FACULTEIT
WETENSCHAPPEN

TESTed: one judge to rule them all

Een universele judge voor het beoordelen van software in een educative context

Niko Strijbol

Studentennummer: 01404620

Promotoren: prof. dr. Peter Dawyndt, dr. ir. Bart Mesuere
Begeleiding: Charlotte Van Petegem

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de Informatica

Academiejaar: 2019 – 2020

UNIVERSITEIT
GENT

Samenvatting

Veel programmeeroefeningen zijn niet inherent gebonden aan een programmeertaal en kunnen opgelost worden in meerdere programmeertalen. Het manueel vertalen van oefeningen van de ene programmeertaal naar de andere is een tijdrovende bezigheid. De focus van deze masterproef is de onderzoeksvraag „Is het mogelijk om oefeningen slechts één keer op een programmeertaalonafhankelijke manier te schrijven en toch oplossingen in verschillende programmeertalen te aanvaarden?”.

Als antwoord hierop introduceert de masterproef TESTed, een prototype van een judge voor het Dodona-platform, die dezelfde oefening in meerdere programmeertalen kan beoordelen. Dodona is een online platform om programmeeroefeningen op te lossen en voorziet oplossingen in real time van feedback, zoals de juistheid.

Een kernaspect van TESTed is het testplan. Dit is een specificatie van hoe een oplossing voor een oefening beoordeeld moet worden. Bij bestaande judges in Dodona gebeurt dit op een programmeertaalafhankelijke manier: jUnit in Java, een eigen formaat in Python, enz. Het testplan bij TESTed is programmeertaalonafhankelijk en wordt opgesteld in JSON. Op deze manier wordt één testplan opgesteld, waarna de oefening kan opgelost worden in alle programmeertalen die TESTed ondersteunt. Momenteel zijn dat Python, Java, Haskell, C en JavaScript.

Het programmeertaalonafhankelijk zijn van het testplan verhindert niet dat een testplan geschreven kan worden dat bepaalde functionaliteit gebruikt die niet aanwezig is in elke programmeertaal. Een voorbeeld hiervan is een testplan met klassen en objecten, dat enkel opgelost zal kunnen worden in objectgericht programmeertalen.

Een testplan bestaat uiteindelijk uit een reeks testgevallen. Een testgeval is een invoer gekoppeld aan een uitvoer. Als invoer ondersteunt het testplan `stdin`, functieoproepen en commandoargumenten. De uitvoer, die beoordeeld wordt, bestaat uit `stdout`, `stderr`, exceptions, returnwaarden, aangemaakte bestanden en de exitcode van de uitvoering.

Voor de vertaling van het testplan naar testcode in de programmeertaal van de oplossing gebruikt TESTed een sjabloonsysteem genaamd Mako, gelijkaardig aan sjabloonsystemen bij webapplicaties (ERB bij Rails, Blade bij Laravel, enz.). Deze sjablonen worden opgesteld bij de configuratie van een programmeertaal in TESTed, samen met een configuratieklasse, die details over de programmeertaal bevat, zoals hoe de programmeertaal gecompileerd moet worden (dit is optioneel) en hoe ze uitgevoerd moet worden.

Doordat het testplan onafhankelijk is van de programmeertaal kunnen bij het configureren van een nieuwe programmeertaal bestaande oefeningen zonder wijziging gebruikt opgelost worden in de nieuwe programmeertaal.

TESTed: one judge to rule them all

Niko Strijbol
Supervisors: Peter Dawyndt, Bart Mesuere

Abstract

Writing a programming exercise is a time-consuming activity. When the goal is using the same exercise in multiple programming languages, the required time grows fast. One needs to manually translate the exercise to each language one wishes to support, although most exercises do not make use of language specific constructs. The translation is not intellectually stimulating work: it is an almost mechanical job. This article introduces TESTed, a prototype of an exercise framework in which a test plan is written in a language-independent format. This allows evaluating submissions in multiple programming languages for the same exercise. TESTed also integrates with Dodona, an online exercise platform. First the inner workings of the framework are described, followed by a discussion of the current limitations and future work.

1. Introduction

Technology is becoming increasingly important in our society: digital solutions are used to solve tasks and problems in more and more domains. As a result of this evolution, students have to become familiar with computational thinking: translating problems from the real world into problems understood by a computer [1]. Computational thinking is broader than just programming, yet programming is a very good way to train students in computational thinking. However, learning to program is often perceived as difficult by students [2]. As solving exercises is beneficial to that learning process, it is important to provide students with exercises that are high in both quality and quantity. This requirement imposes two challenges on educators:

- Educators have to write suitable exercises: they need to take into account what concepts the students are familiar with, how long it takes to solve an exercise, etc.
- The students' submissions must be provided with high quality feedback. This feedback allows students to learn from their mistakes and improve their programming skills (and by proxy their computational thinking skills).

A significant part of programming exercises are for novices: they are intended to teach how to program, while the language details are not very important. Another type of exercise are algorithmic exercises, in which the algorithm is the most important aspect. Both of these types of exercises are prime candidates for being available in multiple programming languages. However, making an exercise available in multiple programming languages adds a non-negligible time cost to the already time-consuming and labour-intensive process of writing exercises. The exercise must be translated manually to each language one wishes to support. This translation work is not intellectually stimulating or interesting: in most cases, it is a fairly mechanical translation. The TESTedframework, introduced in Section 3, is an answer to the question "is it feasible to create a framework, in which an exercise is written once, but is solvable in multiple programming languages?".

2. Dodona

To cope with the second challenge, providing good feedback, educators often use a platform in which submissions for exercises are evaluated automatically. One such platform is Dodona[1], an online platform for programming exercises. It supports multiple programming languages and provides real-time automatic feedback for submissions. Dodona is freely available for schools and is used by multiple courses at Ghent University.

Dodona is a modular system: the programming language specific code for evaluating a submission (the *judge*) is separated from the platform. This judge is run in a Docker container and communicates with Dodona via a JSON interface, using `stdin` for the input and `stdout` for the output. The input are mainly configuration parameters, such as memory and time limits, in addition to the submission and the exercise code. The output of the Docker container is the result of the evaluation. Naturally, Dodona supports determining the correctness of a submission. However, the judge system is a flexible one: other types of feedback are possible. For example, some existing judges run a linter on the submissions, which can contain useful hints for students to improve the quality of their code.

TESTed, introduced in the next section, is implemented as a judge and thus fully integrates with Dodona. As the Dodona JSON interfaces are sufficiently flexible and generic, we believe it is independently usable as well.

[1] https://dodona.ugent.be/

3. TESTed

TESTed is a prototype of a framework for writing programming exercises and evaluating submissions for those exercises in multiple programming languages. The framework can be split into three distinct parts:

- The *test plan* and serialization format. The test plan is a specification on how to evaluate a submission for a given exercise. Combined with the serialization format, it makes writing language-independent exercises possible.
- The *core*, which takes care of handling input and output (in the Dodona JSON format), generating test code from the test plan (using the language configurations), executing said code, and evaluating the results of said execution.
- The *language configurations*, which are the subsystem responsible for translating the test plan into actual code. By separating the language configurations from the main logic, it is easy to add support for new programming languages in TESTed.

3.1. The test plan

The test plan is a programming language independent format that specifies how a submission for an exercise must be evaluated. It contains elements such as the different tests, the inputs, the expected outputs, etc. A concrete example of a simple test plan is included in Listing 3.1. The structure of the test plan is heavily inspired by the Dodona JSON format and consists of a hierarchy of the following elements:

Tabs A tab is the top-level grouping mechanism for the tests. One test plan can consist of multiple tabs, which are shown as distinct tabs in the Dodona user interface.

Contexts Each tab consists of one or more contexts. A context is an independent execution and evaluation of a submission. Each context is run in a new subprocess.

Testcase A context consists of one or more testcases. A testcase is the evaluation of one input and the resulting outputs. We distinguish two types of testcases:

Context testcase The testcase containing the call to the main function (or script execution). A context can have at most one context testcase.

Normal testcase Testcases with other inputs, such as function calls or assignments.

Test Finally, a testcase contains multiple tests, each for a different output channel. Currently, there are tests for `stdout`, `stderr`, return values, exceptions and exit codes. The testcase contains the input, while the tests contain the different expected outputs.

```
{
  "tabs": [
    {
      "name": "Tab",
      "contexts": [
        {
          "context_testcase": {
            "input": {
              "main_call": true,
              "stdin": {
                "type": "text",
                "data": "input-1"
              }
            },
            "output": {
              "stdout": {
                "type": "text",
                "data": "input-1"
              }
            }
          },
          "testcases": [
            {
              "input": {
                "type": "function",
                "name": "echo",
                "arguments": [
                  {
                    "type": "integer",
                    "data": 50
                  }
                ]
              },
              "output": {
                "result": {
                  "value": {
                    "type": "integer",
                    "data": 50
                  }
                }
              }
            }
          ]
        }
      ]
    }
  ]
}
```

Listing 1: An example of a test plan for a fictional exercise. A solution for this exercise should print `stdin` to `stdout` when executed, but also contain a `echo` function, which returns its argument. Both scenario's are tested in the same context in this test plan. In real usage, these would probably be separate contexts.

An important part of the test plan that deserves further attention is the input. As we've mentioned, each testcase has a single input. There are basically two types of input: `stdin` and a statement. Since our goal is not to create a universal programming language, statements have intentionally been kept simple. A statement is either an expression or an assignment. An assignment must be understood as giving a name to the result of an expression. An expression can be a function call, an identifier (referring to a variable previously created using an assignment) or a literal value. The arguments of a function call are also expressions.

Additionally, TESTed defines a serialization format for values. This format consists of two pieces: the data type of a value and the encoding of the value. Since the serialization format is defined in JSON, as is the test plan, the encoding is simply a JSON type. For example, numerical values are encoded as a JSON number. The data type of the value is complexer: since we support multiple programming languages, we must support generic data types. To this end, TESTed defines two kinds of data types:

- Basic types, which include integral numbers, rational numbers, booleans, strings, sequences, sets and maps. These are abstract types, and we don't concern ourselves with implementation details for these types. For example, all integer types in C (`int`, `long`, etc.) map to the same integral number type in the serialization format.
- Advanced types, which are more detailed (`int64`) or programming language specific. Here we do concern ourselves with implementation details.

The advanced types are associated with a basic type, which acts as a fallback. If a programming language does not support a certain advanced type, the basic fallback type will be used. For example, a `tuple` in Python will be considered equal to an `array` in Java, since they both have the basic type `sequence`. Programming languages can also indicate that there is no support for certain types (e.g. no support for sets in C), in which case the exercise will not be solvable in that programming language if the exercise uses that data type.

The serialization format is used for both encoding return values and as the format for literal types in the test plan. However, it is important to note that this is not a formal type system. While a formal type system's main goal is describing which categories of data a construct may be or receive, the type system in the serialization format is used to describe concrete values. For example, the test plan does not specify "a function with return type `int64`". Instead, it specifies "a function whose expected return value is 16, which is of type `int64`". As such, TESTed does not use nor need techniques normally associated with a type system, such as a formal type checker.

3.2. Running an evaluation

A first part of the core of TESTed is responsible for running the evaluation of a submission for an exercise. Each evaluation runs trough a list of steps, as illustrated by Figure 1 and described below:

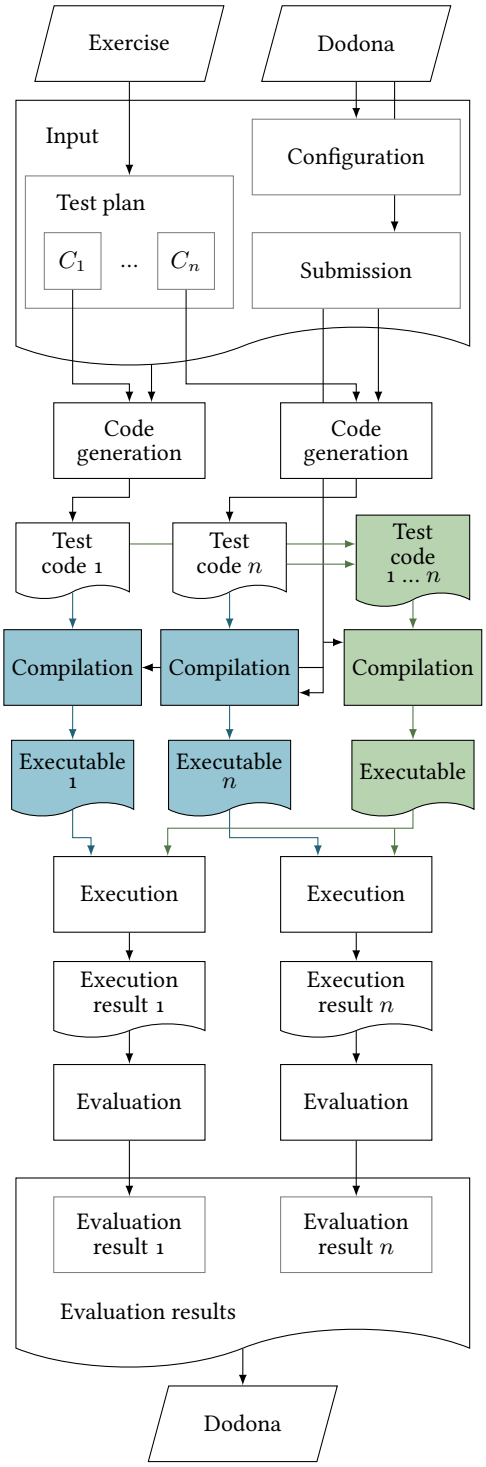


Figure 1: A flow chart depicting the various steps of an evaluation in TESTed. C_1 and C_n stand for context 1 and context n of the test plan. The two colours indicate the different possible paths for the compilation step: blue for context compilation and green for batch compilation.

1. The Docker container for TESTed is started by Dodona, and the submission and configuration are made available to TESTed.
2. The test plan is checked to verify that the exercise is solvable in the programming language of the submission.
3. For each context in the test plan, the test code is generated.
4. The test code is compiled in one of two ways:

   Batch compilation The test code of every context is bundled and compiled together in one compilation. This mode results in one executable.

   Context compilation The test code for each context is compiled separately. For n contexts, there will be n compilations, resulting in n executables.

   This compilation step is, of course, optional. Note that we use the term executable to denote the output of the compilation step, although the output is not always an executable, as is the case in Java.
5. The result of the previous step (the compilation results or the test code if there is no compilation) is executed. Each context is executed in a new subprocess, to combat sharing information between contexts.
6. The collected results of the execution are then evaluated, which is described in Section 3.3.
7. As a final step, TESTed sends the evaluation results to Dodona.

One might ask why we need two compilation modes. The answer is performance. Since the evaluation of a submission happens in real-time, it is desirable to keep the evaluation short. One bottleneck is the compilation step, which can be quite slow in languages such as Java or Haskell. It is common to have test plans of 50 contexts or more, which does mean there would be 50 or more compilations. As a solution, we don't compile each context independently: instead we compile all contexts in one go. Note that the contexts are still executed independently.

3.3. Evaluating the results

A second part of the core is evaluating the results of the previous execution step. As we've mentioned, each output channel is denoted by a different test in the test plan. Currently, TESTed has support for following output channels: `stdout`, `stderr`, exceptions, return values, created files and the exit code. In most cases, the test plan would specify how a channel should be evaluated for each relevant output channel. If an output channel is not relevant, TESTed provides sane defaults (e.g. not specifying `stderr` means there should be no output on `stderr`; if there is output, the submission is considered wrong).

Generally, there are three ways in which an output channel can be evaluated:

Language-specific evaluation The evaluation code is included in the test code generated for the context and is executed directly in the same subprocess as the test code. As a result, the evaluation code must be written in the same programming language as the submission. This mode is intended to evaluate programming language specific aspects.

Programmed evaluation The evaluation code is executed separately from the test code, in a different process. The results of the execution pass through TESTed: they are first serialized in the test process and deserialized in the evaluation process. This means the programming language of the submission and the evaluation code can differ. For example, the evaluation code can be written in Python, and used to evaluate the results of submissions in Java, JavaScript, Haskell, etc.

Generic evaluation TESTed has built-in support for simple evaluations, like comparing a produced value against an expected value contained in the test plan. For example, if a function call with argument a should result in value b, there is no need to write evaluation code. There is support for evaluating textual results (`stdout`, `stderr`), return values, exceptions and the exit code. The evaluator for values is intelligent and takes the data types into account. If the test plan specifies that the return value should be a tuple, TESTed will apply strict comparisons in language supporting tuples (Python and Haskell), but loose comparisons in other languages. This means that for Python and Haskell solutions, only tuples will be accepted. In other languages, all data types with the corresponding basic type will be accepted (such as arrays and lists).

Not all modes are available for all output channels. For example, the language-specific evaluation mode is only available for return values and exceptions. It is assumed that textual output (e.g. `stout`) is not programming language specific.

4. Configuring programming languages

Support for a programming language in TESTed consists of three parts:

1. A configuration file
2. A configuration class
3. Templates

The configuration file is used to record properties of the programming language, such as the file extension or which data structures are supported.

The configuration class handles the language-specific aspects of the compilation and execution step. TESTed expects a command to perform during those steps (e.g. for C, the command would along the lines of `gcc -std=c11 file1.c file2.c`).

The third component are the templates. TESTed uses the Mako templating system [3] to generate the test code and translate language independent concepts from the test plan into actual code (such as literal values). The templating system works

similar to the one used by webapps (such as ERB in Rails, Blade in Laravel or EEX in Phoenix), but generates code instead of HTML/JSON.

Since the test plan is language independent, existing exercise require no change to work with a newly added programming language.

5. Future work

In this section, we briefly discuss the current limitations of the prototype and identify area's where future improvements are possible.

5.1. Test plan

A first area in which future improvements are possible is the test plan. The test plan currently does not support dynamic scheduling of contexts: the contexts to execute are decided before the execution happens. The test plan can be extended to allow some form of decision, for example deciding if a context should run depending on the result of the previous context. Related are repeated executions of the same context. This is useful for exercises with randomness, where the output is not deterministic. For example, generating a list of random numbers with a minimum and maximum constraint would need multiple runs to validate that random numbers are used.

Another aspect of the test plan are the statements and expressions. While these are currently intentionally simple, this does impose some limitations. For example, an expression with mathematical operators (`5 + 5`) is not possible. The expressions and statements could be extended to provide more capabilities. It is also worth investigating if, when more capabilities are added, it might be worth switching to an existing language and transpile this language to the various programming languages TESTed supports.

The test plan is also fairly verbose. We believe this could be solved by introducing a preprocessing step, that translates a more compact format to the test plan. A promising upcoming format is [4]. We also envision different formats for different exercise types. For example, an IO exercise and exercise with function calls have different requirements.

5.2. Programming paradigms

Secondly, TESTed does not translate programming paradigms. For example, a given exercise might be solved using two top-level functions in Python, but would often be solved with an object in Java. Functional languages also have different paradigms compared to object-oriented languages. It might be worth researching common patterns and their equivalent in other programming languages and providing translation for those common patterns.

5.3. Performance

Lastly, while performance is not bad, there is room for improvement. One area in particular are the programmed evaluations. Each evaluation is currently compiled and executed, even though it is common for multiple contexts to use the same evaluation code.

6. Conclusion

We have presented TESTed, a judge for the Dodona platform, capable of evaluating submissions in multiple programming languages for the same exercise. At the moment, TESTed supports Python, Java, Haskell, C and JavaScript. While there still some limitations on the type of exercises TESTed can evaluate, it can already be used for a variety of exercises.

References

[1] B. Bastiaensen, J. De Craemer, Zo denkt een computer, Vlaamse Overheid, 2017.

[2] A. Luxton-Reilly, Simon, I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, L. Ott, J. Paterson, M. J. Scott, J. Sheard, C. Szabo, Introductory programming: A systematic literature review, in: Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2018 Companion, Association for Computing Machinery, New York, NY, USA, 2018, p. 55–106. doi:10.1145/3293881.3295779.

[3] M. Bayer, G. T. Dairiki, P. Jenvey, D. Peckam, A. Ronacher, B. Bangert, B. Trofatter, Mako templates for python.
URL https://www.makotemplates.org/

[4] P. Conrad, C. Bart, S. Edwards, Programming exercise markup language.
URL https://cssplice.github.io/peml/

Vulgariserende samenvatting

In onze maatschappij wordt technologie, en informatica in het bijzonder, alsmaar belangrijker. In steeds meer sectoren worden problemen opgelost met behulp van digitale apparatuur. Het is daarom belangrijk dat iedereen een basis digitale geletterdheid heeft. Het is niet voldoende om te kunnen werken met de programma's en technologie van vandaag. Wie weet of programma's zoals Word of PowerPoint binnen twintig jaar nog relevant zijn? De technologie verandert snel, waardoor het nodig is een begrip van de onderliggende systemen te hebben.

Zo komen we aan bij het begrip *computationeel denken*. Computationeel denken is een breed begrip, maar een goede definitie is het oplossen van problemen met behulp van de computer. Het gaat om het vertalen van het probleem uit de „echte wereld" naar de informaticawereld, zodat het probleem kan begrepen worden door een computer. Dit computationeel denken is recent ook opgenomen in de eindtermen van zowel het basisonderwijs als het secundair onderwijs.

Een goede manier om computationeel denken aan te leren aan studenten is programmeren. Studenten ervaren programmeren echter vaak als moeilijk. Het spreekwoord „oefening baart kunst" indachtig, denken we dat het maken van veel oefeningen een goede manier is om programmeren onder de knie te krijgen. Het aanbieden van veel oefeningen leidt tot uitdagingen voor de lesgevers:

1. Lesgevers moeten geschikte oefeningen opstellen, aangepast aan het niveau van de studenten. Oefeningen moeten rekening houden met welke concepten studenten al kennen, hoeveel werk het oplossen is, enzovoort.
2. De oplossingen voor deze oefeningen moeten voorzien worden van kwalitatieve feedback. Enkel en alleen oefeningen oplossen is niet voldoende: goede feedback laat toe dat studenten hun vaardigheden verbeteren, doordat ze een idee krijgen wat beter kon of waar ze fout zaten.

Voor de tweede uitdaging wordt vaak gebruikt gemaakt van een platform voor programmeeroefeningen, dat een eerste vorm van automatische feedback geeft, zoals de correctheid van de ingediende oplossing voor een oefening. Aan de onderzoeksgroep Computationele Biologie van de UGent is hiervoor het Dodona-platform ontwikkeld.

Een aan de eerste uitdaging gerelateerd aspect is dat er veel programmeertalen zijn en elk van die programmeertalen oefeningen nodig heeft. Momenteel gebeurt het vertalen van een oefening van één programmeertaal naar de andere manueel. Dit neemt ook veel tijd in beslag, terwijl veel oefeningen niet programmeertaalafhankelijk zijn. Veel programmeeroefeningen zijn eenvoudig en komen neer op een lijst sorteren, woorden zoeken in een bestand, iets berekenen op basis van gegevens enzovoort. Deze taken kunnen in elke programmeertaal.

In deze masterproef zoeken we naar een oplossing voor dit laatste probleem: is het mogelijk om een oefening slechts één keer op een programmeertaalonafhankelijke manier te schrijven en toch oplossingen in verschillende programmeertalen te beoordelen?

Als onderdeel van het antwoord op deze vraag hebben we een prototype van een nieuwe judge voor het Dodona-platform ontwikkeld (een judge in Dodona is het onderdeel dat verantwoordelijk is voor het beoordelen van een ingediende oplossing): TESTed. In TESTed moet de lesgever voor een oefening een programmeertaalonafhankelijk testplan opstellen. Dit testplan bevat de specificatie van hoe een ingediende oplossing beoordeeld moet worden. Daarna vertaalt TESTed dit testplan naar de verschillende programmeertalen, waardoor één oefening opgelost kan worden in de programmeertalen die TESTed momenteel ondersteunt: Python, Java, JavaScript, Haskell en C.

Daarnaast hebben we ook aandacht besteed aan het zo eenvoudig mogelijk houden om nieuwe programmeertalen aan TESTed toe te voegen. Als we een nieuwe programmeertaal toevoegen aan TESTed, zullen bestaande oefeningen ook meteen opgelost kunnen worden in deze nieuwe programmeertaal, zonder dat er iets moet veranderen aan het testplan van deze oefeningen.

Dankwoord

De masterproef als culminatie van de opleiding maakt het mijn aangename plicht om ieder te bedanken met wie ik in contact ben gekomen tijdens deze opleiding, in het bijzonder de professoren, assistenten en ondersteunend personeel, die deze opleiding mogelijk gemaakt hebben. Daarnaast zijn er enkele personen wier steun een expliciete vermelding verdient.

In de eerste plaats wil ik mijn promotoren, prof. dr. Peter Dawyndt en dr. Bart Mesuere, en mijn begeleiding, Charlotte Van Petegem, bedanken. Niet alleen voor het aanbieden van dit onderwerp, waarzonder deze masterproef niet zou bestaan, maar ook voor al hun tijd en moeite die ze in deze masterproef gestoken hebben. Zonder de wekelijkse thesismeetings, het nalezen van de verschillende iteraties van de tekst en de vele suggesties zou de masterproef niet zijn wat ze nu is. Ik wil ook prof. Dawyndt in het bijzonder bedanken voor het grondig nalezen van de tekst en het uitproberen van wat ik geschreven heb.

Verder wil ik de leden van Zeus WPI bedanken, die ervoor zorgden dat er in de kelder altijd wel iets anders te doen was dan werken aan de masterproef. Het is jammer dat ik hen in het tweede semester heb moeten missen.

Tot slot wil ik mijn familie bedanken voor hun onvoorwaardelijke steun en gezelschap, waardoor het ook thuis aangenaam werken was.

Niko Strijbol

Toelating tot bruikleen

De auteur geeft de toelating deze masterproef voor consultatie beschikbaar te stellen en delen van de masterproef te kopiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de bepalingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze masterproef.

Niko Strijbol
29 mei 2020

Inhoudsopgave

1. Educational software testing **1**
- 1.1. Computationeel denken . . . 1
- 1.2. Programmeeroefeningen . . . 2
- 1.3. Het leerplatform Dodona . . . 2
- 1.4. Beoordelen van oplossingen . . . 3
- 1.5. Probleemstelling . . . 4
- 1.6. Opbouw van de masterproef . . . 7

2. TESTed **8**
- 2.1. Architecturaal ontwerp . . . 8
- 2.2. Stappenplan van een beoordeling . . . 8
- 2.3. Beschrijven van een oefening . . . 12
- 2.4. Oplossingen uitvoeren . . . 23
- 2.5. Oplossingen beoordelen . . . 26
- 2.6. Performantie . . . 32
- 2.7. Bijkomende taken . . . 37
- 2.8. Robuustheid . . . 37
- 2.9. Implementatiekeuzes . . . 39

3. Configuratie van een oefening **41**
- 3.1. Oefeningen in het Dodona-platform . . . 41
- 3.2. Lotto . . . 42
- 3.3. Echo . . . 45
- 3.4. Echofunctie . . . 47
- 3.5. ZeroDivisionError . . . 50
- 3.6. Som . . . 52
- 3.7. ISBN . . . 54
- 3.8. EqualChecker . . . 57

4. Configuratie van een programmeertaal **61**
- 4.1. TESTed lokaal uitvoeren . . . 61
- 4.2. Globaal stappenplan voor het configureren van een programmeertaal . . . 63
- 4.3. De programmeertaal C . . . 63
- 4.4. Hoe lang duurt het configureren van een programmeertaal? . . . 80
- 4.5. Stabiliteit van TESTed . . . 81
- 4.6. Ondersteunde programmeertalen . . . 83

5. Conclusie, beperkingen en toekomstig werk **84**
- 5.1. Dodona-platform . . . 84
- 5.2. Testplan . . . 86
- 5.3. Performantie . . . 92
- 5.4. TESTed . . . 94

5.5. Extra functionaliteit 96

Bibliografie **98**

A. Mapstructuur na uitvoeren **101**

B. Echo-oefening **102**
B.1. Python 102
B.2. Java 104
B.3. C 109
B.4. Haskell 113
B.5. JavaScript 117

C. Echofunctie-oefening **120**
C.1. Python 120
C.2. Java 122
C.3. C 125
C.4. Haskell 127
C.5. JavaScript 130

1. Educational software testing

De evoluties op technologisch vlak hebben ervoor gezorgd dat onze maatschappij de laatste decennia in hoge mate gedigitaliseerd is, een proces dat nog steeds aan de gang is. Bovendien kan, door de snelheid waarmee deze veranderingen vaak optreden, eerder gesproken worden van een revolutie dan een evolutie: de veranderingen zijn vaak ingrijpend en veranderen fundamentele aspecten van de sectoren waarin de digitalisering plaatsvindt. Dit gaat over het ontstaan van nieuwe sectoren, zoals de deeleconomie, maar ook over ingrijpende veranderingen bij bestaande sectoren, zoals de opkomst van *ride sharing* in de taxisector. Ook de impact op maatschappelijk vlak, zoals de sociale media in de politiek, mag niet vergeten worden [1].

Ook op educatief vlak heeft de digitalisering een grote impact. Enerzijds biedt digitalisering nieuwe mogelijkheden aan voor onderwijsdoeleinden, zoals het lesgeven op afstand, het online aanbieden van leermateriaal en het online indienen en verbeteren van opdrachten.

Anderzijds biedt het ook uitdagingen: om studenten voor te bereiden op de steeds digitalere maatschappij is een basis van digitale geletterdheid nodig. Net door de snelle evolutie op technologisch vlak volstaat het niet om studenten te leren werken met de technologie van vandaag; een grondige kennis van de onderliggende werking van de technologie is onontbeerlijk.

Een belangrijk aspect hierin is het concept van *computationeel denken*. Dat het aanleren van digitale vaardigheden nodig is, bewijst ook de opname van dat computationeel denken in de eindtermen, bijvoorbeeld in het katholieke basisonderwijs [2] of in het secundair onderwijs [3].

1.1. Computationeel denken

Op de vraag wat computationeel denken nu precies betekent lopen de antwoorden uiteen. Het Departement Onderwijs en Vorming van de Vlaams Overheid [4] definieert de term als volgt:

> *Computationeel denken verwijst dus naar het menselijke vermogen om complexe problemen op te lossen en daarbij computers als hulpmiddel te zien. Met andere woorden, computationeel denken is het proces waarbij aspecten van informaticawetenschappen herkend worden in de ons omringende wereld, en waarbij de methodes en technieken uit de informaticawetenschappen toegepast worden om problemen uit de fysische en virtuele wereld te begrijpen en op te lossen.*

Computationeel denken is dus ruimer dan programmeren, maar programmeren vormt wel een uitstekende manier om het computationeel denken aan te leren en te oefenen. Bovendien is programmeren op zich ook een nuttige vaardigheid om studenten aan te leren.

1.2. Programmeeroefeningen

Het aanleren van programmeren is niet eenvoudig en wordt door veel studenten als moeilijk ervaren [5]. Het maken van oefeningen kan daarbij helpen, indachtig het spreekwoord „oefening baart kunst". Studenten veel oefeningen laten maken resulteert wel in twee uitdagingen voor de lesgevers:

1. Lesgevers moeten geschikte oefeningen opstellen, die rekening houden met welke programmeerconcepten studenten al kennen, tijdslimieten, moeilijkheidsgraden, enzovoort. Het opstellen van deze oefeningen vraagt veel tijd.
2. De oplossingen voor deze oefeningen moeten voorzien worden van kwalitatieve feedback. Bij het aanleren van programmeren is feedback een belangrijk aspect om de programmeervaardigheden van de studenten te verbeteren [6].

Deze masterproef focust op de eerste uitdaging, al wordt ook ingespeeld op de tweede uitdaging, onder andere in paragraaf 2.8. De uitdagingen worden beschouwd binnen de context van Dodona, een online leerplatform voor automatische feedback op ingediende oplossingen voor programmeeroefeningen.

1.3. Het leerplatform Dodona

Sinds 2011 wordt aan de onderzoeksgroep Computationele Biologie van de Universiteit Gent gewerkt met programmeeroefeningen waarbij oplossingen in een online systeem ingediend en beoordeeld worden. Oorspronkelijk werd hiervoor gebruik gemaakt van de *Sphere Online Judge* (SPOJ) [7]. Op basis van ervaringen met SPOJ ontwikkelde de onderzoeksgroep een eigen leerplatform, Dodona, dat in september 2016 beschikbaar werd. Het doel van Dodona is eenvoudig: lesgevers bijstaan om hun studenten niet alleen zo goed mogelijk te leren programmeren, maar dit ook op een zo efficiënt mogelijke manier te doen.

Het online leerplatform Dodona kan opgedeeld worden in verschillende onderdelen:

1. Het **leerplatform** zelf is een webapplicatie, die verantwoordelijk is om de verschillende onderdelen samen te laten werken en die ook de webinterface aanbiedt die de studenten en lesgevers gebruiken. Het is via deze interface dat lesgevers oefeningen beschikbaar maken en dat studenten hun oplossingen indienen. Het platform zelf is zo opgebouwd dat het onafhankelijk is van de programmeertaal van de oefeningen.
2. De **judges** binnen Dodona zijn verantwoordelijk voor het beoordelen van ingediende oplossingen. Dit onderdeel wordt uitgebreid besproken in paragraaf 1.4.
3. De **oefeningen** worden niet in Dodona zelf aangemaakt, maar worden door de lesgever aangeleverd via een git-repository. De oefeningen bevatten de beschrijving van de opgave en de testen die uitgevoerd worden tijdens het automatisch beoordelen van een oplossing.

Met Dodona kunnen lesgevers een leertraject opstellen door een reeks oefeningen te selecteren. Studenten die dit leertraject volgen, zien onmiddellijk hun voortgang binnen het traject. Bij het indienen van hun oplossingen ontvangen de studenten ook onmiddellijk feedback over hun oplossing: deze feedback bevat niet alleen de correctheid van de oplossing, maar kan ook andere aspecten belichten, zoals de kwaliteit van de oplossing (onder andere de programmeerstijl), het resultaat van het uitvoeren van linters, en de performantie van de oplossing.

1.4. Beoordelen van oplossingen

In Dodona wordt elke ingediende oplossing beoordeeld door een evaluatieprogramma, de *judge*. In wezen is dit een eenvoudig programma: via de standaardinvoerstroom (`stdin`) krijgt het programma een configuratie van Dodona. Deze configuratie bevat onder andere de programmeertaal van de ingediende oplossing, de locatie van de oefeningenbestanden (via de git-repository van de oefening), de locatie van de ingediende oplossing zelf en configuratieopties, zoals geheugen- en tijdslimieten. Het resultaat van de beoordeling wordt uitgeschreven naar de standaarduitvoerstroom (`stdout`). Zowel de invoer als de uitvoer van de judge zijn JSON, waarvan de structuur vastgelegd is in JSON Schema.[1]

Concreet wordt elke beoordeling uitgevoerd in een Docker-container. Deze Docker-container wordt gemaakt op basis van een Docker-image die bij de judge hoort, en alle dependencies bevat die de judge in kwestie nodig heeft. Bij het uitvoeren van de beoordeling zal Dodona een *bind mount*[2] voorzien, zodat de code van de judge zelf, de code van de oefening en de code van de ingediende oplossing beschikbaar zijn in de container. Via de configuratie geeft Dodona aan de judge mee waar deze bestanden zich bevinden.

Samenvattend bestaat de interface tussen de judge en Dodona uit drie onderdelen:

1. De judge zal uitgevoerd worden in een Docker-container, dus een Docker-image met alle dependencies moet voorzien worden. Deze Docker-image moet ook de judge opstarten.
2. Dodona stelt de invoer van een beoordeling ter beschikking aan de judge. Bestanden worden via een bind mount aan de Docker-container gekoppeld. De paden naar deze bestanden binnen de container en andere informatie (zoals programmeertaal van de oplossing of natuurlijke taal van de gebruiker) worden via de configuratie aan de judge gegeven (via de standaardinvoerstroom).
3. De judge moet het resultaat van zijn beoordeling uitschrijven naar de standaarduitvoerstroom, in de vastgelegde structuur.

Buiten deze interface legt Dodona geen vereisten op aan de werking van judge. Door deze vrijheid lopen de manieren waarop de bestaande judges geïmplementeerd zijn uiteen. Sommige judges beoordelen oplossingen in dezelfde programmeertaal als de taal waarin ze geschreven zijn. Zo is de judge voor Python-oplossingen geschreven in Python en de judge voor Java-oplossingen in Java. Bij andere judges is dat niet het geval: de judges voor Bash en Prolog zijn bijvoorbeeld ook in Python geschreven. Daarnaast heeft elke judge een eigen manier waarop de testen voor de beoordeling van een oplossing opgesteld moeten worden. Zo worden in de Java-judge jUnit-testen gebruikt, terwijl de Python-judge doctests (en een eigen formaat) ondersteunt.

De beoordeling van een oplossing van een student verloopt als volgt:

1. De student dient de oplossing in via de webinterface van Dodona.
2. Dodona start een Docker-container voor de judge.
3. Dodona voorziet de container van de bestanden van de judge, de oefening en de ingediende oplossing via de bind mount.

[1] Dit schema en een tekstuele beschrijving ervan is te vinden in de handleiding op https://dodona-edu.github.io/en/guides/creating-a-judge/.

[2] Informatie over deze term is te vinden op https://docs.docker.com/storage/bind-mounts/

4. De judge wordt uitgevoerd, waarbij de configuratie via de standaardinvoerstroom beschikbaar is.
5. De judge beoordeelt de oplossing zoals aangegeven door de lesgever (via jUnit-tests, doctests, ...). Judges kunnen ook bijkomende taken uitvoeren, zoals linting, beoordeling van de performantie of *grading* van de code van de oplossing.
6. De judge vertaalt zijn beoordeling naar het Dodona-formaat en schrijft het resultaat naar de standaarduitvoerstroom.
7. Dodona slaat dat resultaat op in de databank.
8. Op de webinterface krijgt de student het resultaat te zien als feedback op de ingediende oplossing.

1.5. Probleemstelling

De manier waarop de huidige judges werken resulteert in twee belangrijke nadelen. Bij het bespreken hiervan is het nuttig een voorbeeld in het achterhoofd te houden, teneinde de nadelen te kunnen concretiseren. Als voorbeeld gebruiken we de „Lotto"-oefening[3], waarvan de opgave hieronder gegeven wordt. Oplossingen voor deze oefening in twee programmeertalen, Python en Java, staan in codefragment 1.1.

