+ECOTE – final project

Semester: 2020L

Author: Minh Hieu Do - 288414

Subject: Task 6. CD1

Write a program reading a subset of C++, C# or Java code and constructing class inheritance tree (including also interface realizations, if appropriate). Output the tree. Specify appropriate code limitations.

I. General overview and assumptions

Write a program reading a subset of Java code and constructing class inheritance tree (including also interface realizations, if appropriate). Output the tree. Specify appropriate code limitations.

In the project, ANTLR (ANother Tool for Language Recognition) v4.8 tool will be used to parse input according to the grammar.

Program should recognize incorrect syntax and inform user about it with the proper error message.

The assumptions are:

- All input classes, interfaces are in the single file
- All interfaces are somewhere implemented in library or package or in the same file otherwise program will output error.

Grammar

- Parser rules and lexer rules of Java are defined in “JavaParser.g4” and “JavaLexer.g4” respectively, grammar file which can be found and downloaded in ANTLR's Github grammar repo.
- Parser rules start with lowercase letters, lexer rules with uppercase.

II. Functional requirements

(with algorithms at design level, detailed syntax of an input language if appropriate, transformation rules if appropriate, etc.)

Only one file with *.java extension processed to construct the inheritance tree is passed as argument in the command file.

Syntax of input language should be acceptable according to Java grammar. There is only single inheritance which means a class can only inherit from one parent class.

Program should construct one class inheritance trees of the input file. All independent classes should be children of *java.lang.Object* class.

III. Implementation

General architecture

Classes

- ConstructInheritanceTree

- Main class of the program. It handles all errors during construction process, takes input file from command line and prints the output.
- It handles 3 important exceptions
  - File not found
  - File is not runnable
  - Grammar errors
- Variables
  - *inputFile* – input file name
  - *cs* – character stream of input file
  - *lexer* – a sequence of characters as a sequence of tokens
  - *parser* – parser of grammar
  - *tree* – assign beginning rule for parsing
  - *walker* – trigger listeners waiting for event
  - *inheritanceTree* – instance of ConstructInheritanceTreeListener
- Methods
  - *parseFile(String… args)* – parse input file and construct inheritance tree
  - *runProcess(String inputFile)*- runs input file and throws error if process failed

- ConstructInheriatnceTreeListener
  - Class derives from *JavaParserBaseListener* and overrides methods according to program needs. Listener methods are phrase elements matched by the rules. Each method rule can be overrided if needed and will be triggered by the walker when match rule appears in parse tree.
  - Variables
    - *classHierarchy* – HashMap stores children of each class
    - *interfaceSet* – HashMap stores interfaces each class implements
    - *root* – "Object", root of inheritance tree
  - Methods
    - *ConstructInheritanceTreeListener()* – constructor of the class
    - *enterClassDeclaration(JavaParser.ClassDeclarationContext ctx)* – enter this rule to construct the inheritance tree
    - *printTree(String cl, int depth)* – print class having name *cl* at the depth *depth*, interfaces implemented and its children if any
    - *printInheritanceTree()* – print the complete inheritance tree

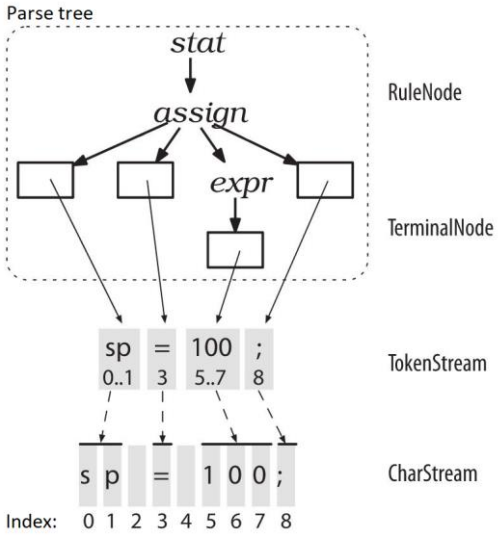
- NotRunnableProcess
  - Class extending *Exception* informs error during file compilation error.
  - Variable
    - *proc* – proccess run from input file
  - Method
    - *toString()* – returns error messsage

- *ThrowingErrorListener*
  - Class extending *BaseErrorListner* handles by default grammar generated by parser and throws exception.
  - Variable

        - *INSTANCE* – stores “this” created instance
    - Methods
        - *syntaxError(Recognizer recognizer, Object offendingSymbol, int line, int charPositionInLine, String msg, RecognitionException e)* – throws ParseCancellationException containing error message

Data structures

According to compiler structure, lexers process characters and pass tokens to the parser, which in turn checks syntax and creates a parse tree. The corresponding ANTLR classes are *CharStream*, *Lexer*, *Token*, *Parser*, and *ParseTree*. The “pipe” connecting the lexer and parser is called a *TokenStream*. The diagram below illustrates how the object of these types is connected in memory.

