02141 Computer Science Modelling
Mandatory Assignment on Context Free Languages:

–Due: 2017-April-27–

Feedback starts: 2017-April-28 Feedback Due: 2016-May-02

Both the assignment and the feedback are required exercises.

1 Part I: A simple Programming Language

This part of the mandatory assignment is about designing a small programming language called *while* and an interpreter for it. We provide you a running parser and interpreter that works for a subset of the language and you will be asked to extend it. On file sharing you find a zip archive that contains already most of the interpreter and you can do this assignment simply by extending these files. You are however free to make your own extensions and change the *concrete syntax* of the language to your personal taste. We only require that on the abstract syntax level your language supports all features that the handed-out version supports and all the extensions that this assignment asks for.

Overview of the files in the assignment archive on campusnet:

- `antlr-3.5.2-complete.jar` is the used version of ANTLR that we use; consult the file
- `Parsing.pdf` for installing and a small tutorial on ANTLR.
- There is a folder `Calculator` which contains the first example from the lecture, where the input program can be only an arithmetic expression without variables. In this example, the actions in the grammar describe directly the evaluation of the given expression, without generating abstract syntax.
- The folder `while_language` contains the entire source codes of the *while* language that we give you. On the top level you find
  - `WhileLanguage.java` – the main file of the interpreter, with two other Java files
  - Folder `parsing` contains initially only the ANTLR grammar `WhileLanguage.g` for our language.
  - Folder `ast` contains all files for the abstract syntax tree. Every class in the abstract syntax tree has a method `void evaluate(Environment)` where the `Environment` maps every variable (that has been assigned to so far) to a concrete integer value. This `evaluate` function in fact defines the *semantics* (meaning) of the language. When you design new elements of the abstract syntax, you may leave the `evaluate` function empty, but probably you have an idea what would be an appropriate evaluation function!

- On the top level there are two programs, `program1.while` and `program2.while`. The first already works with the interpreter we give you, the second one requires features you implement in this assignment. There are also output files that contain the expected output of these programs.

The first task is to install ANTLR and compile the interpreter for our language. You can do this in Eclipse or on a command line. This involves to first run ANTLR on the grammar in the `parsing` folder. Then all java files should be compiled; for that you should be in the folder above `while_language`. This produces the `WhileLanguage.class` that you can run with java and the program files as input.

Question 1 *Consider either the given* ***`WhileLanguage.g`*** *or your modification of it (please include in the submission). Give a context-free grammar for this language: extract from the ANTLR grammar the syntactic rules and present them in standard notation of a context-free grammar. Due to ANTLR's restriction that rules cannot be left-recursive, the ANTLR grammar is more complicated than it has to be; try to give a context-free grammar that is as simple as possible without introducing ambiguities.*

Question 2 *Make the following extensions to your language:*

- *In* `base_bool_expr` *we currently allow checking equalities* `e1=e2` *between arithmetic expressions. Add also comparison* `e1<=e2`*. This requires a new class in the abstract syntax, similar to* `EqualsExpr`*.*

- *In the same rule, we also add* `e1!=e2`*. Can you implement this without a new abstract syntax class? (Hint: the abstract syntax for boolean expressions contains negation.)*

Please clearly mark all modifications you have made to the grammar for this extension. Also please provide an example program in your language that showcases the extensions.

Question 3 *Extend* `statement` *to also include while loops. A while loop consists of a condition (i.e. a boolean expression) and a statement (see for instance* `program2.while`*). Again you must define a new class in the abstract syntax for this construct. Hint: get inspired by how we implement the* `if` *construct, both in the concrete syntax and the abstract syntax.*

Again, please clearly mark all modifications you have made to the grammar for this extension. Also please provide an example program in your language that showcases the extensions.

When you have finished all questions up to here, also the second example program should work (with appropriate modifications if you changed the concrete syntax of your language, of course).

Question 4 *As operators on booleans, the language contains so far conjunction (logical* and*, written* `&&` *) and* negation *(logical* not*, written* `!` *). We now want also disjunction (logical* or*, written* `||` *). Here we want that the negation binds strongest, then conjunction, and disjunction binds the weakest. For instance,* `b1 || ! b2 && b3` *is parsed like* `b1 || ((!b2) && b3)` *.*

Extend your grammar appropriately and give a short argument why your extension is correct. Implement the extension and develop a while program that can be used to test the new feature. Hint: get inspired by how we handle the priority of multiplication over addition in arithmetic expressions.

Again, please clearly mark all modifications you have made to the grammar for this extension. Also please provide an example program in your language that showcases the extensions.