Opgave

Schrijf een functie `loterij` waarmee een lottotrekking kan gesimuleerd worden. De functie moet twee parameters `aantal` en `maximum` hebben. Aan de parameter `aantal` (`int`) kan doorgegeven worden hoeveel balletjes a er moeten getrokken worden (standaardwaarde 6). Aan de parameter `maximum` (`int`) kan doorgegeven worden uit hoeveel balletjes m er moet getrokken worden (standaardwaarde 42). Beide parameters kunnen ook weggelaten worden, waarbij dan de standaardwaarde gebruikt moet worden. De balletjes zijn daarbij dus genummerd van 1 tot en met m. Je mag ervan uitgaan dat $1 \leq a \leq m$. De functie moet een string (`str`) teruggeven die een strikt stijgende lijst (`list`) van a natuurlijke getallen (`int`) beschrijft, waarbij de getallen van elkaar gescheiden zijn door een spatie, een koppelteken (-) en nog een spatie. Voor elk getal n moet gelden dat $1 \leq n \leq m$.

Voorbeeld

```
> loterij()
'2 - 17 - 22 - 27 - 35 - 40'
> loterij(8)
'5 - 13 - 15 - 31 - 34 - 36 - 39 - 40'
> loterij(4, 38)
'16 - 20 - 35 - 37'
```

[3] Vrij naar een oefening van prof. Dawyndt. De originele oefening is beschikbaar op https://dodona.ugent.be/nl/exercises/2025591548/.

Codefragment 1.1. Oplossing in Python en Java voor de voorbeeldoefening Lotto.

```python
from random import randint


def loterij(aantal=6, maximum=42):
    getallen = set()
    while len(getallen) < aantal:
        getallen.add(randint(1, maximum))
    return " - ".join(str(x) for x in sorted(getallen))
```

```java
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.ThreadLocalRandom;
import java.util.stream.Collectors;

class Submission {
    public static String loterij(int aantal, int maximum) {
        var r = ThreadLocalRandom.current();
        var result = new HashSet<Integer>();
        while (result.size() < aantal) {
            result.add(r.nextInt(1, maximum + 1));
        }
        return result.stream()
                .sorted()
                .map(Object::toString)
                .collect(Collectors.joining(" - "));
    }

    public static String loterij(int aantal) {
        return loterij(aantal, 42);
    }

    public static String loterij() {
        return loterij(6, 42);
    }
}
```

1.5.1. Opstellen van oefeningen

Het eerste en belangrijkste nadeel aan de werking van de huidige judges heeft betrekking op de lesgevers en komt voor als zij een oefening willen aanbieden in meerdere programmeertalen. Enerzijds is dit een zware werklast: de oefening, en vooral de code voor de beoordeling, moet voor elke judge opnieuw geschreven worden. Voor de Python-judge zullen doctests nodig zijn, terwijl de Java-judge jUnit-testen vereist. Anderzijds lijdt dit ook tot verschillende versies van dezelfde oefening, wat het onderhouden van de oefeningen moelijker maakt. Als er bijvoorbeeld een fout sluipt in de beoordelingscode, zal de lesgever er aan moeten denken om de fout te verhelpen in alle varianten van de oefening. Bovendien geeft elke nieuwe versie van de oefening een nieuwe mogelijkheid voor het introduceren van fouten.

Kijken we naar de Lotto-oefening, dan valt op dat het gaat om een eenvoudige opgave en een eenvoudige oplossing. Bovendien zijn de verschillen tussen oplossingen in verschillende programmeertalen niet zo groot. In de voorbeeldoplossingen in Python en Java zijn de verschillen minimaal, zij het dat de Java-oplossing wat langer is. De Lotto-oefening zou zonder problemen in nog vele andere programmeertalen opgelost kunnen worden. Eenvoudige programmeeroefeningen, zoals de Lotto-oefening, zijn voornamelijk nuttig in twee gevallen: studenten die voor het eerst leren programmeren en studenten die een nieuwe programmeertaal leren. In het eerste geval is de eigenlijke programmeertaal minder relevant: het zijn vooral de concepten die belangrijk zijn. In het tweede geval is de programmeertaal wel van belang, maar moeten soortgelijke oefeningen gemaakt worden voor elke programmeertaal die aangeleerd moet worden. In beide gevallen is het dus een meerwaarde om de oefening in meerdere programmeertalen aan te bieden.

Eenzelfde constatatie kan gemaakt worden bij meer complexe oefeningen die zich concentreren op algoritmen: ook daar zijn de concepten belangrijker dan in welke programmeertaal een algoritme uiteindelijk geïmplementeerd wordt. Een voorbeeld hiervan is het vak „Algoritmen en Datastructuren", dat gegeven wordt door prof. Fack binnen de opleiding wiskunde[4]. Daar zijn de meeste opgaven op Dodona vandaag al beschikbaar in de programmeertalen Java en Python, maar dan als afzonderlijke en onafhankelijke oefeningen.

Een ander aspect is de beoordeling van een oefening. Voor de Lotto-oefening is de beoordeling niet triviaal, door het gebruik van niet-deterministische functies. Het volstaat voor dit soort oefeningen niet om de uitvoer geproduceerd door de oplossing te vergelijken met een op voorhand vastgelegde verwachte uitvoer. De geproduceerde uitvoer zal moeten gecontroleerd worden met code, specifiek gericht op deze oefening, die de verwachte vereisten van de oplossing controleert. Deze evaluatiecode moet momenteel voor elke programmeertaal en dus elke judge opnieuw geschreven worden. In de context van de Lotto-oefening controleert deze code bijvoorbeeld of de gegeven getallen binnen het bereik liggen en of ze gesorteerd zijn. Toch is deze evaluatiecode niet inherent programmeertaalafhankelijk: controleren of een lijst gesorteerd is, heeft weinig te maken met de programmeertaal van de oplossing.

1.5.2. Implementeren van judges

Een tweede nadeel aan de huidige werking zijn de judges zelf: voor elke programmeertaal die men wil aanbieden in Dodona moet een nieuwe judge ontwikkeld worden. Ook hier is er dubbel werk: dezelfde concepten en features, die eigenlijk programmeertaalonafhankelijk zijn, moeten in elke judge opnieuw geïmplementeerd worden. Hierbij denken we aan bijvoorbeeld de logica om te bepalen wanneer een beoordeling juist of fout is.

[4]De studiefiche is beschikbaar op https://studiegids.ugent.be/2019/NL/studiefiches/C002794.pdf

1.5.3. Onderzoeksvraag

Het eerste nadeel wordt beschouwd als het belangrijkste nadeel en de focus van deze masterproef. Het nadeel valt te formuleren als de onderzoeksvraag waarop deze masterproef een antwoord wil bieden:

> Is het mogelijk om een judge zo te implementeren dat de opgave en beoordelingsmethoden van een oefening slechts eenmaal opgesteld dienen te worden, waarna de oefening beschikbaar is in alle programmeertalen die de judge ondersteunt? Hierbij is het wenselijk dat eens een oefening opgesteld is, deze niet meer gewijzigd moet worden wanneer talen toegevoegd worden aan de judge.

Als bijzaak is het ook interessant om na te gaan of het antwoord op de onderzoeksvraag een voordeel kan bieden voor het implementeren van judges zelf.

De aandachtige lezer zal opmerken dat de opgave voor de Lotto-oefening programmeertaalspecifieke en taalspecifieke elementen bevat. Zo is de beschrijving van de opgave (de voorbeelden en de gegevenstypes) in Python en zijn de namen van functies en parameters in het Nederlands. Het ondersteunen van opgaves met programmeertaalonafhankelijke voorbeelden en vertalingen wordt voor deze masterproef expliciet als *out-of-scope* gezien en zal niet behandeld worden, zij het in paragraaf 5.4.1.

1.6. Opbouw van de masterproef

Hoofdstuk 2 handelt over het antwoord op bovenstaande onderzoeksvraag, waar een prototype van een dergelijke judge wordt voorgesteld.

In hoofdstuk 3 wordt een reeks oefeningen met verschillende functionaliteiten besproken om te illustreren hoe een testplan opgesteld moet worden en om aan te tonen welke soorten oefeningen met TESTed gebruikt kunnen worden. Daarna volgt een gedetailleerde beschrijving van hoe een programmeertaal geconfigureerd moet worden voor dit prototype in hoofdstuk 4. Deze twee hoofdstukken hebben ten doel een handleiding te vormen voor de lesgevers die oefeningen willen opstellen en voor de programmeurs die een nieuwe programmeertaal aan TESTed willen toevoegen. Daarom sluiten deze hoofdstukken qua vorm en stijl meer aan bij een handleiding, wat zich bijvoorbeeld manifesteert als een toename in het aantal codefragmenten.

Tot slot volgt met een hoofdstuk over beperkingen van het huidige prototype, en waar er verbeteringen mogelijk zijn (het „toekomstige werk”).

Na de bronvermeldingen volgen nog twee appendices waar voor twee eenvoudige oefeningen de opgave, het testplan, een oplossing en de door TESTed gegenereerde code getoond worden.

2. TESTed

In het kader van deze masterproef werd een prototype geïmplementeerd van een judge voor Dodona. Het doel hiervan is een antwoord te bieden op de onderzoeksvraag uit het vorige hoofdstuk en de beperkingen van de gekozen aanpak in kaart te brengen. Deze judge heeft de naam TESTed gekregen. Bij TESTed is een oefening programmeertaalonafhankelijk en kunnen oplossingen in verschillende programmeertalen beoordeeld worden aan de hand van een en dezelfde specificatie. Dit hoofdstuk begint met het toelichten van het ontwerp en de algemene werking van de judge, waarna elk onderdeel in meer detail besproken wordt.

2.1. Architecturaal ontwerp

Figuur 2.1 toont het architecturaal ontwerp van TESTed. De twee stippellijnen geven programmeertaalbarrières aan, en verdelen TESTed in drie logisch omgevingen:

1. TESTed zelf is geschreven in Python (zie paragraaf 2.9): in de middelste omgeving staat de programmeertaal dus vast. Deze omgeving is verantwoordelijk voor de regie van de beoordeling op basis van het testplan.
2. De ingediende oplossing wordt uitgevoerd in de *uitvoeringsomgeving*, waar de programmeertaal overeenkomt met de programmeertaal van de ingediende oplossing.
3. Tot slot is er nog de *evaluatieomgeving*, waar door de lesgever geschreven evaluatiecode wordt uitgevoerd. Deze moet niet in dezelfde programmeertaal als de ingediende oplossing of als TESTed geschreven zijn.

2.2. Stappenplan van een beoordeling

De rest van dit hoofdstuk bespreekt in detail alle onderdelen van en stappen die gebeuren bij de beoordeling van een ingediende oplossing. In figuur 2.2 zijn deze stappen gegeven als een flowchart, en een uitgeschreven versie volgt:

1. De Docker-container voor TESTed wordt gestart. Dodona stelt de invoer ter beschikking aan de container: het testplan komt uit de oefening, terwijl de ingediende oplossing en de configuratie uit Dodona komen.
2. Als eerste stap wordt gecontroleerd of het testplan de programmeertaal van de ingediende oplossing ondersteunt (zie paragraaf 2.3.4). De programmeertaal van de oplossing wordt gegeven via de configuratie uit Dodona. Merk op dat de ingediende oplossing hierbij zelf niet nodig is: deze controle zou idealiter gebeuren bij het importeren van de oefening in Dodona, zodat Dodona weet in welke programmeertalen een bepaalde oefening aangeboden kan worden (zie paragraaf 5.1.1). Als het testplan bijvoorbeeld programmeertaalspecifieke

code bevat die enkel in Java geschreven is, zal een oplossing in Python niet beoordeeld kunnen worden. Bevat het testplan een functie die een verzameling moet teruggeven, dan zullen talen als Bash niet in aanmerking komen.

3. Het testplan (zie paragraaf 2.3.1) bestaat uit verschillende contexten. Elke context is een onafhankelijke uitvoering van de ingediende oplossing en kan verschillende aspecten van die uitvoering beoordelen. Voor elk van die contexten wordt in deze stap de testcode gegenereerd. Deze stap is de overgang naar de *uitvoeringsomgeving*.

4. De testcode wordt optioneel gecompileerd. Dit kan op twee manieren (details in paragraaf 2.4.1):

   Batchcompilatie Hierbij wordt de testcode van alle contexten gebundeld en in één keer gecompileerd. In sommige programmeertalen, zoals Haskell en C, leidt dit tot één uitvoerbaar bestand. In andere talen, zoals Java, is dit niet het geval. We noemen voor het gemak de uitvoer van een compilatiestap altijd een *executable*, ook al gaat het technisch niet altijd om een uitvoerbaar bestand. Het grootste voordeel van één compilatiestap is de winst op vlak van performantie.

   Contextcompilatie Hier wordt de testcode voor elke context afzonderlijk gecompileerd. Bij deze manier worden er *n* executables geproduceerd tijdens de compilatiestap.

   Bij geïnterpreteerde programmeertalen wordt deze stap overgeslagen.

5. Nu kan het uitvoeren van de beoordeling zelf beginnen: de gegenereerde code wordt uitgevoerd (nog steeds in de uitvoeringsomgeving). Elke context uit het testplan wordt in een afzonderlijk subproces uitgevoerd, teneinde het delen van informatie tegen te gaan.

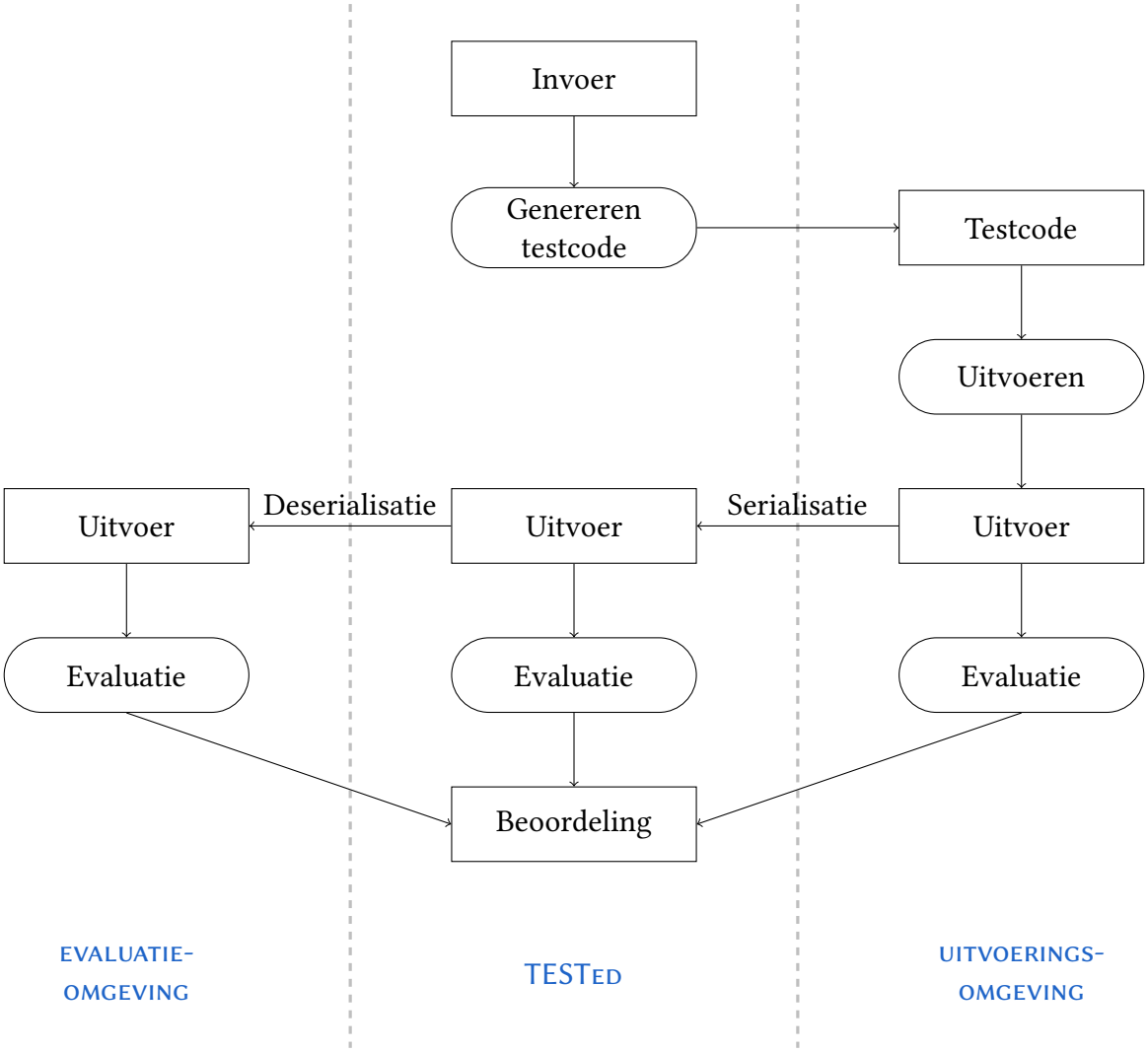
6. Het uitvoeren in de vorige stap produceert resultaten (voor elke context of elk testgeval), zoals de standaarduitvoerstroom, de standaardfoutstroom, returnwaarden, exceptions of exitcodes. Deze bundel resultaten wordt nu geëvalueerd op juistheid. Hiervoor zijn drie mogelijke manieren:

   Programmeertaalspecifieke evaluatie (SE) De code voor de evaluatie is opgenomen in de executable en wordt onmiddellijk uitgevoerd in hetzelfde proces. Via deze mogelijkheid kunnen programmeertaalspecifieke aspecten gecontroleerd worden. Daar de evaluatie in hetzelfde proces gebeurt, blijft dit in de uitvoeringsomgeving.

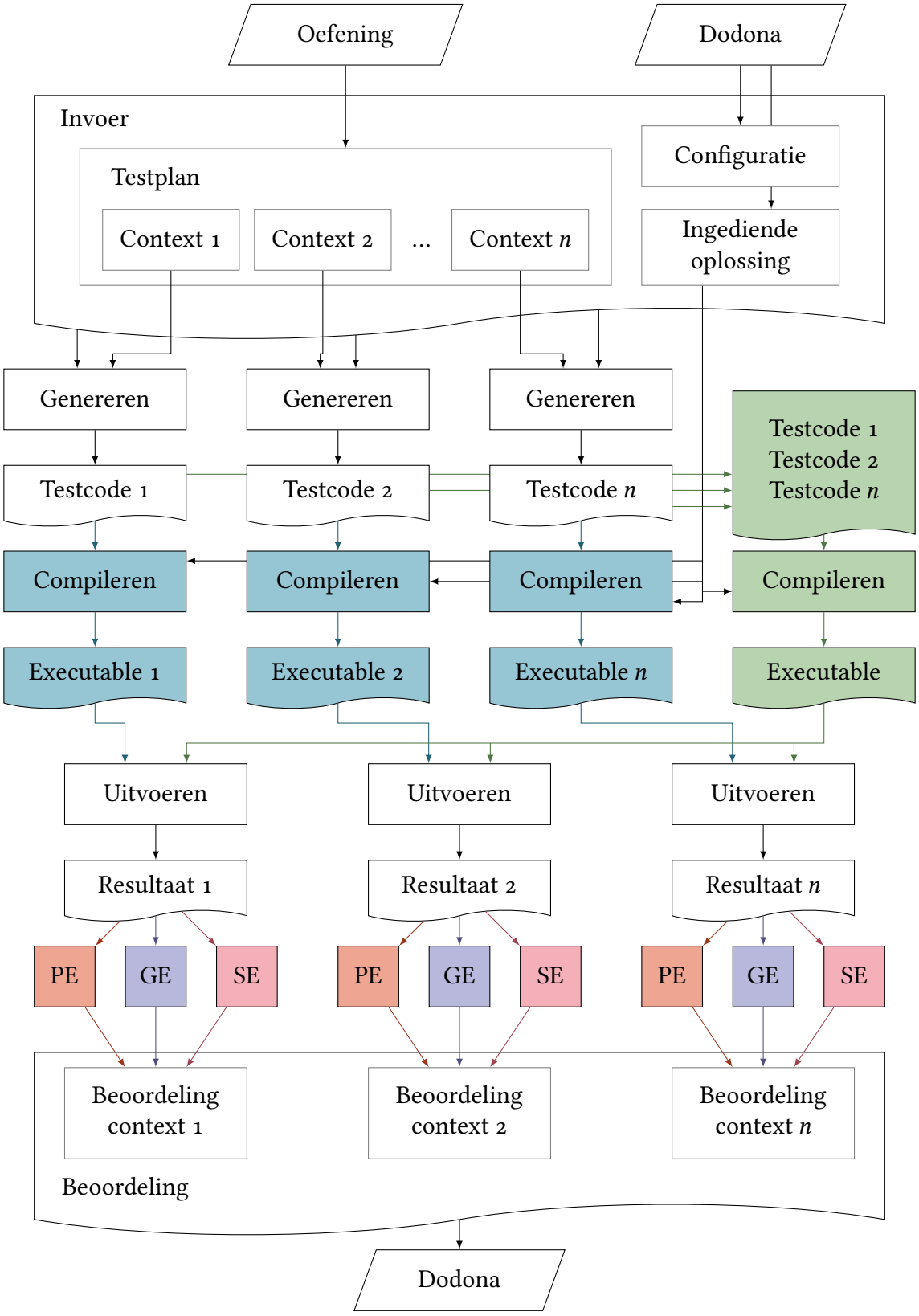
   Geprogrammeerde evaluatie (PE) Hierbij is er evaluatiecode geschreven die los staat van de oplossing, waardoor deze evaluatiecode ook in een andere programmeertaal geschreven kan zijn. De code voor de geprogrammeerde evaluatie wordt uitgevoerd in een nieuw proces, de evaluatieomgeving. Het doel van deze modus is om complexe evaluaties toe te laten op een programmeertaalonafhankelijke manier.

   Generieke evaluatie (GE) Hierbij evalueert TESTed zelf het resultaat. Deze manier is bedoeld voor gestandaardiseerde evaluaties, zoals het vergelijken van geproduceerde uitvoer en verwachte uitvoer. Hier gebeurt de evaluatie binnen TESTed zelf.

7. Tot slot verzamelt TESTed alle evaluatieresultaten en stuurt ze gebundeld door naar Dodona, waarna ze getoond worden aan de gebruiker.



Figuur 2.1. Schematische voorstelling van het architecturale ontwerp van de TESTed.



Figuur 2.2. Flowchart van een beoordeling door TESTed. Controles die de beoordeling voorafgaan zijn niet opgenomen. In het schema worden kleuren gebruikt als er een keuze gemaakt moet worden voor een volgende stap. Er kan steeds slechts één mogelijkheid gekozen worden. De afkortingen PE, GE en SE staan respectievelijk voor geprogrammeerde evaluatie, generieke evaluatie en (programmeertaal)specifieke evaluatie.

2.3. Beschrijven van een oefening

De beoordeling van een ingediende oplossing begint bij de invoer die TESTed krijgt. Centraal in deze invoer is het *testplan* voor een oefening, een specificatie die op een programmeertaalonafhankelijke manier beschrijft hoe een oplossing moet beoordeeld worden. Het vervangt de programmeertaalspecifieke testen van de bestaande judges (i.e. de jUnit-tests of de doctests in respectievelijk Java en Python). Het testplan *sensu lato* wordt opgedeeld in verschillende onderdelen, die hierna besproken worden.

2.3.1. Het testplan

Het testplan *sensu stricto* beschrijft de structuur van de beoordeling van een ingediende oplossing voor een oefening. Deze structuur lijkt qua opbouw sterk op de structuur van de feedback zoals gebruikt door Dodona. Dat de structuur van de beoordeling in Dodona en in het testplan op elkaar lijken, heeft als voordeel dat er geen mentale afbeelding moet gemaakt worden tussen de structuur van het testplan en dat van Dodona. Dit is nuttig omdat de structuur van het testplan uiteindelijk een bijkomende zaak is: het belangrijkste is hoe de feedback aan de studenten getoond zal worden door Dodona. Concreet is de structuur een hiërarchie met volgende elementen:

Plan Het top-level object van het testplan. Dit object bevat de tabbladen en de naam van de klasse of module van de ingediende oplossing. Zo moet in Java standaard een klasse `Submission` gebruikt worden. Configuratieopties voor TESTed zijn niet opgenomen in het testplan, maar horen thuis in het configuratiebestand van de oefening, samen met de configuratieopties voor Dodona.

Tab Een testplan bestaat uit verschillende *tab*s of tabbladen. Deze komen overeen met de tabbladen in de gebruikersinterface van Dodona. Een tabblad kan een naam hebben, die zichtbaar is voor de gebruikers.

Context Elk tabblad bestaat uit een of meerdere *context*en. Een context is een onafhankelijke uitvoering van de code van een ingediende oplossing. Hierbij wordt de nadruk gelegd op de onafhankelijkheid: elke context wordt in een nieuw proces en nieuwe map (op het bestandssysteem) uitgevoerd, zodat de kans op het delen van informatie kleiner is. Hierbij willen we vooral onbedoeld delen van informatie (zoals statische variabelen of het overschrijven van bestanden) vermijden. De gemotiveerde student zal nog steeds informatie kunnen delen tussen de uitvoeringen, door bijvoorbeeld in een andere locatie een bestand aan te maken en later te lezen. Deze onafhankelijkheid laat ook toe om contexten in parallel te verwerken, waarover meer in paragraaf 2.6.3.

Testcase Een context bestaat uit een of meerdere *testcase*s of testgevallen. Een testgeval bestaat uit invoer en een aantal tests. De testgevallen kunnen onderverdeeld worden in twee soorten:

Context testcase of contexttestgeval. Van deze soort is er maximaal één per context (geen is ook mogelijk). Dit testgeval heeft als doel het uitvoeren van de main-functie (of de code zelf als het gaat om een scripttaal zoals Bash of Python). Als invoer voor dit testgeval kunnen enkel de standaardinvoerstroom, de commandoargumenten en de bestanden in de *workdir* meegegeven worden. De exitcode van een uitvoering kan ook enkel in dit testgeval gecontroleerd worden.

Normal testcase of normaal testgeval. Hiervan kunnen er nul of meer zijn per context. Deze testgevallen dienen om andere aspecten van de ingediende oplossing te testen, nadat de code van de gebruiker met success ingeladen is. De invoer is dan ook uitgebreider: het kan gaan om de standaardinvoerstroom, functieoproepen en variabeletoekenningen. Een functieoproep of variabeletoekenning is verplicht (zonder functieoproep of toekenning aan een variabele is er geen code die een resultaat zou kunnen produceren).

Het contexttestgeval wordt altijd als eerste uitgevoerd. Dit is verplicht omdat bepaalde programmeertalen (zoals Python en andere scripttalen) de code onmiddellijk uitvoeren bij het inladen. Om te vermijden dat de volgorde van de testgevallen zou verschillen tussen de programmeertalen, wordt het contexttestgeval altijd eerst uitgevoerd.

Test De beoordeling van een testgeval bestaat uit meerdere *test*s, die elk één aspect van het testgeval controleren. Met aspect bedoelen we de standaarduitvoerstroom, de standaardfoutstroom, opgevangen uitzonderingen (*exceptions*), de teruggegeven waarden van een functieoproep (returnwaarden) of de inhoud van een bestand. De exitcode is ook mogelijk, maar enkel in het contexttestgeval. Het beoordelen van de verschillende aspecten wordt in meer detail beschreven in paragraaf 2.5.

Bij de keuze van een formaat voor het testplan (JSON, XML, ...), zijn vooraf enkele vereisten geformuleerd waaraan het gekozen formaat moet voldoen. Het moet:

- leesbaar zijn voor mensen,
- geschreven kunnen worden met minimale inspanning, met andere woorden de syntaxis dient eenvoudig te zijn, en
- programmeertaalonafhankelijk zijn.

Uiteindelijk is gekozen om het testplan op te stellen in JSON. Niet alleen voldoet JSON aan de vooropgestelde voorwaarden, het wordt ook door veel talen ondersteund.

Toch zijn er ook enkele nadelen aan het gebruik van JSON. Zo is JSON geen beknopte of compacte taal om met de hand te schrijven. Een oplossing hiervoor gebruikt de eigenschap dat veel talen JSON kunnen produceren: andere programma's kunnen desgewenst het testplan in het JSON-formaat genereren, waardoor het niet met de hand geschreven moet worden. Hiervoor denken we aan een DSL (*domain specific language*), maar dit valt buiten de masterproef en wordt verder besproken in paragraaf 5.2.1.

Een tweede nadeel is dat JSON geen programmeertaal is. Terwijl dit de implementatie van de judge bij het interpreteren van het testplan weliswaar eenvoudiger maakt, is het tevens beperkend: beslissen of een testgeval moet uitgevoerd worden op basis van het resultaat van een vorig testgeval is bijvoorbeeld niet mogelijk. Deze beperking wordt uitgebreider besproken in paragraaf 5.2.2.

Tot slot bevat codefragment 2.1 ter illustratie van het formaat een testplan met één context voor de voorbeeldoefening Lotto uit hoofdstuk 1.

2.3.2. Dataserialisatie

Bij de beschrijving van de structuur van een testplan wordt gewag gemaakt van returnwaarden en variabeletoekenningen. Aangezien het testplan programmeertaalonafhankelijk is, moet er dus een manier zijn om data uit de verschillende programmeertalen voor te stellen en te vertalen: het *serialisatieformaat*.

Codefragment 2.1. Een ingekorte versie van het testplan voor de voorbeeldoefening Lotto. Het testplan bevat één tabblad met één context.

```
{
  "tabs": [
    {
      "name": "Feedback",
      "contexts": [
        {
          "input": {
            "function": {
              "type": "function",
              "name": "loterij",
              "arguments": [
                { "type": "integer", "data": 6 },
                { "type": "integer", "data": 15 }
              ]
            }
          },
          "output": {
            "result": {
              "value": {
                "type": "text",
                "data": "1 - 6 - 7 - 11 - 13 - 14"
              },
              "evaluator": {
                "type": "custom",
                "language": "python",
                "path": "./evaluator.py",
                "arguments": [
                  { "type": "integer", "data": 6 },
                  { "type": "integer", "data": 15 }
                ]
              }
            }
          }
        }
      ]
    }
  ]
}
```

Keuze van het formaat

Zoals bij het testplan, werd voor de voorstelling van waarden ook een keuze voor een bepaald formaat gemaakt. Daarvoor werden opnieuw enkele voorwaarden vooropgesteld waaraan het serialisatieformaat moet voldoen. Het formaat moet:

- door mensen geschreven en gelezen kunnen worden,
- onderdeel van het testplan kunnen zijn,
- in meerdere programmeertalen bruikbaar zijn,
- het mogelijk maken om de basisgegevenstypes te ondersteunen die we willen aanbieden in het programmeertaalonafhankelijke deel van het testplan. Deze gegevenstypes zijn:
  - Primitieven: gehele getallen, reële getallen, Boolese waarden en tekenreeksen.
  - Collecties: rijen (eindige, geordende reeks; `list` of `array`), verzamelingen (eindige, ongeordende reeks zonder herhaling; `set`) en afbeeldingen (elk element wordt afgebeeld op een ander element; `map`, `dict` of `object`).

Een voor de hand liggende oplossing is om ook hier JSON te gebruiken, en zelf in JSON een structuur op te stellen voor de waarden. In tegenstelling tot de situatie bij het testplan bestaan er al een resem dataserialisatieformaten, waardoor het de moeite loont om na te gaan of er geen bestaand formaat voldoet aan de vereisten. Hiervoor is gestart van een overzicht op Wikipedia [8]. Uiteindelijk is niet gekozen voor een bestaand formaat, maar voor de JSON-oplossing. De redenen hiervoor zijn samen te vatten als:

- Het gaat om een binair formaat. Binaire formaten zijn uitgesloten op basis van de eerste twee voorwaarden die we opgesteld hebben: mensen kunnen het niet schrijven zonder hulp van bijkomende tools en het is moeilijk in te bedden in een JSON-bestand (zonder gebruik te maken van encoderingen zoals base64). Bovendien zijn binaire formaten moeilijker te implementeren in sommige talen.
- Het formaat ondersteunt niet alle gewenste types. Sommige formaten hebben ondersteuning voor complexere datatypes, maar niet voor alle complexere datatypes die wij nodig hebben. Uiteraard kunnen de eigen types samengesteld worden uit basistypes, maar dan biedt de ondersteuning voor de complexere types weinig voordeel, aangezien er toch een eigen dataschema voor die complexere types opgesteld zal moeten worden.
- Sommige formaten hebben een ander doel, wat zich uit in functionaliteit die wij niet nodig hebben, met als koploper OPC Unified Architecture, waarvan de specificatie ongeveer 1250 pagina's beslaat. Als deze overbodige functionaliteit optioneel is, vormt deze functionaliteit geen nadeel voor het formaat, maar ook geen voordeel.
- Het formaat is niet eenvoudig te implementeren in een programmeertaal waarvoor geen ondersteuning is. Sommige dezer formaten ondersteunen weliswaar veel programmeertalen, maar we willen niet dat het serialisatieformaat een beperkende factor wordt in welke talen door TESTed ondersteund worden. Het mag niet de bedoeling zijn dat het implementeren van het serialisatieformaat het meeste tijd in beslag neemt.

Voor alle bestaande formaten die geen variant van JSON zijn, is er nog een bijkomend nadeel: het inbedden in het JSON-testplan wordt moeilijker. Een lijst van de overwogen formaten met een korte beschrijving:

Apache Avro Een volledig „systeem voor dataserialisatie”. De specificatie van het formaat gebeurt in JSON (vergelijkbaar met JSON Schema), terwijl de eigenlijke data binair geëncodeerd wordt. Heeft uitbreidbare types, met veel ingebouwde types [9].

Apache Parquet Minder relevant, dit is een bestandsformaat voor Hadoop [10].

ASN.1 Staat voor *Abstract Syntax Notation One*, een formaat uit de telecommunicatie. De hoofdstandaard beschrijft enkel de notatie voor een dataformaat. Andere standaarden beschrijven dan de serialisatie, bijvoorbeeld een binair formaat, JSON of XML. De meerdere serialisatievormen zijn in theorie aantrekkelijk: elke taal moet er slechts een ondersteunen, terwijl de judge ze allemaal kan ondersteunen. In de praktijk blijkt echter dat voor veel talen er slechts één serialisatieformaat is, en dat dit vaak het binaire formaat is [11].

Bencode Schema gebruikt in BitTorrent. Het is gedeeltelijk binair, gedeeltelijk in text [12].

Binn Binair dataformaat [13].

BSON Een binaire variant op JSON, geschreven voor en door MongoDB [14].

CBOR Een lichtjes op JSON gebaseerd formaat, ook binair. Heeft een goede standaard, ondersteunt redelijk wat talen [15].

FlatBuffers Lijkt op ProtocolBuffers, allebei geschreven door Google, maar verschilt wat in implementatie van ProtocolBuffers. De encodering is binair [16].

Fast Infoset Is eigenlijk een manier om XML binair te encoderen (te beschouwen als een soort compressie voor xml), waardoor het minder geschikt voor ons gebruik wordt [17].

Ion Een superset van JSON, ontwikkeld door Amazon. Het heeft zowel een tekstuele als binaire voorstelling. Naast de gebruikelijke JSON-types, bevat het enkele uitbreidingen. [18].

MessagePack Nog een binair formaat dat lichtjes op JSON gebaseerd is. Lijkt qua types sterk op JSON. Heeft implementaties in veel talen [19].

OGDL Afkorting voor *Ordered Graph Data Language*. Daar het om een serialisatieformaat voor grafen gaat, is het niet nuttig voor ons doel [20].

OPC Unified Architecture Een protocol voor intermachinecommunicatie. Complex: de specificatie bevat 14 documenten, met ongeveer 1250 pagina’s [21].

OpenDLL Afkorting voor de *Open Data Description Language*. Een tekstueel formaat, bedoeld om arbitraire data voor te stellen. Wordt niet ondersteund in veel programmeertalen, in vergelijking met bijvoorbeeld JSON [22].

ProtocolBuffers Lijkt zoals vermeld sterk op FlatBuffers, maar heeft nog extra stappen nodig bij het encoderen en decoderen, wat het minder geschikt maakt [23].

Smile Nog een binaire variant van JSON [24].

SOAP Afkorting voor *Simple Object Access Protocol*. Niet bedoeld als formaat voor dataserialisatie, maar voor communicatie tussen systemen over een netwerk [25].

SDXF Binair formaat voor data-uitwisseling. Weinig talen ondersteunen dit formaat [26].

Thrift Lijkt sterk op ProtocolBuffers, maar geschreven door Facebook [27].

UBJSON Nog een binaire variant van JSON [28].

Codefragment 2.2. Een lijst bestaande uit twee getallen, geëncodeerd in het serialisatieformaat.

```
{
  "type": "sequence",
  "data": [
    { "type": "integer", "data": 5 },
    { "type": "integer", "data": 15 }
  ]
}
```

Dataschema

JSON is slechts een formaat en geeft geen semantische betekenis aan JSON-elementen. We moeten nog de structuur van het serialisatieformaat vastleggen: het dataschema. Dit dataschema bestaat uit twee grote onderdelen:

- Het encoderen van waarden. (*Hoe zet ik deze lijst getallen om naar JSON?*)
- Het beschrijven van de gegevenstypes van deze waarden. (*Hoe geef ik mee dat de lijst integers van 64 bit bevat?*)

Elke waarde wordt in het serialisatieformaat voorgesteld als een object met twee elementen: de geëncodeerde waarde en het bijhorende gegevenstype. Dit gebeurt recursief: bij het serialiseren van een verzameling worden alle elementen van de verzameling ook geserialiseerd (en zullen dus ook bestaan uit twee elementen: het gegevenstype en de geëncodeerde waarde). Een concreet voorbeeld is codefragment 2.2.

Het encoderen van waarden slaat op het voorstellen van waarden in JSON. JSON heeft slechts een beperkt aantal gegevenstypes, dus worden alle waarden voorgesteld als een van deze types. Zo worden bijvoorbeeld zowel `arrays` en `sets` voorgesteld als een JSON-lijst. Het verschil tussen beiden wordt dan duidelijk gemaakt door het bijhorende gegevenstype. Er is dus nood aan een systeem om aan te geven wat het gegevenstype van een waarde is.

Codefragment 2.3 bevat onder andere de structuur van een waarde (een `Literal`) in het serialisatieformaat, in een vereenvoudigde versie van JSON Schema. Hierbij staat `<types>` voor een van de gegevenstypes die hierna besproken werden.

