1 Define Synthasized attributes and inheritted attributes

$$S \to L\ .\ L \mid L$$
$$L \to L\ B \mid B$$
$$B \to 0 \mid 1$$

Design an L-attributed SDD to compute S.val, the decimal-number value of an input string. For example, the translation of string 101.101 should be the decimal number 5.625.

Hint: use an inherited attribute L. Side that tells which side of the decimal point a bit is on.

Synthesized and Inherited Attributes

In syntax-directed definitions (SDDs), attributes are associated with grammar symbols to hold information about the parse tree. These attributes can be of two types: **synthesized attributes** and **inherited attributes**.

1. **Synthesized Attributes**:
   - Synthesized attributes are those whose values are computed from the attribute values of the children (or subnodes) in the parse tree.
   - They flow upwards in the parse tree.
2. **Inherited Attributes**:
   - Inherited attributes are those whose values are assigned from the attribute values of the parent and/or siblings.
   - They flow downwards or horizontally in the parse tree.
3. **Example Grammar and SDD**
4. Given the grammar:

S -> L.L
L -> L B
L -> B
B -> 0 | 1

We want to design an L-attributed SDD to compute `S.val`, the decimal value of the input string. Let's add attributes:

- `S.val`: synthesized attribute representing the final decimal value.
- `L.val`: synthesized attribute representing the decimal value of the substring on the left side of the decimal point.
- `L.frac_val`: synthesized attribute representing the fractional value of the substring on the right side of the decimal point.
- `L.side`: inherited attribute indicating whether `L` is on the left (`"left"`) or right (`"right"`) side of the decimal point.
- `B.val`: synthesized attribute representing the binary value (0 or 1).

Semantic Rules

Let's define the semantic rules to compute the attributes:

1. **Grammar: `S -> L.L`**
   - `S.val = L1.val + L2.frac_val`
     - Where `L1` is the `L` on the left side of the decimal point.
     - Where `L2` is the `L` on the right side of the decimal point.
   - `L1.side = "left"`
   - `L2.side = "right"`
2. **Grammar: `L -> L B`**
   - For `L.side == "left"`:
     - `L1.val = 2 * L2.val + B.val`
     - `L1.side = L.side`
   - For `L.side == "right"`:
     - `L1.frac_val = L2.frac_val + B.val * 2^(-L2.length - 1)`
     - `L1.length = L2.length + 1`
     - `L1.side = L.side`
3. **Grammar: `L -> B`**
   - For `L.side == "left"`:
     - `L.val = B.val`
   - For `L.side == "right"`:
     - `L.frac_val = B.val * 2^(-1)`
     - `L.length = 1`
4. **Grammar: `B -> 0 | 1`**
   - `B.val = 0` if `B -> 0`
   - `B.val = 1` if `B -> 1`

Attribute Computation Example

Let's compute the value for `101.101`:

Step-by-Step Derivation

1. Parse tree for `101.101`:
```
css
Copy code
    S
   / \
  L   L
 /|\  /|\
L B L B B
| | | | |
B B B B B
| | | | |
1 0 1 1 0 1
```
2. **Attribute values**:
   - For the left `L` (`101`):
     - `B1.val = 1`
     - `L2.val = 2 * 1 + 0 = 2`
     - `L1.val = 2 * 2 + 1 = 5`
   - For the right `L` (`101`):
     - `B1.val = 1 * 2^-1 = 0.5`
     - `L2.frac_val = 0.5 + 0 * 2^-2 = 0.5`
     - `L1.frac_val = 0.5 + 1 * 2^-3 = 0.5 + 0.125 = 0.625`
   - `S.val = 5 + 0.625 = 5.625`

Thus, the translation of `101.101` to its decimal value is `5.625`.

Suppose that we have a production A →BCD. Each of the four non-terminals A, B, C, and D have two attributes: s is a synthesized attribute, and i is an inherited attribute. For each of the sets of rules below, tell whether.

(i) the rules are consistent with an S-attributed definition

(ii) the rules are consistent with an L-attributed definition, and

(iii) whether the rules are consistent with any evaluation order at all?

