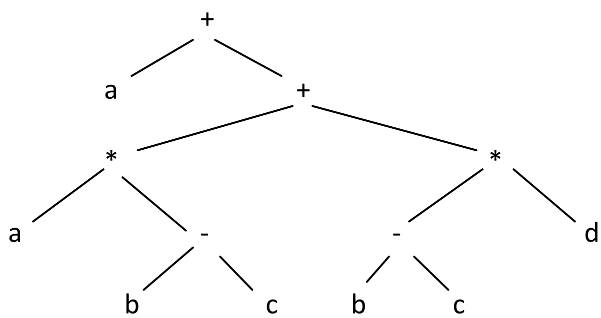
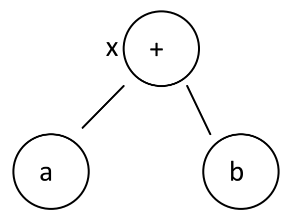
Directed Acyclic Graph (DAG)

Expression: a + a*(b-c) + (b-c)*d

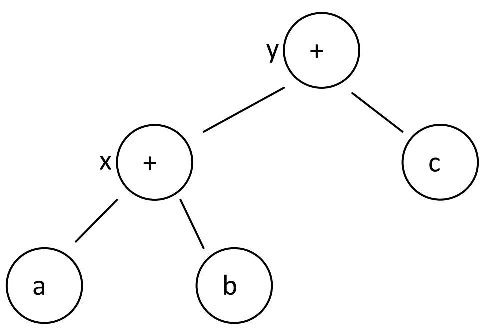


Find common sub-expressions and reduce the tree

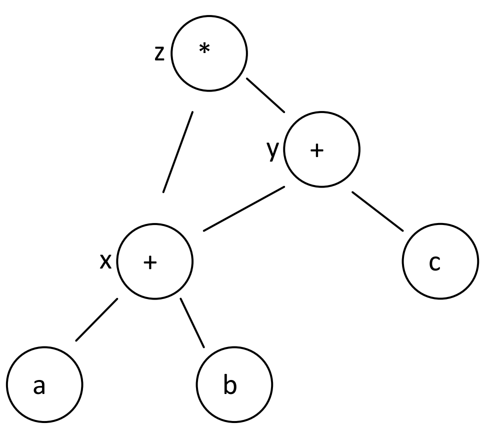
EXAMPLE: (a+b) * (a+b+c)
x = a + b
y = x + c
z = x * y



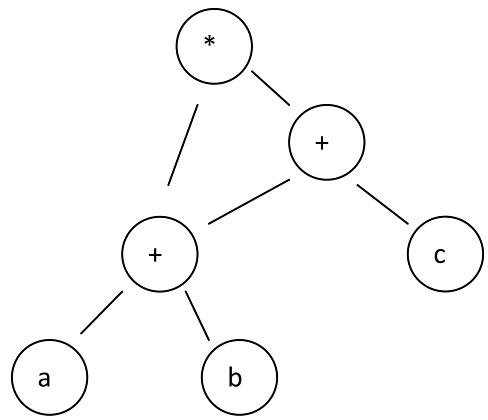
Step 2:



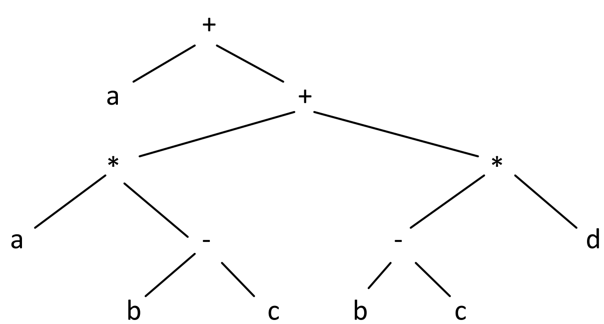
Step 3:



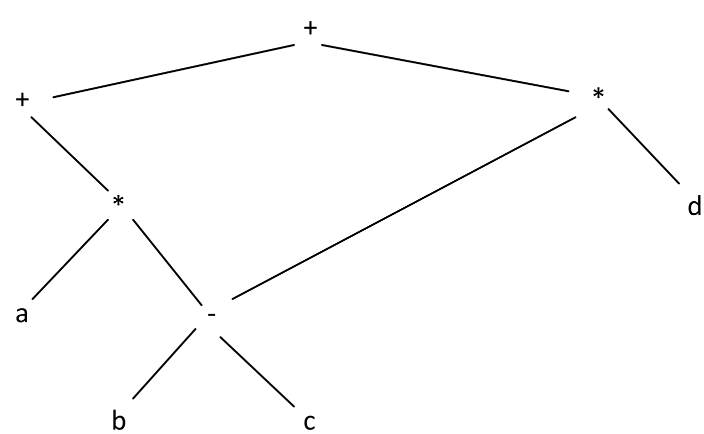
Final:



