

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



Inhoudsopgave

1. Educational software testing	4
1.1. Computationeel denken	4
1.2. Programmeeroefeningen	5
1.3. Het leerplatform Dodona	5
1.4. Beoordelen van oplossingen	6
1.5. Probleemstelling	7
1.6. Opbouw van de thesis	11
2. TESTed	12
2.1. Overzicht	12
2.2. Beschrijven van een oefening	16
2.3. Oplossingen uitvoeren	27
2.4. Oplossingen beoordelen	31
2.5. Performantie	36
2.6. Bijkomende taken	41
2.7. Robuustheid	42
2.8. Implementatiekeuzes	44
3. Configuratie van een programmeertaal in TESTed	46
3.1. TESTed lokaal uitvoeren	46
3.2. Globaal stappenplan voor het configureren van een programmeertaal	48
3.3. De programmeertaal C	48
3.4. Hoe lang duurt het implementeren van een programmeertaal?	65
3.5. Stabiliteit van TESTed	65
4. Oefeningen schrijven voor TESTed	67
4.1. Oefeningen in het Dodona-platform	67
4.2. Lotto	68
4.3. Echo	71
4.4. Echofunctie	71
4.5. ZeroDivisionError	71
4.6. Som	73
5. Beperkingen en toekomstig werk	76
5.1. Dodona-platform	76
5.2. Testplan	78
5.3. Performantie	84
5.4. TESTed	85
5.5. Extra functionaliteit	87
A. Echo-oefening	92
A.1. Opgave	92

A.2.	Testplan	93
A.3.	Python	94
A.4.	Java	96
A.5.	C	101
B.	Echofunctie-oefening	106
B.1.	Opgave	106
B.2.	Testplan	106
B.3.	Python	108
B.4.	Java	110
B.5.	C	113

Dankwoord

Dank aan iedereen!

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 [12].

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 [33] of in het secundair onderwijs [11].

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 [22]. 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 [18].

Deze thesis focust op de eerste uitdaging, al wordt ook ingespeeld op de tweede uitdaging, onder andere in paragraaf 2.7. De uitdaging wordt 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 Computationale Biologie van de Universiteit Gent gewerkt met programmeeroefeningen die in een online systeem ingediend en beoordeeld worden. Oorspronkelijk werd hiervoor gebruik gemaakt van de *Sphere Online Judge* (SPOJ) [20]. 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 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. De judge 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 een 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 van de bind mount.
4. De judge wordt uitgevoerd met de configuratie op de standaardinvoerstroom.

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

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

5. De judge beoordeelt de oplossing aan de beoordelingsmethodes opgesteld door de lesgever (d.w.z. de jUnit-test, de 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 standaarduitvoerstream.
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”-oefening³, waarvan de opgave gegeven is in codefragment 1.1. Oplossingen voor deze oefening staan in codefragmenten 1.2 en 1.3, voor respectievelijk Python en Java.

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 leidt dit ook tot verschillende versies van dezelfde oefening, wat het onderhouden van de oefeningen moeilijker 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 constatacie kan gemaakt worden bij meer complexe oefeningen die zich concentreren op algoritmen: ook daar zijn de concepten belangrijker dan in welke programmeertaal een algoritme

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

Codefragment 1.1. De opgave van de voorbeeldoefening, Lotto.

De **lotto** is een vorm van loterij die voornamelijk bekend is vanwege de genummerde balletjes, waarvan er een aantal getrokken worden. Deelnemers mogen zelf hun eigen nummers aankruisen op een lottoformulier. Hoe groter het aantal overeenkomstige nummers tussen het formulier en de getrokken balletjes, hoe groter de geldprijs.

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

```
1 > loterij()  
2 '2 - 17 - 22 - 27 - 35 - 40'  
3 > loterij(aantal=8)  
4 '5 - 13 - 15 - 31 - 34 - 36 - 39 - 40'  
5 > loterij(aantal=4, maximum=38)  
6 '16 - 20 - 35 - 37'
```

Codefragment 1.2. Oplossing in Python voor de voorbeeldoefening Lotto.

```
1 from random import randint  
2  
3  
4 def loterij(aantal=6, maximum=42):  
5     getallen = set()  
6     while len(getallen) < aantal:  
7         getallen.add(randint(1, maximum))  
8  
9     return " - ".join(str(x) for x in sorted(getallen))
```

Codefragment 1.3. Oplossing in Java voor de voorbeeldoefening Lotto.

```
1  import java.util.HashSet;
2  import java.util.Set;
3  import java.util.concurrent.ThreadLocalRandom;
4  import java.util.stream.Collectors;
5
6  class Submission {
7
8      public static String loterij(int aantal, int maximum) {
9          var r = ThreadLocalRandom.current();
10         var result = new HashSet<Integer>();
11         while (result.size() < aantal) {
12             result.add(r.nextInt(1, maximum + 1));
13         }
14         return result.stream()
15             .sorted()
16             .map(Object::toString)
17             .collect(Collectors.joining(" - "));
18     }
19
20     public static String loterij(int aantal) {
21         return loterij(aantal, 42);
22     }
23
24     public static String loterij() {
25         return loterij(6, 42);
26     }
27 }
```

uiteindelijk geïmplementeerd wordt. Een voorbeeld hiervan is het vak „Algoritmen en Datastructuren”, dat gegeven wordt door prof. Fack binnen de opleiding wiskunde⁴. 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 programmeertaalafhankelijk zijn, moeten in elke judge opnieuw geïmplementeerd worden. Hierbij denken we aan bijvoorbeeld de logica om te bepalen wanneer een beoordeling positief of negatief moet zijn.

1.5.3. Onderzoeksvraag

Het eerste nadeel wordt beschouwd als het belangrijkste nadeel en de focus van deze thesis. Het nadeel valt te formuleren als de onderzoeksvraag waarop deze thesis 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 (codefragment 1.1) programmeertaalspecifieke en taalspecifieke elementen bevat. Zo zijn de voorbeelden in Python en zijn de namen van functies en argumenten in het Nederlands. Het ondersteunen van opgaves met programmeertaalafhankelijke voorbeelden en vertalingen wordt voor deze thesis expliciet als *out-of-scope* gezien en zal niet behandeld worden, zij het in paragraaf 5.4.1.

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

1.6. Opbouw van de thesis

Hoofdstuk 2 handelt over het antwoord op bovenstaande onderzoeksvraag, waar een prototype van een dergelijke judge wordt voorgesteld. Daarna volgt ter illustratie een gedetailleerde beschrijving van hoe een programmeertaal geconfigureerd moet worden voor dit prototype in hoofdstuk 3. In hoofdstuk 4 worden een reeks oefeningen met verschillende functionaliteiten besproken om te illustreren dat het prototype flexibel is. Daar deze twee hoofdstukken voornamelijk ten doel hebben zij die met het prototype (zoals lesgevers) moeten werken te informeren, nemen deze hoofdstukken de vorm aan van meer traditionele softwarehandleidingen, zoals meer codefragmenten. Tot slot volgt met een hoofdstuk over beperkingen van de 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 en de 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 gezoken aanpak in kaart te brengen. Deze judge heeft de naam *TESTed* gekregen. Bij *TESTed* is een oefening programmeertaalafhankelijk en kunnen oplossingen in verschillende programmeertalen beoordeeld worden aan de hand van een en dezelfde specificatie. Dit hoofdstuk begint met het ontwerp en de algemene werking van de judge toe te lichten, waarna elk onderdeel in meer detail besproken wordt.

2.1. Overzicht

2.1.1. 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 ook paragraaf 2.8): 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.

2.1.2. Stappenplan van een beoordeling

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

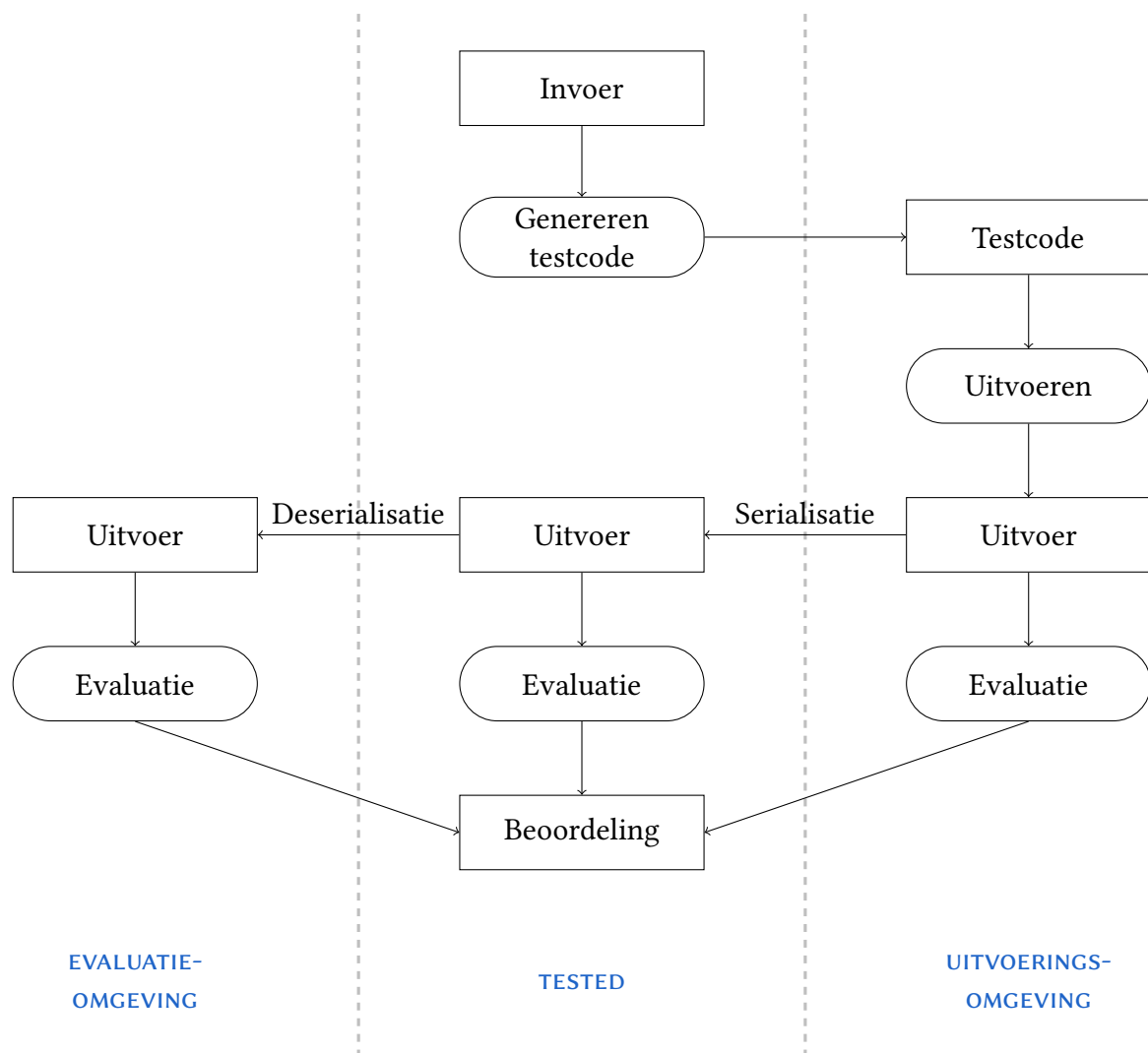
1. De Docker-container voor *TESTed* wordt gestart. Dodona stelt de invoer ter beschikking aan de container: het testplan komt uit de oefening, terwijl de ingediende oplossing en de configuratie uit Dodona komen.
2. Als eerste stap wordt gecontroleerd dat het testplan de programmeertaal van de ingediende oplossing ondersteunt (zie paragraaf 2.2.4). De programmeertaal van de oplossing wordt gegeven via de configuratie uit Dodona. Merk op dat de ingediende oplossing zelf hierbij 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.