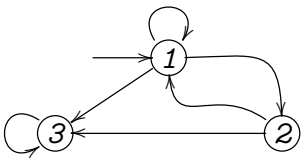
2 Part II: Parsing for a Model Checker

This part of the mandatory assignment is about building the parser of a model checker, like the one you developed in Mandatory Assignment 2.

Question 5 *Design a context-free grammar for transition systems as seen in Mandatory Assignment 2 and implement the grammar in ANTLR. The grammar should accept transition systems denoted as regular expressions of the form*

$$(\ '\{'\ S\ ('*')?\ '['\ P*\ ']'\ S*\ '\}'\ )*$$

where S is a natural number representing a state and P is a lower case string representing an atomic proposition. As an example, the transition system



where states 1, 2 and 3 are respectively labelled with $\{p\}$, $\{q, r\}$ and $\emptyset$, is denoted by

```
{ 1* [ p ] 1 2 3} { 2 [ q , r ] 1 3} { 3 [  ] 3}
```

Provide a context-free grammar for transition systems based on the above syntax and implement it in ANTLR. If you had to transform the grammar to be accepted by ANTLR then explain the changes. Also, provide at least 3 representative examples of transition systems that your grammar admits. NOTE: You can of course use regular expressions in your grammar. Bear in the next question when designing your grammar.

Question 6 *Decorate the grammar with code to actually build a transition system as implemented in Mandatory Assignment 2. If you implemented your model checker in Java (or any other language supported by ANTLR) try to actually integrate the generated code in your model checker, so that it can directly import transition systems from files. For example, if you developed a data structure class* `TransitionSystem` *for transition systems, the initial symbol of your ANTLR grammar (say* `system`*) could return an object of class* `TransitionSystem` *and then your model checker would use the parser* `parserTS` *as follows:*

```
TransitionSystem myTS = parserTS.system();
```

Note that in this case your class `TS` *would be the abstract syntax!*

If you did not use a language supported by ANTLR just provide a sketch of how you would do it.

Question 7 *Implement an ANTLR grammar for the fragment of CTL provided in Mandatory Assignment 2, which we re-propose here in BNF format and using unicode symbols instead of mathematical symbols:*

$$\Phi\ ::=\ tt \mid AP \mid \Phi_1\ and\ \Phi_2 \mid not\ \Phi \mid EX[\Phi] \mid EF[\Phi] \mid AX[\Phi] \mid AG[\Phi]$$

As in question 5, you should provide a context-free grammar using the notation seen in class and explain whether and how you had to transform it to be accepted by ANTLR. Also, provide at least 3 representative examples of formulae that your grammar admits.

Question 8 *Decorate the grammar of CTL with code so that it builds a CTL formula as implemented in Mandatory Assignment 2. If you implemented your model checker in Java (or any other language supported by ANTLR) try to actually integrate the generated code in your model checker, so that it can directly import CTL formulae from files. If you did not use language supported by ANTLR just provide a sketch of how you would do it.*

Consider the following as a sketch of a possible solution for the sake of inspiration. You could design a set of Java classes for the abstract syntax of formulae (you can get inspired by arithmetic and boolean expressions in the while language) unless you already did it in Mandatory Assignment 2. For example, you could define an abstract super class `Formula` *for storing formulae, and some other concrete classes for all kinds of formule. For example, a class* `EXFormula` *with a* `Formula subformula` *attribute could be useful to store formula of the form* $EX[\Phi]$*. The initial symbol of your grammar (say* `formula`*) should then return a value of class* `Formula`*. Your model checker would then use the parser* `parserF` *as follows:*

```
Formula myFormula = parserF.formula();
```

You could then add a set of overloaded methods `check` *to your class of transition systems specialised for each of the concrete classes of* `Formula`*. For example, using the notation suggested in Mandatory Assignment 2 you would implement the method for checking formulae like* $EX[\Phi]$ *as follows:*

```
public ... check(EXFormula F){
    return ctlEX (F.subformula);
}
```

to invoke the method `ctlEX` *of the transition system.*

Note that all the pieces developed in the above questions 5–8 can be easily integrated so that your model checker can actually read transition systems and formulas from files!

Submission

- You are allowed to submit as groups of up to three people.
- Please submit a report that contains an answer (1–2 paragraphs) to each question that also explains what you have thought. It may contain essential parts of your solutions, e.g. some lines of source code.
- Also please submit all your ANTLR and Java source codes as an archive with its directory structure, so that others can easily compile your solution.