06 - Recursive-Descent Parsing

Building a simple recognizer:

1. Convert BNF grammar to EBNF and syntax diagrams.
2. There must be an identifier “token” that points to the current token.
3. There must be a function “match” that:
    a. tests whether current token is a particular value, and
    b. advances the token pointer (“token”) if it is.
4. Make a function for each non-terminal in the grammar.
5. The control flow for each function is the same as the syntax diagram. When a non-terminal is encountered, call the corresponding function. When a token *t* is encountered, call match(t).
6. Add a test for end-of-token-stream to the syntax diagrams. (you may need to add a separate “start” rule).
7. Start with “token” set to the first token in the incoming token stream.

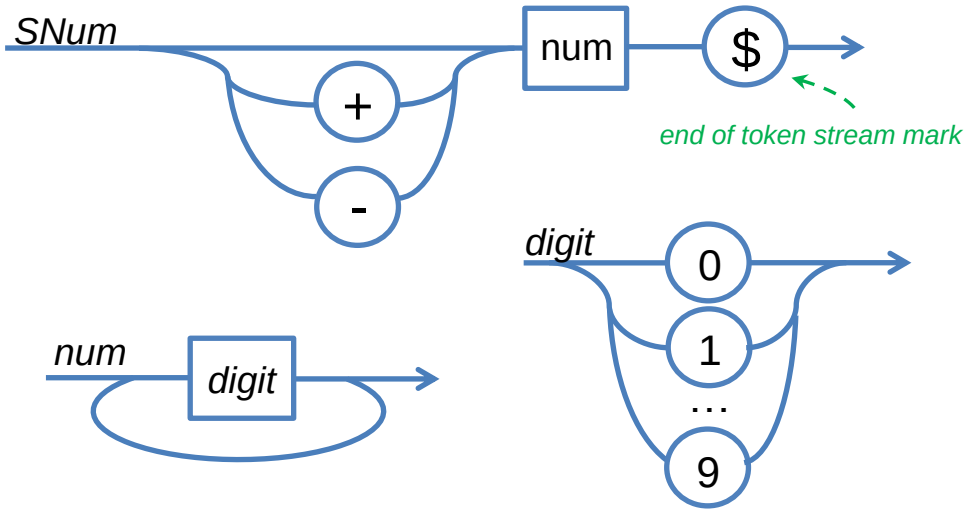
Example:

```
<SNum>   ::= + <num> | - <num> | <num>
<num>    ::= <num> <digit> | <digit>
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Convert to EBNF:

```
<SNum>   ::= [(+|-)] <num>
<num>    ::= <digit> { <digit> }
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Convert to Syntax Diagrams:



Recursive-Descent pseudocode:

```
SNum()
  if token=='+'
    match('+')
  else if token=='-'
    match('-')
  num()
  match($)
```

```
digit()
  if token in [0,1,...,9]
    match(token)
  else
    error
```

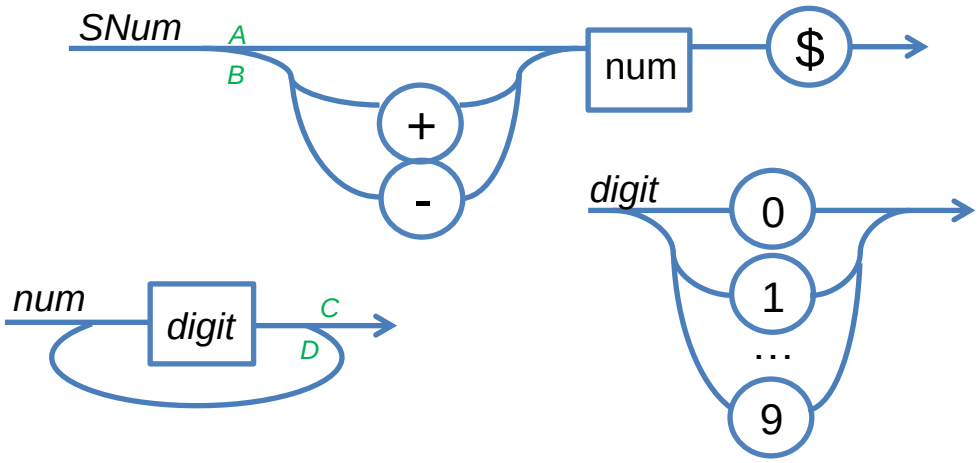
note subtle trick!

```
num()
  do
    digit()
  while token in [0,1,..,9]
```

```
match(t)
  if token==t
    advanceTokenPtr
  else
    error
```

