Compilers - CSCI 468 - Spring 2016

Kevin Hainsworth, Jonathan Johnson, and Josh McCleary

May 3, 2016

1 Program

The listing of the program is attached below. The source code for the compiler program includes the following: Main class, ExtractInterfaceListener class, IRNode class, TinyCode class, Little.g4 grammar file, Little token file, LittleBaseListener class, LittleListener class, LittleLexer class, LittleLexer token file, and LittleParser class.

2 Teamwork

The team worked on this project during collaborative team meetings and their own spare time. The team spent approximately 450 hours of combined time (about 6 total days of work per team member) to the project. Team member 1 contribution to the team consist of providing the codes for step 1, step 2, and eighty percent of step 4. Member 1 also provided the technical writing to on the step 2 and step 3 while drafted the major difficulties of the step 4 paper. Member 1 provided the writings for section 3.5 of the report including the step 2, step 3, and part of step 4. Member 1 contributed an overall 40 percent of the time into the project. Team member 2's contributions to the project include aiding with code design throughout all parts of the project, worked out most of the bugs found throughout the project, performed the majority of communications with the professor and any other outside assistance, and attributed the introduction, background, step one, parts of step two, three, four, and the full fledged compiler, and the conclusion to the report. Member 2 contributed an overall 33 percent of the time. Team member 3's contribution included: writing and debugging the structure and rules for the Little grammar, writing and debugging several methods required for steps 2 and 3 of the project, debugging and modifying code and adding the symbol table to be used in step 4 of the project, final editing and contributions to the final report.

3 Design Pattern

The team employed the builder creational design pattern in a section of this project. It was used to help the program organize the complex functionality and organization of the various compiler phases. The builder pattern was used primarily in the generation of the step three symbol table and the step four semantic actions components of the compiler. Both used classes from the base listener generated by the ANTLR parser generator. This design allowed the program to share the same resources while maintaining its own functionality for each step. In this case, the step three symbol table can walk through the syntax tree and generate the necessary print statements while the step four semantic action can see and walk the same syntax tree while deciphering the required node of the tree for further analysis and computation. This can be seen in our main method code where the ExtractInterfaceListener walks the tree and the TinyCode walks the tree.

4 Technical Writing

Our technical report is presented with this document.

5 UML

Image of the UML diagram is attached at the end of this document.

6 Design Trade-Offs

During the implementation of our project, we made several design decisions that affected time performance and size of our program. During this process, we generally chose to implement code in a way that was easy to understand for us and would be just as easy to implement. For example, when creating the symbol table for step 3 of the project, we chose to implement this using a HashMap data structure. We used strings to store variable names and types so it was

easily readable. Choosing to use integers may have been more efficient space-wise, but would not have been as easy to implement or as readable. However, using the HashMap data type was a positive choice for our running time of the program as searching through a hash is algorithmically constant time. If we had used a different structure, such as a linked list, we would be running slightly slower. In the future, we would like to analyze our code to increase size and time performance, if possible.

7 Software Development Life Cycle Model

During the capstone project development, the Rapid Application Development (RAD) model with prototyping was used. With each portion of progress being made for the compiler development, the initial group meeting functioned much more like techniques describe in RAD. After the initial meeting, the program turned into a prototyping life cycle model. Every time the team met during the discussion of the design, work load distribution, and method approach for each project step the team checked the requirement for the step and designed the approach for programming the functionality. Parallelizing the work load and the production of the program proved to be difficult since one team member would work on one functionality of the program while the rest will try to debug one's original design or the other will create the next step for the program. This course of action clearly replicates the RAD development because the team is primarily focused on building the code for each step and mending the needed maintenance for fixing the bugs that arose and generating the necessary classes and function to complete the project step. The RAD model helped the team to program the parts needed to complete each of the program steps and the team was able to provide additional solutions when problem arose the spot. This model also allows the flow of the development to be much more fluid since the team does not have to worry about combining multiple coding approaches and style into one and test every functionality if they are compatible or not.

Due to the unbalance construction of the code, the team utilizes the pair

programming strategy since it offers a linear approach to the construction of the program and the code versions is only being generated as the team progresses in the functionality of the program. After the team meeting prototyping becomes the design approach. When team members work on their own sometimes the coded program does not work flawlessly so it requires other to view it and hopefully fix the problem. Once the team is able to meet again, the prototyped functionality is discussed by the team and if there are problem, the team will try to collaboratively fix it until it is resolve. If the prototype functionality was not able to be mend the team will provide a redesign to approach the needed component of the program.

