Compiler V2

Versionen

Datum	Notiz	Version
31.03.2019	Integration des Policy-Subsystems Differenzierung zwischen Definitionen und Instanzen (Aktivitäten und Policies) Integration des ContextInformation-Subsystems Diverse Überarbeitungen der Texte	2.0
30.09.2018	Erste Veröffentlichung Umsetzung der Spezifikation V1.0 vom 30.06.2018	1.0

Autoren

Name	Institution	Email
Fabian Bruckner	Fraunhofer ISST	fabian.bruckner@isst.fraunhofer.de

Compiler

Der Compiler für D° setzt stets eine konkrete Version der D°-Spezifikation um und erlaubt es D°-Code, der zu der entsprechenden Version der Spezifikation konform ist, in ausführbare Data Apps zu übersetzten. Bei der Version des Compilers, die in diesem Dokument beschrieben wird, handelt es sich um die Version 2.0, welche die Spezifikation mit der Version 2.0 umsetzt.

Teile des D°-Compilers wurden in der Programmiersprache Kotlin implementiert. Bei Kotlin handelt es sich um eine JVM Sprache, welche zusammen mit Java verwendet werden kann. Aus diesem Grund sind einige der Codebeispiele in diesem Dokument in Kotlin, wohingegen andere in Java implementiert sind.

Voraussetzungen

Um den D°-Compiler zu verwenden müssen gewisse Anforderungen erfüllt werden, welche in diesem Abschnitt erläutert werden.

	Vorraussetzung	Notiz	Änderungen in Version (en)	Hinzugefügt in Version
1	Funktionsfähiges JDK (>= 8)	Der Compiler ist eine Java Applikation, weswegen eine Java Runtime Environment (JRE) zur Ausführung benötigt wird. Da der Compiler aber auch selbstständig den Übersetzungsvorgang des generierten Java-Codes auslöst, wird darüber hinaus das Java Development Kit (JDK) benötigt. Da das JDK ebenfalls eine JRE mitliefert, kann von einer zusätzlichen Installation abgesehen werden. Die verwendete Java Version muss mindestens Java 8 entsprechen. Es ist zu beachten, dass die (vom Betriebssystem abhängigen) Umgebungsvariablen gesetzt werden, damit Java direkt von der Kommandozeile/Bash aufgerufen werden kann.	-	V1.0
2	Maven (3.3.9) Installation	Bei der generierten Java Applikation handelt es sich um ein Maven-Projekt. Aus diesem Grund wird eine ordnungsgemäß eingerichtete Installation von Apache Maven benötigt. Insbesondere ist darauf zu achten, dass die notwendigen Umgebungsvariablen gesetzt werden, damit der Compiler die Ausführung von Maven auslösen kann. Beispielsweise muss unter Windows die Variable `M2_HOME` gesetzt werden und auf das Wurzelverzeichnis der Installation von Maven zeigen.	-	V1.0

Features

Die nachfolgende Tabelle listet (einige) der Funktionalitäten, welche der D°-Compiler enthält, auf und gibt weiterführende Informationen zu den genannten Features.

	Funktionalität	Notiz	Änderunen in Version (en)	Hinzugefügt in Version
1	Übersetzung von D°-Code in ausführbaren Java-Code	Der Compiler erlaubt es aus einem validen D°-Programm ein ausführbares Java-Programm zu erzeugen. Dabei findet die Übersetzung D°-Code Zielsprachen-Code & die anschließende Übersetzung Zielsprachen-Code Ausführbare Data App in einem Schritt statt. Bei der erzeugten Data App handelt es sich aktuell immer um ein Java-Projekt, welches technologisch auf Spring Boot basiert und eine REST-Schnittstelle bereitstellt, welche gemäß der Angaben im D°-Code ausgestaltet ist. Zur Übersetzung des generierten Java-Codes wird Apache Maven verwendet. D° und der dazugehörige Compiler sind darauf vorbereitet verschiedene Arten von Data Apps (bspw. REST-Data-App, CMD-Data-App, ...) zu generieren, aber im aktuellen Entwicklungsstand sind nur Data Apps, welche eine REST-Schnittstelle bereitstellen verfügbar.	V2.0	V1.0
2	Laden & Verwenden der D°-Corepakete (Datentypen und Aktivitäten)	Die Spezifikation von D° enthält weder integrierte Datentypen noch Funktionalitäten (- Im Kontext von D° werden diese Funktionalitäten Aktivitäten genannt -), welche zur Programmierung verwendet werden können. Stattdessen werden diese Teile der Sprache dem Compiler zur Verfügung gestellt. Der Compiler lädt die entsprechenden Definitionen von Datentypen und Aktivitäten ein und verwendet sie zur Syntaxüberprüfung während des Übersetzungsvorgangs. Um Entwicklern grundlegende Funktionalitäten zu liefern wurden die D°-Corepakete für Datentypen und Aktivitäten entwickelt, welche gemeinsam mit dem Compiler ausgeliefert werden und auch in jeder ausführbaren Data App integriert sind.	V2.0	V1.0
3	Laden & Verwenden des D°-Corepakets für Policies	Analog zu den Erweiterungssystemen für Aktivitäten und Datentypen, verfügt D° über ein erweiterbares Subsystem zur Definition von Policies. Wie auch bei den anderen Subsystemen wird ein Corepaket mitgeliefert, welches einige Policies enthält. Diese Policies können direkt verwendet werden, oder als Orientierungshilfe bei der Implementierung neuer Policies verwendet werden.	-	V2.0
4	Dynamisches Laden & Verwenden von Spracherweiterungen	Neben den Policies, Aktivitäten und Datentypen (s. Glossar) der Corepakete kann jeder Entwickler die für ihn notwendigen Sprachelemente selber definieren und hierdurch die Mächtigkeit der Sprache erweitern. Die Nutzung dieses Mechanismus zur Spracherweiterung soll möglichst komfortabel sein. Darüber hinaus sollen die einzelnen Erweiterungen einfach in beliebige D°-Übersetzungsumgebungen integriert werden können. Aus diesem Grund ist es nicht sinnvoll die Erweiterungen fest mit dem D°-Compiler zu verweben, da jede Erweiterung eine neue Version, bzw. eine neue Übersetzung des Compilers notwendig machen würde. Stattdessen ist es dem Compiler möglich beliebige Spracherweiterungen zur Startzeit dynamisch zu laden und im Anschluss zu verwenden. Aktuell existieren zwei Orte an denen sich Spracherweiterungen befinden können: 1. Im D°-Home Verzeichnis (`${USER_HOME}/.degree`) 2. Im selben Ordner wie die zu übersetzende D°-Data App	V2.0	V1.0
5	Überführung von im Compiler geladenen Sprachelementen in die generierte Data App	Aufgrund des oben beschriebenen Aufbaus von D° verfügen auch die generierten Data Apps über keine festen Policies, Funktionalitäten oder verwendbare Datentypen. Hier werden ebenfalls die zuvor angesprochenen Spracherweiterungen verwendet. Diese Erweiterungen müssen auch in den generierten Data Apps zur Verfügung stehen, damit die Funktionalität gewährleistet werden kann. In der aktuellen Version des Compilers findet keine intelligente Auswahl der Erweiterungspakete statt, welche in der generierten Data App verfügbar gemacht werden müssen. Stattdessen werden sämtliche Sprachelemente, welche in den Compiler geladen werden auch in der generierten Data App verfügbar gemacht.	V2.0	V1.0
6	Wahrung von Variablensichtbarkeit	Die Grammatik von D° enthält ein Sprachelement mit dem Namen `block`. Der Block stellt im Bezug auf die Sichtbarkeit von Variablen das zentrale Konstrukt in D° dar. Alle Elemente die sich innerhalb eines Blockes befinden sind im selben Sichtbarkeitsraum. Darüber hinaus ist es möglich auf Elemente des übergeordneten Blocks zuzugreifen. Ein Zugriff von außen auf ein Element, welches innerhalb eines Blockes definiert wurde, ist nicht möglich.	-	V1.0

<table>
<tr>
<td>7</td>
<td>Syntaxüberprüfungen</td>
<td>Da die Grammatik von D° weder Datentypen noch Aktivitäten beschreibt ist es notwendig, dass an mehreren Stellen "Freitext", in Form von gültigen Identifiern, vom Parser akzeptiert wird. Somit ist eine spätere Überprüfung des D°-Codes zur Compilierzeit notwendig.

