Recursive Descent Parsing

- Predictive parsing method for LL(1) grammar (LL with one token lookahead)
- Based on recursive subroutines
  - Each nonterminal has a subroutine that implements the production(s) for that nonterminal so that calling the

subroutine will parse a part of a string described by the nonterminal
- When more than one alternative production exists for a nonterminal, lookahead token from scanner should decide which production is to be applied

LL(1) for a simple calculator language

```
<expr> -> <term> <term_tail>

<term_tail> -> <add_op> <term> <term_tail> | ε

<term> -> <factor> <factor_tail>

<factor_tail> -> <mult_op> <factor> <factor_tail> | ε

<factor> -> ( <expr> ) | - <factor>
          | identifier | unsigned_integer

<add_op> -> + | -

<mult_op> -> * | /
```

A Recursive Descent Parser

Pseudo-code outline of recursive descent parser for the calculator grammar

```
procedure expr()
  term(); term_tail();
procedure term_tail()
  case (input_token())
  of '+'or '-': add_op(); term(); term_tail();
  otherwise: /* skip */
procedure term()
  factor(); factor_tail();
procedure factor_tail()
  case (input_token())
  of '*' or '/': mult_op(); factor(); factor_tail();
  otherwise: /* skip */
procedure factor()
```

```
  case (input_token())
  of '(': match('('); expr(); match(')');
  of '-': factor();
  of identifier: match(identifier);
  of number: match(number);
  otherwise: error;
procedure add_op()
  case (input_token())
  of '+': match('+');
  of '-': match('-');
  otherwise: error;
procedure mult_op()
  case (input_token())
  of '*': match('*');
  of '/': match('/');
  otherwise: error;
```

Exercise: Write a recursive descent parser in Java for this grammar. Answer:

Example Recursive Descent Parsing

- The *dynamic call graph* of a recursive descent parser corresponds exactly to the parse tree of input
- Call graph of input string `1+2*3`