Assignment 1

CS 327 - Compilers

Due date: 24th February, 2022 (11:59 pm IST)

1 Introduction

In this assignment, you will be building a front-end for ChocoPy in Java that consists of a scanner, which performs lexical analysis, and a parser, which performs syntax analysis. Instead of building these components from scratch by hand, you will be using two tools: JFlex, a scanner generator (`https://jflex.de/manual.pdf`), and CUP, a parser generator (`http://www2.cs.tum.edu/projects/cup/docs.php`). JFlex is a tool that converts a specification of lexical analysis, written in a specific format, into a Java class ChocoPyLexer. The ChocoPyLexer class processes an input string and produces a sequence of tokens. The CUP tool converts a specification of a program grammar into a Java class ChocoPyParser. The ChocoPyParser class performs syntax analysis on the sequence of tokens produced by ChocoPyLexer and executes user-specified actions while parsing. These actions contain code to build an AST.

2 Getting Started

We are going to use the GitHub Classroom platform for managing programming assignments and submissions.

- Visit `https://classroom.github.com/a/qAMN1eWI` for the assignment. You will need a GitHub account to join.
- The first team member accepting the assignment should create a new team with some reasonable team name. The other team members can then find the team in the list of open teams and join it when accepting the assignment. A private GitHub repository will be created for your team. It should be of the form https://github.com/IITGN-CS327-2022/Assignment1-¡team¿ where ¡team¿ is the name of your team.
- Ensure you have Git, Apache Maven and JDK 8+ installed. See Section 3 for more information regarding software.
- Run `git clone git@github.com:IITGN-CS327-2022/Assignment1-<team>.git` where ¡team¿ is the name of your team, to clone the repository. It will contain all the files required for the assignment. Your repository must be private.
- Run `mvn clean package`. This will compile the starter code, which parses a tiny subset of ChocoPy. Your goal is to develop a parser that conforms to the grammar listed in the ChocoPy language manual completely and produces output as described in this document.
- Run the following command (on a single line) to test the generated parser against sample inputs and expected outputs—only one test will pass with the starter code:

  ```
  java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=s --dir
  src/test/data/pa1/sample --test
  ```

  Windows users should replace the colon between the JAR names in the classpath with a semicolon:
  `java -cp "chocopy-ref.jar;target/assignment.jar"` .... This applies to all Java commands listed in this document.

3 Files and Directories

The assignment repository contains a number of files that provide a skeleton for the project. Some of these files should not be modified, as they are essential for the assignment to compile correctly. Other files must be modified in order to complete the assignment. You may also have to create some new files in this directory structure. The list below summarizes each file or directory in the provided skeleton.

- `pom.xml`: The Apache Maven build configuration. You should not modify this as it is set up to compile the entire pipeline.
- `src/`: The `src` directory contains manually editable source files, some of which you must modify for this assignment.
  - `src/main/jflex/chocopy/pa1/ChocoPy.jflex`: This file contains the specifications for the JFlex scanner generator tool. You will need to modify this file to write specifications for tokenizing programs written in ChocoPy.
  - `/main/cup/chocopy/pa1/ChocoPy.cup`: This file contains the grammar for the CUP parser generator tool. You will need to modify this file to specify the syntax of ChocoPy and the actions to be executed while parsing an input.
  - `src/test/data/pa1`: This directory contains ChocoPy programs for testing your parser
    - `/sample/*.py` - Sample test programs covering a variety of features of the ChocoPy language that you need to implement in this assignment.
    - `/student contributed/good.py` - A test program that parses successfully. You have to modify this file to test various features of your parser.
    - `/student contributed/bad.py` - A test program that does not parse. You have to modify this file to test various types of syntax errors and error recovery.
- `target/`: The target directory will be created and populated after running `mvn clean package`. It contains automatically generated files that you should not modify by hand. This directory will be deleted before your submission.
  - `target/generated-sources/jflex/chocopy/pa1/ChocoPyLexer.java`: Generated by JFlex, this file will contain the DFAs constructed from the lexical specifications along with any Java code provided as actions. DO NOT MODIFY THE GENERATED FILE BY HAND. However, you may want to inspect this file for compilation errors. In case any of the generated code leads to compilation errors, you should fix the `ChocoPy.jflex` file instead. This file may reference tokens defined in `ChocoPyTokens.java`, which means you should run `mvn clean package` every time you add or modify a terminal declaration in `ChocoPy.cup`.
  - `target/generated-sources/cup/chocopy/pa1/ChocoPyTokens.java`: This file is generated by CUP. This file simply contains a list of token identifiers generated from the terminals declared in `ChocoPy.cup`. These symbols are used both in the lexer and in the parser. DO NOT MODIFY THE GENERATED FILE BY HAND. It is overwritten every time CUP is executed.
  - `cup/chocopy/ChocoPyParser.java`: This file is the main parser generated by CUP. It will contain the LALR parsing tables as well as any action code that you specify in the `ChocoPy.cup` file. DO NOT MODIFY THE GENERATED FILE BY HAND. If you notice any compilation errors in this file, you probably need to fix the action code embedded in `ChocoPy.cup`.
  - `target/assignment.jar`: This is where your compiled parser will be packaged.
- `chocopy-ref.jar`: A reference implementation of the parser, provided by the instructors.
- `README.md`: You will have to modify this file with a writeup.

