ACTION & GOTO

Method:

let {I₀, I₁ ... In} be the set of items the states of Parsers are 0, 1, 2 ... n where state i is constructed from Ip

The parsing actions for state i are determined as follows.

Step 1: If $A \to \alpha . a \beta$ in I_i & goto $(I_i, a) = I_J$ then set action to Shift J where a is terminal or non terminal

Step 2: If $A \to \alpha .$ is in I_i then set Action[i, a] to reduce $A \to \alpha$ for Each a in follow(A)

Step 3: If $S' \to S .$ is in I_i then set Action [i, $] to accept where S' is start symbol

Prob:- S' → S

S → (L)

) S → x

) L → S

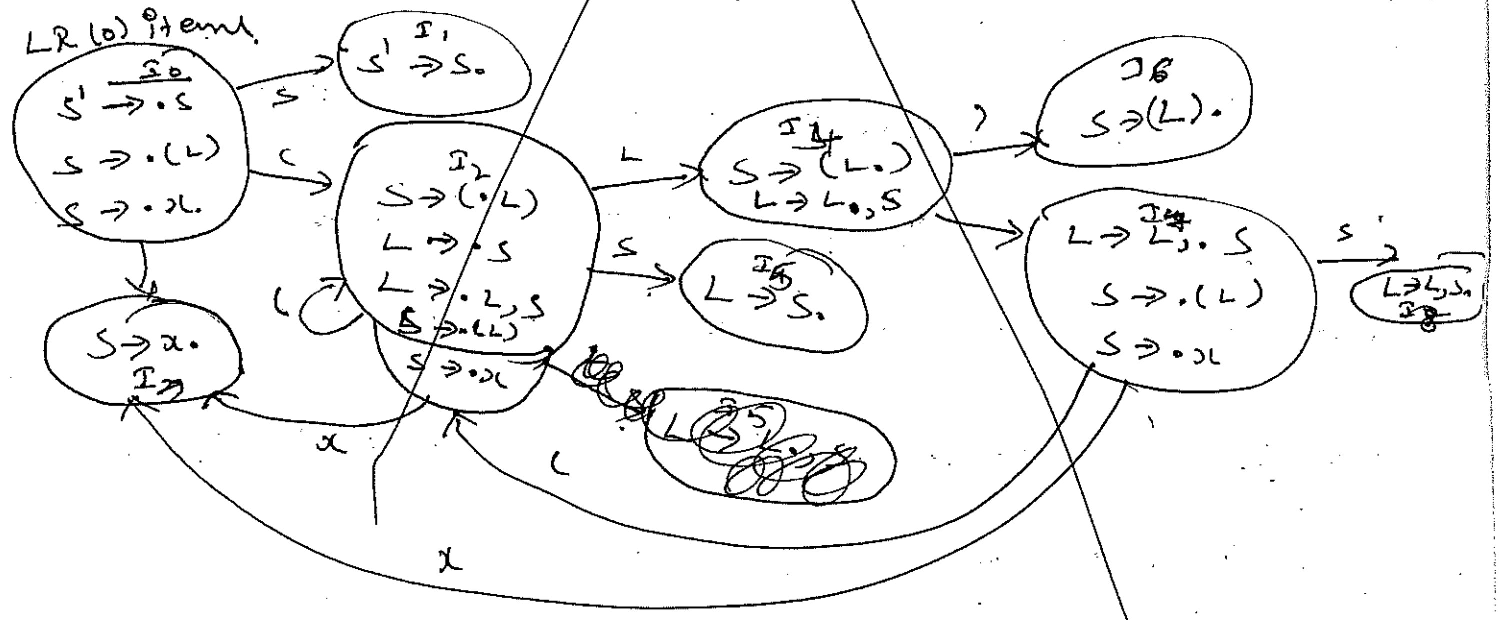
) L → L,S

compute first & follow set

	S	L
first	(x	(x
follow	$),	) ,

NT	Prodn	follow
S	S → (L) A → α B β	foll(L) = first())
L	L → S A → α B	foll(L) = foll(S)
L	L → L, S α B β	foll(L) = first(,S) ,
	L → L, S A α. B	foll(S) = foll(L)

LR(0) item



Construct Parsing table

← action → ← Goto →

	(	)	,	x	$	S	L
0	s2			s3		1	
1					accept		
2	s2			s3		5	4
3		r2	r2		r2		
4		s6	s7				
5		r3	r3				
6		r1	r1		r1		
7	s2			s3		8	
8		r4	r4				

Parse string (x, x)

Stack	I/P	action
$	(x, x)$	shift
$(2	x, x)$	shift
$(2x3	, x)$	reduce S → x
$(2S5	, x)$	reduce L → S
$(2L4	, x)$	shift
$(2L4,7	x)$	shift
$(2L4,7x3	)$	reduce S → x
[illegible]	$	reduce S → (L)
$		

Prob 1:

(S1) augment gramar.

gramar : S' → S

1) S → (L)
2) S → x
3) L → S
4) L → L, S

first(S) = {(x}

first(L) = {(x}

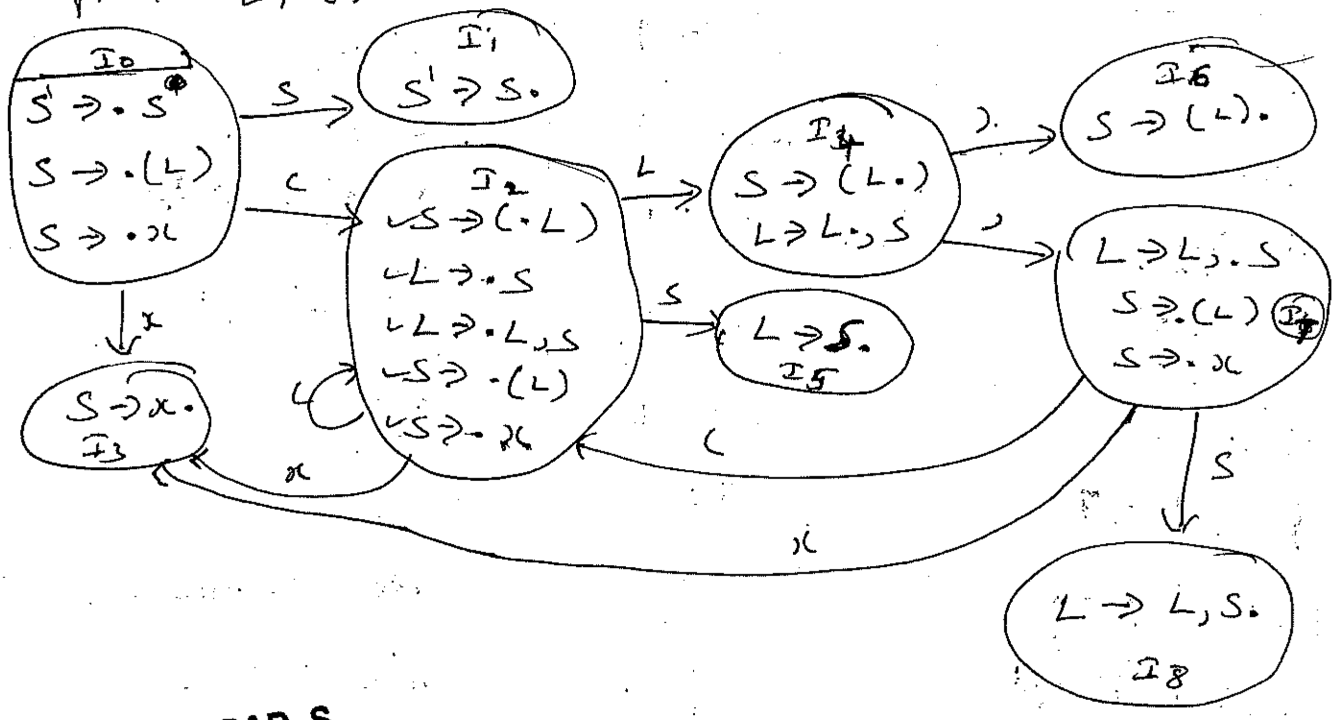
	S	L
first	(x	(x
foll	$),	),

(S2) find first & follow

foll

NT	Prodn	follow
S	S → (L) A → αBβ	S → $ foll(L) = first())
L	L → L, S A → αB β L → L, S A → α B	foll(L) = first(, S) foll(S) = foll(L)

(S3) find LR(0) items.



④ construct Parsetable

	action					Goto	
	(	)	,	x	$	S	L
0	s2			s3		1	
1					accept		
2	s2			s3		5	4
3		r2	r2		r2		
4		s6	s7				
5		r3	r3				
6		r1	r1		r1		
7	s2			s3		8	
8		r4	r4				

⑤ Parse String (x, x)

Stack	I/P	action
0	(x, x) $	shift
0(2	x, x) $	shift
0(2x3	, x) $	reduce S→x
0(2S5	, x) $	reduce L→S
0(2L4	, x) $	shift
0(2L4,7	x) $	shift
0(2L4,7x3	) $	reduce S→x.
0(2L4,7S8	) $	reduce L→L,S
0(2L4	) $	shift
0(2L4)6	$	reduce S→(L)
0S1	$	accept

$\Rightarrow E' \to E$
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

$E' \to E$
1) $E \to E + T$
2) $E \to T$
3) $T \to T * F$
4) $T \to F$
5) $F \to (E)$
6) $F \to id$

	E	T	F
first	(id	(id	(id
foll	$ +)	$ + *)	$ + *)

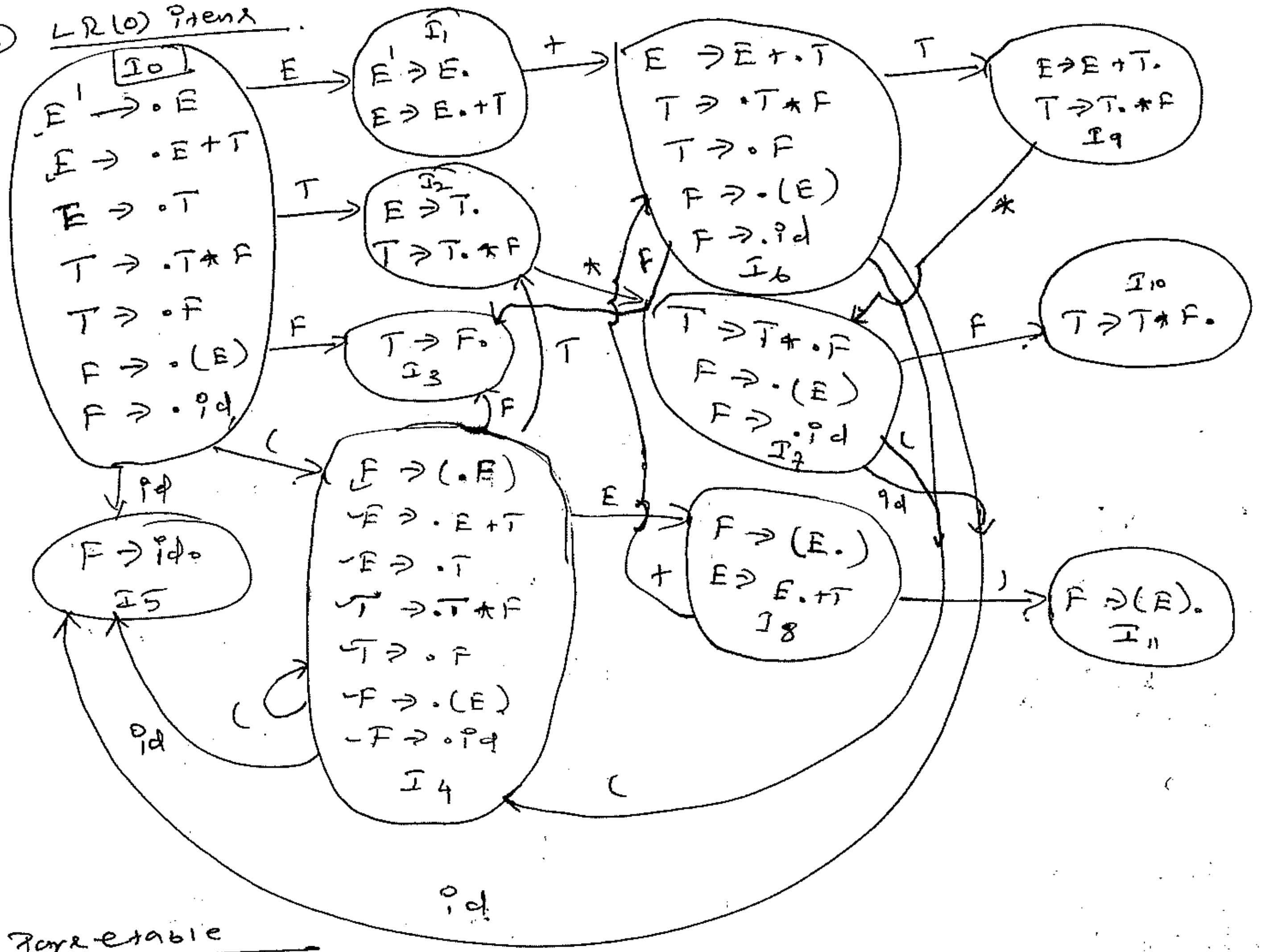
S) first (F) = { (id }

first (T) = { (id }

first (E) = { (id }

NT	Prodn	follow
E	$E \to E + T$ $A \to \alpha B \beta$	foll(E) = first(+T) = +
	$E \to E + T$ $A \to \alpha B$	foll (T) = foll (E)
	$E \to T$ $A \to \alpha B$	foll(T) = foll(E)
T	$T \to T * F$ $A \to \alpha B \beta$	foll(T) = first(* F) = *
	$T \to T * F$ $A \to \alpha B$	foll(F) = foll (T)
	$T \to F$ $A \to \alpha B$	foll (F) = foll (T)
F	$F \to (E)$ $A \to \alpha B \beta$	foll (E) = first{)} =)

(S2) LR(0) items.



(S3) Parse table

	←	action			→	←	goto	→
	+	*	(	id	$	)	E	T	F
0			s4	s5		0	1	2	3
1	s6				accept				
2	r2	s7			r2	r2			
3	r4	r4			r4	r4	8	2	3
4			s4	s5					
5	r6	r6			r6	r6		9	3
6			s4	s5				10	10
7			s4	s5					
8	s6					s11			
9	r1	s7			r1	r1			
10	r3	r3			r3	r3			
11	r5	r5			r5	r5			

Parse String id*id+id.

Stack	i/p	action
0	id*id+id $	shift
0 id 5	* id + id $	reduce F→id
0 F 3	* id + id $	reduce T→F
0 T 2	* id + id $	shift
0 T 2 * 7	id + id $	shift
0 T 2 * 7 id 5	+ id $	reduce F→id
0 T 2 * 7 F 10	+ id $	reduce T→T*F
0 T 2	+ id $	reduce E→T
0 E 1	+ id $	shift
0 E 1 + 6	id $	shift
0 E 1 + 6 id 5	$	reduce F→id
0 E 1 + 6 F 3	$	reduce T→F
0 E 1 + 6 T 9	$	reduce. E→E+T
0 E 1	$	accept

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Prob 3: $S' \to \cdot S$

1) $S \to eA$

2) $S \to d$

3) $A \to ad$

4) $A \to a.$

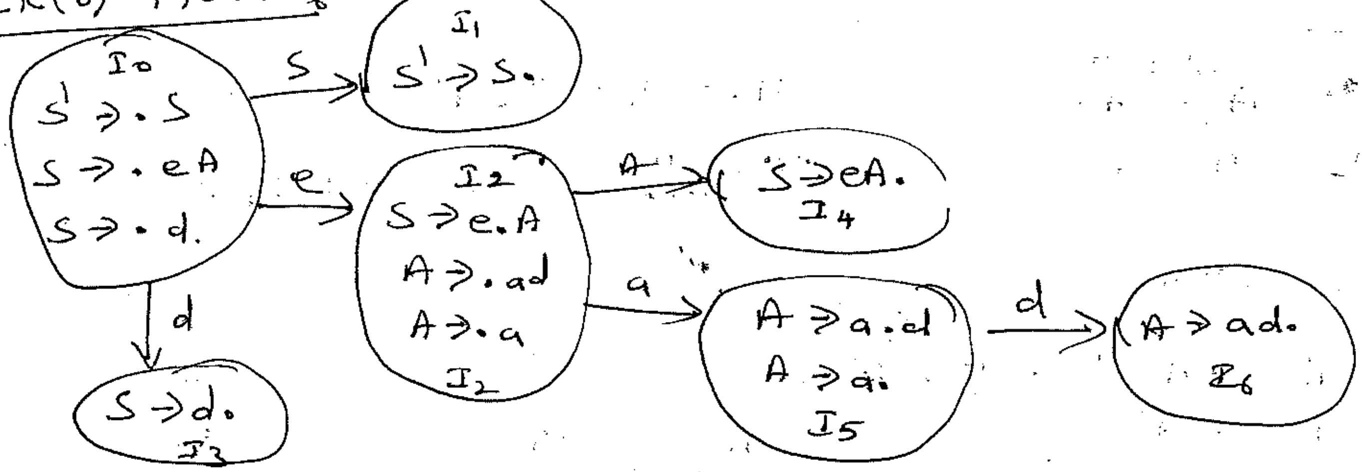
first (S) = {e d}

first (A) = {a}

	S	A
first	e d	a
foll	$	$.

NT	Prodn	foll
S	$S \to e\ A$ $A \to \underline{\alpha}\ B$	foll (A) = foll (S)

LR(0) items :



Parse table

	e	a	d	$	S	A	
0	S2		S3		1		
1				accept			
2		S5				4	
3				r_2			
4			[illegible]	r_1			
5			S6	r_4			
6				r_3			

Parse String   ead

stacks	i/p	action
0	ead $	shift
0e2	ad$	shift
0e2a5	d$	shift
0e2a5d6	$	reduce A → ad.
0e2A4	$	reduce S → eA
0S1	$	accept

$A' \to A$

Prob45 1) $A \to (A)$

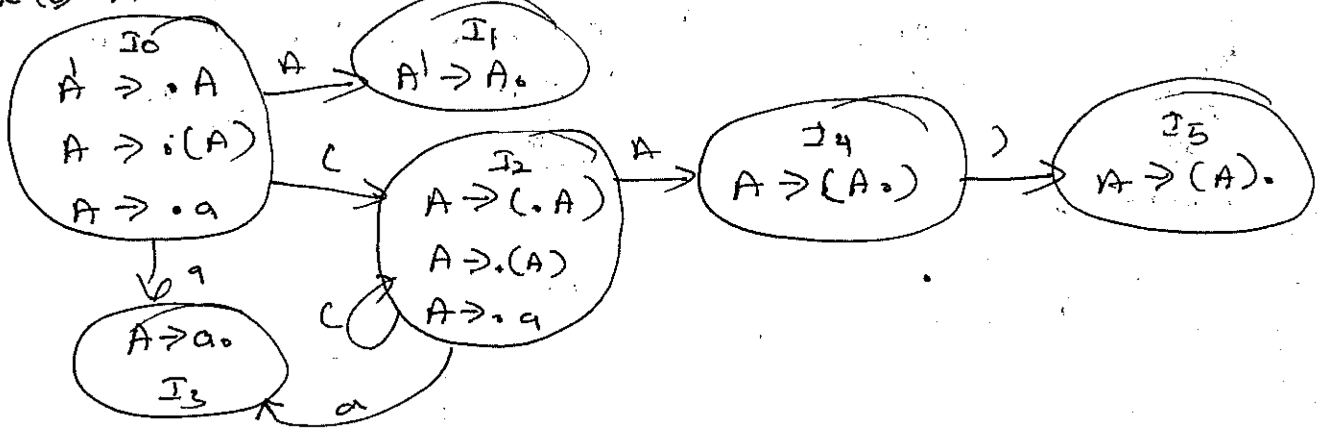
Prob46 2) $A \to a$

first(A) = {(, a}

$A \to \underset{\alpha B \beta}{(A)}$

foll(A) = {$,)}

LR(0) items



Parse table

	(	)	a	$	A
0	S2		S3		1
1				accept	
2	S2		S3		4
3		r2		r2	
4		S5			
5		r1		r1	

Parse string ((a))

Stack	i/p	actions
0	((a))$	shift
0(2	(a))$	shift
0(2(2	a))$	shift
0(2(2a3	))$	reduce A→a
0(2(2A4	))$	shift
0(2(2A4)5	)$	reduce A→(A)
0(2A4	)$	shift
0(2A4)5	$	reduce A→(A)
0A1	$	accept

Prob 5:

$S \to C_1 C_1$

$C_1 \to c C_1$

$C_1 \to d.$

w= cdcd

Prob 6:

$S \to L^{*} = R$

$S \to R$

$L \to * R$

$L \to id$

$R \to L.$

w= id = id

Prob 5:

$S' \to S$

1) $S \to CC$

2) $C \to cC$

3) $C \to d$

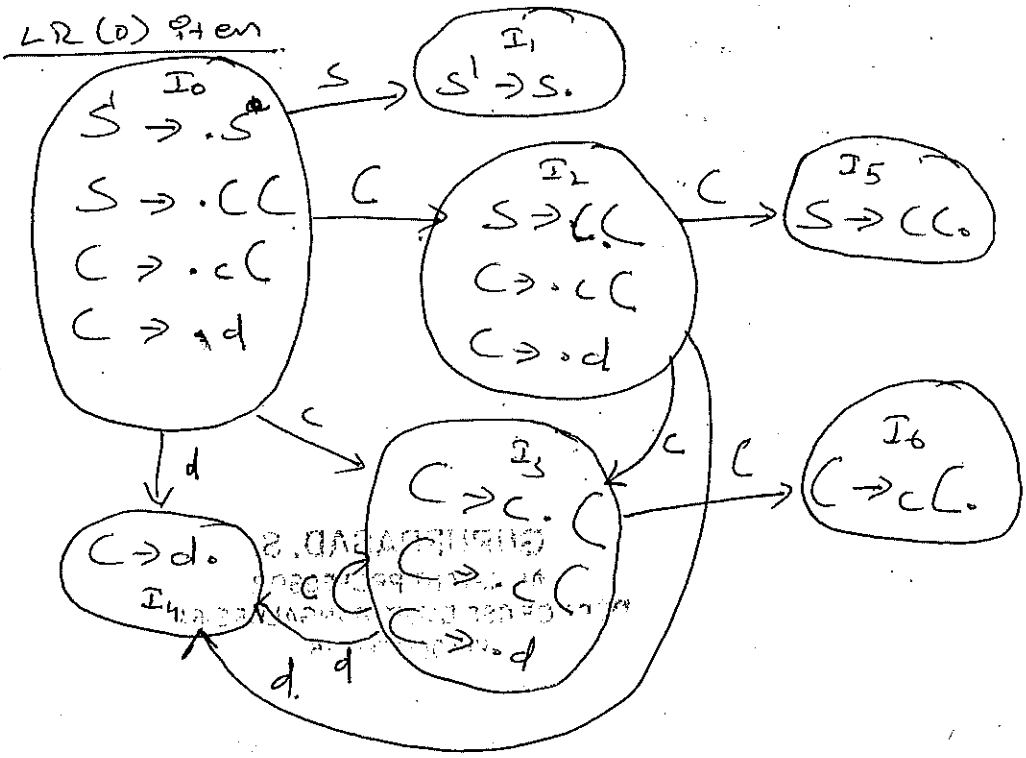
first(S) = first(C)

first(C) = {c d}

	S	C
first	c d	c d
foll	$	$ c d

NT	Prodn	foll
S	$S \to CC$ $A \to \alpha\, B\, B$	$S \to \$$ foll(C) = first(C) foll(C) = foll(S)
	$S \to CC$ $A \quad \alpha\, B$	foll(C) = foll(S)
C	$C \to cC$ $A \to \alpha\, B$	foll(C) = foll(C)

LR(0) item



	action			goto	
	c	d	$	S	C
0	s3	s4		1	2
1			accept		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

stack	i/p	action
0	cdcd$	shift
0c3	dcd$	shift
0c3d4	cd$	reduce C→d.
0c3C6	cd$	reduce C→cC.
0C2	cd$	shift
0C2c3	d$	shift
0C2c3d4	$	reduce C→d
0C2c3C6	$	reduce C→cC
0C2C5	$	reduce S→CC
0S1	$	accept

Prob 6:

$S' \to S$

1) $S \to L = R$

2) $S \to R$

3) $L \to * R$

4) $L \to id$

5) $R \to L$.

