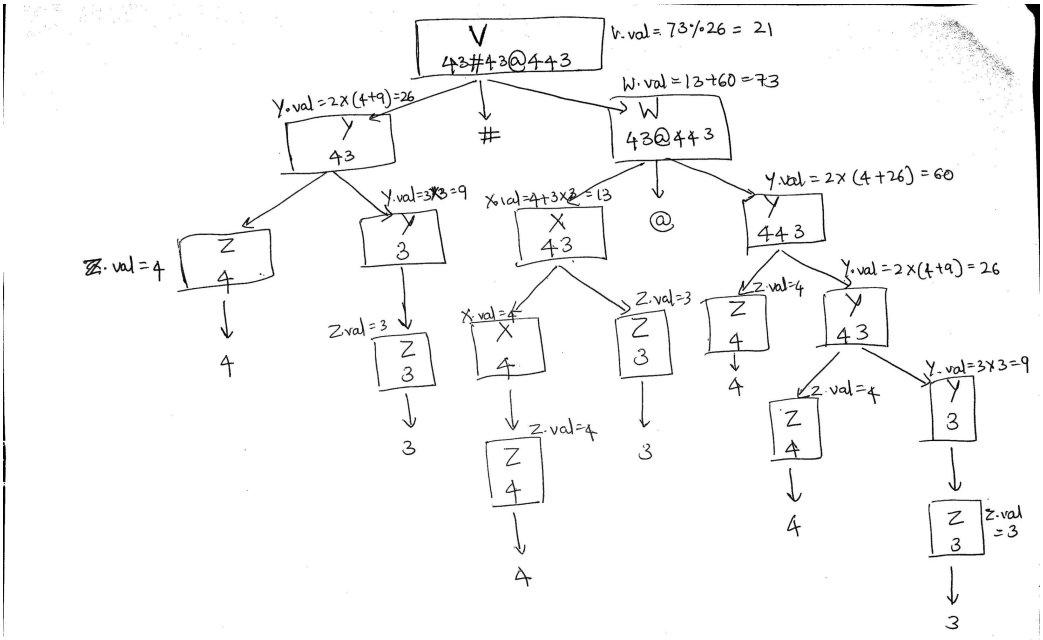
CS335A: Assignment 3

Shrilakshmi S K (211012)

April 5, 2024

Problem 1

(i) Annotated Parse Tree



(ii)

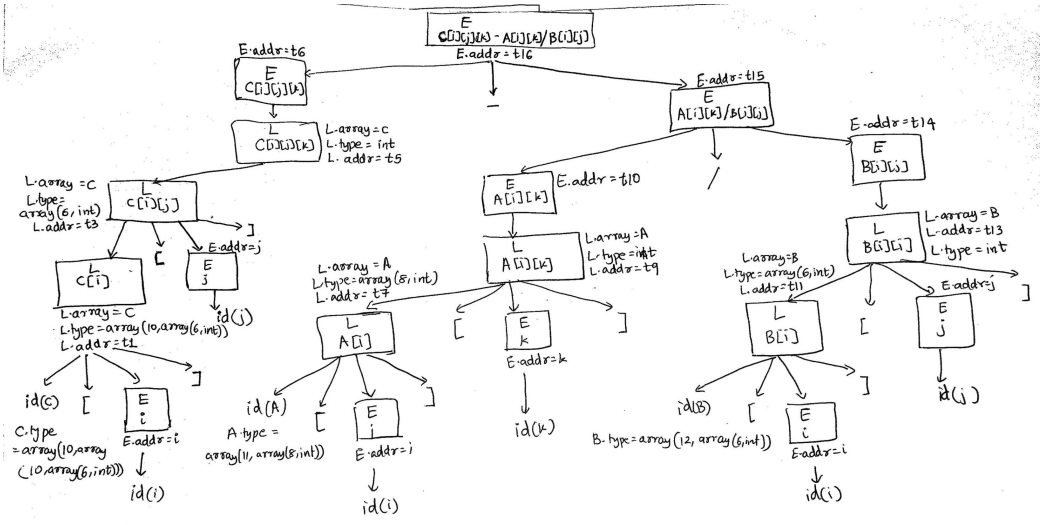
For the given input string 43#43@443, the value of V computed is 21.

(iii)

The grammar is S-attributed because all the attributes (in this case, there is only one attribute val) are synthesized attributes. Parent of each rule computes its attribute val from attribute val of the children of the same rule.

