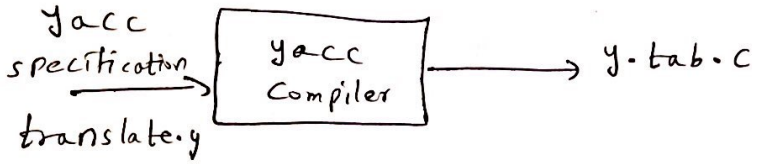
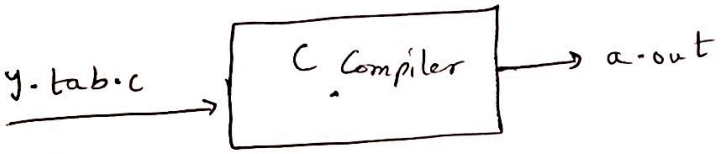
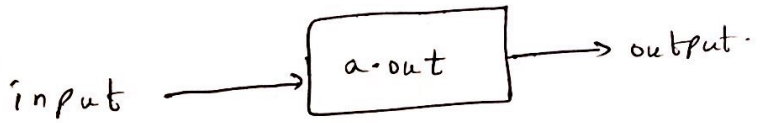
Parser Generators

→ Yet Another Compiler - Compiler (YACC)







→ Unix Command `yacc translate.y`

→ The above command translates the file translate.y into a C program called y.tab.c using LALR method.

→ y.tab.c is a representation of an LALR parser written in C, along

→ By Compiling y.tab.c along with ly library that contains the LR parsing program using command

`cc y.tab.c -ly.`

→ The yacc has three parts

declarations

%%

Translation rules

%%

supporting C routines

→ The Declarations Part

→ There are two sections in the declarations Part of a YACC program (Both are optional).

→ In the first section, we put ordinary C declarations, delimited by %{ and %}.

→ The Translation Rules Part

→ In the part of the YACC specification after the first %% pair, we put the translation rules.

→ Each Rule consists of a grammar production and associated semantic action.

<head> ⟶ <body>$_1$ | <body>$_2$ | ..... | <body>$_n$

would be written in YACC as

```
<head> : <body1>   { <semantic action1> }
       | <body2>   { <semantic action2> }
       :
       | <bodyn>   { <semantic actionn> }
       ;
```

→ A YACC semantic action is a sequence of 'C' statements. In a semantic action, the symbol $$ refers to the attribute value associated with the Nonterminal of the head, while $i refers to value associated with ith grammar symbol (Terminal or) Nonterminal) of the body.

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

↓

```
expr : expr '+' term { $$ = $1 + $3; }
     | term
     ;
```

→ The 'term' is the third grammar symbol of the body, '+' is second symbol.

→ The semantic Action associated with the first production adds value of expr and term of body and assign result to expr of head.

```
$$ = $1 + $3;
```

```
$$ = $1;
```
→ default action.

The Supporting C-Routines part

→ The third part of a YACC specification consists of supporting C-routines.

→ A lexical analyzer by name yylex() must be provided.

→ The lexical analyzer yylex() produces token consisting of a token name and its associated attribute value.

→ The attribute value associated with a token is communicated to the parser through a YACC-defined variable yylval.

YACC specification of a simple Desk calculator

```
%{
#include <ctype.h>
%}
% token DIGIT
%%
line : expr '\n'   { printf("%d\n", $1); }
     ;
expr : expr '+' term { $$ = $1 + $3; }
     | term
     ;
```

```
term : term '*' factor { $$ = $1 * $3; }
     | factor
     ;
factor : '(' expr ')'    { $$ = $2; }
       | DIGIT
       ;
%%
yylex()
{
  int c;
  c = getchar();
  if (isdigit(c))
  {
    yylval = c - '0';
    return DIGIT;
    }
  return c;
  }
```

→ YACC will ~~not~~ resolve all parsing action conflicts using two following rules:

(i) A reduce/reduce conflict is resolved by choosing the conflicating production listed first in the YACC specification.

(ii) A shift/reduce conflict is resolved in favour of shift.

Eg:- %left '+' '-' [left Associative]

+ & - same precedence

→ YACC resolves shift/reduce conflict by attaching a precedence & associativity to each production involved in a conflict, as well as to each terminal involved in a conflict.

→ If it must choose between shifting input symbol a and reducing by production $A \rightarrow \alpha$, YACC reduces if the precedence of the production is greater than a, or if the precedences are the same and the associativity of production is left, otherwise shift action is performed

Unit - 3

Syntax Directed Translation

→ A Syntax-directed definition specifies the values of attributes by associating semantic rules with grammar productions.

→ A Syntax directed Definition (SDD) is a Context free grammar together with attributes and rules.

→ Attributes are associated with grammar symbols and rules are associated with productions.

→ Eg. If X is a symbol and a is one of its attributes, then we write $X.a$ to denote the value of a at a particular parse tree node labeled X.

Inherited and Synthesized Attributes

→ A synthesized attribute for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. A synthesized Attribute at N node is defined only in terms of attribute values at the children of N and at N itself.

→ An Inherited Attribute for a nonterminal B at a parse tree node N is defined by a semantic rule associated with production at the parent of N.

→ An inherited attribute at Node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

→ Terminals can have synthesized attributes, but not inherited attributes.

→ Attributes for terminals have lexical values that are supplied by a lexical Analyzer.

→ There is no semantic Rules in the SDD itself for computing value of an attribute for a Terminal.

SDD for a simple Desk Calculator

Production	Semantic Rules
L → E n	L.val = E.val
E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * 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

→ For the NonTerminals E, T and F the values can be obtained using the attribute "val"

→ The token digit has synthesized attribute lexval whose value can be obtained from lexical Analyzer.

→ The rule for production 1, L → En, sets the L.val to E.val, which is numerical value of Entire expression.

→ Production 2, E → E1+T, also has one rule, which computes the val attribute for the head E as sum of values at E1 and T. The ~~The value of E~~ The val for E is sum of the values of val at the children node.

→ Production 3, E → T, value for E has to be same as the value of val at child for T.

→ Production 4, is a similar to 2nd production, it multiplies the values of children.

→ Productions 5 and 6 copy values at child, like that of production 3.

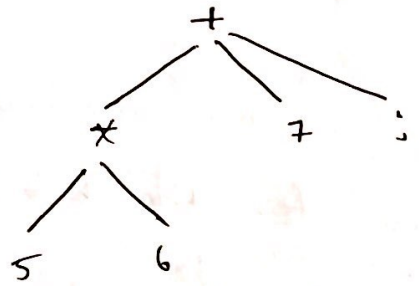
→ Production 7 gives F.val the value of the digit, ~~the~~ i.e the numerical value of token digit that the lexical

Analyzer returned.

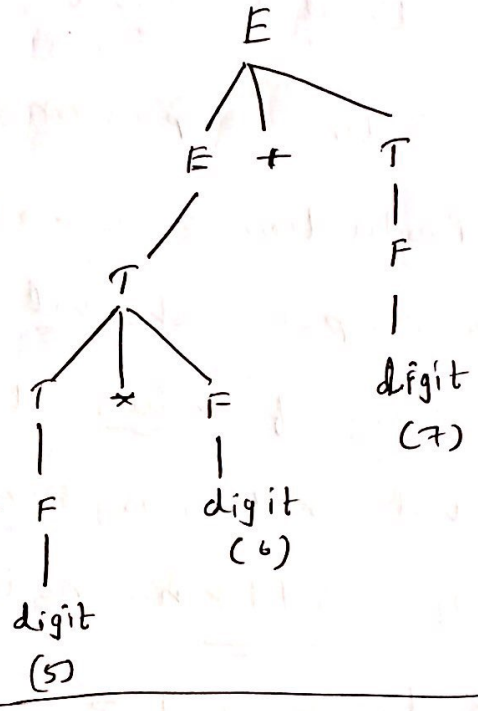
→ An SDD that involves only synthesized attributes is called S-attributed SDD.

Eg:- 5 * 6 + 7

Syntax tree

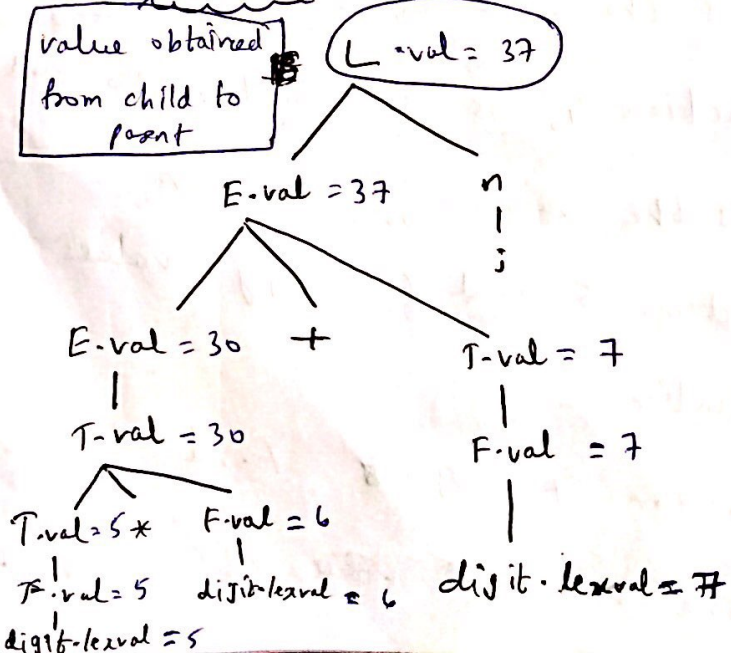


Parse tree



Annotated Parse Tree

value obtained from child to parent



→ The S-attributed definition, can be computed by Bottom-up fashion

→ A Parse Tree, showing the value(s) of its attribute(s) is called an Annotated Parse Tree.

