UNIT 2 & 3

LR Parsers

SLR, CLR and LALR Parsers

Top-down parser

Non Recursive Predictive parser [LL(1) Parser]

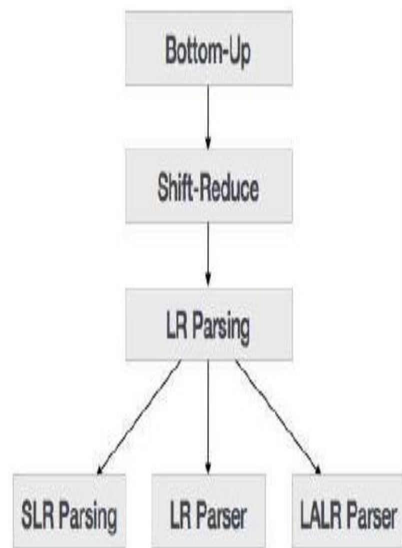
Parser generator tool

YACC – Yet Another Compiler Compiler

Intermediate Code generation

Intermediate Languages - Declarations - Assignment statement - Boolean expressions - Case statement – Back-patching - Procedure call - Type checking

LR Parsers



- The LR parser is a non-recursive, shift-reduce, bottom-up parser.
- It uses a wide class of context free grammar which makes it the most efficient syntax analysis technique.
- LR parsers are also known as **LR(k)** parsers,
  Where
  - L stands for left-to-right scanning of the input stream;
  - R stands for the construction of right-most derivation in reverse, and
  - K denotes the number of look-ahead symbols to make decisions.
- There are three widely used algorithms available for constructing an LR parser:
  1. **SLR(1) – Simple LR Parser**:
     - Works on smallest class of grammar
     - Few number of states, hence very small table
     - Simple and fast construction
  2. **CLR(1) – Canonical LR Parser**:
     - Works on complete set of LR(1) Grammar
     - Generates large table and large number of states
     - Slow construction
  3. **LALR(1) – Look-Ahead LR Parser**:
     - Works on intermediate size of grammar
     - Number of states are same as in SLR(1)

Organization of LR Parsers: The LR Parser consists of

Input Buffer – to hold the input string to be parsed

Stack – holds a sequence of states of LR Items set and the grammar symbols.

Driver Program – is the same for all LR Parsers. Only the parsing table changes from one parser to another.

Parsing table: The parsing table consists of two parts:

(1) a parsing function ACTION and

(2) a goto function GOTO.

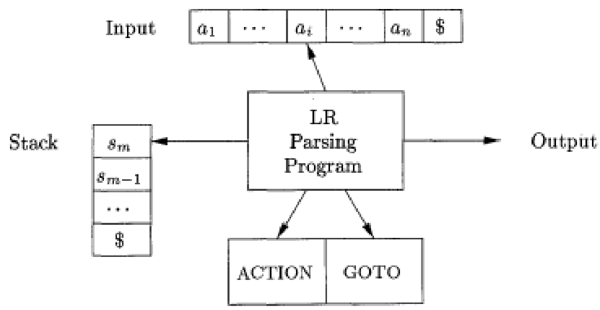


Figure 4.35: Model of an LR parser

1. The ACTION function takes as arguments a state *i* and a terminal *a* (or $, the input end marker). The value of ACTION[*i*, *a*] can have one of four forms:
    a. Shift *j*, where *j* is a state. The action taken by the parser effectively shifts input *a* to the stack, but uses state *j* to represent *a*.
    b. Reduce A → β. The action of the parser effectively reduces β on the top of the stack to head A.
    c. Accept. The parser accepts the input and finishes parsing.
    d. Error. The parser discovers an error in its input and takes some corrective action.
2. Extend the GOTO function, defined on sets of items, to states: if GOTO[I_i, A] = I_j, then GOTO also maps a state *i* and a non terminal A to state *j*.

SLR – Simple LR Parser

- SLR Grammar: A grammar for which an SLR parser can be constructed is said to be SLR Grammar.
- **Steps to construct the SLR Parser**
    1. Computation of LR (0) Items for the given grammar G
    2. Construction of SLR Parsing table using LR (0) items
    3. LR Parsing the Input String

LR (0) Item

- An LR Item of a grammar G is a production of G with a DOT at some position of the RHS.

    (1) A production A→ XYZ yields the four items

    A → .XYZ
    A → X.YZ
    A→ XY.Z
    A→ XYZ.

(2) A Production $A \rightarrow \epsilon$ generates only one item as $A \rightarrow .$

- LR (0) Items provides basis for constructing SLR Parser.

LR Items – Classification

The LR Items can be classified into two different classes as

Kernel Items – It include the Initial Item $S' \rightarrow S$ and all other items whose dots are not at the leftmost end.

Non-Kernel Items - It include items which have their dots at their leftmost end.

Computation of LR (0) Items – Steps

LR (0) Items can be computed using three steps

1. Augmented Grammar
2. Closure Function
3. GOTO Function

(i) Augmented Grammar: If G is a grammar with the start symbol S, then the augmented grammar G' for G is formed with new start symbol S' and production $S' \rightarrow S$.

Example:

G: $E \rightarrow E + E \mid id$

The augmented grammar

G': $E' \rightarrow E$

$E \rightarrow E + E \mid id$

Significance of Augmented Grammar:

- An Augmented Grammar indicates to the parser, when it should stop parsing and announce the acceptance of the inputs.
- The parsing process is completed when a parser is about to reduce $S' \rightarrow S$.

(ii) CLOSURE Function:

Let I is a set of Items for a grammar G, then Closure (I) is constructed from I by using two rules,

Initially every Item in I is added to CLOSURE (I)

If $A \rightarrow \alpha.B\beta$ is in CLOSURE (I) and $B \rightarrow \gamma$ is a production then Add item $B \rightarrow .\gamma$ into I, if it is not there.

(iii) GOTO Function:

The function GOTO (I, X), where X is a Grammar Symbol,

If $A \rightarrow \alpha.X\beta$ is in I then GOTO (I, X) defined as the CLOSURE ($A \rightarrow \alpha X.\beta$)

2. Construction of SLR Parsing table – Algorithm

Let C = {I_0, I_1, I_2, …I_n} be the collection of LR (0) Items for G' and Let I_j is a set in C,

(R1) If GOTO (I_j , a) = I_k then set ACTION [j,a] = SHIFT K (or) S_k

(R2) If GOTO (I_j , A) = I_k then set GOTO [j,A] = K

(R3) If A → α. is in set I_j then set ACTION [j,a] = REDUCE BY A → α for every symbol 'a' in FOLLOW (A)

(R4) If S' → S. is in set I_j then set ACTION [j,$] = ACCEPT

(R5) All the undefined entries are ERROR.

Example: Construct the Simple LR Parsing table for the following grammar S → CC, C → eC, C → d. And parse the string eded.

