1. Develope a leftmost derivation for the identifier value a2i using the BNF syntax given in Figure 2.7.

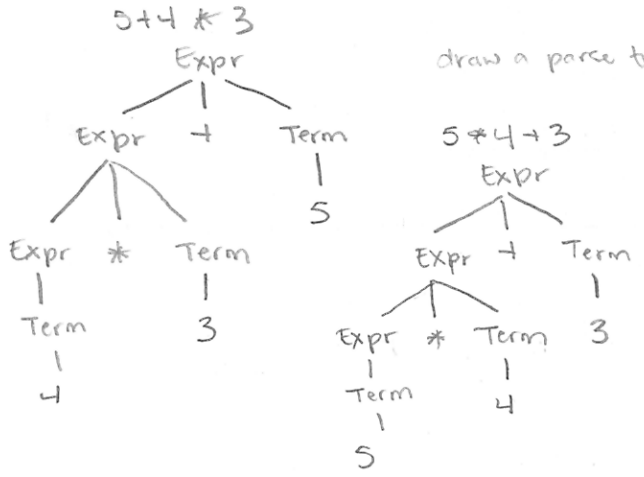
Identifier → Letter {Letter | Digit}
Letter → a|b|...|z|A|B|...|Z
Digit → 0|1|...|9

Identifier → Letter Digit Letter
→ a Digit Letter
→ a 2 Letter
→ a 2 i
→ a2i

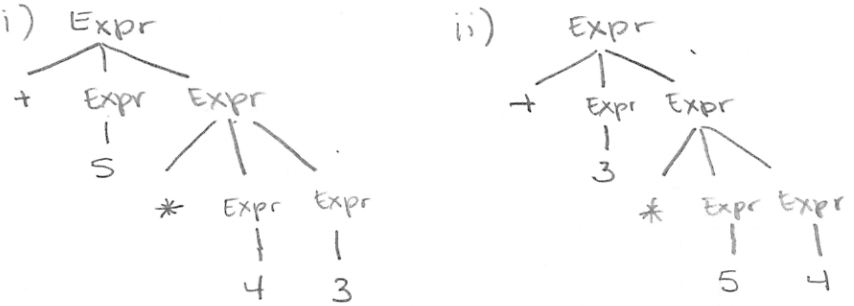
2. Identifier → Letter Digit Letter
→ Letter Digit i
→ Letter 2 i
→ a 2 i
→ a2i

3. Using the grammar: Expr → Expr + Term | Expr * Term | Term
Term → 0|1|...|9|(Expr)

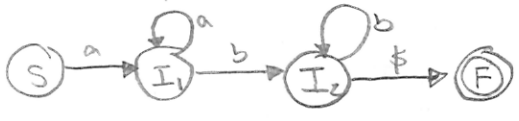
draw a parse tree for each 5+4*3; 5*4+3



4. Using Expr → + Expr Expr | * Expr Expr | 0|1|...|9
draw
i) +5*43
ii) +*543



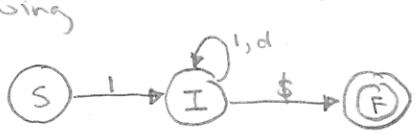
5. Try to define the language $\{a^n b^n\}$ using a DFSA. Discuss why this might not be possible.



This DFSA will not compensate for the set $\{a^n b^n\}$ because its configuration doesn't allows us to count the number of a's or b's. Thus strings aab and abb are legal when clearly they aren't given $\{a^n b^n\}$

6. Show the moves made using the DFSA for identifiers (3.2.2) accepting the following

i) a
ii) a2
iii) a2i
iv) abc



i) $(S, a\$) \vdash (I, \$) \vdash (F,)$

ii) $(S, a2\$) \vdash (I, 2\$) \vdash (I, \$) \vdash (F,)$

iii) $(S, a2i\$) \vdash (I, 2i\$) \vdash (I, i\$) \vdash (I, \$) \vdash (F,)$

iv) $(S, abc\$) \vdash (I, bc\$) \vdash (I, c\$) \vdash (I, \$) \vdash (F,)$

7. For numbers in bases of the form base#number# (w/o embedded ws) give:
(a) a right regular grammar
(b) a regular expression
(c) a DFSA

base expressed in decimal;

Number → Digit Pound Choice1 Choice2 Pound
Choice1 → Letter | Choice 2
Pound → #
Digit → {0|1|...|9}Digit | λ
Letter → {A|B|...|Z}Letter | λ
Choice2 → Digit

N → DPCP → (1D)PCP → (10D)PCP → 10(λ)PCP → 10(#)CP
→ 10#(9C)P → 10#(98C)P → 10#98(λ)P → 10#98#✓

b) {0-9}+#[A-Z]*{0-9}*#

c)

Here I have my expression compensate for numbers in base10+ by allowing uppercase letters to only precede digits. This grammar doesn't restrict illegal numbers like 8#99# however, as I'm unsure how to restrict "numbers" to their respective base.

