Parser Coursework

CM50260 Foundations of Computation 2021-22

- **Deadline:** 8pm on Monday 10 January 2022.
- **Structure:** This assignment consists of four parts (A–D) worth 100 marks in total. These are detailed in the main part of the document below.
- **Submission:** You should submit your coursework on Moodle. Details of what to include are given at the end of this document.
- **Weight:** This coursework is worth 40% of your overall grade for the unit.
- **Plagiarism:** The coursework is an individual assignment. General discussion at a conceptual level is allowed, for example in the tutorials or Q&A forum, but the work you submit should be your own. Disclosing your solutions to others, and using other people's solutions in your work, both constitute plagiarism. See `https://www.bath.ac.uk/publications/qa53-examination-and-assessment-offences/`.

Overview

A parser is a key component of most compilers. Its job is to analyse a string of symbols, which includes

(1) checking that they form a syntactically correct expression,

(2) converting them to a suitable data structure.

The syntax of programming languages can often be described by context-free grammars G. In this case, (1) above can viewed as deciding whether or not a given input string w belongs to the language $L(G)$, while (2) could be interpreted as building a parse tree for w. In this assignment, your challenge will be to implement a parser for context-free languages. The assignment is broken down into the following parts:

A. You will be given a concrete context-free grammar for arithmetic expressions which you must convert to Chomsky normal form.

B. You will implement your converted grammar, so that it can be used to run tests.

C. You will implement a general algorithm for deciding whether or not an input string w is generated by an arbitrary input grammar in Chomsky normal form.

D. You will modify your algorithm from C so that rather than a simple "yes" or "no" it returns a parse tree whenever w is generated by the grammar.

Background knowledge

Context-free grammars

For this assignment, it is essential that you understand the concept of a context-free grammar (CFG), together with the procedure for converting any CFG into an equivalent grammar in Chomsky normal form. The basic theory of CFGs was presented in Lecture 10 during Week 5, while the conversion to Chomsky normal form algorithm will be explained via prerecorded videos along with a pdf summary.

Java

The parser should be implemented in Java, and therefore a basic understanding of Java is essential for the assignment. You will be taught Java in the Principles of Programming unit during the second half of the semester. However, it's important to note that this coursework is primarily designed to give you the opportunity to **apply your theoretical knowledge in a more practical environment**. We are not testing your abilities and sophistication in programming. We provide you with an extensive library of helper code in order to assist you with your implementation.

A context-free grammar for arithmetic expressions

The following grammar $G = (\Sigma, V, R, S)$ generates a language of syntactically correct bracketed arithmetic expressions involving addition and multiplication over literals 0, 1 and x. The terminals Σ and variables V are given by

$$\Sigma := \{+, *, (,), 1, 0, x\} \quad \text{and} \quad V := \{S, T, F\}$$

where S is our start symbol, and the rules R of the grammar are given by

$$\begin{aligned} S &\to S + T \mid T \\ T &\to T * F \mid F \\ F &\to (S) \mid 1 \mid 0 \mid x \end{aligned}$$

Examples of words which belong to the generated language $L(G)$ include:

$$(x) \qquad 1 + x \qquad x * 0 \qquad (x + 0) * 1$$

ASSIGNMENT PART A (10 MARKS)

Convert the grammar G above to an equivalent grammar G_0 in Chomsky normal form. There is no programming involved here, and you should submit your solution, including details of your working, in pdf form, either typeset or as a neatly handwritten scan. You might find the document *Converting a grammar to Chomsky normal form* helpful.

The skeleton code

The remainder of the assignment will involve filling out some skeleton Java code with your own code. The skeleton code is contained in a zip file which can be downloaded from your Moodle page by clicking *Skeleton code: Download*. The skeleton code contains the following key classes:

- `MyGrammar.java`, an empty source file in which you will implement your context-free grammar.
- `Parser.java` which implements the interface `IParser.java`, and which contains two empty methods
  - `isInLanguage(ContextFreeGrammar cfg, Word w)`
  - `generateParseTree(ContextFreeGrammar cfg, Word w)`

  that will eventually be filled in by you. You will need to include any additional methods you define within this source file.
- `Main.java`, a script with which you can demo some features of the skeleton code and run tests on your own programs.

In addition, you will find an extensive library of auxiliary classes which are there to help you implement your parser, together with full documentation. To access the documentation you should navigate to the folder named 'doc' and open `index.html`, or alternatively follow the *Skeleton code: Documentation* link on Moodle. To help get to grips with the skeleton code, you should view the video *Getting Started with the code*. Note that this video - created by Andrew Chinery - refers to a slightly different platform and grammar, but uses essentially the same code.

