CSE340 Spring B 2021 Project 3: Type Checking

Due: **Sunday, April 25, 2021 by 11:59 pm AZ time**

The goal of this project is to give you experience in Hindley-Milner type checking.

We begin by introducing the grammar of our language which is based on the previous project with additional constructs. Then we will discuss the semantics of our language and type checking rules. Finally we will go over a few examples and formalize the expected output.

1. Lexical Specification

Here is the list of tokens that your lexical analyzer should recognize (the new tokens are listed first):

```
INT = "int"
REAL = "real"
BOOL = "bool"
TRUE = "true"
FALSE = "false"
IF = "if"
WHILE = "while"
SWITCH = "switch"
CASE =  "case"
NOT = "!"
PLUS = "+"
MINUS = "-"
MULT = "*"
DIV = "/"
GREATER = ">"
LESS = "<"
GTEQ = ">="
LTEQ = "<="
NOTEQUAL = "<>"
LPAREN = "("
RPAREN = ")"
NUM = (pdigit digit*) + 0
REALNUM = NUM "." digit digit*
PUBLIC = "public"
PRIVATE = "private"
EQUAL = "="
COLON = ":"
COMMA = ","
SEMICOLON = ";"
LBRACE = "{"
RBRACE = "}"
ID = letter (letter + digit)*
```

2. Grammar

Here is the grammar for our input language:

```
program          -> global_vars body
global_vars      -> ε
global_vars      -> var_decl_list
var_decl_list    -> var_decl
var_decl_list    -> var_decl var_decl_list
var_decl         -> var_list COLON type_name SEMICOLON
var_list         -> ID
var_list         -> ID COMMA var_list
type_name        -> INT
type_name        -> REAL
type_name        -> BOOL
body             -> LBRACE stmt_list RBRACE
stmt_list        -> stmt
stmt_list        -> stmt stmt_list
stmt             -> assignment_stmt
stmt             -> if_stmt
stmt             -> while_stmt
stmt             -> switch_stmt
assignment_stmt -> ID EQUAL expression SEMICOLON
expression       -> primary
expression       -> binary_operator expression expression
expression       -> unary_operator expression
unary_operator   -> NOT
binary_operator -> PLUS | MINUS | MULT | DIV
binary_operator -> GREATER | LESS | GTEQ | LTEQ | EQUAL | NOTEQUAL
primary          -> ID
primary          -> NUM
primary          -> REALNUM
primary          -> TRUE
primary          -> FALSE
if_stmt          -> IF LPAREN expression RPAREN body
while_stmt       -> WHILE LPAREN expression RPAREN body
switch_stmt      -> SWITCH LPAREN expression RPAREN LBRACE case_list RBRACE
case_list        -> case
case_list        -> case case_list
case             -> CASE NUM COLON body
```

3. Language Semantics

3.1. Types

The language has three built-in types: `int`, `real` and `bool`.

3.2. Variables

Programmers can declare variables either explicitly or implicitly.

- Explicit variables are declared in an `var_list` of a `var_decl`.
- A variable is declared implicitly if it is not declared explicitly but it appears in the program body.

Example

Consider the following program written in our language:

```
x: int;
y: bool;

{
    y = x;
    z = 10;
    w = * z 5;
}
```

This program has four variables declared: `x`, `y`, `z`, and `w`, with `x` and `y` explicitly declared and `z` and `w` implicitly declared.

3.3. Type System

Our language uses structural equivalence for checking type equivalence. Implicit types will be inferred from the usage (in a simplified form of Hindley-Milner type inference).

Here are all the type rules/constraints that your type checker will enforce (constraints are labeled for reference):

- **C1:** The left hand side of an assignment should have the same type as its right hand side
- **C2:** The operands of a binary operator (`GTEQ` `PLUS`, `MINUS`, `MULT`, `DIV`, `GREATER`, `LESS`,), `LTEQ`, `EQUAL` and `NOTEQUAL` should have the same type (it can be any type)
- **C3:** The operand of a unary operator (`NOT`) should be of type `bool`

- **C4:** Condition of `if` and `while` statements should be of type `bool`
- **C5:** The expression that follows the `switch` keyword in `switch_stmt` should be of type `int`
- The type of expression `binary_operator op1 op2` is the same as the type of `op1` and `op2` if operator is `PLUS`, `MINUS`, `MULT` or `DIV`. Note that `op1` and `op2` must have the same type due to C2
- The type of expression `binary_operator op1 op2` is `bool` if operator is `GREATER`, `LESS`, `GTEQ`, `LTEQ`, `EQUAL` or `NOTEQUAL`
- The type of expression `unary_operator op` is `bool`
- `NUM` constants are of type `int`
- `REALNUM` constants are of type `real`
- `true` and `false` values are of type `bool`

4. Output

There are two scenarios:

- There is a type error in the input program
- There are no type errors in the input program

4.1. Type Error

If any of the type constraints (listed in the Type System section above) is violated in the input program, then the output of your program should be:

```
TYPE MISMATCH <line_number> <constraint>
```

Where `<line_number>` is replaced with the line number that the violation occurs and `<constraint>` should be replaced with the label of the violated type constraint (possible values are **C1** through **C5**). Note that you can assume that anywhere a violation can occur it will be on a single line.