Problem 2

Annotated Parse Tree



3AC

1) t1 = i * 240
2) t2 = j * 24
3) t3 = t1 + t2
4) t4 = k * 4
5) t5 = t3 + t4
6) t6 = C[t5]
7) t7 = i * 32
8) t8 = k * 4
9) t9 = t7 + t8
10) t10 = A[t9]
11) t11 = i * 24
12) t12 = j * 4
13) t13 = t11 + t12
14) t14 = B[t13]
15) t15 = t10 / t14
16) t16 = t6 - t15

Generated 3AC Code

```
1  t1 = i * 240
2  t2 = j * 24
3  t3 = t1 + t2
4  t4 = k * 4
5  t5 = t3 + t4
6  t6 = C[t5]
7  t7 = i * 32
8  t8 = k * 4
9  t9 = t7 + t8
10 t10 = A[t9]
11 t11 = i * 24
12 t12 = j * 4
13 t13 = t11 + t12
14 t14 = B[t13]
15 t15 = t10 / t14
16 t16 = t6 - t15
```

In 3AC code, sizeof(int) = 4.
Line 1: L.type.width = 10 * 6 * sizeof(int) = 240
Line 2: L.type.width = 6 * sizeof(int) = 24
Line 4: L.type.width = sizeof(int) = 4
Line 7: L.type.width = 8 * sizeof(int) = 32
Line 8: L.type.width = sizeof(int) = 4
Line 11: L.type.width = 6 * sizeof(int) = 24
Line 12: L.type.width = sizeof(int) = 4

Problem 3

(i) SDT Translation Scheme and Semantic Actions

```
S :
{ stack <vector <int>, int, int> state;
stack <string> temp_stack;
vector <int> list_type; }
id = E {
    gen(symtop.get(id.lexeme)" = "E.addr);
}
| { stack <vector <int>, int, int> state;
stack <string> temp_stack;
vector <int> list_type; }
L = E {
    if(L.indexed){
        gen(L.array.base"["L.addr"]"" = "E.addr);
    }
    else{
        gen(L.addr" = "E.addr);
    }
}

////////////////////////////////////////////////////

E :
E_1 + E_2 {
    E.addr = newtemp();
    gen(E.addr" = "E_1.addr" + "E_2.addr);
}
| L {
    if(L.indexed){
        E.addr = newtemp();
        gen(E.addr" = "L.array.base"["L.addr"]");
    }
    else{
        E.addr = L.addr;
    }
}
////////////////////////////////////////////////////
```

```
L :
id {
    L.indexed = 0;
    L.addr = symtop.get(id.lexeme);
}
| id [ {
    list_type = symtop.get(id.lexeme).type;
    state.push({list_type, 0, 4});

}
Elist {
    L.indexed = 1;
    state.pop();
    if(!state.empty())list_type = state.top();
    L.addr = Elist.offset;
    L.array = symtop.get(id.lexeme);
}

//////////////////////////////////////////////////////

Elist :
E ] {
    t = newtemp();
    gen(t" = "E.addr" * "state.top().third);
    state.top().third *= list_type[state.top().second];
    state.top().second += 1;
    Elist.offset = t;
}
| E {
    t = newtemp();
    gen(t" = "E.addr" * "state.top().third);
    state.top().third *= list_type[state.top().second];
    state.top().second += 1;
    temp_stack.push(t);
} , Elist_1 {
    Elist.offset = newtemp();
    gen(Elist.offset" = "temp_stack.top()" + "Elist_1.
        ↪ offset);
    temp_stack.pop();
}
```

(ii) Attributes, Data Structures & Auxiliary Functions

Attributes of nonterminal *L*

- `L.indexed`: Since L has two production rules as a parent ($L \rightarrow$ **id** and $L \rightarrow$ **id** [*Elist*), we need to differentiate both as in the former case, we need not create a new temporary variable in E's production, whereas in latter case, we do. So, when $L \rightarrow$ **id**, we set `L.indexed = 0` and when $L \rightarrow$ **id** [*Elist*, we set `L.indexed = 1`

- `L.addr`: The meaning of `L.addr` varies depending upon `L.indexed`. If `L.indexed = 0`, then `L.addr` is just lexeme of the **id**. If `L.indexed = 1`, then `L.addr` is the column major offset of the given indices in the array in bytes.

- `L.array`: It is set to the the symbol table entry of that array. It will be later used to retrieve the base address when we need to add offset and locate the exact address of the indexed array.

Attributes of nonterminal *E*

- `E.addr`: It represents the the variable or temporary variable where the result of the expression is stored.

Attributes of nonterminal *Elist*

- `Elist.offset`: It is a temporary variable that stores the sum of offsets from last dimension of indices to the dimension where this particular *Elist* occurs.