Note the control flow in each of the functions matches the control flow in the corresponding syntax diagram. Also note that whenever a non-terminal is encountered in a diagram, a call is made to the corresponding function.

It is possible to prove whether or not a particular grammar is parse-able by recursive-descent. This entails first identifying each "branch", or decision point in the syntax diagrams, and then showing that the sets of possible tokens one might encounter first on each alternative branch don't overlap. In our diagrams, the branches are labeled *A*, *B*, *C*, *D*, as follows (the decision point in `<digit>` is a trivial case – clearly each branch leads to a different token – so we don't bother to label them):



In general, there are two cases:

1. the branches lead to different items within the rule, or
2. one branch leads to an item within the rule, and the other branch exits the rule.

In case 1, it is necessary to show that the "first" set of each branch is disjoint. In case 2, it is necessary to show that the "first" set of the branch that stays within the rule is disjoint from the "follow" set of the rule itself.

"First" sets are usually fairly simple to determine by inspection, by noting those tokens that can first be encountered when traversing a path. Sometimes finding a "first" set involves unioning together many items… for example, *first(SNum)* = *first(num)* $\cup$ {+} $\cup$ {-} = *first(digit)* $\cup$ {+,-} = {0..9,+,-}. First sets can be determined either for non-terminals, or for branches.

"Follow" sets can be trickier, and involve examining the entire grammar to find the various tokens that might follow a given non-terminal. For example, to find the follow set of `num`, we ***wouldn't*** look in the `num` rule itself, but instead look for occurrences of `num` in the grammar. We would then union together all of the tokens that can immediately follow those occurrences. In the case of `num`, there is only one such occurrence, in the rule `SNum`. Immediately following num is the token "$". So, *follow(num)* = {$}. Follow sets are only ever determined for non-terminals.

Before we proceed with the proof for the above grammar, let's first discuss the *end-of-token-stream* marker. Here it is denoted with $. The marker is inserted by the lexical scanner (not the programmer!). So the parser can be assured that there will always be exactly one, and it will be at the end of the token stream. It is the parser's job to verify that the marker exists after a successful parse of a given expression. Treating the marker like a token, and performing "match" on it, is a convenient way of doing this.

It is worth noting that, without the end-of-token-stream marker, the parser might erroneously accept an illegal stream, if it appears to be a legal one followed by garbage. This can commonly happen, for example if there is a missing left parenthesis.

Back to our grammar, above. To prove that it is parse-able by recursive descent, we must identify each decision point, label the branches for each one, and utilize either case 1 or case 2 as appropriate.

Our grammar happens to have one of each such case.

At decision point A-B (above), there are three disjoint branches:

```
First(branch #1) = First(num) = First(digit) = {0,1,…,9}
First(branch #2) = {+}
First(branch #3) = {-}
```

By inspection it is easy to see that the three "first" sets are disjoint.

At decision point C-D (above), there are two disjoint branches, branch "C" that exits the rule, and branch "D" that leads to an item within the rule. We therefore use the method shown in case 2 (above) as follows:

```
First(branch #2) =  First(digit) = {0,1,…,9}
Follow(num) = {$}
```

By inspection we can see that the two sets are disjoint.

Since all of the decision points satisfy the condition(s) for recursive-descent parsing outlined in the two cases at the top of the page, the grammar is parse-able by recursive-descent. Of course, we already knew that, because we wrote the parser!