Expression: a + a*(b-c) + (b-c)*d



DAG:



Production	Sematic Rule
1. E → E_1 + T	E.node = new Node('+',E_1.node,T.node)
2. E → E_1 - T	E.node = new Node('-',E_1.node,T.node)
3. E → E_1 * T	E.node = new Node('*',E_1.node,T.node)
4. E → T	E.node = T.node
5. T → (E)	T.node = E.node
6. T → **id**	T.node = new Leaf(**id**,**id**.val)
7. T → **num**	T.node = new Leaf(**num**,**num**.val)

Steps for constructing DAG:

p1 = Leaf(**id**, entry-a)
p2 = Leaf(**id**, entry-a) = p1
p3 = Leaf(**id**, entry-b)
p4 = Leaf(**id**, entry-c)
p5 = Node('-',p3,p4)
p6 = Node('*',p1,p5)
p7 = Node('+',p1,p6)
p8 = Leaf(**id**,entry-b) = p3
p9 = Leaf(**id**,entry-c) = p4
p10 = Node('-',p3,p4) = p5
p11 = Leaf(**id**,entry-d)
p12 = Node('*',p5,p11)
p13 = Node('+',p7,p12)

Symbol Table

- Data structure used by the compiler to hold information about the source program constructs
- Constructs: Objects, Classes, Variable names, Functions etc.
- Used by both Analysis Phase and Synthesis Phase
    - Analysis Phase: Front-End (Lexical Analysis, Syntactic Analysis, Semantic Analysis and IR Generations)
    - Synthesis Phase: Back-End (IR, Code Optimization, Target Code)
- Used to
    - Store the names of all entities in a structured form in a data structure
    - Verify if the variable has been declared
    - Determine the scope of a variable
    - Implement type checking by verifying assignments and expressions in the source code and Check these semantically correct
    - Generate the IR and the Target code

Data Structure:

- Unordered List (Small data)
- Linear Lists (Sorted/Unsorted)
- Hash Tables
    - Collisions
- Binary Search Trees

Creation: Token identification -> Names stored in the symbol table

Operations:

- Insert:
  insert(symbol,type), e.g. insert(a,int) for the statement int a.

a (int)	.

add (function)	.

- Lookup: lookup(symbol), e.g. lookup(a).
    - Existence of the symbol
    - Declaration before usage
    - Scope of the symbol
    - Initialization of the symbol
    - Multiple declarations of a symbol etc.

```
var: x,y: integer
Procedure P:
  var: x,a: Boolean
  Procedure Q:
    var x,y,z: integer
    begin
    …
    end
```

Approach: Separate Table for each LEVEL

z	integer
y	integer
x	integer

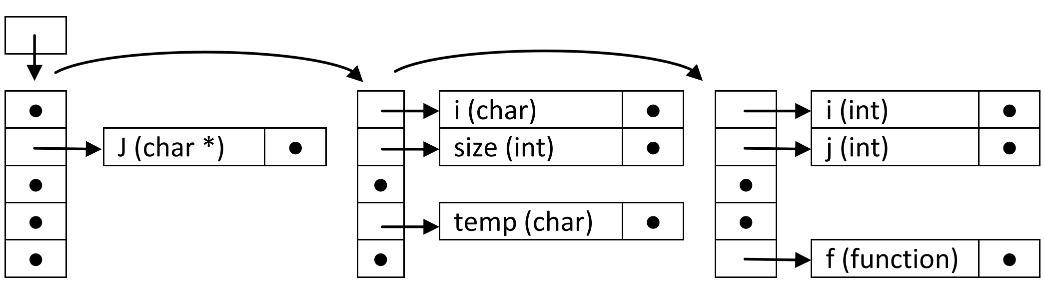
Q	Procedure
a	boolean
x	boolean

P	Procedure
y	integer
x	integer

```
int i,j;
int f(int size)
{
   char i, temp;
   .....
   {
      char *j;
      .....
   }
}
```

```
int i,j;
int f(int size)
{
   char i, temp;
   .....
   {
      double j;
   }
   .....
   {
      char *j;
      .....
   }
}
```

Approach: Hash Table (Separate Table for Each LEVEL)