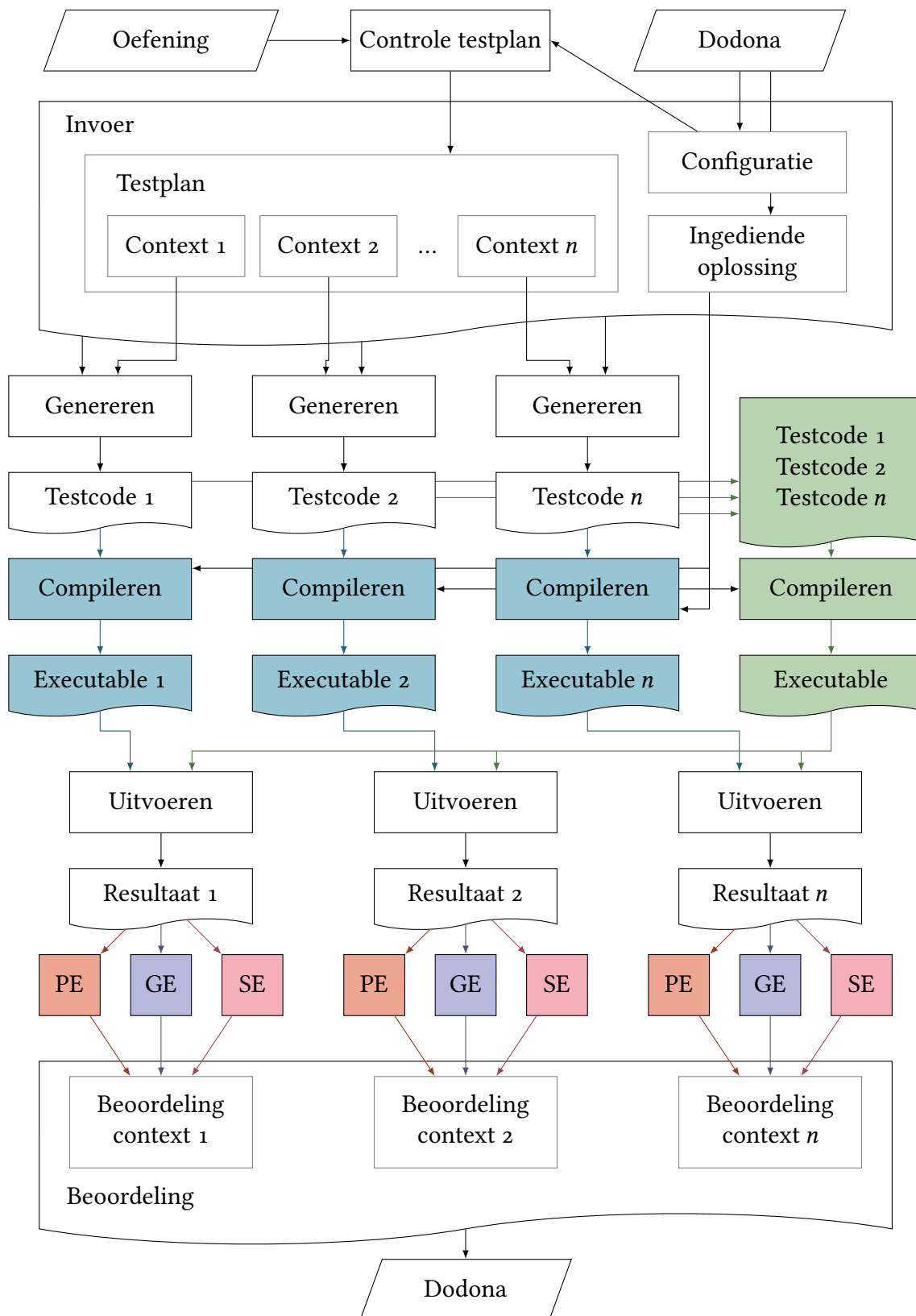
3. Het testplan (zie paragraaf 2.2.1) bestaat uit verschillende contexten. Elke context is een onafhankelijke uitvoering van de ingediende oplossing en kan verschillende aspecten van die uitvoering beoordelen. Voor elk van die contexten wordt in deze stap de testcode gegenereerd. Deze stap is de overgang naar de *uitvoeringsomgeving*.
4. De testcode wordt optioneel gecompileerd, en dit kan op twee manieren (details in paragraaf 2.3.1):
 - a) Batchcompilatie. Hierbij wordt de testcode van alle contexten verzameld 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.
 - b) Contextcompilatie. Hier wordt de testcode voor elke context afzonderlijk gecompileerd. Bij deze manier worden er n executables geproduceerd tijdens de compilatiestap.

In talen die geen compilatie nodig hebben of ondersteunen, 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), zoals de standaard-uitvoerstream, de standaardfoutstream, returnwaardes, exceptions of exitcodes. Deze bundel resultaten wordt nu geëvalueerd op juistheid. Hiervoor zijn drie mogelijke manieren:
 - a) 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.
 - b) 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 ter uitvoering van de geprogrammeerde evaluatiecode wordt gegenereerd en dan uitgevoerd. Het doel van deze modus is om complexe evaluaties toe te laten op een programmeertaalafhankelijke manier. Deze stap vindt plaats in de evaluatieomgeving.
 - c) 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. 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.

2.2. Beschrijven van een oefening

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

2.2.1. 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 oplossing in Dodona en van 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. Concreet is de structuur een hiërarchie met volgende elementen:

Plan Het top-level object van het testplan. Dit object bevat enkel de tabbladen.

Tab Een testplan bestaat uit verschillende *tabs* 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 *contexten*. 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 in een eigen map uitgevoerd, zodat de kans op het delen van informatie klein 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 bv. in een andere locatie een bestand aan te maken en later te lezen. Deze onafhankelijkheid laat ook toe om contexten in parallel te beoordelen, waarover meer in paragraaf [2.5.3](#).

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

Main testcase of hoofdtestgeval. Van deze soort is er maximaal één per context (geen hoofdtestgeval 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 en de programma-argumenten meegegeven worden. De exitcode van een uitvoering kan ook enkel in het hoofdtestgeval 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 het standaardinvoerkanal, functieoproepen en variabeletoekenningen. Een functieoproep of variabeletoekenning is verplicht (zonder functieoproep of toekenning aan een variabele is er geen code om te testen).

Het hoofdtestgeval 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 hoofdtestgeval altijd eerst uitgevoerd.

Test De beoordeling van een testgeval bestaat uit meerdere *tests*, die elk één aspect van het testgeval controleren. Met aspect bedoelen we de standaarduitvoerstream, de standaardfoutstream, 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 hoofdtestgeval. Het beoordelen van de verschillende aspecten wordt in meer detail beschreven in paragraaf 2.4.

Bij de keuze voor 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
- programmeertaalonaafhankelijk 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 thesis 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.

2.2.2. Dataserialisatie

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

```

1 {
2   "tabs": [
3     {
4       "name": "Feedback",
5       "contexts": [
6         {
7           "input": {
8             "function": {
9               "type": "function",
10              "name": "loterij",
11              "arguments": [
12                { "type": "integer", "data": 6 },
13                { "type": "integer", "data": 15 }
14              ]
15            }
16          },
17          "output": {
18            "result": {
19              "value": {
20                "type": "text",
21                "data": "1 - 6 - 7 - 11 - 13 - 14"
22              },
23              "evaluator": {
24                "type": "custom",
25                "language": "python",
26                "path": "./evaluator.py",
27                "arguments": [
28                  { "type": "integer", "data": 6 },
29                  { "type": "integer", "data": 15 }
30                ]
31              }
32            }
33          }
34        ]
35      }
36    ]
37  }
38 }

```

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

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, en
- mogelijk maken om de basisgegevenstypes ondersteunen die we willen aanbieden in het programmeertaalafhankelijke 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 [\[10\]](#). 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 encodings 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 zijn omslachtig in gebruik. Vaak ondersteunen dit soort formaten meer dan wat wij nodig hebben.
- 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 de judge 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 [2].

Apache Parquet Minder relevant, dit is een bestandsformaat voor Hadoop [3].

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 [15].

Bencode Schema gebruikt in BitTorrent. Het is gedeeltelijk binair, gedeeltelijk in text [8].

Binn Binair dataformaat [30].

BSON Een binaire variant op JSON, geschreven voor en door MongoDB [7].

CBOR Een lichtjes op JSON gebaseerd formaat, ook binair. Heeft een goede standaard, ondersteunt redelijk wat talen [6].

FlatBuffers Lijkt op ProtocolBuffers, allebei geschreven door Google, maar verschilt wat in implementatie van ProtocolBuffers. De encoding is binair [25].

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 [16].

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. [1].

MessagePack Nog een binair formaat dat lichtjes op JSON gebaseerd is. Lijkt qua types sterk op JSON. Heeft implementaties in veel talen [13].

OGDL Afkorting voor *Ordered Graph Data Language*. Daar het om een serialisatieformaat voor grafen gaat, is het niet nuttig voor ons doel [24].

opc Unified Architecture Een protocol voor intermachinecommunicatie. Complex: de specificatie bevat 14 documenten, met ongeveer 1250 pagina's [26].

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 [21].

ProtocolBuffers Lijkt zoals vermeld sterk op FlatBuffers, maar heeft nog extra stappen nodig bij het encoderen en decoderen, wat het minder geschikt maakt [28].

Smile Nog een binaire variant van JSON [17].

SOAP Afkorting voor *Simple Object Access Protocol*. Niet bedoeld als formaat voor dataserialisatie, maar voor communicatie tussen systemen over een netwerk [23].

SDXF Binair formaat voor data-uitwisseling. Weinig talen ondersteunen dit formaat [34].

Thrift Lijkt sterk op ProtocolBuffers, maar geschreven door Facebook [31].

UJSON Nog een binaire variant van JSON [32].

```

1 {
2   "type": "sequence",
3   "data": [
4     { "type": "integer", "data": 5 },
5     { "type": "integer", "data": 15 }
6   ]
7 }

```

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

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. Een concreet voorbeeld is codefragment 2.2.

Het encoderen van waarden slaat op het voorstellen van waarden als JSON-waarden. 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 het onder andere de structuur van een waarde in het serialisatieformaat, in een vereenvoudigde versie van JSON Schema. Hierbij staat <types> voor een van de gegevenstypes die hierna besproken werden.

Gevenstypes

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.

```

1 {
2   "Value": {
3     "type": "object",
4     "properties": {
5       "data": {"type": ["number", "string", "boolean", "object", "array", "null"]},
6       "type": {"type": "string", "enum": ["<types>"]}
7     }
8   },
9   "FunctionCall": {
10    "type": "object",
11    "properties": {
12      "type": {
13        "type": "string",
14        "enum": ["function", "namespace", "constructor", "property"]
15      },
16      "namespace": {"type": "string"},
17      "name": {"type": "string"},
18      "arguments": {
19        "type": "array",
20        "items": {"type": "#Expression"}
21      }
22    }
23  },
24  "Identifier": {"type": "string"},
25  "Expression": {
26    "anyOf": [{"$ref": "#Identifier"}, {"$ref": "#Value"}, {"$ref":
    ↳ "#FunctionCall"}]
27  },
28  "Assignment": {
29    "type": "object",
30    "properties": {
31      "name": {"type": "string"},
32      "expression": {"type": "#Expression"},
33      "type": "<datatype>"
34    }
35  },
36  "Statement": {
37    "anyOf": [{"$ref": "#Assignment"}, {"$ref": "#Expression"}]
38  }
39 }

```

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

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

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

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 dat 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 is 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 talen die bijvoorbeeld geen `tuple` uit Python ondersteunen, zal het type omgezet worden naar `list`. Er is ook de mogelijkheid 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 encoding van waarden. In de praktijk gaat het 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).

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.

2.2.3. 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. Bijvoorbeeld zou een opgave 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 hieronder geïllustreerd in het fictieve voorbeeld van codefragment 2.4.

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

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

¹ Uiteraard met de gebruikelijke aliasen van `short`, `unsigned`, ...

Codefragment 2.4. Fictief voorbeeld van wat nodig is bij evaluatie van objectgerichte programmeertalen.

```
1 var variabele = new DoorDeStudentGemaakteKlasse();
2 assert variabele.testfunctie1() = 15;
3 assert variabele.testfunctie2() = "Vijftienduizend";
4 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 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 dezer drie dingen:

1. Een waarde, zoals hiervoor besproken in subparagraaf *Dataschema* van paragraaf 2.2.2.
2. Een identifieer, voorgesteld als een string.
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 Deze 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. Deze soort functie heeft dezelfde semantiek van 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 expressies.

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 de 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.

Codefragment 2.5. Toekennen van de string 'Dodona' aan een variabele.

```
1 {  
2   "variable": "name",  
3   "expression": {  
4     "type": "text",  
5     "value": "Dodona"  
6   },  
7   "type": "text"  
8 }
```

Codefragment 2.6. Geavanceerde constructies die voorgesteld kunnen worden in het testplan.