4 Assignment Goals

The objective of this assignment is to build a front-end for ChocoPy that parses an input ChocoPy program and produces an abstract syntax tree (AST) in JSON format. For a single input file, the parser is invoked by running the following command (on a single line):

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=s <input file>
```

where `<input file>` is a placeholder for the path to a ChocoPy program file.

4.1 Expected output

The parser should output the AST of the program in a JSON format, which is described in the rest of this section.

4.1.1 JSON format

JSON is a notation for representing a tree of objects. A JSON object is a set of key-value pairs called properties, represented using curly braces:
`<key1>: <value1>, <key2>: <value2>, ... .`
For example,
`"product" : "iPad Pro", "company": "Apple", "year": 2016, "released": true.`
Keys are always strings delimited by double quotes; the values can be strings, integers, booleans (`true/false`), the value `null`, other JSON objects, or JSON arrays. Arrays are represented as a list of values delimited by square brackets: `[<value1>, <value2>, ...]`. You can find a complete specification for JSON at `https://json.org`.

In our AST representation, we denote each AST node using a JSON object. Such a JSON object has a particular kind which specifies what keys the object must contain and what types the corresponding values will take. For example, the `Identifier` kind specifies one property, with a key called `name`, whose value must be a string corresponding to the name of the identifier. Similarly, the `UnaryExpr` kind specifies two properties: a string-valued `operator`, and a property with key operand whose value is of kind `Expr`. Kinds can extend other kinds, by including the properties specified by the extended kind as a subset of their own properties. Both `Identifier` and `UnaryExpr` extend kind `Expr`, and therefore JSON objects of these kinds may appear as values whenever an object of kind `Expr` is expected. All kinds in our AST directly or indirectly extend the `Node` kind which specifies two properties: (1) a string-valued property called `kind` that simply contains the kind of the node and (2) `location`, an array of integers. The following is a sample JSON representation of the AST corresponding to the unary expression (`-foo`):