Type Checking

- Devoted to the type checking activities
  - Operator applied to the correct operand type(s): a = 3 % 1.5; b = a / c; (Data Types of a and c do not match)
  - Flow Control: goto lbl (lbl valid/invalid)
  - Uniqueness: Multiple declarations of a variable.
  - Name Checking: add(int x, int y, int z) {}, Call: add(a,b), Call: aadd(a,b,c)
  - Declaration of the variables:
    - Static: Java, C/C++
    - Dynamic: Python, Java Script, e.g. INPUT a (a has not been declared)
  - etc.
- Depends upon
  - Static Structure of the Language Construct
  - Type expression of the language used
  - Rules to assign type to the construct
- Varies from Language to Language
- Static: Compile-Time, Dynamic: Run-Time
-

Source Code -> Scanner -> Token Stream -> Parser -> Syntax Tree -> Type Checker -> …

C Language: Basic Data Types – 5 : void, char, int, float, double
C Language: Type Modifiers – 4 : signed, unsigned, shot, long

- Every language has simple/basic data types
- Some languages allow the creation of new/customized simple data types, i.e. typdef, enum, …

Array:

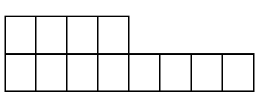
- C/C++/Java : int a[5];
- Pascal : array [1..5] of integer
- Bound Control also comes under type checking

Two-Dimensional Arrays: int a[2][3];
Programming: a[row][col] OR a[col][row]

Records (C/C++): **Assumption: int 4B and float 8B**

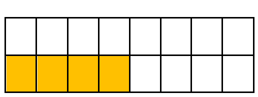
- union
- struct

```
union {
   int a;
   float b;
}
```

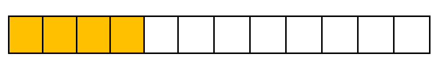


Total size: 8 bytes

Register
AX:
AH,AL



```
struct {
   int a;
   float b;
}
```



Total size of the variable: 4B of a + 8B of b = 12B

```
union {
   int AX;
   struct {
      byte AH;
      byte AL;
   }
}
```

AX – 2B	
AH – 1B	AL – 1B

AX = 3 = 0000 0000 0000 0011
AH = 0 = 0000 0000
AL = 3 = 0000 0011

Pointers:

```
int *a, b;
int c[5];

a = &b
b = 5;

cout << b << endl;        // 5
cout << *a << endl;       // 5

for( int ctr=0 ; ctr<5 ; ctr++ )
   c[ctr] = ctr;

a = &c[3];
cout << *a << endl;       // 3
```

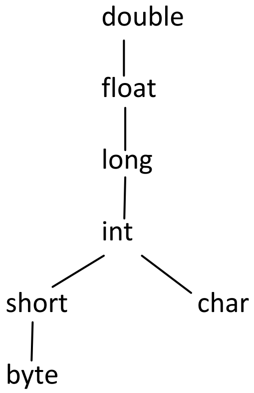
```
int a=3;
float b=a;                Not Accepted (Some compilers may accept)
float b=(float) a;        // 3.0

b = 23.4562;
a = b;                    Not Accepted (Some compilers may accept)
a = (int) b;              // 23
```

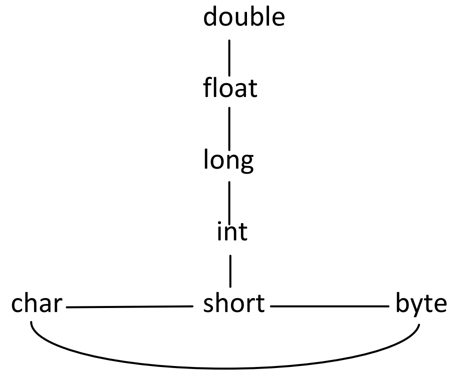
```
E → E1 + E2
if ( E1.type = integer and E2.type = integer )        E.type = integer;
else if ( E1.type = float and E2.type = integer )     E.type = integer;
else …
```

Conversions between primitive types in JAVA

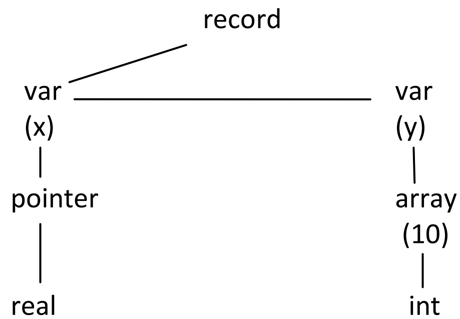
Widening conversion:



Narrowing Conversion:



record
x: pointer to real;
y: array [1..10] of int
end



proc(bool,union a:real; b:char end, int): void