Ob ein D°-Code valide ist kann erst vom Compiler bestimmt werden, sobald die verfügbaren Spracherweiterungen geladen wurden, da erst zu diesem Zeitpunkt die gültigen Werte für die "Freitext"-Stellen bekannt sind. Darüber hinaus sind "klassische" Syntaxüberprüfungen, die nicht von den verfügbaren Spracherweiterungen abhängig, durchzuführen, um dem Entwickler eventuelle Fehler aufzuzeigen und die Generierung von defekten Data Apps in der Zielsprache zu verhindern.

Die aktuelle Version des D°-Compilers unterstützt die folgenden Syntaxüberprüfungen:
<table>
<tr><th></th><th>Überprüfung</th><th>Notiz</th><th>Änderung in Version (en)</th><th>Hinzugefügt in Version</th></tr>
<tr><td>1</td><td>Existenz von referenzierten Variablen</td><td>Wird eine Variable als Eingabeparameter einer Aktivität verwendet ist die Existenz der Variablen zwingend notwendig (dies gilt nicht für Ausgabeparameter). Gleiches gilt für den Zugriff auf (Teil-) Werte einer Variable.</td><td>-</td><td>V1.0</td></tr>
<tr><td>2</td><td>Existenz von referenzierten Datentypen</td><td>Wird eine Variable von einem bestimmten Datentyp angelegt muss sichergestellt werden, dass der entsprechende Datentyp als Teil einer Spracherweiterung geladen wurde.</td><td>-</td><td>V1.0</td></tr>
<tr><td>3</td><td>Existenz von aufgerufenen Aktivitäten</td><td>Wird eine Aktivität aufgerufen ist es notwendig, dass die Aktivität Teil einer der geladenen Spracherweiterungen ist</td><td>-</td><td>V1.0</td></tr>
<tr><td>4</td><td>Existenz von zugegriffenen Feldern einer Variable</td><td>Wird (in einem beliebigem Kontext) auf ein Feld einer Variable zugegriffen (ebenfalls verschachtelt möglich), ist es notwendig, dass die Zugriffe bzgl. Existent und Typkompatibilität überprüft werden.</td><td>-</td><td>V1.0</td></tr>
<tr><td>5</td><td>Typkompatibilität von Eingabeparametern für Aktivitäten</td><td>Werden Variablen als Eingabeparameter an eine Aktivität übergeben ist es notwendig, dass die Datentypen kompatibel sind.

Die aktuelle Version des D°-Compilers orientiert sich an der Reihenfolge der Eingabeparameter in der Aktivitätsdefinition und unterstützt nur exakte Typkompatibilität (keine Beachtung von Sub- und Supertypen).</td><td>-</td><td>V1.0</td></tr>
<tr><td>6</td><td>Typkompatibilität von Ausgabeparametern für Aktivitäten</td><td>Soll eine bereits existierende Variable zur Abspeicherung eines Rückgabewerts einer Aktivität verwendet werden muss die Typkompatibilität sichergestellt werden.

Die aktuelle Version des D°-Compilers orientiert sich an der Reihenfolge der Ausgabeparameter in der Aktivitätsdefinition und unterstützt nur exakte Typkompatibilität (keine Beachtung von Sub- und Supertypen).</td><td>-</td><td>V1.0</td></tr>
<tr><td>7</td><td>Typkompatibilität bei der Zuweisung von Werten zu Variablen</td><td>Soll einer Variablen ein neuer Wert zugewiesen werden ist es notwendig, dass die Typkompatibilität sichergestellt wird.

Die aktuelle Version des D°-Compilers unterstützt nur exakte Typkompatibilität (keine Beachtung von Sub- und Supertypen).</td><td>-</td><td>V1.0</td></tr>
<tr><td>8</td><td>Verwendung gültiger Sprachkonstrukte</td><td>Wie bereits zuvor erwähnt, setzt D° an diversen Stellen auf Identifier, welche je nach Kontext eine unterschiedliche Bedeutung haben können. Aus diesem Grund muss der Compiler sicherstellen, das die verwendeten Sprachkonstrukte in ihrem jeweiligem Kontext gültig sind.

Beispielsweise darf der Aufruf einer Aktivität nicht zur Speicherung eines Rückgabewertes eines anderen Aktivitätsaufrufs verwendet werden.</td><td>-</td><td>V1.0</td></tr>
<tr><td>9</td><td>Validierung von Eingabeparametern</td><td>Wenn eine Data App Eingabeparameter erfordert wird (zur Laufzeit) überprüft, ob diese vorhanden und richtig getypt sind.</td><td>-</td><td>V2.0</td></tr>
</table>
</td>
<td>-</td>
<td>V1.0</td>
</tr>
</table>

Command Line Interface

Die Verwendung des D°-Compilers erfolgt über die Kommandozeile. Zur Steuerung des Verhaltens des D°-Compilers stehen diverse Kommandozeilenparameter zur Verfügung welche nachfolgend erläutert werden. Das Command Line Interface ist konform zum POSIX-Standard inklusive der GNU Erweiterung für `getopt_long`,

Kurze Option	Lange Option	Beschreibung	Änderung in Version(en)	Hinzugefügt in Version
-v	--version	Gibt die aktuell verwendete Version des D°-Compilers zurück.	-	V1.0
-h	--help	Zeigt alle verfügbaren Optionen und deren Verwendung an.	-	V1.0

Neben den Optionen akzeptiert der Compiler exakt einen optionslosen Parameter. Dieser Parameter gibt das Verzeichnis an, in dem sich das zu kompilierende D°-Programm befindet.

Der D°-Compiler ist eine Java-Applikation und kann daher über den Befehl `java -jar <<DEGREE_COMPILER>>.jar` ausgeführt werden. Da ein Teil der Funktionalität aber durch einen sog. Java Agent, welcher in der Applikation des Compilers enthalten ist, bereitgestellt wird muss dieser zusätzlich an die ausführende JVM übergeben werden. Hierdurch ergibt sich folgender Befehl zum Aufruf des D°-Compilers:

```
java -javaagent:<<ABSOLUTER_PFAD>>/<<DEGREE_COMPILER>>.jar -jar <<DEGREE_COMPILER>>.jar [Options] [Code Location]
```

Um den Aufruf des D°-Compilers zu vereinfachen werden Skripte für die Kommandozeilen unter Windows (`degreec.bat`) und Linux (`degreec.sh`) mitgeliefert.

Core Sprachmodule

Der D°-Compiler wird gemeinsam mit einem Erweiterungsmodul für Datentypen und einem für Aktivitäten ausgeliefert, damit in der Sprache eine Grundfunktionalität zur Verfügung steht, ohne das zunächst eine eigene Erweiterung implementiert werden muss. Diese Core Sprachmodule werden nachfolgend betrachtet. Alle Sprachelemente, die in diesen Core Modulen definiert werden, können entweder ohne die Angabe eines Namensraums oder mit dem Namensraum `core` verwendet werden.