```
{
  ”kind”: ”UnaryExpr”,
  ”operator”: ”-”,
  ”operand”: {
  ”kind”: ”Identifier”,
  ”name”: ”foo”,
  ”location” : [ 1, 3, 1, 5 ]
  },
  ”location” : [ 1, 2, 1, 5 ]
}
```

The location array always contains four integers and describes source code location information for the corresponding AST node: (1) the line number of the first character, (2) the column number of the first character, (3) the line number of the last character, and (4) the column number of the last character.

4.1.2 AST node kinds

For this assignment, the set of all kinds required to serialize ASTs are listed in Figure 1. The syntax `kind K {...}` defines a kind and `kind K extends S {...}` defines a kind `K` that extends kind `S`. Properties are defined as `<k>:<v>` where `<k>` is the name of the key and `<v>` is the type of the value. Value types are one of `string`, `int`, `bool`, a JSON object of kind `K`, or a JSON array of type `t` represented as `[t]`. Properties that may contain null values are suffixed with a question mark.

When provided with a ChocoPy program, the output of the parser should be a JSON object of kind Program. Most AST node kinds correspond directly to production rules in the grammar. A notable exception is the `IfStmt` kind, which only contains one `elseBody` even though the grammar allows a sequence of `elif` statements. The `if-elif-else` form is syntactic sugar; the parser de-sugars `elifs` as an `elseBody` with exactly one `IfStmt` in its body. Refer to ChocoPy language reference for an example of this equivalence. The file `src/test/data/pa1/sample/coverage.py` contains a sample ChocoPy program that covers almost all syntax rules and AST node kinds; the corresponding AST JSON can be found in `src/test/data/pa1/sample/coverage.py.ast`. You can also run any input ChocoPy program through the provided reference implementation of the parser, which should produce the JSON-formatted AST that you need to produce. To parse an input program using the reference implementation, run the command (on a single line):

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=r <input file>
```

The output of the reference implementation will be the expected behavior of your parser (unless there are some bugs).

4.2 Error handling

Your lexer should not report errors for any reason. Whenever a token is unrecognized, your lexer should emit the dummy token UNRECOGNIZED. Your parser should not use this token in any grammar rule, thereby leading to a syntax error. You may introduce other dummy tokens for specific purposes, if you desire.

Your parser should be able to recover from simple errors and continue parsing. You can use the predefined `error` nonterminal (refer to the CUP manual) to recover from syntax errors. In particular, your parser should be able to (1) recover from errors within a statement and continue parsing following statements, (2) recover from errors parsing a variable declaration, function definition, or class definition and continue parsing the rest of the declarations and statements.

The `Program` object produced by the parser contains a JSON object of kind `Errors`, which contains a list of `CompilerError` objects, each having a message and the location of an erroneous construct in the source code. For a correct program, the `Errors` object will contain an empty list of `CompilerError` objects. Syntax errors should contain messages of the form `Parse error near token <TOKEN>: <text>`, where `<TOKEN>` is the name of the lexical token where the parse error occurred, and `<text>` is the actual text of the recognized token. You must make every effort to match the error reporting behavior of the reference parser. However, the next section describes how we will validate this.

4.3 Validation

The JSON output from your submission will be compared to the JSON output from the reference implementation. The JSON files need not match verbatim, since whitespace is ignored and the order of properties in a JSON object is not specified. Instead, the JSON object produced by your parser will be compared with the JSON object produced by the reference implementation by recursively comparing all properties and their values. You can use an online tool such as http://www.jsondiff.com to compare two JSON objects for semantic equivalence or to find where in the tree they are different.

For any given test input program, if your parser outputs a string that is not valid JSON, then the test is considered failed. When the input is a syntactically valid ChocoPy program, the test fails if the JSON object produced by your parser is different from the JSON object produced by the reference implementation. In case of invalid ChocoPy programs that lead to syntax errors, the error messages produced by your parser may be slightly different from the reference implementation due to differences in implementing error recovery mechanisms. For this reason, we will only compare the line and column number of the first token that results in error. You can run the following command to the test the output of your parser by comparing it to the outputs of the samples provided:

