UNIT - III
INTERMEDIATE CODE GENERATION

1. Intermediate code forms:

An intermediate code form of source program is an internal form of a program created by the compiler while translating the program created by the compiler while translating the program from a high –level language to assembly code(or)object code(machine code).an intermediate source form represents a more attractive form of target code than does assembly. An optimizing Compiler performs optimizations on the intermediate source form and produces an object module.

Analysis + syntheses=translation

Creates an generate targe code
Intermediate code

In the analysis –synthesis model of a compiler, the front-end translates a source program into an intermediate representation from which the back-end generates target code, in many compilers the source code is translated into a language which is intermediate in complexity between a HLL and machine code .the usual intermediate code introduces symbols to stand for various temporary quantities.

Parser Static Intermediate Code
checker code generator generator
position of intermediate code generator

We assume that the source program has already been parsed and statically checked.. the various intermediate code forms are:

a) Polish notation
b) Abstract syntax trees(or)syntax trees
c) Quadruples
d) Triples three address code
e) Indirect triples
f) Abstract machine code(or)pseudocopde a. postfix notation:

The ordinary (infix) way of writing the sum of a and b is with the operator in the middle: a+b. the postfix (or postfix polish)notation for the same expression places the operator at the right end, as ab+. In general, if e1 and e2 are any postfix expressions, and Ø to the values denoted by e1 and e2 is indicated in postfix notation nby e1e2Ø.no parentheses are needed in postfix notation because the position and priority (number of arguments) of the operators permits only one way to decode a postfix expression.

Example:

1. (a+b)*c in postfix notation is ab+c*,since ab+ represents the infix expression(a+b).
2. a*(b+c)is abc+* in postfix.
3. (a+b)*(c+d) is ab+cd+* in postfix.

Postfix notation can be generalized to k-ary operators for any k>=1.if k-ary operator Ø is applied to

postfix expression e1,e2,.....ek, then the result is denoted by e1e2.....ek Ø. if we know the priority of each operator then we can uniquely decipher any postfix expression by scanning it from either end.
Example:
Consider the postfix string ab+c*.
The right hand * says that there are two arguments to its left. since the next –to-rightmost symbol is c, simple operand, we know c must be the second operand of *.continuing to the left, we encounter the operator +.we know the sub expression ending in + makes up the first operand of *.continuing in this way ,we deduce that ab+c* is "parsed" as (((a,b)+),c)*.
b. syntax tree:
The parse tree itself is a useful intermediate-language representation for a source program, especially in optimizing compilers where the intermediate code needs to extensively restructure. A parse tree, however, often contains redundant information which can be eliminated, Thus producing a more economical representation of the source program. One such variant of a parse tree is what is called an (abstract) syntax tree, a tree in which each leaf represents an operand and each interior node an operator.

/

= :=
a ba + -
Exmples:
1) Syntax tree for the expression a*(b+c)/d

* d
a +
b c
2) syntax tree for if a=b then a:=c+d else b:=c-d

If---then---else

c d d

Three-Address Code:
• In three-address code, there is at most one operator on the right side of aninstruction; that is, no built-up arithmetic expressions are permitted.
x+y*z t1 = y * z t2 = x + t1
• Example

Problems:
Write the 3-address code for the following expression
1. if(x + y * z > x * y +z) a=0;
2. (2 + a * (b – c / d)) / e
3. A :=b * -c + b * -c

Address and Instructions

•

• Example Three-address code is built from two concepts: addresses and instructions.

• An address can be one of the following:

– A name: A source name is replaced by a pointer to its symbol table entry.

• A name: For convenience, allow source-program names to Appear as addresses in three-address code. In an Implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.

– A constant

• A constant: In practice, a compiler must deal with many different types of constants and variables

– A compiler-generated temporary

• A compiler-generated temporary. It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

A list of common three-address instruction forms: Assignment statements

– x= y op z, where op is a binary operation

– x= op y, where op is a unary operation

– Copy statement: x=y

– Indexed assignments: x=y[i] and x[i]=y

– Pointer assignments: x=&y, *x=y and x=*y

Control flow statements

– Unconditional jump: goto L

– Conditional jump: if x relop y goto L ; if x goto L; if False x goto L

– Procedure calls: call procedure p with n parameters and return y, is Optional

param x1 param x2

...

param xn call p, n

– do i = i +1; while (a[i]<v);

The multiplication i * 8 is appropriate for an array of elements that each take 8 units of space.

