Spezifikation V2

Versionen

Datum	Notiz	Version
31.12.2018	Zweite Veröffentlichung Verfeinerung von Konzepten Ausarbeitung des Policy & Instanz-Konzepts	2.0
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 die folgenden Punkte zu beachten:

- Knoten mit abgerundeten Ecken enthalten den Namen einer anderen produzierenden Regel (Nonterminalsymbole)
- Rechteckige Knoten enthalten Zeichen(-ketten) die so auch in den konkreten Ausprägungen der Grammatik vorkommen müssen (Terminalsymbole)
- Knoteninhalte in Hochkommata (') sind Zeichenketten, die in der Ausprägung der Grammatik vorkommen müssen
- 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 Schreibrichtung der Diagramme ist von links nach rechts
- Das Diagramm beginnt auf der linken Seite an der Linie, die auf der linken Seite keinen Knoten hat und endet auf der rechten Seite an der Linie, die auf der rechten Seite keinen Knoten hat
- In jedem Knoten darf am Ausgang eine Linie verfolgt werden, um zum nächsten Knoten bzw. zum Ende zu gelangen
- Der Knoten mit dem Inhalt "EOF" hat eine Sonderrolle, da er das Ende der aktuellen Datei markiert (end-of-file)
- Die Diagramme wurden automatisch generiert, weswegen nicht in allen Fällen eine optimale Darstellung der Regeln vorliegt

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.

Schlüsselworte

Während der Entwicklung von D° wird versucht die Menge an notwendigen Schlüsselwörtern so klein wie möglich zu halten. Dies verringert die Einstiegshürde bei der Verwendung von D° und gibt den Entwicklern größere Freiheiten bei der Wahl von Namen für Sprachelemente.

Im Folgenden werden alle Schlüsselwörter von D° aufgelistet. Diese dürfen nur an den vorgesehenen Stellen verendet werden und können nicht anderweitig in Data Apps verwendet werden (beispielsweise als Variablenname). Die Liste ist nicht final und kann sich in zukünftigen Versionen ändern. Dabei können sowohl neue Elemente hinzugefügt, als auch bestehende entfernt werden.

Es sei darauf hingewiesen, dass die Grammatik von D° die Groß- und Kleinschreibung bei Schlüsselwörtern aktuell ignoriert, weswegen beispielsweise `if`, `If`, `iF` & IF als Schlüsselwort if erkannt werden.

```
activity, app, application, begin, code, configuration, constraint, else, end, if, name, namespace, policy, port, return, version
```

In den nachfolgenden Abbildungen werden die Schlüsselwörter nicht direkt angegeben, sondern die jeweils erzeugenden Regeln. Diese sind alle nach dem Schema `KEYWORD_<<`**`KEYWORD_NAME`**`>>` aufgebaut.

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
Verfeinerte Beschreibung	2.0
Erste Veröffentlichung	1.0

Syntax

Innerhalb von D° wird für Definitionen neuer Elemente die Zeichenkette '`:=`' als Operator verwendet. Die dazugehörige Grammatikregel heißt `DEFINE_OPERATOR`. 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, eine Policy, 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 Ergebnis der Auswertung des Ausdrucks auf der rechten Seite 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.

Die `qualified_name` auf der linken Seite des Operators bilden in ihrer Gesamtheit die dynamische Menge an Schlüsselworten, welche in D° zur Verfügung stehen. Aus diesem Grund müssen die gewählten Bezeichner eindeutig sein, was auch über die verschiedenen Subsysteme hinweg gilt.

Zuweisung (Operator)

Notiz	Version
Verfeinerte Beschreibung	2.0
Erste Veröffentlichung	1.0

Syntax

Um einfache Zuweisungen durchzuführen wird in D° das einfache Gleichheitszeichen '`=`' als Operator verwendet. Die dazugehörige Grammatikregel heißt `ASSIGNMENT_OPERATOR`. 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 eine oder mehrere Sprachkonstrukte erzeugen, welche den `IDENTIFIER`n zugewiesen werden. Es kann sich dabei sowohl um Sprachelemente (beispielsweise Instanzen von Policies und Aktivitäten) handeln, als auch um klassische Variablen (Instanzen von Datentypen). Die Anzahl der `IDENTIFIER` auf der rechten Seite müssen mit der Anzahl der erzeugten Elemente des Ausdrucks auf der rechten Seite übereinstimmen. Somit ist ein gewisses Vorwissen über den verwendeten Ausdruck notwendig.

Generell lässt sich zwischen Aufrufen von Aktivitäten und der Instanziierung von Sprachelementen unterscheiden. Aktivitäten können mehrere Rückgabewerte haben, weswegen in diesem Fall immer die ein (eventuell einelementiges) Array verwendet werden muss. Instanziierungen erzeugen immer ein einzelnes Element, weswegen hier kein Array verwendet werden darf.

Beispiel

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

Semantik

Der oder die IDENTIFIER auf der linken Seite des Operators werden dem aktuellen Sichtbarkeitsbereich (Scope) hinzugefügt. In einer Data App wäre dies der aktuelle BLOCK, wogegen bei Instanziierungen in Sprachmodulen eine globale Sichtbarkeit verwendet wird.

Referenz (Operator)

Notiz	Version
Erste Veröffentlichung	1.0

Syntax

Um die Sprachelemente (Datentypen, Aktivitäten, Policies, Validatoren) in D° zu referenzieren wird das Zeichen '$' verwendet. Es wird dabei dem qualified_name des jeweiligen Sprachelements vorangestellt. Der Operator wird in der Grammatik durch die Regel REFERENCE_OPERATOR abgebildet.

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) - Entfernt

Entferntes Element

Das beschriebene Element ist nicht länger Bestandteil der Grammatik von D° und ist hier nur aus Gründen der Vollständigkeit und Nachvollziehbarkeit in diesem Dokument enthalten.

