1 INTRODUCTION AND LEXICAL ANALYSIS

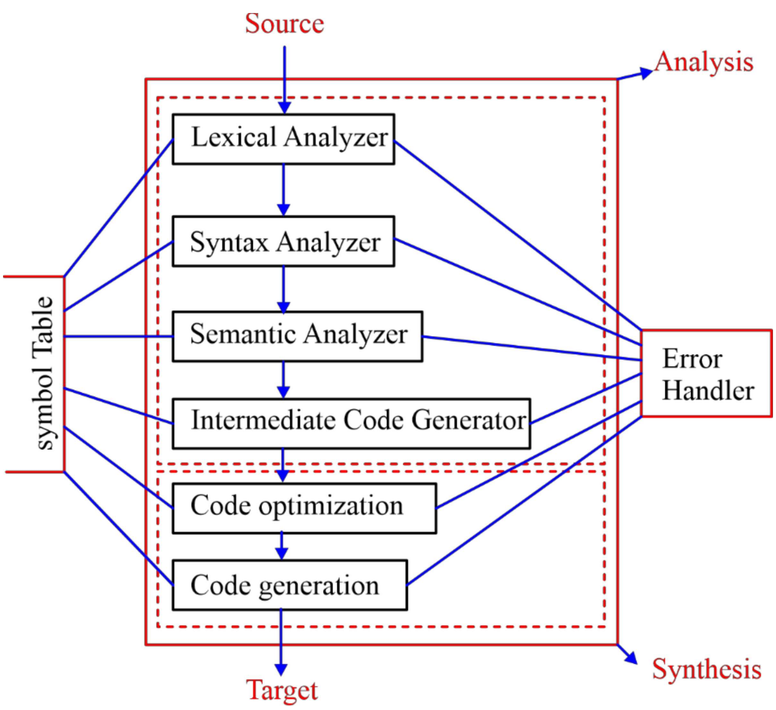
1.1 CD Introduction

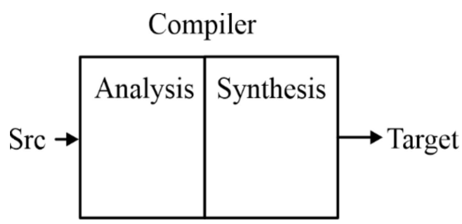
1.1.1 Definition

Convert High Level Language to Low Level Language.

- High Level Language can perform more than one operation in a single Statement.
- lol Level Language can perform atmost one operation in a statement.

1.2 Analysis and Synthesis model of Compiler





- There are 6 phases of the Compiler

1. Lexical Analyzer:

- Program of DFA, it checks for spelling mistakes of program.
- Divides source code into stream of tokens.

2. Syntax Analyzer:

- Checks grammatical errors of the program (Parser).
- Parser is a DPDA.

3. Semantic Analyzer:

Checks for meaning of the program.

Example:

Type miss match, stack overflow

4. Intermediate Code Generation:

- This phase makes the work of next 2 phases much easier.
- Enforces reusability and portability.

5. Code optimization:

- Loop invariant construct
- Common sub expression elimination
- Strength Reduction
- Function in lining

Deadlock elimination

6. Symbol Table:

(1) Data about Data (Meta data)

(2) Data structure used by compiler and shared by all the phrase.

❑❑❑

2 PARSING

2.1 CD - Grammar

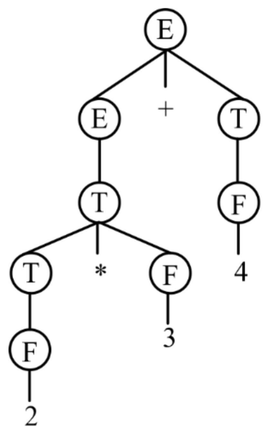
→ In compiler we only use: Type – 2 (CFG) and Type – 3 Regular grammar.
→ Compiler = Program of Grammar
→ Compiler = Membership algorithm
- Every programming Language is Context Sensitive Grammar (Context Sensitive Language)

2.1.1 Parse Tree and Syntax Tree

G: $E \rightarrow E + T / T$ $\quad E \rightarrow E + T \rightarrow T + T \rightarrow T * F + T \rightarrow F * F + T \rightarrow 2 * F + T \rightarrow 2 * 3 + T \rightarrow 2 * 3 + F$
$T \rightarrow T * F / F$

