UNIT - III

Syntax-Directed Translation: Syntax-Directed Definitions, Evaluation Orders for SDD's, Applications of Syntax-Directed Translation, Syntax-Directed Translation Schemes, Implementing L-Attributed SDD's.

Intermediate-Code Generation: Variants of Syntax Trees, Three-Address Code, Types and Declarations, Type Checking, Control Flow, Switch-Statements, Intermediate Code for Procedures.

3.1 SYNTAX-DIRECTED DEFINITIONS

Syntax Directed Definition (SDD) is a kind of abstract specification. It is generalization of context free grammar in which each grammar production **X –> a** is associated with it a set of production rules of the form s = f(b1, b2, ……bk) where s is the attribute obtained from function f. The attribute can be a string, number, type or a memory location. Semantic rules are fragments of code which are embedded usually at the end of production and enclosed in curly braces ({ }).

Example:

E --> E1 + T { E.val = E1.val + T.val}

Annotated Parse Tree – The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

Features –

- High level specification
- Hides implementation details
- Explicit order of evaluation is not specified

Types of attributes – There are two types of attributes:

1. **Synthesized Attributes –** These are those attributes which derive their values from their children nodes i.e. value of synthesized attribute at node is computed from the values of attributes at children nodes in parse tree.

Example:

E --> E1 + T { E.val = E1.val + T.val}

In this, E.val derive its values from E1.val and T.val

Computation of Synthesized Attributes –

- Write the SDD using appropriate semantic rules for each production in given grammar.
- The annotated parse tree is generated and attribute values are computed in bottom up manner.
- The value obtained at root node is the final output.

Example: Consider the following grammar

S --> E

E --> E1 + T

E --> T

T --> T1 * F

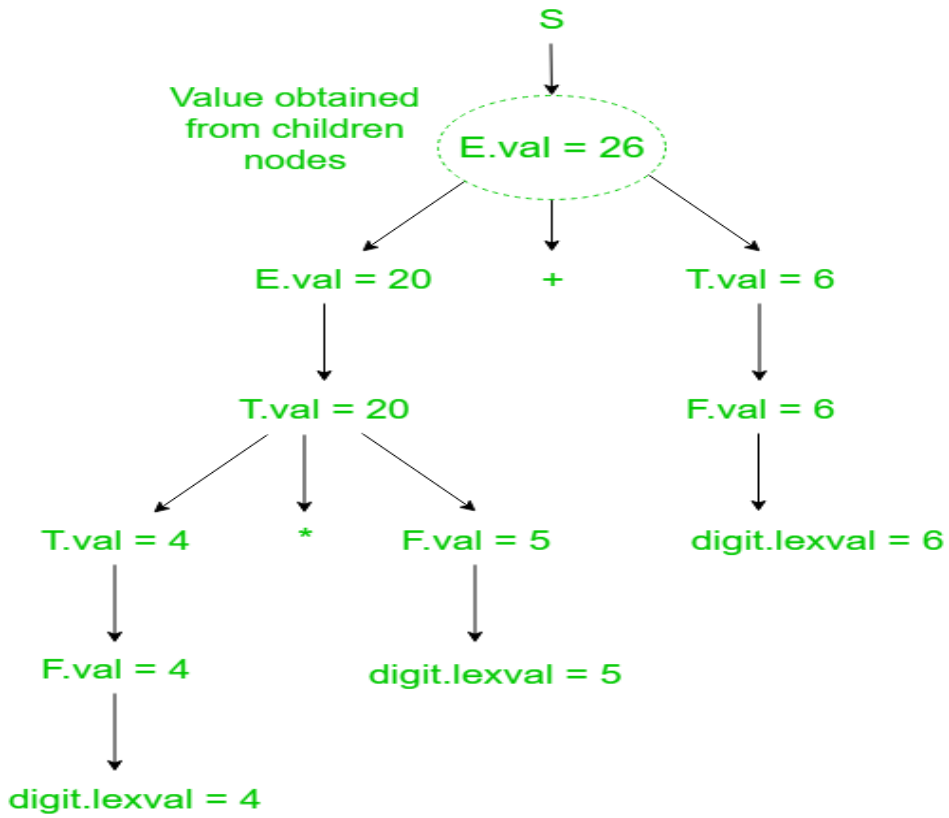
T --> F

F --> digit

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> E	Print(E.val)
E --> E_1 + T	E.val = E_1.val + T.val
E --> T	E.val = T.val
T --> T_1 * F	T.val = T_1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval

Let us assume an input string **4 * 5 + 6** for computing synthesized attributes. The annotated parse tree for the input string is



Annotated Parse Tree

For computation of attributes we start from leftmost bottom node. The rule F –> digit is used to reduce digit to F and the value of digit is obtained from lexical analyzer which becomes value of F i.e. from semantic action F.val = digit.lexval. Hence, F.val = 4 and since T is parent node of F so, we get T.val = 4 from semantic action T.val = F.val. Then, for T –> T1 * F production, the corresponding semantic action is T.val = T1.val * F.val . Hence, T.val = 4 * 5 = 20

Similarly, combination of E1.val + T.val becomes E.val i.e. E.val = E1.val + T.val = 26. Then, the production S –> E is applied to reduce E.val = 26 and semantic action associated with it prints the result E.val . Hence, the output will be 26.

2. Inherited Attributes – These are the attributes which derive their values from their parent or sibling nodes i.e. value of inherited attributes are computed by value of parent or sibling nodes.
Example:

A --> BCD { C.in = A.in, C.type = B.type }

Computation of Inherited Attributes –

- Construct the SDD using semantic actions.
- The annotated parse tree is generated and attribute values are computed in top down manner.

