



Faculteit Toegepaste Wetenschappen  
Vakgroep Informatietechnologie  
Voorzitter: Prof. Dr. Ir. P. LAGASSE

# **Onderzoek naar Interpretatie van COBOL Applicaties op de JVM**

door

Stijn BAES

Promotor: Prof. Dr. Ir. H. TROMP  
Scriptiebegeleider: K. DE SCHUTTER

Scriptie ingediend tot het behalen van de academische graad van  
licentiaat in de informatica

Academiejaar 2004–2005

# Voorwoord

Een combinatie van COBOL en de JVM. Geen hot item zou je zeggen, niets is minder waar. COBOL wordt nog steeds op grote schaal wereldwijd gebruikt in de bank- en verzekeringssector. Er is echter een groot probleem met COBOL, de kennis over de taal is met uitsterven bedreigd. Door het werken met COBOL en de diepere kennis van de JVM, wordt mijn marktwaarde als beginnend informaticus zeker verhoogd. Een bedanking van de mensen die dit mogelijk hebben gemaakt is hier dus op zijn plaats.

In de eerste plaats wil ik mijn promotor Professor Herman Tromp en mijn begeleider Kris De Schutter bedanken voor het aanbieden van dit interessante onderwerp. Het onderzoek zorgt voor een mooie aanvulling en toepassing van mijn genoten opleiding.

Ook mijn ouders wil ik bedanken, zonder hen zouden ik, m'n broers en zus nooit de kansen hebben gekregen waarvan we nu kunnen genieten.

Naast studeren en werken aan de scriptie, moest er ook tijd zijn voor ontspanning, waarvoor dank aan al m'n vrienden. In het bijzonder Dieter, om stukken van het werk te herlezen.

Stijn Baes, mei 2005

# Toelating tot bruikleen

“De auteur geeft de toelating deze scriptie voor consultatie beschikbaar te stellen en delen van de scriptie te kopiëren voor persoonlijk gebruik.

Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze scriptie.”

Stijn Baes, mei 2005

# Onderzoek naar Interpretatie van COBOL Applicaties op de JVM

door

Stijn BAES

Scriptie ingediend tot het behalen van de academische graad van  
licentiaat in de informatica

Academiejaar 2004–2005

Promotor: Prof. Dr. Ir. H. TROMP  
Scriptiebegeleider: K. DE SCHUTTER  
Faculteit Toegepaste Wetenschappen  
Universiteit Gent

Vakgroep Informatietechnologie  
Voorzitter: Prof. Dr. Ir. P. LAGASSE

## Samenvatting

De scriptie onderzoekt hoe COBOL kan toegepast worden op de JVM. Hiervoor worden types in COBOL omgezet naar types voor de JVM en statements in COBOL gemapt naar een combinatie van JVM opcodes. Het aanmaken van Java bytecode vanuit COBOL sourcecode wordt bekomen door het schrijven van een bijhorende applicatie.

## Trefwoorden

COBOL, JVM, Java, bytecode, statement, flow control

# Inhoudsopgave

<b>1</b>	<b>Inleiding</b>	<b>1</b>
1.1	Situering . . . . .	1
1.2	Doelstelling . . . . .	1
1.3	Opbouw van de thesis . . . . .	2
1.4	Gebruikte afkortingen . . . . .	2
<b>2</b>	<b>Gebruikte technologieën</b>	<b>3</b>
2.1	COBOL (Common Business Oriented Language) . . . . .	3
2.1.1	Geschiedenis van COBOL . . . . .	3
2.1.2	Structuur van een COBOL applicatie . . . . .	5
2.2	Java bytecode . . . . .	5
2.2.1	Geschiedenis van Java . . . . .	5
2.2.2	De JVM (Java Virtual Machine) . . . . .	6
2.2.3	Types en waardenbereik . . . . .	7
2.2.4	Stack, registers en geheugengebruik . . . . .	9
2.2.5	JVM bytecodes . . . . .	11
2.3	Jasmin . . . . .	13
2.4	DOM en XPath . . . . .	14
<b>3</b>	<b>Mapping van variabelen naar bytecode</b>	<b>16</b>
3.1	Variabelen in COBOL . . . . .	16
3.2	Tekstuele data . . . . .	18
3.3	Numerieke data . . . . .	20
3.4	Numeriek-edited data . . . . .	22
3.5	Records . . . . .	22

---

3.6	Booleans . . . . .	24
3.7	Implementatie van de variabelen transformatie . . . . .	26
3.7.1	XML omvorming van de DATA DIVISION . . . . .	26
3.7.2	Bijhouden en omzetten van de variabelen . . . . .	30
<b>4</b>	<b>Mapping van statements naar bytecode</b>	<b>36</b>
4.1	Overzicht statements en hun mapping . . . . .	36
4.1.1	DISPLAY . . . . .	36
4.1.2	ACCEPT . . . . .	38
4.1.3	COMPUTE . . . . .	39
4.1.4	SUBTRACT . . . . .	43
4.2	Toepassing op de applicatie . . . . .	44
<b>5</b>	<b>Flow control</b>	<b>46</b>
5.1	Flow control in bytecode . . . . .	46
5.1.1	Onvoorwaardelijke sprong . . . . .	46
5.1.2	Voorwaardelijke sprong . . . . .	46
5.1.3	Terugkerende sprong . . . . .	47
5.2	Flow control in COBOL . . . . .	47
5.2.1	IF ELSE . . . . .	47
5.2.2	PERFORM . . . . .	50
5.3	Implementatie . . . . .	51
<b>6</b>	<b>Een uitgewerkt voorbeeld</b>	<b>55</b>
6.1	Het MOVE statement . . . . .	55
6.1.1	MOVE statement in COBOL . . . . .	55
6.1.2	Equivalent in Java bytecode . . . . .	58
6.1.3	Toepassing op programma . . . . .	61
<b>7</b>	<b>Conclusies en verder onderzoek</b>	<b>64</b>
<b>A</b>	<b>Uitvoeren van de applicatie</b>	<b>66</b>
<b>B</b>	<b>COBOL op een mobiel toestel</b>	<b>70</b>

# Hoofdstuk 1

## Inleiding

### 1.1 Situering

Java bytecode is een intermediaire taal tussen de JVM en sourcecode. De bekende binaire voorstelling van de bytecode is omvat in `.class` bestanden. Java bytecode is ontworpen voor de taal Java door Sun Microsystems. Naarmate het groeien aan populariteit van Java worden er ook tools ontwikkeld om andere sourcecode om te zetten voor de JVM [1, 2]. Intermediaire talen worden veel gebruikt door moderne compilers. Typisch worden ze aangemaakt door een compiler front-end (parsen en vertalen) en gebruikt door de back-end (code generatie voor een bepaalde machine architectuur). Intermediaire talen vergemakkelijken het proces om een compiler te porten naar een andere architectuur door de front-end te behouden.

COBOL is een van de eerste high-level programmeertalen, ontwikkeld in 1959. Naar schatting zijn nog steeds tussen de 60% en 80% van applicaties in gebruik, geschreven in COBOL. Nog steeds worden er COBOL applicaties ontworpen en wordt de omgeving uitgebreid. Deze thesis combineert een oude programmeertaal met een recent platform.

### 1.2 Doelstelling

De doelstelling van deze thesis is nagaan of het omzetten van COBOL sourcecode naar Java bytecode mogelijk is. Er werd een applicatie ontwikkeld die COBOL variabelen inleest en omzet naar Java bytecode. Ook de COBOL statements worden omgezet door ze één voor één te verwerken. De applicatie is ontwikkeld in Java, met behulp van de Eclipse omgeving.

## 1.3 Opbouw van de thesis

In een inleidend hoofdstuk, gaande over de gebruikte technologieën, wordt aangetoond wat COBOL en Java bytecode is. Voor Java bytecode wordt de JVM wat dieper bekenen, dit geeft de werking en de types van de JVM terug. Hier worden ook enkele vaak gebruikte opcodes van de JVM uitgelegd.

Daarna wordt aangetoond hoe de omzetting gebeurt van COBOL applicaties naar applicaties voor de JVM. Dit wordt onderverdeeld in het verwerken van de variabelen en de statements. Het eerste deel zal verduidelijken welke types een COBOL applicatie kan bevatten en hoe deze types worden voorgesteld op de JVM. Daarnaast wordt ook aangetoond hoe de variabelen van een COBOL applicatie bewaard worden. De behandelde statements worden stuk voor stuk besproken. Statements die een invloed hebben op flow control worden in een afzonderlijk hoofdstuk behandeld. Na de mapping van de statements te hebben doorlopen wordt er een voorbeeld gegeven hoe de ontworpen applicatie kan uitgebreid worden met de mapping van een nieuw statement.

## 1.4 Gebruikte afkortingen

COBOL (Common Business Oriented Language)

JVM (Java Virtual Machine)

XML (Extensible Markup Language)

DOM (Document Object Model)

XPATH (XML Path Language)

EBCDIC (Extended Binary Coded Decimal Interchange Code)

ASCII (American National Standard Code for Information Interchange)

CODASYL (Conference on Data Systems Languages)

ANSI (American National Standards Institute)

ISO (International Organization for Standardization)

PC (Program Counter)

EBNF (Extended Backus-Naur Form)

UML (Unified Modeling Language)

NaN (Not a Number)

## Hoofdstuk 2

# Gebruikte technologieën

## 2.1 COBOL (Common Business Oriented Language)

### 2.1.1 Geschiedenis van COBOL

COBOL of de Common Business Oriented Language was één van de eerste hoog niveau programmeertalen. De taal is ontworpen in 1959 door een groep van vertegenwoordigers uit verschillende organisaties, waaronder de overheid, consultants, universiteiten en ontwikkelaars van computersystemen. De groep is bekend als de Conference on Data Systems Languages of CODASYL. Het team stond voor de taak om een gemeenschappelijke business programmeertaal te ontwerpen. Na slechts 6 maanden waren de specificaties voor COBOL compleet. Deze specificaties werden na goedkeuring uitgebracht als COBOL-60. Sindsdien heeft COBOL echter nog grote veranderingen en uitbreidingen ondergaan. In tabel 2.1 staat een overzicht van de belangrijkste ontwikkelingen van COBOL tot op vandaag.

Door een groeiende populariteit van COBOL kwamen er dialecten van de taal in omloop. De Amerikaanse overheid droeg zijn steentje bij aan de initiële populariteit van COBOL, doordat aangekochte computers, COBOL software beschikbaar moesten hebben. Om de verspreiding van verschillende dialecten tegen te gaan werden de ANSI standaarden uitgewerkt.

COBOL applicaties kunnen zeer groot zijn, typisch meer dan 1 miljoen lijnen code. Door grote investeringen in het ontwerpen van COBOL applicaties met een groot aantal lijnen code, hebben deze een lange levensduur. Dit komt tevens door het milieu waarin COBOL applicaties worden toegepast, ze worden veelal gebruikt in kritische toepassingen. Zo wordt onder andere 95% van de data in de bank- en verzekeringswereld verwerkt mbv. COBOL. Deze applicaties kunnen niet zomaar genegeerd worden wanneer een nieuwe programmeertaal z'n intrede doet.

Jaar	Opmerking
1959	Het Amerikaanse Department of Defense (DOD) vraagt een groep specialisten om een business taal te ontwikkelen.
1960	Een eerste voorstel voor COBOL, genoemd COBOL-60. Het American National Standards Institute (ANSI) wordt verantwoordelijk voor het ontwikkelen van COBOL standaarden.
1961	De eerste compilers worden in gebruik genomen.
1965	Het gebruik van COBOL neemt sterk toe.
1968	ANSI stelt de eerste officiële COBOL standaard voor: COBOL-68.
1970	De International Organization for Standardization (ISO) maakt ANSI's COBOL-68 een internationale standaard.
1974	COBOL-74 wordt als nieuwe standaard voorgesteld.
1985	COBOL-85 wordt als nieuwe standaard voorgesteld.
1989	Intrinsieke functies worden aan de standaard toegevoegd.
2002	Toevoeging van object oriëntatie aan COBOL door het invoeren van de COBOL 2002 standaard.

Tabel 2.1: De mijlpalen in de geschiedenis van COBOL

### 2.1.2 Structuur van een COBOL applicatie

COBOL programma's bezitten een hiërarchische structuur. Elk element van de hiërarchie bestaat uit één of meer onderliggende elementen. De verschillende niveaus van de hiërarchie zijn Divisions, Sections, Paragraphs, Sentences en Statements.

Er zijn 4 hoofddivisies en elke divisie zorgt voor een stuk van de informatie nodig voor de compiler. De volgorde van de divisies ligt vast en wordt verduidelijkt door figuur 2.1. De IDENTIFICATION DIVISION geeft informatie over het programma voor de programmeur en de compiler. De ENVIRONMENT DIVISION wordt gebruikt om de omgeving waar het programma wordt uitgevoerd te beschrijven. De DATA DIVISION geeft een beschrijving van de data-items die gebruikt worden in het programma, de variabelen van een applicatie. De PROCEDURE DIVISION bevat de code die de data-items beschreven in de DATA DIVISION verwerkt. Het is hier dat de programmeur z'n algoritme uitwerkt met behulp van statements.

```
000100 IDENTIFICATION DIVISION.  
000200 PROGRAM-ID. HELLOWORLD.  
000300  
000400 ENVIRONMENT DIVISION.  
000500  
000600 DATA DIVISION.  
000700     77 HELLOSTR PIC X(20) VALUE "Hello World".  
000800  
000900 PROCEDURE DIVISION.  
001000  
001100 MAIN.  
001200     DISPLAY HELLOSTR.  
001300 STOP RUN.
```

Figuur 2.1: Een 'Hello World' - voorbeeld.

## 2.2 Java bytecode

### 2.2.1 Geschiedenis van Java

Rond 1990 begonnen James Gosling, Bill Joy en anderen bij Sun Microsystems aan de ontwikkeling van de taal Oak. Ze wilden de taal gebruiken om microprocessorsen aan te spreken bij ingebouwde consumentenartikelen. Om aan de voorwaarde van een embedded taal te voldoen, moest Oak platformonafhankelijk, zeer betrouwbaar en compact zijn. Door het succes van het

internet veranderde Sun van strategie en ging Oak gebruiken voor internetapplicaties. De naam van de taal veranderde naar Java.

Java is nu vooral populair bij applicaties zoals online webwinkels, transactie processing en database interfaces. Ook op de mobiele telefonie markt heeft Java zijn plaats weten in te nemen, door het gebruik van GSM-software, geschreven in Java. Tabel 2.2 geeft een overzicht van de versies van de Java Standard Edition, sinds het ontstaan van Java.

Jaar	Versie	Aantal packages	Aantal klassen	Opmerking
1995	1.0	8	212	Java gelicentieerd door Microsoft en andere bedrijven
1997	1.1	23	504	verbeterde JVM; Microsoft ontwikkelde een eigen 1.1 compatibele JVM; introductie van Swing
1999	1.2 ook Java2 Platform genoemd	59	1520	code en tools uitgebracht als de 'Software Development Kit (SDK)'  ondersteuning voor verschillende lijsten, verzamelingen en hash maps
2000	1.3	76	1842	verbetering van performantie
2002	1.4	135	2991	verbeterde IO en XML ondersteuning
2004	5.0 (voordien 1.5)	165	+3000	sneller opstarten; kleiner geheugengebruik

Tabel 2.2: Versies van het Java platform

Andere specifieke edities van Java zijn de Micro Edition (J2ME) en de Enterprise Edition (J2EE).

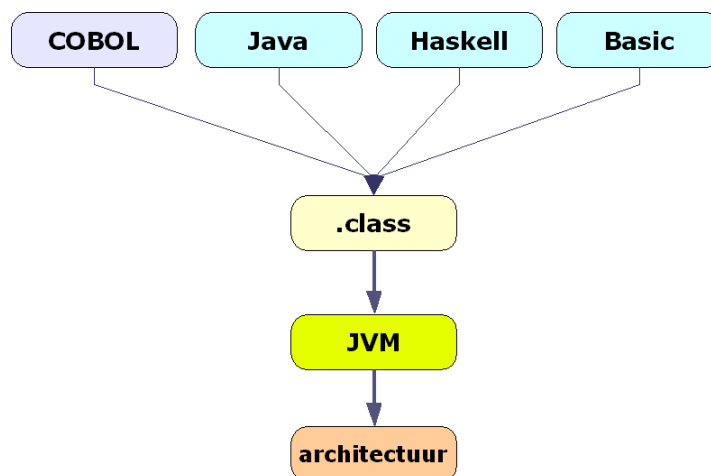
### 2.2.2 De JVM (Java Virtual Machine)

#### Wat is de JVM

De JVM [3] is de belangrijkste component van Java en het Java 2 platform. Het is verantwoordelijk voor de overdraagbaarheid van gecompileerde code naar andere architecturen.

De JVM is een abstracte machine met zijn eigen instructieset en geheugengebruik. Het zorgt voor de back-end van een compiler. De JVM staat onafhankelijk van de Java programmeertaal, er wordt enkel een binair formaat, het `.class` bestand gebruikt. Een `.class` bestand bevat virtuele machine instructies, bytecodes genoemd.

Voor de veiligheid stelt de JVM strenge structurele beperkingen aan een `.class` bestand. Toch kunnen ook andere talen buiten Java gebruik maken van de JVM, zolang deze om te zetten zijn naar een geldig `.class` bestand. Zo kan er op een relatief eenvoudige manier gezorgd worden voor systeemafhankelijkheid.



Figuur 2.2: Gebruik van de JVM als back-end

### 2.2.3 Types en waardenbereik

De JVM bezit 2 soorten types, primitieve types en referentie types. De bijhorende waarden kunnen worden gebruikt voor het opslaan in variabelen, het doorgeven als argumenten, het teruggeven door methodes of het uitvoeren van bewerkingen. De instructieset van de JVM bevat voor elke bewerking een reeks opcodes die gebruik maken van de verschillende primitieve types. Zo zijn er voor de vermenigvuldiging van 2 getallen 4 verschillende instructies: `imul`, `lmul`, `fmul` en `dmul`. Rechtstreekse ondersteuning voor een boolean, byte en short zijn er niet, deze worden intern eerst omgezet naar het int type. Ondersteuning voor objecten en arrays, zit ingebakken in de JVM als een referentie type. Een referentie kan gezien worden als een pointer naar een object. Tabel 2.3 geeft een overzicht van de beschikbare types voor de JVM.

