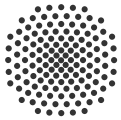

University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2023

Exercise 4: Types

(Deadline for uploading solutions: June 22, 2023, 11:59pm Stuttgart time)

The materials provided for this homework are:

- a pdf file with the text of the homework (this);
- a zip file with **the folder structure and the templates that must be used for the submission**.

The folder structure is shown in Figure 1.

```
exercise4.zip
├─ task_1
│  └─ task1.csv (Edit/replace this file with your solution)
├─ task_2
│  └─ task2.py (Edit/replace this file with your solution)
└─ task_3
   ├─ CMakeLists.txt
   ├─ Expression.cpp
   ├─ Expression.h
   ├─ ExpressionVisitor.h
   ├─ Type.h
   ├─ TypeChecker.cpp (Edit/replace this file with your solution)
   ├─ TypeChecker.h
   ├─ TypeCheckerTest.cpp
   ├─ TypeError.cpp
   ├─ TypeError.h
   └─ TypeUtils.cpp
```

Figure 1: Directory structure in the provided .zip file and in your uploaded solution file.

The submission must be compressed in a zip file using the given folder structure. The names of folders and files must not be changed or moved, otherwise the homework will not be evaluated.

General tips for making sure your submission is graded as you expect:

- Use only the zip format for the archive (not rar, 7z, gz, or anything else).
- Do not rename files or folders, simply open the files provided and put your solutions.

1 Task I (20% of total points of the exercise)

This task is to check your understanding of some concepts around types. For that, we give you code snippets written in different made-up programming languages. We describe relevant parts of the static semantics (e.g., errors during compilation) and dynamic semantics (i.e., runtime behavior) of each language with comments inside the snippet. You have to **answer one question per snippet** by selecting **exactly one** correct answer out of multiple choices.
To submit your answer, please fill in the file *exercise4/task_1/task1.csv*. One row corresponds to one code snippet, which corresponds to one answer. Fill in **A** in the second column of the CSV file to select option A as the answer, **B** to select option B, etc.

Code Snippet 1

```
a = "gato"
b = 3.14
c = a * b # At runtime: "TypeError: can't multiply sequence and float"
```

What property of the language's type system can you deduct from Code Snippet 1

A This is an example of a dynamically typed language.

B This is an example of a statically typed language.

C This is an example of a language without types.

Code Snippet 2

```
int a, c;
char b[] = "Geeks";
a = 100;
c = a / b; // At compilation time: "Error: invalid operands to binary /"
```

What property of the language's type system can you deduct from Code Snippet 2

A This is an example of a dynamically typed language.

B This is an example of a statically typed language.

C This is an example of a language without types.

Code Snippet 3

```
function search<T>(element: T, arr: T[]): boolean {
    var found = false;
    for (let e of arr) {
        if (e === element) {
            found = true;
        }
    }
    return found;
}

let numbers = [1, 2, 3];
let number_found = search<number>(1, numbers); // true

let words = ["hello", "world"];
let word_found = search<string>("hallo", words); // false
```

Which statement is true for Code Snippet 3?

A This is an example of Polymorphic variables.

B This is an example of Parametric polymorphism.

C This is an example of Subtype polymorphism.

Code Snippet 4

```
1 a = 3
2 b = 5.0
3
4 print(float(a)) # 3.0
5
6 print(a + b) # 8.0
```

Which statement is true for Code Snippet 4?

A Line 4 and Line 6 both show implicit type coercion.

B Line 4 and Line 6 both show explicit type coercion.

C Line 4 and Line 6 show implicit and explicit type coercions, respectively.

D Line 4 and Line 6 show explicit and implicit type coercions, respectively.

2 Task II (30% of total points of the exercise)

This task is about **writing type annotations** in Python. Recent versions of Python support optional type annotations for functions (since Python 3.5[1]) and local variables (since Python 3.6[2]). Those can be checked by a third-party program (e.g., `mypy`) before running the program. You can find a quick overview of the format of the annotations and the possible types here: `https://mypy.readthedocs.io/en/latest/cheat_sheet_py3.html`.

Your task is to extend a Python 3 program by adding type annotations. You should add the most specific types possible such that the program is still well-typed. E.g., the assignment `some_variable = 3` should be annotated with type `int` and not with `Any` (since the latter is always trivially type-correct but not very useful). That is, the correct solution is `some_variable: int = 3`. Please annotate all the methods (both arguments and return types) in the file *exercise4/task_2/task2.py*. Local variables do not need to be annotated. Note that types can also be generic, e.g., `List[str]` is a valid type. For such generic types, please specify all type arguments, e.g., `Dict[int, int]` not just `Dict`.

