CS 164 : Programming Languages and Compilers

PA6: A Type-Checker for CLASSES

Professor: Koushik Sen, prepared by Alex Reinking *Due: 4/24/17*

1. Introduction

In this homework assignment, you will implement the type-checker for the CLASSES programming language, described below. As with the previous typechecker, you will annotate the types of expression nodes with their types.

1.1. **Scope and Collaboration.** As before, you may share knowledge of the workings of Atrai with your classmates, as well as discuss general strategies for implementation. However, all code must be written independently, and **you must list all of your collaborators** in your `README`.

1.2. **Setup.** You will write your type-checker in Java. The staff has supplied a skeleton for you to complete on GitHub. While logged into your GitHub student account, visit `https://tinyurl.com/cs164s17pa6`. It will redirect you to GitHub classroom. Accept the assignment, then run `git clone https://github.com/cs164spring17/pa6-<username>.git`, where `<username>` is your GitHub username. The cloned repository will contain all the necessary files for the assignment.

1.3. **Software.** To develop locally, you will need the following software installed:

- JDK 1.8 (or newer). Available from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
- Apache Maven 3.3.9 (or newer). Available from `https://maven.apache.org/download.cgi`
- JUnit 4. This dependency will be installed automatically by Maven, but the documentation is available here: `http://junit.org/junit4/`
- (optional) A Java IDE such as IntelliJ IDEA (free community edition; ultimate edition free for students) that can manage Maven projects.

If you are running Mac or Linux, we suggest installing these tools from your package manager, for example Homebrew on Mac, or Apt on Debian/Ubuntu. Your mileage may vary with Chocolatey on Windows or Apt on the Windows Subsystem for Linux (WSL). We recommend, if possible, using an IDE on Windows. Be sure you are familiar with the new features in Java 8, including lambdas and static imports. A fairly comprehensive overview of the new features is available here: `https://leanpub.com/whatsnewinjava8/read`

1.4. **Testing.** This assignment uses Maven to build and run the tests. To do so, simply:

(1) Open a terminal and `cd` into your git repository
(2) Run `mvn clean test`. You should be in the same directory as `pom.xml`

This will take care of building all the source files beneath `src/main/java`, generating the ANTLR sources from `src/main/antlr4`, and executing all the tests beneath `src/test/java`.

Your code will be tested for correctness, meaning strict adherence to the type checking rules. You have been given 15 of the 50 tests that will be used to grade your code.

1.5. **Documentation.** In addition to tests, we have throughly annotated the APIs with Javadoc strings. The compiled javadocs are located in the `docs/javadocs` folder of the git repository. There is also a set of *slides* describing the Atrai API that is located in `docs/slides/atrai-slides.pdf`.

1.6. **Staff Solution.** To help you test your programs, we have set up a website that lets you run the official CLASSES type checker. The URL of the website is: `http://alexreinking.com/cs164/index.php`.

1.7. **Running your code.** You can run your own code by either adding tests to `src/test/java/atrai/interpreters/CLASSES/ClassesCheckerTest.java` and running `mvn clean test` or by running `mvn -DskipTests clean package` and executing the following command:

```
java -jar target/atrai-1.0.jar [parse|interpret]
atrai.interpreters.CLASSES.ClassesChecker <file name>
```

This command reads a CLASSES program from a file. Then, if you pass `parse` as the first argument, it will dump the untyped tree using the APIs you have been provided. If you instead pass `interpret` as the first argument, it will print the type-annotated tree generated by the checker. You can also run the provided TYPEDLET compiler by replacing `CLASSES.ClassesChecker` with `TYPEDLET.TypedLetChecker` in the command.

During debugging, you might find it helpful to pass the flag `-Ddebug1=true` or additionally `-Ddebug2=true` to the `java` command. You can also set these in the JVM flags area of your IDE's run-command dialog. These flags produce extra debugging output. Passing the `debug1` flag shows traces of the interpreter execution when rules match. Additionally passing `debug2` will show output for rules that don't match, helping you find bugs when a rule fails to match.

1.8. **Submission.** To submit, simply push to origin with `git push origin`. You must tag a commit with `pa6final` using the `git tag` command so we know which commit to grade. If you do not tag a commit, we will use the latest before the deadline, which could potentially be worse than a better, but slightly late, submission.

2. Files and Directories

This is a standard Maven project, so you will find the following files:

- `docs/javadocs` - This is where the pre-compiled Javadocs are.
- `docs/slides` - This folder contains the relevant lecture notes and slides on Atrai.
- `pom.xml` - This is the Maven project file. There should be no need to modify it.
- `src/main/antlr4` - This is where all the grammar files are located, including the grammar for CLASSES.
- `src/main/java` - This is where all the source is. It contains several packages:
  - `atrai.interpreters` - This package contains the example interpreters and some common utilities. You will place your code in `CLASSES/ClassesChecker.java`. There are some data structures you should use in this directory. You may also add more files to this directory.

