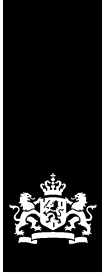


Ministerie van Binnenlandse Zaken en Koninkrijksrelaties

Technisch ontwerp BRP Expressietaal

1.1

Datum 30-03-2017
Status Definitief

Inhoudsopgave

Documenthistorie

Versiehistorie

Datum	Versie	Omschrijving	Auteur
10-01-2017	1.0	1e Versie BRP Expressietaal	Operatie BRP
30-03-2017	1.1	Versie na review commentaar I&T	Operatie BRP

Reviewhistorie

Versienummer	**Datum**	**Namen**
1.0	7-03-2017	Operatie BRP
1.0	30-03-2017	Operatie BRP

1 Inleiding

De BRP-expressietaal (of verkort expressietaal) is een taal om expressies te beschrijven die gaan over gegevens uit de BasisRegistratie Personen. Met de taal kan bijvoorbeeld uitgedrukt worden waar een populatie van personen aan voldoet, zoals "alle vrouwen", "iedereen ouder dan 18 jaar" of "iedereen die woont in gemeente Neerijnen". Met vergelijkbare expressies kan worden vastgesteld of gegevens van een persoon gewijzigd zijn, wat kan leiden tot het sturen van wijzigingen aan afnemers (het zogenaamde 'attenderen').

Dit document is bedoeld voor ontwikkelaars van de taal. In hoofdlijnen worden de volgende onderwerpen behandeld:

• Hoofdstuk 2, "Definitie van een expressie" beschrijft wat een expressie precies is en aan welke voorwaarden het moet voldoen.
• Hoofdstuk 3, "Parsen van de expressie" beschrijft hoe de expressie in String formaat geïnterpreteerd en vertaald wordt naar een Java Expressie Object structuur.
• Hoofdstuk 4 "Evalueren van een expressie" beschrijft hoe het evalueren van expressie werkt. Het beschrijft de belangrijkste interfaces en implementaties van de Java Expressie Object structuur, de typering van expressie en hoe vergelijkingen en berekeningen gedaan worden.
• Hoofdstuk 5 "Foutafhandeling" beschrijft de foutsituaties die kunnen optreden tijdens het parsen en evalueren.
• Hoofdstuk 6 "Expressies in de BRP" beschrijft een aantal specifieke onderwerpen, met betrekking tot gebruik van de expressietaal.

1.1 Referenties

Nr.	Documentnaam	Locatie
1	ANTLR Documentatie	www.antlr.org

2 Definitie van een expressie

Expressies worden gedefinieerd als een enkele String waarde en moeten voldoen aan een strikte syntax om het te kunnen interpreteren.

2.1 Eisen aan de taal

Voor het ontwerp en de implementatie van de expressietaal zijn de volgende eisen meegenomen:

- De expressietaal moet leesbare en begrijpelijke expressies opleveren.
- Expressies mogen alleen concrete antwoorden geven als dat op basis van de brongegevens gerechtvaardigd is.
- Concepten in de taal moeten zoveel mogelijk Nederlands zijn, met eventuele uitzonderingen voor specifiek (Engelstalig) jargon.
- De implementatie moet, waar mogelijk, ontkoppeld zijn van andere onderdelen van de BRP.
- De implementatie moet efficiënt zijn. Het bepalen van gegevens van een persoon, bijvoorbeeld, moet 'snel' uitgevoerd kunnen worden.
- Fouten in expressies moeten zo vroeg mogelijk gevonden worden.

2.2 Syntax

De syntax definieert alle mogelijke logische expressies, functies, operatoren en literals. De syntax heeft geen inhoudelijke kennis van de gegevensset in de BRP (d.w.z. het kent geen adressen, indicaties, geboortegegevens etc.). Deze ontkoppeling heeft als voordeel dat mocht de gegevensset van de BRP uitgebreid worden met nieuwe persoonsgegevens, dan hoeft de taal niet aangepast te worden. Het gebruik van persoonsgegevens in de expressietaal is beschreven in Section 4.2.6,“Persoonsgegevens”.