Type	Voorstelling	Bereik	Opmerking
byte	8-bit signed integer	$-2^{(8-1)}$ tot $2^{(8-1)-1}$	
short	16-bit signed integer	$-2^{(16-1)}$ tot $2^{(16-1)-1}$	
int	32-bit signed integer	$-2^{(32-1)}$ tot $2^{(32-1)-1}$	optelling: <code>iadd</code>
long	64-bit signed integer	$-2^{(64-1)}$ tot $2^{(64-1)-1}$	optelling: <code>ladd</code>
char	16-bit unsigned integer	Unicode karakter, 0 tot 65535	
float	32-bit single-precision		optelling: <code>fadd</code> ; NaN wordt gebruikt als uitkomst van foute bewerkingen zoals delen door 0
double	64-bit double-precision		optelling: <code>dadd</code> ; NaN wordt gebruikt als uitkomst van foute bewerkingen zoals delen door 0
boolean	true en false	0 en 1	er zijn geen instructies op het boolean type, de instructies van het type <code>int</code> moeten gebruikt worden; 1 stelt true voor en 0 false
returnAddress	pointer naar opcodes	32-bit adres	wordt gebruikt door de JVM instructies <code>jsr</code> , <code>ret</code> en <code>jsr_w</code>
referentie	er zijn 3 verschillende types: klasse types, array types en interface types	32-bit pointer naar object	een speciale waarde is <code>null</code> , een referentie naar 'geen' object

Tabel 2.3: Types beschikbaar voor de JVM

### 2.2.4 Stack, registers en geheugengebruik

De stack van de JVM bestaat uit verschillende stack frames. Telkens een methode wordt opgeroepen, wordt er een nieuw stack frame aangemaakt. Een stack frame bestaat uit een operand stack, een execution environment en een rij van lokale variabelen. Bij het oproepen van een methode wordt het nieuw aangemaakt stack frame boven op de stack geplaatst, dit is dan het actief stack frame. Bij het beëindigen van een methode wordt het actief stack frame gepopt.

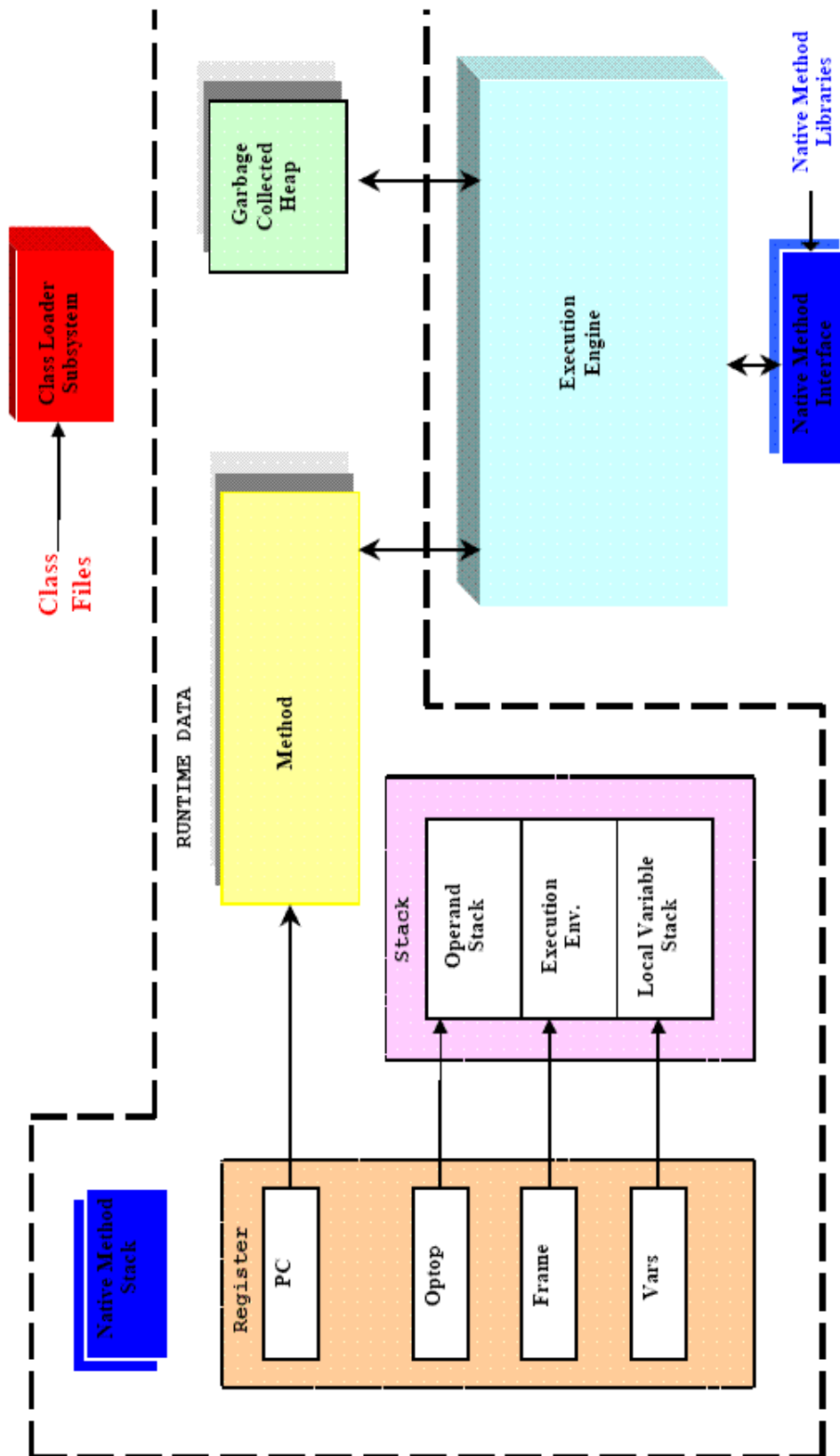
De operand stack wordt gebruikt door de bytecode instructies, hier moeten de operanden voor de opcodes staan, wanneer deze worden uitgevoerd. Het is ook hier dat de uitkomst van een instructie zich zal bevinden. De execution environment wordt gebruikt om de operaties van de stack zelf te onderhouden. De rij van lokale variabelen is de plaats waar alle variabelen van de methode worden opgeslagen. Er kan gebruik gemaakt worden van de instructie `load`, om waarden van de rij van lokale variabelen op de operand stack te plaatsen of `store` om waarden van de stack op te slaan in de lokale variabelen. Figuur 2.3 verduidelijkt dit.

De JVM bezit slechts vier registers. Deze kunnen niet worden gebruikt om bewerkingen mee uit te voeren maar worden intern door de JVM benut om het programmaverloop te bepalen en als pointer naar onderdelen van het actief stack frame. Dit maakt de JVM een stackgebaseerde architectuur.

Het Program Counter (PC) register wijst een instructie aan in de method area. De method area is de plaats in het geheugen waar de bytecode zich bevindt. Het is een pointer naar de momenteel uitgevoerde instructie van de uitgevoerde methode. Nadat een bytecode instructie is uitgevoerd, zal de PC het adres van de volgende uit te voeren instructie bevatten. Elke JVM thread heeft een eigen PC.

De 3 andere registers worden gebruikt voor operaties op de stack. Het Optop register wijst naar het eerste element van de operand stack. Het Frame register wijst naar het eerste element van de execution environment. De lokale variabelen worden geadresseerd door middel van het Vars register.

Objecten en arrays worden opgeslaan op de heap. Elke referentie naar een object of een array is dus een referentie naar een geheugenplaats van de heap. Het is niet nodig om zelf expliciet geheugen vrij te geven. De JVM bezit een garbage collection algoritme om het gebruikte geheugen van onnodige objecten vrij te geven. Er is slechts 1 heap per JVM. Doordat een



Figuur 2.3: De virtuele architectuur van de JVM

adresreferentie in de JVM 32 bits groot is, kan er per JVM 4 gigabytes aan geheugen worden geadresseerd. De stack, de heap en de method area bevinden zich in deze 4 gigabytes. De exacte plaats in het geheugen van deze onderdelen hangt af van de implementatie van de JVM.

### 2.2.5 JVM bytecodes

Bytecodes definiëren de instructieverzameling die zorgt voor de uitvoering van een applicatie. Als een JVM een `.class` bestand laadt, krijgt het per methode een reeks instructies. De instructies noemen we bytecodes omdat de instructies byte-gealigneerd zijn in de method area. Alle stackgebaseerde instructies bestaan uit een 1-byte opcode, gevolgd door 0 of meer parameters van 1-byte. De verzameling van instructies is compact, er zijn 200 standaard opcodes gedefinieerd in de JVM, 25 snellere versies van opcodes en 3 gereserveerd. De opcodes maken duidelijk aan de JVM welke actie moet gebeuren.

Alle berekeningen van de JVM gebeuren op de stack. De JVM heeft geen extra registers om tijdelijke berekeningen te bewaren. Opcodes zijn een tekstuele representatie van bytecode. Bytecodes zijn de binaire representatie van instructies. Zo staat de bytecode `hex(68)` voor de `imul` opcode en `hex(36 04)` voor `istore 4`.

Nu gaan we wat dieper in op verschillende vaak gebruikte opcodes. Er bestaan verschillende manieren om constante waarden op de stack te plaatsen. De constante waarde kan impliciet vermeld zijn in de opcode zelf, kan de opcode volgen als een parameter of de waarde kan genomen worden uit de constant pool.

Sommige opcodes definiëren een type en een constante te pushen waarde. Bijvoorbeeld zal de `iconst_1` opcode staan voor het pushen van een integer met waarde 1 op de stack. Voor veel voorkomende waarden van verschillende types bestaan er zo'n bytecodes. Deze instructies zijn enkel 1 byte lang en zorgen dus voor efficiënte bytecode. Een ander voorbeeld van zo'n opcode is `dconst_0` die een double met waarde 0.0 op de stack plaatst. Merk op dat het pushen van een double 2 plaatsen op de stack zal verbruiken, doordat de woordlengte van de stack slechts 32-bit groot is en een double 64-bit. Een bijzondere opcode is `aconst_null`, deze opcode plaatst een referentie op de stack met waarde null. De 'a' in de opcode slaat op het referentie type.

Er zijn 2 opcodes beschikbaar die een waarde meegekregen als parameter op de stack plaatsen. Dit zijn `bipush` en `sipush`. De `bipush` opcode heeft 1 parameter, `bipush` wordt gebruikt voor waarden die kunnen bevat worden in 1 byte. De te pushen byte wordt eerst omgezet naar een

int type omdat de stack namelijk enkel 32 bits waarden kan bevatten. De `sipush` opcode neemt 2 parameters van elk 1 byte. De maximale waarde die kan gepusht worden met `sipush` is de maximale waarde van het short type. Ook hier worden de 2 bytes eerst omgezet naar 1 int vooraleer op de stack te plaatsen.

Drie opcodes pushen constante waarden uit de constant pool. De constant pool is een stuk van de header van een `.class` bestand. Hier worden waarden (eventueel gecodeerd) opgeslagen die gebruikt worden door de bytecodes. De constantpool kan gezien worden als een rij van waarden. Deze waarden in de constantpool moeten echter niet bytegealligneerd zijn zoals wel het geval is met de bytестroom zelf. De opcodes zijn `ldc1`, `ldc2` en `ldc2w` en vragen 1 of 2 parameters. De parameters zijn geen waarden maar een indexbyte die verwijst naar de plaats in de constantpool waar deze waarden zich bevinden. De JVM zal aan de hand van de indexbyte de waarde opzoeken in de constantpool en op de stack plaatsen. Door het gebruik van Jasmin zullen we ons geen zorgen hoeven te maken over de constantpool, zie daarvoor verder.

Ook voor het pushen van lokale variabelen op de stack zijn er opcodes beschikbaar. Hier wordt een onderscheid gemaakt aan de hand van het type van de variabele die men op de stack wil plaatsen. De bijhorende parameter is de index in de rij van lokale variabelen waar de variabele zich bevindt. Zo zet `iload vindex` de variabele van het type int die zich op de locatie `vindex` bevindt, op de stack. De opcode `aload vindex` zorgt voor het pushen van een referentietype. Zoals bij het pushen van constante waarden zijn er ook opcodes die geen parameter vragen, zoals `iload_0`.

Het opslaan of poppen van waarden op de stack naar de lokale variabelen gebeurt op dezelfde manier als het pushen. In plaats van load wordt dan store gebruikt. Zo zorgt `fstore vindex` voor het poppen van een float waarde van de stack en het opslaan ervan in de rij van lokale variabelen met index `vindex`.

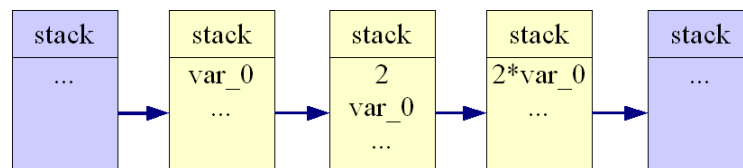
Een voorbeeld verduidelijkt dit. Als we een (int) variabele moeten verdubbelen in waarde, wordt er gebruik gemaakt van de volgende bytecode sequentie.

```
(1) iload_0
(2) iconst_2
(3) imul
(4) istore_0
```

1. De int waarde, die in de rij van lokale variabelen voorkomt op plaats 0 wordt op de stack

geduwd.

2. De constante int waarde 2 wordt op de stack geduwd.
3. Uitvoering van de int vermenigvuldiging. Deze haalt de 2 int waarden van de stack en duwt het resultaat op de stack.
4. Het resultaat wordt van de stack gehaald en opgeslagen op plaats 0 van de lokale variabelen van de methode.



Figuur 2.4: Voorstelling van de stack tijdens het uitvoeren

## 2.3 Jasmin

Jasmin [4] is een Java assembler tool voor de JVM. De tool zet bytcodes in tekstueel Jasmin formaat om in binaire `.class` bestanden. Jasmin is de eerste assembler tool die geschreven is voor de virtuele machine, Sun had zelf nog geen assembler formaat beschikbaar. Daardoor is Jasmin heel populair en de meest gebruikte assembler voor de JVM.

Het Jasmin formaat heeft een gemakkelijk te gebruiken syntax. Er is geen diepe kennis nodig over de JVM om applicaties te maken via Jasmin. Dit in tegenstelling tot het rechtstreeks aanpassen van een `.class` bestand. Er moet geen rekening gehouden worden met de constantpool. Het is Jasmin zelf die de constant pool opbouwt.

Er kan gebruik gemaakt worden van 1 instructie om waarden die normaal uit de constantpool moeten gehaald worden te duwen op de stack. Dit kan via `ldc`, deze instructie vraagt als parameter niet de index uit de constantpool, maar de waarde. Ook `String` objecten kan je zo op de stack plaatsen. Aangezien een `String` geen primitief type is, wordt er intern een `String` object aangemaakt en wordt de referentie naar het object op de stack geplaatst.

Het gemak in gebruik van Jasmin is niet enkel door het nalaten van de constantpool maar ook omdat bij het springen naar een ander stuk code, enkel het label moet gegeven worden,

waarnaar er moet worden gesprongen. De opcode `goto` vraagt normaal een index in de method area waar de instructies staan, waarnaar moet worden gesprongen. In Jasmin kan je `goto` samen met een label gebruiken waarnaar moet worden gesprongen.

Om beter inzicht te krijgen in het gebruik van Jasmin, heb ik gebruik gemaakt van D-java. D-java [5] is een `.class` disassembler. Deze zet binaire `.class` bestanden om in Jasmin-formaat.

## 2.4 DOM en XPath

Het omzetten van COBOL code gebeurt door deze eerst om te zetten naar een XML bestand. Een XML bestand wordt aangemaakt door een intern in INTEC ontworpen tool. Deze tool zorgt voor het parsen van de sourcecode. Het begin van elke compiler.

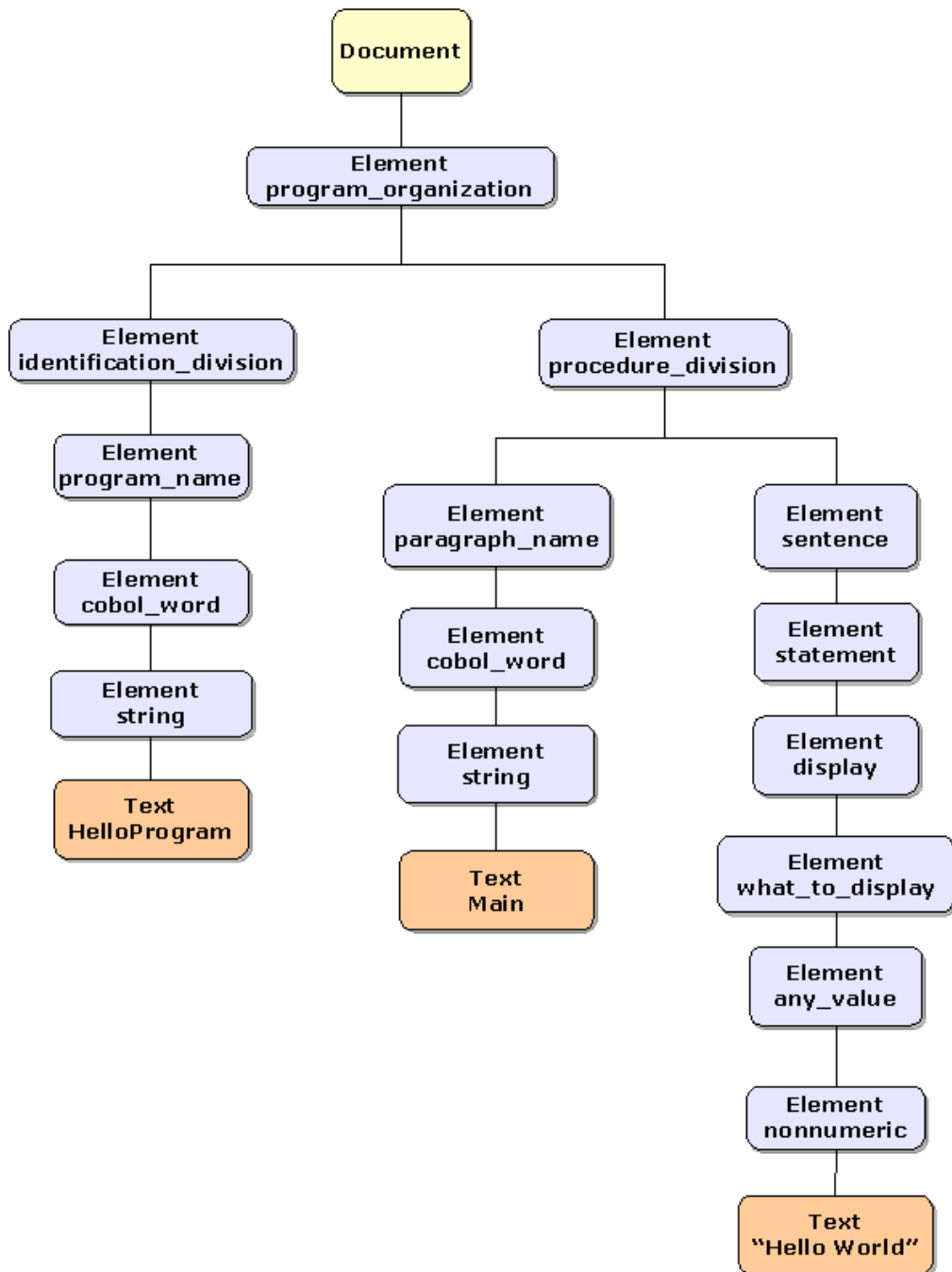
Er zijn verschillende manieren om een XML bestand te lezen, ik heb gekozen voor DOM. DOM of het Document Object Model is een platform en taal neutrale interface die programma's en scripts toelaat om dynamisch XML bestanden te lezen en te wijzigen. DOM maakt een boomstructuur van een XML bestand en beschouwt elk onderdeel als een node. Ik heb gebruik gemaakt van de standaard DOM implementatie in Java.