  - `atrai.core` - This package contains a library for implementing tree transformations. All the other interpreters are built on top of this. Read the Javadocs and look at the example interpreters to understand its use.
  - `atrai.antlr` - This package contains generic utilities for processing ANTLR parse trees.
- `src/test/java/atrai/interpreters/CLASSES/ClassesCheckerPublicTest.java` - This file contains 15 test cases. The rest of the tests will go here during grading, so expect this file to be overwritten. Feel free to add more tests to this file.

The project should compile as-is, but will always reject inputs, and will not pass any test cases. We will overwrite every existing file, except for `ClassesChecker.java` and any new files you add in that same directory.

3. CLASSES Grammar

The ANTLR grammar for CLASSES is reproduced below.

```
prog: cls*;

cls: 'class' type 'extends' type '{' field* method* '}';

field: iden ':' type ';';

method: iden '(' paramlist ')' ':' type '=' expr ';';

expr: 'new' type
    | expr '[' type ']' '.' iden '('  arglist ')'
    | expr '.' iden '('  arglist ')'
    | expr ('*' | '/') expr
    | expr ('+' | '-') expr
    | expr ('=='|'!='|'>'|'<'| '>=' | '<=') expr
    | iden '=' expr
    | 'let' iden ':' type '=' expr 'in' expr
    | 'if' expr 'then' expr 'else' expr
    | 'if' expr 'is' iden ':' type 'then' expr 'else' expr
    | 'while' expr 'do' expr
    | 'print' expr
    | 'isnull' expr
    | 'self'
    | num
    | iden
    | string
    | '{' exprseq '}'
    | '(' expr ')'
    ;

paramlist: param (',' param)* | ;
param: iden ':' type;
arglist: expr (',' expr)* | ;
```

```
exprseq: expr (';' expr)*;

type: 'int' | 'object' | 'boolean' | 'string' | ID ;
num: INT;
iden: ID;
string: STRING;

ID  :   ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
INT :   '0'..'9'+ ;
WS  :   (' '|'\t' | '\r' | '\n')+ -> skip ;
STRING : '"' (~'"')* '"';
```

4. Grading - 60 points

The rubric is as follows:

- 50 points. One point per automated test, including the 15 provided tests.
- 5 points. General code cleanliness and style. Clear variable names, consistent spacing and brace conventions, no monolithic functions, etc.
- 5 points. README containing a description of any challenges you faced while completing this assignment.

5. Typing Rules

The typing rules were covered in detail in lecture and in the slides which are available in `docs/slides/lecture16classes.pdf`. The basic idea is that the type hierarchy, along with the method and field types for each class, is collected in a first pass over the program. Then, with that information fixed, the rest of the program is type-checked. In a departure from TYPEDFUN, you should represent types as simple Java Strings, since the helper classes we provide store string representations of types.

There are many error cases, so be thorough in your testing, and try to stress the staff solution online. The complete information is in the slides, but the most important are:

(1) All identifiers are declared
(2) Types match everywhere
(3) A class must be defined before it is extended
(4) Classes may be defined only once
(5) Methods and attributes in a class defined only once
(6) Attributes from super classes cannot be redefined
(7) Methods from super classes can be redefined provided that the signature is identical
(8) Reserved identifiers are not misused
(9) Assignment to self must be disallowed
(10) No declared identifier name should be a keyword
(11) Two parameters cannot have the same name in a method
(12) One must not extend T or call new T where T is int, boolean, or string

In this assignment, there are two exception types that must be thrown appropriately: SemanticException and SubtypeError. SemanticExceptions are thrown when a name cannot be resolved, when an incorrect number of arguments is passed to a function, when a primitive

value is extended or passed to "new", or when a name is improperly redefined. In any other case where two types do not match, a SubtypeError should be thrown instead.

6. Simple Example

As before, you will be wrapping expressions with typed nodes to store inferred type information. The following is a class with a single field and a getter and setter for that field.

```
class X extends object {
  a: int;
  get(): int = a;
  set(x:int): int = (a = x);
}
```

Then the complete annotated tree looks like this:

```
(%CLS 0 prog
    (%CLS 1 cls
        class (%CLS 2 type X%) extends (%CLS 3 type object%) {
        (%CLS 4 field
            (%CLS 5 iden a%) : (%CLS 6 type int%) ;
        %)

        (%CLS 7 method
            (%CLS 8 iden get%) ( ) : (%CLS 9 type int%) =
            (%CLS 11 typed
                (%CLS 11 iden a%) int
            %)
             ;
        %)

        (%CLS 12 method
            (%CLS 13 iden set%) (
            (%CLS 14 paramlist
                (%CLS 15 param
                    (%CLS 16 iden x%) : (%CLS 17 type int%)
                %)
            %)
             ) : (%CLS 18 type int%) =
            (%CLS 20 typed
                (%CLS 20 expr
                    (%CLS 21 iden a%) =
                    (%CLS 23 typed
                        (%CLS 23 iden x%) int
                    %)
                %)
                 int
            %)
             ;
        %)
         }
```

%)

%)