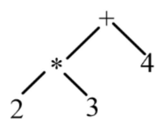
$\boxed{E \rightarrow 2 * 3 + 4}$

Parse Tree:



→ Procedural
→ Int Node = Variable
→ Whole Derivation is represented

Syntax Tree:



→ non procedural
→ Int node = Operation
→ Discuss Concept only

- To check to priority / Associativity:
  Randomly derive till you have enough operators, then check which one is done first.
- If priority of 2 operators is same and both Left and Right associative → **Ambiguous Grammar** [Useless]

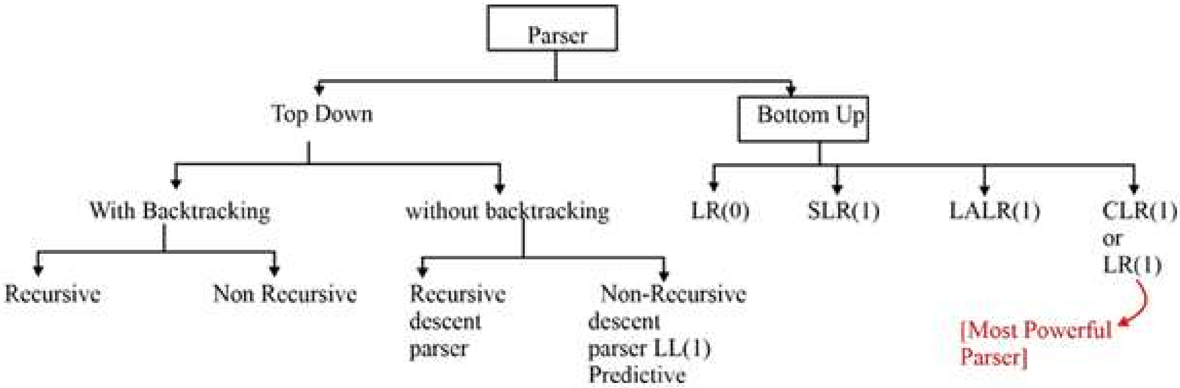
2.1.2 Types of Parsers in Bottom Up: [CD - Parser]

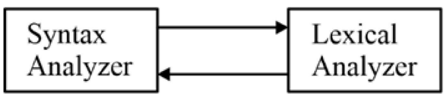
- item
  - reduced $(X \to \alpha \cdot)$ dot(.) at the end
  - unreduced
    - State $(X \to \alpha \cdot t\beta \mid \& \in \text{non-terminal})$
      Variable after (.)
    - Shift $(X \to \alpha \cdot t\beta \mid \& \in \text{terminates})$
      Terminal after (.)

2.2 CD-Syntax Analysis / Parsing

Grammatical Errors are checked by the help of parsers.

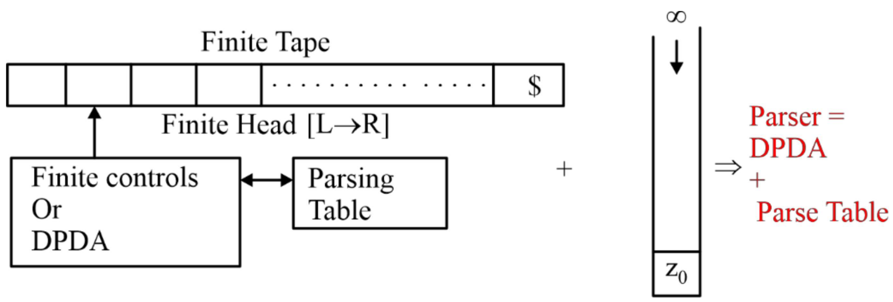
- Parsers are basically DPDA





- All of these parsers are table driven.

2.2.1 Mathematical Model of Parser



- Parsers generate Parse Tree, for a given string by the given grammar.

2.2.2 Top-Down Parser (LL (1))

- It uses LMD and is equivalent to DFS in graph.

2.2.3 Algorithm to construct Parsing Table

1. Remove Left Recursion if any.
2. Left Factor [Remove Common Prefix]
3. Find first and follow set
4. Construct the table.