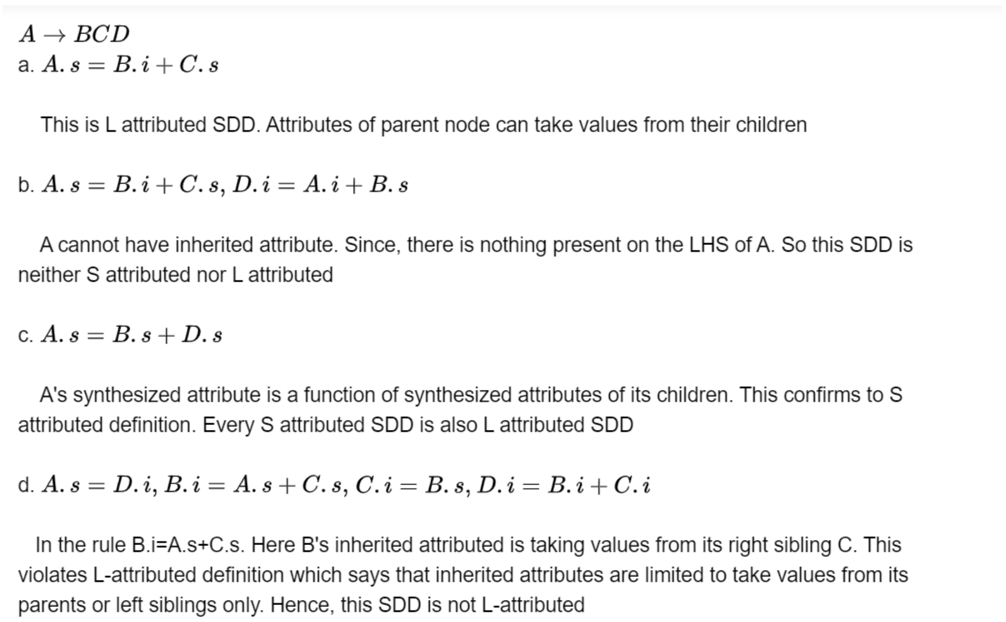
(iv) Draw dependency graph for each set of rules

a) A.s = B.i + C.s.

b) A.s = B.i + C.s and D.i = A.i + B.s.

c) A.s = B.s + D.s.

d) A.s = D.i, B.i = A.s + C.s, C.i = B.s, and D.i = B.i + C.i.



$A \to BCD$

a. $A.s = B.i + C.s$

This is L attributed SDD. Attributes of parent node can take values from their children

b. $A.s = B.i + C.s, D.i = A.i + B.s$

A cannot have inherited attribute. Since, there is nothing present on the LHS of A. So this SDD is neither S attributed nor L attributed

c. $A.s = B.s + D.s$

A's synthesized attribute is a function of synthesized attributes of its children. This confirms to S attributed definition. Every S attributed SDD is also L attributed SDD

d. $A.s = D.i, B.i = A.s + C.s, C.i = B.s, D.i = B.i + C.i$

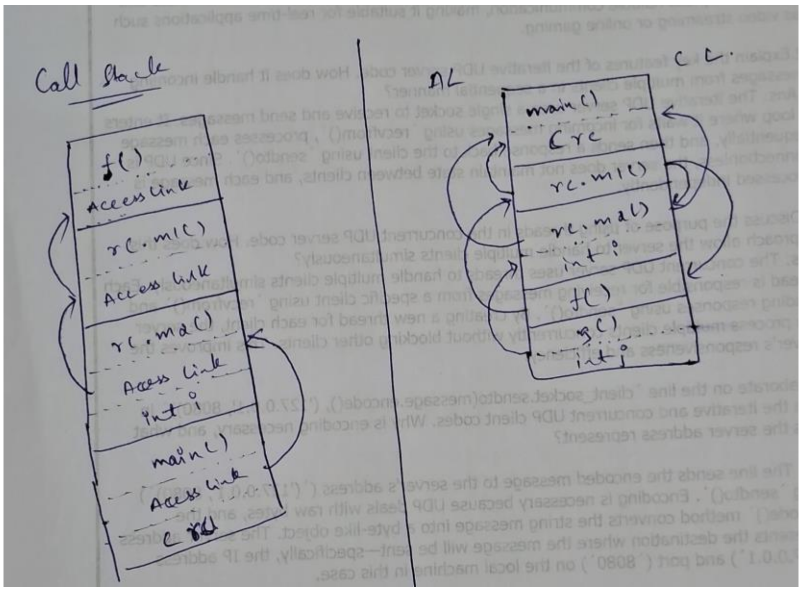
In the rule B.i=A.s+C.s. Here B's inherited attributed is taking values from its right sibling C. This violates L-attributed definition which says that inherited attributes are limited to take values from its parents or left siblings only. Hence, this SDD is not L-attributed

a) In this task we are given a Java-like language where methods can have locally defined methods. Furthermore, it is possible to declare variables and methods at the other most program level. That is supposed to work as usual in languages with static scoping. The following is a program in this language.

```
{
    class C {
        void m1 () {
            void f() {};

            f ();
        }
        void m2 () {
            int i;
            void g() {
                int j;
                j = i;
            };
            i = 1;
            rC.m1();
        };
    };

    C rc ;
    void main () {
        rc =  new C{} ; rC.m2{};
    }
}
```