Example: Consider the following grammar

S --> T L

T --> int

T --> float

T --> double

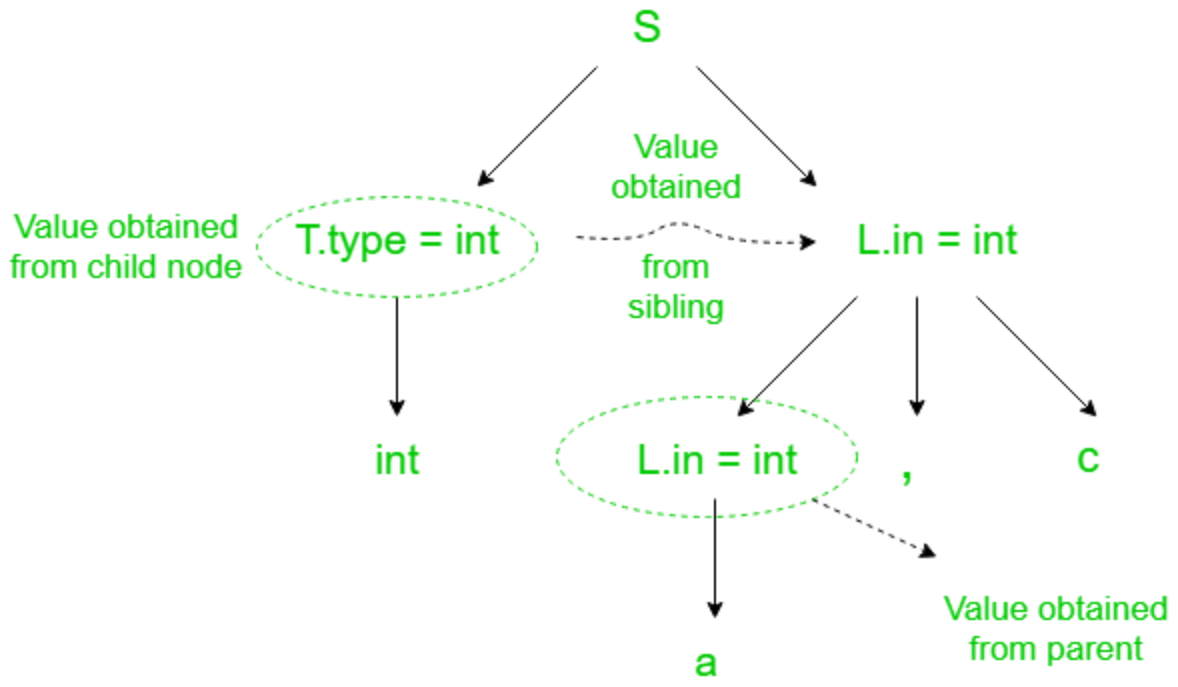
L --> L1, id

L --> id

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> T L	L.in = T.type
T --> int	T.type = int
T --> float	T.type = float
T --> double	T.type = double
L --> L_1 , id	L_1.in = L.in Enter_type(id.entry , L.in)
L --> id	Entry_type(id.entry , L.in)

Let us assume an input string **int a, c** for computing inherited attributes. **The annotated parse tree for the input string is**



The value of L nodes is obtained from T.type (sibling) which is basically lexical value obtained as int, float or double. Then L node gives type of identifiers a and c. The computation of type is done in top down manner or preorder traversal. Using function Enter_type the type of identifiers a and c is inserted in symbol table at corresponding id.entry.

3.2 EVALUATION ORDER FOR SDD

Evaluation order for SDD includes how the SDD(Syntax Directed Definition) is evaluated with the help of attributes, dependency graphs, semantic rules, and S and L attributed definitions. SDD helps in the semantic analysis in the compiler so it's important to know about how SDDs are evaluated and their evaluation order. This article provides detailed information about the SDD evaluation. It requires some basic knowledge of grammar, production, parses tree, annotated parse tree, synthesized and inherited attributes.

Terminologies:

- **Parse Tree:** A parse tree is a tree that represents the syntax of the production hierarchically.
- **Annotated Parse Tree:** Annotated Parse tree contains the values and attributes at each node.
- **Synthesized Attributes:** When the evaluation of any node's attribute is based on children.
- **Inherited Attributes:** When the evaluation of any node's attribute is based on children or parents.

Dependency Graphs:

A dependency graph provides information about the order of evaluation of attributes with the help of edges. It is used to determine the order of evaluation of attributes according to the semantic rules of the production. An edge from the first node attribute to the second node attribute gives the information that first node attribute evaluation is required for the evaluation of the second node attribute. Edges represent the semantic rules of the corresponding production.

Dependency Graph Rules: A node in the dependency graph corresponds to the node of the parse tree for each attribute. Edges (first node from the second node)of the dependency graph represent that the attribute of the first node is evaluated before the attribute of the second node.

Ordering the Evaluation of Attributes:

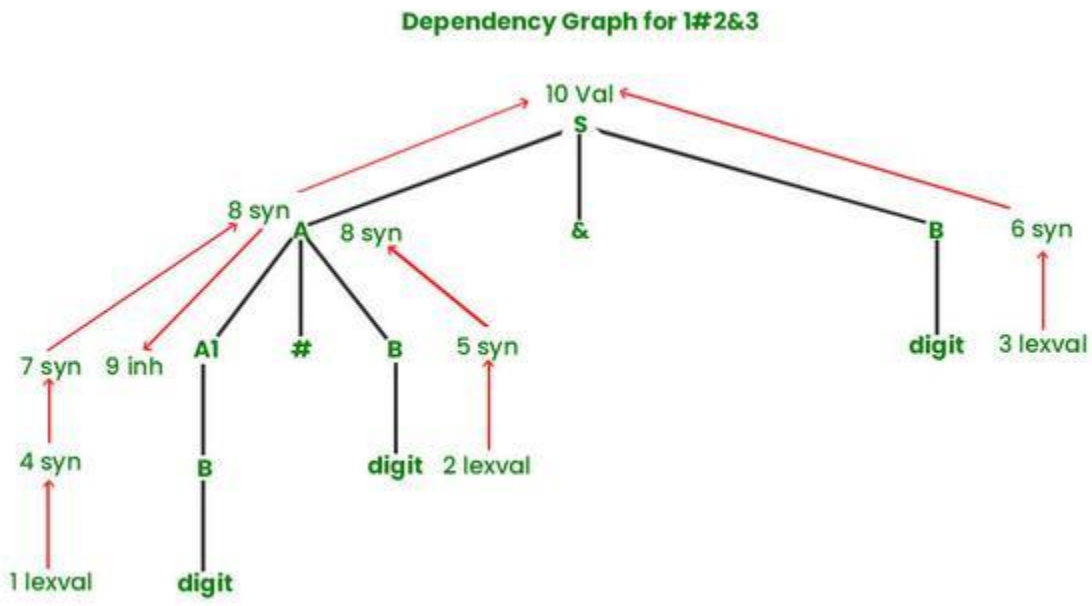
The dependency graph provides the evaluation order of attributes of the nodes of the parse tree. An edge(i.e. first node to the second node) in the dependency graph represents that the attribute of the second node is dependent on the attribute of the first node for further evaluation. This order of evaluation gives a linear order called **topological order**.