Grund der Entfernung: Vereinheitlichung der Syntax über verschiedene Sprachkonstrukte hinweg.

Notiz	Version
Entfernt	2.0
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 IDENTIFIER findet die Zuordnung der Variablen auf der linken Seite für alle Einträge statt.

Funktion (Operator)

Notiz	Version
Verfeinerte Beschreibung	2.0
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. Die dazugehörige Grammatikregel heißt `AT`. 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°, sondern in darunterliegenden Komponenten wie den Compiler, definiert sind. Dies wird damit begründet, dass D° in der aktuellen Version 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
Beispiel aktualisiert	2.0
Erste Veröffentlichung	1.0

Syntax

Bei der Definition von Sprachelementen (Datentypen, Aktivitäten, Policies, 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 durch die Verwendung von Kommata '`,`'.

Beispiel

```
Year2019Validator := $DateValidator(
    @set["day", "^([1-9]|[1-2][0-9]|3[0-1])$"],
    @set["month", "^([1-9]|1[0-2])$"],
    @set["year", "2019"]
)
```

Semantik

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

Listen & Arrays (Klammerung)

Notiz	Version
Verfeinerte Beschreibung	2.0
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 durch die Verwendung von Kommata '`,`'.

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.

Dabei ist der Begriff der Zusammengehörigkeit weiter gefasst als beispielsweise in Java, wo ein Array einen festen Typ haben muss und nur entsprechende Werte aufnehmen kann. Die Zusammengehörigkeit in D° ist allgemeiner. Beispielsweise kann es ein Array von Werten oder von Bezeichnern geben. Wie eng die Zusammengehörigkeit im einzelnen zu verstehen ist, hängt von den einzelnen Konzepten der Sprache ab.

Skriptausdrücke (Klammerung)

Notiz	Version
Verfeinerte Beschreibung	2.0
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 aktuellen 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.

Auf die Verwendung dieser Skriptausdrücke sollte, sofern möglich verzichtet werden, da sie in einer zukünftigen Version von D° entfernt werden könnten.

Datentypen

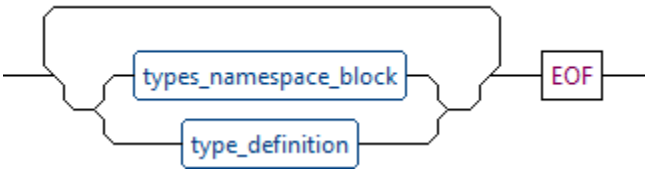
Das Typsystem ist eines der zentralen, erweiterbaren Systeme von D°. Dieser Abschnitt beschreibt die Sprachkonstrukte, die dazu verwendet werden können um neue Datentypen zu definieren, welche in den anderen Systemen der Sprache (bspw. Aktivitäten, Policies, Data Apps) verwendet werden können. Das Typsystem von D° ist initial leer. Eine Grundlage an Datentypen und Validatoren zur Verwendung wird durch das `core`-Modul mitgeliefert.

`types_file`

Notiz	Version
Neue Abbildung, verfeinerte Beschreibung	2.0
Erste Veröffentlichung	1.0

Das oberste Element bei der Definition von neuen Datentypen ist das Datentyp-Sprachmodul, welches durch die `types_file` umgesetzt wird. 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. Hierdurch werden auch Namenskonflikte mit Sprachmodulen dritter vermieden. Innerhalb eines `types_namespace_block` 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
Neue Abbildung, verfeinerte Beschreibung	2.0
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_block` nicht möglich, stattdessen werden alle enthaltenen Typdefinitionen mit dem Namensraum des Blocks versehen.

Ebenfalls ist zu beachten das es momentan nicht möglich ist mehrere Elemente des Typs `types_namespace_block` ineinander zu verschachteln.

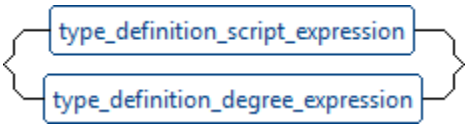
Beispiel

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

myCompany [

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

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

type_definition

Notiz	Version
Neue Abbildung	2.0
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.

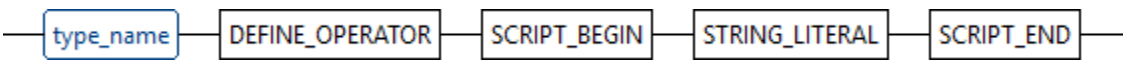
Beispiel

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

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

type_definition_script_expression

Notiz	Version
Neue Abbildung	2.0
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.

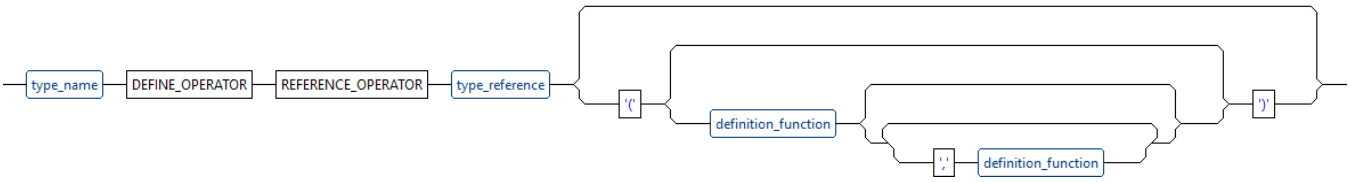
Beispiel

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

`type_definition_degree_expression`

Notiz	Version
Neue Abbildung, verfeinerte Beschreibung, Erweiterung des Beispiels	2.0
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_function` zur Typdefinition hinzugefügt werden. Diese Funktionen beschreiben den Datentypen näher und benennen beispielsweise Validatoren, weitere generalisierende Datentypen und Attribute.