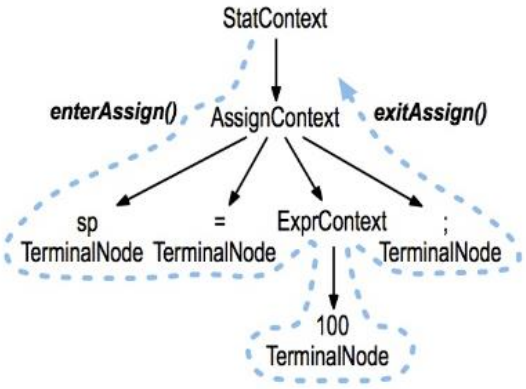


Each non-terminal symbol (in our case, each rule in grammar) is a context object and records everything known about the recognition of a phrase by a rule. Each context object knows the start and stop tokens for the recognized phrase since ANTLR uses tree data structure provides access to all the elements of that phrase.

ANTLR provides support for two tree-walking mechanisms in its runtime library, Parse-Tree Listeners and Visitors. In this project, listener mechanism is used.

To walk a tree and trigger calls into a listener, ANTLR’s runtime provides class *ParseTreeWalker*. The methods in a listener are just callbacks. To make a language application, we build a *ParseTreeListener* implementation containing application-specific code that typically calls into a broader surrounding application.

ANTLR generates a *ParseTreeListener* subclass specific to each grammar with enter and exit methods for each rule. As the walker encounters the node for rule *assign*, for example, it triggers *enterAssign* and passes it the *AssignContext* parse-tree node. After the walker visits all children of the *assign* node, it triggers *exitAssign*.



It also identifiers where in the walk *PareseTreeWalker* calls the enter and exit methods for rule *assign.*

The diagarm below shows the complete sequence of calls made to the listener by *ParseTreeWalker* for the statement tree.

Module descriptions

(with algorithms at implementation level)
Running ANTLR generates code (a parser and a lexer) that recognizes sentences in the language described by the grammar. It also creates listeners interface.

Name of the files is combination of name of the grammar and parts of our compiler.

JavaParser.java - This file contains the parser class definition specific to grammar JavaParser.g4 that recognizes our array language syntax. It contains a method for each rule in the grammar as well as some support code.

JavaLexer.java – This file contains the lexer class definition, which ANTLR generated by analyzing the lexical rules as well as the grammar literals. Recall that the lexer tokenizes the input breaking up into vocabulary symbols.

JavaParser.tokens, **JavaLexer.tokens** - ANTLR assigns a token type number to each token we define and stores these values in this file. It's needed when we split a large grammar into multiple smaller grammars so that ANTLR can synchronize all the token type numbers.

JavaParserListener.java, **JavaParserBaseListener.java** - By default, ANTLR parsers build a tree from the input. By walking that tree, a tree walker can trigger events (callbacks) to a listener object that we provide. JavaListener is the interface that describes the callbacks we can implement. *JavaParserBaseListener* is a set of empty default implementations which we inherit from. This class makes it easy for us to override just the callbacks we're interested in, otherwise we would have to implement all methods in JavaListeners interface.

Input/output description

Input: Single file containing Java code includes all classes, interfaces needed for processing. The input file should compile and have the correct syntax.
Output:

- The inheritance trees of the code with implemented interfaces of each class if any if code is compilable. Otherwise, print error of the input.
- All classes are printed at an increased indentation level.
- Class and implemented interfaces are separated by a colon.
- All interfaces implemented by a class are separated by commas.

See the example below.