Evaluation Criteria: Your code will be evaluated by type-checking it with `mypy` version 1.3.0 with default settings to ensure that it is well-typed. Additionally, every provided type annotation is compared with the correct, most specific one for that argument/return value. (If multiple types are equivalent, we will permit those as well.) Your resulting program must still be a valid Python 3.9 program and must not remove any statements or change any of the behavior.

3 Task III (50% of total points of the exercise)

This task is about implementing a very **simple type checker** based on the formal type systems introduced in the lecture. That is, given a grammar of a language, a set of type rules, and an expression in the language, the type checker should perform a typing derivation of the expression until either all the hypotheses of all type rules are fulfilled (i.e., the expression is well-typed) or no type rules can be applied (i.e., the expression is not well-typed).

Figure 2 specifies the language for this task and its type system. Each expression in the language has one out of two types: Num (for integer numbers) and $Bool$ (for booleans).

Before starting to implement an automated type checker, we recommend writing down (with pen and paper) the typing derivations for a few expressions in the given language.

A template for your implementation is given in *exercise4/task_3/*. You will need to implement the `visit` methods in the `TypeChecker` class, in the *TypeChecker.cpp* file. Each `visit` method therein takes an expression

[1] `https://www.python.org/dev/peps/pep-0484/`
[2] `https://www.python.org/dev/peps/pep-0526/`

$$
\begin{array}{rll}
e ::= & \texttt{true} \mid \texttt{false} & \text{boolean literals} \\
& \mid 0 \mid 1 \mid 2 \mid \ldots \mid n & \text{integer literals} \\
& \mid -e & \text{unary minus} \\
& \mid e + e & \text{integer addition} \\
& \mid e \mid\mid e & \text{boolean disjunction} \\
& \mid e = e & \text{equality} \\
& \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e & \text{if-then-else}
\end{array}
$$

(a) Expression grammar. The intuition for each construct is given in gray on the right.

$$\frac{}{\texttt{true} : \mathit{Bool}}\ \text{T-True} \qquad \frac{}{\texttt{false} : \mathit{Bool}}\ \text{T-False} \qquad \frac{}{n : \mathit{Num}}\ \text{T-Num}$$

$$\frac{e : \mathit{Num}}{-e : \mathit{Num}}\ \text{T-Neg} \qquad \frac{e_1 : \mathit{Num} \quad e_2 : \mathit{Num}}{e_1 + e_2 : \mathit{Num}}\ \text{T-Add}$$

$$\frac{e_1 : \mathit{Bool} \quad e_2 : \mathit{Bool}}{e_1 \mid\mid e_2 : \mathit{Bool}}\ \text{T-Or} \qquad \frac{e_1 : T \quad e_2 : T}{e_1 = e_2 : \mathit{Bool}}\ \text{T-Eq}$$

$$\frac{e_1 : \mathit{Bool} \quad e_2 : T \quad e_3 : T}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T}\ \text{T-If}$$

(b) Type rules. The hypotheses are above the line, the conclusion is below the line, the rule name to the right. Type rules without hypotheses are axioms.

Figure 2: Grammar and type rules for a simple language with boolean and arithmetic expressions.

as argument and returns its type if type checking is successful, or throws a `TypeError` exception, if type checking fails. The `TypeError` constructor takes two types as arguments, which can be two incompatible types or the expected vs. the actual type of an expression.

There is no need to parse the input expression because you are already given its AST. For the AST definition, see *Expression.h*. For debugging, you can print expressions via their `toString` method. (It always adds parentheses, to make the infix operators unambiguous.)

To test your implementation, use the tests in *TypeCheckerTest.cpp* as a starting point. We strongly recommend implementing additional tests, e.g., ill-typed expressions and more complex expressions.

Environment Setup: For this assignment you will need:

- A compatible C++ compiler that supports at least C++14 (e.g. GCC 11.3.0, which will be used for grading)
- CMake

Evaluation Criteria: Your solution will be built with CMake and then be tested against a set of type-correct and type-incorrect expressions. Solutions that cannot be built with CMake cannot be graded. Your solution must not modify any code besides in the *TypeChecker.cpp* file. In case you implement additional tests in *TypeCheckerTest.cpp*, which we strongly recommend, do not send them. *TypeCheckerTest.cpp* should be submitted exactly as received.

Useful Commands: To build and run the tests, execute the following:

- `cmake -S . -B build`
- `cmake --build build`
- `cd build && ctest`