→ For an SDD with both inherited & synthesized attributes, there is no garuntee that there is even one order in which to evaluate attributes at nodes.

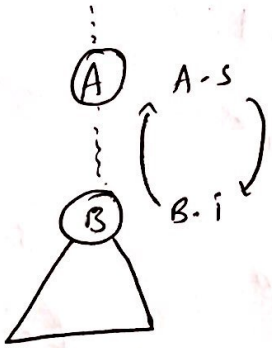
Eg:

PRODUCTION	SEMANTIC RULE
$A \rightarrow B$	$A.s = B.i;$
	$B.i = A.s + 1;$

A.s = synthesized Attribute

B.s = Inherited Attribute

→ There will be Circular Dependency of A.s and B.i



Evaluation Orders for SDD's

→ "Dependency Graphs" is useful tool for determining an evaluation order for the attribute instances in a given parse tree

→ Annotated parse Tree shows the values of attributes, A dependency graph helps us determine how those values can be computed.

→ A Dependency Graph depicts the flow of information among the attribute instances in particular parse tree.

→ An edge from one attribute instance to another means that the value of the first is needed to compute the second.

→ Edges express Constraints Implied by the semantic rules.

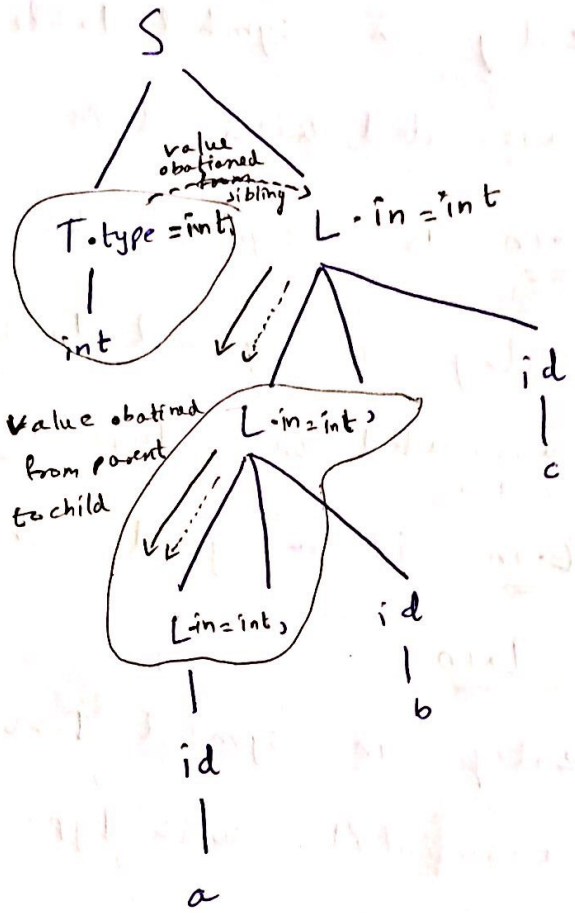
Eg:-

Production	Semantic Rules
$S \rightarrow TL$	L.in = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ char	T.type = character
$T \rightarrow$ float	T.type = float
$T \rightarrow$ double	T.type = double

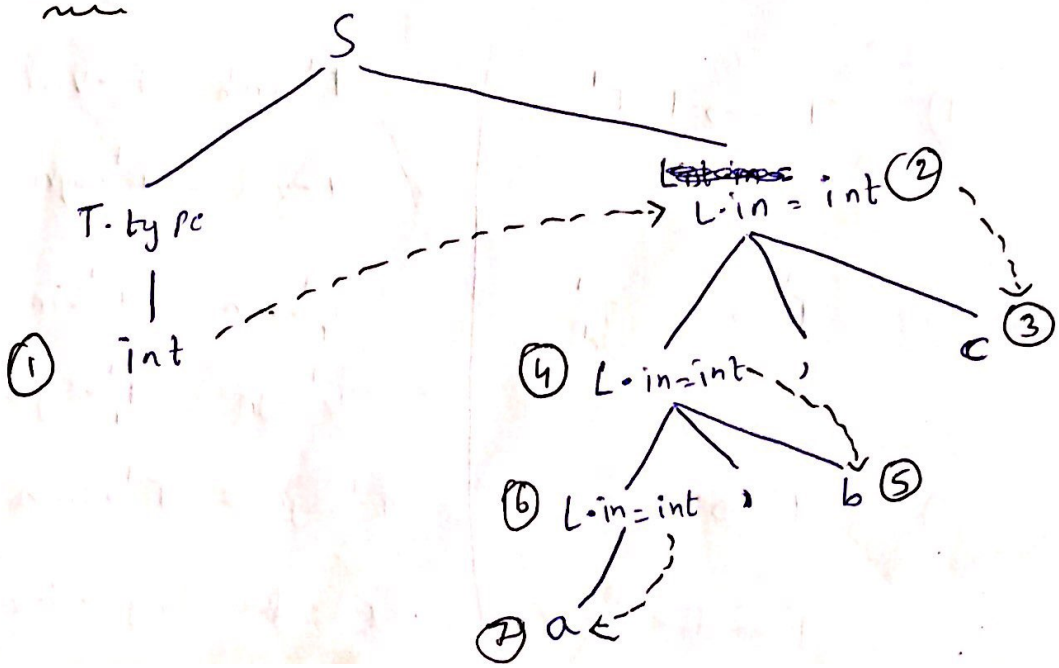
$L \rightarrow L_1, id$ $\quad L_1.in = L.in$
entry_type (id_entry, L.in)

$L \rightarrow id$ $\quad$ entry_type (id_entry, L.in)

Eg: int a, b, c;



Evaluation order



The evaluation order can be decided as follows

① The type int is obtained from lexical analyzer by analyzing the input token.

② The L.in is assigned the type int from sibling T.type

③ The entry in symbol Table for identifier c gets associated with type int.

④ The L.in is assigned the type int from parent L.in

⑤ The entry in symbol Table for identifier b gets associated with type int.

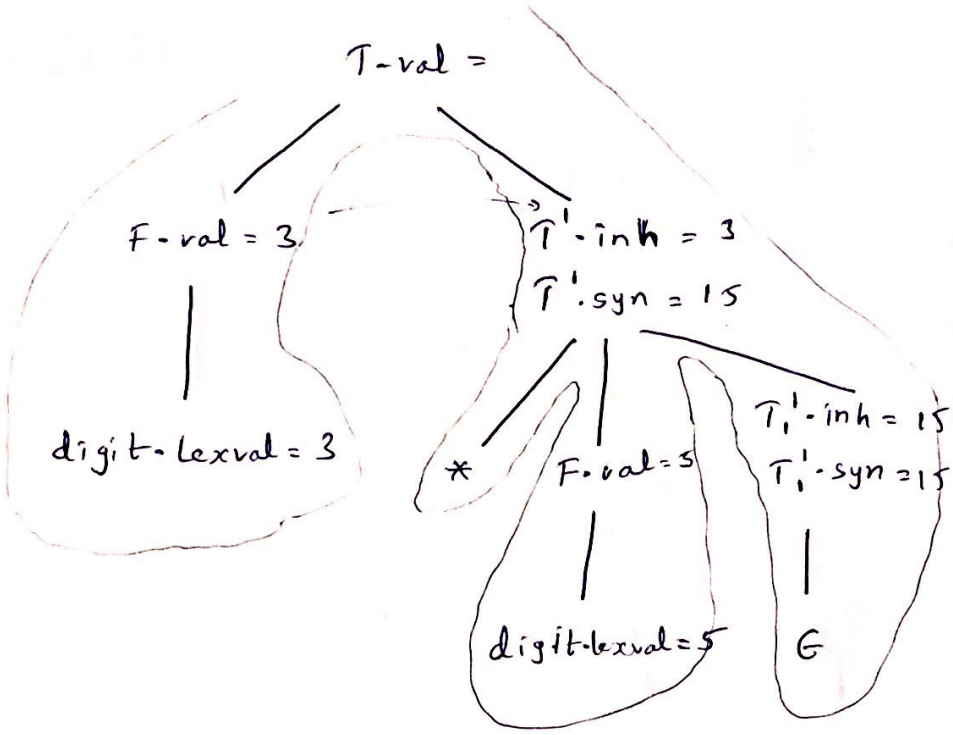
⑥ The L.in is assigned the type int from the parent L.in

⑦ The entry in symbol Table for identifier a gets associated with type int.

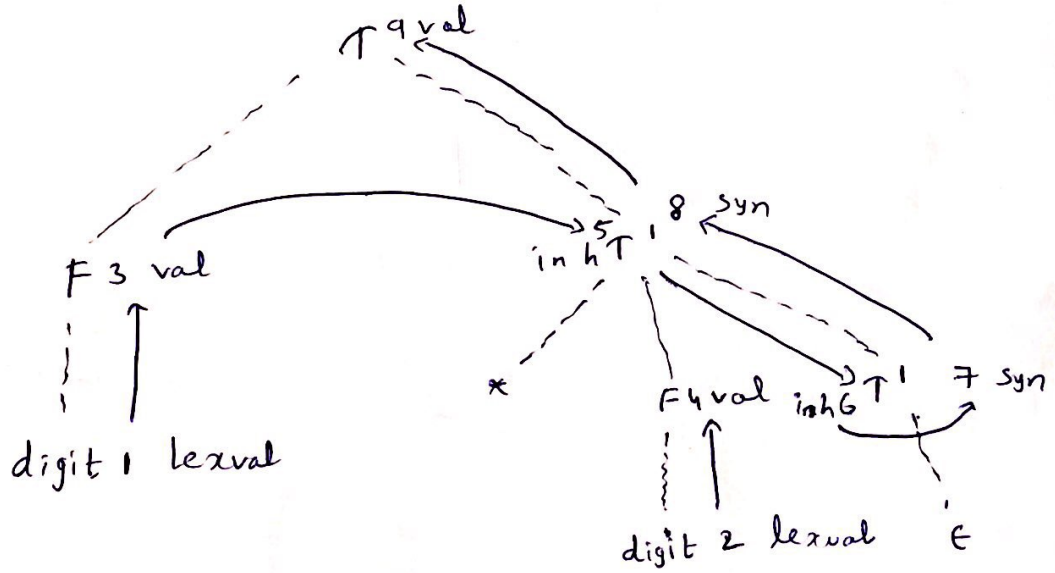
→ An SDD suitable for top-down parsing

1)	$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *F T_1'$	$T_1'.inh = T'.inh * F.val$ $T'.syn = T_1'.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
	$F \rightarrow digit$	$F.val = digit.lexval$

Annotated parse tree for 3 * 5



Dependency graph for above Annotated parse Tree



Applications of Syntax-Directed Translation

Construction of syntax Trees

→ Each node in a syntax tree represents a construct, the children of the node represent the meaningful components of the construct.

