Type Checking



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

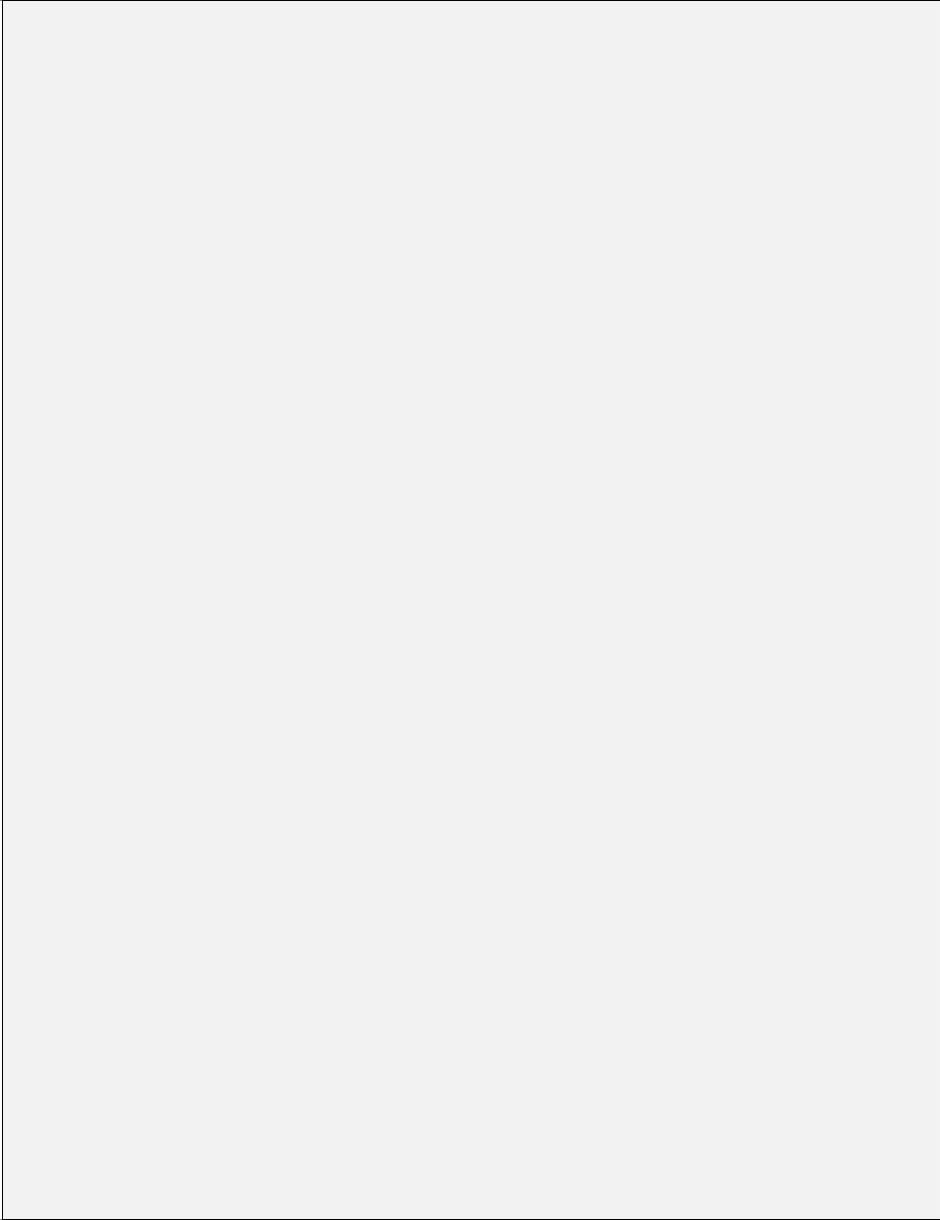
We begin by introducing the grammar of our language which is based on the previous project with additional constructs. Then we will discuss the semantics of our language and type checking rules. Finally, we will go over a few examples and formalize the expected output.