- Ouput example:

  Object
  |-- ClassA: InterfaceA,InterfaceB
  |-- ClassB
  |-- ClassC: InterfaceA

- Explaination:

  ClassA and *ClassC* are children of *Object.*
  ClassB is a child of *ClassA.*
  ClassA implements *InterfaceA* and *InterfaceB.*
  ClassC implements *InterfaceA.*

Others

There is a JAR file can be found named "MinhHieuDo_ECOTE_Project.jar" to run the program.
It will construct the inheritance tree of the input file passed as command-line argument parameter. If there is no input file given, the program will read and analyze all .java files can be found in the directory "testInput".

IV. Functional test cases

Testcase 1. Successful case

- Example:

```
import java.util.*;
interface As {
  void msg();
}
interface Ds {}
class TestCase1 implements As, Ds {
  public void msg(){System.out.println("Hello");}
```

```
}

class B extends D {
   void msg(){System.out.println("Welcome");}
}
class C extends TestCase1 {
   public static void main(String args[]){
      C obj = new C();
      obj.msg();
   }
}

class D implements Ds {}
```

- Expected output:

```
Object
   |-- TestCase1: As,Ds
      |-- C
   |-- D: Ds
      |-- B
```

Test case 2. Missing ‘{‘ at class declaration

- Example:

```
import java.util.*;
interface As {
   void msg();
}

interface Ds {}
class TestCase2 implements As, Ds
   public void msg(){System.out.println("Hello");}
}
```

- Expected output:

```
Line 7:4 missing '{' at 'public'
```

Test case 3. Missing comma separate interfaces implemented

- Example:

```
import java.util.*;
interface As {
   void msg();
}
interface Ds {}
class TestCase3 implements As Ds {
   public void msg(){System.out.println("Hello");}
}
```

- Expected output:

```
Line 6:30 extraneous input 'Ds' expecting '{'
```

Test case 4. Missing semicolon

- Example:

```
import java.util.*;
interface As {
  void msg();
}
interface Ds {}
class TestCase4 implements As, Ds {
  public void msg(){System.out.println("Hello")}
}
```

- Expected output:

```
Line 7:49 missing ';' at '}'
```

Test case 5. Missing interface followed comma

- Example:

```
import java.util.*;
interface As {
  void msg();
}
interface Ds {}
class TestCase5 implements As, {
  public void msg(){System.out.println("Hello");}
}
```

- Expected output:

```
Line 6:31 mismatched input '{' expecting {'boolean', 'byte', 'char', 'double', 'float', 'int',
'long', 'short', '@', IDENTIFIER}
```

Test case 6. Missing ‘class’ at class declaration

- Example:

```
import java.util.*;
interface As {
  void msg();
}
interface Ds {}
TestCase6 implements As {
  public void msg(){System.out.println("Hello");}
}
```

- Expected output:

```
Line 6:0 extraneous input 'TestCase6' expecting {<EOF>, 'abstract', 'class', 'enum', 'final',
'interface', 'private', 'protected', 'public', 'static', 'strictfp', ';', '@'}
```

Test case 7. Missing necessary package

- Example:

```
interface As {
  void msg();
```

```
}
interface Ds {}
class TestCase7 implements As, Ds {
   Scanner s = new Scanner(System.in);
   public void msg(){System.out.println("Hello");}
}
```

- Expected output:

```
Input file is not compilable
.\testInput\TestCase7.java:6: error: cannot find symbol
   Scanner s = new Scanner(System.in);
   ^
 symbol:   class Scanner
 location: class TestCase7
.\testInput\TestCase7.java:6: error: cannot find symbol
   Scanner s = new Scanner(System.in);
               ^
 symbol:   class Scanner
 location: class TestCase7
2 errors
```

Test case 8. Missing 'implements' when implementing interface

- Example:

```
import java.util.*;
interface As {
   void msg();
}
interface Ds {}
class TestCase8 As {
   public void msg(){System.out.println("Hello");}
}
```

- Expected output:

```
Line 6:16 extraneous input 'As' expecting {'extends', 'implements', '{', '<'}
```

Test case 9. Missing return type at method declaration

- Example:

```
import java.util.*;
interface As {
   void msg();
}
interface Ds {}
class TestCase9 implements As {
   public msg(){System.out.println("Hello");}
}
```

- Expected output:

```
Input file is not compilable
.\testInput\TestCase9.java:7: error: invalid method declaration; return type required
   public msg(){System.out.println("Hello");}
```

```
      ^
1 error
```