→ A syntax-tree node representing an expression $E_1 + E_2$ has label '+' and two children representing the subexpressions E_1 and E_2.

→ Each object will have an op field that is the label of the node.

→ The objects will have additional fields as follows:

- If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function Leaf(op, val) creates a leaf object.
- If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function Node takes two or more arguments

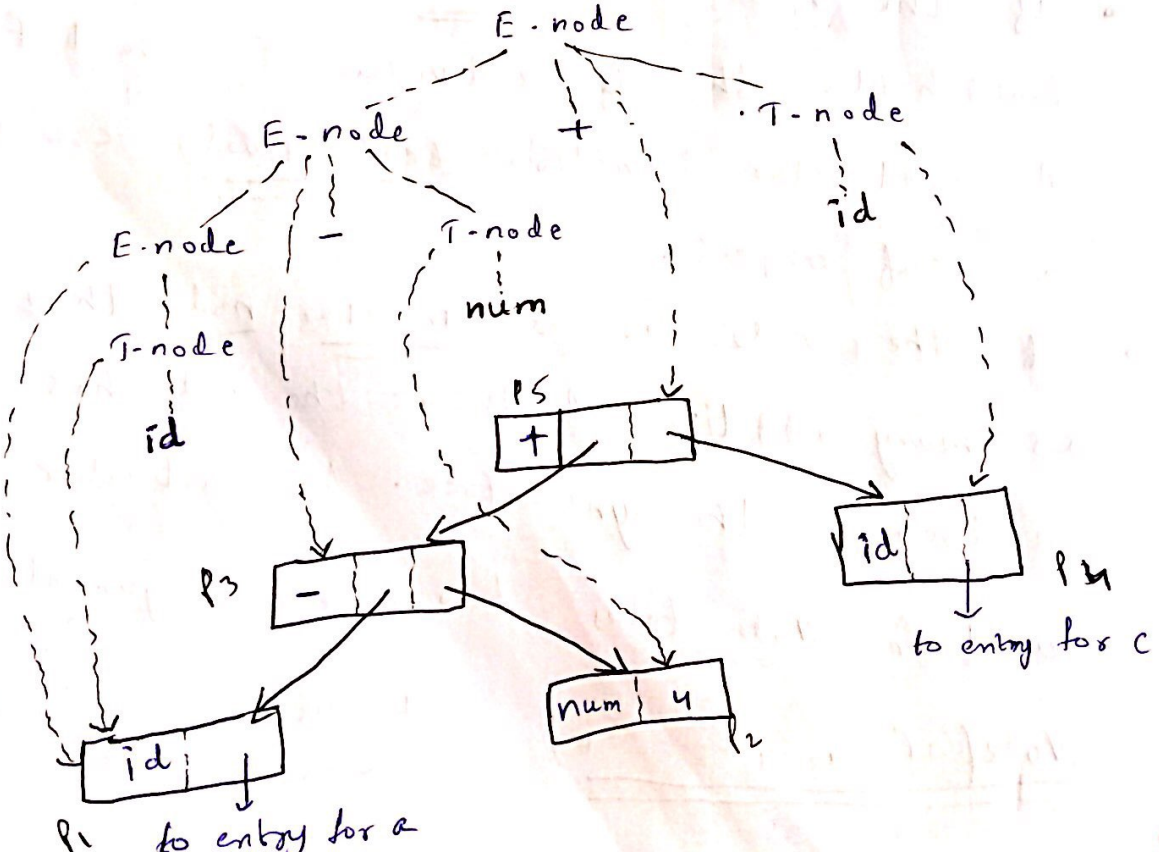
Node(op, c_1, c_2 ..... c_k) // k additional fields for k children

Eg:

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

Syntax tree for $a - 4 + c$

p_1 = new Leaf (id, entry-a);
p_2 = new Leaf (num, 4);
p_3 = new Node ('-', p_1, p_2);
p_4 = new Leaf (id, entry-c);
p_5 = new Node ('+', p_3, p_4);



Directed Acyclic Graph for Expression

→ The directed Acyclic graph usually referred as DAG is a directed graph drawn by identifying the common subexpressions.

→ Like syntax tree DAG has nodes representing the subexpressions in the expression.

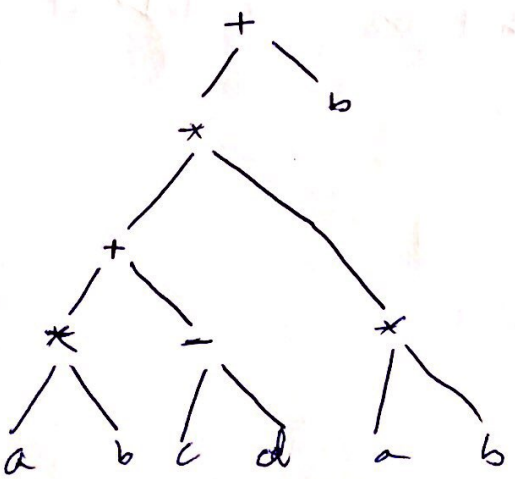
→ These nodes have operator, operand$_1$ and operand$_2$ where operands are children of that node.

→ The difference b/w DAG & syntax tree is that common subexpressions has more than one parent and in syntax tree the common subexpressions would be represented as duplicated subtree.

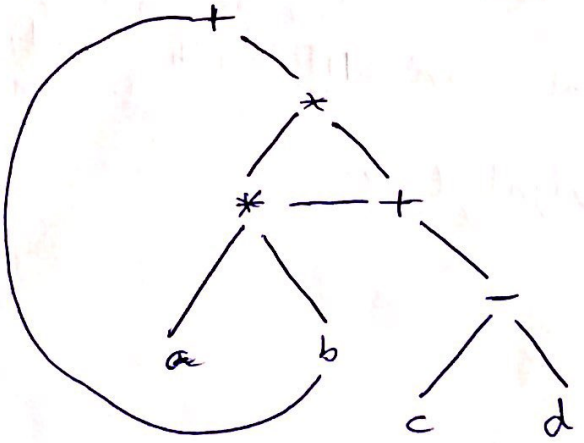
Eg:- $(a * b) + (c - d) * (a * b) + b$

Postorder Traversal $ab* cd- + ab* * b+$

Syntax Tree



DAG



Syntax Directed Translation Schemes

→ These are complementary notation to syntax directed definitions

→ Any SDT can be implemented by first building a parse Tree and then performing the actions in a left-to-right depth first order, i.e during a preorder traversal.

→ SDT's, are implemented during parsing, without building a parse Tree.

→ SDT's are used to implement two important classes of SDD's:

(1) The underlying grammar is LR-parsable, and the SDD is S-attributed.

(2) The underlying grammar is LL-parsable, and the SDD is L-attributed.

→ The semantic rules in an SDD can be converted into an SDT with actions that are executed at right time.

Postfix Translation Schemes

→ The simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed.

→ In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production

→ SDT's with all actions at the right ends of the production bodies are called Postfix SDD's.

Eg: Postfix SDT implenting for Desk Calculator

$L \rightarrow E\ n$	{ Print (E.val); }
$E \rightarrow E_1 + T$	{ E.val = E_1.val + T.val; }
$E \rightarrow T$	{ E.val = T.val; }
$T \rightarrow T_1 * F$	{ T.val = T_1.val * F.val; }
$T \rightarrow F$	{ T.val = F.val; }
$F \rightarrow (E)$	{ F.val = E.val; }
$F \rightarrow$ digit	{ F.val = digit.lexval; }

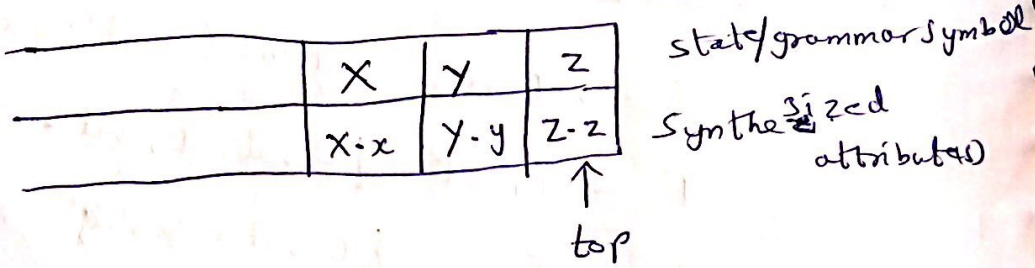
Parser stack Implementation of postfix SDT's

→ postfix SDT's can be implemented during LR parsing by executing the actions when reduction occur.

→ The attribute(s) of each Grammar symbol can be put on the stack in a place where they can be found during the reduction.

→ Place the attributes along with Grammar symbols in records on the stack itself.

→ Eg:- Parser stack with a field for synthesized attributes



→ The three symbols XYZ are on the top of the stack, about to be reduced using production $A \rightarrow XYZ$.

→ If the attributes are all synthesized, and the actions occur at the ends of the productions, compute the attributes for the head

When we reduce the body to the head.