De syntax is gedefinieerd in Backus-Naur-vorm [https://nl.wikipedia.org/wiki/Backus-Naur-vorm]; een algemene notatie voor het definiëren van de syntax/grammatica van een taal. De syntax de BRP Expressietaal is te vinden in het bestand BRPExpressietaal.g4:

```
grammar BRPExpressietaal;
@header {
package nl.bzk.brp.domain.expressie.parser.antlr;
}
brp_expressie :         exp ;
exp :                   closure ;
closure :               booleanExp assignments? ;
assignments :           OP_WAARBIJ assignment (COMMA assignment)* ;
assignment :            variable OP_EQUAL exp ;
booleanExp :            booleanTerm (OP_OR booleanExp)? ;
booleanTerm :           equalityExpression (OP_AND booleanTerm)? ;
equalityExpression :    relationalExpression (equalityOp
relationalExpression)? ;
equalityOp :            OP_EQUAL
    |                   OP_NOT_EQUAL
    |                   OP_LIKE
```

```
    ;
relationalExpression :  ordinalExpression (relationalOp
ordinalExpression)? ;
relationalOp :          OP_LESS
    |                   OP_GREATER
    |                   OP_LESS_EQUAL
    |                   OP_GREATER_EQUAL
    |                   OP_AIN
    |                   OP_AIN_WILDCARD
    |                   OP_EIN
    |                   OP_EIN_WILDCARD
    |                   collectionEOp
    |                   collectionAOp
    ;
collectionEOp :         EOP_EQUAL
    |                   EOP_NOT_EQUAL
    |                   EOP_LESS
    |                   EOP_GREATER
    |                   EOP_LESS_EQUAL
    |                   EOP_GREATER_EQUAL
    |                   EOP_LIKE
    ;
collectionAOp :         AOP_EQUAL
    |                   AOP_NOT_EQUAL
    |                   AOP_LESS
    |                   AOP_GREATER
    |                   AOP_LESS_EQUAL
    |                   AOP_GREATER_EQUAL
    |                   AOP_LIKE
    ;
ordinalExpression :     negatableExpression (ordinalOp
ordinalExpression)? ;
ordinalOp :             OP_PLUS | OP_MINUS ;
negatableExpression :   negationOperator? unaryExpression ;
negationOperator :      OP_MINUS | OP_NOT ;
unaryExpression :       expressionList
    |                   function
    |                   existFunction
    |                   element
    |                   literal
    |                   variable
    |                   bracketedExp
    ;
bracketedExp :          LP exp RP;
expressionList :        emptyList
    |                   nonEmptyList ;
emptyList :             LL RL ;
nonEmptyList :          LL exp (COMMA exp)* RL ;
element :               element_path ;
variable :              IDENTIFIER ;
function :              functionName LP ( exp (COMMA exp)* )? RP ;
functionName :          'IS_NULL'
    |                   'DATUM'
    |                   'VANDAAG'
    |                   'DAG'
    |                   'MAAND'
    |                   'JAAR'
    |                   'AANTAL_DAGEN'
```

```
    |                   'LAATSTE_DAG'
    |                   'AANTAL'
    |                   'ALS'
    |                   'HISM'
    |                   'HISM_LAATSTE'
    |                   'HISF'
    |                   'GEWIJZIGD'
    |                   'KV'
    |                   'KNV'
    |                   'ACTIE'
    |                   'AH'
    ;
existFunction :         existFunctionName LP exp COMMA variable COMMA exp
RP ;
existFunctionName :     'ER_IS'
    |                   'ALLE'
    |                   'FILTER'
    |                   'MAP'
    ;
literal :               stringLiteral
    |                   booleanLiteral
    |                   numericLiteral
    |                   dateLiteral
    |                   dateTimeLiteral
    |                   periodLiteral
    |                   element
    |                   elementCodeLiteral
    |                   nullLiteral
    ;
stringLiteral :         STRING ;
booleanLiteral :        TRUE_CONSTANT | FALSE_CONSTANT ;
numericLiteral :        INTEGER | '-' INTEGER ;
dateTimeLiteral :       year '/' month '/' day '/' hour '/' minute '/'
second ;
dateLiteral :           year '/' month '/' day
    ;
year :                  numericLiteral | UNKNOWN_VALUE;
month :                 numericLiteral | UNKNOWN_VALUE
    |                   monthName ;
monthName:              MAAND_JAN
    |                   MAAND_FEB
    |                   MAAND_MRT
    |                   MAAND_APR
    |                   MAAND_MEI
    |                   MAAND_JUN
    |                   MAAND_JUL
    |                   MAAND_AUG
    |                   MAAND_SEP
    |                   MAAND_OKT
    |                   MAAND_NOV
    |                   MAAND_DEC
    ;
day :                   numericLiteral | UNKNOWN_VALUE;
hour :                  numericLiteral ;
minute :                numericLiteral ;
second :                numericLiteral ;
periodLiteral :         '^' relativeYear '/' relativeMonth '/'
relativeDay ;
```

```
relativeYear :          periodPart | UNKNOWN_VALUE;
relativeMonth :         periodPart | UNKNOWN_VALUE;
relativeDay :           periodPart | UNKNOWN_VALUE;
periodPart :            ('-'|) numericLiteral ;
elementCodeLiteral:   '[' element_path ']' ;
nullLiteral :           NULL_CONSTANT ;
element_path :        IDENTIFIER | IDENTIFIER (DOT IDENTIFIER)+ ;
STRING :                '"' (.|' ')*? '"';
INTEGER :               [0-9]+ ;
WS :                    [ \t\r\n]+ -> skip ;
LP :                    '(' ;
RP :                    ')' ;
LL :                    '{' ;
RL :                    '}' ;
COMMA :                 ',' ;
DOT :                   '.' ;
UNKNOWN_VALUE :         '?' ;
EOP_EQUAL :             'E=' ;
AOP_EQUAL :             'A=' ;
OP_EQUAL :              '=' ;
EOP_NOT_EQUAL :         'E<>' ;
AOP_NOT_EQUAL :         'A<>' ;
OP_NOT_EQUAL :          '<>' ;
EOP_LIKE :              'E=%' ;
AOP_LIKE :              'A=%' ;
OP_LIKE :               '=%' ;
EOP_LESS :              'E<' ;
AOP_LESS :              'A<' ;
OP_LESS :               '<' ;
EOP_GREATER :           'E>' ;
AOP_GREATER :           'A>' ;
OP_GREATER :            '>' ;
EOP_LESS_EQUAL :        'E<=' ;
AOP_LESS_EQUAL :        'A<=' ;
OP_LESS_EQUAL :         '<=' ;
EOP_GREATER_EQUAL :     'E>=' ;
AOP_GREATER_EQUAL :     'A>=' ;
OP_GREATER_EQUAL :      '>=' ;
OP_AIN_WILDCARD :       'AIN%' ;
OP_AIN :                'AIN' ;
OP_EIN_WILDCARD :       'EIN%' ;
OP_EIN :                'EIN' ;
OP_PLUS :               '+' ;
OP_MINUS :              '-' ;
OP_OR :                 'OF' ;
OP_AND :                'EN' ;
OP_NOT :                'NIET' ;
OP_REF :                '$' ;
OP_WAARBIJ :            'WAARBIJ' ;
TRUE_CONSTANT :         'WAAR' | 'TRUE' ;
FALSE_CONSTANT :        'ONWAAR' | 'FALSE' ;
NULL_CONSTANT :         'NULL' ;
MAAND_JAN :             ('JAN' | 'JANUARI') ;
MAAND_FEB :             ('FEB' | 'FEBRUARI') ;
MAAND_MRT :             ('MRT' | 'MAART');
MAAND_APR :             ('APR' | 'APRIL');
MAAND_MEI :             ('MEI' ) ;
MAAND_JUN :             ('JUNI' | 'JUN');
```

```
MAAND_JUL :             ('JULI' | 'JUL');
MAAND_AUG :             ('AUGUSTUS' | 'AUG');
MAAND_SEP :             ('SEPTEMBER' | 'SEP');
MAAND_OKT :             ('OKTOBER' | 'OKT');
MAAND_NOV :             ('NOVEMBER' | 'NOV');
MAAND_DEC :             ('DECEMBER' | 'DEC');
IDENTIFIER :            [a-zA-Z][a-zA-Z0-9_]* ;
```