```
1 var variabele1 = functie1();  
2 var variabele2 = [variabele1, 15, functie2()];  
3 var object = new Test();  
4  
5 assert object.process(variabele2, [functie1()]) = 85;
```

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. Een gecombineerd voorbeeld staat in codefragment 2.5. Hier wordt de string 'Dodona' toegekend aan een variabele met naam name.

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 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 2.6 bevat een voorbeeld van geavanceerde constructies die voorgesteld kunnen worden in het testplan.

2.2.4. Controle ondersteuning voor programmeertalen

In het stappenplan uit paragraaf 2.1 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 4):

```
1 >>> is_isbn("9789027439642")
2 True
3 >>> is_isbn(9789027439642)
4 False
```

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

2.3. 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 zal uitvoeren.

2.3.1. Testcode genereren

Het genereren van de testcode gebeurt met een sjabloonsysteem genaamd Mako [5]. 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. Ook zijn de sjablonen verantwoordelijk voor het genereren van de code die de oplossing van de student zal oproepen en evalueren.

Sjablonen

`TESTED` heeft een aantal standaardsjablonen nodig, 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 uit een context uit te voeren.

selector* Leest een parameter in en voert aan de hand van die parameter de juiste context uit bij de batchcompilatie (zie de volgende subparagraaf en paragraaf 2.5). Dit sjabloon is niet in elke programmeertaal nodig: in Python kunnen de contexten samen gecompileerd worden zonder selector, terwijl dit in Java en Haskell niet mogelijk is.

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.

In de meeste talen zullen ook nog andere sjablonen nodig zijn, die dan gebruikt worden door de verplichte sjablonen. Een voorbeeld is het omzetten van functieoproepen uit het testplan naar code.

Tot slot moet ook het serialisatieformaat geïmplementeerd worden in de taal, en dit in beide richtingen: waarden encoderen in het serialisatieformaat en waarden uit het serialisatieformaat omzetten naar de types van de programmeertaal (het decoderen). Hoe het encoderen van waarden moet gebeuren is vrij, maar in alle bestaande programmeertalen wordt geïmplementeerd met een bijkomende klasse of module (die vaak de naam `Value` krijgt). `TESTED` zelf gaat nooit waarden decoderen, maar dit is toch nodig, bijvoorbeeld binnen de sjablonen voor expressies of statements. In alle bestaande talen is dit geïmplementeerd met bijkomende sjablonen. Dit wordt geïllustreerd in hoofdstuk 3, dat het toevoegen van een nieuwe programmeertaal aan `TESTED` volledig uitwerkt.

Testcode compileren

`TESTED` ondersteunt twee manieren waarop de code gecompileerd kan 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 parameter de juiste context uitgevoerd (met het `selector`-sjabloon).

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

Dit wordt getoond in figuur 2.2 (uit paragraaf 2.1) door twee kleuren te gebruiken: de stappen die enkel plaatsvinden bij batchcompilatie zijn in het **groen**, terwijl stappen die enkel bij contextcompilatie plaatsvinden in het **blauw** staan. Stappen die altijd uitgevoerd worden staan in de flowchart in het zwart.

Batchcompilatie resulteert een grote tijdswinst (zoals beschreven in paragraaf 2.5), 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 uit een tussenoplossing toe te passen, door bijvoorbeeld compilatie op het niveau van tabbladen proberen.

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

Codefragment 2.7. Mapstructuur na het uitvoeren van de testcode van een oplossing in Python.

```

1  workdir                                //Werkmap van TESTed
2  └─ common                             //Gemeenschappelijke code
3  │   └─ context_0_0.py                 //Broncode voor context 0 van tabblad 0
4  │   └─ context_0_0.pyc                //Gecompileerde code voor context 0 van tabblad 0
5  │   └─ context_0_1.py
6  │   └─ context_0_1.pyc
7  │   ...
8  │   └─ context_0_49.py
9  │   └─ context_0_49.pyc
10 │   ...
11 │   └─ context_1_49.py                 //Broncode voor context 49 van tabblad 1
12 │   └─ context_1_49.pyc
13 │   └─ submission.py                  //Ingediende oplossing
14 │   └─ submission.pyc
15 │   └─ values.py                       //Values-module
16 │   └─ values.pyc
17 └─ context_0_0                        //Map voor context 0-0
18 │   └─ FaLd6WGRN_exceptions.txt       //Uitzonderingskanaal
19 │   └─ FaLd6WGRN_values.txt           //Kanaal voor returnwaarden
20 │   └─ context_0_0.pyc                //Code voor context 0-0
21 │   └─ submission.pyc
22 │   └─ values.pyc
23  ...

```

Codefragment 2.8. Mapstructuur na het uitvoeren van een geprogrammeerde evaluator.

```

1  temp                                //Tijdelijke map
2  └─ buzzchecker.py                     //Evaluatiecode
3  └─ evaluation_utils.py                //Hulpfuncties
4  └─ evaluator_executor.py              //Code om de evaluatiecode uit te voeren
5  └─ values.py                           //Values-module

```

2.3.2. Testcode uitvoeren

Vervolgens wordt de (gecompileerde) testcode voor elke context uit het testplan afzonderlijk uitgevoerd 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.

Codefragment 2.7 illustreert dit met een voorbeeld voor een ingediende oplossing in de programmeertaal Python. Deze mapstructuur stelt de toestand van de werkmap van TESTED voor na het uitvoeren van de code. 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, bv. `context_0_1`, wat de map is voor context 1 van tabblad 0 van het testplan. De mapstructuur voor een geprogrammeerde evaluatie is codefragment 2.8.

2.3.3. 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 standaarduitvoerstream. Dit wordt verzameld als tekstuele uitvoer.
- De standaardfoutstream. Ook dit wordt als tekst verzameld.
- Fatale uitzonderingen. Hiermee bedoelen we uitzonderingen die tot aan 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 geëncodeerd en ook verzameld in een bestand.
- Exitcode. Het gaat om de exitcode 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 door de ingediende oplossing gemaakte bestanden te bekijken.

De standaarduitvoer- en standaardfoutstream 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 hier verantwoordelijk voor, en schrijft een *separator* naar alle uitvoerkanalen tussen elk testgeval, zoals te zien is in codefragment 2.9.

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

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

Codefragment 2.9. Voorbeeld van het uitvoerkanaal voor returnwaarden na het uitvoeren van de eerste context uit de voorbeeldoefening Lotto.

- 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.

2.4. Oplossingen beoordelen

In de vorige stap heeft `TESTED` de resultaten verzameld. Het geheel van de verzamelde resultaten noemen we de uitvoer van een beoordeling. 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, geschreven door degene die de oefening opgesteld heeft, in een aparte omgeving (de evaluatieomgeving).
3. Programmeertaalspecifieke evaluatie: de uitvoer wordt onmiddellijk na het uitvoeren van de testcode beoordeeld in het hetzelfde proces.

2.4.1. Generieke evaluatie

Voor eenvoudige beoordelingen (bijvoorbeeld `stdout` vergelijken met een verwachte tekst) volstaat de generieke evaluatie binnen `TESTED`. Het is mogelijk om de verwachte resultaten in het testplan op te nemen. `TESTED` zal deze 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.

Codefragment 2.10. Fragment uit een testplan dat tekstevaluatie gebruikt.

```
1 {
2   "output": {
3     "stdout": {
4       "type": "text",
5       "data": "3.14",
6       "evaluator": {
7         "type": "builtin",
8         "name": "text",
9         "options": {"ignoreWhitespace": true, "tryFloatingPoint": true,
10                    ↪ "applyRounding": true, "roundTo": 2}
11       }
12     }
13   }
```

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. Deze functie controleert of twee zwevendekommagetallen „dicht bij elkaar” liggen. De standaardfoutmarges van Python worden gebruikt. Een punt voor de toekomst is het configureerbaar maken van deze foutmarges.

applyRounding Of zwevendekommagetallen afrond 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 2.10 toont een fragment uit een testplan: de uitvoerspecificatie van een testgeval waarbij de tekstevaluatie gebruikt wordt.

Bestandsevaluatie

In deze evaluatievorm worden twee bestanden vergeleken met elkaar. Hiervoor bevat het testplan enerzijds een pad naar een bestand die met de oefening gegeven 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 dezer manieren:

exact Beide bestanden moet exact hetzelfde zijn, inclusief geregeleindes.

lines Elke regel wordt vergeleken met overeenkomstige regel in het andere bestand. De evaluatie van de lijnen is exact, maar zonder de geregeleindes. Dit betekent dat de witruimte bijvoorbeeld ook moet overeenkomen.

¹Documentatie is hier te vinden: <https://docs.python.org/3/library/math.html#math.isclose>

Codefragment 2.11. Fragment uit een testplan dat bestandsevaluatie gebruikt.

```
1 {
2   "output": {
3     "file": {
4       "expected_path": "./bestand-uit-de-oefening.txt",
5       "actual_path": "./waar-het-verwachte-bestand-komt.txt",
6       "evaluator": {
7         "type": "builtin",
8         "name": "file",
9         "options": {"mode": "values", "ignoreWhitespace": true, "tryFloatingPoint":
10           ↪ true, "applyRounding": true, "roundTo": 2}
11       }
12     }
13   }
```

values Elke regel in het bestand wordt afzonderlijk vergeleken met de tekstevaluatie. Indien deze modus gebruikt wordt, kunnen ook alle opties van de tekstevaluatie meegegeven worden.

Een voorbeeld van hoe dit eruitziet is codefragment 2.11. In dit fragment wordt de modus `values` 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 (zie codefragment 2.7 voor een overzicht van de structuur).

Waarde-evaluatie

Voor uitvoerkanalen zoals de returnwaarden moet meer dan alleen tekst met elkaar vergeleken kunnen worden. Staat er in het testplan welke waarde verwacht wordt (geëncodeerd in het serialisatieformaat), dan kan `TESTED` dit vergelijken met de eigenlijke waarde die geproduceerd werd door de ingediende oplossing.

Het vergelijken van een waarde bestaat uit twee stappen:

1. Het gegevenstype wordt vergeleken, waarbij beide waarden (de verwachte waarde uit het testplan en de geproduceerde waarde uit de ingediende oplossing) hetzelfde type moeten hebben. Hierbij wordt rekening gehouden met de vertalingen tussen de verschillende programmeertalen, waarbij twee gevallen onderscheiden kunnen worden:
 - a) Specificeert 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) Specificeert 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.

2. De twee waarden worden vergeleken op inhoud (indien de vergelijking van de gegevenstypes uit de vorige stap positief is). Hierbij maakt `TESTED` gebruik van de ingebouwde vergelijking van Python om twee waarden te evalueren. Dit betekent dat de regels voor *value comparisons* uit Python² gevolgd worden. Eén uitzondering is zwevendekommagetallen, waarvoor opnieuw `math.isclose()` gebruikt wordt in plaats van `=`.

Bij deze evaluatievorm zijn geen configuratieopties. Een voorbeeld van het gebruik binnen een testplan is codefragment 2.12. Hier wordt als returnwaarde een verzameling met drie elementen (5, 10 en 15) verwacht.

Codefragment 2.12. Fragment uit een testplan dat waarde-evaluatie gebruikt.

```
1 {
2   "output": {
3     "return": {
4       "value": {
5         "type": "set",
6         "data": [
7           { "type": "integer", "data": 5 },
8           { "type": "integer", "data": 10 },
9           { "type": "integer", "data": 15 }
10        ]
11      },
12      "evaluator": {"type": "builtin", "name": "value"}
13    }
14  }
15 }
```

2.4.2. 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 bio-informatica, volstaat een vergelijking met een verwachte waarde uit het testplan niet.

Toch is deze evaluatie niet 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

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

geschreven zijn. Binnen `TESTED` wordt dit mogelijk gemaakt door geproduceerde waarden uit 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 functie³ met als naam `evaluate`. Deze functie krijgt drie argumenten mee:

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 gegeven worden als returnwaarde. Hoe deze returnwaarde er exact uit ziet, is opnieuw programmeertaalafhankelijk. In Python en Java gaat het om een klasse, terwijl het in Haskell om een record gaat. Deze returnwaarde moet vier gegevens bevatten:

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

readableExpected (Optioneel) Een `string` met een 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 (Optioneel) Een `string` met een tekstuele voorstelling van de geproduceerde waarde. Indien niets gegeven wordt, zal `TESTED` zelf een voorstelling maken op basis van de geproduceerde waarde.

messages (Optioneel) Een lijst van berichten om te tonen aan de studenten.

Een concreet voorbeeld van de evaluatiecode voor een geprogrammeerde evaluatie geschreven in Python wordt gegeven in codefragment 2.13. Deze evaluatiecode is eenvoudig: er wordt gekeken 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.

2.4.3. 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 meer programmeertalen te ondersteunen beperkt zich tot een minimum.
- Het werk om een nieuwe programmeertaal toe te voegen aan `TESTED` is kleiner dan een volledig nieuwe judge te implementeren.

³Deze functie wordt dan vertaald door `TESTED`. In Java zal dit bijvoorbeeld resulteren in een statische functie.

Codefragment 2.13. Voorbeeld van evaluatiecode in Python voor een geprogrammeerde evaluatie.

```
1 from evaluation_utils import EvaluationResult
2
3 def evaluate(expected, actual, arguments):
4     # Controleer of de gekregen waarde overeenkomt.
5     correct = expected == actual
6     messages = []
7     if not correct:
8         messages.append("Hallo, dit is niet juist!")
9     # We geven geen verwachte waarde mee; TESTed neemt de waarde uit het testplan.
10    return EvaluationResult(correct, None, actual, messages)
```

Codefragment 2.14. Voorbeeld van evaluatiecode in Python voor een programmeertaalspecifieke evaluatie.