- If we increase the look ahead symbol:

2.2.4 Removal of common Prefix: (Left Factor)

1. $S \to a \mid a\ b \mid a\ A$ $S \to a\ Y$

   $Y \to \in |b|\ A$

2. $A \to a\ b\ A \mid a\ A \mid b$

   $A \to a \times \mid b$ $A \to a \times \mid b$

   $X \to b\ A \mid A$ $X \to b\ A \mid a\ x \mid b$

   $A \to a\ X \mid b$

   $X \to a\ X \mid b$

   $Y\ Y \to A \mid \in$

2.2.5 First and Follow

- First Set $\to$ extreme Left terminal from which the string of that variable starts.

  $\to$ it never contains variable, but may contain '$\in$'.

  $\to$ we can always find the first of any variable.

- Follow set $\to$ Follow set contains terminals and \$.

  It can never contain variable and '$\in$'.

  How to find follow set?

1. Include \$ in follow of start variable.
2. If production is of type $\to$

   $A \to \alpha B\beta$ $\alpha\beta \to$ strings of grammar symbol].

   follow (B) = first (β).

   If, $\beta \to \in$, i.e. $A \to \alpha B$, then follow(B) = follow(A).

- Productions like: $A \to \alpha A$ gives no follow set.

2.2.6 Example of first and follow set

1.

S	→	AB	|	CD
A	→	aA	|	a
B	→	bB	|	b
C	→	cC	|	c
D	→	dD	|	d

	First	Follow
S	a, c	$
A	a	b
B	b	$
C	c	D
D	D	$

2.2.7 Entity into Table: Top Down

1. No of Rows = number of unique variable in Grammar.
2. No of columns = [Terminals + $]
3. For a Variable (Row) fill the column (Terminal) if its P, there in its first with the production required.
4. If $\in$ is in first put $V \rightarrow \in$ under $ and its follow.

- If any cell has multiple items, then its not possible to have LL (1) Parser, since that will be ambiguous.
- In top down we do: Derivation
  In Bottom up we do: Production.

Question: Construct LL (1) Parsing Table for the given Grammar: $E \rightarrow E + T \mid T;\ T \rightarrow T * F \mid F;\ F \rightarrow (E) \mid id; \Rightarrow G_0$

$\Rightarrow$ Removing Left Recursion:

$E \rightarrow TE'$
$E' \rightarrow + TE' \mid \in$
$T \rightarrow FT'$ $\quad G_1$
$T' \rightarrow *FT' \mid \in$
$F \rightarrow (E) \mid id$

	First	Follow
E	C, id	$,)
E'	+, ∈	$,)
T	C, id	+, $,)
T'	*, ∈	+, $,)
F	C, id	*, +, $,)

- Left Factoring not Required:
  Construction of Table: [LL (1)]

	+	*	(	)	id	$
E	Error	Error	E→TE'	Error	E→ TE'	Error
E'	E'→TE'	Error	Error	E'→∈	Error	E'→∈
T	Error	Error	T→FT'	Error	T→ FT'	error
T'	T'→∈	T'→*FT'	Error	T'→∈	Error	T'→∈
F	Error	Error	F→(∈)	Error	F→ id	error

- Since for G_1, Table constructed with no multiple entries. Hence successfully completed. Hence G_1 is LL (1)

Question: Construct LL (1) Parsing Table for the following Grammar:
$S \rightarrow L = R \mid R;\ L \rightarrow * R \mid id;\ R \rightarrow L \Rightarrow G_0$

Solution: Left Factoring:

$S \rightarrow L = R \mid L$
$L \rightarrow *R \mid id$
$R \rightarrow L$

$\Rightarrow$

$S \rightarrow LX$
$\times \Rightarrow = R \mid \in$
$L \rightarrow *R \mid id$
$R \rightarrow L$

$\Rightarrow G_1$

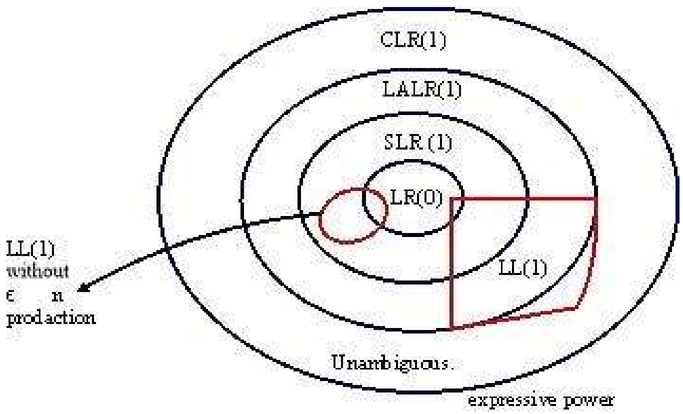
	First	Follow
S	*, id	$
×	=, ∈	$
L	*, id	$
R	*, id	$

Construction of Table:

	*	=	Id	$
S	S→LX	Error	S→L X	Error
L	L→*R	Error	L→id	Error
R	R→L	Error	R→L	Error
×	Error	X→ = R	Error	X→∈

- G_1 is a LL (1) Grammar.