Datentypen

Dieser Abschnitt betrachtet die Validatoren und Datentypen, welche im entsprechenden Core Sprachmodul definiert werden.

Validatoren

	Name	Parameter	Beschreibung	Änderung in Version (en)	Hinzugefügt in Version
1	VoidValidator	-	Akzeptiert alle Werte.	-	V1.0
2	RegexValidator	`pattern` - Ein Regulärer Ausdruck	Dieser Validator akzeptiert alle Werte, die `pattern` entsprechen. Der Defaultwert von `pattern` akzeptiert alles.	-	V1.0
3	IntegerValidator	-	Ausprägung des RegexValidators. `pattern` wurde so gesetzt das jeder Integer akzeptiert wird. Es existieren keine Maximal- und Minimalwerte.	-	V1.0
4	DecimalValidator	-	Ausprägung des RegexValidators. `pattern` wurde so gesetzt das jede Dezimalzahl akzeptiert wird. Es existieren keine Maximal- und Minimalwerte.	-	V1.0
5	ScriptValidator	`script` - Ein Stück JavaScript-Code	Der ScriptValidator verfügt über ein Stück JavaScript-Code welches ausgeführt wird um Werte auf Gültigkeit zu überprüfen. Die letzte Anweisung im JavaScript-Code muss einen boolschen Wert zurückgeben. Defaultmäßig ist kein Script gesetzt.	-	V1.0
6	MonthValidator	-	Ausprägung des ScriptValidators. `script` wurde so gesetzt, dass nur Integerwerte im Wertebereich [1, 12] akzeptiert werden.	-	V1.0
7	DayValidator	-	Ausprägung des ScriptValidators. `script` wurde so gesetzt, dass nur Integerwerte im Wertebereich [1, 31] akzeptiert werden.	-	V1.0
8	HourValidator	-	Ausprägung des ScriptValidators. `script` wurde so gesetzt, dass nur Integerwerte im Wertebereich [0, 23] akzeptiert werden.	-	V1.0
9	Minutes Validator	-	Ausprägung des ScriptValidators. `script` wurde so gesetzt, dass nur Integerwerte im Wertebereich [0, 59] akzeptiert werden.	-	V1.0
10	Seconds Validator	-	Ausprägung des ScriptValidators. `script` wurde so gesetzt, dass nur Integerwerte im Wertebereich [0, 59] akzeptiert werden.	-	V1.0
11	DateValidator	-	Ausprägung des ScriptValidators. `script` wurde so gesetzt, dass für eine Variable vom Typ Date (s.u.) geprüft wird, ob das Datum gültig ist (bpsw. Prüfung für Schaltjahre).	-	V1.0

Datentypen

	Name	Supertyp (en)	Attribut(e)	Defaultwert	Validator (en)	Beschreibung	Änderungen in Version (en)	Hinzugefügt in Version
1	Text	-	-	*Leerer Text*	-	Elementarer primitiver Datentyp der von vielen anderen Datentypen als Supertyp verwendet wird	-	V1.0
2	Composite	-	-	-	VoidValidator	Leerer zusammengesetzter Datentyp. Wird von vielen weiteren zusammengesetzten Datentypen als Supertyp verwendet	-	V1.0
3	Integer	Text	-	0	IntegerValidator	Datentyp für positive und negative Ganzzahlen.	-	V1.0
4	Decimal	Text	-	0.0	DecimalValidator	Datentyp für positive und negative Dezimalzahlen.	-	V1.0
5	Month	Integer	-	1	MonthValidator	Darstellung des Monats als positiven, numerischen Ganzzahlwert.	-	V1.0
6	Day	Integer	-	1	DayValidator	Darstellung des Tags (im Monat, nicht Wochentag) als positiven, numerischen Ganzzahlwert.	-	V1.0
7	Hour	Integer	-	0	HourValidator	Darstellung der Stunde des Tages als positiven, numerischen Ganzzahlwert.	-	V1.0
8	Minute	Integer	-	0	MinuteValidator	Darstellung der Minuten in einer Stunde als positiven, numerischen Ganzzahlwert.	-	V1.0
9	Second	Integer	-	0	SecondValidator	Darstellung der Sekunden in einer Minute als positiven, numerischen Ganzzahlwert.	-	V1.0
10	Date	Composite	day - Day month - Month year - Integer - Defaultwert 1980	01. 01. 1980	DateValidator	Numerische Darstellung eines validen Datums.	-	V1.0

11	Time	Composite	hour - Hour minute - Minute second - Second	00: 00: 00	-	Numerische Darstellung einer Uhrzeit.	-	V1.0
12	DateTime	Composite	date - Date time - Time	00:00:00 01.01.1980	-	Darstellung eines Datums als Kombination aus Datum und Uhrzeit.	-	V1.0
13	Address	Composite	street - Text city - Text	*Leerer Text* *Leerer Text*	-	Eine Adresse, bestehend aus Angaben zur Straße und Stadt.	-	V1.0
14	Person	Composite	forname - Text surname - Text birthday - Date address - Address	*Leerer Text* *Leerer Text* 01.01.1980 *Leerer Text*	-	Repräsentation einer Person mit Vor- und Nachnamen, einem Geburtstag und einer Adresse.	-	V1.0
15	Student	Person	studentNumber - Text course - Text matriculationDate - Date	*Leerer Text* *Leerer Text* 01.01.1980	-	Student als Erweiterung einer Person. Verfügt über eine Matrikelnummer, einen belegten Kurs und ein Immatrikulationsdatum.	-	V1.0

Aktivitäten

	Name	Eingabeparameter	Ausgabeparameter	Beschreibung	Änderungen in Version(en)	Hinzugefügt in Version
1	PrintToConsoleActivity	`text` - Text	-	Schreibt den Inhalt von `text` in die Standardausgabe der ausführenden Data App.	-	V1.0
2	WriteFileActivity	`filePath` - Text `content` - Text	-	Schreibt den kompletten Inhalt von `content` in die Datei `filePath`.	-	V1.0
3	ReadFileActivity	`filePath` - Text	content - Text	Liest den gesamten Inhalt der Datei `filePath` und schreibt ihn in `content`.	-	V1.0

Policies

	Name	Typ	Eingabeparameter	Abhängigkeiten	Beschreibung	Änderungen in Version(en)	Hinzugefügt in Version
1	UseNotBeforeTimestamp	Constraint	timestamp - DateTime	-	Erlaubt die Ausführung einer Aktivität/Zugriff auf Daten nur nach einem gegebenem Zeitpunkt	-	V2.0
2	UseNotAfterTimestamp	Constraint	timestamp - DateTime	-	Erlaubt die Ausführung einer Aktivität/Zugriff auf Daten nur vor einem gegebenem Zeitpunkt	-	V2.0
3	AllowedTimeInterval	Policy	-	UseNotBeforeTimestamp UseNotAfterTimestamp	Erlaubt die Ausführung einer Aktivität/Zugriff auf Daten nur in einem definierten Zeitintervall	-	V2.0

Spracherweiterungen

Policies, Datentypen und Aktivitäten werden in D° über spezielle Spracherweiterungen zur Verfügung gestellt. Jeder Entwickler der D° nutzt kann für seinen eigenen Bedarf passende Erweiterungen definieren. In diesem Abschnitt werden einige Hilfestellungen gegeben, die entsprechende Personen bei der Entwicklung ihrer Erweiterungen unterstützten soll.

