```ruby
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

require 'logger'

class Rule
  Match = Struct.new :pattern, :block

  def initialize(name, parser)
    @logger = parser.logger
    # The name of the expressions this rule matches
    @name = name
    # We need the parser to recursively parse sub-expressions occurring
    # within the pattern of the match objects associated with this rule
    @parser = parser
    @matches = []
    # Left-recursive matches
    @lrmatches = []
  end

  # Add a matching expression to this rule, as in this example:
  #   match(:term, '*', :dice) {|a, _, b| a * b }
  # The arguments to 'match' describe the constituents of this expression.
  def match(*pattern, &block)
    match = Match.new(pattern, block)
    # If the pattern is left-recursive, then add it to the left-recursive set
    if pattern[0] == @name
      pattern.shift
      @lrmatches << match
    else
      @matches << match
    end
  end

  def parse
    # Try non-left-recursive matches first, to avoid infinite recursion
    match_result = try_matches(@matches)
    return nil if match_result.nil?
    loop do
      result = try_matches(@lrmatches, match_result)
      return match_result if result.nil?
      match_result = result
    end
  end

  private

  # Try out all matching patterns of this rule
  def try_matches(matches, pre_result = nil)
    match_result = nil
    # Begin at the current position in the input string of the parser
    start = @parser.pos
    matches.each do |match|
      # pre_result is a previously available result from evaluating expressions
      result = pre_result.nil? ? [] : [pre_result]

      # We iterate through the parts of the pattern, which may be e.g.
      #   [:expr,'*',:term]
      match.pattern.each_with_index do |token,index|

        # If this "token" is a compound term, add the result of
        # parsing it to the "result" array
        if @parser.rules[token]
          result << @parser.rules[token].parse
          if result.last.nil?
            result = nil
            break
          end
          @logger.debug("Matched '#{@name} = #{match.pattern[index..-1].inspect}'")
        else
```

```
71            # Otherwise, we consume the token as part of applying this rule
72            nt = @parser.expect(token)
73            if nt
74              result << nt
75              if @lrmatches.include?(match.pattern) then
76                pattern = [@name]+match.pattern
77              else
78                pattern = match.pattern
79              end
80              @logger.debug("Matched token '#{nt}' as part of rule '#{@name} <= #{pattern.inspect}'")
81            else
82              result = nil
83              break
84            end
85          end
86        end
87        if result
88          if match.block
89            match_result = match.block.call(*result)
90          else
91            match_result = result[0]
92          end
93          @logger.debug("'#{@parser.string[start..@parser.pos-1]}' matched '#{@name}' and generated '#
    {match_result.inspect}'") unless match_result.nil?
94          break
95        else
96          # If this rule did not match the current token list, move
97          # back to the scan position of the last match
98          @parser.pos = start
99        end
100     end
101
102     return match_result
103   end
104 end
105
106 class Parser
107
108   attr_accessor :pos
109   attr_reader :rules, :string, :logger
110
111   class ParseError < RuntimeError
112   end
113
114   def initialize(language_name, &block)
115     @logger = Logger.new(STDOUT)
116     @lex_tokens = []
117     @rules = {}
118     @start = nil
119     @language_name = language_name
120     instance_eval(&block)
121   end
122
123   # Tokenize the string into small pieces
124   def tokenize(string)
125     indentation = 0
126     indent_stack = []
127     @string = string.clone
128     until string.empty?
129       # Unless any of the valid tokens of our language are the prefix of
130       # 'string', we fail with an exception
131       raise ParseError, "unable to lex '#{string}" unless @lex_tokens.any? do |tok|
132         match = tok.pattern.match(string)
133         # The regular expression of a token has matched the beginning of 'string'
134         if match
135           @logger.debug("Token #{match[0]} consumed")
136           # Also, evaluate this expression by using the block
137           # associated with the token
138           @tokens << tok.block.call(match.to_s) if tok.block
139           # consume the match and proceed with the rest of the string
```

```
            string = match.post_match

            # check indentation and generate tokens if necessary
            if tok.pattern == /\A\n+/
              @tokens << :newline if @tokens && @tokens.last != :newline
              new_indentation = /\A */.match(string)[0].length
              if new_indentation < indentation
                while not indent_stack.empty? and indent_stack.last > new_indentation
                  indent_stack.pop
                  @tokens << :dedent
                end
                @tokens << :newline
              elsif new_indentation > indentation
                indent_stack << new_indentation
                @tokens << :indent
              end
              indentation = new_indentation
            end
            true
          else
            # this token pattern did not match, try the next
            false
          end # if
        end # raise
      end # until
    end

    def parse(string, interactive=false)
      @tokens = interactive ? [:prompt] : []
      # First, split the string according to the "token" instructions given.
      # Afterwards @tokens contains all tokens that are to be parsed.
      tokenize(string)

      # These variables are used to match if the total number of tokens
      # are consumed by the parser
      @pos = 0
      @max_pos = 0
      @expected = []
      # Parse (and evaluate) the tokens received
      result = @start.parse
      # If there are unparsed extra tokens, signal error
      if @pos != @tokens.size
        @tokens = @tokens[0..@max_pos]
        @tokens.delete(:indent)
        i = (i = @tokens.reverse.index(:newline)).nil? ? 0 : -i
        @tokens.shift if @tokens.first == :newline
        @tokens.pop if @tokens.last == :newline
        @tokens.shift if @tokens.first == :prompt
        raise ParseError, "Unexpected '#{@tokens.last}' in '#{@tokens[i..-1].join(' ')}'"
      end
      return result
    end

    def next_token
      @pos += 1
      return @tokens[@pos - 1]
    end

    # Return the next token in the queue
    def expect(tok)
      return tok if tok == :empty
      t = next_token
      if @pos - 1 > @max_pos
        @max_pos = @pos - 1
        @expected = []
      end
      return t if tok === t
      @expected << tok if @max_pos == @pos - 1 && !@expected.include?(tok)
      return nil
    end
```

```ruby

    def to_s
      "Parser for #{@language_name}"
    end

    private

    LexToken = Struct.new(:pattern, :block)

    def token(pattern, &block)
      @lex_tokens << LexToken.new(Regexp.new('\\A' + pattern.source), block)
    end

    def start(name, &block)
      rule(name, &block)
      @start = @rules[name]
    end

    def rule(name,&block)
      @current_rule = Rule.new(name, self)
      @rules[name] = @current_rule
      instance_eval &block
      @current_rule = nil
    end

    def match(*pattern, &block)
      @current_rule.send(:match, *pattern, &block)
    end

  end
```