```
1 from evaluation_utils import EvaluationResult
2
3 def evaluate(actual):
4     correct = isinstance(actual, int)
5     messages = []
6     if not correct:
7         messages.append("Dit is geen int!")
8     return EvaluationResult(correct, str(5), str(actual), messages)
```

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 beoordeelt 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 functie `evaluate` slechts één parameter meekrijgt. Ook moet de evaluatiecode geschreven worden voor elke programmeertaal waarin de oefening aangeboden wordt. Een concreet voorbeeld, opnieuw in Python, is codefragment 2.14. In dit voorbeeld wordt gekeken of de waarde een integer is of niet.

2.5. Performantie

Zoals eerder vermeld (paragraaf 2.3.2), 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.

2.5.1. 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 [19]. 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 die uitvoer 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 programmeertaalafhankelijk protocol bovenop een REPL (een *read-eval-print loop*). Deze keuze voor Jupyter-kernels als uitvoering was gebaseerd op volgende argumenten:

- 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 programmeertaalafhankelijk 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 [27], 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 als 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, 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 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 standaarduitvoer-stroom, wat in bepaalde gevallen problemen gaf bij het verzamelen van de resultaten van de uitvoer van de testcode.

2.5.2. Sjablonen

We kozen ervoor om verder te gaan met een systeem van sjablonen (zie ook paragraaf 2.3.1). Hierbij wordt de code gegenereerd en vervolgens uitgevoerd 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 toe in hoe de resultaten (gedrag en neveneffecten) van het uitvoeren van de testcode verzameld worden.
- Het implementeren en configureren van een programmeertaal in TESTED met de basisfunctionaliteit 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 hanteert echter het designprincipe dat optionele functionaliteit geen bijkomend werk mag vragen voor programmeertalen die er geen gebruik van maken.
- Er is 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.
- Herbruikbaar in die zin dat het genereren van testcode in een programmeertaal op basis van de programmeertaalafhankelijke 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 kernels⁴, goed voor ondersteuning voor meer dan tachtig programmeertalen. Niet elke kernel is onmiddellijk bruikbaar, maar er kan verder gewerkt worden op wat bestaat.

⁴Zie deze pagina voor een actuele lijst van kernels: <https://github.com/jupyter/jupyter/wiki/Jupyter-kernels>.

- Het uitvoeren op de commandoregel is bij veel programmeertalen trager indien er geen reset moet gebeuren tussen de verschillende uitvoeringen. Bij het uitvoeren op de commandoregel is er bij veel programmeertalen een performantiekost bij het opstarten van de interpreter (zoals Python) of virtuele machine (zoals Java). Bij een kernel is de opstartkost weliswaar groter, maar deze wordt maar één keer opgestart als er geen reset nodig is.

Bij een eerste iteratie van het systeem met sjablonen bestond enkel de **contextcompilatie** (paragraaf 2.3.1). 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 2.3.1.

2.5.3. Parallele uitvoering van contexten

Een ander aspect is de ondersteuning voor parallele 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 kleiner waarschijnlijk 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 er verschillende beperkingen of opmerkingen zijn in de implementatie:

- De parallele 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.

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.

Oefening	Compilatiemodus	Python (s)		Java (s)	
		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

- De tijdslimieten zullen minder nauwkeurig zijn. Door de parallelle uitvoering zal er meer variatie zitten op de totale tijd die nodig is voor het beoordelen van een oefening. Dit heeft geen effect op de tijdslimieten bij hun functie voor het detecteren van programmeerfouten, zoals oneindige lussen. Waar er wel moet opgelet worden is indien de tijdslimieten gebruikt worden om te controleren dat een oplossing efficiënt is. Door de parallelle uitvoering is het mogelijk dat een oplossing die normaal nooit onder de tijdslimiet zou zitten, nu wel aanvaard wordt.

2.5.4. Snellere geprogrammeerde evaluatie

Een laatste performantieverbetering werd behaald op het vlak van geprogrammeerde evaluatie (zie paragraaf 5.3.2 voor verdere ideeën voor verbeteringen). Zoals beschreven in paragraaf 2.4.2, wordt normaliter voor elke geprogrammeerde evaluatie de evaluatiecode opnieuw gecompileerd en uitgevoerd in een nieuw subprocess. 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.

2.5.5. 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 elks één testgeval bevatten.
- Een eenvoudigere oefening: Echo. Bij deze oefening bestaat de opgave er uit de invoer van de standaardinvoerstream te lezen en te printen naar de standaarduitvoerstream. 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).

2.6. 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 3 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. Ook geeft dit veel vrijheid: 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. 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 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 uitbreiding 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.

2.7. 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 bevat.

2.7.1. 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 ISBN-oefening. Om een idee te krijgen van hoe TESTED werkt in de praktijk 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 2.5) 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 subprocess. 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 2.6.
- TESTED heeft geen ondersteuning voor vertalingen op het vlak van natuurlijke talen. Deze functie wordt besproken als uitbreiding in paragraaf 5.2.5.

Verder heeft professor Dawyndt veel fouten en scenario's uitgeprobeerd. Deze worden hieronder besproken.

2.7.2. 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 „syntaxfout 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. TESTED beperkt tekstuele uitvoer tot tienduizend tekens. Dit is niet van toepassing op andere vormen van uitvoer: het inkorten van grote verzamelingen (bijvoorbeeld een lijst van duizend elementen) is opgenomen als uitbreiding in paragraaf 5.5.

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, zal de oefening juist gerekend worden.

2.8. Implementatiekeuzes

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

TESTED is geschreven in Python. De hoofdreden daarvoor is pragmatisch: zoals besproken in paragraaf 2.5.1, 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.

Het grootste nadeel aan Python is dat het niet de snelste programmeertaal is op het gebied van uitvoeringstijd. Dit is voor TESTED echter 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 2.2.1 en 2.2.2) 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 pydantic⁵. 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 2.15 toont hoe de definitie van het testplan en het serialisatieformaat er uitziet. In codefragment 2.16 staat dan de JSON-variant van dezelfde component: het moge duidelijk zijn dat de omzetting van Python naar JSON voor de hand ligt. Het JSON Schema voor respectievelijk het testplan en het serialisatieformaat kunnen zo verkregen worden:

```
1 > python -m tested.testplan
2 <JSON Schema voor testplan>
3 > python -m tested.serialisation
4 <JSON Schema voor serialisatieformaat>
```

⁵Zie <https://pydantic-docs.helpmanual.io/>

Codefragment 2.15. Een voorbeeld van hoe het testplan met pydantic gedefinieerd wordt. Hier wordt een geprogrammeerde evaluator voorgesteld in Python-code.

```
1 @dataclass
2 class ProgrammedEvaluator:
3     language: str
4     function: EvaluationFunction
5     arguments: List[Value] = field(default_factory=list)
6     type: Literal['programmed'] = 'programmed'
```

Codefragment 2.16. Hoe een geprogrammeerde evaluator er uitziet in JSON.

```
1 {
2     "type": "programmed",
3     "function": {
4         "file": "Evaluator.hs",
5         "name": "evaluate_value"
6     },
7     "language": "haskell",
8     "arguments": [
9         { "type": "text", "data": "arg1" }
10    ]
11 }
```

3. Configuratie van een programmeertaal in TESTed

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 dicht 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 A en B 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 van het originele bestand.

3.1. 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.

3.1.1. De broncode

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

```
1 universal-judge
2 |— docker/      # Een Docker-image om TESTed uit te voeren.
3 |— exercise/    # De map met oefeningen die dienen als voorbeeld en voor de tests.
4 |— judge/       # TESTed zelf.
5 |   |— src/      # Broncode van TESTed.
6 |   |— tests/    # Testcode van TESTed.
7 |   └─ workdir/  # De uitvoer van TESTed bij manuele uitvoer.
8 |— thesis/      # De thesistekst, dus deze tekst.
9 |— config.json  # Configuratiebestand voor gebruik in Dodona.
10 └─ 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.

3.1.2. 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:

```
1 > 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

3.1.3. 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`.

```
1 > python -m tested
```

¹Gebruikers 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:

```
1 > 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 werkmapij van de judge de map `workdir` zal zijn. Dit laat toe om de gegenereerde code te inspecteren.

3.2. 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:

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

Merk op dat dit enkel bestanden genereert. De stappen in paragraaf 3.3.6 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.

3.3. De programmeertaal C

Voor we beginnen aan de configuratie, overlopen we kort welke functionaliteit we langs beide kanten 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 return-waarde 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.

3.3.1. 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:

```
1 universal-judge
2 |─ judge/
3 |   |─ src/
4 |       |─ tested/
5 |           |─ languages/
6 |               |─ c/           ← nieuwe map
7 |               |─ haskell/
8 |               |─ java/
9 |               |─ python/
10 |              |─ config.py
11 |              ...
12 |          ...
13 |      ...
14 |  ...
```

3.3.2. 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 [2.3.1](#) 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.

```
2  "general": {
3    "dependencies": [
4      "values.h",
5      "values.c",
6      "evaluation_result.h",
7      "evaluation_result.c"
8    ],
9    "selector": true
10 },
11 "extensions": {
12   "file": "c",
13   "templates": ["c", "mako"]
14 },
```

Codestijl

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

```
15 "naming_conventions": {  
16     "namespace": "snake_case",  
17     "function": "snake_case"  
18 },
```

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 dit 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:

```
19 "constructs": {  
20     "objects": false,  
21     "exceptions": false,  
22     "function_calls": true,  
23     "assignments": true,  
24     "heterogeneous_collections": false,  
25     "heterogeneous_arguments": false,  
26     "evaluation": false  
27 },
```

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 assignment. 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 verzamelingen 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.

Dan moeten we nu de ondersteuning voor de gegevenstypes vastleggen:

```
28 "datatypes": {
29     "integer": "supported",
30     "rational": "supported",
31     "char": "supported",
32     "text": "supported",
33     "boolean": "supported",
34     "sequence": "unsupported",
35     "set": "unsupported",
36     "map": "unsupported",
37     "nothing": "supported",
38     "int8": "supported",
39     "uint8": "supported",
40     "int16": "supported",
41     "uint16": "supported",
42     "int32": "supported",
43     "uint32": "supported",
44     "int64": "supported",
45     "uint64": "supported",
46     "bigint": "reduced",
47     "single_precision": "supported",
48     "double_precision": "supported",
49     "double_extended": "unsupported",
50     "fixed_precision": "unsupported",
51     "array": "unsupported",
52     "list": "unsupported",
```

```
53     "tuple": "unsupported"
54 }
```

Zoals uitgelegd in paragraaf 2.2.2 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.

3.3.3. 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 `Language` uitbreidt:

```
10 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:

```

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

```

Het eerste argument 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.

Het andere argument van deze methode is een lijst van bestanden mee 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 zal plaatsvinden. 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 deze parameter er bijvoorbeeld als volgt uit:

```

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

```

Als returnwaarde moet deze methode een tuple met twee elementen 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:

```
1 >>> compilation(['submission.c', 'evaluation_result.c', 'context_0_0.c',  
2   ↪ 'selector.c'])  
3 (  
4     ['gcc', '-std=c11', '-Wall', 'evaluation_result.c', 'values.c', 'selector.c',  
5     '-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:

```
19 def execution(self, config: Config,  
20     cwd: Path, file: str, arguments: List[str]) → Command:  
21     local_file = cwd / executable_name(Path(file).stem)  
22     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:


```

1 >>> execution('/test/path', 'executable.exe', ['arg1', 'arg2'])
2 ['/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 heeft, voegen we die ook toe.

Vertaald naar de configuratieklasse wordt dit:

```

25 def solution(self, solution: Path, bundle: Bundle):
26     with open(solution, "r") as file:
27         contents = file.read()
28         # We use regex to find the main function.
29         # First, check if we have a no-arg main function.
30         # If so, replace it with a renamed main function that does have args.
31         no_args = re.compile(r"(int|void)\s+main\s*(\s*\)\s*{")
32         replacement = "int solution_main(int argc, char** argv){"
33         contents, nr = re.subn(no_args, replacement, contents, count=1)
34         if nr == 0:
35             # There was no main function without arguments. Now we try a main
36             # function with arguments.
37             with_args = re.compile(r"(int|void)\s+main\s*(\s*\s*int")
38             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)

```

3.3.4. Sjablonen

De derde stap bestaat uit het schrijven van de sjablonen. We hebben uiteraard de verplichte sjablonen nodig (zie paragraaf 2.5.2 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 de contextcode

selector.c het sjabloon voor de selector voor batchcompilatie

declaration.mako vertaalt het type 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 waarde naar code

value_arguments.mako hulpsjabloon voor **value.mako**

value_basic.mako hulpsjabloon voor **value.mako**

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. Voorbeeld 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.

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.

```

3  #include <stdio.h>
4
5  #include "values.h"
6  #include "${submission_name}.c"
7
8  % for name in evaluator_names:
9      #include "${name}.c"
10
11 % endfor

```

Nuttig om te 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:

```

1  #include "context_0_0.c"
2  #include "context_0_1.c"

```

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 moet de conventies 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 [A.4.2](#) staat bijvoorbeeld de gegenereerde code in Java, waar de `exception`-functies wel gebruikt worden. Ook gebruiken we in C een marco 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(${context_name}_value_file, value)
27
28 #undef send_specific_value
29 #define send_specific_value(value) write_evaluated(${context_name}_value_file,
30 ↪ 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 in 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.