Die Verwendung von `definition_function`-Elementen ist optional. Aus Gründen des Nutzungskomforts kann die Klammerung entfallen, sofern keine `definition_functions` verwendet werden.

Beispiel

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


Float := $Text
```

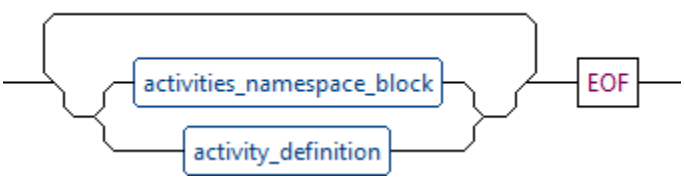
Aktivitäten

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
Neue Abbildung, verfeinerte Beschreibung	2.0
Erste Veröffentlichung	1.0

Genau wie bei der Definition neuer Datentypen, ist das oberste Element zur Definition neuer Aktivitäten das Sprachkonstrukt `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_block` organisiert sein können. Jede Instanz dieses Sprachkonstrukts bildet ein eigenes Sprachmodul.



Aktivitäten die innerhalb eines `activities_namespace_block` 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.

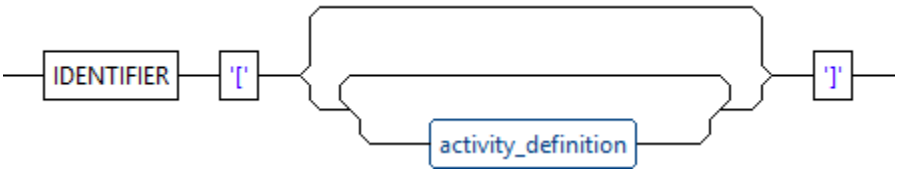
Beispiel