2.2.8 Hierarchy of Parsers: [for ∈- free Grammar]



- For ∈ - producing grammars every LL (1) may not be LALR (1).

> Note:
> We can't construct any parser for ambiguous grammar. Except: Operator precedence, parser possible for some ambiguous grammar.

Example:

G: S → a S a | b S b | a | b (Unambiguous but no parser) L(G) = ω(a + b) ω R
(Odd palindrome)

- Every RG is not LL (1) as it may be ambiguous, or Recursive or Common Prefix.
- Parsers exists only for the grammar if its Language is DCFL.
- There are some grammars whose Language is DCFL but no parser is possible for it.

2.4.9 Operator Precedence Grammar

Format: (1) No 2 or more variable side by side

(2) No $\in$ production.

Example:

$$\begin{array}{l} E \to E+T \mid T \\ T \to T*F \mid F \\ F \to (E) \mid id \\ \text{O.G.} \end{array} \quad \text{or} \quad \begin{array}{l} E \to E+E \\ E \to E*E \\ E \to a/b \\ \text{O.G} \end{array} \quad \text{or} \quad \begin{array}{l} S \to aSa \mid bSb \mid a \mid b \\ \text{O.G} \end{array}$$

Or

$$\left.\begin{array}{l} S \to AB \\ A \to aA \mid \in \\ B \to bB \mid \in \end{array}\right\} \text{Not O.G}$$

2.2.10 Checking LL (1) without Table

$A \to \alpha_1 \mid \alpha_1 \mid \alpha_1$ then $\to$	$A \to \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \in$
first $(\alpha_1) \cap$ first $(\alpha_2) = \varnothing$	first $(\alpha_1) \cap$ first $(\alpha_2) = \varnothing$
first $(\alpha_1) \cap$ first $(\alpha_3) = \varnothing$	first $(\alpha_1) \cap$ first $(\alpha_3) = \varnothing$
first $(\alpha_2) \cap$ first $(\alpha_3) = \varnothing$	first $(\alpha_2) \cap$ first $(\alpha_3) = \varnothing$
	follow (A) $\cap$ first $(\alpha_1) = \varnothing$
	follow (A) $\cap$ first $(\alpha_2) = \varnothing$
	follow (A) $\cap$ first $(\alpha_3) = \varnothing$

2.2.11 Bottom-UP Parser

- It uses RMD in reverse and has no problem with:
  (a) Left recursion (b) Common Prefix
  SLR (1) LR (0) CLR (1) LR (1)
  LR (0) items LALR (1) items
  LR (1) = LR(0) +1 Lookahead
- No parser possible for ambiguous grammar.
- There are some unambiguous grammars for which, there are no Parser.

2.2.12 Basic Algorithm for Construction

- Augment the grammar and expand it, and give numbers to it
- Construct LR (0) or LR (1) item set.
- From that fill the entries in the Table Accordingly.

2.2.13 Types of Entries

(1) Shift Entries: Transitions or Terminals

$$I_0 \xrightarrow[\text{...Action...}]{a} I_1$$

Entry:

	a
I_0	S_1

a∈ Terminal I_0, I_1: Item sets.

- Shift entries are some for all Bottom – up Parser.

(2) State Entry: Transition or non – terminal (variable)

$$I_0 \xrightarrow[\text{<...goto...>}]{A} I_1$$ A → Variable

Entry:

	A
I_0	1

- Same for all Bottom-up Parser.