Eg: Implementing the desk calculator on a bottom up parsing stack.

Production	Actions
$L \rightarrow En$	{ print (stack [top - 1] . val); top = top - 1; }
$E \rightarrow E_1 + T$	{ stack [top - 2] . val = stack [top-2] . val + stack [top] . val ; top = top - 2; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ stack [top-2] . val = stack [top-2] . val * stack [top] . val ; top = top - 2; }
$T \rightarrow F$	
$F \rightarrow (E)$	{ stack [top-2] . val = stack [top-1] . val ; top = top - 2; }
$F \rightarrow$ digit	

SDT's with Actions inside Productions

→ An action may be placed at any position within the body of a production.

→ It is performed immediately after all symbols to its left are processed.

Eg: $B \rightarrow X \{a\} Y$

→ The Action a is done after X is recognized (X is a terminal) or all the terminals derived from X (if X is nonTerminal)

- If the parse is bottomUp, then action a is performed as soon as X appears on top of the stack.
- If the parse is top-down, then action a is performed just before we attempt to expand this occurence of Y (if Y is a nonterminal) or Check for Y on the input (if Y is terminal).

Eg:

$L \rightarrow En'$

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

$E \rightarrow T$

$T \rightarrow \{print('*');\}\ T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

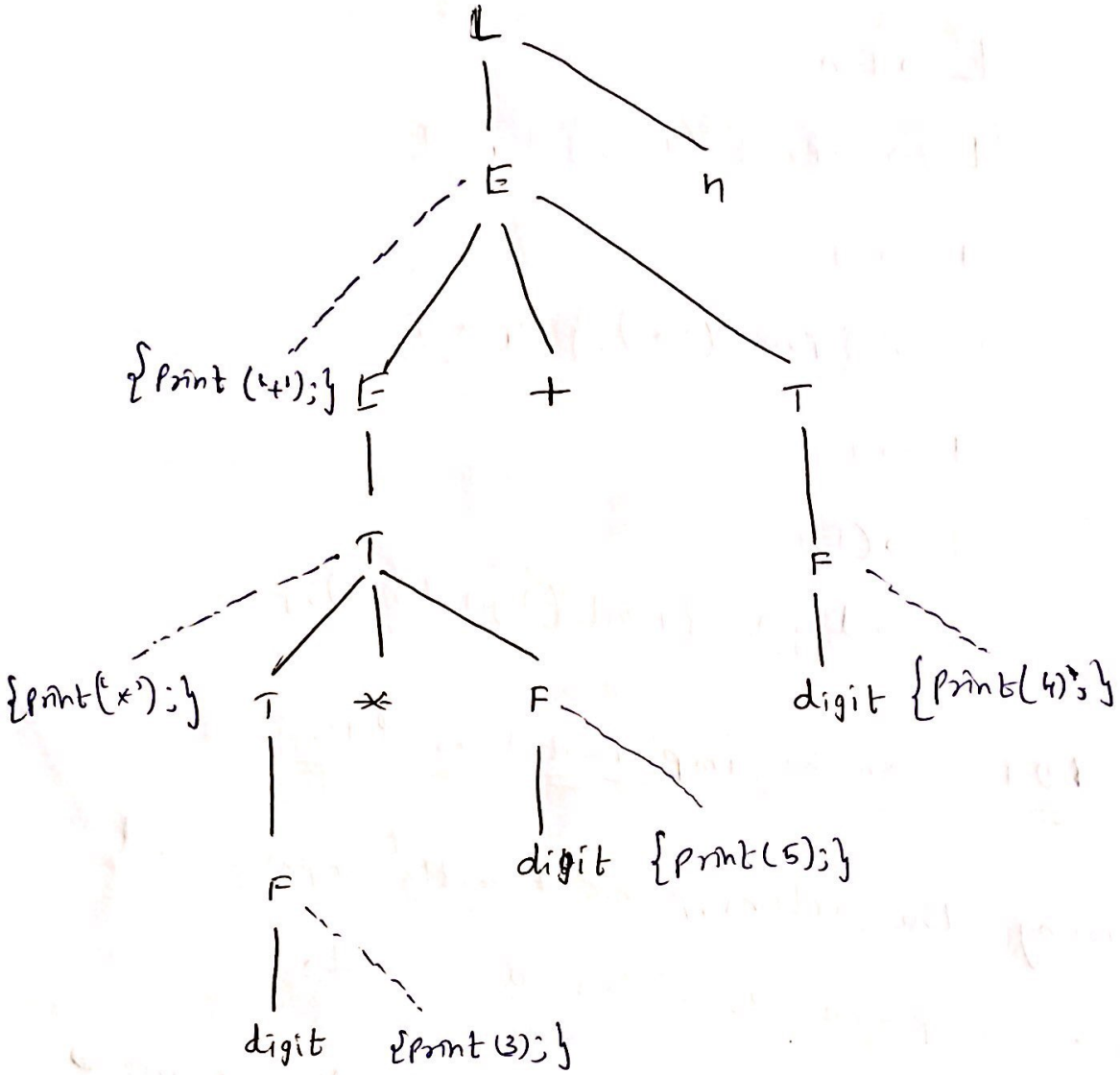
$F \rightarrow digit\ \{print(digit.lexval);\}$

Any SDT can be implemented as follows:

① Ignoring the actions, parse the input and produce a parse Tree as a result.

② Then, examine each interior node N, say one for production $A \rightarrow \alpha$, Add additional childeren to N for the actions in α, so the children of N from left to right have exactly the symbols and actions of α.

③ perform a preorder traversal of the tree, and as soon as a node labeled by an action is visited, perform that action.



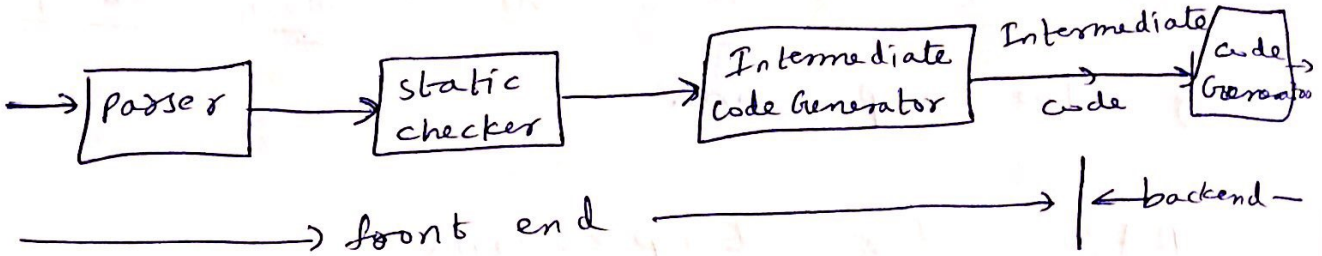
+ * 3 5 4

Implementing L-Attributed SDD's

→ Build the parse Tree and annotate: This approach works for S-attribute SDD.

→ Build the parse Tree, add actions and execute the actions in preorder: This approach works for any L-attributed definition.

Intermediate - Code Generation



→ Static checking includes type checking, which ensures that operators are applied to compatible operands.

→ It also includes any syntactic checks that remain after parsing.

Eg:- Break-statement in "C" is enclosed within a while, for (or) switch statement (or) an error is reported if such an enclosing statement does not exist.

Variants of syntax Trees

→ Nodes in a syntax tree represent constructs in source program.

→ The children of that node represent the meaningful components of a construct.

→ A directed Acyclic graph (DAG) for an expression identifies the common subexpressions of the expression.

DAG for Expressions

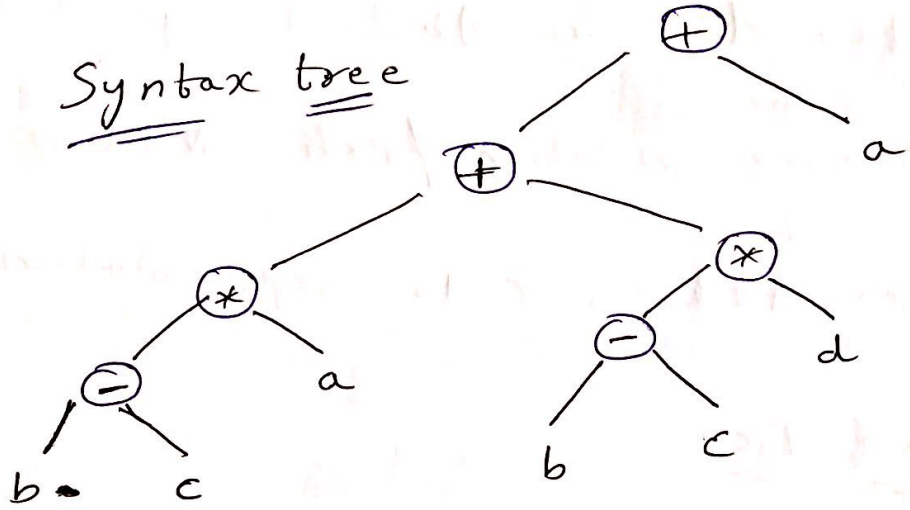
→ Like the syntax tree for an expression, a DAG has leaves corresponding to operands and interior nodes corresponding to operators.

→ Node 'N' in a DAG has more than one parent if 'N' represents a common subexpression in a syntax tree.