C. quadruples:

• Three-address instructions can be implemented as objects or as record with fields for the operator and operands.

• Three such representations

– Quadruple, triples, and indirect triples

• A quadruple (or quad) has four fields: op, arg1, arg2, and result.

Example D. Triples

• A triple has only three fields: op, arg1, and arg2

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

Example

d. Triples:

• A triple has only three fields: op, arg1, and arg2

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

Fig: Representations of a = b * - c + b * - c

Fig: Indirect triples representation of 3-address code
-> The benefit of Quadruples over Triples can be seen in an optimizing compiler, where instructions are often moved around.
->With quadruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change. With triples, the result of an operation is referred to by its position, so moving an instruction may require changing all references to that result. This problem does not occur with indirect triples.
Single-Assignment Static Form
Static single assignment form (SSA) is an intermediate representation that facilitates certain code optimization.
• Two distinct aspects distinguish SSA from three –address code.

– All assignments in SSA are to variables with distinct names; hence the term static single-assignment.

2. Type Checking:
•A compiler has to do semantic checks in addition to syntactic checks. •Semantic Checks
–Static –done during compilation
–Dynamic –done during run-time
•Type checking is one of these static checking operations.
–we may not do all type checking at compile-time.
–Some systems also use dynamic type checking too.
•A type system is a collection of rules for assigning type expressions to the parts of a program.
•A type checker implements a type system.
•A sound type system eliminates run-time type checking for type errors.

•A programming language is strongly-typed, if every program its compiler accepts will execute without type errors.
In practice, some of type checking operations is done at run-time (so, most of the programming languages are not strongly yped).
–
Type Expression:
•The type of a language construct is denoted by a type expression.
•A type expression can be:
–A basic type
•a primitive data type such as integer, real, char, Boolean, ...
•type-error to signal a type error

•void: no type
–A type name
•a name can be used to denote a type expression.
–A type constructor applies to other type expressions.
•arrays: If T is a type expression, then array (I,T)is a type expression where I denotes index range. Ex: array (0..99,int)
•products: If T1and T2 are type expressions, then their Cartesian product T1 x T2 is a type expression. Ex: int x int
•pointers: If T is a type expression, then pointer (T) is a type expression. Ex: pointer (int)
•functions: We may treat functions in a programming language as mapping from a domain type D to a range type R. So, the type of a function can be denoted by the type expression D→R where D are R type expressions. Ex: int→int represents the type of a function which takes an int value as parameter, and its return type is also int.
Type Checking of Statements:
S ->d= E { if (id.type=E.type then S.type=void

else S.type=type-error }

S ->if E then S1 { if (E.type=boolean then S.type=S1.type

else S.type=type-error }
S->while E do S1 { if (E.type=boolean then

S.type=S1.type else S.type=type-error }

Type Checking of Functions:
E->E1(E2) {

else E.type=type-error }

Ex: int f(double x, char y) { ... }
f: double x char->int
argume types return type

Structural Equivalence of Type Expressions:
•How do we know that two type expressions are equal?
•As long as type expressions are built from basic types (no type names), we may use structural equivalence between two type expressions

Structural Equivalence Algorithm (sequin):
if (s and t are same basic types) then return true
else if (s=array(s1,s2) and t=array(t1,t2)) then return (sequiv(s1,t1) and sequiv(s2,t2)) else if (s = s1 x s2and t = t1 x t2) then return (sequiv(s1,t1) and sequiv(s2,t2))
else if (s=pointer(s1) and t=pointer(t1)) then return (sequiv(s1,t1))

else if (s = s1 else return false

Names for Type Expressions:

•In some programming languages, we give a name to a type expression, and we use that name as a type expression afterwards.

type link = ↑cell; ? p,q,r,s have same types ? var p,q : link;

var r,s : ↑cell

•How do we treat type names?

–Get equivalent type expression for a type name (then use structural equivalence), or

–Treat a type name as a basic type

3.Syntax Directed Translation:

 A formalist called as syntax directed definition is used fort specifying translations for programming language constructs.

 A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has associated set of attributes and each and each productions is associated with a set of semantic rules

Definition of (syntax Directed definition) SDD :

SDD is a generalization of CFG in which each grammar productions X->α is associated with it a set of semantic rules of the form

a: = f(b1,b2.....bk)

Where a is an attributes obtained from the function f.

• A syntax-directed definition is a generalization of a context-free grammar in which:

– Each grammar symbol is associated with a set of attributes.

