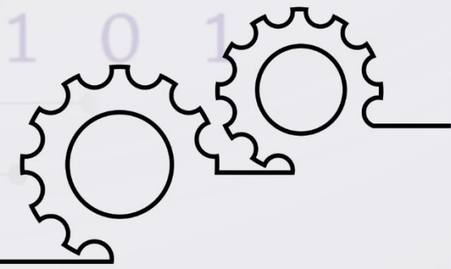
SIMATS
School of Engineering

Compiler Design

Computer Science and Engineering

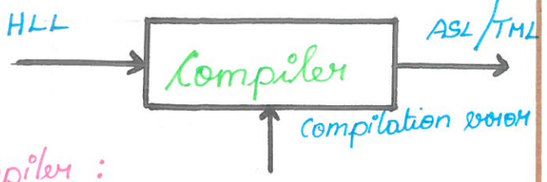


Saveetha Institute of Medical And Technical Sciences,Chennai.

SAVEETHA SCHOOL OF ENGINEERING
SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES
COMPUTER SCIENCE AND ENGINEERING
CSA14 - COMPILER DESIGN

Engineer to Excel

Compiler



Compiler :

* Software → converts High level language into Assembly language.

⇒ HLL - C, C++

⇒ LLL - Binary (or) machine language

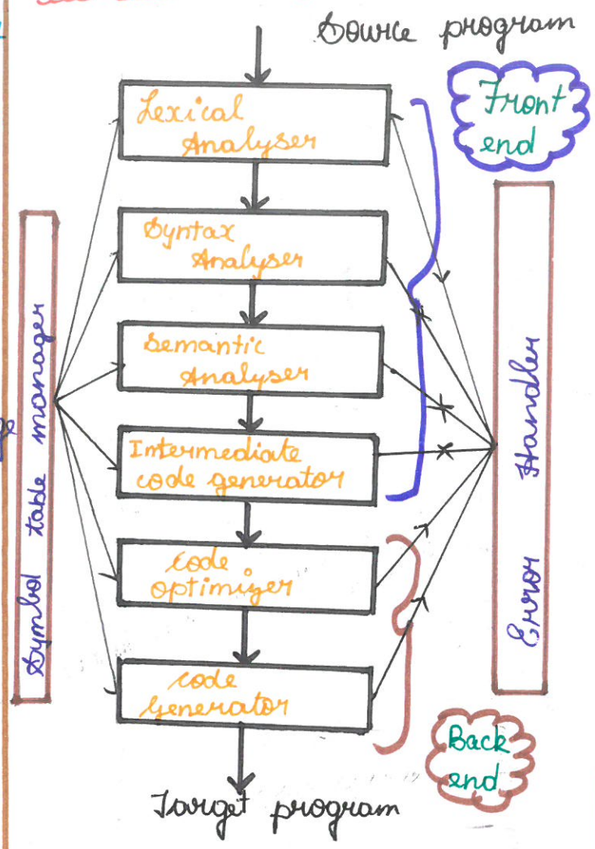
⇒ ASL - Mnemonics (i.e) MOV, ADD

Language processing System

High level language

- **pre-processor** — * Macro processing, * Header files
- Pure HLL
- **Compiler** — Translate source to target code
- Assembly language
- **Assembler** — Microprocessor 8085, 8086
- Relocatable machine code (object code)
- **Loader / Linker** — Loader - Executable code; Linker - resolves External memory address
- Absolute Machine code (Executable code)

The phases of a compiler



(i) **Lexical Analyser :**

* Produce tokens for the input lexeme.
* It is also called scanning

(ii) **Syntax Analyser :**

* construct parse tree (or) syntax tree for the tokens

(iii) **Semantic Analyser :**

* check for semantically consistent
* eg: Type checking Array-Bound Checking Scope - redeclaration

(iv) **Intermediate code generator**

* Convert syntax tree into intermediate code
* eg: Three address code.

(v) **code optimizer :**

* Produce optimized code
* (eg) Avoid unnecessary temp variable Less lines of coding

(v) **code generator :**

* convert optimized code into Assembly language code

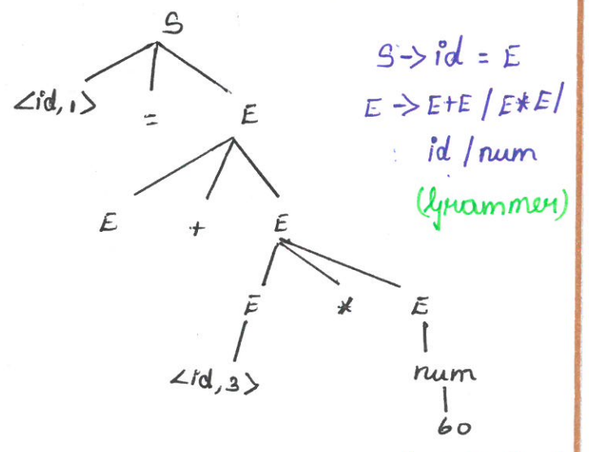
Example :

Position = inital + rate * 60

↓ Lexical Analyzer

<id,1><=><id,2><+><id,3><*><60>

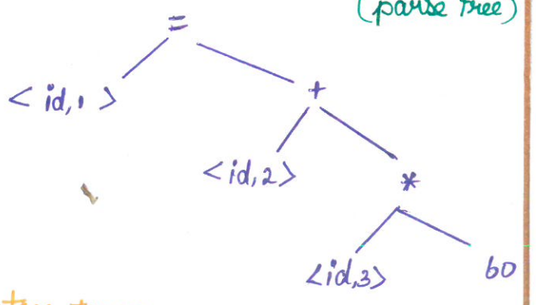
↓ Syntax Analyser



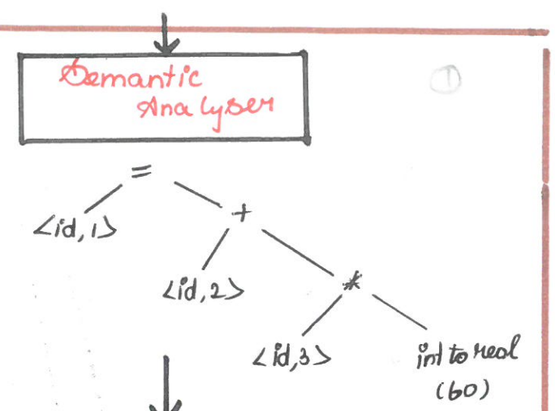
S -> id = E

E -> E+E / E*E / id / num (Grammer)

(parse tree)



(Syntax tree)



↓ Intermediate code generator

temp1 = int to real 60
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3

↓ code optimizer

temp1 = id3 * 60.0
id1 = id2 + temp1

↓ code generator

MOVF id3, R2
MVLF #60.0, R2
MOV F id2, R1
Add F R2, R1
MOV R1, id1

Grouping of phases

Two phase of compilation

a.) Analysis / Front end : L, S
Machine independent : S, I

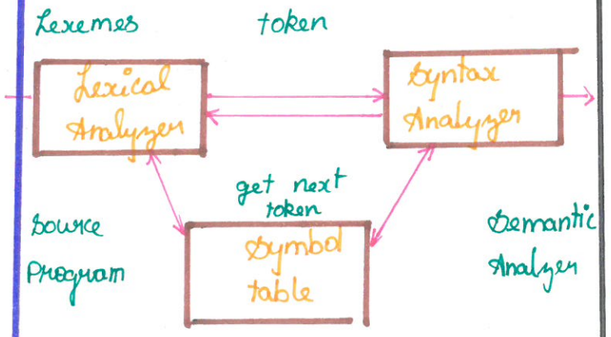
b) Synthesis / Back end/ CO,
Machine Dependent CO1

Compiler Construction Tools

Scanner generator	Parser generator	Syntax directed translation Engine	Data flow engine	Automatic code generator
I/p: Regular expression	I/p: context free grammer	I/p: Syntax tree	I/p Intermediate code	I/p optimized code
o/p: lexical analyzer	o/p: Syntax Analyzer	o/p: Intermediate code	o/p optimized code.	o/p object code

Role of Lexical Analyzer

* Read input character (lexemes)
* produce tokens



Function :

* Generate tokens
* Remove white space
* Remove comments
* correlates error message.

Need :

* simple design
* complier efficiency is improved
* complier portability is enchanced

Error Recovery Schemes :

* Panic mode recovery
* Local correction
* Gobal correction

Tokens, Patterns, lexemes

Tokens : * valid sequence of character

1. Keyword - if while
2. Constants - Pi,
3. Identifier - a, b
4. String - "Name", "age"
5. Number - 0, 1, 2 . . .
6. operators Punctuation - +, *, (,)

Patterns :

* Regular expression or grammer rule used to form tokens

Lexemes :

* instance of token (or) Sequence of character

Ex: a * b

C = a * b + 80

Lexemes	Tokens
a	Identifier
*	multiplication
b	Identifier

Specification of Tokens

Three Specification

1. String
2. language
3. Regular expression

Alphabet : * Character -> Finite Set of Symbols

String : * Finite Sequence of Symbols

Language : * Countable Set of Strings

operation on Strings :

* Length of String : |s|
* prefix : substring at beginning
* Suffix : substring at end.
* Substring : string at start, end & middle.

Operation on languages

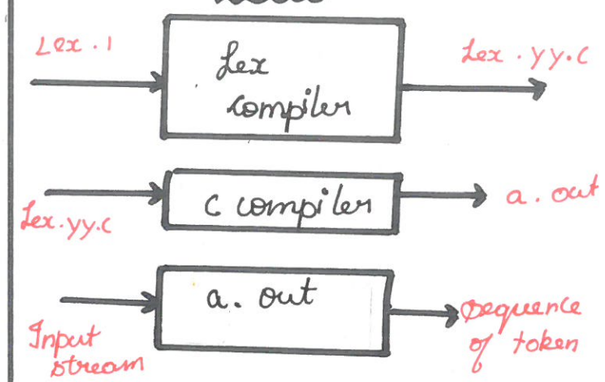
* union : L ∪ S
* concatenation : LS
* Kleene closure : L^*
* positive closure : L^+

language for Specifying lexical Analyser

* 1. ε -RE then L(ε) = {ε}
* 2. a - RE then L(a) = {a}
* 3. r, s -> RE then L(r) & L(s)
* r | s -> L(r) ∪ L(s)
* (r)(s) -> L(r) . L(s)
* r^* -> $(L(r))^*$

* unary * Highest precedence - left associate
* concatenation - 2nd highest - Left associate
* | - lowest Precedance - left associate

LEX



Lex Specification

* consist of three parts

(i) Declaration :

{ definitions }

%. %.

* Includes declaration of variables, constants & Regular expression

(ii) Translation Rules:

* Includes Statement of form

P_1 { action 1 }
P_2 { action 2 }
:
P_n { action n }

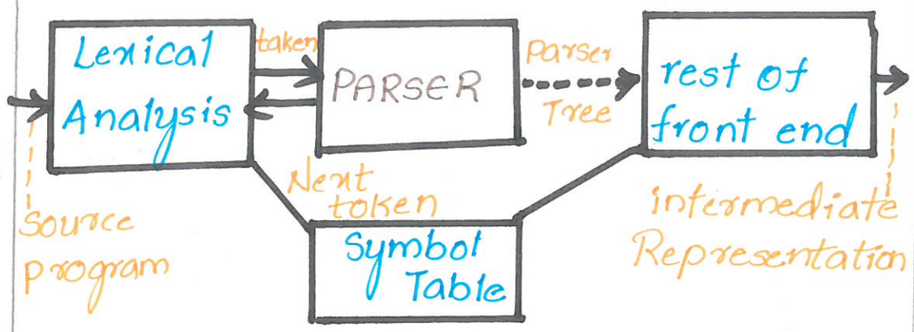
(iii) Auxillary procedure :

$D_1 = R_1$
:
$D_n = R_n$

* optional
* compiled Separately & loaded with lexical Analyzer

ROLE OF THE PARSER

- Input - tokens (or) stream of Tokens
- Output - Parse Tree.



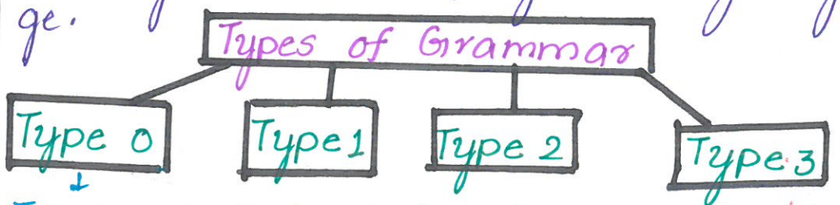
Functions :-

- Verify structure based on Grammar
- Construct parse tree
- Reports the error.
- Perform error recovery

Issues :- Cannot detect error such as

1. Variable redeclaration
2. Variable initalization before use
3. Data type mismatch

Grammar: It is used to describe the syntax of a programming language.



Type 0 - Unrestricted Grammar

Type 1 - Context Sensitive content free Grammar

Type 2 - Context free Grammar

Type 3 - Regular Grammar

Context Free Grammar:-

CFG is a Quadruple that consists of Terminals, Non-terminals, start symbol and production. It is type 2 Grammar

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

Non-Terminal are variables

V - variables
T - Terminals
P - production
S - start Symbol

Context Free Language :-

Collections of input string which are terminals derived from Start Symbol.

Terminals :-

Lower case a, b | Operator +, * | Digit 0, 1 ... 9 | id, if | ( )

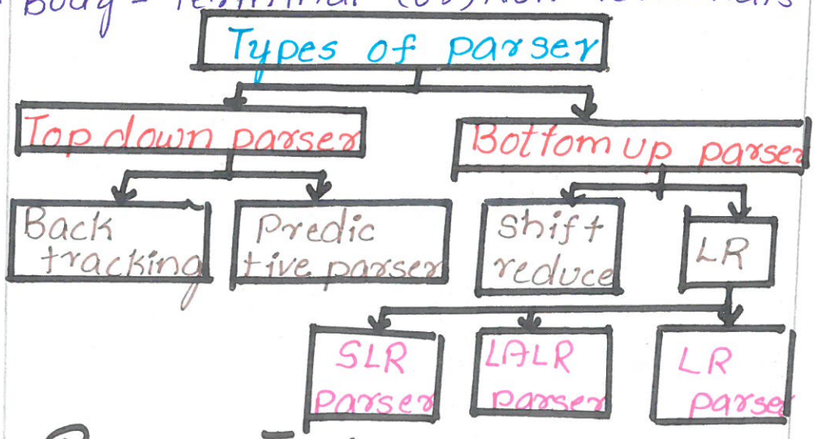
Non-Terminals :-

Upper case A, B, C | Lowercase italic expr, stmt.

Start Symbol :-

First in Grammar | Head of production

Production :- LHS → RHS
Head → Body
Head - only Non-Terminals
Body - Terminal (or) Non-Terminals



Parsing Techniques

Recursive Descent parsing

- Top Down parser
- Collection of recursive procedure

⇓

parsing given i/p string

Steps :-

1. Non terminal - call the procedure
2. Terminal - Matched with look ahead from input
3. Production rule has many alternat

⇓

All these alternates has to be combined into single body of proce

4. Parser - activated

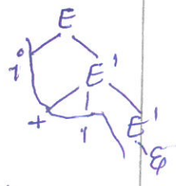
⇓

Procedure corresponding to start symbol.

Example:

$E \rightarrow iE'$ $E' \rightarrow +iE' / \varepsilon$

```
som
    E()
    {
    if (input == 'i')
    input ++;
    E PRIME ();
    }
    E PRIME()
    {
     if (input == '+')
     {
      input ++;
     if (input == 'i')
      input ++;
      E PRIME ();
     }
     else
      return;
     }
  main () {
   E()
   if (input == '$')
printf("parsing success");
  i/p string - i+i
```



PREDICTIVE PARSING

Implementation

1. Elimination of left recursion, left factoring & ambiguous Grammar.
2. Construct FIRST ( ) & FOLLOW ( ) for all non-Terminals
3. Construct Predictive Parsing Table.
4. Parse the input String given using Stack & parsing Table.

(i) Last Recursion Elimination :

$A \rightarrow A\alpha \mid B \Rightarrow A \rightarrow BA'$

$A' \rightarrow \alpha A' \mid \varepsilon$

$\alpha, \beta \rightarrow$ String

(ii) Left Factoring :

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \Rightarrow A \rightarrow \alpha A' \mid \varepsilon$

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

$\alpha, \beta_1, \beta_2 \rightarrow$ String

(iii) FIRST ( ) :

① If x is terminal, then FIRST (x) is {x}.

② If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST (x).

③ If x is NT & $X \rightarrow a\alpha$ is a production then add a to First (x)

④ If x is NT. $X \rightarrow Y_1Y_2$ is a production

FIRST (x) = FIRST (Y_1)

If, $Y_1 = \varepsilon$, FIRST (x) = FIRST (Y_2)

$Y_2 = \varepsilon$, FIRST (x) = FIRST (Y_3)

$Y_K = \varepsilon$, FIRST (x) = {ε}

FOLLOW (x)

① If S is start Symbol, then FOLLOW (S) = {$}

② If $A \rightarrow \alpha B\beta$ production, then FOLLOW (B) = FIRST (β) except ε

③ If $A \rightarrow \alpha\beta$ or $A \rightarrow \alpha B\beta$ where FIRST (β) contain ε then, FOLLOW (B) = FOLLOW (A).

Predictive parsing (LL (1))

Construct predictive parsing table for the grammar.

$E \rightarrow E+T \mid T$

$T \rightarrow T+F \mid F$

$F \rightarrow (E) \mid id$ and i/p string is id+id * id

Step 1 : Eliminate left Recursion

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Left Recursion → ε start with ε

Step 2: No left Factoring bcos No two Choice.

