1DV517 - Assignment 2

April 27, 2021

- Please submit your solutions in a single .pdf file in addition to commented code, your lexer, grammar and executable files with instructions regarding execution.
- Submit each file separately in uncompressed format.
- The deadline for submissions is 18 May 2019.
- Only typed solutions will be accepted. Please show the steps when solving the questions.
- *Reports that do not comply with the aforementioned requirements will not be considered.*

Question 1.(**10 points**) For each of the following languages, design a CFG that recognizes it:

- Strings that do not contain aa with $\Sigma = \{a, b\}$.
- All strings where there are at least n number of a's before the n-th b, for example it should accept ab or $aacbcacbab$. The alphabet is $\Sigma = \{a, b, c\}$.
- Odd-length strings whose first, middle, and last symbols are all the same over $\Sigma = \{a, b, c\}$.
- $L = \{uwxx^{-1} \mid |u| > |w|\}$ where w^{-1} is the inverse of w, $|u|$ is the length of u, and the alphabet is $\Sigma = \{a, b, c\}$.

Question 2.(**4 points**) Let L be the language produced by the following grammar. Show that the strings of this language contain no substring 001:

$$S \to \epsilon \mid S0 \mid 1S \mid 01S$$

Note: You should show that none of the derivations contain 001 in a semi-formal way.

Question 3. (**10 points**) Design a Turing machine that checks whether a string belongs to the language of following grammar or not where $\Sigma = \{a, b\}$.

$$S \to bA \mid a$$
$$A \to aS \mid bA \mid \epsilon$$

Informally explain how your machine works.

Question 4. **(10 points)**

- Demonstrate that the following grammar with the start symbol S and $\Sigma = \{a, b, c\}$ is ambiguous.:

$$S \to aS \mid ASc$$
$$A \to bB \mid \epsilon$$
$$B \to ASa$$

- Explain the reasons of ambiguity.

- Remove all of its ambiguities.

Question 5.**(16 points)** In this task, you will implement a simple compiler from XML to JSON files.

XML Specification

The XML language used for this exercsise is as follows:

- XML files consist of the following components: *elements*, *tags*, *attributes* and *plain old data(POD)*
- A *POD* can take three forms: a *number*, a *string* of characters or a *macro*.
- A *number* is represented by an optional sign, (*'+'* or *'-'*) and a sequence of decimal digits (tex'0'-*'9'*), followed optionally by *'.'* and another sequence of decimal digits.
- A *string* is a sequence of alphanumeric characters, which include *'a'-'z'*, *'A'-'Z'* and *'0'-'9'*, but must start with a non-digit.
- a *macro* starts with *'!'*, is followed by either *'+'* or *'-'* representing the macro operation and ends with a string that shows the elements to apply that operation on.
- An *attribute* comprises a *string*, corresponding to its name, followed by '=' and a *POD* inside double quotes *'"'* that is its value. Example attributes include: `name = "Harry"`, `year = "2021"`, `PI = "3.1415"`.
- *Tags* can be either a starting tag or an ending tag. A starting tag that starts with *'<'*, is followed by a *string* and optionally a number of *attributes*, and terminates with *'>'* . An ending tag starts with "</", is followed by a string, and concludes with '>' . Examples of tags include: `<Person Name="Harry">`, and `</Person>`.
- *Elements* begin with a *starting tag*, are followed by their *content*, which can be other *elements*, or *PODs*, and end with a corresponding *closing tag*. An example element is:

```
<Person Name="Harry">
<Year month="05">2021</Year>
<Employment>
```

```
<Position> Doctoral Candidate </Position>
<Year started="10">2017</Year>
</Employment>
<YearsEmployed> !- Year </YearsEmployed>
</Person>
```

The target JSON grammar, which you need to transform your XML input to while evaluating XML macros, consists of: objects, keys and values.

- A *key* is a quoted *string*.
- A *value* is a *POD*, where macros have been replaced with their evaluated values and all strings are quoted. Example values are: -1, +133.7, "I am a quoted string".
- *Objects* start with a *key*, followed by *':'* and *'{'*, then follows the object content, and the object ends with a *'}'*. The object content is a sequence of comma separated *objects* or *key-values*, where a key-value consists of a *key* followed by *':'* and a *value*. An Example object is:

```
"Person":{
"Name" : "Harry",
"Year" : { "today" : "23-05", "POD" : 2021 },
"Employment" : {
"Position" : "Doctoral Candidate",
"Year" : { started : "10", "POD" : 2017 },
},
"YearsEmployed" : 4
}
```

Macro Operations

Two types of macros exist and can only be used with number POD values:

- Sum macros, starting with "!+", that sum up the values of all PODs of elements of their last argument.
- Difference starting with "!-" which will take the running difference of all PODs of elements specified by the last argument.

Macros bind to their surrounding element, i.e., they only consider the values of their sibling elements. In the following example, !+ num evaluates to 3 and !- num evaluates to 1.

```
<equation>
<num>1</num> <num>2</num> <sum>!+ num</sum>
</equation>

<equation>
<num>3</num> <num>2</num> <diff>!- num</diff>
</equation
```

- Give two CFGs to express the XML and JSON specifications.
- Describe how you map each XML element to a JSON element and how your macro evaluation works.
- Implement the translator from XML to JSON files in a language of your choice.

Notes/Tips:

- You are free to provide an ANTLR based parser and translator or implement your own recursive descent parser and associated translator.
- There are multiple ways to implement the evaluation. The simplest is possibly to keep a hash map of POD values for the children of each element that have number PODs.
- You can find the example input.xml and output.json files alongside the assignment.