FACULTEIT
WETENSCHAPPEN

TESTED: ONE JUDGE TO RULE THEM ALL

Niko Strijbol

Studentennummer: 01404620

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

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

Academiejaar: 2019 – 2020

UNIVERSITEIT
GENT

INHOUDSOPGAVE

1. INLEIDING

Programmeren -> steeds belangrijker en nuttigere kennis om te hebben Goed programmeren -> vereist veel oefening Zeker in cursussen met meer mensen -> goede ondersteuning lesgever Automatiseren van beoordeling programmeeroefeningen -> Dodona

TODO: eventueel samenvoegen met volgende hoofdstuk?

Dit is de inleiding van mijn thesis. Hallo aan iedereen! Omdat we met een initiaal werken, is het aangewezen dat de eerste alinea redelijk wat tekst bevat.

Verder is dit aan de orde:

$$y = 5 + 6$$

Hallo ik ben kleinkapitaal!

Hallo, dit is tekst met code: `dit is dan de code`.

Zeekomkommers (Holothuroidea) vormen een groep van ongewervelde dieren die behoren tot de klasse van stekelhuidigen. De meeste soorten hebben een langwerpig en worstvormig lichaam dat zowel aan de voor- als achterzijde stomp eindigt. Er zijn ook vormen met een sliertige of een bolle lichaamsvorm.

2. DODONA

2.1. Wat is Dodona?

Intro over Dodona: korte geschiedenis, terminologie, hoe Dodona werkt (oefeningen, judges, enz.) -> Over de judge wordt in het deel hierna meer verteld.

2.2. Evalueren van een oplossing

Zoals reeds vermeld worden de oplossingen van studenten geëvalueerd door een zelfgeschreven evaluatiescript, de *judge*. In wezen is dit een eenvoudig programma: het krijgt de configuratie via de standaardinvoerstroom (stdin) en schrijft de resultaten van de evaluatie naar de standaarduitvoerstroom (stdout). Zowel de invoer als de uitvoer van de judge zijn json. Het formaat van deze uitwisseling ligt vast in een json-schema, dat publiekelijk beschikbaar is. [1]

Een judge ondersteunt één programmeertaal. In de praktijk ondersteunt elke judge oplossingen in de programmeertaal waarin hij geschreven is, m.a.w. de Java-judge ondersteunt Java, de Python-judge Python, enz. Ook heeft elke judge een eigen manier waarop de testen voor een oplossing opgesteld moeten worden. Zo worden in de Java-judge jUnit-testen gebruikt, terwijl de Python-judge doctests en een eigen formaat ondersteunt.

In grote lijnen verloopt het evalueren van een oplossing van een student als volgt:

1. De student dient de oplossing in via de webinterface van Dodona.
2. Dodona start een Docker-image met de judge.
3. De judge wordt uitgevoerd, met als invoer de configuratie, zoals hierboven vermeld.
4. De judge evalueert de oefening aan de hand van de code van de student en de evaluatiecode opgesteld door de lesgever (ie. de jUnit-test, de doctests, ...).
5. De judge vertaalt het resultaat van deze evaluatie naar het Dodona-formaat en schrijft het uit.
6. Dodona vangt die uitvoer op, en toont het resultaat aan de student.

2.3. Probleemstelling

Het huidige systeem waarop de judges werken resulteert in twee nadelen. Bij het bespreken van de nadelen is het nuttig een voorbeeld in het achterhoofd te houden, teneinde de nadelen te kunnen concretiseren. Als voorbeeld gebruiken we de „Lotto”-oefening[2], met volgende opgave:

[1] Een tekstuele beschrijving is te vinden in de handleiding (Dodona-team 2020).
[2] Vrij naar een oefening van prof. Dawyndt.

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` kan doorgegeven worden hoeveel balletjes a er moeten getrokken worden (standaardwaarde 6). Aan de parameter `maximum` 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 $a \leq m$. De functie moet een string teruggeven die een strikt stijgende lijst van a natuurlijke getallen beschrijft, waarbij de getallen van elkaar gescheiden zijn door een spatie, een koppelteken (-) en nog een spatie. Voor elk getal n moet gelden dat $1 \leq n \leq m$.