Ook roepen we 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 geïmplementeerd worden.

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 3.3.3). Als de ingediende oplossing geen main-functie bevatte, maar het testplan verwachtte die wel, dan zal de compilatie falen.

```
38     ${before}
39
40     ${context_name}_write_separator();
41
42     % if context_testcase.exists:
43         char* args[] = {\
44             % for argument in ["solution"] + context_testcase.arguments:
45                 "${argument}", \
46             % endfor
47         };
48         solution_main(${len(context_testcase.arguments) + 1}, args);
49     % 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.

```
52     % for testcase in testcases:
53         ${context_name}_write_separator();
54         <include file="statement.mako" args="statement=testcase.input_statement()"
55     ↪ />;
56     % 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.

```

58     ${after}
59
60     fclose(${context_name}_value_file);
61     fclose(${context_name}_exception_file);
62     return 0;
63 }
```

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.

```

65 #ifndef INCLUDED
66 int main() {
67     return ${context_name}();
68 }
69 #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.

```

1  #include <string.h>
2  #include <stdio.h>
3
4  #define INCLUDED true
5
6  % for cont in contexts:
7      #include "${cont}.c"
8  % endfor
9
10 int main(int argc, const char* argv[]) {
11
```

```

12     if (argc < 1) {
13         fprintf(stderr, "No context selected.");
14         return -2;
15     }
16
17     const char* name = argv[1];
18     % for cont in contexts:
19         if (strcmp("${cont}", name) == 0) {
20             return ${cont}();
21         }
22     % endfor
23     fprintf(stderr, "Non-existing selector '%s' selected.", name);
24     return -1;
25 }

```

Het statementsjabloon

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

```

1  ## Convert a statement and/or expression into C code.
2  <%! from tested.utils import get_args %>
3  <%! from tested.serialisation import Value, Identifier, FunctionCall, Assignment %>
4  <%page args="statement,full=False"/>
5  % if isinstance(statement, Identifier):
6      ## If the expression is an identifier, just echo it.
7      ${statement}\
8  % elif isinstance(statement, FunctionCall):
9      ## Delegate to the function template for function calls.
10     <%include file="function.mako" args="function=statement"/>
11 % elif isinstance(statement, get_args(Value)):
12     ## We have a value, delegate to the value template.
13     <%include file="value.mako" args="value=statement" />
14 % else:
15     <% assert isinstance(statement, get_args(Assignment)) %>
16     % if full:
17         <%include file="declaration.mako" args="value=statement.expression" /> \
18     % endif
19     ${statement.variable} = <%include file="statement.mako"
↪     args="statement=statement.expression"/>;
20 % endif

```

De implementatie van dit sjabloon komt conceptueel neer op een grote switch, waarbij we delegeren naar het juiste sjabloon op basis van wat het statement juist is.

Een aspect dat 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:

```
1 int variabele = 5; // met declaration
2 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 type van het 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]`. 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.

```
1 >>> isinstance('string', Expression)
2 TypeError
3 >>> isinstance('string', get_args(Expression))
4 False
5 >>> isinstance('string', get_args(str)) # Ook gewone types
6 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.

3.3.5. 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.

3.3.6. Afsluiting

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`:

```
17 LANGUAGES = {
18     'python':    Python,
19     'java':      Java,
20     'haskell':   Haskell,
21     'runhaskell': RunHaskell,
22     'c':         C
23 }
```

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:

```
1 > 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.

3.3.7. Resultaat

We hebben nu een werkende C-judge voor TESTed en bij uitbreiding voor het Dodona-platform. Daarmee hebben we ondersteuning voor een eerste programmeertaal waarvoor er nog geen judge bestond in Dodona.

3.4. Hoe lang duurt het implementeren van een programmeertaal?

TODO

3.5. Stabiliteit van TESTed

Met stabiliteit wordt hier bedoeld hoe weinig er bij het toevoegen van een nieuwe programmeertaal nog veranderd moet worden aan TESTed zelf, configuraties van bestaande programmeertaal 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 thesis 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. We spreken 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 dan 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 thesis. 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. Oefeningen schrijven voor TESTed

In dit hoofdstuk behandelen we het schrijven van oefeningen voor TESTed. Dit hoofdstuk bevat vooral een beschrijving van welke soort oefeningen ondersteund worden door TESTed en hoe een opgave vertaald moet worden naar een testplan.

4.1. 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 worden meegegeven met de configuratie bij het uitvoeren van TESTed.

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:

```
1 oefening                                # Map van de oefening
2 |— config.json                         # Configuratiebestand van Dodona
3 |— description
4 |   |— description.en.html             # Beschrijving in het Engels
5 |   |— description.nl.html            # Beschrijving in het Nederlands
6 |   |— media                           # Mediabestanden van de beschrijvingen
7 |       |— ISBN.gif
8 |— evaluation                          # Evaluatiecode
9 |   |— plan.json                       # Het testplan
10 |— preparation
11 |   |— generator.py                   # Code die het testplan genereert
12 |— solution                           # Voorbeeldoplossingen voor de oefeningen
13 |   |— solution.java
14 |   |— solution.py
```

Voor meer details en bijkomende informatie (want het voorbeeld hierboven is slechts een basis, er is meer functionaliteit), raden we aan om de relevante handleiding uit de Dodona-documentatie te lezen¹.

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

4.2. Lotto

Als eerste oefening beschouwen we de voorbeeldoefening Lotto uit paragraaf 1.5, die we uitgebreid zullen bespreken. De opgave en voorbeeldoplossingen van deze oefeningen hebben we al vermeld, in respectievelijk codefragment 1.1 en codefragmenten 1.2 en 1.3.

4.2.1. 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 het resultaat vergelijken met een verwachte waarde.

Zoals vermeld in paragraaf 2.2.1 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 ons geval lijkt het logisch om één tabblad te gebruiken: per slot van rekening beoordelen we één functie.

Elke functieoproep is onafhankelijk van elkaar, wat ons 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 volgende vorm aanneemt:

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

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

4.2.2. Evaluatie

Iets dat we in de vorige paragraaf genegeerd hebben is hoe we het vergelijken met een verwachte waarde exact gaan doen. Lottonummers zijn namelijk willekeurig: de pseudocode van hierboven zou dus slechts heel zelden tot een juiste beoordeling leiden. We zullen dit oplossen door een geprogrammeerde evaluatie te gebruiken. Dit is een functie die door `TESTED` zal opgeroepen worden en wier verantwoordelijkheid het vergelijken van de door de ingediende oplossing geproduceerde waarde met een verwachte waarde is. Conceptueel kunnen we dat ook vertalen naar dezelfde pseudocode:

```

1 assert geprogrammeerde_evaluatie(loterij(), [])
2 assert geprogrammeerde_evaluatie(loterij(8), [8])
3 assert geprogrammeerde_evaluatie(loterij(4, 38), [4, 38])
4 # ... enzovoort

```

We zullen de geprogrammeerde evaluatie in Python doen: dit is de aanbevolen programmeertaal voor geprogrammeerde evaluaties, 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.

```

42 def evaluate(expected, actual, arguments):
43     count = arguments[0]
44     maximum = arguments[1]
45     valid, message = valid_lottery_numbers(actual, count, maximum)
46     messages = [Message(message)] if message else []
47     if valid:
48         expected = actual
49     return EvaluationResult(valid, expected, actual, messages)

```

Wat doet deze functie nu juist?

1. Een evaluatiefunctie van een geprogrammeerde evaluatie kan ook argumenten meekrijgen. In ons geval geven we de parameters van de loterij-functie mee als argument aan de evaluatiecode. Eerst worden dus de parameters uit de argumenten gehaald.
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, gebruiken we die waarde 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.

4.2.3. Het testplan

Nu we weten welke structuur we willen en we weten hoe we gaan beoordelen kunnen we een testplan in JSON opstellen. 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. De reden hiervoor is dat deze verwachte waarde getoond zal worden als de geproduceerde waarde verkeerd is. Het is mogelijk om in de geprogrammeerde evaluatie geavanceerde zaken te doen, zoals indien de geproduceerde waarde juist is, maar niet gesorteerd is, de geproduceerde waarde te sorteren en als verwachte waarde te gebruiken.

```

6 {
7   "testcases": [
8     {
9       "output": {
10        "result": {
11          "value": {
12            "type": "text",
13            "data": "7 - 37 - 48 - 54 - 70 - 78 - 81 - 90 - 102 - 103 - 113 - 119 -
↪ 120 - 137 - 140 - 154 - 157 - 159"
14          },
15          "evaluator": {
16            "type": "programmed",
17            "language": "python",
18            "function": {
19              "file": "./evaluator.py"
20            },
21            "arguments": [
22              {
23                "type": "integer",
24                "data": 18
25              },
26              {
27                "type": "integer",
28                "data": 172
29              }
30            ]
31          }
32        },
33        "input": {
34          "type": "function",
35          "name": "loterij",
36          "arguments": [
37            {
38              "type": "integer",
39              "data": 18
40            },
41            {
42              "type": "integer",
43              "data": 172
44            }
45          ]
46        }
47      }
48    ]
49  },
50 }

```

4.3. Echo

Een volgende oefening die we bekijken is de oefening uit appendix A. Deze oefening bestaat uit een invoer lezen van `stdin` en deze invoer schrijven naar `stdout`.

Voor de structuur van de beoordeling geldt grotendeels hetzelfde als bij paragraaf 4.2.1: 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.

Qua evaluatie is deze oefening eenvoudig: we kunnen de ingebouwde evaluatie van `TESTED` gebruiken. De geproduceerde waarde moet vergeleken worden met de verwachte waarde.

Een testplan is al opgenomen in appendix A.2. Dit is een testplan voor twee contexten, maar bij het gebruiken van de oefening zullen uiteraard meer contexten nodig zijn. Ook hier wordt dit testplan gegenereerd door een script en niet met de hand geschreven.

4.4. Echofunctie

Deze oefening is praktisch volledig analoog aan de echo-oefening uit paragraaf 4.3, behalve dat hier een echo-functie moet geïmplementeerd worden. Ook deze oefening is opgenomen als bijlage, in appendix B.

Het testplan is ook te zien in appendix B.2. Ter illustratie bevat dit testplan een context met twee testgevallen. In het eigenlijke testplan dat gebruikt zou worden voor deze oefening zal elk testgeval ook afzonderlijk in een context geplaatst worden.

4.5. ZeroDivisionError

Een interessante oefening is de oefening *ZeroDivisionError* uit *De Programmeursleerling*².

In deze oefening moet een programma geschreven worden dat bij het uitvoeren een `ZeroDivisionError` gooit.

4.5.1. Structuur van de beoordeling

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

²Beschikbaar hier: <https://dodona.ugent.be/nl/courses/293/series/2535/activities/270198713/>

4.5.2. Evaluatie

Het feit dat specifiek een `ZeroDivisionError` moet gegoooid worden, zorgt ervoor dat we hier een programmeertaalspecifieke evaluatie zullen moeten gebruiken. In Java gaat het bijvoorbeeld om een `ArithmeticException`, terwijl delen door nul in 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.

```
1 public class Evaluator {
2     public static EvaluationResult evaluate(Object actual) {
3         if (actual instanceof ArithmeticException) {
4             return EvaluationResult.builder(true)
5                 .withReadableExpected(actual.toString())
6                 .withReadableActual(actual.toString())
7                 .build();
8         } else {
9             return EvaluationResult.builder(false)
10                .withReadableExpected("ArithmeticException")
11                .withReadableActual(actual == null ? "" : actual.toString())
12                .withMessage(new EvaluationResult.Message("Expected
13                    ↪ ArithmeticException, got something else."))
14                .build();
15        }
16    }
17 }
```

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.

4.5.3. Testplan

Ook het testplan is een vrij eenvoudige vertaling van de structuur die we hiervoor hebben bedacht:

```
1 {
2     "tabs": [
3         {
4             "name": "Correctheid",
5             "contexts": [
6                 {
7                     "context_testcase": {
8                         "description": "Uitvoeren code",
9                         "input": {
10                             "main_call": true
11                         }
12                     }
13                 }
14             ]
15         }
16     ]
17 }
```

```

11     },
12     "output": {
13         "exception": {
14             "evaluator": {
15                 "type": "specific",
16                 "evaluators": {
17                     "python": {
18                         "file": "evaluator.py"
19                     },
20                     "java": {
21                         "file": "Evaluator.java"
22                     },
23                     "haskell": {
24                         "file": "Evaluator.hs"
25                     },
26                     "runhaskell": {
27                         "file": "Evaluator.hs"
28                     }
29                 }
30             }
31         }
32     }
33 }
34 ]
35 ]
36 }
37 ]
38 }

```

4.6. Som

Deze oefening is opnieuw een oefening uit *De Programmeursleerling*³. Deze oefening is interessant om te illustreren hoe programma-argumenten werken en hoe de exitcode werkt. Ditmaal bestaat de opgave eruit om een reeks getallen in te lezen uit de programma-argumenten en de som ervan uit te schrijven. Bijvoorbeeld:

```

1 > python ./som
2 0
3 > python ./som 1 2 3 4 5 6 7 8 9 10
4 55
5 > python ./som 1 -2 3 -4 5 -6 7 -8 9 -10
6 -5

```

³<https://dodona.ugent.be/nl/courses/293/series/2556/exercises/1653208777/>

De structuur van deze oefening is eenvoudig: een reeks contexten die elk het programma oproepen met verschillende programma-argumenten. Ook de evaluatie is eenvoudig: we gebruiken de ingebouwde evaluatie van TESTED.