Op Figuur 2.5 zie je de boomstructuur van een 'Hello world' applicatie. Het is niet altijd gemakkelijk om een element aan te spreken door enkel DOM te gebruiken. Als men in het voorbeeld van het 'Hello World' programma de uit te printen tekst wil te weten komen, is dit namelijk nogal omslachtig. Indien men bezig is met het processen van de `display` node, moeten eerst de nodes `what_to_display`, `any_value` en `nonnumeric` behandeld worden, vooraleer de `string` node met als inhoud "Hello World" te verkrijgen is. Een hulpmiddel hiervoor is het gebruik van XPath expressies.

XPath is een taal om bepaalde nodes in een XML document te adresseren. Het is gemakkelijk in gebruik en nodes kunnen relatief ten opzichte van een huidige node of vanaf het begin van het XML document verkregen worden. In het 'Hello World' voorbeeld bekom je de uit te printen tekst door het evalueren van de XPath expressie

```
what_to_display/any_value/nonnumeric/string::text()
```

in plaats van elke node zelf eerst te moeten overlopen.



Figuur 2.5: 'Hello World' als DOM document

## Hoofdstuk 3

# Mapping van variabelen naar bytecode

### 3.1 Variabelen in COBOL

De initialisatie van alle variabelen in een COBOL-programma gebeurt in de `DATA DIVISION`, de data kan verwerkt worden in de `PROCEDURE DIVISION`. De datatypes worden beschreven door een levelnummer, een data-name en een picture clause. De syntax van een elementair data-item is:

```
level-number data-name PIC character-string VALUE value
```

#### Het levelnummer

Het levelnummer van een data-item dient om de relatie tussen elementaire items en group items weer te geven. Een group-item is een hiërarchische groepering van gerelateerde dataitems. Elk element van een group-item is een onderliggende group-item of een elementair data-item. Een group-item kan op die manier gezien worden als een record van elementaire data-items. Data-items kunnen als levelnummer een getal hebben van 01 tot 49, 66, 77 of 88. Data-items onmiddellijk onder een group-item moeten hetzelfde levelnummer hebben dat numeriek groter is dan het levelnummer van het group-item. In figuur 3.1 is `DateOfBirth` een group-item, `Day`, `Month` en `Year` zijn elk elementaire Numerieke data-items.

Er zijn echter enkele speciale levelnummers, dit zijn level-66, level-77 en level-88. Een level-66 data-item beschrijft een data-item die hernoemt wordt via de 'rename-clause'. Een level-77 data-item is een alleenstaand, elementair data-item. Zo'n data-item kan geen andere data-items

```

01 DateOfBirth
  02 Day    PIC 99
  02 Month  PIC 99
  02 Year   PIC 9(4)

```

Figuur 3.1: DateOfBirth als group item.

meer bezitten. Een level-88 dataitem is een boolean data-item, deze zal een bepaalde waarde aannemen afhankelijk van de waarde van een conditionele variabele.

### De picture clause

Het gebruikte formaat om variabelen voor te stellen in COBOL is gebaseerd op het uiterlijk van een variabele bij het uitprinten. Het uiterlijk van een data-item wordt beschreven mbv. zo'n picture clause. Een picture clause kan enkel voor elementaire items worden gespecificeerd en niet voor group-items. De picture van een data-item is een karakterstring die enkele specifieke karakters kan bevatten. Een nummer tussen haakjes in de picture clause is een repetitie factor en indiceert het aantal keer het linker karakter moet herhaald worden. X(3) staat dus gelijk aan XXX. Aan de hand van de picture clause kunnen data-items worden onderverdeeld in verschillende categorieën. Er zijn 5 categorieën van data-items die gegroepeerd worden in 3 klassen: alfabetisch, numeriek en alfanumeriek. In tabel 3.1 staat de onderverdeling van de klassen in categorieën.

Type	Klasse	Categorie
Elementair	Alfabetisch	Alfabetisch
	Numeriek	Numeriek
	Alfanumeriek	Alfanumeriek Numeriek-Edited Alfanumeriek-Edited
Group	Alfanumeriek	Numeriek Alfabetisch Alfanumeriek Numeriek-Edited Alfanumeriek-Edited

Tabel 3.1: De verschillende COBOL categorieën

De opdeling van de data in categorieën op basis van de picture string gaat als volgt:

- Alfabetische data:

Een item is alfabetisch indien z'n picture enkel bestaat uit een sequentie van het symbool A. Een alfabetisch item kan letters uit het alfabet en spaties bevatten.

- Alf numerieke data:

Een item is alfanumeriek indien z'n picture bestaat uit een combinatie van de symbolen A,X en 9. (Enkel A's of 9's definieert geen alfanumerieke data.) Deze data kan zowel letters als cijfers bevatten.

- Alf numeriek-edited data:

Een item is alfanumeriek-edited indien de picture bestaat uit een combinatie van de A,X,9,B,0 en / symbolen. De picture moet minimum 1 A of X en minimum 1 B,0 of / bevatten.

- Numerieke data:

Een item is numeriek indien de picture bestaat uit enkel de symbolen 9,P,S en V. Het aantal digitale posities in de picture moet van 1 tot 18 zijn.

- Numeriek-edited data:

Een item is numeriek edited indien minstens 1 karakter in de picture string een B,/Z,0,,.,\*,+,-,CR,DB of \$ is.

Een overzicht van de meest gebruikte symbolen met hun betekenis en de types waarin ze kunnen voorkomen vind je in tabel 3.2.

## 3.2 Tekstuele data

De behandelde tekstuele data valt onder de categorieën alfabetisch en alfanumeriek. Dit wil zeggen dat de picture string bestaat uit de symbolen A,X en eventueel 9. Bij het mappen naar bytecode wordt de lengte van de picture string gebruikt om een array aan te maken met dezelfde lengte. De aangemaakte array bezit elementen van het type char. Een item met als picture string X(5) zal voorgesteld worden als een array van chars met lengte 5, met de meegegeven initiële waarde of spaties indien de waarde niet gespecificeerd werd. Het aanmaken van een char array kan mbv. het Jasmin formaat door de opcode `newarray char`. Het aantal elementen in de array moet wel eerst op de stack worden gezet door `bipush <lengte>`. Het invullen van een waarde in

Karakter	Betekenis	Type(s)	Voorbeeld
A	enkel letters, geen cijfers	Alfabetisch Alfanumeriek Alfanumeriek-edited	A(5)
X	alle karakters	Alfanumeriek Alfanumeriek-edited	X(5)
9	cijfer	Alfanumeriek Alfanumeriek-edited Numeriek Numeriek-edited	999
S	teken	Numeriek	S9(3)
V	veronderstelde decimale punt	Numeriek Numeriek-edited	S9(3)V99
Z	<ul style="list-style-type: none"> <li>○ onderdrukken van voorafgaande nullen</li> <li>○ spatie indien nul</li> </ul>	Numeriek-edited	ZZ9
,	toegevoegde komma	Numeriek-edited	ZZZ,Z99
.	werkelijke decimale punt	Numeriek-edited	ZZ,ZZZ.99
-	<ul style="list-style-type: none"> <li>○ minus teken indien negatief</li> <li>○ spatie indien positief</li> </ul>	Numeriek-edited	ZZ,ZZZ-

Tabel 3.2: Symbolen van de picture clause

de array kan via de sequentie `bipush <index>` , `bipush <waarde>` , `castore`. Een referentie naar de array opslaan in de rij van lokale variabelen doe je door `astore <index>`.

Het omzetten van `77 STATUS PIC XX VALUE OK` geeft:

```
bipush 2          ; de lengte van de array (2) op de stack plaatsen
newarray char    ; referentie naar een nieuwe array van chars

dup              ; dupliceert de top van de stack, nl. de referentie
bipush 0         ; de index in de array
bipush 79        ; de decimale waarde van het karakter '0'
castore          ; bewaart de spatie op plaats index in de array
                 ; castore poept daarvoor de 3 bovenste elementen

dup
bipush 1
bipush 75
castore

astore 1         ; referentie naar de array opslaan op index 1 van
                 ; de rij van lokale variabelen
```

Indien data wordt doorgegeven van een ander item moet deze data dezelfde lengte hebben als de picture string. Indien de data te kort is wordt de rest rechts opgevuld met spaties. Indien de data te lang is, wordt de data rechts afgeknot zodat ze in de array kan bevat worden. Dit wordt gesimuleerd door de methode

```
public static void setContent (char [] toThis, char [] setThis).
```

van de klasse `CobolString`.

### 3.3 Numerieke data

De picture string van de behandelde numerieke data kan enkel de elementen `9,S` en `V` bevatten. Het aantal cijfers van een geheel getal moet liggen tussen 1 en 18 inclusief.

Het vinden van een geschikte representatie voor numerieke data, is niet gemakkelijk. Er moet rekening gehouden worden met het decimaal afronden van een getal en het uitvoeren van berekeningen. Ik heb gekozen om ook de numerieke data voor te stellen als een array van chars op de JVM. Deze voorstelling zorgt voor een gemakkelijkere input en output, deze kan gebeuren door dezelfde methodes als voor tekstuele data.

Indien er moet gerekend worden met de data, moet deze omgezet worden in een ander type, beschikbaar voor de JVM. Een numeriek data-item met een `V` in de picture stelt een kommagetal voor, deze wordt door een zelfgeschreven methode omgezet in een `double` die op de stack wordt

geplaatst. Een geheel getal wordt omgezet in een int of een long, naar gelang de lengte van de picture. Een geheel getal tot 10 cijfers kan omgezet worden in een int, een getal met meer dan 10 cijfers wordt omgezet in een long. Tabel 3.5 geeft de beschikbare methodes.

De uitgeprinte vorm van een getal is standaard in het gebruikte ACUCOBOL [6] het EBCDIC formaat. Via het EBCDIC formaat wordt een getal als 1234 voorgesteld door de karakterstring 1234 of hexadecimaal F1F2F3F4. De eerste helft van de laatste byte in de hexadecimale representatie wordt gebruikt om het teken van een getal mee te geven. Typisch staat 0xC als representatie voor een positief getal, 0xD staat voor een negatief getal en 0xF dient zoals in het 1234 voorbeeld, voor een getal zonder teken.

Java maakt echter geen gebruik van het EBCDIC formaat voor het voorstellen van karakters, maar van Unicode, een extensie van ASCII. Een methode om een getal om te zetten in een ASCII karakterstring is gebruik maken van 'Sign EBCDIC Custom'. Hier wordt de karakterwaarde van een getal behouden, maar zal de hexadecimale representatie verschillen. In Tabel 3.3 zie je hoe deze omzetting kan gebeuren.

Cijfer	Positieve waarden	Negatieve waarden	Geen teken
0	{(C0)	}(D0)	0(F0)
1	A(C1)	J(D1)	1(F1)
2	B(C2)	K(D2)	2(F2)
3	C(C3)	L(D3)	3(F3)
4	D(C4)	M(D4)	4(F4)
5	E(C5)	N(D5)	5(F5)
6	F(C6)	O(D6)	6(F6)
7	G(C7)	P(D7)	7(F7)
8	H(C8)	Q(D8)	8(F8)
9	I(C9)	R(D9)	9(F9)

Tabel 3.3: Het gebruik van ASCII voor EBCDIC.

De plaats van het decimale punt (het V symbool) wordt intern bijgehouden in het programma maar niet opgeslagen. Hier wordt ook de gelijkens behouden met het uitprinten van een getal via ACUCOBOL. Als voorbeeld wordt het getal 123.12 voorgesteld als een char array van lengte 5, namelijk de string 12312. Reële getallen met een teken worden tevens voorgesteld dmv. 'Sign EBCDIC Custom'. In Tabel 3.4 wordt voor enkele getallen de EBCDIC waarde gegeven.

<b>Getal</b>	<b>Hexadecimale EBCDIC waarde</b>	<b>Karakterstring</b>
1234	F1F2F3F4	1234
+1234	F1F2F3C4	123D
-1234	F1F2F3D4	123M
123.98	F1F2F3F9F8	12398
+123.98	F1F2F3F9C8	1239H
-123.98	F1F2F3F9D8	1239Q

Tabel 3.4: Numerieke data als karakterstrings.

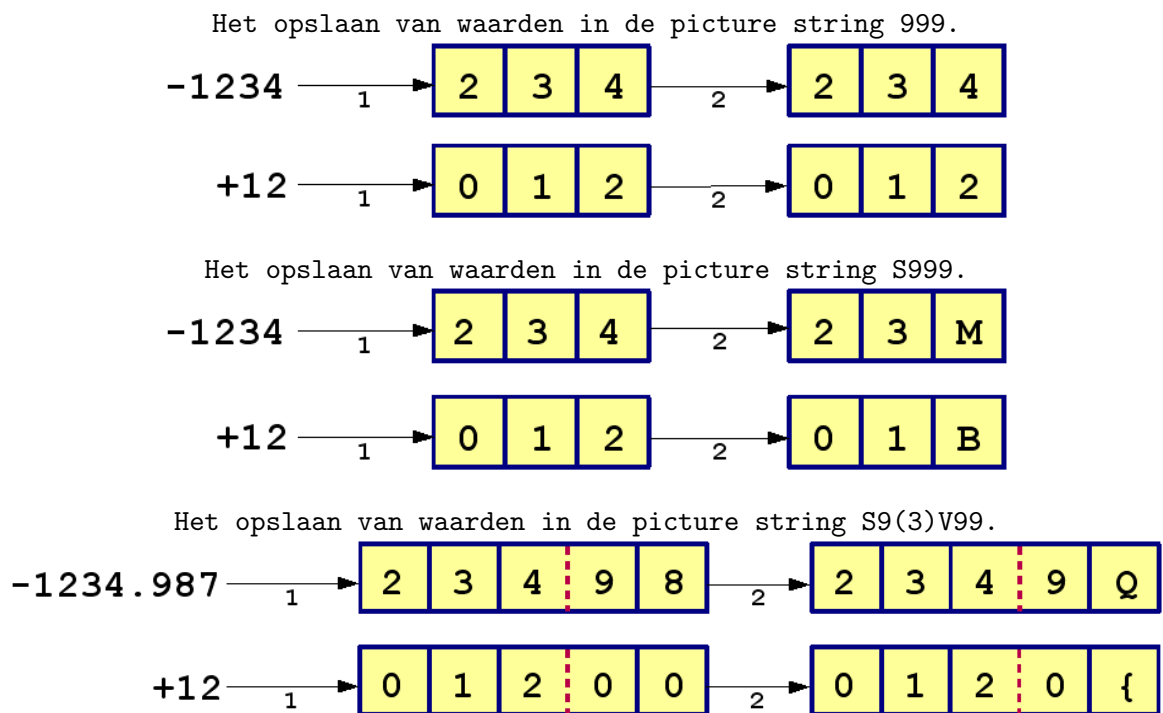
Ook bij de numerieke data moet er rekening worden gehouden met enkele regels ivm het doorgeven van data. De data wordt eerst altijd gealigneerd, zodat de plaats van het decimale punt overeenkomt. Daarna wordt de data links en rechts afgekapt indien de data te lang is of er worden nullen toegevoegd indien de data te kort zou zijn. Steeds moet ervoor gezorgd worden dat de informatie omtrent het teken behouden blijft. Deze alignatie regels kan men in de JVM simuleren door een getal in de char array representatie om te zetten in een int, long of double en daarna de geschikte methode te gebruiken die een int, long of double terug omzet in een char array. De regels ivm de alignatie worden verduidelijkt ahv. Figuur 3.2.

### 3.4 Numeriek-edited data

Numeriek-edited data dient om numerieke data in een printvriendelijk kleedje te krijgen. De data wordt tevens opgeslagen als een array van chars, dit om het uitprinten op het scherm te vergemakkelijken. Indien er numerieke data moet doorgegeven worden aan een numeriek-edited type is er een methode beschikbaar. Zie Tabel 3.5 voor de beschikbare methodes, voor het omvormen van de datatypes. De lengte van de char array is de lengte van de picture string. De behandelde karakters in de picture string van dit type zijn de symbolen 9,Z,-,. en ,.

### 3.5 Records

Een record of een group-item wordt niet omgezet in bytecode. In het programma wordt een record bijgehouden als een lijst van elementaire data-items. Indien een record wordt gebruikt door een statement, dan wordt de bewerking toegepast op de elementaire data. Het uitprinten



- (1) Alignatie volgens het decimale punt.
- (2) Aanpassing van het teken op basis van de picture string en het teken van de opgegeven waarde.

Figuur 3.2: Alignatie van numerieke data.

van een record via het `DISPLAY` statement wordt zo verwerkt tot het uitprinten van de verschillende elementaire data-items van de record na elkaar.

## 3.6 Booleans

Een boolean is een data-item, die wordt aangeduid door het levelnummer 88. Dit type van COBOL hoort bij een ander data-item. Tijdens het initialiseren van de variabelen in de `DATA DIVISION`, wordt bij een boolean in COBOL, een waarde meegegeven die moet zorgen voor het evalueren naar waar. De boolean evalueert namelijk naar waar, indien de waarde horend bij de boolean gelijk is aan de waarde van het bijhorende data-item. De boolean evalueert naar vals, indien deze 2 waarden niet gelijk zijn. Dit type wordt in bytecode voorgesteld als het Java object `CobolBoolean`. De constructor

```
public CobolBoolean(char [] trueValue, char [] associatedData)
```

dient een referentie naar de ware waarde en het geassocieerd data-item mee te krijgen. Via een methode `public boolean isTrue()` wordt een boolean verkregen die aangeeft of de `trueValue` overeenkomt met de `associatedData`. De methode `public void setTrue()` zorgt ervoor dat de char array van het geassocieerde data-item een waarde verkrijgt die overeenkomt met de ware waarde van de boolean.