Oplossingen voor deze opgave staan in codefragmenten 2.1 en 2.2, voor respectievelijk Python en Java.

Het belangrijkste nadeel aan de huidige werking is het bijkomende werk voor lesgevers indien zij hun oefeningen in meerdere talen willen aanbieden. De Lotto-oefening heeft een eenvoudige opgave en oplossing. Bovendien zijn de verschillen tussen de versie in Python en Java minimaal, zij het dat de Java-versie wat langer is. Deze oefening zou zonder problemen in nog vele andere programmeertalen geïmplementeerd kunnen worden.

Ook bij ingewikkeldere oefeningen die zich concentreren op algoritmen, waar de uiteindelijke taal van de implementatie niet relevant is. Een voorbeeld hiervan is het vak „Algoritmen en Datastructuren" dat gegeven wordt door prof. Fack aan de wiskunde[3]. Daar zijn de meeste opgaven vandaag al beschikbaar in Java en Python op Dodona, maar dan als afzonderlijke oefeningen.

Het evalueren van een oplossing voor de Lotto-oefening is minder eenvoudig, daar er met willekeurige getallen gewerkt wordt: het volstaat niet om de uitvoer gegenereerd 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 ons voorbeeld controleert deze code bijvoorbeeld of de gegeven getallen binnen het bereik liggen en of ze gesorteerd zijn.

Voor de lesgevers is het opnieuw opstellen van deze evaluatiecode veel en repetitief werk. Duur het twee minuten om deze code te schrijven en een vak heeft 30 oefeningen, dan duurt het een uur. In twee talen duurt al twee uur, tien talen vraagt al tien uur.

Het tweede nadeel aan de huidige werking, maar wel veel kleiner van belang, betreft het implementeren van de judges zelf. Hoewel de interface voor de judges eenvoudig is, blijkt in de praktijk dat het implementeren van een judge verre van eenvoudig is. Uiteraard is een deel van die complexiteit ingevolge het evalueren van de code, een taak die niet in alle talen eenvoudig is. Doch is

[3]De studiefiche is voor de geïnteresseerden beschikbaar op https://studiegids.ugent.be/2019/NL/studiefiches/C002794.pdf

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

class Main {

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

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

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

Codefragment 2.1.: Voorbeeldoplossing in Java.

```python
from random import randint


def loterij(aantal=6, maximum=42):
    getallen = set()
    while len(getallen) < aantal:
        getallen.add(randint(1, maximum))

    return " - ".join(str(x) for x in sorted(getallen))
```

Codefragment 2.2.: Voorbeeldoplossing in Python.

de complexiteit ten dele te wijten aan de noodzaak om in elke judge opnieuw de hele evaluatieprocedure te implementeren. Aan het deel van de code dat het resultaat van een evaluatie omzet naar het Dodona-formaat is niets taalspecifieks.

Het probleem hierboven beschreven laat zich samenvatten als volgende onderzoeksvraag, waarop deze thesis een antwoord wil bieden:

> Is het mogelijk om een judge zo te implementeren, dat de opgave en evaluatiecode van een oefening slechts eenmaal opgesteld dienen te worden, waarna de oefening beschikbaar is in alle talen die de judge ondersteunt?

2.4. Opbouw

Het volgende hoofdstuk van deze thesis handelt over het antwoord op bovenstaande vraag. Daarna volgt ter illustratie van het gebruik van de judge een gedetailleerde beschrijving van hoe een nieuwe taal moet toegevoegd worden aan de thesis. Dat hoofdstuk heeft ook ten doel te dienen als documentatie voor zij die de judge willen gebruiken. Tot slot wordt afgesloten met een hoofdstuk over beperkingen van de huidige implementaties, en waar er verbeteringen mogelijk zijn (het „toekomstige werk").

3. DE UNIVERSELE JUDGE

Het antwoord op de onderzoeksvraag uit het vorige hoofdstuk manifesteert zich als de *universele judge*. Deze judge voor het Dodona-platform kan dezelfde oefening in meerdere talen evalueren. Dit hoofdstuk licht de werking en implementatie van deze judge toe, beginnend met een algemeen overzicht, waarna elk onderdeel in meer detail besproken wordt.