Step 3: FIRST & FOLLOW

FIRST (E) = {FIRST (T) = FIRST (F)

FIRST (F) = {(, id}

FIRST (E') = {+, ε}

FIRST (T') = {*, ε}

Follow:

FOLLOW (E) = {$,)}

FOLLOW (E') = {$,)}

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

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

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

Since Follow of E' is Follow ④

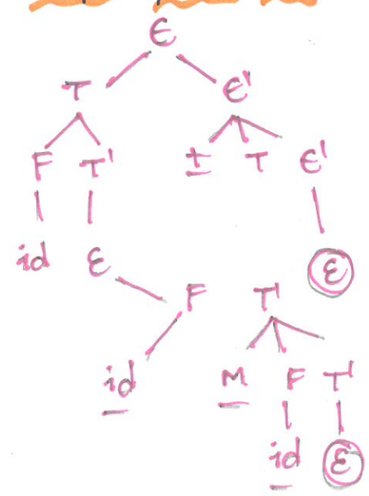
Predictive parsing Table: Step 4

Non-Terminal	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Step 5: Stack Table

Stack	Input	Out Put
$E	id+id * id$	
$E'T	id+id * id$	$E \rightarrow TE'$
$E'T'F	id+id * id$	$T \rightarrow FT'$
$E'T'id	id+id * id$	$F \rightarrow id$
$E'T'	+id * id$	
$E'	+id * id$	$T' \rightarrow \varepsilon$
$E'T+	+id * id$	$E' \rightarrow +TE'$
$E'T	id * id$	
$E'T'F	id * id$	$T \rightarrow FT'$
$E'Tid	id * id$	$F \rightarrow id$
$E'T'	* id$	
$E'F*	* id$	$T' \rightarrow *FT'$
$E'T'F	id$	
$E'T'id	id$	$F \rightarrow id$
$E'T'	$	
$E'	$	$T' \rightarrow \varepsilon$
$	$	$E' \rightarrow \varepsilon$

Step 6: parse Tree



id + id * id

Parse Tree Generated.

Shift Reduce parser

* bottom-up-parser
* leaves to the root
* Reduction are done
* While constructing the parsetree

Consider the following grammer

$S \to cc$

$C \to cC$

$C \to d$

input string : cdcd

$\Rightarrow$ cdcd = Reduction by "$C \to d$"

$\Rightarrow$ cdcC = Reduction by "$C \to cC$"

$\Rightarrow$ cdC = Reduction by "$C \to d$"

$\Rightarrow$ cC = Reduction by "$C \to cC$"

$\Rightarrow$ S

Handles :

* Substring that matches the right side of a production

Example : $A \to aXb$, $X \to c$

$\Rightarrow$ Here 'c' is a handle

$\Rightarrow$ It reduces to X

I/p string acb

$\Rightarrow$ a cb Reduction by "$X \to c$"

$\Rightarrow$ a Xb Reduction by "$A \to aXb$"

$\Rightarrow$ A

Handle pruning :

* process of obtaining the starting non-terminal.

↳ reducing handles to the respective non-terminal

Stack Implementation of Shift reduce parser

* Stack - grammer symbols
* Initially pushed with the terminal $
* i/p buffer - i/p string $

Parsing Actions

* Shift next i/p symbol

Shift - * When there is no handle for reduction

Reduce - * Reduce by a non terminal
* non terminal $\Rightarrow$ pushed $\Rightarrow$ stack
  ⇓ by replacing the handles

Accept : input string $\to$ valid

Error - * Syntax error
* parser calls error recovery routine

Example :

$S \to cC$

$C \to cC$

$C \to d$

I/p string : ccdd

State	I/p string	Action
$	ccdd$	Shift
$c	cdd$	Shift
$cc	dd$	Shift
$ccd	d$	Reduce by $C \to d$
$ccC	d$	Reduce by $C \to cC$
$cC	d$	Reduce by $C \to cC$
$C	d$	Shift
$Cd	$	Reduce by $C \to d$
$CC	$	Reduce by $S \to CC$
$S	$	accept : parsing is successfully done

Example :

Consider the following grammer

$S \to (L) / a$

$L \to L, S / S$

parse the input string (a,(a,a)) using a shift-reduce parser.

Solution :

Stack	Input	Action
$	(a,(a,a))$	Shift
$(	a,(a,a))$	Shift
$(a	,(a,a))$	Reduce $S \to a$
$(S	,(a,a))$	Reduce $L \to S$
$(L	,(a,a))$	Shift
$(L,	(a,a))$	Shift
$(L,(	a,a))$	Shift
$(L,(a	,a))$	Reduce $S \to a$
$(L,(S	,a))$	Reduce $L \to S$
$(L,(L	,a))$	Shift
$(L,(L,	a))$	Shift
$(L,(L,a	))$	Reduce $S \to a$
$(L,(L,S	))$	Reduce $L \to L,S$
$(L,(L	))$	Shift
$(L,(L)	)$	Reduce $S \to (L)$
$(L,S	)$	Reduce $L \to L,S$
$(L	)$	Shift
$(L)	$	Reduce $S \to (L)$
$S	$	Accept

Applications :

1. Implementation of High-level programming.
2. Design of new Computer architectures.
3. Software productivity tool

SLR Parsing

SLR Parsing

Implementation Steps:

1. Augment G and Produce G'
2. Construct the canonical collection of set of items C for G'
3. Construct the parsing action function action & goto using algorithm that require FOLLOW(A) for each non-Terminal of Grammar.

Example for SLR Parsing:

Construct SLR parsing for the following Grammar:

G: $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

Step 1 : Augment G:

G: $E' \to E$

$E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to (E)$

$F \to id$

Step 2: LR(0)

I_0 : $E' \to .E$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

GOTO (I_0, E):

I_1 : $E' \to E.$

$E \to E.+T$

GOTO (I_0, T):

I_2 : $E \to T.$

$T \to T.*F$

GOTO (I_0, F):

I_3 : $T \to F.$

GOTO (I_0, C):

I_4 : $F \to (.E)$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

GOTO (I_0, id):

I_5 : $F \to id.$

GOTO (I_1, +):

I_6 : $E \to E+.T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

GOTO (I_2, *):

I_7 : $T \to T*.F$

$F \to .(E)$

$F \to .id$

GOTO (I_4, E):

I_8 : $F \to (E.)$

$E \to E.+T$

GOTO (I_4, T):

I_9 : $E \to T.$

$T \to T.*F$

GOTO (I_4, F):

$T \to F.$

GOTO (I_4, C):

$F \to (.E)$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

GOTO (I_6, id):

$F \to id$

GOTO (I_6, T)

$E \to E+T.$ I_9

$T \to T.*F$

GOTO (I_6, F)

$T \to F.$ I_3

GOTO (I_6, C)

$E \to (.E)$

$E \to .E+T$

$E \to .T$ I_4

$T \to .T*F$

$T \to .F$

GOTO (I_6, id)

$F \to id.$ I_5

GOTO (I_7, F)

$T \to T*F.$ I_{10}

GOTO (I_7, C)

$E \to (.E)$ I_4

⋮

$E \to .id$

GOTO (I_7, id)

$F \to id$ I_5

GOTO (I_8,)):

$F \to (E).$ I_{11}

GOTO (I_8, +)

$E \to E+.T$ I_6

⋮

$E \to .id$

GOTO (I_9, *)

$T \to T*.F$ I_7

$F \to .(E)$

$F \to .id$

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

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

Step 3: Parsing Table:

	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
I_0	S_5			S_4			1	2	3
I_1		S_6				ACC			
I_2		r_2	S_7		r_2	r_2			
I_3		r_4	r_4		r_4	r_4			
I_4	S_5			S_4			8	2	3
I_5		r_6	r_6		r_6	r_6			
I_6	S_5			S_4				9	3
I_7	S_5			S_4					10
I_8		S_6			S_{11}				
I_9		r_1	S_7		r_1	r_1			
I_{10}		r_3	r_3		r_3	r_3			
I_{11}		r_5	r_5		r_5	r_5			

Check id+id*id is valid or NOT

Stack	Input	Action
0	id+id*id$	GOTO(I_0, id) = S_5, shift
0 id 5	+id*id$	GOTO(I_5, +) = r_6, R, F→id
0 F 3	+id*id$	GOTO(I_0, F) = 3 GOTO(I_3, +) = r_4, R, T→F
0 T 2	+id*id$	GOTO(I_0, T) = 2 GOTO(I_2, +) = r_2, R→E→T
0 E 1	+id*id$	GOTO(I_0, E) = 1 GOTO(I_1, +) = S_6, shift
0 E 1 +6	id*id$	GOTO(I_6, id) = S_5, shift
0 E 1 +6 id 5	*id$	GOTO(I_5, *) = r_6 : R, F→id
0 E 1 +6 F 3	*id$	GOTO(I_6, F) = 3 GOTO(I_3, *) = r_4, R, T→F
0 E 1 +6 T 9	*id$	GOTO(I_6, T) = 9 GOTO(I_9, *) = S_7, shift
0 E 1 +6 T 9 *7	id$	GOTO(I_7, id) = S_5, shift
0 E 1 +6 T 9 *7 id 5	$	GOTO(I_5, $) = r_6, R, F→id
0 E 1 +6 T 9 *7 F 10	$	GOTO(I_7, F) = 4 GOTO(I_4, $) = r_3, R→T→T*F
0 E 1 T 6 T 9	$	GOTO(I_6, T) = 9 GOTO(I_0, $) = r_1 R→E→E+T
0 E 1	$	GOTO(I_0, E) = 1 GOTO(I_1, $) = accept

Operator Precedence Parsing & YACC

Operator Precedence Parsing

- Bottom up parsing
- Class of Grammars → Operator Grammar

Operator Grammar Properties :-

- no two variables are adjacent
- no ε production

Ex:
$E \to E+E / E*E / id$
↳ operator grammar

$E \to EAE / id$ (EAE: Adjacent)

$A \to + / *$

$E \to E+E / E*E / id$

Operator Precedence Relations

Relation	Meaning
$a <\cdot b$	a has less precedence than b
$a \cdot> b$	a has more precedence than b
$a \doteq b$	a has equal precedence than b

Precedence & Associativity

Operator	Precedence
( )	1
-(unary minus)	2
↑	3
*, /	4
+, -	5

Left associative - +, *, - /

Right associative - ↑

id - highest precedence compared to all the operators

$ - least precedence

Ex
$E \to E+E / E*E / id$

	id	+	*	$
fd		·>	·>	·>
f+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

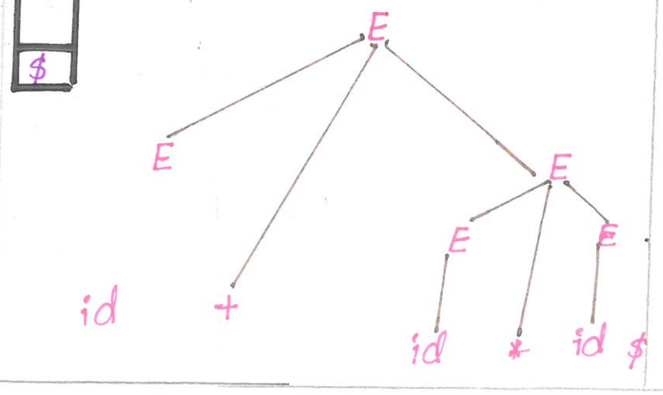
Blank - error

·> (or) ≐ → push

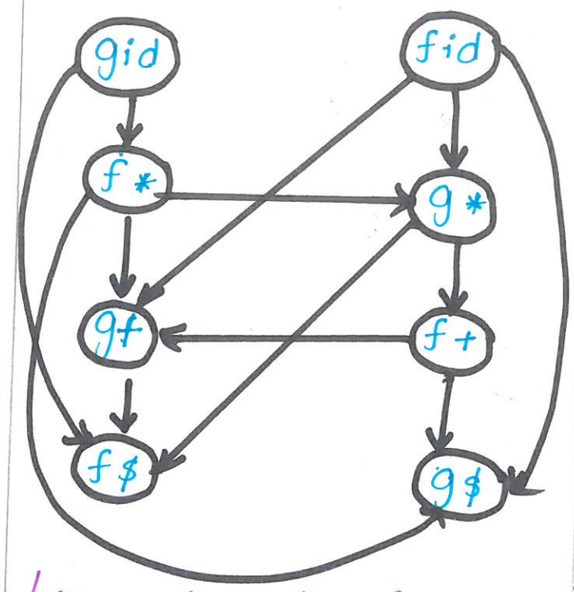
·> → Pop

id	+	id	*	id	$

$



Operator Precedence Flow Graph :-



Longest Path for each node :-

fid ⇒ fid ⇒ g* → f+ → g+ ⇒ f$ ⇒ 4

f+ ⇒ f+ → g+ → f$ ⇒ 2

f* ⇒ f* → g* → f+ → g+ → f$ ⇒ 4

gid ⇒ gid → f* → g* → f+ → g+ → f$ ⇒ 5

g+ ⇒ g+ → f$ ⇒ 1

g* ⇒ g* → f+ → g+ → f$ ⇒ 3

f$ ⇒ 0

g$ ⇒ 0

Operator Precedence relation function :-

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

Operator precedence parser → ambiguous grammar

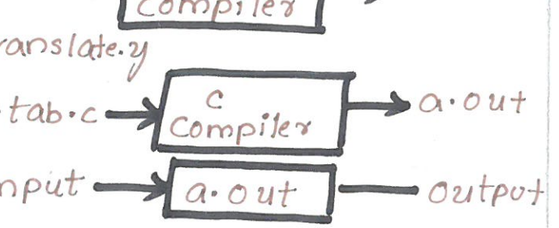
YACC - Yet Another Compiler Compiler

- Stephen Johnson
- LALR parser

token → LALR → parse tree

Creating an Input/output translation with Yacc

Yacc specification



A Yacc source Program - 3 parts

1. declarations %% %%
2. translation rules %% %%
3. suporting C routines

declarations

- Ordinary C declarations delimited by %{ and %}
- Declarations of gramar tokens

Translation Rules :-

Grammar production → Semantic-action General production form
<head> → <body1> / <body2> / --- <body n>

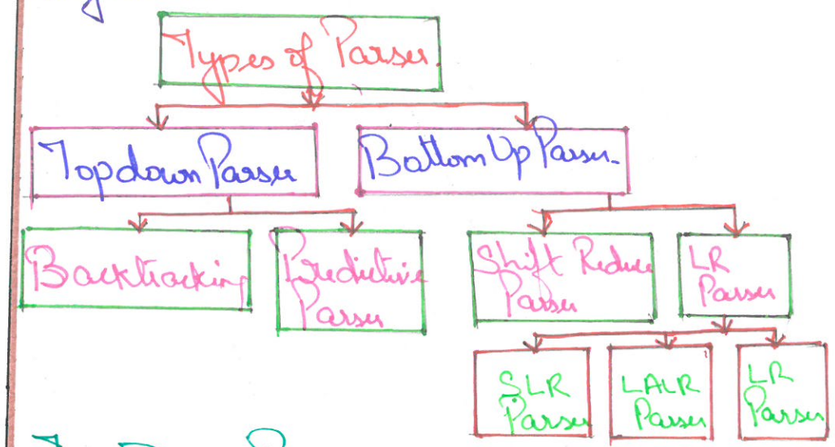
Supporting C routines :-

yylex() - lexical analyzer

TOP DOWN PARSER, ERROR HANDLING & RECOVERY

PARSING TECHNIQUES:-

* Parser Scans the Input string from left to right and Identifies that the derivation is leftmost or rightmost.



TOP-DOWN PARSER:-

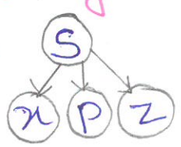
* Tree Generated from top to bottom
* Derivation terminates when the Requied input string terminates.
* Leftmost derivation matches this Requirement
* Main task is to find the appropriate Production Rule in order to Produce the Correct I/p String.

Eg:- Grammar: $S \rightarrow xPy$, $P \rightarrow yw/y$.

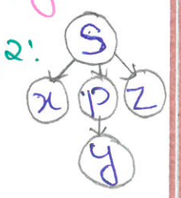
x	y	z

↑ Input Buffer

Step 1:



Step 2:

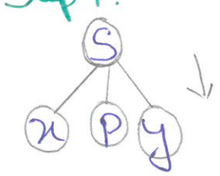


BACKTRACKING:-

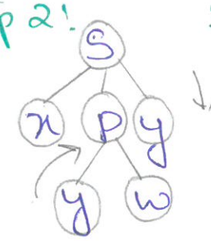
Technique in which for Expansion of non terminal Symbol Choose One alternative and if mismatch they another alternate any.

Eg:- Grammar: $S \rightarrow xPy$, $P \rightarrow yw/y$.

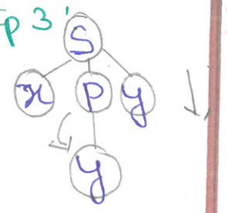
Step 1:



Step 2:



Step 3:



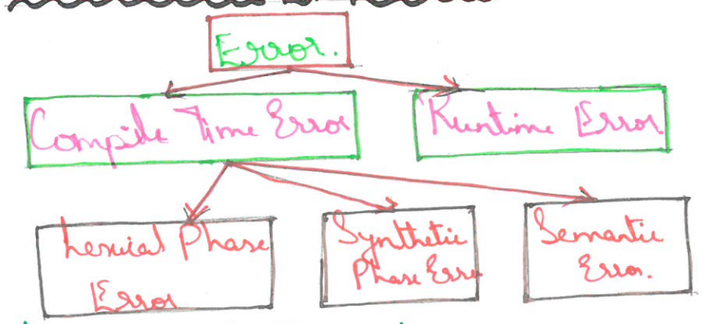
ERROR DETECTION AND RECOVERY.