Given:

G: S → CC
C → eC
C → d

Computation of LR (0) Items

1. Augmented Grammar

G': S' → S
S → CC
C → eC
C → d

2. CLOSURE (S'→ S)

S' → .S
S → .CC
C → .eC
C → .d ………..(I_0)

3. GOTO (I_0,S)

S' → S. ………(I_1)

GOTO (I_0,C)

S → C.C
C → .eC
C → .d ………(I_2)

GOTO (I_0,e)

C → e.C
C → .eC
C → .d ………(I_3)

GOTO (I_0,d)

C → d. ………(I_4)

GOTO (I_2,C)

S → CC. ………(I_5)

GOTO (I_2,e)

C → e.C
C → .eC
C → .d ………(I_3)

GOTO (I_2,d)

C → d. ………(I_4)

GOTO (I_3,C)

C → eC. ………(I_6)

GOTO (I_3,e)

C → e.C
C → .eC
C → .d ………(I_3)

GOTO (I_3,d)

C → d. ………(I_4)

Construction of SLR Parsing Table

State	ACTION			GOTO	
	e	d	$	S	C
0	S_3	S_4		1	2
1			ACC		
2	S_3	S_4			5
3	S_3	S_4			6
4	R_3	R_3	R_3		
5	R_1	R_1	R_1		
6	R_2	R_2	R_2		

SLR Parsing

No	STACK	BUFFER	ACTION
1	0	eded $	ACTION [0,e] = S_3 Shift
2	0e3	ded $	ACTION [3,d] = S_4 Shift
3	0e3d4	ed $	ACTION [4,e] = R_3 Reduce by C → d
	0e3C6	ed $	Pop-off d4, Push C. GOTO(3,C)=6. Push 6
4	0e3C6	ed $	ACTION [6,e] = R_2 Reduce by C → eC
	0C2	ed $	Pop-off e3C6, Push C. GOTO(0,C)=2. Push 2
5	0C2	ed $	ACTION [2,e] = S_3 Shift
6	0C2e3	d $	ACTION [3,d] = S_4 Shift
7	0C2e3d4	$	ACTION [4,$] = R_3 Reduce by C → d
8	0C2e3C6	$	ACTION [6,$] = R_2 Reduce by C → eC
9	0C2C5	$	ACTION [5,$] = R_1 Reduce by S → CC
10	0S1	$	ACTION [1,$] = Accept

Problems on SLR Parsers – Please refer the Class Work Note-Book.

CLR – Canonical LR Parser

- The CLR parser incorporates extra information in the state by redefining items to include a terminal symbol as a look-ahead as a second component.
- The general form of an Item is

  $A \rightarrow \alpha.B\beta$ for SLR | LR (0) Item

  becomes

  $A \rightarrow \alpha.B\beta, a$

  Where a is a terminal or a right-end marker $, and the item is said to be LR (1) Item.
- **Steps to construct the SLR Parser**
  1. Computation of LR (1) Items for the given grammar G
  2. Construction of CLR Parsing table using LR (1) items
  3. Parsing the Input String

(ii) CLOSURE to compute LR(1) Item:

An Item of the form

[A → α.Bβ, a] in the set I and B→ γ is a production it can be added into I as

[A → α.Bβ, a]

[B→ .γ, b]

Where b = FIRST(**βa**).

Example: Construct the canonical LR Parsing table for the following grammar

S → CC, C → eC, C → d.

Computation of LR (1) Items

1. Augmented Grammar

G': S' → S
S → CC
C → eC
C → d

2. CLOSURE (S'→ S)

S' → .S, $
S → .CC, $
C → .eC, e|d
C → .d, e|d ....(I_0)

3. GOTO (I_0,S)

S' → S., $ ....(I_1)

GOTO (I_0,C)

S → C.C, $
C → .eC, $
C → .d, $ ....(I_2)

GOTO (I_0,e)

C → e.C, e|d
C → .eC, e|d
C → .d, e|d ....(I_3)

GOTO (I_0,d)

C → d., , e|d ....(I_4)

GOTO (I_2,C)

S → CC., $ ....(I_5)

GOTO (I_2,e)

C → e.C, $
C → .eC, $
C → .d, $ ....(I_6)

GOTO (I_2,d)

C → d., $ ....(I_7)

GOTO (I_3,C)

C → eC. , e|d ....(I_8)

GOTO (I_3,e)

C → e.C, e|d
C → .eC, e|d
C → .d, e|d ....(I_3)

GOTO (I_3,d)

C → d. , e|d ....(I_4)

GOTO (I_6,C)

C → eC., $ ....(I_9)

GOTO (I_6,e)

C → e.C, $
C → .eC, $
C → .d, $ ....(I_6)

GOTO (I_6,d)

C → d., $ ....(I_7)

Construction of CLR Parsing Table

State	ACTION			GOTO	
	e	d	$	S	C
0	S_3	S_4		1	2
1			ACC		
2	S_6	S_7			5
3	S_3	S_4			8
4	R_3	R_3			
5			R_1		
6	S_6	S_7			9
7			R_3		
8	R_2	R_2			
9			R_2		

More Problems on CLR Parsers – Please refer the Class Work Note-Book.

LALR – Look-ahead LR Parser

- From the LR (1) Items of G' of G,
  - Take a pair of similar looking states; say I_j and I_k each of these states are differentiated only by the look-ahead symbols.
  - Replace I_j and I_k by I_{jk}, the union of I_j and I_k
  - The Goto's on any symbol X to I_j or I_k from any other states now replaced with I_{jk}.

Example: Construct the canonical LR Parsing table for the following grammar
S → CC, C → eC, C → d. And parse the string eed.

From Collection of LR (1) Items computed in CLR parser

State I_3 and I_6 are differentiated only by their look-ahead symbols,

(i) C → e.C, e|d
C → .eC, e|d
C → .d, e|d ....(I_3)
and
C → e.C, $
C → .eC, $
C → .d, $ ....(I_6)
becomes
C → e.C, $|e|d
C → .eC, $|e|d
C → .d, $|e|d ....($I_{36}$)

(ii) C → d., e|d_.........(I_4)
and
C → d., $ ....(I_7)
becomes
C → d., $|e|d ....($I_{47}$)

(iii) C → eC., e|d_.........(I_8)
and
C → eC., $ ....(I_9)
becomes
C → eC., $|e|d ....($I_{89}$)

Construction of LALR Parsing Table

State	ACTION			GOTO	
	e	d	$	S	C
0	**S_{36}**	**S_{47}**		**1**	**2**
1			**ACC**		
2	**S_{36}**	**S_{47}**			**5**
36	**S_{36}**	**S_{47}**			**89**
47	**R_3**	**R_3**	**R_3**		
5			**R_1**		
89	**R_2**	**R_2**	**R_2**		

LALR Parsing

No	STACK	BUFFER	ACTION
1	0	eed $	ACTION [0,e] = S_{36} Shift
2	0e36	ed $	ACTION [36,e] = S_{36} Shift
3	0e36e36	d $	ACTION [36,d] = S_{47} Shift
4	0e36e36d47	$	ACTION [47,$] = R_3 Reduce by C → d
5	0e36e36C89	$	ACTION [89,$] = R_2 Reduce by C → eC
6	0e36C89	$	ACTION [89,$] = R_2 Reduce by C → eC
7	0C2	$	ACTION [2,$] = Undefined, Error

Top Down Parser – Non Recursive Predictive Parser / LL (1) Parser

- Top Down parsing can be viewed as the process of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder.
- Top-down parsing can also be viewed as finding a leftmost derivation for an input string.