TODO: terminologie uitleggen (oplossing, evaluatie, ...) Een groot deel hiervan zal waarschijnlijk uitgelegd zijn bij de werking van Dodona.

3.1. Overzicht

Figuur 3.1 toont de opbouw van de judge op schematische wijze. De judge kan worden opgedeeld in drie gebieden, volgens hun verantwoordelijkheid:

1. Het evaluatieproces, dat de verkregen resultaten interpreteert en beoordeelt.
2. Het kernproces, dat zorgt voor de coördinatie tussen de andere processen, alsook de basistaken vervult. Dit proces is dan ook het start- en eindpunt van een evaluatie.
3. Het uitvoeringsproces, dat de code van de student uitvoert om zo resultaten te bekomen.

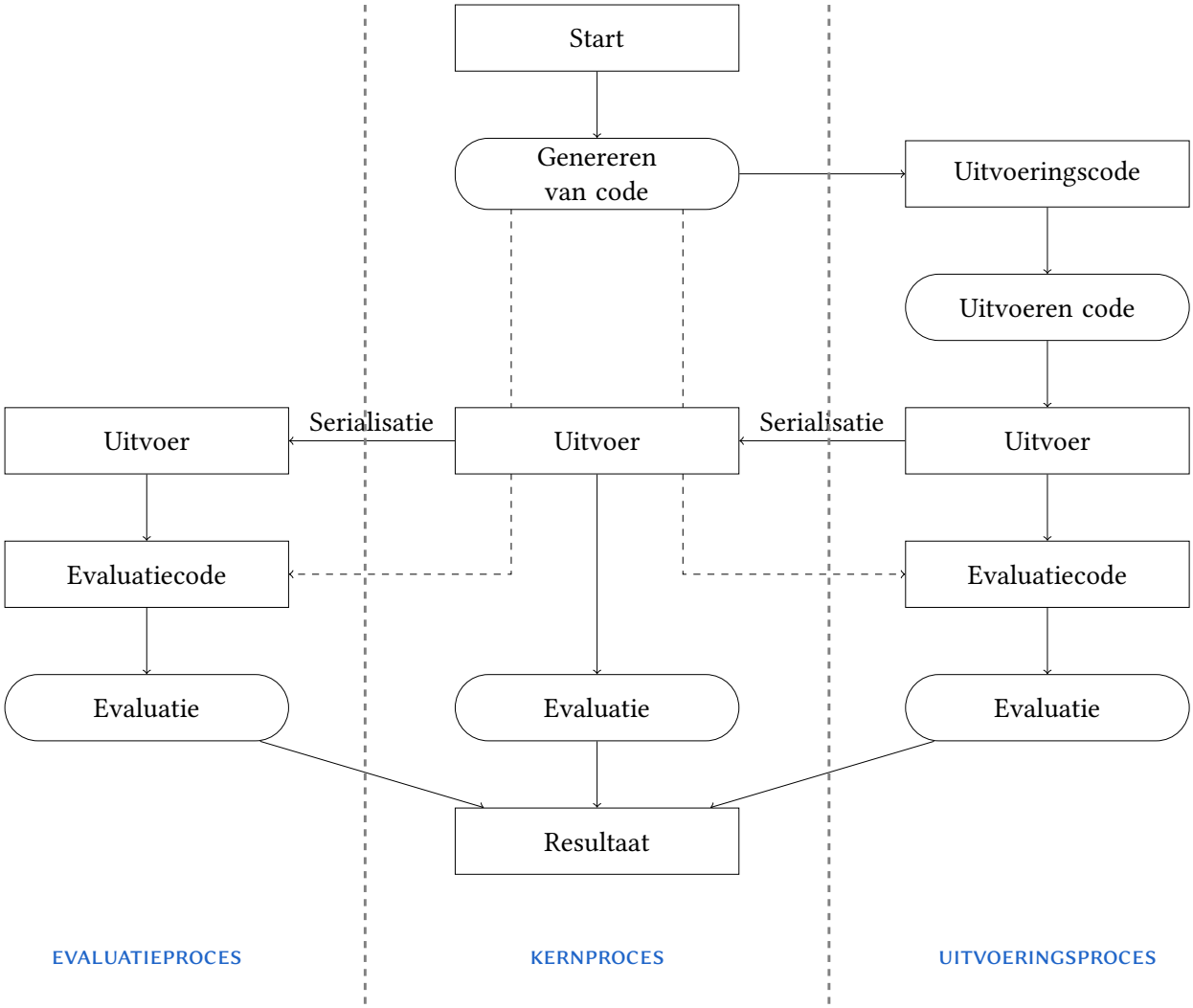
3.2. Beschrijven van een oefening

Naast de drie delen die hierboven beschreven zijn, is er ook nog het *testplan*. Dit is een document dat de oefening beschrijft en de evaluatietesten bevat. Het vervangt de taalspecifieke testen van de bestaande judges (dus de jUnit-tests of de doctests in respectievelijk Java en Python).

3.2.1. Het testplan

Het testplan neemt de vorm aan van een json-bestand. Bij de keuze voor het formaat van het testplan zijn er enkele doelen vooropgesteld. Het testplan moet:

- leesbaar zijn voor mensen,
- geschreven kunnen worden door mensen met minimale informatica-kennis, met andere woorden de syntaxis dient eenvoudig te zijn, en
- taalonafhankelijk zijn.



Figuur 3.1.: Schematische voorstelling van de opbouw van de universele judge.

Json voldoet aan deze vereisten: het is geen binair formaat en de syntaxis is eenvoudig. Een bijkomend voordeel is dat json door veel talen ondersteund wordt.

Er zijn ook enkele nadelen aan het gebruik van json. Een nadeel is dat json geen beknopte taal is voor het met de hand schrijven. Een oplossing hiervoor gebruikt de eigenschap dat veel talen json kunnen produceren: andere programma's of gereedschappen kunnen desgewenst het testplan in het json-formaat genereren, waardoor het niet met de hand geschreven moet worden. Hiervoor is waarschijnlijk een *DSL* (*domain specific language*) geschikt, maar dit valt buiten de thesis en wordt verder besproken in hoofdstuk 5.