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=s --dir
src/test/data/pa1/sample --test
```

Look at the files in `src/test/data/pa1/sample` ending with extension `.ast` for the expected JSONs of individual test cases. For example, the expected JSON for `expr_plus.py` is the file `expr_plus.py.ast`.

4.4 Project skeleton

You have been provided a skeleton project in the assignment repository with some starter code. The project contains code for generating a lexer and parser for a tiny subset of ChocoPy. The starter code also contains a hierarchy of Java classes corresponding to the AST structure. You are welcome to extend this code in any way you like or to discard the starter code and develop a new project from scratch. The only requirement is that the submitted project must build using the command `mvn clean package`, and the generated parser must be executable and testable using the exact commands listed in this document. You must not modify classes in the `chocopy.common` package directly, since the test framework and reference implementation also depend on them. However, feel free to duplicate classes in a custom package and then modify them.

4.5 Writeup

Before submitting your completed assignment, you must edit the README.md and provide the following information: (1) names of the team members who completed the assignment, (2) acknowledgements for any collaboration or outside help received, and (3) how many grace hours have been consumed (refer to the Canvas page for the policy). Further, you must answer the following questions in your write-up by editing the `README.md` file (one or two paragraphs per question is fine):

1. What strategy did you use to emit INDENT and DEDENT tokens correctly? Mention the filename and the line number(s) for the core part of your solution.
2. What was the hardest language feature (not including indentation) to implement in this assignment? Why was it challenging? Mention the filename and line number(s) for the core part of your solution.

5 Hints and Notes

ChocoPy, like Python, uses an indentation-based syntax for determining boundaries of blocks of statements. The ChocoPy manual goes into some detail about how to emit `INDENT` and `DEDENT` tokens, but you need to figure out the best way to engineer this in JFlex. There is definitely more than one way to implement this scheme. You may want to go through the JFlex manual to get some idea on the various features it supports. Read about `%state, YYINITIAL`, and the `yypushback()/yyreset()` methods for controlling the behavior of the lexer.

You must be careful about declaring types for terminals and nonterminals. Typed terminals and nonterminals can be referenced via the colon syntax in production rules to bind their values to variables for use in action rules (look at the rules ChocoPy.cup in the starter code). Do not try to force symbols to be of a particular type by adding unnecessary type casts — this will open up the possibility of unexpected class cast exceptions in corner cases. Instead, modularize your production rules by declaring accurate types for nonterminals depending on what AST nodes they resolve to. The starter code also provides hints on how to collect the start and end source location information (i.e., line and column numbers) from the leftmost and rightmost symbols matched in a production rule.

You can run your parser with the –debug flag to see additional information about the lexer and parser generation process. In particular, JFlex prints NFAs and DFAs constructed from the specifications in `ChocoPy.jflex`, and CUP prints LALR parsing tables and transition rules generated from the grammar in `ChocoPy.cup`. For a single input file, the usage is:

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=s --debug <input
file>
```

If you are interested in observing the sequence of tokens produced by your lexer, you can run the following command:

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.pa1.ChocoPyLexer <input file>
```

You can replace `pa1` with `reference` in the above command to see the corresponding output by the reference lexer. This command will likely be very useful when working on logic for `INDENT` and `DEDENT` tokens. Note: The output of your lexer need not match exactly the output of the reference lexer, since there are different ways in which you can process blank lines and indentation to parse the same syntax. This command is for debugging purposes only. You will only be graded on the JSON outputs of your parser.

6 Submission

Submitting your completed assignment requires the following steps:

1. Run `mvn clean` to rid your directory of any unnecessary files.
2. Add and commit all your progress and push changes to the repository. Run `git commit` followed by `git push origin` to achieve this.
3. Tag the desired commit with `pa1final`. If the desired commit is the latest one, run `git tag pa1final`. Otherwise, run `git tag pa1final <commit-id>` where `<commit-id>` is the commit you want to tag as your final submission. Only the tagged submission will be considered as the final submission and the time of the tag will be considered for the late submission policy.
4. Push the tag using `git push origin pa1final`.

7 Grading (20 points)

The grading shall be as follows:

- 16 points for autograder tests.
- 1 point for the README. (Refer Writeup.)
  - 0.5 points for Q1 of writeup.
  - 0.5 points for Q2 of writeup.
- 2 points for custom test cases.
  - 1 point for `src/test/data/pa1/student_contributed/good.py`. This file must be manually edited to cover as many syntax features as your parser can support. Running your parser on this file should not result in an error.
  - 1 point for `src/test/data/pa1/student_contributed/bad.py`. This file must be manually edited to demonstrate how your parser recovers from errors. Running your parser on this file should report at least two meaningful errors on distinct lines.
- 1 point for code cleanliness: Full points awarded for clear naming for variables and other symbols, consistent spacing and punctuation conventions, reasonable modularization of functions and other components, code comments explaining non-obvious logic. Zero points if little to no effort to organize and document.