

PARSERS IN JAVA

Introduction

This lab will give you practice on hand-coding an LL(1) parser for arithmetic expressions and introduce the ANTLR tool you will use in coursework for automatically generating scanners and parsers.

Setup

Download the zip file from Moodle for this lab. It contains a basic Gradle project to get you started, along with an implementation of a simple token type your parser can use and a main function to test it.

Grammar

This lab uses the grammar shown in Figure 1 below. It is a simplified representation of the Lisp programming language, which is built around **s-expressions**. An s-expression is either a single **atom** (number or identifier), or a sequence of s-expressions contained in parentheses. For our purposes, a number is one or more digits, and an identifier is a lowercase letter followed by any number of lowercase letters or digits. Remember that for a programming language the terminal symbols in a grammar are tokens, not literal values.

Hand-coded LL(1) parser

You should construct an LL(1) parser for the grammar in Figure 1 following the method in the lecture notes. Construct it on paper first by computing the *first*, *follow*, and *start* sets and building the parse table, then consider how to realise this in Java.

Task: Design the parse table on paper, or in Excel. Remember that a real-world parser receives **tokens** as input, not necessarily individual characters.

Task: Create a class called `SexprParser`. At minimum it will need the following:

- a parse method that takes a list of tokens as its only argument and returns a Boolean indicating if the parse was successful;
- some way of maintaining the parse state (but you might choose to keep this local to the parse method); and
- a parse table which the parse method can refer to when updating its state.

The Map interface (`java.util.Map`) is the common interface for all classes in Java that can store key-value pairs. Look up its documentation and choose a suitable implementation to use for your parse table.

$\begin{aligned} SExpr &\rightarrow Atom \\ &\quad List \\ \\ Atom &\rightarrow \underline{number} \\ &\quad \underline{identifier} \\ \\ List &\rightarrow (ListContent) \\ \\ ListContent &\rightarrow SExpr ListContent \\ &\quad \epsilon \end{aligned}$
--

FIGURE 1 – GRAMMAR OF S-EXPRESSIONS

Task: Decide on the data structures you will need to represent the right-hand side of your productions. In your class's constructor, populate the parse table with all the rules.

Task: Implement the parse method, using the pseudocode in the lecture slides as a guide. Check your implementation by running the provided main method.

ANTLR4 parser

ANTLR is a scanner and parser generator. It is written in Java but can produce code for scanners and parsers in many languages. There are two ways of using it in your projects: download the tool and run it manually or incorporate it into the build process of a Gradle project. We will do the latter.

The reading list contains a link to *The Definitive ANTLR4 Reference* which you should refer to at any time if you get stuck and the lab team can't help (or if you don't want to talk to them).

Configuring ANTLR

We will add ANTLR using the Gradle build file. There are two lines to add: a plugin, and a dependency.

Task: In the plugins section of `build.gradle.kts` add a line that just reads `antlr`. The ANTLR plugin tells Gradle how to handle grammar files.

Task: In the dependencies section of `build.gradle.kts`, add the following line:

```
antlr("org.antlr:antlr4:4.13.2")
```

Note that the function name is `antlr`, not `implementation`. Also note the quote marks in this document have automatically been replaced with curved ones, you will need to fix this if you just copy and paste.

Task: At the bottom of `build.gradle.kts` add the following block of code to configure Gradle:

```
tasks.generateGrammarSource {  
    arguments = arguments + listOf("-package", "uk.ac.nott.cs.comp3012")  
}
```

Again, watch the quote marks if you copy and paste. This configuration specifies the package which the classes ANTLR generates should belong to.

Task: Reload the Gradle project so that Gradle downloads the new dependencies.

Creating grammar files

Gradle expects to find ANTLR grammar files in a particular place in the project. It looks for Java source files in `src/main/java`, and it looks for grammar files in `src/main/antlr`.

Task: Create a folder named `antlr` inside `src/main`.

We want our generated grammar files to belong to the same package as our source, so we need to replicate the folder structure of our Java directory inside the ANTLR directory. Classes that belong to the package `foo.bar.me` would live in the directory `foo/bar/me`.

Task: Inside the ANTLR directory, create the following directory hierarchy:
`uk/ac/nott/cs/comp3012`.

Finally, we can create our grammar! ANTLR4 grammar files use the file extension .g4 and the name of the file must match the grammar declaration at the start of the grammar.

Task: Right click on the comp3012 directory you just created and create a new file called SexprAntlr.g4. In the first line of the file, write grammar SexprAntlr; (including the semicolon).

ANTLR syntax

Each rule begins with the name of the symbol, followed by a colon, followed by the symbol definition. The terminal symbol NUMBER is defined using a regular expression and referred to by name. The terminal representing the plus sign is written directly.

```
add: NUMBER '+' add
    | NUMBER
    ;
NUMBER: [0-9]+ ;
```

FIGURE 2 – ANTLR GRAMMAR FOR ADDITION

The rule for add has two alternatives separated by a pipe symbol just as in the notation on paper. Each rule ends with a semicolon. Note that all the parser rules must appear before any scanner rules.

Task: Translate the s-expression grammar from Figure 1 into ANTLR syntax. Remember that in ANTLR, nonterminals must **not** begin with an uppercase letter.