Draw the call stack in the situation where the activation record for

f is on top of the stack for the first time. Draw the stack including variables, access-links, and control-links, but without access-links for methods which are directly declared in a class.



b) Write a small program that will produce different values depending on which kind of variable scoping mechanism is used, static or dynamic. Explain your answer.

To demonstrate the difference between static (lexical) and dynamic scoping, let's write a small program and explain how the variable values differ based on the scoping mechanism used.

Program Example

```java
int x = 10;

void function1() {
    int x = 20;
    function2();
}

void function2() {
    print(x);
}
```

```
void main() {
    function1();
}
```
Explanation
- Static Scoping (Lexical Scoping) :
  - In static scoping, the scope of a variable is determined by the program's structure, specifically the block in which the variable is declared. The compiler resolves each variable reference to the most closely nested scope where the variable is declared.
  - When `function2` is called, it looks for the nearest declaration of `x` in the environment where `function2` itself is defined. In this case, `function2` is defined at the global scope where `x` is 10. Therefore, the output will be `10`.
- Dynamic Scoping :
  - In dynamic scoping, the scope of a variable is determined by the call stack at runtime. The interpreter or runtime system resolves each variable reference by searching the call stack from the current function backwards until it finds a variable with the same name.
  - When `function2` is called, it looks for the nearest declaration of `x` in the call stack. At runtime, `function1` is on the call stack, and it has a local `x` with the value `20`. Therefore, the output will be `20`.

Expected Output Based on Scoping Mechanism

- Static Scoping (Lexical Scoping) :
  - The output of the program will be `10`.
- Dynamic Scoping :
  - The output of the program will be `20`.

Code Explanation

Here's the breakdown of what happens in each scoping mechanism:
1. Static Scoping :
   - `main` calls `function1`.
   - `function1` declares a local `x` with the value `20` and then calls `function2`.
   - `function2` looks for `x` in its defining environment (the global scope) and finds `x = 10`.
   - `function2` prints `10`.
2. Dynamic Scoping :
   - `main` calls `function1`.
   - `function1` declares a local `x` with the value `20` and then calls `function2`.
   - `function2` looks for `x` in the runtime call stack. It finds the local `x` declared in `function1` with the value `20`.
   - `function2` prints `20`.

Conclusion

The program demonstrates how the value of `x` printed by `function2` differs based on whether the language uses static or dynamic scoping. Static scoping resolves `x` in the lexical scope of `function2`'s definition, while dynamic scoping resolves `x` based on the runtime call stack.

a) Consider the following object-oriented program in Java style:

```
class A { int a,b; };
class B extends A { int c,d1,d2,d3,d4,d5,d6,d7,d8,d9; };
...
static void f(A x) { x.a = 1; }
static void g(B x) { x.c = 2; }
static void h() { A p = new A(); f(p); g(p); }
static void k() { B p = new B(); f(p); g(p); }
static void main() { h(); k(); }
```

Explain the run-time structure of values of type A and B. Indicate a constraint on the layout of these structures needed to support inheritance.

Run-Time Structure of Values of Type A and B

In Java and many other object-oriented languages, objects are allocated on the heap, and their memory layout supports inheritance. Let's explore the structure of objects of type `A` and `B` and the necessary constraints to support inheritance.

Class Definitions and Memory Layout

Given the class definitions:
```java
class A {
    int a, b;
```

}
class B extends A {
    int c, d1, d2, d3, d4, d5, d6, d7, d8, d9;
}
``
 # Object Layout
- Class A : Contains two integer fields, `a` and `b`.
- Class B : Inherits from `A`, so it contains `a` and `b` from `A`, plus its own fields: `c`, `d1`, `d2`, `d3`, `d4`, `d5`, `d6`, `d7`, `d8`, and `d9`.