# Requirements

* This project expects inputbuf.h/cc lexer.h/cc parser.h/cc. If you use something else, please submit your own makefile named makefile in order to build your program.
* You should use C/C++, no other programming languages are allowed.
* You can only use C++ standard library.
* You should test your code on Ubuntu Linux 19.04 or greater with gcc 4.9 or higher.
* You can write helper methods or have extra files.

# Lexical Specification

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



INT = “int” REAL = “real” BOO = “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”

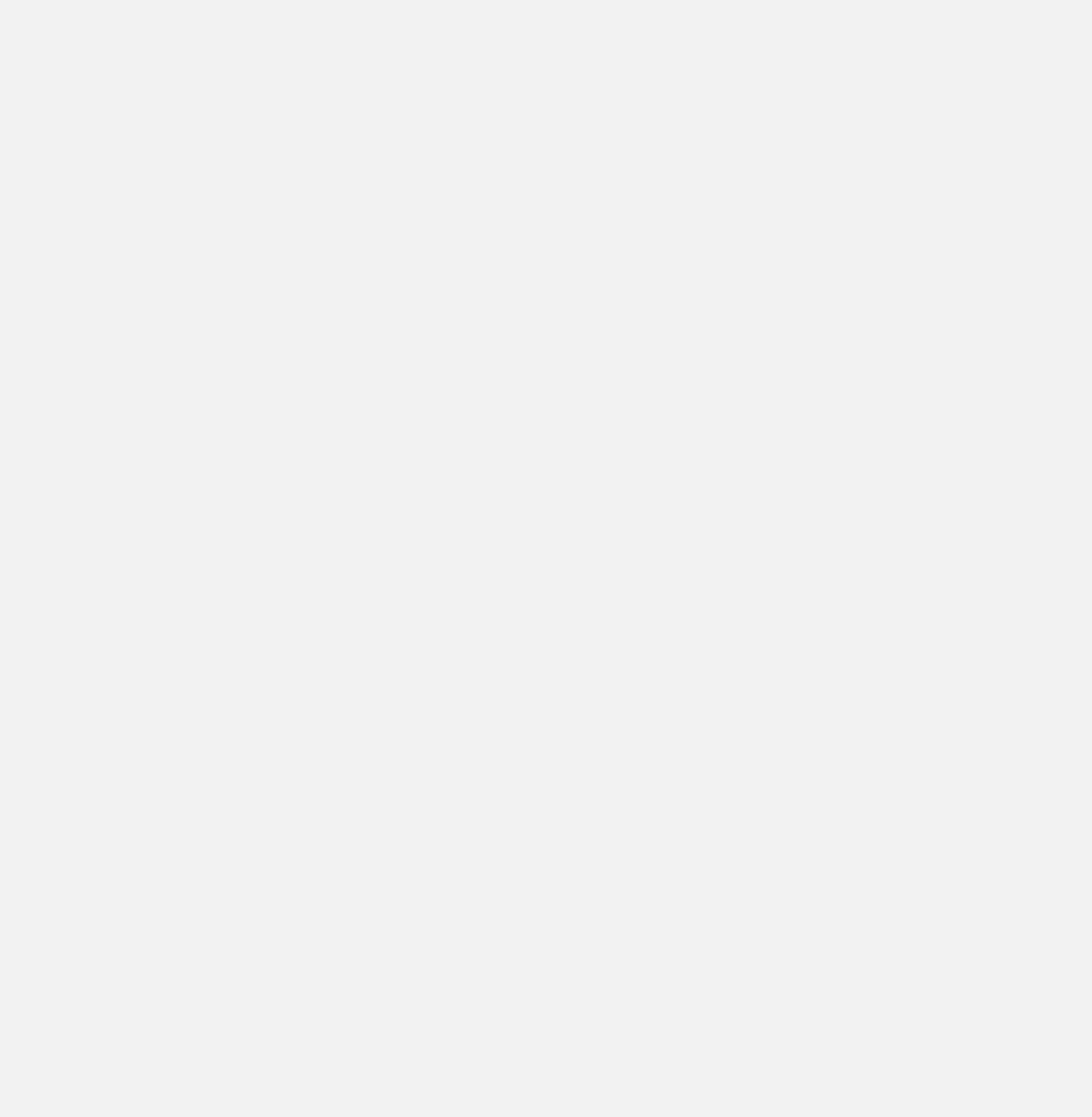
PRIVATE = “private” EQUAL = “=” COLON = “:”