There is no way to evaluate SDD on a parse tree when there is a cycle present in the graph and due to the cycle, no topological order exists

. Production Table		
S.No.	Productions	Semantic Rules
1.	S ⇢ A & B	S.val = A.syn + B.syn
2.	A ⇢ A1 # B	A.syn = A1.syn * B.syn A1.inh = A.syn
3.	A1 ⇢ B	A1.syn = B.syn
4.	B ⇢ digit	B.syn = digit.lexval

S.No.	Productions	Semantic Rules
1.	S ⇢ A & B	S.val = A.syn + B.syn
2.	A ⇢ A1 # B	A.syn = A1.syn * B.syn A1.inh = A.syn
3.	A1 ⇢ B	A1.syn = B.syn
4.	B ⇢ digit	B.syn = digit.lexval

Annotated Parse Tree For 1#2&3



Dependency Graph For 1#2&3

Explanation of dependency graph:

Node number in the graph represents the order of the evaluation of the associated attribute. Edges in the graph represent that the second value is dependent on the first value.

Table-1 represents the attributes corresponding to each node.
Table-2 represents the semantic rules corresponding to each edge.

Table-1

Node	Attribute
1	digit.lexval
2	digit.lexval
3	digit.lexval
4	B.syn
5	B.syn
6	B.syn
7	A1.syn
8	A.syn
9	A1.inh
10	S.val

S-Attributed Definitions:

S-attributed SDD can have only synthesized attributes. In this type of definitions semantic rules are placed at the end of the production only. Its evaluation is based on bottom up parsing.

Example: S ⇢ AB { S.x = f(A.x | B.x) }

L-Attributed Definitions:

L-attributed SDD can have both synthesized and inherited (restricted inherited as attributes can only be taken from the parent or left siblings). In this type of definition, semantics rules can be placed anywhere in the RHS of the production. Its evaluation is based on inorder (topological sorting).

Example: S ⇢ AB {A.x = S.x + 2} or S ⇢ AB { B.x = f(A.x | B.x) } or S ⇢ AB { S.x = f(A.x | B.x) }

Note:

- Every S-attributed grammar is also L-attributed.
- For L-attributed evaluation in order of the annotated parse tree is used.
- For S-attributed reverse of the rightmost derivation is used.

Semantic Rules with controlled side-effects:

Side effects are the program fragment contained within semantic rules. These side effects in SDD can be controlled in two ways: Permit incidental side effects and constraint admissible evaluation orders to have the same translation as any admissible order.

3.3 APPLICATION OF SYNTAX DIRECTED TRANSLATION :

1. SDT is used for Executing Arithmetic Expression.
2. In the conversion from infix to postfix expression.
3. In the conversion from infix to prefix expression.
4. It is also used for Binary to decimal conversion.
5. In counting number of Reduction.
6. In creating a Syntax tree.
7. SDT is used to generate intermediate code.
8. In storing information into symbol table.
9. SDT is commonly used for type checking also.

3.4 SYNTAX DIRECTED TRANSLATION SCHEME

- The Syntax directed translation scheme is a context -free grammar.
- The syntax directed translation scheme is used to evaluate the order of semantic rules.
- In translation scheme, the semantic rules are embedded within the right side of the productions.
- The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

Example

Production	Semantic Rules
S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

Implementation of Syntax directed translation

Syntax direct translation is implemented by constructing a parse tree and performing the actions in a left to right depth first order. SDT is implementing by parse the input and produce a parse tree as a result.

Example

Production	Semantic Rules
S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }

E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

Parse tree for SDT:

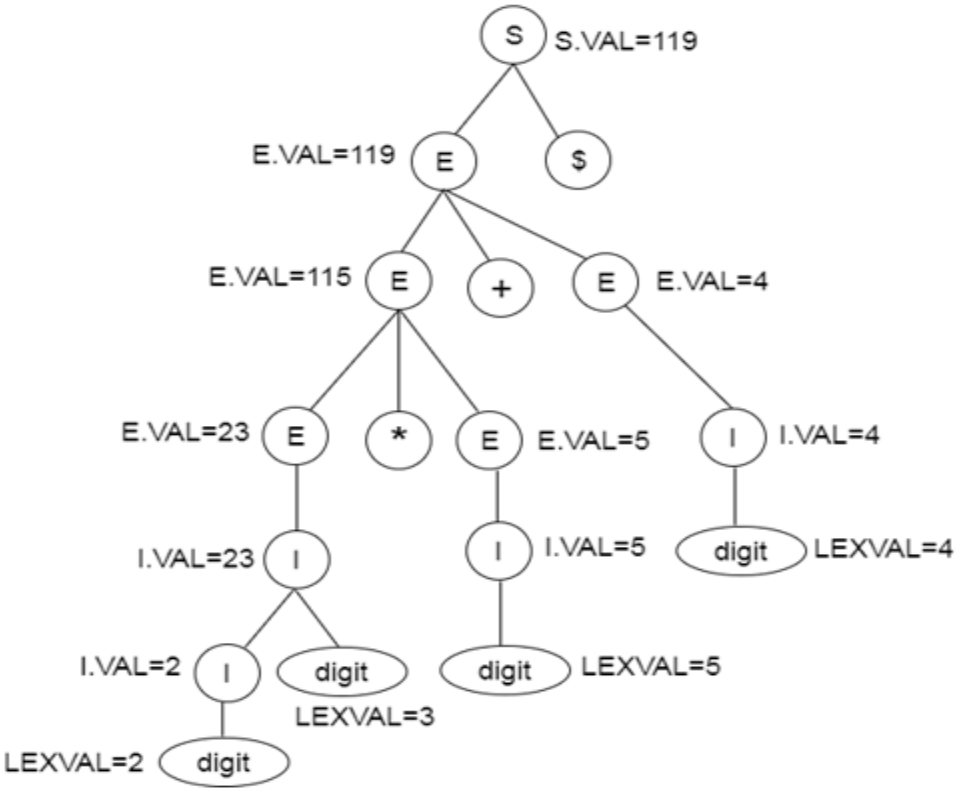


Fig: Parse tree

3.5 SYNTAX-DIRECTED TRANSLATION SCHEMES

Syntax Directed Translation is a set of productions that have semantic rules embedded inside it. The syntax-directed translation helps in the semantic analysis phase in the compiler. SDT has semantic actions along with the production in the grammar. This article is about postfix SDT

and postfix translation schemes with parser stack implementation of it. Postfix SDTs are the SDTs that have semantic actions at the right end of the production. This article also includes SDT with actions inside the production, eliminating left recursion from SDT and SDTs for L-attributed definitions.

