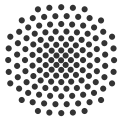

University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2022

Exercise 4: Types

(Deadline for uploading solutions: June 15, 2022, 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 (Insert your solution in this file)
├─ task_2/
│  └─ task2.py (Insert your solution in this file)
├─ task_3/
│  ├─ structA.csv (Insert your solution in this file)
│  └─ structB.csv (Insert your solution in this file)
└─ task_4/
   ├─ lib/
   │  ├─ hamcrest-core-1.3.jar
   │  └─ junit-4.13.2.jar
   └─ src/
      └─ types/
         ├─ Expression.java
         ├─ ExpressionVisitor.java
         ├─ Type.java
         ├─ TypeError.java
         ├─ TypeChecker.java (Insert your solution in this file)
         └─ TypeCheckerTest.java (Add your own tests in this file)
```

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

The submission must be compressed in a zip file (not rar, 7z, gz, or anything else) using exactly the folder structure and names in Figure 1. To ensure that your submission has the expected structure, check it with the validator.jar tool provided on the course page, using the following command (where exercise4.zip is the original file provided to you, and your_exercise4.zip is your modified version of it):

```
java -jar validator.jar exercise4.zip your_exercise4.zip
```

1 Task I (15% 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 (if in doubt, choose the one that you think matches best).

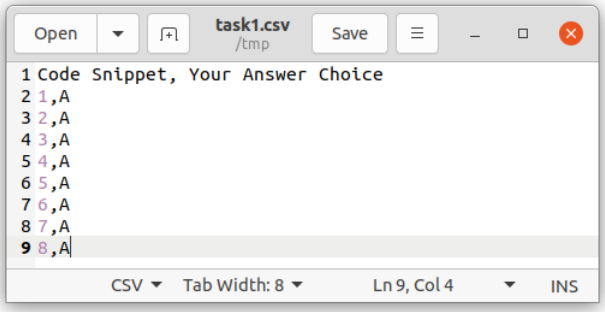

Figure 2: Example of the file format for the answers of Task I.

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. Figure 2 shows an example.

Code Snippet 1

```
let a = 3;  // Variable declaration and initialization.
let b = "hello"
let c = a+b // At compile time: "Type error: cannot add string and integer"
```

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

```
def increment(x: int) -> int:  // Function definition.
    return x + 1

increment("foobar")  // At runtime: "Type error: expected int, got string"
```

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

```
type Sometype = int;  // See below.
int a = 3;            // Variable declaration and initialization.
Sometype b = some_function();  // Store return value of some_function in b.
a = b;  // At compile time: "Error: incompatible types int and Sometype"
```

What can you deduct from the described behavior about the construct in the first line of Code Snippet 3?

A The `type` keyword seems to introduce a new, distinct type, similar to the *newtype idiom* in Haskell or Rust.

B The `type` keyword seems to introduce an equivalent synonym for an existing type, similar to a *typedef* in C or C++.

Code Snippet 4

```
type Student = { id: Integer, name: String }  // Type definition.
type University = { id: Integer, name: String }  // Type definition.
func print_name(university: University) {  // Function definition.
  print(university.name)
}
let john_doe: Student = { id: 1, name: "John Doe" }
print_name(john_doe)  // Prints "John Doe" at runtime, no error.
```

What property of the language's type system can you deduct from Code Snippet 4?

A This is an example of a nominally typed language ("nominal equivalence for types").

B Because the last line executes without error, this language has no type system.

C This is an example of a structurally typed language ("structural equivalence for types").

Code Snippet 5

```
int a = 3;    // Variable a holds integer 3 in binary twos-complement.
float b = a;  // Now b holds the IEEE 754 floating point number '3.0'.
```

Which statement is true for Code Snippet 5?

A This example is only possible in a dynamically typed language.

B This is an example of an implicit type coercion.

C The bit representation of variable b is exactly the same as the bit representation of variable a.

Code Snippet 6

```
a = 3.14    // Variable a holds an IEEE 754 floating point number.
b = int(a)  // Variable b now holds the integer value 3.
```

Which statement is true for Code Snippet 6?

A There is no information loss when going from a to b in the second line.

B This is an example of an implicit type coercion.

C This is an example of an explicit type conversion.

Code Snippet 7

```
print(2 + true) // No compile error, outputs the number 3 at runtime.
print(2 + "4")  // No compile error, outputs the string "24" at runtime.
print(2 * "4")  // No compile error, outputs the number 8 at runtime.
```

Which statement is true for Code Snippet 7?

A This is an ill-typed program.

B The + and * operator perform the same coercion, given arguments with the same types.

C Implicit coercions can make this program hard to understand.

Code Snippet 8