(3) Reduce Entity: done for each separate production in the item set of type:

$i > \times \to \alpha$

where, $i \to$ Production Number

$\times \to$ Producing Variable

$\alpha \to$ Grammar String

(a) LR (0) Parser:

Put Ri in every cell of the set-in action table **(ALL).**

(b) SLR (1) Parser:

Put Ri only in the follow(x) form the Grammar. **(Follow(x))**

(c) LALR (1) and CLR (1):

Put Ri only in the look ahead of the production **(Lookaheads)**

2.2.14 Conflicts

LR (0) Parser:

SR: Shift Reduce Conflict

$S \to \quad X \to \alpha \cdot t\beta \qquad \Rightarrow$ then SR

$R \to Y \to \delta \cdot$ conflict

RR: Reduce Reduce conflict

$R \to X \to \alpha \cdot \qquad \Rightarrow$ then RR

$R \to X \to \beta \cdot$

SLR (1) Parser:

SR:

$S \to X \to \alpha \cdot t\beta$ $\$$ $\Rightarrow$then SR

$R \to Y \to \delta \cdot$ conflict

$t \in$ follow (Y)

RR:

$X \to \alpha \cdot$ $\$$ $\Rightarrow$then RR

$Y \to \beta \cdot$

Follow (x)n follow (y) $\neq \varnothing$

LALR (1) and CLR (1): Same as SLR (1), but instead

SR:

$X \to \alpha \cdot t\,\beta \cdot L_1$ $\$\Rightarrow$then SR

$Y \to \delta \cdot L_2$ conflict

$T \in L_2$

RR:

$X \to \alpha \cdot, L_1$ $\$$ then RR

$Y \to \beta \cdot, L_2$ conflict

$L_1 \cap L_2 \neq \varnothing$

2.4.15 Inadequate Static

A static having ANY conflict is called a conflicting state or independent state.

> Note:
> The state S' $\to$ S. or S' $\to$ S.,$ is accepted state, and this is not a reduction.

- The only difference between CLR (1) and LALR (1) is that, the states with the similar items, but different Lookaheads are merged together to Reduce space.

Important Points

1. If CLR (1) doesn't have any conflict, then conflict may or may not arise after merging in LALR (1).
2. If LALR (1) has SR – conflict, then we can conclude that CLR (1) also has SR – Conflict.
3. LALR (1) has SR – conflict if and only if CLR (1) also has SR.

- We can construct parser for every unambiguous regular grammar: [CLR (1) Parser].

	L Left to right scan of i/p	R Using reverse RMD	(0) No Lookhead	
S Simple	L L to R Scan	R Using Reverse RMD	(1) Lookahead	
L Look	A Ahead	L L to R Scan	R Revers RMD	(1) Lookahead
C Canonical	L L to R Scan	R reverse RMD	(1) Lookahead	

Very Important Point:

LALR (1) Parser can Parse non LALR (1) grammar, when only non-SR – Conflict by favouring shift over reduce.

Example: $E \to E + E \mid E * E \mid id \mid 2 + 3 * 5 \Rightarrow E + E \cdot * 5$

❑❑❑

3 LEXICAL ANALYSIS

3.1 Lexical Analysis

Char Stream
↓
Lexical analysis
↓
Token Stream

→ Token Generation
→ Scanner
→ Linear Phase
→ Token Recognizer

3.1.1 Definition

Scan the whole program, char by char and produces the corresponding token.

- Also produces /Lexical Errors (if any).
- Functions of Lexical Analyzer.

(1) Scans all the character of the program.
(2) Token recognizer.
(3) Ignores the comment & spaces.
(4) Maximal Munch Rule [Longest Prefix Match].

> Note:
> The Lexical Analyser uses, the Regular Expression.
> Prioritization of Rules.
> Longest Prefix match .

- Lexeme → smallest unit of program or Logic.
- Token → internal representation of Lexeme.

3.1.2 Types of Token

(1) Identifier
(2) Keywords
(3) Operators
(4) Literals/Constants
(5) Special Symbol

3.1.3 Token Separation

(1) Spaces
(2) Punctuation

3.1.4 Implementation

- LEX tool ⇒ Lex.yy.e

- All identifiers will have entry in symbol Table/ LA, gives entries into the symbol Table.
  Regular Expression → DFA → Lexical Analyzer

3.1.5 Find number of Tokens

(1) void main () {
Printf("gate");
}
{11 Tokens}

(2) int x, *P;
x = 10; p = &x; x++;
[18 Tokens]

(3) int x;
x = y;
x == y;
[11 Tokens]

(4) int 1x23; [Lexical Error]

(5) char ch = 'A';
[5 Token]

(6) char ch = 'A;
Lexical Error.

(7) char *p = "gate";
[6 Tokens]

(8) char * p = "gate;
Error.