When one team member work alone on the program it is beneficial for the progress while it is also a troublesome. It is difficult because the team has to be provided a brief understanding of the new functionality characteristic. This approach could lead to misunderstandings and could use more time of discussing the code versus completing the development of the program.

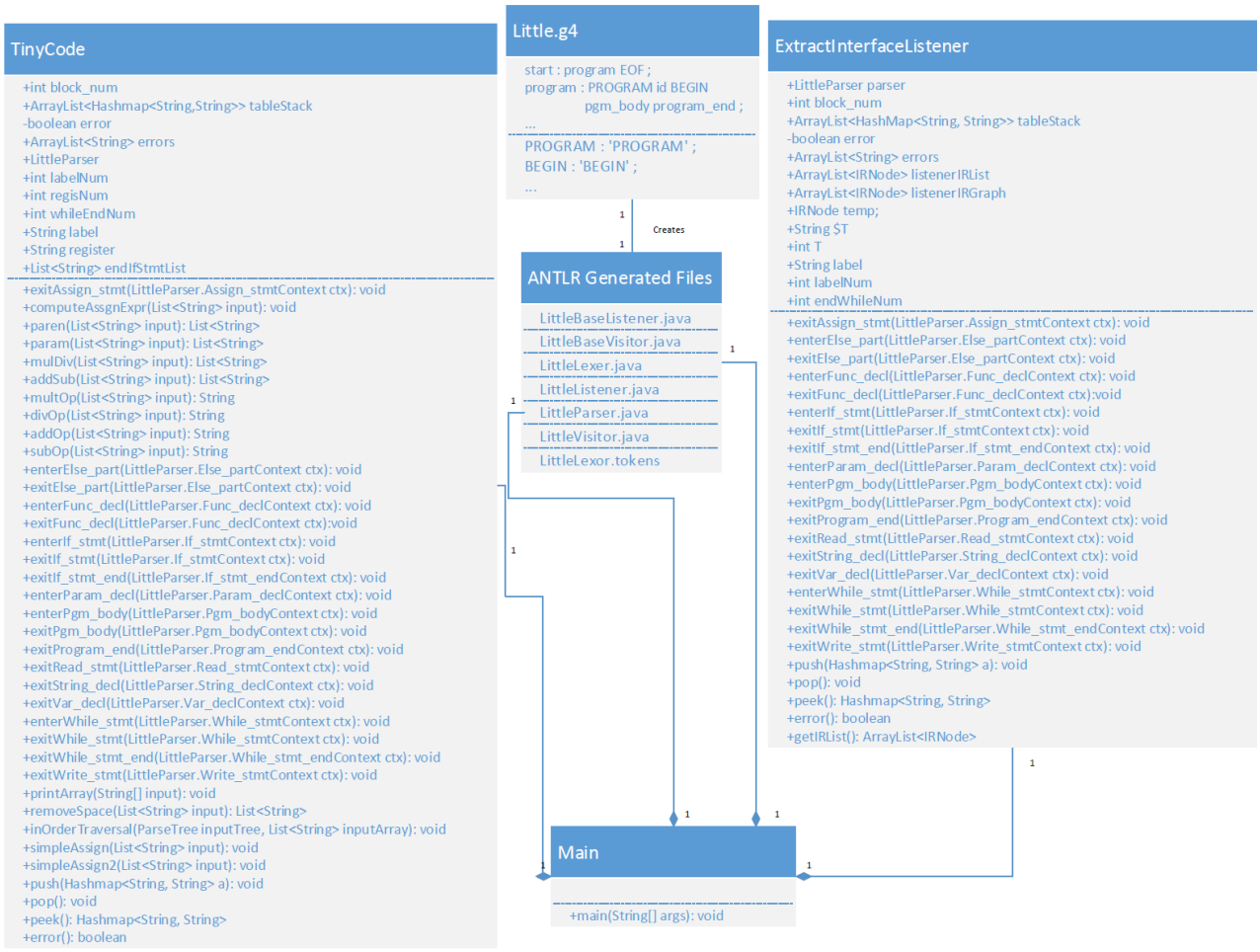

Figure 1: This is the UML class diagram for this compiler.