# CS335 - Compiler Design

Keshav Bansal(170335) Prateek Varshney(170494)
Snehal Raj(170705)

Milestone - 1

## 1 Introduction

For **Milestone 1**, we have used **python** (both versions 2 and 3) as our preferred language and used (source: ANTLR (Another Tool for Language Recognition)) as our parser generator.

The java documentations[1][2] comprehensively describe the rules for identifying each of the tokens for the lexer, as well as the grammar rules for the parser. We then use the parsed output on a given input file to generate the parse tree (using maketrees.py) and subsequently the Abstract Syntax Tree (AST) (using ast.py).

## 2 Method for generating AST

From the antlr4 we receive a parse tree for the given program. Antlr provides an interface to access the tree of RuleContext objects created during a parse that makes the data structure look like a simple parse tree. This node represents both internal nodes, rule invocations, and leaf nodes, token matches. The payload is either a Token or a RuleContext object.

To generate the Abstract Syntax Tree, we used ANTLR's listener interface. We chose the listener methods becuase they are called automatically by the ANTLR provided walker object, whereas visitor methods must walk their children with explicit visit calls.Moreover, Listener uses an explicit stack allocated on the heap, whereas visitor uses call stack to manage tree traversals. This might lead to StackOverFlow exceptions while using visitor on deeply nested ASTstraverse over each node of the parse tree and compress the nodes having only one child node(the only exception to this being when the non terminal is parent to a terminal). Now, while using the listener interface, we deleted any nodes which had only one child. The only exception to this rule was when the non terminal is parent to a terminal, in which case we keep the literal. The AST is first generated in the bracket notation which is then converted into a dot script using python graphviz library.

## 3 Execution Instructions

Before execution make sure that the environment has both python3, python2 and antlr4 runtime.
To install the antlr4 runtime, use the following command

```
$ pip install −r requirements.txt
```

Command line arguments for python script **ast.py** are as follows -

| Flag | Description |
|---|---|
| -h, –help | show this help message and exit |
| –verbose, -v | Display intermediate steps before outputing the parse tree |
| -input INPUT | input file name from which the program must be fetched |
| -out OUT | output file name to which the postscript is written |

To get the syntax tree for program **testcase1.java** and store it in **output.ps** use the command

```
$ python3 ast.py −input testcase1.java −out output.ps
```

To view the postscript file execute the command -

```
$ xdg−open output.ps
```

## 4 Optional Features

### 4.1 Support for Interfaces

An interface is a special abstract structure in Java that resembles a class except a Java interface can only contain method signatures and fields. Instead of containing implementations of the methods, it only contains the signature (name, parameters and exceptions) of the method.
We have included the support for Interfaces in our grammar and have provided **test_1.java** to test the same.

### 4.2 Static polymorphism via method overloading

Static polymorphism refers to the kind of polymorphism which is resolved at compile time. Method overloading in Java allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters. We have added support for Static polymorphism via Method overloading in our grammar and have provided **test_2.java** to test the same..

## 4.3 Dynamic polymorphism via method overriding

Dynamic polymorphism (or Dynamic Method Dispatch) is a process in which a call to an overridden method is resolved at runtime. It might happend when a child class has an object assigned to the parent class. So in order to determine which method would be called, the type of the object would be determined at run-time. We have included support for Dynamic polymorphism via method overriding in our grammar and have provided **test_3.java** to test the same.

## 4.4 Type casting

Typecasting, or type conversion, is a method of changing an entity from one data type to another. There are specifically two kinds of typecasting in java,Widening or Automatic type converion and Narrowing or Explicit type conversion. Automatic Type casting take place when, the two types are compatible and the target type is larger than the source type. Narrowing or Explicit type conversion is needed when we assign a larger type value to a variable of smaller type. Our includes support for typecasted data types which can be test from the file **test_4.java**.

## 4.5 Generics

Java Generic methods enable support for specifying, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively. Using the rules to define Generic Methods in Java, we have added support for methods, bounded parameter types, classes etc. The support for Generics can be tested via **test_5.java**.

## References

[1] https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html

[2] https://docs.oracle.com/javase/specs/jls/se8/html/jls-19.html

[3] https://tomassetti.me/antlr-mega-tutorial/python-setup