```
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]
        )
]
```

`activities_namespace_block`

Notiz	Version
Neue Abbildung, verfeinerte Beschreibung	2.0
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 `activities_namespace_block` ist die explizite Angabe des Namensraums für neue Aktivitäten nicht möglich. Eine Verschachtelung von `activities_namespace_block`-Elementen ist nicht möglich.

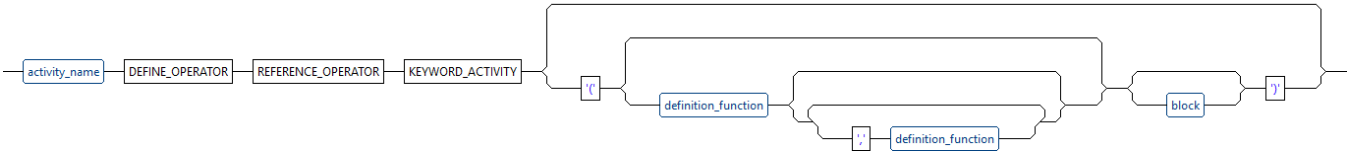
Beispiel

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

`activity_definition`

Notiz	Version
Neue Abbildung	2.0
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

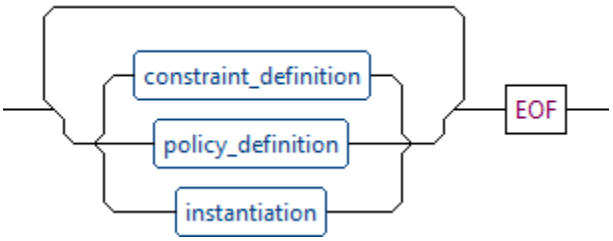
Das dritte Erweiterbare System in D° wird verwendet um Policies zu definieren. Policies sind dafür da, die Nutzungsbedingungen zu definieren, welche den sachgemäßen Umgang mit Daten gewährleisten.

`policies_file`

Notiz	Version
Überarbeitung und Erweiterung des Sprachkonstrukts, neue Abbildung, erweiterte Beschreibung, angepasstes Beispiel	2.0
Erste Veröffentlichung	1.0

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

1. `constraint_definition`: Bei der Definition von Policies setzt D° auf die Verwendung von feingranularen, atomaren Bedingungen, welche verbunden werden um komplexe Policies darzustellen. Constraints sind das Sprachkonstrukt zur Erstellung dieser atomaren Bedingungen
2. `policy_definition`: Zur Erstellung von komplexen Policies verwendet D° Container, welche selber keine Bedingungen enthalten, sondern andere Container und atomare Bedingungen enthalten. Die Policy ist dieser Container.
3. `instantiation`: Die Definitionen der einzelnen Sprachkonstrukte müssen vor der eigentlichen Verwendung instanziiert werden. Hierfür verwendet D° das generische Konzept der `instantiation`, welches verwendet wird, um verschiedene Sprachkonstrukte (bspw. Aktivitäten und Policies) zu instanziieren.



Eine Policy in D° ist somit eine Menge von Definitionen von Constraints und Policies, welche vor der Verwendung instanziiert werden müssen und anschließend an andere Sprachelemente geheftet werden können.

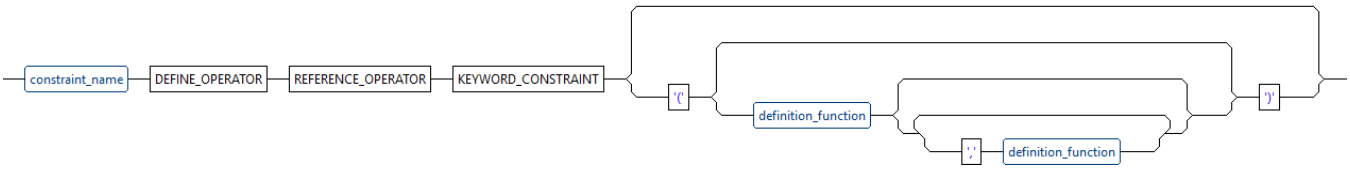
Beispiel

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

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

AllowedTimeInterval := $Policy(
    @dependency["useNotBefore", $UseNotBefore],
    @dependency["useNotAfter", $UseNotAfter]
)

UseNotAfter2020  = $UseNotAfter(
    @set["timestamp", /* VALUE */]
)

UseNotBefore2010  = $UseNotBefore(
    @set["timestamp", /* VALUE */]
)

AllowedTimeInterval2010to2020 = $AllowedTimeInterval(
    @set["useNotBefore", $UseNotBefore2010],
    @set["useNotAfter", $UseNotAfter2020]
)

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

`constraint_definition`

Notiz	Version
Neu hinzugefügt	2.0

In D° werden komplexe Policies durch die Kombination von einzelnen atomaren Bedingungen erzeugt. Zur Erzeugung dieser Bedingungen wird das Sprachkonstrukt `constraint_definition` verwendet. Neben der Definition in einem Sprachmodul muss zusätzlich noch eine Implementation mitgeliefert werden, welche die Logik zur Überprüfung/Durchsetzung enthält. Dieses Sprachkonstrukt kann nicht direkt verwendet werden, sondern muss vor der Nutzung instanziiert werden.



Die Verwendung dieses Sprachelements, sowie die einzelnen Elemente sind analog zur Definition anderer Sprachelemente, wie beispielsweise Aktivitäten.

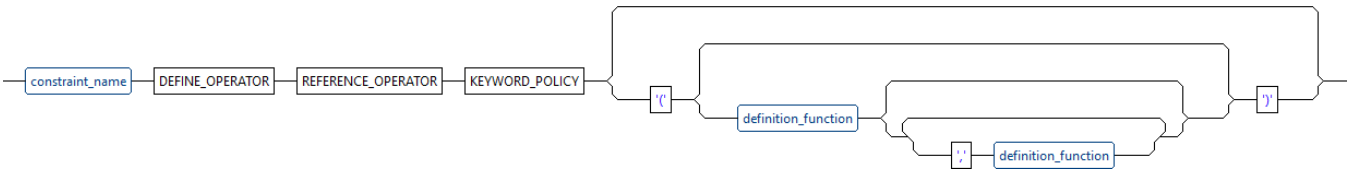
Beispiel

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

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

`policy_definition`

Notiz	Version
Neu hinzugefügt	2.0

In D° werden komplexe Policies durch die Kombination von einzelnen atomaren Bedingungen erzeugt. Zur Organisation und Zusammenführung dieser Bedingungen wird das Sprachkonstrukt `policy_definition` verwendet. Dabei enthält eine `policy_definition` eine Menge von `constraint_definition`, welche in ihrer Summe die Policy bilden.



Die Verwendung dieses Sprachelements, sowie die einzelnen Elemente sind analog zur Definition anderer Sprachelemente, wie beispielsweise Aktivitäten.

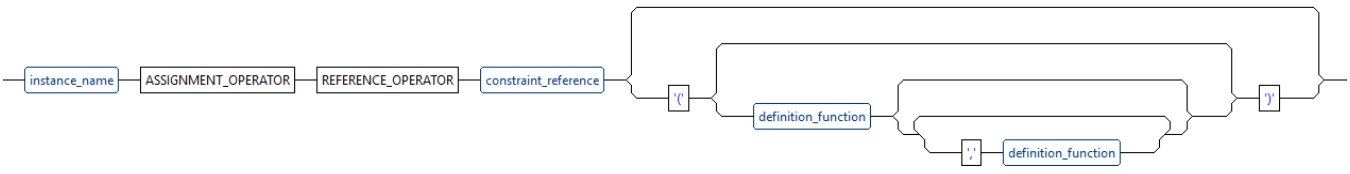
Beispiel

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

`instantiation`

Notiz	Version
Neu hinzugefügt	2.0

Die Definitionen der einzelnen Sprachelemente von D° sind nicht direkt nutzbar. Stattdessen müssen zuerst Instanzen aus den Objekten erzeugt werden, welche im Anschluss verwendet werden können. Die Instanziierung ist auch der Zeitpunkt, an dem ggf. vorhandene Parameter gesetzt/verbunden werden können. Das `instantiation`-Sprachkonstrukt ist generisch gestaltet und kann Instanziierungen für Aktivitäten, Policies und Constraints enthalten. Der Compiler und die Laufzeitumgebung müssen dies korrekt auflösen, was auch ein weiterer wichtiger Punkt ist, wieso verwendete Namen für Sprachelemente global einzigartig sein müssen.



Die Verwendung dieses Sprachelements, sowie die einzelnen Elemente sind analog zur Definition anderer Sprachelemente, wie beispielsweise Aktivitäten.

Beispiel

```
UseNotAfter2020  = $UseNotAfter(
    @set["timestamp", /* VALUE */]
)

UseNotBefore2010  = $UseNotBefore(
    @set["timestamp", /* VALUE */]
)

AllowedTimeInterval2010to2020 = $AllowedTimeInterval(
    @set["useNotBefore", $UseNotBefore2010],
    @set["useNotAfter", $UseNotAfter2020]
)

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

`validator_definition` - Entfernt

Entferntes Element

Das beschriebene Element ist nicht länger Bestandteil der Grammatik von D° und ist hier nur aus Gründen der Vollständigkeit und Nachvollziehbarkeit in diesem Dokument enthalten.

Grund der Entfernung: Entfällt durch vollständige Überarbeitung des Policy Konzepts.

Notiz	Version
Entfernt	2.0
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` - Entfernt

Entferntes Element

Das beschriebene Element ist nicht länger Bestandteil der Grammatik von D° und ist hier nur aus Gründen der Vollständigkeit und Nachvollziehbarkeit in diesem Dokument enthalten.