Codefragment 2.3. Het schema voor waarden, expressies en statements, in een vereenvoudigde versie van JSON Schema.

```
{
  "Literal": {
    "type": "object",
    "properties": {
      "data": {"type": ["number", "string", "boolean", "object", "array", "null"]},
      "type": {"type": "string", "enum": ["<types>"]}
    }
  },
  "FunctionCall": {
    "type": "object",
    "properties": {
      "type": {"enum": ["function", "namespace", "constructor", "property"]},
      "namespace": {"type": "string"},
      "name": {"type": "string"},
      "arguments": {
        "type": "array",
        "items": {"anyOf":  [{"$ref": "#Expression"}, {"$ref": "#NamedArgument"}]}
      }
    }
  },
  "Identifier": {"type": "string"},
  "Expression": {
    "anyOf": [{"$ref": "#Identifier"}, {"$ref": "#Literal"}, {"$ref":
    ↪  "#FunctionCall"}]
  },
  "NamedArgument": {
    "type": "object",
    "properties": {
      "name": {"type": "string"},
      "value": {"type": "#Expression"}
    }
  },
  "Assignment": {
    "type": "object",
    "properties": {
      "name": {"type": "string"},
      "expression": {"type": "#Expression"},
      "type": "<datatype>"
    }
  },
  "Statement": {
    "anyOf": [{"$ref": "#Assignment"}, {"$ref": "#Expression"}]
  }
}
```

Tabel 2.1. Implementaties van de basistypes in de verschillende programmeertalen.

Basistype	Python	Java	Haskell	C
`integer`	`int`	`long`	`Integer`	`long long`
`rational`	`float`	`double`	`Double`	`double`
`text`	`str`	`String`	`String`	`char*`
`char`	`str`	`char`	`Char`	`char`
`boolean`	`bool`	`boolean`	`Boolean`	`bool`
`sequence`	`list`	`List<>`	`List`	-
`set`	`set`	`Set<>`	-	-
`map`	`dict`	`Map<>`	-	-
`nothing`	`None`	`null`	`Nothing`	`NULL`

Tabel 2.2. Voorbeeld van de implementatie van geavanceerde types voor gehele getallen, met als basistype `integer`.

	`int8`	`uint8`	`int16`	`uint16`	`int32`	`uint32`	`int64`	`uint64`
Python	`int`	`int`	`int`	`int`	`int`	`int`	`int`	`int`
Java	`byte`	`short`	`short`	`int`	`int`	`long`	`long`	-
C[1]	`int8_t`	`uint8_t`	`int16_t`	`uint16_t`	`int32_t`	`uint32_t`	`int64_t`	`uint64_t`
Haskell	`Int8`	`Word8`	`Int16`	`Word16`	`Int32`	`Word32`	`Int64`	`Word64`

[1] Uiteraard met de gebruikelijke aliassen van `short`, `unsigned`, ...

Gegevenstypes

Het systeem om de gegevenstypes aan te duiden vervult meerdere functies. Het wordt gebruikt om:

- het gegevenstype van concrete data aan te duiden (beschrijvende modus). Dit gaat om de serialisatie van waarden uit de uitvoeringsomgeving naar TESTed, zoals het geval is bij returnwaarden van functies.
- te beschrijven welk gegevenstype verwacht wordt (voorschrijvende modus). Een voorbeeld hiervan is het aangeven van het gegevenstype van een variabele.
- zelf code te schrijven (letterlijke modus). Dit gaat om serialisatie vanuit het testplan zelf naar de uitvoeringsomgeving. Een voorbeeld hiervan is het opnemen van functieargumenten in het testplan: deze argumenten worden tijdens de serialisatie omgezet naar echte code.

Bij het ontwerp van het systeem voor de gegevenstypes zorgen deze verschillende functies soms voor tegenstrijdige belangen: voor het beschrijven van een waarde moet het systeem zo eenvoudig mogelijk zijn. Een waarde met bijhorend gegevenstype `union[string, int]` is niet bijster nuttig: een waarde kan nooit tegelijk een `string` en een `int` zijn. Aan de andere kant zijn dit soort complexe gegevenstypes wel nuttig bij het aangeven van het verwachte gegevenstype van bijvoorbeeld een variabele.

Daarnaast moet ook rekening gehouden worden met het feit dat deze gegevenstypes in veel programmeertalen implementeerbaar moeten zijn. Een gegevenstype als `union[string, int]` is eenvoudig te implementeren in Python, maar dat is niet het geval in bijvoorbeeld Java of C. Ook heeft elke programmeertaal een eigen niveau van details bij gegevenstypes. Python heeft bijvoorbeeld enkel `integer` voor gehele getallen, terwijl C beschikt over `int`, `unsigned`, `long`, enz.

Tot slot heeft het schrijven van code bijkomende vereisten: als functieargument zijn waarden alleen niet voldoende, ook variabelen en andere functieoproepen moeten gebruikt kunnen worden. Om deze redenen zijn de gegevenstypes opgedeeld in drie categorieën:

1. De basistypes. Deze gegevenstypes zijn bruikbaar in alle modi. De lijst van basistypes omvat:

   `integer` Gehele getallen, zowel positief als negatief.

   `rational` Rationale getallen. Het gaat hier om `floats`, die ook vaak gebruikt worden als benadering van gehele getallen.

   `text` Een tekenreeks of string (alle vormen).

   `char` Een enkel teken.

   `boolean` Een Boolese waarde (of boolean).

   `sequence` Een wiskundige rij: de volgorde van de elementen is belangrijk en dubbele elementen zijn toegelaten.

   `set` Een wiskundige verzameling: de volgorde van de elementen is niet belangrijk is en dubbele elementen zijn niet toegelaten.

   `map` Een wiskundige afbeelding: elk element wordt afgebeeld op een ander element. In Java is dit bijvoorbeeld een `Map`, in Python een `dict` en in Javascript een `object`.

   `nothing` Geeft aan dat er geen waarde is, ook wel `null`, `None` of `nil` genoemd.

   Een lijst van de implementaties in de verschillende programmeertalen staat in tabel 2.1. Elke implementatie van een programmeertaal moet een keuze maken wat de standaardimplementatie van deze types is. Zo implementeert de Java-implementatie het gegevenstype `sequence` als een `List<>`, niet als een `array`. Een implementatie in een programmeertaal kan ook aangeven dat een bepaald type niet ondersteund wordt, waardoor testplannen met dat type niet zullen werken.

2. De uitgebreide types: dit zijn een hele reeks bijkomende types. Deze gegevenstypes staan toe om meer details over de types te serialiseren en in het testplan op te nemen. Een voorbeeld is de lijst van types in tabel 2.2, die voor een reeks gegevenstypes voor gehele getallen de concrete types in verschillende programmeertalen geeft. Het grote verschil is dat deze uitgebreide types standaard vertaald worden naar een van de basistypes. Voor programmeertalen die bijvoorbeeld geen `tuple` uit Python ondersteunen, zal het type omgezet worden naar `list`. Er is ook de mogelijk dat implementaties voor programmeertalen expliciet een bepaald type niet ondersteunen.

3. Voorschrijvende types. Gegevenstypes in deze categorie kunnen enkel gebruikt worden bij het aangeven welk gegevenstype verwacht wordt, niet bij de eigenlijke encodering van waarden. In de praktijk gaat het enkel om het type van variabelen. In deze categorie zouden gegevenstypes als `union[str, int]` komen. Er is echter expliciet gekozen om dit soort types niet te ondersteunen, door de moeilijkheid om dit te implementeren in statisch getypeerde talen, zoals Java of C. Twee types die wel ondersteund worden in deze modus zijn:

   `any` Het `any`-type geeft aan dat het type van een variabele onbekend is. Merk op dat dit in sommige talen tot moeilijkheden zal leiden: zo zal dit in C-code als `long` beschouwd worden (want C heeft geen equivalent van een `any`-type). Dit gegevenstype is ook niet bijster nuttig, en we raden aan het niet te gebruiken.

custom Een eigen type, waarbij de naam van het type gegeven wordt. Dit is nuttig om bijvoorbeeld variabelen aan te maken met als gegevenstype een eigen klasse, zoals een klasse die de student moest implementeren.

Het is het vermelden waard dat TESTed geen formeel typesysteem gebruikt. Het doel van een formeel typesysteem is het beschrijven van welke categorieën data een constructie (een variabele, parameter, enz.) kan zijn. Dit is in tegenstelling tot TESTed, waar meestal concrete gegevens beschreven worden. Zo bevat specificeert het testplan niet „een functie met returntype `int64`", maar wel „een functie met als verwachte waarde 16, wier gegevenstype `int64` is". De enige uitzondering is het gegevenstype van variabelen aanduiden, maar zelfs daar gaat het altijd om het type van een variabele met een concrete expressie die een resultaat oplevert. Als gevolg maakt TESTed geen gebruik van technieken die normaliter geassocieerd worden met typesystemen, zoals een formele typeverificatie.

2.3.3. Expressions en statements

Een ander onderdeel van het testplan verdient ook speciale aandacht: toekennen van waarden aan variabelen (*assignments*) en functieoproepen.

In heel wat oefeningen, en zeker bij objectgerichte en imperatieve programmeertalen, is het toekennen van een waarde aan een variabele, om deze later te gebruiken, onmisbaar. Een opgave zou bijvoorbeeld kunnen bestaan uit het implementeren van een klasse. Bij de evaluatie dient dan een instantie van die klasse aangemaakt te worden, waarna er methoden kunnen aangeroepen worden, zoals geïllustreerd in het fictieve voorbeeld hieronder:

```
var variabele = new DoorDeStudentGemaakteKlasse();
assert variabele.testfunctie1() == 15;
assert variabele.testfunctie2() == "Vijftienduizend";
assert variabele.eigenschap == 42;
```

Om deze reden is het testplan uitgebreid met ondersteuning voor statements en expressies. Toch moet meteen opgemerkt worden dat deze ondersteuning beperkt is tot wat er nodig is om het scenario uit het voorbeeld van hiervoor te kunnen uitvoeren; het is zeker niet de bedoeling om een volledige eigen programmeertaal te ontwerpen.

In codefragment 2.3 staat onder andere ook de structuur van een expressie en een functieoproep in een vereenvoudigde versie van JSON Schema. Een expressie is een van deze drie dingen:

1. Een waarde, zoals besproken in subparagraaf *Dataschema* van paragraaf 2.3.2.
2. Een `identifier`, voorgesteld als een string. Dit om te verwijzen naar een variabele die eerder gemaakt is met een assignment.
3. Een functieoproep, die bestaat uit volgende elementen:

   `type` Het soort functie. Kan een van deze waarden zijn:

   `function` Een *top-level* functie. Afhankelijk van de programmeertaal zal deze functie toch omgezet worden naar een `namespace`-functie. Zo worden dit soort functies in Java omgezet naar statische functies.

`namespace` Een methode (functie van een object) of een functie in een namespace. De invulling hiervan is gedeeltelijk programmeertaalafhankelijk: in Java gaat het om methodes, terwijl het in Haskell om functies van een module gaat. Bij dit soort functies moet de `namespace` gegeven worden.

`constructor` Dit soort functie heeft dezelfde semantiek als een top-level functie, met dien verstande dat het om een constructor gaat. In Java zal bijvoorbeeld het keyword `new` vanzelf toegevoegd worden. De functienaam doet dienst als naam van de klasse.

`property` De property van een instantie wordt gelezen. Dit soort functie heeft dezelfde semantiek als een `namespace`-functie, maar heeft geen argumenten.

`namespace` De namespace van de functie.

`name` De naam van de functie.

`arguments` De argumenten van de functie. Dit is een lijst van ofwel expressies ofwel *named arguments*. Een named argument bestaat uit een naam en een expressie.

In het serialisatieformaat is een statement ofwel een variabeletoekenningen of *assignment*, ofwel een expressie. Een expressie is zoals reeds besproken. Een assignment kent een naam toe aan het resultaat van een expression. Codefragment 2.3 toont ook het vereenvoudigde JSON Schema van een statement (en dus van een assignment, daar er maar één soort statement bestaat). Hier staat `<datatype>` voor een van de gegevenstypes die hiervoor besproken zijn.

De `name` is de naam die aan de variabele gegeven zal worden. Het veldje `expression` moet een expressie zijn, zoals reeds besproken. Ook moet het gegevenstype van de variabele gegeven worden. Hiervoor kunnen types uit het serialisatieformaat gebruikt worden, inclusief de types uit de letterlijke modus. Hieronder staat een concreet voorbeeld van hoe dit eruitziet in. De string `'Dodona'` wordt toegekend aan een variabele met naam `orakel`.

```
{
  "variable": "orakel",
  "expression": {
    "type": "text",
    "value": "Dodona"
  },
  "type": "text"
}
```

Tot slot is het nog het vermelden waard dat waarden van de gegevenstypes `sequence` en `map` als elementen geen andere waarden hebben, maar expressies. Dit is niet het geval in de beschrijvende modus van de gegevenstypes, bijvoorbeeld bij het aangeven wat de verwachte returnwaarde van een functie is. Het testplan biedt namelijk geen ondersteuning voor het serialiseren van identifiers en functieoproepen, enkel waarden. Codefragment 2.4 bevat een voorbeeld van geavanceerde constructies die voorgesteld kunnen worden in het testplan.

Codefragment 2.4. Geavanceerde constructies die voorgesteld kunnen worden in het testplan.

```
var variabele1 = functie1();
var variabele2 = [variabele1, 15, functie2()];
var object = new Test("een string als argument");

assert object.process(variabele2, [functie1()]) == 85;
```

2.3.4. Controle van ondersteuning voor programmeertalen

In het stappenplan uit paragraaf 2.2 is al vermeld dat vóór een beoordeling start, een controle plaatsvindt om zeker te zijn dat het testplan uitgevoerd kan worden in de programmeertaal van de ingediende oplossing. Bij de controle worden volgende zaken nagekeken:

- Controle of de programmeertaal de nodige gegevenstypes ondersteunt. Dit gaat van de basistypes (zoals `sequence`) tot de geavanceerde types (zoals `tuple`). Voor elke programmeertaal binnen TESTed wordt bijgehouden welke types ondersteund worden en welke niet. Bevat een testplan bijvoorbeeld waarden met als type `set` (verzamelingen), dan kunnen enkel programmeertalen die verzamelingen ondersteunen gebruikt worden. Dat zijn bijvoorbeeld Python en Java, maar geen Bash.
- Controle of de programmeertaal over de nodige taalconstructies beschikt, zoals `exceptions` of `objects`. Bij deze controle wordt ook gecontroleerd of dat de programmeertaal optionele functieargumenten of functieargumenten met heterogene gegevenstypes nodig heeft. Een voorbeeld van een functie met argumenten met heterogene gegevenstypes komt bijvoorbeeld uit de ISBN-oefening (deze oefening wordt besproken in hoofdstuk 3):

  ```
  >>> is_isbn("9789027439642")
  True
  >>> is_isbn(9789027439642)
  False
  ```

  In talen als Python en Java kan deze functie geïmplementeerd worden, maar in talens als C is dat veel moeilijker.

2.4. Oplossingen uitvoeren

De eerste stap die wordt uitgevoerd bij het beoordelen van een ingediende oplossing is het genereren van de testcode, die de ingediende oplossing en de testgevallen zal uitvoeren.

2.4.1. Testcode genereren

Het genereren van de testcode gebeurt met een sjabloonsysteem genaamd Mako [29]. Dit soort systemen wordt traditioneel gebruikt bij webapplicaties (zoals Ruby on Rails met ERB, Phoenix met EEX, Laravel met Blade, enz.) om bijvoorbeeld html-pagina's te genereren. In ons geval zijn de sjablonen verantwoordelijk voor de vertaling van programmeertaalonafhankelijke specificaties in het testplan naar concrete testcode in de programmeertaal van de ingediende oplossing. Hierbij

denken we aan de functieoproepen, assignments, enz. De sjablonen zijn ook verantwoordelijk voor het genereren van de code die de oplossing van de student zal oproepen en evalueren.

Sjablonen

TESTed gebruikt een aantal standaardsjablonen, waaraan vastgelegde parameters meegegeven worden en die verantwoordelijk zijn voor specifieke functionaliteit. Deze verplichte sjablonen zijn:

`statement` Vertaalt een statement uit het testplan naar code.

`context` Genereert de code om de testgevallen van een context uit te voeren.

`selector`* Leest een argument in en voert aan de hand van dat argument de juiste context uit bij de batchcompilatie (zie de volgende subparagraaf en paragraaf 2.6).

`evaluator_executor`* Genereert de code om een geprogrammeerde evaluatie te starten.

De sjablonen met een ster zijn enkel verplicht indien de gerelateerde functionaliteit gebruikt wordt. Zo zal het niet nodig zijn om het selectorsjabloon te implementeren in Python, want Python gebruikt geen selector.

Verplichte sjablonen kunnen zelf ook andere sjablonen gebruiken als hulpmiddel voor de configuratie of om delen van een sjabloon op meerdere plaatsen te kunnen gebruiken. Een voorbeeld is het vertalen van een functieoproep naar code, iets dat op meerdere plaatsen moet gebeuren. Dit is een standaardvoorziening van Mako.

Tot slot moet ook het serialisatieformaat geïmplementeerd worden in de taal. Waarden moeten geserialiseerd kunnen worden en statements en expressies moeten gedeserialiseerd kunnen worden. Hoe het serialiseren van waarden moet gebeuren is vrij, maar wordt in alle bestaande programmeertalen geïmplementeerd met een bijkomende klasse of module (die de naam `Value` krijgt). Het deserialiseren ligt wel vast: hiervoor moet het sjabloon `statement` uit bovenstaande lijst geïmplementeerd worden. In alle bestaande programmeertalen is het deserialiseren verder opgesplitst in een sjabloon voor expressies, een sjabloon voor functieoproepen en meerdere sjablonen voor waarden. Zoals al vermeld is deze verdere opsplitsing echter geen vereiste van TESTed. Dit alles wordt geïllustreerd in hoofdstuk 4, dat volledig uitwerkt hoe een nieuwe programmeertaal aan TESTed toegevoegd kan worden.

Testcode compileren

TESTed ondersteunt twee manieren waarop de code kan gecompileerd worden (bij programmeertalen die geen compilatie ondersteunen wordt deze stap overgeslagen):

Batchcompilatie Bij deze manier wordt de code voor alle contexten in een keer gecompileerd. Dit wordt gedaan om performantieredenen. In talen die resulteren in een uitvoerbaar bestand (zoals Haskell of C/C++), resulteert deze modus in één uitvoerbaar bestand voor alle contexten. Bij het uitvoeren wordt dan aan de hand van een argument de juiste context uitgevoerd (met het `selector`-sjabloon).

Contextcompilatie Hierbij wordt elke context afzonderlijk gecompileerd tot een uitvoerbaar bestand.

Dit wordt weergegeven door verschillende kleuren in figuur 2.2: de stappen die enkel plaatsvinden bij batchcompilatie zijn in het groen, terwijl de stappen die enkel bij contextcompilatie plaatsvinden in het blauw staan. Stappen die altijd uitgevoerd worden staan in de flowchart in het zwart.

Batchcompilatie resulteert in een grote tijdswinst (zie paragraaf 2.6), maar heeft wel een groot nadeel: bij statische talen zorgen compilatiefouten in één context ervoor dat geen enkele context beoordeeld kan worden, daar de compilatiefout ervoor zorgt dat er geen code gegenereerd wordt. Stel bijvoorbeeld een oefening waarbij studenten twee functies moeten implementeren. Met batchcompilatie in bijvoorbeeld Java kan de student niet de oplossing voor de eerste functie indienen en laten beoordelen: omdat de tweede functie ontbreekt zullen compilatiefouten optreden.

Als oplossing hiervoor bevat TESTed de mogelijkheid om contextcompilatie en batchcompilatie te combineren, waarbij TESTed terugvalt op de contextcompilatie indien er iets misgaat bij de batchcompilatie. Dit terugvalmechanisme is geen geavanceerd systeem: faalt de compilatie, wordt onmiddellijk elke context afzonderlijk gecompileerd. Een mogelijke optimalisatie bestaat er in een tussenoplossing toe te passen, door bijvoorbeeld compilatie op het niveau van tabbladen proberen.

In appendices B en C wordt de gegenereerde testcode getoond voor enkele eenvoudige oefeningen.

2.4.2. Testcode uitvoeren

Vervolgens wordt de (gecompileerde) testcode afzonderlijk uitgevoerd voor elke context uit het testplan en worden de resultaten (het gedrag en de neveneffecten) verzameld. Het uitvoeren zelf gebeurt op de normale manier waarop code voor de programmeertaal uitgevoerd wordt: via de commandoregel (bijvoorbeeld `python context0.py`). Deze aanpak heeft als voordeel dat er weinig verschil is tussen hoe TESTed de ingediende code uitvoert en hoe de student zijn code zelf uitvoert op zijn eigen computer. Dit voorkomt dat er subtiele verschillen in de resultaten sluipen.

In appendix A staat de concrete mapstructuur van de werkmap na het uitvoeren van een beoordeling, alsook de mapstructuur na het uitvoeren van een geprogrammeerde evaluatie.

2.4.3. Beoordelen van gedrag en neveneffecten

Het uitvoeren van de testcode genereert resultaten (het gedrag en de neveneffecten) die door TESTed beoordeeld moeten worden. Er zijn verschillende soorten gedragingen en neveneffecten die interessant zijn. Elke soort gedrag of neveneffect wordt een *uitvoerkanaal* genoemd. TESTed verzamelt volgende uitvoerkanalen:

- De standaarduitvoerstroom. Dit wordt verzameld als tekstuele uitvoer.
- De standaardfoutstroom. Ook dit wordt als tekstuele uitvoer verzameld.
- Fatale uitzonderingen. Hiermee bedoelen we uitzonderingen die tot aan het uitvoeren van de testcode geraken. Een uitzondering die afgehandeld wordt door de ingediende oplossing wordt niet verzameld. De uitzonderingen worden verzameld in een bestand.
- Returnwaarden. Deze waarden worden geserialiseerd (met het serialisatieformaat) en ook verzameld in een bestand.

- Exitcode. Het gaat om de exitcode na het uitvoeren van de testcode voor een context. Daar de code per context wordt uitgevoerd, wordt de exitcode ook verzameld per context (en niet per testgeval zoals de andere uitvoerkanalen).
- Bestanden. Tijdens het beoordelen van de verzamelde resultaten is het mogelijk de bestanden te beoordelen die aangemaakt zijn bij het uitvoeren van de ingediende oplossing of de testgevallen uit de context.

De standaarduitvoer- en standaardfoutstroom worden rechtstreeks opgevangen door TESTed. De andere uitvoerkanalen (uitzonderingen en returnwaarden) worden naar een bestand geschreven. De reden dat deze niet naar een andere *file descriptor* geschreven worden is eenvoudig: niet alle talen (zoals Java) ondersteunen het openen van bijkomende file descriptors.

Alle uitvoerkanalen (met uitzondering van de exitcode en de bestanden) worden per testgeval verzameld. Aangezien de uitvoerkanalen pas verzameld worden na het uitvoeren van de context, moet er een manier zijn om de uitvoer van de verschillende testgevallen te onderscheiden. De testcode is hiervoor verantwoordelijk, en schrijft een *separator* naar alle uitvoerkanalen voor elk testgeval. Nemen we bijvoorbeeld het kanaal voor returnwaarden, dan ziet de separator gevolgd door een returnwaarde er zo uit:

```
--gL9koJNv3-- SEP{"data":"1 - 3 - 6 - 8 - 10 - 15","type":"text"}
```

Tijdens het genereren van de code krijgen de sjablonen een reeks willekeurige tekens mee: de *secret*. Deze secret wordt gebruikt voor verschillende dingen, zoals:

- De separator. Door het gebruik van de willekeurige tekens is de kans dat de separator overeenkomt met een echte waarde praktisch onbestaand.
- Bestandsnamen. De testcode is verantwoordelijk voor het openen van de bestanden voor de uitvoerkanalen die naar een bestand geschreven worden. Bij het openen zal de testcode de secret in de bestandsnaam gebruiken. Dit is om het per abuis overschrijven van deze bestanden door de ingediende oplossing tegen te gaan.

2.5. Oplossingen beoordelen

In de vorige stap heeft TESTed de resultaten voor elke context verzameld. Het geheel van de verzamelde resultaten noemen we de uitvoer van een context. Deze uitvoer moet vervolgens beoordeeld worden om na te gaan in hoeverre deze voldoet aan de verwachte uitvoer. Dit kan op drie manieren:

1. Generieke evaluatie: de uitvoer wordt beoordeeld door TESTed zelf.
2. Geprogrammeerde evaluatie: de uitvoer wordt beoordeeld door evaluatiecode, in de evaluatieomgeving (een nieuw proces).
3. Programmeertaalspecifieke evaluatie: de uitvoer wordt onmiddellijk na het uitvoeren van de testcode beoordeeld in hetzelfde proces.

Codefragment 2.5. Fragment uit een testplan dat tekstevaluatie gebruikt.

```
{
  "stdout": {
    "type": "text", "data": "3.14",
    "evaluator": {
      "type": "builtin",
      "name": "text",
      "options": {"ignoreWhitespace": true, "tryFloatingPoint": true,
       ↪  "applyRounding": true, "roundTo": 2}
    }
  }
}
```

2.5.1. Generieke evaluatie

Voor eenvoudige beoordelingen (bijvoorbeeld `stdout` vergelijken met een verwachte tekst) volstaat de generieke evaluatie binnen TESTed. TESTed zal verwachte resultaten uit het testplan dan vergelijken met de resultaten geproduceerd door het uitvoeren van de testcode. Als *proof of concept* zijn drie eenvoudige evaluatiemethoden ingebouwd in TESTed, die hieronder besproken worden.

Tekstevaluatie

Deze evaluator vergelijkt de verkregen uitvoer van een uitvoerkanaal (standaarduitvoer, standaardfout, ...) met de verwachte uitvoer uit het testplan. Deze evaluator biedt enkele opties om het gedrag aan te passen:

`ignoreWhitespace` Witruimte voor en na het resultaat wordt genegeerd. Dit gebeurt op de volledige tekst, niet regel per regel.

`caseInsensitive` Er wordt geen rekening gehouden met het verschil tussen hoofdletters en kleine letters.

`tryFloatingPoint` De tekst zal geïnterpreteerd worden als een zwevendekommagetal (*floating-point*). Bij het vergelijken met de verwachte waarde zal de functie `math.isclose()`[1] uit de standaardbibliotheek van Python gebruikt worden.

`applyRounding` Of zwevendekommagetallen afgrond moeten worden tijdens het vergelijken. Indien wel wordt het aantal cijfers genomen van de optie `roundTo`. Na de afronding worden ze ook vergeleken met de functie `math.isclose()`. Deze afronding is enkel van toepassing op het vergelijken, niet op de uitvoer.

`roundTo` Het aantal cijfers na de komma. Enkel nuttig als `applyRounding` waar is.

Deze configuratieopties worden op het niveau van de testen meegegeven. Dit laat toe om voor elke test (zelfs binnen eenzelfde testgeval) andere opties mee te geven. Codefragment 2.5 toont een fragment uit een testplan: de specificatie van `stdout` van een testgeval waarbij de tekstevaluatie gebruikt wordt.

[1] Documentatie is beschikbaar op: https://docs.python.org/3/library/math.html#math.isclose

Codefragment 2.6. Fragment uit een testplan dat bestandsevaluatie gebruikt.

```
{
  "file": {
    "expected_path": "./bestand-uit-de-oefening.txt",
    "actual_path": "./waar-het-verwachte-bestand-komt.txt",
    "evaluator": {
      "type": "builtin",
      "name": "file",
      "options": {
        "mode": "line",
        "ignoreWhitespace": true, "tryFloatingPoint": true, "applyRounding": true,
        ↪  "roundTo": 2
      }
    }
  }
}
```

Bestandsevaluatie

In deze evaluatievorm worden twee bestanden met elkaar vergeleken. Hiervoor bevat het testplan enerzijds een pad naar een bestand dat met de oefening meegeleverd wordt met de verwachte inhoud en anderzijds de naam (of pad) van de locatie waar het verwachte bestand zich moet bevinden. De bestandsevaluatie ondersteunt enkel tekstuele bestanden, geen binaire bestanden. Het vergelijken van de bestanden gebeurt op één van deze manieren:

`full` Beide bestanden worden in hun geheel met elkaar vergeleken.

`line` Elke regel wordt vergeleken met een overeenkomstige regel in het andere bestand. Standaard worden regeleindes niet in overweging genomen bij de vergelijking, maar de optie `stripNewlines` kan op `False` gezet worden om dit toch te doen.

In beide gevallen gebeurt de eigenlijke evaluatie door de tekstevaluator die we zonet besproken hebben. De opties van de bestandsevaluator worden ook doorgegeven aan de tekstevaluator, met als gevolg dat aan de bestandsevaluator ook alle opties van de tekstevaluator meegegeven kunnen worden.

Een voorbeeld van hoe dit eruitziet staat in codefragment 2.6. In dit fragment wordt de modus `line` gebruikt, en worden de opties van de tekstevaluatie ook meegegeven. Het bestand met de verwachte inhoud heeft als naam `bestand-uit-de-oefening.txt` gekregen, terwijl de ingediende oplossing een bestand moet schrijven naar `waar-het-verwachte-bestand-komt.txt`.

Beide paden zijn relatief, maar ten opzichte van andere mappen: het bestand met verwachte inhoud is relatief tegenover de map van de oefening, terwijl het pad waar de ingediende oplossing naar moet schrijven relatief is ten aanzien van de werkmap van de context waarin de oplossing wordt uitgevoerd. De opties in het voorbeeld zullen ervoor zorgen dat elke regel in de bestanden met elkaar wordt vergeleken, met ondersteuning voor kommagetallen en afronding.

Codefragment 2.7. Fragment uit een testplan dat waarde-evaluatie gebruikt.

```
{
  "return": {
    "value": {
      "type": "set",
      "data": [{"type": "integer", "data": 5}, {"type": "integer", "data": 10},
       ↪ {"type": "integer", "data": 15}]
    },
    "evaluator": {"type": "builtin", "name": "value"}
  }
}
```

Waarde-evaluatie

Voor uitvoerkanalen zoals de returnwaarden moet meer dan alleen tekst met elkaar vergeleken kunnen worden. Bevat het testplan de verwachte waarde, dan kan TESTed deze vergelijken met de geproduceerde waarde van de ingediende oplossing. Het vergelijken van een waarde bestaat uit twee stappen:

1. Het gegevenstype wordt recursief vergeleken, waarbij beide waarden hetzelfde type moeten hebben. Hierbij wordt rekening gehouden met de vertalingen tussen de verschillende programmeertalen, waarbij twee gevallen onderscheiden worden:
   a) Specifieert het testplan een basistype, dan zullen alle types die tot dit basistype herleid kunnen worden als hetzelfde beschouwd worden. Is de verwachte waarde bijvoorbeeld `sequence`, zullen ook `arrays` uit Java en `tuples` uit Python goedgekeurd worden.
   b) Specifieert het testplan een uitgebreid type, dan zal het uitgebreid type gebruikt worden voor talen die dat type ondersteunen, terwijl voor andere talen het basistype gebruikt zal worden. Stel dat het testplan bijvoorbeeld een waarde met als gegevenstype `tuple` heeft. In Python en Haskell (twee talen die dat gegevenstype ondersteunen) zullen enkel `tuples` goedgekeurd worden. Voor andere talen, zoals Java, worden alle gegevenstypes goedgekeurd die herleidbaar zijn tot het basistype. Concreet zullen dus `Lists` en `arrays` goedgekeurd worden. Merk op dat momenteel bij collecties (`sequences`, `sets` en `maps`) enkel het type van de collectie gecontroleerd wordt, niet de types van de elementen.
2. De twee waarden worden vergeleken op inhoud (indien de vergelijking van de gegevenstypes uit de vorige stap slaagt). Hierbij maakt TESTed gebruik van de ingebouwde vergelijking van Python om twee waarden te evalueren. De waarden worden eerst naar Python gedeserialiseerd, waarna de vergelijking gebeurt. Dit betekent dat de regels voor *value comparisons* uit Python[2] gevolgd worden. Eén uitzondering is zwevendekommagetallen, waarvoor `math.isclose()` gebruikt wordt.

Bij deze evaluatievorm zijn geen configuratieopties. Een voorbeeld van het gebruik binnen een testplan is codefragment 2.7. Hier wordt als returnwaarde een verzameling met drie elementen (5, 10 en 15) verwacht.

[2] Zie https://docs.python.org/3/reference/expressions.html?highlight=comparison#value-comparisons

2.5.2. Geprogrammeerde evaluatie

Bij oefeningen met niet-deterministische resultaten, zoals de voorbeeldoefening Lotto, kunnen de verwachte waarden niet in het testplan komen. Ook andere oefeningen waar geen directe vergelijking kan gemaakt worden, zoals het uitlijnen van sequenties (*sequence alignment*) uit de bioinformatica, volstaat een vergelijking met een verwachte waarde uit het testplan niet.

Toch is deze evaluatie niet noodzakelijk programmeertaalafhankelijk: de logica om een sequentie uit te lijnen is dezelfde ongeacht de programmeertaal waarin dit gebeurt. Voor dergelijke scenario's is geprogrammeerde evaluatie een oplossing: hierbij wordt code geschreven om de evaluatie te doen, maar deze evaluatiecode staat los van de ingediende oplossing en moet ook niet in dezelfde programmeertaal geschreven zijn. Binnen TESTed wordt dit mogelijk gemaakt door waarden die geproduceerd werden door de ingediende oplossing te serialiseren bij het uitvoeren van de testcode, en terug te deserialiseren bij het uitvoeren van de evaluatiecode.

Deze evaluatiecode kan geschreven worden in een programmeertaal naar keuze, al moet de programmeertaal wel ondersteund worden door TESTed. Hoewel de exacte implementatie verschilt van programmeertaal tot programmeertaal, komt het neer op het implementeren van een top-level functie[3] met als naam `evaluate`. Deze functie heeft drie parameters:

`expected` De verwachte waarde uit het testplan.

`actual` De waarde die geproduceerd werd door de ingediende oplossing.

`arguments` Een lijst van opties voor de geprogrammeerde evaluatie uit het testplan.

Het resultaat van de beoordeling moet als returnwaarde teruggegeven worden. Hoe deze returnwaarde er exact uitziet, is programmeertaalafhankelijk. TESTed legt geen vereisten op: dit wordt volledig bepaald door de configuratie van de programmeertaal. TESTed vereist een JSON-object met vier velden:

`result` Een `boolean` die aangeeft of de geproduceerde waarde juist is of niet.

`readableExpected` Een optionele `string` met de tekstuele voorstelling van de verwachte waarde. Deze tekstuele voorstelling wordt gebruikt om te tonen aan de gebruiker. Indien niets gegeven wordt, zal TESTed zelf een voorstelling maken op basis van de verwachte waarde uit het testplan.

`readableActual` Een optionele `string` met de tekstuele voorstelling van de geproduceerde waarde. Indien niets gegeven wordt, zal TESTed zelf een voorstelling maken op basis van de geproduceerde waarde.

`messages` Een optionele lijst van berichten, die doorgegeven worden aan Dodona. Deze berichten ondersteunen dezelfde functionaliteit als de berichten van Dodona: het formaat en wie het bericht kan zien kunnen ingesteld worden.

In Java en Python wordt voor de returnwaarde een klasse gedefinieerd. In Haskell gaat het om een `record`. Deze returnwaarden worden vervolgens geserialiseerd tot het vereiste `JSON`-object. Dit is opnieuw niet verplicht, maar in de bestaande programmeertalen gebeurt dit door de `values`-module, die ook het serialiseren van waarden doet.

Een concreet voorbeeld van de evaluatiecode voor een geprogrammeerde evaluatie geschreven in Python wordt gegeven in codefragment 2.8. Deze evaluatiecode is eenvoudig: er wordt gekeken

[3]Deze functie wordt dan vertaald door TESTed. In Java zal dit bijvoorbeeld resulteren in een statische functie.

Codefragment 2.8. Voorbeeld van evaluatiecode in Python voor een geprogrammeerde evaluatie.

```python
from evaluation_utils import EvaluationResult, Message

def evaluate(expected, actual, arguments):
    # Controleer of de gekregen waarde overeenkomt.
    correct = expected == actual
    messages = []
    if not correct:
        messages.append(Message("Hallo, dit is niet juist!"))
    # We geven geen verwachte waarde mee; TESTed neemt de waarde uit het testplan.
    return EvaluationResult(correct, None, actual, messages)
```

of de verwachte waarde overeenkomt met de geproduceerde waarde. Indien dat niet het geval is, wordt een bericht meegegeven. In realiteit zal deze code wel complexer zijn, want eenvoudige gevallen zoals dit kunnen namelijk met de ingebouwde evaluatie, waardoor het niet nodig is om een geprogrammeerde evaluatie te gebruiken. De naam van de evaluatiefunctie, `evaluate`, kan in het testplan ingesteld worden.

2.5.3. Programmeertaalspecifieke evaluatie

In sommige scenario's moeten programmeertaalspecifieke concepten beoordeeld worden. Een mogelijkheid is deze oefeningen niet aanbieden in TESTed, maar in de programmeertaalspecifieke judges. Toch zijn er nog voordelen om ook deze oefeningen in TESTed aan te bieden:

- Het bijkomende werk om een oefening in meerdere programmeertalen te ondersteunen is beperkt.
- Het werk om een nieuwe programmeertaal toe te voegen aan TESTed is kleiner dan een volledig nieuwe judge te implementeren.

Het is desalniettemin het vermelden waard dat het niet zeker is of deze evaluatiemethode (en dit scenario meer algemeen) veel zal voorkomen. Oefeningen die programmeertaalspecifieke aspecten moeten beoordelen zijn, net door hun programmeertaalspecifieke aard, moeilijker aan te bieden in meerdere programmeertalen. Een oefening in de programmeertaal C die bijvoorbeeld beoordeeld wordt op juist gebruik van pointers zal weinig nut hebben in Python.

In gebruik is een programmeertaalspecifieke evaluatie zo goed als identiek aan de geprogrammeerde evaluatie, met dat verschil dat de evaluatiefunctie slechts één argument meekrijgt: de door de ingediende oplossing geproduceerde waarde. Een verwachte waarde is niet voorzien, omdat we verwachten dat in veel gevallen de waarde niet ondersteund zal worden door het testplan. Ook moet de evaluatiecode geschreven worden voor elke programmeertaal waarin de oefening aangeboden wordt. Een concreet voorbeeld, opnieuw in Python, is codefragment 2.9. In dit voorbeeld wordt gekeken of de waarde een integer is of niet.

Codefragment 2.9. Voorbeeld van evaluatiecode in Python voor een programmeertaalspecifieke evaluatie.

```python
from evaluation_utils import EvaluationResult, Message

def evaluate(actual):
    correct = isinstance(actual, int)
    messages = []
    if not correct:
        messages.append(Message("Dit is geen int!"))
    return EvaluationResult(correct, str(5), str(actual), messages)
```

2.6. Performantie

Zoals eerder vermeld (paragraaf 2.4.2), wordt de testcode voor elke context afzonderlijk uitgevoerd. Dat de contexten strikt onafhankelijk van elkaar uitgevoerd worden, werd reeds in het begin als een doel vooropgesteld. Dit geeft wel enkele uitdagingen op het vlak van performantie. Het belang van performante judges in Dodona is niet te verwaarlozen, in die zin dat Dodona een interactief platform is, waar studenten verwachten dat de feedback op hun ingediende oplossing onmiddellijk beschikbaar is. Deze paragraaf beschrijft de evolutie van de implementatie van TESTed vanuit het perspectief van de performantie.