Een ander nadeel is dat json geen programmeertaal is. Terwijl dit de implementatie wel eenvoudiger maakt, is het tevens beperkend: beslissen of een testgeval moet uitgevoerd worden op basis van het resultaat van een vorig testgeval is bij wijze als voorbeeld niet mogelijk. Ook deze beperking wordt uitgebreider besproken in hoofdstuk 5.

Qua structuur lijkt het testplan op de structuur opgelegd door Dodona, aangezien dat de verschillen tussen het opstellen van een oefening en hoe ze gebruikt zal worden minimaliseert. Bij het beschrijven van de structuur is zoveel mogelijk gebruik gemaakt van de bestaande terminologie, zoals die gebruikt wordt door Dodona. Een overzicht van de structuur op Dodona is (Dodona-team 2020).

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

Context Elk tabblad bestaat uit een of meerdere *context*en. Een context is een onafhankelijke uitvoering van een evaluatie. De nadruk ligt op de „onafhankelijkheid". Vaak wordt elke context in een nieuw proces uitgevoerd, zodat er geen informatie tussen contexten kan uitgewisseld worden.

Testcase Een context bestaat uit een of meerdere *testcase*s of testgevallen. Een testgeval bestaat uit invoer en een aantal tests. Een context bevat twee soorten testgevallen:

Main testcase of hoofdtestgeval. Van deze soort is er maximaal één per context. Dit testgeval is voor het uitvoeren van de main-functie (of gewoon de code als het gaat om een scripttaal zoals Bash of Python). Als invoer voor dit testgeval kan enkel het standaardinvoerkanaal en de programma-argumenten meegegeven worden.

Normal testcase of normaal testgeval. Hiervan kunnen er nul of meer zijn per context. Deze testgevallen zijn voor andere dingen te testen, nadat de code van de gebruiker met success ingeladen is. De invoer is dan ook uitgebreider: het kan gaan om het standaardinvoerkanaal, functieoproepen en variabeletoekenningen. Een functieoproep of variabeletoekenning is verplicht.