4.2. No Type Error

If there are no type errors in the input program, then you should output type information for all variables in the input program in the order they appear in the program. There are two cases:

- If the type of the variable is determined to be one of the builtin types, then output one line in the following format:

  ```
  <variable>: <type> #
  ```

  where `<variable>` should be replaced by the variable name and `<type>` should be replaced by the type of the variable.

- If the type of the variable could not be determined to be one of the builtin types, then you need to list all variables that have the same type as the target variable and mark all of them as printed (so as to not print a separate entry for those later). You should output one line in the following format:

  ```
  <variable_list>: ? #
  ```

  where `<variable_list>` is a comma-separated list of variables that have the same type in the order they appear in the program.

5. Examples

Given the following:

```
a, b: int;
{
    a = < b 2;
}
```

The output will be the following:

```
TYPE MISMATCH 3 C1
```

This is because the type of `< b 2` is `bool`, but `a` is of type `int` which is a violation of C1.

Given the following:

```
a, b: int;
{
    a = + b 2.5;
}
```

The output will be the following:

```
TYPE MISMATCH 3 C2
```

This is because the type of `b` is `int` and the type of `2.5` is `real` which means in the expression `+ b 2.5`, C2 is violated

Given the following:

```
a, b: int;
{
    a = b;
}
```

The output will be the following:

```
a: int #
b: int #
```

Given the following:

```
{
    a = b;
}
```

The output will be the following:

```
a, b: ? #
```

Note that `b` is not listed separately because it is marked as printed when listed with `a` on the first line of the output.

Given the following:

```
{
    a = + 1 b;
}
```

The output will be the following:

```
a: int #
b: int #
```

Given the following:

```
{
    if (<= a b)
    {
        a = 2.4;
    }
}
```

The output will be the following:

```
a: real #
b: real #
```

Given the following:

```
{
    if (a)
    {
        b = * 2 b;
    }
}
```

The output will be the following:

```
a: bool #
b: int #
```

```
a, b: int;
c: int;

{
    x = + a * b c;
    y = ! true;
}
```

The output will be the following:

```
a:  int  #
b:  int  #
c:  int  #
x:  int  #
y: bool #
```

Given the following:

```
{
    x = + a * b c;
    y = ! < a x;
    z = w;
}
```

The output will be the following:

```
x, a, b, c: ? #
y: bool #
z, w: ? #
```

Note that `z` and `w` are not listed with `x` .

6. Requirements

Here are the requirements of this project:

- You should submit **all your project files (source code [.cc] and headers[.h]) on Gradescope.** Do not zip them, and no need to submit on canvas anymore.
- You should use C/C++, no other programming languages are allowed.
- You should test your code on Ubuntu Linux 19.04 or greater with gcc 4.9 or higher.
- You **cannot use library methods** for type checking, parsing or regular expression (regex) matching in projects. You will be implementing them yourself. If you have doubts about using a library method, please check it with the instructor or TA beforehand.
- You can write helper methods or have extra files, but they **<u>should have been written by you.</u>**

7. Evaluation

The submissions are evaluated based on the automated test cases on the Gradescope. Your grade will be proportional to the number of test cases passing. If your code does not compile on the submission website, you will not receive any points. On Gradescope, when you get the results back, ignore the “Test err” case, it is not counted toward the grade.

There will be three categories of test cases:

- Test cases with assignment statements (no `if` , `while` or `switch`):
- Test cases with assignment, if and while statements (no `switch`):
- Test cases with all types of statements

You can access the Gradescope through the left side bar in canvas, or using this link: (https://www.gradescope.com/courses/254734). You have already been enrolled in the grade scope class, and using the left side bar in canvas you will automatically get into the Gradescope course. If you want to used the link directly, than you should have received an email with title "Welcome to Gradescope for CSE 340 - Spring B" which provides info on setting up your account and password. If you already had a Gradescope account with your asu email, then most probably the course should show up in your Gradescope dashboard.