– This set of attributes for a grammar symbol is partitioned into two subsets called synthesized and inherited attributes of that grammar symbol.

– Each production rule is associated with a set of semantic rules.

• Semantic rules set up dependencies between attributes which can be represented by a dependency graph.

• This dependency graph determines the evaluation order of these semantic rules.

• Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

The two attributes for non terminal are :

1) Synthesized attribute (S-attribute) : (↑)

An attribute is said to be synthesized attribute if its value at a parse tree node is determined from attribute values at the children of the node

2) Inherited attribute: (↑,→)

An inherited attribute is one whose value at parse tree node is determined in terms of attributes at the parent and | or siblings of that node.

 The attribute can be string, a number, a type, a, memory location or anything else.

 The parse tree showing the value of attributes at each node is called an annotated parse tree.

The process of computing the attribute values at the node is called annotating or decorating the parse tree.Terminals can have synthesized attributes, but not inherited attributes.

Annotated Parse Tree

• A parse tree showing the values of attributes at each node is called an Annotated parse tree.
• The process of computing the attributes values at the nodes is called annotating (or decorating) of the parse tree.
• Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

Ex1:1) Synthesized Attributes : Ex: Consider the CFG :
S→ EN E→ E+T E→E-T E→ T T→ T*F T→T/F T→F F→ (E) F→digit N→;

Solution: The syntax directed definition can be written for the above grammar by using semantic actions for each production.
Production rule Semantic actions

S →EN S.val=E.val
E →E1+T E.val =E1.val + T.val
E →E1-T E.val = E1.val – T.val
E →T E.val =T.val
T →T*F T.val = T.val * F.val
T →T|F T.val =T.val | F.val
F → (E) F.val =E.val
T →F T.val =F.val
F →digit F.val =digit.lexval
N →; can be ignored by lexical Analyzer as; I

is terminating symbol

For the Non-terminals E,T and F the values can be obtained using the attribute “Val”.
The taken digit has synthesized attribute “lexval”.
In S→EN, symbol S is the start symbol. This rule is to print the final answer of expressed.
Following steps are followed to Compute S attributed definition
1. Write the SDD using the appropriate semantic actions for corresponding production rule of the given Grammar.
2. The annotated parse tree is generated and attribute values are computed. The Computation is done in bottom up manner.
3. The value obtained at the node is supposed to be final output.

PROBLEM 1:

Consider the string 5*6+7; Construct Syntax tree, parse tree and annotated tree.
Solution:
The corresponding annotated parse tree is shown below for the string 5*6+7;

Syntax tree:

Annotated parse tree :

Advantages: SDDs are more readable and hence useful for specifications
Disadvantages: not very efficient.

Ex2:

PROBLEM : Consider the grammar that is used for Simple desk calculator. Obtain the Semantic action and also the annotated parse tree for the string
3*5+4n. L→En E→E1+T

E→T
T→T1*F
T→F
F→ (E)
F→digit
Solution :
Production rule Semantic actions
L→En 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
The corresponding annotated parse tree U shown below, for the string 3*5+4n.

Dependency Graphs:

Dependency graph and topological sort:

 For each parse-tree node, say a node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with X.

 If a semantic rule associated with a production p defines the value of synthesized attribute A.b in terms of the value of X.c. Then the dependency graph has an edge from X.c to A.b

 If a semantic rule associated with a production p defines the value of inherited attribute B.c in terms of the value X.a. Then , the dependency graph has an edge from X.a to B.c.

Applications of Syntax-Directed Translation

• Construction of syntax Trees

– The nodes of the syntax tree are represented by objects with a suitable number of fields.
– 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 nodes are viewed as records, the Leaf returns a pointer to a new record for a leaf.
• 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 , c1,c2,.....ck) creates an object with first field op and k additional fields for the k children c1,c2,.....ck

Syntax-Directed Translation Schemes

A SDT scheme is a context-free grammar with program fragments embedded within production bodies .The program fragments are called semantic actions and can appear at any position within the production body.

Any SDT can be implemented by first building a parse tree and then pre-forming the actions in a left-to-right depth first order. i.e during preorder traversal.

The use of SDT's to implement two important classes of SDD's

1. If the grammar is LR parsable, then SDD is S-attributed.
2. If the grammar is LL parsable, then SDD is L-attributed.

Postfix Translation Schemes

The postfix SDT implements the desk calculator SDD with one change: the action for the first production prints the value. As the grammar is LR, and the SDD is S-attributed.