2.6.1. Jupyter-kernels

Het eerste prototype van TESTed gebruikte Jupyter-kernels voor het uitvoeren van de testcode. Jupyter-kernels zijn de achterliggende technologie van Jupyter Notebooks [30]. De werking van een Jupyter-kernel kan als volgt samengevat worden: een Jupyter-kernel is een lokaal proces, dat code kan uitvoeren en de resultaten van dat uitvoeren teruggeeft. Zo kan men naar de Python-kernel de expressie `5 + 9` sturen, waarop het antwoord `14` zal zijn. Een andere manier om een Jupyter-kernel te bekijken, is als een programmeertaalonafhankelijk protocol bovenop een REPL (een *read-eval-print loop*). Deze keuze voor Jupyter-kernels als uitvoering was gebaseerd op volgende overwegingen:

- Hergebruik van bestaande kernels. Hierdoor is het niet nodig om voor elke programmeertaal veel tijd te besteden aan de implementatie of de configuratie: aangezien het protocol voor Jupyter-kernels programmeertaalonafhankelijk is, kunnen alle bestaande kernels gebruikt worden.
- De functionaliteit aangeboden door een Jupyter-kernel is de functionaliteit die nodig is voor TESTed: het uitvoeren van fragmenten code en het resultaat van die uitvoering verzamelen.
- Eerder werk [31], dat gebruik maakt van Jupyter-kernels voor een gelijkaardig doel, rapporteert geen problemen met het gebruik van Jupyter-kernels.

Omdat contexten per definitie onafhankelijk van elkaar zijn, moet de gebruikte Jupyter-kernel gestopt en opnieuw gestart worden tussen elke context. Dit brengt een onaanvaardbare performantiekost met zich mee, daar de meeste oefeningen niet computationeel intensief zijn. Het beoordelen van eenvoudige oefeningen, zoals de Lotto-oefening, duurde al snel meerdere minuten.

Enkele ideeën om de performantiekost te verkleinen waren:

- Het gebruiken van een *pool* kernels (een verzameling kernels die klaar staan voor gebruik). De werkwijze is als volgt:

  1. Bij de start van het beoordelen van een ingediende oplossing worden meerdere kernels gestart.
  2. Bij elke context wordt een kernel uit de pool gehaald om de testcode uit te voeren.
  3. Na het uitvoeren wordt de kernel op een andere draad (*thread*) opnieuw opgestart en terug aan de pool toegevoegd.

  Het idee achter deze werkwijze is dat door op een andere draad de kernels te herstarten, er altijd een kernel klaarstaat om de testcode uit te voeren, en er dus niet gewacht moet worden op het opnieuw opstarten van die kernels. In de praktijk bleek echter dat zelfs met een twintigtal kernels in de pool, het uitvoeren van de testcode van eenvoudige oefeningen dermate snel gaat in vergelijking met het opnieuw opstarten van de kernels, dat de kernels nooit op tijd herstart zijn.

- De kernels niet opnieuw opstarten, maar de interne toestand opnieuw instellen. In bepaalde kernels, zoals de Python-kernel, is dit mogelijk. De Python-kernel (IPython) heeft bijvoorbeeld een magisch commando `%reset`, dat de toestand van de kernel opnieuw instelt. Het probleem is dat er maar weinig kernels zijn die een gelijkaardig commando hebben.

Op dit punt is besloten dat de overhead van de Jupyter-kernels niet naar tevredenheid kon opgelost worden. Bovendien is een bijkomend nadeel van het gebruik van Jupyter-kernels ondervonden: de kernels voor andere programmeertalen dan Python zijn van gevarieerde kwaliteit. Zo schrijven de Java-kernel en de Julia-kernel bij het opstarten altijd een boodschap naar de standaarduitvoerstroom, wat in bepaalde gevallen problemen gaf bij het verzamelen van de resultaten van de uitvoer van de testcode.

2.6.2. Sjablonen

We kozen ervoor om verder te gaan met een systeem van sjablonen (zie ook paragraaf 2.4.1). Hierbij wordt eerst testcode gegenereerd, die vervolgens uitgevoerd wordt via de commandoregel.

Voordelen ten opzichte van de Jupyter-kernels zijn:

- Het is sneller om twee onafhankelijke programma's uit te voeren op de commandolijn dan een Jupyter-kernel tweemaal te starten.
- Het laat meer vrijheid in hoe de resultaten (gedrag en neveneffecten) van het uitvoeren van de testcode verzameld worden.
- Het implementeren en configureren van een programmeertaal met de basisfunctionaliteit in TESTed is minder werk dan het implementeren van een nieuwe Jupyter-kernel. Optionele functionaliteit, zoals linting, kan ervoor zorgen dat meer tijd en werk nodig is bij het toevoegen van een programmeertaal. Het systeem met de sjablonen volgt echter zoveel mogelijk het ontwerpprincipe dat optionele functionaliteit geen bijkomend werk mag vragen voor programmeertalen die er geen gebruik van maken. Programmeertalen die bijvoorbeeld geen linting ondersteunen, zullen geen bijkomende configuratie nodig hebben.

- Er is doorgaans[4] geen verschil tussen hoe TESTed de ingediende oplossing uitvoert en hoe de student diezelfde oplossing uitvoert. Bij de Jupyter-kernels is dat soms wel het geval: bij de R-kernel zitten subtiele verschillen tussen het uitvoeren in de kernel en het uitvoeren op de commandoregel.
- Het is herbruikbaar in die zin dat het genereren van testcode in een programmeertaal op basis van de programmeertaalonafhankelijke specificatie in het testplan sowieso nodig is.
- De gegenereerde testcode bestaat uit normale codebestanden (bv. `.py`- of `.java`-bestanden), wat het toevoegen van een programmeertaal aan TESTed eenvoudiger te debuggen maakt: alle gegenereerde testcode is in een bestand beschikbaar voor inspectie.

Elke medaille heeft ook een keerzijde:

- Het gebruik van sjablonen zorgt ervoor dat het uitvoeren van de testcode minder dynamisch is. Daar de testcode eerst gegenereerd moet worden, moet vóór het uitvoeren bepaald worden welke testcode zal uitgevoerd worden.
- Er moet meer zelf geïmplementeerd worden: het ecosysteem van Jupyter is groot. Er bestaan meer dan honderden kernels[5], goed voor ondersteuning van meer dan tachtig programmeertalen. Niet elke kernel is onmiddellijk bruikbaar, maar er kan verder gewerkt worden op wat bestaat.
- Het uitvoeren op de commandoregel is bij veel programmeertalen trager indien er geen *cold start* moet gebeuren tussen de contexten. Dit is bijvoorbeeld het geval bij de Python- en Java-kernel, die een speciaal commando hebben dat de toestand van de kernel opnieuw kan instellen. In deze gevallen is de initiële opstartkost van de kernel groter dan het starten van de interpreter (in Python) of de virtuele machine (Java), maar deze kost wordt geamortiseerd over de verschillende contexten, waardoor de opstartkost per context kleiner is.

Bij een eerste iteratie van het systeem met sjablonen bestond enkel de **contextcompilatie** (paragraaf 2.4.1). Dit heeft een grote performantiekost bij alle programmeertalen, maar in het bijzonder bij talen zoals Java. Daar moest voor elke context eerst een compilatie plaatsvinden, waarna de gecompileerde testcode werd uitgevoerd. Bij een testplan met bijvoorbeeld 50 contexten vertaalt dit zich in 50 keer de `javac`-compiler uitvoeren en ook 50 keer het uitvoeren van de code zelf met `java`.

In een tweede iteratie werd een **partiële compilatie** geïmplementeerd. Het idee hier is dat er veel testcode is die hetzelfde blijft voor elke context (denk aan de ingediende oplossing en hulpbestanden van TESTed). In de partiële compilatie wordt de gemeenschappelijke testcode eerst gecompileerd (bij programmeertalen die dat ondersteunen). Bij het beoordelen van een context wordt dan enkel de testcode specifiek voor de te beoordelen context gecompileerd.

In een derde iteratie werd de partiële compilatie uitgebreid naar **batchcompilatie**, zoals beschreven in paragraaf 2.4.1.

[4] Soms is een wijziging nodig om de code uitvoerbaar te maken binnen TESTed. Zo zal in C de `main`-functie hernoemd worden en in Haskell bestaat de mogelijkheid om de moduledeclaratie (`module Solution where`) toe te voegen aan de ingediende oplossing.

[5] Zie deze pagina voor een actuele lijst van kernels: https://github.com/jupyter/jupyter/wiki/Jupyter-kernels.

2.6.3. Parallelle uitvoering van contexten

Een ander aspect is de ondersteuning voor parallelle uitvoering van de contexten. Zoals al enkele malen aangehaald zijn de contexten van het testplan volledig onafhankelijk van elkaar, waardoor ze zich ertoe lenen om parallel uitgevoerd te worden.

Een eerste opmerking hierbij is dat de tijdswinst op Dodona waarschijnlijk kleiner zal zijn dan de tijdswinst geobserveerd in de tijdsmetingen (tabel 2.3). De reden hiervoor is dat Dodona een beperkt aantal *workers* heeft die beoordelingen uitvoeren. Doordat er meerdere gebruikers tegelijk oplossingen indienen, zal er niet altijd ruimte zijn voor parallellisatie binnen de judge.

Een tweede opmerking is dat momenteel er verschillende beperkingen of opmerkingen zijn in de implementatie:

- De parallelle uitvoering heeft enkel betrekking op het uitvoeren van de contexten, niet op de beoordeling ervan. De beoordeling gebeurt nog steeds sequentieel.
- De parallellisatie gebeurt enkel binnen tabbladen. De tabbladen zelf worden sequentieel uitgevoerd. Dit betekent dat de contexten van tabblad 1 parallel worden uitgevoerd, en pas aan tabblad 2 begonnen wordt als alle contexten van tabblad 1 klaar zijn.
- De volgorde waarin het resultaat van de beoordeling van een context naar Dodona gestuurd wordt blijft dezelfde volgorde als in het testplan. Dit impliceert dat het mogelijk is dat, bijvoorbeeld bij het overschrijden van de tijdslimiet, de uitvoer van uitgevoerde contexten niet getoond wordt. Stel dat contexten 1, 2, 4 en 5 klaar zijn wanneer de tijdslimiet overschreden wordt. Omdat TESTed aan het wachten was op context 3, zullen contexten 4 en 5 ook als niet uitgevoerd beschouwd worden.

Er is echter geen reden om aan te nemen dat deze beperkingen niet opgelost kunnen worden. We hebben deze opmerkingen dan ook opgenomen in paragraaf 5.3.3.

2.6.4. Snellere geprogrammeerde evaluatie

Een laatste performantieverbetering werd behaald op het vlak van de geprogrammeerde evaluatie (zie paragraaf 5.3.2 voor verdere ideeën voor verbeteringen). Zoals beschreven in paragraaf 2.5.2, wordt normaliter voor elke geprogrammeerde evaluatie de evaluatiecode opnieuw gecompileerd en uitgevoerd in een nieuw subproces. Voor evaluatiecode die geschreven is in Python worden deze stappen overgeslagen. Er is speciale ondersteuning in TESTed ingebouwd om geprogrammeerde evaluaties waarvan de evaluatiecode in Python geschreven is, rechtstreeks in TESTed zelf uit te voeren.

Dit conflicteert niet met het onafhankelijk uitvoeren van de code: bij een geprogrammeerde evaluatie wordt tijdens de evaluatie geen code van de student uitgevoerd. We kunnen dus voldoende zeker zijn dat de uitgevoerde code *in orde* is. Het blijft bovendien mogelijk een geprogrammeerde evaluatie op de tragere manier uit te voeren, indien daar nood aan is.

Tabel 2.3. Tijdsmetingen voor de oefeningen Lotto en Echo, voor de programmeertalen Python en Java, in contextcompilatie, partiële compilatie en batchcompilatie. Bij Lotto is deze laatste uitgevoerd met en zonder optimalisatie voor geprogrammeerde evaluaties.

		Python (s)		Java (s)	
Oefening	Compilatiemodus	1 thread	4 threads	1 thread	4 threads
Lotto	Context	13	9	51	38
	Partieel	9	7	46	25
	Batch	9	6	12	10
	Batch+Eval	5	**3**	9	**6**
Echo	Context	6	3	48	28
	Partieel	5	3	46	26
	Batch	5	**3**	8	**5**

2.6.5. Tijdsmetingen

Tabel 2.3 toont enkele tijdsmetingen van de verschillende implementaties van het systeem met sjablonen, voor de programmeertalen Python en Java. In deze tabel is de tijd gemeten in seconden om een (juiste) ingediende oplossing voor twee oefeningen te beoordelen:

- De Lotto-oefening, zoals beschreven in paragraaf 1.5. Deze oefening heeft een geprogrammeerde evaluatie en bestaat uit 50 contexten, die elk één testgeval bevatten.
- Een eenvoudigere oefening: Echo. Bij deze oefening bestaat de opgave er uit de invoer van de standaardinvoerstroom te lezen en te printen naar de standaarduitvoerstroom. Deze oefening bevat enkel een generieke evaluatie en bestaat ook uit 50 contexten, met telkens één testgeval per context.

Elke beoordeling is uitgevoerd met contextcompilatie, partiële compilatie en batchcompilatie. De Lotto-oefening is ook gemeten met de optimalisatie voor geprogrammeerde evaluatie (daar de evaluatiecode voor deze oefening geschreven is in Python). Dit is aangeduid met de naam „Batch+Eval" in de kolom „Compilatiemodus". Deze variant is niet uitgevoerd op de Echo-oefening, daar die oefening geen geprogrammeerde evaluatie heeft. Elke tijdsmeting is ook uitgevoerd met en zonder parallelle uitvoering van de contexten. Deze tijdsmetingen zijn uitgevoerd op een standaardcomputer (Windows 10, Intel i7-8550U, Python 3.8.1, 64-bit), niet op de Dodona-server en niet in een Docker-container. Die laatste twee factoren kunnen ervoor zorgen dat uitvoeringstijden op Dodona sterk verschillen van deze metingen.

Zoals verwacht levert batchcompilatie het meeste tijdswinst op bij Java: daar heeft de compilatiestap ook een veel groter aandeel in de uitvoeringstijd. Bij Python is de compilatie veel sneller en ook veel minder belangrijk (en zelfs optioneel, de stap wordt enkel uitgevoerd om bepaalde syntaxisfouten vroeg op te vangen). Een gevolg van de grotere winst bij programmeertalen met „zware" compilatiestap is dat het verschil tussen die programmeertalen en niet-gecompileerde programmeertalen significant kleiner geworden is. Waar voor bijvoorbeeld de Lotto-oefening de beoordeling van een oplossing in Java 38 seconden trager was dan een oplossing in Python beoordelen, is dit verschil gedaald naar 3 seconden.

2.7. Bijkomende taken

Sommige judges doen meer dan enkel het oordelen over de juistheid van de ingediende oplossing. Zo heeft onder andere de Python-judge ondersteuning voor *linting*. Dit betekent dat de code van de ingediende oplossing geanalyseerd (maar niet uitgevoerd) wordt om zo allerlei mogelijke problemen op te sporen, zoals stijlfouten, mogelijke bugs, verdachte constructies, enz.

TESTed voorziet hier ondersteuning voor middels een optionele functie in de configuratie van een programmeertaal (zie hoofdstuk 4 voor de details over het configureren van een taal). Deze functie krijgt als argumenten onder andere de configuratie, het testplan en het pad naar de code van de ingediende oplossing mee. Als resultaat geeft de functie een lijst van berichten en annotaties die naar Dodona gestuurd worden. Ter illustratie dat deze aanpak werkt, is linting geïmplementeerd voor Python. Dit een flexibele oplossing: de implementatie in Python gebruikt een bestand voor het configureren van de linter, en het is mogelijk om de naam van een eigen configuratiebestand (uit de oefeningenrepository) mee te geven via de programmeertaalspecifieke configuratie van TESTed, zonder dat daar speciale ondersteuning voor nodig is binnen TESTed. Tot slot nog opmerken dat het formaat van de annotaties wordt voorgeschreven door Dodona, maar dit is voldoende generiek voor praktisch alle scenario's. Het formaat ondersteunt onder andere:

- Het aangeven van een gedetailleerde plaats in de ingediende oplossing, zoals de regel en kolom. Het aanduiden van meerdere opeenvolgende regels of kolommen is ook mogelijk.
- De boodschap is een vrij tekstveld.
- De ernst van de boodschap, met ondersteuning voor een veelvoorkomende indeling: informatiebericht, waarschuwing en foutbericht.

Een mogelijke uitbreiding is bijvoorbeeld het toevoegen van identificatienummers aan de berichten. Bij veel linters heeft elke boodschap een unieke code. Aan de hand van die code zou Dodona de uitleg over de boodschap kunnen tonen (maar dit vereist wel dat Dodona de uitleg voor specifieke linters heeft). Een alternatief zou zijn dat TESTed deze zelf toevoegt. De gemakkelijkste oplossing is echter de annotatie uitbreiden met ondersteuning voor een URL: bij veel linters kan met de identificatiecode van het bericht een URL geconstrueerd worden waarop meer informatie staat. Deze URL zou dan aan de studenten kunnen getoond worden.

2.8. Robuustheid

Een belangrijk aspect bij *educational software testing* is de feedback als het verkeerd loopt. De feedback bij een verkeerde oplossing is in veel gevallen zelfs belangrijker dan de feedback bij een juiste oplossing: het is namelijk de bedoeling dat als studenten een verkeerde oplossing indienen, dat de feedback ze terug op weg kan helpen om hun oplossing te verbeteren. De feedback die Dodona toont is afkomstig van de judges: het is de taak van TESTed om kwalitatief hoogstaande feedback te voorzien. Met kwalitatief hoogstaand wordt bedoeld dat de feedback nuttige informatie bevat, maar ook geen verkeerde of misleidende informatie.

2.8.1. In de praktijk

In het vak *Scriptingtalen* van de bachelor Informatica krijgen studenten elke week een reeks oefeningen om op te lossen (op Dodona). Een van die oefeningen is de Python-versie van de *ISBN*-oefening. De opgave van deze oefening is opgenomen als onderdeel van paragraaf 3.7. Om een idee te krijgen van hoe TESTed in de praktijk werkt, is besloten om tijdens een van de weken de *ISBN*-oefening aan te vullen door een oefening met dezelfde opgave, maar die beoordeeld wordt door TESTed in plaats van de Python-judge.

De algemene bevindingen die uit de praktijktest naar boven kwamen waren:

- TESTed heeft geen ondersteuning voor de Python Tutor. Dit is al vermeld, en wordt verder besproken in paragraaf 5.5.
- De evaluatie met TESTed duurt ongeveer dubbel zo lang als met de Python-judge. Aan de ene kant zou de optimalisatie van de geprogrammeerde evaluatie (zie paragraaf 2.6) dit moeten verbeteren (maar wel in dit specifieke geval, moest de evaluatiecode in een andere programmeertaal dan Python geschreven zijn, zou het geen verschil maken). Aan de andere kant ligt een tragere uitvoer in vergelijking met de Python-judge binnen de verwachtingen: TESTed voert elke context uit in een afzonderlijk subproces. De Python-judge doet dit niet. Qua uitvoer worden wel contexten gebruikt, maar de interne werking van de Python-judge kan het best vergeleken worden met een oefeningen waar alle testen in dezelfde context opgenomen zijn.
- TESTed heeft geen verwerking van stacktraces. Deze functionaliteit is opgenomen als mogelijke uitbreiding als onderdeel van paragraaf 5.5.
- TESTed heeft geen linting. Naar aanleiding van de praktijktest is hier ondersteuning voor toegevoegd, zie paragraaf 2.7.
- TESTed heeft geen ondersteuning voor vertalingen op het vlak van natuurlijke talen. Deze functie wordt besproken als uitbreiding in paragraaf 5.2.6.

Verder heeft professor Dawyndt veel fouten en scenario's uitgeprobeerd. Deze worden hierna in detail besproken.

2.8.2. Soorten fouten

TESTed moet robuust zijn tegen allerlei vormen van fouten in de ingediende oplossingen. Hieronder volgt een lijst (van categorieën) van fouten waarvoor TESTed nuttige feedback geeft. TESTed is zo opgebouwd dat er altijd iets van feedback komt, ook in onvoorziene omstandigheden, maar voor de soorten fouten op onderstaande lijst is expliciet gecontroleerd wat de kwaliteit van de feedback is. Waar nuttig zijn deze gevallen omgezet naar een unit test voor TESTed.

Compilatiefouten De uitvoer van de compiler wordt altijd getoond aan de studenten, dus op dat vlak wordt juist afgehandeld. Er bestaat ook de mogelijkheid om de compilatie-uitvoer te verwerken en bijvoorbeeld om te zetten naar annotaties, die dan in de code worden getoond. Hierbij wordt vooral gedacht aan foutboodschappen als „syntaxisfout op regel 5, kolom 2". De stacktraces bij deze uitvoer bevatten wel referenties naar code die TESTed gegenereerd heeft. Deze verwijzingen naar interne code wegfilteren is opgenomen als uitbreiding als onderdeel van paragraaf 5.5. Bij een batchcompilatie wordt de uitvoer in een nieuw tabblad getoond, terwijl bij contextcompilatie de uitvoer bij de relevant context staat.

Uitvoeringsfouten Hier gaat het om crashes tijdens de uitvoering, zoals delen door nul. Ook hier wordt nog geen verdere verwerking van de foutboodschap gedaan (ook onderdeel van paragraaf 5.5).

Tijdslimieten TESTed heeft ondersteuning voor tijdslimieten in de judge. Dit laat toe om meer uitvoer te tonen dan als de tijdslimiet aan Dodona wordt overgelaten. Momenteel werkt de implementatie ervan als volgt: TESTed houdt bij hoe lang de beoordeling al duurt, en gebruikt de resterende tijd als een limiet op de uitvoering van een context. De eerste context heeft dan als limiet de volledige toegestane tijd (minus een percentage dat voorbehouden wordt voor TESTed zelf). De laatste context zal de kleinste limiet krijgen (de tijd die nog overschiet). Scenario's waar dit voorkomt zijn bijvoorbeeld oplossingen die te traag werken, maar ook oplossingen met bijvoorbeeld oneindige lussen. Als uitbreiding (opgenomen in paragraaf 5.5) wordt beschreven dat er ook een tijdslimiet per context mogelijk zou kunnen zijn (zodat de eerste context niet de volledige beschikbare tijd krijgt, maar slechts een deel ervan). De hoofdreden om zelf de tijdslimiet te implementeren in TESTed is om zo de overige, niet-uitgevoerde testen ook te kunnen tonen. Momenteel is dat met een boodschap die uitlegt dat ze niet zijn uitgevoerd, maar op termijn is het de bedoeling in Dodona een bijkomende status toe te voegen.

Te grote uitvoer Bij te grote uitvoer zal TESTed deze limiteren. Hierbij gaat het om scenario's zoals een oneindige lus die blijft schrijven naar `stdout`. De totale grootte van de uitvoer van alle uitvoerkanalen samen wordt gelimiteerd op 8 MiB (80 % van de limiet opgelegd door Dodona). Bij het overschrijden van deze limiet zal TESTed de evaluatie fout rekenen en de juiste status doorgeven aan Dodona. Deze limiet werkt goed op `stdout` en `stderr`, maar minder goed op andere vormen van uitvoer. Het inkorten van grote verzamelingen (bijvoorbeeld een lijst van duizend elementen) is opgenomen als uitbreiding in paragraaf 5.5. Dat laatste is ook nuttig zelfs al wordt de uitvoerlimiet niet overschreden: het is overzichtelijker.

Te veel uitvoer Hierbij wordt gedacht aan zaken zoals uitvoer op `stdout` of `stderr` terwijl die niet verwacht wordt. Standaard wordt een oefening als fout beschouwd indien er te veel uitvoer is, maar de auteur van de oefening kan kiezen om het teveel aan uitvoer te negeren (bijvoorbeeld als de studenten debugberichten schrijven naar stdout is dit niet in elke oefening een probleem).

Vroegtijdig stoppen van uitvoering Hier gaat het om code van de studenten die bijvoorbeeld `exit(-5)` bevat. Dit zal standaard fout gerekend worden door TESTed. De exitcode is echter een uitvoerkanaal zoals elk ander, waar de verwachte waarde ingesteld kan worden. Is het dus de bedoeling om exitcode `-5` te krijgen, dan zal het testgeval juist gerekend worden.

2.9. Implementatiekeuzes

In deze paragraaf lichten we enkele keuzes toe die we gemaakt hebben bij de implementatie van wat we in dit hoofdstuk besproken hebben.

TESTed is geschreven in Python. De hoofdreden daarvoor is pragmatisch: zoals besproken in paragraaf 2.6.1, gebruikte de eerste versie van TESTed Jupyter-kernels om code uit te voeren. Het raamwerk rond de Jupyter-kernels is geschreven in Python, dus was het een voor de hand liggende keuze om Python te gebruiken voor TESTed. Als TESTed later weg van de Jupyter-kernels geëvolueerd is, is niet van programmeertaal veranderd. Dit omdat er geen goede reden was om de bestaande code te herschrijven in een andere programmeertaal.

Codefragment 2.10. Een voorbeeld van hoe het testplan met `pydantic` gedefinieerd wordt. Hier wordt een functieoproep voorgesteld in Python-code.

```python
@dataclass
class FunctionCall:
    type: FunctionType
    name: str
    namespace: Optional[str] = None
    arguments: List[Union['Expression', NamedArgument]] =
     ↪  field(default_factory=list)
```

Het grootste nadeel aan Python is dat het niet de snelste programmeertaal is op het gebied van uitvoeringstijd. Dit is voor TESTed minder relevant: het gros van de beoordelingstijd wordt besteed aan het uitvoeren van de ingediende oplossing, niet aan TESTed zelf.

Binnen TESTed worden twee dependencies intensief gebruikt:

1. `Mako` voor de sjablonen.
2. `Pydantic` voor het definiëren van het testplan en het serialisatieformaat.

In TESTed is niet onmiddellijk gekozen voor Mako als sjabloonsysteem. Een eerdere versie gebruikte Jinja2. Later is overgeschakeld op Mako, omdat er bij Jinja2 problemen waren met witruimte in de gegenereerde code. Deze witruimte is belangrijk bij het genereren van bijvoorbeeld Python-code. Deze twee packages zijn ongeveer de populairste packages in het Python-ecosysteem om sjablonen te gebruiken, buiten de ingebouwde sjablonen van Django. Die laatste zijn niet overwogen, omdat het niet duidelijk is of de sjablonen van Django ook gebruikt kunnen worden zonder het hele framework te moeten gebruiken.

Na de keuze voor JSON als testplan en serialisatieformaat (paragrafen 2.3.1 en 2.3.2) is er beslist om de implementatie in Python de gezaghebbende definitie voor het formaat te maken. Oorspronkelijk zijn het testplan en het serialisatieformaat opgesteld met `dataclasses` uit Python. Dit bleek niet voldoende: er waren betere validaties nodig en het bleek handig om het formaat van het testplan en het serialisatieformaat te kunnen exporteren als JSON Schema.

Hiervoor gebruiken we nu `pydantic`[6]. Dit package laat ons toe om nog steeds `dataclasses` te gebruiken, maar toch validaties te kunnen toevoegen en de structuur te kunnen exporteren naar JSON Schema. Codefragment 2.10 toont hoe de definitie van een functieoproep er in Python-code uitziet. Deze klasse wordt letterlijk vertaald naar JSON, zoals duidelijk is in bijvoorbeeld het testplan in paragraaf 3.4.3.

Het JSON Schema voor respectievelijk het testplan en het serialisatieformaat kunnen zo automatisch verkregen worden:

```
> python -m tested.testplan
<JSON Schema voor testplan>
> python -m tested.serialisation
<JSON Schema voor serialisatieformaat>
```

[6] Zie https://pydantic-docs.helpmanual.io/

3. Configuratie van een oefening

In dit hoofdstuk behandelen we het configureren van oefeningen voor TESTed. We bespreken verschillende eenvoudige oefeningen, met als doel dat deze oefeningen als voorbeeld kunnen dienen bij het samenstellen van complexere oefeningen of oefeningen die meerdere functionaliteiten tegelijk gebruiken. We bespreken oefeningen:

- die invoer (`stdin`) lezen en uitvoer (`stdout`) produceren (*I/O*-oefeningen),
- waarbij functies beoordeeld worden door TESTed,
- waarbij een programmeertaalspecifieke evaluator gebruikt wordt,
- waarbij een geprogrammeerde evaluator gebruikt wordt,
- waarbij commandoargumenten gelezen worden,
- waarbij statements (assignments) gebruikt worden en
- waarbij objectgerichte zaken gebruikt worden.

Bij elke oefening bespreken we de opgave, hoe we willen dat de beoordeling van een oplossing er zal uitzien en hoe we dat vertalen naar een testplan.

3.1. Oefeningen in het Dodona-platform

TESTed zelf legt geen structuur of formaat vast voor de oefeningen, buiten het formaat van het testplan. De locatie van de relevante bestanden wordt meegegeven bij het uitvoeren van TESTed. TESTed kan daardoor ook los van Dodona gebruikt worden.

De configuratie in de manuele uitvoeringsmodus van TESTed gaat er wel vanuit dat de mappenstructuur van de oefening de structuur van Dodona volgt. Voor de volledigheid volgt hieronder een mappenstructuur met de belangrijkste elementen van de oefening:

```
oefening                      # Map van de oefening in Dodona
├── config.json               # Configuratiebestand van Dodona
├── description
│   ├── description.en.html   # Beschrijving in het Engels
│   ├── description.nl.html   # Beschrijving in het Nederlands
├── evaluation                # Evaluatiecode
│   └── plan.json             # Het testplan voor TESTed
├── preparation
│   └── generator.py          # Code die het testplan genereert
└── solution                  # Voorbeeldoplossingen voor de oefeningen
    ├── solution.java
    └── solution.py
```

Voor meer details en bijkomende informatie (want het voorbeeld hierboven is slechts een basis), raden we aan om de relevante handleiding uit de Dodona-documentatie te lezen[1].

3.2. Lotto

Als eerste oefening beschouwen we de voorbeeldoefening Lotto uit paragraaf 1.5, die we uitgebreid zullen bespreken. De opgave is al opgenomen in die paragraaf, terwijl de voorbeeldoplossingen in codefragment 1.1 staan.

3.2.1. Structuur van de beoordeling

Na het lezen van de opgave is het duidelijk dat de oefening bestaat uit het implementeren van een functie. In abstracto zullen we, om te beoordelen of deze functie correct geïmplementeerd is, de functie een aantal keren oproepen met verschillende argumenten en dan telkens de geproduceerde returnwaarde vergelijken met een verwachte waarde.

Zoals vermeld in paragraaf 2.3.1 bestaat het testplan uit een hiërarchie van elementen, beginnend met een aantal tabbladen, die elk een aantal contexten hebben, die op hun beurt bestaan uit testgevallen, die tot slot bestaan uit testen. In dit geval lijkt het logisch om één tabblad te gebruiken: per slot van rekening beoordelen we één functie.

Alle functieoproepen zijn onafhankelijk van elkaar, wat suggereert dat elke functieoproep in een aparte context dient te gebeuren. Een context bestaat uit een optioneel testgeval voor de `main`-functie en een reeks normale testgevallen. Bij de Lotto-oefening hebben we geen `main`-functie, dus zullen we enkel normale testgevallen hebben. Er zal één testgeval per context zijn, want in elke context willen we de functie éénmaal oproepen.

Vertaald naar pseudocode willen we dus dat onze beoordeling de volgende vorm aanneemt:

```
assert loterij() == '2 - 17 - 22 - 27 - 35 - 40'
assert loterij(8) == "5 - 13 - 15 - 31 - 34 - 36 - 39 - 40"
assert loterij(4, 38) == "16 - 20 - 35 - 37"
# ... enzovoort
```

Hierbij is het geheel van `asserts` dus ons tabblad, terwijl we elke `assert` in een aparte context en dus ook apart testgeval onderbrengen.

[1]Beschikbaar op https://dodona-edu.github.io/en/references/exercise-directory-structure/

3.2.2. Evaluatie

Iets dat we in de vorige paragraaf genegeerd hebben, is hoe we het vergelijken met een verwachte waarde exact gaan doen. Lottonummers worden namelijk willekeurig getrokken: de pseudocode van hierboven zou dus slechts heel zelden tot een juiste beoordeling leiden. We lossen dit op door een geprogrammeerde evaluatie te gebruiken. Dit is een functie die TESTed oproept om de door de ingediende oplossing geproduceerde waarde te vergelijken met een verwachte waarde. Hierbij is het mogelijk om in het testplan argumenten mee te geven aan deze evaluatiefunctie, iets dat we hier ook doen. De argumenten die we meegeven aan de `loterij`-functie geven we ook mee aan de evaluatiefunctie. Conceptueel kunnen we dat ook vertalen naar pseudocode:

```
assert geprogrammeerde_evaluatie(loterij(), [])
assert geprogrammeerde_evaluatie(loterij(8), [8])
assert geprogrammeerde_evaluatie(loterij(4, 38), [4, 38])
# ... enzovoort
```

We zullen de geprogrammeerde evaluatie in Python doen: dit is de aanbevolen programmeertaal voor geprogrammeerde evaluaties in TESTed, met als eenvoudige reden dat ze het snelst is. Hieronder is een fragment van de evaluatiecode: dit is de functie die door TESTed zal opgeroepen worden.

```
def evaluate(expected, actual, arguments):
    count, maximum = arguments
    valid, message = valid_lottery_numbers(actual, count, maximum)
    messages = [Message(message)] if message else []
    if valid:
        expected = actual
    return EvaluationResult(valid, expected, actual, messages)
```

Wat doet deze functie nu juist?

1. Een evaluatiefunctie van een geprogrammeerde evaluatie kan argumenten meekrijgen. In ons geval geven we de argumenten van de `loterij`-functie mee als argument aan de evaluatiecode.
2. De functie `valid_lottery_numbers` wordt opgeroepen. We hebben deze functie niet opgenomen in het codefragment hierboven omdat het een lange functie is, maar deze functie controleert in feite of de geproduceerde waarde voldoet aan de vereisten (klopt het aantal getallen, is de lijst gesorteerd, enzovoort).
3. Indien de geproduceerde waarde geldig is, dan gebruiken we die ook als verwachte waarde. Dit voorkomt dat de oplossing juist is, maar Dodona toch een verschil toont tussen de geproduceerde en verwachte waarde.

4. Tot slot construeren we een `EvaluationResult` als returnwaarde. Dit object verwacht vier waarden: het resultaat van de evaluatie (juist of fout), optioneel de verwachte waarde indien deze overschreven moet worden, optioneel de geproduceerde waarde indien deze overschreven moet worden en een optionele lijst van feedbackberichten (meer details in paragraaf 2.5.2).

3.2.3. Het testplan

Nu we weten welke structuur we willen en hoe we gaan beoordelen, kunnen we een testplan opstellen in JSON. Hieronder volgt ter illustratie één context uit het testplan. In werkelijkheid (en ook bij deze oefening) wordt het testplan niet met de hand geschreven; het wordt gegenereerd door een Python-script. Dit script bevindt zich in de `preparation`-map van de oefening.

We merken op dat we wel een verwachte waarde opnemen in het testplan, ook al gaat het om willekeurige returnwaarden. Bij deze oefening wordt deze waarde niet gebruikt bij de geprogrammeerde evaluatie. Toch is het nuttig ze op te nemen, omdat deze verwachte waarde in de feedback getoond zal worden als de geproduceerde waarde verkeerd is of als de tijdslimiet bijvoorbeeld overschreden wordt. Het is mogelijk om in de geprogrammeerde evaluatie geavanceerde zaken te doen, zoals indien de geproduceerde waarde juist is, maar niet gesorteerd, de geproduceerde waarde te sorteren en als verwachte waarde te gebruiken.

```
{
  "testcases": [
    {
      "input": {
        "type": "function",
        "name": "loterij",
        "arguments": [
          {
            "type": "integer",
            "data": 18
          },
          {
            "type": "integer",
            "data": 172
          }
        ]
      },
      "output": {
        "result": {
          "value": {
            "type": "text",
            "data": "7 - 37 - 48 - 54 - 70 - 78 - 81 - 90 - 102 - 103 - 113 - 119 -
              ↪  120 - 137 - 140 - 154 - 157 - 159"
          },
          "evaluator": {
            "type": "programmed",
            "language": "python",
            "function": {
```

```
              "file": "./evaluator.py"
            },
            "arguments": [
              {
                "type": "integer",
                "data": 18
              },
              {
                "type": "integer",
                "data": 172
              }
            ]
          }
        }
      }
    }
  ]
},
```

3.3. Echo

Een volgende oefening die we bekijken, is een eenvoudige *invoer-uitvoer*-oefening. Deze oefening bestaat uit een invoer lezen van `stdin` en deze invoer schrijven naar `stdout`. Bij deze oefeningen zijn voor een aantal programmeertalen de volledige testcode die door TESTed gegenereerd wordt en de voorbeeldoplossingen opgenomen in appendix B.

3.3.1. Opgave

De volledige opgave voor deze oefening volgt hieronder:

> Je krijgt één regel invoer en je moet eenvoudigweg deze invoer herhalen. Je moet dus een programma schrijven dat als echo fungeert.
>
> #### Invoer
>
> Eén regel tekst zonder verdere restricties.
>
> #### Uitvoer
>
> Dezelfde tekst als je hebt ingelezen bij de invoer.

Voorbeeld

Invoer

```
42
```

Uitvoer

```
42
```

Voorbeeld

Invoer

```
ECHO
```

Uitvoer

```
ECHO
```

3.3.2. Structuur van de beoordeling en evaluatie

Voor de structuur van de beoordeling geldt grotendeels hetzelfde als bij paragraaf 3.2.1: we hebben een tabblad, met daarin een aantal contexten. In elke context gebruiken we een andere waarde als `stdin`. In tegenstelling tot de Lotto-oefening hebben we nu wel een `main`-functie. De context zal dus bestaan uit het testgeval voor de `main`-functie en geen normale testgevallen hebben. Het testgeval voor de `main`-functie wordt in het testplan apart aangeduid met `context_testcase`.

Qua evaluatie is deze oefening eenvoudig: we kunnen de ingebouwde evaluatie van TESTed gebruiken. De geproduceerde tekstuele waarde moet vergeleken worden met de verwachte waarde (en moet exact overeenkomen).