3 Parsen van de expressie

Een expressie in String formaat kan niet direct geëvalueerd worden. Er moet een vertaling gemaakt worden naar een Java Object. Deze vertaalslag noemen we het parsen en deze bestaat uit twee stappen:

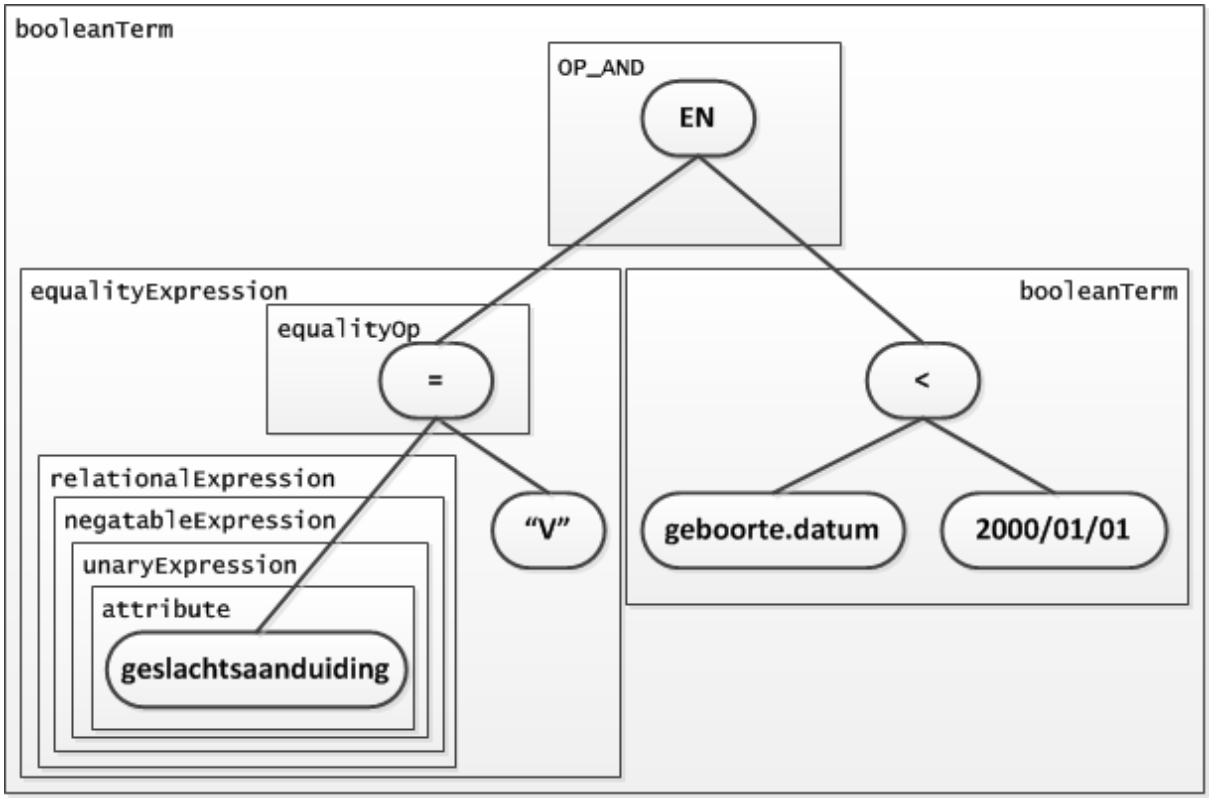
1. Het vertalen van de expressie-String naar een parsetree.
2. Het vertalen van de parsetree naar een Java Objectstructuur.

De volgende paragrafen gaan hier verder op in.

3.1 **De Parsetree**

Allereerst dient er een zogenaamde Parsetree [https://nl.wikipedia.org/wiki/Syntaxisboom] gemaakt te worden van de expressiestring. Dit is een boomstructuur die als tussenstap gebruikt wordt bij het vertaling naar een datastructuur.

Onderstaand een voorbeeld van de parsetree van de expressie: geslachtsaanduiding = "V" EN geboorte.datum < 2000/0101



3.1.1 *Genereren van de parsercode*

Voor het interpreteren van de syntax het genereren van de parsetree en de vertaling naar een datastructuur wordt gebruik gemaakt van ANTLR (versie 4, www.antlr.org).

ANTLR genereert een Parser, een Lexer en een Visitor welke specifiek zijn voor de BRP. Deze generatie gebeurt development-time en is enkel nodig als de syntax wijzigt (dus niet als er een nieuw element in BRP wordt toegevoegd!).

Het genereren van de parser wordt gedaan door een Maven plugin in de module brp-expressietaal. Om te voorkomen dat generatie standaard gebeurt is profiel genereer-parser) nodig. Het volgende commando genereert de code in de package nl.bzk.brp.domain.expressie.parser.antlr

mvn install -Pgenereer-parser

3.2 De Expressietree

Het verschil tussen de parsetree en de expressietree, is dat de parsetree een 'opgeknipt' stuk tekst is en nog geen notie van expressies heeft. In de expressietree is de oorspronkelijke tekst vervangen door 'echte' objecten die voldoen aan de Expressie interface.