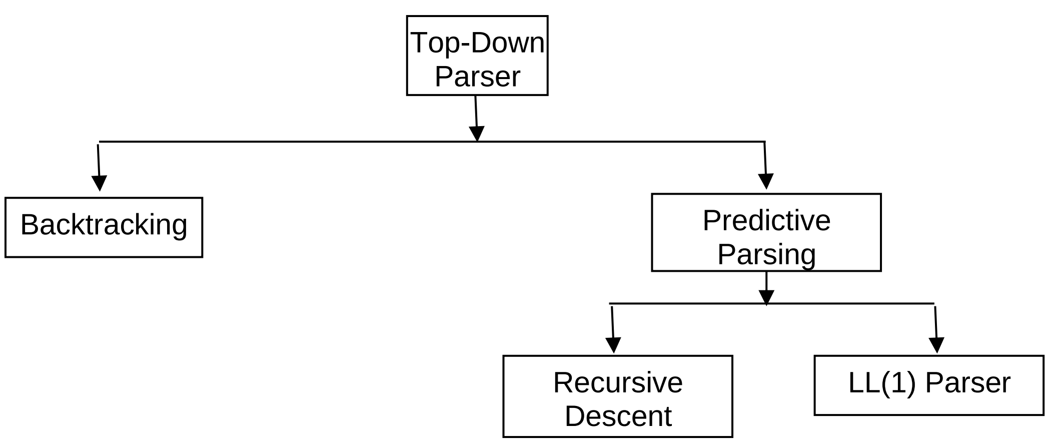


Fig 3.2: Types of Top-Down Parsers

Construction of Predictive LL (1) Parser

1. Before constructing the Predictive LL (1) parsers we have to
   a. Eliminate ambiguity,
   b. Eliminate left-recursion and
   c. Perform left factor where required.
2. For construction of Predictive LL (1) parser, follow the following steps:
   a. Computation of FIRST
   b. Computation of FOLLOW
   c. Construct the Predictive Parsing Table using FIRST and FOLLOW sets
3. Parse the input string

Computation of FIRST: The Computation of FIRST allows the parser to choose which production to apply, based on the first input symbol.

Definition – FIRST: FIRST (α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α.

Algorithm

(R1) If x is terminal then **FIRST(x) = {x}**

(R2) If there is a production A → ε, then **FIRST (A) = {ε}**

(R3) If A → $X_1X_2X_3 \ldots X_k$ is a production, where X is a grammar symbol, then

If FIRST(X1) ≠ {ε} then **FIRST (A) = FIRST(X1).**

If FIRST(X1) = {ε} then FIRST (A) = FIRST(X2) − {ε}.

If FIRST(X1) = {ε} and FIRST (X2) = {ε} then FIRST(A) = FIRST(X3) − {ε}.

…

If ε is in the FIRST set for every X_k then FIRST (A) = ε

Computation of FOLLOW: Follow (A) for a non-terminal A, to be the set of terminals a such that can appear immediately to the right of A in some sentential form.

Algorithm

(R1) Include **$ in the Follow (S)**, where S is the start symbol of the grammar

(R2) If A → αBβ, then **FOLLOW (B) = FIRST (β) except ϵ**

(R3) If (A → αB) or (A → αBβ and FIRST (β) has ϵ), then **FOLLOW (B) = FOLLOW (A)**

Construction of Parsing Table – Algorithm

(R1) For each terminal 'a' in FIRST (α), **add A → α to M [A, a]**

(R2) If ϵ is in FIRST (α), then **add A→ ϵ to M [A, b] for every symbol 'b' in FOLLOW (A).**

(R3) If ϵ is in FIRST (α) and $ is in FOLLOW (A) then **Add A→ α to M [A, $].**

(R4) Make all undefined entries of M be **ERROR.**

<u>Parsing the Input string - Algorithm</u>

Let X is the symbol on top of the stack and ‘a’ is the current input symbol,

(R1) If X is a terminal and X=a=S, then the parser halts and announce the successful completion of parsing.

(R2) If X is a terminal and X=a≠S, then Pop-off X from the stack and advance the input pointer.

(R3) If X is a Non-terminal then consult an entry M [X, a] of the parsing table,

- If M [X, a] = {X → UVW}, then the parser replaces X on top of the stack by UVW, with U on top of the stack.
- If M [X, a] = Undefined, then the parser calls an error recovery routine.

<u>Example</u>:

Construct the LL (1) parsing table for the following grammar and parse the string id + id * id.

E → E + T | T

T → T * F | F

F → (E) | id

Given:

G: E → E + T | T

T → T * F | F

F → (E) | id

<u>Step -1: Elimination of Left Recursion</u>

After Eliminating the Left Recursion from E and T productions,

E→TE'

E'→+TE' | ϵ

T→ FT'

T'→*FT' | ϵ

F→(E) | id

<u>Step -2: Computation of FIRST</u>

<u>FIRST (E)</u>

E → TE’

First (E) = First (T)……….(1)

<u>FIRST (E’)</u>

E'→+TE' E'→ ϵ

First (E’) = {+, ϵ}

<u>FIRST (T)</u>

T → FT’

First (T) = First (F)……….(2)

FIRST (T’)

T'→*FT'   T'→ ϵ

First (T’) = {*, ϵ}

FIRST (F)

F→(E)   F→ id

First (F) = {(, id}

Substituting First (F) in (2)

First (T) = **First (F) = {(, id**}

Substituting First (T) in (1)

First (E) = First (T) = **First (F) = {(, id**}

Step -3: Computation of FOLLOW

FOLLOW (E)

FOLLOW (E) = {$} by [R1]

F→(E)

FOLLOW (E) =First ()) = {$,)} by [R2]

FOLLOW (E’)

E→TE'

Follow(E’) = Follow(E) by [R3] ={$,)}

E'→+TE'

Follow(E’) = Follow(E’) by [R3] ={$,)}

FOLLOW (T)

E→TE'