Test Een testcase bestaat uit meerdere *test*s, die elk één aspect van een testcase controleren. In het testplan kan voor elke soort uitvoer in een testcase het verwachte resultaat of de gewenste controleactie aangegeven worden. Het gaat om het standaarduitvoerkanaal, het standaardfoutkanaal, opgevangen uitzonderingen, de teruggegeven waarden van een functieoproep en de inhoud van een bestand.

3.2.2. Dataserialisatie

In de paragraaf hiervoor wordt bij de beschrijving van het testplan gewag gemaakt van de teruggegeven waarden van een functieoproep. Aangezien het testplan taalonafhankelijk is, moet er dus een manier zijn om data uit de verschillende programmeertalen voor te stellen.

Bij het kiezen van een serialisatieformaat, waren er enkele vereisten waaraan voldaan moest worden. Het formaat moet:

- de types ondersteunen die we willen aanbieden in het taalonafhankelijke deel van het testplan,
- het formaat moet eenvoudig te schrijven zijn vanuit meerdere programmeertalen (dus liever niet binair),
- toelaten op redelijk eenvoudige wijze ondersteuning voor nieuwe programmeertalen toe te voegen.

In de eerste plaats is gekeken of een bestaand formaat kan voldoen aan deze vereisten. Hiervoor is gestart van een overzicht op Wikipedia, zie (Wikipedia-bijdragers 2020). Een alternatief is zelf een formaat vast te leggen om waarden te encoderen als json. Voor dit alternatief is uiteindelijk gekozen. Samenvattend zijn de redenen waarom niet gekozen is voor een bestaande oplossing:

- Het is een binair formaat. Hoewel dit vaak voordelen oplevert op het vlak van geheugengebruik en snelheid, wordt het encoderen naar en decoderen van het formaat een stuk ingewikkelder om te implementeren in andere programmeertalen. Sommige bestaande oplossing zijn weliswaar reeds beschikbaar in een hoeveelheid talen, maar er is toch gekozen om het zelf implementeren zo gemakkelijk mogelijk te houden, daar de judge misschien talen wilt ondersteunen waarvoor er nog niets bestaat.
- De bestaande oplossing is niet uitbreidbaar met eigen types. Sommige oplossingen hebben een breed arsenaal aan ingebouwde types, maar de judge vereist soms zeer specifieke types.
- De bestaande oplossing is omslachtig in gebruik voor ons doel. Dit is het geval wanneer een oplossing veel meer doet dan vereist voor ons doel.
- De bestaande oplossing heeft geen voordelen tegenover een eigen oplossing in json, terwijl er wel het nadeel van een bijkomende taal is.

Hieronder volgt nu per overwogen serialisatie-oplossing een korte beschrijving en de grootste min- of pluspunten.

Apache Avro Een volledig „systeem voor dataserialisatie”. De specificatie van het formaat gebeurt in json, terwijl de eigenlijke data binair geëncodeerd wordt. Heeft uitbreidbare types, met veel ingebouwde types (*Apache Avro™ 1.9.1 Documentation* 2019).

Apache Parquet Minder relevant, dit is een bestandsformaat voor Hadoop (*Apache Parquet* 2020).

ASN.1 Staat voor *Abstract Syntax Notation One*, een ouder formaat uit de telecommunicatie. De hoofdstandaard beschrijft enkel de notatie voor een dataformaat. Andere standaarden beschrijven dan de serialisatie, in bv. binair formaat, json of xml. De meerdere serialisatievormen zijn in theorie aantrekkelijk: elke taal moet er slechts één ondersteunen, terwijl de judge ze allemaal kan ondersteunen. In de praktijk blijkt echter dat voor veel talen, er slechts een serialisatieformaat is, en dat dit vaak het binaire formaat is (*Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation* 2015).

Bencode Schema gebruikt in BitTorrent. Het is gedeeltelijk binair, gedeeltelijk in text. Beperkte hoeveelheid ingebouwde types, uitbreidbaar met eigen types via een omweg (Cohen 2017).

Binn Binair dataformaat. Ondersteunt redelijk wat ingebouwde types, alsook zelfgedefinieerde types (Ramos 2019).

BSON Een binaire variant op json, geschreven voor en door MongoDB. Heeft geen voordelen tegenover json, alleen nadelen, aangezien het binair is (*BSON 1.1* 2019).