In het testplan zijn twee contexten opgenomen: een waarbij getallen gegeven zijn en een waarbij andere argumenten gegeven worden. In een echt testplan zullen er uiteraard meer contexten zijn.

```
1 {
2   "tabs": [
3     {
4       "name": "Feedback",
5       "contexts": [
6         {
7           "context_testcase": {
8             "input": {
9               "arguments": ["-1", "-23", "72", "84", "-38", "-61", "49", "45"],
10              "main_call": true
11            },
12            "output": {
13              "stdout": {
14                "type": "text",
15                "data": "127"
16              }
17            }
18          },
19          {
20            "context_testcase": {
21              "input": {
22                "arguments": [],
23                "main_call": true
24              },
25              "output": {
26                "stdout": {
27                  "type": "text",
28                  "data": "0"
29                }
30              }
31            }
32          }
33        ],
34        {
35          "context_testcase": {
36            "input": {
37              "arguments": ["E", "y", "j", "2", "f", "N", "a"],
38              "main_call": true
39            },
40            "output": {
41              "stderr": {
42                "type": "text",
```

```
43         "data": "som: ongeldige argumenten"
44     },
45     "exit_code": {
46         "value": 1
47     }
48 }
49 }
50 }
51 ]
52 }
53 ]
54 }
```

TODO: ISBN oefening

5. Beperkingen en toekomstig werk

De doelstelling van deze thesis is het formuleren van een antwoord op de vraag of programmeertaalafhankelijke oefeningen mogelijk zijn (zie paragraaf 1.5). Als antwoord hierop is een prototype, TESTed, ontwikkeld. Op basis van het prototype blijkt dat het antwoord op de onderzoeksvraag positief is. Eigen aan een prototype is dat het nog niet *production ready* is. Dit hoofdstuk handelt over ontbrekende functionaliteit of beperkingen van TESTed, en reikt mogelijke ideeën voor een oplossing 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, 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:

¹Zie [https://en.wikipedia.org/wiki/Shebang_\(Unix\)](https://en.wikipedia.org/wiki/Shebang_(Unix)).

```
1  #!/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 werd 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 2.2.4, 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.

De volgende vraag die zich stelt is hoe dit praktisch moet aangepakt worden. Het bepalen van de ondersteunde programmeertalen is iets specifiek 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-team². Het voorstel is om oefeningen en judges te combineren in Dodona tot één Docker-container. Interessant binnen `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 plaatsvinden.

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

²Zie <https://github.com/dodona-edu/dodona/issues/1754>

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 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 2.2.4 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 gemaakt kan 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 de lesgevers op het niveau van het vak, de reeks of de oefening een *whitelist* of witte lijst van programmeertalen 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 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 2.2.1 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 2.2.1). Bovendien is een designkeuze 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 oefening, 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 voorbeeld-oefening 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*) [29]. 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 beschikbaar online. De beschrijving van de *Palindroom*-oefening is codefragment 5.1.

Een tweede standaard is PEXL (*Programming Exercise Markup Language*) [9]. PEXL is een tekstueel formaat (een soort DSL) dat er veelbelovend uitziet om met de hand te schrijven. Op het moment van schrijven van deze thesis is PEXL nog in volle ontwikkeling. Aangezien een van de *design goals* van PEXL is dat de omzetting van PEXL 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 PEXL.

5.2.2. Dynamische scheduling van testen

In TESTED is gekozen om het testplan in JSON op te stellen (zie paragraaf 2.2.1). Een nadeel van deze aanpak 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 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 kunnen worden met verschillende mogelijkheden. Zo is reeds voorzien in het testplan


```

1 <specification line_terminator="\n" id="input">
2   <line>
3     <data type="text" value="racecar"/>
4   </line>
5   <line>
6     <data type="text" value="Flintstone"/>
7   </line>
8 </specification>

```

```

1 <specification line_terminator="\n" id="output">
2   <line>
3     <data type="text">
4       "racecar" is a palindrome."
5     </data>
6   </line>
7   <line>
8     <data type="text">
9       "Flintstone" is not a palindrome.
10    </data>
11  </line>
12 </specification>

```

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

```

1  exercise_id: edu.vt.cs.cs1114.palindromes
2
3  title: Palindromes (A Simple PEXL Example)
4
5  topics: Strings, loops, conditions
6  prerequisites: variables, assignment, boolean operators
7
8  instructions: -----
9  Write a program that reads a single string (in the form of one line of text) from
10 its standard input, and determines whether the string is a _palindrome_. A
11 palindrome is a string that reads the same way backward as it does forward, such as
12 "racecar" or "madam". Your program does not need to prompt for its input, and
13 ↪ should
14 only generate one line of output, in the following format:
15
16 <pre>"racecar" is a palindrome.</pre>
17
18 or
19
20 <pre>"Flintstone" is not a palindrome.</pre>
21 -----
22
23 assets.test_format: stdin-stdout
24
25 [assets.tests]
26 stdin: racecar
27 stdout: "racecar" is a palindrome.
28
29 stdin: Flintstone
30 stdout: "Flintstone" is not a palindrome.
31
32 stdin: url(some/local/input.txt)
33 stdout: url(some/local/output.txt)
34
35 stdin: url(http://example.com/some/local/generator/input)
36 stdout: url(http://example.com/some/local/generator/output)

```

Codefragment 5.2. Beschrijving van de oefening „palindroom” in PEXL. Overgenomen uit [9].

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 preconditionie gegeven worden: welke contexten moet geslaagd zijn om de huidige context uit te voeren. Zijn niet alle contexten uit de preconditionie 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. 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.

5.2.4. Beperkte expressies in het testplan

De expressies in het testplan zijn opzettelijk eenvoudig gehouden (zie paragraaf 2.2.3). 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 hierin gaat, begint dit wel een universele programmeertaal te worden (wat ook expliciet buiten het bestek van deze thesis 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).

5.2.5. Ondersteuning voor 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 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:

```
1 {  
2   "name": "is_isbn"  
3 }
```

Ondersteuning voor vertaling kan dan deze vorm aannemen:

```
1 {  
2   "name": {  
3     "en": "are_isbn",  
4     "nl": "zijn_isbn"  
5   }  
6 }
```

Als een functie binnen een testplan veel wordt opgeroepen, 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:

```
1 {  
2   "translations": {  
3     "are_isbn": {  
4       "en": "are_isbn",  
5       "nl": "zijn_isbn"  
6     }  
7   }  
8 }
```

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

```
1 {  
2   "name": "$are_isbn"  
3 }
```

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's³.

5.3. Performantie

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

5.3.1. Performantie uitvoeren contexten

Het uitvoeren van elke context in een afzonderlijk subprocess brengt een inherente performantie-kost 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 verminderd kan 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` voorzien kunnen worden, waarmee alle contexten uitgevoerd zullen 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.

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 getoond wordt in Dodona aan de studenten. 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.

³Zie <https://docs.python.org/3/library/gettext.html>

5.3.2. Performantie geprogrammeerde evaluatie

Het uitvoeren van een geprogrammeerde evaluatie (zie paragraaf 2.4.2) 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 2.5 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 die programmeerde evaluatie gebruikmaken. Wordt er n keer andere evaluatiecode in een andere programmeertaal gebruikt, 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 rechte reeks in TESTed gedaan zonder subproces): hiervoor is het gebruik van Jupyter-kernels niet nuttig.

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. Programmeertaalafhankelijke opgaven

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

Een idee is hier 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 programmeertaalafhankelijk, 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. Programmeertaalafhankelijke programmeerparadigma

TESTED vertaalt geen programmeerparadigma 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 denkpijsten om hier een oplossing voor te bieden:

- Voorzie binnen TESTED ook vertalingen van programmeerparadigma. 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 programmeertaalafhankelijk 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 programmeertaalafhankelijke evaluatie te doen van de resultaten.

Het omgekeerde, programmeertaalafhankelijk invoer en programmeertaalspecifieke beoordeling, bestaat al binnen TESTED (zie paragraaf 2.4.3). Als tot slot zowel de invoer als de uitvoer verschilt van programmeertaal tot programmeertaal, kan er niet meer gesproken worden van een programmeertaalafhankelijke oefeningen. In dat geval is het beter een bestaande programmeertaalspecifieke judge van Dodona te gebruiken.

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 `Date` 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 minder evaluatiecode in elke programmeertaal geschreven worden.

Hieraan gerelateerd is een nieuwe evaluatiemodus, die we een *orakevaluatie* 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 orakevaluatie 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.

5.5. Extra functionaliteit

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

- Ondersteuning voor Python Tutor. Dit is een visuele debugger, die origineel geschreven is voor Python [14]. 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 2.6), maar bij sommige talen worden ook veel nuttige waarschuwingen getoond bij het compileren van de code. Deze waarschuwingen wordt momenteel getoond in TESTED, en er is ondersteuning om deze waarschuwingen om te zetten in 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 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 weer-
geven van waarden om het vergelijken met de verwachte uitvoer gemakkelijker te maken.
Een voorbeeld is verzamelingen gesorteerd tonen, zodat het vinden van de verschillen een-
voudiger 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 eenvou-
dig 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 binnen Dodona een status voor een test die niet uitgevoerd
is⁴. 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.

⁴Zie de issue: <https://github.com/dodona-edu/dodona/issues/1785>

Bibliografie

- [1] *Amazon Ion*. Amazon. 15 januari 2020. URL: <https://amzn.github.io/ion-docs/> (bezocht op 27-01-2020).
- [2] *Apache Avro™ 1.9.1 Documentation*. The Apache Foundation. 9 februari 2019. URL: <http://avro.apache.org/docs/1.9.1/> (bezocht op 27-01-2020).
- [3] *Apache Parquet*. The Apache Foundation. 13 januari 2020. URL: <https://parquet.apache.org/documentation/latest/> (bezocht op 27-01-2020).
- [4] Ben Bastiaensen en Jan De Craemer. *Zo denkt een computer*. Vlaamse Overheid, 2017. Hfdstk. Computationeel denken, p. 8–9.
- [5] Michael Bayer e.a. *Mako Templates for Python*. 20 januari 2020. URL: <https://www.makotemplates.org/>.
- [6] Carstenn Bormann en Paul Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049. Internet Engineering Task Force, oktober 2013. URL: <https://tools.ietf.org/html/rfc7049>.
- [7] *BSON 1.1*. MongoDB. 19 juli 2019. URL: <http://bsonspec.org/> (bezocht op 17-01-2020).
- [8] Bram Cohen. *The BitTorrent Protocol Specification*. 4 februari 2017. URL: http://bittorrent.org/beps/bep_0003.html.
- [9] Phill Conrad, Cory Bart en Stephen Edwards. *Programming Exercise Markup Language*. Standards, Protocols, en Learning Infrastructure for Computing Education (SPLICE). URL: <https://csssplice.github.io/peml/> (bezocht op 31-03-2020).
- [10] Wikipedia Contributors. *Comparison of data-serialization formats*. Wikipedia, The Free Encyclopedia. 25 januari 2020. URL: https://en.wikipedia.org/w/index.php?title=Comparison_of_data-serialization_formats&oldid=937433197.
- [11] „Decreet van 14 december 2018 betreffende de onderwijsdoelen voor de eerste graad van het secundair onderwijs”. In: *Belgisch Staatsblad* 26/04/2019 (14 december 2018), p. 40558. URL: http://www.ejustice.just.fgov.be/mopdf/2019/04/26_1.pdf#page=44.
- [12] Marc Duranton e.a. *HiPEAC Vision 2019*. Januari 2019. Hfdstk. Impact of computing technology on society, p. 110–111. ISBN: 9789090313641.
- [13] Sadayuki Furuhashi. *MessagePack*. 17 september 2018. URL: <https://msgpack.org/>.
- [14] Philip 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*. SIGCSE '13. Denver, Colorado, USA: Association for Computing Machinery, 2013, p. 579–584. ISBN: 9781450318686. DOI: [10.1145/2445196.2445368](https://doi.org/10.1145/2445196.2445368).
- [15] *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*. Recommendation X.608. International Telecommunications Union, augustus 2015. URL: <https://www.itu.int/rec/T-REC-X.680-201508-I/en>.
- [16] *Information technology – Generic applications of ASN.1: Fast infosec*. Recommendation X.881. International Telecommunications Union, 14 mei 2005. URL: <https://www.itu.int/rec/T-REC-X.891-200505-I/en>.

- [17] Jackson JSON team. *Smile Data Format*. 2010. URL: <https://github.com/FasterXML/smile-format-specification>.
- [18] Hieke Keuning, Johan Jeuring en Bastiaan 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*. ITiCSE '16. Arequipa, Peru: Association for Computing Machinery, 2016, p. 41–46. ISBN: 9781450342315. DOI: [10.1145/2899415.2899422](https://doi.org/10.1145/2899415.2899422).
- [19] Thomas Kluyver e.a. „Jupyter Notebooks – a publishing format for reproducible computational workflows”. In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Red. door F. Loizides en B. Schmidt. IOS Press. 2016, p. 87–90. DOI: [10.3233/978-1-61499-649-1-87](https://doi.org/10.3233/978-1-61499-649-1-87).
- [20] Adrian Kosowski, Michał Małafiejski en Tomasz Noiński. „Application of an Online Judge & Contester System in Academic Tuition”. In: *Advances in Web Based Learning – ICWL 2007*. Red. door Howard Leung e.a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 343–354. ISBN: 978-3-540-78139-4. DOI: [10.1007/978-3-540-78139-4_31](https://doi.org/10.1007/978-3-540-78139-4_31).
- [21] Eric Lengyel. *Open Data Description Language (OpenDDL)*. 17 januari 2017. URL: <http://openddl.org/>.
- [22] Andrew Luxton-Reilly e.a. „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. Larnaca, Cyprus: Association for Computing Machinery, 2018, p. 55–106. ISBN: 9781450362238. DOI: [10.1145/3293881.3295779](https://doi.org/10.1145/3293881.3295779).
- [23] Nilo Mitra en Yves Lafon. *SOAP Version 1.2 Part 0: Primer (Second Edition)*. TR. W3C, april 2007. URL: <http://www.w3.org/TR/2007/REC-soap12-part0-20070427/>.
- [24] *OGDL 2018.2*. 25 februari 2018. URL: <https://msgpack.org/> (bezocht op 28-01-2020).
- [25] Wouter van Oortmerssen. *FlatBuffers*. Google. 24 april 2019. URL: <https://google.github.io/flatbuffers/>.
- [26] *OPC unified architecture - Part 1: Overview and concepts*. IEC TR 62541-1:2016. International Electrotechnical Commission, 10 mei 2016. URL: <https://webstore.iec.ch/publication/25997>.
- [27] Charlotte Van Petegem. „Computationele benaderingen voor deductie van de computationele complexiteit van computerprogramma's”. Universiteit Gent, 2018. URL: <https://lib.ugent.be/catalog/rug01:002479652>.
- [28] *ProtocolBuffers*. Google. 13 december 2019. URL: <https://developers.google.com/protocol-buffers/>.
- [29] Ricardo Queirós en José Paulo 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](https://doi.org/10.400.22/4748).
- [30] Bernardo Ramos. *Binn*. 25 september 2019. URL: <https://github.com/liteserver/binn>.
- [31] Mark Slee, Aditya Agarwal en Marc Kwiatkowski. „Thrift: Scalable cross-language services implementation”. In: (2007). URL: <http://thrift.apache.org/static/files/thrift-20070401.pdf>.
- [32] *Universal Binary JSON*. 25 februari 2018. URL: <http://ubjson.org/>.
- [33] Dienst Curriculum & vorming. *Computationeel denken in Zin in leren! Zin in leven!* Katholiek Onderwijs Vlaanderen, 25 september 2017.

- [34] Max Wildgrube. *Structured Data Exchange Format (SDXF)*. RFC 3072. Internet Engineering Task Force, maart 2001. URL: <https://tools.ietf.org/html/rfc3072>.

A. Echo-oefening

In deze bijlage is een volledige oefening beschikbaar, met opgave, testplan en de gegenereerde code. Om de codefragmenten enigszins kort te houden, zijn er maar twee contexten in de oefening.

A.1. Opgave

De opgave van deze oefening luidt als volgt:

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

A.2. Testplan

Een testplan met twee testgevallen, elk in een andere context:

