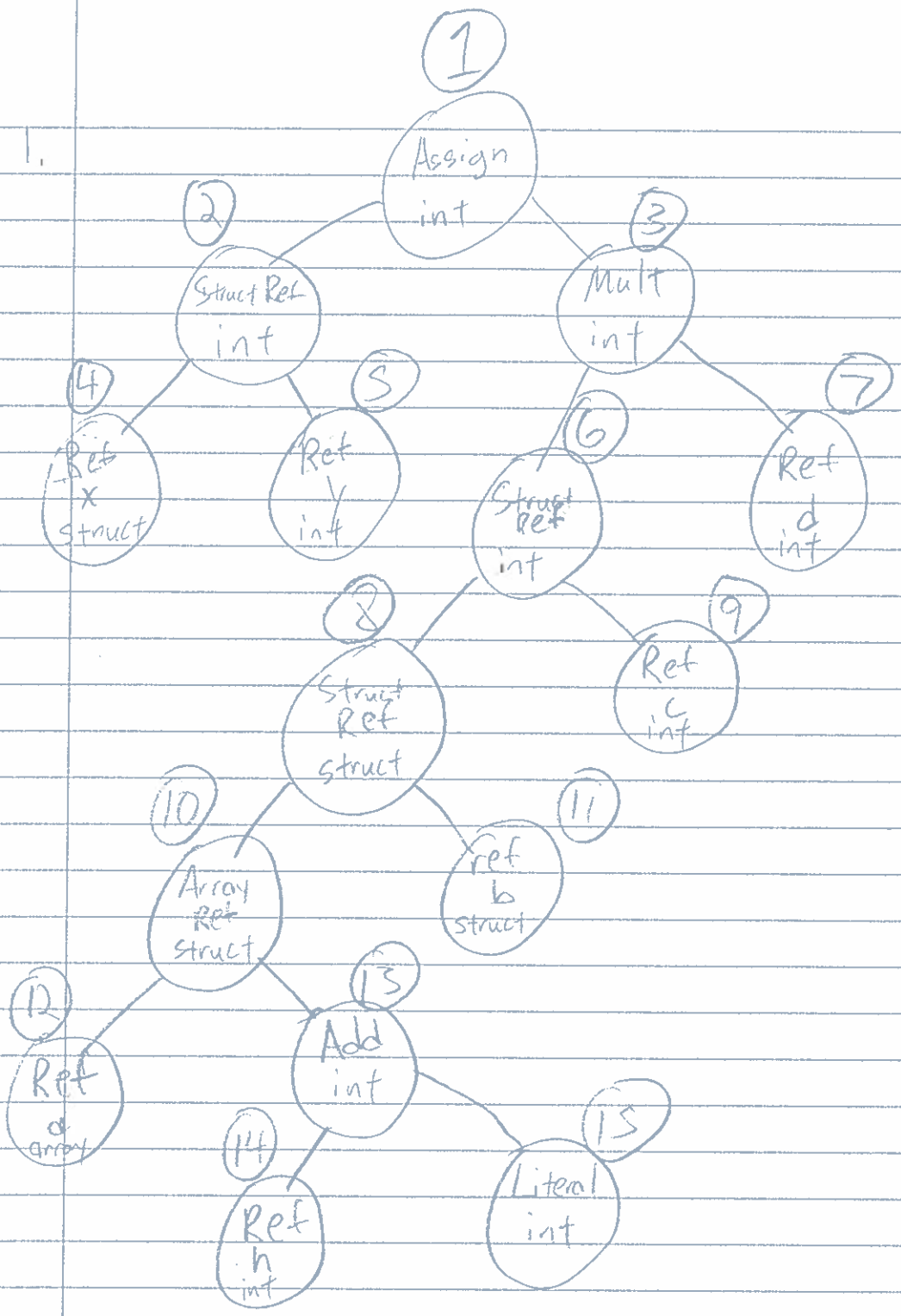
1.


1
Assign
int
2
Struct Ref
int
3
Mult
int
4
Ref
x
struct
5
Ref
y
int
6
Struct Ref
int
7
Ref
d
int
8
Struct
Ref
struct
9
Ref
c
int
10
Array
Ref
struct
11
ref
b
struct
12
Ref
a
array
13
Add
int
14
Ref
h
int
15
Literal
int

2. Start at node 1, travel to node 2 then node 4.

Lookup "x" in symbol table, set type to struct. Back to node 2 then node 5.

Lookup "y" in x's symbol table. Set type to int. Back to 5.

Set 5's type to int. Back to 1. Move to 3, 6, 8, 10, 12.

Lookup "c" in symbol table. Set type to array. Back to 10. Move to 13 then 14.

Lookup "n" in symbol table. Set type to int. Back to 13.

Node 15's type was set during creation of AST. Set 14's type to int, back to 10.

Set 10's type to struct (the type being stored in the array). Back to 8. Move to 11.

Lookup b, set type to struct. Back to 8.

Set 8's type to struct. Back to 6 then 9.

Lookup c, set type to int. Back to 6.

Set 6's type to int. Back to 3. Move to 7.

Lookup d, set 7's type to int. Back to 3.

Set 3's type to int, Back to 1.

Set 1's type to int.

	isR	tN
3. public int method3(int x, String y){	T	F
int a;	T	T
String[] b = new String[x];	T	T
a = 0;	T	T
while(a < 20){	T	T
if(a == 10){	T	F
break;	T	F
} (if)	T	T
a++;	T	T
}		
try{	T	T
if(a > 2){	T	T
b[a] = y;	T	T
}else{	T	T
b[x] = y;	T	T
} (if)	T	T
}catch(ArrayIndexOOBE e){	T	F
return -1;	T	F
a = -1;	F	T
} (try)	T	T
return a;	T	F
}		

4. To handle default parameters, simply put all versions of the method into the symbol table. For the given example:

```
int power(int)
int power(int, int)
```

would both be in the symbol table. Semantic analysis would proceed as usual, trying to match the number of params and their types.

This behavior also has implications on overloading. It is possible to create ambiguous method calls when overloading and default parameters are both used. The responsibility for avoiding such a situation usually falls on the programmer.