Spezifikation V1

Versionen

Datum	Notiz	Version
30.06.2018	Erste Veröffentlichung	1.0

Autoren

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

Spezifikation

Dieses Dokument ist in der jeweils aktuellsten Version die offizielle Spezifikation für die domänenspezifische Programmiersprache D°. Im Rahmen des IDS kann D° als Ausführungssprache verwendet werden, beispielsweise zur Implementation der Data Services. Es dokumentiert die einzelnen Sprachkonstrukte der Grammatik und liefert an geeigneten Stellen Beispiele. Zur Veranschaulichung der einzelnen Konzepte wird eine spezielle Form von Syntaxdiagrammen (sogenannte Railroad Diagrams) verwendet.

Innerhalb der Diagramme sind drei wichtige Elemente zu unterscheiden:

- Rechteckige Knoten enthalten den Namen einer anderen produzierenden Regel (Nonterminalsymbole)
- Knoten mit abgerundeten Ecken enthalten Zeichen(-ketten) die so auch in den konkreten Ausprägungen der Grammatik vorkommen müssen (Terminalsymbole)
- Bei kursivem Text, der die Linien des Diagramms unterbricht handelt es sich um Kommentare, welche nur der Veranschaulichung dienen und keine Auswirkung auf die Syntax der Sprache haben

Die Konzepte der Sprache sind in sechs logische Gruppen unterteilt, welche nacheinander dokumentiert werden. Zunächst werden die vier zentralen Module der Sprache betrachtet: Datentypen, Aktivitäten, Policies, Data Apps. Anschließend werden gemeinsam genutzte Datenflusskonzepte und zuletzt allgemeine Konzepte, die an mehreren Stellen verwendet werden, spezifiziert. Bevor mit der Spezifikation der Sprachkonstrukte begonnen wird, werden die Operatoren von D° sowie die Semantik unterschiedlicher Klammerungen erläutert.

Kommentare, Operatoren & Klammerung

Bei der Entwicklung von D° wird versucht die Menge an Schlüsselworten möglichst klein zu halten. Stattdessen werden verschiedene Operatoren verwendet und unterschiedliche Klammerungen mit eigener Semantik belegt. Nachfolgend werden diese Sprachelemente dargestellt.

Kommentare

Notiz	Version
Erste Veröffentlichung	1.0

D° erlaubt die Verwendung von Kommentaren an jeder Stelle. Dabei ist die Syntax identisch zu der aus anderen Programmiersprachen (beispielsweise Java, C++).

Um ein einzeiliges Kommentar einzuleiten wird die Zeichenkette '`//`' verwendet, welche bewirkt, dass alle Zeichen bis zum Zeilenende während der Übersetzung ignoriert werden.

Mehrzeilige Kommentare werden durch die Zeichenkette '`/*`' eingeleitet. Es werden alle nachfolgenden Zeichen bis zur Zeichenkette '`*/`' ignoriert.

Darüber hinaus werden sämtliche Zeilenumbrüche, Leerzeichen & Tabulatoren ignoriert, was eine individuelle Formatierung des Codes erlaubt.

Definition (Operator)

Notiz	Version
Erste Veröffentlichung	1.0

Syntax

Innerhalb von D° wird für Definitionen neuer Elemente die Zeichenkette '`:=`' als Operator verwendet. Auf der linken Seite des Operators steht ein `qualified_name` und auf der rechten Seite ein Ausdruck. Dabei führt die Auswertung des Ausdrucks zur Erzeugung eines neuen Sprachelements, welches unter dem Bezeichner auf der linken Seite des Operators erreichbar ist. Bei diesem Element kann es sich um eine Aktivität, einen Datentypen oder einen Validator handeln. Die so definierten Sprachelemente werden während der Übersetzung mit der Data App verwoben und stehen somit atomar zur Verfügung.

Beispiel

```
RegexValidator := {# "ValidatorBuilder.create(\"Regex\").set(\"pattern\", \".*\").build()" #}
IntegerValidator := $RegexValidator(
    @set["pattern", "^(0|[1-9][0-9]*)$"]
)
```

Semantik

Die Verwendung des Operators bewirkt, dass der Wert des Ausdrucks auf der rechten Seite geklont wird und anschließend durch die Verwendung des `qualified_name` auf der linken Seite des Operators erreicht werden kann. Der Operator wird zur Definition von Sprachelementen aus den erweiterbaren Teilen der Sprache (Typsystem, Aktivitäten, Policies) verwendet und sorgt für eine Einbettung der Elemente in die Sprache.