first(S) = first(L) = {*, id}

first(L) = {*, id}

first(R) = first(L) = {*, id}

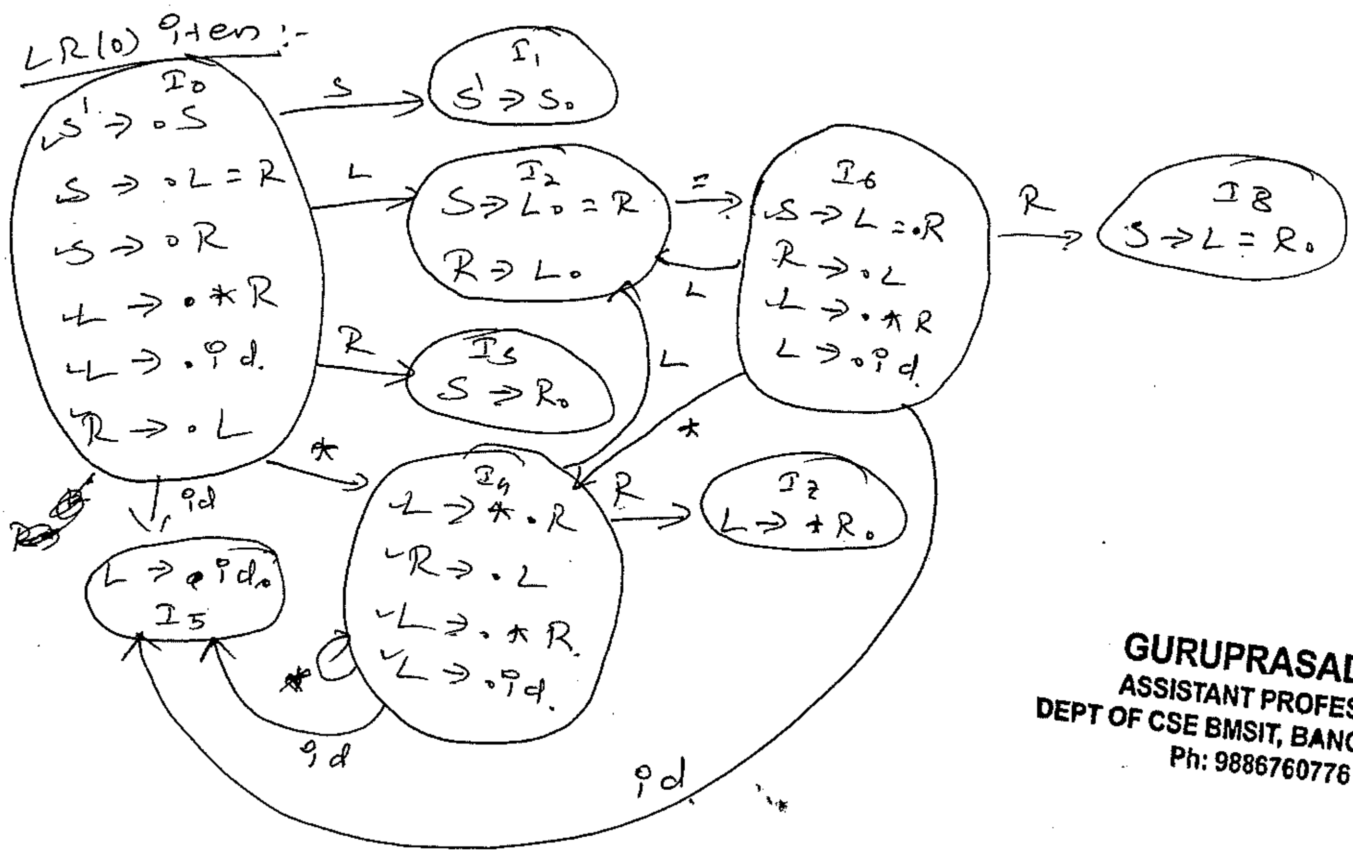
	S	L	R
first	* id	* id	* id.
foll	$	= $	$ =

NT	Prodn	foll
S	$S \to L = R$ $A \to \alpha B \beta$	foll(L) = first(=R) = =
	$S \to L = R$ $A \to \alpha B$	foll(R) = foll(S)
	$S \to R$ $A \to \alpha B$	foll(R) = foll(S)
L	$L \to * R$ $A \to \alpha B$	foll(R) = foll(L)
	$L \to id$	foll(L) = foll(R)
R	$R \to L$ $A \to \alpha B$	

Parsing table

0

LR(0) item :-



GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

	*	id	=	$	S	L	R
0	s4	s5			1	2	3
1				accept			
2			r5 s6	r5			
3				r2			
4	s4	s5				2	7
5			r4	r4			
6	s4	s5				2.	8
7			r3	r3			
8				r1			

Stack	i/p	action
0	id = id $	shift
0id5	= id $	reduce L → id
0L2	= id $	shift
0L2=6	id $	shift
0L2=6id5	$	reduce L → id
0L2=6L2	$	reduce R → L
0L2=6R8	$	reduce S → L=R
0S1	$	accept

End of unit ③

Syntax Directed Translation

Semantic Analysis

why:

→ Syntatic correctness was checked during Parsing

→ Another level of correctness is not captured like

- * var is declared/not
- * Types are consistant
- * if x = y is y assignable to x
- * fn calls have right no & type of parameters
- * if p.q is q is member of obj p
- * is var x is initilized before use etc

what:

→ Semantic action include

(i) symbol table handling

- * maintain info abt declared symbols.
- * info abt types
- * info abt scopes.

(ii) Checking context cond^ns

- * scope rule
- * type checking

(iii) Invocation of code generation routines.

It is done during reduction during Parsing

How:

* It is done by using a techinque called Syntax Directed Defination (SDD)

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

The semantic analysis involve both the description of analysis & the Implementation algorithms

In Parsing CFG is used as description & BU & TD parsing used as algorithms

but

There are no standard method to specify semantics of lang as it varies from lang to lang

One method used is to identify attributes of Prg lang entities that must be computed and to write. attribute equations or semantic rules

This method is called Syntax Directed Definations.

In SDD attributes are directly associated with grammar symbols

If X is a grammar symbol and a is an attribute then

X. a repn the value of a associated to X

In SDD grammar symbols (T & NT) are attached with attributes based on info of Prg lang constructs

Values of these attributes are computed by semantic rules associated with grammar production.

Each node in Parse tree now act as record holdg attrib values

Attributes in SDD could be of any kind like.

- numbers. → tablereference
- types - name etc

Each accourance of grammar Symbol will have seperate instance of attribute.

Types of Attributes

There are two types of attribs.

① Synthesized

② Inherited

① Synthesized Attribute.

* The value of these attributes is computed by the value of attributes of its children in parse tree.

Eg:- $E \to E + num \mid num$

$E_2.val = E_1.val + num.val$

* It is used with LR Parsers.
* The SDD is S-attributed if every attrib is synthesized.

② Inherited Attributes

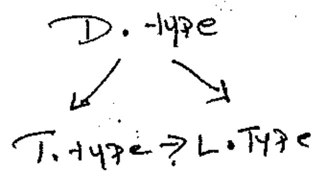
* The value of attribute is computed from the attribute value of its parent | sibling in parse tree.

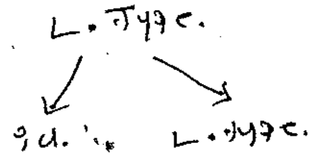
$D \to TL;$

$T \to int \mid float$

$L \to L, id \mid id$

Eg:-





* The SDD is L-attributed if val of attrib is synthesized / inherited

Semantic Rules in SDD

They are the rules define the value of attribute of a grammar symbol.

Eg:-

Production	Semantic Rules
$L \to En$	$L.val = E.val$
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val \times F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to digit$	$F.val = digit.lexval$

Evaluating an SDD at Nodes of a Parse Tree

The Parse tree showing values of it's attributes is called an "Annotated Parse tree".

To Construct Annotated Parse Tree:-

(**) Before we evaluate an attribute at a node of Parse tree we must evaluate all the attributes upon which its value depends.

So with synthesized we can evaluate all children before parent in bottom up manner.

but both with inherited & synthesized attribute there will no one order

Eg:- $A \rightarrow B$

Semantic rule

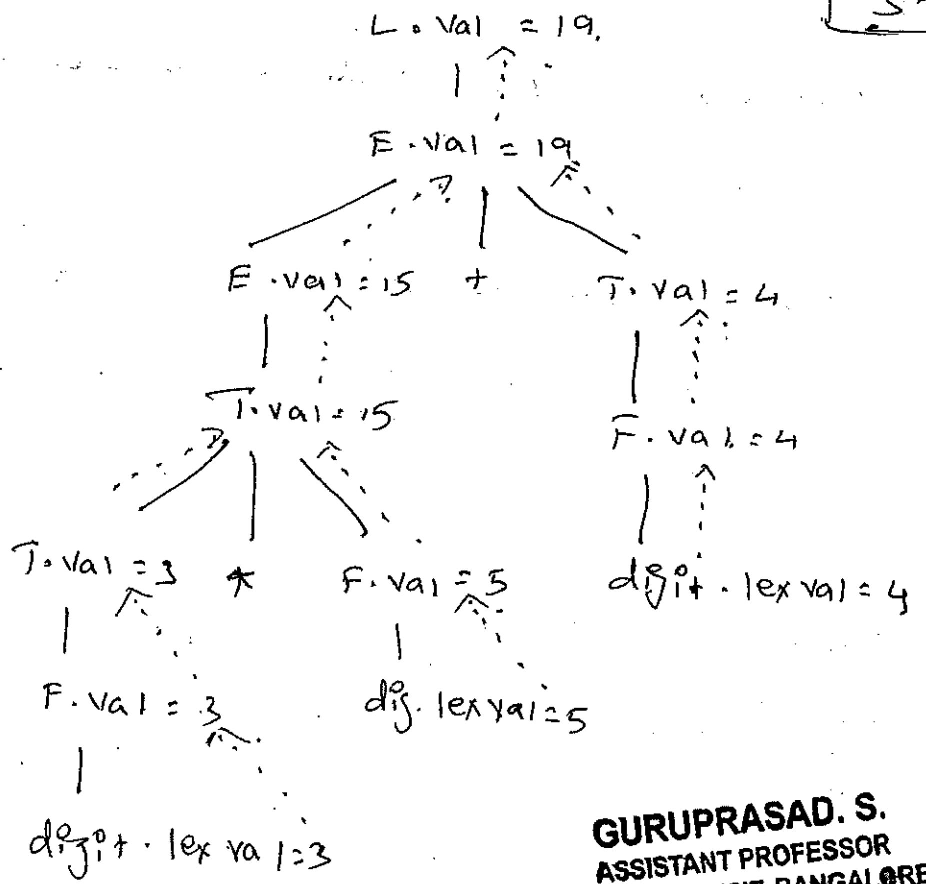
$A.s = B.i$

$B.i = A.s+1$

this is impossible to evaluate due to circularity.

Annotated Parse tree. Eg continued.

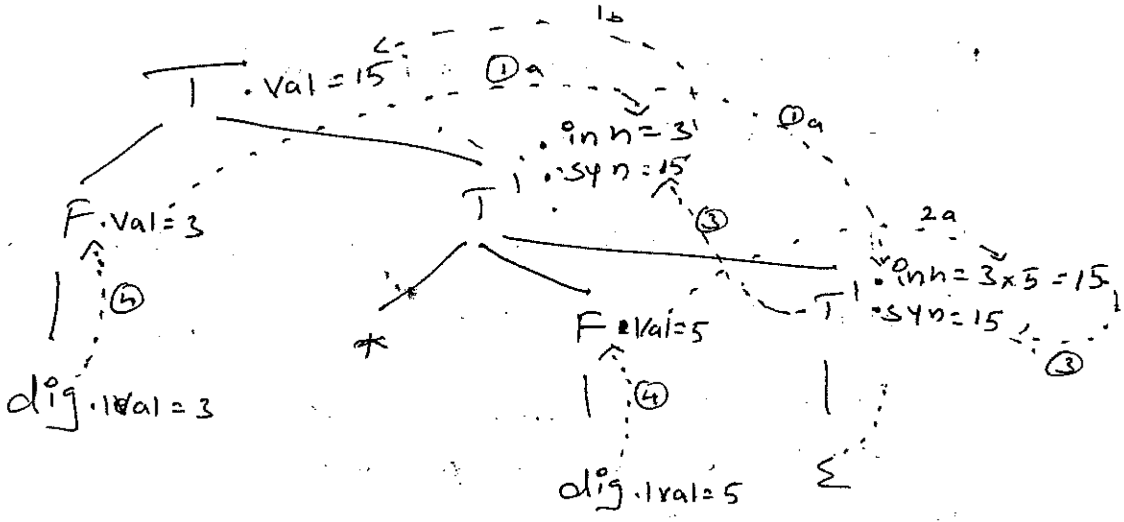
$\boxed{3*5+4=19}$



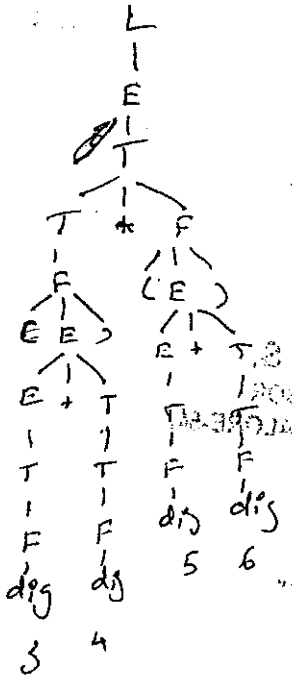
Eg ②:

Production	Semantic Rules
$T \to FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' = * F T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
$T' \to \varepsilon$	$T'.syn = T'.inh$
$F \to dig$	$F.val = dig.lexval$

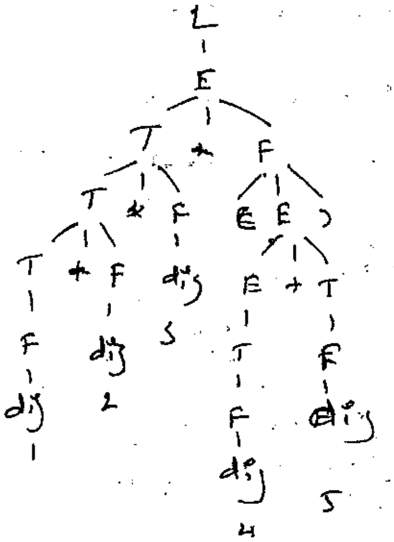
3 * 5



① (3+4)* (5+6)n



② 1 * 2 * 3 * (4+5)n



Evaluation orders of SDDs.

ependancy Graph:

Dependancy Graphs are usefull tools for ~~evaluating~~ determining an valuation order for attrib instance in given Parse tree.

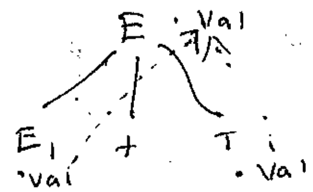
shows how values are calculated in annotated Parse tree.

It depicts the flow of info among the attrib instance in a articular Parse tree.

An edge from one attrib to another means that the value of first need to be computed before the second.

- * for each Parse node with symbol X the DG has node associated with X
- *. If the semantic rule of $Prod^n$ P define value of synthesized attrib A.b in terms of val of X.c then DG will have edge from X.c to A.b
  (where X is always child)
- * If semantic rule of $Prod^n$ P define value of inherited attrib B.c in terms of val of X.a then DG has an edge from X.a to B.c
  (where X is Parent or Sibling)

Eg: Synthesized.

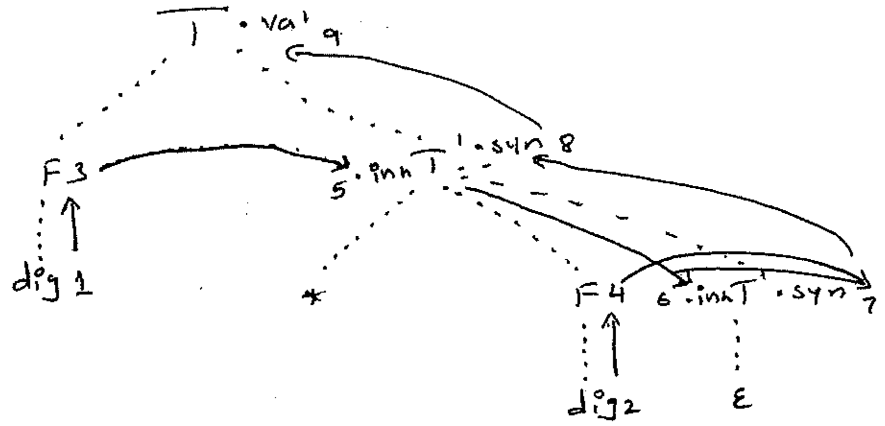


GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64.
Ph: 9886760776

Eg:- Inherited

order start from

1 - 9.



DG specify the order in which we can Evaluate the attrib at various nodes of Parse tree.

If DG has edge from node M to N then M has to be Evaluated before N, each ordering makes a directed graph to linear Order. and is called as Topological Sort

If there is a cycle in the graph then there are no topological sorts, then there is no way to Evaluate the SDD.

S- attributed Definations

An SDD is S-attributed if Every attrib is Synthesized.

When an attrib is S-attributed. we can Evaluate. it's attrib in any BU order i.e we apply Post order traversal

It is impl^n with LR Parsers. using Parsing stack.

L - attributed Definations

SDD is L-attributed if Each attrib must be Either.

(i) Synthesized

(ii) Inherited. but with rules that
if Prodn $A \to X_1, X_2 \cdots X_n$ then $X_i.a$ can be computed. only by
(a) inherited attrib associated with 'A'

(b) Either inherited or synthesized attrib associated with $X_1 \cdots X_{i-1}$ located to left of X_i

(c) Either inherited or synthesized attrib associated with X_i itself but no way cycles in DG.

Semantic Rules with Controlled Side Effects

In practice translation involve side effects,

Eg:- Calculator print result

Entr ST for type of id etc.

We shall controll the side effects in SDD to allow consistant evaln with DG. viz

* Permit incidental side effects that do not constrain attribute evaluation.

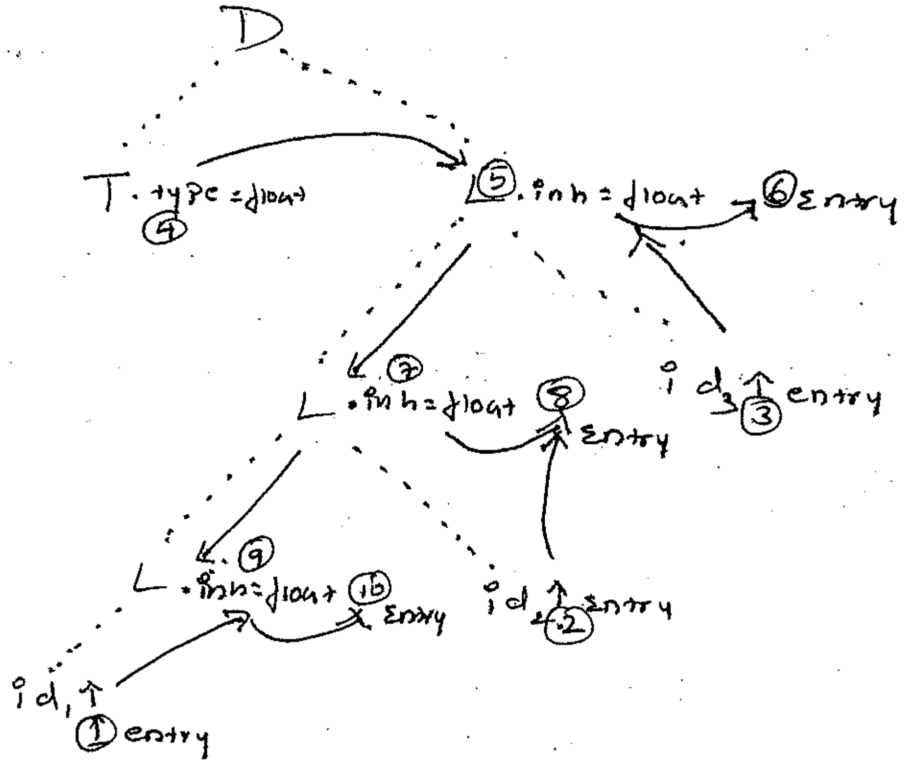
* Constrain the allowable evaluation orders, so that the same translation is produced for any allowable order.

Eg:-

Productions.	Semantic Rules
1) D → T L	L.inh = T.type
T → int	T.type = integer
T → float	T.type = flow
L → L₁, id	L_1.inh = L.inh addType (id.entry, L.inh)
L → id	add type (id.entry, L.inh)

D - Declrⁿ   T - type   L - list of int

w: float id_1, id_2, id_3



& int a b c.   float w x y z

Application of Syntax Directed Translation

Compilers use. Syntax tree as intermediate Repⁿ i.e. SDD converts i/p string to syntax tree - the compiler walks through these tree using another set of rules to translate it into intermediate Code.

Construction of syntax tree.

~~* Each object will have an 'op' field. that is the label of nodes~~

* Each node in syntax tree repⁿ one object, it will be record/ construct holding info abt obj & ptrs to child nodes.

* If node is leaf it holds lexical value for the leaf.