(9) int x = 10;
/* comment x = x + 1; ERROR
x = x + 1; [11 Tokens]

❑❑❑

4 SYNTAX DIRECTED TRANSACTION

4.1 Syntax Directed Transaction

CFG + Transition
Syntax + Transition } SDT

SDT: CFG + Transition → 1) Meaning 2) Semantic

4.1.1 Application of SDTL

(1) Used to perform Semantic Analysis.
(2) Produce Parse Tree.
(3) Produce intermediate representation.
(4) Evaluate an expression.
(5) Convert infix to prefix or postfix.

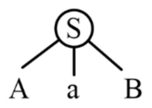
Example: $S \to S_1S_2$ [S.count = S_1.count + S_2.count]
$S \to (S_1)$ [S.count = S_1.count + 1]
$S \to \epsilon$ [S.count = 0]

- Count is an attribute for non–terminal

4.1.2 Attributes

(1) Inherited Attribute
(2) Synthesized Attribute

$S \to AaB$



S is parent to A, a, B
A is child to S
A is Left Sibling to a, B

(1) Inherited Attribute: (RHS)

$S \to \boxed{A}B$ {A.x = f (B.x | S.x)}

- The computation at any node (non-terminal) depends on parent or siblings (S).

- In above Example, x is inherited attribute.

(2) Synthesized Attribute: (LHS)

$S \rightarrow A\ B\ \{S.x = f(A.x \mid B.x)\}$

x is synthesized attribute.
The computation at any node (non-terminal) depends on children.

4.1.3 Identifying Attribute Type

- Always check every Translation.

1) $D \rightarrow T : L\ ;\ \{L.Type = T.Type\}$ inherited
$L \rightarrow L1,\ id\ ;\ \{L1.Type = L.Type\}$ inherited
$L \rightarrow id$
$T \rightarrow integer\ \{T.Type = int\}$ synthezised

Type is neither inherited nor synthesized

2) $E \rightarrow E1 + T\ \{E.val = E1.val + T.val\}$ synthesized
$E \rightarrow T\ \{E.val = T.val\}$ Synthesized
$T \rightarrow T1 - F\ \{T.val = T1.val - F.val\}$ Synthesized
$F \rightarrow id\ \{F.val = id.val\}$ synthesized

Val is synthesized attribute

3) $S \rightarrow AB\ \{Aa = B.x;\ S.y = A.x\}$ x is inherited | y is synthesized
$A \rightarrow a\ \{A.y = a\}$ y is synthesized
$B \rightarrow b\ \{B.y = a.y\}$ y is synthesized

$x \Rightarrow$ inherited
$y \Rightarrow$ synthesized

4) $D \rightarrow T\ L\ \{L.in = T.type\}$ inherited(in)
$T \rightarrow int\ \{T.type = int\}$ /synthesized
$L \rightarrow id\ \{Add\ type(id.entry, L.in)\}$

in $\Rightarrow$ inherited
type $\Rightarrow$ synthesized

4.1.4 Syntax Directed Definitions (SDDs): (Attribute Grammar)

(1) L-Attributed Grammar

→ Attribute is synthesized or restricted inherited. (1) Parent 2) Left sibling only).
→ Translation can be appended anywhere is RHS of production.
→ Example: $S \rightarrow AB\ \{A.x = S.x + 2\}$
or, $S \rightarrow AB\ \{B.x = f(A.x \mid S.x)\}$
or, $S \rightarrow AB\ \{S.x = f(A.x \mid B.x)\}$
→ Evaluation: In Order (Topological).

(2) S-Attributed Grammar:

→ Attribute is synthesized only.
→ The transaction is placed only at the end of production.
→ Example: $S \rightarrow AB\ \{S.x = f(Ax \mid Bx)\}$.
→ **Evaluation:** Reverse RMD (Bottom-Up Parsing).

4.1.5 Identify SDD

(1) $E \rightarrow E_1 + E_2$ {E.type = if (E_1.type = = int & & E.type = = int) then int} Synthesizer else type – error.

$E \rightarrow id$ {E.type = Lookup (id.entry)} synthesizer.

- type is synthesized, hence S-attribute and also L-attributed Grammar.
- Every S-attributed Grammar is also L-attributed Grammar.
- For L-attributed Evaluation, use the In-order of annotated Parser Tree.
- For S-attributed, reverse of RMD is used.

→ find RMD Order.

→ Consider its reverse.