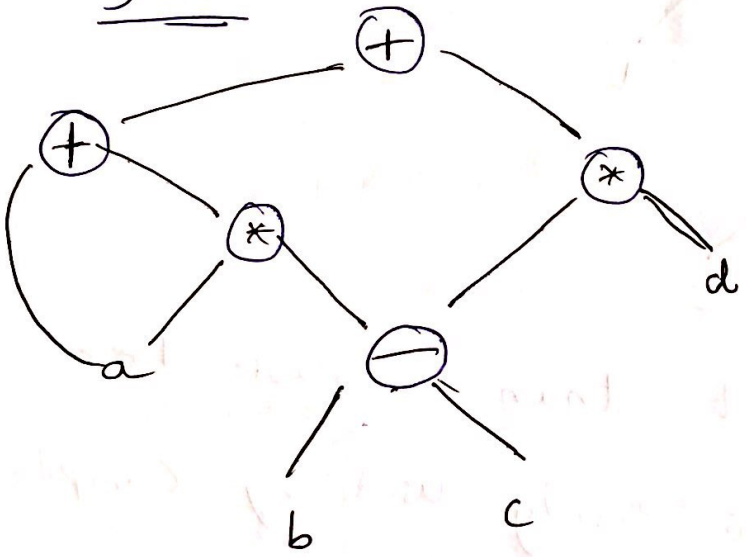
→ The tree for common subexpression would be replicated as many times as the subexpression appears in the original expression.

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

Syntax tree

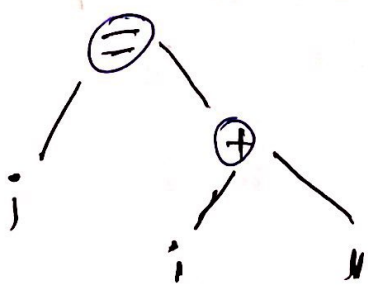


DAG

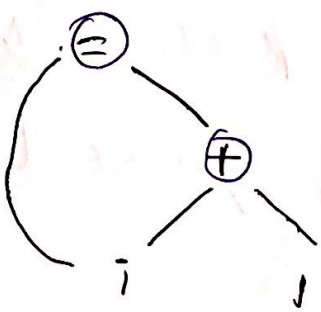


Eg 2:- $i = i + 1$

Syntax tree



DAG

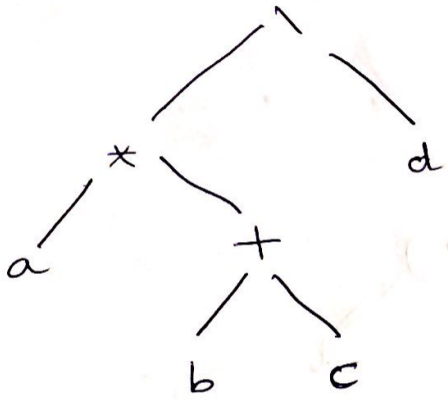


Intermediate Code:

1. Syntax tree (or) Abstract syntax tree
2. Postfix notation / Prefix Notation
3. Three Address Code representation.

1. Syntax Tree

Eg: a * (b + c) / d



→ More compact than a parse Tree

→ They can be easily used by Compiler.

2. Postfix notation (Suffix Notation (or) Reverse Polish Notation)

~~[illegible] is useful~~

→ Postfix Notation is a linear representation of syntax tree.

Eg: $x + y = xy +$

Productions	Semantic Rules
$E \rightarrow E_1$ op E_2	E.code = E_1.code ‖ E_2.code ‖ op
$E \rightarrow (E_1)$	E.code = E_1.code
$E \rightarrow id$	E.code = id.

Eg: $Z = (A - B) * (C - D) + [E + (F / G)]$

$Z = AB - * CD - + [E + FG/]$

$Z = AB - * CD - + EFG/+$

$Z = AB - CD - * + EFG/+$

$Z = AB - CD - * EFG/+ +$

Three address code

→ It is an intermediate code. It is used by the optimizing compilers.

→ In three address code, the given expression is broken down into several separate instructions.

→ These instructions can be easily translate into assembly Language.

→ Each Three-address code instruction has atmost three operands.

→ There is atmost one operator on the right side of an instruction.

$x + y * z$

Eg: $t_1 = y * z$

$t_2 = x + t_1$

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

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

Three address code

Quadruples

→ A quadruple has four fields

op → contains an internal code for operator

arg_1 → one operand of Binary expression

arg_2 → another operand of Binary expression

result → The result of Binary expression on R-H S

→ The following are some exceptions to this

(i) Instructions with unary operators like

$x = minus\ y$ (or $x - y$ donot use arg_2

(ii) operators like param use neither arg_2 nor result

(iii) Conditional and unconditional Jumps put the target label in result

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

$t_1 = minus\ c$

$t_2 = b * t_1$

$t_3 = minus\ c$

$t_4 = b * t_3$

$t_5 = t_2 + t_1$

$a = t_5$

	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

Triples

→ A triple has only fields

op

arg1

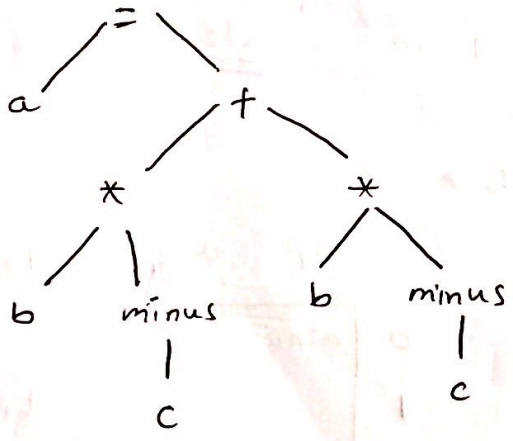
arg2

→ Using triples, we refer to the result of an operation x op y by its position, rather than by an explicit temporary name.

→ DAG and triple representations of expressions are equivalent.

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

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



(i) Syntax tree

	op	arg1	arg2
0	minus	c	—
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Triples

Indirect Triples

→ Indirect Triples consist of a listing of pointers to triples, rather than a listing of triples themselves.

→ With Indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves.

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

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

	op	arg1	arg2
0	minus	c	—
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Types and Declarations

→ The applications of types can be grouped under

(i) Type checking - uses logical rules to reason about the behaviour of a program at runtime. Specifically, it ensures that the types of the operands match the type expected by an operator.

Eg: '&&' operator in Java expects its two operands to be booleans.

(ii) Translation Applications - From the type of a name, a compiler can determine the storage that will be needed for that name at runtime. Type information is also needed to calculate the address denoted by an array reference.

Type expressions

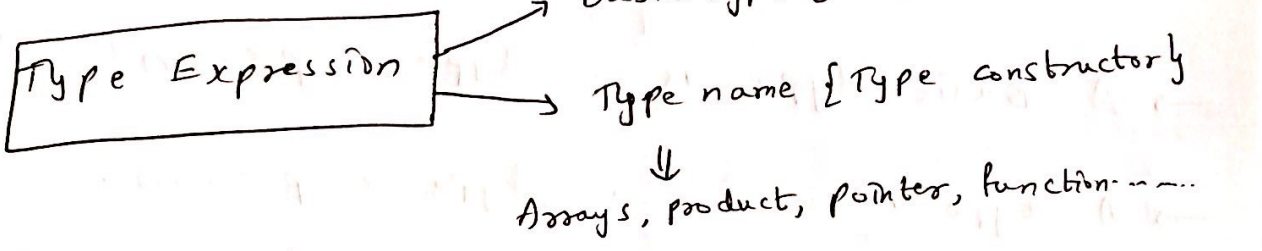
→ Types have structure, which we shall represent using type expressions

→ A Type expression is either a basic type or

is formed by applying an operator called a type constructor to a type expression

→ The sets of basic types and constructors depend on Language to be checked.

→ Type expression will denotes the type of Language construction.



Eg:- T is a Type Expression (Basic)

array (I, T) is a Type Expression

I - Index range

T - Type expression

Basic Type

array of 10 integers of int ⇒ array ([1...10], int)

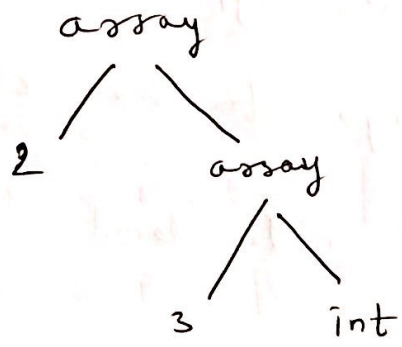
Type Construct

Eg:- The array type int [2][3] can be read as "array of 2 arrays of 3 integers each"

Type expression array (2, array (3, integer))

→ The operator array takes two parameters,

(i) a number and (ii) a type



Type expression for int [2][3]

→ A Basic type is a type expression [Eg: int, float, char]

→ A type name is a type expression.

→ A type expression can be formed by applying the array type constructor to a number and a type expression.

→ A record is a data structure with fields. A type expression can be formed by applying the record type constructor to field names and their types.

→ A type expression can be formed by using the type constructor "→" for function types.

s → t for "function from type s to type t"

→ If s and t are type expressions, then their Cartesian Product sxt is a type expression.

→ Type expressions may contain variables whose values are type expressions.

Type equivalence

→ Many Type-checking rules have the form "if two type expressions are equal then return a certain type else error"

→ When two expressions are represented by graphs, two types are "structurally equivalent" if and only if one of the following conditions is true:

Nameequivalence

- They are the same basic type.
- They are formed by applying the same constructor to structurally equivalent types.
- One is a type name that denotes the other.

Eg: char: a, b;
a = 'A';
b = a; /Name equivalence/

struct a, b;
/structural equivalence/

Declarations

→ Declarations with lists of names can be handled as

Eg: Grammar

$$D \rightarrow T\ id;\ D \mid \epsilon$$

$$T \rightarrow B\ C \mid record\ '\{'\ D\ '\}'$$

$$B \rightarrow int \mid float$$

$$C \rightarrow \epsilon \mid [num]\ C$$

→ The above grammar deals with the basic and array Types.

→ NonTerminal 'D' generates a sequence of declarations.

→ NonTerminal 'T' generates basic, array or) record (structure) types.

→ Nonterminal 'B' generates one of basic types int and float

→ NonTerminal 'C' (for component) generates string of 0 or) more integers, each integer surrounded by brackets.