Auxiliary Functions

- `newtemp()`: It generates a new temporary variable. This is used in intermediate code to hold intermediate results of expressions.

- `gen()`: It is a function that generates the actual three-address code. The strings within the gen function are the instructions themselves. For example, `gen("t1 = t2 + t3")` would generate a line of three-address code where the contents of t2 and t3 are added and the result is stored in t1.

- `symtop.get(id.lexeme)`: It is a function that retrieves the symbol table entry for the identifier given by `id.lexeme` from the current symbol table and its ancestors in symbol table hierarchy.

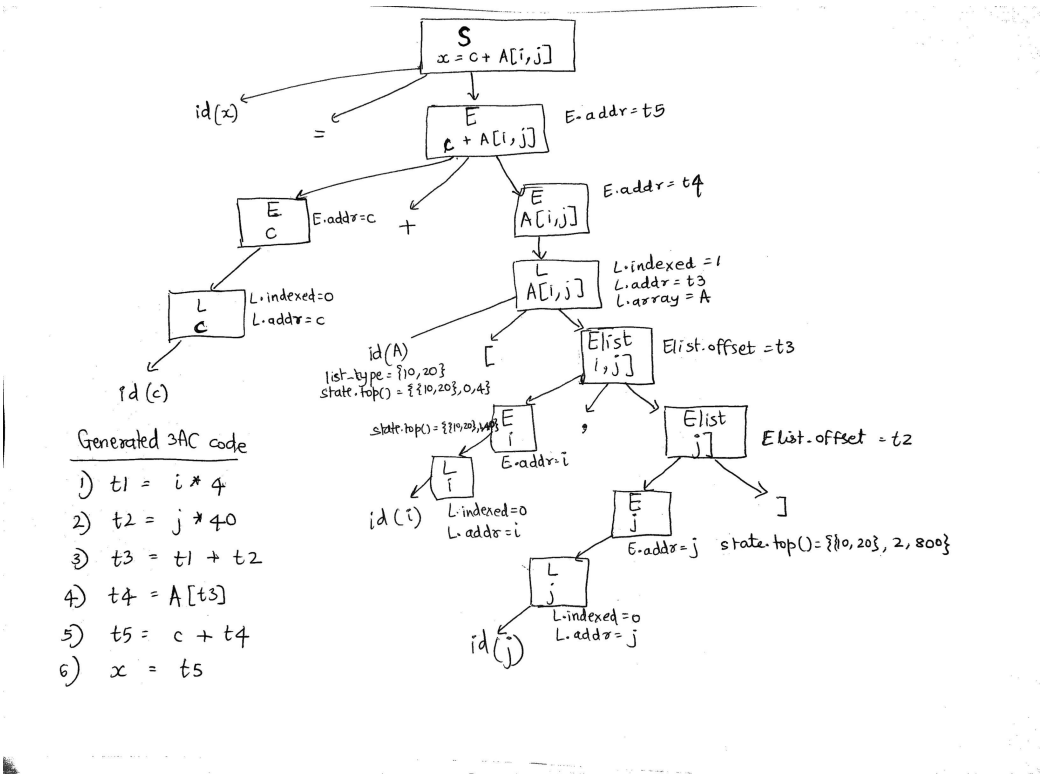
Data Structures

- `vector <int> list_type`: This simply stores the dimensions of the current array being parsed. If the dimensions of array A is 2 x 9 x 4, then `list_type = {2, 9, 4}`

- `stack <vector <int>, int, int> state`: This is a stack of triplet that stores all the incomplete array indexings. The first element in this triplet is the dimensions/type of array. The second element is the dimension of last indexing. The third element is a number which we will eventually multiply with the index E to find the offset of that dimension. With every passing index, we multiply this third element by the dimension's length. This ensures correct calculation of column major index. At the beginning of indexing, we set third element to 4, as 4 is the number that should be multiplied to find the column major offset in first dimension in bytes.
  We use stack of triplets instead of just a global triplet to handles nested indexing cases like `A[i, j, B[m, n], k]`. We use a similar analogy used in creating hierarchy of symbol tables in milestone 2.

- `stack <string> temp_stack`: This stack stores the temporaries of `Elist.offset`. We need to sum up all the temporaries holding offsets of individual dimension to obtain total offset. This stack will be used to sum up all the temporaries in the end. Note that, this stack does not affect nested array indexing cases as the element is pushed and popped in same production, which guarantees either full overlapping or full disjointness among the nested array indexings.

(iii) Annotated Parse Tree



(iv) Generated 3AC Code

```
t1 = i * 4
t2 = j * 40
t3 = t1 + t2
t4 = A[t3]
t5 = c + t4
x = t5
```