ASSIGNMENT PART B (10 MARKS)

Take your converted grammar G_0 in normal form from Part A and implement this in the skeleton code by filling in the relevant part of `MyGrammar.java`. Note that the auxiliary files contain methods that allow you to print your grammar, which is useful for testing your implementation. One way to print is to simply run `Main.java` with custom code (Option 3), adding the line

```
System.out.println(cfg);
```

to the `customCode()` method.

ASSIGNMENT PART C (50 MARKS)

Suppose that G is now some *arbitrary* context-free grammar in Chomsky normal form. If $w \in L(G)$ has length $n \geq 1$ then any derivation of w from the start variable must have exactly $2n - 1$ steps. This observation leads us to an algorithm for solving the following decision problem:

- **Input:** A context-free grammar G in Chomsky normal form together with a word w
- **Output:** A boolean value indicating whether or not $w \in L(G)$

The algorithm works as follows:

1. List all derivations in G with $2n-1$ steps, where n is the length of w, unless $n = 0$, in which case list all derivations with one step.
2. If any of these derivations generate w, return `true`. If not, return `false`.

In this part of the assignment, you should implement this algorithm by filling out the following method of the `Parser.java` source file

```
boolean isInLanguage(ContextFreeGrammar cfg, Word w)
```

so that it now returns `true` if and only if the input word `w` is generated by input grammar `cfg`, which we assume is in Chomsky normal form. Any auxiliary methods you create should be included as part of the `Parser.java` file.

ASSIGNMENT PART D (30 MARKS)

Adapt your code so that it now returns parse trees in the case that words are generated by the grammar. More precisely, you should fill out the following method of the `Parser.java` source file

```
ParseTreeNode generateParseTree(ContextFreeGrammar cfg, Word w)
```

so that it returns a parse tree whenever the input word `w` has a derivation in the input grammar `cfg`, and just `null` otherwise.

In your implementation of the main algorithm in Part C, you should find some way of keeping track of the entire derivation from start variable to the final string. The `Derivation` class is there to help you do this. For Part D, you could then write a procedure which converts a valid derivation into a parse tree, working backwards through the steps you took. Your parse tree will be represented as an object of the `ParseTreeNode` class.

In building your parse tree, you should work from the bottom of the tree upwards. First of all create a number of ParseTreeNode objects – one for each of the final terminals in the parse tree. You will also need to find a way to keep track of these objects. Then, by following the steps of the full derivation backwards, you can work out which ParseTreeNode objects need to be combined into new ParseTreeNodes. Eventually you'll reach the start variable of the grammar, and this will be your final ParseTreeNode object.

Note that iteration over objects of the `Derivation` class runs backwards by default (see the documentation for more details). To illustrate this, you could experiment with the following method:

```
public void printDerivation(Derivation d) {
   for(Step s : d) {
      System.out.println(s);
   }
}
```

which prints out all steps in the input derivation.

Note on testing and marking for Parts C and D

The code you submit for Parts C and D will be subject to automatic code testing on different context-free grammars. You are expected to check that your program compiles and works correctly by testing it using the demo script.

It is important to emphasise that we will test your code by simply inserting your submitted file `Parser.java` into our script. This means that if you made any changes to the helper code, or used any additional source files, there is a risk your submission will not compile when we run it.

If you use your own choice of IDE to develop your code and are concerned that it may not run on other machines, a good test would be to follow the repl.it link we provide, where you can copy-and-paste your code and run it within the repl.it environment.

Above all, please double check that your submission compiles **directly before you upload it** and after making any last minute changes!

- If you run tests on your user defined grammar G_0 which you implemented in Part B, you may find that your program encounters memory issues for input words of length 4 or greater, depending on the data structure you use. This is not necessarily a sign that your program is incorrect: The algorithm described in Part C is essentially a "brute force" search and is therefore inefficient. There exist more sophisticated algorithms for parsing context-free grammars, but those are beyond the scope of this assignment.

What to submit

Your submission should be a .zip file which includes the following:

1. A pdf file named `PartA.pdf` containing your solution to Part A.
2. Your completed `MyGrammar.java` file containing your solution to Part B.
3. Your completed `Parser.java` file containing your solutions for Parts C and D.

   To repeat: any auxiliary methods you create must also be contained within your submitted code, as we will simply be replacing the default `Parser.java` file in the skeleton code with your completed version. Please refrain from submitting additional source code.

GOOD LUCK!