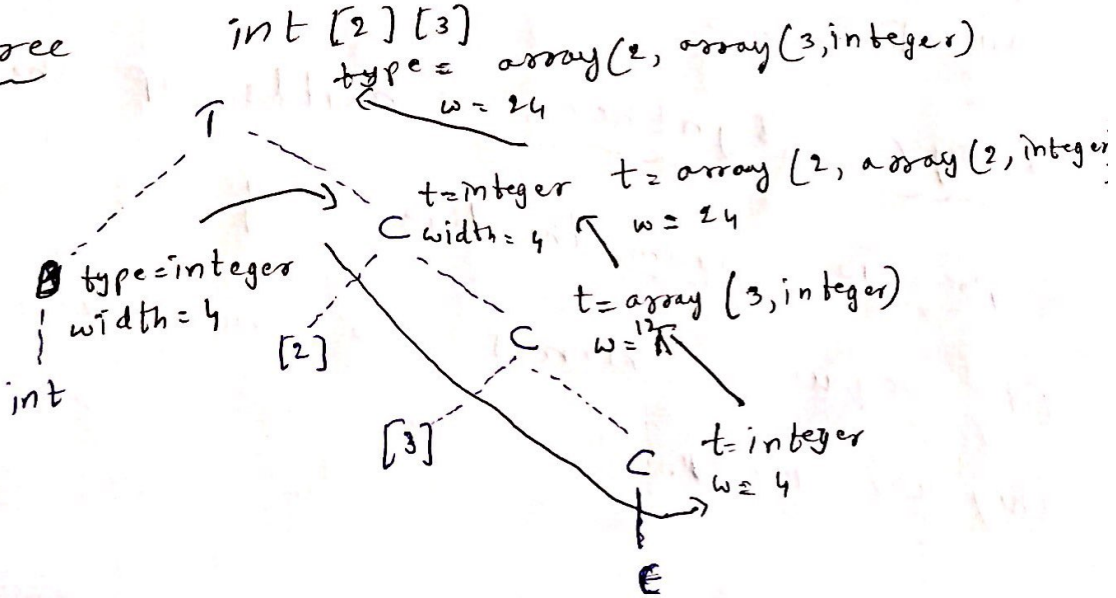
→ An array type consists of a basic type specified by B, followed by array components specified by non terminal 'C'.

→ A record type is a sequence of declarations for the fields of the record, all surrounded by curly braces.

SDT that computes types and their widths for basic and Array types.

$T \rightarrow B\ C$	{ t = B.type; w = B.width; }
$B \rightarrow int$	{ B.type = integer; B.width = 4; }
$B \rightarrow float$	{ B.type = float; B.width = 8; }
$C \rightarrow \epsilon$	{ C.type = t; C.width = w; }
$C \rightarrow [num]\ C_1$	{ array(num.value, C_1.type); C.width = num.value × C_1.width; }

Eg: Parse tree



Type checking:

→ To do Type checking a compiler needs to assign a type expression to each component of the source program.

→ The compiler must then determine that these type expressions conform to a collection of logical rules that is called the Type system for the source language.

→ Type checking has potential for catching errors in programs.

Rules for Type checking

→ Type checking can take on two forms:

(i) Syntensis
(ii) Intesence

→ Type Synthesis builds up the type of an expression from the types of its subexpressions. It requires names to be declared before they are used.

Eg:- The type of $E_1 + E_2$ is defined in terms of types of E_1 and E_2

A typical rule for type synthesis has the form

> if f has type $s \to t$ and x has type s
> then expression $f(x)$ has type t.

→ f and x denote the expressions, and $s \to t$ denotes function from s to t.

Type Inference

→ It determines the type of a language construct from the way it is used

A typical Rule for Type Inference has the form

> if $f(x)$ is an expression,
> then for some α and β, f has type $\alpha \to \beta$ and x has type α

Eg:-

Type syntesis

		type
(f)	void add ( )	int → void
(x)	int x	int
f(x)	void add (int x)	void

Type Inference

f(x)	void null (x)	void
f	void null ( )	$\alpha \to \beta$
	(x)	int → void
		int

Type checking

→ Type checker is a module of a compiler devoted to type checking tasks.

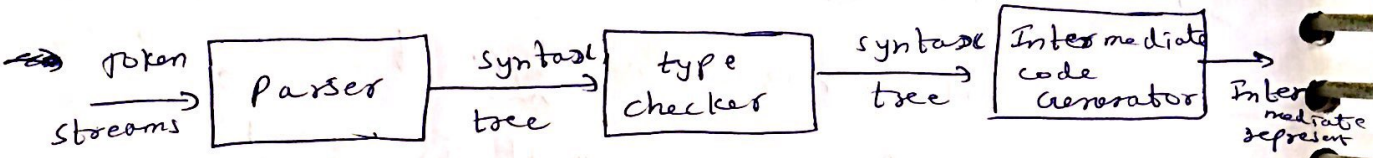
⇓

int - within range
float - within range
arrays - indexing done

→ Type checking is of two types

(i) static Type checking [Done at compile time]

(ii) Dynamic Type checking. [Done at Run time]



Type conversion (or) (Type casting)

→ A Type cast is basically a conversion from one Type to another

→ These are two types of conversions

(i) Implicit type conversion

(ii) Explicit type conversion

Implicit Type Conversion (Also called coercions)

→ If a Compiler converts one type of data into another type of data automatically

Eg:- short a = 20;

int b = a; // Implicit type conversion

→ No data loss in Implicit type conversions

bool → char → short int → int → long int → float

(ii) Explicit type conversion (Also called casts)

→ when data of one type is converted explicitly to another type with the help of predefined functions.

→ There may be Data loss in Explicit type conversion

Eg:- int a;

float f = 8.38;

a = f; // Explicit type conversion

a stores 8 (data loss)

Control Flow

→ The translation of statements such as if-else-statements and while-statements is tied to translation of boolean expressions.

→ In programming languages, boolean expressions are often used to

(i) Alter the flow of control

→ Boolean expressions are used as conditional expressions in statements that alter the flow of control.

Eg: If (E) S, the expression E must be true if statement S is reached.

(ii) Compute logical values

→ A Boolean expression can represent true or false as values.

Boolean expressions

→ Boolean expressions are composed of the boolean operators (&&, ||, !) applied to elements

that are boolean variables or relational expressions.

→ Relational expressions are of the form E_1 rel. E_2, where E_1 and E_2 are arithmetic expressions.

Eg; Grammar

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

rel. op op attribute with $(<, <=, >, >=, ==, !=)$

Short-circuit code

→ In short-circuit (or jumping), the boolean operators &&, || and ! translate into jumps.

→ The operators themselves donot appear in the code, instead, the value of boolean expression is represented by a position in the code sequence.

Eg:- if ($x < 100$ || $x > 200$ && $x != y$) $x = 0$;

Jumping code

```
if  x < 100  goto  L2
if  False  x > 200  goto  L1
if  False  x != y  goto  L1
L2: x = 0
L1:
```

Flow of Control Statements

Grammar

$S \longrightarrow$ if (B) S_1

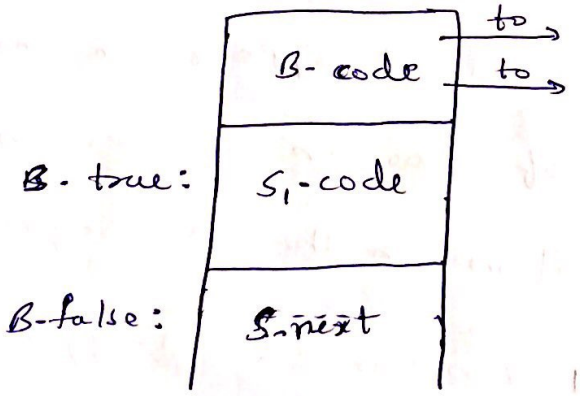
$S \longrightarrow$ if (B) S_1 else S_2

$S \longrightarrow$ while (B) S_1

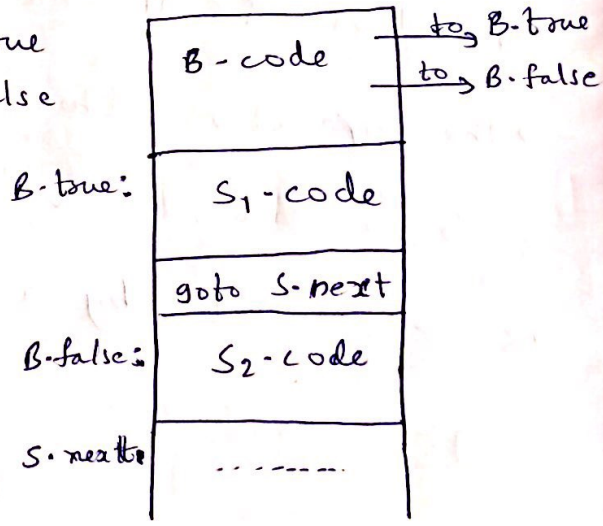
→ Non-Terminal 'B' represents a Boolean Expression

→ Non-Terminal 'S' represents a statement.

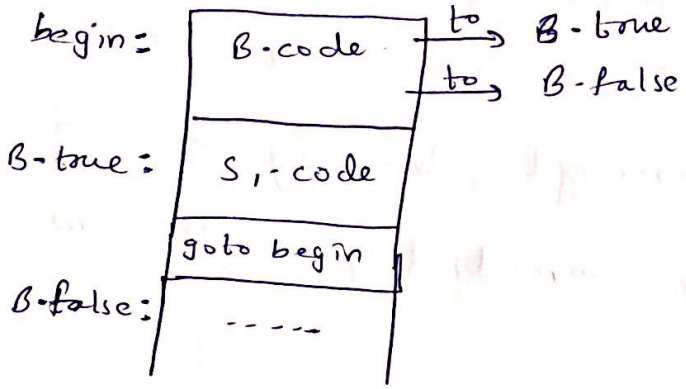
→ Both B and S have a synthesized attribute code, which gives the translation into three address code (instructions)



(a) if



(b) if else



(c) while

→ The translation of if (B) S_1 consists of B-code followed by S_1-code

→ within B-code are jumps based on the value of B. If B is true, control flows to first instruction of S_1-code. If B is false, control flows to the instruction immediately following S_1-code.