The memory layout for instances of `A` and `B` can be visualized as follows:
- Instance of A :
      a,b
- Instance of B :
     a,b inherited from A
    c , d1 – d9

  Constraint on the Layout to Support Inheritance

The primary constraint to support inheritance is that the memory layout of a subclass must extend the layout of its superclass. Specifically:

1. Prefix Constraint : The memory layout of `B` must start with the memory layout of `A`. This ensures that any `B` object can be treated as an `A` object, allowing polymorphism and correct method dispatch.
   - This means that the fields of `A` (`a` and `b`) must be at the same offset in both `A` and `B`.

  Explanation of the Program

Let's walk through the program execution:

```java
static void f(A x) { x.a = 1; }

static void g(B x) { x.c = 2; }

static void h() {
    A p = new A();
    f(p); // p.a = 1
    g(p); // Error: g(B) cannot be called with an A reference
}

static void k() {
    B p = new B();
    f(p); // p.a = 1 (B instance is treated as an A instance)
    g(p); // p.c = 2
}

static void main() {
    h(); // Error in g(p)
    k(); // Works fine
}
```

  Details of Method Execution

- Method `f(A x)` :
  - Sets `x.a` to 1.
  - Can be called with an instance of either `A` or `B` due to polymorphism. When called with an instance of `B`, `x.a` refers to the `a` field inherited from `A`.

- Method `g(B x)` :
  - Sets `x.c` to 2.
  - Can only be called with an instance of `B`, since `c` is not a member of `A`.

Execution Flow

- Method `h` :
  - Creates an instance of `A`.
  - Calls `f(p)` which sets `p.a` to 1.
  - Attempts to call `g(p)` with an `A` instance, leading to a compile-time error because `g` expects a `B` instance.

- Method `k` :
  - Creates an instance of `B`.
  - Calls `f(p)` which sets `p.a` to 1. Here, `p` is treated as an instance of `A` due to inheritance.
  - Calls `g(p)` which sets `p.c` to 2.

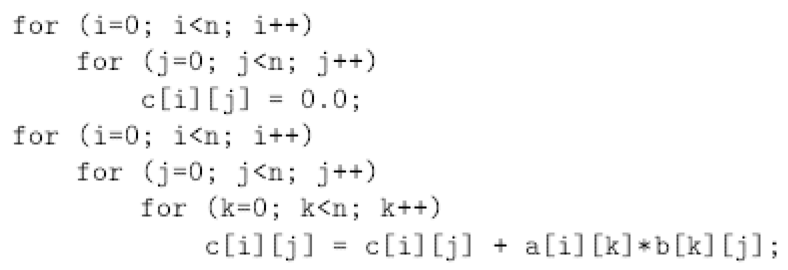
Conclusion

To support inheritance, the memory layout of a subclass must extend that of its superclass. The fields of the superclass must appear at the beginning of the subclass's layout. This allows methods expecting a superclass type to operate correctly on subclass instances. This is crucial for polymorphism and method dispatch to work correctly in object-oriented languages like Java.

b) Explain how it is possible to “leak memory” using a reference counting garbage collector and describe any technique that might be used to address this problem.

Pages 468 -

Define basci block and control flow graph.Discuss the algorithm to construct basic blocks from the given three address code ((Definitiona and algorithm page 525 – 529)

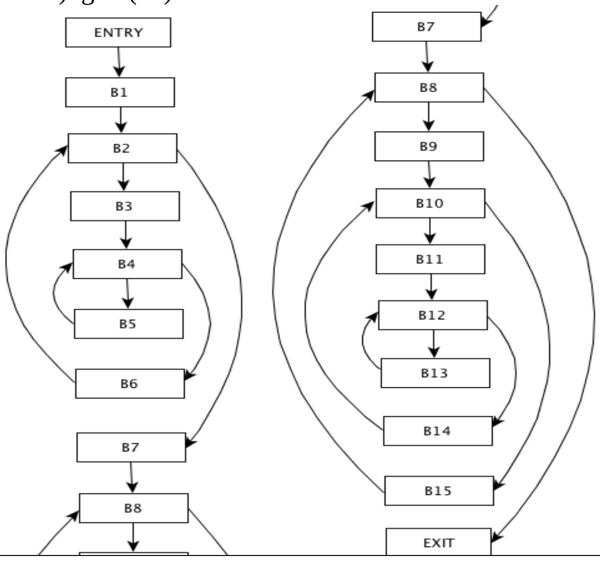


```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0.0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

Construct the control flow graph for the above code

```
B1      1)  i = 0

B2      2)  if i >= n goto(13)

B3      3)  j = 0

B4      4)  if j >= n goto(11)

