**Syntax Directed Translation**

**Arrays**

Array defined as 'int a[10][20][30][40]'

Array reference is 'a[5][6][7][8]'

|  |
| --- |
| 0000 |
| 0001 |
| . |
| . |
| . |
| 0039 |
| 00100 |
| . |
| . |
| 00139 |

Assuming arrays are allocated as

From 0000 to 1000, the number of integers which can be stored = 24000 (10\*20\*30\*40)

Therefore, difference between the addresses of

a[0][0][0][0] and a[1][0][0][0] = 24000\*sizeof(int)

1. To go to a[5][0][0][0], we need to traverse 5\*24000 locations
2. From a[5][0][0][0] to a[5][6][0][0], we need to traverse 6\*30\*40 locations
3. From this point to a[5][6][7][0], 7\*40 locations
4. From this position to a[5][6][7][8], 8 locations

Summarising points (i) to (iv), to go from a[0][0][0][0] to a[5][6][7][8], we need to traverse (5\*24000 + 6\*30\*40 + 7\*50 + 8) = 127488 locations

**Calculating the array access number**

**Method I:** access number = (first offset \* limit1 \* limit2 \* limit3) + (second offset \* limit 2 \* limit3) + (third offset \* limit 3) + (fourth offset)

**Method II: ((**(5 \* 20) + 6) \* 30 + 7) \* 40 + 8

**Grammar for this recursive definition**:

**I) elist 🡪 elist1[id]**

{

elist.arrayname = elist1.arrayname

elist.offset = elist1.offset \* current dimension + id

elist.dimension = elist1.dimension + 1

}

**elist 🡪 id[id]**

{

Get name of the array in elist.arrayname

elist.offset = id

elist.dimension = 1

}

*Example - array reference is a[5][6][7][8]*

elist a[5][6][7][8]

elist [id] a[5][6][7]

elist\*\* [id] a[5][6]

elist\* [id]

(second offset)

id [id]

(base of the array) (first offset - a[5])

\* at this point, name\_of\_array = 'a' (stored in elist.arrayname)

dimension = 1

offset = 5

\*\* name\_of\_array = 'a'

dimension = 2

offset = (5\*20) + 6

**II)** We can define a queue and put all the offsets there. And when we have processed everything, then we'll multiply to calculate the array access number. The changed grammar for this will be -

**L 🡪 id elist** //unroll at this point

**elist 🡪 [id] elist**

**elist 🡪 [id]**

*Example - array reference is a[5][6][7][8]*

L --5, multiply all here

id elist --6

id elist --7

id --8

**Translation of Switch Statements**

switch(E)

{

case constant1 : statement1

case constant2 : statement2

default statement3

}

There are 2 techniques to translate **switch**:

1. **Converting switch to if-else ladder**

Calculate E

if E == constant1

statement1

else if E == constant2

statement2

else

statement3

This technique is slightly difficult.

**Disadvantage:**

While translation we don’t know value of E (it will be further up the tree). Therefore, we'll need **placeholders**.

1. **Using labels** (Decoupling translation of statements with value of E)

We translate above switch as :

E

goto Test

L1: statement1

L2: statement2

Ln: statement3

Test: if E == constant1 goto L1

if E == constant2 goto L2

goto Ln

**Grammar:**

**C -> case id: MS**

{

Add in queue (id.value , M.Quad)

}

**C -> case id : MS C**

{

Add in queue (id.value , M.Quad)

//Queue associated with non-terminal C

}

**S -> switch(E) N {C}**

{

//E.value is known

Bachpatch(N.Nextlist,NextQuad);

//For all values in Process Queue

compare ( E.value == id.value) to goto M.Quad

}

After calculating E, we need to jump to start of table. For that we need to backpatch N to nextquad and then directly compare.

If there are **‘break'** statements, we need to **create nextlist for each case:**

**c.nextlist = (S.nextlist,c(1).nextlist)**

**For 1st case:**

C.nextlist= S.nextlist

**For 2nd case and all following:**

C.nextlist=C(1).nextlist

**Advantages:**

* Works in all cases, for example, when there is switch inside a switch.
* Easy because we don’t need a lot of information.