Project 5: Typed Jam

Points: 100

Provided Code

In contrast to earlier assignments, the only provided code for this project is a skeleton Junit test file **`Assign5Test.java`** for this project. For this assignment, you must revise the code from your solution to Project 4.

Overview

Task Your assignment is to convert Jam to a typed programming language.

Step 1 As the first step in the project, you will strip your solution for Project 4 down to a family of two interpreters: call-by-value/eager (value/value) and call-by-value/lazy (value/need).

The public interface to the interpreter is the same as in Assignment 4 except for the fact the the **`Interpreter`** class will only support two public methods: **`eagerEval()`** and **`lazyEval()`** which correspond to **`valueValue()`** and **`valueNeed()`** in Assignment 4.

Save you solution to this step in a separate directory. You will use it as the starting point for Project 6.

Step 2 As the second step of the assignment, you will modify the parser and AST representation to

- eliminate the primitives **`list?`**, **`number?`**, **`function?`**, **`ref?`**, and **`arity`** from the language syntax
- require the syntax **`Id : Type`** for *binding occurrences* of variables, which occur on the left-hand side of definitions and in the parameter lists of **`map`** expressions
- require the syntax **`empty : Type`** in place of **`empty`**, where Type indicates the element type of the list type to which **`empty`** belongs
- force any **`Factor`** that is a **`Prim`** to be followed by an application argument list. This restriction prevents a **`Prim`** from being used as a general value.

Hence, the grammar looks like this:

Expressions

```
Exp               ::= Term { Binop Exp }
                    | if Exp then Exp else Exp
                    | let Def+ in Exp
                    | map TypedIdList to Exp
                    | "{" PropStatementList "}"
Term              ::= Unop Term
                    | Factor { ( ExpList ) }
                    | Prim ( ExpList )
                    | Empty
                    | Int
                    | Bool
Factor            ::= ( Exp ) | Id
ExpList           ::= { PropExpList }
PropExpList       ::= Exp | Exp , PropExpList
TypedIdList       ::= { PropTypedIdList }
PropTypedIdList   ::= TypedId | TypedId, PropTypedIdList
TypedId           ::= Id : Type
PropStatementList ::= Exp | Exp ; PropStatementList
```

Definitions

```
Def ::= TypedId := Exp ;
```

Primitive Constants, Operators, and Operations

```
Empty  ::= empty : Type
Bool  ::= true | false
Unop  ::= Sign | ~ | ! | ref
Sign  ::= "+" | -
Binop ::= Sign | "*" | / | = | != | < | > | <= | >= | & |
"|" | <-
Prim  ::= empty? | cons? | cons | first | rest
```

Identifiers

```
Id ::= AlphaOther {AlphaOther | Digit}*
```

Please note that `Id` does *not* contain the anything that matches `Prim` or the keywords **`if`**, **`then`**, **`else`**, **`map`**, **`to`**, **`let`**, **`in`**, **`empty`**, **`true`**, and **`false`**.

Types

```
Type         ::= unit
               | int
               | bool
               | list Type
               | ref Type
               | (TypeList -> Type)
TypeList     ::= { PropTypeList }
PropTypeList ::= Type | Type, PropTypeList
```

Numbers

```
Int ::= Digit+
```

Note that there are three type constructors **`list`**, **`->`**, and **`ref`**. You will need to define an abstract syntax for types.

You are not responsible for producing slick error diagnostics for syntax errors. You may still prohibit the omitted primitive operations recognized by the lexer/parser from being used as variable names.

Step 3 Before you can interpret a typed program, you must type check it.

Define a static method in the **`Interpreter`** class called **`checkTypes`** that takes an expression (an AST) as an argument, type checks it, and returns an expression suitable for interpretation. You can simply use inheritance to define an AST representation for typed program syntax that extends the untyped AST representation. If your type checker encounters a type error, it should abort the computation by throwing a **`TypeException`** with an error message explaining the type error. **`TypeException`** looks like this:

```
class TypeException extends RuntimeException {
    TypeException(String msg) { super(msg); }
}
```

The top-level methods **`eagerEval()`** and **`lazyEval()`** should first invoke the parser, then the context-sensitive syntax checker from Assignment 3, then the type checker, and finally the appropriate evaluation method.