Follow(T) = First (E’) Except ϵ by (R2) = {+}

Follow(T) = Follow (E) Since First(E’) has ϵ

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

E'→+TE'

Follow (T) = Follow (E’) by [R3] = {+,$,) }

FOLLOW (T’)

T→ FT'

Follow(T’) = Follow (T) by [R3] = {+,$,) }

T'→*FT'

Follow(T’) = Follow(T’) by [R3] = {+,$,) }

FOLLOW (F)

T→ FT'

Follow(F) = First (T’) Except ϵ by (R2) = {*}

Follow(F) = Follow (T) Since First(E’) has ϵ

Follow (F) = Follow(T) = {+,$,) }

T'→*FT'

Follow (F) = Follow (T’) by [R3] = {*,+,$,) }

Symbol	FIRST	FOLLOW
E	(, id	$,)
E’	+,ϵ	$,)
T	(, id	+, $,)
T’	*, ϵ	+, $,)
F	(, id	*, +, $,)

Step – 4: Construction of Parsing Table

M	id	+	*	(	)	$
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→ ϵ	E’→ ϵ
T	T→FT’			T→FT’		
T’		T’→ ϵ	T’→*FT’		T’→ ϵ	T’→ ϵ
F	F→id				F→(E)	

Step 5: Parsing the String id+id*id

Stack	Input buffer	Action taken
$E	id+id*id$	M[E,id] = E → TE’
$E'T	id+id*id$	M[T,id] = T → FT’
$E'T'F	id+id*id$	M[F,id] = F → id
$E'T'~~id~~	~~id~~+id*id$	Pop off & Advance the Input pointer
$E'T'	+id*id$	M[T’,+] = T’ → ϵ
$E'	+id*id$	M[E’,+] = E’→+TE’
$E'T~~+~~	~~+~~id*id$	Pop off & Advance the Input pointer
$E'T	id*id$	M[T,id] = T → FT’
$E'T'F	id*id$	M[F,id] = F → id
$E'T'~~id~~	~~id~~*id$	Pop off & Advance the Input pointer
$E'T'	*id$	M[T’,*] = T’→*FT’
$E'T'F~~*~~	~~*~~id$	Pop off & Advance the Input pointer
$E'T'F	id$	M[F,id] = F → id
$E'T'~~id~~	~~id~~$	Pop off & Advance the Input pointer
$E'T'	$	M[T’,$] = T’→ ϵ
$E'	$	M[E’,$] = E’→ ϵ
$	$	Accepted

More problems on LL (1) Parser: please refer the class work note-book.

Intermediate Code Generation

- ICG is the final phase of the compiler front-end.
- It translates the program into a format expected by the compiler back-end
- In typical compilers: ICG followed by code optimization and machine code generation
- Techniques for intermediate code generation can be used for final code generation

Why use an intermediate representation?

- It’s easy to change the source or the target language by adapting only the front-end or back-end (**portability**)
- It makes **optimization** easier: one needs to write optimization methods only for the intermediate representation.
- The intermediate representation can be directly interpreted.

How to choose the intermediate representation?

- It should be easy to translate the source language to the intermediate representation.
- It should be easy to translate the intermediate representation to the machine code.
- The intermediate representation should be suitable for optimization.
- It should be neither too high level nor too low level.
- One can have more than one intermediate representation in a single compiler.

General forms of Intermediate Representations (IR)

1. Graphical IR (parse tree, abstract syntax trees, DAG)
2. Linear IR (POSTFIX)
3. Three Address Code (TAC) - instructions of the form "**result = op1 operator op2**"

Three Address Code Statements

- Three Address code is a sequence of statements of the general form **x = y op z**, where x, y and z are names, constants and op stands for operator.
- In TAC, no multiple arithmetic expressions are permitted. Each statement contains almost three addresses, two for operands and one for the result.
- For example, the statement a=b + c * d become

  t1 = c *d

  t2 = b + t1

  a = t2

Types of TAC Statements

Type	General form
Assignment statement	**x = y op z**, where op – arithmetic / logical operator
Unary assignment	**x:=op y**, where op is any unary operator
Copy statement	**x = y**
Unconditional Jump	**goto L**, where L is a label
Conditional Jump	**if x relop y goto L**, where relop is any relational operator & L is a label
Function call / procedure call statement	For p (a_1,a_2,…a_n); **param a_1** **param a_2** **…** **param a_n** **call p, n** where a_i - argument, p – function name, n – no. of arguments
Indexed statement	**x = y[i]** and **x[i] = y**
Address & Pointer Assignment statement	**x = &y, x =*y** and ***x = y**

Implementation of Three Address Code (TAC) Statements

The TAC statements are implemented as RECORD structure with fields as arguments and operators.

There are three kinds of representations as

1. **Quadruples**
2. **Triples**
3. **Indirect Triples**

Quadruples

A Quadruple is a **record structure with four fields** as

op	arg1	arg2	result

- The contents of the fields arg1, arg2 and result are normally a pointer to the Symbol Table entries for the names represented by these fields.
- **Drawbacks**: Entering compiler generated temporaries into the symbol table requires additional memory , which leads to high space complexity.
- **Example**: The TAC sequence for the statement a=b + c * d

t1 = c *d

t2 = b + t1

a = t2

op	arg1	arg2	result
*	c	d	t1
+	b	t1	t2
ASSIGN	t2	--	a

Triples

To avoid entering the compiler generated temporaries into the symbol table, a temporary value is referred by the position of the statement that computes it.

In triples the TAC statements can be represented as record with three fields: op, arg1 and arg2.

op	arg1	arg2

Example: The TAC sequence for the statement a=b + c * d

1. t1 = c *d

2. t2 = b + t1

3. a = t2

Statement Position	op	arg1	arg2
1	*	c	d
2	+	b	[1]
3	ASSIGN	a	[2]

In triples, the array references requires two entries, Triples for statement x[i] = y which generates two records is as follows

Statement Position	op	arg1	arg2
0	[]=	x	i
1	ASSIGN	[0]	y

Similarly Triples for statement y = x[i] which generates two records is as follows

Statement Position	op	arg1	arg2
0	=[]	x	i
1	ASSIGN	y	[0]

Indirect Triples

In Indirect Triples, list the pointer to triples rather than listing the triples themselves.

Example: The TAC sequence for the statement a=b + c * d

1. t1 = c *d

2. t2 = b + t1

3. a = t2

Statement Position	Pointer
1	100
2	103
3	106

Address	op	arg1	arg2
100	*	c	d
103	+	b	[1]
106	ASSIGN	a	[2]

Comparison of TAC Statements

1. **By Storage Requirement**: Bothe Quadruples and Indirect Triples requires same amount of memory but using triples can save the memory.
2. **Support for Code Optimization:** Quadruples find best usage in an optimizing compiler where statements are freely moved around. If the statement computing x is moved, the statement using the value of x requires no change.
   A statement can be moved by recording the statement list in Indirect Triples, whereas in Triples moving a statement requires changes in all references to that temporary.