  Leaf (op, val)

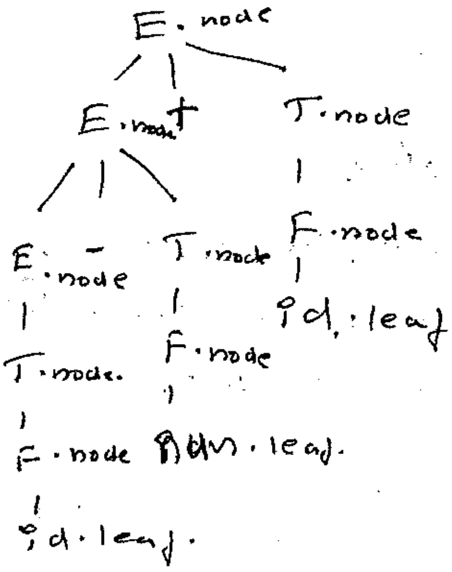
* If node is an interior node. it holds ptrs to its children

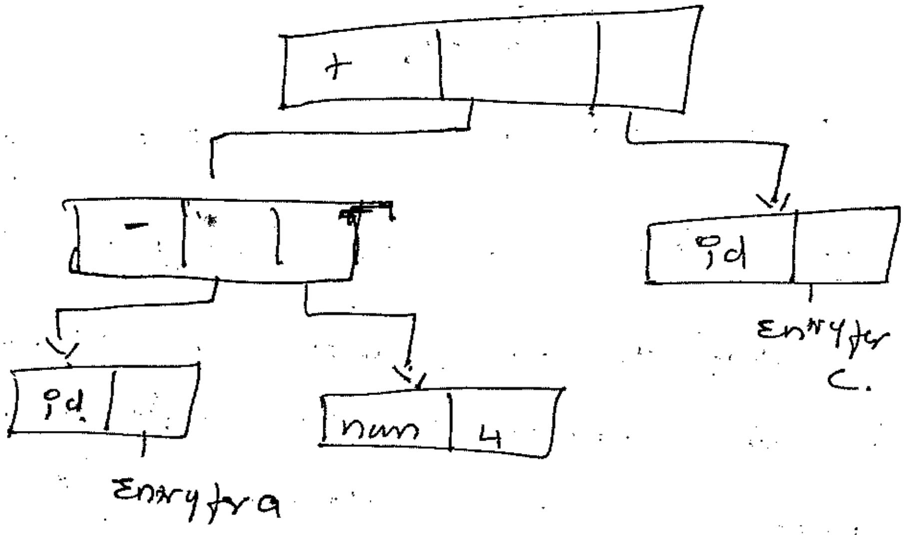
  Node $(op, c_1, c_2 \cdots c_n)$

Eg:-

Prodn	Semantic Rules
$E \rightarrow E_1 + T$	E.node = new Node ('+', E_1.node, T.node)
$E \rightarrow E_1 - T$	E.node = new Node ('-', E_1.node, T.node)
$E \rightarrow T$	E.node = T.node
$T \rightarrow (E)$	T.node = E.node
$T \rightarrow id$	T.node = new Leaf (id, id.entry)
$T \rightarrow num$	T.node = new Leaf (num, num.val)

a - 4 + c





P_1 = new leaf (id, Entry a)

P_2 = new leaf (num, 4)

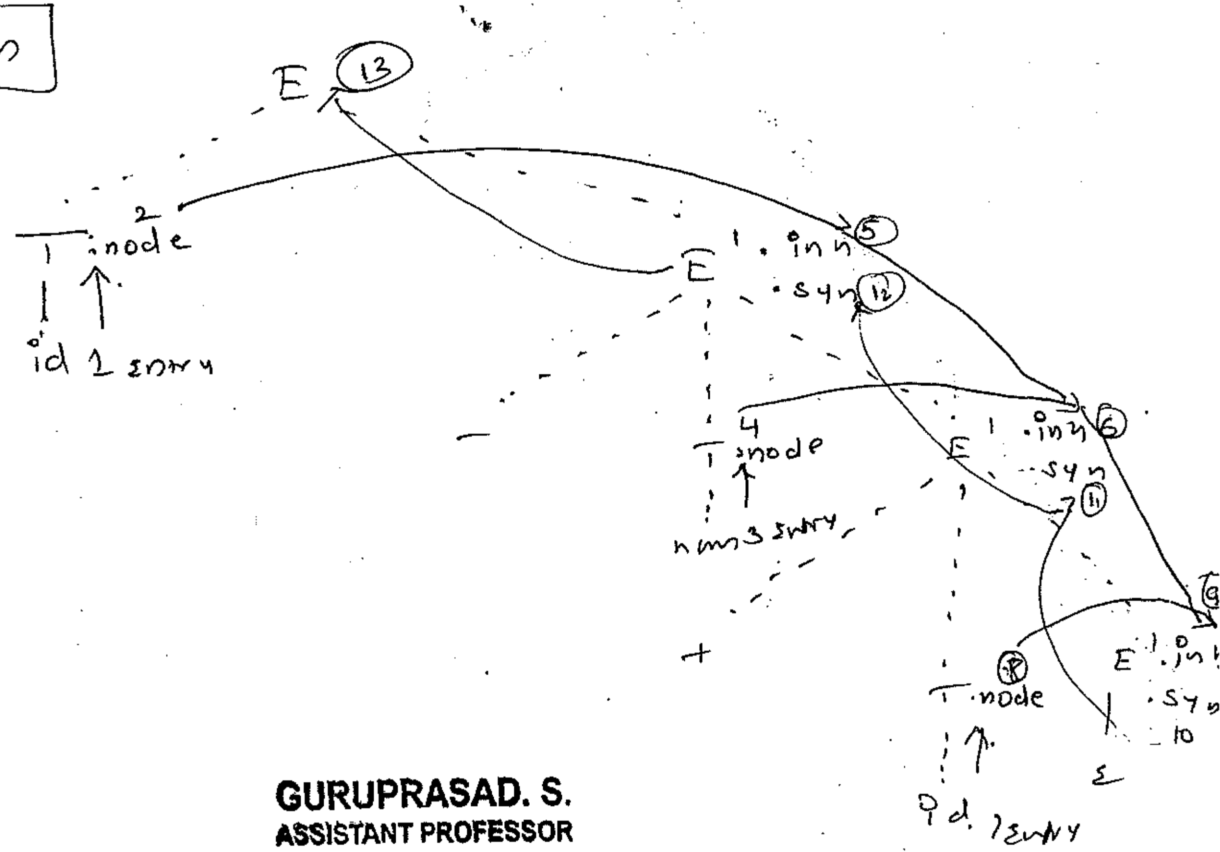
P_3 = new Node ('-', P_1, P_2)

P_4 = new leaf (id, entry c)

P_5 = new Node ('+', P_3, P_4)

Production	Semantic Rules
$E \to T E'$	$E.node = E'.syn$ $E'.inh = T.node$
$E' \to + T E'_1$	$E_1'.inn = new\ Node('+', E.inh . T.node .$ $E'.syn = E_1'.syn$
$E' \to - T E'$	$E_1'.inn = newNode('-', E.inh . T.node)$ $E'.syn = E_1'.syn$
$E' \to \varepsilon$	$E'.syn = E'.inh$
$T \to (E)$	$T.node = E.node.$
$T \to id$	$T.node = new\ leaf(id, id.entry)$
$T \to num$	$T.node = newleaf(num, num.entry)$

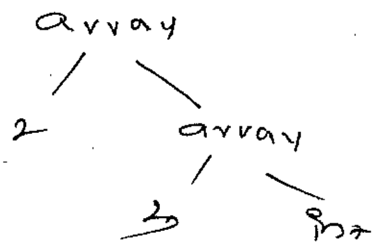
Top down



eg2: Structure of type.

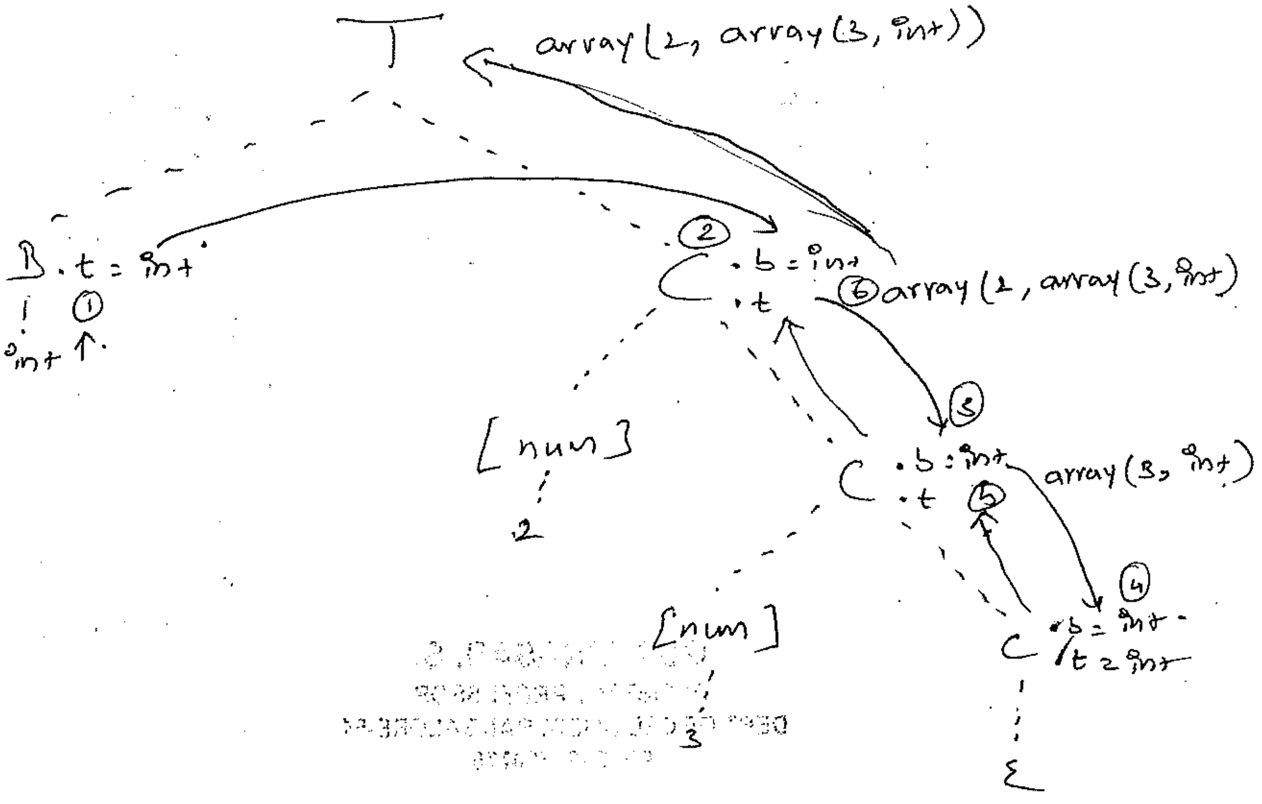
In C int [2] [3] array of 2 array of 3 integers.

repn as



Prodn	Static Rules
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow int$	$B.t = int$
$B \rightarrow float$	$B.t = float$
$C \rightarrow [num]\ C_1$	$C.t = array(num.val, C_1.t)$ $\boxed{C_1.b = C.b}$
$C \rightarrow \varepsilon$	$C.t = C.b$

int [2] [3]



Syntax Directed Translation Scheme

SDT schemes are complementary notation to SDD, it is ~~application~~ implimentation of SDD

SDT ~~is~~ contains CFG embeded with semantic actions appears at any position in production body

Any SDT can be impl^n by first building a parse tree & then performing the actions from left to right in depth first order.

SDT's are implemented on two important classes of SDDs

1) underlying grammar is LR parsable & SDD is S-attributed.
2) underlying grammar is LL parsable & SDD is L-attributed.

The objective is to convert semantic rules in SDD into SDT with actions executed at right time during parsing.

SDT impl^n during parsing via a marker nonterminal M where
$M \rightarrow \varepsilon$

Postfix translation Schemes

SDT for S-attributed SDD will contain actions at the end of prod^n and is executed along with reduction of the body to head of prod^n they are called Postfix SDT's

Eg:-

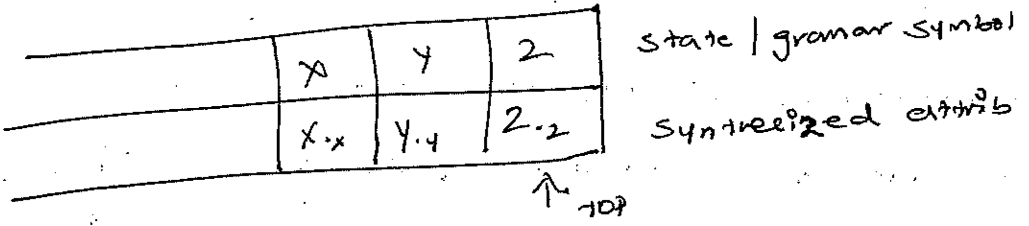
1) $L \rightarrow E n$ $\{Print(E.val);\}$
2) $E \rightarrow E_1 + T$ $\{E.val = E_1.val + T.val;\}$
3) $E \rightarrow T$ $\{E.val = T.val\}$
4) $T \rightarrow T_1 * F$ $\{T.val = T_1.val \times F.val\}$
5) $T \rightarrow F$ $\{T.val = F.val\}$
6) $F \rightarrow (E)$ $\{F.val = E.val\}$
7) $F \rightarrow dig$ $\{F.val = dig.lexval\}$

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Parser stack impl^n of Postfix SDT

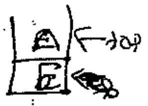
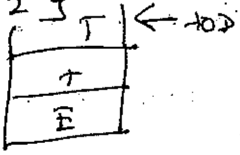
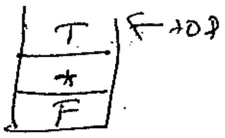
Postfix SDT can be impl^n during LR parsing by executing the ction when reduction occurs.
he attribute of each grammar symbol could be put on the stack & can be used while reduction.

Eg:- A → X Y Z



If the attribute are synthesized the action occurs at the end of Production i.e during reduction.

Eg:-

Prodn	Action	
$L \to E n$	{ print (stack[top-1].val); top = top-1;}	
$E \to E_1 + T$	{ stack[top-2].val = stack[top-2].val + stack[top].val top = top-2 }	
$E \to T$		
$T \to T * F$	{ stack[top-2].val = stack[top-2].val + stack[top].val top = top-2 }	
$T \to F$		
$F \to (E)$	{ stack[top-2].val = stack[top-1].val top = top-2;}	
$F \to dig$		

SDT's with action inside Productions

An action may be Placed at any Pos'n within the body of Prodn & is Performed immediatly after all symbols to its left are Processed.

thus if we have Prodn $B \rightarrow X \{a\} Y$ the action a is done after we have recognized X (if X is terminal) or all terminals derived by X (if X is non terminal)

In BU Parse Perform action 'a' as soon as the occurance of X on Parsing stack.

In TD Parse Perform action 'a' before we attempt to Expand the occurance of Y.

Not all SDT's can be impl'n during Parsing

Eg:- $L \rightarrow En$

$E \rightarrow \{Print('+');\} \; E_1 + T$

$E \rightarrow T$

$T \rightarrow \{Print('*');\} \; T * F$

$T \rightarrow F$

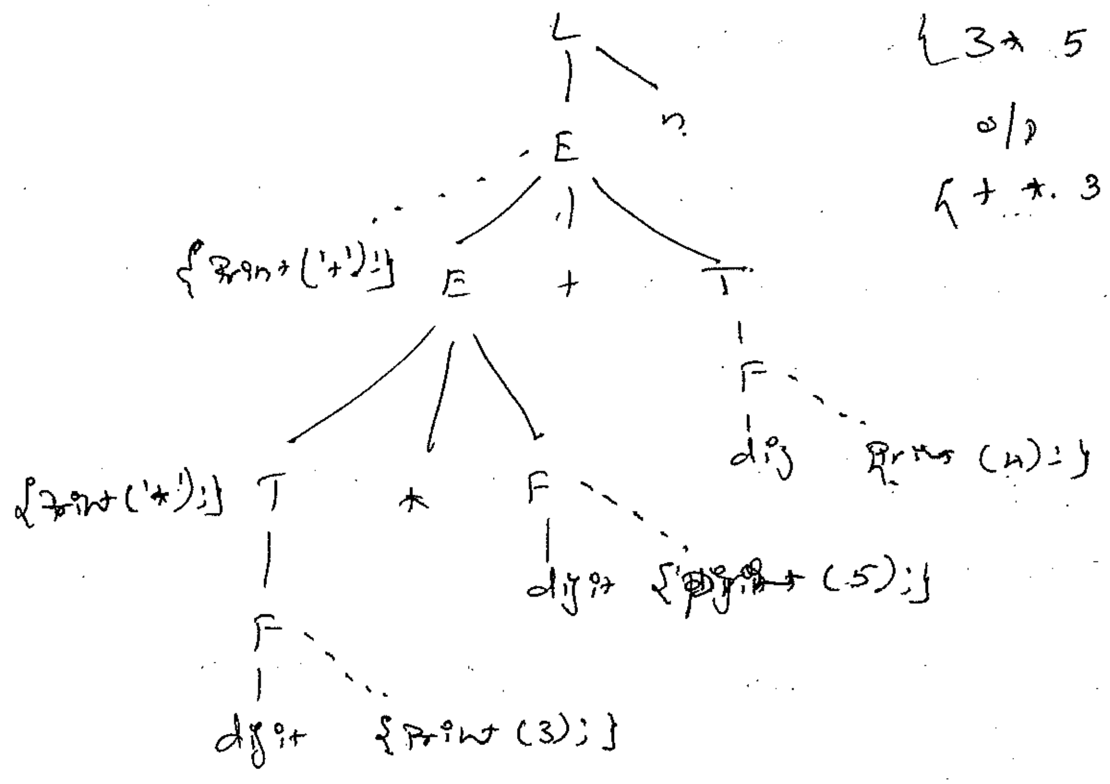
$F \rightarrow (E)$

$F \rightarrow digit \; \{Print (dig.lex val)\}$

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64.
Ph: 9886760776

It is impossible to impl'n this SDT either TD / BU parsing as Parser has to Print instances of * + before knowing weather it appears or not

So we marker nonterminals M_2 M_4 for 2 & 4 Prod'n actions where $M_2 \rightarrow \varepsilon$ & $M_4 \rightarrow \varepsilon$ & shift digits then perform Pre order traversal of the tree & as soon as a node labeled by an action is visited perform action



{3* 5+4}

o/p

{+ *. 3 5 4}

Eliminating Left Recursion from SDT

Since we cant Parse a grammar with left recursion in TD it has to be Eliminated.

In case of SDT also we should eliminate left recursion to know the order in which the actions in an SDT are performed.

So $A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta R$

$R \rightarrow \alpha A \mid \varepsilon$

Eg:- $E \rightarrow E_1 + T$ {print('+');}

$E \rightarrow T$

$E \rightarrow TR$

$R \rightarrow + T$ {print('+');} $R \mid \varepsilon$

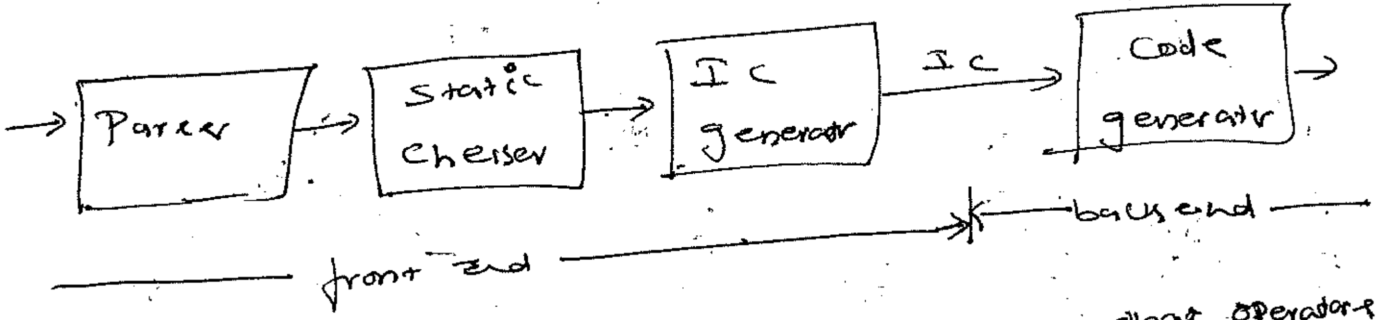
End of unit 5

Unit 6°.. Part B

Intermediate Code Generation

In analysis & synthesis model of Compiler front end analyze the src prg & creates Intermediate repn for which back end generates target code.

So details of src prg is confined to front End while target m/c details are confined to back End.



Static Checking includes type checking to ensure that operators are applied to compatible operands. also include addnl syntactic checks.

Before src prg is converted to target prg a sequence of intermediate repn are constructed. some High level close to prg lang like Syntax tree, 3-addr code. & low level close to target m/c repn

Etc.

Variants of Syntax Tree

The nodes in syntax tree repn constructs of src prg; the children of a node repn the meaning full compont. of construct

A Directed Acyclic Graph (DAG) for an Exprn identifies common sub expressions (sub exprn that occur more than once) of the Exprn.

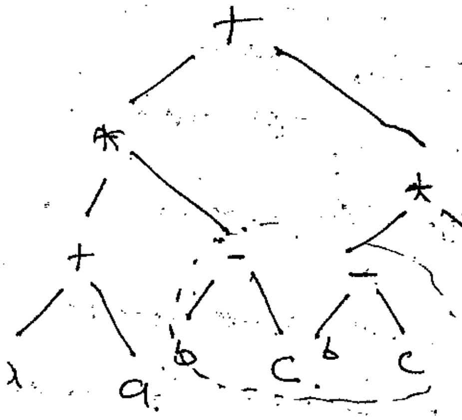
DAG can be constructed using same technique that construct syntax tree / Abstract syntax tree

DAG for Expressions

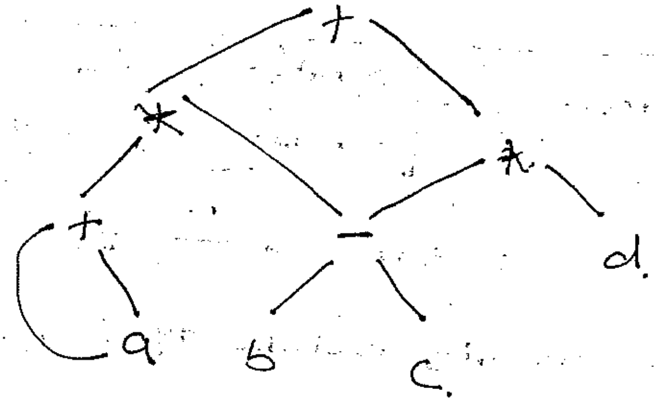
It repn Exprn & gives compiler important clues regarding the generation of efficient code to Evaluate the exprn

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

AST:

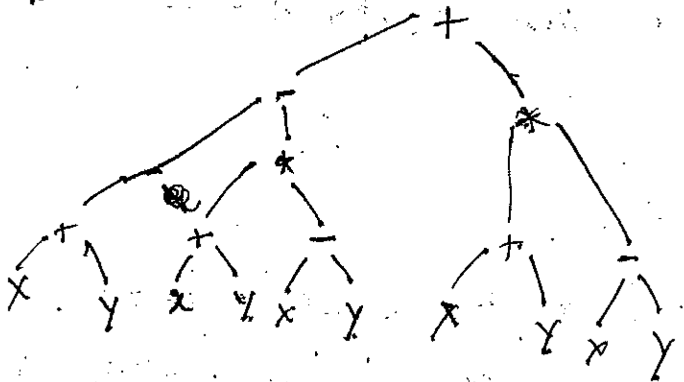


DAG:



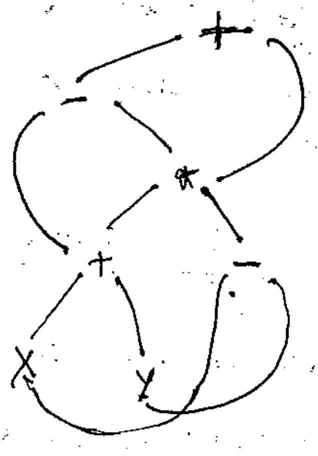
$((x+y) - ((x+y) * (x-y))) + ((x+y) * (x-y))$

AST



19 nodes

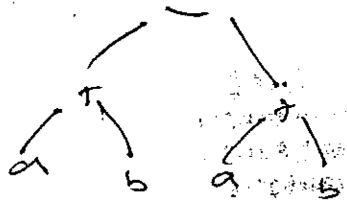
DAG:



7 nodes

$(a+b) - (a+b)$

AST



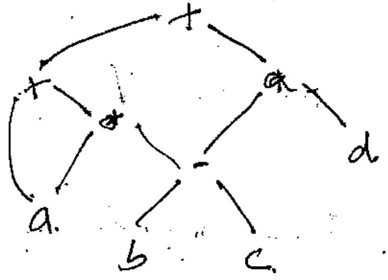
DAG.

Construction of syntax tree for simple exprn

The SDD given below can construct either syntax tree/DAG

Prodn	Semantic Rules
$E \rightarrow E_1 + T$	E.node = newNode('+', E_1.node, T.node)
$E \rightarrow E_1 - T$	E.node = newNode('-', E_1.node, T.node)
$E \rightarrow T$	E.node = T.node.
$T \rightarrow (E)$	T.node = E.node.
$T \rightarrow id$	T.node = newleaf (id, id.Entry)
$T \rightarrow num$	T.node = new leaf (num, num.val)

Construct syntax tree for $a + a * (b - c) + (b - c) * d$.



P_1 = new leaf (id, Entry a)

P_2 = new leaf (id, Entry a) = P_1

P_3 = new leaf (id, Entry b)

P_4 = newleaf (id, Entry c)

P_5 = newnode ('-', P_3, P_4)

P_6 = newnode ('*', P_1, P_5)

P_7 = newnode ('+', P_1, P_6)

P_8 = newleaf (id, Entry b) = P_3

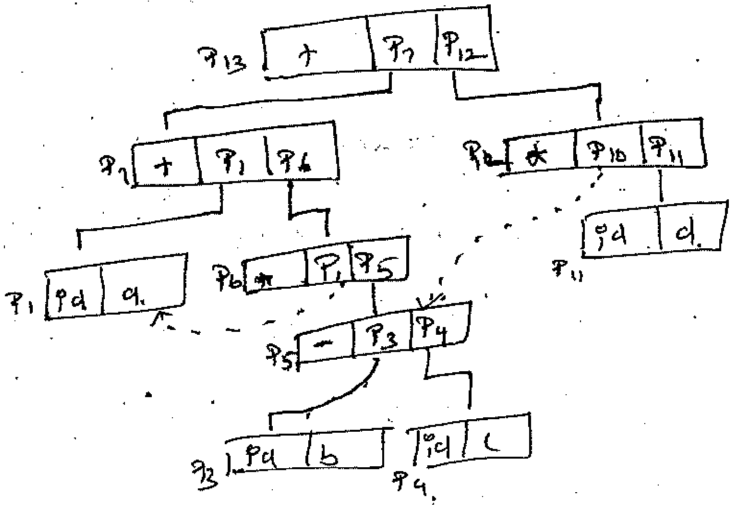
P_9 = new leaf (id, Entry c) = P_4

P_{10} = newnode (-, P_8, P_9) = P_5

P_{11} = newleaf (id, Entry d)

P_{12} = newnode (*, P_{10}, P_{11})

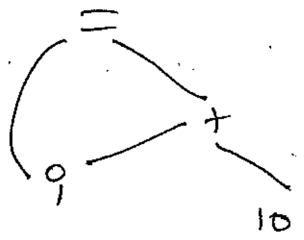
P_{13} = newnode (+, P_7, P_{12})



GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64.
Ph: 9886760776

The value number method to construct DAG.