CBOR Een lichtjes op json gebaseerd formaat, ook binair. Heeft een goede standaard, ondersteunt redelijk wat talen. De hoeveelheid ingebouwde types is redelijk groot, en eigen types zijn mogelijk (Bormann en Hoffman 2013).

FlatBuffers Lijkt op ProtocolBuffers, allebei geschreven door Google, maar verschilt wat in implementatie van ProtocolBuffers. De encodering is binair. Met een schema op te stellen kunnen eigen types eenvoudig samengesteld worden vanuit de ingebouwde basistypes (Oortmerssen 2019).

Fast Infoset Is eigenlijk een manier om xml binair te encoderen (te beschouwen als een soort compressie voor xml), waardoor het niet bruikbaar is (*Information technology – Generic applications of ASN.1: Fast infoset* 2005).

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. Het encoderen van bijkomende, eigen types is mogelijk door S-expressies, al is dat minder elegant (*Amazon Ion* 2020).

MessagePack Nog een binair formaat dat lichtjes op json gebaseerd is. Lijkt qua types sterk op json. Heeft implementaties in zeer veel talen (Furuhashi 2018).

OGDL Afkorting voor *Ordered Graph Data Language*. Daar het om een serialisatieformaat voor grafen gaat, is het niet nuttig voor ons doel (*OGDL 2018.2* 2018).

OPC Unified Architecture Een protocol voor intermachinecommunicatie. Zeer complex: de specificatie bevat 14 documenten, met ongeveer 1250 pagina's (*OPC unified architecture - Part 1: Overview and concepts* 2016).

OpenDLL Afkorting voor de *Open Data Description Language*. Een tekstueel formaat, bedoeld om arbitraire data voor te stellen. Nieuwe type's kunnen gedefinieerd worden. Wordt niet veel programmeertalen ondersteunen dit. Qua functionaliteit zelfs iets boven json; door de mogelijkheid om zelf types te definiëren, maar de grote ondersteuning voor json in andere talen zorgt ervoor dat json toch de voorkeur geniet (Lengyel 2017).

ProtocolBuffers Lijkt zoals vermeld sterk op FlatBuffers, maar heeft nog extra stappen nodig bij het encoderen en decoderen, wat het minder geschikt maakt (*ProtocolBuffers* 2019).

Smile Nog een binaire variant van json (Jackson JSON team 2010).

SOAP Afkorting voor *Simple Object Access Protocol*. Niet bedoeld als formaat voor dataserialisatie, maar voor communicatie tussen systemen over een netwerk (Mitra en Lafon 2007).

SDXF Binair formaat voor data-uitwisseling. Weinig talen ondersteunen dit formaat. Er zijn weinig ingebouwde types, maar complexere types maken uit de basistypes is mogelijk (Wildgrube 2001).

Thrift Lijkt sterk op ProtocolBuffers, maar geschreven door Facebook (Slee, Agarwal en Kwiatkowski 2007).

UBJSON Nog een binaire variant van json (*Universal Binary JSON 2018*).

Door de redenen hierboven voldoen geen van alle hiervoor genoemde oplossingen aan alle vereisten. Zoals al vermeld is dan gekozen om zelf een formaat op te stellen. Uiteindelijk is hierbij gekozen om data te serialiseren in json. Er is niet gekozen voor xml, omdat het testplan al json gebruikt, en xml geen meerwaarde biedt om het introduceren van een nieuwe taal te rechtvaardigen. Ook is niet gekozen voor een variant, zoals yaml, om het zo eenvoudig mogelijk te maken het formaat te implementeren.

Het serialisatieformaat beschrijft voor elke data welk type het is. Dit vormt een json-object bestaande uit het type en de waarde. Codefragment 3.1 toont hoe dit er uit ziet voor een voorbeeld van een lijst van twee getallen. Een formelere definitie van het formaat is appendix A.

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

Codefragment 3.1.: Voorbeeld van een lijst van twee getallen in het serialisatieformaat.

Het serialisatieformaat wordt zowel gebruikt in het testplan zelf om waarden te encoderen, als los ervan om resultaten van functieoproepen te encoderen.

3.2.3. Functieoproepen en assignments

Een ander onderdeel van het testplan verdient ook speciale aandacht: het toekennen van variabelen (*assignment*) en de functieoproepen.

