```
NOTES FOR MARCH 1st:

Context Free Grammers:  Define valid token sequences
                        Define structure of a program

  Example:  Rules:
             E -> E + T
                | T

             T -> T * P
                | P

             P -> ID
                | (E)

                 Parse Tree:

                    E       A+B*C Operator predence
                   / \
                  E + T
                  |  / \
                  T T * P
                  | |   |
                  P P   |
                  | |   |
                 ID ID  ID

          On the right is the entire subtree that does multiplication,
          then it gets added to the left side.
             A+B  Expression
                    /  \
                   E + Term

          -> captures operator precedence automatically
          -> operator associativity ( tells order in which you do operations)
          -> computers are left associative in the absence of explicit
                        parenthesis

          When do you need associativity right to left?
                                    -> assignment
                                    -> exponentiation
                        To change associativity, just change the first rule:
                                    from E -> E + T  to E -> T + E

              REMEMBER: operators closest to identifier have higher priority

          Grammers give a great deal of power!

**Problems you can create and then how to deal with them:

            Example:

             Grammer:   E -> E - E
```

```
                          | ID

          This can give two different parse trees:

        E            OR                  E
       / \                              / \
      E - E                            E - E     (wrong associativity)
     /    |                            |    \
    E-E   ID                           ID   E-E
    | |                                     | |
   ID ID                                   ID ID

                     ID - ID - ID gives two different parse trees (BAD)
```

-> Grammers that allow > 1 parse tree are AMBIGUOUS

-> Context Free Grammers allow more than one derivation for same problem

YAK, CUP reject ambigous grammers

NOTE: In our program, IF THEN ELSE is ambigous
-> definition is valid but not what you want, error can not autom. machine

Example: (non Term in CAPS; term is in lower)

```
S -> A B    (has bugs independent of strings, user may
   | x                   or may not have meant to generate)
B -> b
A -> a A
C -> d
```

has flaws:

1. Every non-term must generate at least one string of all terminals (including lambda)

   -> for our example: infinite recursion, not stopping criteria criteria -> can't generate a string of only terms

2. All non-terminals must be reachable from start string

   -> for our example: no way to get to C; production never used

**How can we program this? Grammer analysis program; essentially worklist algorithm.

Example: Algorithm to find non-terminals that fail rule #1

```
    A) mark all terminal symbols
B) REPEAT:
            if all symbols in the right hand side (RHS) are
               marked, then mark left hand side (LHS) symbols
       UNTIL NO CHANGES
```

Example: Algorithm to march reachable non-terminals (from S)

```
A) mark S (trivially reachable)
B) REPEAT
          if LHS of production is marked, then mark all non-
             terminals on RHS
     UNTIL NO CHANGES
```

Example: Alogrithm to mark lambdas

```
          (example: optional semicolon -> ;
                                             | lambda  )
```

```
A) if A -> lambda in a production, then mark A (all occurances)
B) REPEAT
          if B->CD......X in a production and the entire RHS is
             marked, then mark LHS
     UNTIL NO CHANGES
```

Any questions? email piltch@cs.wisc.edu (Naomi)