<b>Datatype</b>	<b>Methode</b>	<b>Returntype</b>	<b>Opmerking</b>
numeriek	setContent (double nmr,char[] data,int decimalPlace)	void	zet de double nmr om in een char array
numeriek	setContent (int nmr,char[] data,int decimalPlace)	void	zet de int nmr om in een char array
numeriek	setContent (long nmr,char[] data,int decimalPlace)	void	zet de long nmr om in een char array
numeriek-edited	setContent (double nmr,char [] data, String picture)	void	zet de double nmr om in een char array
numeriek-edited	setContent (int nmr,char [] data, String picture)	void	zet de int nmr om in een char array
numeriek-edited	setContent (long nmr,char [] data, String picture)	void	zet de long nmr om in een char array
tekstueel	setContent (char [] toThis, char [] setThis)	void	copieert de inhoud van toThis naar setThis met toepassing van de alignatieregels
numeriek	toDouble(char[] data, int decimalPlace)	double	zet een char array om in een double
numeriek	toInteger(char[] data, int decimalPlace)	int	zet een char array om in een int
numeriek	toLong(char[] data, int decimalPlace)	long	zet een char array om in een long
numeriek-edited	toDouble(char[] data, String picture)	double	zet een char array om in een double
numeriek-edited	toInteger(char[] data, String picture)	int	zet een char array om in een int
numeriek-edited	toLong(char[] data, String picture)	long	zet een char array om in een long

Tabel 3.5: Methodes beschikbaar voor het omvormen van data.

## 3.7 Implementatie van de variabelen transformatie

De omvorming van de DATA DIVISION gebeurt in 2 stappen. Eerst wordt de DATA DIVISION van het gegenereerde XML document, uit de COBOL source code, omgezet naar een andere, gemakkelijker te gebruiken vorm. Daarna wordt dit aangepaste XML document gebruikt voor het inlezen en omzetten van de data-items.

### 3.7.1 XML omvorming van de DATA DIVISION

Het initieel gegenereerde XML-bestand bezit alle informatie die nodig is, maar toch heb ik er voor geopteerd, om de DATA DIVISION om te vormen naar een ander XML formaat. Het is namelijk niet altijd duidelijk of een bepaald item nu tot de bovenliggende record behoort of niet. Het niveaunummer moet altijd gecontroleerd worden om dit te weten. Ik heb gebruik gemaakt van de mogelijkheid om een structuur weer te geven in de XML-boom die geen nood meer heeft aan het niveaunummer, reeds een versimpeling op syntaxisgebied. Hieronder staat de DATA DIVISION zoals deze gegenereerd wordt voor de source code

```
01 Num1 PIC S(9) VALUE ZEROES.
01 Num2 PIC S(9) VALUE ZEROES.
01 Result PIC Z,Z99- VALUE ZEROES.

<data_division>
  <string>DATA</string>
  <!-- -->
  <string>DIVISION</string>
  <string>.</string>
  <!-- @n -->
  <string>WORKING-STORAGE</string>
  <!-- -->
  <string>SECTION</string>
  <string>.</string>
  <!-- @n -->
  <record_description_entry>
    <data_description_entry>
      <data_description_entry_format_b>
        <level_number>
          <numeric>
            <string>01</string>
          </numeric>
          <!-- -->
        </level_number>
        <data_name>
          <cobol_word>
            <string>Num1</string>
```

```

    </cobol_word>
    <!-- -->
</data_name>
<string>PIC</string>
<!-- -->
<picture_string>
  <character_string>
    <string>S9(3)</string>
  </character_string>
<!-- -->
</picture_string>
<string>VALUE</string>
<!-- -->
<value_lit>
  <figurative_constant>
    <string>ZEROS</string>
  </figurative_constant>
</value_lit>
<string>.</string>
<!-- @n -->
</data_description_entry_format_b>
</data_description_entry>
<data_description_entry>
<data_description_entry_format_b>
  <level_number>
    <numeric>
      <string>01</string>
    </numeric>
    <!-- -->
  </level_number>
  <data_name>
    <cobol_word>
      <string>Num2</string>
    </cobol_word>
    <!-- -->
  </data_name>
  <string>PIC</string>
  <!-- -->
  <picture_string>
    <character_string>
      <string>S9(3)</string>
    </character_string>
    <!-- -->
  </picture_string>
  <string>VALUE</string>
  <!-- -->
  <value_lit>
    <figurative_constant>

```

```

        <string>ZEROS</string>
    </figurative_constant>
</value_lit>
<string>.</string>
<!-- @n -->
</data_description_entry_format_b>
</data_description_entry>
<data_description_entry>
<data_description_entry_format_b>
    <level_number>
        <numeric>
            <string>01</string>
        </numeric>
        <!-- -->
    </level_number>
    <data_name>
        <cobol_word>
            <string>Result</string>
        </cobol_word>
        <!-- -->
    </data_name>
    <string>PIC</string>
    <!-- -->
    <picture_string>
        <character_string>
            <string>Z,Z99-</string>
        </character_string>
        <!-- -->
    </picture_string>
    <string>VALUE</string>
    <!-- -->
    <value_lit>
        <figurative_constant>
            <string>ZEROS</string>
        </figurative_constant>
    </value_lit>
    <string>.</string>
    <!-- @n -->
</data_description_entry_format_b>
</data_description_entry>
</record_description_entry>
</data_division>

```

De transformatie wordt uitgelegd ahv. Figuur 3.3, die een overzicht geeft van de stappen die doorlopen worden. In de hoofdklasse van het programma, `Cobol2Bytecode` wordt het XML-bestand omgezet naar een DOM document. Dit DOM document wordt doorgegeven aan een klasse `PreProcess`. Deze zal een lege `outNode` aanmaken die de `data_division` node moet

vervangen, de `data_division` wordt dus opnieuw aangemaakt. Deze `outNode` wordt samen met de `inNode`, de `data_division` node doorgegeven aan de `DataDivisionParser`.

Door de `DataDivisionParser` wordt er een `data_division` node toegevoegd aan de `outNode` en worden alle kinderen van de `inNode`, de `data_division` overlopen. Indien er een `file_description` node wordt ontmoet, zal deze in de `outNode` een `file_description` node aanmaken. De volgende `record_description` zal dan onder deze nieuwe `file_description` worden aangemaakt. Elke `record_description_entry` wordt afgehandeld door een `RecordDescriptionParser` object.

De `RecordDescriptionParser` zal nagaan of de doorgegeven node wel degelijk een record is, dit wordt nagegaan door te kijken naar het niveaunummer van de eerste en de volgende `data_description_entry` node. Indien een record, wordt er een `record_description` node aangemaakt in de `outNode`, alle bijhorende elementaire datatypes worden doorgegeven aan een `DataDescriptionParser` object.

Het `DataDescriptionParser` object zal in de eventueel bijhorende `record_description` een `data_description` node aanmaken. De enige informatie die behouden blijft is `data_name`, `picture` en `value`. Het is in deze klasse dat de vereenvoudiging van `picture` doorgevoerd wordt. De `picture` kan geen haakjes meer bevatten, `9(3)` wordt `999`. In het geval van een level-88 datatype, krijgt het bijbehorende data-item een extra node, dit is de `boolean_data` node. Deze node bevat de nodige informatie over een boolean, z'n naam (`data_name`) en z'n ware waarde (`value`). Zo wordt het meteen een stuk gemakkelijker om op een eenvoudige manier de verschillende DOM `data_description` nodes om te zetten naar bijhorende bytecode. De items bevatten nu alle data die nodig is, de XML representatie is dan:

```
<data_division>
  <data_description>
    <data_name>Num1</data_name>
    <picture>S999</picture>
    <value>0</value>
  </data_description>
  <data_description>
    <data_name>Num2</data_name>
    <picture>S999</picture>
    <value>0</value>
  </data_description>
  <data_description>
    <data_name>Result</data_name>
    <picture>Z,Z99-</picture>
    <value>0</value>
```

```
</data_description>  
</data_division>
```

Meer algemeen zal de `data_division` er nu uitzien als beschreven door de volgende regels, in EBNF.

```
data_division = (file_description)* || (record_description)* ||  
                (data_description)*;  
file_description = file_name record_description;  
record_description = record_name ((record_description)* || (data_description)*);  
data_description = data_name picture value (boolean_data)*;  
boolean_data = data_name value;
```

### 3.7.2 Bijhouden en omzetten van de variabelen

#### Het `DataKeeper` object

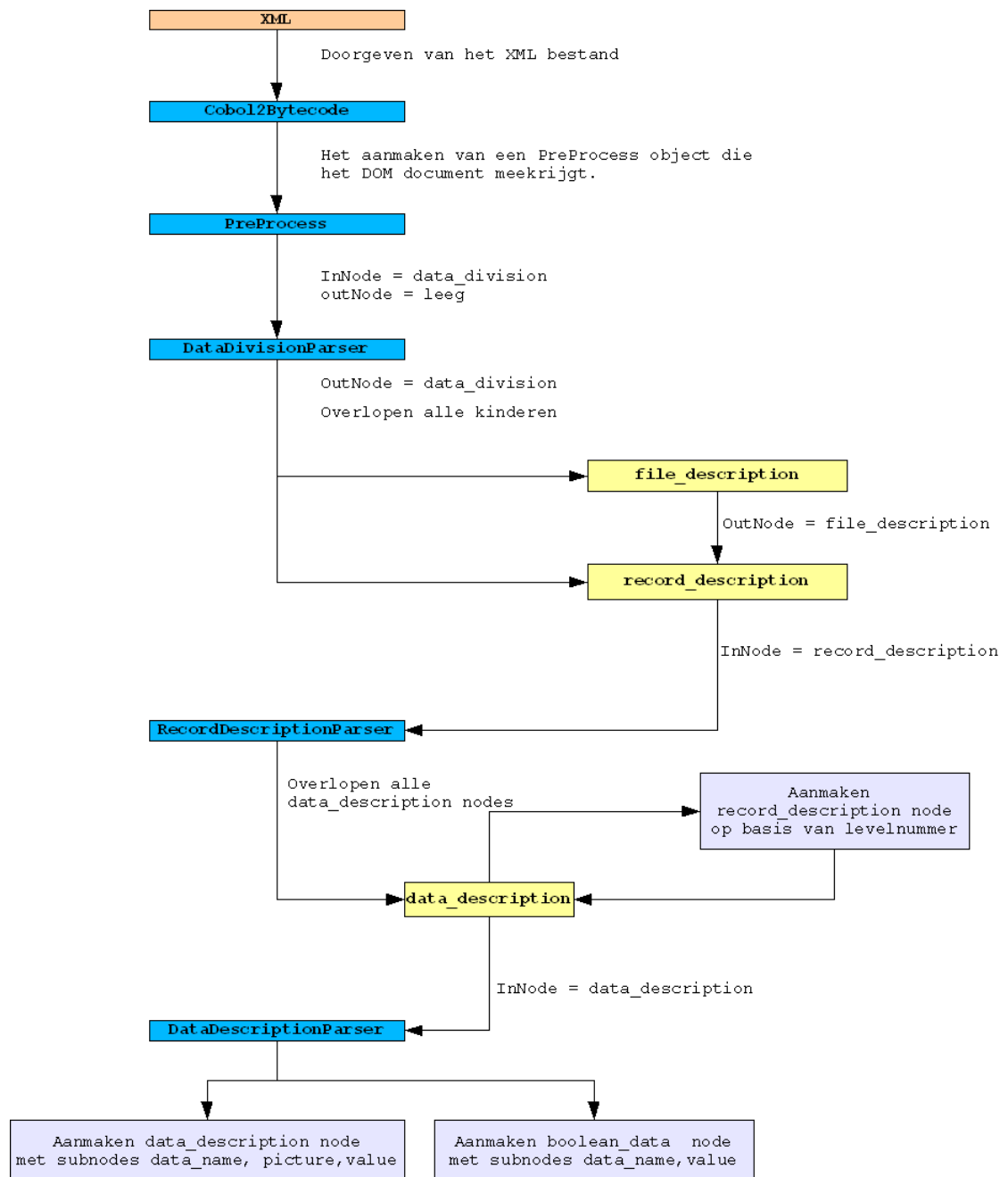
De data moet niet enkel omgezet worden in bytecode, de namen van de variabelen moeten ook bijgehouden worden in het programma. Het is het `DataKeeper` object dat hiervoor zorgt. Deze wordt doorgegeven aan elk object dat een transformatie moet uitvoeren. Het `DataKeeper` object houdt een mapping bij van de variabelen en hun namen. Deze variabelen worden bijgehouden als een `DataItem` object. De `DataKeeper` zorgt tevens voor de index in de rij van lokale variabelen, van de JVM. Naast het bijhouden van de gewone data-items, worden ook de records, de booleans en de filedescriptions bijgehouden.

#### Het `JVMSimulation` object

Het `JVMSimulation` object is tevens een object dat gebruikt wordt door andere objecten die een transformatie moeten uitvoeren. Dit object is een verzameling van methodes die zorgen voor het aanmaken van bytecode. De verschillende methodes zorgen voor het genereren van bytecode die veel wordt gebruikt. Het is ook mogelijk om rechtstreeks bytecode te schrijven door de `write` methode. De bytecode zelf wordt bijgehouden in een `BytecodeFile` object. Enkel via het `JVMSimulation` object kan het `BytecodeFile` object gewijzigd worden.

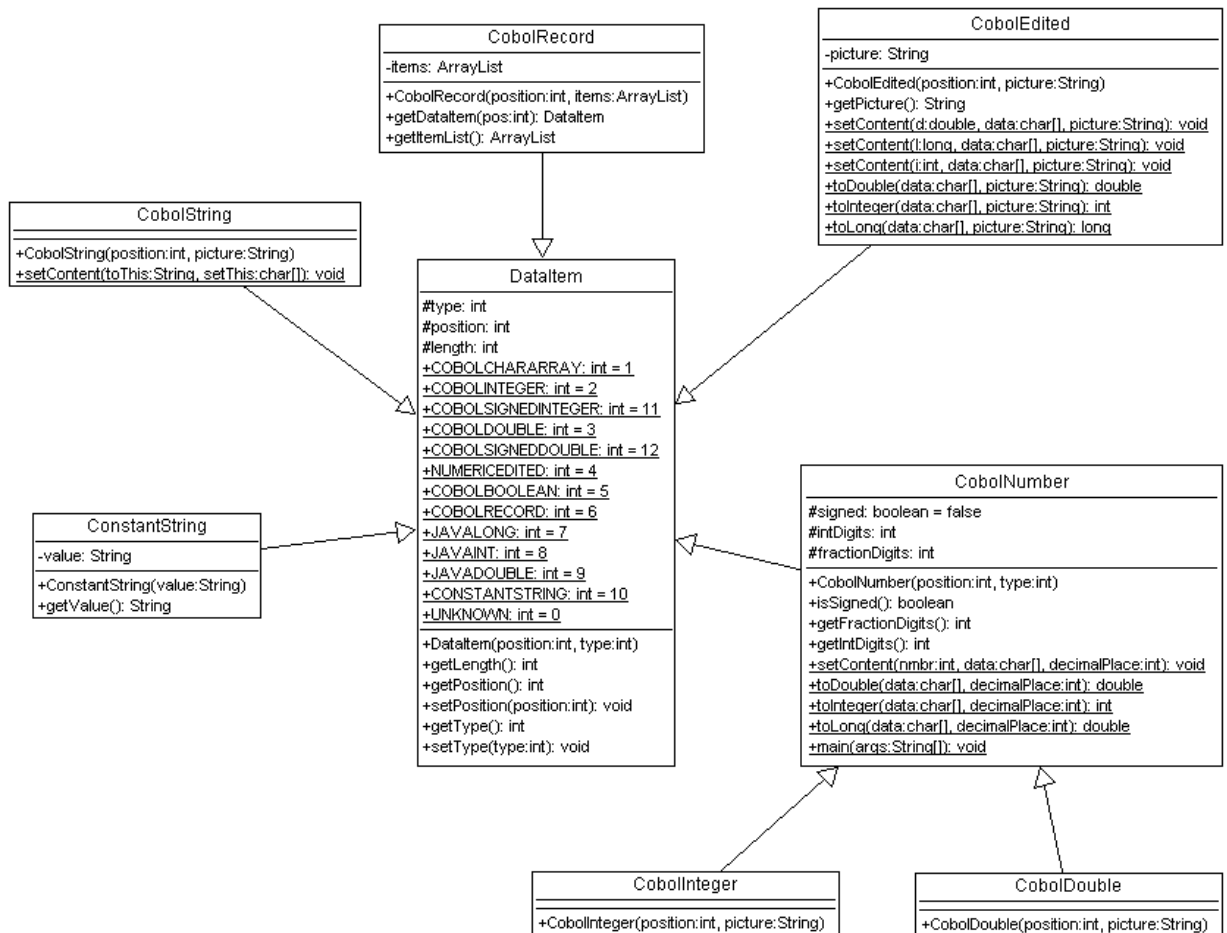
#### Het `DataItem` object

Elke variabele wordt opgeslagen als een `DataItem` object. Een `DataItem` object houdt niet de waarde bij van een variabele maar enkel de index in de rij van lokale variabelen, de lengte van de array en het type van de variabele. Dit is de informatie die nodig is voor het `JVMSimulation`



Figuur 3.3: XML transformatie van de DATA DIVISION.

object, om bytecode aan te maken. Het type wordt gebaseerd op de picture clause. Naargelang het type zal een `DataItem` een specifiekere klasse krijgen. Via deze klassen zijn er specifiekere methodes die van toepassing zijn op dat type. Het is ook hier waar de statische methodes van Tabel 3.5 zich bevinden. Figuur 3.4 geeft een overzicht van de extensies van het `DataItem` object.



Figuur 3.4: UML schema van de `DataItem` klasse.