Zuweisung (Operator)

Notiz	Version
Erste Veröffentlichung	1.0

Syntax

Um einfache Zuweisungen durchzuführen wird in D° das einfache Gleichheitszeichen '=' als Operator verwendet. Auf der linken Seite des Operators steht der Bezeichner unter dem der Ausdruck auf der rechten Seite des Operators erreichbar sein soll. Bei dem Bezeichner handelt es sich je nach Kontext um einen `IDENTIFIER` oder ein ein Array vom Typ `IDENTIFIER`.

Der Ausdruck auf der rechten Seite des Operators muss bei der Auswertung Instanzen von Sprachkonstrukten erzeugen, welche den `IDENTIFIER` zugewiesen werden. Es handelt sich bei dem Ausruck entweder um direkte Instanziierungen von Elementen (Aktivitäten, Validatoren) oder den Aufruf einer Aktivität. Da Aktivitäten in D° mehrere Rückgabewerte haben können ist es notwendig mehrere `IDENTIFIER` als Array auf der linken Seite des Operators anzugeben, damit alle Rückgabewerte in folgenden Anweisungen verwendet werden können.

Beispiel

```
MyIntegerValidator = $IntegerValidator()
[geoLocation, postAddress] = RetrieveAddress[person];
```

Semantik

Der oder die `IDENTIFIER` auf der linken Seite des Operators werden in jeder Situation als Referenzen auf Variablen interpretiert. Somit wird der Zuweisungsoperator ausschließlich für Variablenzuweisungen verwendet.

Referenz (Operator)

Notiz	Version
Erste Veröffentlichung	1.0

Syntax

Um die Sprachelemente (Datentypen, Aktivitäten, Validatoren) in D° zu referenzieren wird das Zeichen '$' verwendet. Es wird dabei dem `qualified_name` des jeweiligen Sprachelements vorangestellt.

Beispiel

```
MyIntegerValidator = $IntegerValidator()
```

Semantik

D° verwendet an vielen Stellen `IDENTIFIER` und `qualified_name` ohne Einschränkungen oder zusätzliche Schlüsselworte. Hierdurch ist es für den Anwender in manchen Situation nicht direkt erkennbar ob durch einen Ausdruck ein Sprachelement oder eine Instanz eines Sprachelements referenziert wird. Um diese Uneindeutigkeiten zu vermeiden wird der Referenzoperator verwendet. Wenn er einem `qualified_name` vorangestellt wird, ist klar erkennbar, dass ein Sprachelement referenziert wird.

Abbildung (Operator)

Notiz	Version
Erste Veröffentlichung	1.0

Syntax

Um eine Menge von Variablen zu einem Validator zuzuordnen verwendet D° die Zeichenkette '->'. Der Operator wird nur bei der Instanziierung von Aktivitäten verwendet, um diese Zuordnung vorzunehmen. Auf der linken und der rechten Seite des Operators ist je ein Array vom Typ `IDENTIFIER`.

Beispiel

```
onlyInHomeDirMax14BytesPerFile_WriteFileActivity = $WriteFileActivity(
    [filePath] -> [onlyInHomeDirValidator],
    [content] -> [max14BytesValidator]
)
```

Semantik

Wenn eine Instanz einer Aktivität erzeugt wird ist es notwendig, dass Instanzen von Validatoren (welche die definierte Policy durchsetzten sollen) mit den entsprechenden Parametern belegt werden, damit die Durchsetzung der Nutzungsbedingungen erfolgen kann. Das Array auf der linken Seite des Operators enthält die `IDENTIFIER` von Variablen, welche Daten enthalten. Die `IDENTIFIER`, welche sich im Array auf der rechten Seite des Operators befinden verweisen auf die Instanzen der Validatoren, welche in der Aktivität enthalten sind. Befinden sich auf der rechten Seite des Operators mehrere `ID ENTIFIER` findet die Zuordnung der Variablen auf der linken Seite für alle Einträge statt.

Funktion (Operator)

Notiz	Version
Erste Veröffentlichung	1.0

Syntax

Um bei der Definition von Sprachelementen und bei der Instanziierung von Validatoren auf externe Funktionalitäten zuzugreifen verwendet D° das '@' als Operator. Dabei wird der Operator einem `IDENTIFIER` vorangestellt und bewirkt, dass der nachfolgende Bezeichner als externer Funktionsname interpretiert wird.