Often the nodes of a syntax tree or DAG are stored in array of records as shown below.



Val no			
1	id		
2	num	10	
3	+	1	2
4	=	1	3

entry for i

Nodes of DAG for $i = i + 10$ allocated in an array

Algorithm for value number method

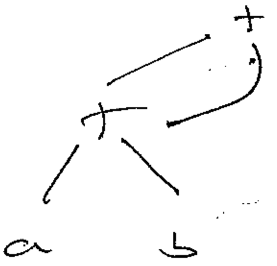
Input: label op, node l and node r

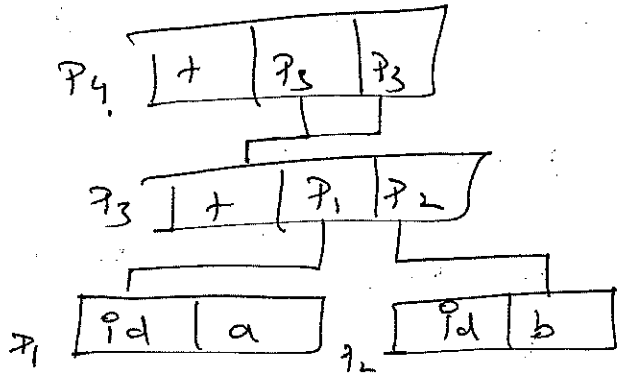
Output: The value number of a node in the array with signature ⟨op, l, r⟩

Method: Search the array for a node M with label op, left child l, and right child r. If there is such a node return the value number of M. If not create in the array a new node N with label op, left child l and right child r and return the value number.

A more efficient approach is to use a hash table in which the nodes are put into "buckets"

Eg:- $a + b + (a + b)$





P_1 = newleaf (id, Entry a)

P_2 = newleaf (id, Entry b)

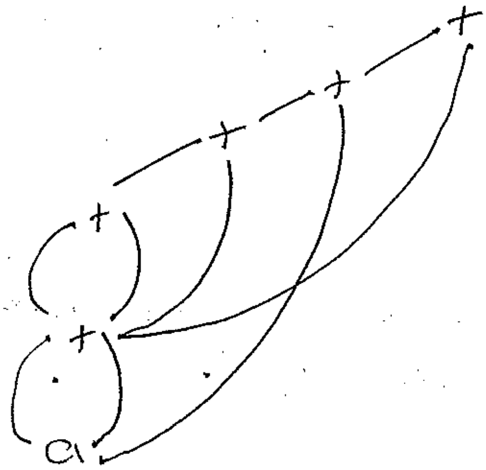
P_3 = newnode (+, P_1, P_2)

P_4 = newnode (+, P_3, P_3)

Value num

1	id			→ Entry a
2	id			→ Entry b
3	+	1	2	
4	+	3	3	

Eg:- $a + a + ((a + a + a + (a + a + a + a))$



1	id			→ Entry a
2	+	1	1	
3	+	2	2	
4	+	3	2	
5	+	4	1	
6	+	5	2	

P_1 = new leaf (id, entry a)

P_2 = new node (+, P_1, P_1) a+a

P_3 = new node (+, P_2, P_2) a+a+a+a

P_4 = new node (+, P_3, P_2)

P_5 = new node (+ P_4, P_1)

P_6 = newnode (+ P_5, P_2)

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-8
Ph: 9886760776

Three address Code

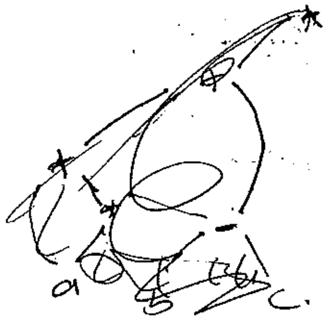
It is of the form $A = B \text{ op } C$.

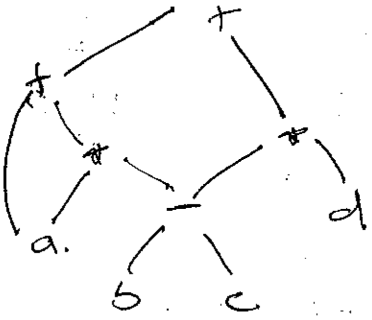
$x * y * z$ might be translated into sequence of 3 addr instrn

$t_1 = y * z$ $\quad$ t_1 t_2 are temp names.

$t_2 = x + t_1$

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





3-addr code

$t_1 = b - c$

$t_2 = t_1 * a$

$t_3 = t_2 + a$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

Three addr code is built from two concepts

(i) address $\quad$ (ii) instruction.

Address can be one of the following

a 1) a name - source prog names appear as 3 addr code, in impl
name is replaced by a ptr to ST entry.

2) a constant -

3) compiler - generated temporary names

Instructions of 3 - addr code

1) Assignment instrn of the form $\underbrace{x = y \text{ op } z}_{\text{binary}}$ or of the form $\underbrace{x = \text{op } y}_{\text{unary}}$

② Copy instrn - of the form $x = y$

3) uncondnl Jump - of the form if x goto L

4) condnl Jump - of the form if x relop y goto L

5) Procedure call & return

6) indexed copy instrn - $x = y[i]$, $a[i] = y$

7) address & pointer assignment of the form $x = \&y$ $x = *y$

Eg:- do $i = i+1$
while $(a[i] < v)$;

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

L: $t_1 = i+1$	100 : $t_1 = i+1$
$i = t_1$	101 : $i = t_1$
$t_2 = i * 4$	102 : $t_2 = i * 4$
$t_3 = a[t_2]$	103 : $t_3 = a[t_2]$
if $t_3 < v$ goto L	104 : if $t_3 < v$ goto 100
a) Symbolic labels	b) Position numbers

Quadruples

The intermediate repn of 3-addr instrn in Data Structures can be done in 3 ways (i) Quadruples (ii) Triples (iii) Indirect triples.

Quadruples has 4 fields op, arg1, arg2, result

Operator - Contains internal code for operators.

$A = -B * (C+D)$

$t_1 = -B$
$t_2 = c+d$
$t_3 = t_1 * t_2$
$A = t_3$

OP	Arg1	Arg2	Result
uminus	B	-	t_1
+	c	d	t_2
*	t_1	t_2	t_3
=	t_3	-	A

Note: unary operators & assignment op do not use arg2

Param operators do not use either arg2 nor result

uncondnl & condnl Jumps put the target label in result

The contents of arg1 arg2 & results are normally points to ST. Entries for the names, so temporary names must be Entered into ST when they are created.

Triples

The 3-addr stmts are repn by a structure with only 3 fields OP, arg1, arg2, where arg1 & arg2 are arguments of OP. they are Either ptr to ST or ptr to Structure.

we use Parenthesized nos to repn ptrs into triple structure while ST ptrs are repn by ~~the~~ names themselves.

1 ① A = -B * (C+D)

$t_1 = -B$

$t_2 = C+D$

$t_3 = t_1 * t_2$

$A = t_3$

	OP	arg1	arg2
(0)	uminus	B	—
(1)	+	C	D
(2)	*	(0)	(1)
(3)	=	A	

② A[1] = B

	OP	arg1	arg2
(0)	[]=	A	1
(1)	=	(0)	B

③ A = B[1]

	OP	arg1	arg2
(0)	=[]	B	1
(1)	=	A	(0)

i) a+a*(b-c)+(b-c)*d

$t_1 = b-c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

	OP	arg1	arg2
(0)	minus	b	c
(1)	*	a	(0)
(2)	+	a	(1)
(3)	*	(0)	d
(4)	+	(2)	(3)

$a = b * -c + b * -c$

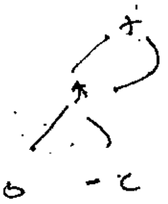
Dag:

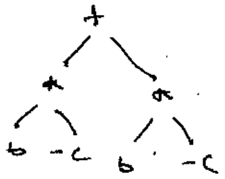
$t_1 = -c$

$t_2 = b * t_1$

$t_3 = t_2 + t_2$

$a = t_3$





3 addr:

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

	op	arg1	arg2.
(0)	uminus	c	-
(1)	*	b	(0)
(2)	uminus	c	-
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

Indirect Triples

Consists of listing pointers to triples rather than listing the triples themselves

Eg:- $a = b * -c + b * -c$.

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	......

$\Rightarrow$

	op	arg1	arg2.
(0)	uminus	c	-
(1)	*	b	(0)
(2)	uminus	c	-
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

Static Single Assignment Form

SSA is an intermediate repⁿ that facilitates certain code optimization. The diff. b/w SSA & 3-addr code is:

ⓐ all assignments in SSA are to var with distinct names

ⓑ SSA uses a notational convention called Φ function to combine the two defⁿ of x

ⓐ Eg:- $(a+b-c) + (e-a+b-c*d)$

$P = a+b$

$q = P-c$

$P = q*d$

$P = e-P$

$q = P+q$

$P_1 = a+b$

$q_1 = P_1 - c$

$P_2 = q_1 * d$

$P_3 = e - P_2$

$q_2 = P_3 + q_1$

SSA

$t_1 = a+b$

$t_2 = t_1 - c$

$t_3 = t_2 * d$

$t_4 = e - t_3$

$t_5 = t_4 + t_2$

ⓑ Eg:- if (flag) x = -1; else x = 1;

y = x * a;

SSA

if (flag) $x_1 = -1$; else $x_2 = 1$;

$x_3 = \phi(\underset{\text{true}}{x_1}, \underset{\text{false}}{x_2})$

Ex: ① $a + -(b+c)$ A Syntax tree, DAG, QT, T, IT Vn

② $a[i] = b*c - b*d$

③ $X = *p + \&y$

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Type Design

type check - runtime

type appln - stge

type Expn - no Expn

Type Expn

type Equivalence -

$\text{type check} \begin{cases} \text{Syntn} \\ \text{Int} \end{cases}$

$\text{type conversion} \begin{cases} \text{narrow} \\ \text{wide} \end{cases}$

Types & Declarations.

The appln of types can be grouped under (i) type checking & (ii) Translation

→ Type checking: uses logical rules to reason about the behaviour of a program at run time

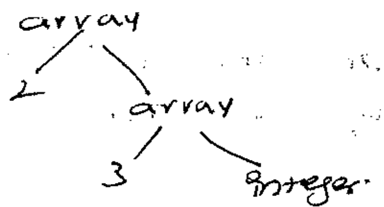
Eg:- && needs both opnds to be logical/boolean

→ Translation appln: from the type of name, a compiler can determine the storage that will be needed for the name at runtime.

Type Expressions

Types have structure, which is repn using "type expression" - a type exprn is either formed by a basic type or formed by applying an operator called a type constructor to a type exprn

Eg:- int [2][3] can be read as array of 2 array of 3 integers each & written as array (2, array (3, integer))



operator array takes two parameters number & type.

The following are defn of type Exprn:

- * A basic type is a type exprn like int, float, char, boolean, void.
- * A type name is type exprn - int a int float
- * A type exprn can be formed by applying the array type constructor to number & a type exprn. int a[10]
- * A record is a data structure with named fields, A type exprn can be formed by applying record type constructor to field names & their types

Structured & int a struct ar

- * A type exprn can be formed by using type constructor → for functiontypes int fun (a b)
- * A convinent way to repn a type exprn is to use graph or value number method.

Type Equivalence

Many type checking rules have the form "if two type Exprⁿ are equal then return a certain type else Error" this rule we need to say weather two type Exprⁿ are Equivalent

When the Exprⁿs are repⁿ by graphs two types are structurally Equivalent iff the following condⁿ is true

- * They are the same basic type.
- * They are formed by applying the same Constructor to structurally Eqt types.
- * one is a type name that denotes other.

Storage layout for local names.

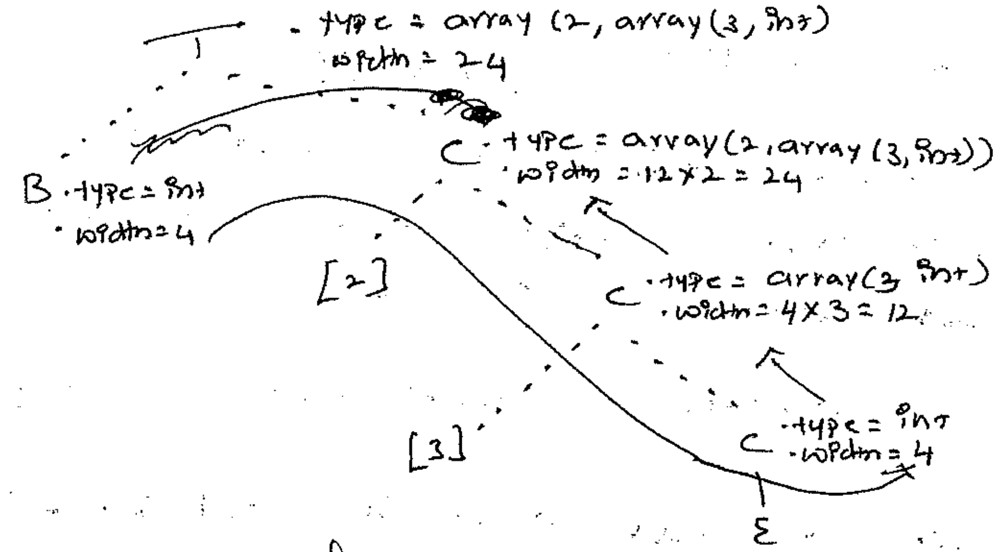
From the type of a name, we can determine the amount of storage that will be needed for the name at runtime.

At Compile time, we can use these amount to assign each name a relative address. The type & relative addr. are saved in symbol table Entry for name.

The width of a type is the no of storage units needed for objects of that type.

Prodn	Semantic Rule
T → B C	{ t = B.type ; w = B. width ; }
B → int	B.type = Integer B.width = 4;
B → float	B.type = float B. width = 8;
C → ε	C.type = t C. width = w;
C → [num] C₁	C.type = { array (num.val , C₁.type); C.width = num.val x C₁.width }

int [2] [3]



(f) Sequence of Declarations

We can calculate the relative addr of a variable in SymTab this is achived by using the method top.put which creates a SymTab entry for any variable. top indicates the current symbol table.

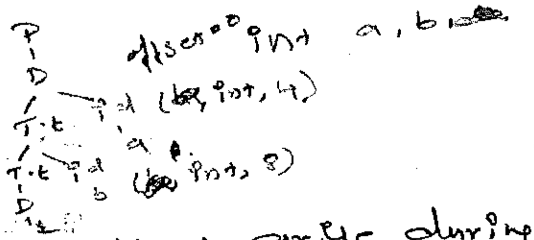
Initially offset is 0, As each new name x is enterd into SymTab with its relative addr is set to current value of offset which is incremented by width of x.

Eg:- P → D {offset = 0}

D → T id {top.put (id.lexeme, T.type, offset);
offset = offset + T.width; }

D → D1

D → ε



Translation of Expressions

Here we explore issues that arise during the translation of Expression and statements.

An Exprn with more than one operator like a+b*c will translate into instrns with atmost one operator per instruction

Translation of Expressions

Operations within Expression ④

The SDD builds up 3-addr code for an assignment stmt S using attrib code for S and attrib addr and code for E

S.code E.code - 3addr code for S & E

E.addr - addr that will hold value of E.

Prod n	Semantic Rules
S → id = E	S.code = E.code || gen (top.get (id.lexeme) '=' E.addr)
E → $E_1 + E_2$	E.addr = newTemp () E.code = E_1.code || E_2.code|| gen (E.addr '=' E_1.addr '+' E_2.addr)
E → - E_1	E.addr = newTemp () E.code = E_1.code|| gen (E.addr '=' 'minus' E_1.addr)
E → (E)	E.addr = E_1.addr E.code = E_1.code
E → id	E.addr = top.get (id.lexeme) E.code = ' '

Addressing array Elements

Array Elements can be accessed quickly if they are stored in blocks of ~~sequence~~ continuous locations, i is 0 ..... n-1 if width of each array element is w then ith element of array A begin at location

$$base + i \times w$$

where base is the relative addr of A[0] i.e A

so for 2-dimensional array

A[i_1][i_2] the relative addr is

$$base + i_1 \times w_1 + i_2 \times w_2$$

In general for k dimensional array

$$base + i_1 \times w_1 + i_2 \times w_2 + \ldots\ldots + i_k \cancel{\times} w_k$$

A two dimensional array is normally stored in one of two forms

(i) row major (row by row)

(ii) column major (column by column)

11	12	13
21	22	23

array A[2][3]

↑ A[11]
first row A[12]
↓ A[13]
↑ A[21]
sec row A[22]
↓ A[23]

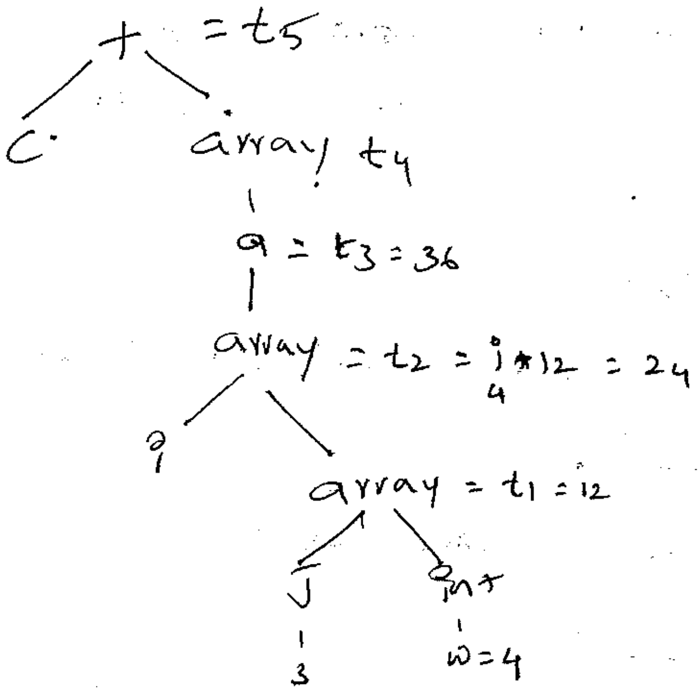
Row major

A[11] ↑ 1st col
A[21] ↓
A[12] ↑ 2nd col
A[22] ↓
A[13] ↑ 3rd col
A[23] ↓

col major

Consider the Exprn c + a[i][j] with i = 2 j = 3

the Parse tree is



$t_1 = j * 4 = 12$

$t_2 = i * t_1$

i.e.

$4 * 12 = 24$

$t_3 = t_1 + t_2$

$t_4 = a[t_3]$

$t_5 = c + t_4$

V Type checking

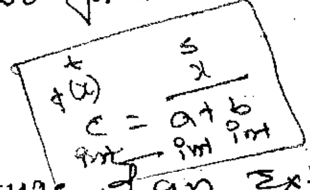
To do type checking, a compiler needs to assign a type Expr to Each component of the src Prg, the compiler must then determine that these type Expr confirm to a collection of logical rules that called the type system for src Prg

Type checking has potential for catching Errors in Prg.

Rules for type checking

Type checking can take on two forms:

(i) Synthesis (ii) Inference



(i) Type synthesis builds up the type of an Exprn from the types of its sub Exprn. it requires names to be declared before they are used.

The type $E_1 + E_2$ is defined in terms of types of E_1 & E_2

A typical rule for type synthesis.

if f has type $s \rightarrow t$ and x has type s,
then Exprn $f(x)$ has type t
where. f & x denote Exprn and $s \rightarrow t$ denotes a function of from s to t

(ii) Type inference determines the type of a language construct from the way it is used.

Eg:- null(x) tests if list is empty then x must be list of some type.

$\underset{float}{c} = \underset{float}{a} + \underset{float}{\underline{2.5 \cdot 3}}$

A typical rule for type inference is.

if $f(x)$ is an Exprn,
then for some α and β, f has type $\alpha \rightarrow \beta$ and x has type α
it is used in languages like ML where declrn of names is not required.

Type Conversion.

Type conversion is needed when types of the opnds in an Exprn is different because the operation/m/c instrn will be diff for diff type.

Eg:- $\underset{\text{int}}{\underline{2}} * \underset{\text{float}}{\underline{3.14}}$ └ float

so $t_1 = \text{float}(2)$
$t_2 = t_1 * 3.14$

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

The rule is. if $E \rightarrow E_1 + E_2$

if (E_1.type = integer and E_2.type = integer) E.type = integer;

else if (E_1.type = float and E_2.type = integer E.type = float ...

There are two major reasons for type conversion

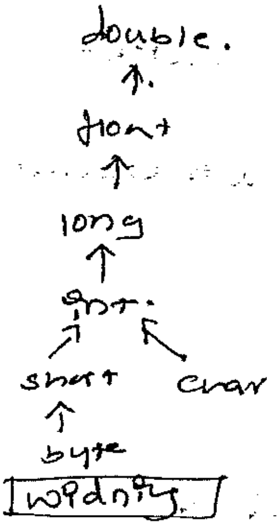
(i) widening (ii) Narrowing.

implicit / coreion conversion - done by compiler.

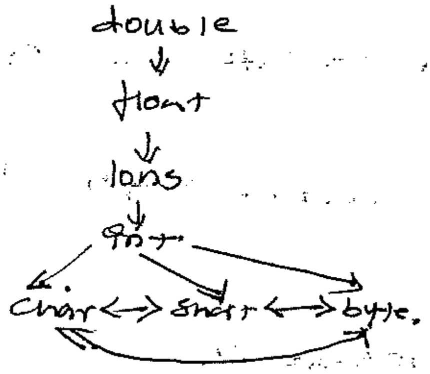
explicit / caste conversion - done by Programmar.

Widning - Intension is to Preserve info so convert from lower to higher.

Narrowing - may lose info so convert from higher to lower.



Widning



Narrowing

We can use two functions max(t1, t2) - returns max value of type.
widen (a, t, w) a-addr t-type w-widened type.

Unification :

Unification is the problem of determining whether two exprs s & t can be made identical by substituting expr for the variables in s & t. If s and t have constants but no variables, then s & t unify iff they are identical.

Eg:- consider two expressions.

$$((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2) \quad -①$$

$$((\alpha_3 \rightarrow \alpha_4) \times list(\alpha_3)) \rightarrow \alpha_5 \quad -②$$

the following substitution S is the most general unifier for these exprs

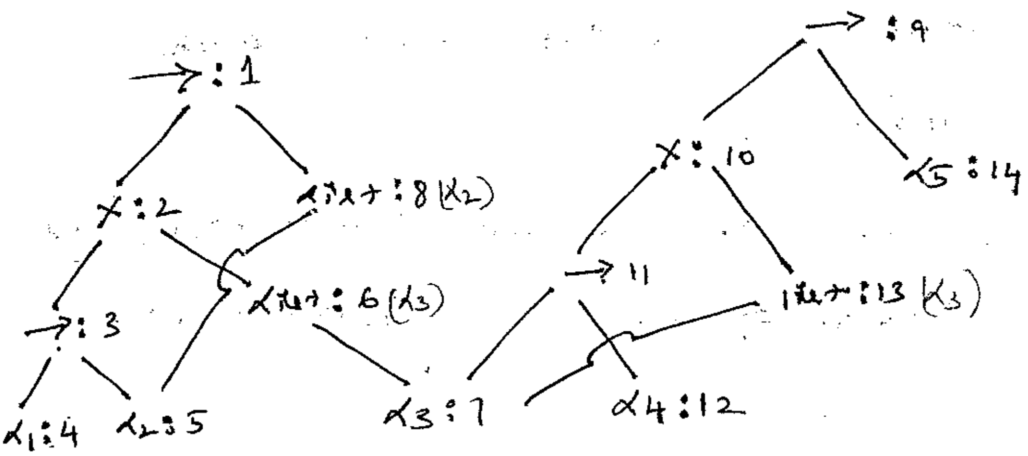
α	$S(\alpha)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$list(\alpha_2)$

wrt expr②

This substitution make the two type expr^n as,

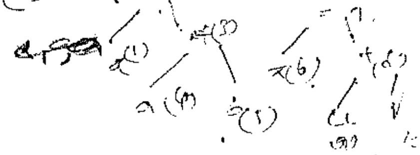
$$((\alpha_1 \to \alpha_2) \times list(\alpha_1)) \to list(\alpha_2) \quad — ③$$

Suppose that two Expr^n are rep^n by initial graph where each node be in its own equivalance class.