* All Possible errors made by the Programmer are detected & reported to user in message form.
* This Process is called Error Handling Process.

FUNCTIONS OF ERROR HANDLER

* Should Identify all Possible Error from Source Code
* Should report these error with appropriate message to User.
* Should repair the error at instance in order to Continue Processing of Pgm.
* Should recover from errors to detect as many as Possible.

CLASSIFICATION OF ERRORS:



FUNCTIONS OF ERROR HANDLER:

1) Error Detection
2) Error Report
3) Error Recovery.

Lexical: Misspelling of Identifier, Keywords or Operators

Syntactic: A missing Semicolon or Unbalanced Parenthesis.

Semantic: Incompatible Value assignment or type mismatch b/w operator & Operand.

ERROR RECOVERY:

Error Recovery Method	Lexical Phase Error	Syntactic Phase Error	Semantic Phase Error
Panic Mode	✓	✓	✗
Phrase-(Lv) Mode	✗	✓	✗
Error Production	✗	✓	✗
Global Production	✗	✓	✗
Using Symbol Table	✗	✗	✓

ERROR RECOVERY STRATEGIES:-

i) PANIC MODE RECOVERY: Parser discards the I/p Symbol One at a time Until One of the designated Set of Synchronizing token is found.

ii) PHASE LEVEL RECOVERY: Parser Performs local Correction On the remainy I/p if Error found, means Corrective mode.

iii) ERROR PRODUCTION: Incaporated if user is aware of common mistakes that are Encounted in Pgm in Conjuction with error.

iv) GLOBAL CORRECTION: Tries to find closest match for it which is error free. The closest match is One that does not do any insertion, deletion & Changes of token.

Syntax Directed Definitions & Annotated Parse tree

Semantic Analysis

* To check the correctness of program
* Enable proper execution action
* Type checking -> number and type of argument
  ↓ in
  ⇒ function calls + function header of function definition
* Object binding -> Associating variable
  ↓ respective function definition
* Automatic type conversion
  ↳ integer in mixed mode operation
  ⇒ Help in intermediate code generation
  ⇒ Display appropriate error message

Semantics of a languages
↳ described using two notations
⇒ Syntax directed definition (SDD)
⇒ Syntax directed Translation (SDT)

Syntax Directed Definitions

SDD = CFG + attributes + Semantic rules

Eg:
$E \to E_1 + T$

Production	Semantic Rule
$E \to E_1 + T$	$E.val = E_1.val + T.val$ Val -> attribute

Attribute:

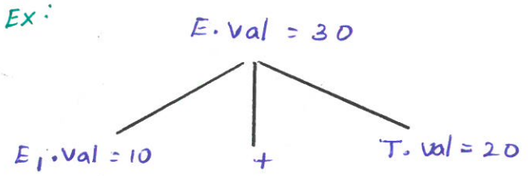
* property of programming language construct
* Associated with grammer symbols
* X -> grammer symbol
* a -> attribute
* X.a -> value of attribute 'a' at a particular nodes

Synthesized Attribute:

Parent ← child (Bottom up)

* Child -> parent (BUP)

Ex:



Inherited Attribute: (I-Attribute)

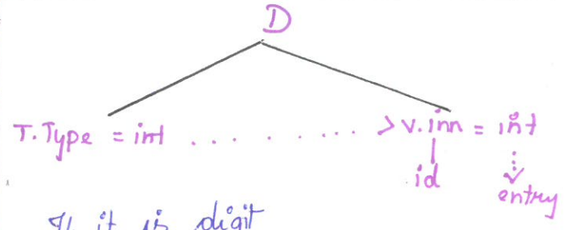
Parent → Sibling → child (Top Down)

Eg:

D -> TV
int sum

D - Declaration
T - Type -> int
V - variable -> sum

production	Semantic rule
D -> TV ↳ Sibling	V.Inh = T.Type ↳ T - attribute



If it is digit
↓
digit . lexval.

Annotated parse tree

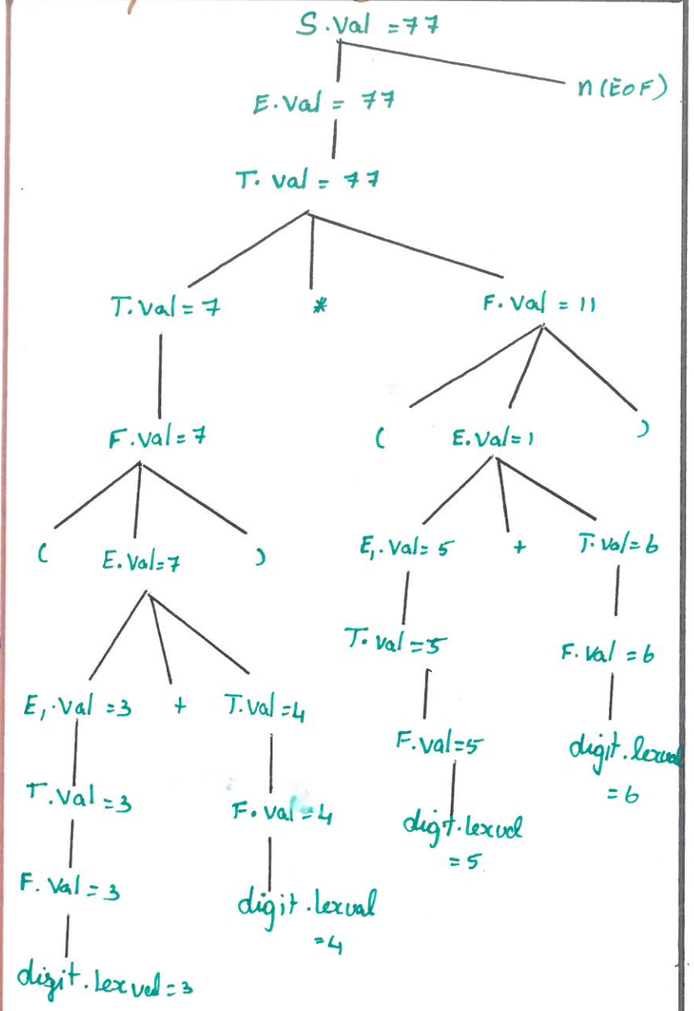
parse tree
⇓
attribute values of each node.

* Terminal => Synthesized attribute
  ⇓
  obtained directly from lexical analyser
* Other nodes => Synthesized (or) inherited attribute

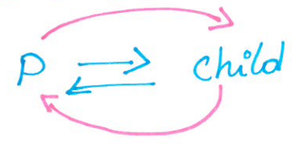
Eg: Write the SDD for a simple desk calculator and draw annotated parse tree for the expression (3+4)*(5+6) n

$S \to E\,n$
$E \to E_1 + T \mid E_1 - T \mid T$
$T \to T_1 * F \mid T_1 \mid F \mid F$
$F \to (E) \mid digit$

production	Semantic rules
$S \to E\,n$	S.val = E.val
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to E_1 - T$	$E.val = E_1.val - T.val$
$E \to T$	E.val = T.val
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to T_1 \mid F$	$T.val = T_1.val / F.val$
$T \to F$	T.val = F.val.
$F \to (E)$	F.val = E.val
$F \to digit$	F.val = digit.lexval



Circular dependency:



Eg:

production	Semantic Rule
A -> B	A.S = B.i B.i = A.S + 6.

partial annotated parse tree

circular dependency

Construction of Syntax tree, Bottom up Evaluation of S & L Attribute

Construction of Syntax tree

leaf → operands

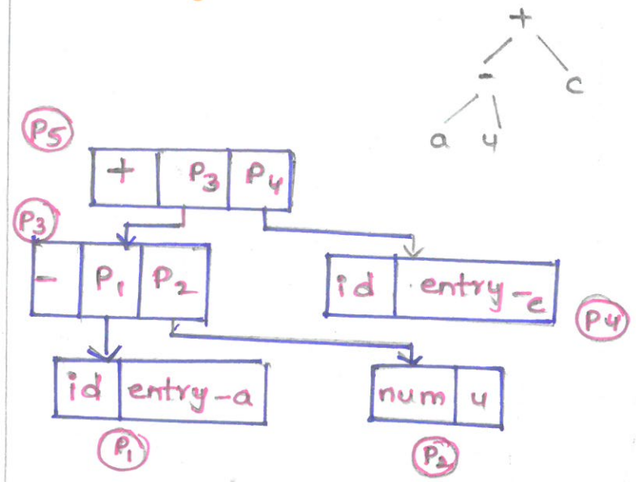
Interior node → operator

Functions

1. mx node (OP, left, right)
2. mx leaf (id, entry to symbol table)
3. mx leaf (num, value)

Steps for construction of Syntax tree for a-4+c

Symbol	operation
a	P_1 = mx leaf (id, entry-a)
4	P_2 = mx leaf (num, 4)
-	P_3 = mx node (-, P_1, P_2)
c	P_4 = mx leaf (id, entry-c)
+	P_5 = mx node (+, P_3, P_4)



Syntax Directed Definition (SDD)

Productions	Semantic rule
$E \to E_1 + T$	E.node = mx node (+, E_1.node, T.node)
$E \to E_1 - T$	E.node = mx node (-, E_1.node, T.node)
$E \to T$	E.node = T.node
T → id	T.node = mx leaf (id, id.entry)
T → num	T.node = mx leaf (num, num.value)

Bottom-up

S-Attributed Definitions - only Synthesized attributes → SDD

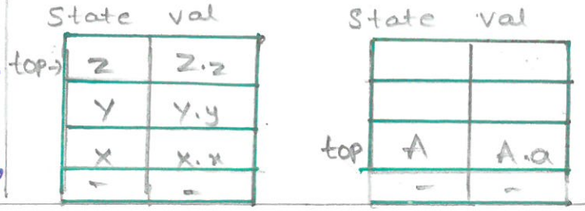
L-Attributed Definitions - Synthesized Attribute & inherited at attribute.

Bottom-up Evaluation of S-Attributed Definitions ⇒ implemented - LR parser

S-attributed definition
⇓
Parser generator
⇓
Construct translator
⇓
Evaluates attributes

- Stack ⇒ Grammar symbols, value of attribute
⇒ Pair of arrays val & state.

A → XYZ A.a = f (X.x, Y.y, Z.z)



Production	Semantic Rules
L → En	Print (val [top-1])
$E \to E_1 + T$	val [ntop] = val [top-2] + val [top]
E → T	
$T \to T * F$	val [ntop] = val [top-2] * val [top]
T → F	
F → (E)	val [ntop] = val [top-1]
F → digit	

- digit → push digit.lexval into - val stack
- other terminals - no shift.

Input	State	Val	Semantic rule
$S+3\times4n$	-	-	
$+3\times4n$	5	5	
$+3\times4n$	F	5	F → digit
$+3\times4n$	T	5	T → F
$+3\times4n$	E	5	E → T
$3\times4n$	E+	5+	
$\times4n$	E+3	5+3	
$\times4n$	E+F	5+3	F → digit
$\times4n$	E+T	5+3	T → F
4n	E+T×	5+3×	
n	E+T×4	5+3×4	
n	E+T×F	5+3×4	F → digit
n	E+T	5+12	T → T, ×F
n	E	17	E → E_1 + T
	En	17 -	L → En
	L	17	

L-Attributed Definitions

X - Synthesized attribute X.S

Y - inherited attribute X.i

Y.i = X.S

D → TL

T → int | float

L → L_1 , id | id

" float p, q, r"

SDD for Simple type declarations

production	Semantic rule
D → TL	L.inh = T.type
T → int	T.type = integer
T → float	T.type = float
$L \to L_1$, id	L_1.inh = L.inh add type (id.entry, L.inh)
L → id	add type (id.entry, L.inh)

Input	State	productions
float p,q,r	-	-
p,q,r	float	-
p,q,r	T	T → float
,q,r	Tp	-
,q,r	TL	L → id
,q,r	TL,	-
,r	TL,q	-
,r	TL	L, L, id
r	TL,	-
	TL,r	-
	TL	L → L, id
	D	D → TL

TOP DOWN TRANSLATION

Top Down Translation:

- I-attribute needed
- Elimination of left-recursion

Obtain SDD for the following grammar

$S \to En$
$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid digit$

- annotated parse tree for the expression $(3+4)*(5+6)n$

- After eliminating left recursion

$S \to En$
$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid digit$

- Compute attribute value

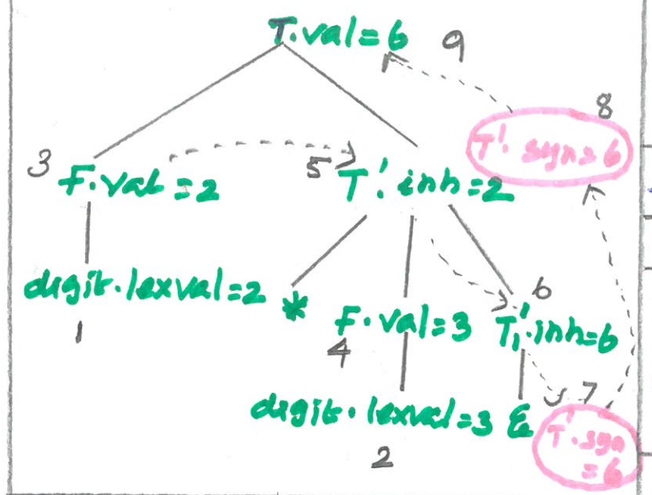
$L.H.S \leftarrow R.H.S$

⇒ synthesized attribute

production	Semantic rule	Type
$S \to En$	$S.val = E.val$	synthesized
$F \to (E)$	$F.val = E.val$	synthesized
$F \to digit$	$F.val = digit.lexval$	synthesized

ex
- consider the following production
- draw the annotated parse tree
- for the expression 2 *3

productions
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to digit$



	Semantic rule	production
$F.val = 2$ is copied to $T'.inh$	$T'.inh = F.val$	$T \to FT'$
$T'.inh * F.val$ is copied to $T_1'.inh$	$T_1'.inh = T'.inh * F.val$	$T' \to *FT'$
$T'.inh$ is copied to $T'.syn$	$T'.syn = T_1'.inh$	$T' \to \varepsilon$
$T'.syn$ is moved to its parent	$T'.syn = T_1'.syn$	$T' \to *FT'$
$T'.syn$ is moved to its parent T	$T.val = T'.syn$	$T \to FT'$

- production & their respective rules

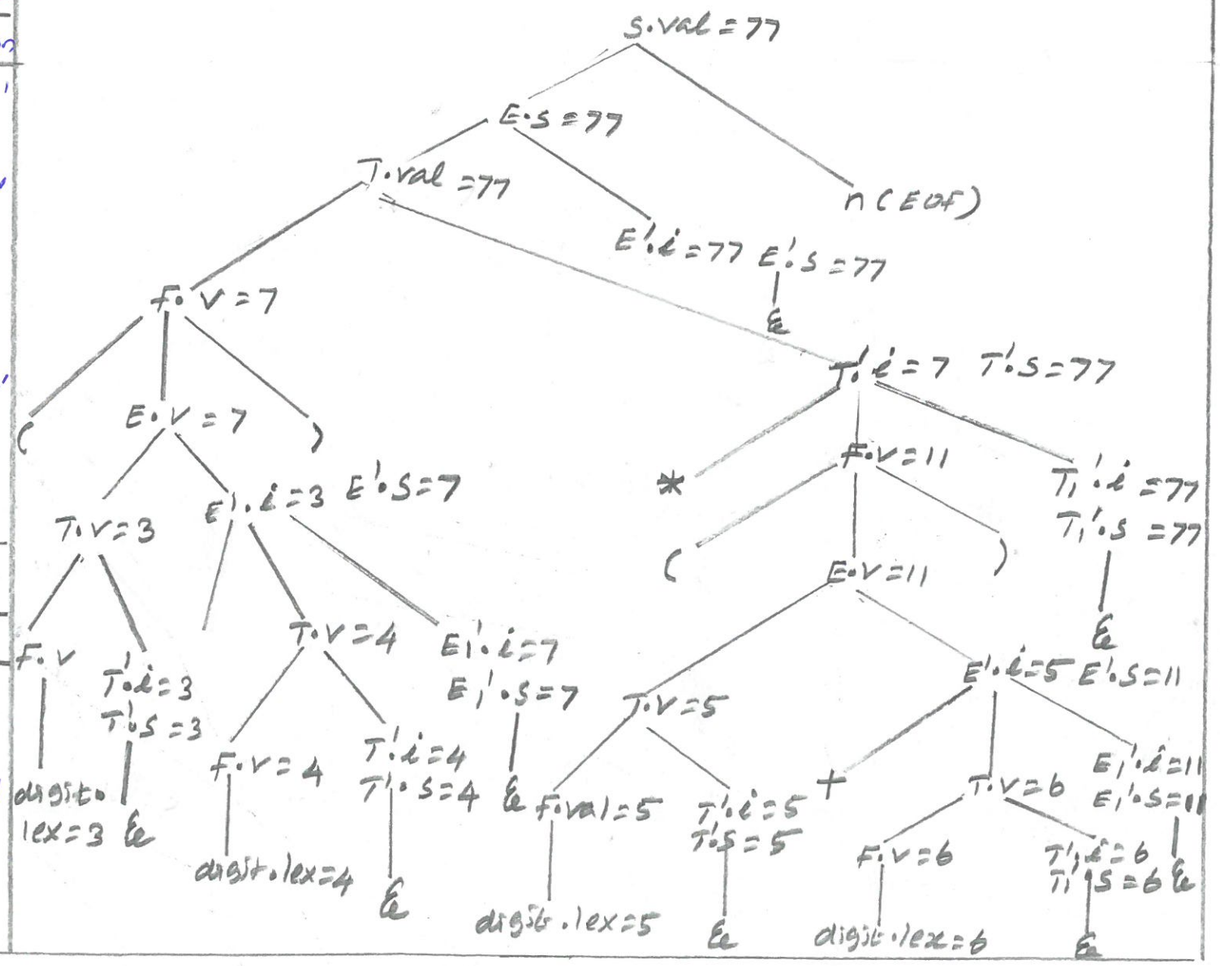
production	Semantic rules
$T \to FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \to *FT'$	$T_1'.inh = T'.inh * F.val$ $T'.syn = T_1'.syn$
$T' \to \varepsilon$	$T'.syn = T_1'.inh$