3.3.3. Testplan

Als voorbeeldtestplan nemen we een testplan met twee contexten. Bij het gebruik van deze oefening zal het testplan vijftig contexten bevatten. Het wordt ook niet met de hand geschreven: een Python-script genereert het.

```
{
  "tabs": [
    {
      "name": "Feedback",
      "contexts": [
        {
          "context_testcase": {
            "input": {
              "main_call": true,
              "stdin": {
                "type": "text",
```

```
                "data": "input-1"
              }
            },
            "output": {
              "stdout": {
                "type": "text",
                "data": "input-1"
              }
            }
          }
        },
        {
          "context_testcase": {
            "input": {
              "main_call": true,
              "stdin": {
                "type": "text",
                "data": "input-2"
              }
            },
            "output": {
              "stdout": {
                "type": "text",
                "data": "input-2"
              }
            }
          }
        }
      ]
    }
  ]
}
```

3.4. Echofunctie

Een variant van de vorige oefening is de oefening *Echofunctie*, waarbij een `echo`-functie geïmplementeerd dient te worden. Ook bij deze oefening zijn voor een aantal programmeertalen de gegenereerde testcode en voorbeeldoplossingen opgenomen in appendix C.

3.4.1. Opgave

De volledige opgave luidt als volgt:

Schrijf een functie `echo` die steeds teruggeeft wat als argument aan de functie wordt doorgegeven.

Het argument van de functie zal steeds een `string` zijn.

Voorbeeld

```
>>> echo("5");
"5"
>>> echo("ok");
"ok"
```

Het is nuttig om stil te staan bij waarom de opgave vermeldt dat de invoer altijd een `string` zal zijn. Dit is om de oefening in zoveel mogelijk programmeertalen te kunnen aanbieden. In bepaalde programmeertalen, zoals C, is het moeilijk om een functie te schrijven die een argument van een willekeurig type aanvaardt (we noemen dit heterogene argumenten binnen TESTed). Om die reden verhindert TESTed dat oefeningen waar dit vereist is opgelost kunnen worden in die programmeertalen. Toch is het mogelijk om oefeningen op te stellen waar heterogene argumenten gebruikt worden. Deze oefeningen zullen dan in minder programmeertalen beoordeeld kunnen worden. Stel dat we in het testplan bijvoorbeeld de `echo`-functie oproepen met een getal in plaats van een `string`. TESTed zal dan automatisch detecteren dat deze oefening enkel opgelost kan worden in programmeertalen die heterogene functieargumenten ondersteunen. Als we dan toch zouden proberen om een oplossing geschreven in C te laten beoordelen, zal TESTed een foutmelding tonen (zie paragraaf 2.3.4 voor meer details over dit mechanisme).

3.4.2. Structuur van de beoordeling en evaluatie

De structuur is volledig analoog aan de *Echo*-oefening. We zullen een reeks contexten hebben, waarbij we in elke context de te implementeren functie oproepen met andere invoer en het resultaat controleren. Ook de evaluatie is analoog: we gebruiken de ingebouwde evaluatie van TESTed. Het verschil zit in welke kanalen de invoer en uitvoer zich bevinden: bij deze oefening gebruiken we functieargumenten in plaats van `stdin` en het resultaat is een returnwaarde in plaats van `stdout`.

3.4.3. Testplan

Ter illustratie tonen we hier een testplan met één context, die twee testgevallen bevat. Dit komt met opzet niet overeen met wat we hierboven bij de structuur van de beoordeling besproken hebben. De reden hiervoor is dat we ook een illustratie van meerdere testgevallen willen in appendix C. In het testplan dat gebruikt zou worden bij het beoordelen van oplossingen van studenten zal elk testgeval wel in een eigen context geplaatst worden (en zullen er opnieuw meer contexten zijn).

```
{
  "tabs": [
    {
      "name": "Feedback",
```

```
      "contexts": [
        {
          "testcases": [
            {
              "input": {
                "type": "function",
                "name": "echo",
                "arguments": [
                  {
                    "type": "text",
                    "data": "input-1"
                  }
                ]
              },
              "output": {
                "result": {
                  "value": {
                    "type": "text",
                    "data": "input-1"
                  }
                }
              }
            },
            {
              "input": {
                "type": "function",
                "name": "echo",
                "arguments": [
                  {
                    "type": "text",
                    "data": "input-2"
                  }
                ]
              },
              "output": {
                "result": {
                  "value": {
                    "type": "text",
                    "data": "input-2"
                  }
                }
              }
            }
          ]
        }
      ]
    }
  ]
}
```

3.5. ZeroDivisionError

Een interessante oefening is de oefening *ZeroDivisionError* uit het boek *De Programmeursleerling* van Pieter Spronck [32]. Een interactieve versie van deze oefening is beschikbaar in Dodona.[2] In deze oefening moet een programma geschreven worden dat bij het uitvoeren een exception gooit. In Python gaat het om een `ZeroDivisionError`. We nemen de opgave hier niet op, omdat ze weinig bijdraagt aan ons doel, het uitleggen hoe oefeningen voor TESTed geschreven moeten worden.

3.5.1. Structuur van de beoordeling

Daar het programma een exception moet gooien als het uitgevoerd wordt, lijkt het aangewezen dat we een `main`-functie zullen gebruiken. Ook uniek aan deze oefening is dat we de oplossing één keer moeten uitvoeren, dus zullen we ook één context hebben. De structuur zal een tabblad met een context met een testgeval voor de `main`-functie zijn.

3.5.2. Evaluatie

Het feit dat specifiek een `ZeroDivisionError` moet gegooid worden in Python, zorgt ervoor dat we hier een programmeertaalspecifieke evaluatie zullen moeten gebruiken. In Java gaat het bijvoorbeeld om een `ArithmeticException`, terwijl delen door nul is Haskell zal zorgen voor een `DivideByZero`.

We bekijken hier eens niet de evaluatiecode in Python, maar die in Java. De evaluatiecode is ook beschikbaar in Haskell.

```java
public class Evaluator {
    public static EvaluationResult evaluate(Object actual) {
        if (actual instanceof ArithmeticException) {
            return EvaluationResult.builder(true)
                    .withReadableExpected(actual.toString())
                    .withReadableActual(actual.toString())
                    .build();
        } else {
            return EvaluationResult.builder(false)
                    .withReadableExpected("ArithmeticException")
                    .withReadableActual(actual == null ? "" : actual.toString())
                    .withMessage(new EvaluationResult.Message("Expected
                     ↪  ArithmeticException, got something else."))
                    .build();
        }
    }

}
```

[2] Op https://dodona.ugent.be/nl/courses/293/series/2535/activities/270198713/

De evaluatiecode is redelijk rechtdoorzee: indien het een exception van het juiste type is, wordt de oplossing als juist beschouwd, terwijl alle andere exceptions (of `null` als er geen exception is) fout gerekend worden.

3.5.3. Testplan

Ook het testplan is een vrij eenvoudige vertaling van de structuur die we hiervoor hebben bedacht. Uniek hier is dat dit testplan ook het volledige testplan is zoals het gebruikt wordt bij de oefening in Dodona.

```
{
  "tabs": [
    {
      "name": "Feedback",
      "contexts": [
        {
          "context_testcase": {
            "description": "Uitvoeren code",
            "input": {
              "main_call": true
            },
            "output": {
              "exception": {
                "evaluator": {
                  "type": "specific",
                  "evaluators": {
                    "python": {"file": "evaluator.py"},
                    "java": {"file": "Evaluator.java"},
                    "haskell": {"file": "Evaluator.hs"},
                    "runhaskell": {"file": "Evaluator.hs"}
                  }
                }
              }
            }
          }
        }
      ]
    }
  ]
}
```

3.6. Som

Deze oefening is andermaal afkomstig uit het boek *De Programmeursleerling*[3]. De oefening is interessant om te illustreren hoe commandoargumenten werken en hoe de exitcode werkt. De opgave bestaat eruit om een reeks getallen in te lezen uit de commandoargumenten en de som ervan uit te schrijven op `stdout`. Als een van de commandoargumenten geen geldig getal is, dan moet een foutboodschap naar `stderr` geschreven worden en moet het programma eindigen met exitcode `1`. Bijvoorbeeld:

```
> python ./som
0
> echo $?
0
> python ./som 1 2 3 4 5 6 7 8 9 10
55
> python ./som 1 -2 3 -4 5 -6 7 -8 9 -10
-5
> python ./som spam eggs bacon
som: ongeldige argumenten
> echo $?
1
```

Ook hier nemen we de opgave niet op door haar lengte en geringe nut.

3.6.1. Structuur van de beoordeling en evaluatie

Qua structuur en evaluatie lijkt deze oefening sterk op de *Echo*-oefening, met dat verschil dat we hier commandoargumenten hebben. Om de ingediende oplossing te beoordelen, zullen we de oplossing meerdere malen uitvoeren met telkens andere commandoargumenten. We plaatsen elk stel argumenten in een eigen context. Bij deze opgave is er geen keuze: per context is er maximaal één stel commandoargumenten, want de `main`-functie wordt hoogstens eenmaal opgeroepen per context. Het verwachte resultaat op `stdout` en de foutboodschappen zijn opnieuw deterministisch te berekenen op basis van de commandoargumenten, dus kunnen we de ingebouwde evaluatie van TESTed gebruiken.

3.6.2. Testplan

In het testplan zijn drie contexten opgenomen: één waarbij getallen gegeven zijn, één waarbij geen argumenten gegeven worden en één waar geen geldige getallen als argumenten gegeven worden. Dit opnieuw om het testplan kort te houden; bij gebruik in Dodona zal het testplan meer contexten bevatten.

[3]Ook interactief beschikbaar op Dodona: https://dodona.ugent.be/nl/courses/293/series/2556/exercises/1653208777/

```
{
  "tabs": [
    {
      "name": "Feedback",
      "contexts": [
        {
          "context_testcase": {
            "input": {
              "arguments": ["-1", "-23", "72", "84", "-38", "-61", "49", "45"],
              "main_call": true
            },
            "output": {
              "stdout": {
                "type": "text",
                "data": "127"
              }
            }
          }
        },
        {
          "context_testcase": {
            "input": {
              "arguments": [],
              "main_call": true
            },
            "output": {
              "stdout": {
                "type": "text",
                "data": "0"
              }
            }
          }
        },
        {
          "context_testcase": {
            "input": {
              "arguments": ["spam", "eggs", "bacon"],
              "main_call": true
            },
            "output": {
              "stderr": {
                "type": "text",
                "data": "som: ongeldige argumenten"
              },
              "exit_code": {
                "value": 1
              }
            }
          }
```

```
        }
      ]
    }
  ]
}
```

3.7. ISBN

Een volgende oefeningen die we behandelen is de *ISBN*-oefening. Deze oefening is al vermeld in paragraaf 2.8.1, waar we besproken hebben dat we deze oefening al hebben laten oplossen door studenten. Vanuit het oogpunt van het schrijven van oefeningen voor TESTed is deze oefening interessant doordat het een „ingewikkeldere" oefening is, waarbij ook statements (*in casu* assignments) gebruikt worden.

3.7.1. Opgave

Hieronder volgt (een fragment van) de opgave:

Opgave

- Schrijf een functie `is_isbn` waaraan een string *c* (`str`) moet doorgegeven worden. De functie moet een Booleaanse waarde (`bool`) teruggeven, die aangeeft of *c* een geldige ISBN-code is. De functie heeft ook nog een optionele tweede parameter `isbn13` waaraan een Booleaanse waarde (`bool`) kan doorgegeven worden die aangeeft of het om een ISBN-10 code (`False`) of om een ISBN-13 code (`True`, standaardwaarde) moet gaan.
- Schrijf een functie `are_isbn` waaraan een lijst (`list`) met $n \in \mathbb{N}$ codes moet doorgegeven worden. De functie moet voor alle codes uit de gegeven lijst aangegeven of ze geldige ISBN-codes voorstellen. De functie heeft ook nog een tweede optionele parameter `isbn13` waaraan een Booleaanse waarde (`bool`) kan doorgegeven worden die aangeeft of het om ISBN-10 codes (`False`) of om ISBN-13 codes (`True`) moet gaan.

  Als er niet expliciet een waarde wordt doorgegeven aan de parameter `isbn13`, dan moet het type van elke code uit de lijst bepaald worden op basis van de lengte van die code. Als een code geen string (`str`) is, dan wordt die *a priori* als ongeldig bestempeld. Voor codes van lengte 13 moet getest worden of het geldige ISBN-13 codes zijn, en voor codes van lengte 10 of het geldige ISBN-10 codes zijn. Codes met afwijkende lengtes (geen 10 en geen 13) worden ook *a priori* als ongeldige ISBN-codes bestempeld.

  De functie moet een nieuwe lijst (`list`) met *n* Booleaanse waarden (`bool`) teruggeven, die aangeven of de code op de corresponderende positie in de gegeven lijst een geldige ISBN-code is.

Voorbeeld

```
>>> is_isbn('9789027439642', False)
False
>>> is_isbn('9789027439642', True)
True
>>> is_isbn('9789027439642')
True
>>> is_isbn('080442957X')
False
>>> is_isbn('080442957X', False)
True

>>> codes = ['0012345678', '0012345679', '9971502100', '080442957X', 5,
 ↪  True, 'The Practice of Computing Using Python', '9789027439642',
 ↪  '5486948320146']
>>> are_isbn(codes)
[False, True, True, True, False, False, False, True, False]
>>> are_isbn(codes, True)
[False, False, False, False, False, False, False, True, False]
>>> are_isbn(codes, False)
[False, True, True, True, False, False, False, False, False]
```

3.7.2. Structuur van de beoordeling en evaluatie

Uit de opgave volgt dat er twee functies geïmplementeerd zullen moeten worden. Het is gebruikelijk bij Dodona om elk van deze functies in een apart tabblad te beoordelen.

De contexten in het eerste tabblad voor de functie `is_isbn` zijn niet speciaal. We roepen de functie `is_isbn` per context één keer op met andere argumenten.

In het tweede tabblad voor de functie `are_isbn` ligt de situatie iets anders. De eerste parameter van deze functie is een lijst van potentiële ISBN's. Om de overzichtelijkheid te verbeteren willen we, zoals in het voorbeeld in de opgave, deze lijst eerst toekennen aan een variabele (een assignment) en dan de variabele gebruiken als argument voor de functie.

In TESTed heeft een testgeval altijd één statement als invoer. In de situatie hierboven hebben we twee statements: eerst de assignment en vervolgens de functieoproep. We zullen dus per context twee testgevallen hebben.

Op het vlak van evaluatie is deze oefening eenvoudig: door deterministische resultaten kunnen we de ingebouwde evaluatie van TESTed gebruiken.

3.7.3. Testplan

Als testplan tonen we hier een testplan met een context uit het tweede tabblad (dus met de assignment). Om het testplan niet te lang te maken hebben we geen context opgenomen uit het eerste tabblad, vermits deze contexten niets nieuws doen.

Het statement bestaat uit een naam en een expressie. De expressie is in dit geval een waarde: een lijst van elementen. Dit is de eerste keer dat we een collectie gebruiken als waarde, dus loont het de moeite om daar even stil bij te staan. Het gebruikt illustreert dat de types van de elementen in een collectie niet hetzelfde gegevenstype moeten hebben: deze lijst bevat tekst, getallen en booleans. Ook dit is niet mogelijk in alle programmeertalen: TESTed detecteert dit als een heterogene collectie. Dit zal ervoor zorgen dat we deze oefening niet in alle programmeertalen kunnen oplossen, naast het feit dat de programmeertaal ook collecties moet ondersteunen, wat bijvoorbeeld (nog) niet het geval is in C.

```
{
  "tabs": [
    {
      "name": "are_isbn",
      "contexts": [
        {
          "testcases": [
            {
              "input": {
                "type": "sequence",
                "variable": "codes01",
                "expression": {
                  "data": [
                    {"data": "0012345678", "type": "text"},
                    {"data": "0012345679", "type": "text"},
                    {"data": "9971502100", "type": "text"},
                    {"data": "080442957X", "type": "text"},
                    {"data": 5, "type": "integer"},
                    {"data": true, "type": "boolean"},
                    {"data": "The Practice of Computing Using Python", "type":
                     ↪  "text"},
                    {"data": "9789027439642", "type": "text"},
                    {"data": "5486948320146", "type": "text"}
                  ],
                  "type": "sequence"
                }
              }
            },
            {
              "input": {
                "type": "function",
                "name": "are_isbn",
                "arguments": ["codes01"]
              },
              "output": {
                "result": {
                  "value": {
                    "data": [
                      {"data": false, "type": "boolean"},
                      {"data": true, "type": "boolean"},
                      {"data": true, "type": "boolean"},
```

```
                    {"data": true, "type": "boolean"},
                    {"data": false, "type": "boolean"},
                    {"data": false, "type": "boolean"},
                    {"data": false, "type": "boolean"},
                    {"data": true, "type": "boolean"},
                    {"data": false, "type": "boolean"}
                ],
                "type": "sequence"
            }
        }
    }
  }
 ]
 }
]
}
```

3.8. EqualChecker

Als laatste oefeningen bekijken we een oefeningen die gebruik maakt van een klasse, om te illustreren hoe objectgerichte oefeningen ook mogelijk zijn. Zoals duidelijk zal zijn uit de opgave is dit een fictieve oefening.

3.8.1. Opgave

Hieronder volgt de opgave:

Opgave

Implementeer een klasse `EqualChecker`, met een constructor die één getal neemt. De klasse moet één methode `check` hebben, die een `boolean` teruggeeft als het argument van de methode hetzelfde is als het argument dat we aan de constructor gegeven hebben.

Voorbeeld

```
>>> instance = EqualChecker(5)
>>> instance.check(8)
False
```

```
>>> instance.check(5)
True
```

3.8.2. Structuur van de beoordeling en evaluatie

Bij het evalueren van deze oefeningen zullen we opnieuw meerdere statements nodig hebben: een statement om een instantie van de klasse `EqualCheck` te initialiseren, gevolgd door een statement die een methode van het aangemaakte object oproept. Hier hebben we de keuze gemaakt dat we per context drie testgevallen zullen hebben: één om het object te maken en twee die een methode oproepen.

De evaluatie gebeurt met de ingebouwde evaluatie van TESTed.

3.8.3. Testplan

Als testplan tonen we hier een testplan met één context, die zoals gezegd drie testgevallen heeft. Nieuwe elementen in dit testplan zijn:

- In het eerste testgeval gebruiken we een assignment met een constructor om het object aan te maken. In het testplan wordt een constructor voorgesteld als een speciale functie, waarbij de naam van de functie de naam van de klasse is. Ook het type van de variabele die we maken bij de assignment is speciaal: het gaat hier namelijk niet om een ingebouwd type van TESTed, maar om de klasse `EqualChecker`. Aangezien dit een apart testgeval is, gebeurt de evaluatie van dit testgeval zoals voor elke functie. Stel dat de constructor uitvoer genereert op `stdout`, dan zal dit testgeval fout gerekend worden.
- In de volgende twee testgevallen gebruiken we een `namespace`-functie. Als `namespace` geven we de naam van de variabele op die we hiervoor gemaakt hebben: `instance`. Voor het overige gebeurt dit analoog aan de functies die we tot nu toe gezien hebben.

```
{
  "tabs": [
    {
      "name": "Feedback",
      "contexts": [
        {
          "testcases": [
            {
              "input": {
                "variable": "instance",
                "type": {
                  "type": "custom",
                  "data": "EqualChecker"
                },
                "expression": {
                  "type": "constructor",
                  "name": "EqualChecker",
```

```
                "arguments": [
                  {
                    "type": "integer",
                    "data": 5
                  }
                ]
              }
            }
          },
          {
            "input": {
              "type": "namespace",
              "namespace": "instance",
              "name": "check",
              "arguments": [
                {
                  "type": "integer",
                  "data": 25
                }
              ]
            },
            "output": {
              "result": {
                "value": {
                  "type": "boolean",
                  "data": false
                }
              }
            }
          },
          {
            "input": {
              "type": "namespace",
              "namespace": "instance",
              "name": "check",
              "arguments": [
                {
                  "type": "integer",
                  "data": 5
                }
              ]
            },
            "output": {
              "result": {
                "value": {
                  "type": "boolean",
                  "data": true
                }
              }
```

```
                }
              }
            ]
          }
        ]
      }
    ]
}
```

4. Configuratie van een programmeertaal

In dit hoofdstuk wordt in detail uitgelegd hoe een nieuwe programmeertaal aan TESTed kan toegevoegd worden. We doen dat door te beschrijven hoe de programmeertaal C aan TESTed toegevoegd is. Dit hoofdstuk sluit qua vorm en stijl dan ook dichter aan bij een handleiding. Enkele nuttige links en verwijzingen hierbij zijn:

- Bestaande configuraties: `https://github.com/dodona-edu/universal-judge/tree/new-master/judge/src/tested/languages`
- Appendices B en C bevatten specificaties van volledige oefeningen: de opgave, het testplan en de gegenereerde code. Dat laatste kan nuttig zijn om ook het concrete resultaat te zien van wat de sjablonen genereren.

In dit hoofdstuk worden regelmatig codebestanden getoond, verspreid over meerdere codefragmenten (bijvoorbeeld eerst *n* regels, daarna wat tekst en dan pas de rest van de regels). Om dit duidelijk te maken toont de regelnummering aan de linkerkant de regelnummers in het originele bestand.

4.1. TESTed lokaal uitvoeren

Tijdens het configureren van een programmeertaal is het nuttig om TESTed lokaal uit te voeren, zonder daarvoor het volledige Dodona-platform te moeten uitvoeren. Buiten de *dependencies* voor de bestaande programmeertalen is TESTed een Python-package, dat op de normale manier uitgevoerd kan worden.

4.1.1. De broncode

Na het klonen van de repository van TESTed beschikken we over volgende mappenstructuur:

```
universal-judge
├── docker/      # Een Docker-image om TESTed uit te voeren.
├── exercise/    # De map met oefeningen die dienen als voorbeeld en voor de tests.
├── judge/       # TESTed zelf.
│   ├── src/     # Broncode van TESTed.
│   ├── tests/   # Testcode van TESTed.
|   └── workdir/ # De uitvoer van TESTed bij manuele uitvoer.
├── thesis/      # De thesistekst, dus deze tekst.
├── config.json  # Configuratiebestand voor gebruik in Dodona.
└── run          # Wordt gebruikt door Dodona om TESTed uit te voeren.
```

Merk op dat dit de toestand is op het moment van het schrijven van deze tekst. Het is te voorzien dat in een later stadium alles behalve de mappen `judge` en `exercise` verhuizen naar een andere repository. In dit hoofdstuk interesseren we ons enkel in die mappen, dus we voorzien geen grote problemen.

4.1.2. Dependencies

De dependencies van TESTed zelf zijn opgelijst in een `requirements.txt`-bestand, zoals gebruikelijk is bij Python-projecten. Vereisten voor het uitvoeren van tests staan in `requirements-test.txt`. TESTed gebruikt Python 3.8 of later. Het installeren van deze vereisten gebeurt op de gebruikelijke manier:

```
> pip install -r requirements.txt
```

Voor de programmeertalen die momenteel reeds geconfigureerd zijn, zijn volgende dependencies nodig:

Python Indien de linter gebruikt wordt, is `pylint` een dependency. Daarnaast moet `python` beschikbaar zijn in het `PATH`. Door optimalisaties is het momenteel aan te raden om dezelfde Python-versie te gebruiken voor TESTed als voor de Python-oefeningen.

Java TESTed vereist Java 11, maar heeft verder geen dependencies. De commando's `javac` en `java` moeten beschikbaar zijn in het `PATH`.

Haskell Voor Haskell is GHC 8.6 of later nodig. Daarnaast is `aeson` nodig. Beiden moeten globaal beschikbaar zijn in het `PATH`.

Merk op dat de dependencies voor de programmeertalen optioneel zijn. Om bijvoorbeeld enkel Python-oplossingen te beoordelen zijn geen andere dependencies nodig.

Voor de programmeertaal C gaan we gebruikmaken van GCC, waarbij versie 8.1 of later nodig is.

TESTed werkt op elk besturingssysteem dat ondersteund wordt door Python. Sommige dependencies, zoals GCC, vragen wel meer moeite om te installeren op Windows.[1]

4.1.3. Uitvoeren

We gaan er voor de rest van het hoofdstuk van uit dat commando's uitgevoerd worden in de map `judge/src`.

Er zijn twee manieren om TESTed uit te voeren. Ten eerste is er de „gewone" manier; dit is ook hoe Dodona TESTed uitvoert. Bij het uitvoeren op deze manier zal TESTed een configuratie lezen van `stdin` en zal het resultaat van de beoordeling in Dodona-formaat uitgeschreven worden naar `stdout`.

```
> python -m tested
```

[1] Gebruikers op Windows kunnen MinGW of MSYS2 proberen.

Bij het configureren van een programmeertaal of het werken aan TESTed is het echter nuttiger om meer uitvoer te zien en is het vervelend om telkens een configuratie te lezen vanop `stdin`. Daarom is er een tweede manier:

```
> python -m tested.manual
```

Deze uitvoer verschilt op een aantal vlakken van de gewone uitvoering:

1. Er wordt geen configuratie gelezen van `stdin`. De configuratie is gedefinieerd in de code zelf en gebruikt een van de oefeningen die in de map `exercise` zitten.
2. Er worden, naast de resultaten van de beoordeling, logs uitgeschreven naar `stdout` die aangeven wat TESTed doet. Als er bijvoorbeeld een fout optreedt tijdens het compileren zullen deze logs nuttig zijn: zo wordt uitgeschreven welk commando TESTed exact uitvoert voor de compilatie en ook in welke map dat gebeurt.
3. De configuratie is zo opgesteld dat de werkmap van de judge de map `workdir` zal zijn. Dit laat toe om de gegenereerde code te inspecteren.

4.2. Globaal stappenplan voor het configureren van een programmeertaal

Het configureren van een programmeertaal in TESTed bestaat uit drie grote onderdelen:

1. Het configuratiebestand, met enkele opties voor de programmeertaal.
2. De configuratieklasse, met de meer dynamische opties, zoals het compilatiecommando.
3. De sjablonen, die gebruikt worden om code te genereren.

TESTed voorziet een hulpmiddel om de bestanden op de juiste plaats te genereren. Op basis van enkele vragen worden *stubs* gegenereerd voor het configuratiebestand, de configuratieklasse en de sjablonen. Dit hulpmiddel kan als volgt uitgevoerd worden:

```
> python -m tested.generation
```

Merk op dat dit enkel bestanden genereert. De stappen in paragraaf 4.3.6 voor het registreren van de nieuwe programmeertaal in TESTed zijn nog steeds nodig.

We overlopen nu elk onderdeel in functie van de programmeertaal C. We gaan er telkens vanuit dat bovenstaande hulpmiddel niet gebruikt is en dat de bestanden dus nog gemaakt moeten worden. Is bovenstaande hulpmiddel wel gebruikt, dan kunnen de instructies voor het maken van bestanden genegeerd worden.

4.3. De programmeertaal C

Voor we beginnen aan de configuratie, overlopen we kort welke functionaliteit we willen ondersteunen: welke functionaliteit uit C kunnen we aanbieden in TESTed en welke functionaliteit uit TESTed kunnen we implementeren in C? Uiteraard willen we zoveel mogelijk ondersteunen, maar vooral op het vlak van gegevenstypes zijn er momenteel beperkingen.

Welke basistypes gaan we niet ondersteunen?

`sequence` Arrays zijn een speciaal geval in C: statische arrays kunnen bijvoorbeeld niet als returnwaarde dienen, en ook als functieargument zijn ze niet ideaal. Dynamische arrays nemen de vorm aan van een pointer en een grootte. TESTed heeft momenteel geen ondersteuning voor datatypes die als twee waarden geïmplementeerd moeten worden, dus worden arrays momenteel niet ondersteund.

`set` C heeft geen ingebouwde verzamelingen.

`map` C heeft geen ingebouwde map of dict. Er zijn wel structs, maar daarvan is het niet mogelijk om de velden at runtime op te vragen, waardoor we ze niet kunnen serialiseren.

Welke geavanceerde types gaan we niet ondersteunen?

`big_int` C heeft geen ingebouwd type voor getallen van arbitraire grootte.

`fixed_precision` C heeft geen ingebouwd type voor kommagetallen met willekeurige precisie.

Andere datastructuren Het gaat hier om datastructuren zoals `array` en `list` (om dezelfde redenen als hierboven). Ook `tuple` wordt niet ondersteund, omdat het niet bestaat in C.

4.3.1. Locatie van de code

De eerste stap in het configureren van een programmeertaal is het aanmaken van een map waarin we de code voor de programmeertaal zullen zetten. Deze map moet de naam van de programmeertaal krijgen en op de juiste plaats binnen TESTed aanwezig zijn. Maak een nieuwe map `judge/src/tested/languages/c`. Na het aanmaken van de map moet de mappenstructuur er zo uitzien:

```
universal-judge
├── judge/
│   ├── src/
│   │   └── tested/
│   │       ├── languages/
│   │       │   ├── c/          ← nieuwe map
│   │       │   ├── haskell/
│   │       │   ├── java/
│   │       │   ├── python/
│   │       │   ├── config.py
│   │       │   ...
│   │       ...
│   ...
...
```

4.3.2. Configuratiebestand

Het configuratiebestand is een JSON-bestand met enkele eigenschappen van de programmeertaal. Dit configuratiebestand maakt het implementeren van de configuratieklasse een stuk eenvoudiger, omdat de implementatie van die klasse daardoor veel minder lang zal zijn. Maak eerst het configuratiebestand aan: `judge/src/tested/languages/c/config.json`.

Merk op dat het configuratiebestand slechts een hulpmiddel is: indien gewenst kunnen al deze opties ook ingesteld worden door de juiste methodes te implementeren in de configuratieklasse, maar we verwachten dat dit in veel gevallen niet nodig zal zijn.

Algemene opties

`general.dependencies` Dit zijn bestanden die beschikbaar zullen zijn tijdens het compileren en tijdens het uitvoeren van de beoordeling. Dit betekent dat deze dependencies gebruikt kunnen worden in de testcode voor de contexten en de evaluatiecode voor de geprogrammeerde en programmeertaalspecifieke code. In het geval van C is dit de `values`-module, waarvan we de implementatie later bespreken. Deze dependencies zijn bedoeld om gebruikt te worden in de code gegenereerd door de sjablonen, niet in de ingediende oplossing (hoewel dat momenteel technisch mogelijk is).

`general.selector` Dit geeft aan of de programmeertaal gebruikmaakt van een selector tijdens het uitvoeren van code die gecompileerd is in batchcompilatie. Voor de meeste talen met compilatie zal dit `true` zijn. Zo ook bij C (paragraaf 2.4.1 legt dit mechanisme in meer detail uit).

`extensions.file` Geeft de voornaamste bestandsextensie aan van de bestanden. Met voornaamste bedoelen we de extensie van de bestanden die gegenereerd worden. Bijvoorbeeld in C bestaan zowel `.h` en `.c`, maar de gegenereerde code gebruikt `.c`.

`extensions.templates` - wordt gebruikt om aan te geven welke extensies gebruikt worden voor de sjablonen. Standaard is dit de bestandsextensie van hierboven en `.mako`. Het is vaak niet nodig om dit op te geven.

```
"general": {
  "dependencies": [
    "values.h",
    "values.c",
    "evaluation_result.h",
    "evaluation_result.c"
  ],
  "selector": true
},
"extensions": {
  "file": "c",
  "templates": ["c", "mako"]
},
```

Codestijl

Programmeertaalelementen zoals functies en namespaces worden omgezet in functie van de codestijl die gebruikelijk is in de programmeertaal:

```
"naming_conventions": {
  "namespace": "snake_case",
  "function": "snake_case"
},
```

De mogelijke waarden zijn:

`snake_case` Tussen elk woord staat een underscore: `dit_is_een_voorbeeld`.

`camel_case` Elk woord, buiten het eerste, start met een hoofdletter: `ditIsEenVoorbeeld`. Deze variant wordt ook wel *lowerCamelCase* genoemd.

`pascal_case` Elk woord, ook het eerste, start met een hoofdletter: `DitIsEenVoorbeeld`. Deze variant wordt ook wel *UpperCamelCase* genoemd.

Standaard wordt `snake_case` gebruikt, dus bij C is het niet strikt nodig om deze optie in de configuratie op te nemen.

Functionaliteit

De laatste twee blokken in de configuratie geven aan welke constructies en gegevenstypes de programmeertaal ondersteunt. We hebben reeds besproken welke functionaliteit we willen ondersteunen en welke niet. We beginnen met de taalconstructies vast te leggen:

```
"constructs": {
  "objects": false,
  "exceptions": false,
  "function_calls": true,
  "assignments": true,
  "heterogeneous_collections": false,
  "heterogeneous_arguments": false,
  "evaluation": false,
  "named_arguments": false,
  "default_parameters": false
},
```

Hier kan voor elke taalconstructie opgegeven worden of ze ondersteund wordt of niet (met een `boolean`). Standaard wordt geen enkele taalconstructie ondersteund. Dit zorgt ervoor dat alle ondersteunde constructies expliciet in het configuratiebestand staan en dat nieuwe taalconstructies toegevoegd kunnen worden zonder dat bestaande configuraties van programmeertalen aangepast moeten worden.

De mogelijke taalconstructies zijn deze uit de enum `tested.features.Construct`. Hieronder volgt een lijst van elke taalconstructie en een korte beschrijving:

`objects` Objectgeoriënteerde zaken zoals klassen.

`exceptions` Exceptions en uitzonderingen.

`function_calls` Functieoproepen. Merk op dat constructors in het testplan een speciale soort functie zijn, maar deze hangen af van de taalconstructie `objects`.

`assignments` Het toekennen van een waarde aan een variabele. Een „assignment" moet ruim geïnterpreteerd worden als ondersteuning voor iets dat neerkomt op een assigment. Zo kent Haskell bijvoorbeeld geen assignments: `x = 5` definieert technisch gezien een functie met een constante returnwaarde `5`. Dit moet ook onder `assignments` gerekend worden.

`heterogeneous_collections` Hiermee bedoelen we collecties met elementen met verschillende gegevenstypes. Dit is bijvoorbeeld geen probleem in Python (`[5, 52.23]`), gaat al iets moeilijker in Java (`List<Object> = List.of(1, 52.23)`), maar zal niet lukken in Haskell.

`heterogeneous_arguments` Hiermee bedoelen we functieoproepen waarbij dezelfde functie meerdere keren wordt opgeroepen met argumenten met verschillende datatypes (bijvoorbeeld eerst `check(True)` daarna `check('hallo')`). Dit zal lukken in Python en Java, maar niet in Haskell en C.

`evaluation` Of een geprogrammeerde evaluatie mogelijk is in deze programmeertaal. Dit is technisch gezien geen taalconstructie, maar dezelfde infrastructuur wordt gebruikt om dit te controleren.

`named_arguments` Of benoemde argumenten ondersteund worden. Dit betekent dat de argumenten voor een functie niet enkel positioneel, maar ook op naam meegegeven kunnen worden.

`default_parameters` Of de programmeertaal standaardwaarden voor parameters ondersteunt, wat betekent dat ze kunnen weggelaten worden.

Dan moeten we nu de ondersteuning voor de gegevenstypes vastleggen:

```
30 "datatypes": {
31   "integer": "supported",
32   "rational": "supported",
33   "char": "supported",
34   "text": "supported",
35   "boolean": "supported",
36   "sequence": "unsupported",
37   "set": "unsupported",
38   "map": "unsupported",
39   "nothing": "supported",
40   "int8": "supported",
41   "uint8": "supported",
42   "int16": "supported",
43   "uint16": "supported",
44   "int32": "supported",
45   "uint32": "supported",
46   "int64": "supported",
47   "uint64": "supported",
```

```
  "bigint": "reduced",
  "single_precision": "supported",
  "double_precision": "supported",
  "double_extended": "unsupported",
  "fixed_precision": "unsupported",
  "array": "unsupported",
  "list": "unsupported",
  "tuple": "unsupported"
}
```

Zoals uitgelegd in paragraaf 2.3.2 zijn er twee soorten gegevenstypes in TESTed: de basistypes en de geavanceerde types. De basistypes zijn abstracte types voor concepten (zoals een sequentie of een geheel getal), terwijl de geavanceerde types concreter zijn (zoals een geheel getal van 8 bits). Een gegevenstype kan drie niveaus van ondersteuning hebben:

`supported` volledige ondersteuning

`reduced` wordt ondersteund, maar wordt herleid tot een basistype (bijvoorbeeld een `list` wordt geïnterpreteerd als een `sequence`)

`unsupported` geen ondersteuning, dit is de standaardwaarde

Een opmerking hierbij is dat de status `reduced` voor de basistypes equivalent is aan `supported`: een basistype reduceren tot een basistype blijft hetzelfde type.

Het is de bedoeling dat de meeste programmeertalen voor het merendeel van de datatypes ten minste `reduced` hebben. Toch is gekozen om `unsupported` als standaardwaarde te nemen; dit zorgt ervoor dat de ondersteunde datatypes expliciet uitgeschreven zijn. Dit laat ook toe om datatypes toe te voegen aan TESTed zonder bestaande configuraties van programmeertalen te moeten aanpassen. Ter illustratie vermelden we hier voor C alle datatypes, ook de niet-ondersteunde.

4.3.3. Configuratieklasse

De configuratieklasse is de schakel tussen de generieke aspecten van TESTed en het programmeertaalafhankelijke gedrag. Omdat TESTed in Python geschreven is, moet deze klasse ook in Python geïmplementeerd worden.

Maak een nieuw Python-bestand `judge/src/tested/languages/c/config.py` aan. Hierin moet een klasse komen die van `Language` overerft:

```
class C(Language):
```

In de rest van deze paragraaf overlopen we de verschillende methodes die geïmplementeerd moeten worden in deze klasse. In de superklasse, `Language`, zijn de abstracte methodes voorzien van uitgebreide documentatie.

Compileren van de code

Een eerste en belangrijke methode is de callback voor de compilatiestap:

```
def compilation(self, config: Config, files: List[str]) → CallbackResult:
    main_file = files[-1]
    exec_file = Path(main_file).stem
    result = executable_name(exec_file)
    return (["gcc", "-std=c11", "-Wall", "evaluation_result.c", "values.c",
             main_file, "-o", result], [result])
```

De eerste parameter van deze methode is een klasse met enkele configuratie-opties, zoals de tijdslimiet, geheugenlimiet en de programmeertaalspecifieke opties. Dit zou bijvoorbeeld gebruikt kunnen worden om de versie van C mee te geven (zoals C11 of C99). Dit wordt momenteel niet gedaan in C, want TESTed vereist C11, maar de mogelijkheid bestaat. Wel moeten we opmerken dat de tijdslimiet zelden nuttig zal zijn, daar TESTed de uitvoeringstijd bijhoudt. In de configuratieklasse is het dus niet nodig om daar rekening mee te houden.