### De `DataDivision` klasse

De hoofdklasse van het programma, `Cobol2Bytecode` zal een `DataDivision` object aanmaken. Het proces dat doorlopen wordt, wordt uitgelegd ahv Figuur 3.5. Aan dit object wordt de `data_division` node meegegeven, samen met de globaal aangemaakte `DataKeeper` om de variabelen in bij te houden en het globaal `JVMSimulation` object dat zal gebruikt worden om bytecode te genereren (1). De `DataDivision` zal elk kind van de `data_division` node overlopen en

aan de hand van het type een private methode oproepen (2).

```
private DataItem ProcessDataNode (Node item) throws XPathException {
    if((item.getNodeName().equals("data_description"))
        return ProcessDataDescription (item);
    else if((item.getNodeName().equals("record_description"))
        return ProcessRecordDescription (item);
    else if ((item.getNodeName().equals("file_description"))
        return ProcessFileDescription(item);
    return null;
}
```

Als voorbeeld wordt hieronder het geval van een `data_description` node weergegeven. De data wordt opgehaald uit het DOM document en doorgegeven aan de `DataKeeper` (3) die zorgt voor een `DataItem` object (4). Dit `DataItem` object wordt dan doorgegeven aan `JVMSimulation` (5) die bytecode aanmaakt (6).

```
private DataItem ProcessDataDescription (Node item) throws XPathException {
    // register the data and set the initial value
    String name = (String)xpath.evaluate("data_name",item,
        XPathConstants.STRING);
    String picture = (String)xpath.evaluate("picture",item,
        XPathConstants.STRING);
    String value = (String)xpath.evaluate("value",item,XPathConstants.STRING);
    // register with DataKeeper
    DataItem data = keeper.registerData(name,picture);
    // correct representation of initial data
    char [] initialData = formatValue(data,value);
    JVM.initializeData(data,initialData); // make bytecode
    // if there are associated CobolBoolean,
    // initialize it and set his associated data and value
    NodeList bList =
        (NodeList)xpath.evaluate("boolean_data",item,XPathConstants.NODESET);
    for (int i = 0; i < bList.getLength(); i++)
    {
        Node tmp = bList.item(i);
        String boolName =
            (String) xpath.evaluate("data_name", tmp , XPathConstants.STRING);
        String boolValue =
            (String) xpath.evaluate("value", tmp , XPathConstants.STRING);
        DataItem bool = keeper.registerCobolBoolean(boolName);
        JVM.initializeCobolBoolean(bool,boolValue,data);
    }
    return data;
}
```

De implementatie van de methode `registerData` van de `DataKeeper` is:

```

public DataItem registerData(String name , String picture){
    variableCounter++;
    DataItem d;
    int type = getType(picture);
    if (type==DataItem.COBOLCHARARRAY)
        d = new CobolString(variableCounter,picture);
    else if (type==DataItem.COBOLDOUBLE)
        d = new CobolDouble(variableCounter,picture);
    else if (type==DataItem.COBOLINTEGER)
        d = new CobolInteger(variableCounter,picture);
    else if (type==DataItem.NUMERICEDITED)
        d = new CobolEdited(variableCounter,picture);
    else // type should be UNKNOWN
        d = new DataItem(variableCounter,type);
    dataItems.put(name , d);
    return d;
}

```

De methode `initializeData` van `JVMSimulation` geeft:

```

public void initializeData(DataItem a, char [] initialValue) {
    // the bytecode, the position is the index in the local vars array
    code.initializeArray(initialValue,a.getPosition());
}

```

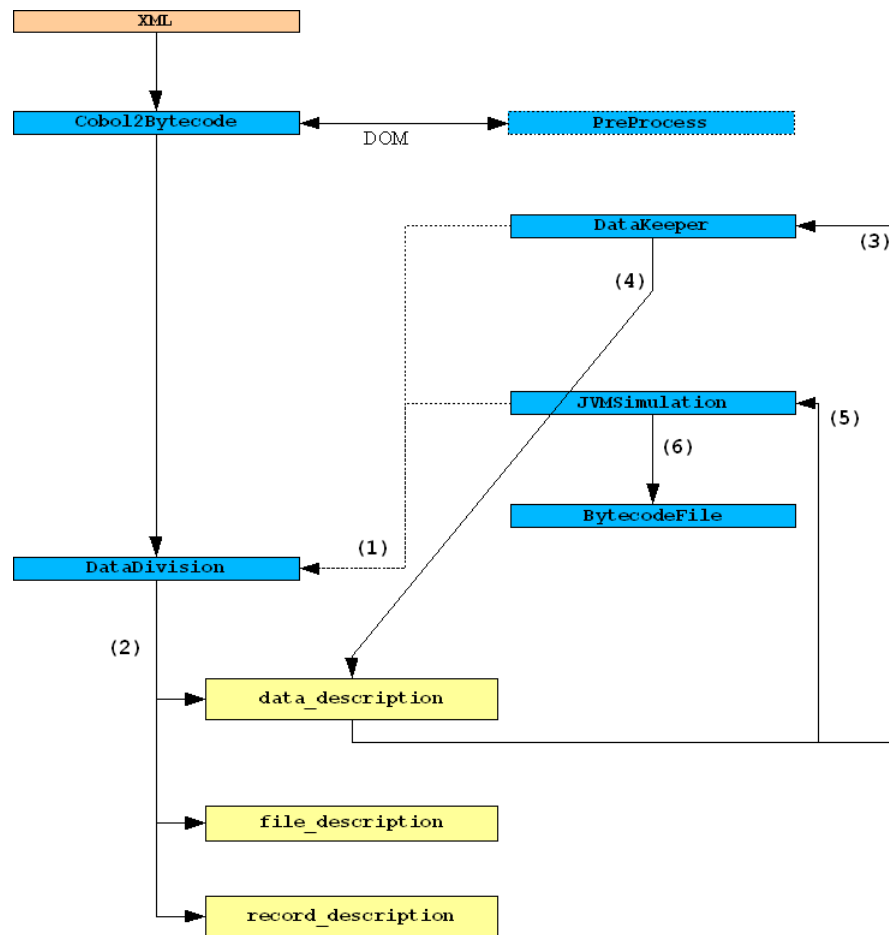
Deze maakt gebruik van de methode `initializeArray` van `BytecodeFile`:

```

public void initializeArray(char [] value , int position) {
    write("bipush " + value.length);
    write("newarray char");
    for (int i = 0; i < value.length; i++)
    {
        write("dup");
        write("bipush " + i);
        write("bipush " + (byte) value[i]);
        write("castore");
    }
    write("astore " + position);
}

```

In het geval van een `record_description` of een `file_description` wordt een gelijkaardig proces doorlopen.



Figuur 3.5: Aanmaken van bytecode voor variabelen.

## Hoofdstuk 4

# Mapping van statements naar bytecode

De opzet van het onderzoek is aantonen dat COBOL kan gemapt worden naar bytecode. Om dit te onderzoeken is er gekozen voor een representatieve subset van COBOL statements. In wat volgt krijg je een overzicht van de gemapte statements, statements die een invloed hebben op de flow control, worden in het volgend hoofdstuk behandeld. Merk op dat niet elk statement volledig is geïnterpreteerd. Een statement heeft in COBOL veelal verschillende vormen [7, 8], er is meestal voor één bepaalde vorm gekozen.

### 4.1 Overzicht statements en hun mapping

#### 4.1.1 DISPLAY

Het DISPLAY statement verzorgt het uitprinten van variabelen of constante waarden op het scherm. De nagegane vorm heeft als constructie:

```
DISPLAY item1,item2,...,itemn.
```

Hier kan een item een variabele, een record of een constante string zijn.

Indien er maar 1 item aanwezig is, wordt het item uitgeprint op het scherm en gesprongen naar de volgende lijn. Als er meerdere items zijn, worden de items na elkaar getoond en pas na het laatste item, wordt er naar een nieuwe lijn gegaan.

Voor de JVM zijn er echter geen opcodes beschikbaar voor printen op het scherm. Er zal daardoor gebruik gemaakt worden van het oproepen van Java methodes. De methodes die

we hiervoor gebruiken zijn `print` en `println` van het `PrintStream` object, bekomen door `java.lang.System.out`. Een referentie naar het `PrintStream` object op de stack plaatsen kan mbv. de opcode `getstatic`. De methodes van het object, kunnen worden opgeroepen via de opcode `invokevirtual`.

Een constante string wordt door de Jasmin opcode `ldc` op de stack geplaatst. De Java functie van `print` (of `println`) die een `String` vraagt als input wordt dan opgeroepen. Het uitschrijven van een variabele wordt bekomen door de Java printfunctie die een char array vraagt als parameter.

Voor de implementatie, wordt het `DISPLAY` statement behandeld door het aanmaken van een `Display` object. Het `Display` object zal een `ArrayList` van `DataItem` objecten bijhouden die moeten uitgeschreven worden. Het `DataItem` van een variabele bekom je door de methode `getDataItem(variableName)` op te roepen van het `DataKeeper` object. Indien de opgehaalde variabele een record is, wordt niet het record maar alle data-items waaruit het record bestaat toegevoegd aan de `ArrayList`. Het `DataItem` van een constante string wordt nieuw aangemaakt, dit is dan een `ConstantString` object die enkel z'n waarde meekrijgt voor initialisatie. Elk `DataItem` wordt dan doorgegeven aan de methode `printData` van de `JVMSimulation`.

```
public void printData(DataItem data, boolean newline) {
    code.write("getstatic java/lang/System/out Ljava/io/PrintStream;");
    String print = "print";
    if(newline)
        print = "println";
    if (data.getType() == DataItem.CONSTANTSTRING)
    {
        String value = ((ConstantString) data).getValue();
        code.write("ldc \""+value+"\"");
        code.write("invokevirtual java/io/PrintStream/" + print
            + "(Ljava/lang/String;)V");
    }
    else
    {
        code.write("aload "+data.getPosition());
        code.write("invokevirtual java/io/PrintStream/" + print
            + "([C)V");
    }
}
```

De implementatie van de methode `printData` zorgt dus voor het aanmaken van bytecode. De referentie naar de array wordt dmv de bewaarde positie in het `DataItem` object op de stack

geplaatst. Door ervoor te zorgen dat `printData` een `newline` boolean krijgt als parameter, kan de `Display` klasse worden uitgebreid met de optie `WITH (NO) ADVANCING`, van het `DISPLAY` statement.

Het omzetten van `DISPLAY "The result is: ",Result.` geeft in Jasmin formaat dan:

```
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "The result is: "
invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
getstatic java/lang/System/out Ljava/io/PrintStream;
aload 3
invokevirtual java/io/PrintStream/println([C)V
```

### 4.1.2 ACCEPT

Het `ACCEPT` statement verzorgt het inlezen van data van standaard input en het bewaren van de data in een variabele. De nagegane vorm van het statement is:

```
ACCEPT data-item.
```

Net zoals voor het uitprinten, zijn er ook voor het inlezen van data geen opcodes beschikbaar. De Java methode die hier gebruikt wordt is een zelfgeschreven methode.

Het omzetten van het `ACCEPT` statement wordt behandeld door de klasse `Accept`. In analogie aan de `Display` klasse, wordt eerst de variabelenaam uit de XML opgevraagd en doorgegeven aan de `DataKeeper` om het `DataItem` object te verkrijgen. Indien het een record is, worden de verschillende items die behoren tot de record apart behandeld. Het aanmaken van bytecode voor het inlezen verkrijg je door de `readData` methode op te roepen van `JVMSimulation`. De methode vraagt een `DataItem` en een boolean, deze boolean kan wederom gebruikt worden voor de optie `WITH (NO) ADVANCING`. De methode `readData` zal voor elke `DataItem`, code aanmaken die een char array inleest van standaard input. Dit met behulp van de zelfgeschreven methode `readFromInput`. Het oproepen van deze methode via bytecode vraagt dat er reeds een char array en een boolean op de stack staan. De ontwerpbeslissing van deze methode steunt op het gebruik van `ACUCOBOL`.

```
public void readData (DataItem data , boolean newline)
{
    code.write("aload "+data.getPosition());
    if (newline)
        code.write("iconst_1");
    else
```

```

        code.write("iconst_0");
        code.write("invokestatic cobol/HelperClass/readFromInput([CZ)V");
    }

```

Omzetting van ACCEPT Number1. geeft in Jasmin formaat dan:

```

aload 1
iconst_1
invokestatic cobol/HelperClass/readFromInput([CZ)V

```

### 4.1.3 COMPUTE

Het COMPUTE statement evalueert een aritmetische expressie en zet 1 of meer data-items gelijk aan het resultaat. Het nagegane COMPUTE statement is in COBOL van de vorm:

```
COMPUTE data-item = arithmetic-expression.
```

Een voorwaarde is dat `data-item` numeriek of numeriek edited moet zijn. Deze voorwaarde houdt in dat `data-item` kan voorgesteld worden als een int, long of double, evenals de uitkomst van de aritmetische expressie.

Hierdoor kan het COMPUTE statement worden omgevormd tot het berekenen van een aritmetische expressie, die de uitkomst onder de vorm van een int, long of double op de stack plaatst en het toepassen van de bijhorende `setContent` methode van `CobolNumber` of `CobolEdited` (zie Tabel 3.5), om het resultaat op te slaan in `data-item` als een char array.

De Java klasse die dit COMPUTE statement verzorgt is `Compute`. Deze maakt een `Computational` object aan waaraan de `arithmetic_expression` node wordt doorgegeven. Het gehele proces die de omzetting verzorgt, wordt uitgelegd mbv. het voorbeeld

```
COMPUTE Result = 10 + Number1 * Number2.
```

of in XML formaat:

```

<compute>
  <string>COMPUTE</string>
  <result>
    <cobol_word>
      <string>Result</string>
    </cobol_word>
  </result>
  <string>=</string>
  <arithmetic_expression>
    <multiplicative_expression>
      <power_expression>
        <unary_expression>

```

```

    <basic_expression>
      <number>
        <numeric>
          <string>10</string>
        </numeric>
      </number>
    </basic_expression>
  </unary_expression>
</multiplicative_expression>
<string>+</string>
<multiplicative_expression>
  <power_expression>
    <unary_expression>
      <basic_expression>
        <data_item>
          <cobol_word>
            <string>Number1</string>
          </cobol_word>
        </data_item>
      </basic_expression>
    </unary_expression>
  </power_expression>
  <string>*</string>
  <power_expression>
    <unary_expression>
      <basic_expression>
        <data_item>
          <cobol_word>
            <string>Number2</string>
          </cobol_word>
        </data_item>
      </basic_expression>
    </unary_expression>
  </power_expression>
</multiplicative_expression>
</arithmetic_expression>
</compute>

```

De bytecode voor het voorbeeld aangemaakt door `Compute`, wordt in Jasmin:

```

;--hier code aangemaakt door Computational.java--
aload 3
ldc "Z,Z99-"
invokestatic cobol/CobolEdited/setContent(I[CLjava/lang/String;)V

```

### arithmetic expression

Om het resultaat van de expressie uit te werken, zullen alle variabelen omgezet worden naar getallen van het type `int`, `long` of `double`. Dit zijn types van de JVM waarmee kan worden gerekend. De bewerkingen die beschikbaar zijn, zijn `add`, `sub`, `mul` en `div`. Dit zijn type-afhankelijke bewerkingen, enkel waarden van hetzelfde type kunnen gebruikt worden. Voor bijvoorbeeld de optelling van 2 `int` waarden hebben we de opcode `iadd`. Hoe de verwerking van de expressie gebeurt, wordt hier verduidelijkt dmv. de implementatie.

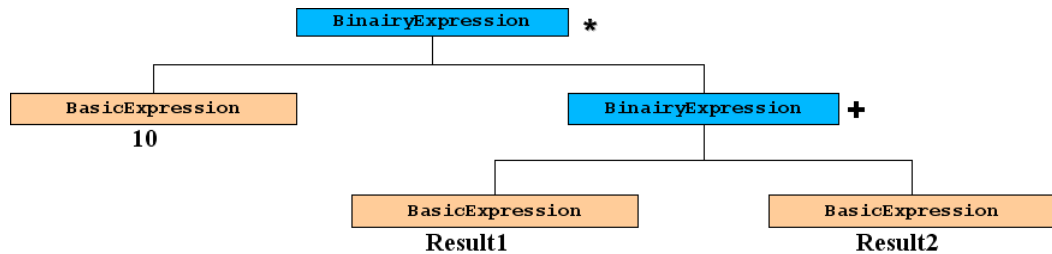
De `Computational` klasse zal ervoor zorgen, dat voor elke bewerking, een `Expression` object wordt aangemaakt. `Expression` is een abstracte klasse die een rekenkundige bewerking voorstelt. Tabel 4.1 geeft aan welke uitbreiding van `Expression` er wordt gebruikt voor de verschillende nodes. Figuur 4.2 geeft het UML schema van `Expression`.

Teken	Type	XML-node	Expression	Voorbeelden
+	optelling	<code>arithmetic_expression</code>	<code>BinaryExpression</code>	<code>5+2</code>
-	aftrekking	<code>arithmetic_expression</code>	<code>BinaryExpression</code>	<code>5-2</code>
*	vermenigvuldiging	<code>multiplicative_expression</code>	<code>BinaryExpression</code>	<code>5*2</code>
/	deling	<code>multiplicative_expression</code>	<code>BinaryExpression</code>	<code>5/2</code>
**	exponentiatie	<code>power_expression</code>	<code>BinaryExpression</code>	<code>5**2</code>
+	geen effect	<code>unary_expression</code>	<code>UnaryExpression</code>	<code>+5</code>
-	negatie	<code>unary_expression</code>	<code>UnaryExpression</code>	<code>-5</code>
	variabele, getal of figuurlijke constante	<code>basic_expression</code>	<code>BasicExpression</code>	<code>Result</code> , <code>5</code> of <code>ZERO</code>

Tabel 4.1: Aanmaken van `Expression` objecten.

De `Expression` objecten worden recursief opgebouwd en hebben het aanmaken van een `BasicExpression` als eindvoorwaarde. Een `BinaryExpression` krijgt 2 andere `Expression` objecten mee waarop de bewerking moet worden uitgevoerd. Het toepassen van `Computational` op ons voorbeeld, zal de twee `BasicExpression` objecten aanmaken die `Result1` en `Result2` voorstellen. Deze worden meegegeven bij het aanmaken van een `BinaryExpression` van het type `MUL` die `Result1*Result2` voorstelt en op zijn beurt, samen met de `BasicExpression` voor `10` wordt gebruikt voor het aanmaken van een `BinaryExpression` van het type `ADD` die `10 + (Result1*Result2)` voorstelt. Dit is te zien op Figuur 4.1. Aan elke `Expression` wordt

ook het `DataItem` van het resultaat meegegeven, een `Expression` moet immers weten met welk JVM type het moet kunnen werken, bv `imul` of `dmul`.



Figuur 4.1: Boomstructuur van een Expression.

Het zijn de verschillende `Expression` objecten die zorgen voor het aanmaken van bytecode, door methodes van het `JVMSimulation` object te gebruiken. Een `BasicExpression` zorgt voor bytecode die de variabele of een constante waarde op de stack plaatst. Een `BinaryExpression` zal z'n twee `Expression` objecten eerst bytecode laten aanmaken, om dan de bytecode aan te maken die de bewerking voorstelt. Een `UnaryExpression` dient voor de negatie van een andere `Expression`.

Toegepast op ons voorbeeld zal de `BinaryExpression` die de optelling verzorgt, als Jasmin code geven:

```

;--hier code van BasicExpression voor 10
;--hier code van BinaryExpression voor Result1*Result2
iadd
  
```

De `BasicExpression` voor 10 zet een int met waarde 10 op de stack via `ldc 10`.