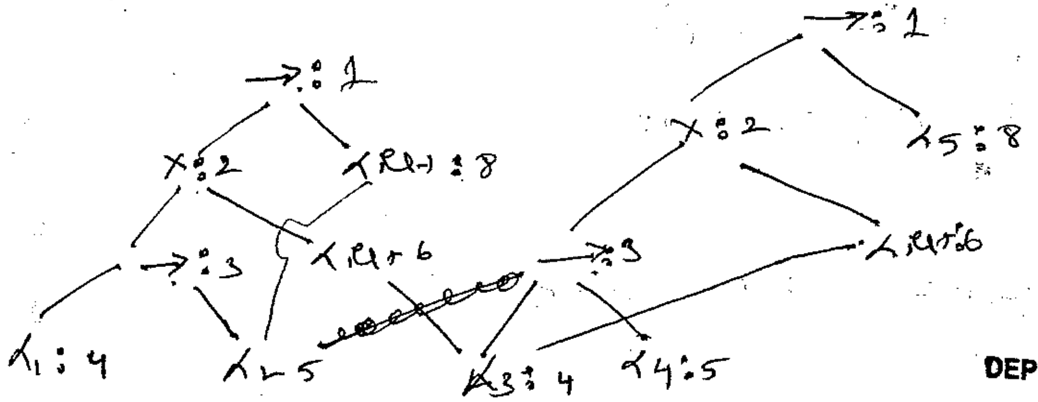
x = a + b
y = c + d

(1, 9)	(4, 7)
(2, 10)	(5, 4)
(8, 14)	x = a + b
(3, 11)	z = c + v
(6, 13) =	



Now if alg of unification be applied to compute (1, 9), it notices that nodes 1 & 9 both rep^n same operator. i.e 1 & 9 merges into same equivalance class then it calls unify (2 & 10) and unify (8, 14)

The equivalance class after unification are resulting graph.



GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Algorithms:

Input: a graph rep^{n}ing a type and a pair of nodes m & n to be unified.

Output: boolean value true if the expr^n rep^n by the nodes m & n unify else false.

ethod:

```
boolean unify (Node m, Node n)
{
    s = find (m);   t = find (n);
    if (s = t) return true;
    else if ( nodes s and t repnte same basic type)
            return true;
    else if (s is an op-node with children s1, s2 and
            t is an op node with children t1, t2)
        {
            union (s, t);
            return unify (s1, t1) and unify (s2, t2);
        }
    else if  s or t  repn  a variable {
        {
            union (s, t);
            return true;
        }
    else return false
}
```

find(n) : returns the representative node of equivalance class containing n

union (m, n): merges the equivalance class containing node m & n

2016-20 Batch, Dept. of EEE

SL.	USN	NAME OF THE CANDIDATE	QUOTA	I SGPA	II SGPA	III SGPA	IV (800)	V	VI	VII	VIII	Total	Percentag	Class	Only 1st yr per	Only 2nd yr per	Only 3rd yr per	REMARKS
1	1BY16EE001	AAFAQ ALTAF WANI	Aicte-JKSS	6.08	4.25	422	452								5.17	54.63		
2	1BY16EE002	ABHIJIT KUMAR	Comed-K	8.67	9.08	671	689								8.88	85.00		
3	1BY16EE003	ABHIJITH BISWAS	Comed-K	6.67	6.42										6.55			Left
4	1BY16EE004	ADITI D A	CET	7.5	8.58	584	644								8.04	76.75		
5	1BY16EE005	AKANKSHA P		1.75	1.68										1.72			Lost Eligibility
6	1BY16EE006	ANAMIKA BASU		1.25	3.1	257	299								2.18	34.75		Lost Eligibility
7	1BY16EE007	ANIRUDH ARAVIND	CET	7.5	8.25	529	572								7.88	68.81		
8	1BY16EE008	ANIRUDHA K S	Comed-K	5.17	6	461	527								5.59	61.75		
9	1BY16EE009	AWATI SPANDANA	CET	6.42	8.08	522	552								7.25	67.13		
10	1BY16EE010	B ASHWINI	CET	5.58	7.14	533	588								6.36	70.06		
11	1BY16EE011	B MANJUNATH	CET	8.75	8.75	651	673								8.75	82.75		
12	1BY16EE012	BALAJI SUBRAMANYAM M K	CET	8.17	7.75	554	567								7.96	70.06		
13	1BY16EE013	BHARGAVREDDY B	CET	7.67	7.25	500	530								7.46	64.38		
14	1BY16EE014	BHASKARA RAJU A V		5.33	4.07	408	292								4.70	43.75		Lost Eligibility
15	1BY16EE015	CHAITHRA B C	CET	6.92	7	489	554								6.96	65.19		
16	1BY16EE016	DEEPA M S	CET	8.33	8.58	595	645								8.46	77.50		
17	1BY16EE017	ENTHA ROHITH	Comed-K	8.25	8.17	601	645								8.21	77.88		
18	1BY16EE018	G SANKESH BOTHRA																COB
19	1BY16EE019	GAUTAMA BHARADWAJ	CET	8.42	8.33	593	626								8.38	76.19		
20	1BY16EE020	GOVARDHAN		6.33	6.25	462	553								6.29	63.44		
21	1BY16EE021	H VISHAL		3.75	3.2	348	357								3.48	44.06		Lost Eligibility
22	1BY16EE022	HARIKRISHNAN U	Comed-K	8.58	8.58	626	660								8.58	80.38		
23	1BY16EE023	HARSHIT VAISH	CET	8.25	8.25	606	644								8.25	78.13		
24	1BY16EE024	KACHARLA BALASAI NIKHIL		7.17	6.58	502	506								6.88	63.00		
25	1BY16EE025	KALPIT CHATURVEDI	Comed-K	5.08	4.17	397	427								4.63	51.50		Left
26	1BY16EE026	KAUSHIK GOWDA H N	Comed-K	7.42	8.42	555	438								7.92	62.06		
27	1BY16EE027	KAVYA P	CET	8	8.67	559	605								8.34	72.75		
28	1BY16EE028	KEERTHI GURURAJ G S	CET-SNQ	7.33	8.58	551	580								7.96	70.69		
29	1BY16EE029	KISHLAY SINHA		3.25	2.38	249	234								2.82	30.19		Lost Eligibility
30	1BY16EE030	KUSHAGRA DHAWAN	CET	7.58	8.42	554	573								8.00	70.44		
31	1BY16EE031	MADHU CHANDRA M		6.33	4.08	355	394								5.21	46.81		
32	1BY16EE032	MADHUSUDHAN H K	CET	7.17	7.33	485	541								7.25	64.13		
33	1BY16EE033	MALLARI SURESH		1.17	3.5	333	420								2.34	47.06		Lost Eligibility
34	1BY16EE034	MOHAMMAD ADIL ANSARI		6.5	7.08	566	575								6.79	71.31		
35	1BY16EE035	NAVEDYA OJHA		7.67	8.33	569	657								8.00	76.63		
36	1BY16EE036	NIKITA CHAUHAN		7.92	7.42	473	578								7.67	65.69		
37	1BY16EE037	NISHA CHAURASIA		6.17	5.25	387	468								5.71	53.44		Lost Eligibility
38	1BY16EE039	PULKIT KUMAR DAGUR		0	0													Lost Eligibility
39	1BY16EE040	RAKESH M		0.5	0													Lost Eligibility
40	1BY16EE041	ROHAN CHINNI C L		7.33	8	582	645								7.67	76.69		
41	1BY16EE042	RUCHITHA D		5.33	7	443	531								6.17	60.88		
42	1BY16EE043	RUPANSHA KHARE		4.33	4.29	367	417								4.31	49.00		
43	1BY16EE044	SADIQ HASAN ABBASI		7.42	7.67	566	575								7.55	71.31		
44	1BY16EE045	SARTHAK GUPTA		6.58	7.67	516	564								7.13	67.50		
45	1BY16EE046	SAUMYA MISHRA		6.33	6.92	454	526								6.63	61.25		
46	1BY16EE047	SAYALEE SANJAY KAHANDAL		7.5	9.08	549	611								8.29	72.50		
47	1BY16EE048	SESHA GOPAL S		7.08	6.58	536	603								6.83	71.19		

48	1BY16EE049	SHIVAPRASAD K R		8.08	7.67	559	611								7.88	73.13		
49	1BY16EE050	SINCHANA GOWDA		6	6.5	478	495								6.25	60.81		
50	1BY16EE051	SINCHANA S		8.83	9.17	639	630								9.00	79.31		
51	1BY16EE052	SOMPALLI SAKETH		8.17	8.67	645	694								8.42	83.69		
52	1BY16EE053	SRIJAN MRISHRA		7.25	7.58	604	608								7.42	75.75		
53	1BY16EE054	SUHANA KHANUM N		7.83	8	529	597								7.92	70.38		
54	1BY16EE055	SUJATA		4.08	4.87	447	490								4.48	58.56		
55	1BY16EE056	SWATI		7.83	8.25	546	644								8.04	74.38		
56	1BY16EE057	SYED IMTIAZ ALI		3.42	2.19										2.81			Lost Eligibility
57	1BY16EE058	VAISHNOVI H		4.67	6.93	399	534								5.80	58.31		
58	1BY16EE059	VENU BHARGAV VALLURI		7.25	7.33	484	572								7.29	66.00		
59	1BY16EE060	VINOD		5.92	6.5	527	613								6.21	71.25		
60	1BY16EE062	ABHISHEK KUMAR YADAV		5.5	5.57	481	533								5.54	63.38		
61	1BY16EE063	DEEPAK SINGH		7.17	7.17	437	448								7.17	55.31		
62	1BY16EE064	RUCHITHA YADAV		2.83	3.39	310	267								3.11	36.06		Lost Eligibility n Left
63	1BY17EE400	AKILESH S K	Dip			385	513									56.13		
64	1BY17EE401	FIRAS AHMED	Dip			483	551									64.63		
65	1BY17EE402	GIRISH KUMAR L N	Dip			447	498									59.06		
66	1BY17EE403	GOKUL MANIKANTA T	Dip			441	491									58.25		
67	1BY17EE404	GURUPRASAD AGALI	Dip			368	489									53.56		
68	1BY17EE405	NITHIN P	Dip			326										20.38		Left
69	1BY17EE406	RAVI	Dip			370	482									53.25		
70	1BY17EE407	SHARATH S K	Dip			460	471									58.19		
71	1BY17EE408	SHIVA KUMAR K R	Dip			263	499									47.63		
72	1BY17EE409	SHIVAKUMAR SOMASAGAR	Dip			423	462									55.31		
73	1BY17EE410	SYED SHA SAYAAFF	Dip			313	221									33.38		Lost Eligibility
74	1BY17EE411	YAQOOB QASIM SOFI	Dip			485	505									61.88		
															6.58	62.74		

No. of Students	61	61	68	67				
ZERO BACKLOG	44	40	32	30				
WITH BACKLOG	60	56	68	58				

Control flow

The translation of stmts such as if-else-stmt and while-stmt is tied to the translation of boolean expr.

In prg lang, boolean exprn are used to

(i) Alter the flow of control - boolean expr are used as condnl expr in stmts that alter the flow of control. for eg: if(E) then S, the expr E must be true if stmt S is reached.

(ii) Compute logical values - a boolean expr can repn true or false values.

Boolean Expressions

Boolean exprn are composed of boolean operators. like (&&, ||, !) applied to elements that are boolean variables or rational expressions., rational exprns are of the form E_1 relop E_2. where E_1 & E_2 are arithmetic exprs.

Consider the boolean exprn gen by the grammar.

$B \rightarrow B || B \mid B \&\& B \mid !B \mid (B) \mid E \text{ rel } E \mid \text{true} \mid \text{false}$.

the rel op may be. <, <=, =, !=, >, >= repn by relop

In exprn $B_1 || B_2$ if either B_1 or B_2 is true entire exprn is true but in $B_1 \&\& B_2$ if either B_1 or B_2 is false entire exprn is false.

The semantic defn of the prg lang determines weather all parts of boolean expr must be evaluated.

Short Circuit Code

In short-circuit (jumping) code, the boolean operators &&, ||, ! translate into jumps, - the operators themselves donot appear in the code instead the value of a boolean expr is repn by a position in the code sequence.

Eg:-

if (x<100 || x>200 && x != y) x = 0;

is translated into

```
if (x < 100) goto L2
if false x>200 goto L1
if false x != y goto L1
L2 : x = 0           - true
L1 :                 - false
```

Flow Control Statements

Now consider the translation of boolean exprn into 3-addr code in the context of the gramar.

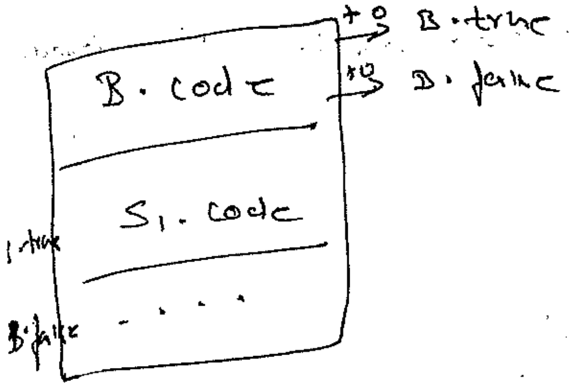
$S \rightarrow if\ (B)\ S_1$

$S \rightarrow if\ (B)\ S_1\ else\ S_2$

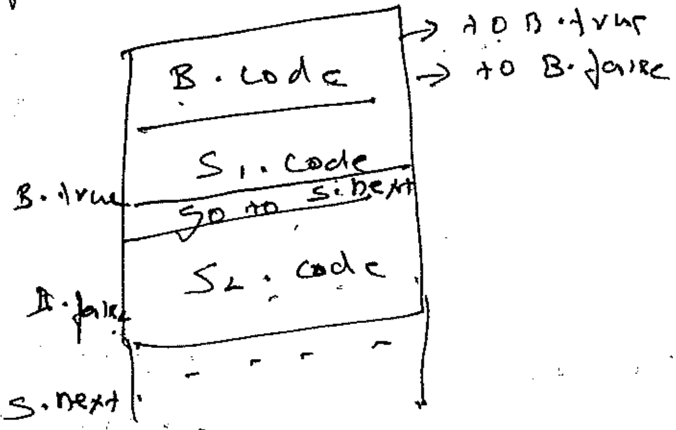
$S \rightarrow while\ (B)\ S_1$

In this gramar 'B' repn boolean exprn & 'S' repn a statement both B & S have the synthesized attrib code which gives the transln into 3-addr-code

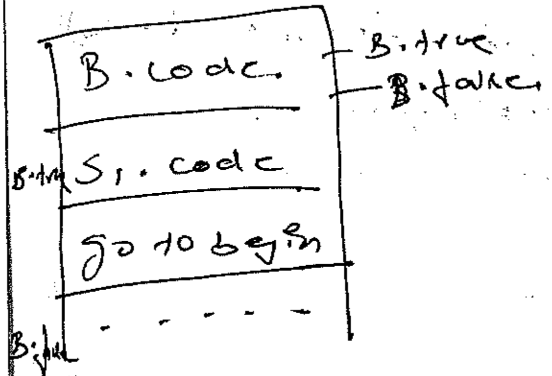
The tranln of if (B) S_1 Consists of B.code folled by S_1.code within B.code are Jumps based on the value of B. If B is true control flows to the first instrn of S_1.code & if B is false control flows to the instrn immediately followed S_1.code.



(a) if.



(b) if-else



(c) while.

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

The SDD for flow of control statements Produce 3-addr code for Bol Exp

Prodn	Semantic Rule
P → S	S.next = newlabel() P.code = S.code ∥ label(S.next)
S → assign	S.code = assign.code.
S → if (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code ∥ label(B.true) ∥ S_1.code

$S \rightarrow$ if (B) S_1 else S_2

B.true = newlabel()
B.false = new label()
S_1.next = S_2.next = S.next.
S.code = B.code || label(B.true) || S_1.code
|| gen('goto' S.next) || label(B.false) || S_2.cod

$S \rightarrow$ while (B) S_1

begin = newlabel()
B.true = newlabel()
B.false = S.next
S_1.next = begin
S.code = label(begin) || B.code || label(B.true)
|| S_1.code || gen('goto' begin)

$S \rightarrow S_1 S_2$

S_1.next = newlabel()
S_2.next = S.next
S.code = S_1.code || label(S_1.next) || S_2.code

---

newlabel() creates a newlabel each time it is called.

label(L) attaches label L to next 3-addr item to be generated.

Control flow translation of boolean Expression

The semantic rules for bool expn is as shown.

Prodn	Semantic Rule
B → B_1 $\Vert$ B_2.	B_1.true = B.true B_1.false = newlabel() B_2.true = B.true B_2.false = B.false. B.code = B_1.code $\Vert$ label(B_1.false) $\Vert$ B_2.code
B → B_1 && B_2.	B_1.true = new label() B_1.false = B.false B_2.true = B.true B_2.false = B.false B.code = B_1.code $\Vert$ label(B_1.true) $\Vert$ B_2.code
B → ! B_1	B_1.true = B.false B_1.false = B.true B.code = B_1.code
B → E_1 relop E_2	B. code = E_1.code $\Vert$ E_2.code $\Vert$ gen ('if' E_1.addr relop E_2.addr 'goto' B.true) $\Vert$ gen ('goto' B.false)
B → true	B.code = gen ('goto' B.true)
B → false	B.code = gen ('goto' B.false)

if (x<100 || x>200 && x!=y) x=0

for

```
      if x<100  goto L2
           goto L3
   L3:  if x>200 goto L4
           goto L1
   L4:  if x!=y  goto L2
           goto L1
   L2 :  x=0
   L1 :
```

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Boolean Values & Jumping Code.

The use of Boolean Expr^n is to alter the flow of control in stmt, A boolean Expr^n may also be evaluated for its value as in assignment stmts such as X = true or x = a<b

A clear way of handling both roles of bool Expr^n is first build a syntax tree for Expr^n using either of following approach

(i) Use two passes: Construct a complete syntax tree for the i/p & then walk the tree in depth first order, computing the translations specified by semantic rules

(ii) Use one pass for stmts: but two passes for Expr^n with this approach, we would translate E in while (E) S1 before S1 is examined.

Back Patching :

The key problem when generating code for bool exprn & flow of control stmts is that of matching a jump instrn with the target of jump. So the approach to solve is back patching, in which lists of jumps are passed as synthesized attributes. Specifically when a jump is generated, the target of jump is temporarily left unspecified. Each such jump is put on a list of jumps & all jumps on a list have the same target label. The labels are filled when proper label can be determined

One Pass Code generation using Back Patching

Back Patching can be used to generate code for bool exprn & flow of contl stmts in one pass.

The ~~[illegible]~~ synthesized attribs truelist & falselist of NT 'B' are used to manage labels in jumping code for boolean exprn

B.truelist - list of ~~labels~~ jumps to reach if B is true

B.falselist - list of ~~labels~~ jumps to reach if B is false

S.nextlist - list of jumps to instrn immediately follows S

Instrns are generated into an array & labels will be indices into the array 3 fns are used to manipulate list of jumps.

(i) makelist(i) :- Creates the list containing only i, an index into the array of instrns, it returns ptr to newly created list

(ii) merge(P₁, P₂) : Concatenates the lists pointed to by P₁ & P₂ & return a ptr to concatenated list

(iii) backpatch(P, i): Inserts i as the target label for each of the instrs on list pointed by P.

Back Patching for Boolean Exprn

The following are the translation scheme suitable for generating code for bool exprn during BU parsing

M - marker NT :- calls semantic action to pick up at appropriate times & the index of next inst to be generated.

B - boolean   E - Exprn

$B \to B_1 \,||\, M B_2 \mid B_1 \,\&\&\, M B_2 \mid ! B_1 \mid (B) \mid E_1 \text{ rel } E_2 \mid \text{true} \mid \text{false}$

$M \to \epsilon$

The translation scheme:

$B \to B_1 \,||\, M B_2$

if B1 is false

{ backpatch (B1. falselist, M. instr);

B1 true

B. truelist = merge (B1.truelist, B2.truelist);

B. falselist = B2.falselist; }

$B \to B_1 \,\&\&\, M B_2$

B1 true

{ backpatch (B1.truelist, M. instr);

B. truelist = B2 . truelist;

B. falselist = merge (B1.falselist, B2.falselist);

B1 false

$B \to ! B_1$

{ B. truelist = B1 . falselist;

B. falselist = B1 .truelist }

$B \to (B_1)$

{ B .truelist = B1 .truelist;

B . falselist = B1 . falselist; }

$B \rightarrow E_1$ rel E_2 { B.truelist = makelist (nextinstr);
B.falselist = makelist (nextinstr+1);
emit ('if' E_1.addr rel.op E_2.addr 'goto _');
emit ('goto _'); }

B → true { B.truelist = makelist (nextinstr);
emit ('goto _'); }

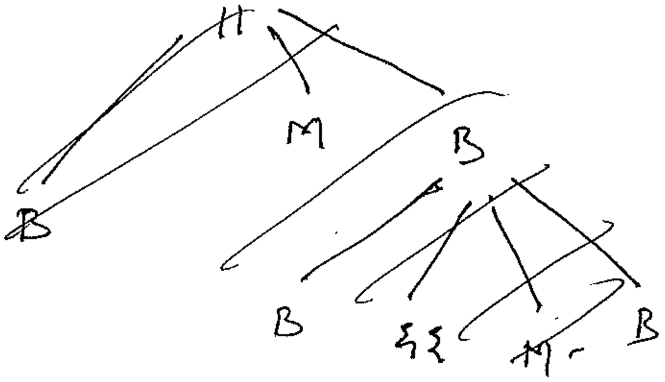
B → false { B.falselist = makelist (nextinstr);
emit ('goto _'); }

M → ε { M.instr = nextinstr }

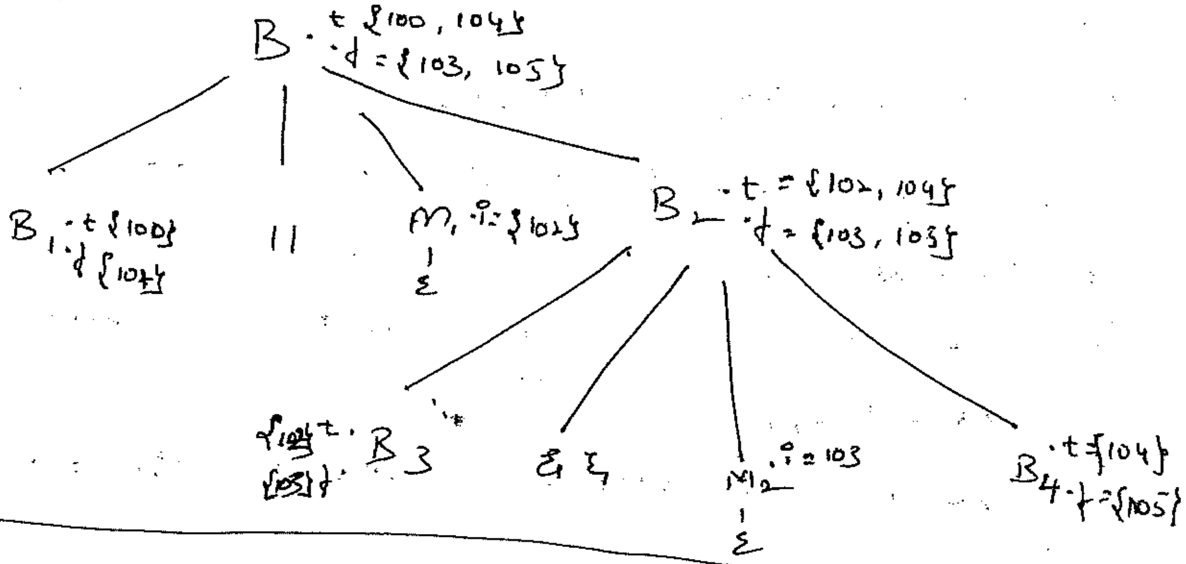
Back Patching:

in if (B) S Jump to S if B is true else Jump to next instr of S so B must be translated first then label has to be found where to Jump if true & if false if these labels are forward referenced then we need two passes to compute & assign labels to address? to solve this problem Back Patching is used wherein Jumps are left unspecified temporarily — see back

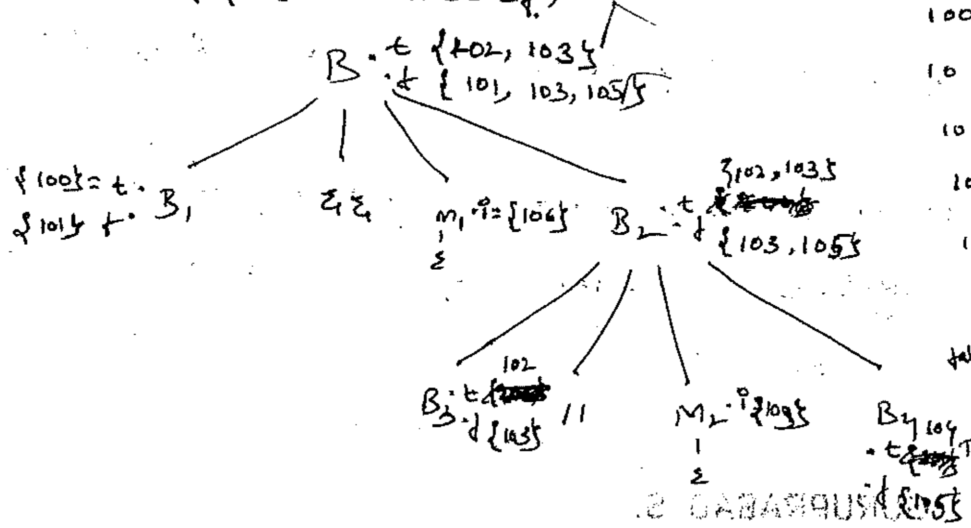
Eg:- $x < 100 \,||\, x > 200 \;\&\&\; x \,!= y$



```
          100:  if x < 100  go to _{200}   false
          101:  goto _  {102}
          102:  if x > 200  go to _{104}
          103:  go to _{106}
          104:  if x != y go to {200}
          105:  go to 106
false 106:
          ⋮
True 200:
```



Eg2: $a == b \;\&\&\; (c == d \,||\, e == f)$



```
100 :  if (a == b) goto 102
101 :  goto 106
102 :  if (c == d) goto 200
103 :  goto 104
104 :  if (e == f) goto 200
105 :  goto 106
fal 106
     ⋮
     200
```

③ $(a = b \,||\, c == d) \,||\, e == f$

④ $(a == b \;\&\&\; c == d) \;\&\&\; e == f$

Switch Statements

The switch or case statement is available in a variety of languages as shown below

```
Switch (E)
{
 case V1 : S1
  case V2 : S2
     :
   case Vn-1 : Sn-1
    default : Sn
}
```

Translation of switch Statement

The intend translation of switch is code to :

① Evaluate the Expr E
② Find the value V_J in the list of case that is the the value of Expr
③ Execute the Stmt S_J associated with value found.