De andere parameter van deze methode is een lijst van bestanden waarvan TESTed vermoedt dat ze nuttig kunnen zijn voor de compilatiestap. Het bevat onder andere de dependencies uit het configuratiebestand, de ingediende oplossing en de uit de sjablonen gegenereerde bestanden. Die laatste bestanden zijn bijvoorbeeld de verschillende contexten bij een batchcompilatie, maar kunnen ook de evaluator zijn bij een geprogrammeerde evaluatie. De bestanden bestaan uit de naam en een bestandsextensie.

De conventie is om het bestand met de main-functie als laatste te plaatsen.

Al deze bestanden zullen zich in de map bevinden waarin de compilatie plaatsvindt. Het is niet verplicht om al deze bestanden ook effectief te gebruiken: sommige programmeertalen hebben zelf een detectiesysteem voor bestanden. Zo is het in C voldoende om enkel het laatste bestand met de main-functie te gebruiken: alle andere bestanden worden gevonden door GCC.

Concreet ziet een argument voor deze parameter er bijvoorbeeld als volgt uit:

```
['values.py', 'evaluation_utils.py', 'context_0_0.py']
```

Als returnwaarde moet deze methode een tuple met twee element teruggeven: het compilatiecommando en een lijst van resulterende bestanden of een filter.

Het compilatiecommando neemt de vorm aan van een lijst van de elementen waaruit het commando bestaat. Bij het uitvoeren van dit commando zal deze lijst aan de Python-module `subprocess` gegeven worden.

Na het uitvoeren van het compilatiecommando moet TESTed weten welke bestanden relevant zijn om mee te nemen naar een volgende stap in de beoordeling. Daarom moet een lijst van resulterende bestanden teruggegeven worden. Enkel bestanden in deze lijst zullen bijvoorbeeld beschikbaar zijn bij het uitvoeren van de contexten. Een lijst van bestanden teruggeven is mogelijk indien op voorhand geweten is in welke bestanden de compilatie resulteert. Dit is bijvoorbeeld hier het geval (in C resulteert de compilatie in één uitvoerbaar bestand), of ook bij Python, waar de compilatie

voor elk `.py`-bestand resulteert in een `.pyc`-bestand. Ook hier moet de conventie gerespecteerd worden dat het bestand met de `main`-functie als laatste komt.

Het is echter niet altijd mogelijk om op voorhand te weten in welke bestanden de code zal resulteren. Zo resulteert compilatie van één `.java`-bestand mogelijk in meerdere `.class`-bestanden, afhankelijk van de inhoud van de bestanden. Om dit op te lossen kan in plaats van een lijst ook een filterfunctie teruggegeven worden.

Nadat de compilatie uitgevoerd is, zal TESTed deze filter toepassen elk bestand in de map waarin de compilatie uitgevoerd is. De filterfunctie krijgt als argument de naam van een bestand en moet `True` of `False` teruggeven als het bestand respectievelijk wel of niet moet meegenomen worden naar een volgende stap.

Een voorbeeld van de in- en uitvoer van de compilatiemethode:

```
>>> compilation(['submission.c', 'evaluation_result.c', 'context_0_0.c',
 ↪  'selector.c'])
(
    ['gcc', '-std=c11', '-Wall', 'evaluation_result.c', 'values.c', 'selector.c',
     '-o', 'selector.exe'], ['selector.exe']
)
```

Als een leeg compilatiecommando wordt teruggegeven, dan wordt er geen compilatie gedaan. Dit is ook de standaardimplementatie van deze methode. Voor programmeertalen waar geen compilatie nodig is, moet deze methode niet geïmplementeerd worden.

Uitvoeren van de testcode

Na het compileren moeten we een methode implementeren om de gecompileerde code uit te voeren:

```
def execution(self, config: Config,
              cwd: Path, file: str, arguments: List[str]) → Command:
    local_file = cwd / executable_name(Path(file).stem)
    return [str(local_file.absolute()), *arguments]
```

Deze functie heeft vier parameters:

`config` Dezelfde configuratie-opties als bij de compilatiemethode. Bij Java wordt dit bijvoorbeeld gebruikt om de geheugenlimiet van de JVM juist in te stellen.

`cwd` de map waarin het uitvoeren plaatsvindt

`file` het uitvoerbaar bestand dat moet uitgevoerd worden

`arguments` argumenten die aan het proces moeten meegegeven worden

Als resultaat moet het commando teruggegeven worden, dat ook aan `subprocess` doorgegeven wordt.

In het geval van C is dit commando eenvoudig: we geven het absolute pad naar het uitvoerbare bestand mee en geven ook de argumenten mee. Het absolute pad is nodig omdat de executable die we willen uitvoeren (en gemaakt hebben in de compilatiestap) niet in het `PATH` zit.

Een voorbeeld van deze functie in werking is:

```
>>> execution('/test/path', 'executable.exe', ['arg1', 'arg2'])
['/test/path/executable.exe', 'arg1', 'arg2']
```

De basisimplementatie van de configuratie is nu klaar. Voor de meeste programmeertalen kan nu overgegaan worden naar de sjablonen, maar in C moeten we nog een extra methode implementeren.

Aanpassen van de ingediende oplossing

De testcode die door TESTed gegenereerd wordt, kan meerdere `main`-functies bevatten:

- De ingediende oplossing kan een `main`-functie hebben.
- Zowel de contexten als de selector kunnen `main`-functies hebben.

In C kan er slechts één `main`-functie per compilatie zijn.

Een ander probleem is dat de selector elke context insluit (zoals we later zullen zien bij de sjablonen), en elke context ook de oplossing insluit.

Om deze redenen moeten we de code van de ingediende oplossing een beetje aanpassen:

- We voegen aan `guard` toe, zodat de oplossing slechts eenmaal geladen wordt.
- We hernoemen de `main`-functie naar `solution_main` indien die bestaat. Als de `main`-functie geen argumenten had dan voegen we die ook toe.

Vertaald naar de configuratieklasse wordt dit:

```
def solution(self, solution: Path, bundle: Bundle):
    with open(solution, "r") as file:
        contents = file.read()
    # We use regex to find the main function.
    # First, check if we have a no-arg main function.
    # If so, replace it with a renamed main function that does have args.
    no_args = re.compile(r"(int|void)\s+main\s*\(\s*\)\s*{")
    replacement = "int solution_main(int argc, char** argv){"
    contents, nr = re.subn(no_args, replacement, contents, count=1)
    if nr == 0:
        # There was no main function without arguments. Now we try a main
        # function with arguments.
        with_args = re.compile(r"(int|void)\s+main\s*\(\s*int")
        replacement = "int solution_main(int"
```

```
39          contents = re.sub(with_args, replacement, contents, count=1)
40      with open(solution, "w") as file:
41          header = "#pragma once\n\n"
42          file.write(header + contents)
```

4.3.4. Sjablonen

De derde stap bestaat uit het schrijven van de sjablonen. We hebben uiteraard de verplichte sjablonen nodig (zie paragraaf 2.6.2 voor een beschrijving van welke sjablonen verplicht zijn en welke niet), maar om code te hergebruiken kiezen we ervoor om enkele bijkomende sjablonen te schrijven:

`context.c` het sjabloon voor contextcode

`selector.c` het sjabloon voor de selector voor batchcompilatie

`declaration.mako` vertaalt de declaratie van een variabele naar code

`function.mako` vertaalt een functieoproep naar code

`statement.mako` vertaalt een statement of een expressie naar code

`value.mako` vertaalt een letterlijke waarde naar code

`value_arguments.mako` hulpsjabloon voor `value.mako` (opsomming van recursieve gegevenstypes, zoals lijsten, maar ook van functieargumenten)

`value_basic.mako` hulpsjabloon voor `value.mako` (vertaalt de basistypes)

Al deze sjablonen komen in de map `judge/src/tested/languages/c/templates`.

Het is vrij om de bestandsextensie van de sjablonen te kiezen, zolang het een extensie is uit de configuratie. Standaard zijn de toegelaten extensies `.mako` en een programmeertaalafhankelijke extensie, hier `.c`. Een conventie die gebruikt wordt binnen TESTed, is de volgende:

- Sjablonen eindigen op de programmeertaalafhankelijke extensie (`.c`) indien het sjabloon resulteert in een op zichzelf staand bestand. Voorbeelden zijn het contextsjabloon en de selector.
- Sjablonen die resulteren in een codefragment en dus vooral gebruikt worden als onderdeel van andere sjablonen eindigen op `.mako`. Dit zijn bijvoorbeeld de sjablonen om functies en statements om te zetten.

Dit wordt niet afgedwongen door TESTed; alle sjablonen hadden de extensie `.c` of `.mako` kunnen krijgen, of een mengeling. Dit geldt voornamelijk voor de verplichte sjablonen. De andere sjablonen (die als hulpmiddel gebruikt worden door de verplichte sjablonen) kunnen eender welke extensie krijgen, want bij het gebruiken van een sjabloon in Mako moet de bestandsextensie opgegeven worden.

Het contextsjabloon

Dit is veruit het grootste en het meest ingewikkelde sjabloon. Het is verantwoordelijk voor het genereren van de testcode voor één context.

We importeren de values-module (hierover later meer) en de ingediende oplossing. De variabele `submission_name` zal de naam van het oplossingsbestand bevatten. Een overzicht van alle beschikbare variabelen in het contextsjabloon is te vinden in de klasse `_ContextArguments` uit de module `tested.languages.generator`.

We importeren ook alle programmeertaalspecifieke evaluatoren die we nodig zullen hebben. De variabele `evaluator_names` bevat een verzameling van deze namen.

```
#include <stdio.h>

#include "values.h"
#include "${submission_name}.c"

% for name in evaluator_names:
    #include "${name}.c"
% endfor
```

Witruimte in Mako Nuttig om weten is dat TESTed een extensie heeft toegevoegd aan Mako, waardoor de indentatie van Mako-gerelateerde taalconstructies zal verdwijnen. De `for`-loop in het fragment hierboven resulteert bijvoorbeeld in deze code:

```
#include "context_0_0.c"
#include "context_0_1.c"
```

Regeleindes in Mako Ook nuttig om weten is dat een regeleinde in een sjabloon in Mako resulteert in een regeleinde in het geproduceerde bestand. Mako voorziet hier een oplossing voor: door een *backslash* op het einde van de regel te plaatsen zal er geen regeleinde komen in het geproduceerde bestand. Volgende codefragment (let op de \):

```
int test = \⏎
"test";⏎
```

Zal bijvoorbeeld resulteren in deze code:

```
int test = "test";⏎
```

Vervolgens maken we twee variabelen aan waarin de bestanden komen die dienst doen als return- en exception-channel. We noemen deze bestanden de uitvoerbestanden. Merk op dat C geen exceptions ondersteunt, maar TESTed verwacht toch een bestand voor het exception-channel. Anders

zal TESTed ervan uitgaan dat er iets verkeerd liep tijdens het uitvoeren. We definiëren direct ook een functie om de separator naar alle uitvoerkanalen te schrijven.

In onderstaand codefragment en in de rest van het contextsjabloon wordt regelmatig de naam van de context als prefix gebruikt voor functies en variabelen. Dit is omdat het in C niet mogelijk is om in meerdere bestanden functies met dezelfde naam te hebben. Als we dus meerdere contexten samen compileren en elke context heeft zijn eigen `write_separator`-functie, dan zou het compileren mislukken.

```
13 static FILE* ${context_name}_value_file = NULL;
14 static FILE* ${context_name}_exception_file = NULL;
15
16 static void ${context_name}_write_separator() {
17     fprintf(${context_name}_value_file, "--${secret_id}-- SEP");
18     fprintf(${context_name}_exception_file, "--${secret_id}-- SEP");
19     fprintf(stdout, "--${secret_id}-- SEP");
20     fprintf(stderr, "--${secret_id}-- SEP");
21 }
```

Als een resultaat geproduceerd wordt voor de return- of exception-channel, dan moet dat resultaat geserialiseerd worden en naar de uitvoerbestanden geschreven worden. TESTed verwacht dat volgende functies beschikbaar zijn:

`send_value(value)` schrijf een waarde naar een bestand.

`send_exception(exception)` schrijf een exception naar een bestand.

`send_specific_value(value)` schrijf het resultaat van een programmeertaalspecifieke evaluatie naar de return-channel.

`send_specific_exception(exception)` schrijf het resultaat van een programmeertaalspecifieke evaluatie naar de exception-channel.

Bij het implementeren moeten de conventies voor naamgeving van de programmeertaal toegepast worden: zo zal TESTed in Java een oproep naar een functie met naam `sendValue` genereren.

We zullen later zien dat we zelf de oproepen naar deze functies in het sjabloon schrijven. Toegepast op C zijn er wat wijzigingen, omdat C geen exceptions ondersteunt. Als gevolg daarvan zullen we de exception-functies niet implementeren en zullen we ook geen oproep naar deze functies genereren. In appendix B.2.2 staat bijvoorbeeld de gegenereerde code in Java, waar de exception-functies wel gebruikt worden. In C gebruiken we ook een macro in plaats van een functie: dit opnieuw omdat we niet dezelfde functie in meerdere bestanden kunnen definiëren.

```
25 #undef send_value
26 #define send_value(value) write_value(${context_name}_value_file, value)
27
29 #undef send_specific_value
30 #define send_specific_value(value) write_evaluated(${context_name}_value_file,
    ↪ value)
```

We zien ook dat de implementatie eenvoudig is: we geven de gekregen waarde of exception door aan de juiste functie uit de `values`-module en geven ook het bestand mee waarin de waarde of exception moet komen.

De lezer zal zich misschien afvragen waarom het nodig is om deze functies te gebruiken: als TESTed een functieoproep naar deze functies kan definiëren, waarom kan TESTed dan niet direct de `values`-module gebruiken, zonder daar deze functies tussen te plaatsen?

Het antwoord is dat de `values`-module niet verplicht is. Dit is een conventie die in alle ondersteunde programmeertalen gebruikt wordt, maar het is evengoed mogelijk om bij de implementatie van bijvoorbeeld `send_value` de waarde rechtstreeks naar het bestand te schrijven. Deze functies moeten beschouwd worden als de „interface" tussen TESTed en de programmeertaal: TESTed verwacht dat deze functies bestaan en de waarde of exception naar het juiste bestand schrijven, maar hoe dat gebeurt maakt voor TESTed niet uit.

Nu zijn we aangekomen bij het uitvoeren van de testgevallen zelf. In C gebeurt dit in een functie die de naam van de context krijgt. Als eerste stap maken we de bestanden voor de return- en exception-channel aan.

```
33 int ${context_name}() {
34
35     ${context_name}_value_file = fopen("${value_file}", "w");
36     ${context_name}_exception_file = fopen("${exception_file}", "w");
```

Vervolgens printen we de `before`-code. De `before`-code is een fragment code dat uitgevoerd wordt voor het uitvoeren van de context. Deze kan opgegeven worden in het testplan.

Verder schrijven we de *separator* naar de uitvoerbestanden door gebruik te maken van de functie die we eerder gedefinieerd hebben in ons sjabloon. Zoals we reeds bespraken, komt de uitvoer van de return- en exception-channel van alle testgevallen in dezelfde bestanden terecht. Het is nodig om de waarden van elkaar te kunnen onderscheiden, om goed te weten waar de resultaten van een testgeval stoppen en waar de resultaten van het volgende testgeval beginnen. Hiervoor gebruiken we de *separator*.

Het is belangrijk om de separator altijd vóór aanvang van een testgeval naar de uitvoerbestanden te schrijven. TESTed is daar zo op voorzien: de separator na het testgeval uitschrijven zal tot verkeerde resultaten leiden.

We roepen ook de `main`-functie van de oplossing op indien het testplan dat vereist. Oefeningen waar geen `main`-functie opgeroepen wordt zijn bijvoorbeeld deze waarbij de student een functie moet implementeren.

In het codefragment hieronder wordt een oproep gedaan naar de functie `solution_main` uit de ingediende oplossing. Deze functie hebben we zelf gemaakt door de gewone `main`-functie te hernoemen (zie paragraaf 4.3.3). Als de ingediende oplossing geen `main`-functie bevatte, maar het testplan verwachtte die wel, dan zal de compilatie falen.

```
    ${before}

    ${context_name}_write_separator();

    % if context_testcase.exists:
        char* args[] = {\
        % for argument in ["solution"] + context_testcase.arguments:
            "${argument}", \
        % endfor
        };
        solution_main(${len(context_testcase.arguments) + 1}, args);
    % endif
```

Vervolgens genereren we de code voor alle normale testgevallen. Omdat C geen exceptions ondersteunt, is deze implementatie eenvoudig: we schrijven de separator naar de uitvoerbestanden en voeren het invoerstatement uit.

```
    % for testcase in testcases:
        ${context_name}_write_separator();
        <%include file="statement.mako" args="statement=testcase.input_statement()"
↪   />;

    % endfor
```

Dat invoerstatement is `testcase.input_statement()`, wat een geserialiseerd statement zal teruggeven. Wat dat statement juist is, is eigenlijk niet relevant voor het sjabloon, maar het kan toch geen kwaad om het te weten:

- Als de invoer van het testgeval een assignment is, zal dit resulteren in code die er zo uitziet: `int variable = functieoproep();`
- Is de invoer een uitdrukking (*expression*) en zijn we geïnteresseerd in de returnwaarde (het is dus niet van het type `void`), dan zal de gegenereerde code er als volgt uitzien: `send_value(functieoproep());`

Als afsluiter zetten we de `after`-code en sluiten we de bestanden. De `after`-code is analoog aan de `before`-code.

```
    ${after}

    fclose(${context_name}_value_file);
    fclose(${context_name}_exception_file);
    return 0;
}
```

Omdat TESTed zowel contextcompilatie als batchcompilatie ondersteunt, moet elke context een `main`-functie hebben. C laat slechts 1 `main`-functie toe. Indien we in batchcompilatie zitten, zal de selector gebruikt worden, en zal `INCLUDED` op `TRUE` staan. In dat geval voegen we geen `main`-functie toe.

```
#ifndef INCLUDED
int main() {
    return ${context_name}();
}
#endif
```

Het selectorsjabloon

Het is nuttig om er meteen het selectorsjabloon bij te halen: dit wordt gebruikt als TESTed in batchcompilatie werkt en is verantwoordelijk om de juiste context uit te voeren op basis van een argument. Het is in dit sjabloon dat de macro `INCLUDED` op `true` gezet wordt, waardoor de `main`-functies in andere contexten niet gebruikt worden.

```
#include <string.h>
#include <stdio.h>

#define INCLUDED true

% for cont in contexts:
    #include "${cont}.c"
% endfor

int main(int argc, const char* argv[]) {

    if (argc < 1) {
        fprintf(stderr, "No context selected.");
        return -2;
    }

    const char* name = argv[1];
    % for cont in contexts:
        if (strcmp("${cont}", name) == 0) {
            return ${cont}();
        }
    % endfor
    fprintf(stderr, "Non-existing context '%s' selected.", name);
    return -1;
}
```

Het statementsjabloon

Dit sjabloon wordt door TESTed gebruikt om statements te vertalen naar code. Dit omvat onder andere assignments, functieoproepen en waarden:

```
## Convert a statement and/or expression into C code.
<%! from tested.utils import get_args %>\
<%! from tested.serialisation import Value, Identifier, FunctionCall, Assignment
 ↪  %>\
<%page args="statement,full=False"/>\
% if isinstance(statement, Identifier):
    ## If the expression is an identifier, just echo it.
    ${statement}\
% elif isinstance(statement, FunctionCall):
    ## Delegate to the function template for function calls.
    <%include file="function.mako" args="function=statement"/>\
% elif isinstance(statement, get_args(Value)):
    ## We have a value, delegate to the value template.
    <%include file="value.mako" args="value=statement" />\
% else:
    <% assert isinstance(statement, get_args(Assignment)) %>\
    % if full:
        <%include file="declaration.mako" args="value=statement.expression" /> \
    % endif
    ${statement.variable} = <%include file="statement.mako"
 ↪  args="statement=statement.expression"/>;
% endif
```

De implementatie van dit sjabloon komt conceptueel neer op een grote `switch`, waarbij we delegeren naar het juiste sjabloon op basis van welk soort statement of expressie het is.

Een aspect dat wat meer uitleg vraagt, is de `full`-parameter. Dit geeft aan dat het gegevenstype van de variabele bij een assignment ook nodig is. Het verschil is duidelijk met een voorbeeld:

```
int variabele = 5; // met declaration
variabele = 6; // zonder declaration
```

In C is deze parameter minder relevant omdat de tweede variant zelden nodig is, maar deze is vooral nodig in talen zoals Java.

Wat ook nuttig kan zijn, is de functie `get_args`, geïmporteerd uit `tested.utils`. We willen op verschillende plaatsen in het sjabloon een andere actie doen op basis van het soort statement. Normaliter zou een eenvoudige oproep met `isinstance` volstaan. Nu zijn verschillende types, zoals `Value`, `Expression` en `Statement` geen echt type: ze zijn geïmplementeerd als een `Union`. Zo luidt de definitie van `Expression` als volgt: `Expression = Union[Identifier, FunctionCall, Value]`. `Union`-types kunnen niet gebruikt worden in `isinstance`. De `get_args`-functie lost dit probleem op door de `Union` om te zetten naar een `tuple` van echte types. Het is beter de functie `get_args` te veel dan te weinig te gebruiken: de functie werkt ook voor gewone types.

```
>>> isinstance('string', Expression)
TypeError
>>> isinstance('string', get_args(Expression))
False
>>> isinstance('string', get_args(str))  # Ook gewone types
True
```

Overige

De overige sjablonen vertalen elk een taalelement op een gelijkaardige wijze als het statementsjabloon. Het gaat om volgende sjablonen:

- `declaration.mako`
- `function.mako`
- `value.mako`
- `value_arguments.mako`
- `value_basic.mako`

We hebben ze niet opgenomen in dit hoofdstuk, omdat ze sterk lijken op het statementsjabloon. De implementatie van deze sjablonen is te bekijken in de repository.

4.3.5. Hulpmodules

Zoals we in het begin van dit hoofdstuk vermeld hebben, zijn er twee bestanden die als „dependency" opgegeven zijn: `values.c` en `values.h`. Deze bestanden implementeren het serialiseren van data naar het serialisatieformaat en vormen samen de `values`-module. De elementen die geserialiseerd moeten worden:

- Waarden, zoals returnwaarden.
- Exceptions (niet het geval in C, want die bestaan niet in C).
- Resultaten van geprogrammeerde en programmeertaalspecifieke evaluaties.

Hier nemen we de implementatie opnieuw niet op, daar de implementatie van deze module volledig programmeertaalafhankelijk is. In Python is de implementatie eenvoudig door de ingebouwde module `json`, terwijl de implementatie in C een stuk langer is.

4.3.6. Registratie

Als laatste rest nu nog om de nieuwe programmeertaal te registreren bij TESTed. Hiervoor volstaat het om de programmeertaal en de bijhorende configuratieklasse toe te voegen aan het bestand `judge/src/tested/languages/__init__.py`, in de dictionary `LANGUAGES`:

```
LANGUAGES = {
    'c':          C,
    'haskell':    Haskell,
    'java':       Java,
    'javascript': JavaScript,
    'python':     Python,
    'runhaskell': RunHaskell,
}
```

Om de programmeertaal manueel te testen is volgend stappenplan aanbevolen:

1. Implementeer oplossingen voor een of meerdere oefeningen uit de map `exercises` in de nieuwe programmeertaal.
2. Wijzig `judge/src/tested/manual.py` zodat dit bestand de oefening gebruikt waarvoor een oplossing bestaat (en stel ook de juiste programmeertaal in).
3. Voer uit, zoals we in het begin van het hoofdstuk besproken hebben:

```
> python -m tested.manual
```

TESTed heeft ook een testsuite met verschillende oefeningen en scenario's. Om de nieuwe programmeertaal hieraan toe te voegen, moeten de juiste oplossingen geïmplementeerd worden. Hiervoor wordt best gekeken naar `judge/tests/test_functionality.py`. In dat bestand staan de verschillende testen. Bij elke test staat welke oplossing gebruikt wordt; indien het niet duidelijk zou zijn wat de oplossing voor een bepaalde test moet doen, kunnen de bestaande oplossingen in de bestaande programmeertalen een grote hulp zijn.

4.4. Hoe lang duurt het configureren van een programmeertaal?

Een belangrijke factor bij het configureren van een programmeertaal is de kennis over de programmeertaal. Het is nodig om minstens een goede kennis te hebben van volgende aspecten van de programmeertaal:

- Het compileren (indien nodig) en uitvoeren van programma's in de programmeertaal vanop de commandoregel, met alle aspecten die daarbij horen. Een voorbeeld is hoe dependencies meegegeven moeten worden aan het compilatiecommando. In sommige programmeertalen (zoals Java) moeten deze expliciet meegegeven worden, terwijl dat in andere programmeertalen (zoals C) juist niet mag.
- Het gebruik van een `main`-functie of equivalent. In talen als Python is dit niet nodig, daar een bestand met code zo kan uitgevoerd worden. Bij andere talen, zoals Java, C en Haskell is dit wel nodig. Bovendien moet deze `main`-functie programmatisch (vanuit andere code) opgeroepen kunnen worden. Elke context moet namelijk uitvoerbaar zijn op zichzelf, maar ook als onderdeel van een groter programma. In C vraagt dit bijvoorbeeld enige inspanning en wat goochelen met macro's om de `main`-functies juist te krijgen.
- Het schrijven van tekst naar bestanden.

- Hoe de gegevenstypes die ondersteund worden door TESTed voorgesteld kunnen worden in de programmeertaal.
- Het serialiseren van die gegevenstypes naar het JSON-formaat van TESTed. Een mogelijk struikelblok hier is dat TESTed een functie verwacht die het serialiseren doet voor alle gegevenstypes. Dit is in sommige programmeertalen zoals Haskell of C minder voor de hand liggend.
- Andere zaken die de programmeertaal mogelijk ondersteunt, zoals assignments, functieoproepen en objectgerichte acties (objecten aanmaken en gebruiken).

Kennis over hoe TESTed werkt wordt niet verwacht of nodig geacht, met uitzondering van twee onderdelen:

- Informatie over hoe de verschillende compilatiemanier werken, uit paragraaf 2.4.1.
- Informatie over de verschillende evaluatievormen, uit paragraaf 2.5.

Voor de rest zou het moeten volstaan om dit hoofdstuk door te nemen en de stappen die er in beschreven staan uit te voeren. Op verschillende plaatsen wordt verwezen naar andere onderdelen van de masterproef: deze kunnen nuttig zijn om het hoe en waarom te beantwoorden, maar zijn niet strikt nodig voor het implementeren van een programmeertaal. Ook de bestaande configuraties zijn een grote bron van informatie, zeker als het op syntaxis aankomt. Een voorbeeld hiervan zijn de Mako-sjablonen. We verwachten dat het doornemen van dit hoofdstuk en het bekijken van de bestaande sjablonen alle informatie verschaft die nodig is om aan de slag te kunnen, zonder dat de handleiding van Mako doorgenomen moet worden.

Zoals we bespreken in paragraaf 4.5, zijn er wel inspanningen geleverd om het configureren van een programmeertaal flexibel te maken, maar dat neemt niet weg dat het configureren van een programmeertaal grotendeels opgesteld is op basis van de noden van de bestaande programmeertalen. Hoe meer de programmeertaal afwijkt van de bestaande programmeertalen, des te meer creativiteit en tijd nodig zal zijn om de programmeertaal in TESTed te configureren of om meer hooks en callbacks te voorzien in de configuratieklasse.

Dit alles neemt niet weg dat we optimistisch zijn dat het configureren van een programmeertaal eerder snel kan verlopen. Gegeven een goede kennis over de programmeertaal (en dat de programmeertaal niet te exotisch is), schatten we in dat het configureren van een programmeertaal ongeveer een dag in beslag neemt. Hiermee bedoelen we dit hoofdstuk lezen en de stappen die erin beschreven staan uitvoeren, zodat er een werkende configuratie is. Om de programmeertaal tot in de puntjes uit te werken voorzien we een kleine twee dagen. Hiermee bedoelen we bijvoorbeeld het toevoegen van de programmeertaal aan de testen, waarvoor er verschillende oefeningen opgelost moeten worden.

4.5. Stabiliteit van TESTed

Met stabiliteit wordt hier bedoeld hoe weinig er bij het toevoegen van een nieuwe programmeertaal of het opstellen van nieuwe soorten oefeningen nog veranderd moet worden aan TESTed zelf, configuraties van bestaande programmeertalen en de testplannen van bestaande oefeningen.

We kunnen de interne werking van TESTed in grote lijnen verdelen in drie onderdelen:

- De interface naar de oefeningen toe. Hieronder vallen vooral het testplan en het serialisatieformaat.
- De interface naar de programmeertalen toe. Hieronder vallen vooral het configuratiebestand, de configuratieklasse en de verplichte sjablonen.
- Het interne deel waar het testplan uitgevoerd wordt, de ingediende oplossingen beoordeeld worden en de resultaten aan Dodona doorgegeven worden.

De grenzen tussen deze onderdelen zijn niet strikt: zo zal de configuratieklasse ook gebruikt worden in het interne deel tijdens de uitvoering van het testplan.

We willen opmerken dat de focus in deze masterproef vooral lag op de interface naar de oefeningen toe (dus het formaat van het testplan en het serialisatieformaat). We verwachten dan ook niet dat het toevoegen van een programmeertaal grote onverwachte wijzigingen vereist in het formaat van testplan of het serialisatieformaat. Met grote wijzigingen bedoelen we fundamentele wijzigingen aan de manier waarop deze systemen werken. Dit in tegenstelling tot kleinere wijzigingen: een voorbeeld hiervan is een onderscheid maken tussen constanten en veranderlijken (de `final` uit Java en de `const` uit C). Momenteel maakt TESTed dit onderscheid niet, simpelweg omdat er geen *use case* voor is. Zou dit onderscheid toch nodig zijn in bepaalde programmeertalen, dan is het voldoende om een veldje toe te voegen aan het serialisatiemodel van de assignment.

Daarnaast spreken we van onverwachte wijzigingen, omdat sommige delen van TESTed juist voorzien zijn op wijzigingen. Een voorbeeld zijn de gegevenstypes uit het serialisatieformaat. Daar is het juist de bedoeling dat nieuwe programmeertalen bijkomende geavanceerde gegevenstypes toevoegen indien nodig. Als een nieuwe programmeertaal bijvoorbeeld ondersteuning wil bieden voor oneindige generators (bijvoorbeeld een functie die getallen blijft teruggeven), dan is dat mogelijk. Er zullen ook geen wijzigingen nodig zijn aan de bestaande programmeertalen bij het toevoegen van nieuwe gegevenstypes. Aan de testplannen zelf (dus niet het formaat, maar de JSON-bestanden van de oefeningen) verwachten we niet dat er wijzigingen nodig zijn.

Het omgekeerde is ook waar: we verwachten dat er wijzigingen nodig kunnen zijn aan de configuratieklasse en/of de sjablonen bij nieuwe programmeertalen. Een stabiele interface tussen de programmeertalen enerzijds en de kern van TESTed anderzijds was geen expliciet doel in deze masterproef. Hierbij wel twee nuances:

- Hoewel er een poging gedaan is om de configuratieklasse flexibel te maken, is de functionaliteit ervan vooral voorzien op de ondersteunde programmeertalen en de programmeertalen die er op lijken. Het is ook op voorhand moeilijk in te schatten welke functionaliteit nodig zal zijn voor nieuwe programmeertalen.
- De wijzigingen zullen naar verwachting vooral in de configuratieklasse zijn. De methodes om de code te compileren en uit te voeren krijgen momenteel als argumenten de nodige informatie. Het is goed mogelijk dat een nieuwe programmeertaal meer informatie nodig zal hebben. Een ander aspect waar wijzigingen mogelijk zijn, is welke informatie beschikbaar wordt gesteld aan de sjablonen. Ook hier is het mogelijk dat een programmeertaal meer informatie nodig heeft dan de informatie die TESTed momenteel aanreikt.

De ontwikkeling van TESTed vond grotendeels plaats met drie programmeertalen: Python, Java en Haskell. C is pas in een later stadium toegevoegd, waar de meeste functionaliteit van TESTed reeds bestond. Bij het configureren van C zijn wijzigingen nodig geweest aan de configuratieklasse. Een voorbeeld is dat bij de methode voor het uitvoeren van de code nu ook het volledige pad naar de map waarin het uitvoeren gebeurt een argument is. C compileert naar een uitvoerbaar bestand, wat tot dan nog niet gebeurde (Haskell gebruikte nog `runhaskell`). Om het uitvoerbare bestand

uit te voeren is een absoluut pad naar dat uitvoerbaar bestand nodig, maar die informatie was niet voorhanden in de methode. Hierbij moest de configuratieklasse van de bestaande programmeertalen ook licht gewijzigd worden, want de `execute`-functie die geïmplementeerd was in deze klassen kreeg een bijkomende parameter.

4.6. Ondersteunde programmeertalen

Momenteel ondersteunt TESTed 5 programmeertalen:

- Python
- Java
- Haskell (met GHC, al is er een variant die werkt met `runhaskell`)
- C (toegevoegd in dit hoofdstuk als voorbeeld)
- JavaScript (grotendeels toegevoegd door prof. Dawyndt als controle dat dit hoofdstuk slaagt in zijn opzet: duidelijk maken hoe een programmeertaal geconfigureerd moet worden)

Met het toevoegen van C hebben we ondersteuning voor een eerste programmeertaal waarvoor er nog geen judge bestond in Dodona.

5. Conclusie, beperkingen en toekomstig werk

De doelstelling van deze masterproef was het formuleren van een antwoord op de vraag of programmeertaalonafhankelijke oefeningen mogelijk zijn (zie paragraaf 1.5). Als antwoord hierop is een prototype, TESTed, ontwikkeld. Op basis van het prototype kunnen we concluderen dat het antwoord op de onderzoeksvraag positief is.

Eigen aan een prototype is dat het nog niet *production ready* is. Dit rest van dit hoofdstuk handelt over ontbrekende functionaliteit of beperkingen van TESTed, en reikt mogelijke ideeën voor oplossingen aan.

5.1. Dodona-platform

Geen enkele bestaande judge in het Dodona-platform ondersteunt meerdere programmeertalen. Dit heeft gevolgen voor hoe Dodona omgaat met programmeertalen, zoals hieronder duidelijk wordt.

5.1.1. Programmeertaalkeuze in het Dodona-platform

Een eerste probleem is dat binnen Dodona de programmeertaal gekoppeld is aan een oefening en aan een judge, niet aan een oplossing. Dit impliceert dat elke oplossing van een oefening in dezelfde programmeertaal geschreven is. Een aanname die niet meer geldt bij TESTed. Een uitbreiding van Dodona bestaat er dus in om ondersteuning toe te voegen voor het kiezen van een programmeertaal per oplossing. Dit moet ook in de gebruikersinterface mogelijk zijn, zodat studenten kunnen kiezen in welke programmeertaal ze een oefening oplossen.

Als *workaround* zijn er twee suboptimale opties:

- Dezelfde oefening in meerdere programmeertalen aanbieden, door bijvoorbeeld met symlinks dezelfde opgave, testplan en andere bestanden te gebruiken, maar met een ander configbestand. Op deze manier zullen er in Dodona meerdere oefeningen zijn. Deze aanpak heeft nadelen. Zo zijn de oefeningen effectief andere oefeningen vanuit het perspectief van Dodona. Dient een student bijvoorbeeld een oplossing in voor één van de oefeningen, worden de overige oefeningen niet automatisch als ingediend gemarkeerd. Ook is de manier waarop de opgave, het testplan, enzovoort gedeeld worden (de symlinks) niet echt handig.
- TESTed zelf de programmeertaal laten afleiden op basis van de ingediende oplossing. Hiervoor zijn verschillende implementaties mogelijk, zoals een heuristiek die op basis van de ingediende broncode de programmeertaal voorspelt of de student die programmeertaal expliciet laten aanduiden.

Er is voorlopig gekozen om de tweede manier te implementeren: TESTed ondersteunt een speciale *shebang*[1] die aangeeft in welke programmeertaal de ingediende oplossing beoordeeld moet worden. De eerste regel van de ingediende oplossing kan er als volgt uitzien:

```
#!tested [programmeertaal]
```

Hierbij wordt `[programmeertaal]` vervangen door de naam van de programmeertaal, zoals `java` of `python`. Dit is een voorlopige oplossing die toelaat om oefeningen in meerdere programmeertalen aan te bieden zonder dat Dodona moet aangepast worden. Een nadeel aan deze oplossing is dat de student niet kan zien in welke programmeertalen ingediend kan worden; dit moet door de lesgever aangeduid worden.

5.1.2. Detectie ondersteunde programmeertalen

Een ander gevolg van de koppeling van de programmeertaal aan een oefening binnen Dodona is dat Dodona momenteel altijd weet voor welke programmeertaal een oplossing wordt ingediend: de programmeertaal van de oefening. Het is echter niet voldoende dat de programmeertaal van de oplossing gekend is bij het indienen, het is ook nuttig te weten in welke programmeertalen een oefening gemaakt kan worden. Dit om enerzijds dingen als syntaxiskleuring te kunnen toepassen, maar ook om ervoor te zorgen dat een student geen nutteloos werk verricht en een oplossing maakt in een niet-ondersteunde programmeertaal.

Zoals besproken in paragraaf 2.3.4, controleert TESTed of de programmeertaal van de ingediende oplossing ondersteund wordt door het testplan van de oefening. Die controle komt echter te laat, namelijk bij het beoordelen van een oplossing. Aan de andere kant is de controle onafhankelijk van de oplossing: enkel het testplan is nodig. Een beter ogenblik om de ondersteunde programmeertalen van een oefening te bepalen, is bij het importeren van de oefening in het Dodona-platform. Bij het importeren gebeurt al een verwerking van de oefening door Dodona, om bijvoorbeeld de configuratiebestanden te lezen en de oefening in de databank op te slaan. Deze verwerking zou kunnen uitgebreid worden om ook de ondersteunde programmeertalen te bepalen. Merk op dat deze controle opnieuw moet uitgevoerd worden elke keer dat een nieuwe programmeertaal aan TESTed toegevoegd wordt. Dat deze controle steeds opnieuw moet gebeuren is misschien gewenst: zo is meteen duidelijk welke oefeningen in de nieuwe programmeertaal opgelost kunnen worden.