Uiteindelijk vormt zich een boomstructuur met leaf objecten voor String-, Getal- en Datumwaarden (de literals, ofwel de constanten) en parent objecten, de operatoren (expressies die bewerkingen kunnen uitvoeren wat resulteert in een nieuwe expressie).

3.2.1 *Parse-visit*

Het aflopen van de parsetree door ANTLR resulteert in callbacks in de visitor implementatie `ExpressieVisitor`. In deze visit methode dient aan de hand van de parsecontext een implementatie van `Expressie` gemaakt te worden.

Aan de hand van onderstaande expressie zullen we de werking van de parser-visit ontleden:

```
geslachtsaanduiding = "V" EN geboorte.datum < 2000/01/01
```

Dit resulteert in de volgende aanroepen en expressieobjecten:

geslachtsaanduiding resulteert in de callback:

```
public Expressie visitElement(
final BRPExpressietaalParser.ElementContext ctx) {
//maakt een ElementExpressie welke het attribuut
//geslachtsaanduiding kan vinden op de persoonslijst
}
```

"V" resulteert in de callback:

```
public Expressie visitStringLiteral(
final BRPExpressietaalParser.StringLiteralContext ctx) {
//maakt een StringLiteral expressie
}
```

= resulteert in de callback:

```
public Expressie visitEqualityExpression(
final BRPExpressietaalParser.EqualityExpressionContext ctx) {
//maakt een VergelijkingOperator expressie voor de
//ElementExpressie(geslachtsaanduiding)
//en de StringLiteral("V").
```

```
}
```

geboorte.datum resulteert in de callback:

```
public Expressie visitElement(
final BRPExpressietaalParser.ElementContext ctx) {
//maakt een ElementExpressie die geboorte.datum kan vinden
//op de persoonslijst
}
```

2000/01/01 resulteert in de callback:

```
public Expressie visitDateLiteral(
final BRPExpressietaalParser.DateLiteralContext ctx) {
//maakt een DatumLiteral expressie
}
```

< resulteert in de callback:

```
public Expressie visitRelationalExpression(
final BRPExpressietaalParser.RelationalExpressionContext ctx) {
//maakt een VergelijkingOperator expressie voor het vergelijken van
//het resultaat van de ElementExpressie(geboorte.datum)
//en de DatumLiteral(2000/01/01)
}
```

EN resulteert in de callback:

```
public Expressie visitBooleanExp(
final BRPExpressietaalParser.BooleanExpContext ctx) {
//maakt een EnOperator expressie voor de VergelijkingOperator(=)
//en de VergelijkingOperator(<)
}
```

3.3 De Parsecontext

Bij het parsen van een expressie wordt gebruik gemaakt van een *Context* object. Deze context is nodig voor het kunnen parsen van expressies die afhankelijk zijn van variabelen die pas evaluatie-tijd beschikbaar komen. Om de expressie dan toch te kunnen parsen dient op de context een variabele gedeclareerd te worden tezamen met het *ExpressieType*.

Expressies in de BRP zijn grofweg in te delen in twee smaken, *'normale'* en *attendering* expressies. Voor beide expressies geldt dat ze afhankelijk zijn van persoonsgegevens. Naar deze persoonsgegevens wordt verwezen middels variabelen welke door de expressietaal-code reeds correct gedefinieerd zijn.

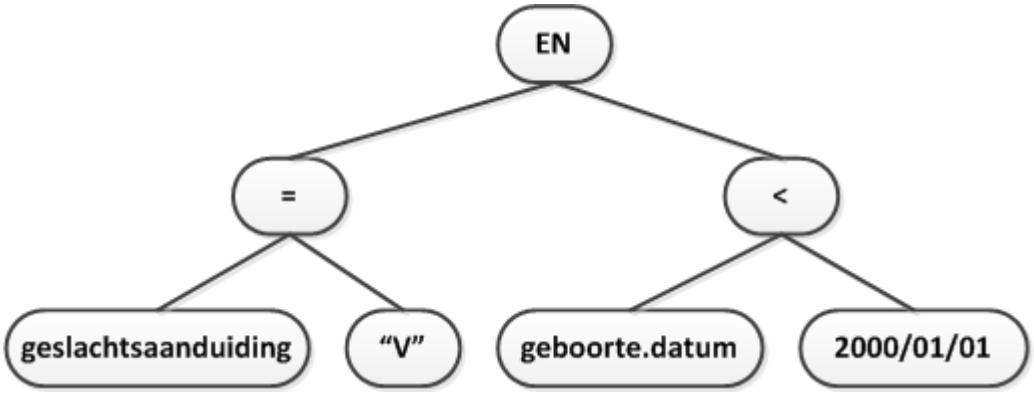
Voor *'normale'* expressies bestaat er daarom de `persoon` variabele. Deze variabele hoeft niet expliciet in de expressie gebruikt te worden als een persoonsgegeven wordt aangewezen, maar het mag wel. Indien er geen variabele opgegeven is valt het systeem terug op deze impliciete variabele. Bijvoorbeeld, de expressie `persoon.BSN` definieert de variabele expliciet, waarbij de expressie `BSN` dat niet doet.

Voor attendering expressies als `GEWIJZIGD(oud, nieuw)` wordt gebruik gemaakt van twee persoonslijsten, De variabele 'oud' wijst naar het oude persoonsbeeld en de variabele *'nieuw'* wijst naar het nieuwe persoonsbeeld. In tegenstelling tot de persoon variabele dienen deze variabelen altijd expliciet gebruikt te worden. De variabelen worden automatisch gedefinieerd in

de *parse* en `evalueerAttenderingsCriterium` methoden in BRPExpressies. Zie de paragraaf BRPExpressies API.

4 Evalueren van een expressie

Het evalueren van de expressie wordt simpelweg gedaan door het aanroepen van de evalueer methode op het Expressie object. Evaluatie is recursief, onderliggende expressies worden eerst geëvalueerd. Hierbij wordt ook een Context object doorgegeven welke gebruikt wordt voor het resolven van variabelen, of het definiëren van een tijdelijke scope.