Semantic Analysis

- **Semantic analysis**, also context sensitive **analysis**, is a process in compiler construction, usually after parsing, to gather necessary **semantic** information from the source code.
- It usually includes type checking, scope resolution, array index bound checking or makes sure a variable is declared before use which is impossible to detect in parsing.

- Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other.
- Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.
- Lexical Analysis and Syntactic Analysis produce a “context free” analysis of the input, whereas the Semantic Analysis performs context sensitive checks.

CFG + semantic rules = Syntax Directed Definitions

Semantic Error: Some of the semantic errors that the semantic analyzer is expected to recognize:
1. Type mismatch
2. Undeclared variable
3. Reserved identifier misuse.
4. Multiple declaration of variable in a scope.
5. Accessing an out of scope variable.
6. Actual and formal parameter mismatch.

Attribute Grammar
- Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.
- Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.
- Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language.
- Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

There are two notations used for associating semantic rules with productions,
1. **SDD** – Syntax Directed Definition
2. **SDT** – Syntax Directed Translation

SDD – Syntax Directed Definition: A SDD is a generalization of CFG in which each grammar symbol has an associates set of attributes partitioned into two subsets called the synthesized attributes and inherited attributes of the grammar symbol.

SDT – Syntax Directed Translation: A SDT is a notation used to attach the semantic action to the production rules of the grammar.

SDT for Declaration statement

- For every declaration, it is necessary to lay out storage for the declared variables.
- For every local name in a procedure, needs to create a ST(Symbol Table) entry containing:
    - The type of the name
    - How much storage the name requires
    - A relative offset from the beginning of the static data area or beginning of the activation record.
- To keep track of the current offset into the static data area, the compiler maintains a global variable called OFFSET.
- OFFSET is initialized to 0 when compilation begins.
- After each declaration, OFFSET is incremented by the size of the declared variable

Variable:

Offset - a variable that keep track of next available relative address.

Attributes:

T. Type – defines the type of a data object, like integer, float, etc.

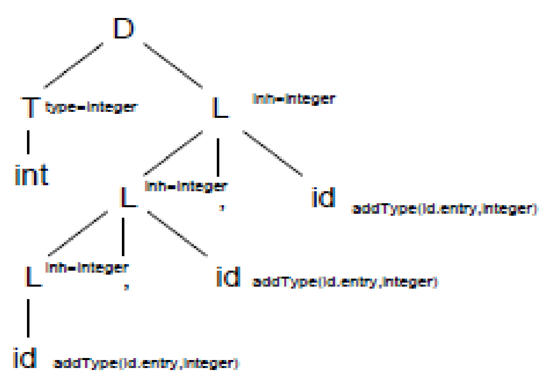
T. width – defines the amount of memory units taken by the data object.

Routines:

Enter (name, type, offset) - This procedure should create an entry in the symbol table, for Variable *name*, having its type set to type and relative address *offset* in its data area.

Grammar	Semantic Rules
D → T L	{ L.type = T.type; Offset =0;}
T → int	{T.type = integer; T.width = 2; offset = offset + T.width;}
T → float	{T.type = float; T.width = 4; offset = offset + T.width;}
L → L1, id	{L1.type = L.type; enter(id.name, L.type, offset)}
L → id	{enter(id.name, L.type, offset)}

Example: int a, b, c;



Symbol Table Entry ..

Name	type	width	offset	…
…	…	…	…	…
a	integer	2	0	…
b	Integer	2	2	…

C	Integer	2	4	…
…	…	…	…	…

SDT for Assignment Statement

- The Syntax Directed Translation to generate the TAC statements for the an assignment statement includes the following attributes and routines,

Attributes:

E. Place – refers to the address that will hold the value of E

Routines:

Emit () – routine to generate the TAC statement to an output file. For example the notation emit(x '=' y '+' z) represent the TAC instruction x=y+z. Here the expression appearing in place of variables like x,y and z are evaluated when passed to emit and quoted strings like '=' are taken literally.

Look-up (name) – This routine checks is there is an entry for the occurrence of the name in the symbol table. If so, a pointer to the entry is returned, otherwise returns NULL.

NewTemp() – routine to create a new compiler-generated temporary.

Grammar	Semantic Rules
S → id = E	{P = LOOKUP (id.name) If P ≠ NULL then EMIT(P '=' E.place); else ERROR; }
E → E_1 + E_2	{ E. Place = NEWTEMP(); EMIT (E.place '=' E_1.place '+' E_2.place); }
E → E_1 * E_2	{ E. Place = NEWTEMP(); EMIT (E.place '=' E_1.place '*' E_2.place); }
E → - E_1	{ E. Place = NEWTEMP(); EMIT (E.place '=' 'UMINUS' E_1.place); }
E → (E_1)	{ E. Place = E_1.place ; }
E → id	{P = LOOKUP (id.name) If P ≠ NULL then E.place = P; else ERROR; }
E → num	{E.place = num.val; }

Example: please refer Class work Note-book.

SDT for Array References

The elements of an array are stored in a block of consecutive locations.

Let

A be a Single Dimensional Array, **W** be the width of an array element

The **i^{th}** entry of A is in location

$$A[i] = base + (I - low) * w$$

Where **low** – lower bound on the array subscript

Base – base address of the array (ie) relative address of A [low].

The same expression can be rewritten as

$$A[i] = i * w + C_A \qquad \text{where } C_A = (base - low * w)$$

*The Sub-expression C_A can be evaluated when the declaration of an array is seen and the value of C_A is kept available in the symbol table entry for A. Hence the relative address for A[i] is obtained by simple adding i*w to C_A.*

Example: For the assignment B[i] = 100, in TAC form,

$t_1 = i*w$

$C_B [t_1] = 100$

Grammar	Semantic Rules
S → L = E	{ if L.offset= NULL Emit(L.place '=' E.place); else Emit(L.place '[' L.offset']' '=' E.place); }
L → id [num] | id	{ L.place=NEWTEMP(); L.offset=NEWTEMP(); Emit(L.Place '=' $C_{id.arrayname}$); Emit(L.offset'='num*val '*' WIDTH(id.type)); }
E → L	{ E.place=L.place; Emit(E.place'='L.place '['L.offset']'); }
E → E_1 + E_2	{ E. Place = NEWTEMP(); EMIT (E.place '=' E_1.place '+' E_2.place); }
E → E_1 * E_2	{ E. Place = NEWTEMP(); EMIT (E.place '=' E_1.place '*' E_2.place); }
E → - E_1	{ E. Place = NEWTEMP(); EMIT (E.place '=' 'UMINUS' E_1.place); }
E → (E_1)	{ E. Place = E_1.place ; }
E → id	{P = LOOKUP (id.name) If P ≠ NULL then E.place = P; else ERROR; }
E → num	{E.place = num.val; }