De volgende vraag die zich stelt is hoe dit praktisch moet aangepakt worden. Het bepalen van de ondersteunde programmeertalen is iets specifieks voor TESTed, terwijl judge-specifieke code zoveel mogelijk buiten Dodona wordt gehouden. Een mogelijke oplossing is de verwerking van de oefeningen uit te breiden, zodat de judge ook een verwerking kan doen van de oefening. Deze aanpak heeft als voordeel dat het een generieke oplossing is: alle judges krijgen de mogelijkheid tot het verwerken van de oefening.

Een andere variant van deze oplossing wordt overwogen door het Dodona-team[2]. Het voorstel is om oefeningen en judges te combineren in Dodona tot één Docker-container. Interessant voor TESTed is het toevoegen van een `prepare`-stap aan de judge (momenteel heeft een judge slechts één stap, namelijk `run`, het beoordelen). Deze voorbereidende stap resulteert dan in een Docker-container op maat van de oefening. Het is in deze voorbereidende stap dat het detecteren van de ondersteunde programmeertalen zou kunnen plaatsvinden.

[1] Zie https://en.wikipedia.org/wiki/Shebang_(Unix).
[2] Zie https://github.com/dodona-edu/dodona/issues/1754

Een andere oplossing voor dit probleem is de ondersteunde programmeertalen manueel opgeven in de configuratiebestanden van de oefening. Hier is dan weer het voordeel dat de verwerkingsstap van Dodona niet uitgebreider en trager moet worden. Aan de andere kant mag de auteur van de oefening dan niet vergeten om de programmeertalen in de configuratie op te nemen. Eventueel kan dit gemakkelijker gemaakt worden door een script te voorzien dat de ondersteunde programmeertalen detecteert en het resultaat in de configuratiebestanden schrijft. In de twee eerdere voorstellen wordt telkens gesproken over een verwerkingsstap, waarin het detecteren plaatsvindt. Bij deze laatste oplossing verschuift de verantwoordelijkheid voor het uitvoeren van deze stap van Dodona naar de auteur van de oefening. Als er een nieuwe programmeertaal aan TESTed wordt toegevoegd, dan verwacht deze oplossing ook dat de auteur voor elke oefening het configuratiebestand aanpast (indien de nieuwe oefening beschikbaar moet zijn in de nieuwe programmeertaal).

5.1.3. Beperken van ondersteunde programmeertalen

De vorige paragraaf heeft het steeds over het automatisch detecteren van de ondersteunde programmeertalen. Om twee redenen kan het echter nuttig zijn om deze automatisch detectie te overschrijven.

Ten eerste is de automatische detectie niet altijd voldoende streng, wat ertoe leidt dat TESTed een programmeertaal als ondersteund beschouwd, terwijl de oefening niet oplosbaar is in die programmeertaal. Een concreet voorbeeld hiervan is de detectie van functieargumenten met heterogene gegevenstypes (zie paragraaf 2.3.4 voor een voorbeeld). De automatische detectie houdt enkel rekening met *literal* functieargumenten. De gegevenstypes van functieoproepen en identifiers worden niet gecontroleerd, aangezien deze informatie niet eenvoudig af te leiden is uit het testplan. In dit eerste geval is het dus nuttig dat de auteur van de oefening op het niveau van een oefening een *blacklist* of zwarte lijst van programmeertalen kan meegeven: programmeertalen waarin de oefening niet kan gemaakt worden. Een belangrijk detail is dat als er nieuwe programmeertalen bijkomen in TESTed, de oefening standaard wel ondersteund zou worden in die nieuwe programmeertaal.

De andere grote reden is dat lesgevers de programmeertalen waarin een oefening opgelost kan worden willen beperken. Dit kan bijvoorbeeld zijn omdat de lesgever slechts programmeertalen wil toelaten die hij machtig is, of dat de oefening gebruikt wordt in een vak waar één bepaalde programmeertaal onderwezen wordt. Binnen Dodona stelt een lesgever binnen een vak reeksen oefeningen op. Die reeksen kunnen bestaan uit eigen, zelfgeschreven oefeningen, maar kunnen ook bestaan uit oefeningen die beschikbaar zijn in het Dodona-platform. De auteur van de oefening en lesgever zijn dus niet altijd dezelfde persoon. Om die reden is het nuttig als lesgevers op het niveau van het vak, de reeks of de oefening een *whitelist* of witte lijst van programmeertalen zouden kunnen opgeven: oefeningen in dat vak of die reeksen zullen enkel in de toegelaten programmeertalen kunnen opgelost worden. Bij deze beperking worden nieuwe programmeertalen niet automatisch ondersteund. Het zou ook nuttig zijn om ook hier met een zwarte lijst te kunnen werken, waarbij nieuwe talen wel automatisch ondersteund worden.

5.2. Testplan

De gekozen aanpak voor het testplan heeft bepaalde beperkingen (zie paragraaf 2.3.1 voor de gemaakte keuze). Deze paragraaf bespreekt drie grote beperkingen en mogelijke oplossingen.

5.2.1. Handgeschreven testplannen zijn omslachtig

In TESTed is gekozen om het testplan in JSON op te stellen (zie paragraaf 2.3.1). Bovendien is het een ontwerpkeuze geweest om zoveel mogelijk informatie expliciet te maken. Dit heeft als voornaamste voordeel dat de implementatie in TESTed eenvoudiger blijft. Anderzijds zorgt dit wel voor redelijk wat herhaalde informatie in het testplan.

De combinatie van JSON met de herhaalde informatie zorgt ervoor dat een testplan vaak lang is. Te lang om nog met de hand te schrijven. Een oplossing hiervoor, die al gebruikt wordt, is om het testplan niet met de hand te schrijven, maar te laten generen door bijvoorbeeld een Python-script.

Een andere oplossing is het toevoegen van een bijkomende stap voor het testplan: het testplan wordt dan in een ander formaat geschreven, en bijvoorbeeld tijdens het importeren in Dodona omgezet naar het JSON-bestand. Voor dat andere formaat kan een DSL (*domain-specific language*) nuttig zijn. Dit is een programmeertaal die specifiek ontworpen is voor een bepaald onderwerp of doel. Hier zou ze specifiek ontworpen zijn om een testplan te schrijven.

Een bijkomend voordeel van JSON als formaat is dat het een neutraal formaat is. Verschillende soorten oefeningen hebben andere noden op het vlak van schrijven van het testplan. Het testplan voor een oefening die enkel `stdin` en `stdout` gebruikt zal er helemaal anders uitzien dan een testplan voor een oefening waar functieoproepen gebruikt worden. Het zou bijvoorbeeld kunnen dat voor verschillende soorten oefeningen een andere DSL ontworpen wordt.

Zo zijn er bijvoorbeeld meerdere standaarden voor het schrijven van programmeeroefeningen. Een programma dat oefeningen, die in deze formaten beschreven zijn, omzet naar het eigen formaat van TESTed kan een nuttige uitbreiding zijn. Op het moment van schrijven lijken er echter nog geen wijdverspreide standaarden te zijn. Bij wijze van voorbeeld worden twee standaarden besproken, telkens met een beschrijving van dezelfde oefening: de *Palindroom*-oefening. Bij deze voorbeeldoefening moet invoer gelezen worden en dan als uitvoer moet aangegeven worden of de invoer al dan niet een palindroom is.

Een eerste standaard is PEXIL (*Programming Exercises Interoperability Language*) [33]. Dit is een op XML-gebaseerd formaat, gepubliceerd in 2011. Ontwikkeling van deze standaard lijkt gestopt te zijn. We hebben geen online oefeningen gevonden die in dit formaat beschreven zijn en de nodige resources voor het zelf schrijven van een oefeningen (zoals het XML Schema) zijn niet meer online beschikbaar. De beschrijving van de *Palindroom*-oefening staat in codefragment 5.1.

Een tweede standaard is PEML (*Programming Exercise Markup Language*) [34]. PEML is een tekstueel formaat (een soort DSL) dat er veelbelovend uitziet om met de hand te schrijven. Op het moment van schrijven van deze masterproef is PEML nog in volle ontwikkeling. Aangezien een van de *design goals* van PEML is dat de omzetting van PEML naar JSON eenvoudig zal zijn, lijkt dit een interessant formaat. De beschrijving van de *Palindroom*-oefening staat in codefragment 5.2. Dit voorbeeld is overgenomen uit de documentatie van PEML.

5.2.2. Dynamische scheduling van testen

Een ander nadeel van de keuze voor JSON als formaat voor het testplan, is dat de *scheduling* (het plannen van welke testcode wanneer uitgevoerd worden) statisch gebeurt. Een analogie is bijvoorbeeld een gecompileerde taal zoals C/C++ en een dynamische taal zoals Python: TESTed vervult in

```xml
<specification line_terminator="\n" id="input">
    <line>
        <data type="text" value="racecar"/>
    </line>
    <line>
        <data type="text" value="Flintstone"/>
    </line>
</specification>
```

```xml
<specification line_terminator="\n" id="output">
    <line>
        <data type="text">
            "racecar" is a palindrome."
        </data>
    </line>
    <line>
        <data type="text">
            "Flintstone" is not a palindrome.
        </data>
    </line>
</specification>
```

Codefragment 5.1. Beschrijving van de oefening „palindroom” (gesplitst in invoer en uitvoer) in PEXIL.

```
exercise_id: edu.vt.cs.cs1114.palindromes

title: Palindromes (A Simple PEML Example)

topics: Strings, loops, conditions
prerequisites: variables, assignment, boolean operators

instructions: ----------
Write a program that reads a single string (in the form of one line of text) from
its standard input, and determines whether the string is a _palindrome_. A
palindrome is a string that reads the same way backward as it does forward, such as
"racecar" or "madam". Your program does not need to prompt for its input, and
 ↪  should
only generate one line of output, in the following format:

<pre>"racecar" is a palindrome.</pre>

or

<pre>"Flintstone" is not a palindrome.</pre>
----------

assets.test_format: stdin-stdout

[assets.tests]
stdin: racecar
stdout: "racecar" is a palindrome.

stdin: Flintstone
stdout: "Flintstone" is not a palindrome.

stdin: url(some/local/input.txt)
stdout: url(some/local/output.txt)

stdin: url(http://example.com/some/local/generator/input)
stdout: url(http://example.com/some/local/generator/output)
```

Codefragment 5.2. Beschrijving van de oefening „palindroom” in PEML. Overgenomen uit [34].

deze analogie de rol van compiler. Door dit statische testplan is het niet mogelijk om in het algemeen op basis van het resultaat van een vorig testgeval het verloop van de volgende testgevallen te bepalen.

Het statisch testplan *an sich* is echter voor specifieke gevallen geen probleem. Het testplan kan uitgebreid worden met verschillende mogelijkheden. Zo is reeds voorzien in het testplan om een testgeval als essentieel aan te duiden: faalt het testgeval, dan zullen volgende testgevallen niet meer uitgevoerd worden. Dit zou uitgebreid kunnen worden zodat het ook op het niveau van contexten werkt.

Dit systeem kan verder uitgebreid worden door aan elke context een unieke identificatiecode toe te kennen. Daarna kan voor elke context een preconditie gegeven worden: welke contexten moet geslaagd zijn om de huidige context uit te voeren. Zijn niet alle contexten uit de preconditie geslaagd, dan wordt de context niet uitgevoerd. Het testplan kan bevatten dat context 15 enkel uitgevoerd kan worden indien contexten 1, 7 en 14 ook succesvol waren.

5.2.3. Programmeertaalkeuze voor contexten

Een andere variant van dynamische testplannen situeert zich op het niveau van de programmeertalen. Het kan nuttig zijn om per context optioneel aan te duiden in welke programmeertalen deze uitgevoerd kan worden. Dit gaat zowel over een expliciete lijst van programmeertalen als een automatische bepaling op basis van gebruikte functionaliteit. Als er momenteel functionaliteit gebruikt wordt die niet ondersteund wordt door een programmeertaal, kan het volledige testplan niet uitgevoerd worden in die programmeertaal, ook al is het slechts één context die niet ondersteunde functionaliteit gebruikt. Bij deze uitbreiding zou dan enkel die ene context niet uitgevoerd worden.

Een scenario waar dit nuttig kan zijn is bijvoorbeeld het gebruik van benoemde argumenten. Het kan nuttig zijn om dergelijke argumenten te vertalen naar positionele argumenten in programmeertalen die geen ondersteuning hebben voor benoemde argumenten. Dit is echter niet mogelijk voor contexten waar de argumenten in een andere volgorde gegeven zijn. Deze laatste contexten zouden dan enkel uitgevoerd worden als de programmeertaal benoemde argumenten ondersteunt. Bijvoorbeeld:

```
1 functie(argument1='Hallo', argument2=5)  # In alle programmeertalen
2 functie(argument2=10, argument1='Dag')   # Enkel in bepaalde programmeertalen
```

5.2.4. Herhaalde uitvoeringen van testgevallen

Een andere, maar aan het dynamisch uitvoeren verwante, beperking is het ontbreken van de mogelijkheid om contexten of testgevallen meerdere keren uit te voeren. Dit herhaald uitvoeren is nuttig bij niet-deterministische oefeningen. Een oefening waarbij waarden bijvoorbeeld uit een normale verdeling getrokken moeten worden, zal een duizendtal keren uitgevoerd moeten worden vooraleer met voldoende zekerheid kan geconstateerd worden of er effectief een normale verdeling gebruikt wordt of niet.

Dit is ook op te lossen met een uitbreiding van het testplan: per testgeval of per context zou kunnen aangeduid worden hoeveel keer deze moet uitgevoerd worden. Om de performantie te verbeteren, zouden ook stopvoorwaarden kunnen meegegeven worden. Een aspect dat dit complexer maakt,

is dat de evaluatoren (zowel de ingebouwde evaluatie binnen TESTed als de geprogrammeerde en programmeertaalspecifieke evaluatie) aangepast zullen moeten worden om niet enkel met één resultaat, maar met een reeks resultaten te werken. Dit vermijden is mogelijk door de bestaande evaluatoren niet te wijzigen en ze per uitvoering op te roepen, zoals nu gebeurt. We kunnen dan een finale, globale evaluator toevoegen die de resultaten in hun geheel beoordeelt. Dit geeft ook een mogelijkheid voor optimalisatie: als een evaluatie van een individuele uitvoering faalt, kan het uitvoeren al gestop worden. Stel bijvoorbeeld de *Lotto*-oefening. Als bij de eerste uitvoering al blijkt dat de returnwaarde (een lijst) niet voldoende getallen bevat, kan de evaluatie al stoppen.

5.2.5. Beperkingen van de statements en expressies in het testplan

De statements en expressies in het testplan zijn opzettelijk eenvoudig gehouden (zie paragraaf 2.3.3). Dit levert wel beperkingen op: zo is het niet mogelijk om bijvoorbeeld een functieoproep als `test(5 + 5)` in het testplan te schrijven. Een idee zou dus kunnen zijn om het testplan uit te breiden met meer taalconstructies, waarbij het testplan dan dienst doet als een soort AST (abstract syntax tree). Afhankelijk van hoe ver men we hierin willen gaan, begint de vorm van een universele programmeertaal aan te nemen (wat ook expliciet buiten het bestek van deze masterproef valt). In dat geval loont het de moeite om te onderzoeken of geen eenvoudige, bestaande programmeertaal *transpiled* zou kunnen worden naar het testplan, in plaats van zelf een nieuwe taal op te stellen. Hiervoor moet het testplan wel uitgebreid worden met functies van een AST of IR (intermediary language). Ook TESTed zelf zal uitgebreid moeten worden, aangezien er meer zal moeten omgezet kunnen worden naar de programmeertalen (de sjablonen zullen dus uitgebreid moeten worden). We kunnen nog verder gaan door een transpiler en/of compiler in TESTed zelf te integreren. In dat geval bevat het testplan geen AST of IR, maar de programmeertaal (in tekstuele vorm).

5.2.6. Ondersteuning voor meerdere natuurlijke talen

Sommige judges voor specifieke programmeertalen in Dodona hebben ondersteuning voor het vertalen op het gebied van natuurlijke talen, zoals de vertaling van bijvoorbeeld namen van functies of variabelen. De vertaling van de beschrijving van de opgave is dan weer geïmplementeerd in Dodona zelf door een tweede bestandsextensie. Zo zullen `description.nl.md` en `description.en.md` gebruikt worden voor respectievelijk Nederlands en Engels.

Een eenvoudige oplossing voor het vertalen in de code is iets gelijkaardigs doen met het testplan: `testplan.nl.json` en `testplan.en.json` voor respectievelijk Nederlands en Engels. Deze aanpak heeft wel als nadeel dat veel zaken in het testplan onnodig zullen gedupliceerd moeten worden; een waarde `5` is niet taalafhankelijk.

Een idee dat dit zou kunnen voorkomen is om binnen eenzelfde testplan ondersteuning te bieden voor vertalingen, bijvoorbeeld door telkens meerdere talen te aanvaarden. Momenteel ziet een functienaam er bijvoorbeeld als volgt uit:

```
{
  "name": "zijn_isbn"
}
```

Ondersteuning voor vertaling kan dan deze vorm aannemen:

```
{
  "name": {
    "en": "are_isbn",
    "nl": "zijn_isbn"
  }
}
```

Als een functie binnen een testplan veel wordt opgeroepen, dan zorgt dit nog steeds voor veel herhaling. Dit kan opgelost worden door een DSL, maar dat neemt niet weg dat het testplan onnodig lang zal worden. Een mogelijke oplossing hiervoor is om een soort vertaalwoordenboek mee te geven in het testplan, waar met sleutelwoorden gewerkt wordt. Een voorbeeld illustreert dit:

```
{
  "translations": {
    "are_isbn": {
      "en": "are_isbn",
      "nl": "zijn_isbn"
    }
  }
}
```

De functieoproep gebruikt dan een sleutelwoord uit het vertaalwoordenboek, waarna TESTed bij het uitvoeren de juiste naam gebruikt:

```
{
  "name": "$are_isbn"
}
```

Ook de foutboodschappen die TESTed zelf genereert zijn bijvoorbeeld nog niet vertaald: de boodschappen met studenten als doelpubliek zijn in het Nederlands, terwijl de boodschappen voor cursusbeheerders in het Engels zijn. Dit is echter eenvoudig op te lossen: TESTed krijgt de natuurlijke taal mee van Dodona in de configuratie, en Python heeft meerdere internationalisatie-API's[3]. Het is waarschijnlijk ook nodig om de evaluatoren uit te breiden zodat zij ook de taal als een argument meekrijgen, zodat zij hun boodschappen ook kunnen vertalen.

5.3. Performantie

Ook op het gebied van performantie zijn er nog mogelijkheden tot verbetering.

[3]Zie https://docs.python.org/3/library/gettext.html

5.3.1. Uitvoeren van contexten

Het uitvoeren van elke context in een afzonderlijk subprocess brengt een inherente performantiekost met zich mee. Een mogelijke uitbreiding is het toelaten om meerdere contexten in hetzelfde proces en na elkaar uit te voeren, waardoor de totale tijd die nodig is om een oplossing te beoordelen kan verminderd worden, of minstens onderzoeken of wat het effect hiervan is. De meeste infrastructuur om dit te doen bestaat al: de selector voert bijvoorbeeld een context uit op basis van een argument. Er zou een speciaal argument `all` kunnen voorzien worden, waarmee alle contexten zullen uitgevoerd worden. Mogelijke problemen of zaken die nog moeten opgelost worden:

- Programmeertalen waar geen selector nodig is, zoals Python, zullen dan toch een selector nodig hebben.
- Bij oefeningen waarbij bestanden aangemaakt moeten worden, mogen die bestanden elkaar niet overschrijven.
- Geprogrammeerde evaluaties zullen pas na het uitvoeren van alle contexten kunnen gebeuren. Ook de resultaten van contexten zullen pas erna verwerkt kunnen worden. Dit omdat de geprogrammeerde evaluatie niet noodzakelijk in dezelfde programmeertaal als de oefening gebeurt en dus via TESTed moet gaan. Momenteel voert TESTed de testcode uit, wacht op het resultaat en voert vervolgens de evaluatiecode uit (per context). Het uitvoeren van alle contexten in een keer verandert niets aan de volgorde waarin deze stappen gebeuren. Dit wijzigen zal een ingrijpende verandering aan TESTed zijn.

Ook blijven alle redenen waarom contexten afzonderlijk uitgevoerd worden gelden, zoals het feit dat studenten dan bijvoorbeeld met statische variabelen gegevens kunnen delen tussen de contexten. We zouden dan ook enkel aanraden om deze modus te gebruiken bij eenvoudige oefeningen.

Hetzelfde effect kan nu al bereikt worden door alle testgevallen in het testplan in dezelfde context onder te brengen. Het „probleem" hierbij is dat dat dan ook zo aan de studenten getoond wordt in Dodona. Een alternatieve oplossing bestaat er dan uit om toe te laten dat TESTed aan Dodona doorgeeft dat elk testgeval in een eigen context uitgevoerd werd, terwijl dat niet zo is. Een bedenking hier is dat dit een verschil introduceert tussen de structuur van het testplan en de structuur van de uitvoer die Dodona toont, net iets dat we proberen te vermijden.

5.3.2. Geprogrammeerde evaluatie

Het uitvoeren van een geprogrammeerde evaluatie (zie paragraaf 2.5.2) zorgt voor een niet te verwaarlozen kost op het vlak van performantie (zie tabel 2.3 voor enkele tijdsmetingen). De reden hiervoor is eenvoudig te verklaren: zoals de contexten wordt elke geprogrammeerde evaluatie in een afzonderlijk subproces uitgevoerd (afhankelijk van de programmeertaal ook met eigen compilatiestap). Zoals vermeld in paragraaf 2.6 is TESTed oorspronkelijk gestart met het uitvoeren van code in Jupyter-kernels. De grootste reden dat we daar vanaf gestapt zijn, is dat de kost voor het opnieuw opstarten van een kernel te groot is, en het heropstarten noodzakelijk is om de onafhankelijkheid van de contexten te garanderen. Bij n contexten moet de kernel dus n keer opnieuw gestart worden.

Bij een geprogrammeerde evaluatie wordt geen code van de student uitgevoerd: het opnieuw starten van de kernel is dus niet nodig. Hierdoor wordt de opstartkost van de kernel verspreid over alle contexten die van een programmeerde evaluatie in dezelfde programmeertaal gebruikmaken. Wordt er n keer andere evaluatiecode in een andere programmeertaal gebruikt, dan zal dit uiteraard

geen voordeel opleveren, omdat er dan nog steeds *n* keer een kernel gestart wordt. In de meeste gevallen wordt echter dezelfde evaluatiecode gebruikt voor alle testgevallen. Het *worst case scenario* is dan weliswaar trager, maar in de meeste gevallen zal de geprogrammeerde evaluatie sneller zijn.

Hierop is één uitzondering: een geprogrammeerde evaluatie waarbij de programmeertaal van de evaluatiecode in Python geschreven is. Voor Python is er speciale ondersteuning (vermits TESTed zelf ook in Python geschreven is, wordt deze evaluatie ook rechtsreeks in TESTed gedaan zonder subproces): hiervoor is het gebruik van Jupyter-kernels niet nuttig.

5.3.3. Optimalisaties bestaande parallellisatie

Zoals we vermeld hebben in paragraaf 2.6.3, zijn er nog een aantal beperkingen en opmerkingen bij de implementatie van het parallel uitvoeren van de contexten, waarvan de belangrijkste zijn:

- De contexten per tabblad geparallelliseerd worden, terwijl tabbladen nog steeds sequentieel uitgevoerd worden. In het ergste geval, waarbij elk tabblad één context heeft, gebeurt niets in parallel.
- Enkel het uitvoeren gebeurt in parallel, niet de beoordeling van het resultaat van het uitvoeren. Een gevolg hiervan is dat geprogrammeerde evaluaties ook sequentieel uitgevoerd worden, terwijl dit ook in parallel zou kunnen gebeuren. De reden hiervoor is technisch van aard: de uitvoer van de beoordeling moet in volgorde naar Dodona gestuurd worden, maar intern stuurt TESTed een resultaat meestal direct door nadat het beschikbaar is. De beoordelingen in parallel uitvoeren zou er dus voor zorgen dat de uitvoer door elkaar staat.

5.3.4. Optimalisatie uitvoering

Bij het uitvoeren wordt voor elke context een nieuwe map gemaakt, waarna de verschillende benodigde bestanden naar die map gekopieerd worden. Het is het onderzoeken waard of hier optimalisaties mogelijk zijn, door bijvoorbeeld de bestanden niet te kopiëren, maar te *symlinken*. Ook zou onderzocht kunnen worden of die nieuwe map nodig is: misschien kan een speciaal bestandssysteem gebruikt worden dat *copy-on-write* ondersteunt, waardoor geen nieuwe map moet gemaakt worden voor elke context. Dit laatste systeem zou ook het voordeel hebben dat beter gecontroleerd kan worden welke bestanden aangemaakt zijn door de ingediende oplossing.

5.4. TESTed

In deze paragraaf komen de beperkingen aan bod die ofwel enkel betrekking hebben op TESTed zelf, ofwel betrekking hebben op meerdere onderdelen, bijvoorbeeld zowel TESTed als het testplan.

5.4.1. Programmeertaalonafhankelijke opgaven

Bij veel oefeningen bevat de opgave een stuk code ter illustratie van de opgave. Dit is expliciet buiten het bestek van deze masterproef gehouden, maar deze voorbeelden zijn idealiter in de programmeertaal waarin de student wenst in te dienen.

Een idee is hier om de codevoorbeelden uit de opgave ook in het formaat van het testplan op te stellen. TESTed bevat namelijk alles om het testplan om te zetten naar concrete code.

Concreet gaat het om volgende zaken:

1. De programmeertaalkeuze in het Dodona-platform moet uitgebreid worden zodat de opgave ook programmeertaalafhankelijk is en dus kan gewijzigd worden als de student een andere programmeertaal kiest.
2. De uitleg bij een opgave (bijvoorbeeld omkadering of bijkomende informatie) is vaak programmeertaalonafhankelijk, terwijl de meer technische onderdelen programmeertaalafhankelijk zijn. Voorbeelden van dat laatste zijn datastructuren (`list`/`array`) of codevoorbeelden. Hiervoor zijn ook meerdere mogelijkheden: zo zou alles binnen eenzelfde bestand kunnen, of kan er één bestand per programmeertaal zijn, waarbij de gemeenschappelijke delen in een apart bestand komen.

5.4.2. Programmeertaalonafhankelijke programmeerparadigmata

TESTed vertaalt geen programmeerparadigmata tussen verschillende programmeertalen. Dit kan ervoor zorgen dat bepaalde oefeningen in sommige programmeertalen op onnatuurlijke wijze opgelost moeten worden. Stel als voorbeeld de ISBN-oefeningen. In Python is het *pythonic* om hiervoor twee top-level functies te schrijven (`is_isbn` en `are_isbn`). In Java zal TESTed dan twee statische functies verwachten, terwijl deze opgave in de Java-wereld ook vaak opgelost zal worden met bijvoorbeeld een klasse `IsbnValidator`, met twee methoden `check` en `checkAll`.

Er zijn meerdere denkpistes om hier een oplossing voor te bieden:

- Voorzie binnen TESTed ook vertalingen van programmeerparadigmata. Zo zou voor een oefening opgegeven kunnen worden of de Java-oplossing een klasse of statische methodes moet gebruiken.
- Maak een systeem met hybride oefeningen, waarbij de invoer programmeertaalafhankelijk is, terwijl de evaluatie van resultaten programmeertaalonafhankelijk blijft. In het voorbeeld hierboven zou een lesgever dan voor elke programmeertaal opgeven hoe een resultaat bekomen moet worden (in Python twee functieoproepen, in Java een instantie van een klasse maken en twee methoden oproepen), waarna TESTed overneemt om een programmeertaalonafhankelijke evaluatie te doen van de resultaten.

Het omgekeerde, programmeertaalonafhankelijk invoer en programmeertaalspecifieke beoordeling, bestaat al binnen TESTed (zie paragraaf 2.5.3). Als tot slot zowel de invoer als de uitvoer verschilt van programmeertaal tot programmeertaal, kan er niet meer gesproken worden van een programmeertaalonafhankelijke oefeningen. In dat geval is het beter een bestaande programmeertaalspecifieke judge van Dodona te gebruiken of voor elke programmeertaal een eigen testplan op te stellen.

5.4.3. Flexibelere evaluatievormen

Momenteel moet bij de evaluatie een keuze gemaakt worden: het is ofwel een ingebouwde evaluatie, ofwel een geprogrammeerde evaluatie, ofwel een programmeertaalspecifieke evaluatie. In sommige scenario's is het echter wenselijk om een combinatie te kunnen gebruiken, vooral bij de geprogrammeerde evaluatie. Een voorbeeld is het volgende: stel dat de oefening vereist dat een functie geïmplementeerd moet worden die een getal moet teruggeven. Dit getal is het resultaat van een niet-deterministische berekening, dus is een geprogrammeerde evaluatie aangewezen. Het zou dan nuttig zijn dat TESTed al controleert of de door de oplossing geproduceerde waarde het juiste gegevenstype heeft. In talen zoals Haskell of C is het niet eenvoudig om een evaluatiefunctie te schrijven waaraan arbitraire types kunnen meegegeven worden.

Een ander scenario waar dit nuttig kan zijn, is als er voorbereidend werk nodig is voor de evaluatie. Stel dat een functie geïmplementeerd moet worden die een datum teruggeeft. In veel programmeertalen is dit een object (denk aan `DateTime` in Java of `datetime.Date` in Python). Het zou nuttig kunnen zijn als in de programmeertaalspecifieke evaluatie deze datum bijvoorbeeld omgezet wordt naar een string, waarna TESTed de evaluatie kan overnemen. Op deze manier moet er in elke programmeertaal minder evaluatiecode geschreven worden.

Hieraan gerelateerd is een nieuwe evaluatiemodus, die we een *orakelevaluatie* kunnen noemen. In deze evaluatiemodus staat de waarde waarmee vergeleken moet worden niet in het testplan, maar wordt deze waarde berekend, door bijvoorbeeld de voorbeeldoplossing uit te voeren met dezelfde invoer als de ingediende oplossing. Het voordeel van deze evaluatiemodus is dat de vergelijking nog steeds kan gebeuren door TESTed en niet moet gebeuren in een geprogrammeerde evaluatie. Vanuit dat oogpunt is de orakelevaluatie op te vatten als een combinatie van een geprogrammeerde evaluatie en een ingebouwde evaluatie, wat ons weer bij de vorige alinea's brengt. De geprogrammeerde evaluatie zou dan de waarde berekenen, waarna TESTed de evaluatie overneemt en de ingebouwde evaluatie uitvoert. Dit kan nuttig zijn bij oefeningen die bijvoorbeeld iets berekenen op basis van de huidige datum.

Het samenvoegen van beide concepten leidt tot een evaluatiepijplijn, waar de evaluatie doorheen verschillende stadia gaat:

1. Het verwachte resultaat kan geleverd worden door de orakelevaluatie.
2. Een programmeertaalspecifieke evaluatie wordt uitgevoerd.
3. Een geprogrammeerde evaluatie wordt uitgevoerd.
4. De ingebouwde evaluatie van TESTed wordt uitgevoerd.

Elk van de stappen is optioneel, al moet minstens één stap uitgevoerd worden. Is er bijvoorbeeld geen orakelevaluatie, dan wordt de waarde uit het testplan gebruikt. Elke stap krijgt als argument dan een evaluatiepakket, dat bestaat uit onder andere het resultaat van de vorige evaluatie en de geproduceerde waarde.

5.5. Extra functionaliteit

Er is in de loop van de masterproef extra functionaliteit naar boven gekomen waarvoor ook ondersteuning zou toegevoegd kunnen worden:

- Ondersteuning voor de Python Tutor. Dit is een visuele debugger, die origineel geschreven is voor Python [35]. Ondertussen worden ook andere talen ondersteund, zoals Java, C++, JavaScript en Ruby. In Dodona ondersteunt momenteel enkel de Python-judge de Python Tutor. Het zou een mooie toevoeging zijn indien dit bij TESTed voor meerdere programmeertalen kan toegevoegd worden.
- Interpreteren van output van de compiler. Momenteel ondersteunt TESTed linting (zie paragraaf 2.7), maar bij sommige talen worden ook veel nuttige waarschuwingen getoond bij het compileren van de code. Deze waarschuwingen worden momenteel getoond in TESTed, en er is ondersteuning om deze waarschuwingen om te zetten naar annotaties op de code van de oplossing. Dit laatste is momenteel geïmplementeerd voor de compilatiestap in Python. De ondersteuning hiervoor zou uitgebreid kunnen worden naar andere talen en flexibeler gemaakt kunnen worden. Zo wordt de uitvoer van de compiler altijd getoond, ook al is die uitvoer omgezet naar annotaties (dezelfde informatie wordt dus dubbel getoond).
- Foutboodschappen beter afhandelen. Als momenteel een fout optreedt, wordt de stacktrace zonder verdere verwerking getoond aan de gebruiker. Het is echter nuttig hier een verwerkingsstap toe te voegen, zodat:
  - De code gegenereerd door TESTed weggefilterd kan worden. Het kan verwarrend zijn voor studenten als de stacktrace verwijst naar code die zij niet ingediend hebben.
  - Het aanbrengen van links naar specifieke plaatsen in de ingediende code. Als er bijvoorbeeld in de foutboodschap staat „fout op regel 5, kolom 10”, dan zou een snelkoppeling naar die plaats in de code nuttig zijn. Dodona biedt hier ondersteuning voor en dit wordt ook gebruikt in de Python-judge.
- Lange uitvoer beperkt tonen. TESTed bevat momenteel een eenvoudige limiet: de uitvoer wordt beperkt tot tienduizend tekens. Deze limiet houdt echter geen rekening met de aard van de uitvoer (bijvoorbeeld een boodschap van TESTed of de uitvoer van de compiler). Zo zou een stacktrace anders ingekort kunnen worden.
- Datastructuren mooier weergeven (*pretty printing*). Enerzijds gaat het om grote datastructuren verkort weer te geven. Is de uitvoer bijvoorbeeld een lijst van duizend elementen, dan zouden enkel de eerste en laatste tien getoond kunnen worden. Momenteel worden alle waarden verbatim aan de studenten getoond. Anderzijds gaat het ook om het beter weergeven van waarden om het vergelijken met de verwachte uitvoer gemakkelijker te maken. Een voorbeeld is verzamelingen gesorteerd tonen, zodat het vinden van de verschillen eenvoudiger wordt. Ook het tonen van complexe datastructuren op meerdere regels is hier een voorbeeld van.
- Tijdslimieten op het niveau van de contexten. Momenteel is het enkel mogelijk om op het niveau van een oefening een tijdslimiet in te stellen. De implementatie hiervan zou eenvoudig moeten zijn: intern werkt TESTed al met een tijdslimiet per context, namelijk de totale tijdslimiet van de oefening minus de verstreken tijd.
- Gerelateerd aan de tijdslimieten is er binnen Dodona nog geen status voor een test die niet uitgevoerd is[4]. Momenteel toont TESTed de niet-uitgevoerde testen met de status „fout”. Het is echter wenselijk deze oefening als „niet-uitgevoerd” aan te duiden. Ook visueel zou er een verschil kunnen zijn, door bijvoorbeeld een grijze achtergrond te gebruiken in plaats van een rode achtergrond.

[4]Zie de issue: https://github.com/dodona-edu/dodona/issues/1785

Bibliografie

[1] M. Duranton, K. De Bosschere, B. Coppens, C. Gamrat, M. Gray, H. Munk, E. Ozer, T. Vardanega en O. Zendra, *HiPEAC Vision 2019*. jan 2019, hfdstk. Impact of computing technology on society, p. 110–111, ISBN: 9789090313641.

[2] Dienst Curriculum en vorming, „Computationeel denken in Zin in leren! Zin in leven!" Katholiek Onderwijs Vlaanderen, 25 sep 2017.

[3] „Decreet van 14 december 2018 betreffende de onderwijsdoelen voor de eerste graad van het secundair onderwijs," *Belgisch Staatsblad*, jrg. 26/04/2019, p. 40 558, 14 dec 2018. adres: http://www.ejustice.just.fgov.be/mopdf/2019/04/26_1.pdf#page=44.

[4] B. Bastiaensen en J. De Craemer, *Zo denkt een computer*. Vlaamse Overheid, 2017, hfdstk. Computationeel denken, p. 8–9.

[5] A. Luxton-Reilly, Simon, I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, L. Ott, J. Paterson, M. J. Scott, J. Sheard en C. Szabo, „Introductory Programming: A Systematic Literature Review," in *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, reeks ITiCSE 2018 Companion, Larnaca, Cyprus: Association for Computing Machinery, 2018, p. 55–106, ISBN: 9781450362238. DOI: 10.1145/3293881.3295779.

[6] H. Keuning, J. Jeuring en B. Heeren, „Towards a Systematic Review of Automated Feedback Generation for Programming Exercises," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, reeks ITiCSE '16, Arequipa, Peru: Association for Computing Machinery, 2016, p. 41–46, ISBN: 9781450342315. DOI: 10.1145/2899415.2899422.

[7] A. Kosowski, M. Małafiejski en T. Noiński, „Application of an Online Judge & Contester System in Academic Tuition," in *Advances in Web Based Learning – ICWL 2007*, H. Leung, F. Li, R. Lau en Q. Li, red., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 343–354, ISBN: 978-3-540-78139-4. DOI: 10.1007/978-3-540-78139-4_31.

[8] Wikipedia Contributors. (25 jan 2020). „Comparison of data-serialization formats, Wikipedia, The Free Encyclopedia," adres: https://en.wikipedia.org/w/index.php?title=Comparison_of_data-serialization_formats&oldid=937433197.

[9] *Apache Avro™ 1.9.1 Documentation*, The Apache Foundation, 9 feb 2019. adres: http://avro.apache.org/docs/1.9.1/ (bezocht op 27-01-2020).

[10] *Apache Parquet*, The Apache Foundation, 13 jan 2020. adres: https://parquet.apache.org/documentation/latest/ (bezocht op 27-01-2020).

[11] „Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation," International Telecommunications Union, Recommendation X.608, aug 2015. adres: https://www.itu.int/rec/T-REC-X.680-201508-I/en.

[12] B. Cohen. (4 feb 2017). „The BitTorrent Protocol Specification," adres: http://bittorrent.org/beps/bep_0003.html.

[13] B. Ramos. (25 sep 2019). „Binn," adres: https://github.com/liteserver/binn.

[14] (19 jul 2019). „BSON 1.1," MongoDB, adres: http://bsonspec.org/ (bezocht op 17-01-2020).

[15] C. Bormann en P. Hoffman, „Concise Binary Object Representation (CBOR)," Internet Engineering Task Force, RFC 7049, okt 2013. adres: https://tools.ietf.org/html/rfc7049.