Datentypen

Zur Definition neuer Datentypen reicht die textuelle Definition aus und es muss keine zusätzliche Implementation geliefert werden.

Im Kern unterscheidet das Typsystem zwischen primitiven Typen und zusammengesetzten Typen. Für beide dieser Kategorien ist im Core-Sprachmodul ein Datentyp definiert (`Text` & `Composite`), welcher als Supertyp für alle weiteren Datentypen verwendet werden kann. Dabei wurden diese beiden Datentypen über die Verwendung von Skriptausdrücken definiert, welche von der sonstigen Syntax der Sprachdefinitionen abweicht. Bei Bedarf können Skriptausdrücke verwendet werden um neue "Wurzel-Datentypen" zu definieren, die keinen Supertyp haben.

Generell wird von der Verwendung von Skriptausdrücken abgeraten, da in der Skriptengine nur bestimmte Klassen aus dem Typsystem zur Verfügung stehen. Darüber hinaus sind die verfügbaren Schnittstellen und die Funktionalität nicht dokumentiert, da die Verwendung von Skriptausdrücken in zukünftigen Versionen ggf. eingeschränkt oder sogar vollständig entfernt wird. Der nachfolgend Codeblock zeigt die beiden verwendeten Skriptausdrücke, welche verwendet wurden um die Datentypen Text und Composite zu definieren.

Definition von Datentypen - Beispiel 1

```
Text := {# "TypeBuilder.define(\"Text\").initialValue(\"\").register()" #}
Composite := {# "TypeTaxonomy.getInstance().register(new CompositeType(QualifiedIdentifier.create(\"core\", \"
Composite\"), ValidatorBuilder.create(\"Void\").build()))" #}
```

Auf Basis der beiden Grundtypen Text und Composite können nun beliebige andere Datentypen definiert werden. Dabei sind einige Punkte zu beachten:

- Ein primitiver Datentyp kann keinen zusammengesetzten Datentypen als Supertypen haben
- Ein zusammengesetzter Datentyp kann keinen primitiven Datentypen als Supertypen haben
- Alle Eigenschaften des/der Supertypen sind ebenfalls in den Subtypen vorhanden (Validatoren, Attribute, Defaultwerte)

Im folgenden Codeblock ist die Definition des Datentyps `Integer` zu sehen. Der Supertyp von `Integer` ist `Text`, zur Validierung wird der `IntegerValidator` verwendet und der Defaultwert lautet 0. Der $-Operator wird verwendet um bereits definierte und somit existierende Elemente zu referenzieren, wogegen der @-Operator verwendet wird um Aktionen (sog. Definition Functions) im Typsystem einzuleiten.

Definition von Datentypen - Beispiel 2

```
Integer := $Text(
    @validate[$IntegerValidator],
    @initialValue["0"]
)
```

Bei der Definition von zusammengesetzten Datentypen ist die Syntax identisch, aber es stehen teilweise andere Definition Functions zur Verfügung. Durch die Definition Function `@attribute` wird dem aktuell definierten Datentyp ein Attribut hinzugefügt. Dabei müssen zwingend Name und Datentyp angegeben werden. Als dritter Parameter kann ein Defaultwert angegeben werden, falls nicht der Standardwert des referenzierten Datentyps verwendet werden soll. Es ist zu beachten das zusammengesetzte Datentypen selber über keinen Defaultwert verfügen. Stattdessen bildet der der Defaultwert aus den Standardwerten aller Attribute.

Definition von Datentypen - Beispiel 3

```
Date := $Composite(
    @validate[$DateValidator],
    @attribute["day", $Day],
    @attribute["month", $Month],
    @attribute["year", $Integer, "1980"]
)
```

Es gibt zwei Möglichkeiten um innerhalb von D° neue Validatoren zu erzeugen. Zum einen ist es möglich aus einem generischen (parametrisierbaren) Validator, wie bspw. dem `RegexValidator`, einen spezifischen Validator zu erzeugen indem die Parameter mit passenden Werten belegt werden. Darüber hinaus ist es auch möglich vollständig neue (generische) Validatoren zu implementieren und als Sprachmodul in den Compiler zu laden.

Syntaktisch ist Ableitung eines neuen Validators von einem bestehendem identisch zur Definition neuer Datentypen, aber es steht nur eine Definition Function zur Verfügung. Die Definition Function `@set` wird verwendet um vorhandene Parameter des Validators für die konkrete Ausprägung zu setzen.

Definition von Datentypen - Beispiel 4

```
IntegerValidator := $RegexValidator(
    @set["pattern", "^(\\+|-)?(0|[1-9][0-9]*)$"]
)
```

Wenn ein vollständig neuer Validator implementiert werden soll müssen einige Punkte beachtet werden:

- Bei der Implementation handelt es sich um eine Java-Klasse
- Die Klasse ist mit der `@ValidatorKey`-Annotation markiert
    - Die Annotation erwartet einen String Parameter, welcher den Namen des Validators bestimmt, welcher vom Typsystem zum auffinden verwendet wird.
- Die Klasse muss das Interface `Validator` implementieren
    - Die im Interface definierte `accept()`-Methode ist der Einstiegspunkt für die Validierung eines Wertes
    - Die im Interface definierte `save()`-Methode verfügt über eine Standardimplementierung und muss nur überschrieben werden, wenn der Validator Parameter verwendet
    - Neben dem Default-Constructor (parameterlos) muss ein ein Constructor implementiert werden, der als einzigen Parameter ein `Struct` entgegen nimmt.
        - Das Struct ist der Persistenzcontainer des Typsystems
        - von der `save()`-Methode werden ebenfalls Stucts verwendet

- Das Struckt ist ein key-value-Store

Der nachfolgende Codeblock zeigt die (gekürzte) Implementation des RegexValidators.

Definition von Datentypen - Beispiel 5

```
...


@ValidatorKey("Regex")
public class RegexValidator implements Validator {

    RegexValidator() {
        pattern = ".*";
    }

    public RegexValidator(String regex) {
        this.pattern = regex;
    }

    public RegexValidator(Struct saved) {
        Validate.isTrue(saved.getStringLiteral(NAME_KEY).equals(getKey().toString()));
        this.pattern = saved.getStringLiteral(PATTERN_KEY);
    }

    @Override
    public void accept(ValidationContext context) {
        context.getStyle().validate(ValidationContext.Style.PRIMTIVE);
        if(PatternCache.mismatches(pattern, context.getValue()))
            throw new IllegalArgumentException(String.format("Value does not match pattern! '%s', '%s'",
pattern, context.getValue()));
    }

    @Override
    public Struct save() {
        Struct save = Validator.super.save();
        save.addLiteral(PATTERN_KEY, pattern);
        return save;
    }

    @Override
    public String toString() {
        return new ToStringBuilder(this, new TypesToStringStyle())
                .append("pattern", pattern)
                .build();
    }

    public static Validator create(String pattern) {
        return new RegexValidator(pattern);
    }

    public static Validator create(String pattern, boolean allowNull, boolean allowEmpty) {
        return new OptionalValidator(new RegexValidator(pattern), allowNull, allowEmpty);
    }

        ...
}
```

Aktivitäten

Bei der Definition von neuen Aktivitäten ist sowohl die textuelle Definition als auch eine Implementation der eigentlichen Funktionalität notwendig. Syntaktisch ist die Definition von Aktivitäten analog zur Definition von Datentypen und Validatoren. Es stehen die Definition Functions `@input` und `@output` zur Verfügung, um Ein- und Ausgabeparameter durch Angabe des Namens und des Datentyps zu definieren. Insbesondere ist zu beachten, dass eine Aktivität beliebig viele Ein- und Ausgabeparameter besitzen kann.