8.
Expr → Op Expr Expr | Primary
Op → +|-|*|/
Primary → Integer | Letter
Integer → Digit Integer
Letter → a|b|...|z
Digit → 0|1|..|9

⇒

Program = Declarations decpart; Statements body;
Declarations = Declaration*
Declaration = VariableDecl
VariableDecl = Variable v; Type t
Type = int
statements = Statement*
Statement = Assignment
Assignment = VariableRef target; Expr src
Expr = VariableRef | Value | Binary
VariableRef = Variable
Binary = Operator op; Expr t1,t2
Operator = ArithmeticOp
ArithmeticOp = +|-|*|/
Variable = a|b|...|z
Value = 0|1|..|9

8 (cont.)

```
class Program {
    Declarations decPart;
    Statemnts body; }

    Program (Declarations d, Statemnts s) {
        decpart = d
        body    = s

class Declarations extends ArrayList<Declaration> {}
class Declaration {
    Variable v;
    Type     t;
    Declaration (Variable var, Type tytpe) {
        v = var
        t = type
class Type {
    final static Type INT = new Type("int");
    priv String id;
    priv Type (string t) { id = t; }
    return id;
abstract class Statemnt { // statemnt = assignment }
class Statemnts extends Statemnt {
    public ArrayList<Statemnt> mumbers = new ArrayList<Statemnt>(); }
class Assignmnt extends Statemnt {
    Variable target;
    Expr source;
    Assignmnt (Variable t, Expr e) {
        target = t;
        source = e; } }
abstract class Expr { }
class Variable extends Expr {
    priv S id;
    Var (S s) { id = s }
      return id
abstract class Value ext. Expr
    Type type
      int intValue () { ret 0 }
class IntValue ext Value {
    priv int value = 0
    IntValue () { type = Type.INT }
      ret "" + value
class Binary exts Expr {
    Operator op;
    Expr t1, t2
    Binary (Operator o, Expr l, Expr r) {
      op = o; t1 = l, t2 = r;
class Operator {
    final static String PLUS = "+";
                  ...    = "-"; etc.
```

Interp

8.)
sample return for x = 5 *4 + 2
Program
  Declarations:
    Declarations = { int }
  Statements:
    Assignment:
      Variable: X
      Binary:
        Operator: +
        Binary:
          Operator: *
          IntValue: 5
          IntValue: 4
        IntValue: 3

9.) a) lexical scope - a name is bound to a collection of statements in accordance w/ its position in the source program. Scoping based on the grammatical structure of a program
A C++/Java ex. of lex scope is the for loop structure, where the scope of i in for(int i=0; i<10; i++){} is limited to within the brackets
b) dynamic scope - A name is bound to its most recent declaration based on the program's execution history. ex. Line 4 i = 2, Line 6 i = 3, i = 3.
c) lifetime - The lifetime of a variable is the time interval during which the variable has been allocated a block of memory. ex. a variable assignment declared within a function gets memory deallocated once its finished working.
d) visibility - A name is visible to a reference if its referencing includes that reference and the name is not redeclared in an inner scope.
ex. int x = 0; person(string n, int x){ name = n; age = x;} x refers to the constructors' x"
e) nested scope - Scope of a variable or value within a controlled structure that uses the var/val within a specific context
int sum=0; for(int i=0; i<10; i++){
    sum += i // 1 + 2 + 3 ... + 10
F) hidden variables - a variable is declared in a global scope and is then declared again in a local scope ex. instance variables called in a function, which alters the i.v..
g) forward reference - declaring instance variables and methods anywhere within a script; not limited to a particular spot.
ex. private int myInt = 2; - constructor method; private int myOtherInt = 3;
h) Overloading - Uses the # or type of arguements to distinguish among identical function names or operators.
public String toString(){ getClass().getName()... }
public String toString(){ myObject.getName() +" something"; }

10.) ST if the types of all variables are fixed when they are declared at compile time
DT if the type of a variable can vary at runtime depending on the value assign
ST int i; float j; i + j = error incompatible types, DT i coerced to float.
Strongly typed specifies 1+ restrictions on how ops involving values of diff data types can be intermixed. weakly typed is the opposite
WT a = 1, b = "2", concat(a, b) = "22" ST error restriction on type mixing
Explicit typing refers to the declaration of every type associated w/ a value
Implicit typing can "infer" what type a value is based on its context
ET casting.. (int)5 + 3; IT long l; int i; if (l > i) l = i