Little Language [Grand]

Intent

Given the grammar of a simple language, provide a parser.

Motivation

Many problems can be expressed using small grammars. Applications then must provide a parser that, for example, allows to build an abstract syntax tree (AST) which in turn can be interpreted by an interpreter (see also the Interpreter pattern).

Such problems often fall into the following parts:

- the definition of the lexicals of the language;
- the definition of the syntax of a language;
- the definition of its semantics.

The task of the *scanner* (also referred to as *lexical analyzer*) is to extract the lexicals of the language. The details of a scanner, especially the specification of lexicals by using regular expressions, are not discussed here. The Java class `java.io.StreamTokenizer` provides a simple and configurable scanner.

The grammar can be defined by a set of production rule P_i:

$$P_i := P_{i1}\ P_{i2}\ \ldots\ P_{in}$$

To determine the top-level production, one of the production P_i is uniquely specified (not shown, here, the first one). Example of a grammar using a EBNF-like grammar style:

```
expression           := orExpression
orExpression         := andExpression { "+" orExpression }
andExpression        := simpleExpression { "*" andExpression }
simpleExpression     := "(" orExpression ")"
                     | notExpression
                     | variableExpression
                     | constantExpression
notExpression        := "~" ( variableExpression |
                              constantExpression )
variableExpression   := NAME
constantExpression   := "true"
                     | "false"
```

where quoted words and words in capital letters are terminals; all other words are non-terminals.

Note that the above grammar does not produce the “natural” grouping of composed expressions. For example, expression `a * b * c` would be grouped as `a * (b * c)`.

During the process of recognizing productions, the parser needs to invoke semantic actions on the productions recognized so far. If, for example, an AST must be built, then the corresponding node of a tree must be created, possibly with zero or more sub-trees corresponding to the non-terminal symbols of the RHS of the production.

Mapping of Productions to Methods. We can write in a systematic way a *recursive-descent* parser. We write a class, say, `Parser`, which has a public method, say, `parse`, and a set of private methods. These private methods correspond to the productions of the EBNF grammar given.

A production of the form

$$\texttt{non-terminal}_1 := \texttt{non-terminal}_2\ \{\ \texttt{"op"}\ \texttt{non-terminal}_1\ \}$$

maps on a corresponding method of a class `Parser` having the following structure (pseudo code):

```
private BooleanExp method-for-NT1() {
   BooleanExp exp = method-for-NT2();
   while (token == "op") {
      nextToken(); // Consume next token.
      exp = new NT1-Node(exp, method-for-NT1());
   }
   return exp;
}
```

A production of the form

```
non-terminal1  := "(" non-terminal2 ")"
                | non-terminal3
                ...
                | non-terminaln
```

maps on a corresponding method of a class `Parser` having the following structure (pseudo code):

```
private BooleanExp method-for-NT1() {
   BooleanExp exp = null;
   if (token == "(") {
       nextToken(); // Read next token in order to procede.
       exp = method-for-NT2();
       expect(")"); // Check existence of closing parenthesis
       nextToken(); // Read next token in order to procede.
   } else if (token == denotes start of NT3) {
       exp = method-for-NT3();
   } else ...
   } else if (token == denotes start of NTn) {
       exp = method-for-NTn();
   } else {
       throw new SyntaxException
      ("Invalid nonterminal1, current token is: " +
       LexicalAnalyzer.tokenName(token));
   }
   return exp;
}
```

Helper method `nextToken` tells the scanner to read and return the next token:

```
private void nextToken() {
   token = lexer.nextToken();
}
```

Helper method `expext(Token)` must be used for checking the closing terminal symbol of block structures (or end-of.file markers):

```
private void expect(int t) throws SyntaxException
{
   if (token != t) {
      String msg = "Found " + LexicalAnalyzer.tokenName(token)
      + " when expecting token: "
      + LexicalAnalyzer.tokenName(t);
      throw new SyntaxException(msg);
   }
}
```