Im nachfolgenden Codeblock befindet sich ein Beispiel, wie die `ReadFileActivity` im Core Sprachmodul definiert wurde.

Definition von Aktivitäten - Beispiel 1

```
ReadFileActivity := $Activity (
    @input["filePath", $Text],
    @output["content", $Text]
)
```

Neben der Definition der Aktivität muss ebenfalls die Implementierung der Funktionalität vorhanden sein. Hier sind wenig Rahmenbedingungen gegeben:

- Bei der Implementation handelt es sich im eine Java-Klasse
- Die Klasse ist mit der `@ActivityAnnotation`-Annotation markiert
    - Die Annotation erwartet einen String Parameter, welcher dem (voll qualifiziertem) Namen aus der Definition entsprechen muss
- Die Klasse muss das Interface `ActivityAPI` implementieren
    - Alternativ kann von der abstrakten Klasse `ActivityInstance` geerbt werden
    - Die im Interface definierte `run()`-Methode ist der Einstiegspunkt für die Ausführung der Aktivität
- Die Aktivität darf beliebige Abhängigkeiten zu externen Bibliotheken besitzen, muss diese aber im fertigen Jar-File mitliefern

Der nachfolgende Codeblock zeigt beispielhaft die (gekürzte) Implementation der `ReadFileActivity` in der Programmiersprache Kotlin.

Definition von Aktivitäten - Beispiel 2

```
package de.fhg.isst.degree.activities.core.io

import de.fhg.isst.degree.activities.ActivityInstance
...

@ActivityAnnotation("core.ReadFileActivity")
class ReadFileActivity : ActivityInstance() {

    override fun run(input: InputScope): OutputScope {
        ...
    }
}
```

Policies

Das Policy-Subsystem von D° unterscheidet zwischen zwei verschiedenen Konstrukten: Constraints und Policies.

Bei Constraints handelt es sich um (feingranulare) Regeln, welche auch technisch umgesetzt werden können. Aus diesem Grund besteht eine Constraint sowohl aus einer textuellen Beschreibung als auch aus einer Implementation, welche die Logik zur Durchsetzung/Überprüfung enthält. Dabei besteht die Prüf und Durchsetzungslogik einer Constraint aus drei Teilen: Pre- & Postconditions, sowie einen dritten Teil der benötigt wird unmittelbar bevor eine Aktion ausgeführt werden soll, die mit der Constraint verbunden ist.

Der aktuelle Entwicklungsstand erlaubt nur die Verwendung von Pre- & Postconditions.

Bei Policies handelt es sich im Kontext von D° um Container, welche eine Menge von Policies und Constraints kapseln und selber über keine Logik verfügen. Aus diesem Grund besteht eine Policy nur aus einer textuellen Definition, welche die enthaltenen Elemente bestimmt.

Syntaktisch erfolgt die (textuelle) Definition von Policy-Konstrukten analog zu Datentypen und Aktivitäten. Dabei stehen für die Definition von Policy-Konstrukten andere Definition Functions zur Verfügung, als bei den anderen Sprachkonstrukten.

Bei der Definition von Constraints steht die Definition Function `@attribute` zur Verfügung, welche verwendet wird, um die Parameter des Constraints zu definieren. Dabei wird der Name des Parameters bestimmt und der verwendete Datentyp referenziert.

Werden Policies definiert wird die Definition Function `@dependency` verwendet, um zu definieren, welche Policy-Konstrukte in der jeweiligen Policy eingebettet sind.

Das nachfolgende Codebeispiel zeigt die beispielhafte Definition von zwei Constraints.

Definition von Policies - Beispiel 1

```
UseNotBeforeTimeStamp := $Constraint(
    @attribute["timestamp", $DateTime]
)

UseNotAfterTimeStamp := $Constraint(
    @attribute["timestamp", $DateTime]
)
```

Im folgenden Codeblock ist die Definition einer Policy sichtbar, welche die zwei zuvor definierten Constraints verwenden.

Definition von Policies - Beispiel 2

```
AllowedTimeInterval := $Policy(
    @dependency["useNotBefore", $UseNotBeforeTimeStamp],
    @dependency["useNotAfter", $UseNotAfterTimeStamp]
)
```

Für Constraints ist es notwendig, dass neben der Definition auch eine Implementation vorliegt. Bei der Implementation sind die folgenden Punkte zu beachten:

- Bei der Implementation handelt es sich im eine Java-Klasse
- Die Klasse ist mit der `@PolicyAnnotation`-Annotation markiert
  - Die Annotation erwartet einen String Parameter, welcher dem (voll qualifiziertem) Namen aus der Definition entsprechen muss
- Die Klasse muss das Interface `EmbeddedPolicyAPI` implementieren
  - In späteren Versionen von D° werden unter Umständen weitere Arten von Policies definiert und hinzugefügt, aber im aktuellen Stand gibt es nur die EmbeddedPolicyAPI
  - Die im Interface definierten Methoden `acceptPrecondition()`, `acceptSecurityManagerIntervention()` und `acceptPostcondition()` sind die Einstiegspunkte für das Policyenforcement
- Das Constraint darf beliebige Abhängigkeiten zu externen Bibliotheken besitzen, muss diese aber im fertigen Jar-File mitliefern

Das nächste Beispiel zeigt die Implementation der Constraint `UseNotBeforeTimeStamp` in der Programmiersprache Kotlin.

Definition von Policies - Beispiel 3

```
package de.fhg.isst.degree.policies.core.date

import ...

@PolicyAnnotation("UseNotBeforeTimeStamp")
class UseNotBeforeConstraint : EmbeddedPolicyApi {

    override fun acceptPrecondition(input : PolicyInputScope): Boolean {
        val timestamp = input.get("timestamp")!!
        val date : CompositeInstance = (timestamp as CompositeInstance).get("date")
        val time : CompositeInstance = (timestamp as CompositeInstance).get("time")
        val timestampDate = GregorianCalendar(
                Integer.valueOf(date.read("year")),
                Integer.valueOf(date.read("month")),
                Integer.valueOf(date.read("day")),
                Integer.valueOf(time.read("hour")),
                Integer.valueOf(time.read("minute")),
                Integer.valueOf(time.read("second"))
        )
        if (timestampDate.after(GregorianCalendar())) {
            return false
        }

        return true
    }

    override fun acceptSecurityManagerIntervention(input: PolicyInputScope): Boolean {
        return true
    }

    override fun acceptPostcondition(input: PolicyInputScope): Boolean {
        return true
    }

}
```

Differenzierung zwischen Definitionen und Instanzen

In der ersten Version des D° Compilers wurden die vom Nutzer definierten Sprachelemente (Datentypen, Aktivitäten und Policies) direkt in den erzeugten Data Apps verwendet. Hierdurch ergibt sich die ungünstige Situation, dass bspw. Aktivitäten, welche mit verschiedenen Policies in einer Data App verwendet werden sollen, vollständig dupliziert werden müssen.

