CS 164 : Programming Languages and Compilers

PA3: An Interpreter for the FUN Language

Professor: Koushik Sen, prepared by Alex Reinking *Due: 3/13/17*

1. Introduction

In this homework assignment, you will implement the Interpreter for the FUN 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. You should use your code for LETREC as a starting point, but note there are differences in the grammar.

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 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/cs164s17pa3`. It will redirect you to GitHub classroom. Accept the assignment, then run `git clone https://github.com/cs164spring17/pa3-<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 a FUN 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/FUN/FunInterpreterTest.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.FUN.FunInterpreter <file name>
```

This command reads a FUN 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 `FUN.FunInterpreter` 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 `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 `pa3final` 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 FUN.
- `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 `FUN/FunInterpreter.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/FUN/FunInterpreterTest.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 `FunInterpreter.java` and any new files you add in that same directory.

3. FUN Examples

FUN is a language with arithmetic and logical expressions, mutable state, conditionals, loops, and functions. It is similar to LETREC and REFS, but differs in a few ways. Most importantly, the function declaration and application syntaxes are more C-like. We demonstrate with a short factorial function.

```
let fact = fun(x) =
  let prod = 1
   in { while x > 1 do {
          prod = prod * x;
          x = x - 1
        }; prod }
in fact(4)
```

This program evaluates to 24. Notice that it is able to change the value of its argument. However, the function does not have access to `fact`, since `fact` does not exist at the point of the function's declaration. Thus, to write this function recursively, you must make use of mutable state, like so:

```
let factrec = null,
    fact = fun(x) = if x == 0 then 1 else x * factrec(x - 1)
in { factrec = fact; print "Result: " + fact(4) }
```

This allows fact to call itself recursively by way of an auxiliary binding. This example also shows several more features of FUN: printing, string concatenation with implicit casting, null values, and multiple let declarations.

4. Atrai Update

It is now possible to use Kleene stars in grammars with Atrai. When such a construction is used, you may call the `iterate()` function to process its children. The iterate function accepts a lambda, an initial reduction value, and an arbitrary object called the 'context'. The lambda is called on each child, left-to-right, and receives three arguments: first, the child; second, the current reduction value; finally, the context. The lambda is expected to return an updated reduction value.

As an example, you can print all of the children of a sequence operator using the rule:

```
transformer.addTransformer("(%FUN @_ expr { @_ }%)", (c, E) -> {
  return  n(c[2]).iterate((child, redValue, environment) -> {
    System.out.printf("Child %d: %s%n", redValue, String.valueOf(child));
    return (Integer)redValue + 1; // increment the child counter
  }, 0, null);
});
```

5. FUN Syntax

You will implement the FUN language, which has arithmetic and logical expressions, assignment, branching, loops, and functions. The grammar for FUN is reproduced below.