A production of the form (grouping)

```
non-terminal1 := "~" ( non-terminal2 | ... | non-terminaln )
```

maps on a corresponding method of a class `Parser` having the following structure (pseudo code):

```
private BooleanExp method-for-NT1() {
   BooleanExp exp = null;
   nextToken(); // Read next token in order to procede.
   if (token == denotes start of NT2) {
      // perhaps consume next token, then:
      exp = new NT1-Node(method-for-NT2());
   } else ...
   } else if (denotes start of NTn) {
      // perhaps consume next token, then:
      exp = new NT1-Node(method-for-NTn());
   } else {
       throw new SyntaxException("Invalid non-terminal1");
   }
   return exp;
}
```

Reading New Tokens. Upon matching terminal symbols such as “+” or “(” you must read the next token before continuing. There is often more than one place where you can do that. One possible strategy is to that in the code that handles the (non-) terminal that follows the matching terminal symbol. Consider, for example, the production:

```
notExpression := "~" ( variableExpression|constantExpression)
```

In order to handle this production, terminal symbol “~” must have been recognized. Thus, in the body of the method for this production, you must arrange that the next token (which is either a `variableExpression` or a `constantExpression`) must be read:

```
private BooleanExp notExpression() throws SyntaxException
{
   BooleanExp exp = null;
   nextToken(); // Read next token in order to procede.
   if (...) {
       ...
   } else if (...) {
       ...;
   } else {throw new SyntaxException("Invalid NotExpression");
}
```

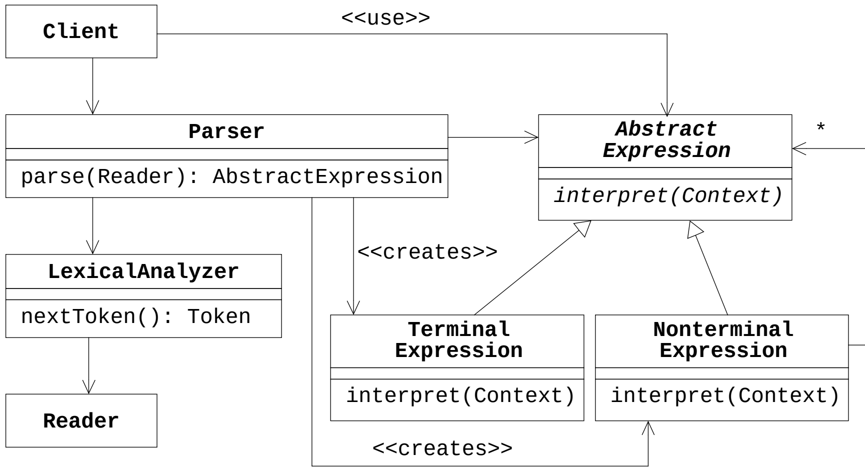
Consuming Closing Block Tokens. Some production have closing terminal tokens. For example, token “`)`” in the following production “closes” an `orExpression`:

```
simpleExpression  := "(" orExpression ")"
                   | notExpression
                   ...
```

The part or the body’s method for `simpleExpression` must consume the “`)`” token:

```
private BooleanExp simpleExpression() throws SyntaxException
{
   BooleanExp exp = null;
   if (token == LexicalAnalyzer.LPAR) {
       nextToken(); // Read next token in order to procede.
       exp = orExpression();
       expect(LexicalAnalyzer.RPAR);
       nextToken(); // Read next token in order to procede.
   } else if (...) {
       ...;
   } else {throw new SyntaxException("Invalid NotExpression");}
}
```

Structure



Applicability

- Use Little Language if the grammar is simple and small.
- Use parser generators if the grammar is complex.

Participants

- **`AbstractExpression`**:
  - declares an abstract `interpret` method that is common to all node in the abstract syntax tree
- **`TerminalExpression`**:
  - implements the `interpret` method associated with terminal symbols of the grammar
  - a concrete class is required for each terminal symbol of the grammar
- **`NonterminalExpression`**:
  - one of such class is required for each rule `R ::=` R_1 R_2 `...` R_n
  - maintains

Collaborations

- `Client` creates instances of `Reader` and `Parser`.
- `Parser` creates an instance of `LexicalAnalyzer`.
- `Parser` uses a `LexicalAnalyzer` to get the next `Token`.
- `Parser` creates an `AbstractExpression`.
- `Client` interpretes an instance of `AbstractExpression`.

Consequences

- It is easy to write a parser for a small language.
- Little Language creates an AST with *heterogeneous* nodes. ASTs with homogenous nodes could easily be created, though.
- The AST is suitable for being interpreted, see Interpreter.