De evaluatievolgorde voor bovenstaand voorbeeld is:

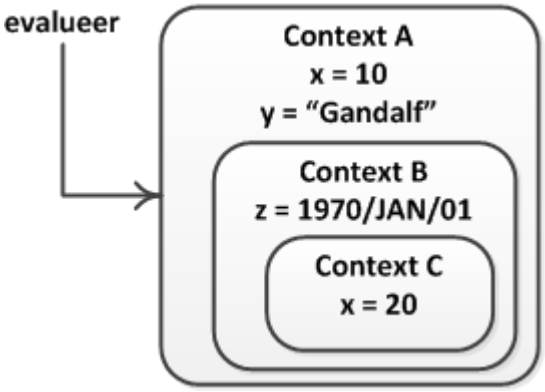
client evalueert EN
EN evalueert =
= evalueert geslachtsaanduiding; resultaat is een `LijstExpressie`
= evalueert "V"; resultaat is een `StringLiteral`
= evaluatie klaar; resultaat is `BooleanLiteral`
EN evalueert <
< evalueert `geboorte.datum`; resultaat is een `LijstExpressie`
< evalueert `2000/01/01`; resultaat is een `DatumLiteral`
< evaluatie klaar; resultaat is een `BooleanLiteral`
EN evaluatie klaar; resultaat is `BooleanLiteral`

4.1 Context

De evaluatie van een expressie vindt plaats in een bepaalde context; gedefinieerde symbolische namen met hun waarde. Een aantal variabelen worden gedefinieerd door de expressietaal zelf; de variabelen persoon, oud, nieuw. Andere variabelen worden gedefinieerd door functies of closures.

Een context kan ook een andere context omvatten, er vormt zich dan een context hiërarchie. Het maken van een nieuwe context resulteert in een tijdelijke scope waarin child-expressies evalueren. Bijvoorbeeld, de MAPFunctie itereert over de waarden van een lijst en voert hier vervolgens een expressie op uit. Om dit te doen wordt een nieuwe context gemaakt, waarbij een nieuwe contextvariabele de actuele iteratiewaarde bevat. Na evaluatie van de functie wordt de context tezamen met de geïntroduceerde variabelen weggegooid.

Als een identifier wordt opgezocht in een context, kijkt de context eerst in de eigen collectie identifiers; indien de naam daar niet gevonden wordt, wordt de vraag doorgespeeld aan de omliggende context enzovoort.

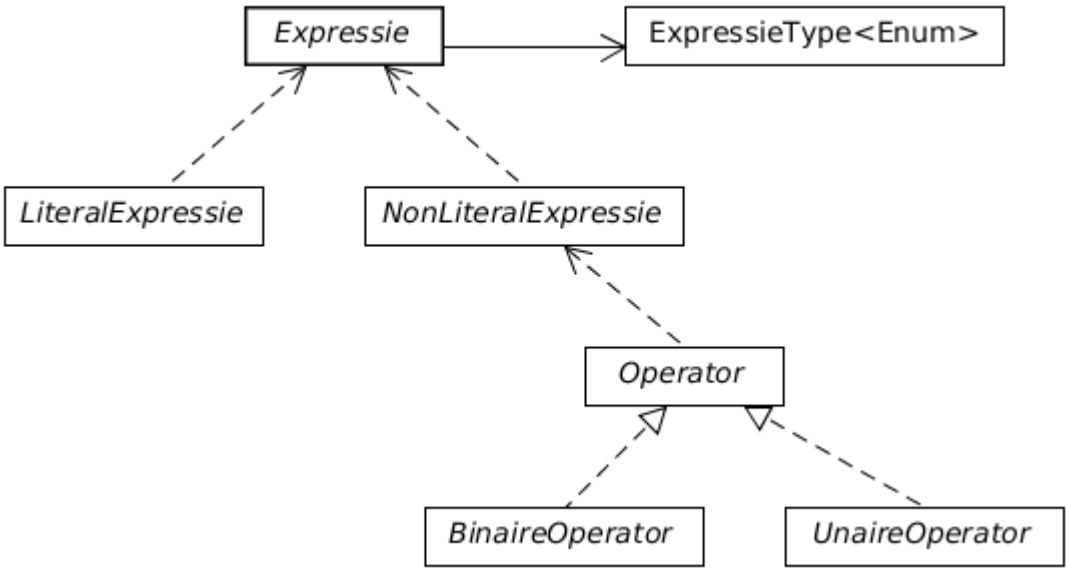


In het bovenstaande voorbeeld zijn drie contexten aan elkaar gekoppeld. Een verwijzing naar x in een expressie levert 10, een verwijzing naar y levert "Gandalf" en een verwijzing z levert 1970/JAN/01.

Een expressie waarbij de bovenstaande (hypothetische) situatie kan ontstaan, is: (((x WAARBIJ x = 10, y = "Gandalf") WAARBIJ z=1970/JAN/01) WAARBIJ x = 20). Deze expressie levert als resultaat dus 10.

4.2 Expressie implementatie

Het resultaat van het parse proces is één root expressie Object dat conformeert aan de interface `nl.bzk.brp.domain.expressie.Expressie`. Dit hoofdstuk behandelt de belangrijkste implementaties.



4.2.1 *Expressie superinterface*

De interface Expressie is de superinterface voor alle expressies. De belangrijkste kenmerken van expressies zijn dat ze een ExpressieType hebben en geëvalueerd kunnen worden. Implementaties zijn herbruikbaar en threadsafe.

De methode `Expressie evalueer(Context context);` Evalueert de expressie gegeven een context van identifiers. In de regel wordt de expressie berekend voor een bepaald BRP-object. Dit object is via de context beschikbaar; standaard is dat "persoon", een identifier die verwijst naar de persoon waarvoor de expressie geëvalueerd wordt).

