University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2021

Exercise 5: Types

(Deadline for uploading solutions: July 2, 2021, 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 directory structure and the templates that must be used for the submission**.

The directory structure is:

```
Exercise5/
├─ Task1/
│   ├─ reference-counting.csv
│   └─ mark-and-sweep.csv
├─ Task2/
│   └─ Task2.zip
├─ task3.csv
└─ Task4/
    ├─ structA.csv
    ├─ structB.csv
    └─ structC.csv
```

The submission must be compressed in a zip file using the given directory structure. The name of files and directories 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, etc.).
- Your zip file should contain exactly one top-level directory `Exercise5/`, as in the zip file provided by us.
- Do not rename files or folders, simply open the files provided and put your solutions in them.

1 Task I (15% of total points of the exercise)

This task is about **garbage collection**, or **automatic memory management**, in "managed" languages like Java, Python, Objective-C, and many others. We give you an excerpt of a program written in Java that allocates several objects on the heap and performs several assignments. You have to determine at certain code locations in the program, how two different types of automatic memory management would keep track of the objects. Objects are identified by their `value` field. For example, "object `A`" refers to the object created via `new Object("A")`.

- For **reference counting**, you shall give the current reference count as an integer number for all objects in the program. The reference count of an object is the number of pointers referring to that object. For example, at line 11 below, you would put `1` as the reference count of object `A`.
- For **mark-and-sweep garbage collection**, you shall determine for each object whether it is marked, that is, whether it is reachable from the current variable definitions. That is, assume a mark-and-sweep garbage collector (GC) is invoked at lines 11, 15, 19, and 23 of the program below.[1] Enter `true` if the object is marked (and thus won't be deallocated), and `false` if not. For example, at line 11 below, you would put `true` for object `A`.

For objects that have not yet been created at a certain point in the program, submit `undefined` as the reference count and for the marked property. For example, object `C` is not yet created at line 11 and thus has `undefined` as the reference count and marked status in the respective answer files.

The Java program is in the following:

```java
 1  class Object { // Simple class with data and pointer to another instance.
 2      String value;
 3      Object next = null;
 4      // Constructor
 5      Object(String value) { this.value = value; }
 6  };
 7
 8  // In a method body somewhere else in the program...
 9  Object o1 = new Object("A");
10  o1.next = new Object("B");
11  // Submit answer in CSV.
12
13  o1.next.next = new Object("C");
14  Object o2 = new Object("D");
15  // Submit answer in CSV.
16
17  o1.next.next.next = o1.next;
18  Object o3 = o2;
19  // Submit answer in CSV.
20
21  o1.next = null;
22  o2 = null;
23  // Submit answer in CSV.
```

Fill your answers in the files *Exercise5/Task1/reference-counting.csv* and *Exercise5/Task1/mark-and-sweep.csv*. The first column gives the line number at which point you shall give the current state of the heap-allocated objects. The remaining columns correspond to the objects in the program.
Please make sure that your CSV file is valid. For example, it should not contain additional commas or tabs or semicolons as delimiters. Try to stay away from Excel or LibreOffice, or double check your files with a plain-text editor after editing them in said programs.

Evaluation Criteria: Your solution will be compared against the correct result for each object and each of the two memory management systems.

[1] In the real-world, the GC is often invoked non-deterministically, which is one of the drawbacks of GCs.

2 Task II (45% of total points of the exercise)

This task is about implementing an automated 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 type rules' hypotheses are fulfilled (and the expression is well-typed) or no more type-rules can be applied (and the expression is not well-typed).

Figure 1 specifies the language for this task and its type system. Each expression in the language has one out of two types: Nat and $Bool$.

$e ::= \texttt{true} \mid \texttt{false}$	boolean literals
$\mid n$	integer literals, so $n \in \mathbb{N}$
$\mid !e$	boolean negation
$\mid e \mathbin{\&\&} e$	boolean conjunction
$\mid e + e$	integer addition
$\mid e = e$	equality
$\mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	if-then-else

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

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

$$\frac{e_1 : Bool \quad e_2 : Bool}{e_1 \mathbin{\&\&} e_2 : Bool}\ \text{T-And} \qquad \frac{e_1 : Nat \quad e_2 : Nat}{e_1 + e_2 : Nat}\ \text{T-Add}$$

$$\frac{e : Bool}{!e : Bool}\ \text{T-Not} \qquad \frac{e_1 : T \quad e_2 : T}{e_1 = e_2 : Bool}\ \text{T-Eq}$$

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

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

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

A template for your implementation is given in *Exercise5/task2/Task2.zip*, which you can import into Eclipse.

