Assignment 0: Syntax and Parsing

Isak Karlsson & Beatrice Åkerblom
{isak-kar, beatrice}@dsv.su.se

1 Introduction

This assignment is the first of four assignments. The assignment is due to 2013-11-24 and shall be uploaded to the course web site. The assignment is done in groups of two (2) students (exceptions are *only* provided by Beatrice).

Please, if in any kind of doubt, use the ilearn2 discussion forum.

1.1 Requirements

1. Your submission should be uploaded (code, documente, etc. as an archive) in ilearn before the deadline.
2. All authors must be specified in the upload (both as a note to the submission and in code, documents, etc.).

2 The Assignment

2.1 Implement a simple tokenizer & parser in Java)

Using (Sebesta, 2010) implement a simple tokenizer, parser and interpreter for the following EBNF[1]-grammar:

```
assign = id '=' expr
expr   = term [{('+' | '-') term}]
term   = factor [{('*' | '/') factor}]
factor = int | '(' expr ')'
```

Where `int` is defined as `(0..9)+` and `id` as `(a..z)+`.

[1] http://en.wikipedia.org/wiki/Extended_Backus-Naur_Form

The first step is to write a simple *scanner* that extract chars from a String. Then, using a *tokenizer* tokenize the chars from the scanner into tokens. The third step is to implement a simple one-look-a-head recursive decent *parser* that produces a parse tree. The final step is to implement a *visitor* that visits each node in the parse tree and produces the result of evaluating an assignment. For example, given the input string ``a = 10 + 20 * 2'', the program shall produce an instance of `Map<String, Number>` where `map.get(``a'')` would return `50`. Your interpreter only need to handle integers. Thus, `11/2` would return 7.

Use the supplied Java-interface and classes to implement the assignment.

2.1.1 Steps to complete

1. Implement a subclass of the interface *Scanner*. The constructor shall take a `String` (i.e. `new MyScanner(``a=1+2'')`) from which characters are extracted. The scanner can be implemented using, e.g., `String#charAt`. Follow the definitions of `int` and `id` from above.

```
Scanner sc = new MyScanner("a=2");
sc.next(); // a
sc.current(); // a
sc.peek(); // =
sc.current(); // a
sc.next(); // =
...
```

2. Implement a subclass of the interface `Tokenizer` that takes a *Scanner*-implementation as a constructor parameter. A start could be:

```
private Token extractToken() {
   char ch = sc.current();
   consumeWhiteSpaces();
   if(ch == Scanner.EOF) {
      return new Token(..., ..., Token.Type.EOF);
   }
   if(Character.isDigit(ch)) {
      return extractNumber();
   } else if (Token.Type.OPERATORS.containsKey(
               String.valueOf(ch))) {
      Token t =  new Token(String.valueOf(ch), String.valueOf(ch),
```

```
                        TokenType.OPERATORS.get(
                          String.valueOf(ch)));
        ch = sc.next();
        return t;
    } else if (Character.isLetter(ch)) {
      return extractIdentifier();
    } else {
      throw new RuntimeException("Unexpected character '" + ch + "'");
    }
}
```

Above, `sc` is an instance of `Scanner`.

3. When the tokenizer is implemented, implement a subclass of the `Parser`-interface. This class should take a `Tokenizer` as constructor parameter. Then, when `Parser#parse()` is called, read each token from the supplied tokenizer and construct a parse tree. For an example, see Sebesta (2009) pp. 188-206. You have been supplied the different *Node*s of the parse tree. Your implementation could look something like this:

```
public Node parse() {
  AssignNode node = new AssignNode();
  Token t = tokenizer.next();
  Token n = tokenizer.next();

  if(t.type() == Token.Type.IDENTIFIER &&
     n.type() == Token.Type.EQ) {
    node.left = new IdentifierNode();
    node.left.value = t.text();

    tokenizer.next(); // Move to start of expression
    node.right = this.parseExpression();
  } else {
    throw new RuntimeException("Invalid start of assignment");
  }

  return node;
}
```

You should implement one method for each rule in the syntax (e.g. `parseTerm()` etc.), that returns a corresponding `Node`. To test your parser (i.e., to see what tree is built) use `Node#toTree()` to construct a string representation of the parse tree.

4. The last (and probably simplest) step is to implement the visitor, that "visits" each node in the tree. Implement each method in `Visitor`. For example, `visitAssign`:

```
public Object visitAssign(AssignNode n) {
  Map<String, Number> assignments =
      new HashMap<String, Number>();
  String left = (String) visit(n.left);
  Number right = (Number) visit(n.right);
  assignments.put(left, right);

  return assignments;
}
```

5. Edit `Test.java` (i.e., insert your subclasses), compile, and run `Test.java`. The output should be:

```
Evaluated: 'a = 1 + 2' Expected: 'a'='3' Got: '3'
Evaluated: 'b= 1+ 2' Expected: 'b'='3' Got: '3'
Evaluated: 'c=1*2+11/2' Expected: 'c'='7' Got: '7'
Evaluated: 'd=(2+3)-(10*(20-3))' Expected: 'd'='-165' Got: '-165'
```

6. Upload the code you have written to "Assignment 0" at the course website.

3 Grading

In order to receive the grade E you have to implement the Scanner and the Tokenizer. To receive a grade better than C, you also have to implement the parser (and visitor).

Grading programming is more an art than a science. In the general case, it is extremely difficult to impossible to say that one program is better than the other. For the obvious reasons, it is impossible to cover the grading criteria completely. In any case, the points below are not totally black/white.

For the programming assignment 0 the following table will be used when grading the assignment.

Grade	Scanner	Tokenizer	Parser
A	Good	Good	Good
B	Good	Good	OK
C	Good	OK	OK-
D	OK	OK	Missing/OK-
E	OK-	OK-	Missing/OK-
Fx	Missing/OK-	Missing	Missing
F	Missing	Missing	Missing

Table 1: Grading scheme

References

R.W. Sebesta. *Concepts of Programming Languages [With Access Code].* Pearson Education, Limited, 2010.