```
1 {
2   "tabs": [
3     {
4       "name": "Feedback",
5       "contexts": [
6         {
7           "context_testcase": {
8             "input": {
9               "main_call": true,
10              "stdin": {
11                "type": "text",
12                "data": "input-1"
13              }
14            },
15            "output": {
16              "stdout": {
17                "type": "text",
18                "data": "input-1"
19              }
20            }
21          },
22          {
23            "context_testcase": {
24              "input": {
25                "main_call": true,
26                "stdin": {
27                  "type": "text",
28                  "data": "input-2"
29                }
30              },
31              "output": {
32                "stdout": {
33                  "type": "text",
34                  "data": "input-2"
35                }
36              }
37            }
38          }
39        ]
40      }
41    ]
42  }
43 }
```

A.3. Python

A.3.1. Oplossing

```
1 i = input()
2 print(i)
```

A.3.2. Gegenerateerde 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

```
1 import values
2 import sys
3
4 value_file = open("DPS2okM9j_values.txt", "w")
5 exception_file = open("DPS2okM9j_exceptions.txt", "w")
6
7 def write_separator():
8     value_file.write("--DPS2okM9j-- SEP")
9     exception_file.write("--DPS2okM9j-- SEP")
10    sys.stderr.write("--DPS2okM9j-- SEP")
11    sys.stdout.write("--DPS2okM9j-- SEP")
12    sys.stdout.flush()
13    sys.stderr.flush()
14    value_file.flush()
15    exception_file.flush()
16
17 def send_value(value):
18     values.send_value(value_file, value)
19
20 def send_exception(exception):
21     values.send_exception(exception_file, exception)
22
```

```

23 def send_specific_value(r):
24     values.send_evaluated(value_file, r)
25
26 def send_specific_exception(r):
27     values.send_evaluated(exception_file, r)
28
29 try:
30     write_separator()
31     from submission import *
32 except Exception as e:
33     send_exception(e)
34 else:
35     send_exception(None)
36
37 value_file.close()
38 exception_file.close()

```

context_0_1.py

```

1 import values
2 import sys
3
4 value_file = open("DPS2okM9j_values.txt", "w")
5 exception_file = open("DPS2okM9j_exceptions.txt", "w")
6
7 def write_separator():
8     value_file.write("--DPS2okM9j-- SEP")
9     exception_file.write("--DPS2okM9j-- SEP")
10    sys.stderr.write("--DPS2okM9j-- SEP")
11    sys.stdout.write("--DPS2okM9j-- SEP")
12    sys.stdout.flush()
13    sys.stderr.flush()
14    value_file.flush()
15    exception_file.flush()
16
17 def send_value(value):
18     values.send_value(value_file, value)
19
20 def send_exception(exception):
21     values.send_exception(exception_file, exception)
22
23 def send_specific_value(r):
24     values.send_evaluated(value_file, r)
25
26 def send_specific_exception(r):

```



```

27     values.send_evaluated(exception_file, r)
28
29     try:
30         write_separator()
31         from submission import *
32     except Exception as e:
33         send_exception(e)
34     else:
35         send_exception(None)
36
37 value_file.close()
38 exception_file.close()

```

A.3.3. Uitvoeren

Bij het uitvoeren worden de gegenereerde bestanden uitgevoerd. In beide gevallen (batchcompilatie en contextcompilatie) wordt de juiste context uitgevoerd:

```

1 > python context_0_0.py
2 <uitvoer context_0_0>
3 > python context_0_1.py
4 <uitvoer context_0_1>

```

A.4. Java

A.4.1. Oplossing

```

1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4
5 public class Submission {
6     public static void main(String[] args) throws Exception {
7         BufferedReader reader = new BufferedReader(new
8             ↳ InputStreamReader(System.in));
9         String name = reader.readLine();
10        System.out.println(name);
11    }
12 }

```

A.4.2. Gegenerateerde code

De testcode die gegenereerd wordt bij het uitvoeren van bovenstaande testplan. Deze code is lichtjes aangepast: overvloedige 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`

```
1  import java.io.*;
2  import java.util.*;
3  import java.util.function.*;
4
5  public class Context00 implements Closeable {
6
7      private final PrintWriter valueWriter;
8      private final PrintWriter exceptionWriter;
9
10     public Context00() throws Exception {
11         this.valueWriter = new PrintWriter("izH9w8XbJ_values.txt");
12         this.exceptionWriter = new PrintWriter("izH9w8XbJ_exceptions.txt");
13     }
14
15     private void writeSeparator() throws Exception {
16         valueWriter.write("--izH9w8XbJ-- SEP");
17         exceptionWriter.write("--izH9w8XbJ-- SEP");
18         System.err.print("--izH9w8XbJ-- SEP");
19         System.out.print("--izH9w8XbJ-- SEP");
20         valueWriter.flush();
21         exceptionWriter.flush();
22         System.err.flush();
23         System.out.flush();
24     }
25
26     private void sendValue(Object value) throws Exception {
27         Values.send(valueWriter, value);
28     }
29
30     private void sendException(Throwable exception) throws Exception {
31         Values.sendException(exceptionWriter, exception);
32     }
33 }
```

```

32     }
33
34     private void sendSpecificValue(EvaluationResult r) {
35         Values.sendEvaluated(valueWriter, r);
36     }
37
38     private void sendSpecificException(EvaluationResult r) {
39         Values.sendEvaluated(exceptionWriter, r);
40     }
41
42     void execute() throws Exception {
43
44         this.writeSeparator();
45         try {
46             Submission.main(new String[]{
47                 sendException(null);
48             } catch (Exception | AssertionError e) {
49                 sendException(e);
50             }
51         }
52     }
53
54     @Override
55     public void close() throws IOException {
56         this.valueWriter.close();
57         this.exceptionWriter.close();
58     }
59
60     public static void main(String[] a) throws Exception {
61         try(Context00 context = new Context00()) {
62             context.execute();
63         }
64     }
65 }

```

Context01.java

```

1  import java.io.*;
2  import java.util.*;
3  import java.util.function.*;
4
5  public class Context01 implements Closeable {
6
7      private final PrintWriter valueWriter;
8      private final PrintWriter exceptionWriter;

```

```

9
10 public Context01() throws Exception {
11     this.valueWriter = new PrintWriter("izH9w8XbJ_values.txt");
12     this.exceptionWriter = new PrintWriter("izH9w8XbJ_exceptions.txt");
13 }
14
15 private void writeSeparator() throws Exception {
16     valueWriter.write("--izH9w8XbJ-- SEP");
17     exceptionWriter.write("--izH9w8XbJ-- SEP");
18     System.err.print("--izH9w8XbJ-- SEP");
19     System.out.print("--izH9w8XbJ-- SEP");
20     valueWriter.flush();
21     exceptionWriter.flush();
22     System.err.flush();
23     System.out.flush();
24 }
25
26 private void sendValue(Object value) throws Exception {
27     Values.send(valueWriter, value);
28 }
29
30 private void sendException(Throwable exception) throws Exception {
31     Values.sendException(exceptionWriter, exception);
32 }
33
34 private void sendSpecificValue(EvaluationResult r) {
35     Values.sendEvaluated(valueWriter, r);
36 }
37
38 private void sendSpecificException(EvaluationResult r) {
39     Values.sendEvaluated(exceptionWriter, r);
40 }
41
42 void execute() throws Exception {
43
44     this.writeSeparator();
45     try {
46         Submission.main(new String[]{
47             sendException(null);
48         } catch (Exception | AssertionError e) {
49             sendException(e);
50         }
51     }
52 }
53
54 @Override
55 public void close() throws IOException {
56     this.valueWriter.close();
57     this.exceptionWriter.close();

```

```

58     }
59
60     public static void main(String[] a) throws Exception {
61         try(Context01 context = new Context01()) {
62             context.execute();
63         }
64     }
65 }

```

Selector.java

```

1  class Selector {
2      public static void main(String[] a) throws Exception {
3          var name = a[0];
4          if ("Context00".equals(name)) {
5              Context00.main(new String[]{});
6          }
7          if ("Context01".equals(name)) {
8              Context01.main(new String[]{});
9          }
10     }
11 }

```

A.4.3. Uitvoeren

Bij batchcompilatie wordt `Selector.java` gecompileerd, wat leidt tot een reeks `.class`-bestanden. Vervolgens wordt de selector tweemaal uitgevoerd:

```

1  > ./java -cp . Selector Context00
2  <uitvoer context_0_0>
3  > ./java -cp . Selector Context01
4  <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 programma-argument zoals gebruikelijk.