1. The grammar of the language is given as an ANTLR grammar in *src/ExprPL.g4*. Generate the parser by running *java -jar antlr-4.9.2-complete.jar src/ExprPL.g4* from the *Task2* directory. This command will generate several *.java* files in the *src* directory. Please submit these files along with your solution.
2. Your implementation of the type checker will be in *src/TypeChecker.java*. The constructor of the `TypeChecker` class takes a parse tree as an argument. To check if this tree is type-correct, one can call the `check` method. Your task is to implement this method.
3. To test your implementation, use the tests in *src/TypeCheckerTest.java* as a starting point. We strongly recommend to implement additional tests, e.g., to test other parts of the syntax of the language and to create more complex expressions.

Hint: A convenient way to implement the type checker is to traverse the given parse tree by extending the `ExprPLBaseListener` class, which itself is a subclass of `ParseTreeListener`. Have a look at the API documentation of `ParseTreeWalker` to understand how to traverse a parse tree with a given `ParseTreeListener`.[2]

A note regarding precedence and associativity: The precedence of the operators in the language is the following (lower to higher): if-then-else, `&&`, `=`, `+`, `!`. All binary operators are left-associative. For example, the expression `2 + 3 + 4 = 9` is evaluated as `(((2 + 3) + 4) = 9)`, which is type-correct and yields the boolean `true`. The parse trees implicitly follow these rules, i.e., evaluating subtrees bottom-up yields the correct evaluation order.

Evaluation Criteria: Your solution will compiled with Java 8 and will then be tested against a set of type-correct and type-incorrect expressions. All given expressions will be syntactically correct, i.e., they can be parsed into a parse tree.

[2] `https://www.antlr.org/api/Java/`

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

This task is about **pointer arithmetic and arrays** in C and C++. You are provided with an incomplete C program and possible code fragments to be used to complete the code. Please fill in some of the blanks with some of the provided code fragments. You can fill in at most one code fragment per blank. Each fragment can be used only once. When completed, the `main` function of the program should be syntactically correct and type-correct, print the value 3, and free all allocated memory before returning from the main function.

Submit the correct assignments of blanks to code fragments in the solution file *Exercise5/task3.csv*. In the provided template file, for each blank either fill in the number of the correct fragment or 0 to indicate that none of the fragments should be inserted.

Incomplete Code

```c
#include<stdlib.h>
#include<stdio.h>

int main() {
        int *p2;
        int *p1;
        __blank1__
        for(int i=0;i<4;i++){
                p1[i] = i;
        }

        __blank2__
        for(int i=0;i<3;i++){
                p2[i] = 1;
        }

        int *pc, c;
        c = 5;
        __blank3__
        __blank4__
        printf("%d", *pc);

        __blank5__
        __blank6__
}
```

Options for code fragments to insert:

- Fragment 1: `pc = &c;`
- Fragment 2: `free(p1);`
- Fragment 3: `pc = c;`
- Fragment 4: `free(p2);`
- Fragment 5: `p1 = malloc(4*sizeof(int));`
- Fragment 6: `c = p2[1] + p1[2];`
- Fragment 7: `p2 = malloc(3*sizeof(int));`
- Fragment 8: `c = p2[1] + p1[3];`
- Fragment 9: `p2 = malloc(3*sizeof(char));`

Evaluation Criteria: Your solution will be compared against a correct assignment of blanks to code fragments.

4 Task IV (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 `char`s, 4 bytes for `int`s and `float`s, 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 `char`s, 4 byte alignment for `int`s, and 8 byte alignment for `float`s 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. Similar to C[3], assume no padding between array elements, but there can be before the array to satisfy the alignment of the first element.

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 three struct definitions are:

Struct A

```c
struct A {
  char   field1;
  float  field2;
  int    *field3;
  char   field4;
};
```

Struct B

```c
struct B {
  int    field1[10];
  float  field2;
  float  *field3;
};
```

Struct C

```c
struct C {
  int     *field1;
  float   field2;
  struct {
    int   *field3;
    int   field4;
  } nested_struct;
};
```

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 *struct*.csv* files in the *Exercise5/Task4/* 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.

[3] see https://stackoverflow.com/questions/1066681/can-c-arrays-contain-padding-in-between-elements

To give an example: Under the **Packed** rule, `struct A` is layed out as follows. `field1` starts at byte offset 0, followed by `field2` at byte offset 1, followed by `field3` at byte offset 5, followed by `field4` at byte offset 13. The overall size of the struct is 14 bytes. This solution is already filled into the second column of the solution template in *Exercise5/Task4/structA.csv*.

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