3.6 SDT FOR L-ATTRIBUTED DEFINITIONS: SDT with L-attributed definitions involves both synthesized and inherited attributes in the production. To convert an L-attributed definition into its equivalent SDT follow the underlying rules:

- When the attributes are inherited attributes of any non-terminal then place the action immediately before the non-terminal in the production.
- When the attributes of the non-terminal are synthesized then, place the action at the right end of that production.

IMPLEMENTING L-ATTRIBUTED SDD'S

Methods to do translation by traversing a parse tree:

1. Build the parse tree and annotate.
2. Build the parse tree , add actions , and execute the actions in preorder. This works for L attributed definition.

Methods for translation for translation during parsing:

1. Use a recursive – descent parser with one function for each non-terminal. The function for Non-terminal. A receives inherited attributed of A as arguments and returns the synthesized attributed of A.
2. Generate code using a recursive – descent parser.
3. Implement a SDT in conjunction with an LL-parser.

Translation during Recursive – Descent Parsing:

A recursive – descent parser has a function A for each non-terminal A.

a) The argument of function A are the inherited attributed of non-terminal A.

b) The return value of function A is the collection of synthesize attributes of non-terminal A.

c) Preserve, in local variable, the values of all attributes needed to compute inherited attribute for non-terminals in the body or synthesize attribute for the head non terminal.

d) Call functions corresponding to non-terminals in the body of the selected production, providing them with the proper arguments.

A -> X1 X2 ---------- (1)

Attributes for A: a.syn , a.inh

Attributes for X1: x1.syn Attributes of X2: x2.inh

We know , inherited attributes for non-terminal in the body of production can be the function of

inherited of parent and attributes of non-terminal left to it.

x2.inh = f(x1.syn, a.inh)

and, synthesize attributes of the head of the production is the function of synthesized attributes of the children.

a.syn = fun(x1.syn, x2.syn)

Equivalent SDT, for production (1):

A -> X1 { x2 = f(x.syn, a.inh) } X2 { a-syn = f(x1.syn, x2.syn) }

1. Inherited attributed rule of the non-terminal of the body of production will come just before that NT in the that production.
2. If A (head of the production) has any synthesized attributed rules, then that will come after all non-terminal of the body of production as the fragmented code.

For example:

T -> F T' { T'.inh = F.val / T.val = T'.syn }

T' -> * F T1' { T1'.inh = T'.inh * F.val / T'.syn = T1'.syn }

T' -> epsilon { T'.syn = T'.inh }

F -> id { F.val = id.val }

Equivalent SDT:

T -> F { T'.inh = F.val } T' { T.val = T'.syn }

T’ -> * F { T1’.inh = T’.inh * F.val } T1’ { T’.syn = T’.inh }

T’ -> epsilon { T’.syn = T’.inh }

F -> id { F.val = id.val }

Writing function for a non terminal

A( )

{

Declare : a-syn, x1-syn, x2.inh, x2-syn;

If(current symbol (a) == Terminal X1)

Move i/p pointer to next symbol of the input string

Else if (X2 is NT)

x1-syn = X1( );

x2.inh = f (x1-syn);

x2-syn = X2(x2.inh);

a.syn = f (x1-syn,x2-syn);

return a-syn;

}

Now, functions for the non-terminals introduced in the above example:

F( )

{

Declaration: F-val;

If(id == T and Id matched with the input symbol)

Move input pointer to the next symbol.

F.val = id.val (get from the lexical analyser)

Return F.val;

}

T( )

{

If(F matched with the input symbol & F == T)

Move input pointer to the next symbol.

If (F == NT)

{

F-val = F( );

T'-inh = F-val;

}

If(T' == a && T' is T)

Move ptr. To next;

If(T' == NT)

{

T'-syn = T'(T'-inh);

T-val = T'-syn;

Return T-val;

}

3.7 VARIANTS OF SYNTAX TREES

A syntax tree basically has two variants which are described below:

1. Directed Acyclic Graphs for Expressions (DAG)
2. The Value-Number Method for Constructing DAGs

Nodes in a syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct. A directed acyclic graph (hereafter called a *DAG)* for an expression identifies the *common subexpressions* (subexpressions that occur more than once) of the expression. As we shall see in this section, DAG's can be constructed by using the same techniques that construct syntax trees.

1. Directed Acyclic Graphs for Expressions

Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior codes corresponding to operators. The difference is that a node *N* in a DAG has more than

one parent if *N* represents a com-mon subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression. Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Example 6.1: Figure 6.3 shows the DAG for the expression

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

The leaf for a has two parents, because a appears twice in the expression. More interestingly, the two occurrences of the common subexpression b - c are represented by one node, the node labeled —. That node has two parents, representing its two uses in the subexpressions a* (b - c) and (b - c)*d . Even though b and c appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression b - c . •

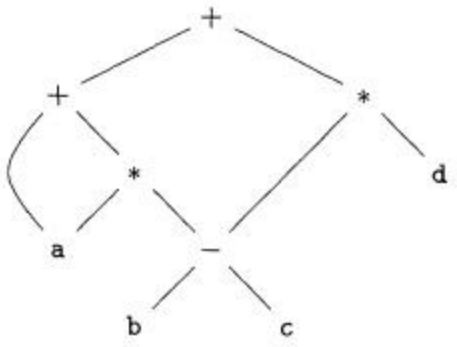


Figure 6.3: Dag for the expression a + a * (b - c) + (b - c) * d

The SDD of Fig. 6.4 can construct either syntax trees or DAG's. It was used to construct syntax trees in Example 5.11, where functions Leaf and Node created a fresh node each time they were called. It will construct a DAG if, before creating a new node, these functions first check whether an identical node already exists. If a previously created identical node exists, the existing node is returned. For instance, before constructing a new node, Node(op, left, right) we check whether there is already a node with label op, and children left and right, in that order. If so, Node returns the existing node; otherwise, it creates a new node.