Example: TAC sequence for a[i] = b*c +c*d as per the above semantic rules will be,

$\mathbf{t_1 = b * c}$

$\mathbf{t_2 = c * d}$

$\mathbf{t_3 = t_1 + t_2}$

t_4 = i * w

C_a [t4] = t3

A [1] [1]	A [1] [2]	A [1] [3]
A [2] [1]	A [2] [2]	A [2] [3]

<u>SDT for Multi-dimensional Array References</u>

The Two-dimensional array is normally stored in one of the two forms,

1. Row-major ordering
2. Column-major ordering

Row-Major ordering of an array A [2] [3]

A [1] [1]
A [1] [2]
A [1] [3]
A [2] [1]
A [2] [2]
A [2] [3]

Column-Major ordering of an array A [2] [3]

A [1] [1]
A [2] [1]
A [1] [2]
A [2] [2]
A [1] [3]
A [2] [3]

A [1] [1]	A [1] [2]	A [1] [3]
A [2] [1]	A [2] [2]	A [2] [3]

The Relative address of A [i] [j] can be calculated as

A[i] [j] = base + [(i – low1) *n2 + (j-low2)] * w, for Row-major ordering

whereas

A[i] [j] = base + [(j – low2) *n1 + (i-low1)] * w, for Column-major ordering

Where

low1 & low2 are the lower bounds on the array subscript

Base – base address of the array (ie) relative address of A [low1] [low2].

n1 & n2 – No of elements in dimension 1 and 2 respectively.

The Compile-time pre-calculation is also being applied to address calculation of multi-dimensional arrays. Hence the same expression can be rewritten as

A[i] [j] = (((i * n2) + j) * w) + C_A where C_A = base – (((low1*n2) + low2)*w)

<u>*The Sub-expression C_A can be evaluated when the declaration of an array is seen and the value of C_A is kept available in the symbol table entry for A. Hence the relative address for A[i][j] is obtained by adding t3 to*</u> <u>C_A as</u>

t1 = i *n2

t2 = t1 + j

t3 = t2 *w

Example: For the assignment B[i] [j] = 100, in TAC form,

t1 = i *n2

t2 = t1 + j

t3 = t2 *w

C_B [t_3] = 100

The Syntax Directed Translation to generate the TAC statements for the an array reference within an assignment statement includes the following attributes and routines,

<u>Attributes:</u>

L. place – refers to the address that will hold the value of L

L. offset – If L-value is a simple name (id) then L.offset is NULL. If it is an array reference it contains the relative address of the array.

EListL. array – represents an array & a pointer to the symbol table entry for that array.

EListL.ndim – records the number of the dimensions in the array.

EListL. place – refers to the address that will hold the value of EList

<u>Routines:</u>

Emit () – routine to generate the TAC statement to an output file.

Limit (array,j) – returns N_j – the size of the j^{th} dimension of the array.

Width (array) – returns the width of an array element.

The Grammar is generalized to support multi-dimensional array as A [n_1,n_2,n_3,….,n_k]

Grammar	Semantic Rules
S → L = E	{ if L.offset= NULL then Emit(L.place '=' E.place); else Emit(L.place '[' L.offset']' '=' E.place); }
E → L	{ if L.offset = NULL then E.place=L.place; else E.place = NEWTEMP(); Emit(E.place'='L.place '['L.offset']'); }
L → EList]	{ L.place=NEWTEMP(); L.offset= NEWTEMP(); Emit(L.place '=' C(EList.array)); Emit(L.offset'=' EList.place '*' Width(Elist.array);}

EList → EList$_1$, E	t=NEWTEMP();
	m= EList$_1$.ndim +1;
	Emit(t'=' EList$_1$.place '*' Limit(EList$_1$.array,m));
	Emit(t'=' t '+' E.place);
	EList.array = EList$_1$. array;
	EList.place = t; EList.ndim = m; }

EList → id[E	{ EList.array = id.place; EList.place = E.place; EList.ndim = 1; }
E → E$_1$ + E$_2$	{ E. Place = NEWTEMP(); EMIT (E.place '=' E$_1$.place '+' E$_2$.place); }
E → E$_1$ * E$_2$	{ E. Place = NEWTEMP(); EMIT (E.place '=' E$_1$.place '*' E$_2$.place); }
E → - E$_1$	{ E. Place = NEWTEMP(); EMIT (E.place '=' 'UMINUS' E$_1$.place); }

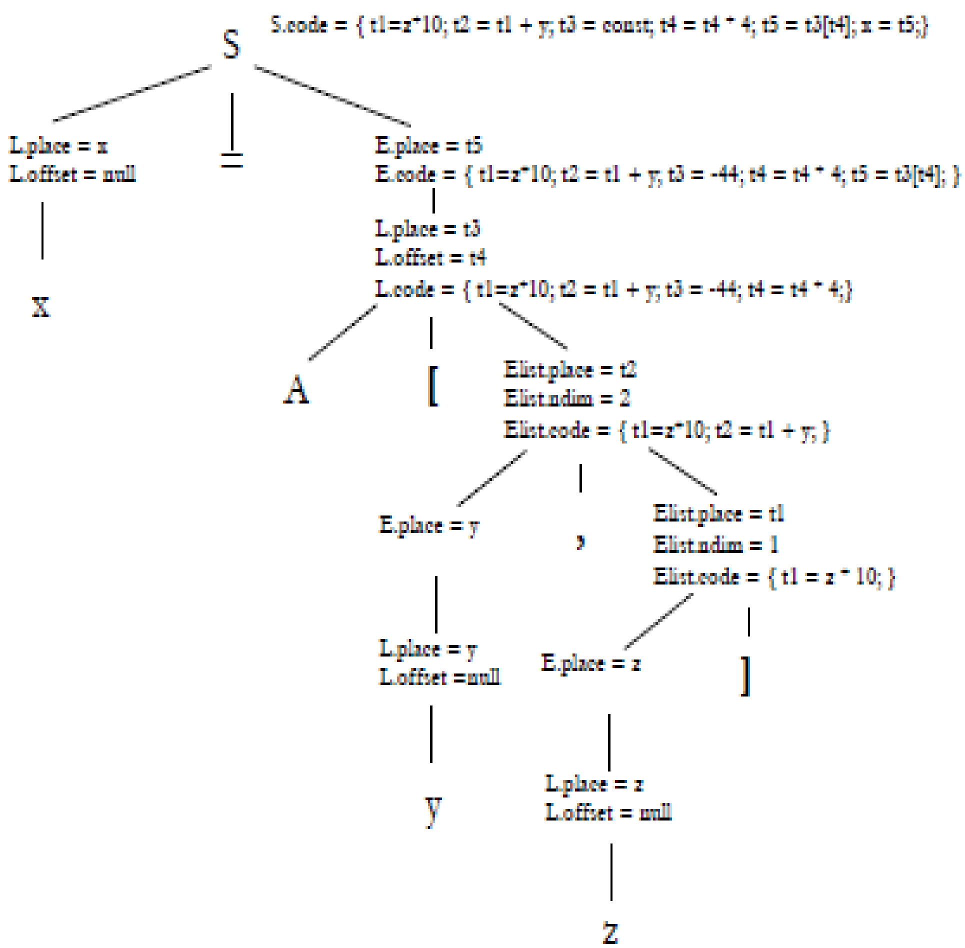
E → (E_1)	{ E. Place = E_1.place ; }
E → id	{P = LOOKUP (id.name) If P ≠ NULL then E.place = P; else ERROR; }
E → num	{E.place = num.val; }