→ The labels for the jumps in B-code and S-code are managed using inherited attributes.

→ with a boolean expression B, we associate two labels: B.true, the label to which control flows if B is true, and B.false, the label

to which control flows if B is false.

→ With a statement S, we associate an inherited attribute S.next denoting a label for the instruction immediately after the code for S.

Syntax-Directed Definition for flow-of-control statements

Production	Semantic Rules
$P \rightarrow S$	S.next = newlabel() P.code = S.code || label(S.next)
$S \rightarrow$ assign	S.code = assign.code
$S \rightarrow$ if (B) S_1	B.true = newlabel() B.false = S_1.next S_1.next = S.next S.code = B.code || label(B.true) || S_1.code
$S \rightarrow$ if (B) S_1 else S_2	B.true = newlabel() B.false = newlabel() S_1.next = S.next S_2.next = S.next S.code = B.code || label(B.true) || S_1.code || gen('goto' S.next) || label(B.false) || S_2.code

$S \rightarrow \text{while}(B)\ S_1$

begin = newlabel()

B.true = newlabel()

B.false = S.next

S_1.next = begin

S.code = label(begin) || B.code ||

label(B.true) || S_1.code || gen('goto' begin)

$S \rightarrow S_1\ S_2$

S_1.next = newlabel()

S_2.next = S.next

S.code = S_1.code || label(S_1.next)

|| S_2.code

→ newlabel() creates a new label each time it is called.

→ label(L) attaches label 'L' to next three-address instruction to be generated.

→ P → S. The semantic rules associated with this production initialize S.next to newlabel. P.code consists of S.code followed by newlabel S.next.

→ $S \rightarrow assign$. S.code is simply assign.code

→ $S \rightarrow if\ (B)\ S_1$. The semantic rules create a new label B.true and attach it to the first three address instruction generated for statement S_1. Thus, jumps to B.true within the code for B will go to code for S_1. By setting B.false to S.next we ensure that control will skip the code for S_1 if B evaluates to false.

→ $S \rightarrow if\ (B)\ S_1$ else S_2 . The code for the Boolean expression B has jumps out of it to the first instruction of the code for S_1 if B is true, and to the first instruction of code for S_2 if B is false. The control flows from both S_1 and S_2 to the three address instruction immediately following the code for S. (its label is given by inherited attribute S.next) An explicit goto S.next appears after

the code for S_1 to skip over the code for S_2.

→ $S \rightarrow$ while (B) S_1 . We use local variable label attached to begin to hold a new label attached to the first instruction for this while-statement which is also first instruction for B.

The inherited labeled S.next marks the instruction that control must flow to if B is false (B.false set to S.next) .

A new label B.true is attached to the first instruction for S_1, the code for B generates a jump to this label if B is true. After the code for S_1 we place the instruction goto begin, which causes jump back to the beginning of code for boolean expression. S_1.next is set to label begin, so jumps from within S_1 code can go directly to begin.

→ $S \rightarrow S_1 S_2$ consists of the code for S_1 followed by code for S_2 - The semantic rules manage the labels, the first instruction after the code for S_1 is beginning of the code for S_2. The instruction after the code for S_2 is also the instruction after the code for S.

Control flow for Translation of Boolean Expressions

Production	Semantic Rules
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$
	$B_1.false = newlabel()$
	$B_2.true = B.true$
	$B_2.false = B.false$
	$B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$

$B \rightarrow B_1 \&\& B_2$

$B_1.true = newlabel()$

$B_1.false = B.false$

$B_2.true = B.true$

$B_2.false = B.false$

$B.code = B_1.code \,||\, label(B_1.true) \,||\, B_2.code$

$B \rightarrow ! B_1$

$B_1.true = B.false$

$B_1.false = B.true$

$B.code = B_1.code$

$B \rightarrow E_1 \; rel \; E_2$

$B.code = E_1.code \,||\, E_2.code \,||$

gen ('if' $E_1.addr$ rel-op

$E_2.addr$ 'goto' $B.true$) ||

gen ('goto' $B.false$)

$B \rightarrow true$

$B.code = gen$ ('goto' $B.true$)

$B \rightarrow false$

~~$B.false$~~ -

$B.code = gen$ ('goto' $B.false$)

→ $B \rightarrow E_1$ rel E_2, is tran

into a comparision th

with jumps to appr

Eg: B is of form E_1 o

```
if E1.addr < E2.addr goto
goto B.false
```

→ $B \rightarrow B_1 \,||\, B_2$ · If

immediately know B

B_1.true = is sam

If B_1 is false, then B

So we make B_1.fa

First instruction in f

If B_2 is true the

If B_2 is false the

→ $B \rightarrow B_1 \,\&\&\, B_2$ · If

immediately know B i

If B_1 is true, then B_2

So we make B_1-true to be label of first instruction in the code for B_2 (B_1-true = newlabel())

If B_2 is true then B is true [B_2-true = B-true]

If B_2 is false then B is false [B_2-false = B-false]

→ $B \rightarrow !B_1$, No code for an expression B of the form $!B_1$.

Interchange the true and false exits of B to get the true and false exits of B_1.

→ $B \rightarrow$ true & $B \rightarrow$ false, The constants true and false translate into jumps to B-true & B-false

Eg:- if ($x < 100$ || $x > 200$ && $x != y$) $x = 0$;

```
        if x < 100 goto L2
        goto L3
L3 :    if x > 200 goto L4
        goto L1
L4 :    if x != y goto L2
L2 :    x = 0
L1 :
```

control flow translation of simple if-statement

Back Patching

→ A key problem when generating code for boolean expressions and flow of control statements is that of matching a jump instruction with the target of the jump.

→ For Example, the translation of Boolean Expression B in if (B) S contains a jump, for when B is false, to the instruction following the code for S.

→ In one pass translation, B must be translated before S is examined. what then is the traget of the goto that jumps over the code for S?

→ This problem is addressed by passing labels as inherited attributes to where the relevant jump instructions were generated. But a seperate pass is

needed to bind labels to addresses.

→ Backpatching, is a technique in which lists of jumps are passed as synthesized attributes.

→ specifically, when a jump is generated, the target of the jump is temporarily left unspecified.

→ Each such jump is put on a list of jumps whose labels are to be filled in when the proper label can be determined. All of the jumps on a list have the same target code.

One pass code Generation Using Backpatching

→ Backpatching can be used to generate code for boolean expressions and flow of control statements in one pass.

→ we use synthesized attributes truelist & falselist of nonterminal B are used

to manage labels in Jumping code for boolean expressions.

→ B.truelist will be a list of Jump or conditional Jump instructions into which we must insert the label to which control goes if B is true.

→ B.falselist will be list of instructions that eventually get the label to which control goes when B is false.

→ As a code is generated for B, Jumps to the true and false exits are left incomplete, with the label field unfilled.

→ These incomplete Jumps are placed on lists pointed by B.truelist & B.falselist

→ S has synthesized attribute S.nextlist, denoting a list of Jumps to the instruction immediately following the code for S.

→ To manipulate lists of jumps, we use three functions.

(i) makelist(i) creates a new list containing only i, an index into the array of instructions. It returns a pointer to newly created list.

(ii) merge(p_1, p_2) concatenates lists pointed by p_1 and p_2, and returns a pointer to the concatenated list.

(iii) backpatch(p, i) inserts i as the target label for each of the instructions on the list pointed by p.

Backpatching for Boolean Expressions

→ We now construct a translation scheme suitable for generating code for boolean expressions during bottom-up parsing.

→ A marker nonterminal (M) in the grammar causes a semantic action to pick up, at

appropriate times, the index of the next instruction to be generated.

Grammar

$B \rightarrow B_1 \,||\, M\, B_2$
$\quad | \; B_1 \,\&\&\, M\, B_2$
$\quad | \; !\, B_1$
$\quad | \; (B_1)$
$\quad | \; E_1 \text{ rel } E_2$
$\quad | \; \text{true}$
$\quad | \; \text{false}$

The Translation scheme for Boolean expressions

$B \rightarrow B_1 \,||\, M\, B_2$ { backpatch (B_1.falselist, M.instr);
B.truelist = merge (B_1.truelist, B_2.truelist);
B.falselist = B_2.falselist; }

$B \rightarrow B_1 \,\&\&\, M\, B_2$ { backpatch (B_1.truelist, M.instr);
B.truelist = B_2.truelist;
B.falselist = merge (B_1.falselist, B_2.falselist); }

$B \rightarrow !\, B_1$ { B.truelist = B_1.falselist;
B.falselist = B_1.truelist }

$B \rightarrow (B_1)$ { B.truelist = B_1.truelist ;
B.falselist = B_1.falselist ; }

$B \rightarrow E_1$ rel E_2 { B.truelist = makelist (nextinstr);
B.falselist = makelist (nextinstr + 1);
emit ('if' E_1.addr rel.op E_2.addr
'goto' _ ');
emit ('goto _'); }