Example 6.2 : The sequence of steps shown in Fig. 6.5 constructs the DAG in Fig. 6.3, provided Node and Leaf return an existing node, if possible, as discussed abve. We assume that entry-a points to the symbol-table entry for a, and similarly for the other identifiers.

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf(\mathbf{id}, \mathbf{id}.entry)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf(\mathbf{num}, \mathbf{num}.val)$

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG's

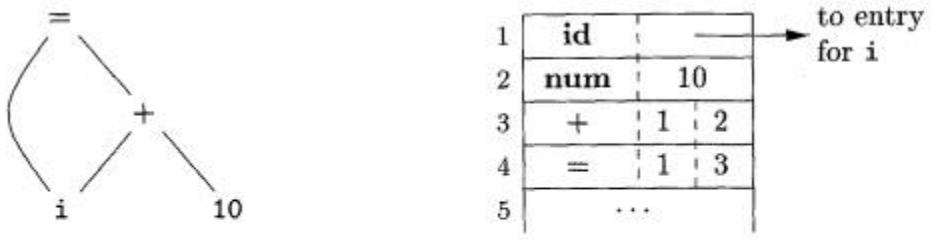
1) $p_1 = Leaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\mathbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\mathbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\mathbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

Figure 6.5: Steps for constructing the DAG of Fig. 6.3

When the call to Leaf (id, entry-a) is repeated at step 2, the node created by the previous call is returned, so p2 = Pi • Similarly, the nodes returned at steps 8 and 9 are the same as those returned at steps 3 and 4 (i.e., ps = p3 and PQ = p^). Hence the node returned at step 10 must be the same at that returned at step 5; i.e., pio = P5- •

2. The Value-Number Method for Constructing DAG's

Often, the nodes of a syntax tree or DAG are stored in an array of records, as suggested by Fig. 6.6. Each row of the array represents one record, and therefore one node. In each record, the first field is an operation code, indicating the label of the node. In Fig. 6.6(b), leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant, in this case), and interior nodes have two additional fields indicating the left and right children.



(a) DAG (b) Array.

Figure 6.6: Nodes of a DAG for $i = i + 10$ allocated in an array

In this array, we refer to nodes by giving the integer index of the record for that node within the array. This integer historically has been called the value number for the node or for the expression represented by the node. For instance, in Fig. 6.6, the node labeled -I- has value number 3, and its left and right children have value numbers 1 and 2, respectively. In practice, we could use pointers to records or references to objects instead of integer indexes, but we shall still refer to the reference to a node as its "value number." If stored in an appropriate data structure, value numbers help us construct expression DAG's efficiently; the next algorithm shows how.

Suppose that nodes are stored in an array, as in Fig. 6.6, and each node is referred to by its value number. Let the signature of an interior node be the triple (op,/,r), where op is the label, I its left child's value number, and r its right child's value number. A unary operator may be assumed to have r = 0-

Algorithm: The value-number method for constructing the nodes of a DAG.

INPUT : Label op, node /, and node r.

OUTPUT : The value number of a node in the array with signature (op, l,r).

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

While Algorithm 6.3 yields the desired output, searching the entire array every time we are asked to locate one node is expensive, especially if the array holds expressions from an entire program. A more efficient approach is to use a hash table, in which the nodes are put into "buckets," each of which typically will have only a few nodes. The hash table is one of several data structures that support *dictionaries* efficiently.1 A dictionary is an abstract data type that allows us to insert and delete elements of a set, and to determine whether a given element is currently in the set. A good data structure for dictionaries, such as a hash table, performs each of these operations in time that is constant or close to constant, independent of the size of the set.

To construct a hash table for the nodes of a DAG, we need a hash function h that computes the index of the bucket for a signature (op, I, r), in a way that distributes the signatures across buckets, so that it is unlikely that any one bucket will get much more than a fair share of the nodes. The bucket index h(op, I, r) is computed deterministically from op, I, and r, so that we may repeat the calculation and always get to the same bucket index for node (op, I, r).

The buckets can be implemented as linked lists, as in Fig. 6.7. An array, indexed by hash value, holds the bucket headers, each of which points to the first cell of a list. Within the linked list for a bucket, each cell holds the value number of one of the nodes that hash to that bucket. That is, node (op,l,r) can be found on the list whose header is at index h(op,l,r) of the array.

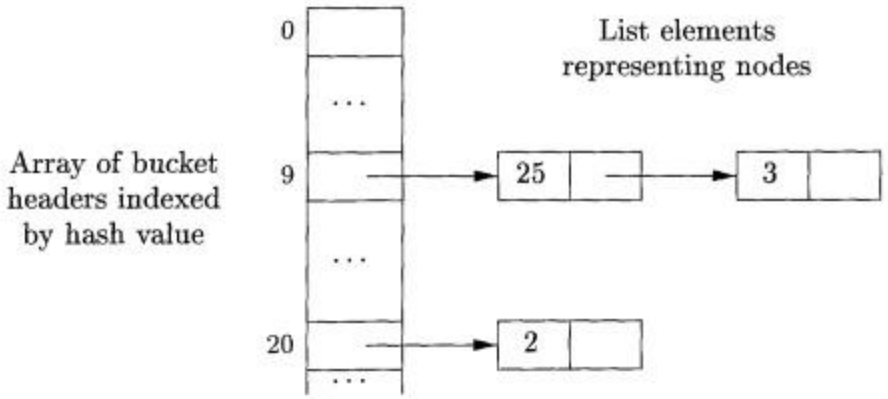


Figure 6.7: Data structure for searching buckets

Thus, given the input node op, I, and r, we compute the bucket index h(op,l,r) and search the list of cells in this bucket for the given input node. Typically, there are enough buckets so that no list has more than a few cells. We may need to look at all the cells within a bucket, however, and for each value number v found in a cell, we must check whether the signature (op,l,r) of the input node matches the node with value number v in the list of cells (as in Fig. 6.7) . If we find a match, we return v. If we find no match, we know no such node can exist in any other bucket, so we create a new cell, add it to the list of cells for bucket index h(op, l,r), and return the value number in that new cell.