De methode `ExpressieType getType(Context context);` Geeft het type van de Expressie, zo precies mogelijk. Voor de meeste expressies is er een intrinsiek type bekend (boolean operator, optellen, aftrekken, enzovoort), maar voor symbolische waarden (variabelen en attributen) is dat niet altijd zo. In dat laatste geval is het type *UNKNOWN*. In die laatste gevallen kan het type pas definitief bepaald worden tijdens evaluatie.

4.2.2 *Literals*

De `nl.bzk.brp.domain.expressie.literal.Literal` interface is de superinterface voor alle constante waarden in een expressie. Dit zijn de leaf-objecten in de expressietree. Met constant wordt bedoeld dat het parsetijd al vast staat wat de waarde is van de expressie. Het evalueren van de
expressie geeft een referentie naar zichzelf:

```
@Override
default Expressie evalueer(final Context context) {
// De evaluatie van een constante is - per definitie - gelijk aan
zichzelf.
return this;
}
```

De Literal implementaties staan in `nl.bzk.brp.domain.expressie.literal`:

- `BooleanLiteral`; bevat de waarden `true` of `false`.
- `DatumLiteral`; bevat datum (deels onbekend) bijvoorbeeld, `2010/04/01` of `2012/00/00`.
- `DatumtijdLiteral`; bevat een volledig bekende datum met tijd `2010/04/01/11/30/30`.
- `GetalLiteral` bevat een nummer van het type `Long;`
- `StringLiteral` bevat een `String` waarde.
- `ElementnaamLiteral`; betreft een element notatie, bijvoorbeeld [`Persoon.Geboorte.GeboorteDatum`].
- `MetaObjectLiteral`; bevat een `MetaObject`.
- `MetaGroepLiteral`; bevat een `MetaGroep`.
- `MetaRecordLiteral`; bevat een `MetaRecord`.
- `ActieLiteral`; bevat een `Actie`.
- `PeriodeLiteral`; bevat een periode aanduiding, wat handig is bij het rekenen met datums.
- `NullLiteral`; representatie van iets dat niet bestaat, bijvoorbeeld een ontbrekend attribuut op de persoonslijst.

4.2.3 *Operatoren*

De `Operator` is de interface van alle operator expressies en heeft als supertype `NonLiteralExpressie`. `Operator-expressies` doen een bewerking op één of meerdere expressies wat resulteert in een nieuwe expressie. Bijvoorbeeld, 1 + 1 evalueert tot 2 en 1 < 2 evalueert tot true. De operatoren zijn zelf weer onderverdeeld in twee categorieën, de unaire operatoren en de binaire operatoren.

4.2.3.1 Unaire operator

Unaire operatoren werken met één operand en hebben als supertype de abstractie nl.bzk.brp.domain.expressie.operator.AbstractUnaireOperator.

Er bestaan twee implementaties:

- LogischeInverseOperator; voor het inverteren van een boolean expressie, bijvoorbeeld !true.
- NumeriekeInverseOperator; voor het inverteren van de signed bit van een getal.

4.2.3.2 Binaire operator

Binaire operatoren werken met met twee operands en hebben als supertype de abstractie `nl.bzk.brp.domain.expressie.operator.AbstractBinaireOperator`.

Er bestaan meerdere implementaties:

- `Rekenoperator`; voor het uitvoeren van berekeningen als plus, minus. Maakt gebruikt van de speciale berekenfuncties.
- `VergelijkingOperator`; voor het uitvoeren van een vergelijkingen; gelijk, ongelijk, klein, kleinergelijk, groter, grotergelijk, wildcard-gelijk. Maakt gebruikt van de speciale vergelijkfunctie.
- `EnOperator`; voor het uitvoeren van een logische EN.
- `OfOperator`; voor het uitvoeren van een logische OF.
- `EAOperator`; voor vergelijkingen met lijsten, bijvoorbeeld alle waarden in de lijst zijn kleiner dan X, of er is een.
- waarde in de lijst groter dan Y. Maakt gebruikt van de speciale vergelijkfunctie;
- `EAINOperator`; voor gelijk-vergelijkingen met lijsten, bijvoorbeeld elke waarde uit de linkercollectie moet voorkomen in de rechtercollectie, of er moet minimaal één waarde uit de linkercollectie moet voorkomen in de rechtercollectie. Maakt gebruikt van de speciale vergelijkfunctie.
- `EAINWildcardOperator`; voor wildcard-vergelijkingen met lijsten, vb voor elke collectiewaarde van het linkeroperand moet er een match zijn met een wildcard-expressie uit het rechteroperand. vb voor minimaal één collectiewaarde van het linkeroperand moet er een match zijn met een wildcard-expressie uit het rechteroperand. Maakt gebruikt van de speciale vergelijkfunctie.

