Problems with Recursive Descent Parsing
Recursive descent is simple. What could possibly go wrong?
Problem 1 -- Recursive Descent can't handle left-recursive rules.

Rule $E ::= E + T \mid T$

becomes

```c
void E() {
    E()
    ..... 
    } 
```

You know that can't work.
This is a real problem. We have already seen that left-recursive rules are important for expression grammars because they give us left-associative operators, and these are an important part of most programming languages.

The solution used in APL was to make all operators right-associative so you don't need left-recursive rules, but this feels wrong to most programmers.
We will handle this by modifying the recursive descent algorithm for left-recursive rules.

Consider a typical left-recursive rule:

\[ E ::= E+T | E-T | T \]

For the moment think of \( T \) as a terminal symbol, as if our grammar was

\[ E ::= E + t | E - t | t \]

This rule generates a chain of \( t \)'s, with a + or - operator between each pair:

\[ t1 \pm t2 \pm t3 \pm \ldots \pm tn \]
t₁ ± t₂ ± t₃ ± ... ± tₙ

Instead of recursing to get the prefix of this, we'll think about it as follows. We know it has to start with a t, so we grab that t, and consume its tokens. If the next token is a + or -, we are still in the E expression so we do a getNextToken() to get past this operator, and get another t. This continues until the token following one of our t's is not a + or -
This leads to the following code:

```c
void E( ) {
    T();
    while (IsAddOp( currentToken )) {
        getNextToken();
        T();
    }
}
```

Here `IsAddOp()` is a simple function that returns true if its argument is a token that represents + or -.
This is fine as far as recognizing strings, but we usually want our parser to build a parse tree. We know we want this to be a left-associative tree, so the expression $t_1 - t_2 - t_3$ parses to
TreeNode E() {
    TreeNode t = T();
    while (IsAddOp(currentToken)) {
        TreeNode t1 = new TreeNode();
        t1.token = currentToken;
        getNextToken();
        t1.leftChild = t;
        t1.rightChild = T();
        t = t1;
    }
    return t;
}
The BPL grammar contains rules

\[
E ::= E+T \mid E-T \mid T \\
T ::= T*F \mid T/F \mid T\%F \\
F ::= -F \mid \&Factor \mid *Factor \mid Factor \\
\]

... 

The E and T procedures need to have loops that build left-associative trees. The F rule is not left-recursive, so you can use the usual recursive descent techniques for it.
There is no general fix for the problem of left-recursive rules -- if you find one in a grammar that you are parsing, you either need to find a trick to modify your recursive descent techniques to fit the rule, or use a different parsing technique. This is one of the reasons that commercial compiler shops generally don't use recursive descent.
Problem 2: Recursive descent only works if we can tell which rule to use. If you have grammar rules $A ::= B \mid C$ and rules $B$ and $C$ start with the same tokens, we can't tell which to use.

For example, consider the grammar

\[
A ::= aBa \mid B \mid a \quad (A \text{ is the start symbol})
\]
\[
B ::= aBb \mid b
\]

This is an unambiguous grammar that generates

\{ $a^n b^n$, $a^n b^n a$: $n \geq 0$ \}
If our input string is aaaaaaabbabbbba we want the first rule we use (the top of our parse tree) to be $A ::= aBa$; if the input string is aaaaaaabbbbbbbb, we want the first rule to be $A ::= B$. We have to read across 15 symbols before we determine which rule to use, and by the time we have done that our current token is the end-of-input symbol.