B5      5)  t1 = n * i
        6)  t2 = t1 + j
        7)  t3 = t2 * 8
        8)  c[t3] = 0.0
        9)  j = j + 1
       10)  goto(4)

B6     11)  i = i + 1
       12)  goto(2)

B7     13)  i = 0

B8     14)  if i >= n goto(40)

B9     15)  j = 0

B10    16)  if j >= n goto(38)

B11    17)  k = 0
```

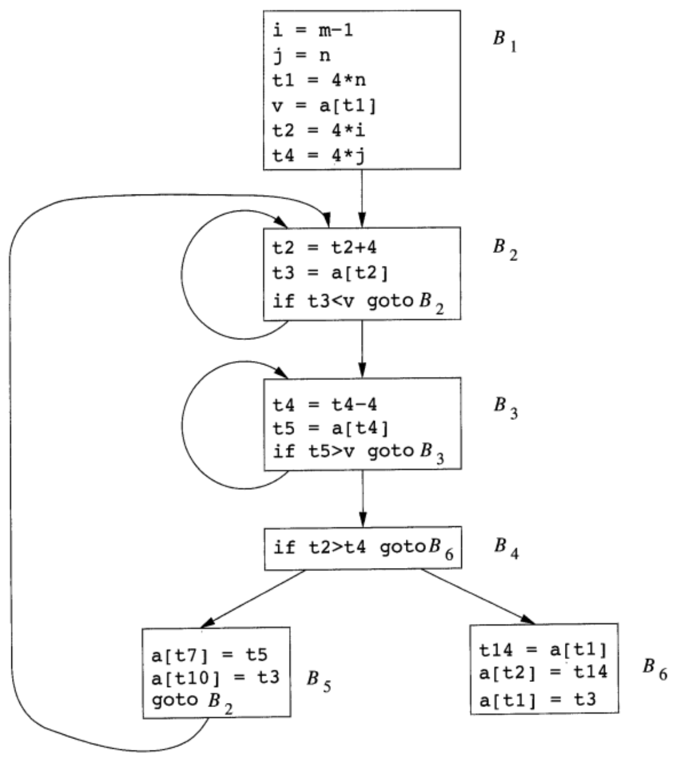
```
B12     18)  if k >= n goto(36)

B13     19)  t4 = n * i
        20)  t5 = t4 + j
        21)  t6 = t5 * 8
        22)  t7 = c[t6]
        23)  t8 = n * i
        24)  t9 = t8 + k
        25)  t10 = t9 * 8
        26)  t11 = a[t10]
        27)  t12 = n * k
        28)  t13 = t12 + j
        29)  t14 = t13 * 8
        30)  t15 = b[t14]
        31)  t16 = t11 * t15
        32)  t17 = t7 + t16
        33)  c[t6] = t17
        34)  k = k + 1
        35)  goto(18)

B14     36)  j = j + 1
        37)  goto(16)

B15     38)  i = i + 1
        39)  goto(14)
```



```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Explain machine independent optimization and convert the above code into three address code and optimize the generated three address code.

```
 (1)    i = m-1
 (2)    j = n
 (3)    t1 = 4*n
 (4)    v = a[t1]
 (5)    i = i+1
 (6)    t2 = 4*i
 (7)    t3 = a[t2]
 (8)    if t3<v goto (5)
 (9)    j = j-1
(10)    t4 = 4*j
(11)    t5 = a[t4]
(12)    if t5>v goto (9)
(13)    if i>=j goto (23)
(14)    t6 = 4*i
(15)    x = a[t6]
(16)    t7 = 4*i
(17)    t8 = 4*j
(18)    t9 = a[t8]
(19)    a[t7] = t9
(20)    t10 = 4*j
(21)    a[t10] = x
(22)    goto (5)
(23)    t11 = 4*i
(24)    x  = a[t11]
(25)    t12 = 4*i
(26)    t13 = 4*n
(27)    t14 = a[t13]
(28)    a[t12] = t14
(29)    t15 = 4*n
(30)    a[t15] = x
```



```
B1:
i = m-1
j = n
t1 = 4*n
v = a[t1]
t2 = 4*i
t4 = 4*j

B2:
t2 = t2+4
t3 = a[t2]
if t3<v goto B2

B3:
t4 = t4-4
t5 = a[t4]
if t5>v goto B3

B4:
if t2>t4 goto B6

B5:
a[t7] = t5
a[t10] = t3
goto B2

B6:
t14 = a[t1]
a[t2] = t14
a[t1] = t3
```