```
1  prog: expr
2
3  expr: expr '(' expr? ')'
4      | expr ('*' | '/') expr
5      | expr ('+' | '-') expr
6      | expr ('==' | '!=' | '>' | '<' | '>=' | '<=') expr
7      | iden '=' expr
8      | 'let' decllist 'in' expr
9      | 'if' expr 'then' expr 'else' expr
10     | 'while' expr 'do' expr
11     | 'fun' '(' iden ')' '=' expr
12     | 'fun' '(' ')' '=' expr
13     | 'print' expr
14     | num
15     | string
16     | 'null'
17     | iden
18     | '{' exprseq '}'
19     | '(' expr ')'
20
21 decllist: decl (',' decl)*
22 decl: iden '=' expr
23 exprseq: expr (';' expr)*
24
25 num:[0-9]+;
26 iden: [a-zA-Z_][a-zA-Z0-9_]*;
27 string: "[^"]*";
```

6. Operational Semantics of FUN

Here is an informal description of the operational semantics of FUN, coupled with the rigorous deduction rules. E refers to the environment, which is a mapping of variable names to memory locations. S and sometimes T are stores, which are mappings from memory locations to values.

6.1. **Literals.** Literal numbers, strings, and null all evaluate to themselves without any side-effects. An identifier evaluates to its value in the current environment and store.

$$\text{NUM}\frac{}{E, S \vdash num : Int(num), S} \qquad \text{STRING}\frac{}{E, S \vdash string : String(string), S}$$

$$\text{NULL}\frac{}{E, S \vdash \texttt{null} : Null, S} \qquad \text{IDEN}\frac{l = E(id) \qquad v = S(l)}{E, S \vdash id : v, S}$$

6.2. **Simple Expressions and Sequencing.** The comparison operators are valid only on integers and operate in the environment produced after their arguments' evaluation.

$$\text{LOGIC}\frac{\begin{array}{c}E, S \vdash e_1 : Int(v_1), S_1 \qquad E, S_1 \vdash e_2 : Int(v_2), S_2 \\ b = v_1 \ \texttt{op}\ v_2 \qquad \texttt{op} \in \{==, !=, <, >, <=, >=\}\end{array}}{E, S \vdash e_1 \ \texttt{op}\ e_2 : Bool(b), S_2}$$

The arithmetic are valid only on integers, with the exception of plus. For two integers, the plus operator computes their sum. However, if at least one is a string, then both are converted to strings and concatenated.

$$\text{ARITH}\frac{\begin{array}{c}E, S \vdash e_1 : Int(v_1), S_1 \qquad E, S_1 \vdash e_2 : Int(v_2), S_2 \\ v = v_1 \ \texttt{op}\ v_2 \qquad \texttt{op} \in \{+, -, *, /\}\end{array}}{E, S \vdash e_1 \ \texttt{op}\ e_2 : Int(v), S_2}$$

$$\text{CONCAT-L}\frac{\begin{array}{c}E, S \vdash e_1 : String(s_1), S_1 \qquad E, S_1 \vdash e_2 : v_2, S_2 \\ s_2 = \texttt{String.valueOf}(v_2) \qquad s = s_1 s_2\end{array}}{E, S \vdash e_1 + e_2 : String(s), S_2}$$

$$\text{CONCAT-R}\frac{\begin{array}{c}E, S \vdash e_1 : v_1, S_1 \qquad E, S_1 \vdash e_2 : String(s_2), S_2 \\ s_1 = \texttt{String.valueOf}(v_1) \qquad s = s_1 s_2\end{array}}{E, S \vdash e_1 + e_2 : String(s), S_2}$$

The concatenation rules use the `toString` method in Java to produce strings from Booleans and Integers. Thus `(0==0) + "foo"` evaluates to "truefoo". Since at least one of the operands must be a string to concatenate, `(0==0) + 1` is invalid.

The sequencing construction (semicolon-delimited exprs in curly braces) evaluates its arguments left-to-right and returns the last one.

$$\text{SEQ}\frac{E, S \vdash e_1 : v_1, S_1 \qquad E, S_1 \vdash e_2 : v_2, S_2 \qquad \cdots \qquad E, S_{n-1} \vdash e_n : v_n, S_n}{E, S \vdash \{e_1; e_2; \cdots; e_n\} : v_n, S_n}$$

Lastly, assignment looks up the given location for the given variable in the environment, evaluates the right hand side, then updates the store. The whole expression evaluates to the

right hand side:

$$\text{ASSN}\frac{E, S \vdash e : v : S_1 \quad l = E(id) \quad S_2 = [l = v]S_1}{E, S \vdash id = e : v, S_2}$$