Beispiel

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

Semantik

Bei der Definition von Sprachelementen (Datentypen, Aktivitäten, Validatoren), sowie bei der Instanziierung von Validatoren kann es notwendig sein auf Funktionen zuzugreifen, welche nicht in D° definiert sind. Dies wird damit begründet, dass D° in der ersten Version noch nicht in der Lage ist den D°-Compiler zu erzeugen. Dabei wird dem `IDENTIFIER`, welcher den Namen der externen Funktionalität enthält, der Operator vorangestellt und hinter dem Bezeichner die benötigten Parameter als Array angegeben.

Menge (Klammerung)

Notiz	Version
Erste Veröffentlichung	1.0

Syntax

Bei der Definition von Sprachelementen (Datentypen, Aktivitäten, Validatoren), sowie bei der Instanziierung von Validatoren und Aktivitäten müssen diverse Angaben gemacht werden, welche das Element/die Instanz beschreiben. Um diese Elemente zusammenzufassen werden sie in ein Klammerpaar '`(  )`' gefasst. Eine Trennung der einzelnen Elemente erfolgt in der Regel über Kommata '`,`', ist aber nicht in jedem Fall notwendig.

Beispiel

```
onlyInHomeDirMax14BytesPerFile_WriteFileActivity = $WriteFileActivity(
    [filePath] -> [onlyInHomeDirValidator],
    [content] -> [max14BytesValidator]
)
```

Semantik

Um eine Menge von zusammengehörigen (aber nicht zwingend gleichartigen/-getypten) Elementen zusammenzufassen wird diese Art der Klammerung verwendet.

Listen & Arrays (Klammerung)

Notiz	Version
Erste Veröffentlichung	1.0

Syntax

An verschiedenen Stellen von D° können bzw. müssen Arrays/Listen verwendet werden. Arrays werden von eckigen Klammern '`[ ]`' umfasst und kenntlich gemacht. Eine Trennung der einzelnen Elemente erfolgt in der Regel über Kommata '`,`', ist aber nicht in jedem Fall notwendig.

Beispiel

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

Semantik

Um eine Menge von zusammengehörigen& gleichartigen Elementen zusammenzufassen wird diese Art der Klammerung verwendet.

Skriptausdrücke (Klammerung)

Notiz	Version
Erste Veröffentlichung	1.0

Syntax

Bei der Definition neuer Datentypen kann in D° ein Stück Programmcode in einer Skriptsprache verwendet werden. Dabei wird dem Skript die Zeichenkette '`{#`' vorangestellt. Um das Skript zu beenden wird die Zeichenkette '`#}`' verwendet.

Beispiel

```
RegexValidator := {# "ValidatorBuilder.create(\"Regex\").set(\"pattern\", \".*\").build()" #}
```

Semantik

Da D° es in der ersten Version nicht ermöglicht den D°-Compiler mit allen Komponenten zu erzeugen, ist es bei der Definition von Datentypen unter Umständen notwendig auf externe Funktionalitäten zuzugreifen. Dies wird über die Verwendung der Skriptausdrücke komfortabel erlaubt. Die gewählte Skriptsprache ist JavaScript und in der Ausführungsumgebung der Skriptausdrücke wurden alle relevanten Teile des Typsystems verfügbar gemacht.

Datentypen

Das Typsystem ist eins der zentralen, erweiterbaren Systeme von D°. Dieser Abschnitt beschreibt die Sprachkonstrukte, die dazu verwendet werden können um neue Datentypen zu definieren, welche in den anderen Systemen der Sprache (Aktivitäten, Policies, Data Apps) verwendet werden können.

Types File

Notiz	Version
Erste Veröffentlichung	1.0

Das oberste Element bei der Definition von neuen Datentypen ist die Types File. Für jede Instanz dieses Sprachelements ist eine eigene Datei vorgesehen, die eine beliebige Anzahl von Typdefinitionen und Namensräumen enthält.

Sofern möglich sollte die Menge der Typdefinitionen innerhalb der Datei ein in sich geschlossenes Modul ergeben, das keine/möglichst wenige externe Abhängigkeiten besitzt.