In heel wat oefeningen, en zeker in objectgerichte 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 geïllustreerd in codefragment 3.2. Dit voorbeeld illustreert meteen dat functieoproepen ook zeker ondersteunt moeten worden.

Concreet is ervoor gekozen om het testplan niet uit te breiden met generieke statements of expressions, maar de ondersteuning te beperken tot assignments en functieoproepen. Dit om de implementatie van de vertaling van het testplan naar de ondersteunde programmeertalen niet nodeloos ingewikkeld te maken.

Conceptueel ondersteunt het testplan het toekennen van een naam aan het resultaat van een functieoproep. Vertaalt naar json wordt dat dan:

```
var variabele = new DoorDeStudentGemaakteKlasse();
assert variabele.testfunctie2();
assert variabele.testfunctie2();
```

Codefragment 3.2.: Een kort conceptueel voorbeeld van het soort oefeningen waar een assignment nodig is.

```
{
  "name": "Naam van de variabele",
  "expression": "<Object voor functieoproep>",
  "type": "Optioneel type"
}
```

De naam is de naam die aan de variabele gegeven zal worden. Het veldje `expression` moet een object zijn dat een functieoproep voorstelt (zie hierna). Ook is er de mogelijk om een optioneel type mee te geven; in eenvoudige gevallen kan de judge dit afleiden, maar bij complexere gevallen niet meer. Dit type moet een van de ondersteunde types zijn uit het serialisatieformaat, dat hiervoor beschreven werd. Een functie ziet er als volgt uit:

```
{
  "type": "Soort functie",
  "name": "Naam van de functie",
  "object": "Optioneel object bij de functie",
  "arguments": ["Lijst van argumenten"]
}
```

Het type van de functie geeft aan welk soort functie het is. Mogelijke waarden zijn momenteel `top`, `object`, `constructor` en `identity`. De laatste soort is een speciaal geval, waarbij geen functienaam moet gegeven worden en slechts één argument toegelaten is. Die functie zal dan dat ene argument teruggeven. De naam van de functie benoemt eenvoudig welke functie opgeroepen moet worden. Het object van de functie laat toe om functies op objecten op te roepen. De lijst van argumenten kan nul of meer waarden bevatten. Deze waarden moeten in het formaat zijn zoals aangegeven in het vorige deel over de serialisatie van waarden.

3.3. Uitvoeren van de oplossing

Nadat de student een oplossing heeft ingediend en de judge is opgestart, begint de evaluatie van de oplossing. Eerst wordt de uit te voeren code gegenereerd, waarna de uitvoering van die code volgt.

3.3.1. Genereren van code

Het genereren van de code gebeurt met een sjabloonsysteem, genaamd Mako (Bayer e.a. 2020). Dit sjabloonsysteem wordt traditioneel gebruikt bij webapplicaties (zoals Ruby on Rails met ERB, Phoenix met EEX, Laravel met Blade, enz.) om een html-pagina te genereren. In ons geval zijn de sjablonen verantwoordelijk voor de vertaling van taalonafhankelijke concepten naar implementaties in specifieke talen. Voorbeelden hiervan zijn functieoproepen, toekennen van variabelen, enz.

Ook zijn de sjablonen verantwoordelijk voor het genereren van de code die de oplossing van de student zal oproepen en evalueren.

Voor elke ondersteunde programmeertaal dienen een aantal sjablonen geïmplementeerd te worden, die elk een functie voor zich nemen.

TODO blablabla

3.3.2. Uitvoeren van de code

Nadat alle code voorhanden is, wordt deze per context uit het testplan uitgevoerd en wordt de uitvoer verzameld. Het uitvoeren zelf gebeurt op de normale manier dat de programmeertaal uitgevoerd wordt via de commandoregel. Dit is een voordeel van onze aanpak: er is geen verschil tussen hoe de judge de code van de student uitvoert en hoe de student zijn code zelf uitvoert op zijn eigen computer. Dit voorkomt dat er subtiele verschillen zouden insluipen.