[16] W. van Oortmerssen. (24 apr 2019). „FlatBuffers," Google, adres: https://google.github.io/flatbuffers/.

[17] „Information technology – Generic applications of ASN.1: Fast infoset," International Telecommunications Union, Recommendation X.881, 14 mei 2005. adres: https://www.itu.int/rec/T-REC-X.891-200505-I/en.

[18] *Amazon Ion*, Amazon, 15 jan 2020. adres: https://amzn.github.io/ion-docs/ (bezocht op 27-01-2020).

[19] S. Furuhashi. (17 sep 2018). „MessagePack," adres: https://msgpack.org/.

[20] (25 feb 2018). „OGDL 2018.2," adres: https://msgpack.org/ (bezocht op 28-01-2020).

[21] „OPC unified architecture - Part 1: Overview and concepts," International Electrotechnical Commission, IEC TR 62541-1:2016, 10 mei 2016. adres: https://webstore.iec.ch/publication/25997.

[22] E. Lengyel. (17 jan 2017). „Open Data Description Language (OpenDDL)," adres: http://openddl.org/.

[23] (13 dec 2019). „ProtocolBuffers," Google, adres: https://developers.google.com/protocol-buffers/.

[24] Jackson JSON team. (2010). „Smile Data Format," adres: https://github.com/FasterXML/smile-format-specification.

[25] N. Mitra en Y. Lafon, „SOAP Version 1.2 Part 0: Primer (Second Edition)," W3C, TR, apr 2007. adres: http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[26] M. Wildgrube, „Structured Data Exchange Format (SDXF)," Internet Engineering Task Force, RFC 3072, mrt 2001. adres: https://tools.ietf.org/html/rfc3072.

[27] M. Slee, A. Agarwal en M. Kwiatkowski, „Thrift: Scalable cross-language services implementation," 2007. adres: http://thrift.apache.org/static/files/thrift-20070401.pdf.

[28] (25 feb 2018). „Universal Binary JSON," adres: http://ubjson.org/.

[29] M. Bayer, G. T. Dairiki, P. Jenvey, D. Peckam, A. Ronacher, B. Bangert en B. Trofatter. (20 jan 2020). „Mako Templates for Python," adres: https://www.makotemplates.org/.

[30] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla en C. Willing, „Jupyter Notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides en B. Schmidt, red., IOS Press, 2016, p. 87–90. DOI: 10.3233/978-1-61499-649-1-87.

[31] C. Van Petegem, „Computationele benaderingen voor deductie van de computationele complexiteit van computerprogramma's," Universiteit Gent, 2018. adres: https://lib.ugent.be/catalog/rug01:002479652.

[32] P. Spronck, *De Programmeursleerling, Leren coderen met Python 3*. 24 okt 2017. adres: http://www.spronck.net/pythonbook/dutchindex.xhtml.

[33] R. Queirós en J. P. Leal, „Pexil: Programming exercises interoperability language," in *Conferência Nacional XATA: XML, aplicações e tecnologias associadas*, ESEIG, 2011, p. 37–48, ISBN: 9789899686311. DOI: 10400.22/4748.

[34] P. Conrad, C. Bart en S. Edwards. (). „Programming Exercise Markup Language,” Standards, Protocols, en Learning Infrastructure for Computing Education (SPLICE), adres: https://cssplice.github.io/peml/ (bezocht op 31-03-2020).

[35] P. J. Guo, „Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education,” in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, reeks SIGCSE '13, Denver, Colorado, USA: Association for Computing Machinery, 2013, p. 579–584, ISBN: 9781450318686. DOI: 10.1145/2445196.2445368.

A. Mapstructuur na uitvoeren

Het eerste codefragment toont de mapstructuur van werkmap na het uitvoeren van een beoordeling voor een ingediende oplossing in Python. In de map common zit alle testcode en de gecompileerde bestanden voor alle contexten. Voor elke context worden de gecompileerde bestanden gekopieerd naar een andere map, bijvoorbeeld `context_0_1`, wat de map is voor context 1 van tabblad 0 van het testplan. Het tweede codefragment toont de mapstructuur na het uitvoeren van een geprogrammeerde evaluatie.

```
workdir                          //Werkmap van TESTed
├── common                       //Gemeenschappelijke code
│   ├── context_0_0.py           //Broncode voor context 0 van tabblad 0
│   ├── context_0_0.pyc          //Gecompileerde code voor context 0 van tabblad 0
│   ├── context_0_1.py
│   ├── context_0_1.pyc
│   ...
│   ├── context_0_49.py
│   ├── context_0_49.pyc
│   ...
│   ├── context_1_49.py          //Broncode voor context 49 van tabblad 1
│   ├── context_1_49.pyc
│   ├── submission.py            //Ingediende oplossing
│   ├── submission.pyc
│   ├── values.py                //Values-module
│   └── values.pyc
├── context_0_0                  //Map voor context 0-0
│   ├── FaLd6WGRN_exceptions.txt //Uitzonderingskanaal
│   ├── FaLd6WGRN_values.txt     //Kanaal voor returnwaarden
│   ├── context_0_0.pyc          //Code voor context 0-0
│   ├── submission.pyc
│   └── values.pyc
...
```

```
temp                        //Tijdelijke map
├── buzzchecker.py          //Evaluatiecode
├── evaluation_utils.py     //Hulpfuncties
├── evaluator_executor.py   //Code om de evaluatiecode uit te voeren
└── values.py               //Values-module
```

B. Echo-oefening

Deze bijlage bevat de door TESTed gegenereerde testcode voor het testplan uit de oefening *Echo* uit hoofdstuk 3 (zie paragraaf 3.3.3 voor het testplan.)

B.1. Python

B.1.1. Oplossing

```
i = input()
print(i)
```

B.1.2. Gegenereerde code

De testcode die gegenereerd wordt bij het uitvoeren van bovenstaande testplan. Deze code is lichtjes aangepast: overtollige witruimte is verwijderd. Anders is de code identiek aan de door TESTed gegenereerde code.

In Python is geen selector nodig, waardoor de gegenereerde code identiek is voor batchcompilatie en voor contextcompilatie. In beide gevallen worden twee bestanden gegenereerd:

1. `context_0_0.py`
2. `context_0_1.py`

`context_0_0.py`

```
import values
import sys

value_file = open("lgo0uvyA1_values.txt", "w")
exception_file = open("lgo0uvyA1_exceptions.txt", "w")

def write_separator():
    value_file.write("--lgo0uvyA1-- SEP")
    exception_file.write("--lgo0uvyA1-- SEP")
    sys.stderr.write("--lgo0uvyA1-- SEP")
```

```
    sys.stdout.write("--lgo0uvyA1-- SEP")
    sys.stdout.flush()
    sys.stderr.flush()
    value_file.flush()
    exception_file.flush()

def send_value(value):
    values.send_value(value_file, value)

def send_exception(exception):
    values.send_exception(exception_file, exception)

def send_specific_value(value):
    values.send_evaluated(value_file, value)

def send_specific_exception(exception):
    values.send_evaluated(exception_file, exception)

try:
    write_separator()
    from submission import *
except Exception as e:
    send_exception(e)
else:
    send_exception(None)

value_file.close()
exception_file.close()
```

context_0_1.py

```
import values
import sys

value_file = open("lgo0uvyA1_values.txt", "w")
exception_file = open("lgo0uvyA1_exceptions.txt", "w")

def write_separator():
    value_file.write("--lgo0uvyA1-- SEP")
    exception_file.write("--lgo0uvyA1-- SEP")
    sys.stderr.write("--lgo0uvyA1-- SEP")
    sys.stdout.write("--lgo0uvyA1-- SEP")
    sys.stdout.flush()
    sys.stderr.flush()
    value_file.flush()
```

```
    exception_file.flush()

def send_value(value):
    values.send_value(value_file, value)

def send_exception(exception):
    values.send_exception(exception_file, exception)

def send_specific_value(value):
    values.send_evaluated(value_file, value)

def send_specific_exception(exception):
    values.send_evaluated(exception_file, exception)

try:
    write_separator()
    from submission import *
except Exception as e:
    send_exception(e)
else:
    send_exception(None)

value_file.close()
exception_file.close()
```

B.1.3. Uitvoeren

Bij het uitvoeren worden de gegenereerde bestanden uitgevoerd. In beide gevallen (batchcompilatie en contextcompilatie) wordt de juiste context uitgevoerd:

```
> python context_0_0.py
<uitvoer context_0_0>
> python context_0_1.py
<uitvoer context_0_1>
```

B.2. Java

B.2.1. Oplossing

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

```java

public class Submission {
    public static void main(String[] args) throws Exception {
        BufferedReader reader = new BufferedReader(new
        ↪  InputStreamReader(System.in));
        String name = reader.readLine();
        System.out.println(name);
    }
}
```

B.2.2. Gegenereerde code

De testcode die gegenereerd wordt bij het uitvoeren van bovenstaande testplan. Deze code is lichtjes aangepast: overtollige witruimte is verwijderd. Anders is de code identiek aan de door TESTed gegenereerde code.

Bij batchcompilatie worden alle testgevallen in één keer gecompileerd. Daarvoor wordt het bestand `Selector.java` gegenereerd. Bij contextcompilatie wordt geen `Selector.java` gegenereerd, maar zijn de bestanden anders identiek. Volgende bestanden werden gegenereerd (met `Selector.java` dus enkel in batchcompilatie):

1. `Context00.java`
2. `Context01.java`
3. `Selector.java`

`Context00.java`

```java
import java.io.*;
import java.util.*;
import java.util.function.*;

public class Context00 implements Closeable {

    private final PrintWriter valueWriter;
    private final PrintWriter exceptionWriter;

    public Context00() throws Exception {
        this.valueWriter = new PrintWriter("raIuhcbB2_values.txt");
        this.exceptionWriter = new PrintWriter("raIuhcbB2_exceptions.txt");
    }

    private void writeSeparator() throws Exception {
        valueWriter.write("--raIuhcbB2-- SEP");
        exceptionWriter.write("--raIuhcbB2-- SEP");
```

```java
        System.err.print("--raIuhcbB2-- SEP");
        System.out.print("--raIuhcbB2-- SEP");
        valueWriter.flush();
        exceptionWriter.flush();
        System.err.flush();
        System.out.flush();
    }

    private void sendValue(Object value) throws Exception {
        Values.send(valueWriter, value);
    }

    private void sendException(Throwable exception) throws Exception {
        Values.sendException(exceptionWriter, exception);
    }

    private void sendSpecificValue(EvaluationResult value) {
        Values.sendEvaluated(valueWriter, value);
    }

    private void sendSpecificException(EvaluationResult exception) {
        Values.sendEvaluated(exceptionWriter, exception);
    }

    void execute() throws Exception {

        this.writeSeparator();
        try {
            Submission.main(new String[]{});
            sendException(null);
        } catch (Exception | AssertionError e) {
            sendException(e);
        }

    }

    @Override
    public void close() throws IOException {
        this.valueWriter.close();
        this.exceptionWriter.close();
    }

    public static void main(String[] a) throws Exception {
        try(Context00 context = new Context00()) {
            context.execute();
        }
    }
}
```

Context01.java

```
import java.io.*;
import java.util.*;
import java.util.function.*;

public class Context01 implements Closeable {

    private final PrintWriter valueWriter;
    private final PrintWriter exceptionWriter;

    public Context01() throws Exception {
        this.valueWriter = new PrintWriter("raIuhcbB2_values.txt");
        this.exceptionWriter = new PrintWriter("raIuhcbB2_exceptions.txt");
    }

    private void writeSeparator() throws Exception {
        valueWriter.write("--raIuhcbB2-- SEP");
        exceptionWriter.write("--raIuhcbB2-- SEP");
        System.err.print("--raIuhcbB2-- SEP");
        System.out.print("--raIuhcbB2-- SEP");
        valueWriter.flush();
        exceptionWriter.flush();
        System.err.flush();
        System.out.flush();
    }

    private void sendValue(Object value) throws Exception {
        Values.send(valueWriter, value);
    }

    private void sendException(Throwable exception) throws Exception {
        Values.sendException(exceptionWriter, exception);
    }

    private void sendSpecificValue(EvaluationResult value) {
        Values.sendEvaluated(valueWriter, value);
    }

    private void sendSpecificException(EvaluationResult exception) {
        Values.sendEvaluated(exceptionWriter, exception);
    }

    void execute() throws Exception {

        this.writeSeparator();
```

```
        try {
            Submission.main(new String[]{});
            sendException(null);
        } catch (Exception | AssertionError e) {
            sendException(e);
        }

    }

    @Override
    public void close() throws IOException {
        this.valueWriter.close();
        this.exceptionWriter.close();
    }

    public static void main(String[] a) throws Exception {
        try(Context01 context = new Context01()) {
            context.execute();
        }
    }
}
```

Selector.java

```
class Selector {
    public static void main(String[] a) throws Exception {
        var name = a[0];
        if ("Context00".equals(name)) {
            Context00.main(new String[]{});
        }
        if ("Context01".equals(name)) {
            Context01.main(new String[]{});
        }
    }
}
```

B.2.3. Uitvoeren

Bij batchcompilatie wordt `Selector.java` gecompileerd, wat leidt tot een reeks `.class`-bestanden. Vervolgens wordt de selector tweemaal uitgevoerd:

```
> ./java -cp . Selector Context00
<uitvoer context_0_0>
> ./java -cp . Selector Context01
<uitvoer context_0_1>
```

Een opmerking hierbij teneinde verwarring tegen te gaan: in de commando's hierboven wordt met het argument `Selector` de klasse meegegeven aan Java die uitgevoerd moet worden. Het argument `Context0*` is een commandoargument zoals gebruikelijk.

Bij contextcompilatie wordt elke context afzonderlijk gecompileerd, wat ook leidt tot een reeks `.class`-bestanden. Deze keer worden de contexten zelf uitgevoerd:

```
> java -cp . Context00
<uitvoer context_0_0>
> java -cp . Context01
<uitvoer context_0_1>
```

B.3. C

B.3.1. Oplossing

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

int main() {

    int c;

    while ((c = getchar()) ≠ EOF) {
        putchar(c);
    }

    return EXIT_SUCCESS;
}
```

B.3.2. Gegenereerde code

De testcode die gegenereerd wordt bij het uitvoeren van bovenstaande testplan. Deze code is lichtjes aangepast: overtollige witruimte is verwijderd. Anders is de code identiek aan de door TESTed gegenereerde code.

Bij batchcompilatie worden alle testgevallen in één keer gecompileerd. Daarvoor wordt het bestand `selector.c` gegenereerd. Bij contextcompilatie wordt geen `selector.c` gegenereerd, maar zijn de

bestanden anders identiek. Volgende bestanden werden gegenereerd (met `selector.c` dus enkel in batchcompilatie):

1. `context_0_0.c`
2. `context_0_1.c`
3. `selector.c`

context_0_0.c

```c
#include <stdio.h>

#include "values.h"
#include "submission.c"

static FILE* context_0_0_value_file = NULL;
static FILE* context_0_0_exception_file = NULL;

static void context_0_0_write_separator() {
    fprintf(context_0_0_value_file, "--bKHsgGplf-- SEP");
    fprintf(context_0_0_exception_file, "--bKHsgGplf-- SEP");
    fprintf(stdout, "--bKHsgGplf-- SEP");
    fprintf(stderr, "--bKHsgGplf-- SEP");
}

#undef send_value
#define send_value(value) write_value(context_0_0_value_file, value)

#undef send_specific_value
#define send_specific_value(value) write_evaluated(context_0_0_value_file, value)

int context_0_0() {

    context_0_0_value_file = fopen("bKHsgGplf_values.txt", "w");
    context_0_0_exception_file = fopen("bKHsgGplf_exceptions.txt", "w");

    context_0_0_write_separator();

    char* args[] = {    "solution",     };
    solution_main(1, args);

    fclose(context_0_0_value_file);
    fclose(context_0_0_exception_file);
    return 0;
}

#ifndef INCLUDED
int main() {
```

```
    return context_0_0();
}
#endif
```

context_0_1.c

```
#include <stdio.h>

#include "values.h"
#include "submission.c"

static FILE* context_0_1_value_file = NULL;
static FILE* context_0_1_exception_file = NULL;

static void context_0_1_write_separator() {
    fprintf(context_0_1_value_file, "--bKHsgGplf-- SEP");
    fprintf(context_0_1_exception_file, "--bKHsgGplf-- SEP");
    fprintf(stdout, "--bKHsgGplf-- SEP");
    fprintf(stderr, "--bKHsgGplf-- SEP");
}

#undef send_value
#define send_value(value) write_value(context_0_1_value_file, value)

#undef send_specific_value
#define send_specific_value(value) write_evaluated(context_0_1_value_file, value)

int context_0_1() {

    context_0_1_value_file = fopen("bKHsgGplf_values.txt", "w");
    context_0_1_exception_file = fopen("bKHsgGplf_exceptions.txt", "w");

    context_0_1_write_separator();

    char* args[] = {    "solution",     };
    solution_main(1, args);

    fclose(context_0_1_value_file);
    fclose(context_0_1_exception_file);
    return 0;
}

#ifndef INCLUDED
int main() {
    return context_0_1();
```

```
}
#endif
```

selector.c

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

#define INCLUDED true

#include "context_0_0.c"
#include "context_0_1.c"

int main(int argc, const char* argv[]) {

    if (argc < 1) {
        fprintf(stderr, "No context selected.");
        return -2;
    }

    const char* name = argv[1];
    if (strcmp("context_0_0", name) == 0) {
        return context_0_0();
    }
    if (strcmp("context_0_1", name) == 0) {
        return context_0_1();
    }
    fprintf(stderr, "Non-existing selector '%s' selected.", name);
    return -1;
}
```

B.3.3. Uitvoeren

Bij batchcompilatie wordt `selector.c` gecompileerd, wat leidt tot een uitvoerbaar bestand `selector`. Dit laatste bestand wordt dan tweemaal uitgevoerd:

```
> ./selector context_0_0
<uitvoer context_0_0>
> ./selector context_0_1
<uitvoer context_0_1>
```

Bij contextcompilatie wordt elke context afzonderlijk gecompileerd, wat leidt tot twee uitvoerbare bestanden: `context_0_0` en `context_0_1`. Deze uitvoerbare bestanden worden vervolgens uitgevoerd:

```
> ./context_0_0
<uitvoer context_0_0>
> ./context_0_1
<uitvoer context_0_1>
```

B.4. Haskell

B.4.1. Oplossing

```haskell
main = do
    s ← getLine
    putStrLn s
```

B.4.2. Gegenereerde code

De testcode die gegenereerd wordt bij het uitvoeren van bovenstaande testplan. Deze code is lichtjes aangepast: overtollige witruimte is verwijderd. Anders is de code identiek aan de door TESTed gegenereerde code.

Bij batchcompilatie worden alle testgevallen in één keer gecompileerd. Daarvoor wordt het bestand `Selector.hs` gegenereerd. Bij contextcompilatie wordt geen `Selector.hs` gegenereerd, maar zijn de bestanden anders identiek. Volgende bestanden werden gegenereerd (met `Selector.hs` dus enkel in batchcompilatie):

1. `Context00.hs`
2. `Context01.hs`
3. `Selector.hs`

Op het einde van alle contexten staat in de Haskell-code `putStr ""`. Dit is met opzet en nodig omdat de `do`-notatie in Haskell moet eindigen met een expressie.

`Context00.hs`

```haskell
{-# LANGUAGE NamedFieldPuns #-}

module Context00 where

import System.IO (hPutStr, stderr, stdout, hFlush)
import System.Environment
import qualified Values
import Control.Monad.Trans.Class
import Control.Exception
import EvaluationUtils

import qualified Submission

value_file = "UOUqZ20sh_values.txt"
exception_file = "UOUqZ20sh_exceptions.txt"

writeSeparator :: IO ()
writeSeparator = do
    hPutStr stderr "--UOUqZ20sh-- SEP"
    hPutStr stdout "--UOUqZ20sh-- SEP"
    appendFile value_file "--UOUqZ20sh-- SEP"
    appendFile exception_file "--UOUqZ20sh-- SEP"
    hFlush stdout
    hFlush stderr

sendValue :: Values.Typeable a => a -> IO ()
sendValue = Values.sendValue value_file

sendException :: Exception e => Maybe e -> IO ()
sendException = Values.sendException exception_file

sendSpecificValue :: EvaluationResult -> IO ()
sendSpecificValue = Values.sendEvaluated value_file

sendSpecificException :: EvaluationResult -> IO ()
sendSpecificException = Values.sendEvaluated exception_file

handleException :: Exception e => (Either e a) -> Maybe e
handleException (Left e) = Just e
handleException (Right _) = Nothing

main = do

    writeSeparator

    let mainArgs = {}
    result <- try (withArgs mainArgs Submission.main) :: IO (Either SomeException
     ↪  ())
    let ee = handleException result in sendException (ee)
```

```
    putStr ""
```

Context01.hs

```
{-# LANGUAGE NamedFieldPuns #-}

module Context01 where

import System.IO (hPutStr, stderr, stdout, hFlush)
import System.Environment
import qualified Values
import Control.Monad.Trans.Class
import Control.Exception
import EvaluationUtils

import qualified Submission

value_file = "UOUqZ20sh_values.txt"
exception_file = "UOUqZ20sh_exceptions.txt"

writeSeparator :: IO ()
writeSeparator = do
    hPutStr stderr "--UOUqZ20sh-- SEP"
    hPutStr stdout "--UOUqZ20sh-- SEP"
    appendFile value_file "--UOUqZ20sh-- SEP"
    appendFile exception_file "--UOUqZ20sh-- SEP"
    hFlush stdout
    hFlush stderr

sendValue :: Values.Typeable a => a -> IO ()
sendValue = Values.sendValue value_file

sendException :: Exception e => Maybe e -> IO ()
sendException = Values.sendException exception_file

sendSpecificValue :: EvaluationResult -> IO ()
sendSpecificValue = Values.sendEvaluated value_file

sendSpecificException :: EvaluationResult -> IO ()
sendSpecificException = Values.sendEvaluated exception_file

handleException :: Exception e => (Either e a) -> Maybe e
handleException (Left e) = Just e
handleException (Right _) = Nothing
```

```
41
42 main = do
43
44     writeSeparator
45
46     let mainArgs = {}
47     result ← try (withArgs mainArgs Submission.main) :: IO (Either SomeException
       ↪  ())
48     let ee = handleException result in sendException (ee)
49
50     putStr ""
```

Selector.hs

```
1  module Selector where
2
3  import System.Environment
4
5  import qualified Context00
6  import qualified Context01
7
8  main = do
9      [n] ← getArgs
10     case n of
11         "Context00" → Context00.main
12         "Context01" → Context01.main
```

B.4.3. Uitvoeren

Bij batchcompilatie wordt `Selector.hs` gecompileerd, wat leidt tot een uitvoerbaar bestand `selector`. Dit laatste bestand wordt dan tweemaal uitgevoerd:

```
1 > ./selector Context00
2 <uitvoer context_0_0>
3 > ./selector Context01
4 <uitvoer context_0_1>
```

Bij contextcompilatie wordt elke context afzonderlijk gecompileerd, wat leidt tot twee uitvoerbare bestanden: `Context00` en `Context01`. Deze uitvoerbare bestanden worden vervolgens uitgevoerd:

```
> ./Context00
<uitvoer context_0_0>
> ./Context01
<uitvoer context_0_1>
```

B.5. JavaScript

B.5.1. Oplossing

```
const fs = require("fs");
const stdinBuffer = fs.readFileSync(0); // STDIN_FILENO = 0
console.log(stdinBuffer.toString());
```

B.5.2. Gegenereerde code

De testcode die gegenereerd wordt bij het uitvoeren van bovenstaande testplan. Deze code is lichtjes aangepast: overtollige witruimte is verwijderd. Anders is de code identiek aan de door TESTed gegenereerde code.

In JavaScript is geen selector nodig, waardoor de gegenereerde code identiek is voor batchcompilatie en voor contextcompilatie. In beide gevallen worden twee bestanden gegenereerd:

1. `context00.js`
2. `context01.js`

`context00.js`

```
const fs = require('fs');
const vm = require('vm');
const values = require("./values.js");

const valueFile = fs.openSync("TMIQgB2wo_values.txt", "w");
const exceptionFile = fs.openSync("TMIQgB2wo_exceptions.txt", "w");

function writeSeparator() {
    fs.writeSync(valueFile, "--TMIQgB2wo-- SEP");
    fs.writeSync(exceptionFile, "--TMIQgB2wo-- SEP");
    fs.writeSync(process.stdout.fd, "--TMIQgB2wo-- SEP");
    fs.writeSync(process.stderr.fd, "--TMIQgB2wo-- SEP");
}

```

```
function sendValue(value) {
    values.sendValue(valueFile, value);
}

function sendException(exception) {
    values.sendException(exceptionFile, exception);
}

function sendSpecificValue(value) {
    values.sendEvaluated(valueFile, value);
}

function sendSpecificException(exception) {
    values.sendEvaluated(exceptionFile, exception);
}

new_args = [process.argv[0]]
new_args = new_args.concat([])
process.argv = new_args

try {
    writeSeparator();

    eval(fs.readFileSync("submission.js") + "");

    sendException(null)
} catch(e) {
    sendException(e)
}

fs.closeSync(valueFile);
fs.closeSync(exceptionFile);
```

context01.js

```
const fs = require('fs');
const vm = require('vm');
const values = require("./values.js");

const valueFile = fs.openSync("TMIQgB2wo_values.txt", "w");
const exceptionFile = fs.openSync("TMIQgB2wo_exceptions.txt", "w");

function writeSeparator() {
    fs.writeSync(valueFile, "--TMIQgB2wo-- SEP");
    fs.writeSync(exceptionFile, "--TMIQgB2wo-- SEP");
```

```
    fs.writeSync(process.stdout.fd, "--TMIQgB2wo-- SEP");
    fs.writeSync(process.stderr.fd, "--TMIQgB2wo-- SEP");
}

function sendValue(value) {
    values.sendValue(valueFile, value);
}

function sendException(exception) {
    values.sendException(exceptionFile, exception);
}

function sendSpecificValue(value) {
    values.sendEvaluated(valueFile, value);
}

function sendSpecificException(exception) {
    values.sendEvaluated(exceptionFile, exception);
}

new_args = [process.argv[0]]
new_args = new_args.concat([])
process.argv = new_args

try {
    writeSeparator();

    eval(fs.readFileSync("submission.js") + "");

    sendException(null)
} catch(e) {
    sendException(e)
}

fs.closeSync(valueFile);
fs.closeSync(exceptionFile);
```

B.5.3. Uitvoeren

Bij het uitvoeren worden de gegenereerde bestanden uitgevoerd. In beide gevallen (batchcompilatie en contextcompilatie) wordt de juiste context uitgevoerd:

```
> node context00.js
<uitvoer context_0_0>
> node context01.js
<uitvoer context_0_1>
```

C. Echofunctie-oefening

Deze bijlage bevat de door TESTed gegenereerde testcode voor het testplan uit de oefening *Echofunctie* uit hoofdstuk 3 (zie paragraaf 3.4.3 voor het testplan.)

C.1. Python

C.1.1. Oplossing

```
def echo(content):
    return content
```

C.1.2. Gegenereerde code

De testcode die gegenereerd wordt bij het uitvoeren van bovenstaande testplan. Deze code is lichtjes aangepast: overtollige witruimte is verwijderd. Anders is de code identiek aan de door TESTed gegenereerde code.

In Python is geen selector nodig, waardoor de gegenereerde code identiek is voor batchcompilatie en voor contextcompilatie. In beide gevallen wordt een bestanden gegenereerd:

1. `context_0_0.py`

`context_0_0.py`

```
import values
import sys

value_file = open("t8QUHe5mW_values.txt", "w")
exception_file = open("t8QUHe5mW_exceptions.txt", "w")

def write_separator():
    value_file.write("--t8QUHe5mW-- SEP")
    exception_file.write("--t8QUHe5mW-- SEP")
    sys.stderr.write("--t8QUHe5mW-- SEP")
    sys.stdout.write("--t8QUHe5mW-- SEP")
    sys.stdout.flush()
```

```
    sys.stderr.flush()
    value_file.flush()
    exception_file.flush()

def send_value(value):
    values.send_value(value_file, value)

def send_exception(exception):
    values.send_exception(exception_file, exception)

def send_specific_value(value):
    values.send_evaluated(value_file, value)

def send_specific_exception(exception):
    values.send_evaluated(exception_file, exception)

try:
    write_separator()
    from submission import *
except Exception as e:
    raise e

write_separator()

try:
    send_value(echo('input-1'))
except Exception as e:
    send_exception(e)
else:
    send_exception(None)
write_separator()

try:
    send_value(echo('input-2'))
except Exception as e:
    send_exception(e)
else:
    send_exception(None)

value_file.close()
exception_file.close()
```

C.1.3. Uitvoeren

Bij het uitvoeren worden de gegenereerde bestanden uitgevoerd. In beide gevallen (batchcompilatie en contextcompilatie) wordt de juiste context uitgevoerd:

```
> python context_0_0.py
<uitvoer context_0_0>
```

C.2. Java

C.2.1. Oplossing

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Submission {
    public static void main(String[] args) throws Exception {
        BufferedReader reader = new BufferedReader(new
        ↪  InputStreamReader(System.in));
        String name = reader.readLine();
        System.out.println(name);
    }
}
```

C.2.2. Gegenereerde code

De testcode die gegenereerd wordt bij het uitvoeren van bovenstaande testplan. Deze code is lichtjes aangepast: overtollige witruimte is verwijderd. Anders is de code identiek aan de door TESTed gegenereerde code.

Bij batchcompilatie worden alle testgevallen in één keer gecompileerd. Daarvoor wordt het bestand `Selector.java` gegenereerd. Bij contextcompilatie wordt geen `Selector.java` gegenereerd, maar zijn de bestanden anders identiek. Volgende bestanden werden gegenereerd (met `Selector.java` dus enkel in batchcompilatie):

1. `Context00.java`
2. `Selector.java`

`Context00.java`

```java
import java.io.*;
import java.util.*;
import java.util.function.*;

public class Context00 implements Closeable {
```

```java

    private final PrintWriter valueWriter;
    private final PrintWriter exceptionWriter;

    public Context00() throws Exception {
        this.valueWriter = new PrintWriter("WQbeY1Lm7_values.txt");
        this.exceptionWriter = new PrintWriter("WQbeY1Lm7_exceptions.txt");
    }

    private void writeSeparator() throws Exception {
        valueWriter.write("--WQbeY1Lm7-- SEP");
        exceptionWriter.write("--WQbeY1Lm7-- SEP");
        System.err.print("--WQbeY1Lm7-- SEP");
        System.out.print("--WQbeY1Lm7-- SEP");
        valueWriter.flush();
        exceptionWriter.flush();
        System.err.flush();
        System.out.flush();
    }

    private void sendValue(Object value) throws Exception {
        Values.send(valueWriter, value);
    }

    private void sendException(Throwable exception) throws Exception {
        Values.sendException(exceptionWriter, exception);
    }

    private void sendSpecificValue(EvaluationResult value) {
        Values.sendEvaluated(valueWriter, value);
    }

    private void sendSpecificException(EvaluationResult exception) {
        Values.sendEvaluated(exceptionWriter, exception);
    }

    void execute() throws Exception {

        this.writeSeparator();

        this.writeSeparator();
        try {
            sendValue(Submission.echo("input-1"));
            sendException(null);
        } catch (Exception | AssertionError e) {
            sendException(e);
        }
        this.writeSeparator();
        try {
```

```
            sendValue(Submission.echo("input-2"));
            sendException(null);
        } catch (Exception | AssertionError e) {
            sendException(e);
        }

    }

    @Override
    public void close() throws IOException {
        this.valueWriter.close();
        this.exceptionWriter.close();
    }

    public static void main(String[] a) throws Exception {
        try(Context00 context = new Context00()) {
            context.execute();
        }
    }
}
```

Selector.java

```
class Selector {
    public static void main(String[] a) throws Exception {
        var name = a[0];
        if ("Context00".equals(name)) {
            Context00.main(new String[]{});
        }
    }
}
```

C.2.3. Uitvoeren

Bij batchcompilatie wordt `Selector.java` gecompileerd, wat leidt tot een reeks `.class`-bestanden. Vervolgens wordt de selector uitgevoerd:

```
> ./java -cp . Selector Context00
<uitvoer context_0_0>
```

Bij contextcompilatie wordt elke context afzonderlijk gecompileerd, wat ook leidt tot een reeks `.class`-bestanden. Deze keer wordt de context zelf uitgevoerd:

```
> java -cp . Context00
<uitvoer context_0_0>
```

C.3. C

C.3.1. Oplossing

We voeren het testplan uit met deze oplossing:

```c
char* echo(char* content) {
    return content;
}
```

C.3.2. Gegenereerde code

De gegenereerde code die gegenereerd wordt bij het uitvoeren van bovenstaande testplan. Deze code is lichtjes aangepast: overtollige witruimte is verwijderd. Anders is de code identiek aan de door TESTed gegenereerde code.

Bij batchcompilatie worden alle testgevallen in één keer gecompileerd. Volgende bestanden werden gegenereerd:

1. `context_0_0.c`
2. `selector.c`

`context_0_0.c`

```c
#include <stdio.h>

#include "values.h"
#include "submission.c"

static FILE* context_0_0_value_file = NULL;
static FILE* context_0_0_exception_file = NULL;

static void context_0_0_write_separator() {
    fprintf(context_0_0_value_file, "--UNeB8zAju-- SEP");
    fprintf(context_0_0_exception_file, "--UNeB8zAju-- SEP");
    fprintf(stdout, "--UNeB8zAju-- SEP");
```

```
    fprintf(stderr, "--UNeB8zAju-- SEP");
}

#undef send_value
#define send_value(value) write_value(context_0_0_value_file, value)

#undef send_specific_value
#define send_specific_value(value) write_evaluated(context_0_0_value_file, value)

int context_0_0() {

    context_0_0_value_file = fopen("UNeB8zAju_values.txt", "w");
    context_0_0_exception_file = fopen("UNeB8zAju_exceptions.txt", "w");

    context_0_0_write_separator();

    context_0_0_write_separator();
    send_value(echo("input-1"));

    context_0_0_write_separator();
    send_value(echo("input-2"));

    fclose(context_0_0_value_file);
    fclose(context_0_0_exception_file);
    return 0;
}

#ifndef INCLUDED
int main() {
    return context_0_0();
}
#endif
```

selector.c

```
#include <string.h>
#include <stdio.h>

#define INCLUDED true

#include "context_0_0.c"

int main(int argc, const char* argv[]) {

    if (argc < 1) {
```

```
        fprintf(stderr, "No context selected.");
        return -2;
    }

    const char* name = argv[1];
    if (strcmp("context_0_0", name) == 0) {
        return context_0_0();
    }
    fprintf(stderr, "Non-existing selector '%s' selected.", name);
    return -1;
}
```

C.3.3. Uitvoeren

Bij het compileren wordt `selector.c` gecompileerd, wat leidt tot een uitvoerbaar bestand `selector`. Dit laatste bestand wordt dan uitgevoerd:

```
> ./selector context_0_0
<uitvoer context_0_0>
```

Bij contextcompilatie wordt elke context afzonderlijk gecompileerd, wat leidt tot een uitvoerbaar bestand: `context_0_0`. Dit bestand wordt vervolgens uitgevoerd:

```
> ./context_0_0
<uitvoer context_0_0>
```

C.4. Haskell

C.4.1. Oplossing

We voeren het testplan uit met deze oplossing:

```
echo a = a
```

C.4.2. Gegenereerde code

De gegenereerde code die gegenereerd wordt bij het uitvoeren van bovenstaande testplan. Deze code is lichtjes aangepast: overtollige witruimte is verwijderd. Anders is de code identiek aan de door TESTed gegenereerde code.

Bij batchcompilatie worden alle testgevallen in één keer gecompileerd. Volgende bestanden werden gegenereerd:

1. `Context00.hs`
2. `Selector.hs`

`Context00.hs`

```haskell
{-# LANGUAGE NamedFieldPuns #-}

module Context00 where

import System.IO (hPutStr, stderr, stdout, hFlush)
import System.Environment
import qualified Values
import Control.Monad.Trans.Class
import Control.Exception
import EvaluationUtils

import qualified Submission

value_file = "yq5AlxCOr_values.txt"
exception_file = "yq5AlxCOr_exceptions.txt"

writeSeparator :: IO ()
writeSeparator = do
    hPutStr stderr "--yq5AlxCOr-- SEP"
    hPutStr stdout "--yq5AlxCOr-- SEP"
    appendFile value_file "--yq5AlxCOr-- SEP"
    appendFile exception_file "--yq5AlxCOr-- SEP"
    hFlush stdout
    hFlush stderr

sendValue :: Values.Typeable a => a -> IO ()
sendValue = Values.sendValue value_file

sendException :: Exception e => Maybe e -> IO ()
sendException = Values.sendException exception_file

sendSpecificValue :: EvaluationResult -> IO ()
sendSpecificValue = Values.sendEvaluated value_file

```

```haskell
sendSpecificException :: EvaluationResult → IO ()
sendSpecificException = Values.sendEvaluated exception_file

handleException :: Exception e ⇒ (Either e a) → Maybe e
handleException (Left e) = Just e
handleException (Right _) = Nothing

main = do

    writeSeparator

    writeSeparator

    result0 ← catch
        (return (Submission.echo ("input-1"))

            >>= \r → sendValue (r)
            >> let ee = (Nothing :: Maybe SomeException) in sendException (ee))
        (\e → let ee = (Just (e :: SomeException)) in sendException (ee))

    writeSeparator

    result1 ← catch
        (return (Submission.echo ("input-2"))

            >>= \r → sendValue (r)
            >> let ee = (Nothing :: Maybe SomeException) in sendException (ee))
        (\e → let ee = (Just (e :: SomeException)) in sendException (ee))

    putStr ""
```

Selector.hs

```haskell
module Selector where

import System.Environment

import qualified Context00

main = do
    [n] ← getArgs
    case n of
        "Context00" → Context00.main
```

C.4.3. Uitvoeren

Bij het compileren wordt `Selector.hs` gecompileerd, wat leidt tot een uitvoerbaar bestand `Selector`. Dit laatste bestand wordt dan uitgevoerd:

```
> ./selector Context00
<uitvoer context_0_0>
```

Bij contextcompilatie wordt elke context afzonderlijk gecompileerd, wat leidt tot een uitvoerbaar bestand: `Context00`. Dit bestand wordt vervolgens uitgevoerd:

```
> ./Context00
<uitvoer context_0_0>
```

C.5. JavaScript

We hebben de code van JavaScript niet opgenomen in deze bijlage, omdat de code weinig verschilt van de code in appendix B.5, en ook conceptueel sterk lijkt op de code voor Python van deze oefening.