Um Datentypen besser gruppieren zu können, ist die Verwendung von Namensräumen möglich. Innerhalb eines Types Namespace Blocks erhalten alle definierten Datentypen den Namensraum des umschließenden Blocks. Eine explizite Angabe des Namensraums ist an dieser Stelle nicht möglich. Typdefinitionen, die außerhalb eines Types Namespace Block erfolgt sind können mit vollqualifiziertem Namensraum definiert werden. Wird auf die Angabe des Namensraums außerhalb der Blockumgebung verzichtet, erhält der definierte Datentyp implizit den Namensraum '`core`'.

Analog können Referenzen auf bereits existierende Typen vollqualifiziert erfolgen oder ohne die Angabe des Namensraums, was zur Verwendung des Namensraums `core` führt.

Beispiel

```
Text := {# "TypeBuilder.define(\"Text\").initialValue(\"\").register()" #}

core.Integer := $Text(
    @validate[$IntegerValidator],
    @initialValue[0]
)

internal [
    Month := $Integer(
        @validate[$MonthValidator],
        @initialValue[1]
    )
]
```

Types Namespace Block

Notiz	Version
Erste Veröffentlichung	1.0

Um die Nutzbarkeit zu erhöhen und die Lesbarkeit zu verbessern können Typdefinitionen, die den selben vollqualifizierten Namensraum verwenden in einem Types Namespace Block gekapselt werden.

Eine individuelle Angabe des Namensraums für einzelne Typdefinitionen ist innerhalb des Types Namespace Blocks nicht möglich, stattdessen werden alle enthaltenen Typdefinitionen mit dem Namensraum des Blocks versehen.

Beispiel

```
Text := {# "TypeBuilder.define(\"Text\").initialValue(\"\").register()" #}


myCompany [

    Integer := $Text(
        @validate[$IntegerValidator],
        @initialValue[1]
    )

    Second := $myCompany.Integer(
        @validate[$SecondValidator],
        @initialValue[0]
    )
]
```

Type Definition

Notiz	Version
Erste Veröffentlichung	1.0

Aktuell erlaubt D° Typdefinitionen auf zwei verschiedene Arten.

Zum einen können Skriptausdrücke verwendet werden, die durch das Typsystem, während der Übersetzung, direkt evaluiert werden und zum anderen können die Typen direkt in D° definiert werden.

Beispiel

```
Text := {# "TypeBuilder.define(\"Text\").initialValue(\"\").register()" #}

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

Script Expression

Notiz	Version
Erste Veröffentlichung	1.0

D° erlaubt es in der aktuellen Version Skriptausdrücke zur Definition von neuen Datentypen zu verwenden. Diese Skriptausdrücke werden unverändert während der Übersetzung durch den Compiler ausgewertet. Relevante Komponenten des Typsystems sind in der Skriptumgebung verfügbar und können verwendet werden. Die verwendete Skriptsprache ist Javascript.

Im aktuellen Entwicklungsstand von D° ist es notwendig, dass es möglich ist Datentypen zu definieren, die nicht ausschließlich durch D° definiert werden. Dies liegt darin begründet, dass das Typsystem initial über keinerlei Typen verfügt. Da bei der Typdefinition aber immer ein generalisierender Datentyp angegeben werden muss, ist es notwendig, das Typsystem zunächst durch die Verwendung von Skriptausdrücken mit rudimentären Datentypen zu befüllen. Diese können auch leer sein und ausschließlich als Container verwendet werden.

Beispiel

```
Text := {# "TypeBuilder.define(\"Text\").initialValue(\"\").register()" #}
```

D° Expression

Notiz	Version
Erste Veröffentlichung	1.0

Neben der Typdefinition durch die Verwendung von Skriptausdrücken, erlaubt D° die Definition neuer Datentypen durch die alleinige Nutzung von D° Mitteln.

Dabei besteht der einfachste Datentyp aus einem (vollqualifiziertem) Bezeichner, unter dem der Typ erreichbar ist und die Referenz auf einen generalisierenden Datentypen. Darüber hinaus kann eine beliebige Menge von Definition Functions zur Typdefinition hinzugefügt werden. Diese Funktionen beschreiben den Datentypen näher und benennen beispielsweise Validatoren, weitere generalisierende Datentypen und Attribute.

Beispiel

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

Aktivitäten

Wie auch beim Typsystem von D° handelt es sich bei den Aktivitäten um eine erweiterbare Menge, die atomar mit der Sprache verwoben wird, sobald die Übersetzung stattfindet. D° bringt keine fest integrierten Aktivitäten mit, sondern nur ein core-Paket, welches den Erweiterungsmechanismus verwendet. Bei Bedarf können Entwickler selbstständig Erweiterungen (neue Datentypen und/oder Aktivitäten) für D° entwickeln.

Activities File

Notiz	Version
Erste Veröffentlichung	1.0

Genau wie bei der Definition neuer Datentypen, ist das oberste Element zur Definition neuer Aktivitäten die Activities File, die jeweils in einer eigenen Datei platziert werden muss. In dieser Datei befindet sich eine beliebige Menge von Aktivitätsdefinitionen die in Activities Namespace Blocks organisiert sein können.

Aktivitäten die innerhalb eines Activities Namespace Blocks definiert werden erhalten alle den Namensraum des Blocks. Eine individuelle Angabe des Namensraums ist an dieser Stelle nicht möglich. Außerhalb der Blockumgebung kann der Namensraum (vollqualifiziert) angegeben werden. Wird auf die Angabe verzichtet, wird automatisch der core-Namensraum verwendet.

Beispiel