Aus diesem Grund wurde in der zweiten Version des D° Compilers eine Trennung zwischen den Definitionen und den später verwendeten Instanzen eingeführt. Das System wird aktuell in zwei der Subsysteme von D° verwendet:

1. Aktivitäten müssen instanziiert werden, um sie mit Policy-Konstrukten zu verbinden. Hierdurch kann ein und dieselbe Aktivität mit verschiedenen Policies verbunden werden und in Form von verschiedenen Instanzen mehrfach (in unterschiedlichen Zusammenhängen) verwendet werden.
2. Auch Elemente aus dem Policy-Subsystem müssen instanziiert werden, bevor sie in Data Apps verwendet werden können.
    a. Constraints müssen instanziiert werden, um die verfügbaren Parameter mit Werten zu belegen.
    b. Policies müssen instanziiert werden, um die eingebetteten Policy-Konstrukte mit Instanzen zu belegen.

Das nachfolgende Codebeispiel zeigt die Definition von 2 Constraints und einer Policy, welche beide Constraints enthält. Anschließend werden sowohl die Constraints als auch die Policy instanziiert. Diese neue Instanz einer Policy wird mit einer neu erzeugten Instanz einer Aktivität verbunden.

Differenzierung zwischen Definitionen und Instanzen - Beispiel 1

```
UseNotBeforeTimeStamp := $Constraint(
    @attribute["timestamp", $DateTime]
)

UseNotAfterTimeStamp := $Constraint(
    @attribute["timestamp", $DateTime]
)

AllowedTimeInterval := $Policy(
    @dependency["useNotBefore", $UseNotBeforeTimeStamp],
    @dependency["useNotAfter", $UseNotAfterTimeStamp]
)

UseNotAfterTimeStamp2020  = $UseNotAfterTimeStamp(
    @set["timestamp", "/**VALUE**/"]
)

UseNotBeforeTimeStamp2010  = $UseNotBeforeTimeStamp(
    @set["timestamp", "/**VALUE**/"]
)

AllowedTimeInterval2010to2020 = $AllowedTimeInterval(
    @set["useNotBefore", $UseNotBeforeTimeStamp2010],
    @set["useNotAfter", $UseNotAfterTimeStamp2020]
)

ConstrainedPrintToConsoleActivity = $PrintToConsoleActivity(
    @controlledBy["timeConstraint", $AllowedTimeInterval2010to2020]
)
```

In der aktuellen Version des D°-Compilers befinden sich sämtliche Instanziierungen (auch die für Aktivitäten) in den Policy-Sprachmodulen (.degreePolicy Dateien). Eine spätere Version wird an dieser Stelle eine Umstrukturierung vornehmen.

Subsystem für Kontextinformationen

Häufig kommt es vor, dass Policies Kontextabhängig sind und zusätzliche Informationen für das Enfocement benötigen. Beispielsweise wenn eine Aktivität nur drei mal ausgeführt werden darf, ist es notwendig irgendwo die Information zu hinterlegen, wie oft die Policy bereits ausgeführt wurde. Das Policy-Subsystem von D° würde es erlauben diese Informationen in der jeweiligen Policy bzw. Constraint direkt zu speichern. Dies funktioniert aber nur für Informationen, welche von einer einzelnen Constraint verwendet werden.

Besteht beispielsweise eine Datenverbindung über die global insgesamt nur 100MB Daten transferiert werden dürfen, so ist es nicht zielführend, wenn jede Aktivität vermerkt wie viele Daten sie versendet hat, da eine globale Überprüfung der Gesamtmenge fehlt. Darüber hinaus existieren Kontextinformationen, die von vielen Constraints (lesend) verwendet werden und es wenig Sinn macht diese Redundant in jedem betroffenen Element abzuspeichern. Hierzu zählen beispielsweise Informationen über den aktuell ausführenden/anfragenden Benutzer.

Um diese Szenarien abzubilden wurde in der zweiten Version des D°-Compilers eine ContextInformation-Subsystem eingefügt. Das Subsystem erlaubt die Baumartige Organisation von Kontextinformationen, welche alle über eindeutige Namen erreichbar sind.

Die im Subsystem enthaltenen Werte werden persistiert und stehen somit auch über mehrere Ausführungen einer Data App hinweg zur Verfügung. Jede Änderung an den Werten wird direkt persistiert und darüber hinaus in einem persistenten Protokoll erfasst.

Die aktuelle Version dieses Subsystems legt keinen besonderen Fokus auf die Sicherheit der erzeugten Dateien, welche die Daten enthalten. So ist es beispielsweise möglich die entsprechenden Dateien zu manipulieren oder zu löschen, wodurch das Verhalten der Data App beeinflusst werden kann. In einer späteren Version des D°-Compilers wird das System diesbezüglich überarbeitet.

Aktuell stehen die folgenden Arten von Kontextinformationen zur Verfügung:

ReadOnly, ReadWrite, Switch, Counter, DecrementCounter, IncrementCounter

Neben dem eigentlichen Wert verfügt jedes Objekt für Kontextinformationen über einen Namen, der zur Identifizierung verwendet wird.

Die einzelnen Werte sind dabei in sog. `ContextModule` organisiert. Dabei soll ein Modul logisch/semantisch zusammengehörige Werte zusammenfassen und gemeinsam verfügbar machen. Jedes Modul verfügt über einen Namen und enthält eine beliebige Menge weiterer Module und Items für Kontextinformationen.

Um einen Wert, der im Subsystem abgelegt ist, wird eine Anfrage an das oberste `ContextModule` übergeben und anschließend aufgelöst. Dabei haben Anfragen die Form 'MODULE_NAME(.MODULE_NAME)+.ITEM_NAME'.

Das folgende Codebeispiel zeigt eine Beispielanfrage an das ContextInformation-Subsystem in der Programmiersprache Java.

Nutzung des ContextInformation-Subsystems - Beispiel 1

```
ExecutionContext.getInstance().resolve("UserInformation.username")
```

Bei der Implementation eigener Module gibt es wenig zu beachten:

- Falls das neue Modul auf der höchsten Ebene angesiedelt ist (- Im Baum über keine Eltern verfügt -), muss die Implementation mit der `@RootContextModule`-Annotation versehen werden, um zur Laufzeit auffindbar zu sein
    - Ist das Modul nur in einem anderen Modul platziert, ist dies nicht notwendig
- Es muss die Klasse `ContextModule` erweitert werden
    - Sie enthält Funktionalitäten für Logging, Persistenz und Verwaltung der Module
- Es muss ein (Java) Default-Constructor angelegt werden, in welchem der Constructor der Superklasse `ContextModule` aufgerufen wird. An dieser Stelle muss der Name des neuen Moduls als einziger Parameter übergeben werden
- Die Methode `crateDefaultContext()` muss implementiert werden, um die Standardwerte für die geplanten Kontextinformationen zu setzten.

Nutzung des ContextInformation-Subsystems - Beispiel 2

```
package de.fhg.isst.oe270.degree.runtime.java.context.core;

import ...

@RootContextModule
public class OsUserInformationContextModule extends ContextModule {

    public OsUserInformationContextModule() {
        super("OsUserInformation");
    }

    @Override
    public HashMap<String, ContextEntity> createDefaultContext() {
        HashMap<String, ContextEntity> defaultConfiguration = new HashMap<>();

        defaultConfiguration.put("username", new ReadOnlyEntity("username", System.getProperty("user.name")));
        defaultConfiguration.put("userroles", new ReadOnlyEntity("userroles", NO_VALUE));

        return defaultConfiguration;
    }

}
```

Das Subsystem für Kontextinformationen kann nicht nur innerhalb von Policy-Konstrukten (bspw. Constraints) verwendet werden, sondern auch an anderen relevanten Stellen. Beispielsweise kann innerhalb von Aktivitäten auch auf Kontextinformationen zugegriffen werden.