- semantic rules for other production

production	Semantic rules
$E \to TE'$	$E'.inh = T.val$ $E.val = E'.syn$
$E' \to +TE'$	$E_1'.inh = E'.inh + T.val$ $E'.syn = E_1'.syn$
$E' \to \varepsilon$	$E'.syn = E_1'.inh$

- Final SDD

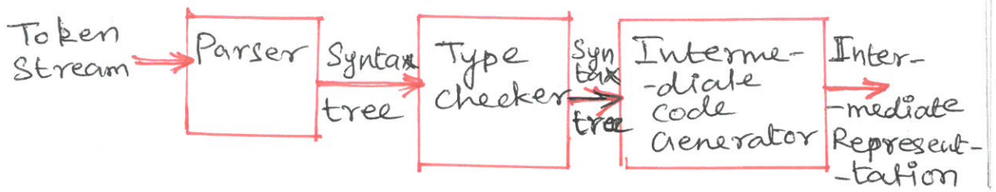
production	Semantic rule	Type
$S \to En$	$S.val = E.val$	Synthesized
$E \to TE'$	$E'.inh = T.val$ $E.val = E'.syn$	Inherited synthesized
$E' \to +TE'$	$E_1'.inh = E'.inh + T.val$ $E'.syn = E_1'.syn$	Inherited synthesized
$E' \to \varepsilon$	$E'.syn = E_1'.inh$	Inherited
$T \to FT'$	$T'.inh = F.val$ $T.val = T'.syn$	Inherited synthesized
$T' \to *FT'$	$T_1'.inh = T'.inh * F.val$ $T'.syn = T_1'.syn$	Inherited synthesized
$T' \to \varepsilon$	$T'.syn = T_1'.inh$	synthesized
$F \to (E)$	$F.val = E.val$	synthesized
$F \to digit$	$F.val = digit.lexval$	synthesized



TYPE SYSTEM – SPECIFICATION OF SIMPLE TYPE CHECKER, EQUIVALENCE OF TYPE EXPRESSION & TYPE CONVERSION

TYPE CHECKING:

- Verifies the type of construct matches that expected by its context.



Two Types of checking

Static checking

Dynamic checking

STATIC checking:

- Compiler checks Source program during compile time
- Verifies Syntactic and Semantic conversions

DYNAMIC CHECKING:

- Compiler checks target program during execution (runtime)

TYPE SYSTEM

Design of a type checker based on ⊛ Syntactic Constructs ⊛ Notion of types ⊛ Rules for assigning types

Type Expressions:

- Type of language construct is called as "type expression".

Types of Type Expressions

Basic Type (Ex: Boolean, char integer and real)

Type Constructor (Ex: Arrays, Records, Pointer Functions)

IMPLEMENTATION OF A SIMPLE TYPE CHECKER

→ Implementation of type checker by Syntax Translation Scheme

→ It synthesizes the type of each expressions from types of its subexpressions

Grammar (CFG) for programms

P → D ; E
D → D ; D | id : T
T → char | integer | array[num] of T | ↑T
E → literal | num | id | E mod E | E[E] | E↑

where P – program statement
D – Sequence of declarations
E – Expressions

Translation Scheme for type of an Identifier

P → D ; E
D → D ; D
D → id : T {addtype(id.entry, T.type)}
T → char {T.type = char}
T → integer {T.type = integer}
T → ↑T1 {T.type = pointer(T1.type)}

Translation Scheme for Expressions

E → literal {E.type = char}
E → num {E.type = integer}
E → id {E.type := lookup(id.entry)}
E → E1 mod E2 {E.type = if E1.type = integer and E2.type = integer then integer else type = error}
E → E1 [E2] {E.type = if E2.type = integer and E1.type = array(S,t) then t else type = error}
E → E1↑ {E.type = if E1.type = pointer(t) then t else type = error}

⊛ Function lookup used to fetch the type saved in Symbol-table entry

Translation scheme for statements (Flow of Control)

S → id := E {S.type := if id.type = E.type then void else type-error}
S → if E then S1 {S.type := if E.type = boolean then S1.type else S1.type}
S → while E do S1 {S.type := if E.type = boolean then S1.type else type-error}

EQUIVALENCE OF TYPE EXPRESSION

(1) **Structural Equivalence:**
Two type expressions are same if they have made up of same basic types and Constructs

(2) **Name Equivalence:**
Two type expressions are same if their constituents have the same Names

TYPE CONVERSION:

→ A data type is automatically converted into another datatype at compile time

→ It happens when destination data type is smaller than source datatype

TYPE CASTING:

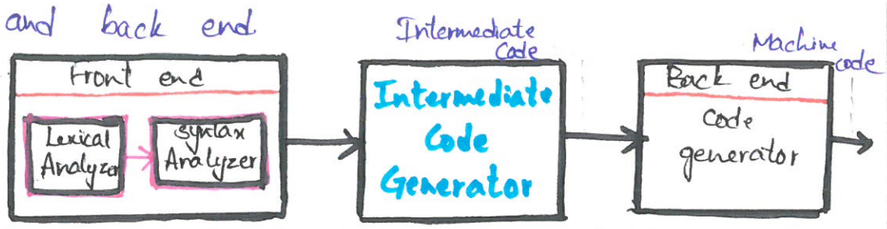
→ A datatype converted into another datatype by programmer or user in program code

→ Programmer manually convert one datatype into another

Destination_datatype = (target_datatype) variable (datatype)

Intermediate Code Generation:

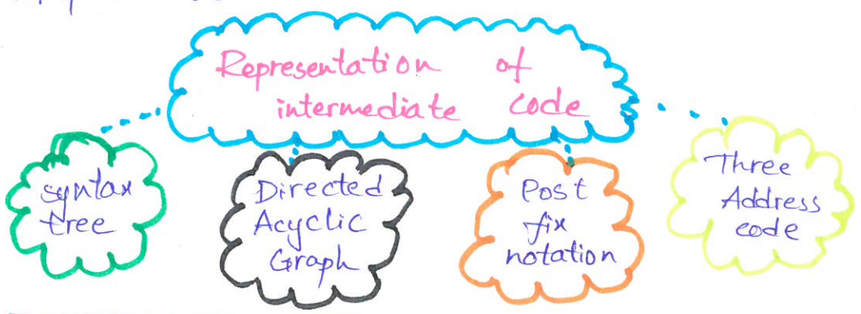
ICG translate the source program into intermediate code. It lies b/w front end and back end.



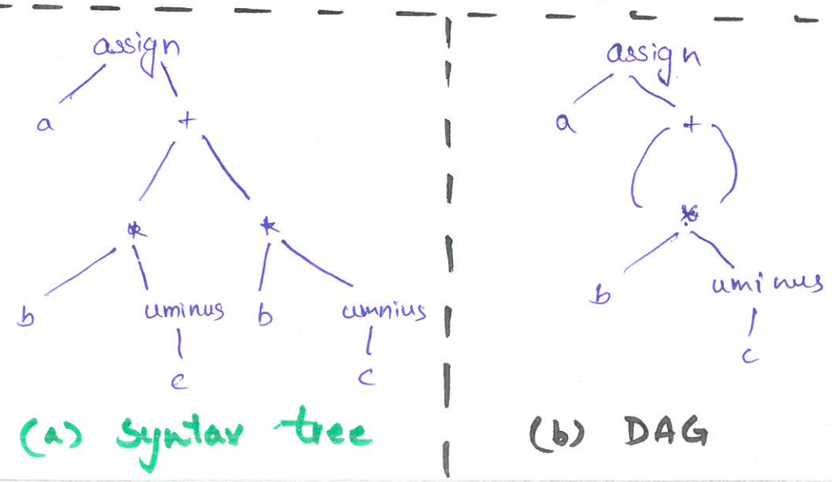
Position of Intermediate Code Gen:

Benefits:

* Retargeting is facilitated
* Machine - independent code optimizer can be applied to intermediate representation



Ex: a := b * -c + b * -c



(a) Syntax tree (b) DAG

Post Fix notation:

Nodes appear immediately after its children.

abc uminus * bc uminus * + assign

Three address code:

These are statement of form $c = a\ op\ b$, where a, b, - operands c - Result, so three address.

Three address code for syntax tree	Three address code for DAG
$t_1 := -c$	$t_1 := -c$
$t_2 := b * t_1$	$t_2 := b * t_1$
$t_3 := -c$	$t_3 := t_2 + t_2$
$t_4 := b * t_3$	$a := t_3$
$t_5 := t_2 + t_4$	
$a := t_5$	

Types of 3-Address Statement:

1. Arithmetic or logical:
   x := y op z
2. Unary operation:
   x := op y
3. Copy statement:
   x := y
4. Unconditional Jump:
   Go to L
5. Conditional Jump:
   if x relop y goto L
6. Param x & Call p,n.
   Param x_1
   Param x_2
   ⋮
   Param x_2
   Call p, n
7. Indexed:
   x := y[i]
8. Address & pointer:
   x := &y
   x := *y

Ex:

Implementation of 3 Address code:

Quadruples, Triples, Indirect Triples

Quadruples

	OP	arg (1)	arg (2)	Result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Triples

	op	arg 1	arg 2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Indirect Triples

	statement		op	arg 1	arg 2
(0)	(14)	(14)	uminus	c	
(1)	(15)	(15)	*	b	(14)
(2)	(16)	(16)	uminus	c	
(3)	(17)	(17)	*	b	(16)
(4)	(18)	(18)	+	(15)	(17)
(5)	(19)	(19)	assign	a	(18)

Declaration :-

A variable or procedure need to be declared before it is used declaration involves allocation of space in memory and entry of type and name is symbol table.

Declaration in a procedure :

offset :- It is a global variable, which keep track the next available relative address.

Translation scheme :

* Non terminal P generates sequence of declaration in the form id:T.
* Initially offset is set to 0. As soon as name is entered in symbol table the offset is incremented by the width of data object.

Procedure syntax :

enter (name, type, offset)

* TYPE is represented as integer, real, pointer and arrays.
* width of each pointer is assumed to be 4.

Computing the type and relative addresses of declared names

P → D {offset := 0}

D → D ; D

D → id : T {enter(id.name, T.type, offset); offset := offset + T.width}

T → integer {T.type := integer; T.width := 4}

T → real {T.type := real; T.width := 8}

T → array [num] of T, {T.type := array (num.val, T1.type); T.width := num.val × T1.width}

T → ↑T1 {T.type := pointer (T1.type); T.width := 4}

D → D

D → D ; D | id : T | proc id ; D ; S

D → proc id D1 S

Semantic Rules :

1. mktable (previous) – Creates a new symbol table and returns a pointer to the new table.
2. enter (table, name, type, offset) – Creates new entry for the name in symbol table.
3. add width (table, width) – records cumulative width & all entires in table in header.
4. enterproc (table, name, newtable) – Creates new entry for procedure name in the symbol, table pointed to by table :

Syntax directed Translation Scheme for Nested Procedures :

P → MD {add width (top (tblptv), top(offset); pop (tblptr); pop (offset)

M → ε {t := mk table (NIL); push (t, tbl. ptr); push (0, offset)}

D → D1 ; D2

D → proc id; {t := top (tbl.ptr); ND1 ; S addwidth (t, top (offset)); pop (tblptr); pop (offset); enterproc (top(tblptv); id.name, b).

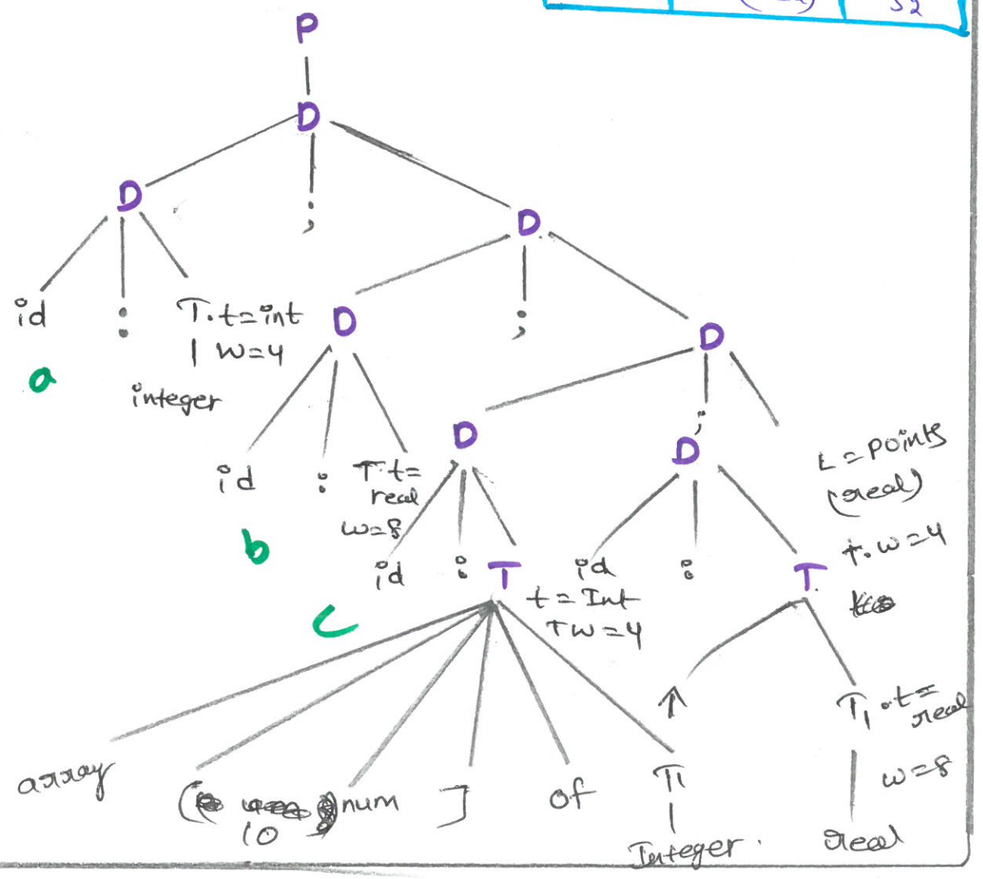
D → id : T {enter (top (tblpbr), id.name ; T.type, top (offset); top(offset) ; = top(offset) + T.width }

N → ε {t := mktable (top (tblptr)) ; push (t, tblptr); push (0, offset)}

Ex :-

```
{ a : integer
  b : real
  c : arrage [10] of integer
  d : ↑ real
}
offset = 0
```

Name	Type	offset
a	integer	0
b	real	4
c	arr (10) integer	12
d	pro ↑ (real)	52



Assignment Statement & Boolean Expression

Topic : Assignment & Boolean

Intermediate code Generation

SDT for Assignment statement :

Two attribute :

E.place - Hold the value of E

E.Code - Sequence of three address st

gen() - Function used to produce Sequence of 3 address st

S → id := E	{S.code := E.code || gen (id.place := E.place)}
E → E1 + E2	{E.place := new tmp . E.code = E1.code || E2.code || gen (E.place := E1.place + E2.place)}
E → E1 * E2	{E.place := newtmp E.code = E1.code || E2.code || gen (E.place := E1.place * E2.place)}
E → - E1	{E.place := newtmp E.code := E1.code || gen (E.place := - E1.place)}
E → (E1)	{E.place := E1.place E.code := E1.code}
E → id	{E.place := id.place E.code := ''}

SDT for Boolean Expression

* used to compute logical value
* change flow of control

E → E or E / E and E / not E / (E) / id relop id / true / false

Logical :

E → E1 or E2	E.place := newtemp emit (E.place ':=' E1.place 'or' E2.place)
E → E1 and E2	E.place := newtemp emit (E.place ':=' E1.place 'and' E2.place)
E → not E1	E.place := newtemp emit (E.place ':=' 'not' E1.place.
E → (E1)	E.place := E1.place
E → id1 relop id2	E.place := new temp emit (if id1.place relop id2.Place goto next stat +3) emit (E.place = 0) emit (goto next stat +2) emit (E.place = 1)
E → True	E.place := newtemp emit (E.place = '1')
E → false	E.place := newtemp emit (E.place = 0)

Flow of control statement

S → if E then S1 / if E then S1 else S2 / while E do S1

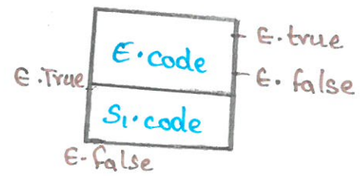
SDT :

S → if E then S1

E.true = new label

E.false = S.next

S1.next = S.next



S.code = E.code || gen(E.true ':') || S1.code

S → if E then S1 else S2

E.true = new label

E.false = newlabel

S1.next = S.next

S2.next = S.next

S.code = E.code = E.code || gen(E.true) || S1.code || gen (go to S.next) || gen (E.false ':') || S2.code

S → while E do S1

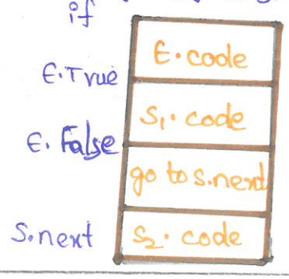
S.begin = new label

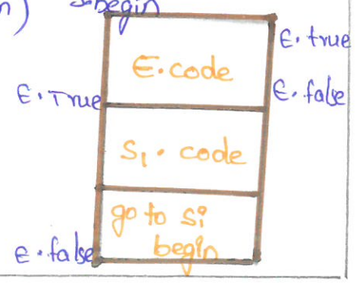
E.True = new label

E.false = S.next

S1.next = S.begin

S.code = gen (S.begin ':') || E.code || gen(E.true ':') || S1.code || gen (go to S.begin)





CASE STATEMENT, BACK PATCHING, PROCEDURE CALLS

* CASE STATEMENT :-

=> Switch statement allow a variable to be tested for equality against a list of value. Each value is called case.