L →E n {print(E.val);}

E → E1 + T { E.val = E1.val + T.val }

E → E1 - T { E.val = E1.val - T.val }

E → T { E.val = T.val }

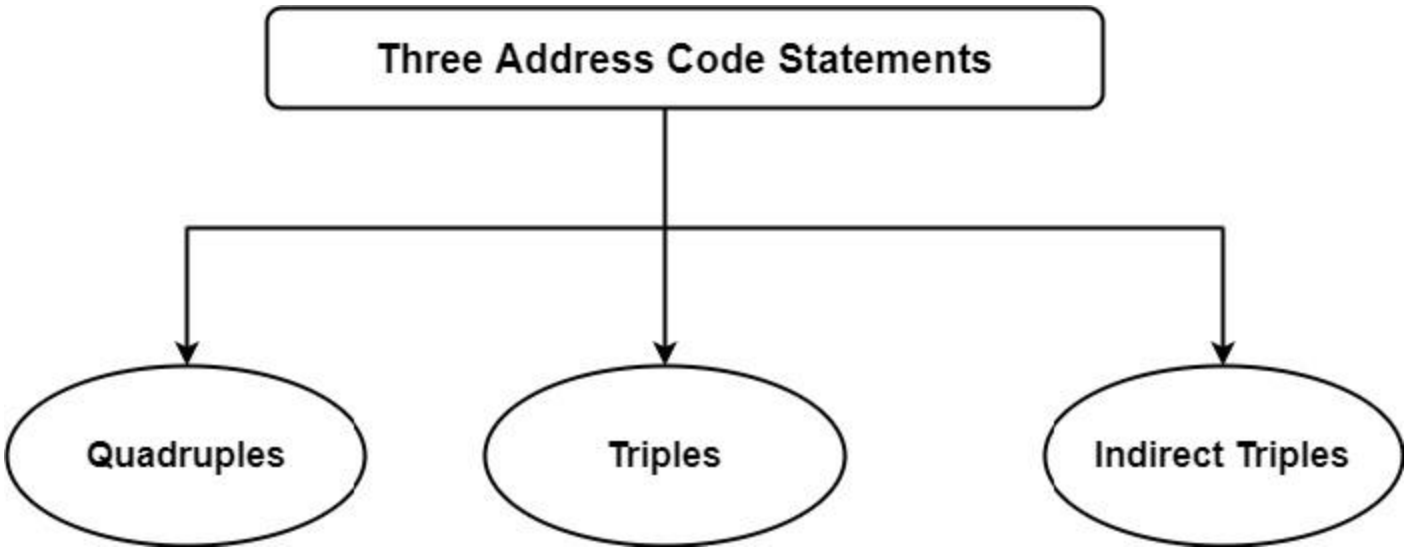
T → T1 * F { T.val = T1.val * F.val } T → F { T.val = F.val }

Implementation of three address code: **Implementation of Three Address Code Statements?**

Compiler DesignProgramming LanguagesComputer Programming

There are three implementations used for three address code statements which are as follows −

- Quadruples
- Triples
- Indirect Triples



Quadruples

Quadruple is a structure that contains atmost four fields, i.e., operator, Argument 1, Argument 2, and Result.

Operator	Argument 1	Argument 2	Result

For a statement a = b + c, Quadruple Representation places + in the operator field, a in the Argument 1 field, b in Argument 2, and c in Result field.

For example− Consider the Statement

a = b + c * d

First, convert this statement into Three Address code

∴ Three Address code will be

t1 = c ∗ d

t2 = b + t1

a = t2.

After construction of the Three Address code, it will be changed to Quadruple representation as follows−

Quadruple

Location	Operator	arg 1	arg 2	Result

Location	Operator	arg 1	arg 2	Result
(0)	*	c	d	t2
(1)	+	b	t1	t1
(2)	=	t2		A

The content of fields arg 1, arg 2 and Result are pointers to symbol table entries for names represented by these entries.

Triples

This three address code representation contains three (3) fields, i.e., one for operator and two for arguments (i.e., Argument 1 and Argument 2)

Operator	Argument 1	Argument 2

In this representation, temporary variables are not used. Instead of temporary variables, we use a number in parenthesis to represent a pointer to that particular record of the symbol table.

For example, consider the statement

a = b + c * d

First of all, it will be converted to Three Address Code

∴ t1 = c ∗ d

t2 = b + t1

a = t2

Triple for this Three Address Code will be −

Triple

Location	Operator	arg 1	arg 2
(0)	∗	C	d

Location	Operator	arg 1	arg 2
(1)	+	B	(0)
(2)	=	A	(1)

Here (0) represents a pointer that refers to the result c * d, which can be used in further statements, i.e., when c * d is added with b. This result will be saved at the position pointer by (1). Pointer (1) will be used further when it is assigned to a.