3.8 THREE-ADDRESS CODE

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code.It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

General representation –

a = b op c

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

Example-1: Convert the expression a * – (b + c) into three address code.

$t_1 = b + c$
$t_2 = uminus\ t_1$
$t_3 = a * t_2$

Example-2: Write three address code for following code

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

$i = 1$
$L : t_1 = x * 5$
$t_2 = \&a$
$t_3 = sizeof(int)$
$t_4 = t_3 * i$
$t_5 = t_2 + t_4$
$*t_5 = t_1$
$i = i + 1$
$if\ i<=10\ goto\ L$

Implementation of Three Address Code –
There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple –
It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage –

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage –

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

Example – Consider expression a = b * – c + b * – c.
The three address code is:

t1 = uminus c

t2 = b * t1

t3 = uminus c

t4 = b * t3

t5 = t2 + t4

a = t5

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

2. Triples –
This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage –

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example – Consider expression a = b * – c + b * – c

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. Indirect Triples –
This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example – Consider expression a = b * – c + b * – c

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

Question – Write quadruple, triples and indirect triples for following expression : (x + y) * (y + z) + (x + y + z)

Explanation – The three address code is:

t1 = x + y

t2 = y + z

t3 = t1 * t2

t4 = t1 + z

t5 = t3 + t4

#	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Quadruple representation

#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples representation

List of pointers to table

#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triples representation

3.9 TYPES AND DECLARATIONS

Type checking is the process of verifying and enforcing constraints of types in values. A compiler must check that the source program should follow the syntactic and semantic conventions of the source language and it should also check the type rules of the language. It allows the programmer to limit what types may be used in certain circumstances and assigns types to values. The type-checker determines whether these values are used appropriately or not.

It checks the type of objects and reports a type error in the case of a violation, and incorrect types are corrected. Whatever the compiler we use, while it is compiling the program, it has to follow the type rules of the language. Every language has its own set of type rules for the language. We know that the information about data types is maintained and computed by the compiler.

The information about data types like INTEGER, FLOAT, CHARACTER, and all the other data types is maintained and computed by the compiler. The compiler contains modules, where the type checker is a module of a compiler and its task is type checking.

Coercion

Conversion from one type to another type is known as **implicit** if it is to be done automatically by the compiler. Implicit type conversions are also called **Coercion** and coercion is limited in many languages.

Example: An integer may be converted to a real but real is not converted to an integer.

Conversion is said to be **Explicit** if the programmer writes something to do the Conversion.

Tasks:

1. has to allow “Indexing is only on an array”
2. has to check the range of data types used
3. INTEGER (int) has a range of -32,768 to +32767
4. FLOAT has a range of 1.2E-38 to 3.4E+38.

3.10 TYPES OF TYPE CHECKING

There are two kinds of type checking:

1. Static Type Checking.
2. Dynamic Type Checking.

Static Type Checking:

Static type checking is defined as type checking performed at compile time. It checks the type variables at compile-time, which means the type of the variable is known at the compile time. It generally examines the program text during the translation of the program. Using the type rules of a system, a compiler can infer from the source text that a function (fun) will be applied to an operand (a) of the right type each time the expression fun(a) is evaluated.

Examples of Static checks include:

- **Type-checks:** A compiler should report an error if an operator is applied to an incompatible operand. For example, if an array variable and function variable are added together.
- **The flow of control checks:** Statements that cause the flow of control to leave a construct must have someplace to which to transfer the flow of control. For example, a break statement in C causes control to leave the smallest enclosing while, for, or switch statement, an error occurs if such an enclosing statement does not exist.
- **Uniqueness checks:** There are situations in which an object must be defined only once. For example, in Pascal an identifier must be declared uniquely, labels in a case statement must be distinct, and else a statement in a scaler type may not be represented.
- **Name-related checks:** Sometimes the same name may appear two or more times. For example in Ada, a loop may have a name that appears at the beginning and end of the construct. The compiler must check that the same name is used at both places.

The Benefits of Static Type Checking:

1. Runtime Error Protection.
2. It catches syntactic errors like spurious words or extra punctuation.
3. It catches wrong names like Math and Predefined Naming.
4. Detects incorrect argument types.
5. It catches the wrong number of arguments.
6. It catches wrong return types, like return “70”, from a function that’s declared to return an int.

Dynamic Type Checking:

Dynamic Type Checking is defined as the type checking being done at run time. In Dynamic Type Checking, types are associated with values, not variables. Implementations of dynamically

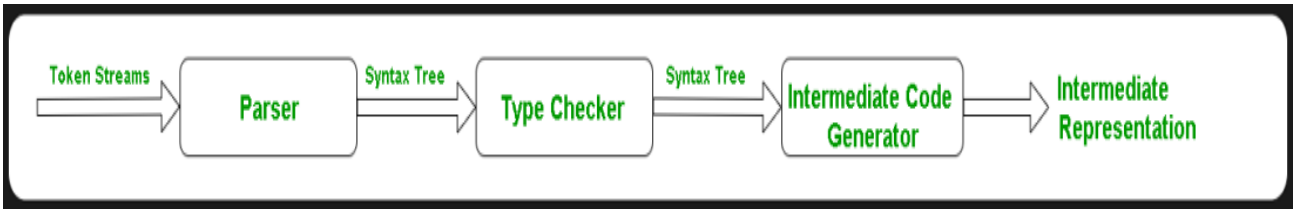
type-checked languages runtime objects are generally associated with each other through a type tag, which is a reference to a type containing its type information. Dynamic typing is more flexible. A static type system always restricts what can be conveniently expressed. Dynamic typing results in more compact programs since it is more flexible and does not require types to be spelled out. Programming with a static type system often requires more design and implementation effort.

Languages like Pascal and C have static type checking. Type checking is used to check the correctness of the program before its execution. The main purpose of type-checking is to check the correctness and data type assignments and type-casting of the data types, whether it is syntactically correct or not before their execution.
Static Type-Checking is also used to determine the amount of memory needed to store the variable.

The design of the type-checker depends on:

1. Syntactic Structure of language constructs.
2. The Expressions of languages.
3. The rules for assigning types to constructs (semantic rules).