Grund der Entfernung: Entfällt durch vollständige Überarbeitung des Policy Konzepts.

Notiz	Version
Entfernt	2.0

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` - Entfernt

Entferntes Element

Das beschriebene Element ist nicht länger Bestandteil der Grammatik von D° und ist hier nur aus Gründen der Vollständigkeit und Nachvollziehbarkeit in diesem Dokument enthalten.

Grund der Entfernung: Entfällt durch vollständige Überarbeitung des Policy Konzepts

Notiz	Version
Entfernt	2.0
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` - Entfernt

Entferntes Element

Das beschriebene Element ist nicht länger Bestandteil der Grammatik von D° und ist hier nur aus Gründen der Vollständigkeit und Nachvollziehbarkeit in diesem Dokument enthalten.

Grund der Entfernung: Vereinheitlichung der Syntax über verschiedene Sprachkonstrukte hinweg.

Notiz	Version
Entfernt	2.0
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]
)
```

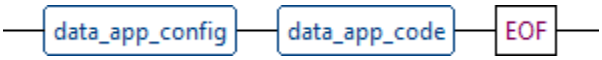
Data Apps

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
Neue Abbildung	2.0
Erste Veröffentlichung	1.0

Bei der Entwicklung einer Data App ist das Sprachkonstrukt `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.

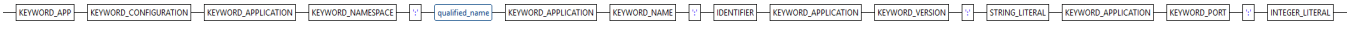
Beispiel

```
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_config`

Notiz	Version
Neue Abbildung, überarbeitete Beschreibung	2.0
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.

Die möglichen Konfigurationselemente sind fest in der Grammatik hinterlegt, was zu der oben sichtbaren, unübersichtlichen Regel führt. Sofern das `data_app_config`-Element langfristig erhalten bleibt, findet hier eine Überarbeitung statt, um das Konzept generischer zu gestalten.

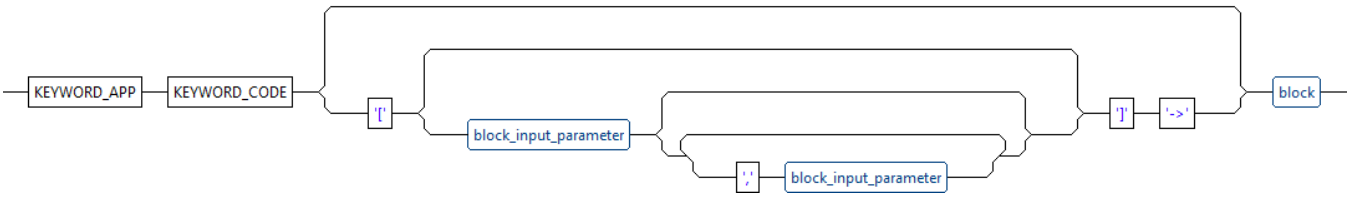
Beispiel

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

`data_app_code`

Notiz	Version
Neue Abbildung	2.0
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.

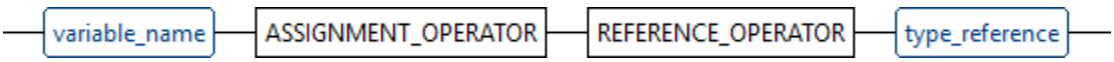
Beispiel

```
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
```

`block_input_parameter`

Notiz	Version
Neu hinzugefügt	2.0

Um definierte Eingaben für Data Apps zu erlauben, müssen diese im Programmcode auftauchen.



Die Angabe von Eingabeparametern für eine Data App ist optional und kann entfallen, wenn keine Eingaben vorgesehen sind.

Beispiel

```
App code
        [cart = $Warenkorb] -> 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
```

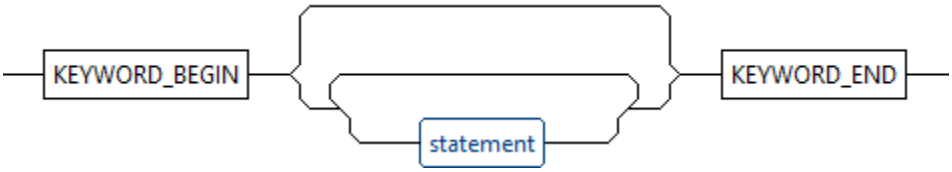
Datenfluss

D° enthällt verschiedene Konstrukte zur Erzeugung des Datenflusses in einer Data App.

`block`

Notiz	Version
Neue Abbildung	2.0
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.

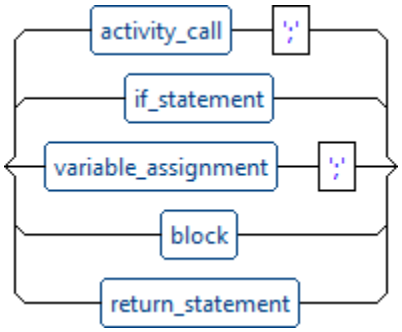
Beispiel

```
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
Neue Arten von Statements hinzugefügt, neue Abbildung, angepasste Beschreibung	2.0
Erste Veröffentlichung	1.0

Um Blöcke mit Funktionalität anzureichern werden Statements verwendet.



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

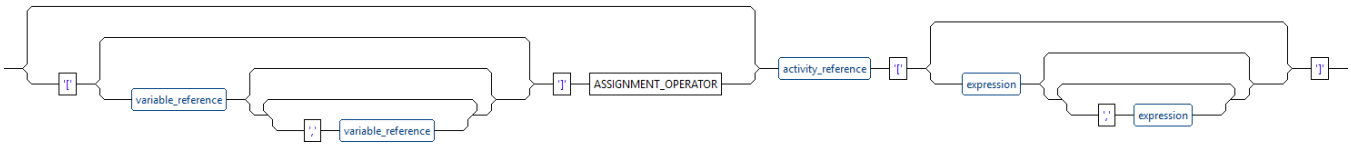
Beispiel

```
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
Neue Abbildung	2.0
Erste Veröffentlichung	1.0

Um die Funktionalität einer Aktivität, die in D° definiert wurde, auszulösen wird das Sprachkonstrukt `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.