De `BinaryExpression` voor de vermenigvuldiging geeft:

```

;--hier code van BasicExpression voor Result1
;--hier code van BasicExpression voor Result2
imul
  
```

De `BasicExpression` voor `Result1` zet de int voorstelling van de variabele op de stack, dit geeft:

```

aload 1
bipush 3
invokestatic cobol/CobolNumber/toInteger([CI)I
  
```

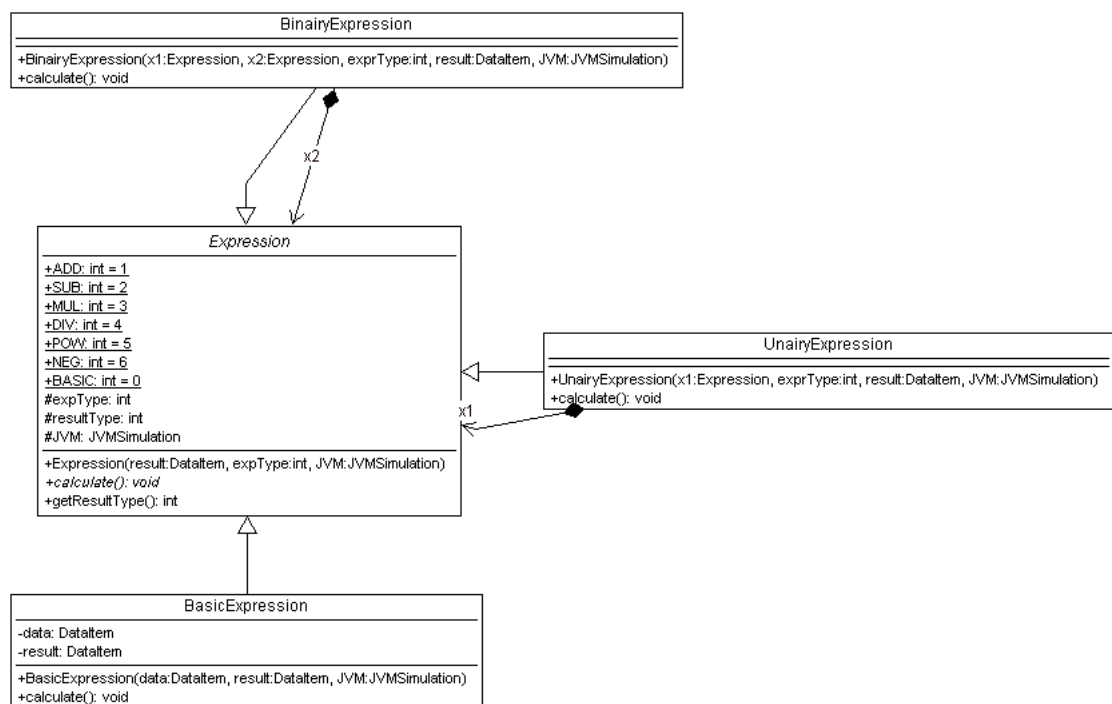
De totaal gegenereerde code voor het `COMPUTE` statement geeft in Jasmin:

```

ldc 10
aload 1
bipush 3
invokestatic cobol/CobolNumber/toInteger([CI)I
aload 2
bipush 3
invokestatic cobol/CobolNumber/toInteger([CI)I
imul
iadd
aload 3
ldc "Z,Z99-"
invokestatic cobol/CobolEdited/setContent(I[CLjava/lang/String;)V

```

De klasse `Computational` wordt later ook nog gebruikt door een klasse `Condition`.



Figuur 4.2: UML schema van de Expression klasse.

#### 4.1.4 SUBTRACT

Het `SUBTRACT` statement trekt een waarde af van een gegeven variabele. De vorm van het statement dat onderzocht is, heeft als voorstelling:

```
SUBTRACT literal-1_or_data-item-1 FROM data-item-2.
```

Dit zal de eerste waarde aftrekken van `data-item-2` en het resultaat bewaren in `data-item-2`.

Het is evident dat de mapping naar bytecode zal zorgen door code die de 2 waarden op de stack plaatst onder de vorm van een `int`, `double` of `long` en het toepassen van de opcodes `isub`, `dsub` of `lsub` respectievelijk. Deze waarde moet dan enkel nog bewaard worden in `data-name-2`, via een `setContent` methode.

Als implementatie worden de `Expression` objecten hergebruikt. Het `SUBTRACT` statement kan gezien worden als een `BinaryExpression` van het type `SUB` waaraan de 2 `BasicExpression` objecten worden doorgegeven die de 2 waarden voorstellen. De gegenereerde bytecode zal hierdoor hetzelfde zijn als code voor:

```
COMPUTE data-item-2 = data-item-2 - literal-1_or_data-item-1
```

De statements `ADD`, `MULTIPLY` en `DIVIDE` werden niet geïmplementeerd maar deze kunnen op dezelfde manier worden omgezet zoals voor het `SUBTRACT` statement. Hiervoor zullen de type-afhankelijke opcodes `add`, `mul` en `div` respectievelijk worden gebruikt.

## 4.2 Toepassing op de applicatie

Nu is de mapping van statements verduidelijkt maar hoe deze objecten, die zorgen voor de mapping, worden toegepast op het volledige programma dient nog enige uitleg.

Elk object die dient voor de mapping van een statement, is een extensie een abstract `TransformFunction` object. Een `TransformFunction` houdt de meegegeven node, het `DataKeeper` object en het `JVMSimulation` object bij. De abstracte methode `processInstruction` moet geïmplementeerd worden zodat door deze methode uit te voeren de gewenste bytecode wordt gegenereerd.

Nadat de variabelen van het COBOL programma gemapt zijn naar bytecode, wordt elk statement in de XML source één voor één overlopen en gemapt. Het is in `Procedure.java` dat de `procedure_division` node wordt aangepakt. Bij het overlopen van het DOM document wordt een `statement` node doorgegeven aan een `Statement` object. Een `Statement` object zal een `TransformFunction` object aanmaken die hoort bij het COBOL statement. Welk object moet aangemaakt worden voor ieder statement staat beschreven in `statement_mapping.xml`. Uitvoeren van de geschikte `TransformFunction` verkrijg je door:

```
// statementType could be display, accept, compute ...  
String statementType = statNode.getNodeName();
```

```
// className is the name of the corresponding Java class eg. Display
String className = (String) xpath.evaluate("classmap/"+statementType,
                                           classDocument , XPathConstants.STRING);

try
{
    // create the corresponding TransformFunction object
    Class c = Class.forName(className);
    TransformFunction o = (TransformFunction) c.newInstance();
    o.setDataKeeper(keeper);
    o.setSimulator(JVM);
    o.setNode(statNode);
    o.processInstruction(); // execute the statement
} catch (ClassNotFoundException e1)
{
    System.err.println("TransformFunction not found for : " +statementType);
}
```

Hierbij is classDocument het DOM document van class\_mapping.xml.

Het XML document class\_mapping.xml heeft als inhoud:

```
<classmap>
  <display>Cobol2Bytecode.transformer.Display</display>
  <accept>Cobol2Bytecode.transformer.Accept</accept>
  <compute>Cobol2Bytecode.transformer.Compute</compute>
  <read>Cobol2Bytecode.transformer.Read</read>
  <perform>Cobol2Bytecode.transformer.Perform</perform>
  <stop>Cobol2Bytecode.transformer.Stop</stop>
  <if>Cobol2Bytecode.transformer.IfStatement</if>
  <subtract>Cobol2Bytecode.transformer.Subtract</subtract>
</classmap>
```

Door deze werkwijze kunnen de gemapte statements worden uitgebreid, zonder aanpassing van het programma. Er dient enkel een nieuwe klasse te worden aangemaakt, die TransformFunction uitbreidt alsook een regel bij te voegen in class\_mapping.xml.

Het aanmaken van een nieuwe mapping wordt besproken in hoofdstuk 6, ahv. het MOVE statement.

## Hoofdstuk 5

# Flow control

Om COBOL statements zoals IF en PERFORM te mappen naar Java bytecode moet onderzocht worden welke opcodes er beschikbaar zijn voor de JVM. Deze opcodes zullen genomen worden uit de verzameling van opcodes voor flow control.

### 5.1 Flow control in bytecode

#### 5.1.1 Onvoorwaardelijke sprong

De eenvoudigste opcode voor flow control is `goto <label>`. Deze opcode zorgt voor een onvoorwaardelijke sprong naar het meegegeven label. Terugspringen via `goto` is niet mogelijk, het PC register van de JVM wordt namelijk niet bewaard.

#### 5.1.2 Voorwaardelijke sprong

Een andere reeks opcodes zijn de opcodes die 2 getallen vergelijken en springen indien een gestelde voorwaarde waar is. Zo zorgen de opcodes `if_icmp<cond> <label>` voor het vergelijken van 2 int waarden op de stack en springen naar het label `<label>` indien aan de voorwaarde `<cond>` voldaan is. Tabel 5.1 toont aan welke condities er kunnen worden gebruikt. Naast de opcodes die 2 getallen met elkaar vergelijkt, bestaan er ook opcodes die een getal vergelijkt met 0. Dit is `if<cond> <label>`.

Indien de 2 waarden die met elkaar moeten vergeleken worden een double of een long zijn, kan er niet gesprongen worden mbv de opcodes hierboven. In het geval van een long kunnen de 2 getallen vergeleken worden door `lcmp`. Deze opcode vergelijkt de 2 long waarden op de stack en zal -1 teruggeven indien `getal1<getal2`, 1 indien `getal1>getal2` en 0 indien de getallen

Opcode	Operandum	Description
<code>if_icmpeq</code>	label	pop int getal2 en getal1, als getal1 == getal2, spring naar label
<code>if_icmpne</code>	label	pop int getal2 en getal1, als getal1 != getal2, spring naar label
<code>if_icmplt</code>	label	pop int getal2 en getal1, als getal1 < getal2, spring naar label
<code>if_icmple</code>	label	pop int getal2 en getal1, als getal1 <= getal2, spring naar label
<code>if_icmpgt</code>	label	pop int getal2 en getal1, als getal1 > getal2, spring naar label
<code>if_icmpge</code>	label	pop int getal2 en getal1, als getal1 >= getal2, spring naar label

Tabel 5.1: Opcodes voor het vergelijken van 2 int waarden.

gelijk zijn. Deze int waarde kan dan vergeleken worden met 0 om te springen naar een label. In het geval van double waarden worden de opcodes `dcmpg` en `dcmpl` gebruikt. Deze doen beide hetzelfde voor double als `lcmp` voor long. Het enige verschil tussen `dcmpg` en `dcmpl`, is het behandelen van NaN (not a number) waarden.

### 5.1.3 Terugkerende sprong

Naast enkel springen naar een label, kan er ook gesprongen worden en teruggekeerd. Dit wordt bekomen door `jsr <label>` te gebruiken. Deze opcode zorgt voor het springen naar het label `<label>` en zal een referentie naar de opcode na `jsr` op de stack plaatsen. Deze referentie kan via `astore` in de rij van lokale variabelen worden opgeslagen. Terugkeren doe je dan via `ret <index>`. De bijhorende index is de plaats in de rij van lokale variabelen waar de referentie is opgeslagen.

In de compilatie van een standaard Java methode worden `jsr` en `ret` niet gebruikt, de opcodes dienen namelijk voor exception handling. Alternatief zou er kunnen gebruik gemaakt worden van het aanmaken en oproepen van een methode, in bytecode.

## 5.2 Flow control in COBOL

Na het onderzoeken welke opcodes voor flow control beschikbaar zijn op de JVM, moet er gezocht worden naar een combinatie van die opcodes om het IF of PERFORM statement voor te stellen.

### 5.2.1 IF ELSE

Het onderzochte IF statement is in COBOL van de vorm:

```
IF condition-1 THEN statement-1 [ELSE statement-2] [END-IF]
```

Indien `condition-1` waar is zullen de statements onder `statement-1` worden uitgevoerd. Indien de conditie niet waar is zullen de statements volgend op het IF statement worden uitgevoerd, of `statement-2` indien er een ELSE clause aanwezig is.

Het IF statement kan omgezet worden naar bytecode door een combinatie van de `goto` en `if_icmp<cond>` opcodes en het toevoegen van labels. Dit ziet er in Jasmin formaat dan als volgt uit:

```

        if_icmp<cond> waarLabel
valsLabel:
        ; code voor statement-2
        goto eindLabel
waarLabel:
        ; code voor statement-1
eindLabel:
        ; code voor statements na IF

```

De eerste regel in het stuk Jasmin code die de conditie voorstelt is echter wel wat te simplistisch. De echte code die gevormt wordt hangt af van de constructie van de COBOL conditie.

### condition

Een conditie in COBOL kan opgebouwt worden als verschillende deelcondities. Een simpele conditie bestaat uit een level-88 data-item of 2 aritmetische expressies die verbonden zijn door een relatie. Een complexe conditie bekom je door het gebruik van de AND,OR en NOT connectoren. In EBNF geeft dit:

```

condition = (simple condition) || (complex condition);
simple condition = (arithmetic expr) <comparator> (arithmetic expr)
                || (cobolBoolean)
complex condition = (condition OR condition) || (condition AND condition)
                || (NOT condition)

```

De condities moeten geëvalueerd worden en aan de hand van het waar of vals zijn van de conditie moet er in bytecode een stuk code uitgevoerd worden, aangeduid door een label. De enige externe informatie nodig om een conditie op te bouwen is de label voor waar en de label voor vals. Door gebruik te maken van die labels, samen met `goto` en `if_icmp<cond>` of `if<cond>` kan een conditie in COBOL, omgezet worden in bytecode. In wat volgt zal de bytecode voor de mogelijke samenstellingen van een conditie gegeven worden.

Een level-88 datatype wordt voorgesteld op de JVM als een `CobolBoolean` object. Er is een methode `isTrue` beschikbaar die het object evalueert en een boolean teruggeeft. Een boolean

wordt op de JVM aangeduid door een 0 indien vals of een 1 indien waar. De evaluatie van een level-88 data-item kan in Jasmin code dan door:

```
; op stack plaatsen CobolBoolean object via aload
invokevirtual cobol/CobolBoolean/isActive(V)B
if_gt waarLabel ; indien de boolean > 0, moet er gesprongen worden
```

Het evalueren van een relatie tussen 2 arithmetic expressions kan natuurlijk via een opcode uit Tabel 5.1 indien de te vergelijken waarden van het int type zijn. Zoals besproken in het vorige hoofdstuk zorgt het uitvoeren van een arithmetic expression voor een int, long of double op de stack. De evaluatie van 2 double types kan door:

```
; evalueren van arithm. expr. 1
; evalueren van arithm. expr. 2
dcmpg
if_<cond> waarLabel
```

Voor het vergelijken van 2 long types moet lcmp worden gebruikt.

Het combineren van 2 condities via AND vraagt dat de eerste conditie wordt geëvalueerd, indien de eerste conditie waar is, moet ook de tweede conditie worden bekeken. Als voorbeeld geeft dit voor  $(2 > 1) \text{ AND } (8 < 9)$ :

```
        ldc 2
        ldc 1
        if_icmpgt andLabel
        goto valsLabel
andLabel:
        ldc 8
        ldc 9
        if_icmplt waarLabel
```

Het gebruik van OR vraagt dat er maar één van de twee condities waar moet zijn. Hierdoor is de bytecode voor de complexe conditie met combinator OR, gelijk aan de bytecode voor de 2 condities, na elkaar. Als voorbeeld geeft dit voor  $(2 > 1) \text{ OR } (8 < 9)$ :

```
        ldc 2
        ldc 1
        if_icmpgt waarLabel
        ldc 8
        ldc 9
        if_icmplt waarLabel
```

De NOT combinator zorgt voor de negatie van een conditie. Dit kan in bytecode verkregen worden door het omdraaien van de label voor waar en de label voor vals, op de hele conditie

waar de negatie op slaat en het toevoegen op het einde van de opcode `goto`, met als parameter de label voor waar. De extra `goto` is nodig omdat na een conditie namelijk de bytecode staat die uitgevoerd moet worden als de conditie naar vals evalueert. Als voorbeeld geeft dit voor `NOT [(2 > 1) AND (8 < 9)]`:

```

        ldc 2
        ldc 1
        if_icmpgt andLabel
        goto waarLabel
andLabel:
        ldc 8
        ldc 9
        if_icmplt valsLabel
        goto waarLabel

```

### 5.2.2 PERFORM

Het `PERFORM` statement zorgt ervoor dat een procedure wordt uitgevoerd, na het uitvoeren, er wordt teruggesprongen en verdergegaan met de statements na `PERFORM`. Dit kan in bytecode verkregen worden door het gebruik van de opcodes `jsr` en `ret`.

Er zijn 2 vormen van het `PERFORM` statement onderzocht, een `PERFORM` zonder- en met conditie. Het onderzochte `PERFORM` statement zonder conditie is in COBOL van de vorm:

```
PERFORM procedure-name.
```

Dit zal er in Jasmin formaat uitzien als:

```

        jsr proclabel
        ; code voor statements na PERFORM

proclabel:
        ; code voor statements van deze procedure
        astore <x> ; opslaan van referentie om terug te keren
        ret <x>   ; terugspringen via referentie

```

Het onderzochte `PERFORM` statement met conditie is in COBOL van de vorm:

```
PERFORM procedure-name UNTIL condition.
```

Hier wordt de procedure uitgevoerd totdat de conditie naar waar evalueert. De conditie wordt op dezelfde manier geconstrueerd zoals uitgelegd bij het `IF` statement. Dit geeft in Jasmin formaat:

```

beginLabel:
        if_icmp<cond> eindLabel ; als de conditie waar is,

```

```

                                ; geen PERFORM meer uitvoeren
    jsr procLabel
    goto beginLabel ; terug testen van de conditie
eindLabel:
    ; code voor statements na PERFORM

procLabel:
    ; code voor statements van deze procedure
    astore <x> ; opslaan van referentie om terug te keren
    ret <x> ; terugspringen via referentie

```

### 5.3 Implementatie

Net zoals bij de statements behandeld in hoofdstuk 4, wordt de omzetting van de statements voor flow control, verzorgd door een uitbreiding van de `TransformFunction` klasse.

Naast het opvragen van de variabelen aan het `DataKeeper` object, moeten de Jasmin labels die gebruikt worden, ook verkregen worden. Het `DataKeeper` object heeft hiervoor de methodes `getLabel(String procedureName)` en `getNewLabel()`. De eerste methode geeft een string terug die een mapping is van een procedurenaam in COBOL naar een label in Jasmin of een nieuw label indien de procedurenaam nog niet gekend is. De tweede methode geeft een string terug die een nieuw label voorstelt. Het teruggegeven label is van de vorm `label<x>` met `<x>` een getal dat geïncrementeerd wordt, telkens een label wordt aangemaakt door het `DataKeeper` object.

