Assignment 0: Syntax and Parsing

1 Introduction

This assignment is the first of four assignments on PROP. The assignment should be 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 under the headline "Block 0: Syntax, semantics and parsers" to ask questions abour this assignment.

1.1 Requirements

1. Your submission should be uploaded (code, documents, etc. as an archive) in ilearn before the deadline (see iLearn for details about the deadlines).
2. All authors must be specified in the upload (both as a note together with the submission and in code, documents, etc.).

2 The Assignment

2.1 Implement a simple tokenizer & parser in Java)

The assignment is to implement an interpreter, including the tokenizer and parser, in Java for assignment statements in the language described below.

If you are aiming for grade C, D or E you should implement an interpreter for the following grammar in EBNF (ISO standard):

Grammar 1

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

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

If you are aiming for grade A or B you should instead implement an interpreter for the following grammar in EBNF (ISO standard).

Grammar2:

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

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

The interpreter should consist of the following parts:

1. Scanner (already implemented for you),
2. Tokenizer,
3. Parser,
4. Evaluator.

2.2 Handed out code

There are some code files with already implemented parts of the assignment for you in the zip-file Assignment0-Code.zip:

1. `Program.java`, with the main program you should use.

2. `IScanner.java`, an interface for the Scanner implementations.
3. `Scanner.java`, an implementation of the Scanner.
4. `ITokenizer.java`, an interface for Tokenizer implementations.
5. `TokenizerException.java`, an exception class for tokenizer exceptions.
6. `Lexeme.java`, a class for holding information about a lexeme.
7. `Token.java`, an enumeration of different tokens.
8. `IParser.java`, an interface for Parser implementations.
9. `ParserException.java`, an exception class for parser exceptions.
10. `INode.java`, which includes an interface for nodes of a parse tree.
11. `program1.txt`, `program2.txt`, `parsetree1.txt` and `parsetree2.txt`, with examples of input and output to the program.

Read all files in Assignment0-Code.zip before you start implementing assignment 0! Do not change the given code files, but write your own code in separate files!

2.3 Tokenizer

The tokenizer should call the Scanner to get a stream of characters, and from that stream of characters create a stream of lexemes/tokens. You should implement a Tokenizer class, which implements the interface ITokenizer

2.4 Parser

The parser should call the Tokenizer to get a stream of lexemes/tokens, and from that stream of lexemes/tokens create a parse tree. The parse tree should be built of instances of the following classes: AssignmentNode, ExpressionNode, TermNode and FactorNode (and for grade A and B: BlockNode and StatementsNode). You should implement these classes, which should all implement the interface INode. For the lexemes in the parse tree you should use the Lexeme class. You should also implement a class Parser, which should implement the interface IParser

The INode interface includes a method buildString(StringBuilder, int), which should be used to create a string representation of the parse tree. Your program should output a parse tree as a string, which should exactly follow the given examples. An interpreter for Grammar 1 with input program1.txt should output parsetree1.txt, and an interpreter for Grammar2 with input program2.txt should output parsetree2.txt.

2.5 Evaluator

The Evaluator should be distributed in an object-oriented way over the nodes of the parse tree. The handed out INode interface includes a method `evaluate(Object[])`, which should return the value of the node. Your program should output a parse tree evaluation as a string, which should exactly follow the given example files program1.txt, parsetree1.txt, program2.txt and parsetree2.txt.

Note that the grammar rules are right recursive (tail recursive) but the arithmetic operators are left associative. This complexity needs to be handled in a correct way to get the grade A.

Note also that in Grammar2 the assignment statement may include identifiers (variables) in the expression. To be able to evaluate expressions including variables the evaluator needs to maintain a data structure including the current value of all found variables. Assume the default value of an unassigned variable is 0.

3 Grading

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.

Each part–Tokenizer, Parser and Evaluator–will be valuated with respect to the quality of the implementation:

1. NotOk means it is missing or not functioning,
2. Ok means it is substantially functioning and the code is ok written and ok structured.

3. Good means it is completely functioning and the code is well written and well structured.

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

Grade	Tokenizer	Parser	Evaluator
A	Good	Good	Good
B	Good	Good	OK
C	Good	OK	NotOK
D	OK	OK	Missing/NotOK
E	NotOK	NotOK	Missing/NotOK
Fx	Missing/NotOK	Missing	Missing
F	Missing	Missing	Missing

Table 1: Grading scheme

Remember that to get a grade A or B you must implement an interpreter for Grammar2, and an evaluator implementing the left associativity of the arithmetic operators.