UNIT – III

INTERMEDIATE CODE GENERATION

Types of intermediate code – Representation of three address code – syntax Directed translation scheme – Intermediate code generation for : Assignment statements – Boolean statements – switch – case statement – Procedure call – Symbol Table Generation.

Types of Intermediate code:

* The translation of the source code into the object code for the target m/c, a compiler can produce middle level language code, which is referred to intermediate code.

Types : 3 types.

1) Postfix Notation
2) Syntax Tree
3) Three Address code
   - Quadruples
   - Triples
   - Indirect Triples.

Postfix Notation:

→ operator comes after an operand.

→ ie, the operator follows an operand

Eg:

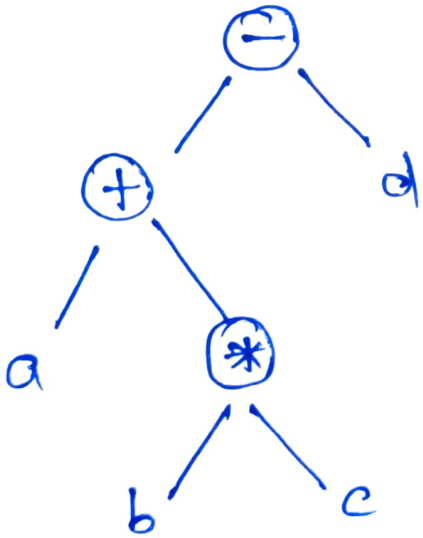
$-(a+b)*(c+d) \longrightarrow ab+cd+*$

$-(a*b)-(c+d) \longrightarrow ab*+cd+-$

Syntax Tree:

→ leaf node describes an operand & each interior node is an operator.

Eg: $a+b*c-d$.



Three Address code:

→ sequence of stmts of the form $A = B\ op\ C$ where A, B, C are either programmer-defined names, constants or compiler generated temp-names,

The op represents an operator.

→ reason for the name 3-addr code is that each stmt generally includes 3 addr, 2 for the operands and one for result.

Types of 3 address :

Quadruples :

→ it is a structure which consists of 4 fields namely op, arg1, arg2 and result.

→ op denotes the operator

→ arg1 & arg2 denotes two operands

→ result is used to store the result of the expn.

Triples :

→ doesn't make use of extra temporary variables to represent a single operation, instead when a reference to another triple's value is needed, a ptr to a triple is used.

→ it consist of only three fields namely op, arg1 and arg2.

Indirect Triples:

→ it makes use of pointer to the listing of all references to computation which is made separately and stored.

→ similar in utility as compared to quadruple representation but requires less space than it.

→ Temporaries are implicit and easier to rearrange code.

Representation of 3 addr code: (Implementation)

* 3-address code is a type of intermediate code, which is easy to generate and can be easily converted to m/c code.

* It makes use of atmost three addresses.

General representation

$a = b \ op \ c$

where

- a, b or c represents operands like names, constants or compiler generated temp names.
- op represents operator.

SYMBOL TABLE GENERATION:

Symbol Table (ST) Important data structure

* Created and maintained by compiler to store the info about various entries such as variable names, function names, objects, classes, interfaces etc.
* It is used in both the analysis & synthesis parts.
* Analysis phase collects the info for symbol table.
* Synthesis phase uses that info to generate code.
* It is built in both lexical & syntax analysis phases.
* It is used by compiler to acheive compile time efficiency.

Use of Symbol Table in various phases of Compiler:

1. Lexical Analysis: Creates new table entries (token) in the table.
2. Syntax Analysis: Adds info about attribute type, scope, dimension, line of reference, use etc in the table.
3. Semantic Analysis: Checks for semantic errors (type checking) by using available info.
4. Intermediate Code Generation: To know how much and what type of run-time is allocated we use the symbol table in this phase.

Code Optimization: Uses info in symbol table for machine dependent optimization.

Target Code Generation: Generates code by using address info of identifier present in the table.

* A symbol table is simply a table that is either linear or a hash table.

* Maintains an entry for each name in the format "<Symbol Name, Type, Attribute>"

Example: Static int interest; → Variable declaration

Stores above in following format in symbol table

<interest, int, static>

Operations of Symbol Table:

1. Allocate: To allocate a new empty symbol table.

2. Free: To remove all entries and free the storage of ST.

3. Insert: Insert() function is used to insert name in a ST and return a pointer to its entry.

Example: int x;

the compiler process the above in the format of {insert(x,int)}

4. Lookup: Used to search for a name and return a pointer to its entry.

5. Set-Attribute: To associate an attribute with a given entry

6. Get-Attribute: To get the associated attribute with a given entry.

Implementation of ST:

* We use some of the data structures commonly to implement the ST. They are:

1. List
2. Linked List
3. Hash Table
4. Binary Search Tree (BST)
5. Scope Management.

1. List: Arrays are used to store names and its associated information.

* New names are added in the order as they arrive.
* For inserting new names, it must not already present or else it occurns an error as "Multiple defined name".
* For searching a name, we start from ~~searching~~ of beginning of list till Available pointer and if not found we get an error as- "use of undeclared name".
* "Available" pointer is used in List.