The Position of the Type checker in the Compiler:



Type checking in Compiler

The token streams from the lexical analyzer are passed to the PARSER. The PARSER will generate a syntax tree. When a program (source code) is converted into a syntax tree, the type-checker plays a Crucial Role. So, by seeing the syntax tree, you can tell whether each data type is handling the correct variable or not. The Type-Checker will check and if any modifications are present, then it will modify. It produces a syntax tree, and after that, INTERMEDIATE CODE Generation is done.

Overloading:

An Overloading symbol is one that has different operations depending on its context.

Overloading is of two types:

1. Operator Overloading
2. Function Overloading

Operator Overloading: In Mathematics, the arithmetic expression “x+y” has the addition operator ‘+’ is overloaded because ‘+’ in “x+y” have different operators when ‘x’ and ‘y’ are integers, complex numbers, reals, and Matrices.

Example: In Ada, the parentheses '()' are overloaded, the ith element of the expression A(i) of an Array A has a different meaning such as a 'call to function 'A' with argument 'i' or an explicit conversion of expression i to type 'A'. In most languages the arithmetic operators are overloaded.

Function Overloading: The Type Checker resolves the Function Overloading based on types of arguments and Numbers.

Example:

```
E-->E1(E2)
        {
          E.type:= if E2.type = s
          E1.type = s -->t then t
                        else type_error
        }
```

3.11 TRANSLATION OF ASSIGNMENT STATEMENTS

In the syntax directed translation, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array and records.

Consider the grammar

1. S → id := E
2. E → E1 + E2
3. E → E1 * E2
4. E → (E1)
5. E → id

The translation scheme of above grammar is given below:

Production rule	Semantic actions

S → id :=E	{p = look_up(id.name);
	If p ≠ nil then
	Emit (p = E.place)
	Else
	Error;
	}

E → E1 + E2	{E.place = newtemp(); Emit (E.place = E1.place '+' E2.place) }
E → E1 * E2	{E.place = newtemp(); Emit (E.place = E1.place '*' E2.place) }
E → (E1)	{E.place = E1.place}

E → id	{p = look_up(id.name);
	If p ≠ nil then
	Emit (p = E.place)
	Else
	Error;
	}

- The p returns the entry for id.name in the symbol table.
- The Emit function is used for appending the three address code to the output file. Otherwise it will report an error.
- The newtemp() is a function used to generate new temporary variables.
- E.place holds the value of E.

3.12 BOOLEAN EXPRESSIONS

Boolean expressions have two primary purposes. They are used for computing the logical values. They are also used as conditional expression using if-then-else or while-do.

Consider the grammarhttps://www.javatpoint.com/boolean-expressions

E → E OR E

E → E AND E

E → NOT E

E → (E)

E → id relop id

E → TRUE

E → FALSE

The relop is denoted by <, >, <, >.

The AND and OR are left associated. NOT has the higher precedence then AND and lastly OR.

Production rule	Semantic actions
E → E1 OR E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'OR' E2.place) }
E → E1 + E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'AND' E2.place) }
E → NOT E1	{E.place = newtemp(); Emit (E.place ':=' 'NOT' E1.place) }
E → (E1)	{E.place = E1.place}
E → id relop id2	{E.place = newtemp(); Emit ('if' id1.place relop.op id2.place 'goto' nextstar + 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') }

The EMIT function is used to generate the three address code and the newtemp( ) function is used to generate the temporary variables.

The E → id relop id2 contains the next_state and it gives the index of next three address statements in the output sequence.

Here is the example which generates the three address code using the above translation scheme:https://www.javatpoint.com/boolean-expressionshttps://www.javatpoint.com/boolean-expressions

p>q AND r<s OR u>r

100: **if** p>q **goto** 103

101: t1:=0

102: **goto** 104

103: t1:=1

104: **if** r>s **goto** 107

105: t2:=0

106: **goto** 108

107: t2:=1

108: **if** u>v **goto** 111

109: t3:=0

110: **goto** 112

111: t3:= 1

112: t4:= t1 AND t2

113: t5:= t4 OR t3

3.13 STATEMENTS THAT ALTER THE FLOW OF CONTROL

The goto statement alters the flow of control. If we implement goto statements then we need to define a LABEL for a statement. A production can be added for this purpose:

S → LABEL : S

LABEL → id

In this production system, semantic action is attached to record the LABEL and its value in the symbol table.

Following grammar used to incorporate structure flow-of-control constructs:

https://www.javatpoint.com/statements-that-alter-the-flow-of-controlhttps://www.javatpoint.com/statements-that-alter-the-flow-of-controlhttps://www.javatpoint.com/statements-that-alter-the-flow-of-controlS → **if** E then S

S → **if** E then S **else** S

S → **while** E **do** S

S → begin L end

S→ A

L→ L ; S

L → S

Here, S is a statement, L is a statement-list, A is an assignment statement and E is a Boolean-valued expression.

Translation scheme for statement that alters flow of control

- We introduce the marker non-terminal M as in case of grammar for Boolean expression.
- This M is put before statement in both if then else. In case of while-do, we need to put M before E as we need to come back to it after executing S.

- In case of if-then-else, if we evaluate E to be true, first S will be executed.
- After this we should ensure that instead of second S, the code after the if-then else will be executed. Then we place another non-terminal marker N after first S.

The grammar is as follows:https://www.javatpoint.com/statements-that-alter-the-flow-of-control

S → **if** E then M S

S → **if** E then M S **else** M S

S → **while** M E **do** M S

S → begin L end

S → A

L→ L ; M S

L → S

M → ∈

N → ∈

The translation scheme for this grammar is as follows:

Production rule	**Semantic actions**
S → if E then M S1	BACKPATCH (E.TRUE, M.QUAD) S.NEXT = MERGE (E.FALSE, S1.NEXT)
S → if E then M1 S1 else M2 S2	BACKPATCH (E.TRUE, M1.QUAD) BACKPATCH (E.FALSE, M2.QUAD) S.NEXT = MERGE (S1.NEXT, N.NEXT, S2.NEXT)
S → while M1 E do M2 S1	BACKPATCH (S1,NEXT, M1.QUAD) BACKPATCH (E.TRUE, M2.QUAD) S.NEXT = E.FALSE GEN (goto M1.QUAD)
S → begin L end	S.NEXT = L.NEXT
S → A	S.NEXT = MAKELIST ()

L → L ; M S	BACKPATHCH (L1.NEXT, M.QUAD) L.NEXT = S.NEXT
L → S	L.NEXT = S.NEXT
M → ∈	M.QUAD = NEXTQUAD
N→ ∈	N.NEXT = MAKELIST (NEXTQUAD) GEN (goto_)

3.14 POSTFIX TRANSLATION

In a production A → α, the translation rule of A.CODE consists of the concatenation of the CODE translations of the non-terminals in α in the same order as the non-terminals appear in α.