```
AddressCheck:= $Activity(
    @input["address", $Address],
    @output["valid", $Boolean],
    @provides["address", "valid", $Boolean]
)


myCompany [
        CreditCheck := $Activity(
                    @input["customer", $myCompany.Customer"],
            @output["solvent", $Boolean]
            @provides["customer", solvent, $Boolean]
        )
]
```

Activity Namespace Block

Notiz	Version
Erste Veröffentlichung	1.0

Zur Verbesserung von Nutzbarkeit und Lesbarkeit der Aktivitätsdefinitionen können mehrere Aktivitäten, die im selben Namensraum platziert werden sollen, in einer gemeinsamen Blockumgebung platziert werden.

Innerhalb eines Activity Namespace Blocks ist die explizite Angabe des Namensraums für neue Aktivitäten nicht möglich.

Beispiel

```
myCompany [
        CreditCheck := $Activity(
                    @input["customer", $myCompany.Customer"],
            @output["solvent", $Boolean]
            @provides["customer", solvent, $Boolean]
        )
]
```

Activity Definition

Notiz	Version
Erste Veröffentlichung	1.0

Die Definition von Aktivitäten erfolgt direkt in D°. Im einfachsten Fall besteht die Aktivitätsdefinition nur aus einem (vollqualifiziertem) Bezeichner unter dem die Aktivität erreichbar ist. Anders als bei Datentypen in D° gibt es bei Aktivitäten keine generalisierenden bzw. spezialisierenden Beziehungen zwischen Aktivitäten. Stattdessen unterscheidet D° zwischen atomaren und zusammengesetzten Aktivitäten.

Bei einer atomaren Aktivität handelt es sich um eine Aktivität, bei der die eigentliche Funktionalität nicht in D°, sondern in einer anderen Technologie implementiert ist. Innerhalb von D° ist die Aktivität aber atomar und kann nicht weiter zerlegt werden. Im Gegensatz dazu hat eine zusammengesetzte Aktivität keine Funktionalität in anderen Technologien sondern besitzt einen eigenen Kontrollfluss, welcher aus beliebig vielen Kontrollflusselementen, sowie atomaren und anderen zusammengesetzten Aktivitäten besteht. Die Unterscheidung zwischen atomaren und zusammengesetzten Aktivitäten erfolgt implizit. Sobald eine Aktivitätsdefinition über ein Block-Element verfügt, handelt es sich um eine zusammengesetzte Aktivität.

Beispiel

```
AddressCheck:= $Activity(
    @input["address", $Address],
    @output["valid", $Boolean],
    @provides["address", "valid", $Boolean]
)


myCompany [
        ProcessOrder := $Activity(
            @input["order", $shipping.Order]
            begin
                [availableCart, notAvailbeCart] = shipping.StockCheck[order.cart];
                [solvent] = myCompany.CredeitCheck[order.customer];
                [addressValid] = AddressCheck[order.deliveryAddress];