* Time complexity for insertion - O(1) - fast
" " Lookup - O(n) - slow for large table

* Advantage: Takes minimum amount of space.

2. Linked List:

* Informations are linked together in the form of list.
* A link field is added to each record.
* Time complexity for insertion - O(1) - fast
" " Lookup - O(n) - slow for large table.

3. Hash Table:

* It is an array with index range of table size 0 to -1.
* To search for a name we use hash function which results in integer b/w 0 to -1.
* ~~Inserti~~
* Time complexity for insertion & Lookup - O(1) - very fast
* Advantage: Search is possible
* Disadvantage: Hashing is complicated to implement.

4. Binary Search Tree (BST):

* We add 2 link fields i.e; left & right childs.
* All names are created as child of root node.
* Time complexity for insertion & Lookup - $O(\log_2 n)$.

5. Scope Management:

* We are having 2 types of symbol Tables.

1. Global Symbol Table: It can be accessed by all procedures.
2. Scope Symbol Table: Created for each scope in the program.

* To determine the scope of a name, symbol tables are arranged in hierarchical structure.

Example:

```
int value=10;
int sum_num()
 {
 int num_1;
 int num_2;
  {
  int num_3;
  int num_4;
  }
int num_5;
  {
  int num_6;
  int num_7;
}
```

```
int sum_id;
 {
 int id_1;
 int id_2;
  {
  int id_3;
  int id_4;
  }
int id_5;
}
```

We represents above example code in Hierarchical structure of ST.

Value	Var	int
Sum_num	Sum	int
Sum_id	Sum	int

→ Global Symbol Table.

Sum_num Symbol table

num_1	Var	int
num_2	Var	int
num_5	Var	int

Sum_id symbol table

id_1	Var	int
id_2	Var	int
id_5	var	int

Local Symbol tables.

num_3	Var	int
num_4	Var	int

inner Scope 1

num_6	Var	int
num_7	Var	int

inner scope 2

id_3	Var	int
id_4	Var	int

inner scope 3

Q2. SDT for assignment statements

eg. Consider the grammar

$S \longrightarrow id = E$

$E \longrightarrow E_1 + E_2$

$E \longrightarrow E_1 * E_2$

$E \longrightarrow -E_1$

$E \longrightarrow (E_1)$

$E \longrightarrow id$

Attributes → E.place stores the value of E_1
→ E.code is the sequence of 3 address statements evaluating E.

gen produces the sequence of 3 address statements.

PRODUCTION	SEMANTIC RULES
$S \rightarrow id = E$	S.code = E.code || gen (id.place = E.place)
$E \rightarrow E_1 + E_2$	E.place = newtemp () E.code = E_1.code || E_2.code || gen (E.place = E_1.place + E_2.place)
$E \rightarrow E_1 * E_2$	E.place = newtemp () E.code = E_1.code || E_2.code || gen (E.place = E_1.place * E_2.place)
$E \rightarrow -E_1$	E.place = newtemp () E.code = E_1.code || gen (E.place = 'uminus' E_1.place)

$E \rightarrow (E_1)$	$E.place = E_1.place$ $E.code = E_1.code$
$E \rightarrow id$	$E.place = id.place$ $E.code = " "$

eg. Generate the three address code for the assignment statement $A = -B * (C + D)$

Sequence of moves ⟶

Input	stack	value	generated code
A=-B*(C+D)			
=-B*(C+D)	id	A	
-B*(C+D)	id=	A_	
B*(C+D)	id=-	A__	
*(C+D)	id=-id	A__B	
*(C+D)	id=-E	A__B	
*(C+D)	id=E	A_T_1	$T_1 = -B$
(C+D)	id=E*	A_T_1_	
C+D)	id=E*(	A_T_1__	
+D)	id=E*(id	A_T_1__C	
+D)	id=E*(E	A_T_1__C_	
D)	id=E*(E+	A_T_1__C_	
)	id=E*(E+id	A_T_1__C_D	
)	id=E*(E+E	A_T_1__C_D	
)	id=E*(E	A_T_1__T_2	$T_2 = C + D$
-	id=E*(E)	A_T_1__T_2_	
-	id=E*E	A_T_1_T_2	
-	id=E	A_T_3	$T_3 = T_1 * T_2$
-	S	S	$A = T_3$