SDT of switch Stmt

The switch Stmt is translated to the intermediate code as shown.

The tests appear at the End so that the simple code generator can recognize the multi way branch and can generate Efficient code for it.

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

```
code to Evaluate E into t
  go to test
L1 :  code for S1
      goto next
L2 :  code for S2.
      go to next
         :
Ln-1: code for Sn-1
      goto next
Ln :  code for Sn
      goto next
test: if t = V1 goto L1
      if t = V2 goto L2
      . . . . .
      if t = Vn-1 goto Ln-1
      goto Ln
next :
```

Intermediate Code for Procedures

In 3-addr code a function call is translated into the Evaluation of parameters in preparation for a call, followed by Call itself. and the parameters are passed by value.

Eg:- If a is array of int & f is a fn from int to int then the call

$$n = f(a[i])$$

translate into following 3-addr code:

```
t1 = i * 4          n = t3
t2 = a[t1]
param t2
t3 = call f
```

GURUPRASAD. B

Function definitions & calls can be translated using concep

① Function types: the type of a fn must encode the return t

& types of formal parameters. , let void be a special type.

that repn no parameters or no return type.

② Symbol Table: let 's' be the top symbol table when fn defn is reached the fn name is inserted into 's' for use in the rest of prg.

③ Type checking: within expn, a fn is treated like any othe operator. for eg f is a fn with parameter of type real then the integer 2 is coerced to a real in the call f(2)

④ function calls: when generating 3-addr code for fn call id(E_1 E_2 ... E) it is sufficient to generate 3-addr instn for evaluating or reducing parameter E. to address, followed by param instn for each parameter.

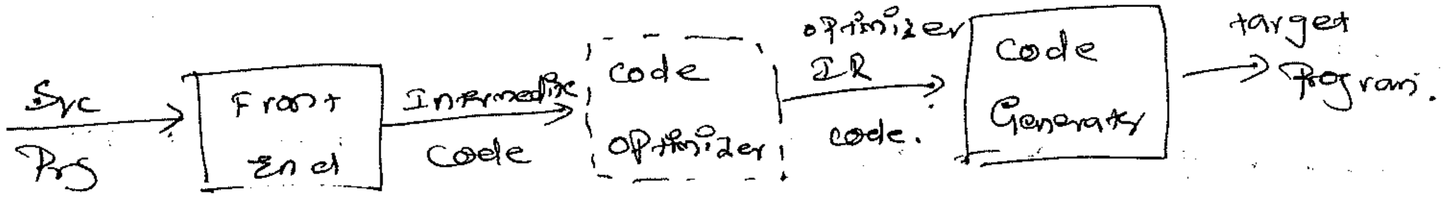
X

End of Part ⓑ unit ⑥

Part -B   Unit - 8

Code Generation

A Code Generator is the final phase of Compiler model. it takes IR code. Produced by front end & relevant ST info to produce semantically equivallent target Program.



A code generator has three Primary tasks: instruction selection Register allocation and instruction ordering.

Issues in the Design of a Code Generator

The most important criterion for a code generator is that it Produce Correct code.

(i) Input to CG
(ii) o/p from CG target code
(iii) Insn seln
(iv) Reg allocn asn & assgn
(v) Evaln order.

i) Input to the code Generator.

The i/p to CG ~~is~~ is. different IR code of Src Prg, Produced by front end. There are many choices for IR include 3-addr repn such as Quadruples, Triples, Indirect Triples. virtual m/c repn such as byte codes, stack-m/c code., graphical repn such as Syntax tree, DAG.

We assume that the front End has scanned parsed & translated the Src Prg into a relatively low level IR. We also assume that Syntactic, semantic errors have been detected alredy.

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

The Target Program

The instn-set architecture of the target m/c has a significant impact on the difficulty of constructing a good code generator that produces high quality m/c code.

The most common target m/c architectures are RISC, CISC & stack based.

RISC m/c has many registers, 3-addr instns, simple addressing modes & simple instn set.

CISC m/c has few regs, 2-addr instns, variety of addressing modes & complex instn set.

- Stack based m/c operations are done by pushing operands to the stack & perform operations. This is Java Virtual machine (JVM).

e object code produced can be either (i) absolute) relative

i) Producing absolute m/c lang prg as o/p is advantageous e it can be placed in a fixed location in memory & can be mmediately executed quickly.

ii) Producing a reallocatable m/c lang prg as o/p allows subprograms to be compiled separately.

(ii) Instruction Selection

The Code Generator must map the IR prog into a code sequence that can be executed by the target m/c. The complexity of performing this mapping is determined by factors such as

- The level of IR
- The nature of instn set architecture.
- The desired quality of generated code.

The nature of the instn set of target m/c has strong impact on instn selection. Eg:- uniformity & completeness of instn set are important factors.

Instn speed & m/c idioms are another factors.

Eg:- every 3-addr code of the form x = y+z are translated to code sequence like

```
LD   R0, y         ; R0 = y
ADD  R0, R0, z.    ; R0 = R0+z
ST   x, R0           x = R0
```

this strategy produces redundant LD & store. Eg.

```
a = b+c
d = a+e

LD R0, b
ADD R0, R0, c
ST a, R0
LD R0, a
ADD R0, R0, e
ST d, R0
```

Eg:- a = a+1 can be written as

```
LD R0, a.
ADD R0, R0, 1
ST a, R0
```

this can be impln using single instn INC

Register Allocation

A key problem in C.G. is deciding what value to hold in
hat registers. Regs are fastest computational unit on target m/c.
+ ~~[illegible]~~ only limited regs are available.
alue not held in regs. need to reside in memory.
e hence, with reg opns are shorter & faster, so effecient
utilization of regs are important.

he use of regs is divided into 2 problems

' Registers allocation : select the set of variable that will reside in registers at each point in program.

' Register assignment : pick the specific register that a variable will reside

:- consider 3-addr code eg.

```
t = a+b
t = t*c.
t = t/d
```

ⓐ

```
t = a+b
t = t+c
t = t/d
```

ⓑ

only diff. b/w ⓐ & ⓑ is operator in second stmt. the shortest assembly code is,

```
L  R1, a.
A  R1, b
M  R1, c
D  R1, d
ST R1, t
```

```
L    R0, a
A    R0, b
A    R0, c.
SRDA R0, 32
D    R1, d
ST   R1, t
```

Notes: SRDA means shift right double arithmetic, SRDA R_0, 32 shifts dividend into R_1 & clears R_0

(v) Evaluation order -

The order in which computations are performed can affect the efficiency of the target code. Some computation order require fewer registers to hold intermediate results than others.

The target Language

Familiarity with target m/c and its instruction set is a prerequisite for designing a good code generator.

Our target computer models a 3 addr m/c with load and store opns, computation opns, jump opns & condnl jmp opns.

The foll are the kinds of instrns available:

(i) load opns - LD dst, addr LD r, x addr → dst loc to reg x → r

(ii) store opns - ST x, r. r → x reg to loc

(iii) computation opns. OP dst, sr_1, sr_2.

OP - ADD, SUB

(iv) un condnl jmp - BR L

(v) condnl jmp - Bcond r, L eg BLTZ r L
rg Jmp if r val is less than zero

eg:- $\boxed{X = y - 2}$

```
LD R1, Y
LD R2, 2.
SUB R1, R1, R2
ST X, R1
```

$\boxed{b = a[i]}$ Each i is 8 bytes.

```
LD R1, i
MUL R1, R1, 8
LD R2, a(R1)
ST b, R2
```

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

a[j] = c.

```
LD R1, c
LD R2, j
MUL R2, R2, 8
ST a(R2), R1      // contents (a + contents (R2)) = R1
```

x = *p

```
LD R1, p
LD R2, 0(R1)
ST x, R2
```

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

*p = y

```
LD R1, p
LD R2, y
ST 0(R1), R2
```

If x < y goto L

```
LD R1, x
LD R2, y
SUB R1, R1, R2
BLTZ R1, M   // if R1 < 0 goto M)
```

Program & Instn Cost

Some common cost measure are the length of compilation time, size, running time & power consumption of target prg

for simplicity we take cost of instn to be one plus the cost associated with the addr mode of opnds.

Eg:-

① LD R0, R1 – cost 1 no mem opn

② LD R0, M cost 2 one mem opn

③ LD R1, *100 (R2) Loads R1 by (contents(contents(100 + contents (R2))))

cost is 3 as 100 is stored in the word following them

Eg:- ① x = b * c
y = a + x

```
LD R0, b
LD R1, c
MUL R0, R0, R1
ST x, R0
LD R1, a
LD R2, x
ADD R1, R1, R2
ST y, R1
```

② x = a[i]
y = b[j]
a[i] = y
b[j] = x

```
LD R0, i
MUL R0, R0, 8      } x = a[i]
ST x, a(R0)

LD R1, j
MUL R1, R1, 8      } y = b[j]
ST y, b(R1)

LD R0, y
LD R1, i           } a[i] = y
MUL R1, R1, 8
ST a(R1), R0

LD R1, x
LD R2, j           } b[j] = x
MUL R2, R2, 8
ST b(R2), R1
```

3) y = * q
q = q + 4
* p = y
p = p + 4

```
LD R0, q
                   } y = *q
ST y, 0(R0)

LD R0, q
ADD R0, R0, 4      } q = q+4
ST q, R0

LD R1, p
LD R2, y           } *p = y

ST 0(R1), R2

LD R0, y
ADD R0, R0, 4      } p = p+4
ST p, R0
```

```
if x < y goto L1          LD R0, x
 z = 0                    LD R1, y
goto L2                   SUB R0, R0, R1
: z = 1                   BLTZ R0, L1
                          ST z, 0
                          BR L2
                       L1 ST z, 1
```

```
if i > n goto L2
LD R0, i
LD R1, n
SUB R1, R1, R0
BLTZ R1, L2
```

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Addressing in the target Code

Here we show how names in the IR can be converted to address in target code.

The executing Prg runs in its own logical addr space that is partitioned into code area and data areas.

) Statically determined area: code that holds the Executable target code.

) Statically determined data area: Data area for holding global constants & other data generated by compiler.

A Dynamically managed area: Heap for holding data objects that are ~~created~~ allocated & freed during Prg exn

A Dynamically managed area: Stack for holding activation records as they are created & destroyed during procedure call & return.

Static Allocation

Code gen for simplified Proc call & return focus on the following 3-addr stmt.

(i) call calee (ii) return (iii) halt (iv) action.

Eg:-

code for C
action 1
call P
action 2
halt

code for P
action 3
return

⇒

100: Action 1
120: ST 364, #140 (store ret addr i.e 140 in locn 364)
132: BR 200 (addr of P)
140: ACTION 2
160: HALT
⋮
200: Action 3
220: BR *364

Stack Allocation

Static allocation can become stack allocation by using relative addresses for storage in activation records. ~~However, in stack allocation~~

However in stack alloc^n the posn of an actv^n record for a procedure is not known until run time.

When a procedure is called the calling procedure increments the stack pointer & transfers the control to the called procedure. After control return to the caller, we decrement stack pointer, there by deallocating the activation record of called procedure.

Eg:- LD SP, # stack start

code for the first procedure

HALT

A procedure call sequence increment stack pointer, saves the return addr & transfer control to called procedure

```
ADD  SP, SP, # caller.record size.

ST   * SP, # here + 16          save return addr

BR   callee.code area.
```

the called procedure transfers control to ret addr using

```
Br  * 0 (SP)
```

and decrement SP by

```
SUB  SP, SP, # caller.record size.
```

Eg:-

```
  action1       codeform
  Call P
  action2
  halt
        code for P
  action3
  return
```

⇒

```
100 : LD SP, #600      initialize stack
108 : Action1
128 : ADD SP, SP, #msize.
136 : ST  *SP, #152
144 : BR 200
152 : SUB SP, SP, #msize
160 : Action2
180 : HALT
      - - - - -
200 : Action3
220 : Br * 0 (SP)
```

Basic Blocks & Flow Graphs

Basic blocks are the graphical representation of IR code that is helpful for code generation. which are constructed as follows:

(i) Partition the IR code into basic blocks, which are sequence of consecutive 3-addr instn with the properties that

ⓐ The flow of control can only enter the B.B through the first instn in the block.

ⓑ Control will leave the block without halting or branching, Except in the last instruction in the block.

② The basic block become the nodes of a flow graph whose edges indicate which blocks can follow which other blocks.

Basic Blocks

The first Job is to Partition a sequence 3-addr stmt into basic blocks

Algorithm: Partitioning 3-addr stmt into B.B

Input: A sequence of 3-addr stmts

Output: A list of B.B for the sequence in which each stmt is assigned to exactly one B.B

Method:

Step ① - determine the set of Leaders, the first stmt of B.B with the following rules.

ⓐ The first stmt is a Leader

ⓑ The target of a Conditional / un-conditional goto is a leader.

ⓒ The stmt which immediatly follows a condn'l / uncondn'l goto is a Leader.

Step ② - For each Leader construct the B.B which consists of a Leader & all the stmts up to next Leader or End of the Program.

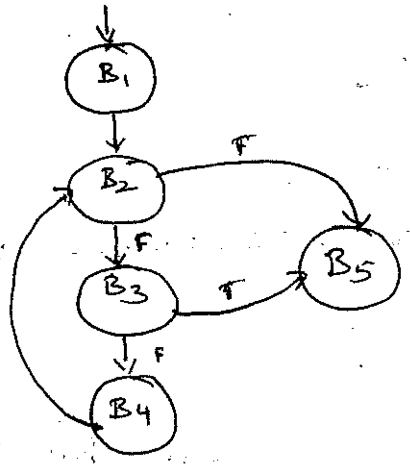
Note: Any stmt not placed in a B.B can never be executed & may be removed.

Eg:- 1) Generate the intermediate code for the following construct and also B.B.

```
i = 1
j = 10
while(i ≤ 5 && j ≥ 6)
    a = b+c+d
    i = i+1
    j = j-1
```

Scalar code:

```
1)  i = 1 ....          L     B1
2)  j = 1
3)  if i ≥ 5  goto 13   L     B2
4)  if j ≤ 6  goto 13   L     B3
5)  t1 = b+c            L
6)  t2 = t1+d
7)  a = t2
8)  t3 = i+1                  B4
9)  i = t3
10) t4 = j-1
11) j = t4
12) goto 3
13) End                 L     B5
```



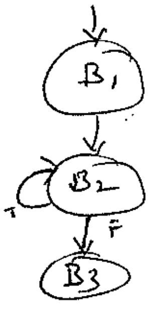
CFG

2)

```
Product = 0;
i = 1
do
Product = Product + a[i] * b[i]
  i = i+1
While (i <= 20);
```

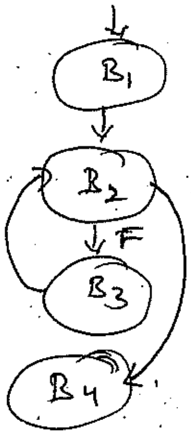
GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

```
1)  Prod = 0            L1    B1
2)  i = 1
3)  t1 = i*4            L2
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<=20) goto 3         B2
15) End                 L3    B3
```



```
Sum = 0
for i = 1 to n do
    sum = sum + a[i]
```

```
1) sum = 0   L
2) i = 1                         B1
3) if i > n  goto End   L        B2
4) t1 = 4*i
5) t2 = a[i]
6) t3 = sum + t2
7) sum = t3                      B3
8) ~~goto~~ t4 = i + 1
9) i = t4
10) goto 3
11) End   L                      B4
```



CFG

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Next use Information

Knowing when the value of a variable will be used next is essential for generating good code.

Algorithm to determine the liveness and next use info for each statement in B.B

Input: A basic Block of 3-addr stmt with S.T initially live for all non temp variables in B.

Output: At each stmt i: $x = y + z$ in B, we attach the liveness and next use info of x, y & z.

Method: Start at the last stmt in B and scan backwards to the beginning of B. At each stmt i: $x = y + z$ in B do the following

(i) attach the stmt i the info currently found in the SymbolTable regarding the next use & liveness of x, y, z.

(ii) In the Symbol Table, set x to "not live & "none"

(iii) In the Symbol Table, set y & z to "live" & next use of y & z to i.

Flow graphs

Once an intermediate code prg is partitioned into B.B. we repn the flow of control b/w them by a flowgraph or Control flow Graph.

The nodes of CFG are the B.B.

There is an edge from block B to block C iff it is possible for the first instrn in the block to immediately flow the last instrn in block.

There are 2 ways such edge could be justified

- * There is a condnl/uncondnl Jump from the End of B to the beginning of C
- * C immediately follows B in the original order of 3-addr instrn and B doesn't end in an uncondnl Jump

Often we add two nodes, called Entry & Exit that do not correspond to executable intermediate instrn

Eg:

```
for i = 1 to 5 do
for j = 1 to 5 do
  a[i, j] = 0.0;
for i = 1 to 5 do
  a[i, i] = 1.0
```

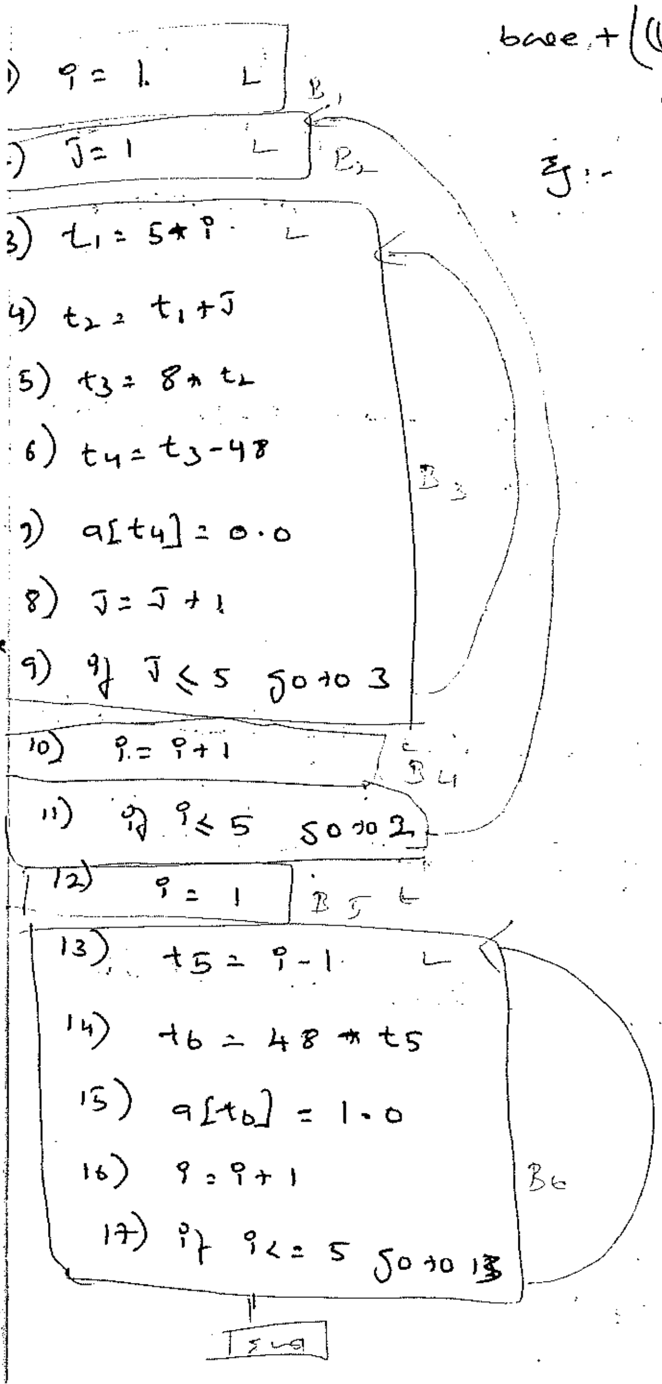
for j=1to5 do
a[i,j]=0.0

matrix is stored in row major order ∴ so a[i j] =
base + ((((n * i) + j) * 8) - 48)
no of rows

Eg :- a[2, 2] = 200 + (((5 * 2) + 2) * 8) - 48)
= 248

base + (i₁ * n₂ + i₂) * w

ue"
wep
regn
e for
s.
he
gnegn
not


1) i = 1
2) j = 1
3) t_1 = 5 * i
4) t_2 = t_1 + j
5) t_3 = 8 * t_2
6) t_4 = t_3 - 48
7) a[t_4] = 0.0
8) j = j + 1
9) if j <= 5 goto 3
10) i = i + 1
11) if i <= 5 goto 2
12) i = 1
13) t_5 = i - 1
14) t_6 = 48 * t_5
15) a[t_6] = 1.0
16) i = i + 1
17) if i <= 5 goto 13

LOOPS:

Prog lang constructs like while stmt do-while stmt & for stmt naturally give rise to loops many code transformations depend upon the idfn of loops in a flow graph.

