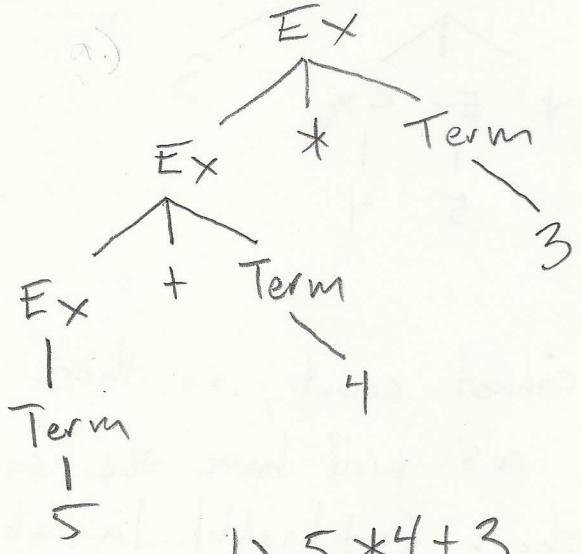
CNC 2012
Programming Langs.
Professor Sheryl Shulman
Midterm
Owen Meyer

1.) $I \rightarrow Let\{Let \mid Dig\} \equiv I \rightarrow Let\ A$
$A \rightarrow Let\ A \mid Dig\ A \mid \lambda$

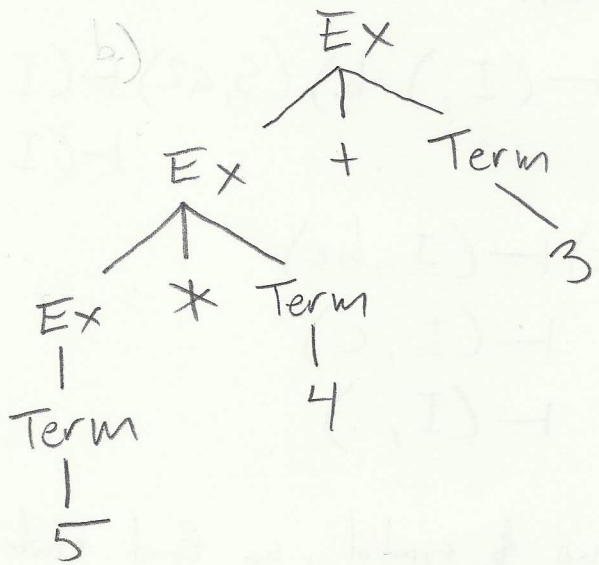
$I \Rightarrow Let\ A$
$\Rightarrow a\ A$
$\Rightarrow a\ Dig\ A$
$\Rightarrow a\ 2\ A$
$\Rightarrow a\ 2\ Let\ A$
$\Rightarrow a\ 2\ i\ A$
$\Rightarrow a\ 2\ i\ \lambda$

2.)
$I \Rightarrow Let\ A$
$\Rightarrow Let\ Dig\ A$
$\Rightarrow Let\ Dig\ Let\ A$
$\Rightarrow Let\ Dig\ Let\ \lambda$
$\Rightarrow Let\ Dig\ i$
$\Rightarrow Let\ 2\ i$
$\Rightarrow a\ 2\ i$

3.) a.) $5+4*3$

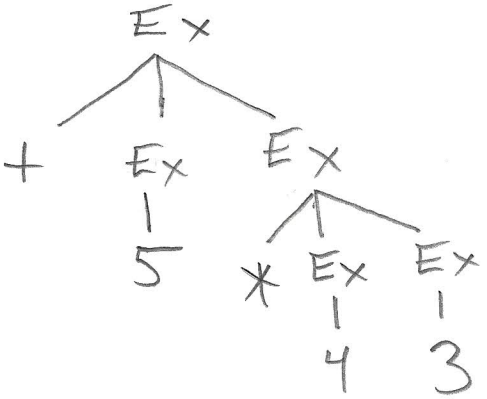


b.) $5*4+3$

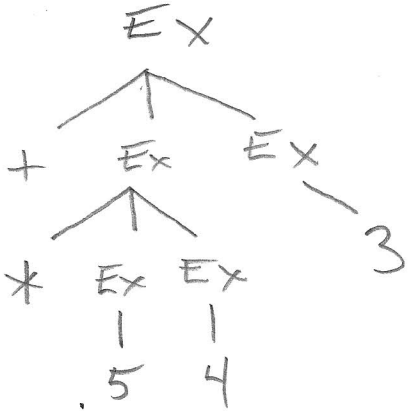


4.)

a.) +5*43



b.) +*543



5.) DFSA's cannot count, so there is no way to determine that the a's will have the same count as the b's. Something like: (ab|aabb|...|a...ab...b) is not a legal regular expression.

