6.0 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.
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.

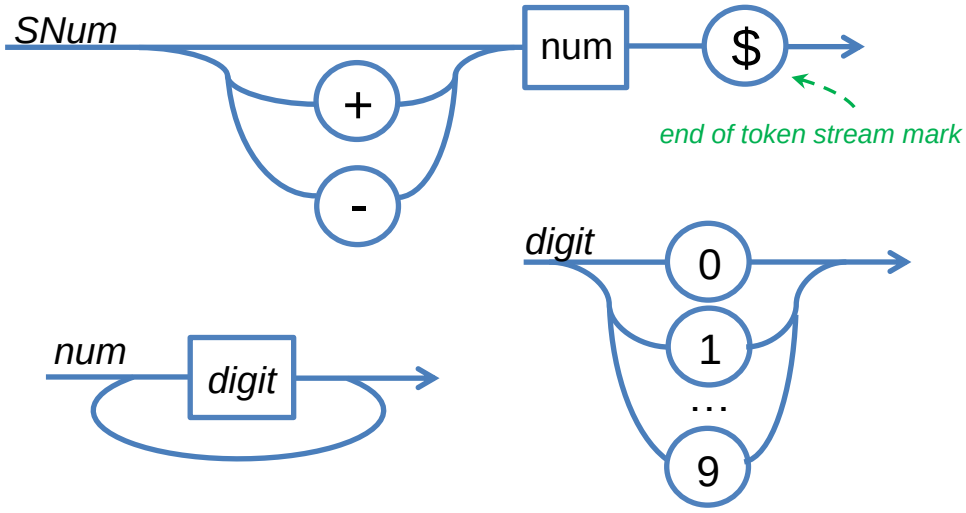
Example:

```
<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($)
```

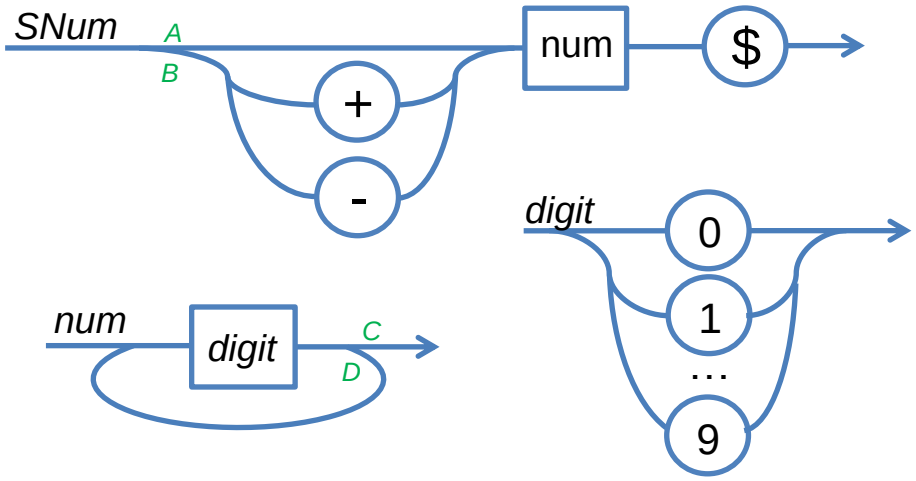
```
num()
  do
    digit()
  while token in [0,1,..,9]
```

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

note subtle trick!

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

Proof that the above grammar is parse-able by recursive descent:



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

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

branch #2 = `{+}`

branch #3 = `{-}`

At decision point C-D (above), there are two disjoint branches:

branch #1 = `Follow(num) = {$}`

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

The decision point in `<digit>` is a trivial case.