$B \rightarrow$ true { B.truelist = makelist (nextinstr);
emit ('goto _'); }

$B \rightarrow$ false { B.falselist = makelist (nextinstr);
emit ('goto _'); }

$M \rightarrow \epsilon$ { M.instr = nextinstr; }

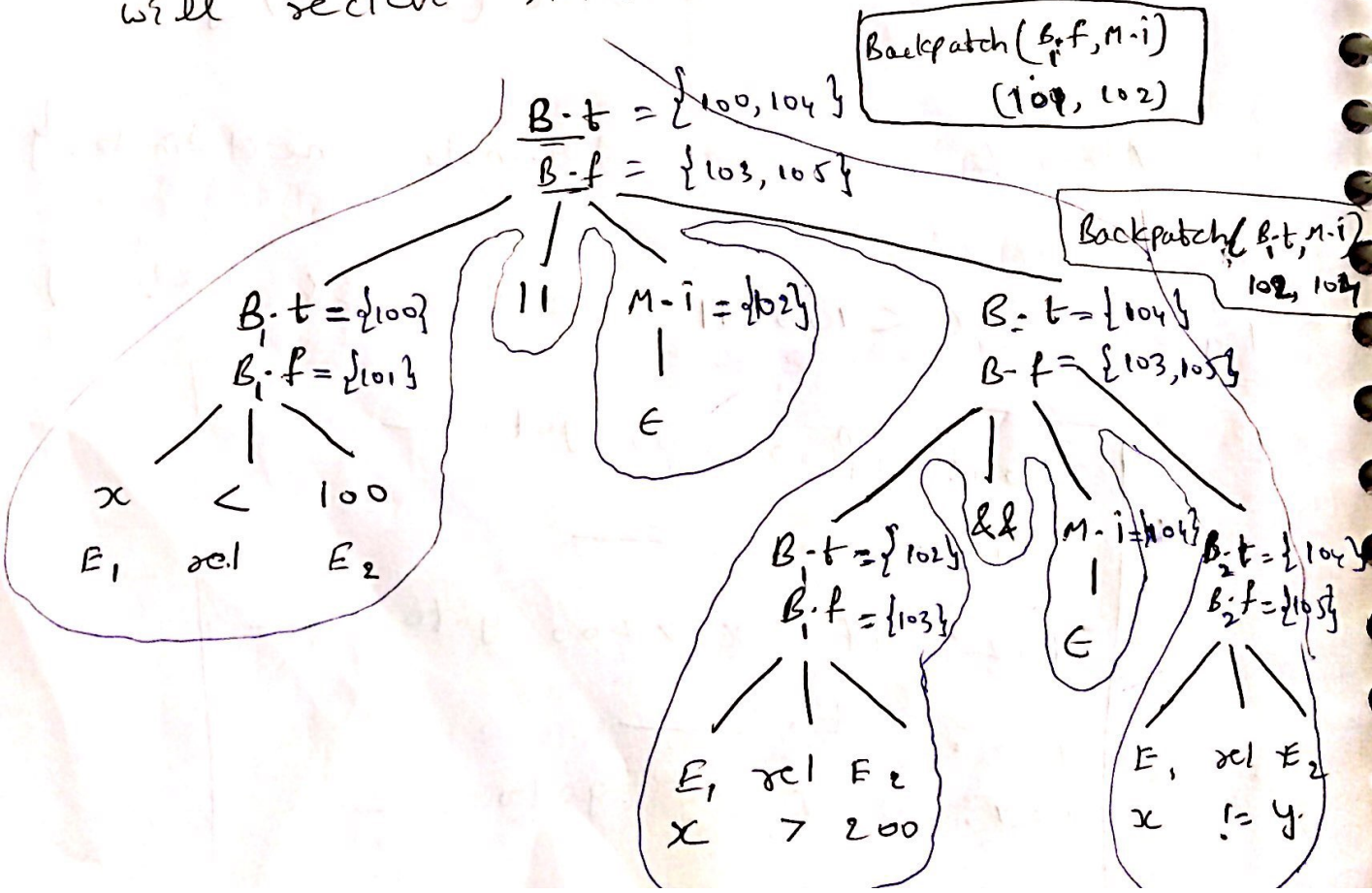
Eg :- $x < 100 \;||\; x > 200 \;\&\&\; x != y$

```
100: if x <100 goto —
101: goto —
102: if x >200 goto —
103: goto —
104: if x != y goto —
105: goto —
```

→ $B \rightarrow B_1 \,||\, M\, B_2$ - If B_1 is true, then B is also true, so the jumps on B_1-truelist become part of B-truelist.

If B_1 is false, we must next test B_2, so the target for the jumps B_1-falselist must be beginning of code generated for B_2.

This target is obtained by using Marker Non-terminal M. The value M-instr will be backpatched onto the B_1-falselist (each instruction on the list of B_1-false list will recieve M-instr as its target label).



(a) After Backpatching 104 into instruction 102

100: if x<100 goto ____

101: goto ____

102: if x>200 goto 104

103: goto ____

104: if x!=y goto ____

105: goto ____

(b) After Backpatching 102 into instruction 101

100: if x<100 goto ____

101: goto 102

102: if x>200 goto 104

103: goto ____

104: if x!=y goto ____

105: goto ____

(c) After filling B.truelist with true (100 and 104 with true)

100: if x<100 goto true

101: goto 102

102: if x>200 goto 104

103: goto ____

104: if x!=y goto true

105: goto ____

(d) After filling B-falselist with false (103 and 105 with false)

100 : if x < 100 goto true

101 : goto 102

102 : if x > 200 goto 104

103 : goto false

104 : if x != y goto true

105 : goto false

Translation scheme of flow of control statements using Backpatching

S → if (B) M S_1 { backpatch (B.truelist, M.instr);
S.nextlist = merge (B.falselist, S_1.nextlist);}

S → if (B) M_1 S_1 N else M_2 S_2 { backpatch (B.truelist, M_1.instr);
backpatch (B.falselist, M_2.instr);
~~temp = mergelist~~
temp = merge (S_1.nextlist, N.nextlist);
S.nextlist = merge (temp, S_2.nextlist);
}

$S \rightarrow$ while M_1 (B) M_2 S_1	{ backpatch(S_1.nextlist, M_1.instr); backpatch(B.truelist, M_2.instr); S.nextlist = B.falselist; emit('goto' M_1.instr); }
$S \rightarrow \{ L \}$	{ S.nextlist = L.nextlist }
$S \rightarrow A$;	{ S.nextlist = null; }
$M \rightarrow \epsilon$	{ M.instr = nextinstr; }
$N \rightarrow \epsilon$	{ N.nextlist = makelist(nextinstr); emit('goto _'); }
$L \rightarrow L_1, M\ S$	{ backpatch(L_1.nextinstr, M.instr); L.nextlist = S.nextlist }
$L \rightarrow S$	{ L.nextlist = S.nextlist }

Translation of Switch Statements

→ The intended translation of a switch is code to:

1) Evaluate the Expression E
2) Find the value v_j in the list of cases that is the same as the value of the expression. The default value matches the expression if none of the values explicitly mentioned in cases does.
3) Execute the statement s_j associated with the value found.

Syntax-Directed Translation of Switch statements

code to evaluate E into t

goto test

L_1 : code for S_1

goto next

L_2 : code for S_2

goto next

:

:

L_{n-1} : code for S_{n-1}

goto next

L_n : code for S_n

goto next

test : if $t = V_1$ goto L_1

if $t = V_2$ goto L_2

⋮

if $t = V_{n-1}$ goto L_{n-1}

goto L_n

next :

→ The translation into the above scheme when we see the keyword switch, we generate two new labels test and next, and a new temporary t.

→ Then as we passe the Expression E, we generate code to evaluate E into t.

After processing E, we generate the jump goto test.

→ Then, as we see each case keyword, we create a new label Li and enter it into Symbol Table.

→ we place in a queue, used only to store cases, a value-label pair consisting of the value V_i of the case constant and L_i (or) a pointer to symbol-Table entry for L_i).

→ we process each statement case $V_i : S_i$ by emitting the label L_i attached to the code for S_i, followed by Jump goto next.

→ When the end of switch is found, we are ready to generate the code for n-way branch.

→ Reading the queue of value-label pairs, we can generate a sequence of three-address statements of the form as follows.

```
case t  V1 L1
case t  V2 L2
 :
case t  Vn-1 Ln-1
case t  t  Ln
label next
```

Three address code instructions used to translate switch statement.

~~The case t~~

→ The case t V_i L_i instruction is a synonym for if $t = V_i$ goto L_i.

Another translation of a switch statement

```
        code to evaluate E into t
        if t != V1 goto L1
        code for S1
        goto next
L1 :    if t != V2 goto L2
        code for S2
        goto next
L2 :
        ------
Ln-2 :  if t != Vn-1 goto Ln-1
        code for Sn-1
        goto next
Ln-1 :  code for Sn
next :
```

Intermediate code for procedures

→ In three-address code, a function call is unraveled into the evaluation of parameters in preparation for a call, followed by call itself.

Eg: Suppose that 'a' is an array of integers, and 'f' is a function from integers to integers.

```
n = f(a[i]);
```

Three address code

1) $t_1 = i * 4$
2) $t_2 = a[t_1]$
3) param t_2
4) $t_3 = call\ f, 1$
5) $n = t_3$

→ The first two lines compute the value of the expression a[i] into temporary t_2.

→ Line 3 makes t_2 an actual parameter for the call on Line 4 of f with one paramter.

→ Line 4 assigns the value returned by the function call to t_3.

→ Line 5 assigns the returned value to n.

$D \rightarrow$ define T id (F) {S}
$F \rightarrow \epsilon \mid T\ id, F$
$S \rightarrow$ return E ;
$E \rightarrow id\ (A)$
$A \rightarrow \epsilon \mid E, A$

→ Non Terminals D and T generate declarations and types.

→ A function definition generated by D consists of keyword define, a return type, ~~and~~ the function name, formal parameters in parantheses and function body consisting of statement.

→ Nonterminal F generates zero (or) more formal parameters, where formal parameters consists of a type followed by identifier.

→ Non Terminals S and E generates statements and Expressions.

→ The production for S adds a statement that returns the value of an expression.

→ The production for E adds function calls, with actual parameters generated by A.

→ An actual parameter is an expression.