LFTC Scanner Documentation

Symbol Table

For the storage of identifiers and contans I will use my implementation of a HashTable.

A list of fixed capacity will hold other lists, one for each bucket.

```
private ArrayList<ArrayList<Map.Entry<String, Integer>>> items;
private int size;
```

The token that is about to be put in the table will pe hashed with a function that sums up the ascii code for each character forming the toString() value of said token and will be divided by the capacity. The remainder will be the position of the list the token will be stored.

```
private int hash(String key) {
    int sum = 0;

    for (int i = 0; i < key.length(); i++)
        sum += key.charAt(i);

    return sum % size;
}
```

The token will be stored alongside a code. 0 for identifiers, 1 for identifiers and -1 forreserved words, operators and separators.

Program Internal Form

For the program internal form I will use a list of pairs. A pair will contain will be made upof another 2 pairs. The first pair will represent the string representation of a token and the code indicating if is an identifier, constant or something else. The second pair will indicate the place in the symbol table of the token. If it is not an identifier or constant the position will be marked with (-1,-1)

```
private List<Map.Entry<Map.Entry<String, Integer>, Map.Entry<Integer, Integer>>> pif = new ArrayList<>();
```

Lists of reserved words, operators and separators:

```
private List<String> reservedWords = Arrays.asList("intreg", "flotant", "sir_caracter", "caracter", "pentru", "cat_timp",
"daca", "altfel", citeste", "afiseaza", "fractional", "continua", "iesi", "caz");
private List<String> operators = Arrays.asList("+", "-", "*", "/", "%", "=", "==", "!=", "<", ">", "<=", ">=");
private List<String> separators = Arrays.asList("(", ")", ";", "{", "}", "[", "]", " ", "\t", "\n", ",");
```

Regex Patterns

Pattern for recognizing an identifier:

^[a-zA-Z]([a-z|A-Z|0-9|_])*$

Pattern for recognizing a

constant:

numeric: ^0|[+|-][1-9]([0-9])*|[1-9]([0-9])*|[+|-][1-9]([0-9])*\.([0-9])*|[1-9]([0-9])*\.([0-9])*$
character: [a-zA-Z0-9_?!#*./%+=<>;)(}{]
string: [a-zA-Z0-9_?!#*./%+=<>;)(}{]+

Scanner

The program file will be parsed line by line, each line being subjected to the tokenizefunction.
The tokenize function parses the line character by character.

```java
public ArrayList<String> tokenize(String line) {
    ArrayList<String> tokens = new ArrayList<>();

    for (int i = 0; i < line.length(); i++) {
        if (languageSpecification.isSeparator(String.valueOf(line.charAt(i)))
                && !(String.valueOf(line.charAt(i))).equals(" ")) {
            tokens.add(String.valueOf(line.charAt(i)));
        } else if (line.charAt(i) == '\"') {
            String constant = identifyStringConstant(line, i);
            tokens.add(constant);
            i += constant.length() - 1;
        } else if (line.charAt(i) == '\'') {
            String constant = identifyCharConstant(line, i);
            tokens.add(constant);
            i += constant.length() - 1;
        } else if (line.charAt(i) == '-') {
            String token = identifyMinusToken(line, i, tokens);
            tokens.add(token);
            i += token.length() - 1;
        } else if (line.charAt(i) == '+') {
            String token = identifyPlusToken(line, i, tokens);
            tokens.add(token);
            i += token.length() - 1;
        } else if (languageSpecification.isPartOfOperator(line.charAt(i))) {
            String operator = identifyOperator(line, i);
            tokens.add(operator);
            i += operator.length() - 1;
        } else if (line.charAt(i) != ' ') {
            String token = identifyToken(line, i);
            tokens.add(token);
            i += token.length() - 1;
        }
    }
    return tokens;
}
```

Each character will be checked if:

-is separator or “ ” → the token craeted so far will pe stored in the tokens lists

-is “ → the function that identifies a string constant will be called and token stored

```
public String identifyStringConstant(String line, int position) {
    StringBuilder constant = new StringBuilder();

    for (int i = position; i < line.length(); i++) {
        if ((languageSpecification.isSeparator(String.valueOf(line.charAt(i)))
                || languageSpecification.isOperator(String.valueOf(line.charAt(i))))
                && ((i == line.length() - 2 && line.charAt(i + 1) != '\"') || (i == line.length() - 1)))
            break;
        constant.append(line.charAt(i));
        if (line.charAt(i) == '\"' && i != position)
            break;
    }

    return constant.toString();
}
```