Syntax :-

```
Switch expression.
begin.
    Case value : statement.
    :
    case value : Statement
    default    : statement.
    end.
```

Ex :- SDT of case statement.

```
switch E.
beign.
    case v1 : S1
    case v2 : S2
    :
    case vn-1 : Sn-1
    Default : Sn.
```

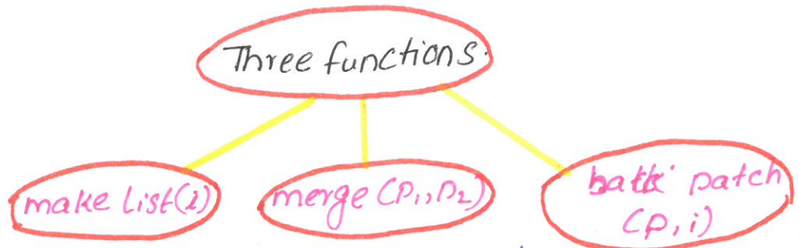
Intermediate code of case statement code to evaluate E into E.go to test.

```
L1 : code for S1
     goto next.
L2 : code for S2
     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 :
```

Back patching :-

-> process of fulfilling unspecified information

=> boolean expression and flow of control statement in one pass.



-> Synthesized attributes truelist and false list of non terminal E are used to generate Jumping code for boolen.

-> Attribute M.Quad record the number of the 1st statement.

{ m.Quad = next Quad }

next Quad - holds index of next quadraple.

Translation scheme :-

1) E -> E1 or M E2
{ back patch E1.false list, M.Quad);
E.true list := merge(E1.true list, E2.true list)
E.false list := E2.false list }

2) E -> not E1
{ E.true list := E1.false list;
E.false list := E1.true list; }

3) E -> id1 relop id2
{ E.true list := make list (next Quad);
E.false list := make list (next Quad+1);
emit('if' id1.place relop.op id2.place 'go to'_)
emit ('goto _') }.

4) E -> E1 and ME2
{ back patch (E1.true list, M.Quad);
E.true list := E2.true list;
E.False list := Merge (E1.false list, E2.false list)}.

5) E -> (E1) { E.true list := E1.true list;
E.false list := E1.false list; }

6) E -> true { E.true list := make list (next Quad);
emit ('goto _')}

7) E -> false { E.false list := make list (next Quad);
emit ('goto _')}.

8) M -> E { M.Quad := next Quad }

PROCEDURE CALLS :-

-> when a procedure calls occurs, space must be allocated for the activation record of the called procedure.

1) S -> call id (E list)
{ for each item p on queue do
emit ('param' p);
emit ('call' id.place)}.

2) Elist -> Elist E.
{ append E.place to the end of Queue}

3) E list -> E
{ initialize Queue to contain only E.place}

Elist : code for s is the code for E list

param : Elist followed by param P Statement.

call : followed by a call statement.

PRINCIPLE SOURCE OF OPTIMIZATION

Principle source of optimization:-

- Improve code
- Consume less resources
- deliver high speed

Types:-

Machine Independent:

- does not involve CPU registers
- absolute memory locations.

```
do
{
item = 10;
value = value + item;
} while (value < 100);
```

- repeated assignment item

```
item = 10;
do
{
value = value + item;
} while (value < 100);
```

Machine dependent:

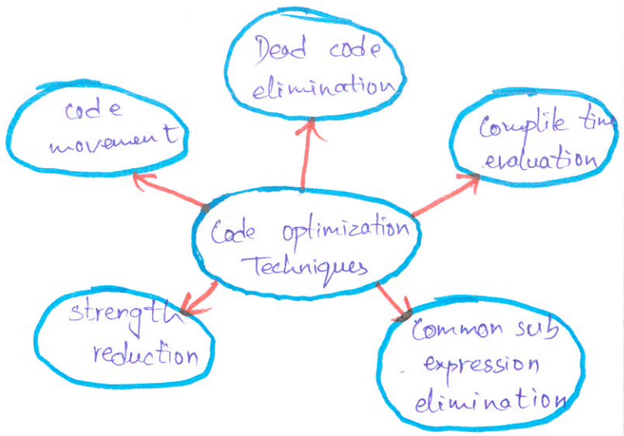
- After the target-code → generated
- Code transformed → according to the target machine
- Involves CPU registers.

Code optimization:

→ Eliminating → unwanted code,
→ Rearranging → statements.

Advantages:

→ Faster execution
→ Utilizes the memory efficiently



1. Compile Time Evaluation:

Constant Folding:

Length = (22/7) * d

⇓

length = 3.14 * d

Constant propagation:

p = 3.14 r = 5

Area = pi * r * r

Area = 3.14 * 5 * 5

2. Common sub-expression elimination:

Code before optimization	Code After Optimization
$S_1 = 4 * i$	$S_1 = 4 * i$
$S_2 = a[S_1]$	$S_2 = a[S_1]$
$S_3 = 4 * j$	$S_3 = 4 * j$
$S_4 = 4 * i$ (redundant)	$S_5 = n$
$S_5 = n$	$S_6 = b[S_1] + S_5$
$S_6 = b[S_4] + S_5$	

3. Code movement:

Code before optimization	Code after optimization
for (int j = 0; j < n; j++) { x = y + z; a[j] = b * j; }	x = y + z; for (int j = 0; j < n; j++) { a[j] = b * j; }

4. Dead code Elimination:

Code before optimization	Code after optimization
i = 0; if (i == 1) { a = x + 5; }	i = 0;

5. Strength reduction:

Code before optimization	Code after optimization
B = A * 2	B = A + A

Loop optimization:-

- Decrease ← inner loop instructions
- running time improved.

1. Code motion:

```
for (i=0; i<n; i++)
{
  x = y + 3;
  a[i] = 6 * i;
}
```

⇒

```
x = y + 3;
for (i=0; i<n; i++)
{
  a[i] = 6 * i;
}
```

2. Induction Variable elimination:

```
i = 1;
while (i < 10)
{
  y = i * 4;
  i = i + 1;
}
```

⇒

```
t = 4
while (t < 40)
{
  y = t
  t = t + 4;
}
```

3. Loop unrolling:

```
for (i=0; i<100; i++)
{
  display ();
}
```

⇒

```
for (i=0; i<50; i++)
{
  display ();
  display ();
}
```

4. Loop jamming:

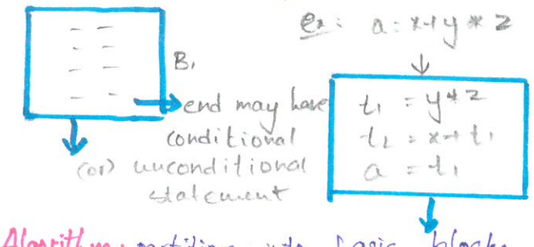
```
for (i=0; i<100; i++)
  a[i] = 1;
for (i=0; i<100; i++)
  b[i] = 2;
```

⇒

```
for (i=0; i<100; i++)
{
  a[i] = 1;
  b[i] = 2;
}
```

Basic blocks & flow graph:

- sequence of three address statements
- do not have any jump statements



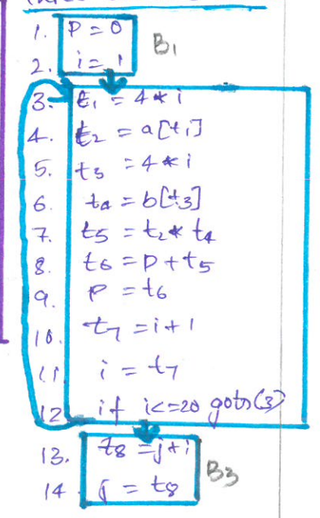
Algorithm: partition into basic blocks
Input: sequence of 3-address statement
Output: A list of basic blocks
Method:-
1. Determine → set of leaders

Rules:
(i) First statement → leader
(ii) Target of conditional (or) unconditional goto → leader.
(iii) Statement that immediately follows goto (or) unconditional goto → leader.

ex: Fragment of source code:

```
begin
  P := 0;
  i = 1;
  do begin
  P = P + a[i] * b[i];
  i = i + 1;
  end
  while i <= 20;
  j = j + 1;
  end
```

Three address code:



leader → 1, 3, 13

Flow graph:

- how the program control passed among the blocks
- Directed graphs
- Nodes → Basic blocks
- Edges → flow of control

* PEEPHOLE OPTIMIZATION *

Peephole optimization :-

* A simple and effective technique for locally improving target code. kind of optimization Performed over a very small set of instruction in a segment of generated code. 'peephole' also called 'window'

The characteristics of peephole optimization :-

1. Redundant - instructions elimination
2. Flow of control optimizations
3. Algebraic simplifications
4. use of machine idioms
5. Unreachable code
6. Reduction in strength.

1. Redudant instructions elimination

Eg: MOV R0,a (1)
MOV a,R0 (2)

2. Flow of control optimizations :-

* unnessary Jump can be eliminated in either intermediate code or target code.
* Replace Jump sequences.

Ex: go to L_1 ... L_1 : goto L_2 → Replaced by → goto L_2 ... L_2

Ex_2: go to L_1 ... L_1 : if a<b goto L_2 ... L_3 : → if a<b goto L_2 goto L_3 ... L_3 :

* Skip the unconditional Jump and reduce execution time.

3. Algebraic specification :-

peephole optimization suits well for algebraic specification.

Ex
```
X := X+0
 (or)
X := X * 1
```

* Produce straight forward intermediate code generation

4. Use of machine Idioms :-

Target machine have hardware instructions to implement certain specific operations efficiently.

Ex: Auto Increment, Auto Decrement

* Greatly improves the Quality of code by pushing and popping in a state.

Ex
```
i := i+1 ⇒ i++
i := i-1 ⇒ i--
```

5. unreachable code :-

Peephole optimization remove unreachable instruction.

```
# desine debug 0'
  :
if (debug){
print Debugging Information
}.
```

For if - statement :-

```
if debug = 1 goto L1
   go to L2
L1 : print debugging Information
L2 :
```

* peephole optimization is to eliminate Jump over Jump.

6. Reduction in strength :-

* Replace expensive operations by equivalent cheapers ones of the target machine.

Ex $X^2 := X * X$

Instead of power function to use multiplication

Application :

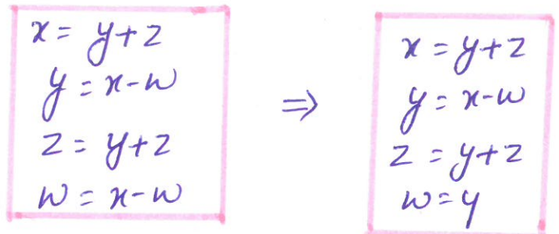
- Optimization of compiles Architectures
- Software productivity tools

Optimization of Basic Blocks, Basic Block & flow Graph, Global Data flow Analysis.

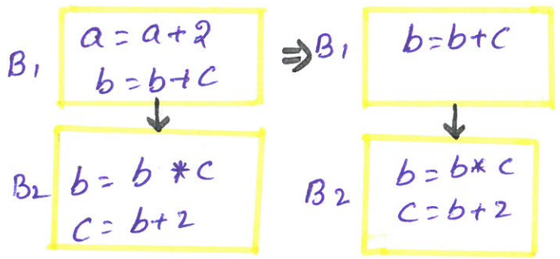
Optimization of Basic Blocks

⇒ Structure preserving transformation.

1. Common subexpression elimination.

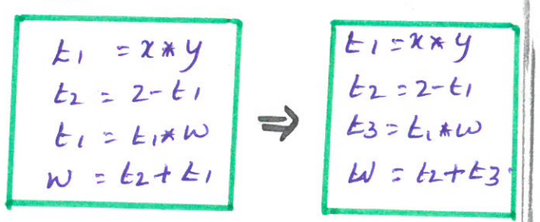


2. Dead code Elimination.



'a' - Dead code.

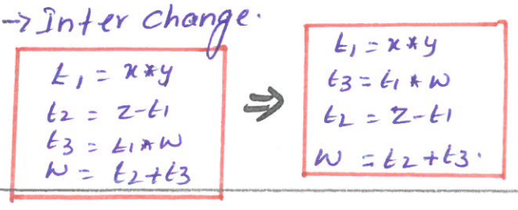
3. Remaining of Temporary variable.



4. Interchange of

→ Two statments are not independant

→ Inter change.



Algebraic Transformations:-

x = x+0
x = x*1
→ value not changed.
→ remove the statement.

c = d**2 ⇒ pow(d,2).
⇓
c = d*d.

Introduction to Global Data Flow Analysis.

→ To do code optimization and code generation.

compiler - collect information about whole program

- Distribute it to each blow in the folow graph.

Data flow equation.

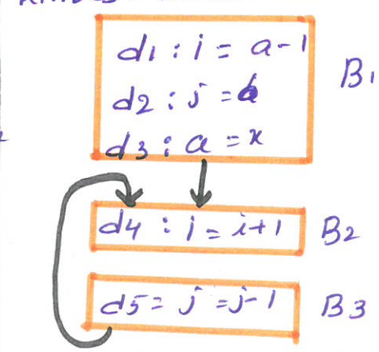
out[s] = gen[s] ∪ (in[s] - kill[s])

out[s] = Information at end of s.

gen[s] = Information generated by s

in[s] = Information enters at the begining of s.

kill[s] = Information killed by s.



Points and paths:-

- within B, there is a point blw 2 adjacent statement.

B1 → 4 points.

Path :-

- A path from p_1 to p_n is a sequence of points $p_1, p_2 \cdots p_n$.
- such that for each i between 1 and n-1 either.

(i) p_i - point immediately preceding a statement.

p_{i+1} - point immediately following that statement in same block

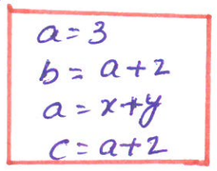
(ii) p_i = End of some block.

p_{i+1} - begining of a successor block

Ex:- path from begining of B_3 to begining of B_2.

Reaching Definition:

- A definition of variable x is a statement that assigns a value of x
- A definition of d reaches a point P, if there is a path from d to P,
- such that d is not 'killed' along the path.

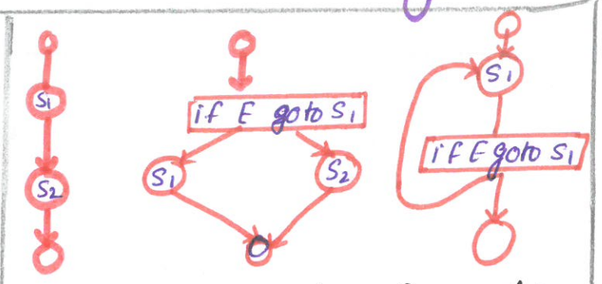


- 'a' - killed ⇒ if between 2 points along path ⇒ definition of 'a'

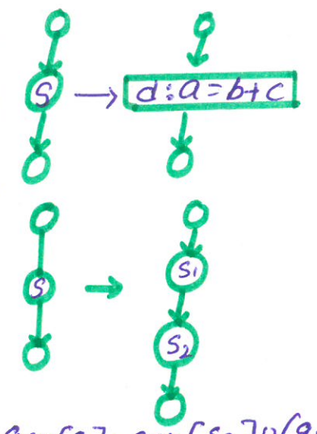
Data flow Analysis of structured program.

S → id := E / S ; S / if E then S else S / do S while E

E → id + id / id.



Data flow Equations for reaching Definitions:-



gen[s] = {d}

kill[s] = Da - {d}

out[s] = gen[s] ∪ (in[s] - kill[s])

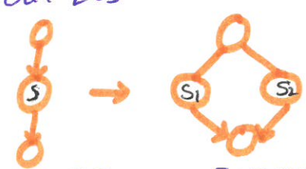
gen[s] = gen[s2] ∪ (gen[s1] - kill[s2])

kill[s] = kill[s2] ∪ (kill[s1] - gen[s2])

in[s1] = in[s]

in[s2] = out[s1]

out[s] = out[s2]



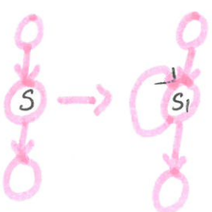
gen[s] = gen[s1] ∪ gen[s2]

kill[s] = kill[s1] ∩ kill[s2]

in[s1] = in[s]

in[s2] = in[s]

out[s] = out[s1] ∪ out[s2].



gen[s] = gen[s1]

kill[s] = kill[s1]

in[s1] = in[s] ∪ gen[s]

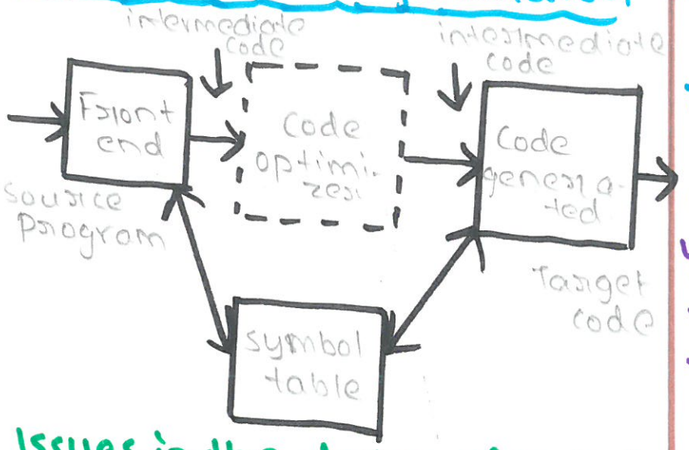
out[s] = out[s].