                if(solvent && addressValid) begin
                    shipping.Ship[order.deliveryAddress, availableCart];
                    accounting.Invoicing[order.billingAddress, availableCart];
                end
            end
        )
]
```

Policies

Das dritte Erweiterbare System in D° wird verwendet um Policies zu definieren. Policies sind dafür da, die Nutzungsbedingungen zu definieren, welche den sachgemäßen Umgang mit Daten gewährleisten.

Policies File

Notiz	Version
Erste Veröffentlichung	1.0

Bei der Definition von Policies sind drei verschiedene Elemente relevant, wobei es sich aber nur bei einem Element um die Definition neuer Sprachelemente handelt.

1. Validator Definition: Um die einzelnen Aspekte von Policies hinsichtlich ihrer Umsetzung zu überprüfen werden in D° Validatoren verwendet, welche zunächst definiert werden müssen.
2. Validator Instantiation: Instanzen der zuvor definierten Validatoren werden mit notwendigen Details der Policy gefüllt, um die Überprüfung durchführen zu können.
3. Activity Instantiation: Aktivitäten, die zuvor im Aktivitätssystem definiert wurden, werden Instanziiert und mit Instanzen der Validatoren verknüpft, welche die Nutzungsbedingungen durchsetzen.

Eine Policy in D° ist somit eine Menge von Validator Definitionen und Instanzen von Validatoren und Aktivitäten.

Beispiel

```
RegexValidator := $Validator(
     @constructorArg["pattern", $Text],
     @input["text", $Text]
)


deliverToEuropeOnly := $RegexValidator(
    @set["pattern", "(?i)^Europe$"]
)

addressCheck = $AddressCheck(
    [address.country] -> [deliverToEuropeOnly]
)
```

Validator Definition

Notiz	Version
Erste Veröffentlichung	1.0

Um die einzelnen Aspekte von Policies zu überprüfen, werden in D° Validatoren verwendet. Diese Validatoren müssen vor ihrere Verwendung definiert werden. Die eigentliche Logik zur Überprüfung des betrachteten Aspekts ist (analog zu atomaren Aktivitäten) in einer anderen Technologie implementiert.

Eine beliebige Menge Definition Functions kann verwendet werden um den definierten Validator genauer zu beschreiben. Beispielsweise werden durch diese Funktionen Eingabewerte und eventuelle Konstruktor-Parameter, die zur Parametrisierung des Verhaltens verwendet werden, angegeben.

Beispiel

```
RegexValidator := $Validator(
     @constructorArg["pattern", $Text],
     @input["text", $Text]
)
```

Validator Instantiation

Notiz	Version
Erste Veröffentlichung	1.0

Da Validatoren häufig durch Parameter in ihrem konkreten Verhalten angepasst werden können, ist es notwendig, dass aus den Definitionen Instanzen erzeugt werden, welche entsprechend konfiguriert sind, um dem jeweiligen Anwendungsfall gerecht zu werden.

Wie auch bei der Definition neuer Validatoren, ist es möglich über Definition Functions die erzeugte Instanz anzupassen. Beispielsweise können Parameter an die Instanz übergeben werden.

Beispiel

```
deliverToEuropeOnly := $RegexValidator(
    @set["pattern", "(?i)^Europe$"]
)
```

Activity Instantiation

Notiz	Version
Erste Veröffentlichung	1.0

Die Definition einer Aktivität ist zunächst nicht mit Validator (Instanzen) zur Durchsetzung von Policies verbunden. Um diese Lücke zu schließen ist es notwendig aus der Definition der Aktivität eine Instanz zu erzeugen, welche mit konkreten Instanzen von Validatoren verbunden ist.

Bei der Zuordnung von Validator-Instanzen zu Aktivitäts-Instanzen ist es notwendig, dass die relevanten Parameter der Aktivität mit den Instanzen der Validatoren verbunden werden. Eine beliebige Menge dieser Abbildungen kann bei der Instanziierung von Aktivitäten angegeben werden.

Beispiel

```
addressCheck = $AddressCheck(
    [address.country] -> [deliverToEuropeOnly]
)
```

Parameter to Validator Mapping

Notiz	Version
Erste Veröffentlichung	1.0

Bei der Verknüpfung von Validator-Instanzen mit der Instanz einer Aktivität ist es notwendig, dass die Parameter der Aktivität auf die Validatoren abgebildet werden, damit die entsprechenden Überprüfungen durchgeführt werden können.

Bei der Angabe dieser Abbildungen wird der Abbildungs-Operator verwendet. Dabei stehen auf der linken Seite des Operators die Parameter der Aktivität und auf der rechten Seite die Validator-Instanzen auf die abgebildet werden soll.

Beispiel