Name-Mapping

Es gibt Szenarien, in denen mehrere Quellen für Kontextinformationen zur Verfügung stehen, unterschiedliche Informationen enthalten und nur eine Quelle die gewünschten (richtigen) Daten enthält. Beispielsweise kann die Frage nach dem ausführenden/eine Anfrage auslösenden Nutzers je nach Art der Data App unterschiedlich. Bei einer Data App, die eine REST-Schnittstelle zur Verfügung stellt ist die Benutzerinformation ggf. in einem JSON-Web-Token (JWT) mitgeliefert, wogegen eine Data App, die nur auf der Kommandozeile ausgeführt wird, Informationen aus dem Betriebssystem benötigt, um den aktuell ausführenden Nutzer zu ermitteln.

In solchen Situationen soll vermieden werden, dass bestehende Data Apps und Sprachkonstrukte angepasst werden müssen, um zu den aktuellen Umständen zu passen. Aus diesem Grund verfügt das ContextInformation-Subsystem über einen Mechanismus für das Abbilden auf von Namen auf Aliase. Zu jedem Zeitpunkt ist jede Kontextinformation unter ihrem Namen erreichbar. Darüber hinaus können Module mit einem Alias versehen werden, welcher erlaubt das Modul unter einem anderen Namen zu erreichen.

Beispielsweise existieren zwei verschiedene Module, welche Informationen über den aktuellen Benutzer enthalten:

- Das `OsUserInformationContextModule` enthält Daten auf Basis von Informationen des Betriebssystems
- Das `JWTUserInformationContextModule` enthält Elemente mit den selben Namen wie das obige Modul (- eine eindeutige Identifizierung ist über die vollqualifizierte Angabe des Namens, inkl. Modulnamen möglich -), bezieht die Informationen bei jedem Request aus dem JWT, welches an die REST-Schnittstelle übergeben wird.

Um eine einheitliche Nutzung zu erlauben, wird je nach Art der Data App ein virtuelles Modul `UserInformation` verfügbar gemacht und erlaubt es die (vom Kontext abhängigen) korrekten Informationen abzufragen. Aktuell ist es für Anwender, die neue Kontextmodule entwickeln nicht möglich eigene Mappings anzulegen. Die Mappings werden auf Basis des Data App Typs erzeugt. Diese Funktion wird in einer späteren Version des D°-Compilers dahingehend erweitert.

Verfügbare Kontextinformationen

	Modul	Mapping	Data App Type	Kontextinformation	Defaultwert	Entity-Typ	Beschreibung	Änderungen in Version (en)	Hinzugefügt in Version
1	OsUserInformation	UserInformation	-	username	`System.getProperty("user.name")`	ReadOnly	Aktueller Benutzername, welcher in Umgebungsvariable abgelegt ist. Wird zur Startzeit initialisiert und ändert sich danach nicht mehr.	-	V2.0
				userroles	NO_VALUE	ReadOnly	Rollen des aktuellen Benutzers. Wird zur Startzeit initialisiert und ändert sich danach nicht mehr.	-	V2.0
2	JWTUserInformation	UserInformation	RestDataApp	username	NO_VALUE	ReadWrite	Benutzername aus JWT Token. Wird bei jedem Request aktualisiert.	-	V2.0
				userroles	NO_VALUE	ReadWrite	Rollen des aktuellen Benutzers. Wird bei jedem Request aktualisiert.	-	V2.0

Policy Enforcement

Zuvor wurde bereits aufgezeigt, dass Policy-Konstrukte in D° drei verschiedene Stellen haben, an denen die Logik der Konstrukte eingefügt werden kann:

- Precondition: Wird ein Element mit einem Policy-Konstrukt geschützt, hat das jeweilige Konstrukt vor der Verwendung (bspw. Aufruf einer Aktivität) des geschützten Elements die Möglichkeit Code auszuführen. Dies kann bspw. genutzt werden, um Kontextinformationen abzufragen.
- Ereignissbasierte Unterbrechungen: Werden in einer Data App bestimmte Aktionen ausgeführt (bspw. Zugriff auf eine Datei, Netzwerkinteraktion) wird die Ausführung unterbrochen und verknüpfte Policy-Konstrukte können nochmal Logik ausführen. Da die Policy-Konstrukte zu diesem Zeitpunkt Zugriff auf die betroffenen Daten (bspw. die Daten, die in eine Datei geschrieben werden sollen) haben, können an dieser Stelle tiefer gehende Prüfungen durchgeführt werden.
- Postcondition: Wird ein Element mit einem Policy-Konstrukt geschützt, hat das jeweilige Konstrukt nach der Verwendung des geschützten Elements die Möglichkeit Code auszuführen. Dies kann bspw. genutzt werden, um Kontextinformationen zu aktualisieren.

D°-Code wird in komplexe Java-Applikationen übersetzt, welche durch die JVM ausgeführt werden. Innerhalb der JVM existiert der `SecurityManager`. Hierbei handelt es sich um ein Konzept das es erlaubt bestimmte Aktionen in Java-Applikationen zu verbieten. Dabei wird eine simple Notation verwendet um die entsprechenden Regeln zu definieren.

Dieses System wird in D° als Grundlage für die Ereignisbasierten Unterbrechungen verwendet und mit den Policy-Konstrukten von D° verknüpft. In der zweiten Version des D°-Compilers sind hierfür einige Vorbereitungen getroffen worden, aber das System ist aktuell noch nicht nutzbar. Dies wird sich in einer der nächsten Versionen des D°-Compilers ändern. Aktuell ist es nur möglich Pre- & Postconditions ausführen zu lassen.

Ausführung von Data Apps

Die Ausgabe des D°-Compilers ist eine ausführbare Java Applikation (JAR-File). Sofern keine Sprachmodule mit Aktivitäten verwendet wurden kann die Applikation ganz normal über `java -jar <<JAR_NAME>>.jar` erfolgen.

Werden zusätzlich auch Aktivitäten aus Sprachmodulen benötigt müssen diese in den Klassenpfad der ausführenden JRE eingefügt werden, da die entsprechenden Implementierungen in der aktuellen Version nicht in die fertige Java Applikation eingefügt werden. Am einfachsten ist eine Ausführung unter diesen Umständen zu erreichen, wenn sich alle Jar-Dateien im selben Ordner befinden und das starten über den folgenden Befehl erfolgt: `java -cp *.jar;. <<DATA_APP_NAMESPACE>>.<<DATA_APP_NAME>>`. Dabei entsprechen `DATA_APP_NAMESPACE` und `DATA_APP_NAME` den Angaben aus dem Konfigurationsteil der .degree Datei.

Road Map

Abschließend wird in diesem Abschnitt ein Überblick über die geplante, langfristige Weiterentwicklungen des D°-Compilers gegeben. Die Auflistung ist dabei weder final noch erhebt sie Anspruch auf Vollständigkeit.

Konformität zu zukünftigen Versionen der Spezifikation

Damit zu jeder Version der D°-Spezifikation eine nutzbare Implementation zur verfügung steht ist es notwendig, dass zu jeder neuen Version der Spezifikation der Compiler weiterentwickelt wird, um konform zur neuen Spezifikation zu sein und neue Features zu unterstützen. Teile der unten aufgeführten Punkte ergeben aus geplanten Funktionalitäten, die in der Zukunft in die Spezifikation eingepflegt werden und werden an dieser Stelle gelistet, um einen besseren Überblick über die allgemeine zukünftige Entwicklung von D° zu geben.