Het aanmaken van de bytecode die de label van een procedure aangeeft, gebeurt in de klasse `Procedure`, door het oproepen van:

```

private void processParaName (Node paraNode) throws XPathException
{
    String name = (String) xpath.evaluate("cobol_word/string", paraNode,
                                         XPathConstants.STRING);
    String javaLabel = keeper.getLabel(name);
    JVM.writeCode(javaLabel+":");
}

```

De klasse `Procedure` zorgt tevens voor het aanmaken van de bytecode voor het terugspringen nadat een procedure is doorlopen. De methode die hiervoor zorgt geeft:

```

private void processSentence (Node sentenceNode)
{
    // statements handling
    NodeList children = sentenceNode.getChildNodes();
    for (int i = 0; i < children.getLength(); i++)

```

```

    {
        Node tmp = children.item(i);
        if (tmp.getNodeName().equals("statement"))
        {
            Statement s = new Statement(keeper, JVM, tmp);
            s.process();
        }
    }
    // use a free position in JVM's local vars
    position = keeper.usePosition();
    // write code for returning to main-procedure
    JVM.writeCode("astore "+position);
    JVM.writeCode("ret "+position);
}

```

### IfStatement

Het object dat zorgt voor de omzetting van een IF statement is `IfStatement`. Hier worden er 3 nieuwe labels aangemaakt, deze horen bij de code voor een ware conditie, een valse conditie en het einde van het IF statement. De conditie wordt verwerkt door een `Condition` object die de `condition` node en de labels voor waar en vals meekrijgt. Nadat de bytecode voor een conditie is aangemaakt worden de bijhorende statements verwerkt door het `Statement` object.

```

condition = new Condition (conditionNode, JVM, keeper, trueLabel, falseLabel);
condition.Execute();
JVM.writeCode(falseLabel+":");
Statement f = new Statement(keeper, JVM, falseNode);
f.process();
JVM.writeCode("goto "+endLabel);
JVM.writeCode(trueLabel+":");
Statement t = new Statement(keeper, JVM, trueNode);
t.process();
JVM.writeCode(endLabel+":");

```

### Perform

Net zoals bij `IfStatement` wordt door `Perform`, ook 3 nieuwe labels aangemaakt, een label dat wijst naar het begin van het statement, een label dat het einde van het `PERFORM` statement aanduidt en een label waarnaar moet gesprongen worden indien de conditie naar vals evalueert. De conditie die wordt aangemaakt zal als label voor waar, het einde van het `PERFORM` statement meekrijgen, samen met de label voor vals, namelijk deze die zorgt voor het springen naar de procedure.

```

condition = new Condition (conditionNode, JVM, keeper, endLabel, falseLabel);

```

```
JVM.writeCode(beginLabel+":");
if (hasCondition)
    condition.Execute();
JVM.writeCode(falseLabel+":");
JVM.writeCode("jsr "+procedureLabel);
if (hasCondition)
    JVM.writeCode("goto "+beginLabel);
JVM.writeCode(endLabel+":");
```

### Condition

Het object dat bytecode aanmaakt voor een conditie is op dezelfde manier opgebouwt als de `Computational` klasse. De klasse `Condition` krijgt de XML node mee die de conditie bevat. Door de kinderen van de node te overlopen wordt een bijpassende methode opgeroepen die de bytecode zal aanmaken. Een `arithmetic_expression` node wordt verwerkt door het aanmaken van een `Computational` object.

```
public void processNode(Node n, String trueJump , String falseJump ) {
    if(n.getNodeName().equals("condition"))
        processOr(n,trueJump,falseJump);
    else if(n.getNodeName().equals("logical_and_expression"))
        processAnd(n,trueJump,falseJump);
    else if(n.getNodeName().equals("abbreviated_combined_condition"))
        processNode(n.getFirstChild(),trueJump,falseJump);
    else if(n.getNodeName().equals("simple_condition"))
        processRelation(n,trueJump,falseJump);
    else if(n.getNodeName().equals("arithmetic_expression"))
    {
        Computational comp = new Computational(keeper,JVM,n);
        comp.process();
    }
    else
        processNode(n.getNextSibling(),trueJump,falseJump);
}
```

Met als voorbeeld de implementatie van de methode die zorgt voor het verwerken van de AND combinator:

```
public void processAnd(Node n , String trueJump, String falseJump) {
    String xprType = (String) xpath.evaluate("string", n,
        XPathConstants.STRING);
    Node xpr1Node = (Node) xpath.evaluate(
        "abbreviated_combined_condition[1]",n,XPathConstants.NODE);
    Node xpr2Node = (Node) xpath.evaluate(
        "abbreviated_combined_condition[2]",n,XPathConstants.NODE);
    if(xprType.equalsIgnoreCase("AND"))
```

```
{
    String secondCmp = keeper.getNewLabel();
    processNode(xpr1Node, secondCmp, falseJump);
    JVM.writeCode("goto "+falseJump);
    JVM.writeCode(secondCmp+":");
    processNode(xpr2Node, trueJump, falseJump);
}
else
    processNode(xpr1Node, trueJump, falseJump);
}
```

## Hoofdstuk 6

# Een uitgewerkt voorbeeld

### 6.1 Het MOVE statement

Bij het omzetten van COBOL statements naar Java bytecode, wordt voor elk statement een gelijkaardig proces doorlopen. Dit proces wordt in dit hoofdstuk meer in detail besproken, door wat dieper in te gaan op de omzetting van het MOVE statement.

#### 6.1.1 MOVE statement in COBOL

Het MOVE statement verplaatst data naar één of meer data-items. Om de syntax en de werking van het statement te verstaan, wordt er gebruik gemaakt van de COBOL-85 taal referentie [7]. Er zijn 7 verschillende formaten beschikbaar van het MOVE statement. Enkel het meest gebruikte formaat wordt onder de loep genomen. Dit ziet er in COBOL uit als:

```
MOVE data-item-1 TO data-item-2.
```

Hierbij is `data-item-1` het zendend item en `data-item-2` het ontvangend item. Er zijn wel enkele beperkingen op het zendend en ontvangend item, zoals welke types er kunnen worden gebruikt. Deze beperkingen worden aangeduid door enkele regels waar het MOVE statement zich moet aan houden. De belangrijkste regels zijn hieronder opgesomt.

1. Elke nodige omzetting van data van de éne vorm van interne voorstelling naar een ander vindt plaats, samen met eventueel editeren of de-editeren. Daarbij gebeurt ook de alignering met het decimale punt en het aanvullen met nullen, behalve waar nullen vervangen worden door editeren.
2. Een alfanumeriek-edited data-item of een alfabetisch data-item kunnen niet verplaatst

worden naar een numeriek-edited data-item.

3. Een numeriek data-item kan niet verplaatst worden naar een alfabetisch data-item.
4. Numerieke niet gehele data kan niet ontvangen worden door een alfanumeriek of door een alfanumeriek-edited data-item.
5. Als het zendend item numeriek met een teken is, wordt het teken niet doorgegeven, indien het ontvangend item alfanumeriek is.
6. Als het zendend item numeriek-edited is, wordt er niet aan de-editeren gedaan, indien het ontvangend item alfanumeriek is.
7. Indien het ontvangend item numeriek of numeriek-edited is en het zendend item numeriek-edited, wordt aan de-editeren gedaan. Deze niet geëditteerde data wordt doorgegeven aan het ontvangend item.
8. Indien het ontvangend item numeriek met een teken is, wordt het teken doorgegeven aan het ontvangend item. Omzetting om aan de gebruikte interne voorstelling van het teken te voldoen gebeurt hierna. Indien het zendend item geen teken heeft, krijgt het ontvangend item een positief teken.
9. Indien het ontvangend item numeriek zonder teken is, wordt de absolute waarde van het zendend item doorgegeven. Dit betekent dat een negatieve waarde zo positief komt.
10. Als het ontvangend item alfanumeriek is, gebeurt er afkapping of spatievulling indien nodig.

Deze regels moeten toegepast worden op de data-items die we voorstellen op de JVM. Tabel 6.1 toont de toepassing van die regels op de beschikbare data-items. Aan de hand van de tabel zal beslist worden welke opcodes we zullen gebruiken.

van/naar	Tekstueel	Numeriek zonder teken	Numeriek met teken	Niet geheel numeriek	Numeriek-edited
<b>Tekstueel</b>	<ul style="list-style-type: none"> <li>o alignatie links</li> </ul>	<ul style="list-style-type: none"> <li>o doorgeven als geheel getal zonder teken</li> <li>o alignatie rechts</li> </ul>	<ul style="list-style-type: none"> <li>o doorgeven als getal met positief teken</li> <li>o alignatie rechts</li> </ul>	<ul style="list-style-type: none"> <li>o doorgeven als geheel getal</li> <li>o alignatie met decimaal punt</li> </ul>	<ul style="list-style-type: none"> <li>o doorgeven als geheel getal</li> <li>o editeren</li> </ul>
<b>Numeriek zonder teken</b>	<ul style="list-style-type: none"> <li>o alignatie links</li> </ul>	<ul style="list-style-type: none"> <li>o alignatie rechts</li> </ul>	<ul style="list-style-type: none"> <li>o doorgeven als getal met positief teken</li> <li>o alignatie rechts</li> </ul>	<ul style="list-style-type: none"> <li>o alignatie met decimaal punt</li> </ul>	<ul style="list-style-type: none"> <li>o editeren</li> </ul>
<b>Numeriek met teken</b>	<ul style="list-style-type: none"> <li>o doorgeven als getal zonder teken</li> <li>o alignatie links</li> </ul>	<ul style="list-style-type: none"> <li>o alignatie rechts</li> </ul>	<ul style="list-style-type: none"> <li>o alignatie rechts</li> </ul>	<ul style="list-style-type: none"> <li>o alignatie met decimaal punt</li> </ul>	<ul style="list-style-type: none"> <li>o editeren</li> </ul>
<b>Numeriek-edited</b>	<ul style="list-style-type: none"> <li>o alignatie links</li> </ul>	<ul style="list-style-type: none"> <li>o de-editeren</li> <li>o alignatie rechts</li> </ul>	<ul style="list-style-type: none"> <li>o de-editeren</li> <li>o alignatie rechts</li> </ul>	<ul style="list-style-type: none"> <li>o de-editeren</li> <li>o alignatie met decimaal punt</li> </ul>	<ul style="list-style-type: none"> <li>o de-editeren</li> <li>o doorgeven als numeriek</li> <li>o editeren</li> </ul>

Tabel 6.1: Regels te ondergaan voor de verschillende types

### 6.1.2 Equivalent in Java bytecode

Het MOVE statement moet data-items van een bepaalde vorm doorgeven naar data-items van een eventueel andere vorm en de gepaste alignatie toepassen. De data-items zijn opgeslagen als char arrays en voor de alignatie beschikken we over de methodes in Tabel 3.5. Om het MOVE statement om te zetten zal er dus gebruik gemaakt worden van een combinatie van deze methodes. In wat volgt geven we een overzicht van de mogelijke verplaatsingen van data-items.

#### Tekstueel naar Tekstueel

Voor het verplaatsen van tekstuele data naar andere tekstuele data moet enkel rekening worden gehouden met de alignatie links. De methode `setContent` van `CobolString` zorgt hiervoor. Het MOVE statement in Jasmin formaat hiervoor is dan:

```
aload <x> ; op stack plaatsen van het zendend item
aload <y> ; op stack plaatsen van het ontvangend item
invokestatic cobol/CobolString/setContent([C[C)V
```

#### Tekstueel naar Numeriek

Hier moet de tekstuele data een getal voorstellen. Dit getal is altijd een geheel getal, aangezien tekstuele data geen plaatshouder heeft voor het decimaal punt. Dit wil zeggen dat de data eerst kan omgezet worden naar een getal van het `int` of `long` type, de keuze van het geschikte type gebeurt op basis van de lengte van de char array van de tekstuele data. Het ontvangend item is numeriek, daardoor mag een teken enkel worden doorgegeven indien het ontvangend item numeriek met teken is. De geschikte `setContent` methode van `CobolNumber` zorgt daarvoor. Dit kan in Jasmin formaat voorgesteld worden als:

```
aload <x> ; op stack plaatsen van het zendend item
ldc <lengte van array> ; plaats van het decimaal punt
; omzetten naar int, of long indien de lengte van de array > 10
invokestatic cobol/CobolNumber/toInteger([CI)I
aload <y> ; op stack plaatsen van het ontvangend item
ldc <decimale plaats van ontvangend item>
invokestatic cobol/CobolNumber/setContent(I[CI)V
```

#### Tekstueel naar Numeriek-edited

Het MOVE statement voor tekstuele data naar numeriek-edited data moet in principe hetzelfde effect hebben als de tekstuele data omzetten naar numerieke data en deze numerieke data doorgeven aan de numeriek-edited data.

In ACUCOBOL wordt dit niet gedaan, daar wordt de data enkel doorgegeven en alignatie toegepast voor tekstuele data. Ik heb ervoor gekozen om dit niet zo te doen, dit leidt namelijk tot inconsistente data.

In plaats van gebruik te maken van een `setContent` methode van `CobolNumber` wordt er gebruik gemaakt van `setContent` van `CobolEdited`, hier moet naast het getal en de char array ook de picture string van het numeriek-edited data-item worden doorgegeven. Dit omzetten naar Jasmin geeft:

```
aload <x> ; op stack plaatsen van het zendend item
ldc <lengte van array> ; plaats van het decimaal punt
invokestatic cobol/CobolNumber/toInteger([CI)I ; of toLong
aload <y> ; op stack plaatsen van het ontvangend item
ldc "<picture>"
invokestatic cobol/CobolEdited/setContent(I[CLjava/lang/String;)V
```

### Numeriek naar Tekstueel

Het verplaatsen van gehele numerieke data zonder teken naar tekstuele data kan gewoon gebeuren door het kopiëren van de inhoud van de char array en het toepassen van de alignatie voor tekstuele data. Dit wil zeggen dat er kan gebruik gemaakt worden van de methode `setContent` van `CobolString`. Dit geeft in Jasmin hetzelfde als het MOVE statement voor tekstuele data naar tekstuele data.

Gehele numerieke data met teken moet volgens regel 5 doorgegeven worden zonder teken. Nadat deze data geen teken meer heeft kan de data op dezelfde manier doorgegeven worden als hierboven, namelijk via `setContent` van `CobolString`. Om het zendend item tekenloos te maken kan er een methode geschreven worden die het teken verwijderd of er kan er een nieuwe char array aangemaakt worden die het zendend item voorstelt, maar dan zonder teken. Deze array krijgt zijn inhoud dan door het toepassen van `toInteger` of `toDouble` op het origineel data-item en `setContent` voor het nieuwe data-item. Er is gekozen voor het aanmaken van een nieuwe char array. In Jasmin formaat geeft dit:

```
aload <x> ; op stack plaatsen van het zendend item
ldc <dec place> ; plaats van het decimaal punt
invokestatic cobol/CobolNumber/toInteger([CI)I ; of toLong
; code voor aanmaken nieuw char array die getal zonder teken voorstelt
ldc <decimale plaats van ontvangend item>
invokestatic cobol/CobolNumber/setContent(I[CI)V
aload <y> ; op stack plaatsen van het ontvangend item
invokestatic cobol/CobolString/setContent([C[C)V
```

Het doorgeven van niet gehele numerieke data kan niet volgens regel 4. Dit is dan ook niet onderzocht.

### Numeriek naar Numeriek

Het doorgeven van numerieke data naar andere numerieke data moet enkel rekening houden met de alignatie van het decimaal punt en het al dan niet doorgeven van het teken. Dit kan eenvoudig bekomen worden door het omzetten van een char array naar een int, long of double en dit getal als parameter te gebruiken voor een `setContent` methode van `CobolNumber`.

### Numeriek naar Numeriek-edited

Het is duidelijk dat dit MOVE statement kan uitgewerkt worden als het omzetten van de numerieke data naar een int, long of double en deze waarde doorgeven aan de `setContent` methode van `CobolEdited`. In Jasmin geeft dit voor een int:

```
aload <x> ; op stack plaatsen van het zendend item
ldc <dec place> ; plaats van het decimaal punt
invokestatic cobol/CobolNumber/toInteger([CI)I
aload <y> ; op stack plaatsen van het ontvangend item
ldc "<picture>"
invokestatic cobol/CobolEdited/setContent(I[CLjava/lang/String;)V
```

### Numeriek-edited naar Tekstueel

Voor deze MOVE moet regel 6 toegepast worden, dit wil zeggen dat de waarde van het zendend item gewoon doorgegeven kan worden aan het ontvangend item, samen met het toepassen van alignatie voor tekstuele data. Dit is dus equivalent met het geval voor tekstueel naar tekstueel.

### Numeriek-edited naar Numeriek

Om numeriek-edited data om te zetten naar numerieke data, moet volgens regel 7, het zendend item eerst in niet geëditteerde vorm verkregen worden. Dit kan door de data om te zetten naar een int, long of double, door gebruik te maken van een statische methode van `CobolEdited`. Dit getal kan dan via `setContent` van `CobolNumber` doorgegeven worden aan de numerieke variabele.

### Numeriek-edited naar Numeriek-edited

Net zoals bij numberiek-edited naar numeriek, moet voldaan zijn aan regel 6. De data wordt eerst in ongeëditeerde vorm gezet, waardoor een getal verkregen wordt. Een `setContent` methode van `CobolEdited` zal dit getal dan terug editeren mbv. de picture van het ontvangend item.

### 6.1.3 Toepassing op programma

Éénmaal de mapping van het MOVE statement gevonden is, kunnen we de ontworpen applicatie uitbreiden. Er moet daarvoor een nieuw object gemaakt worden die de mapping verwerkt. In wat volgt wordt uitgelegd hoe zo'n object moet worden aangemaakt.

#### Aanmaken van de TransformFunction Move

Het object dat het MOVE statement behandelt is `Move`, een uitbreiding van de abstracte klasse `TransformFunction`. Zo'n object krijgt een XML node die verwerkt moet worden. Al het werk gebeurt door de implementatie van de methode `processInstruction`. Deze methode moet de variabelenaam van het zendend item en het ontvangend item ophalen uit de XML en voor elke combinatie van types, de bijhorende bytecode aanmaken.

