CS315 - Assignment1

Nikhil Bansal
170431

Shivam Kumar
170669

Baldip Singh Bijlani
170203

1 Tools Used

1.1 Lex

We used lex for building the lexical analyzer. It translates a set of regular expressions into a corresponding C implementation of finite state machine *lex.yy.c*, which on compiling yields the lexical analyzer *lexer*.

1.2 Bison

Bison is an open-source tool that converts a grammar description for an LALR(1) context-free grammar(*parser.y*) into a C program to parse that grammar. The C program generated, when compiled and linked with the object file of lexical analyzer, finally gives the executable parser. We automatically generated the state actions of the grammar using *gen_parser.py* and *gen_type.py*

1.3 DOT Language

The Parser generated uses DOT language for AST generation. It is a simple language which is used to represent graph textually. It stores the graph in the form of edges between vertices. Each edge is stored in the form of A -> B, where A and B are vertices and there is a directed edge from A to B in the graph.

1.4 DOT Tool

DOT tool is the default tool used in Graphviz , if the edges have directionality. It is used to visualize the graph from a script which represents the graph in textual form. for e.g. If graph.dot is a DOT script describing a graph, then the following command produces an image(.png file), which shows the graph in it's normal form:

```
$ dot -Tpng graph.dot -o graph.png
```

2 Compilation instructions

Source files are present in src/ directory. First, lexical analyzer of the language needs to be created, before moving on to the Syntax Analyzer/Parser level.The lexical structure of the java language is written in *lexer.l*. It can be compiled by :

```
$ lex lexer.l
```

The above produces a C implementation of lexical analyzer, which we compiled into an object file for it's use in parser:

```
$ g++ -c lex.yy.c -DECHO
```

Now, for the Syntax Analyzer part, the Syntactic grammar of java language is written in *parser.y*, which is then compiled with bison to yield a corresponding C implementation of the parser as follows:

```
$ bison -d parser.y
```

The C implementation is then converted to object file by compiling it with g++.

```
$g++ -c parser.tab.c
```

Now, finally the object files of lexer and syntax analyzer part are linked together to form the complete executable *parser*.

```
$ g++ lex.yy.o parser.tab.o -o parser
```

Thereby, the executable formed is capable for correctly parsing any given java file as input. We have provided a Makefile for ease of compilation. To compile the parser, simply run:

```
$ cd <milestone1>/src
$ make
```

To remove the generated files run:

```
$ make clean
```

Finally, we have provided a pre-built version of binary, compiled on our system.

3 Usage Instructions

Now, the parser generated can be used to parse any input file as follows:

```
$ ./parser –inp <inputfile> –out <outputfile>
```

If no input file is provided, it reads the inputs from stdin. Also, if no output file is provided, the generated dot script is written to stdout. Help message can be seen using parser –help.

To Convert output dot file to an image use:

```
$ dot -Tpng file_name.dot -o output_image_name.png
```

We have provided a wrapper script run.sh to run the parser. The Command:

```
$ run.sh test_1
```

, will read the file test_1.java, parse it and output its parse tree in test_1.png .

4 Features Supported

We support all the basic features required in the milestone. Apart from them, following optional features are supported as well:

- Support for Interfaces
- Static polymorphism via method overloading
- Limited annotations(@ Override) to support Dynamic polymorphism via method overriding
- Primitive Type Casting
- Inheritance