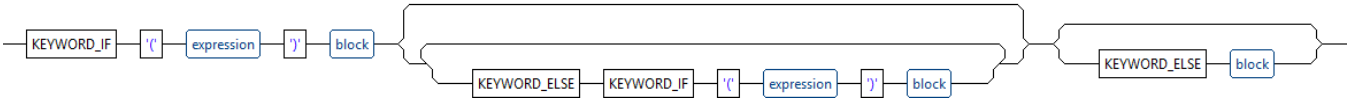
Beispiel

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

`if_statement`

Notiz	Version
Neu hinzugefügt	2.0

Es ist notwendig im Datenfluss von D° (bedingte) Verzweigungen verwenden zu können, welche in unterschiedlichen Ausführungspfaden resultieren können.

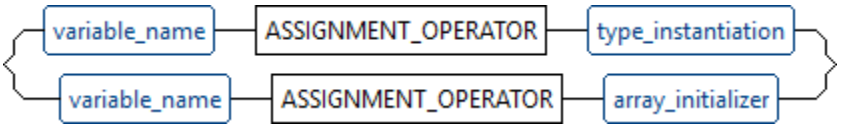


Das `if_statement` kapselt zwei `block`-Umgebungen mit einer Bedingung, deren Auswertung entscheidet welcher ausgeführt wird. Das Konzept ist Teil der Sprache, aber in der aktuellen Version noch nicht vollständig spezifiziert, weswegen die Anwendung nicht analog zu anderen Elementen des Datenflusses funktioniert. Aus diesem Grund ist an dieser Stelle kein Beispiel aufgeführt und von der Verwendung wird aktuell abgeraten.

`variable_assignment`

Notiz	Version
Neu hinzugefügt	2.0

Zur effizienten Verwendung von Variablen ist es notwendig, dass diese mit Werten belegt werden können und nicht nur die Rückgabewerte von Aktivitätsaufrufen abspeichern können. Dabei können die Werte, die einer Variablen zugewiesen werden aus unterschiedlichen Quellen stammen.



Es ist zu beachten, dass auf der rechten Seite des Operators kein Aufruf einer Aktivität sein darf. Der Fall, dass der Rückgabewert eines Aufrufs einer Aktivität in einer/mehreren Variable/n abgespeichert werden soll, ist im Sprachkonstrukt `activity_call` verankert.

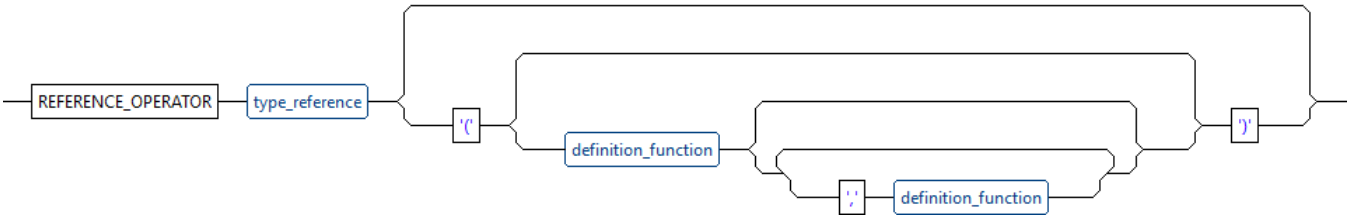
Beispiel

```
myVar = $Text(@write["Hello World!"]);
```

`type_instantiatiaon`

Notiz	Version
Neu hinzugefügt	2.0

Eine mögliche Wertequelle für die Belegung von Variablen ist die Instanziierung von Datentypen.

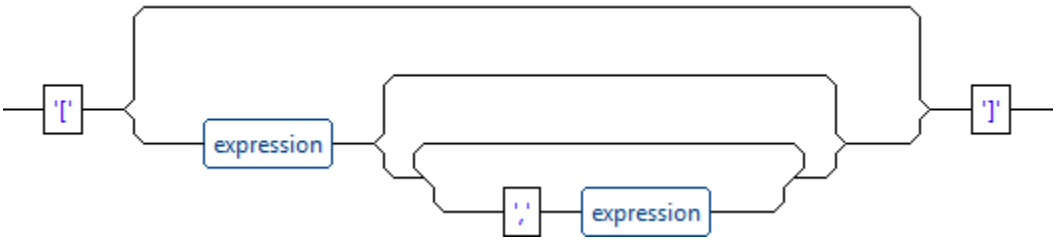


Dabei ist die Verwendung Analog zur Erzeugung von Instanzen für Aktivitäten und Policies aus ihren Definitionen.

`array_initializer`

Notiz	Version
Neu hinzugefügt	2.0

Neben einzelnen Werten, kann eine Variable auch ein Array enthalten.