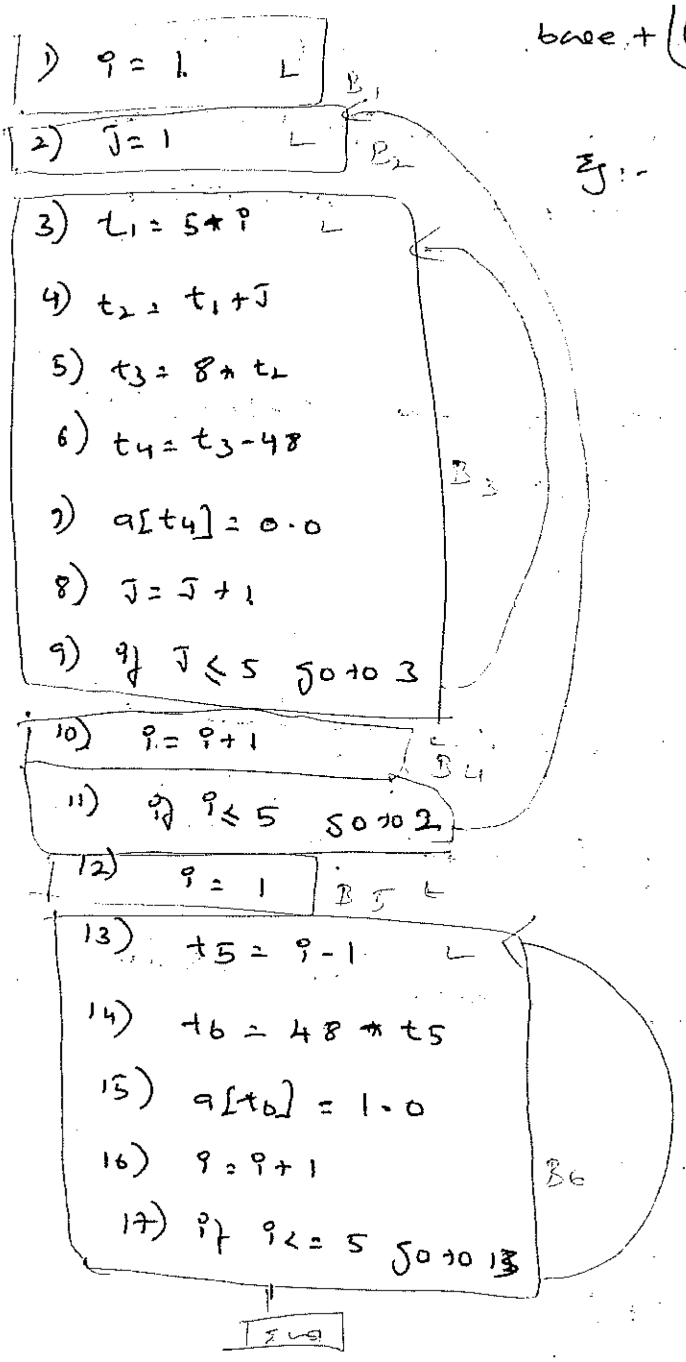
A set of nodes L in a flow graph is loop if -

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

matrix is stored in row major order. so $a[i\,j] =$

$base + (((n * i) + j) * 8) - 48)$
no of rows

Eg:- $a[2,2] = 200 + (((5 * 2) + 2) * 8) - 48)$
$= 248$



```
1) i = 1
2) j = 1
3) t1 = 5 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 48
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 5 goto 3
10) i = i + 1
11) if i <= 5 goto 2
12) i = 1
13) t5 = i - 1
14) t6 = 48 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 5 goto 13
```

$base + (i_1 * n_2 + i_2) * w$

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Loops:

Prog lang constructs like while stmt do-while stmt & for stmt naturally give rise to loops many code transformations depend upon the idfn of loops in a flow graph.

A set of nodes L in a flow graph is loop if -

(i) There is a node in L called the loop entry with the property that no other node in L has a predecessor outside L i.e every path from the entry of the entire flowgraph to any node in L goes to the loop entry.

(ii) Every node in L has a non empty path completely within L to the entry of L i.e self loop.

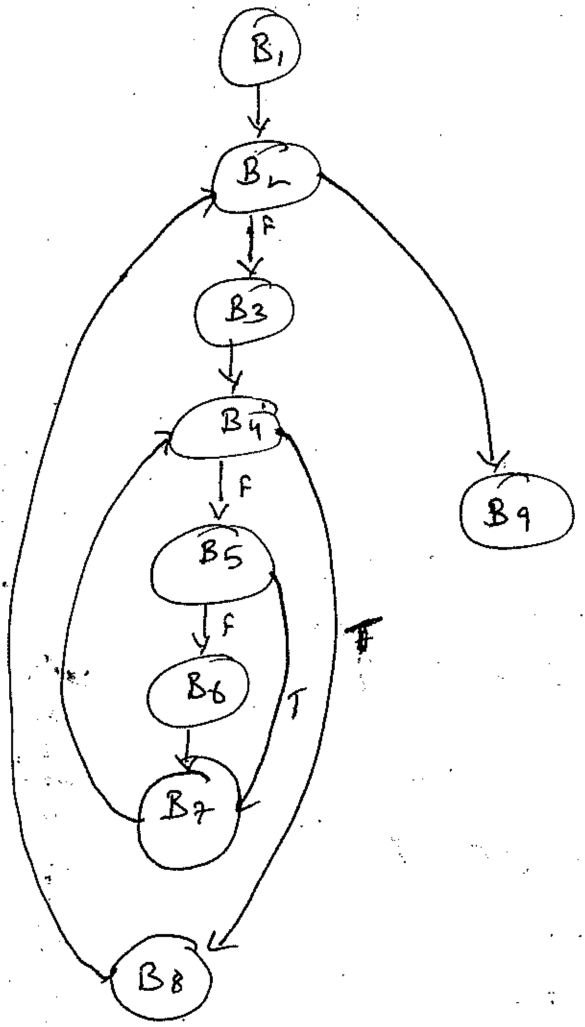
Matrix multiplication

Bubble Sort for asending order

```
for(i=1; i<n; i++)  // To keep track of pass no's
{
    for (j=0; j<n-i; j++)  // To compare the elements
    {
        if (a[j] > a[j+1])  // checking the condition
        {
            temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
        }
    }
}
```

```
1) Param a
2) Parm n
3) call Sort
L4) i = 1                          B1
 5) J = 0
L6) if (i > n) goto 24             B2
L7) t1 = n-i                       B3
L8) if (J > t1) goto 21            B4
L9) t2 = 4 * J                     B5
10) t3 = a[t2]
11) t4 = J+1
12) t5 = 4 * t4
13) t6 = a[t5]
14) if (t3 < t6) go to 18
L 15) temp = t3                    B6
16)   t3 = t6
17)   t6 = temp
L18)  t7 = J+1                     B7
19)   J = t7
20)   go to 8
L21)  t8 = i+1                     B8
22)   i = t8
23)   go to 6
L24)  return                       B9
```

CFG for sort



GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Matrix Multiplication

```
for i = 1 to m do          - r of A
{
    for J = 1 to q do      - col of B
    {
        Sum = 0;
                           - k r of B
        for k = 1 to n do
        {
            Sum = Sum + a[i][k] * b[k J]
        }
        C[i, J] = Sum
    }
}
```

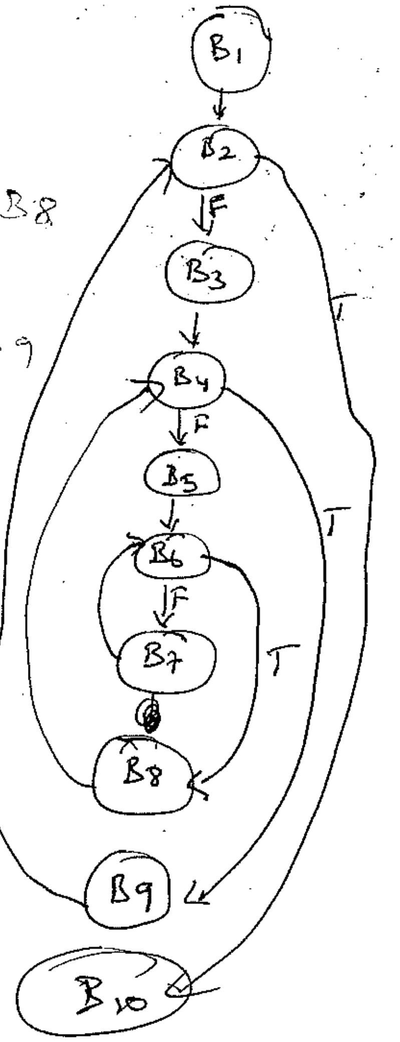
$m \times n$ $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ $n \times q$ $\begin{bmatrix} 2 & 1 \\ 4 & 3 \end{bmatrix}$

no of col in A = no of rows in B

$$\begin{bmatrix} 1\times2+2\times4 & 1\times1+2\times3 \\ 3\times2+4*4 & 3\times1+4\times3 \end{bmatrix}$$

$\underset{m\times n}{a[i\,J]} =$ base + $w*(i*\overset{\text{no of col}}{n}+J)$

```
L 1)  i = 1                      B1
L 2)  if (i > m) goto 32         B2
L 3)  J = 1                      B3
L 4)  if (J > q) goto 29         B4
L 5)  Sum = 0
  6)  k = 1                      B5
L 7)  if (k > n) goto 26         B6
L 8)  t1 = A * i
  9)  t2 = t1 + k
 10)  t3 = t2 * 4
 11)  t4 = a[t3]      // a[i k]
 12)  t5 = k * q
 13)  t6 = t5 + J
 14)  t7 = t6 * 4
 15)  t8 = b[t7]      // b[k J]
 16)  t9 = t4 * t8
 17)  t10 = Sum + t9
 18)  Sum = t10
 19)  t11 = q * i
 20)  t12 = t11 + J
 21)  t13 = c[t12]    // c[i J]
 22)  t13 = Sum
 23)  t14 = k + 1
 ~~24) goto~~
 24)  k = t14
 25)  goto 7                     B7
L 26) t15 = J + 1
  27) J = t15
  28) goto 4                     B8
L 29) t16 = i + 1
  30) i = t16
  31) goto 2                     B9
L 32) End.                       B10
```



$a\; i(m)\; k(n)$ $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ $b\; \overset{k(n)\; J(q)}{\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}}$

base + $(i * \underset{\text{col}}{n} + J) * w$

$c\; \overset{i(m)\; J(q)}{\begin{bmatrix} & \end{bmatrix}}$

i - index
l - limit

Optimization of Basic Blocks

A substantial improvement in the running time of code can be achieved by Local optzn with in each B.B and more can be achieved through Global optzn which looks for info flow b/w blocks.

The DAG repn of B.B

Many local optimization begin by transforming B.B into DAG. We can construct DAG for Expn in each B.B as follows:

(i) There is a node in DAG for each of initial value of variable in the blocks (leafnode)

(ii) There is node N (non leaf) associated with each stmt 'S' in the block. the children of N corresponds to statements that are last defn prior to S' of the opnds used by 'S'

(iii) Node N is labeled by operator ~~labeled by the~~ applied at S also the list of var for which it is last defn with in blocks

(iv) Certain nodes are designated as o/p nodes whose variables are live on Exit from blocks

The DAG representation of B.B lets us perform several code improving transformations on code repn by the blocks

(i) Elimination of local Common Sub Expression i.e the stmts that compute a value that has already been computed.

(ii) Eliminate dead Code i.e stmts that computes a value never needed.

(iii) ~~Reduce~~ Re-order stmts that do not depend on one another

(iv) Apply algebric laws to re-order opnds & simplify computation.

Finding local Common Sub-Expression

Common Sub-Expr^n is noticing a new node M is about to be added, whether there is an Existing node N with same Children in the same order and with same operator. If so N may be used in place of M.

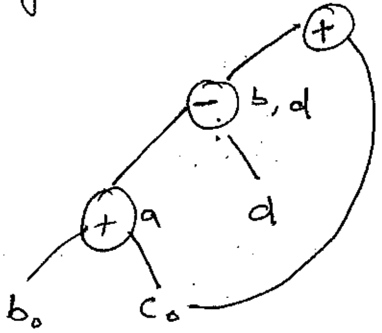
Eg:-

a = b+c

b = a[illegible]d.

c = b+c.

d = a-d.



since there are only 3 non leaf nodes we can rep^n 3-addr Stmt of B.B as,

a = b+c

b = a-d

c = d+c   or   c = b+c

Correct   ×

as b is not live on exit so we use node d rather than b

Eg 2:

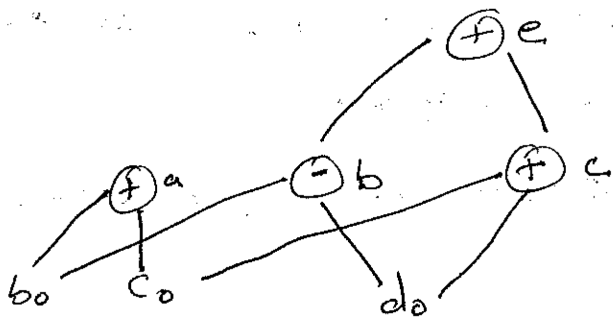
a = b+c — ①

b = b-d

c = c+d

e = b+c — ④

The value of b & c change before it is used in 4th Stmt so new node



Eg③:

DAG:

$t_1 = 4 * i$

$t_2 = a[t_1]$

$t_3 = 4 * i$

$t_4 = b[t_3]$

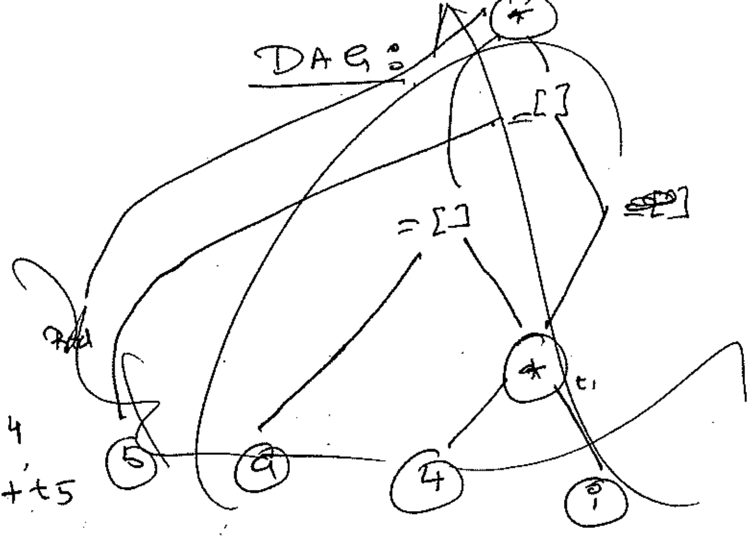
$t_5 = t_2 * t_4$

~~Prod~~ $t_6 = Prod + t_5$

$Prod = t_6$

$i = i + 1$

if $i \leq 20$ goto 3



⇓

$t_1 = 4 * i$

$t_2 = a[t_1]$

$t_4 = b[t_1]$

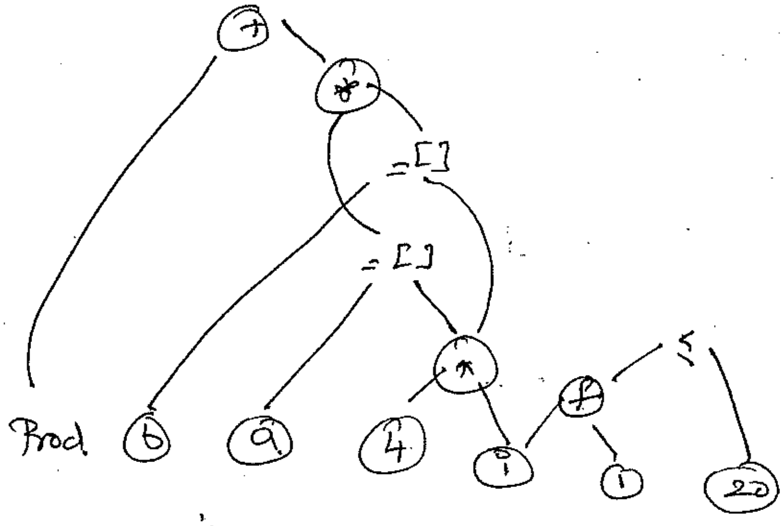
$t_5 = t_2 * t_4$

$t_6 = Prod + t_5$

$Prod = t_6$

$i = i + 1$

if $i \leq 20$ goto③



GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Dead Code Elimination

The optzn of DAG correspondg to dead Code Elmn can be impl as follows:

① Delete from DAG any root that has no live variables attached, a root node is one with no ancestors.

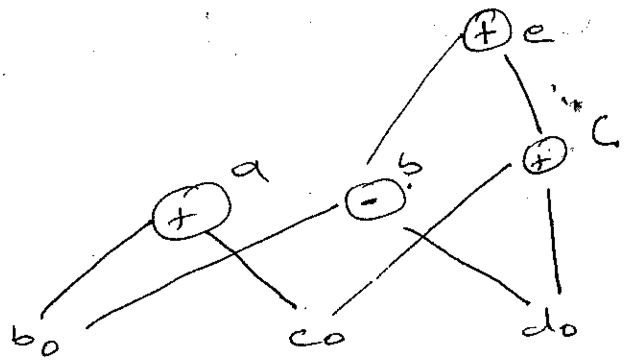
② Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

Eg:- $a = b + c$

$b = b - d$

$c = c + d$.

$e = b + c$.



If a & b are live but c and e are not, we can remove e and c becomes root as it is also not live we can Eliminate it.

to solve:

$d = b * c$

$e = a + b$

$b = b * c$

$a = e - d$.

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64
Ph: 9886760776

The use of Algebraic Identities

① Algebraic Identities repn another important class of optn on basic blocks we may apply algebraic identities such as.

$x + 0 = 0 + x = x$ $\quad$ $x - 0 = x$

$x * 1 = 1 * x = x.$ $\quad$ $x/1 = x.$

② Another classes of algebraic optn include. local reduction in strength i.e replacing a more expensive operator by cheaper one as.

Expensive	Cheaper
x^2	$x \times x$
$2 * x$	$x + x$
$x/2$	$x * 0.5$

③ Third class of related. optn is. Constant folding: here we Evaluate Constant Expn at Compile time. & replace the Expn by re. Values

Eg:- $2 * 3.14$ be replaced by value 6.28

④ DAG Constrn Can help to apply another algebric transformation such as. commutativity and associative.

commutative Eg:- $x * y = y * 2$ if suppose $y * 2$ has already present (node M) then can be used instead of creating node for $x * y$.

Eg:- $x > y$ Can be tested by $x - y$.

associative Eg:- $a = b + c$, $e = c + d + b$ $\Rightarrow$ $a = b + c$, $t = c + d$, $e = t + b$ $\Rightarrow$ $a = b + c.$, $e = \underset{b+c}{\underrightarrow{a}} + d$

Eg:-

```
Prod = 0
i = 0
T1 = 4*i
T2 = a[T1]
T4 = b[T1]
T5 = T2*T4
Prod = Prod + T5
i = i+1
i ≤ 20 goto 3
```

⟹

```
Prod = 0
T1 = 0
T1 = T1+4
T2 = a(T1)
T4 = b[T1]
T5 = T2*T4
Prod = Prod + T5
T1 = T1+1
if T1 ≤ 80 goto B2
```

$T_1 = 4*i$ increments i value by 4 Every time.
So we can replace by cheaper Exprn T_1+4
so $i \leq 20$ is written as $T_1 \leq 80$ ∵ (20×4)

to solve:
$x = a+b+c+d+e+f$
$y = a+c+e$;

Representation of Array References

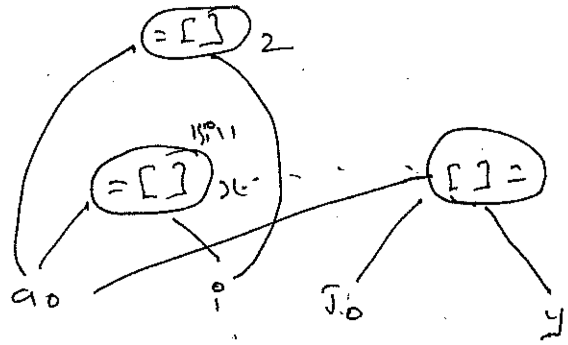
The proper way to repn array access in DAG is as follows.

(i) An assignment of form an array $x = a[i]$, is repn by creating a node with operator $=[\,]$ & two children representing the initial value of array a_0 & index i. Variable x becomes label of this new node.

(ii) An assignment of form $a[i] = y$ is repn by new node $[\,]=$ and the children repn a_0 & i & y there is no label. this node will ($[\,]=$) kill all nodes currently Constructed

Eg :-

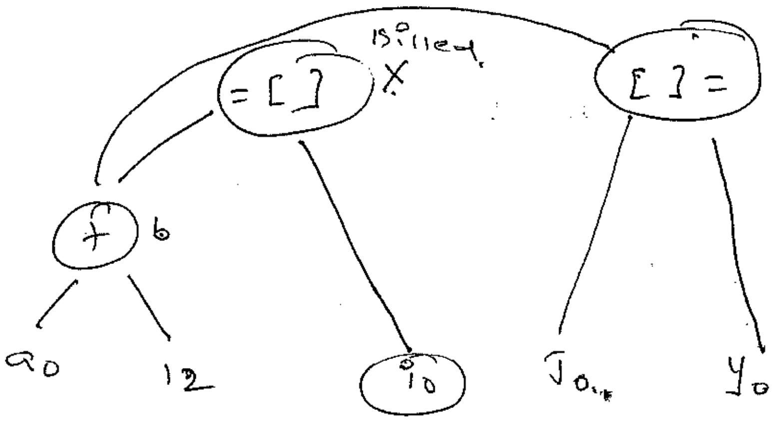
```
x = a[i]
a[j] = y
z = a[i]
```



Node x be killed by
a[j] = y

Eg 2: $b = 12 + a$
$x = b[i]$
$b[j] = y$

to solve:
$a[i] = b$
$*p = c.$
$d = a[j]$
$e = *p$
$*p = a[i]$



GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

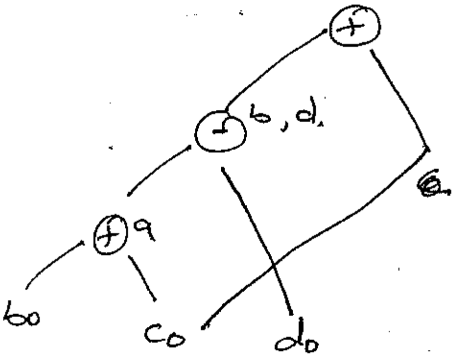
Pointer assignment & Procedure Call

When we assign indirectly through a ptr as $x = *p$ or $*q = y$
we don't know what p or q point to - so
the operator =* must take all nodes that are currently
associated with identifiers as arguments.
The =* kills all other nodes so far constructed in DAG.

Re-assembling B.B from DAG

After we perform whatever optimization possible while constructing DAG or by manipulating DAG.
we must reconstruct the 3-addr code for B.B from which we built DAG

If a node has more than one live Var attached. then we have to introduce Copy stmts to give the correct value to each of those Variables.



Eg:- $a = b + c$

$b = a - d.$

$c = b + c.$

$d = a - d.$

So if b is not live.

$a = b + c$

$d = a - d.$ as $b = d.$

$c = d + c$

if suppose b is also live

then $a = b + c$

$d = a - b$

$b = d.$ ⟹ copy stmt.

$c = d + c.$

Rules to use while reconstructing B.B from DAG

(i) The order of instr must respect order of nodes in DAG i.e nodes computed after children

(ii) Assignments to an array must follow all previous assignmt

(iii) Evaln of array ele must follow any previous assigned to some array

(iv) Any use of a variable must follow all previous procedure call indirect assignmt through ptr

(v) Any Proced call / indirect assignmt must follow all previous Evaln of Var

A Sample Code Generator:

One of the Primary issue during code gen is deciding how to use registers to best advantage.

There are four Principles for use of registers

(i) In most machines, some or all of opnds of an opn must be in registers in order to perform opn.

(ii) Reg make good temporaries

(iii) Reg are used to hold (global) values that are computed in one basic block & used in another block.

(iv) Reg are often used to help the run-time storage mgt.

The m/c instn are of ~~the~~ form

LD reg, mem
ST mem, reg
OP reg, reg, reg.

GURUPRASAD. S.
ASSISTANT PROFESSOR
DEPT OF CSE BMSIT, BANGALORE-64,
Ph: 9886760776

Register & address descriptors

The two data structures that are used in gen of code are

(i) Address Descriptors (AD):- keeps track of the locn where the current value of the variable can be found. the locn may be Reg, mem or stack or combination. the info can be stored in S.T entry for that variable name.

(ii) Register Descriptor - Keeps track of the variable names whose current value is in the register, we assume that initially, all register descriptors are empty. As the codegen progress each reg will hold the value of zero or more names.

The Code-Generation Algorithm

An essential part of alg is function getReg(I); which selects reg for each mem locn associated with 3-addr code instrn I

Fn getReg() has access to all the Regs and Address Descriptors of all variables in B.B.

M/c instrn for opn:

For a 3-addr instrn such as $x = y + z$, do the following

(i) we getReg($x = y + z$) to select reg for x, y and z call there R_x, R_y, R_z

(ii) If y is not in R_y then issue an instrn LD R_y, y' where y' is one of the mem locn for y

iii) 11'4 if z is not in R_z issue LD R_z, z' where z' is locn of z

(iv) Issue instrn ADD R_x, R_y, R_z

M/c instrn for Copy stmts

For stmt of the form $x = y$ getReg() will use same register both x & y

if y is not in reg generate LD R_y, y if already in Reg do nothing.

Ending the Basic Blocks

If the var is used temporarily only within its block then at the end of block value is forgot & reg is empty.

If var is live on Exit or we dont know about liveness, then we assume variable is needed and generate ST x, R.

Managing Register & Address Descriptor.