The type-checking method should rely on the same tree-walking methods that you developed for checking syntax in Assignment 3. It behaves like an "abstract" interpreter that takes a type environment (mapping variables to types) as input and returns a type instead of a value as output. Note that the primitive operations and operators all have types. Moreover, the primitives **`cons`**, **`first`**, **`rest`**, **`empty?`**, and **`cons?`** and and the operators

!, **ref**, **<-**, **=** have schematic types that must be matched against their contexts to determine
how they are instantiated:

```
cons      : 'a , list 'a -> list 'a
first     : list 'a      -> 'a
rest      : list 'a      -> list 'a
ref       : 'a           -> ref 'a
empty?    : list 'a      -> bool
cons?     : list 'a      -> bool
!         : ref 'a       -> 'a
<-        : ref 'a , 'a  -> unit
=         : 'a , 'a      -> bool
!=        : 'a , 'a      -> bool
```

In these type declarations, the symbol **'a** stands for any specific type (a specific type cannot contain the special symbol **'a**). Only these primitives and binary operators have polymorphic type; every program-defined function must have a specific (non-polymorphic) type. The symbol **'a** designating an undetermined type never appears in actual program text.

To simplify the type checking process, the revised syntax prohibits these polymorphic operations from being used as data values (stored in data structures, passed as arguments, or returned as results). To use cons as a data value, you must wrap it in a map which forces it to have a specific type. Hindley-Milner polymorphism (used in ML) is slightly more flexible; it allows you to use polymorphic operations as data values, but their reconstructed types must not be polymorphic!

From the perspective of type-checking, the **if-then-else** construct behaves like a ternary polymorphic function with type

```
if-then-else : (bool, 'a, 'a -> 'a)
```

The remaining primitives and operators have the following specific types:

```
(unary) -  : (int -> int)
~          : (bool -> bool)
+          : (int, int -> int)
(binary) - : (int, int -> int)
*          : (int, int -> int)
/          : (int, int -> int)
<          : (int, int -> bool)
<=         : (int, int -> bool)
&          : (bool, bool -> bool)
|          : (bool, bool -> bool)
```

Some type correct applications of primitives can still generate run-time errors because the collection of definable types and type-checking machinery are too weak to capture the domain of primitive functions precisely. For example, the following expressions are all type-correct:

```
1/0
first(empty : int)
rest(empty : int)
```

To type the applications of polymorphic primitives and operators, your type checker will need to match their schematic types against the types the checker produces for their inputs. Every program sub-expression other than one of the schematic primitives or operators has a specific (non-schematic) type. We augmented **`empty`** with type annotation to obtain this property.

Consider the following example. Assume that the type environment assigns **`x`** the type **`int`**, then the unary operator **`ref`** in the application **`ref x`** has type **`(int -> ref int)`**.

Testing You are expected to write unit tests for all of your non-trivial methods or functions *and submit your test code as part of your program*. For more information, refer back to Assignment 1.

Test that your type checker works and produces code suitable for your interpreter. As usual we will provide a sample Unit testing suite to help you get started on writing your unit tests and sample grading scripts so you can test that your program supports the proper interfaces before submitting it for grading.

Testing and Submitting Your Program

The **`src`** folder in the provided code base only includes the Junit test file **`Assign5Test.java`**. Junit test files from earlier projects will generally fail because your Project 5 interpreter supports different (and fewer) evaluation methods. You obviously need to add more test cases to **`Assign5Test.java`** or create your own Junit test file for Project 5. We will evaluate the test coverage of your unit tests.

As always, each class and method in your program should be preceded by a short `javadoc` comment stating precisely what it does.

Edit the **`README`** file for your repository so that it

- outlines the organization of your program,

- specifies any compatible additions that you made the provided API (not generally necessary), and
- specifes what testing process you used to confirm the correctness of your program.

Note that the **`README`** file is a GitHub “MarkDown” file, text augmented by a few simple formatting annotations. MarkDown (MD) is much less powerful than HTML but much easier to use. The webpage **markdown-cheat-sheet** is a good summary for learning MarkDown and a useful reference.

Since your program resides in a repository owned by our classroom, you do not need to do anything to submit except push you the latest version of your local repository back to the origin on GitHub.

Please make sure to remove or disable all debugging output generated by program. Excessive use of print statements considerably slows down your interpreter and thus the testing of your project. If your interpreter generates a timeout error because debug output drastically slows it down, you may not get credit for the test cases that took too long.

Back to course website