COMMA = “,” SEMICOLON = “;” LBRACE = “{” RBRACE = “}”

ID = letter (letter + digit)\*

# Grammar

Here is the grammar for our input language:



program global\_vars global\_vars var\_decl\_list var\_decl\_list var\_decl var\_list var\_list type\_name type\_name type\_name body stmt\_list stmt\_list stmt

stmt stmt stmt

assignment\_stmt expression expression expression unary\_operator binary\_operator binary\_operator primary

primary primary primary primary if\_stmt while\_stmt switch\_stmt case\_list case\_list

case

-> global\_vars body

-> ε

-> var\_decl\_list

-> var\_decl

-> var\_decl var\_decl\_list

-> var\_list COLON type\_name SEMICOLON

-> ID

-> ID COMMA var\_list

-> INT

-> REAL

-> BOOL

-> LBRACE stmt\_list RBRACE

-> stmt

-> stmt stmt\_list

-> assignment\_stmt

-> if\_stmt

-> while\_stmt

-> switch\_stmt

-> ID EQUAL expression SEMICOLON

-> primary

-> binary\_operator expression expression

-> unary\_operator expression

-> NOT

-> PLUS | MINUS | MULT | DIV

-> GREATER | LESS | GTEQ | LTEQ | EQUAL | NOTEQUAL

-> ID

-> NUM

-> REALNUM

-> TRUE

-> FALSE

-> IF LPAREN expression RPAREN body

-> WHILE LPAREN expression RPAREN body

-> SWITCH LPAREN expression RPAREN LBRACE case\_list RBRACE

-> case

-> case case\_list

-> CASE NUM COLON body

# Language Semantics

## Types

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

## 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.

## 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 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.

# Output

There are two scenarios:

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

## 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:



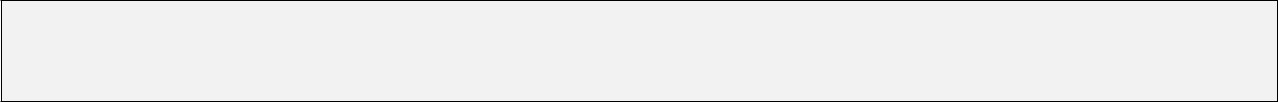
TYPE MISMATCH <line\_number> <constraint>

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.

## 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:

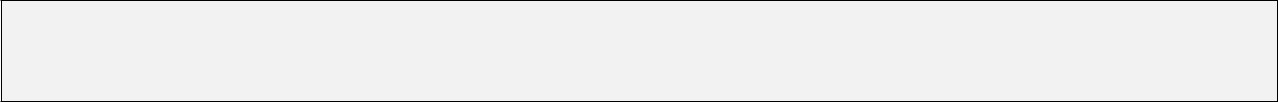
If the type of the variable is determined to be one of the built-in types, then output one line in the following format:



<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 built-in 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.

# 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:



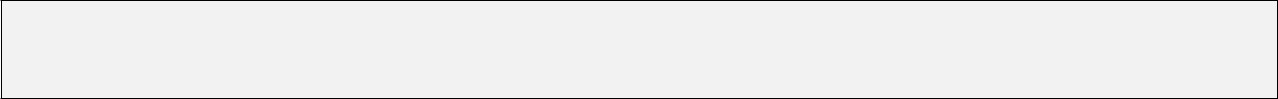
a, b: int;

{

a = b;

}

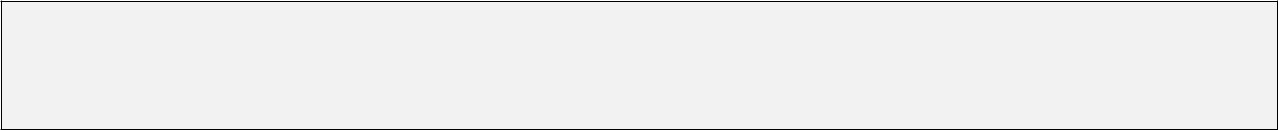
The output will be the following:



a: int #

b: int #

Given the following:



{

a = b;

}

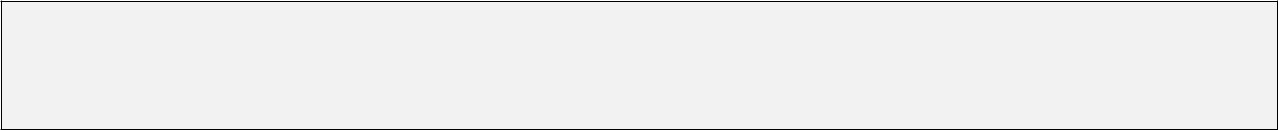
The output will be the following:



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:

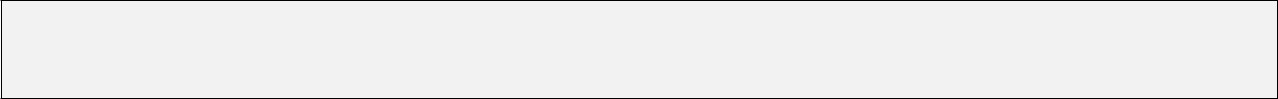


{

a = + 1 b;

}

The output will be the following:



a: int #

b: int #

Given the following:



{

if (<= a b)

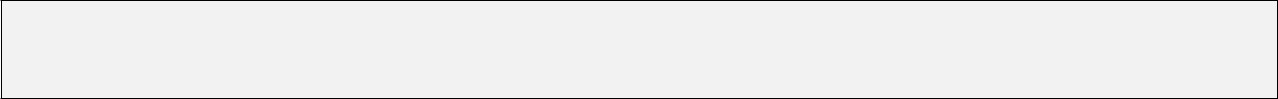
{

a = 2.4;

}

}

The output will be the following:



a: real #

b: real #

Given the following:



{

if (a)

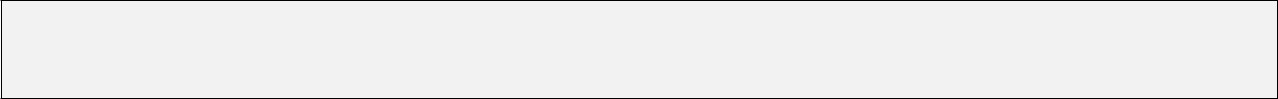
{

b = \* 2 b;

}

}

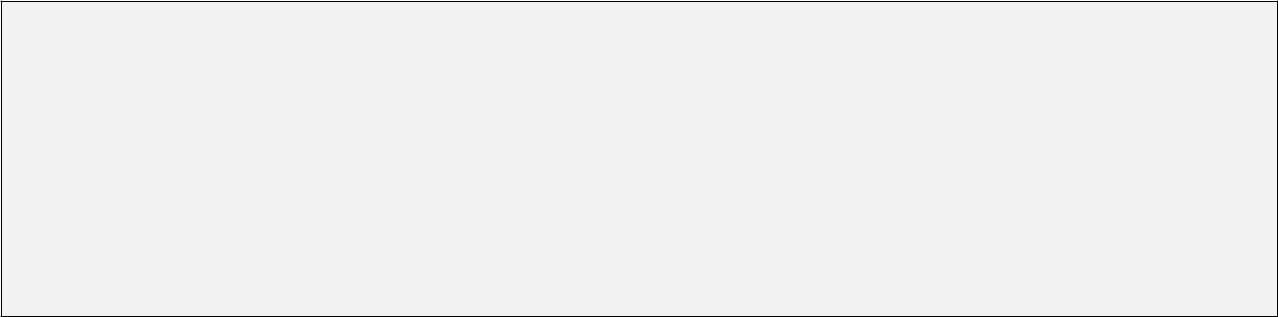
The output will be the following:



a: bool #

b: int #

Given the following:



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.

# Evaluation

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