As the code gen Alg issues load store & other m/c instn it needs to update the register & address descriptor. The rules are as follows:

1) For the instr LD R, x.
   - a) Change RD for reg R to hold only x.
   - b) Change AD for x by adding reg R as addnl locn

2) For the instr ST x, R.
   Change AD of x to indicate its own mem locn

3) For an opn such as ADD Rx, Ry, Rz of $x = y + z$
   - ⓐ Change RD for Rx to hold x only
   - ⓑ Change AD for x so that its only locn is Rx
   - ⓒ Remove Rx from AD of any var other than x

④ When we process copy Stmt $x = y$ $\overset{\text{with}}{}$ load y into Ry
   - ⓐ ADD x to RD for Ry
   - ⓑ Change AD for x so that its only locn is Ry

Eg:-

t = a - b
u = a - c
v = t + u
a = d
d = v + u

t1 = a - b
t2 = a - c
t3 = t1 + t2
a = d
d = t3 + t2

R1, R2, R3

assume: t, u & v are temp and local to block

~~local to block~~ ~~[illegible]~~ ~~a, b, c, d~~

a, b, c, d are live on exit from block/

Stmt	Codegen	RD	AD

t = a-b	LD R1, a
	LD R2, b
	SUB R2 R1, R2
		R1 R2 R3
		a b
		a t
			a b c d t u v
			a b c d
			a b c d t u v
			a R1 b c d R2

u = a-c	LD R3, c. SUB R1, R1, R3	R1 R2 R3 u t c	a b c d t u a b c R3 a R2 R1
v = t+u	ADD R3, R2, R1	R1 R2 R3 u t v	t u v a b c a R2 R1 R3
a = d	LD R2, d	R1 R2 R3 u a, d v	t u v a b c d R2 R1 R3
d = v+u	ADD R1, R1, R3	R1 R2 R3 d a v	a b c d t u v a b c R1 R3
exit	ST a, R2 ST d, R1	d a v	a b c d t u v a R2 b c d R1 R3

Design of the function GetReg

For picking Reg Ry for y the rules are

(i) If y is currently in reg, pick a reg already containing y as Ry, donot issue m/c instn LD to load to reg

(ii) If y is not in reg, but there is reg currently empty pick one such reg as Ry

(iii) The difficult case if y is not in Reg & there are no empty reg then we should pick one of the allowable reg and make it safe to reuse.

Let R be a candidate reg & V is one of the var that the RD for R say, is in R then possibilities are:

ⓐ If the AD for V says that V is also value where beside R, then we are ok.

ⓑ If V is x the value being computed by instr I & x is not also one of the other operands of instn I then we are OK

ⓒ otherwise If V is not used later i.e after instn I and if V is live on exit from the blocks then we are ok

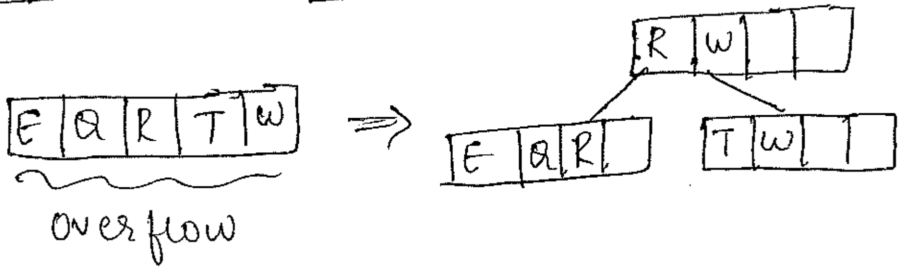
ⓓ If we are not ok by one of the first two case then we need to generate the store instn ST V, R place a copy of V in its own mem locn this opn is called spill.

End of unit 8

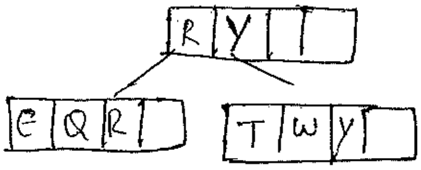
Q W E R T ~~Y~~ ~~U~~ ~~I~~ ~~O~~ ~~P~~ ~~A~~ ~~S~~ ~~D~~ ~~F~~ ~~G~~ ~~H~~ ~~J~~ ~~K~~ ~~L~~

Z X C V B N M

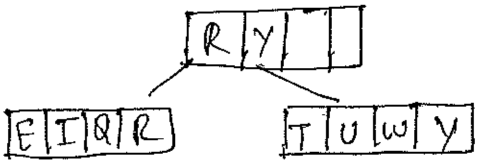
T ⇒



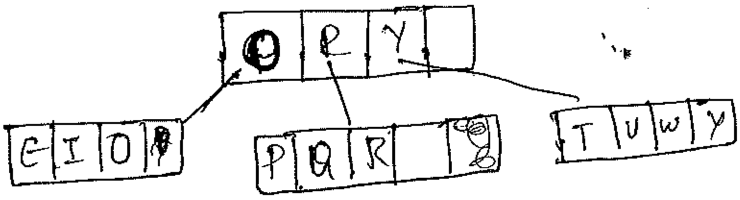
Y =



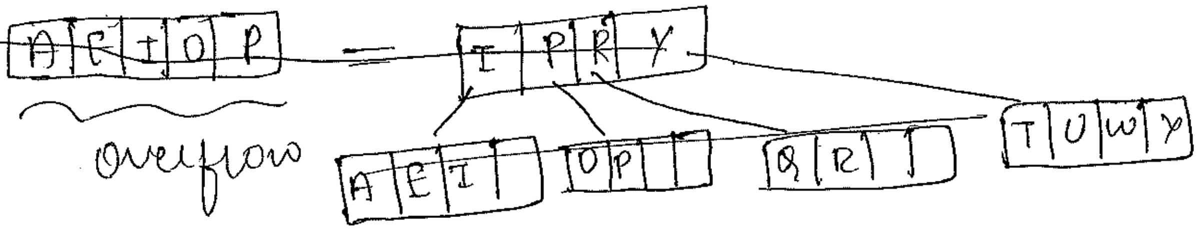
U, I =



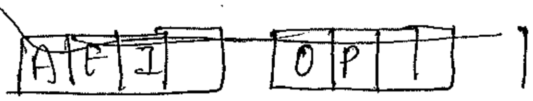
O, P :



A:



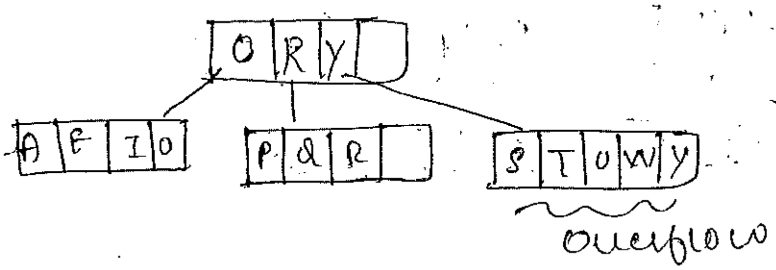
S:



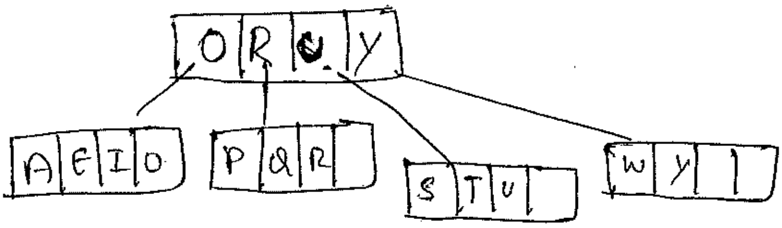
A:


A E I O
P Q R
T U W Y

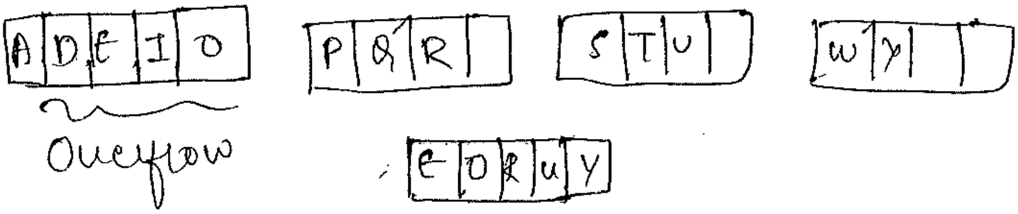
S:

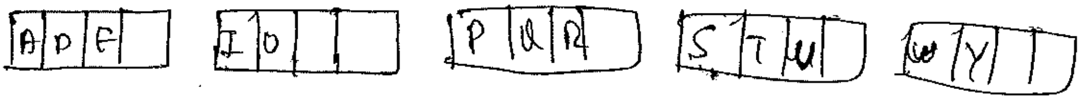

O R Y
A E I O
P Q R
S T U W Y
Overflow

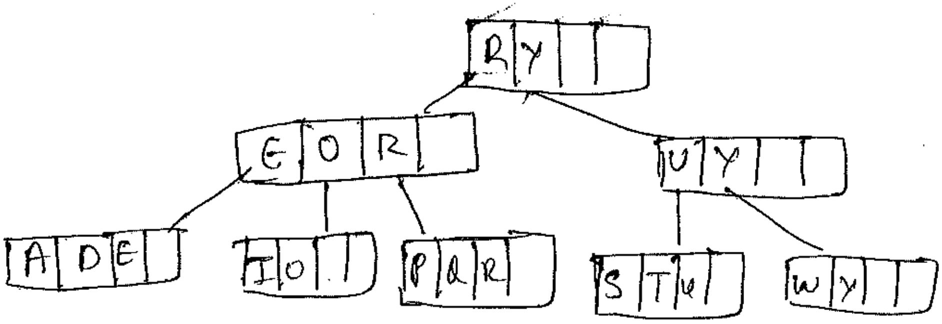
=


O R U Y
A E I O
P Q R
S T U
W Y

D:


A D E I O
P Q R
S T U
W Y
Overflow
E O R U Y

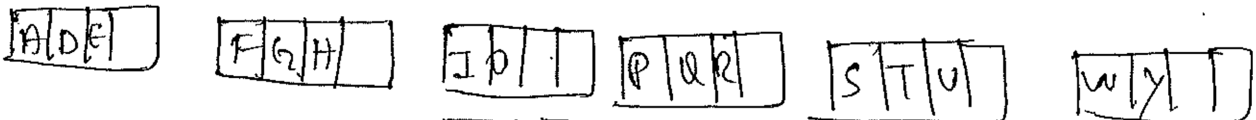

A D E
I O
P Q R
S T U
W Y


R Y
E O R
U Y
A D E
I O
P Q R
S T U
W Y

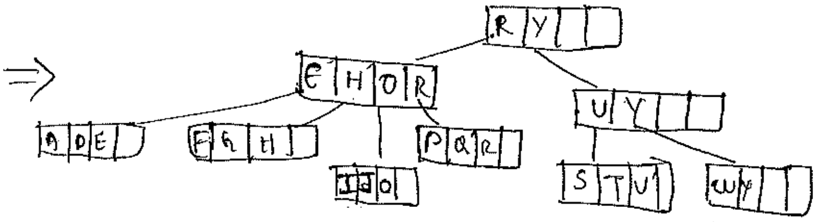
F, G:


A D E
F G I O
P Q R
S T U
W Y

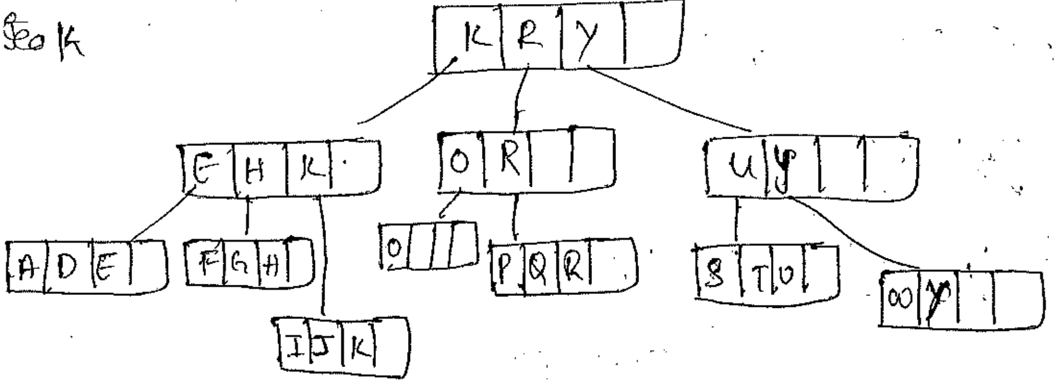
H:


A D E
F G H
I O
P Q R
S T U
W Y

J: ⇒


R Y
E H O R
U Y
A D E
F G H
I J O
P Q R
S T U
W Y

Seo K



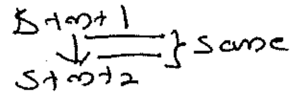
L :

Dataflow Analysis

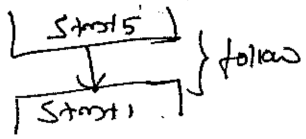
- All optzn depend on data flow analysis.
- DFA is a technique to derive info about flow of data along Prg Exn Path.
- Eg:- $X = a+b$ $X = a+b$ are both X, a, b are having same value or different — determined by DFA.

DFA assumptions / Abstraction

- Prg Exn is series of transformation, Each stmt transform (IR) i/p state to new o/p state.
- i/p State - Prg Pt before stmt and o/p State - Prg Pt after stmt
- To analyze Prg behavior we must consider all possible Prg Paths.
  - within a basic block - the Prg Pt after a Stmt is same as the Prg Pt before next Stmt



  - between B_1 and B_2 the Prg Pt of last stmt of B_1 is immediatly followed by first stmt of B_2



the Execution Path (Path) P_1 to P_n is a sequence of points $P_1, P_2 \ldots P_n$ such that

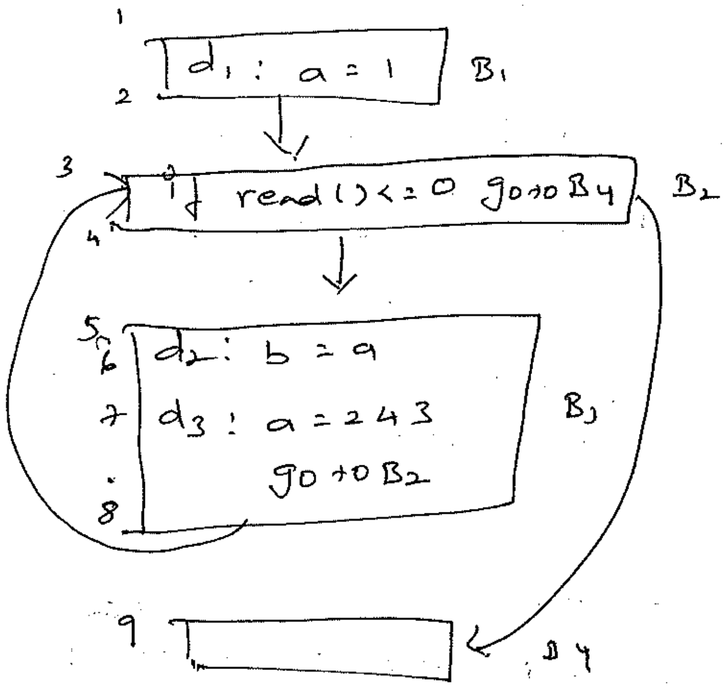
P_i Preceds. P_{i+1}

P_i is End of same block and P_{i+1} is start of Succor blocks

Different (infinate) no of Paths may be possible in a Prg

An analysis choose different State & diff info & no analysis is perfect repn of the State.

Eg:-



Differen possible paths {1, 2, 3, 4 9}, {1, 2, 3, 4, 5, 6, 7, 8, 3, 4 .. 9}

initially a=1 reaches till pt 5 d2 in first itrn in next itrn d3 reaches 3 so {d1, d3} are "reaching definition"

The Dataflow Analysis Schema

For Each appln of DFA

- at Every Prg pt a dataflow value is allocated that repn all possible prg states observed from that pt.
- data flow for Stmt 'S' is repn as In[S] and out[S] repn states before S and after S

two factors affect IN[S] and out[S]

① semantic of Stmt 'S' [transfer function)

② control flow.

Transfer function.

Eg:- $b = a.$ after stmt both b & a will be equal to same value.

transfer functions come in 2 flavours

① foreward flow

f_s takes dataflow value before stmt 'S' and Produce new data flow value after S

$$out[s] = f_s(IN[s])$$

Eg:- $a = a+1$

② backwardflow

f_s converts dataflow value after the stmt to new dataflow value before stmt

$$In[s] = f_s(out[s])$$

Control flow function.

data flow values are derived from flow of cntrl

if B' contains $S_1 \dots S_n$ then

$$IN[S_{i+1}] = out[S_i]$$

but the decision to diffiult between blocks it depends on all definations to reach the leader stmt of new blocks and $out[S_i]$ of last stmt of old blks

If blk B contains $s_1 \cdots s_n$ then,

$IN[B] = IN[s_1]$ $\quad OUT[B] = OUT[s_n]$

$f_B = f_{s_n} \circ \cdots f_{s_2} \circ f_{s_1}$

$$\boxed{OUT[B] = f_B(IN[B])}$$

foreword flow

$$\boxed{IN[B] = \cup_{P \text{ (Predecessor of } B)} OUT[P]}$$

$$\boxed{\begin{array}{l} IN[B] = f_B(OUT[B]) \\ OUT[B] = \cup_{S \text{ successor of } B} IN[S] \end{array}}$$

backward flow

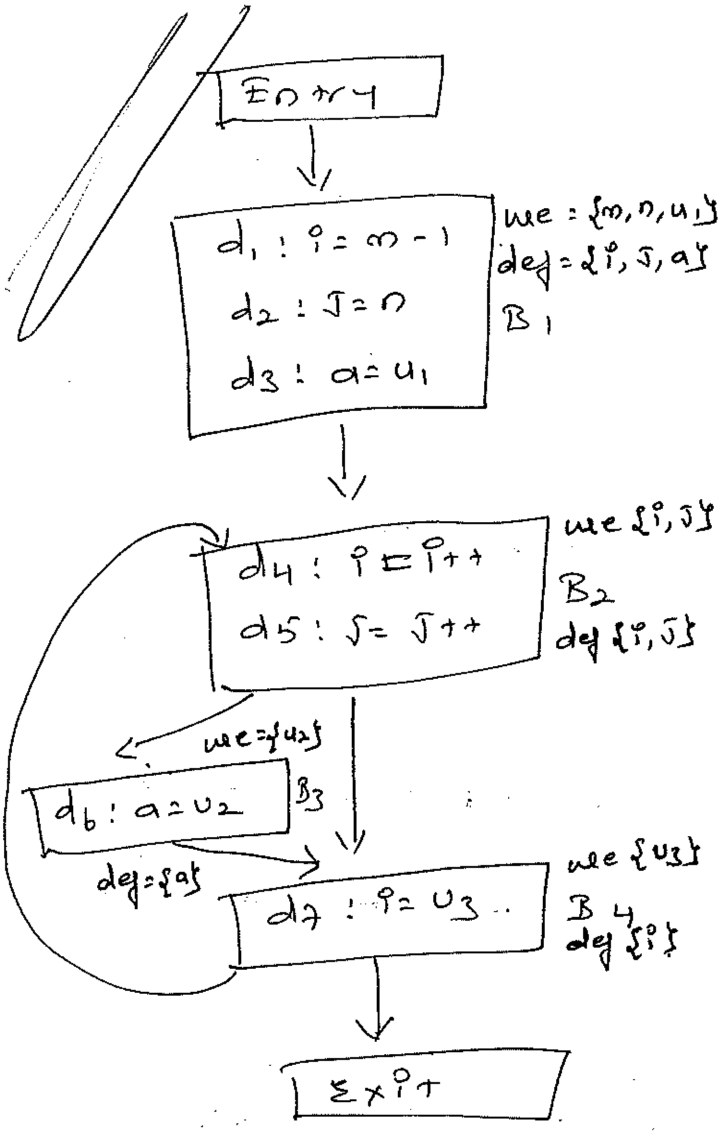
Reaching Definations

- where in a program each variable 'x' is defined.
- a defination d reaches point P if there is a path path from d to P such that d is not killed along the path.
- Kill of defination 'd' happens if 'x' is having any other defenition along the path.
- A defination 'd' is any stmt that assign value to 'x'

$$d: u = v + w$$

d generates a defn d of var u and kills all other defn of u

$$f_d(x) = gen_d \cup (x - killed)$$



gen $B_1 = \{d_1, d_2\ d_3\}$

kill $B_1 = \{d_4, d_5, d_6, d_7\}$

gen $B_2 = \{d_4, d_5\}$

kill $B_2 = \{d_1, d_2\ d_7\}$

gen $B_3 = \{d_6\}$

kill $B_3 = \{d_3\}$

gen $B_4 = \{d_7\}$

kill B_4 $\{d_4, d_1\}$

A.B.B generates set of definitions and kills set of defn gen set - downwards exposed - are set of defn generated in BB (gen takes precedence over kill)

if B_1

$d_1\ a = 3$
$d_2\ a = 4$

gen of $B_1 = d_2$
kill of B_1 = {d~~4~~, d_2}

Live Variable Analysis

Variable x is Live at pt P if x at pt P could be used along some path in flow graph starting at P Else x is dead

Live Var info is useful in register allocation for basic blocks.

Data flow Equations can be defined using

(i) IN[B] and OUT[B]

(ii) transfer function of B.B

- * def B - set of var defined in B prior to any use in B
- * use B - set of values used in B prior to defn of var

So

Var in use B is live on Entrance to B

Var in def B is dead in B as path starts from B

No Variables are live on Exit

Var is live coming into block B if it is used before re defination

Var is live coming out of the block if it is not redefined in the block or *(it is used in the successive block.)*

$$\text{In}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$$

$$\text{OUt}[n] = \bigcup_{s\ \text{successor}[n]} \text{in}[s]$$

Available Expression

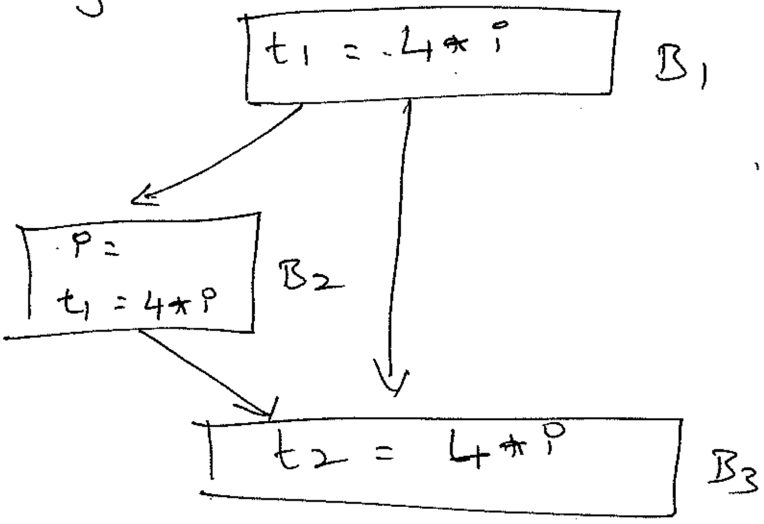
An Expression x+y is available at point P if every path from Entry to P evaluates x+y and after last Evaluation x and y are not changed

A block generates x+y if it Evaluates it

A block kills x+y if it assign x or y and donot re compute x+y

use! to detect global common sub Expression

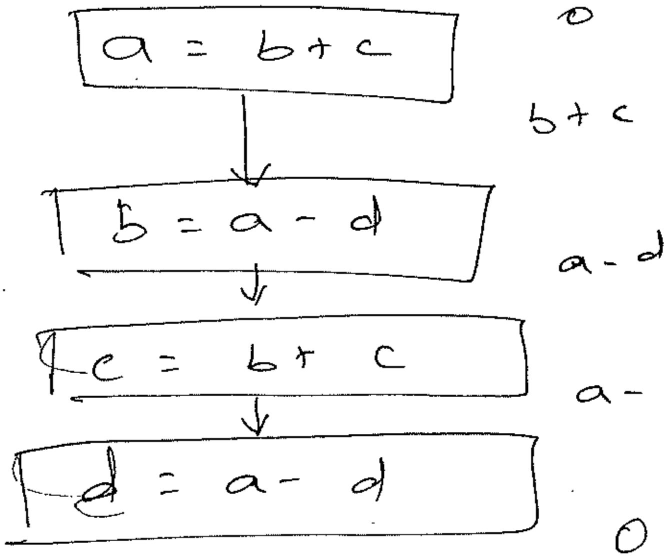
Eg!-



in B3 4*i is available if B2 do not Exist or recompute 4*i

4*i is not available in B3 if B2 only assign i and do not re compute 4*i.

Eg!-



b got Chaged