LL(1) Parsing tables:

- Each row: non-terminal to expand (A)
- Each column: lookahead token (a)
- Entry is the production that the parser needs to call to expand A

Top-down parsing:

- Parse tree is built from the top to the leaves, always expand leftmost nonterminal

FIRST(\alpha) is the set of terminals `a` that begin the strings derived from \alpha

FOLLOW(A) is the set of terminals b (including $$) that can appear immediately to the right of A in some sentential form

Constructing LL(1) parsing table uses PREDICT(A->\alpha) into each parse table entry
If each entry in the table contains at most one production, G is LL(1).
To make LL(1) grammar, don't use left recursion. Place tokens on the left instead of nonterminals

LR parser can handle left recursion
Works by moving tokens off the stack left to right, and when possible reducing the rightmost expressions

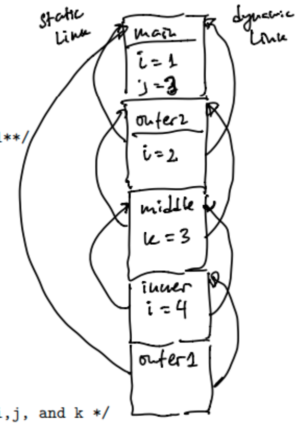
```
E -> E a T | T # this is an unambiguous grammar
T -> T b F | F # with precedence being c -> b -> a
F -> F c | id
```

Scoping rules map variable to declaration

- Most languages use static scoping, where mapping from variable to declaration is made at compile time, also known as lexical scoping

**Question 5.** (Static and dynamic scoping. 20pts) Consider the following program with block structure and nested subroutines.

```
procedure main

  i : integer := 1; j : integer := 3

  procedure outer1()
    write (i)    /* prints i on console */ /**1**/

  procedure outer2()

    i : integer;

    procedure middle(k : integer)
      procedure inner()
        i : integer := 4
        outer1()

      /* body of middle */
      inner()
      return (i,j,k) /* returns the tuple of i,j, and k */
               (2,3,3)
    /* body of outer2 */
    i := 2
    return middle(j)

  /* body of main */
  write (outer2())
  write (i, j) /* prints the tuple of i and j, e.g., write (1, 2) prints (1, 2) on console */
```




(i) (6pts) What does this program print under static scoping rules?

1 /* outer1 */
(2, 3, 3) /* write (outer2()) */
(1, 3) /* write (i,j) */

**Question 7.** (Prolog programming. 21pts) Interesting lists in Prolog.

(i) (6pts) Write a `insertPlus(L,R)` predicate in Prolog that adds a list of numbers. A sample interaction follows:

```
?- insertPlus([2,3,4],R).
R = 9.
```

insertPlus([], 0).
insertPlus([H|T], R) :- integer(H), insertPlus(T, Rt), R is H+Rt.

```
reverse([1,2,3], A)
sum_list([1,2,3], S)
append(A, B, C) # C is result of appending
intersection(A, B, C)
union(A, B, C)
subtract(A, B, C) # subtract B elements from A, result in C
```