6.) a.) $(S, a) \vdash (I,)$ b.) $(S, a2) \vdash (I, 2) \vdash (I,)$ c.) $(S, a2i) \vdash (I, 2i) \vdash (I, i) \vdash (I,)$

d.) $(S, abc) \vdash (I, bc) \vdash (I, c) \vdash (I,)$

<no $ symbol, no final states reached>

7.)

a.) $S \rightarrow 0A|\ldots|9A$

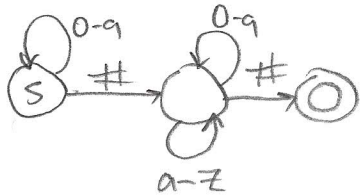
$A \rightarrow 0A|\ldots|9A|0B|\ldots|9B$

$B \rightarrow \#C$

$C \rightarrow 0C|\ldots|9C|aC|\ldots|zC|\#$

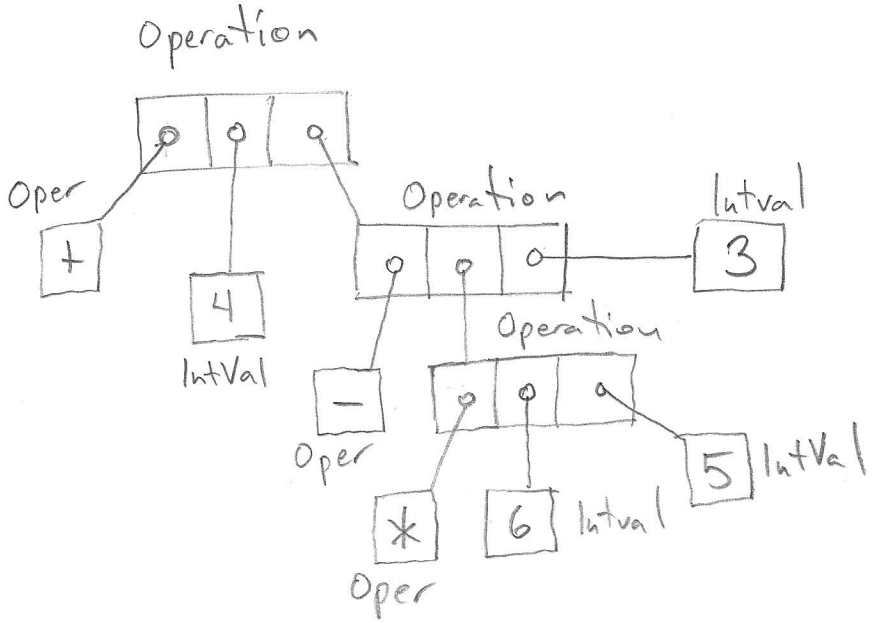
b.) $(0|\ldots|9)^*\#(0|\ldots|9|a|\ldots|z)^*\#$

c.)