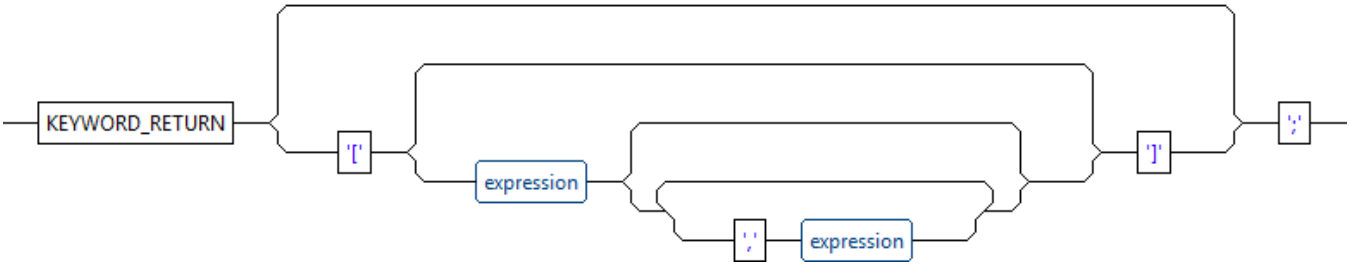


Dabei wird jedes Element des Arrays durch ein Sprachkonstrukt vom Typ `expression` definiert.

`return_statement`

Notiz	Version
Neu hinzugefügt	2.0

Zur expliziten Auszeichnung von Werten die an einen Aufrufer zurückgegeben werden sollen, wird ein spezielles `statement` benötigt.



Syntaktisch können `return_statements` überall verwendet werden, dürfen semantisch aber nur am Ende von zusammengesetzten Aktivitäten und am Ende von Data Apps verwendet werden.

Beispiel

```
return [myVar1, myVar2];

return [];

return;
```

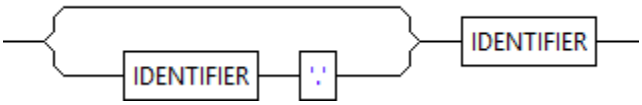
Common

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

`qualified_name`

Notiz	Version
Neue Abbildung, überarbeitete Beschreibung	2.0
Erste Veröffentlichung	1.0

Um verschiedene Elemente von D° (unter Anderem Datentypen und Aktivitäten) zu gruppieren und diese Zugehörigkeit sichtbar zu machen, werden Sprachelemente vom Typ `qualified_name` zur Benennung verwendet. Diese bestehen aus nicht-leeren Menge von `IDENTIFIER`n, wobei der letzte `IDENTIFIER` als Name für das jeweilige Element verwendet wird und die verbleibenden `IDENTIFIER` als Namensraum. Zwischen zwei `IDENTIFIER`n 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`').

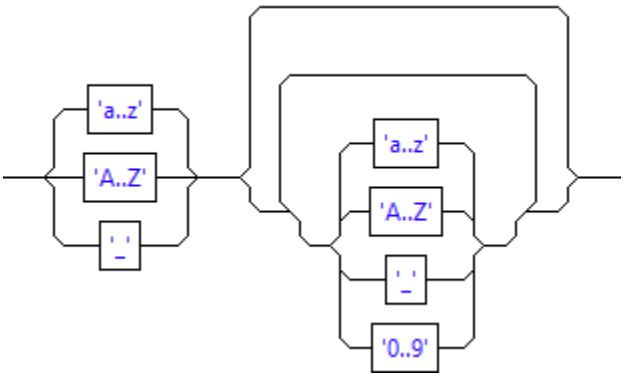
Beispiel

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

`IDENTIFIER`

Notiz	Version
Neue Abbildung, überarbeitete Beschreibung	2.0
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 oder Unterstrich beginnen muss.



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

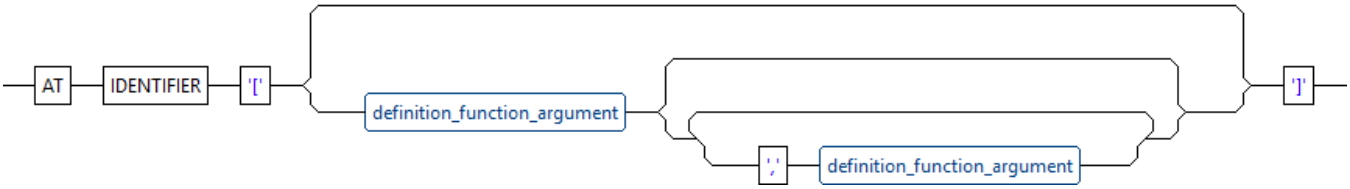
Beispiel

`__shipping`
`myCompany`
`AddressCheck`

`definition_function`

Notiz	Version
Neue Abbildung, überarbeitete Beschreibung	2.0
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 wird das Sprachelement `definition_function` 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.

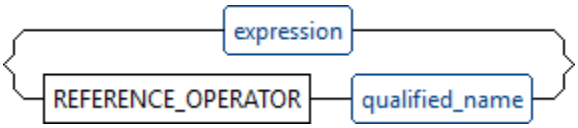
Beispiel

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

`definition_function_argument`

Notiz	Version
Neue Abbildung, überarbeitete Beschreibung	2.0
Erste Veröffentlichung	1.0

Die Argumente, welche an Sprachkonstrukte vom Typ `definition_function` 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.