#### Ophalen van nodige informatie uit de XML-source

De XML die gegenereerd wordt voor het stukje COBOL code

```
MOVE Number1 TO Number2.
```

staat hieronder, via XPATH kunnen de variabelennamen worden opgehaald.

```
<move>
  <string>MOVE</string>
  <!-- -->
  <any_value>
    <cobol_word>
      <string>Number1</string>
    </cobol_word>
  <!-- -->
</any_value>
<string>TO</string>
<!-- -->
<data_item>
  <cobol_word>
    <string>Number2</string>
  </cobol_word>
</data_item>
```

```
</move>
```

De XPATH expressies beschrijven het XML pad naar de data die we nodig hebben. Deze expressies evalueren, haalt deze data op.

```
String fromName = (String) xpath.evaluate("any_value/cobol_word/string::text()",
    inNode, XPathConstants.STRING);
String toName = (String) xpath.evaluate("data_item/cobol_word/string::text()",
    inNode, XPathConstants.STRING);
```

Met behulp van deze variabelennamen worden de corresponderende `DataItem` objecten aangevraagd bij het `DataKeeper` object.

```
DataItem fromData = keeper.getDataItem(fromName);
DataItem toData = keeper.getDataItem(toName);
```

### Inbouwen van bytecode

De bekomen `DataItem` objecten moeten gecontroleerd worden op hun type, aan de hand van het type zal de bytecode hiervoor beschreven, worden aangemaakt. Daarvoor moet er gebruik gemaakt worden van methodes van de klasse `JVMSimulation`. Voor het geval tekstueel naar numeriek wordt de volgende code doorlopen:

```
if (fromData.getType()==DataItem.COBOLCHARARRAY &&
    toData.getType()==DataItem.COBOLINTEGER)
{
    int decCount = fromData.getLength();
    JVM.push(fromData);
    JVM.pushNativeConstant(decCount);
    if(decCount > 10)
        JVM.writeCode("invokestatic cobol/CobolNumber/toLong([CI)L");
    else
        JVM.writeCode("invokestatic cobol/CobolNumber/toInteger([CI)I");
    JVM.push(toData);
    JVM.pushNativeConstant(((CobolNumber)toData).getIntDigits());
    if(decCount > 10)
        JVM.writeCode("invokestatic cobol/CobolNumber/setContent(L[CI)V");
    else
        JVM.writeCode("invokestatic cobol/CobolNumber/setContent(I[CI)V");
}
```

De andere combinaties van elke 2 types, krijgen een gelijkaardige implementatie.

### Registreren van de Move functie

Nadat de implementatie van `Move` geschreven is, moet de klasse nog gecompileerd worden. Hierna is het object klaar voor gebruik. Om de applicatie kennis te geven van het nieuw

gemapte statement moet er enkel nog een regel toegevoegd worden aan een XML file. Aan `statement_mapping.xml` voeg je de onderstaande regel toe.

```
<move>Cobol2Bytecode.transformer.Move</move>
```

De nodenaam is dezelfde als de node die verwerkt moet worden, namelijk de `move` node. De inhoud van de node is het pad waar het object zich bevindt. Het `Move` object bevindt zich in de package `Cobol2Bytecode.transformer`.

## Hoofdstuk 7

# Conclusies en verder onderzoek

### **De doelstelling werd bereikt**

Aan de hand van de thesis werd aangetoond dat het omzetten van COBOL sourcecode naar instructies voor de JVM mogelijk is. Dit werd bekomen door de meest gebruikte statements en data declaraties te mappen naar bytecode. Ook flow control bleek geen probleem te zijn, door gebruik te maken van een combinatie van JVM opcodes die een invloed hebben op het programmaverloop.

### **Nog niet geschikt voor industriële toepassingen**

Er is echter wel een groot nadeel, het vraagt heel veel werkt om heel de specificatie [7] van bijvoorbeeld enkel COBOL-85 om te zetten in bytecode. COBOL bezit namelijk meer dan 500 sleutelwoorden. Naast het groot aantal statements die een COBOL applicatie kan bevatten, heeft elk statement meestal ook nog verschillende vormen. Deze moeten dan ook apart onderzocht en behandeld worden. Om dan de complexiteit van het probleem nog wat door te drijven, bezit de COBOL taal ook veel dialecten. Indien er zich wordt gehouden aan een basis van de COBOL taal, heeft het zin om de thesis verder uit te breiden.

### **Een aanzet tot uitbreiding**

Het uitbreiden van de subset van statements, werd door de opbouw van de bijhorende applicatie vereenvoudigd. Voor elk statement die men wil toevoegen aan de onderzochte subset, moet enkel een Java object geïmplementeerd worden en een XML bestand aangepast. Hierdoor is het tevens mogelijk om de applicatie toe te passen op een ander COBOL dialect. Dit kan namelijk

door andere Java objecten aan te maken en het XML bestand aan te passen voor de nieuwe objecten.

### **De bytecode is toepasbaar**

Eénmaal er bytecode wordt verkregen, is de bytecode ook toepasbaar op andere applicaties. De JVM is immers heel populair en deze populariteit neemt nog steeds toe. Dit heeft tot gevolg dat er een hele reeks tools beschikbaar zijn die inwerken op Java bytecode. Enkele voorbeelden van toepassingen, is het omzetten van Java bytecode naar Java sourcecode en het gebruik van optimalisatietechnieken op de bytecode [10, 11].

### **COBOL wordt platformafhankelijk**

De JVM zorgt als virtuele machine voor platformafhankelijkheid. Door de JVM als doelmachine te kiezen, wordt COBOL meteen platformafhankelijk. Een bijkomend voordeel van de populariteit van de JVM, is dat er voor nieuwe technologieën, er meestal een JVM beschikbaar wordt gesteld. De thesis zorgt ervoor dat ook COBOL beschikbaar wordt voor deze nieuwe technologieën.

Om de platformafhankelijk aan te tonen, werd een programma, geschreven in COBOL, uitgevoerd op een emulatie van een GSM met Java technologie (zie hiervoor Bijlage B). Het is vast niet echt nuttig om COBOL programma's te schrijven voor een GSM, maar dit toont wel aan dat COBOL niet dood is. Zoals m'n thesisbegeleider Kris, onderzoek doet naar "Helping old music to a fresh beat" [9], krijgt de beat hier zelfs een polyfone toon.

### **Combineren van Java en COBOL**

Door de mogelijkheid om rechtstreeks, in Java bytecode, Java methodes aan te roepen, wordt het mogelijk om Java en COBOL te combineren bij het ontwerpen van een applicatie. COBOL programma's kunnen zo Java methodes oproepen en programma's geschreven in Java kunnen gebruik maken van procedures in COBOL. Dit is namelijk heel belangrijk in de toepassing op de bedrijfswereld. Door het uitsterven van kennis over COBOL, is het noodzakelijk om COBOL applicaties op termijn om te zetten naar een andere omgeving.

## Bijlage A

# Uitvoeren van de applicatie

In wat volgt, wordt verduidelijkt hoe de geschreven applicatie uitgevoerd moet worden. Dit gebeurt aan de hand van een voorbeeld.

In de verschillende hoofdstukken van de thesis, werd als verduidelijking, stukjes bytecode gegeven. In deze bijlage wordt de gegenereerde bytecode getoond van een compleet programma. Het COBOL programma dat hier wordt omgezet is een procedure voor het berekenen van de grootste gemene deler van 2 getallen. Dit voorbeeld werd gekozen omdat hier de meeste van de besproken statements worden toegepast. Het bevat namelijk invoer van toetsenbord, uitschrijven naar het scherm, rekenkundige bewerkingen en flow control. De COBOL sourcecode van het programma staat hieronder.

```
identification division.
program-id. grootstegemenedeler.
data division.
working-storage section.
77      getal1          pic 999.
77      getal2          pic 999.
procedure division.
hoofd.
        display "Geef het eerste getal in."
        accept  getal1
        display "Geef het tweede getal in."
        accept  getal2
        perform verwerk until getal1 = getal2
        display "De grootste gemene deler is " getal1
        stop run.

verwerk.
        if getal1 > getal2
        subtract getal2 from getal1
```

```
else
  subtract getal1 from getal2
end-if.
```

Dit programma, samen met het bijhorende XML bestand en de bytecode, is ook te vinden op de CD-ROM, onder de map voorbeelden. Op de CD-ROM vind je naast de sourcecode en de javadocs, van de geschreven applicatie, een `.jar` bestand, in de map `build`. Als parameters bij het uitvoeren, moet het XML bestand worden opgegeven, samen met de naam van de te maken Java klasse. Indien alle nodige bestanden zich in eenzelfde map bevinden, voeren we de applicatie uit door:

```
java -jar Cobol2Bytecode.jar ggd.xml GrootsteGemeneDeler
```

De bytecode die wordt aangemaakt, wordt in het bestand `GrootsteGemeneDeler.j` geplaatst.

De inhoud vindt je hieronder.

```
;Classfile version:
;   Major: 46
;   Minor: 0
.source ggd.xml
.class public synchronized GrootsteGemeneDeler
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
.throws java/lang/Exception
.limit stack 15
.limit locals 15
; DATADESCRIPTION - getal1
bipush 3
newarray char
dup
bipush 0
bipush 48
castore
dup
bipush 1
bipush 48
castore
dup
bipush 2
bipush 48
castore
astore 1
; DATADESCRIPTION - getal2
bipush 3
newarray char
dup
bipush 0
bipush 48
castore
dup
```

```
bipush 1
bipush 48
castore
dup
bipush 2
bipush 48
castore
astore 2
; PROCEDURE - hoofd
Label1:
; DISPLAY
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Geef het eerste getal in."
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
; ACCEPT
aload 1
iconst_1
invokestatic cobol/HelperClass/readFromInput([CZ)V
; DISPLAY
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Geef het tweede getal in."
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
; ACCEPT
aload 2
iconst_1
invokestatic cobol/HelperClass/readFromInput([CZ)V
; PERFORM - verwerk
Label3:
aload 1
bipush 3
invokestatic cobol/CobolNumber/toInteger([CI)I
aload 2
bipush 3
invokestatic cobol/CobolNumber/toInteger([CI)I
if_icmpeq Label4
Label5:
jsr Label2
goto Label3
Label4:
; DISPLAY
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "De grootste gemene deler is "
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
; DISPLAY
getstatic java/lang/System/out Ljava/io/PrintStream;
aload 1
invokevirtual java/io/PrintStream/println([C)V
; STOP
return
; PROCEDURE - verwerk
Label2:
; IF
aload 1
bipush 3
```

```
invokestatic cobol/CobolNumber/toInteger([CI)I
aload 2
bipush 3
invokestatic cobol/CobolNumber/toInteger([CI)I
if_icmpgt Label6
Label7:
; SUBTRACT
aload 1
bipush 3
invokestatic cobol/CobolNumber/toInteger([CI)I
aload 2
bipush 3
invokestatic cobol/CobolNumber/toInteger([CI)I
isub
aload 2
bipush 3
invokestatic cobol/CobolNumber/setContent(I[CI)V
goto Label8
Label6:
; SUBTRACT
aload 2
bipush 3
invokestatic cobol/CobolNumber/toInteger([CI)I
aload 1
bipush 3
invokestatic cobol/CobolNumber/toInteger([CI)I
isub
aload 1
bipush 3
invokestatic cobol/CobolNumber/setContent(I[CI)V
Label8:
astore 4
ret 4
return
.end method
```

Dit aangemaakte bestand wordt omgezet in een `.class` bestand met behulp van Jasmin. De nodige `.jar` hiervoor, bevindt zich tevens op de CD-ROM, onder de map Software. Bij het uitvoeren, geven we als parameter het `.j` bestand op:

```
java -jar jasmin.jar GrootsteGemeneDeler.j
```

Hierdoor wordt het bestand `GrootsteGemeneDeler.class` aangemaakt. Uitvoeren van dit bestand kan natuurlijk met:

```
java GrootsteGemeneDeler
```

Voor het uitvoeren van de klasse, moet echter wel gezorgd worden dat de `CLASSPATH` verwijst naar `Cobol2ByteCode.jar`, de gebruikte methodes uit Tabel 3.5 bevinden zich namelijk ook in dit bestand.

## Bijlage B

# COBOL op een mobiel toestel

Zoals in de conclusie werd vermeld, wordt het platformafhankelijk maken van COBOL, aange-  
toond door een eenvoudige applicatie geschreven in COBOL, uit te voeren op een emulatie van  
een GSM met Java technologie beschikbaar. Deze bijlage bespreekt hoe dit kan worden uitgetest.

Ik merk hierbij wel op dat het zeker niet mogelijk is om alle gegenereerde bytecode uit  
te voeren op een mobiel toestel. De gebruikte JVM bij een GSM is echter beperkt tot J2ME.  
Bepaalde geschreven methodes die data in COBOL omzet in een int,long of double maken echter  
gebruik van klassen die niet behoren tot de beperkte J2ME.

Het gebruikte voorbeeld is `performtest.cbl`, te vinden op de bijgeleverde CD-ROM. Dit  
bestand werd via de ontworpen applicatie omgezet naar bytecode. Deze bytecode werd door  
Jasmin omgezet in `Performtest.class`. De COBOL sourcecode is:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. PerformFormat1.  
  
PROCEDURE DIVISION.  
TopLevel.  
    DISPLAY "In TopLevel. Starting to run program"  
    PERFORM OneLevelDown  
    DISPLAY "Back in TopLevel."  
    STOP RUN.  
  
TwoLevelsDown.  
    DISPLAY ">>>>>>> Now in TwoLevelsDown."  
    PERFORM ThreeLevelsDown.  
    DISPLAY ">>>>>>> Back in TwoLevelsDown."  
  
OneLevelDown.  
    DISPLAY ">>>> Now in OneLevelDown"  
    PERFORM TwoLevelsDown
```



hoe een procedure, geschreven in COBOL, uitgevoerd wordt door een programma, geschreven in Java.

```
import javax.microedition.midlet.*;

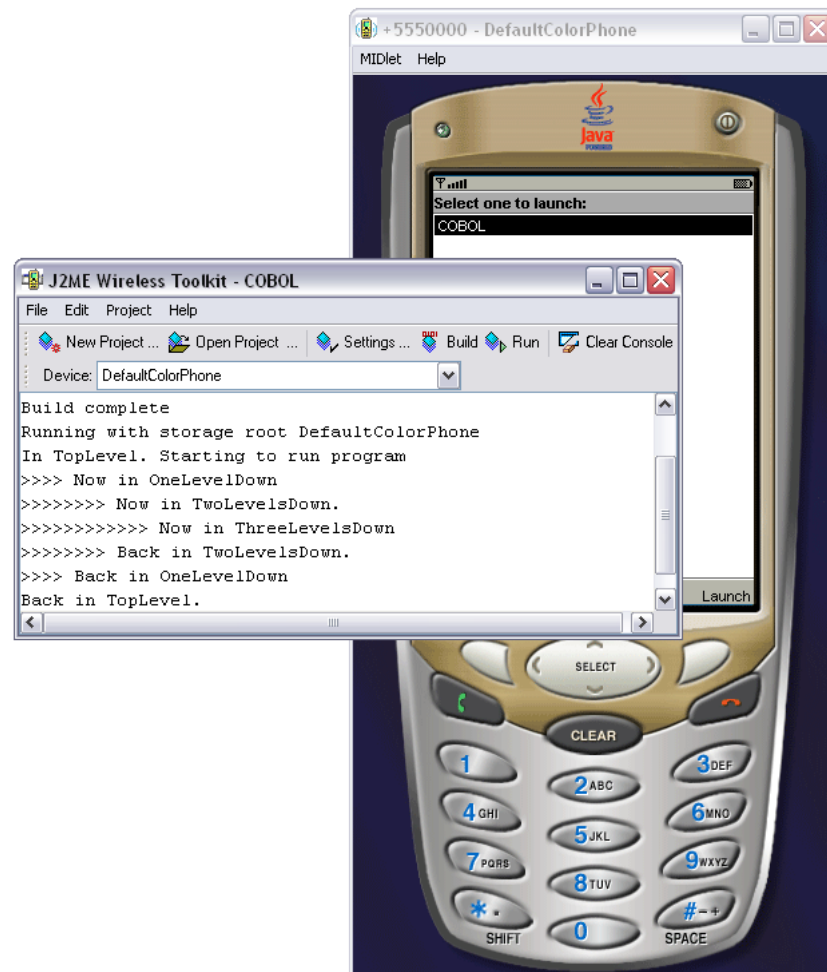
public class CobolApp extends MIDlet
{
    public CobolApp () { }

    public void startApp()
    {
        // uitvoeren COBOL
        try{Performtest.main(null);}catch(Exception e){}
    }

    public void pauseApp() { }

    public void destroyApp(boolean unconditional) { }
}
```

Figuur B.2: Inhoud van CobolApp.java.



Figuur B.3: Het uitvoeren van performttest.cbl op een GSM emulatie.

# Lijst van figuren

2.1	Een 'Hello World' - voorbeeld. . . . .	5
2.2	Gebruik van de JVM als back-end . . . . .	7
2.3	De virtuele architectuur van de JVM . . . . .	10
2.4	Voorstelling van de stack tijdens het uitvoeren . . . . .	13
2.5	'Hello World' als DOM document . . . . .	15
3.1	DateOfBirth als group item. . . . .	17
3.2	Alignatie van numerieke data. . . . .	23
3.3	XML transformatie van de DATA DIVISION. . . . .	31
3.4	UML schema van de DataItem klasse. . . . .	32
3.5	Aanmaken van bytecode voor variabelen. . . . .	35
4.1	Boomstructuur van een Expression. . . . .	42
4.2	UML schema van de Expression klasse. . . . .	43
B.1	Aanmaken van een nieuw project. . . . .	71
B.2	Inhoud van CobolApp.java. . . . .	72
B.3	Het uitvoeren van performttest.cbl op een GSM emulatie. . . . .	73

# Lijst van tabellen

2.1	De mijlpalen in de geschiedenis van COBOL . . . . .	4
2.2	Versies van het Java platform . . . . .	6
2.3	Types beschikbaar voor de JVM . . . . .	8
3.1	De verschillende COBOL categorieën . . . . .	17
3.2	Symbolen van de picture clause . . . . .	19
3.3	Het gebruik van ASCII voor EBCDIC. . . . .	21
3.4	Numerieke data als karakterstrings. . . . .	22
3.5	Methodes beschikbaar voor het omvormen van data. . . . .	25
4.1	Aanmaken van Expression objecten. . . . .	41
5.1	Opcodes voor het vergelijken van 2 int waarden. . . . .	47
6.1	Regels te ondergaan voor de verschillende types . . . . .	57

# Bibliografie

- [1] Languages for the JVM, <http://www.robert-tolksdorf.de/vmlanguages.html>
- [2] J. Engel, “Programming for the Java<sup>TM</sup> Virtual Machine”, *Addison-Wesley Professional*, June 1999.
- [3] T. Lindholm, F. Yellin, “The Java<sup>TM</sup> Virtual Machine Specification, Second Edition”, *Addison-Wesley Professional*, April 1999.
- [4] Jasmin, a Java Assembler Interface, <http://jasmin.sourceforge.net>
- [5] DJava, a Java disassembler, <http://cat.nyu.edu/~meyer/jvm/djava/>
- [6] ACUCOBOL, <http://www.acucorp.com/>
- [7] SRDI, “COBOL 85 language reference”, September 2000.
- [8] M. Murach, A. Prince, R. Menendez, “Murach’s structured COBOL: training & reference”, *Mike Murach and Associates, Inc.*, 2000.
- [9] R. Lämmel, K. De Schutter, “What does aspect-oriented programming mean to Cobol?”, in *Proceedings of Aspect-Oriented Software Development (AOSD 2005)*, Chicago, USA, March 2005.
- [10] S. Srinivasan, K.S. Ranganath, “Simulation and analysis of the Java Platform.”, Fall 2001.
- [11] E. Gagnon, L. Hendren, “SableVM: A Research Framework for the Efficient Execution of Java Bytecode”, in *Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, USA, April 2001.