❑❑❑

5 INTERMEDIATE, CODE OPTIMIZATION

5.1 Introduction

Representations

- Linear Form
  - Postfix code
  - 3-address code
  - state-single assignment (SSA code)
- Non-Linear Form
  - Syntax Tree
  - Directed Acyclic graph (DAG)
  - Control Flow Graph

Example Expression:

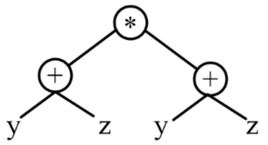
$(y + z) * (y + z)$

Post fix $\rightarrow$ $\boxed{yz+yz+*}$

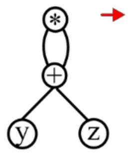
SSA $\rightarrow$ $\boxed{\begin{array}{l} t_1 = y+z \\ t_2 = t_1 * t_2 \end{array}}$ $\Rightarrow f_1$ and f_2 cannot be reassigned

3AC $\rightarrow$ $\boxed{\begin{array}{l} t_1 = y+z \\ t_2 = t_1 \times t_2 \end{array}}$

Syntax Tree $\rightarrow$



DAG $\rightarrow$



Reuse the already existing common sub expression

3-address Code: Code in which, at most 3 addresses. [including LHS] address → name, variable, constant

$x = y + z * y - a$

1, 2, 3

$t_1 = z * y$

$t_2 = y + t_1$

$t_3 = t_2 - a$

$x = t_3$

Equivalent 3 - address Code

Representation of 3AC

Triple not | Quadruple not | Indirect Triple not

0	*	z	y
1	+	y	(0)
2	-	(1)	a
3	=	(2)	

0	*	z	y	t_1
1	+	y	t_1	t_2
2	-	t_2	a	t_3
3	=	t_3		x

6	(0)
7	(1)
8	(2)
9	(3)

→ Indirect notation pointers to the numbers of Triple

Triple Notation
- → Space efficient
- → time inefficient

Quadruple
- → space inefficient
- → time efficient

- 3AC done using operator precedence.

Find minimum number of variables required in equivalent 3AC:

1.

$x = u - t$

$y = x * v$

$x = y + w$

$y = t - z$

$y = *x *y$

7 variables

$x * y \Rightarrow x * (t - z) \Rightarrow (y + w) * (t - z)$

$\Rightarrow ((x * v) + w) * (t - z)$

$\Rightarrow (((u - t) * v) + w) * (t - z)$

$$\begin{bmatrix} u = u - t & u = t - z \\ v = u * u & z - w * v \\ w = v + w & \end{bmatrix} \Rightarrow \text{5 variables only}$$

$\Rightarrow$ Evaluating the expression:

a $\Rightarrow c + b \Rightarrow d - b + b \Rightarrow b + c - b + b$

$\Rightarrow b + a + d - b + b$

$\Rightarrow b + b + c + d - b + b$

$\Rightarrow b + b + c + d$

$a = b + c$
$c = a + d$
$d = b + c$
$c = d - b$
$a = c + b$

$$\boxed{\begin{aligned} t_1 &= b + b \\ t_1 &= t_1 + c \\ a &= t_1 + d \end{aligned}}$$

Minimum 3AC.
[wrong, even t_1 is not used]

∴ Minimum:

$$\left\{\begin{matrix} b = b + b \\ c = b + c \\ a = c + d \end{matrix}\right\} \Rightarrow \text{only 3 variables [most optimal]}$$

5.1.1 Static Single Assignment Code: (SSA Code)

Every variable (address) in the code has single assignment [single meaning] + 3AC.

1. $x = u - t$
$y = x * u$
$x = y + w$
$y = t - z$
$y = x * y$

Find SSA?

$\Rightarrow$ [u, t, v, w, z] are already assigned so we can't use them.

Equivalent SSA Code:

$x = u – t$ $y = x * v$

$p = y + w$

$q = t – x$

$r = p * q$

in use: x, y, p, q, r $\Rightarrow$ additional

∴ Total Variable $\Rightarrow$ 10.

2. $p = a - b$
$q = p * c$
$p = u * v$
$q = p + q$

$\Rightarrow$ [a, b, c, u, v] are already assigned,

Equivalent SSA Code:

$p = a – b$

$q = p * c$

$p_1 = v * u$

$q_2 = p_1 + q$

in use: p, q, p_1, q_2

∴ Total Variable = 9

5.1.2 Control Flow Graphs

- CFG contain group of basic blocks and controls. CFG has nodes and edges to define basic blocks and controls.
- Basic Blocks: Sequence of 3-address code statements, in which control enters only from 1st statement (called as leader), and leaves from last statement.

