Example of recursive-descent parsing

The technique is to use a function for each nonterminal (variable) in grammar. The function will attempt to match one of the rules on the right side of the production.
Here's a grammar for type names:

```
type   -> simple | "*" type | "array" "[" simple "]" "of" type
simple -> "int" | "char" | NUM ".." NUM
```

Note that NUM is a token, not a grammar variable. We assume that a multi-digit number has already been scanned and its value is stored with NUM token. So NUM doesn't need to be broken down further (i.e. into its separate digits - because digits themselves are not tokens).

Below is the pseudo-code for a recursive descent parser of above grammar. In a real compiler, we would also need to create the parse tree or perform some other action as we figure out which grammar rule is being applied. Note: In the first case where the right-hand side begins with a nonterminal called simple, we look at what a simple type can start with.

```
void type()                          // Handle productions of type
{
  if (token is "int" or "char" or a number)     // type -> simple
    simple();
  else if (token is "*")                        // type -> "*" type
  {
    match("*");
    type();
  }
  else if (token is "array")                    // type -> "array", etc.
  {
    match("array");
    match("[");
    simple();
    match("]");
    match("of");
    type();
  }
  else                                          // Otherwise, syntax error!
    error();
}
-------------------------------------------------------------------------
void simple()                        // Handle productions of simple
{
  if (token is "int")                           // simple -> "int"
    match("int");
  else if (token is "char")                     // simple -> "char"
    match("char");
  else if (token is a NUM)                      // simple -> NUM ".." NUM
  {
    match(NUM);
    match("..");
    match(NUM);
  }
  else                                          // syntax error
    error();
}
-------------------------------------------------------------------------
void match(token t)                  // Match means see if the token
{                                    // is what we expect it to be.
  if (token matches t)               // And then we go on to the next token.
    token = find_next_token();
  else
    error();
}
```