CS 164 : Programming Languages and Compilers

PA2: An Interpreter for the LETREC Language

Professor: Koushik Sen, prepared by Alex Reinking *Due: 3/2/17*

1. Introduction

In this homework assignment, you will implement the Interpreter for the LETREC programming language, described below. You have been supplied with a grammar, the operational semantics, and extensive documentation the Atrai tree transformation API, which you will use to complete the interpreter. For your benefit, we have included examples of writing interpreters for a simpler language, LET.

1.1. Scope and Collaboration.

This project should not require more than 50-75 lines of code. However, the Atrai API is large and complicated, and will require some study of the provided tutorials, javadocs, and perhaps source code. Thus, more than for any other assignment, we **insist** that you start early. 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 interpreter 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/cs164s17pa2`. It will redirect you to GitHub classroom. Accept the assignment, then run `git clone https://github.com/cs164spring17/pa2-<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 2.2.1 (or newer, 3.x.y recommended). 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 operational semantics. You have been given 10 of the 30 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. As an aside, you can build the javadocs yourself by running

```
mvn clean javadoc:javadoc
```

This will produce a small website in `target/site/apidocs/index.html`. If you open that file in your browser, you can peruse the documentation.

There is also a set of *slides* 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 a LETREC program with the staff's code. The URL of the website is: `http://alexreinking.com/cs164/index.php`.

1.7. **Running your code.** Similarly, you can run your own code by either adding tests to `src/test/java/atrai/interpreters/LETREC/LetrecInterpreterTest.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.LETREC.LetrecInterpreter <file name>
```

This command reads a LETREC 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 result of parsing and interpreting the whole program. You can also run the provided LET interpreter by replacing `LETREC.LetrecInterpreter` with `LET.LetInterpreter` in the command. In general, you can run any class implementing the `Interpreter` interface by supplying the fully-qualified class name.

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 first flag (`debug1`) 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 `pa2final` 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.
- `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. You can see example grammars in `LET.g4` and `LETREC.g4`.
- `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 `LETREC/LetrecInterpreter.java`. You may 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/LETREC/LetrecInterpreterTest.java` - This file contains 10 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 `LetrecInterpreter.java` and any new files you add in that same directory.

3. Grading - 40 points

The rubric is as follows:

- 30 points. One point for every automated test. You have been given 10 tests to help you debug simple cases.
- 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.

4. LETREC Syntax

You will implement the LETREC language, which has arithmetic and logical expressions, branching, and functions. As a guide, you should consult the slides from Lecture 8 (included in the `docs/slides/` folder). We have also provided a full interpreter for the LET language, which you should use as a guide for structuring your code. You can find that code in `src/main/java/atrai/interpreters/LET/LetInterpreter.java`. The grammar for the LETREC language is given below in ANTLR syntax.

```
1 prog: expr;
2
3 expr: num
4     | iden
```

```
5       | '(' expr ')'
6       | '(' expr expr ')'
7       | expr ('*' | '/') expr
8       | expr ('+' | '-') expr
9       | expr ('=='|'!='|'>'|'<'| '>=' | '<=') expr
10      | 'let' iden '=' expr 'in' expr
11      | 'if' expr 'then' expr 'else' expr
12      | 'letrec' iden '(' iden ')' '=' expr 'in' expr
13      ;
14
15  num: INT;
16
17  iden: ID;
18
19  ID  :   ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
20  INT :   '0'..'9'+ ;
21  WS  :   (' '|'\t' | '\r' | '\n')+ -> skip ;
```

5. Operational Semantics of LETREC

We will describe the semantics of LETREC informally first, to give you a sense of the expected behavior. Then, we will give the formal operational semantics of LETREC to prevent any ambiguity.

5.1. **Literals.** Literal numbers evaluate to themselves without any side-effects. An identifier evaluates to its value in the appropriate environment. Note that there are no boolean literals, like True or False. The three types in the language are denoted $Int(x)$ for some integer x, $Bool(b)$ for b either **True** or **False**, and $Lambda(v, body, env)$ where v is the name of the bound variable (the argument), *body* is the body of the function expression, and *env* is the environment in which it was created.

5.2. **Simple Expressions.** Function application, arithmetic and logical comparisons all evaluate their arguments eagerly and behave as they normally would in Java, or another conventional programming language.

$$\text{ARITH-OP}\frac{E \vdash e_1 : Int(v_1) \quad E \vdash e_2 : Int(v_2) \quad v_1 \ op \ v_2 = v \quad op \in \{+, -, *, /\}}{E \vdash e_1 \ op \ e_2 : Int(v)}$$

$$\text{LOGIC-OP}\frac{E \vdash e_1 : Int(v_1) \quad E \vdash e_2 : Int(v_2) \quad v_1 \ op \ v_2 = b \quad op \in \{==, !=, >, <, >=, <=\}}{E \vdash e_1 \ op \ e_2 : Bool(b)}$$

Here, you should read "$op \in \{+, -, *, /\}$" as "op is one of addition, subtraction, multiplication, or division".

5.3. **Conditionals.** If expressions evaluate their conditions, and depending on whether the condition is true, decide to either evaluate their "then" branch (true) or their "else" branch (false). The overall value of the if expression is the value of the branch it evaluated.

$$\text{IF-TRUE}\frac{E \vdash e_1 : Bool(True) \quad E \vdash e_2 : v}{E \vdash if \ e_1 \ then \ e_2 \ else \ e_3 : v}$$

$$\text{IF-FALSE}\frac{E \vdash e_1 : Bool(False) \qquad E \vdash e_3 : v}{E \vdash if\ e_1\ then\ e_2\ else\ e_3 : v}$$

5.4. **Let and Letrec.** Let and letrec expressions evaluate their arguments eagerly. Let defines values, while letrec defines lambdas. The bindings created by these expressions can only be used inside the "in" part of the expression. However, other "let" bindings can *shadow* existing bindings within their scope. So an interior let can temporarily redefine an existing binding in its expression. The precise semantics for this are:

$$\text{LET}\frac{E \vdash e_1 : v_1 \qquad [id = v_1]E \vdash e_2 : v_2}{E \vdash let\ id = e_1\ in\ e_2 : v_2}$$

and

$$\text{LETREC}\frac{E' = [id_1 = Lambda(id_2, e_1, E')]E \qquad E' \vdash e_2 : v}{E \vdash letrec\ id_1(id_2) = e_1\ in\ e_2 : v}$$

5.5. **Function Application.** Once a lambda has been created in a letrec binding, it may be applied to another expression within the body of the letrec. It works essentially by binding the actual argument value to the parameter name in the saved environment, and then evaluating the lambda's body in that environment.

$$\text{FUN-APP}\frac{E \vdash e_1 : Lambda(id, e', E') \qquad E \vdash e_2 : v \qquad [id = v]E' \vdash e' : v'}{E \vdash (e_1\ e_2) : v'}$$