CS164 Programming Languages and Compilers	**Fall 2018**

Programming Assignment 1

Assigned: September 8, 2018 — **Due:** Oct 2, 2018 at 11:59pm

1 Overview

The three programming assignments in this course will direct you to develop a compiler for ChocoPy, a statically typed dialect of Python. The assignments will cover (1) lexing and parsing of ChocoPy into an abstract syntax tree (AST), (2) semantic analysis of the AST, and (3) code generation.

For this assignment, you are to build 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, and *CUP*, a parser generator. 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.

- Create a github account (if you don't have one already). Visit `https://classroom.github.com/g/YPG3pf1g` for the assignment.
- Accept the assignment, and visit the repository just created for you. It should be of the form `https://github.com/cs164fall2018/pa1-chocopy-parser-<username>` where `<username>` is your GitHub username.
- Run `git clone https://github.com/cs164fall2018/pa1-chocopy-parser-<username>.git` (where `<username>` is your GitHub username) to clone the repository. It will contain all the files required for the assignment. Your repository must be private; otherwise, you will get 0 points in this assignment.
- 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 to test the generated parser against sample inputs and expected outputs. Only one test will pass with the starter code.

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

3 Software dependencies

The software required for this assignment is as follows:

- Git, version 2.5 or newer: `https://git-scm.com/downloads`
- Java Development Kit (JDK), version 8 or newer: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
- Apache Maven, version 3.39 or newer: `https://maven.apache.org/download.cgi`
- (optional) An IDE such as IntelliJ IDEA (free community editor or ultimate edition for students): `https://www.jetbrains.com/idea`.
- (optional) Python, version 3.6 or newer, for running ChocoPy programs in a Python interpreter: `https://www.python.org/downloads`

If you are using Linux or MacOS, we recommend using a package manager such as `apt` or `homebrew`. Otherwise, you can simply download and install the software from the websites listed above. We also recommend using an IDE to develop and debug your code. Since the provided starter code is a Maven project, you can import it in IntelliJ as a project with existing sources that uses Maven to build.

4 External Documentation

- JFlex user's manual: `http://jflex.de/manual.html`
- CUP user's manual: `http://www2.cs.tum.edu/projects/cup/docs.php`

5 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.

- `chocopy_language_reference.pdf`: The ChocoPy language manual: it specifies the ChocoPy syntax and should be used as a reference throughout the programming assignment(s).
- `PA1.pdf`: This document. It contains the specification for the assignment.
- `pom.xml`: The Apache Maven build configuration. You do not need to 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.

  - `src/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. Note that 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 generated parser will be compiled and 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.

6 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:

```
java -cp "target/assignment.jar:chocopy-ref.jar" chocopy.ChocoPy --pa1 chocopy.pa1.StudentParser
--in <input_file>
```

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

6.1 Expected output

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

6.1.1 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>, ...` }. An example JSON object is {`"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>, ...]`. A complete specification for JSON may be found 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 an example 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 leftmost character, (2) the column number of the leftmost character, (3) the line number of the rightmost character, and (4) the column number of the rightmost character.

6.1.2 AST node kinds

For this assignment, we list the set of all kinds required to serialize ASTs in Figure 1. We use the syntax `kind K {...}` to define a kind and `kind K extends S {...}` to define 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 syntactically valid ChocoPy program, the output of the parser should be a JSON object of kind `Program`. 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.out`.

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:

```
java -cp "target/assignment.jar:chocopy-ref.jar" chocopy.ChocoPy --pa1 chocopy.reference.RefParser
--in <input_file>
```

In case of any ambiguities or discrepancies, the output of the reference implementation will be considered the expected behavior of your parser.

6.2 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.

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 catch 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.

If there are one or more errors in parsing an input program, then the output of the parser should be a JSON object of kind `Errors`, which contains a list `Error` objects, each having a message and the location of the errornous token in the source code. 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.