Similarly the TAC Instruction for 3 – dimensional arrays can be generated as

A[i] [j] [k] = ((((i * n2) + j) * n3) * w) + C_A

And the TAC Sequence will be,

t1 = i * n2
t2 = t1 + j
t3 = t2 * n3
t4 = t3 * w
C_A [t4]



SDT for Type Conversion

- In order to facilitate the mixed-type arithmetic operations, the compiler generates appropriate type-conversion instruction or rejects the operation.
- The Syntax Directed Translation to generate the TAC statements for type conversions within an assignment statement includes the following attributes and routines,

  E. type – specify the type of the data item in E

  Int2Float () – routine to convert Integer to Float type.

Grammar	Semantic Rules

$E \rightarrow E_1 + E_2$	{ E. place = NEWTEMP();
	if (E_1.type =**Integer** and E_2.type =**Integer**)
	E.type =**Integer;**
	Emit (E.place '=' E_1.place 'int+' E_2.place);
	else if (E_1.type = **Float** and E_2.type =**Float**)
	E.type =**Float;**
	Emit (E.place '=' E_1.place 'float+' E_2.place);
	else if (E_1.type = **Integer** and E_2.type =**Float**)
	u=NEWTEMP();
	Emit (u '=' 'Int2Float' E_1.place);
	E.type =**Float;**
	Emit (E.place '=' u 'float+' E_2.place);
	else if (E_1.type = **Float** and E_2.type = **Integer**)
	u=NEWTEMP();
	Emit (u '=' 'Int2Float' E_2.place);
	E.type =**Float;**
	Emit (E.place '=' E_2.place 'float+' u);
	else
	E.type = Type-Error();

Example: Let x,y are float and I,j are integer, generate the TAC instruction for x = y+i*j

t1 = i 'int*' j
t2 = 'Int2Float' t1
t3 = y 'Float+' t2
x = t3

Type checking

- **Static:** Done during compilation time. This reduces the run time of the program and the code generation is also faster.
- **Dynamic:** Done during run time. Due to this the code gets inefficient and it also slows down the execution time. But it adds to the flexibility of the program.

Types:

1. Basic Types: int, real, bool, char.
2. Arrays: as Array (length , type).
3. Function Arguments: as T1 x T2 x T3 x ……… x Tn.
4. Pointer: as Pointer (T) .
5. Named Record : If there is a structure defined as :-
   struct record
   {
   int length;
   char word[10];
   };
   Then its type will be constructed as….
   (length x integer) x (word x array(10,char))

Consider a simple C language:

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

Corresponding **Semantic actions**:

T →char	{T.type=char}
T → integer	{T.type = integer}
D→id : T	{ AddEntry (id.entry,T.type)}
T→array [num] of T1	{T.type = Array(num,T1.type)}
E→literal	{ E.type = char }
E→num	{ E.type = iniger }
E→id	{E.type = lookup (id.type)}
E→E1mod E2	{if(E1.type==int) && (E2.type == int) then E.type = int else type error }
E →E1 [E2]	{if(E2.type == int & E1.type == array(s,t)) then E.type = t else type error }

SDT for SWITCH – CASE statement

- In the Switch statement, a selector expression is to be evaluated followed by n constant values that the expression might take including a default value, which always matches if no other value does.

Syntax	Semantic Rules
Switch (Expr) {	Code to evaluate E into 't'
case v_1:	goto **TEST**
S_1;	**L1:** code for S_1
break;	goto **NEXT**
case v_2:	**L2:** code for S_2
S_2;	goto **NEXT**
break;	…
…	**Ln-1:** code for S_{n-1}
case v_{n-1}:	goto **NEXT**
S_{n-1};	**Ln:** code for S_n
break;	goto **NEXT**
default:	**TEST:** if t=1 then goto L1
S_n;	if t=2 then goto L2
break;	…
}	if t=(n-)1 then goto Ln-1
	else goto Ln

SDT for Procedure Call statement

- The translation for procedure call statement includes a calling sequence and return sequence.
- **Calling sequence**- series of actions to be taken while entering into the procedure definition.
- **Return sequence** - series of actions to be taken while leaving from the procedure definition

Grammar	Semantic Rules
D → T id (F) { S }	{ For each item P on QUEUE Emit('PARAM' P); Emit (Call id.name, n);}
F → T id, F | ϵ	{Append id.place to the end of the QUEUE; }

Example: Generate the TAC instruction for n = f (a[i]);

t1 = i *w
t2 = a[t1]
PARAM t2
t3 = CALL f,1
n = t3

SDT for BOOLEAN EXPRESSIONS

- Boolean expressions are composed of the Boolean operators (*and*, *or* and *not*) applied to the elements that are Boolean variables or relational expressions.
- Example: **a *or* b *and not* c**

  t_1 = *not* c
  t_2 = b *and* t_1
  t_3 = a *or* t_2
- There are two methods of translating Boolean Expression,
  1. Numerical Representation
  2. Flow-of-control representation

1. Numerical Representation:

- Encode TRUE as 1 and FALSE as 0 and to evaluate a boolean expression.
- An expression can be evaluated completely from left to right.
- **Routines:**

  NEWTEMP() – used to generate the new compiler-generated temporary.

  EMIT() - used to generate a TAC Instruction

  NEXTSTAT – Gives the index of the next TAC Instruction in the output sequence and every call to EMIT() increments NEXTSTAT by 1.

Grammar: E → E *or* E | E *and* E | *not* E | (E) | id_1 *relop* id_2 | true | false

Grammar	Semantic Rules
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* E	{ E.place := newtemp(); EMIT (E.place ':=' '*not*' E.place); }
E→(E1)	{ E.place := E1.place; }
E →*id1 relop id2*	{ E.place := newtemp; EMIT ('if' *id1*.place *relop*.op *id2*.place '*goto*' nextstat+3); EMIT (E.place ':=' '0'); EMIT ('*goto*' nextstat+2);

	EMIT (E.place ‘:=’ ‘1’); }
E → *true*	{ E.place := newtemp; EMIT (E.place ‘:=’ ‘1’); }
E → *false*	{ E.place = newtemp; emit (E.place ‘:=’ ‘0’); }

Example1: a<b or c<d using numerical representation.