Bij contextcompilatie wordt elke context afzonderlijk gecompileerd, wat ook leidt tot een reeks `.class`-bestanden. Deze keer worden de contexten zelf uitgevoerd:

```
1 > java -cp . Context00
2 <uitvoer context_0_0>
3 > java -cp . Context01
4 <uitvoer context_0_1>
```

A.5. C

A.5.1. Oplossing

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5
6     int c;
7
8     while ((c = getchar()) != EOF) {
9         putchar(c);
10    }
11
12    return EXIT_SUCCESS;
13 }
```

A.5.2. Gegenerateerde code

De testcode die gegenereerd wordt bij het uitvoeren van bovenstaande testplan. Deze code is lichtjes aangepast: overbodige 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

```

1  #include <stdio.h>
2
3  #include "values.h"
4  #include "submission.c"
5
6  static FILE* context_0_0_value_file = NULL;
7  static FILE* context_0_0_exception_file = NULL;
8
9  static void context_0_0_write_separator() {
10     fprintf(context_0_0_value_file, "--H6YnawX92-- SEP");
11     fprintf(context_0_0_exception_file, "--H6YnawX92-- SEP");
12     fprintf(stdout, "--H6YnawX92-- SEP");
13     fprintf(stderr, "--H6YnawX92-- SEP");
14 }
15
16 #undef send_value
17 #define send_value(value) write_value(context_0_0_value_file, value)
18
19 #undef send_specific_value
20 #define send_specific_value(value) send_evaluated(context_0_0_value_file, value)
21
22 int context_0_0() {
23
24     context_0_0_value_file = fopen("H6YnawX92_values.txt", "w");
25     context_0_0_exception_file = fopen("H6YnawX92_exceptions.txt", "w");
26
27     context_0_0_write_separator();
28
29     char* args[] = {    };
30     solution_main(0, args);
31
32     fclose(context_0_0_value_file);
33     fclose(context_0_0_exception_file);
34     return 0;
35 }
36
37 #ifndef INCLUDED
38 int main() {
39     return context_0_0();
40 }
41 #endif

```

context_0_1.c

```
1  #include <stdio.h>
2
3  #include "values.h"
4  #include "submission.c"
5
6  static FILE* context_0_1_value_file = NULL;
7  static FILE* context_0_1_exception_file = NULL;
8
9  static void context_0_1_write_separator() {
10     fprintf(context_0_1_value_file, "--H6YnawX92-- SEP");
11     fprintf(context_0_1_exception_file, "--H6YnawX92-- SEP");
12     fprintf(stdout, "--H6YnawX92-- SEP");
13     fprintf(stderr, "--H6YnawX92-- SEP");
14 }
15
16 #undef send_value
17 #define send_value(value) write_value(context_0_1_value_file, value)
18
19 #undef send_specific_value
20 #define send_specific_value(value) send_evaluated(context_0_1_value_file, value)
21
22 int context_0_1() {
23
24     context_0_1_value_file = fopen("H6YnawX92_values.txt", "w");
25     context_0_1_exception_file = fopen("H6YnawX92_exceptions.txt", "w");
26
27     context_0_1_write_separator();
28
29     char* args[] = {    };
30     solution_main(0, args);
31
32     fclose(context_0_1_value_file);
33     fclose(context_0_1_exception_file);
34     return 0;
35 }
36
37 #ifndef INCLUDED
38 int main() {
39     return context_0_1();
40 }
41 #endif
```


selector.c

```
1  #include <string.h>
2  #include <stdio.h>
3
4  #define INCLUDED true
5
6  #include "context_0_0.c"
7  #include "context_0_1.c"
8
9  int main(int argc, const char* argv[]) {
10
11     if (argc < 1) {
12         fprintf(stderr, "You must select the context");
13         return -2;
14     }
15
16     const char* name = argv[1];
17
18     if (strcmp("context_0_0", name) == 0) {
19         return context_0_0();
20     }
21     if (strcmp("context_0_1", name) == 0) {
22         return context_0_1();
23     }
24     return -1;
25 }
```

A.5.3. Uitvoeren

Bij batchcompilatie wordt `selector.c` gecompileerd, wat leidt tot een uitvoerbaar bestand `selector`. Dit laatste bestand wordt dan tweemaal uitgevoerd:

```
1  > ./selector context_0_0
2  <uitvoer context_0_0>
3  > ./selector context_0_1
4  <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:

```
1  > ./context_0_0
2  <uitvoer context_0_0>
```

```
3 > ./context_0_1
4 <uitvoer context_0_1>
```

B. Echofunctie-oefening

In deze bijlage is een volledige oefening beschikbaar, met opgave, testplan en de gegenereerde code. Om de codefragmenten enigszins kort te houden, is er maar een context, maar wel met twee testgevallen in dezelfde context.

B.1. Opgave

De opgave van deze oefening 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

```
1 >>> echo("5");  
2 "5"  
3 >>> echo("ok");  
4 "ok"
```

B.2. Testplan

Een testplan met een context met twee testgevallen:

```
1 {  
2   "tabs": [  
3     {  
4       "name": "Feedback",  
5       "contexts": [  
6         {  
7           "testcases": [  
8             {  
9               "input": {  
10                "type": "function",  
11                "name": "echo",  
12                "arguments": [  
                ]  
              }  
            }  
          ]  
        }  
      ]  
    }  
  ]  
}
```

```

13         {
14             "type": "text",
15             "data": "input-1"
16         }
17     ]
18 },
19 "output": {
20     "result": {
21         "value": {
22             "type": "text",
23             "data": "input-1"
24         }
25     }
26 },
27 {
28     "input": {
29         "type": "function",
30         "name": "echo",
31         "arguments": [
32             {
33                 "type": "text",
34                 "data": "input-2"
35             }
36         ]
37     },
38     "output": {
39         "result": {
40             "value": {
41                 "type": "text",
42                 "data": "input-2"
43             }
44         }
45     }
46 }
47 ]
48 ]
49 }
50 ]
51 }
52 ]
53 }

```

B.3. Python

B.3.1. Oplossing

```
1 def echo(content):  
2     return content
```

B.3.2. Gegenerateerde 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

```
1 import values  
2 import sys  
3  
4 value_file = open("iPE3Nt21C_values.txt", "w")  
5 exception_file = open("iPE3Nt21C_exceptions.txt", "w")  
6  
7 def write_separator():  
8     value_file.write("--iPE3Nt21C-- SEP")  
9     exception_file.write("--iPE3Nt21C-- SEP")  
10    sys.stderr.write("--iPE3Nt21C-- SEP")  
11    sys.stdout.write("--iPE3Nt21C-- SEP")  
12    sys.stdout.flush()  
13    sys.stderr.flush()  
14    value_file.flush()  
15    exception_file.flush()  
16  
17 def send_value(value):  
18     values.send_value(value_file, value)  
19  
20 def send_exception(exception):  
21     values.send_exception(exception_file, exception)  
22  
23 def send_specific_value(r):  
24     values.send_evaluated(value_file, r)
```

```

25
26 def send_specific_exception(r):
27     values.send_evaluated(exception_file, r)
28
29     try:
30         write_separator()
31         from submission import *
32     except Exception as e:
33         raise e
34
35     write_separator()
36     try:
37         send_value(echo('input-1'))
38
39     except Exception as e:
40         send_exception(e)
41     else:
42         send_exception(None)
43
44     write_separator()
45     try:
46         send_value(echo('input-2'))
47
48     except Exception as e:
49         send_exception(e)
50     else:
51         send_exception(None)
52
53     value_file.close()
54     exception_file.close()

```

B.3.3. Uitvoeren

Bij het uitvoeren worden de gegenereerde bestanden uitgevoerd. In beide gevallen (batchcompilatie en contextcompilatie) wordt de juiste context uitgevoerd:

```

1 > python context_0_0.py
2 <uitvoer context_0_0>

```

B.4. Java

B.4.1. Oplossing

```
1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4
5 public class Submission {
6     public static void main(String[] args) throws Exception {
7         BufferedReader reader = new BufferedReader(new
8             ↳ InputStreamReader(System.in));
9         String name = reader.readLine();
10        System.out.println(name);
11    }
12 }
```

B.4.2. Gegenerateerde 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

```
1 import java.io.*;
2 import java.util.*;
3 import java.util.function.*;
4
5 public class Context00 implements Closeable {
6
7     private final PrintWriter valueWriter;
8     private final PrintWriter exceptionWriter;
9
10    public Context00() throws Exception {
```

```

11     this.valueWriter = new PrintWriter("RJK5zjxUR_values.txt");
12     this.exceptionWriter = new PrintWriter("RJK5zjxUR_exceptions.txt");
13 }
14
15 private void writeSeparator() throws Exception {
16     valueWriter.write("--RJK5zjxUR-- SEP");
17     exceptionWriter.write("--RJK5zjxUR-- SEP");
18     System.err.print("--RJK5zjxUR-- SEP");
19     System.out.print("--RJK5zjxUR-- SEP");
20     valueWriter.flush();
21     exceptionWriter.flush();
22     System.err.flush();
23     System.out.flush();
24 }
25
26 private void sendValue(Object value) throws Exception {
27     Values.send(valueWriter, value);
28 }
29
30 private void sendException(Throwable exception) throws Exception {
31     Values.sendException(exceptionWriter, exception);
32 }
33
34 private void sendSpecificValue(EvaluationResult r) {
35     Values.sendEvaluated(valueWriter, r);
36 }
37
38 private void sendSpecificException(EvaluationResult r) {
39     Values.sendEvaluated(exceptionWriter, r);
40 }
41
42 void execute() throws Exception {
43
44     this.writeSeparator();
45
46     this.writeSeparator();
47     try {
48         sendValue(Submission.echo("input-1"));
49         sendException(null);
50     } catch (Exception | AssertionError e) {
51         sendException(e);
52     }
53     this.writeSeparator();
54     try {
55         sendValue(Submission.echo("input-2"));
56         sendException(null);
57     } catch (Exception | AssertionError e) {
58         sendException(e);
59     }

```



```

60
61     }
62
63     @Override
64     public void close() throws IOException {
65         this.valueWriter.close();
66         this.exceptionWriter.close();
67     }
68
69     public static void main(String[] a) throws Exception {
70         try(Context00 context = new Context00()) {
71             context.execute();
72         }
73     }
74 }

```

Selector.java

```

1  class Selector {
2      public static void main(String[] a) throws Exception {
3          var name = a[0];
4          if ("Context00".equals(name)) {
5              Context00.main(new String[]{});
6          }
7      }
8  }

```

B.4.3. Uitvoeren

Bij batchcompilatie wordt Selector.java gecompileerd, wat leidt tot een reeks .class-bestanden. Vervolgens wordt de selector uitgevoerd:

```

1  > ./java -cp . Selector Context00
2  <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:

```

1  > java -cp . Context00
2  <uitvoer context_0_0>

```

B.5. C

B.5.1. Oplossing

We voeren het testplan uit met deze oplossing:

```
1 char* echo(char* content) {  
2     return content;  
3 }
```

B.5.2. Gegenerateerde code

De gegenereerde code die gegenereerd wordt bij het uitvoeren van bovenstaande testplan. Deze code is lichtjes aangepast: overbodige 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

```
1 #include <stdio.h>  
2  
3 #include "values.h"  
4 #include "submission.c"  
5  
6 static FILE* context_0_0_value_file = NULL;  
7 static FILE* context_0_0_exception_file = NULL;  
8  
9 static void context_0_0_write_separator() {  
10     fprintf(context_0_0_value_file, "--IjxAJWTRu-- SEP");  
11     fprintf(context_0_0_exception_file, "--IjxAJWTRu-- SEP");  
12     fprintf(stdout, "--IjxAJWTRu-- SEP");  
13     fprintf(stderr, "--IjxAJWTRu-- SEP");  
14 }  
15  
16 #undef send_value  
17 #define send_value(value) write_value(context_0_0_value_file, value)  
18  
19 #undef send_specific_value  
20 #define send_specific_value(value) send_evaluated(context_0_0_value_file, value)
```

```

21
22 int context_0_0() {
23
24     context_0_0_value_file = fopen("IjxAJWTRu_values.txt", "w");
25     context_0_0_exception_file = fopen("IjxAJWTRu_exceptions.txt", "w");
26
27     context_0_0_write_separator();
28
29     context_0_0_write_separator();
30     send_value(echo("input-1"));
31
32     context_0_0_write_separator();
33     send_value(echo("input-2"));
34
35     fclose(context_0_0_value_file);
36     fclose(context_0_0_exception_file);
37     return 0;
38 }
39
40 #ifndef INCLUDED
41 int main() {
42     return context_0_0();
43 }
44 #endif

```

selector.c

```

1  #include <string.h>
2  #include <stdio.h>
3
4  #define INCLUDED true
5
6  #include "context_0_0.c"
7
8  int main(int argc, const char* argv[]) {
9
10     if (argc < 1) {
11         fprintf(stderr, "You must select the context");
12         return -2;
13     }
14
15     const char* name = argv[1];
16
17     if (strcmp("context_0_0", name) == 0) {
18         return context_0_0();
19     }
20 }

```

```
19     }  
20     return -1;  
21 }
```

B.5.3. Uitvoeren

Bij het compileren wordt `selector.c` gecompileerd, wat leidt tot een uitvoerbaar bestand `selector`. Dit laatste bestand wordt dan uitgevoerd:

```
1 > ./selector context_0_0  
2 <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:

```
1 > ./context_0_0  
2 <uitvoer context_0_0>
```