```
onlyInHomeDirMax14BytesPerFile_WriteFileActivity = $WriteFileActivity(
    [filePath] -> [onlyInHomeDirValidator],
    [content] -> [max14BytesValidator]
)
```

Data Apps

Eine Data App ist eine Applikation die in D° entwickelt wurde. Sie ist das zentrale Konzept der Sprache. Anders als die Systeme zur Definition von Sprachelementen ist die Struktur von Data Apps fest und kann nicht durch Entwickler verändert werden.

Data App File

Notiz	Version
Erste Veröffentlichung	1.0

Bei der Entwicklung einer Data App ist die Data App File das zentrale Konstrukt. Jede Data App muss in einer eigenen Datei abgelegt werden.

Eine Data App besteht dabei aus zwei verschiedenen Komponenten: Der Konfiguration und dem Code.

Beispiel

```
App configuration
        application namespace : degree.myCompany
        application name : ProcessOrder


App code
        begin
            [availableCart, notAvailbeCart] = shipping.StockCheck[order.cart];
            [solvent] = myCompany.CredeitCheck[order.customer];
        [addressValid] = AddressCheck[order.deliveryAddress];

            if(solvent && addressValid) begin
                shipping.Ship[order.deliveryAddress, availableCart];
                   accounting.Invoicing[order.billingAddress, availableCart];
               end
           end
```

Data App Configuration

Notiz	Version
Erste Veröffentlichung	1.0

Die Konfiguration einer Data App befindet sich oberhalb des Programmcodes und wird (zur besseren Lesbarkeit) durch die Schlüsselwort-Kette 'App `configuration`' eingeleitet. Die eigentliche Konfiguration besteht aus `key : value` Paaren.

Die aktuelle Version von D° erlaubt es im Konfigurationsbereich nur den Namen und den Namensraum der Applikation festzulegen. Weitere Werte bzw. Freiheiten bei der Wertangabe gibt es nicht. Der Aufbau des Data App Configuration Elements kann sich in künftigen Versionen der Grammatik noch ändern bzw. das Element vollständig wegfallen.

Beispiel

```
App configuration
        application namespace : degree.myCompany
        application name : ProcessOrder
```

Data App Code

Notiz	Version
Erste Veröffentlichung	1.0

Das Herzstück einer Data App ist der Programmcode der die Funktionalität definiert. Der Programmcode wird (zur besseren Lesbarkeit) durch die Schlüsselwort-Kette '`App code`' eingeleitet.

Der Programmcode einer Data App besteht aus einem einzelnen Block der beliebige Statements enthalten kann.

Beispiel

```
App code
        begin
            [availableCart, notAvailbeCart] = shipping.StockCheck[order.cart];
            [solvent] = myCompany.CredeitCheck[order.customer];
        [addressValid] = AddressCheck[order.deliveryAddress];

            if(solvent && addressValid) begin
                shipping.Ship[order.deliveryAddress, availableCart];
                   accounting.Invoicing[order.billingAddress, availableCart];
               end
           end
```

Datenfluss

Block

Notiz	Version
Erste Veröffentlichung	1.0

Oberstes Element bei der Erstellung von Programmcode innerhalb von D° ist der Block. Er wird durch die Schlüsselwörter '`begin`' & '`end`' eingeleitet bzw. beendet und ist verwandt zu den Block-Konzepten anderer Programmiersprachen (bspw. C++, Java).

Ein Block besteht aus einer Menge von Statements und wird Sequentiell ausgeführt.

Beispiel

```
begin
           [availableCart, notAvailbeCart] = shipping.StockCheck[order.cart];
    [solvent] = myCompany.CredeitCheck[order.customer];
    [addressValid] = AddressCheck[order.deliveryAddress];

           if(solvent && addressValid) begin
        shipping.Ship[order.deliveryAddress, availableCart];
               accounting.Invoicing[order.billingAddress, availableCart];
           end
end
```

Statement

Notiz	Version
Erste Veröffentlichung	1.0

Um Blöcke mit Funktionalität anzureichern werden Statements verwendet.

Ein Statement kann ein Aktivitätsaufruf oder ein Block sein. Diese Menge wird in folgenden Versionen der Grammatik angepasst bzw. erweitert, um auch andere Sprachkonstrukte zu erlauben.

Beispiel