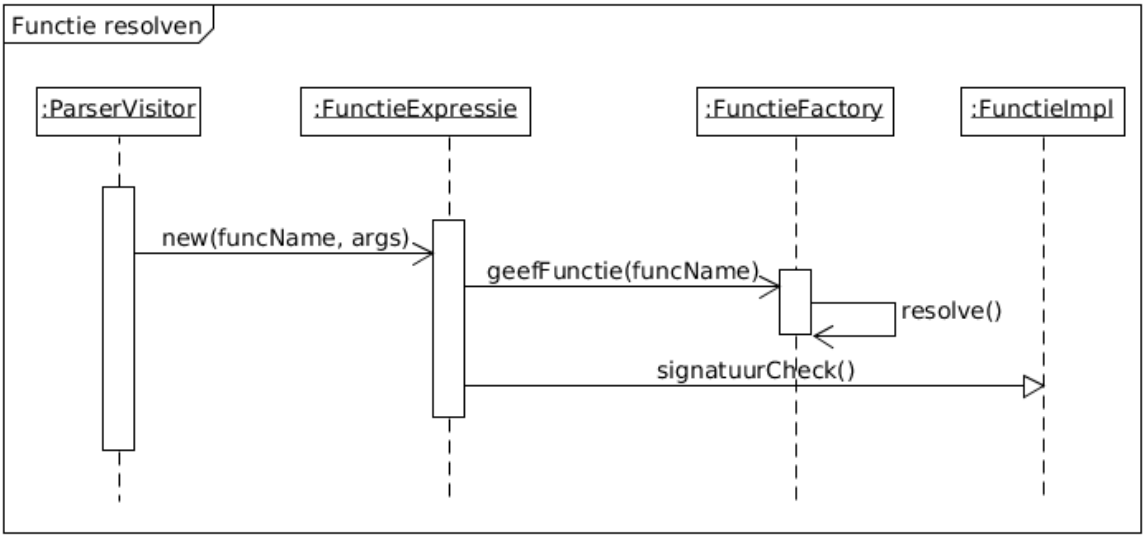
4.2.4 *Functies*

De `FunctieExpressie` maakt het mogelijk om functie constructies te gebruiken in de expressietaal. De namen van de functies zijn hard vastgelegd in de syntax, het aantal argumenten en het type van de argumenten echter niet. Bij het parsen dient deze controle alsnog gedaan te worden middels een `Signatuur` object, welke gebonden is aan de functie.

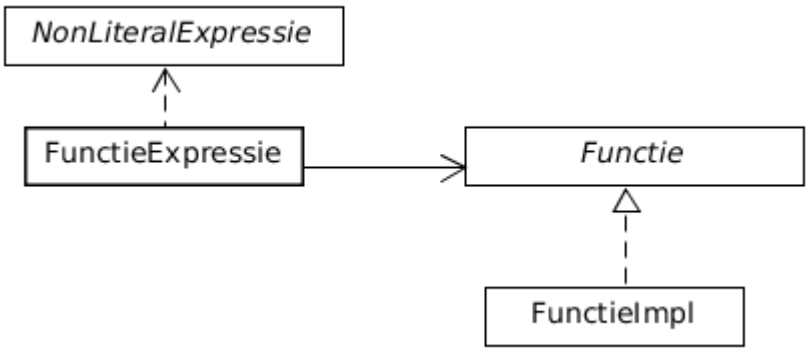
Het niet vastleggen van het aantal argumenten en in minder mate de typering van de argumenten in de expressietaalsyntax is een designkeuze geweest met als rationale de syntax-definitie simpel te houden.

De `FunctieExpressie` maakt gebruikt van de `FunctieFactory` voor het vinden van `Functie`
implementaties. Dit gebeurt parsetijd bij het instantiëren van de `FunctieExpressie`. De `FunctieFactory` scant eenmalig (in de package `nl.bzk.brp.domain.expressie.functie`) naar implementaties van Functie met `@FunctieKeyword` annotatie.

Van belang is hierbij dat de naam van de annotatie bv VANDAAG in `@FunctieKeyword("VANDAAG")` exact overeenkomt met de naam van de functie in de syntaxdefinitie. Alleen dan kan de functie correct geresolved door de `FunctieFactory`.



Er bestaan vele functies in de expressietaal die allemaal worden uitgevoerd door de `FunctieExpressie`. Voor het echte werk delegeert de `FunctieExpressie` naar `Functie` implementatie. Deze implementaties zijn te vinden in de package `nl.bzk.brp.domain.expressie.functie`.
De `nl.bzk.brp.domain.expressie.functie.Functie` is de superinterface van alle functies.



4.2.5 *Variabelen*

De `VariabeleExpressie` wordt gedefinieerd met een identifier. Deze identifier is een verwijzing naar een waarde (de variabele) die aanwezig is op de evaluatiecontext. Het evalueren van de `VariabeleExpressie` maakt het mogelijk om waarden uit de evaluatiecontext te halen en terug te geven. Het Java type van deze waarden is ook altijd een Expressie.

4.2.6 *Persoonsgegevens*

De ElementExpressie is in staat gegevens te selecteren van het MetaModel van Persoonslijst en de gevonden waarden te retourneren als een LijstExpressie. Deze lijst kan leeg zijn of gevuld, maar bevat nooit NULL waarden indien persoonsgegevens niet bestaan.

ElementExpressies op attribuut, groep, object leveren leveren respectievelijk een LijstExpressie met
MetaAttribuut> waarden, MetaGroepLiterals> en MetaObjectLiterals op.

In de syntax wordt een persoonsgegevens gedefinieerd als de reguliere expressie (zie element_path in Syntax. Een element_path is herhaling van een door een punt (.) gescheiden reeks tekens. De syntax controleert alleen op het patroon. Deze soepelheid van de taal houdt de syntax simpel en de taal makkelijker onderhoudbaar. Een tekenreeks dat aan het patroon voldoet hoeft dus niet per sé een geldig aanwijsbaar persoonsgegevens te zijn. Hier zijn extra parsetime controles voor nodig.

4.2.7 *Resolving*

De driver voor persoonsgegevens-expressies is de Element tabel, en dan specifiek de kolommen, naam, alias en identxsd. De waarden in deze kolommen zijn de basis voor expressies op persoonsgegevens.

Voor alle mogelijke expressies waarmee persoonsgegevens opgevraagd worden bestaat een zogenoemde implementatie van de `nl.bzk.brp.domain.expressie.element.Resolver` interface. De `nl.bzk.brp.domain.expressie.element.ResolverMap` is bekend met alle resolvers en wordt gebruikt door `ElementExpressie` om de persoonsgegevens daadwerkelijk te vinden.