Production can be factored to achieve postfix form.

Postfix translation of while statement

The productionhttps://www.javatpoint.com/postfix-translationhttps://www.javatpoint.com/postfix-translation

S → **while** M1 E **do** M2 S1

Can be factored as:https://www.javatpoint.com/postfix-translationhttps://www.javatpoint.com/postfix-translation

S → C S1

C → W E **do**

W → **while**

A suitable transition scheme would be

Production Rule	Semantic Action
W → while	W.QUAD = NEXTQUAD
C → W E do	C W E do
S→ C S1	BACKPATCH (S1.NEXT, C.QUAD) S.NEXT = C.FALSE GEN (goto C.QUAD)

Postfix translation of for statement . The productionhttps://www.javatpoint.com/postfix-translation

1. S **for** L = E1 step E2 to E3 **do** S1

Can be factored as

F → **for** L

T → F = E1 by E2 to E3 **do**

S → T S1

3.15 ARRAY REFERENCES IN ARITHMETIC EXPRESSIONS

Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive location. Array can be one dimensional or two dimensional.

For one dimensional array:

1. A: array[low..high] of the ith elements is at:
2. base + (i-low)*width → i*width + (base - low*width)

Multi-dimensional arrays:

Row major or column major forms

- Row major: a[1,1], a[1,2], a[1,3], a[2,1], a[2,2], a[2,3]
- Column major: a[1,1], a[2,1], a[1, 2], a[2, 2],a[1, 3],a[2,3]

- In raw major form, the address of a[i1, i2] is
- Base+((i1-low1)*(high2-low2+1)+i2-low2)*width

Translation scheme for array elements

Limit(array, j) returns nj=highj-lowj+1

place: the temporaryor variables

offset: offset from the base, null if not an array reference

The production:

S → L := E

E → E+E

E → (E)

E → L

L → Elist]

L → id

Elist → Elist, E

Elist → id[E

A suitable transition scheme for array elements would be:

Production Rule	Semantic Action

S → L := E	{if L.offset = null then emit(L.place ':=' E.place)
	else EMIT (L.place'['L.offset ']' ':=' E.place);
	}

E → E+E	{E.place := newtemp; EMIT (E.place ':=' E1.place '+' E2.place); }
E → (E)	{E.place := E1.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); }
L → id	{L.place = lookup(id.name); L.offset = null; }

Elist → Elist, E	{t := newtemp;
	m := Elist1.ndim + 1;
	EMIT (t ':=' Elist1.place '*' limit(Elist1.array, m));
	EMIT (t, ':=' t '+' E.place);
	Elist.array = Elist1.array;
	Elist.place := t;
	Elist.ndim := m;
	}

Elist → id[E	{Elist.array := lookup(id.name); Elist.place := E.place Elist.ndim := 1; }

Where:

ndim denotes the number of dimensions.

limit(array, i) function returns the upper limit along with the dimension of array

width(array) returns the number of byte for one element of array.

3.16 PROCEDURES CALL

Procedure is an important and frequently used programming construct for a compiler. It is used to generate good code for procedure calls and returns.

Calling sequence:

The translation for a call includes a sequence of actions taken on entry and exit from each procedure. Following actions take place in a calling sequence:

- When a procedure call occurs then space is allocated for activation record.
- Evaluate the argument of the called procedure.
- Establish the environment pointers to enable the called procedure to access data in enclosing blocks.
- Save the state of the calling procedure so that it can resume execution after the call.
- Also save the return address. It is the address of the location to which the called routine must transfer after it is finished.
- Finally generate a jump to the beginning of the code for the called procedure.
-

Let us consider a grammar for a simple procedure call statementhttps://www.javatpoint.com/procedures-callhttps://www.javatpoint.com/procedures-call

S → call id(Elist)

Elist → Elist, E

Elist → E

A suitable transition scheme for procedure call would be:

Production Rule	**Semantic Action**

S → call id(Elist)	for each item p on QUEUE do GEN (param p) GEN (call id.PLACE)
Elist → Elist, E	append E.PLACE to the end of QUEUE
Elist → E	initialize QUEUE to contain only E.PLACE

Queue is used to store the list of parameters in the procedure call.

3.17 DECLARATIONS

When we encounter declarations, we need to lay out storage for the declared variables.

For every local name in a procedure, we create a ST(Symbol Table) entry containing:

1. The type of the name
2. How much storage the name requires

The production:

D → integer, id
D → real, id
D → D1, id

A suitable transition scheme for declarations would be:

Production rule	Semantic action
D → integer, id	ENTER (id.PLACE, integer) D.ATTR = integer
D → real, id	ENTER (id.PLACE, real) D.ATTR = real
D → D1, id	ENTER (id.PLACE, D1.ATTR) D.ATTR = D1.ATTR

3.18 CASE STATEMENTS

Switch and case statement is available in a variety of languages. The syntax of case statement is as follows:https://www.javatpoint.com/case-statementshttps://www.javatpoint.com/case-statements

switch E

begin

case V1: S1

case V2: S2

.

case Vn-1: Sn-1

default: Sn

end

The translation scheme for this shown below:

Code to evaluate E into Thttps://www.javatpoint.com/case-statementshttps://www.javatpoint.com/case-statementshttps://www.javatpoint.com/case-statements

goto TEST

L1: code **for** S1

goto NEXT

L2: code **for** S2

goto NEXT

.

Ln-1: code **for** Sn-1

goto NEXT

Ln: code **for** Sn

goto NEXT

TEST: **if** T = V1 **goto** L1

if T = V2goto L2

.

if T = Vn-1 **goto** Ln-1

goto

NEXT:

- When switch keyword is seen then a new temporary T and two new labels test and next are generated.
- When the case keyword occurs then for each case keyword, a new label Li is created and entered into the symbol table. The value of Vi of each case constant and a pointer to this symbol-table entry are placed on a stack.