Code Generation - Issues in the design of code Generator, Target Machine

CODE GENERATION :-

Final Phase - Code generator.

Position of Code generator



Issues in the design of a code generator :-

1. Input to code generator
Intermediate representation - choices.
   a) postfix notation
   b) syntax tree (or) DAG
   c) Three address code
      - Quadruples
      - Triples
      - Indirect triples

Free from all errors

2. Target program :-
   1. Absolute machine language
   - Fixed memory location
   - CPU access faster
   (ii) Relocatable machine language
   - subprograms & subroutines compiled seperately
   - Relocatable object modules linked together
   - loaded → linking loader.
   (iii) Assembly language :-
   - Generate - code easily

Machine Architecture RISC CISC

3. Memory Management :-
Symbol table ← Memory management.

4. Instruction selection :-
- Quality of generated code
- speed and size

```
a : b+c        MOV b,R0        b → R0
d : a+e        ADD c,R0        R0 → c+R0
               MOV R0,a        R0 → a
               MOV a,R0        a → R0
               ADD e,R0        R0 → e+R0
               MOV R0,d        R0 → d
```

⇓

```
MOV b,R0
ADD c,R0
ADD e,R0
MOV R0,d
```

Instruction speed machine idioms type of code Architecture quality

```
a := a+1
MOV a,R0
ADD #1,R0      ⇒ INC a
MOV R0,a
```

5. Register allocation
- efficient utilization of registers.

i) Register allocation

R0	R1
a	b

- Register Assignment

a → R0 b → R0

Process of storing a value

Ex :- `t = x+y` `t = t*z` `t = t/w`

```
MOV x,R0
ADD y,R0
MUL z,R0
DIV w,R0
MOV R0,t
```

6. Evaluation order
- efficiency of the target code based on sequence of execution.

The Target Machine :-
Two-address instruction of the form.

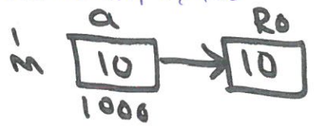
op source, destination

MOV (Move content of source to destination)
ADD (add content of source to destination)
SUB (subtract Content of Source from destination)

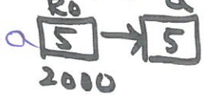
Addressing Modes

Mode	FORM	Address	Added cost
1. Absolute	M	M	1
2. Register	R	R	0
3. Indexed	c(R)	c+ content (R)	1
4. Indirect register	*R	contents (R)	0
5. Indirect Indexed	*c(R)	content (c+content (R))	1
6. Immediate (or) literal	#c	MA	1

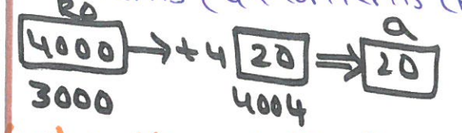
(i) Absolute : MOV a,R0
contents (a) ⇒ R0



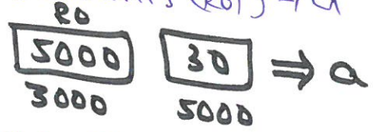
(ii) Register MOV R0,a
contents (R0) ⇒ a



(iii) Indexed :- MOV 4(R0),a
contents (4+ contents (R0)) ⇒ a



(iv) Indirect Register :-
MOV *R0, a contents (contents (R0)) ⇒ a



(v) Indirect Indexed :-
MOV *4(R0),a contents (contents (4+ contents (R0))) ⇒ a

(vi) Immediate (or) literal :-
MOV #4,R0

Cost of Instruction =

1 + cost (source-mode) + cost (destination mode)

Ex : MOV R0,R1
cost = 1+0+0 = 1
MOV R0,M
cost = 1+0+1 = 2
ADD #1,R0
cost = 1+1+0 = 2
SUB 2(R0), *8(R1)
cost = 1+1+1 = 3.

RUNTIME STORAGE MANAGEMENT

RUNTIME STORAGE MANAGEMENT

ACTIVATION RECORD :-

Information needed in execution of a procedure is kept in Block of storage.

Storage Allocation Schemes:-

1. Static Allocation - allocate storage in compile time.
2. Stack allocation - allocate stack storage at runtime.
3. Heap allocation - allocates & deallocate storage at runtime in Heap storage

① Static Allocation:-

The position of an activation record is fixed in memory at compile time

Three address statements:-

Call	Return	Halt	Action.

Runtime Memory is divide into area

Code | static is data | stack

1. Implementation of call statement :-

MOV # here +20, callee.static area
GOTO callee.code-area.

Callee.static - add of AR
Callee.code - first instruct for called procedure
Here +20 - Return address

2. Implementation of Return st:-

GOTO * Callee.static-area.

* Transfer the control to the add at beginning of the activation record.

3. Implementation of action st.

ACTION - used to implement action st.

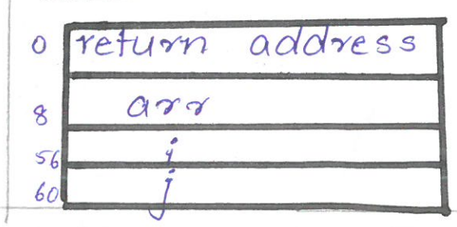
4. Implementation of Halt st:-

HALT - Final instruction - return the control to the Operating system.

Ex:

```
/* Code for c */
action 1
call p
action 2
halt

/* code for p */
action 3
return
```



Ex:

```
100 : ACTION 1 /* code for c */
120 : mov # 140, 364
132 : GOTO 200
140 : ACTION 2
160 : HALT
      :
200 : ACTION 3 /* code for p */
220 : GOTO *364
      :
300:                  /* Activation record for c */
304 :
      :
364: 140              /* Activation record for p */
368 :
```

STACK Allocation:-

- Register store position of AR
- Offsets from the value in Reg.

1. Initalization of stack:

MOV # stackstart, sp
HALT

2. Implement of call Statement:

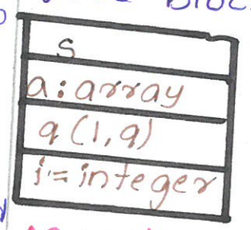
ADD # caller, records size, sp
MOV # here +16, * sp
GOTO callee.code-area.

3. Implementation as Return st:

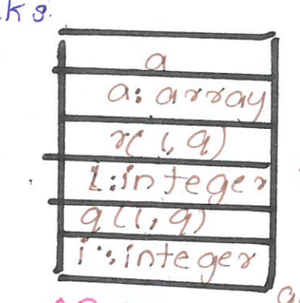
GOTO *0 (SP)
SUB # caller.record size, SP

③ Heap allocation:-

- Parcels out pieces of storage
- Deallocated when not need
- Record is retained when the procedure ends
- Linked List to maintain free blocks.



AR on stacks



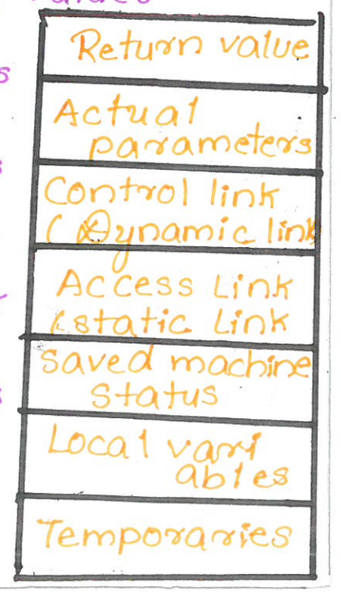
AR on Heap

Activation tree

Activation Record

1. Feelds of activation record

1. Temporary values
2. Local variables
3. Saved machine registers
4. Control link
5. Access link
6. Actual Parameters
7. Return values.



A Simple Code Generator

Simple Code Generation

Final Activity of Compiler

Creates Assembly Language

Properties:

* Correctness.
* High Quality.
* Efficient use of resources
* Quick Code generation.

Function & use:

Uses descriptors to keep track of register Content & addresses for Name.

Register (R) Descriptor	Address (s) Descriptor

Information about values & Data.

Information about memory or locations

Status	Operand Descriptors

Register Descriptor

Attributes	Addressing mode	Storage location

Address Descriptor.

Algorithm for Code Generation

```
Gen_Code ( operator, operand 1, operand 2 )
{
 if (operand 1, address mode = 'R')
  {
  if (operator = '+')
  Generate ('ADD operand 2, R0');
  else if (operator = '-')
    Generate ('SUB operand 2, R0');
  else if (operator = '*')
    Generate ('MUL operand 2, R0');
  else if (operator = '/')
    Generate ('DIV operand 2, R0')
  }
 else if (operand 2, address mode = 'R')
  {
  if (operator = '+')
  Generate ('ADD operand 1, R0');
  else if (operator = '-')
  Generate ('SUB operand 1, R0');
  else if (operator = '*')
  Generate ('MUL operand 1, R0');
  else if (operator = '/')
  Generate ('DIV operand 1, R0');
 } else
  {
  Generate ('Mov operand 2, R0');
    if (operator = '+')
  Generate ('ADD operand 2, R0');
  else if (operator = '-')
    Generate ('SUB operand 2, R0');
  else if (operator = '*')
    Generate ('MUL operand 2, R0');
  else if (operator = '/')
  Generate ('DIV operand 2, R0');
```

① D : (a-b) * (a-c) + (a-c)

Sol: Three address code

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

Three address code	Target code sequence	Register Descriptor	Operand Descriptor
$t_1 = a - b$	Mov a, R_0 SUB b, R_0	R_0 - a R_0 - t_1	t_1, R, R_0
$t_2 = a - c$	Mov a, R_1 SUB c, R_1	R_1 - a R_1 - t_2	t_2, R, R_1
$t_3 = t_1 * t_2$	MUL R_1, R_0	R_0 - t_3	t_3, R, R_0
$t_4 = t_3 + t_2$	ADD R_1, R_0	R_0 - t_4	t_4, R, R_0
$D = t_4$	STORE t_4, D	D - t_4	t_4, R, R_0

Next Use Information

Eliminate Dead Code and register allocation

→ Indicate a variable in the current position will be reused.

Input: Block of Three address statemen

Output: Livelinees & Next - use.

Method:

1. Attach to statement i information regarding the next use & livliness of variables.
2. In Symbol table, Set Variable to liveness & Next use for LHS.
3. Repeat the liveness & use for RHS.

Statement:

i : x = y op z

Symbol table.

Name	Liveliness	Next-use
x	not live	No Next-use
y	live	i
z	live	i

Directed Acyclic Graphs [DAGS] :-

- DAG - Directed Acyclic Graph is a special kind of Abstract syntax Tree.
- Each node contains unique value.
- Not contain any cycles - called Acyclic.

Algorithm :-

Method :-

Step 1 :- if y is undefined then create node (Y)

If z is undefined, create node (z) for case (i)

step 2 :- For case (i) - create a node (op) whose left child is node (y) and right child is node(z) node (y)

case (ii) - Determine whether there is node (op) with one child node (y). If not create such node.

case (iii) - node n will be node (y)

step 3 :-

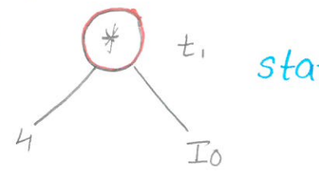
Delete x from the list of identifiers for node (x) Append x to the list of attached identifiers for the node n found in step 2 and set node (x) to n

Ex: Consider the block of three address statement:

1. $t_1 := 4 * i$
2. $t_2 := a[t_1]$
3. $t_3 := 4 * i$
4. $t_4 := b[t_3]$
5. $t_5 := t_2 * t_4$
6. $t_6 := prod + t_5$
7. $prod := t_6$
8. $t_7 := i + 1$
9. $i := t_7$
10. if $i <= 20$ go to (1)

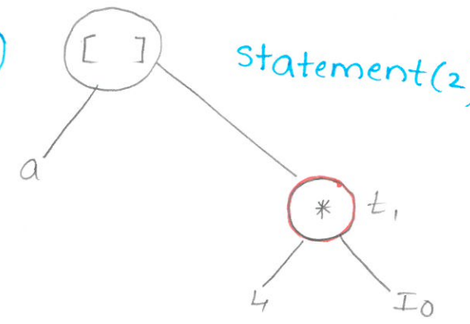
Stages in DAG Construction :-

(a)



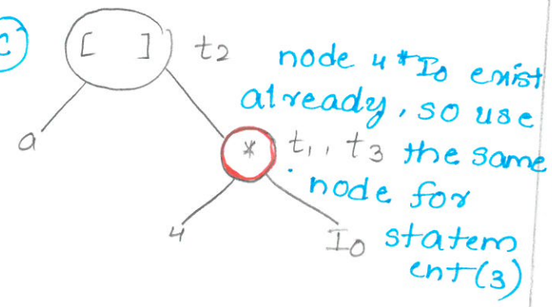
statement (1)

(b)



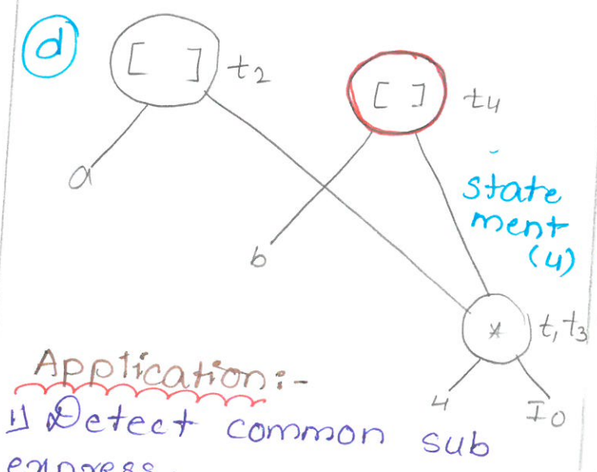
statement (2)

(c)



node 4 * I0 exist already, so use t1, t3 the same node for statement (3)

(d)

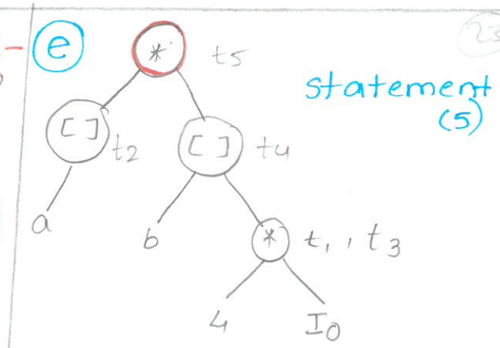


statement (4)

Application :-

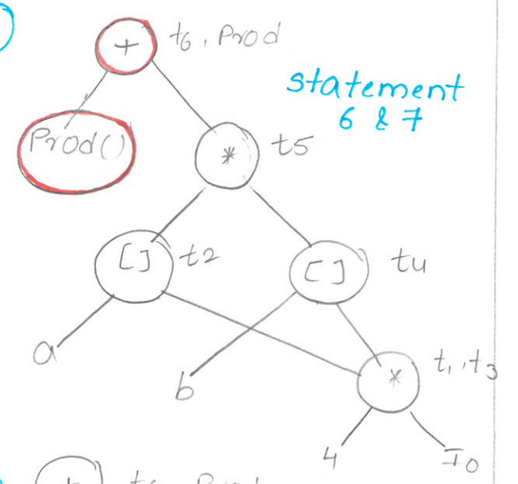
1) Detect common sub express.
2) Determine which identifier have their values
3) Determine compute values used outside the block

(e)



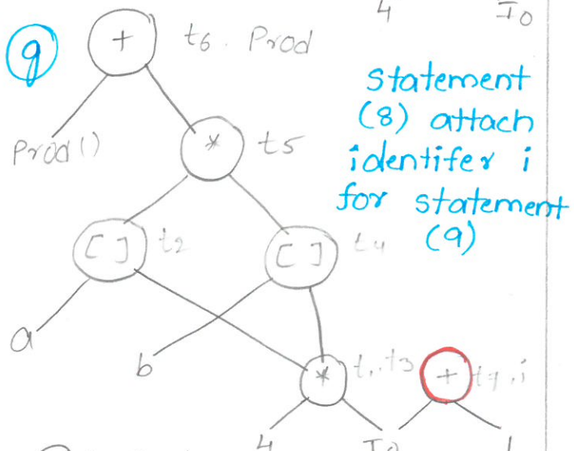
statement (5)

(f)



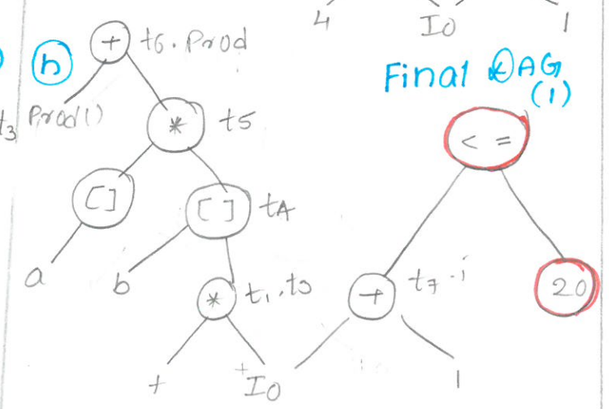
statement 6 & 7

(g)



statement (8) attach identifer i for statement (9)

(h)



Final DAG (1)

Illustrate the output of each phase of compilation of the input "a = (b+c) * (b+c) * 2" and explain in detail the process of compilation.

Lenical Analyzer

- Tokens and their types.