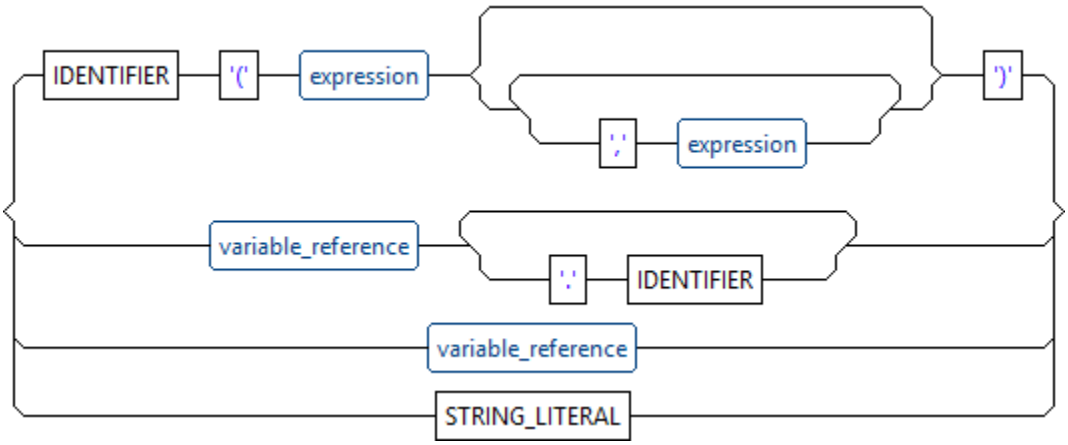
Beispiel

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

`expression`

Notiz	Version
Neu hinzugefügt	2.0

An verschiedenen Stellen in Data Apps ist es notwendig Ausdrücke verwenden zu können die (auf verschiedenen Wegen) zu einem Wert ausgewertet werden können. Zu diesem Zweck verwendet D° das `expression` Sprachkonstrukt.



Aktuell erlaubt die Grammatik die Verwendung von folgenden Ausdrücken:

- Strings
- Referenzen auf Variablen
- Referenzen auf einzelne Komponenten einer Variable
- Aufruf externer Funktionen mit zusätzlichen Parametern
    - Diese Art der Ausdrücke ist zwar Teil der Grammatik, ist aber nicht vollständig ausgearbeitet und wird in einer zukünftigen Version wieder entfernt oder überarbeitet. Von einer Verwendung wird aktuell abgeraten

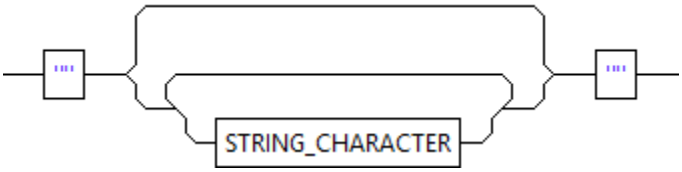
Beispiel

```
method()
myVar.member
myVar
"expression"
```

STRING_LITERAL

Notiz	Version
Neue Abbildung, überarbeitete Beschreibung	2.0
Erste Veröffentlichung	1.0

Um beliebige Zeichenketten an eine `definition_function` zu übergeben, wird das `STRING_LITERAL` verwendet.



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

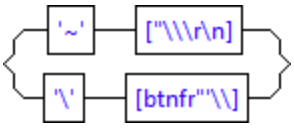
Beispiel

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

STRING_CHARACTER

Notiz	Version
Neue Abbildung, überarbeitete Beschreibung	2.0
Erste Veröffentlichung	1.0

Ein `STRING_LITERAL` besteht aus einer beliebigen Menge von `STRING_CHARACTER`, welche den Inhalt abbilden.



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

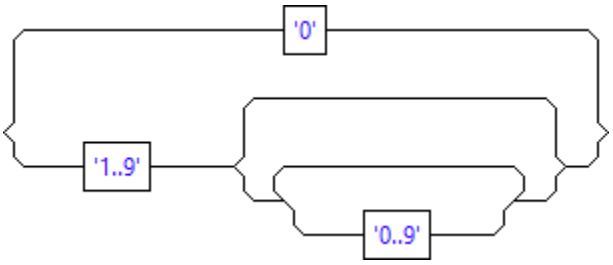
Beispiel

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

`INTEGER_LITERAL`

Notiz	Version
Neue Abbildung, überarbeitete Beschreibung	2.0
Erste Veröffentlichung	1.0

Um positive, ganzzahlige Werte als Argumente an eine `definition_function` zu übergeben, wird das `INTEGER_LITERAL` verwendet.



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

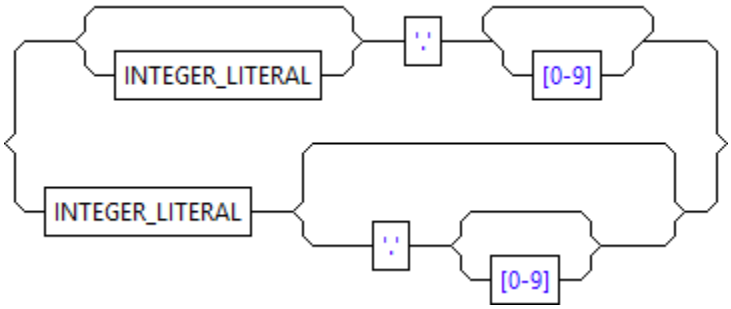
Beispiel

```
0
17
3871
```

`FLOATING_POINT_LITERAL`

Notiz	Version
Neu hinzugefügt	2.0

Um reelle Zahlen als Argumente an eine `definition_function` zu übergeben, wird das `FLOATING_POINT_LITERAL` verwendet.



Dabei ist jede Gleitkommazahl abbildbar; es gibt keine Ober- bzw. Untergrenze. Das Sprachkonstrukt ist bereits in der Grammatik enthalten, wird aktuell aber an keiner Stelle verwendet/unterstützt. Es ist als Vorbereitung für künftige Versionen von D° integriert worden.

Beispiel

```
0
17
3871
```

Zusätzliche Regeln

In diversen der aufgeführten Regeln wurden Elemente verwendet, die an dieser Stelle noch nicht erläutert wurden. Diese bisher unbekannten Regeln existieren nur aus Gründen der einfachen Grammatikverarbeitung und werden intern auf `qualified_name` und `IDENTIFIER` abgebildet.

Regeln die auf `qualified_name` abgebildet werden: `activity_name, activity_reference, constraint_name, constraint_reference, instance_name, type_name, type_reference`

Regeln die auf `IDENTIFIER` abgebildet werden: `variable_name, variable_reference`