Policy Definition und Enforcement

Eines der zentralen Features von D° ist die Definition und Durchsetzung von Policies direkt in der entwickelten Data App. Diese Funktionalität hat es nicht mehr in die erste Version der Spezifikation, welche eine Grundlage für zukünftige Entwicklungen darstellt, geschafft und wird in einer nachfolgenden Version der Spezifikation enthalten sein. Auf der Spezifikationsseite wird ausschließlich die Definition und Verwendung von Policies betrachtet, wogegen auf der Seite des Compilers entsprechender Code generiert werden muss, welcher die Einhaltung der definierten Policies sicherstellt.

Integration von Sprachmodulen in die ausführbaren Applikationen

Jedes Sprachmodul verfügt über eine beliebige Anzahl von Definitionen und unter Umständen auch Implementationen. Während die Definitionen der Erweiterungen bereits in die generierte Data App eingefügt werden, ist dies für die Implementationen nicht der Fall. Stattdessen ist eine Manipulation des Klassenpfades zur Ausführung notwendig. Dieser Umweg ist zum einen unpraktisch und zum anderen wird die Nutzung von D° erschwert. Aus diesem Grund soll in einer zukünftigen Version des Compilers die Generierung so umgestaltet werden, dass am Ende eine einzelne ausführbare Datei erzeugt wird, welche ohne zusätzliche Abhängigkeiten verwendet werden kann.

Werkzeugunterstützung

Um dem Nutzungskomfort für D° zu erhöhen wäre die Existenz spezieller Entwicklungswerkzeuge (IDE, Debugger, Profiler, ...) wünschenswert.

Syntaxüberprüfung

Die aktuelle Version des D°-Compilers bietet bereits einige Arten von Syntaxüberprüfung. Im Laufe der zukünftigen Entwicklungen werden an geeignete Stellen noch weitere Überprüfungen (auch Semantische Prüfungen) in den Compiler eingebaut werden.

Smartes Parametermatching

Werden Parameterlisten (bspw. als Ein- oder Ausgabe für Aktivitäten) verwendet, muss die Reihenfolge der einzelnen Einträgen aktuell streng der Definition entsprechen. Hier ist es in bestimmten Fällen möglich, dass auf die exakte Reihenfolge verzichtet wird, solange eine eindeutige Abbildung zwischen der Definition und den tatsächlichen Parametern gefunden werden kann. Darüber hinaus kann eine Aktivität als Eingabeparameter momentan nicht die Ausgabe einer anderen Aktivität verwenden, da die verwendeten Container (sog. Scopes) nicht automatisch ausgepackt werden.

An dieser Stelle gibt es noch Verbesserungen die in zukünftige Versionen des D°-Compilers integriert werden sollen.

Unterschiedliche Ausprägungen von Data Apps

Aktuell stellt jede Service einen Webservice dar, welcher eine REST-Schnittstelle mit genau einem Endpunkt zur Verfügung stellt. In der Zukunft werden die Grammatik und infolgedessen auch der Compiler weitere Arten von Data Apps (bspw. klassische Applikationen ohne Schnittstelle nach aussen) unterstützen.

Weitere Arten von Aktivitäten

In der aktuellen Version des D°-Compilers werden alle Aktivitäten von Jave-Klassen, die in die finale Data App integriert werden bereitgestellt. Für zukünftige Versionen sind Erweiterungen geplant, die sowohl die Spezifikation, als auch den Compiler betreffen und hierdurch weitere Arten von Aktivitäten erlauben. Beispiele hierfür sind:

- Remote Activities: Werden nicht in die Data App integriert, sondern laufen in einem separaten Prozess auf der selben Maschine, oder auf einer entfernten Maschine
- XLanguage Activities: Sind in einer anderen Programmiersprache als die Data App geschrieben und werden in die Data App integriert

Geschickte Integration von Sprachmodulen in Data Apps

Die meißten Data Apps verwenden nur eine Teilmenge aller verfügbaren Spracherweiterungen. Der aktuelle Stand des D°-Compilers integriert dennoch alle verfügbaren Spracherweiterungen in die finale Data App. Hier soll im Rahmen der Weiterentwicklung eine "intelligente" Auswahl getroffen werden, wodurch (langfristig) wesentlich kleinere Data Apps erzeugt werden können.

Dedizierte Ausführungsumgebung

Data Apps, die mit dem aktuellen D°-Compiler erzeugt werden, werden aktuell direkt in der JVM ausgeführt. In der Zukunft wird eine dedizierte Ausführungsumgebung für D°-Data Apps entwickelt. Die Ausführungsumgebung bietet neben Managementfunktionalitäten für die einzelnen Data Apps auch zusätzliche Funktionalitäten wie bspw. die Isolation einzelner Data Apps durch Containerisierung.

Verknüpfung mit Remote Processing Technologie

In AP 4.7 (Advanced Data Processing) des Leistungszentrums Logistik & IT wurde eine erste Version für das sog. Remote Processing entwickelt. Beim Remote Processing handelt es sich um eine Technologie mit den folgenden Features:

- Versand von Applikationen an entferne Maschinen
- Automatische Isolation von versendeten Applikation durch Containerisierung
- Ausführung der isolierten Applikation
- Rücklieferung der Berechnungsergebnisse an den Absender der Applikation

Diese Technologie soll im Rahmen der Entwicklung von D° erweitert und integriert werden. Zunächst muss der korrekte Umgang mit D°-Data Apps in die Technologie integriert werden. Ein weiteres Ziel ist es eine Funktionalität nachzurüsten, welche es erlaubt, statt fertigen Applikationen, den Programmcode (und weitere eventuell notwendige Artefakte) zu übersenden und eine Übersetzung in eine ausführbare Applikation auf der Empfängerseite vorzunehmen. Hierdurch würden Ausführungsumgebung und Compiler für D° verschmelzen.

Glossar

Dieser Abschnitt erläutert Begriffe, die im D°-Kontext verwendet werden und in anderen Programmiersprachen häufig über eine andere Semantik verfügen oder in diesem Kontext ansonsten nicht geläufig sind.

Aktivität

In D° stellt eine Aktivität eine (komplexe) Funktionalität dar, welche aus der Sichtweise von D° atomar ausgeführt wird. Jede Aktivität besteht aus zwei unterschiedlichen Teilen: Der Definition und der Implementation.

Bei der Definition handelt es sich um eine rein textuelle Beschreibung der Aktivität, welche Informationen bspw. über Ein- und Ausgabeparameter enthält.

Die Implementation wird in einer ausführbaren Programmiersprache verfasst und enthält die eigentliche Funktionalität. Es ist notwendig die Implementation mit entsprechenden Metadaten (bspw. Annotationen in Java) zu versehen, damit der Compiler eine Verbindung zwischen Definition und Implementation vornehmen kann. Aktuell wird ausschließlich Java für die Implementation unterstützt.

Am ehesten ist das Konzept der Aktivität vergleichbar mit einer `native`-Methode in Java.

Datentyp

Der Datentyp in D° ist ein reiner Datencontainer und besitzt keinerlei eigene Funktionalität, abgesehen von dem lesenden und schreibenden Zugriff auf die enthaltenen Werte. D° unterscheidet zwischen zwei verschiedenen Arten von Datentypen: Primitiven Datentypen und Zusammengesetzten (Composite) Datentypen.

Primitive Datentypen sind elementar und enthalten direkt einen Wert. Außerdem verfügen sie über einen Standardwert und einen Validator, der für abgelegte Werte über die Gültigkeit entscheidet.

Zusammengesetzte Datentypen enthalten keine direkten Daten sondern eine Menge von primitiven und zusammengesetzten Datentypen, welche die eigentlichen Daten enthalten. Sie verfügen über keine Standardwerte, sondern nur über einen Validator.