All of these constructs evaluate their arguments left to right, so that `(x = 3) + (x = 5)` evaluates to 8 and the value of `x` is 5 after the operation.

6.3. **Print.** Print first evaluates its arguments, then prints the values to `System.out`. It evaluates to its argument. This is like `trace` in Haskell.

$$\text{PRINT}\frac{E, S \vdash e : v, S' \quad \texttt{System.out.println}(v)}{E, S \vdash \texttt{print}\ e : v, S'}$$

6.4. **Lets and Branching.** Let expressions again evaluate their arguments left to right and assign values to their variables eagerly. Thus an expression like `let x=1, y=x in x+y` evaluates to 2. Variables in let branches override existing names in their scope only. Thus, the expression `let x = 1 in {print x; let x = 2 in print x; print x}` prints "121" and evaluates to 1.

$$\text{LET}\frac{\begin{array}{cccc} E, S \vdash e_1 : v_1, T_1 & l_1 = newloc(T_1) & E_1 = [id_1 = l_1]E & S_1 = [l_1 = v_1]T_1 \\ \vdots & \vdots & \vdots & \vdots \\ E_{n-1}, S_{n-1} \vdash e_n : v_n, T_n & l_n = newloc(T_n) & E_n = [id_n = l_n]E_{n-1} & S_n = [l_n = v_n]T_n \end{array} \quad E_n, S_n \vdash e : v, S'}{E, S \vdash \texttt{let}\ id_1 = e_1\ \cdots\ id_n = e_n\ \texttt{in}\ e : v, S'}$$

"If" expressions first evaluate their conditions and if the condition is true, they evaluate to their "then" branch and leave the "else" branch unevaluated. Respectively, the "then" branch is unevaluated and the "false" branch returned when the condition is false.

$$\text{IF-TRUE}\frac{E, S \vdash e_1 : Bool(True), S_1 \quad E, S_1 \vdash e_2 : v_2, S_2}{E, S \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : v_2, S_2}$$

$$\text{IF-FALSE}\frac{E, S \vdash e_1 : Bool(False), S_1 \quad E, S_1 \vdash e_3 : v_3, S_3}{E, S \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : v_3, S_3}$$

While-loops first evaluate their condition in the current environment. If the condition is true, the loop then evaluates its body and repeats the condition test in the new environment. Once the condition evaluates to false, the while loop evaluates to null.

$$\text{WHILE-T}\frac{E, S \vdash e_1 : Bool(True), S_1 \quad E, S_1 \vdash e_2 : v_2, S_2 \quad E, S_2 \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2 : Null, S'}{E, S \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2 : Null, S'}$$

$$\text{WHILE-F}\frac{E, S \vdash e_1 : Bool(False), S_1}{E, S \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2 : Null, S_1}$$

6.5. **Functions.** A function declaration creates a new lambda, which saves the name of its bound variable, its body, and the current environment. Unlike in LETREC, the lambda is permitted to have no argument, as well.

$$\text{FUN}\frac{}{E, S \vdash \texttt{fun}\ (id) = e : Lambda(E, id, e), S}$$

$$\text{FUN-NA}\frac{}{E, S \vdash \texttt{fun}\ () = e : Lambda(E, e), S}$$

When a function is evaluated, the expression returning the function is evaluated first, then the argument is evaluated if it exists. The argument binding is added to the environment and store in a new location, and the body of the lambda is evaluated in the updated context.

$$\text{APP}\ \frac{\begin{array}{c}E, S \vdash e_1 : Lambda(id, e, E'), S_1 \qquad E, S_1 \vdash e_2 : v_2, S_2 \\ l = newloc(S_2) \qquad [id = l]E', [l = v_2]S_2 \vdash e : v, S'\end{array}}{E, S \vdash e_1(e_2) : v, S'}$$

$$\text{APP-NA}\ \frac{E, S \vdash e_1 : Lambda(e, E'), S_1 \qquad E', S_1 \vdash e : v, S'}{E, S \vdash e_1() : v, S'}$$

Note that the function application syntax has changed. Now, you may write expressions like `(fun(x)=x+1)(1)` which will evaluate to 2. There is also no more letrec keyword. Functions can be bound do names using the let syntax, like any other declaration.

7. 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.