CS 316 Fall 2017
Observe **course policies** in undertaking this project.
All programs must be written in Oracle Standard Edition compliant Java or ANSI/ISO standard compliant C++.

PROJECT 3: Type Checker
Due: 12/03/17, Sunday, 11 PM
Late projects will not be accepted.

The objective of this project is to extend the parser implemented in Project 2 by a type checker.

A commonly used, effective type checking method is by means of a *type evaluation* function. The type evaluation function serves as a formal definition of the type rules of a programming language and can be implemented as a type checker. The definition of the type evaluation function for our language, called *TypeEval*, is given below. To simplify the project, it uses rigid type rules to distinguish all three types *int*, *float*, *boolean* without incorporating the standard implicit conversion from *int* to *float* adopted in many languages.

Definition of TypeEval

The special symbol $\perp_t$ will represent the type error value. For each expression E in the category ⟨exp⟩, TypeEval(E) returns the type of E: *int*, *float*, *boolean*, or $\perp_t$. For each function definition F, TypeEval(F) returns $\perp_t$ or *correct*; the special value *correct* will represent the type correctness of F.

In the following definition, the arguments of TypeEval will be given in abstract syntax.

variables, constant literals

- TypeEval(*x*) = declared type of *x*, for each formal parameter variable *x*
- TypeEval(*c*) = type of *c*, for each constant literal *c*

conditional expressions

- TypeEval(if E_1 then E_2 else E_3) =
  if TypeEval(E_1) = boolean **then**
  {
  if TypeEval(E_2) = int and TypeEval(E_3) = int **then** int
  else if TypeEval(E_2) = float and TypeEval(E_3) = float **then** float
  else if TypeEval(E_2) = boolean and TypeEval(E_3) = boolean **then** boolean
  else $\perp_t$
  }
  else $\perp_t$

arithmetic expressions

- TypeEval((+ E_1 E_2 ··· E_n)) =
  if TypeEval(E_i) = int for all $1 \le i \le n$ **then** int
  else if TypeEval(E_i) = float for all $1 \le i \le n$ **then** float
  else $\perp_t$

Analogously for −, *, /.

boolean expressions

- TypeEval((or E_1 E_2 ··· E_n)) =
  if TypeEval(E_i) = boolean for all $1 \le i \le n$ **then** boolean
  else $\perp_t$

Analogously for "and".

- TypeEval((not E)) =
  if TypeEval(E) = boolean **then** boolean
  else $\perp_t$

comparison expressions

- TypeEval((< E_1 E_2)) =
  if TypeEval(E_1) ∈ {int, float} and TypeEval(E_2) ∈ {int, float} **then** boolean
  else $\perp_t$

  Analogously for <=, >, >=.

- TypeEval(E_1 = E_2) =
  if TypeEval(E_1) ∈ {int, float} and TypeEval(E_2) ∈ {int, float} **then** boolean
  else if TypeEval(E_1) = boolean and TypeEval(E_2) = boolean **then** boolean
  else $\perp_t$

user-defined function call expressions

- TypeEval((f E_1 E_2 ⋯ E_n)) =
  if TypeEval(E_i) = declared type of p_i for all $1 \le i \le n$ **then** declared return type of f
  else $\perp_t$
  where f is any user-defined function and p_i is the i-th formal parameter of f

function definitions

- TypeEval(return-type fun-name parameter-list = E) =
  if return-type = TypeEval(E) **then** correct
  else $\perp_t$

TypeEval is to be implemented by functions in the syntactic-category classes except for ⟨header⟩ and its component categories. The target objects of these TypeEval( ) functions will be actual parse trees instead of abstract syntax. For example, type evaluation of a parse tree object, *exp*, of ⟨exp⟩ will be performed by a call *exp*.TypeEval(...), where arguments for TypeEval will be suitable type maps described below.

Your program is to:

1. Read any text file that contains (what is intended to be) a string in the category ⟨fun def list⟩ and then construct an explicit parse tree as per Project 2.
2. Build necessary type maps of the function return types and the declared types of all parameters, and display them. You will need to build three type maps to record:
   a. the function names mapped to their declared return types
   b. within each function declaration, the formal parameter variables mapped to their declared types
   c. within each function declaration, the sequential positions of the formal parameter variables mapped to their types, to be used to type check user-defined function calls

   These maps are built by functions working on the parse tree for ⟨header⟩. Any reasonably efficient map data structure may be used for this purpose. I used HashMap for all three maps as follows:

```
static HashMap<String,TypeVal> funTypeMap = new HashMap<String,TypeVal>();
        // records declared return types of functions

static HashMap<String,HashMap<String,TypeVal>> paramTypeMap = new HashMap<String,HashMap<String,TypeVal>>();
        // for each function name, records declared types of formal parameters

static HashMap<String,HashMap<Integer,TypeVal>> paramNumTypeMap = new HashMap<String,HashMap<Integer,TypeVal>>();
        // for each function name, records declared types of i-th formal parameters, indexed by i
```

   You may use these three maps and this sample TypeVal class.
3. Apply TypeEval() to the parse tree for the entire ⟨fun def list⟩ to perform type checking.
4. Issue an appropriate error message when the first type error is found (no need to recover from it and continue checking).

The program need not perform semantic checking like detecting undeclared variables in expressions, formal parameter variables declared more than once, and incorrect numbers of arguments of function calls. Presume that all function definitions are properly formed in these regards. In particular, presume the following:

- The operators +, −, *, /, *or*, *and* are applied to two or more arguments.
- The operator *not* is applied to exactly one argument.
- All five comparison operators are applied to exactly two arguments.

To make grading efficient and uniform, the program is to read the input/output file names as external arguments to the main function. How to set external arguments to Java main function in Eclipse.

If your Project 2 program wasn't functional, you may use this sample program.

Here's a sample set of test input files:

in1 | out1
in2 | out2
in3 | out3
in4 | out4
in5 | out5
in6 | out6
in7 | out7
in8 | out8
in9 | out9
in10 | out10
in11 | out11
in12 | out12

You should make your own additional input files to test the program.

Submission

Email the following materials to keitaro.yukawa@gmail.com with the subject header:

CS 316, Project 3, your full name

- All the classes comprising your source code, including the lexical analyzer and parser you used. Since there will be many classes, **make sure to double check no classes are missing in your submission**.
- A list of all class names arranged like this page. This may be in text, html, PDF, or WORD file.
- Concise instructions for how to compile and run your program.

You may email the entire materials in a .zip or .rar compressed file.

The due date is 12/03/17, Sunday, 11 PM. No late projects will be accepted. If you haven't been able to complete the project, you may send an incomplete program for partial credit. In this case, include a description of what is and is not working in your program along with what you believe to be the sources of the problems.