Symbol	category	Attribute
a	identifier	#1
=	operator	Assigment (1)
b	identifier	# 2
+	operator	Arithmatic (1)
c	identifier	#3
*	operator	Arithmatic (2)
(	operator	open Parenthsis (1)
)	operator	closed Parenthsis (1)
2	constant	# 4

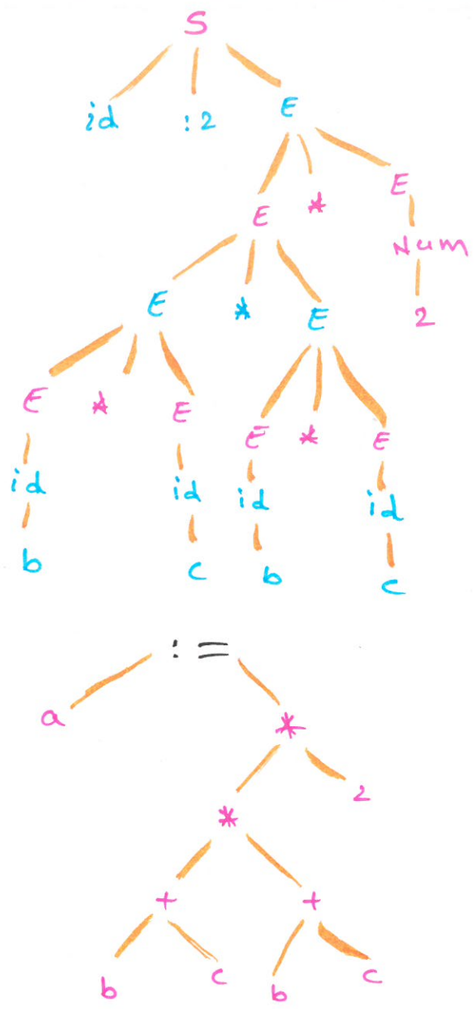
Syntax Analyzer

- Prase tree / syntax tree

S → id := E

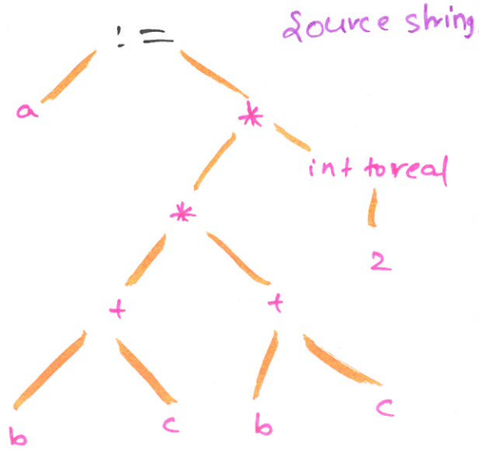
E → E+E | E*E | id | num

↳ content free grammar.



Semantic Analysis:

- To determine meaning of a source string.



Intermediate code generation:

- Three Address code.

T_1 = int to real (2)

T_2 = b+c

T_3 = b+c

T_4 = T_2 * T_3

T_5 = T_4 * T_1

a = T_5

code optimization

- Improve intermediate code
- Run faster, occupy less space.

T_1 = int to real (2)

T_2 = b+c

T_3 = T_2 * T_2

T_4 = T_3 * T_1

a = T_4

code generation:

→ Allocate storage & generate a relocata -ble m/L code.

```
MOV b, R2
ADD R2, c
MUL R2, R2
MUL R2, #2.0
MOV R2, a.
```

content of the symbol Table.

Addr	sym	Attribute	member
1	A	id, real	1000
2	B	id, real	1100
3	C	id, real	1110.

Content of literal Table

Address	literal	Attribute	Member
4	2	const, int	1200

Find the number of tokens in the following statement.

```
int x;
char * p = "SSE";
```

int - keyword
x - Identifier
; - special symbol
char - keyword
* - operator
p - Identifier
= - operator
"sse" - constant
; - special symbol.

} 9

Identify lenical Errors

```
int 1x;
int c = "Hello;
int d = hello";
```

1x - Not token, lenical error.

" - missing, will not generate token
↳ lenical error.

write the regular expression for the language containing the string in which every 0 is immediately followed by 11.

R = (011+1)*

Context Free Grammar, Derivation & Ambiguity

Context Free Grammar

1. $L = \{a^n / n \geq 0\}$

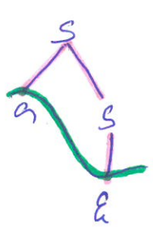
Soln

$L = \{\varepsilon, a, aa \text{ ------}\}$

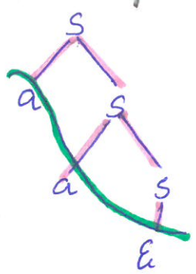
The grammar is

$S \rightarrow aS / \varepsilon$

repetition of 'a'



i/p string a



i/p string aa

2. $L = \{$any combination of a's and b's$\}$

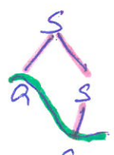
$R = (a+b)^*$

Soln

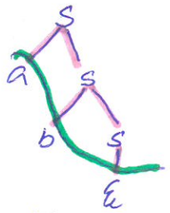
$L = \{\varepsilon, a, b, aa, ba, ab, bb \text{ ---}\}$

The grammar is

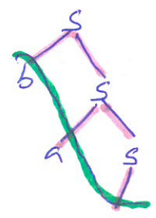
$S \rightarrow aS / bS / \varepsilon$



i/p string - a



i/p string - ab



i/p string - ba

3. Length of string is odd (over 'a' and 'b' input)

$\Sigma = \{a, b\}$

$R = [(a+b)(a+b)]^*(a+b)$

Soln

1. Generate grammar for smallest expression.
2. combine with starting symbol

$(a+b) \Rightarrow A \rightarrow a/b$

$(a+b)(a+b) \Rightarrow B \rightarrow AA$

A A

$[(a+b)(a+b)]^* \Rightarrow C \rightarrow BC/\varepsilon$ (repetition)

$(a+b)[(a+b)(a+b)]^*$

A C

Now The grammar is

$S \rightarrow AC$

$C \rightarrow BC/\varepsilon$

$B \rightarrow AA$

$A \rightarrow a/b$

Derivation

1. Design a derivation tree for the following grammar

$S \rightarrow aS$

$S \rightarrow aSbS$

$S \rightarrow \varepsilon$

Also obtain the leftmost and rightmost derivation for the string "aaabaab"

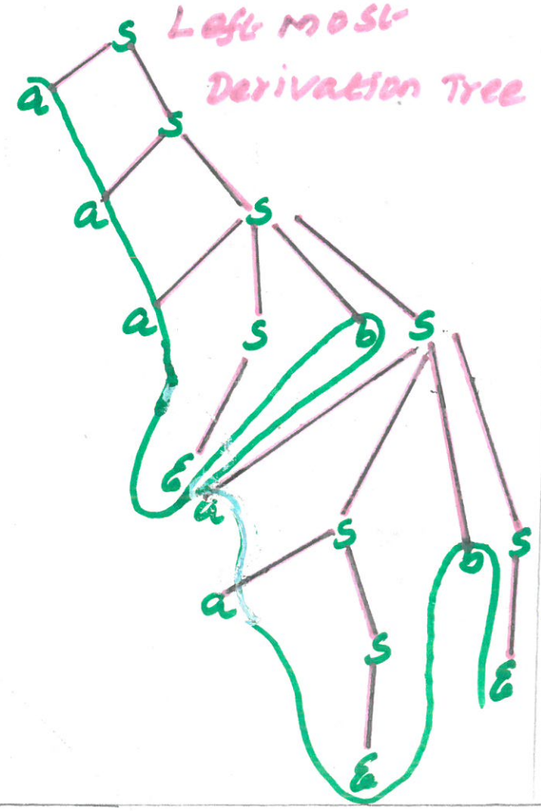
Soln

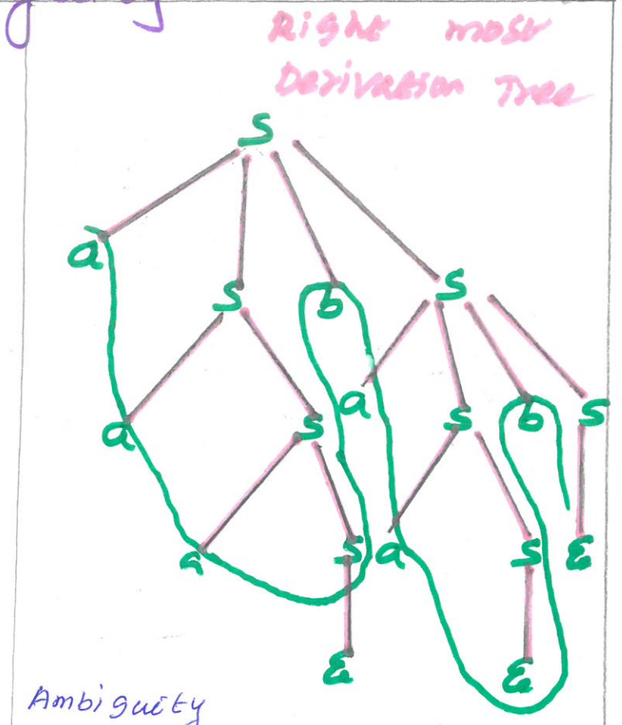
Left Most Derivation

S	$S \rightarrow aS$
aS	$S \rightarrow aS$
aaS	$S \rightarrow aSbS$
aaaSbS	$S \rightarrow \varepsilon$
aaabS	$S \rightarrow aSbS$
aaabaSbS	$S \rightarrow aS$
aaabaaSbS	$S \rightarrow \varepsilon$
aaabaaSbS	$S \rightarrow \varepsilon$
aaabaab	is derived

Rightmost Derivation

S	$S \rightarrow aSbS$
aSbS	$S \rightarrow aSbS$
aSbaSbS	$S \rightarrow \varepsilon$
aSbaSb	$S \rightarrow aS$
aSbaaSb	$S \rightarrow \varepsilon$
aSbaab	$S \rightarrow aS$
aaSbaab	$S \rightarrow aS$
aaaSbaab	$S \rightarrow \varepsilon$
aaabaab	is derived





Ambiguity

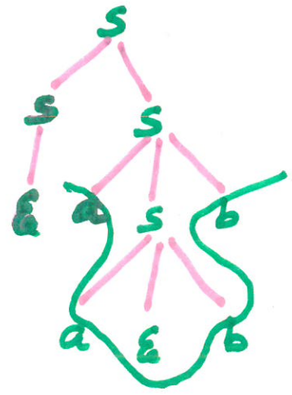
1. Check whether the given grammar G is ambiguous or not.

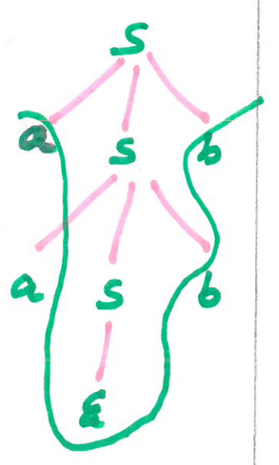
$S \rightarrow aSb / SS$

$S \rightarrow \varepsilon$

Soln

For the string "aabb"





- since there are two parse trees for a single string "aabb", the grammar G is ambiguous.

Construct Stack Implementaion of shift reduce Parsing for the grammar

S → CC

C → CC

C → d

and the Input string cdcd

Stack	Input-string	Action
$	cdcd$	shift
$c	dcd$	shift
$cd	cd$	reduce by c→d
$cC	cd$	Reduce by C→c
$C	cd$	shift
$Cc	d$	shift
$CCd	$	reduce by c→d
$CCc	$	reduce by C→d
$CC	$	reduce by S→cc
$S	$	The i/p string is successfully Passed.

2. Construct operator Precedence Parsing table for the grammar.

P → SR/S

R → bsR/bs

S → wbs/w

W → L*W/L

L → Td

P- Paragraph

S- sentence

R- Recursive no. of sentence.

b- blank

w- word

L- letter

id - Identifier.

- Entire grammar defining the Paragraph.

SR - Two adjacent nonterminals.

- Not a operator grammar.

- Sub- R → bsR/bs

P → SbsR/sbs/s

↳ Again Adjacent nonterminal

- Recursion no.if sentence - P.

Now,

P → SbP/sbs/s

S → wbS/w

W → L*W/L

L → 2d.

* - Right associative.

b - Right associative.

	id	*	b	$
id	–	·>	·>	·>
*	<·	<·	->	·>
b	<·	<·	<·	·>
$	<·	<·	<·	–

3. Construct operator Procedence For the grammar.

E → E+T/T

T → T*F/F

F → (E)/id

and Prase the i/p string id.

Computation of Leading

1. A → Y G B

↳ single numerical (or) & load

(A) = {a}

2. A → B

↳ non terminal

lead(A) = lead(B)

leading(F) = {c, id}

leading(T) = {*, leading(F)}

= {*, c, id}

leading(E) = {+, leading(T)}

= {+, *, c, id}

Computation of Trailing

↳ Terminal

1. A → β X Y

↳ single non terminal(oo) ε

Trailing(A) = {x}

2. A → αB

↳ Nonterminal.

Trialing(A) = Trailing(B)

Traling(F) = {), id}

Traling(T) = {*, Trailing(F)}

= {*,), id}

Relations for the Production

1. E → (E) (T, LN.T) / S → i c t s e s (N.T, T)

≐ i=t , t=e

2. Terminal Followed by nonterminal

eg) aA a <· lead(A)

3. Non terminal followed by Terminal

eg: Aa Trailing(A) ·> a

E+ - Trailing(E) ·> + => {+, *}, id} ·> +

+T - + <· leading(T) => + <· {*, c, id}

T* - Trailing(T) ·> * => {*,), id} ·> *

*F - * <· leading(F) => * <· {c, id}

(E - (<· leading(E) => (<· {+, *, c, id}

E) - Trailing(a) ·>·> {+, *,), id} ·>)-

(E) - (≐)

E$ - Trailing(E) => $ => {+, *,), id} ·> $

$E - $ <· leading(E) => $ <· {+, *, c, id}

	id	+	*	(	)	$
id		·>	·>		·>	·>
+	<·	·>	<·	<·	·>	·>
*	<·	·>	·>	<·	·>	·>
(	<·	<·	<·	<·	≐	
)		·>	·>		·>	·>
$	<·	<·	<·	<·		

In flow graph -≐ → (fcg) - single node.

Stack	i/p string	Action
$	id$	$<· id so shift.
$<· id	·> $	id ·> $ so scan backward and Pop the symbol until <·.
$	$	successful completion The i/p string is Valid.

Construct recursive descent parser for the following grammar.

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

Solve

```
E()
{ T();
  EPRIME();
}
E PRIME()
{ if input - ==+')
  { Input++;
    T();
    E PRIME();
  } else
    return
}
T()
{F()}
  TPRIME()
  }
TPRIME()
{
if (input = ='*')
  { input++;
    F()
    TPRIME();
  }
  else
  return
  }
F()
{if Input = ='(')
  Input ++;
  E();
  if (input = =')')
   Input ++;
  3
  else (Input = =' 'id')
    Input ++;
    }
```

Construct the predictive parser for the following grammar.

$S \to (L)/a$
$L \to L,S/S$

Elimination of left recursion.

$S \to (L)$
$S \to a$
$L \to SL'$
$L' \to ,SL'/\varepsilon$

Computation of FIRST

FIRST (S) = {(,a}
FIRST (L) = {(,a}
FIRST (L') = {,, ε}

Computation of FOLLOW

Follow(S) = {$, ,}
Follow (L) = {)}
Follow (L') = {)}

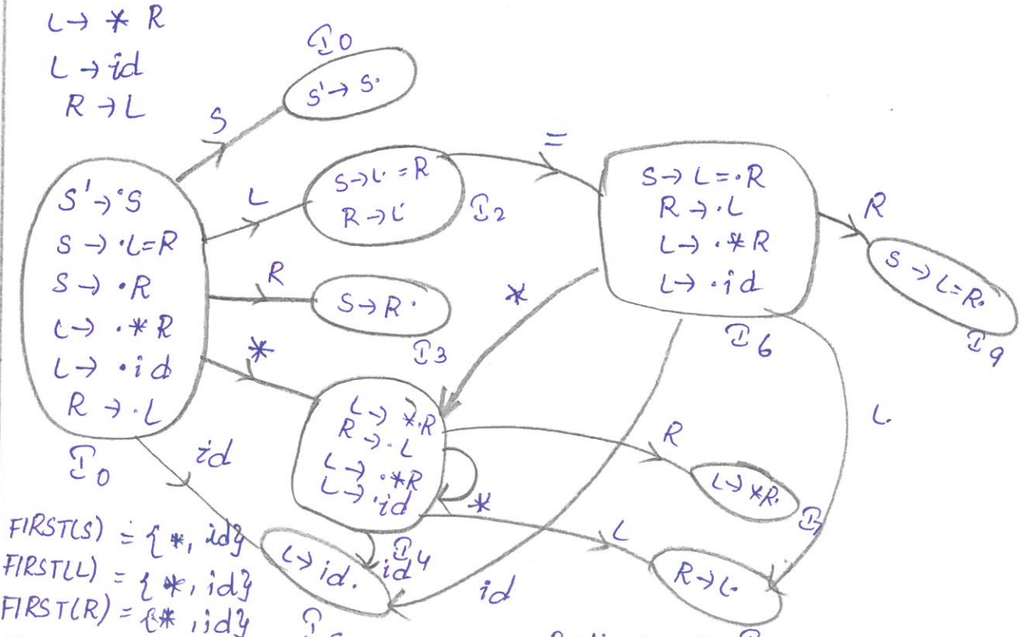
	a	(	)	,	$
S	S→a	S→(L)			
L	L→SL'	L→SL'			
L'			L'→ε	L'→,SL'	

State	Input	Action
$S	(a,a)$	S→(L)
$S(L	(a,a)$	POP'('
$SL	a,a)$	L→SL'
$SL'S	a,a)$	S→a
$SL'a	a,a)$	POP 'a'
$S)R'	,a)$	L'→,SL'
$S)L'S_2	,a)$	POP,
$S)L'S	a)$	S→a
$S)L'a	a)$	POP'a'
$S)L'	)$	L'→ε
$S)	)$	POP)
$S	$	accept.