1000 if a<b goto 1003
1001 t1=0
1002 goto 1004
1003 t1=1
1004 if c<d goto 1007
1005 t2=0
1006 goto 1008
1007 t2=1
1008 t3 = t1 OR t2

Example2: *Translation of a<b or c<d and e<f*

100 if a<b goto 103
101 t1 := 0
102 goto 104
103 t1 := 1
104 if c<d goto 107
105 t2 := 0
106 goto 108
107 t2 := 1
108 if e<f goto 111
109 t3 := 0
110 goto 112

2. Flow-of-Control Representation:

- Control flow translation of Boolean expression representing the value of a Boolean expression by apposition reached in a program.

Attributes:

1. B.True – refers to the label to which control flows if B is true.
2. B. False – refers to the label to which control flows if B is fasle.
3. B.code – refers the sequence of TAC statements in B

Routines:

1. NewLabel () – returns a new label each time it is called.
2. Gen() – attach the label to the TAC instruction.

Grammar	**Semantic Rules**
E→ E1 *or* E2	{E1. true = E.True E1.fasle = NewLabel(); E2.true = E.true E2.false=E.false E.code =E1.code || gen(E1.false)||E2.code }
E → E1 *and* E2	{ E1.true =NewLabel(); E1.false = E.false E2.true = E.true E2.false = E.false E.code = E1.code ||gen(E1.true) || E2.code }

E → *not* E1	{ E.true = E1.false E.false = E1.true E.code = E1.code }
E →*id1 relop id2*	{E.code = E1.code || E2.code || EMIT ('if' *id1*.place *relop*.op *id2*.place '*goto*' E.true); EMIT ('*goto*' E.false);
E → *true*	{ E.code = EMIT (goto E.true); }
E → *false*	{ E.code = EMIT (goto E.false); }

Example*:* *a<b or c<d* using flow of control representation

```
        if a<b goto Ltrue
        goto L1
L1:     if c<d goto Ltrue
        goto Exit
Ltrue:
```

Example2: *Translation of a<b or c<d and e<f*

```
        if a<b goto L1
        goto L2
L2:     if c<d goto L1
        goto Exit
L1:     if e<f goto Ltrue
        goto Exit
```

SDT for Flow of Control Statements

The translation of Boolean expressions into TAC in the context of statement are generated by

S → if (B) S_1
S → if (B) S_1 else S_2
S → while (B) S_1

In this grammar, the non-terminal B represents a Boolean expression and S represents a statement.

S.Code – gives the translation into TAC instructions.
S.next – denoting a label for the instruction immediately after the code for S
NewLabel () – creates a new Label each time it is called.
Gen(L) – attaches the label L to the next TAC Instruction to be generated.

Grammar	Semantic Rules
S → if (B) S_1	B. true = NewLabel(); B.false = S_1.next = S.next; **S.code = B.code || gen(B.true':') ||S_1.code**
S → if (B) S_1 else S_2	B. true = NewLabel(); B.false = NewLabel (); S_1.next = S_2.next = S.next; **S.code = B.code || gen(B.true':') ||S_1.code ||** **gen('goto' S.next) ||** **gen(B.false ':') ||S_2.code**

S → while (B) S_1	begin =NewLabel (); B.true = NewLabel(); B.false = S.next S_1.next = begin **S.code = gen (begin ':') || B.code || gen(B.true ':') ||** **S_1.code || gen ('goto' begin)**

Examples: please refer the Class work notebook.

BACKPATCHING

- A key problem when generating code for Boolean expression and flow-of-control statements is that of matching a jump instruction with the target of the jump.
- TAC for Boolean expressions and flow-of-control statements are generated using two passes
  - **Pass 1** – Generate the TAC instruction with the target of the jumps are unspecified.
  - **Pass 2** – Identifying and filling up the labels at appropriate jump statements.
- To generate TAC instructions in a single pass, a compiler may not know the labels that control must goto at the time of goto statements are generated.
- **Backpatching can be used to generate code for Boolean expressions and flow-of-control statements in one pass**.
- It maintains a list of TAC statements that needed to be completed with the same label. All of the jumps on a list have the same target label.

Attributes:

B.trueList – will be list of jump instructions into which control goes if B is true.
B.falseList – will be list of jump instructions into which control goes if B is false.
S.nextList – denoting a list of jumps to the instruction immediately following the code for S

Routines:

Makelist (i) – creates a new list containing only i.
Merge (p1,p2) – Concatenates the lsits pointed to by p1 with p2
Backpatch (p,i) – inserts i as target label for each of the instruction on the list pointed to by p.

Backpatching for Flow-of-Control Statements:

Grammar	Semantic Rules
S → if (B) M S_1	{ Backpatch (B.trueList, M.Instr); S.nextList = merge (B.falseList, S1.nextList); }

S → if (B) M_1 S_1 N else M_2 S_2	{ Backpatch (B.trueList,M1.Instr); Backpatch (B.falseList, M2.Instr); Temp=merge(S1.nextList,N.nextList); S.nextList = merge (temp,S2.nextList); }
S → while M_1 (B) M_2 S_1	{ Backpatch (S1.nextList, M1.Instr); Backpatch (B.trueList, M2.Instr); S.nextList = B.flaseList; Gen (‘goto’ M1.Instr); }
M → ϵ	{M.Instr = nextInstr; }
N → ϵ	{N.nextList = makelist (nextInstr); Gen (‘goto –‘); }

Backpatching for Boolean Expressions:

Grammar	Semantic Rules
B→ B_1 || *M* B_2	{ Backpatch (B1.falseList, M.Instr); B.trueList = merge (B1.trueList, B2.trueList); B. falseList = B2.falseList; }
B → B_1 *&& M* B_2	{ Backpatch (B1.trueList,M.Instr); B.trueList = B2.trueList; B.falseList = merge (B1.falseList, B2.falseList);}
B → *!* B_1	{ B. trueList = B1.falseList; B.falseList = B1.trueList; }
B →E_1 *relop* E_2	{B.trueList =makelist (nextInstr); B.falseList = makelist(nextInstr +1); Gen (‘if’ E1.place relop.op E2.place ‘goto _’); Gen (‘goto _’); }
B → *true*	{ B.trueList = makeList (nextInstr); Gen (‘goto _’); }
B → *false*	{ B.falseList = makeList (nextInstr); Gen (‘goto _’); }
M → ϵ	{M.Instr = nextInstr; }

Example: please refer the class work notebook.