3.4. Evalueren van een oplossing

Na de uitvoering van elke context heeft de judge alle relevant uitvoer verzamelt, zoals de standaardkanalen. Deze uitvoer moet vervolgens beoordeeld worden om na te gaan in hoeverre deze uitvoer voldoet aan de verwachte uitvoer.

4. CASE-STUDY: TOEVOEGEN VAN EEN TAAL

Allerlei uitleg

5. BEPERKINGEN EN TOEKOMSTIG WERK

Wat kunnen we al en vooral wat niet? Waar kan nog aan gewerkt worden?

Korte samenvatting

5.1. Performance

-> Uitleg over eerste implementatie met jupyter kernels -> Uitleg over verschillende stadia van codegeneratie (alles apart -> zoveel mogelijk samen)

5.2. Functies

-> Dynamisch testplan -> Herhaalde uitvoeringen

A. SPECIFICATIE VAN HET SERIALISATIEFORMAAT

Hier komt een bestand.

BIBLIOGRAFIE

Amazon Ion (15 januari 2020). Amazon. URL: https://amzn.github.io/ion-docs/ (bezocht op 27-01-2020).

Apache Avro™ 1.9.1 Documentation (9 februari 2019). The Apache Foundation. URL: http://avro.apache.org/docs/1.9.1/ (bezocht op 27-01-2020).

Apache Parquet (13 januari 2020). The Apache Foundation. URL: https://parquet.apache.org/documentation/latest/ (bezocht op 27-01-2020).

Bayer, Michael e.a. (20 januari 2020). *Mako Templates for Python*. URL: https://www.makotemplates.org/.

Bormann, Carstenn en Paul Hoffman (oktober 2013). *Concise Binary Object Representation (CBOR)*. RFC 7049. Internet Engineering Task Force. URL: https://tools.ietf.org/html/rfc7049.

BSON 1.1 (19 juli 2019). MongoDB. URL: http://bsonspec.org/ (bezocht op 17-01-2020).

Cohen, Bram (4 februari 2017). *The BitTorrent Protocol Specification*. URL: http://bittorrent.org/beps/bep_0003.html.

Dodona-team (23 januari 2020). *Creating a new Judge*. Universiteit Gent. URL: https://dodona-edu.github.io/en/guides/creating-a-judge/.

Furuhashi, Sadayuki (17 september 2018). *MessagePack*. URL: https://msgpack.org/.

Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation (augustus 2015). Recommendation X.608. International Telecommunications Union. URL: https://www.itu.int/rec/T-REC-X.680-201508-I/en.

Information technology – Generic applications of ASN.1: Fast infoset (14 mei 2005). Recommendation X.881. International Telecommunications Union. URL: https://www.itu.int/rec/T-REC-X.891-200505-I/en.

Jackson JSON team (2010). *Smile Data Format*. URL: https://github.com/FasterXML/smile-format-specification.

Lengyel, Eric (17 januari 2017). *Open Data Description Language (OpenDDL)*. URL: http://openddl.org/.

Mitra, Nilo en Yves Lafon (april 2007). *SOAP Version 1.2 Part 0: Primer (Second Edition)*. TR. http://www.w3.org/TR soap12-parto-20070427/. W3C.

OGDL 2018.2 (25 februari 2018). URL: https://msgpack.org/ (bezocht op 28-01-2020).

Oortmerssen, Wouter van (24 april 2019). *FlatBuffers*. Google. URL: https://google.github.io/flatbuffers/.

OPC unified architecture - Part 1: Overview and concepts (10 mei 2016). IEC TR 62541-1:2016. International Electrotechnical Commission. URL: https://webstore.iec.ch/publication/25997.

ProtocolBuffers (13 december 2019). Google. URL: https://developers.google.com/protocol-buffers/.

Ramos, Bernardo (25 september 2019). *Binn*. URL: https://github.com/liteserver/binn.

Slee, Mark, Aditya Agarwal en Marc Kwiatkowski (2007). „Thrift: Scalable cross-language services implementation". In: URL: http://thrift.apache.org/static/files/thrift-20070401.pdf.

Universal Binary JSON (25 februari 2018). URL: http://ubjson.org/.

Wikipedia-bijdragers (25 januari 2020). *Comparison of data-serialization formats. Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/w/index.php?title=Comparison_of_data-serialization_formats&oldid=937433197.

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