Indirect Triples

The indirect triple representation uses an extra array to list the pointers to the triples in the desired order. This is known as indirect triple representation.

Indirect Triple will be

Indirect Triples

Statement	
(0) (1) (2)	(11) (12) (13)

Location	Operator	arg 1	arg 2
(11)	*	c	d
(12)	+	b	(11)
(13)	=	a	(12)

In this, it can only need to refer to a pointer (0), (1), (2)which will further refer pointers(11), (12), (13) respectively & then pointers (11), (12), (13) point to triples that is why this representation is called indirect triple representation.

Backpatching

Backpatching is the activity of filling up in specified information of labels using appropriate sementic actions during the code generation process.

2. Backpatching refers to the process of resolving forward branches that have been used in the code, when the value of the target becomes known.
3. Backpatching is done to overcome the problem of processing the incomplete information in one pass.
4. Backpatching can be used to generate code for boolean expressions and flow of control statements in one pass.

To generate code using backpatching following functions are used :

1. **Makelist(*i*) :** Makelist is a function which creates a new list from one item where *i* is an index into the array of instructions.
2. **Merge(p_1 , p_2) :** Merge is a function which concatenates the lists pointed by p_1, and p_2, and returns a pointer to the concatenated list.
3. **Backpatch (*p*, *i*) :** Inserts i as the target label for each of the instructions on the list pointed by *p*.

Backpatching in boolean expressions :

1. The solution is to generate a sequence of branching statements wher- the addresses of the jumps are temporarily left unspecified.
2. For each boolean expression *E* we maintain two lists :
   a. *E*.truelist which is the list of the (addresses of the) jump statemen. appearing in the translation of *E* and forwarding to *E*.true.
   b. *E*.falselist which is the list of the (addresses of the) jump statements appearing in the translation of *E* and forwarding to *E*.false.
3. When the label *E*.true (resp. *E*.false) is eventually defined we can walk down the list, patching in the value of its address.
4. In the translation scheme below :
   a. We use emit to generate code that contains place holders to be filled in later by the backpatch procedure.
   b. The attributes *E*.truelist, *E*.falselist are synthesized.
   c. When the code for *E* is generated, addresses of jumps corresponding to the values true and false are left unspecified and put on the lists *E*.truelist and *E*. falselist, respectively.

5. A marker non-terminal *M* is used to capture the numerical address of a statement.
6. nextinstr is a global variable that stores the number of the next statement to be generated.

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 grammar

1. E → E OR E
2. E → E AND E
3. E → NOT E
4. E → (E)
5. E → id relop id
6. E → TRUE
7. 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:

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
```

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:

1. S → LABEL : S
2. 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:

1. S → **if** E then S
2. S → **if** E then S **else** S
3. S → **while** E **do** S
4. S → begin L end
5. S→ A
6. L→ L ; S
7. 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:

1. S → **if** E then M S
2. S → **if** E then M S **else** M S
3. S → **while** M E **do** M S
4. S → begin L end
5. S → A
6. L→ L ; M S

7. L → S
8. M → ∈
9. 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_)

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 statement

1. S → call id(Elist)
2. Elist → Elist, E
3. 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.

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:

1. D → integer, id
2. D → real, id
3. 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

ENTER is used to make the entry into symbol table and **ATTR** is used to trace the data type.

Case Statements

Switch and case statement is available in a variety of languages. The syntax of case statement is as follows:

```
1.    switch E
2.          begin
3.                case V1: S1
4.                case V2: S2
5.                .
6.                .
7.                .
8.  case Vn-1: Sn-1
9.  default: Sn
10.         end
```

The translation scheme for this shown below:

Code to evaluate E into T

```
1.  goto TEST
2.          L1:     code for S1
3.                  goto NEXT
4.          L2:     code for S2
5.                  goto NEXT
6.                  .
7.                  .
8.                  .
9.          Ln-1:   code for Sn-1
10.                 goto NEXT
11.         Ln:     code for Sn
12. goto NEXT
13.         TEST:   if T = V1 goto L1
14.                 if T = V2goto L2
15.                 .
16.                 .
```

17. .
18. **if** T = Vn-1 **goto** Ln-1
19. **goto**
20. 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.