Example:

1.	i = 1	LB_1
2.	j = 1	LB_2
3.	t1 = 5 * i	
4.	t2 = t1 + 5	
5.	t3 = 4*t2	LB_3
6.	t4 = t3	
7.	a[t4] = 1	
8.	j = j + 1	
9.	if j ≤ 5 goto 3	LB_4
10.	i = i + 1	LB_5
11.	if i < 5 goto 2	LB_6

5.2 Code Optimization

Saves space / time. (Basic Objective)

Flow Analysis
- Control Flow Graph
- Data Flow Graph

optimization
- Local (Intra-procedural Basic Blocks)
- Global (Inter Procedural Whole Program)

5.2.1 Optimization Methods

1. Constant folding
2. Copy propagation
3. Strength Reduction
4. Dead code elimination
5. Common sub expression elimination.
6. Loop Optimization
7. Peephole Optimization

1. Constant Folding

(i) $x = 2 * 3 + y \Rightarrow x = 6 + y$

Folding

ii) $x = 2 + y * 3\}$ can't fold the constant

2. Copy Propagation

i) Variable Propagation:

$x = y;$

$z = y + 2;$

$\Rightarrow z = x + 2;$

ii) Constant Propagation:

$x = 3$

$z = 3 + a;$

$\Rightarrow z = x + a$

3. Strength Reduction:

Replace expensive statement / instruction with cheaper one.

(i) $x = 2 * y$ **costly** $\Rightarrow$ $x = y + y;$ **Cheap**

(ii) $x = 2 * y$ $\Rightarrow$ $x = y << 1;$ **Much Cheaper**

(iii) $x = y / 8$ $\Rightarrow$ $x = y >> 3;$

4. Dead Code Elimination:

```
x = 2; FALSE
if (x > 2)
        printf ("code"); DEAD CODE CAN BE REMOVED
else
        printf("optimization");
```

```
x = 2;
printf("optimization");
```

- Hence, above dead code never executes during execution. We can always delete such code.

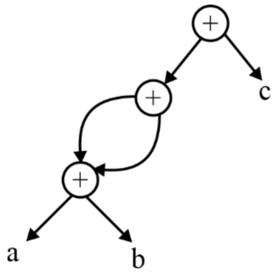
5. Common Sub Expression Elimination:

DAG is used to eliminate common sub expression.

Example: $x = (a + b) + (a + b) + c;$

$\Rightarrow t_1 = a + b$

$x = t_1 + t_1 + c$



6. Loop Optimization:

(i) Code Motion – Frequency Reduction:

Move the loop invariant code outside of loop.

```
for(i = 0; i < n; i++) {
    x = 10;        invariant  →   x = 10;
                                  for(i = 0, i < n; i++){
    y = y + i;                    y = y + i;
}                                 }
```

(ii) Induction Variable elimination:

```
i1 = 0;
i2 = 0;
for (i = 0; i < n; i++) {    →    for(i = 0; i < n; i++) {
    A[i1++] = B[i2++]                 A [i] = B [i];
                                  }
```

i, i_1, i_2 = 3 induction variable → only 1 induction variable: i

(iii) Loop Merging / Combining: (Loop Jamming)

```
3n + 2  for(i = 0; i < n; i++)          for(i = 0; i < n; i++) {
            A [i] = i + 1;       →          A [i] = i + 1
3n + 2  for(j = 0; i < n; j++)              B [i] = i - 1   4n + 2
            B [j] = j - 1;
6n + 4            reduced
```

(iv) Loop Unrolling:

1)
```
for (i = 0; i < 3; i++)          printf("CD");
    printf("CD");                printf("CD");
                                 printf("CD");
```

$3 \times 3 + 2 = 11$ Statements → 3 statements

2)
```
for (i = 0; i < 2n; i ++) {      for (i = 0; i < n; i++) {
    printf("CD");                    printf("CD");
}                                    printf("CD");
                                 }
```

$(2 \times 3n + 2) = 6n + 2$ → $(4n + 2)$

7. **Peephole Optimization:**

Examines a short sequence of target instructions in a window (*peephole*) and replaces the instructions by a faster and/or shorter sequence when possible.

- Applied to intermediate code or target code
- Following Optimizations can be used:

- Redundant instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

❑❑❑