6.2.1 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.

```
kind Node {
  kind: string,
  location: [int]
}

kind Program extends Node {
    declarations: [Declaration],
    statements: [Stmt]
}

kind Declaration extends Node { }

kind ClassDef extends Declaration {
    name: Identifier,
    superClass: Identifier,
    declarations: [Declaration]
}

kind FuncDef extends Declaration {
    name: Identifier,
    params: [TypedVar],
    returnType: TypeAnnotation,
    declarations: [Declaration],
    statements: [Stmt]
}

kind VarDef extends Declaration {
    var: TypedVar,
    value: Literal
}

kind GlobalDecl
  extends Declaration {
    variable: Identifier
}

kind NonlocalDecl
  extends Declaration {
    variable: Identifier
}

kind TypedVar extends Node {
    identifier: Identifier,
    type: TypeAnnotation
}

kind TypeAnnotation
  extends Node { }

kind ClassType
  extends TypeAnnotation {
    className: string
}

kind ListType
  extends TypeAnnotation {
    elementType: TypeAnnotation
}

kind Stmt extends Node { }

kind PassStmt extends Stmt { }

kind ExprStmt extends Stmt {
    expr: Expr
}

kind ReturnStmt extends Stmt {
    value: Expr?
}

kind VarAssignStmt extends Stmt {
    var: Identifier,
    value: Expr
}

kind MemberAssignStmt
  extends Stmt {
    objectMember: MemberExpr,
    value: Expr
}

kind IndexAssignStmt
  extends Stmt {
    listElement: IndexExpr,
    value: Expr
}

kind IfStmt extends Stmt {
    condition: Expr,
    thenBody: [Stmt],
    elseBody: [Stmt]
}

kind WhileStmt extends Stmt {
    condition: Expr,
    body: [Stmt]
}

kind ForStmt extends Stmt {
    identifier: Identifier,
    iterable: Expr,
    body: [Stmt]
}

kind Expr extends Node { }

kind Identifier extends Expr {
    name: string
}

kind BinaryExpr extends Expr {
    left: Expr,
    operator: string,
    right: Expr
}

kind UnaryExpr extends Expr {
    operator: string,
    operand: Expr
}

kind CallExpr extends Expr {
    function: Identifier,
    args: [Expr]
}

kind MethodCallExpr extends Expr {
    method: MemberExpr,
    args: [Expr]
}

kind IndexExpr extends Expr {
    list: Expr,
    index: Expr
}

kind MemberExpr extends Expr {
    object: Expr,
    member: Identifier
}

kind ListExpr extends Expr {
    elements: [Expr]
}

kind Literal extends Expr { }

kind NoneLiteral extends Literal { }

kind StringLiteral extends Literal {
    value: string
}

kind IntegerLiteral extends Literal {
    value: int
}

kind BooleanLiteral extends Literal {
    value: bool
}

kind Errors extends Node {
    errors: [SyntaxError]
}

kind SyntaxError extends Node {
    message: string
}
```

Figure 1: Kinds of JSON objects corresponding to AST nodes

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 than 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 provided samples:

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

6.3 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 be build-able, and the generated parser must be runnable and testable using the exact commands listed in this document.

6.4 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 late hours have been consumed (refer to the course website for grading 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?
2. What was the hardest language feature (not including indentation) to implement in this assignment? Why was it challenging?
3. What kind of parse errors does your parser detect? In what situations can it recover? Explain with the help of examples from the custom tests that you have written.

7 Implementation Notes

JFlex 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. Hint: Read about `%state`, `YYINITIAL`, and the `yypushback()` method for controlling the behavior of the lexer.

CUP 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.

Verbose debugging You can run `mvn clean package -Ddebug=true` 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`. You can also provide the `--debug` parameter to the `StudentParser` when running your parser on some input file; this flag will tell the generated parser to print information about shifts and reduces performed during parsing of an input program.

8 Submission

Submitting your completed assignment requires the following steps:

- Run `mvn clean` to rid your directory of any unnecessary files.
- Add and commit all your progress and push changes to the repository. Run `git commit` followed by `git push origin` to achieve this.
- 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.
- Push the tag using `git push origin pa1final`.

9 Grading (50 points)

The grading rubric is as follows

- 42 points for autograder tests (1 point per correct test / 42 tests)
- 4 points for the README
  - 4: thorough discussion of design decisions (including handling of indentation) and choice of test cases.
  - 2: vague or hard to understand; omits important details
  - 0: little to no effort
- 4 points for code cleanliness:

- 4 : 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
- 2 : effort made but imprecise or lacking in quality
- 0 : little to no effort to organize and document code

9.1 Extra credit: Bug reports

The reference implementation possibly contains some bugs. If you find a bug, report it by making a post on Piazza with a sample input program and describe how the expected output should differ. The first student/team to report a bug gets extra credit (5 points per unique bug with a maximum of 20 extra credits per team).

Bugs in the reference implementation are defined as (1) unexpected exceptions being reported or (2) violations of the specifications of the assignment or the specifications of the ChocoPy manual, which would lead to incorrect results. Note that minor mistakes in the ChocoPy manual or this document itself are not considered bugs in the reference implementation, though we would appreciate any such feedback.

The decision on whether to accept a bug report as valid and distinct from previous bug reports is at the discretion of the instructors.