State	Action				GO TO		
	=	*	id	$	S	L	R
0		S_4	S_5		1	2	3
1				accept			
2	$S6/r_5$			r_5			
3				r_2			
4		S_4	S_5			8	7
5	r_4			r_4			
6		S_4	S_5			8	9
7	r_3			r_3			
8	r_5			r_5			
9				r_1			

Check whether the following grammar is SLR(1) (or) not.

$S \to L=R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$



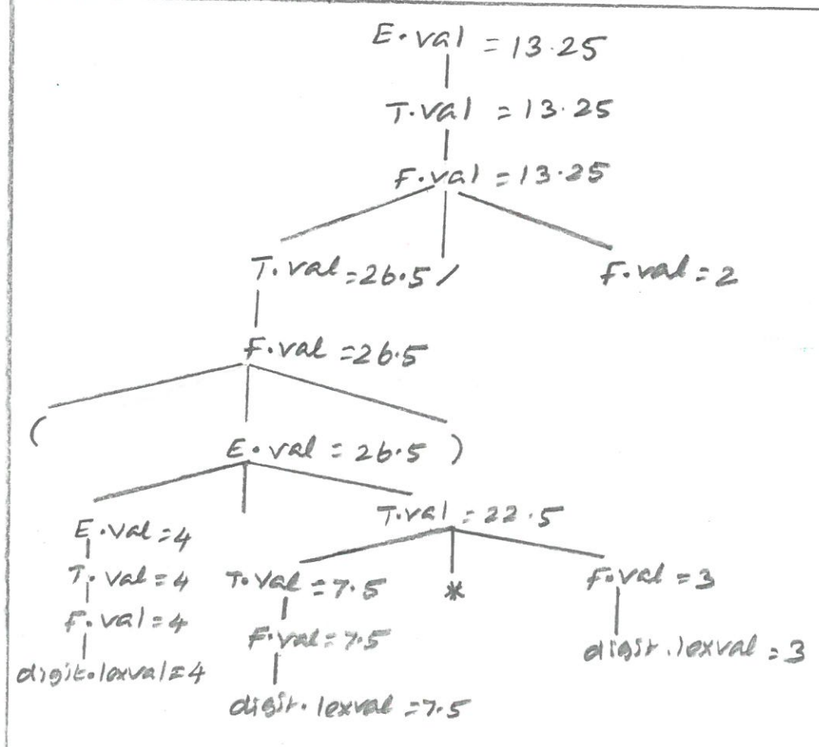
FIRST(S) = {*, id}
FIRST(L) = {*, id}
FIRST(R) = {* ,id}
Follow(S) = {$}
Follow(L) = {=, $}
Follow(R) = {$, =}.

Action(2, =) = S_6/r_6
Shift reduce Conflict
∴ the grammar is not SLR(1).

Construct a decorated parse tree according to the syntax directed definition for the following input statement : (4 + 7.5 * 3)/2

Soln

Production rule	Semantic actions
S → E	print (E.val)
S → E+T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T*F	T.val = T1.val * F.val
T → T/F	T.val = T1.val / F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

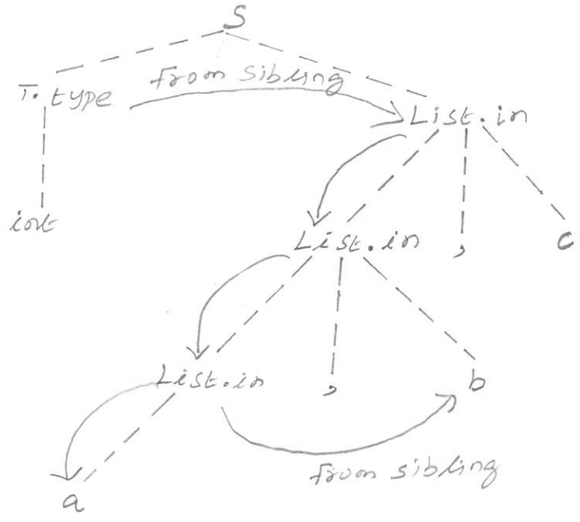


Design the dependency graph for the following grammar.

S → T List
T → int
T → float
T → Char
T → double
List → List1 , id
List → id

Soln

Production rule	Semantic actions
S → T List	List.in = T.type
T → int	T.type = integer
T → float	T.type = float
T → Char	T.type = Char
T → double	T.type = double
List → List1 , id	List1.in = List.in Enter_type (id.entry, List.in)
List → id	Enter_type (id.entry, List.in)

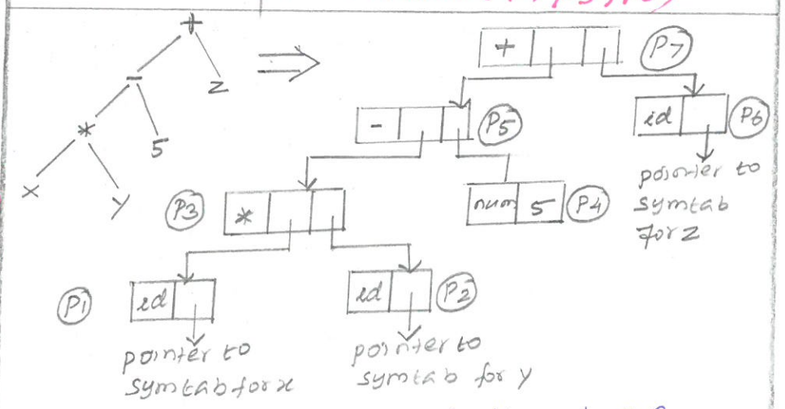


Construct the syntax tree for the expression x * y - 5 + z

Soln

1. convert the expression from infix to postfix xy * 5 - z +
2. make use of the functions Mknode(), mkleaf (id, ptr) and mkleaf (num, val).
3. Sequence of function call postfix expression xy * 5 - z +

symbol	operation
x	P1 = mkleaf (id, ptr to entry x)
y	P2 = mkleaf (id, ptr to entry y)
*	P3 = mknode (*, P1, P2)
5	P4 = mkleaf (num, 5)
-	P5 = mknode (-, P3, P4)
z	P6 = mkleaf (id, ptr to entry z)
+	P7 = mknode (+, P5, P6)



Syntax Directed Definition for the above Grammar

production	Semantic rule
E → E1 + T	E.node = mknode (+, E1.node, T.node)
E → E1 - T	E.node = mknode (-, E1.node, T.node)
E → T	E.node = T.node
T → T1 * F	T.node = mknode (*, T1.node, F.node)
T → F	T.node = F.node
F → id	F.node = mkleaf (id, id.entry)
F → num	F.node = mkleaf (num, num.value)

Implementation of IC

① Translate the following expression to quadruple, triple and indirect triple: $a+b \times c / e \uparrow f + b \times a$

$T_1 = e \uparrow f$

$T_2 = b \times c$

$T_3 = T_2 / T_1$

$T_4 = b \times a$

$T_5 = a + T_3$

$T_6 = T_5 + T_4$

priority or precedence:

↑

× /

+ −

Location	op	arg 1	arg 2	Result
(0)	↑	e	f	T_1
(1)	×	b	c	T_2
(2)	/	T_2	T_1	T_3
(3)	×	b	a	T_4
(4)	+	a	T_3	T_5
(5)	+	T_5	T_4	T_6

Quadruple

Location	op	arg 1	arg 2
(0)	↑	e	f
(1)	×	b	c
(2)	/	(2)	(0)
(3)	×	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

Triple

	Statement
(0)	(20)
(1)	(21)
(2)	(22)
(3)	(23)
(4)	(24)
(5)	(25)

location	op	arg 1	arg 2
(20)	↑	e	f
(21)	×	b	c
(22)	/	(20)	(20)
(23)	×	b	a
(24)	+	a	(22)
(25)	+	(24)	(23)

Indirect triple

Three Address Code

② write three address code for the following code.

```
for (i = 1 ; i <= 10 ; i++)
{ a[i] = x * 5 ;
}
```

Sol: i = 1

```
L : t1 = x * 5
    t2 = &a
    t3 = sizeof (int)
    t4 = t3 * i
    t5 = t2 + t4
   *t5 = t1
    i = i + 1
if i <= 10 goto L
```

③ write three address code for the following expression:

If A < B and C < D then t = 1
else t = 0.

(1) If (A < B) goto (3)
(2) goto (4)
(3) If (C < D) goto (6)
(4) t = 0
(5) goto (7)
(6) t = 1
(7)

BACK PATCHING

④ Generate the three address code and Syntax tree for the following expression.

a < b or c < d and e < f

Three address code:

```
100 : if a < b goto -
101 : goto 102.
102 : if c < d goto 104
103 : goto -
104 : if e < f goto -
105 : goto -
```

E.Tlist = {100, 104}
E.Flist = {103, 105}

E.Tlist = {100}
E.Flist = {101}

a < b

or

M.state = {102}

E

E.Tlist = {104}
E.Flist = {103, 105}

E.Tlist = {102} and M.state = {104} and E.Tlist = {104}
E.Flist = {103}

c < d

E

E.Flist = {105}

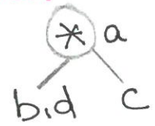
e < f

① Construct the DAG for the following block.

a := b * c
d := b
e := d * c
b := e
f := b + c
g := f + d.

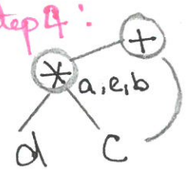
Step 1:

Step 2:

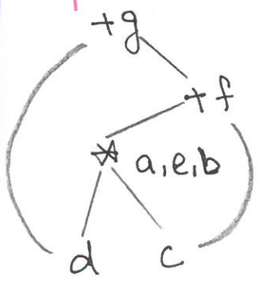


Step 3:

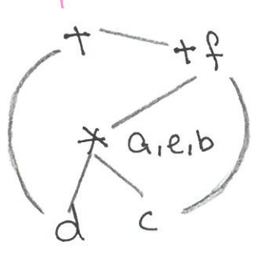
Step 4:



Step 5:

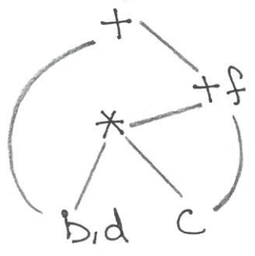


Step 6:



Optimized Code:

a := b * c
d := b
f := a + c
g := f + d



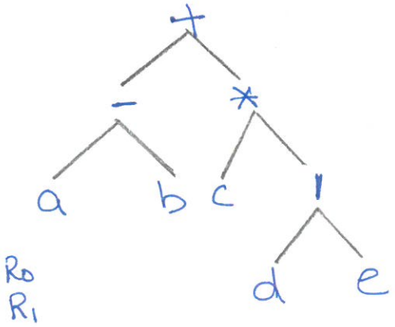
② CODE GENERATION:

d := (a-b) + (a-c) + (a-c)

t := a - b
u := a - c
v := t + u
d := v + u

Statement	Code Generator	Register descriptor Register Empty	Address descriptor
t := a-b	MOV a, R0 SUB b, R0	R0 Contains t	t in R0
u := a-c	MOV a, R1 SUB c, R1	R0 Contains t R1 Contains u	t in R0 u in R1
V := t+u	ADD R1, R0	R0 Contains V R1 Contains u	u in R1 V in R1
d := V+u	ADD R1, R0 MOV R0, d	R0 Contains d	d in R0 d in R0 and Memory

③ Construct a DAG for the given Expression.
(a-b) + c * (d+e) with Code.



Code:
MOV C, R0
MOV d, R1
DIV R1, e → R1 := (d/e)
MUL R1, R0 → R0 := c * (d/e)
MOV R1, a → R1 := a
SUB b, R1 → R1 := R1 - b (i.e) a-b
ADD R1, R0 → R0 := R1 + R0.

FLOW GRAPH:-

B1
W = 0;
x = x+y;
y = 0;
if (x > 2)

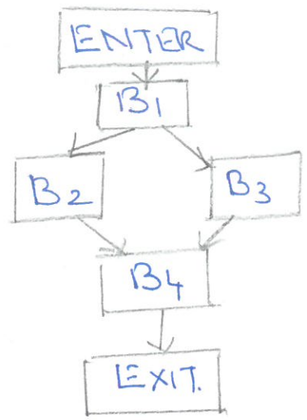
B2
y = x;
x++;

B3
y = z;
z++;

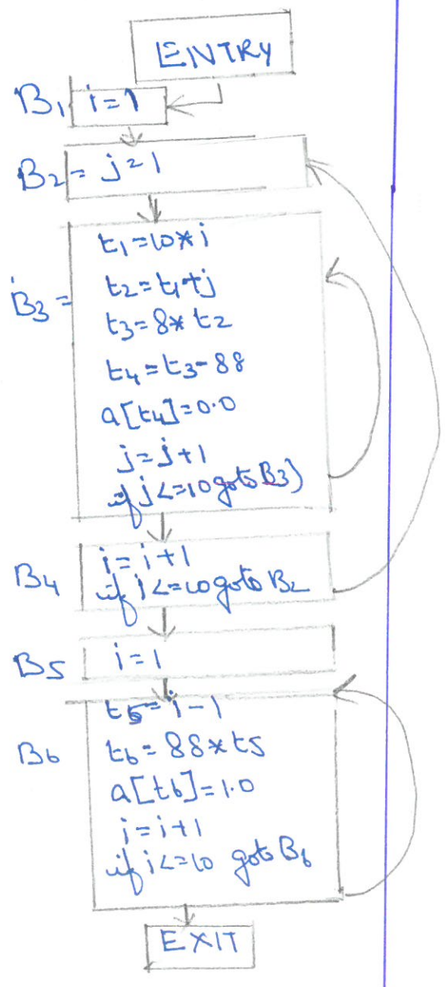
B4
w = x+z;

BASIC BLOCKS

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j+1
9) if j <= 10 goto (3)
10) i = i+1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i-1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i+1
17) if i <= 10 goto (13)



FLOW GRAPH.



Translate the following assignment statement that three address code $x = (a+b) * (c-d) + ((e/f) * (a+b))$ Apply code generation algorithm, generate a code sequence for three address statement.

three address code.	Target code sequence	Register descriptor	operand descriptors
$t_1 := a+b$	MOV a, R_0 ADD b, R_0	Empty R_0 contains t_1	t_1 R R_0
$t_2 := c-d$	MOV c, R_1 SUB d, R_1	R_1 contains c R_1 contains t_2	t_2 R R_1
$t_3 := e/f$	MOV e, R_2 DIV f, R_2	R_2 contains e R_2 contains t_3	t_3 R R_2
$t_4 := t_1 * t_2$	MUL R_0, R_1	R_0 contains t_1 R_0 contains t_2 R_0 contains t_4	t_4 R R_1
$t_5 := t_3 * t_1$	MUL R_2, R_0	R_2 contains t_3 R_0 contains t_1 R_0 contains t_5	t_5 R R_0
$t_6 := t_4 + t_5$	ADD R_1, R_0	R_1 contain t_4 R_0 contain t_5 R_0 contain t_6	t_6 R R_0

*Generate code for the following statements for target machine

1. $x = x+1$
2. $x = a+b+c$
3. $x = a_1 / (b-c) - d * (e+f)$

Solution:- 1. The three address code will be

$t_1 := x+1$
$x := t_1$

The code will be

MOV x, R_0
ADD #1, R_0
MOV R_0, x

2) The three address code will be

$t_1 : b+c$
$t_2 : a+t_1$
$x : t_2$

The code will be -

MOV a, R_0
ADD b, R_0
ADD c, R_0
MOV R_0, x

3. The three address code will be

$t_1 := b-c$
$t_2 := a_1 / t_1$
$t_3 := t_2 - d$
$t_4 := e+f$
$t_5 := t_3 * t_4$
$x := t_5$

The code will be-

MOV b, R_0
SUB c, R_0
MOV a_1, R_1
DIV R_0, R_1
SUB d, R_1
MOV f, R_2
ADD e, R_2
MUL R_1, R_2
MOV R_2, x

* For the following expression obtain optimal code using
i) only two registers
ii) only one register
$(a+b) - (c-(d+e))$

Solution: First of all we will write a three address code for given expression

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

The generated code using simple code generation algorithm will be.

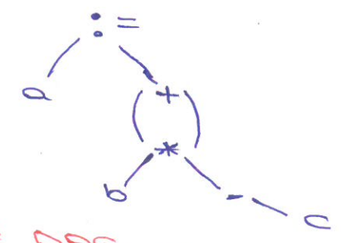
i)
MOV d, R_0
ADD e, R_0
MOV c, R_1
SUB R_0, R_1
MOV a, R_0
ADD b, R_0
SUB R_1, R_0.

ii)
MOV d, R_0
ADD e, R_0
MOV R_0, t_1
MOV c, R_0
SUB t_1, R_0
MOV R_0, t_2
MOV a, R_0
ADD b, R_0
SUB t_2, R_0

* How would you represent the following equation using the DAG?

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

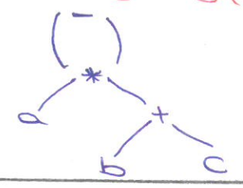
Solution:



* Draw the DAG for the statement

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

Solution:-



SIMATS

SCHOOL OF ENGINEERING

Approved by AICTE | IET-UK Accreditation

Engineer to Excel

Saveetha Nagar, Thandalam, Chennai - 602 105, TamilNadu, India