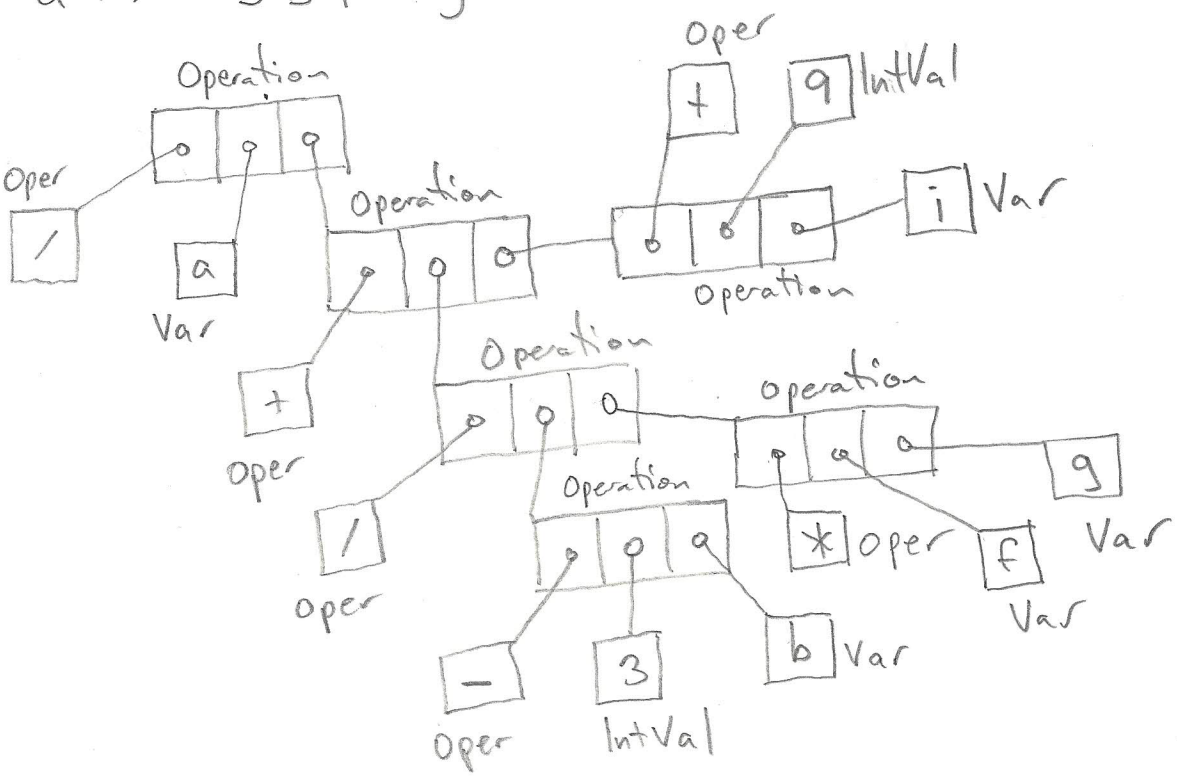


None of these forms discern whether some is string is in legal form. I believe that in order to achieve this counting is necessary, therefore right regular grammars, regular expressions, or DFSA's are incapable of this feat as none of them are capable of counting (all three describe the same set of languages: the regular languages). I have experimented with implimenting expressions such as these where there is a specified upper bound to the number base. I believe in this case it would be possible to discern between legal and illegal strings although very difficult.

8.) +4-*653



/a+/-3b*fg+9i



8.)

```
//Design concept for Programming Langs. Midterm problem 8.)

abstract class Expr{}

class Operation extends Expr{
        Oper op;
        Expr ex1 ex2;

        Operation (Op o, Expr e1, Expr e2){
                op = o;
                ex1 = e1;
                ex2 = e2;
        }
}

class IntVal extends Expr{
        int val;

        IntVal (int v){val = v;}
}

class Var extends Expr{
        char a;

        Var (char i){a=i;}
}

class Oper {
        String op;

        Oper (String o){op=o;}
}

public class Parser {

        public Expr expression(){
                Expression e;
                if (token.isVar() ) e = Var(match(...) );
                else if (token.isIntVal() ) e = IntVal(match(...) );
                else if (token.isOper() ){
                        Oper o = new Oper(match(...) );
                        Expr e1 = expression();
                        Expr e2 = expression();
                }
                else error();
                return e;
        }
}
```

9.)
a.
Lexical scope (also called static scope) is where a name is bound to a collection of statements in accordance with it's position in the source program. This is performed at compile time. For example:

for (int i = 0; i < 3; i++){...}

The lexical scope of i is within the confines of this for loop.

b.
Dynamic scope is when a name is bound to it's most recent point of declaration in terms of the execution history. An example:
← see reverse
given figure 4.2 in Programming Languages, the book goes on to describe how in considering the call history the run-time symbol table looks like:
" B <w, 2>, <j, 3>, <k, 3>
main <a, 15>, <b, 15>, <h, 1>, <i, 1>, B, 2>, <A, 8>, <main, 14>
Where the reference to i now resolves to the one declared globally on line 1." - Programming Languages.

c.
Lifetime is how long a name binding lasts. e.g., a variable within a call on the stack frame has stack frame lifetime.

d.
A name has visibility within the referencing environment it is declared in unless it has been redeclared by an inner scope. e.g.:

```
void eggs(){
        int yoke;
        ...
        for (int yoke;...){..}  // ← yoke declared just above is not visible within the for loop.
}
```

e.
Nested scope is a scope within a scope (think Crosby Stills and Nash). e.g.:

```
void someScope(){
        ...
        while(...){<nested scope is happening..>}
}
```

f.
Hidden variables are variables that are redeclared within a nested scope. (for e.g., see d.)

g.
Forward referencing is when a name is referenced before it's first declaration. Such as:

```
lbeam++;
int lbeam = 0;
```

h.
Overloading happens when some method name has multiple definitions that are distinguished by their type signatures. Here is a case of this phenomenon:

```
void fan(int arrgh){...}
void fan(String ahhh){...}
```

10.)
a.
In static typing all the variable's types are fixed at compile time, while in dynamic typing, the types of the variables can fluctuate during the execution of the program.

b.
Strong typing allows for all the type errors to be caught at compile time or run time. Week typing is the lack of this virtuous attribute.

c.
Explicit typing (such as using a unary casting operator) is type conversion specified by the program writer. Implicit typing is a conversion of the type of some value that is handled by the machine, such as in the case of adding an int to a float where the int is coerced into a float.

11.)
Yikes!
What language is this? I guess I could assume it to be something, but that would be presumptuous. Let us say that this is some unknown language and thus the semantics must be inferred from the syntax. I am thinking that malloc is a dragon of great strength with the ability to char his victim's flesh. I am definitely outgunned on this one.. The internet says that ** is a pointer to a pointer and I can't find what the * between the char and the q says. I can only find cases of malloc (memory allocation) online or in the book (pg 286) where it takes a single argument, while here it seems to be taking two. So this thing is copying one block of memory to another I am guessing (is q a pointer or sometimes a pointer or a pointer to the chars, and why is sizeof() taking a type as an argument?!) by incrementing the pointer *p while reassigning the whizzit to the gramblesneazir until the string length is achieved. This seems pretty sensible to me now.

```
1  int h, i;
2  void B(int w) {
3      int j, k;
4      i = 2*w;
5      w = w+1;
6      ...
7  }
8  void A (int x, int y) {
9      float i, j;
10     B(h);
11     i = 3;
12     ...
13 }
14 void main() {
15     int a, b;
16     h = 5; a = 3; b = 2;
17     A(a, b);
18     B(h);
19     ...
20 }
```

fig. 4.2