```
begin
           [availableCart, notAvailbeCart] = shipping.StockCheck[order.cart];
    [solvent] = myCompany.CredeitCheck[order.customer];
    [addressValid] = AddressCheck[order.deliveryAddress];

           if(solvent && addressValid) begin
        shipping.Ship[order.deliveryAddress, availableCart];
               accounting.Invoicing[order.billingAddress, availableCart];
           end
end
```

Activity Call

Notiz	Version
Erste Veröffentlichung	1.0

Um die Funktionalität einer Aktivität, die in D° definiert wurde, auszulösen wird der Activity Call verwendet. Wird beim Activity Call eine atomare Aktivität aufgerufen, ist dies eine atomare Aktion die entweder vollständig, oder gar nicht durchgeführt wird.

Beim Aufruf einer Aktivität ist es möglich eine beliebige Anzahl von Ein- und Ausgabeparameter anzugeben.

Beispiel

```
[availableCart, notAvailbeCart] = shipping.StockCheck[order.cart];
[solvent] = myCompany.CredeitCheck[order.customer];
[addressValid] = AddressCheck[order.deliveryAddress];
```

Common

Neben den Datenfluss-Elementen gibt es einige Konzepte die an einer Vielzahl von Stellen verwendet werden.

Qualified Name

Notiz	Version
Erste Veröffentlichung	1.0

Um verschiedene Elemente von D° (unter Anderem Datentypen und Aktivitäten) zu gruppieren und diese Zugehörigkeit sichtbar zu machen, werden Qualified Names zur Benennung verwendet. Diese bestehen aus nicht-leeren Menge von Identifiern, wobei der letzte Identifier als Name für das jeweilige Element verwendet wird und die verbleibenden Identifier als Namensraum. Zwischen zwei Identifiern befindet sich immer ein '`.`' zur Trennung.

Insbesondere ist auch ein alleinstehender Identifier (ohne Namensraum) ein Qualified Name. Wird auf die Angabe des Namensraums verzichtet, wird implizit der Standard-Namensraum '`core`' verwendet. Bestimmte Namensräume können geschützt werden und als Folge dessen nicht durch alle Entwickler verwendet werden (beispielsweise der Namensraum '`system`').

Beispiel

```
shipping.StockCheck
myCompany.CredeitCheck
AddressCheck
```

Identifier

Notiz	Version
Erste Veröffentlichung	1.0

Der Identifier ist der zentrale Bezeichner für Elemente innerhalb von D°. Es ist eine alphanumerische Zeichenkette, die Unterstriche enthalten darf und mit einem Buchstaben beginnen muss.

Je nach Kontext wird ein Identifier unterschiedlich interpretiert. Beispielsweise als Variablenname, Namensraum, Aktivitätsname oder Funktionsname.

Beispiel

```
__shipping
myCompany
AddressCheck
```

Definition Function

Notiz	Version
Erste Veröffentlichung	1.0

Bei der Definition und Instantiierung von verschiedenen Elementen innerhalb von D° ist es notwendig zusätzliche Definitionen/Konfigurationen am jeweiligen Element vorzunehmen. Hierfür werden Definition Functions verwendet.

Ein Identifier benennt die Funktionalität die angepasst/aufgerufen werden soll. Erlaubte Werte sind dabei Kontextabhängig. Eine Menge von Argumenten wird als Array an die entsprechende Funktionalität übergeben.

Beispiel

```
@constructorArg["pattern", $Text],
@input["text", $Text]
@set["pattern", "(?i)^Europe$"]
```

Definition Function Argument

Notiz	Version
Erste Veröffentlichung	1.0

Die Argumente, welche an Definition Functions übergeben werden können, können unterschiedliche Typen haben. Sowohl Literale als auch Referenzen auf Elemente/Instanzen sind möglich.

Die Auswahl an möglichen Ausprägungen der Argumente ist mit der aktuellen Version der Grammatik eingeschränkt und kann sich in folgenden Versionen ändern.

Beispiel

```
"Kundennummer: 477567"
15
$myCompany.customer
order
```

String Literal

Notiz	Version
Erste Veröffentlichung	1.0

Um beliebige Zeichenketten an eine Defintion Function zu übergeben, wird das String Literal verwendet.

Es ist verwandt zu String Literalen aus anderen Programmiersprachen (bspw. C++, Java).

Beispiel

```
"Kundennummer: 477567"
""
"Hello\\r\\nWorld!"
```

Integer Literal

Notiz	Version
Erste Veröffentlichung	1.0

Um positive, ganzzahlige Werte als Argumente an Definition Functions zu übergeben, werden Integer Literals verwendet.

Dabei ist jede positive Ganzzahl abbildbar; es gibt keine Obergrenze.

Beispiel

```
0
17
3871
```