```
// Assume Java semantics.
interface Animal {
  public void makeSound();
}
class Horse implements Animal {
  public void makeSound() {
    System.out.println("wheehee");
  }
}
class Cat implements Animal {
  public void makeSound() {
    System.out.println("meow");
  }
}

// Somewhere else in the program...
Animal a = createAnimal();
a.makeSound();
```

Which statement is true for Code Snippet 8?

A The last line makes use of parametric polymorphism.

B The last line makes use of subtype polymorphism.

C The last line makes use of both parametric and subtype polymorphism.

2 Task II (20% 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 that is only partially type-annotated by adding more 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 functions (both arguments and return type) and local variables in the file *exercise4/task_2/task2.py*. The given file is a type-correct, but only partially type-annotated Python 3.9 program. Loop variables (e.g., `i` in `for i in ...`) 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 0.910 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 variable/argument/return value. (If multiple types are exactly 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.

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

3 Task III (20% of total points of the exercise)

This task is about the **memory layout** of structs and **alignment** in C (or other "systems" languages). Given several definitions of structs in C and several sets of rules, you should compute the memory layout of each struct for each set of rules. That is, you should determine at **which offset each field starts** and the **overall size** of the struct in memory.

For the sizes of primitive types, assume 1 byte for `chars`, 4 bytes for `ints` and `floats`, and 8 bytes for pointers. For the natural alignment of primitive types (i.e., where memory access of the underlying architecture would be fastest), assume no alignment requirement for `chars`, 4 byte alignment for `ints`, and 8 byte alignment for `floats` and pointers. For example, an alignment requirement of 4 bytes means that the byte offset of that field must divide without remainder by 4. The first offset in each struct is 0.

You have to compute the **most compact** memory layout for each struct under the following rules:

- **Packed**: The natural alignment requirements (see above) do not have to be respected, i.e., each field can start at an arbitrary offset. The order of fields must not be changed.
- **Default**: The above alignment requirements must be respected for each field. The order of fields must not be changed.
- **Reordered**: The above alignment requirements must be respected for each field, but the order of fields can be changed compared with the declaration. (But not across structs, that would change semantics.)

The two struct definitions are:

Struct A

```
1 struct A {
2   int   field1;
3   char  field2;
4   float field3;
5   char  field4;
6 };
```

Struct B

```
1 struct B {
2   char   field1;
3   float* field2;
4   struct {
5     int  field3;
6     char field4;
7   } nested_struct;
8 };
```

For solving the task, it may be useful to draw layout figures with pen and paper, similar to the lecture. However, your final answers must be submitted in the *structA.csv* and *structB.csv* files in the *exercise4/task_3/* directory. There is one row for each field of the struct and a final row for the overall size in bytes. For the fields, fill in the offset (i.e., the byte index at which the field begins), viewed from the start of the outermost struct, in the columns corresponding to each of the three rules.

To give an example: Under the **Packed** rule, `struct A` is laid out as follows. `field1` starts at byte offset 0, followed by `field2` at byte offset 4, followed by `field3` at byte offset 5, followed by `field4` at byte offset 9. The overall size of the struct is 10 bytes. This solution is already filled into the second column of the solution template in *exercise4/task_3/structA.csv*.

Evaluation Criteria: Your solution will be compared against the correct byte offsets of each struct field and the correct overall struct size under each set of alignment requirements.

4 Task IV (45% 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 3 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).

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

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

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

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

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

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

(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 3: Grammar and type rules for a simple language with boolean and arithmetic expressions.

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_4/*, which you can import into an IDE of your choice. You will need to implement the `visit` methods in the `TypeChecker` class, in the *src/types/TypeChecker.java* file. Each `visit` method therein takes an expression 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.java*. 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.java* as a starting point. We strongly recommend implementing additional tests, e.g., ill-typed expressions and more complex expressions. For constructing complex expressions more conveniently, you can use the shorthand static members (`true_`, `neg`, etc.) in the `Expression` class.

Evaluation Criteria: Your solution will be compiled with Java 8 and then be tested against a set of type-correct and type-incorrect expressions. Solutions that do not compile with Java 8 cannot be graded. Your solution must not modify any code besides in the *src/types/TypeChecker.java* file (and additional tests in *TypeCheckerTest.java*). We will not evaluate the quality of your tests.