-is ‘ → the functions that identifies a char constant will be called and token stored

```
public String identifyCharConstant(String line, int position) {
    StringBuilder constant = new StringBuilder();

    for (int i = position; i < line.length(); i++) {
        if ((languageSpecification.isSeparator(String.valueOf(line.charAt(i))) ||
                languageSpecification.isOperator(String.valueOf(line.charAt(i))))
            && ((i == line.length() - 2 && line.charAt(i + 1) != '\'') ||
                (i == line.length() - 1)))
            break;
        constant.append(line.charAt(i));
        if (line.charAt(i) == '\'' && i != position)
            break;
    }

    return constant.toString();
}
```

-is - → the functions that decides if – is part of a number (stores the whole number astoken) or is an operator (stores just – as token) will be called

```
public String identifyMinusToken(String line, int position, ArrayList<String> tokens) {
    if (languageSpecification.isIdentifier(tokens.get(tokens.size() - 1)) ||
            languageSpecification.isConstant(tokens.get(tokens.size() - 1))) {
        return "-";
    }

    StringBuilder token = new StringBuilder();
    token.append('-');

    for (int i = position + 1; i < line.length() && (Character.isDigit(line.charAt(i)) || line.charAt(i) == '.'); i++) {
        token.append(line.charAt(i));
    }
```

```
    return token.toString();}
```

-is + → same as for -

-is operator → will check if it's part of an operator that is formed by multiple charactersand will store the token

```
public String identifyOperator(String line, int position) {
    StringBuilder operator = new StringBuilder();
    operator.append(line.charAt(position));
    operator.append(line.charAt(position + 1));

    if (languageSpecification.isOperator(operator.toString()))
        return operator.toString();

    return String.valueOf(line.charAt(position));
}
```

-is none of the above → will continue parsing the line until a separator/operator/emptyspace will be encountered and the result will be stored as a token

```
public String identifyToken(String line, int position) {
    StringBuilder token = new StringBuilder();

    for (int i = position; i < line.length()
            && !languageSpecification.isSeparator(String.valueOf(line.charAt(i)))
            && !languageSpecification.isPartOfOperator(line.charAt(i))
            && line.charAt(i) != ' '; i++) {
        token.append(line.charAt(i));
    }

    return token.toString();
}
```

After obtaining the list of tokens the symbol table and program internal form will be crested.First we store the token in the symbol table with the corresponding code (-1,0,1). Then we store it in the program internal form together with the position in the symbol table. Ifa token doesn't match any of the criteria an error message will be displayed.

```
public void constructPifAndSt(List<Map.Entry<String, Integer>> tokens) {
    List<String> invalidTokens = new ArrayList<>();
    boolean isLexicallyCorrect = true;

    for (Map.Entry<String, Integer> tokenPair : tokens) {
        String token = tokenPair.getKey();

        if (languageSpecification.isOperator(token) || languageSpecification.isReservedWord(token)
                || languageSpecification.isSeparator(token)) {
            programInternalForm.add(new AbstractMap.SimpleEntry<>(token, -1), new AbstractMap.SimpleEntry<>(-1, -
1));
        } else if (languageSpecification.isIdentifier(token)) {
            symbolTable.add(new AbstractMap.SimpleEntry<>(token, 0));
            Map.Entry<Integer, Integer> position = symbolTable.getPosition(token);
```

```
            programInternalForm.add(new AbstractMap.SimpleEntry<>(token, 0), position);
        } else if (languageSpecification.isConstant(token)) {
            symbolTable.add(new AbstractMap.SimpleEntry<>(token, 1));
            Map.Entry<Integer, Integer> position = symbolTable.getPosition(token);
            programInternalForm.add(new AbstractMap.SimpleEntry<>(token, 1), position);
        } else if (!invalidTokens.contains(token)) {
            invalidTokens.add(token);
            isLexicallyCorrect = false;
            System.out.println("Error at line " + tokenPair.getValue() + ": invalid token " + token);
        }
    }

    if (isLexicallyCorrect) {
        System.out.println("Program is lexically correct");
    }
}
```