IN2009 Coursework Part 2 (25% of module mark)

This is Part 2 of a single coursework assignment which is delivered in two equally weighted parts. The coursework as a whole is worth 50% of the module mark.

Use of AI Tools

Use of AI tools in developing your solutions is NOT permitted.

The CourseworkPart 2 IntelliJ Project

You will find the zip file in the Coursework Part 2 assignment in Moodle. The AST classes and the LPLParser class in this project are identical to the ones that were provided for Part1. **You should replace LPLParser and the AST classes by your completed versions from Part 1**. You will need to edit the IntelliJ Terminal CLASSPATH configuration setting in the usual way, so that it is correct for your local set up. Both the SBNF and the SSM libraries are required by this project. You should have already installed the relevant Junit libraries from Maven while doing Part 1 of the coursework (if not, follow the instructions below).

The Task

This part of the coursework asks you to extend your parser and compiler from Part 1 to a more complete version of the LPL language. As for Part 1, you are only permitted to use SSM.jar, SBNF.jar, the standard Java libraries, and the Junit libraries.

In addition to the features already supported by the restricted version of LPL which you implemented for Part 1, the complete LPL language also supports methods and arrays. You will need to make various changes:

- Recall from Week 7 that global and local variables are compiled in very different ways. Some of your existing compile methods will need to be modified and you will need to modify the SymbolTable class to allow your compile methods to determine whether they are compiling part of the main program body or part of a method definition, and to distinguish between global variables, method parameters and local variables. The details are for you to decide.
- You will need to extend and modify your parser code to handle the extended LPL grammar.
- You will need to add new AST classes and modify some existing AST classes, in order to be able to build an AST for LPL programs which use the new features. The details are for you to decide. (If you want to be able to pretty-print your new ASTs, you will also need to extend the Visitor interface and define appropriate `accept` methods in your AST classes. However this is unassessed and entirely optional.)

The complete LPL grammar is provided in file `data/LPL-non_LL1.sbnf`. An LL(1) grammar which generates the same language is provided in `LPL-C.sbnf`. The LL(1) grammar in `LPL-A.sbnf` is the same grammar that you used for Part 1 and is included here only for reference. The LL(1) grammar in

`LPL-B.sbnf` is for an intermediate version of LPL which supports methods but not arrays. (File `data/LPL.sbnf` contains just the token definitions and is included for backwards compatibility with the use of constant `LPL_SBNF_FILE` in LPLParser.)

Assessment

Your submission will be graded entirely on the basis of automated testing. The more tests that your code passes, the higher the mark awarded. The marking is divided into three parts.

1) **[30 marks]** for a parser which deals correctly with LPL programs which include method definitions and method calls but which do not include any use of arrays. Your mark for this part will be in proportion to the number of `parseB` tests that your submission passes (see `LPLParserTest` in the test package). To obtain full marks for just this part it is sufficient to implement a parser which is correct for the intermediate grammar in `data/LPL-B.sbnf`. Note that the parser tests do **not** examine the AST which your parser builds (in fact, you could simply return a null AST and still obtain these marks).
2) **[40 marks]** for a compiler (the combination of a parser which builds ASTs and the AST compile methods) which generates correctly functioning SSM code for LPL programs which include method definitions and method calls but which do not include any use of arrays. Your mark for this part will be in proportion to the number of `compileB` tests that your submission passes (see `LPLCompilerTest` in the test package).
3) **[30 marks]** for a parser which deals correctly with LPL programs which include use of arrays. Your mark for this part will be in proportion to the number of `parseC` tests that your submission passes (see `LPLParserTest` in the test package). As for the `parseB` tests, these tests do **not** examine the AST which your parser builds.

Optional Extensions

The following tasks extend the above work to achieve a complete LPL implementation. They are not assessed:

- Implement a complete compiler which generates SSM assembly code for all LPL programs, including those which use arrays. You can test your code using `compileC` in `LPLCompilerTest` (these tests are not comprehensive). You will need to devise some scheme for allocating memory in the heap, to be used when new arrays are created (no need to be overly ambitious here; a simple scheme which progressively increments some kind of "next free" pointer is sufficient for proof-of-concept). The trickiest part of this is initialising all new array elements to zero/null (**Tip**: You can write a simple LPL function for initializing arrays then bootstrap your compiler to compile it into SSM assembly. You will probably need to hand-edit the resulting code a little, but it's a lot easier than hand-writing assembly from scratch.)
- Implement a type-checker. A stub implementation has been provided in the `staticanalysis` package. Note that it uses the Visitor pattern. An alternative would be to add type-check methods to the AST classes, but using the Visitor pattern is better software-engineering, since it separates the definition of the essential AST structure from the details of any particular choice of tree-

processing. (The same comment applies to code-generation; using the Visitor pattern would be better software-engineering than defining compile methods in the AST classes). If you do use the Visitor pattern, then the visit method for each descendant of Exp should either throw a StaticAnalysisException or return the type of the expression; all other visit methods should either throw a StaticAnalysisException or return null. You may need to make additional changes to your SymbolTable class in order to support type-checking. You can test your type-checker using `LPLTypeCheckerTest` (these tests are not comprehensive).

Submission

Submit your work on Moodle as a single Zip archive of your entire project folder. (If you have converted the project to work in some IDE other than IntelliJ, please notify me in advance: s.hunt@city.ac.uk.) The Zip archive must be in a format which is recognised and can be unpacked by standard archiving software (e.g. 7Zip).

Please note that you **DO NOT** need to include a copy of the SSM and SBNF libraries with your submission.

Extenuating Circumstances: Applications for extensions must be made using the usual Extenuating Circumstances procedure at the first available opportunity (forms available from Programmes Office and online). **If your application is approved**, you may be given up to a 1 week extension. In this case, please submit your work in the relevant "EC Late Submissions" area in Moodle (this will only become available once the normal deadline has passed).

Please note the following:

- Applications for EC extensions are managed by the admin team, not by the module leader. Contact **ug.cs@city.ac.uk** if you have any questions.
- You are encouraged to deal with your extenuating circumstances (sickness, family emergency) as your priority. However, if you are capable, consider submitting something (even incomplete work) by the regular deadline as a contingency plan (in case your EC application is not approved).
- EC submissions can be at most one week late. Sometimes EC cases can take longer than this to resolve (if for example, required evidence was not provided or was insufficient):
    - Should your case take longer than the allowed submission timeframe, submit your work within the timeframe even if you haven't received the EC decision yet.
    - Should your case take too long for timely assessment to be practical, then you will unfortunately need to resit in August.

Testing

The project includes support for Junit tests in the `test` package. This code depends on the files in `data/tests`. **You will need to install the necessary JUnit library support from Maven.**

Installing the Junit Libraries from Maven

IntelliJ: *File* => *Project Structure*

1. Under *Platform Settings*, select *Global Libraries*
2. Top of the 2nd column (**not** the third) click **+**
3. Select *From Maven...*
4. Copy-paste the following into the Maven coordinates field:

   `org.junit.jupiter:junit-jupiter:5.9.3`
5. Check the *Download* check-box and *select a location on your local machine*
6. Click all the necessary OK buttons.