**CS 164 : Programming Languages and Compilers**

# PA5: A Compiler for the TYPEDFUN Language

*Professor: Koushik Sen, prepared by Alex Reinking* *Due: 4/3/17*

## 1. Introduction

In this homework assignment, you will implement the compiler for the TYPEDFUN programming language, described below. You will build off of your TYPEDFUN implementation. In the interest of avoiding "double-jeopardy", we have provided a JAR of the staff solution for TYPEDFUN, as COMPILEDFUN relies on the typed parse trees from PA4.

You will transform the parse tree to a low-level sequence of Java commands. This sequence is inserted into a template of code we provide, along with the code for filling in the template.

### 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 compiler 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/cs164s17pa5`. It will redirect you to GitHub classroom. Accept the assignment, then run `git clone https://github.com/cs164spring17/pa5-<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 operational semantics. 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 TYPEDFUN compiler. 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/COMPILEDFUN/FunCompilerTest.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.COMPILEDFUN.FunCompiler <file name>
```

This command reads a TYPEDFUN 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 Java code generated by the compiler. You can also run the provided TYPEDLET compiler by replacing `COMPILEDFUN.FunCompiler` with `COMPILEDLET.LetCompiler` 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 `pa5final` 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 TYPEDFUN.
- `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 `COMPILEDFUN/FunCompiler.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/COMPILEDFUN/FunCompilerPublicTest.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 `FunCompiler.java` and any new files you add in that same directory.

## 3. Compiler Architecture

We've given you a bit more scaffolding this time than in previous assignments. In particular, you should study the code in `src/main/java/atrai/interpreters/common/CodeTemplate.java`, which handles generating and dynamically executing your Java code. If you wish to implement more advanced features + optimizations, you should do so by subclassing the classes declared in this file.

CodeTemplates are constructed from a flat tree that acts as a linked list. That is, your transformer should return a tree with only leaves as children. Each of those children should contain exactly one Low-Level Java (LLJ, described in the slides) instruction. Its `addLambda` function helps you add lambdas to the generated output.

Specifically, the generated code must match the code generated by the reference implementation. The only allowable LLJ commands are those on slide 17 of the CGen slides. You may write the optimization to get rid of the stack pointer (sp), but you must do so in a separate class. Document how to call it in your README.

### 3.1. Semantics and Typing.

The operational semantics and typing rules are the same here as they were in PA3 and PA4, respectively. Those PDFs are available in `docs/PA3.pdf` and `docs/PA4.pdf`. These semantics are meant to give an exact meaning to each construction in the language, but are not meant to prescribe an implementation. In some cases, you might find that an equivalent implementation is easier to write than a strict adherence to the inference rules.

### 3.2. Additional Reading / Resources.

We have provided the lecture slides for code generation in `docs/slides/lecture13cgen.pdf`. You are expected to study these **carefully** as well as the example compiler for LET in order to do this assignment. Again, you may only generate code using the LLJ instructions on slide 17.

## 4. Atrai Update

In the previous assignment, we introduced the `@*_` and `$*n` functions for processing lists of children. In this assignment we extend Atrai's templates with one more special symbol: `$$n`. This symbol is replaced in a template by the children of the associated tree, thus excluding the root. For example, in the LetCompiler, we process addition statements using the following code snippet:

```
transformer.addTransformer("(%LET @_ expr @_ + @_%)", (c, E) -> {
  c[2] = transformer.transform(c[2], E);
  c[3] = transformer.transform(c[3], E);
}, "(%$$2 (%stack[sp++].i=a0.i;%) $$3 (%t1.i=stack[sp-1].i;%)
   (%a0.i=t1.i+a0.i;%) (%sp--;%)%)");
```

Here, the second capture group gets transformed into a one-level tree containing LLJ instructions. In the template, `n` gets replaced by all of those children so that the result of this transformer is also a one-level tree.

For a concrete example, the input: `(% (%a b c%)(%d e f%)%)` would match the pattern `(% @_ @_ %)` and would complete the template `(% $$1 $$2 %)` as `(% a b c d e f %)`

## 5. 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.