Implementation

Parser logic is captured in one class only. This is not quite object-oriented, but makes perfect sense in this case (high cohesion of the `Parser` class).

```
public LexicalAnalyzer(Reader r) {
   input = new StreamTokenizer(r);
   // configure StreamTokenizer:
   input.resetSyntax();
   input.eolIsSignificant(false);
   input.wordChars('a', 'z');
   input.wordChars('A', 'Z');
   input.ordinaryChar('+');  // or
   input.ordinaryChar('*');  // and
   input.ordinaryChar('~');  // not
   input.ordinaryChar('(');
   input.ordinaryChar(')');
   input.whitespaceChars('\u0000', ' ');
}
```

Sample Code

The excerpt of a lexical analyzer is:

```
import java.io.*;

public class LexicalAnalyzer {
   private StreamTokenizer input;
   private int lastToken;

   // Tokens.
   static final int INVALID_CHAR = -1;
   static final int NO_TOKEN = 0;
   static final int AND = 1;
   // etc. for remaining tokens, then:
   static final int EOF = 99;
```

```
   public int nextToken() {
      int token;
      try {
         switch (input.nextToken ()) {
            case StreamTokenizer.TT_EOF:
               token = EOF;
               break;
            case StreamTokenizer.TT_WORD:
               if (input.sval.equalsIgnoreCase("false"))
                  token = FALSE;
               else if (input.sval.equalsIgnoreCase("true"))
                  token = TRUE;
               else token = VARIABLE;
               break;
            case '+':
               token = or;
               break;
            // etc.
         }
      } catch (IOException ex) { token = EOF; }
      return token;
}
```

See exercises for a complete lexical analyzer class. Note also that it is very helpful for debugging purposes to add a `main` method to `LexicalAnalyzer`.

The excerpt of class `Parser` looks like:

```
public class Parser {
   private LexicalAnalyzer lexer; private int token;
   public BooleanExp parse(Reader r) throws SyntaxException {
      lexer = new LexicalAnalyzer(r);
      nextToken();
      BooleanExp exp = orExpression();
      expect(LexicalAnalyzer.EOF);
      return exp;
   }
   private BooleanExp orExpression() throws SyntaxException {
      BooleanExp exp = andExpression();
      while (token == LexicalAnalyzer.OR) {
         nextToken(); // Read next token in order to procede.
         exp = new OrExp(exp, orExpression());
      }
      return exp;
   }
```

```
   private BooleanExp notExpression() throws SyntaxException {
      BooleanExp exp = null;
      nextToken();
      // Handle the two + error cases for not expressions:
      // Rest of body: left as an exercise.
      return exp;
   }
   private BooleanExp variableExpression() throws S.Exception
      BooleanExp exp = null;
      if (token == LexicalAnalyzer.NAME) {
          exp = new Variable(lexer.getString());
      } else { throw new S.E.("Invalid VariableExpression");
      }
      nextToken();
      return exp;
   }
   private BooleanExp constantExpression() throws S.Exception
      BooleanExp exp = null;
      // Handle the two + error cases for constant expressions
      // Rest of body: left as an exercise.
      return exp;
   }
```

```
   private BooleanExp andExpression() throws SyntaxException {
      BooleanExp exp = simpleExpression();
      // Rest of body: left as an exercise.
      return exp;
   }

   private BooleanExp simpleExpression() throws S.Exception {
      BooleanExp exp = null;
      // Handle the four + error cases for simple expressions:
      if (token == LexicalAnalyzer.LPAR) {
         nextToken();
         exp = orExpression();
         expect(LexicalAnalyzer.RPAR);
         nextToken();
      } else if (token == LexicalAnalyzer.NOT) {
         // Treatement of remaining cases: left as an exercise.
         ...
      } else {
         throw new SyntaxException("...missing input");
      }
      return exp;
   }
```

```
   private void expect(int t) throws SyntaxException {
      if (token != t) { // throw SyntaxException...
      }
      nextToken();
   }

   private void nextToken() {
      token = lexer.nextToken();
   }

   // Recommended: main() method or JUnit test class,
   // left as an exercise.
}
```

Related Patterns

- Interpreter pattern to interpret the generated AST.
- Visitor allows to move the interpretation logic away from the node classes of the AST into a separate class.