Onderstaand een aantal voorbeelden van expressies op persoonsgegevens, hoe deze samengesteld zijn en hoe ze geresolved worden:

- `Persoon.Identificatienummers.Burgerservicenummer`. Deze elementnaam is direct te gebruiken als expressie om een attribuut te vinden op de PL.
- `x.Naamgebruik.Voorvoegsel`. Hierbij is x een variabele, en Naamgebruik.Voorvoegsel een samenvoeging van groep alias en attribuut alias. Tijdens evaluatie moet eerst x geresolved worden tot een MetaObject. Binnen de context van dat object wordt vervolgens gezocht naar het attribuut met alias Voorvoegsel binnen de groep met alias Naamgebruik.
- `x.burgerservicenummer`. Hier is x een variabele, en burgerservicenummer attribuut alias. Tijdens evaluatie moet eerst x geresolved worden tot een MetaGroep. Binnen de context van dat object wordt vervolgens gezocht naar het attribuut met alias Voorvoegsel binnen de groep met alias Naamgebruik.
- `persoon.Identificatienummers`. Deze elementnaam is direct te gebruiken als expressie om een groep te vinden op de PL.
- `geboorte`. Dit is de alias van de `Persoon.Geboorte` groep. Er is geen variabele gedefinieerd, dus er zal gezocht worden naar alle geboorte groepen binnen Persoon;
- `Persoon.Adres`. Deze elementnaam is direct te gebruiken als expressie om een object te vinden op de PL.

- `Nationaliteiten`. Deze naam alias van het objecttype `Persoon.Nationaliteit` is direct te gebruiken alle MetaObjecten te vinden van type Persoon.Nationaliteit.

4.3 ExpressieType

Elke expressie heeft een type (net als bijvoorbeeld in Java). Het type van een expressie is gedefinieerd als het type van het resultaat als de expressie succesvol geëvalueerd zou worden. Voorbeelden:

- "`10`" is van type `GETAL`.
- "`WAAR`" is van type `BOOLEAN`.
- "`1 + 2`" is van type `GETAL` (als de expressie geëvalueerd zou worden, zou dat leiden tot een GETAL).
- "`1 < 2`" is van type `BOOLEAN`.
- "`VANDAAG()`" is van type `DATUM`.
- "`geboorte.datum`" is van type `LIJST`.

4.3.1 *Ordinal basistypes*

De expressietaal kent drie ordinal basistypes. Dit zijn types die een grootheid aanduiden; elementen van deze types zijn geordend en kunnen onderling vergeleken worden met groter dan en kleiner dan:

- `GETAL`: gehele getallen, zowel positief als negatief;
- `DATUM`: datum, mogelijk met onbekende delen;
- `DATUMTIJD`: datum met tijd, volledig bekend;
- `PERIODE`: een tijdsperiode (een jaar, een jaar en twee maanden, twee dagen etc..);

4.3.2 *Niet-ordinal basistypes*

De expressietaal kent drie basistypes die niet ordinal zijn:

- `STRING`: tekenreeksen;
- `BOOLEAN`: de logische waarden WAAR en ONWAAR;
- `LIJST`: lijsten van expressies. Ontwerpkeuze Om het type systeem van de expressietaal eenvoudig te houden, zijn lijsten in principe ongetypeerd. De expressietaal kent geen onderscheid tussen lijst van getallen en een lijst van strings. In sommige gevallen echter is het wel zinvol om iets over de typering van elementen te weten; daarvoor zijn methoden in de interface Expressie gedefinieerd.

4.3.3 *Speciale types*

De expressietaal kent een aantal speciale types. Deze worden gebruikt voor bijzondere gevallen. Ze zijn niet ordinal.

- `NULL`: type van waarden die niet bepaald kunnen worden
- `ONBEKEND_TYPE`: type van waarden waarvan het type (nog) niet bepaald kan worden;
- `ELEMENT`: Voor notatie van een element op de persoonslijst. De ElementnaamLiteral en de GEWIJZIGD functie maken hier gebruik van om een element aan te wijzen, zonder dat deze direct geëvalueerd wordt.
- `BRP_METARECORD`: voor MetaRecordLiterals om records van persoonsgegevens door te geven.

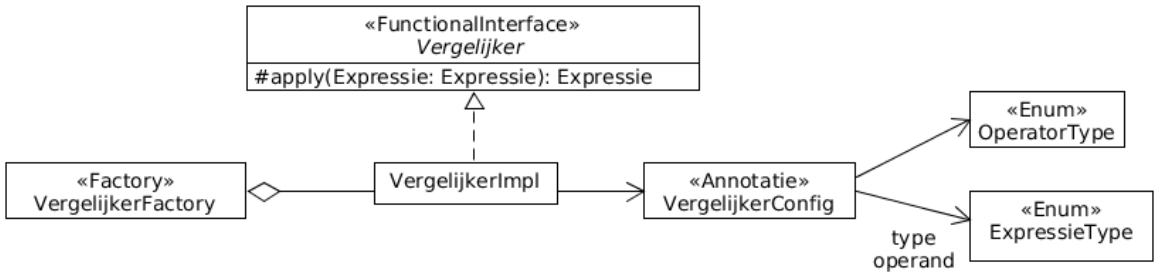
- `BRP_METAGROEP`: voor MetaGroepLiterals om groepen van persoonsgegevens door te geven.
- `BRP_METAOBJECT`: voor MetaObjectLiterals om objecten van persoonsgegevens door te geven.
- `BRP_ACTIE`: voor ActieLiterals om verantwoording van persoonsgegevens door te geven.

4.4 Vergelijkingen

Er is een design keuze gemaakt om de vergelijkingen *kleiner, kleinergelijk, groter, grotergelijk, gelijk, ongelijk* en *wildcardgelijk* voor alle datatypen *getal, String, boolean, null, datum, datumtijd, lijst* los te trekken van de operatoren en van de objecten die met elkaar vergeleken worden.

Bijvoorbeeld, een `GetalLiteral` weet niet hoe deze vergeleken wordt met een andere `GetalLiteral` en de `VergelijkingOperator` weet niet hoe het twee getallen met elkaar moet vergelijken.

Voor het uitvoeren van een vergelijkingen moet er gebruik gemaakt worden van de speciale `Vergelijking` interface, het supertype voor alle vergelijkingen. Een `Vergelijking` is een `BiFunctie` met als parameters en return type een `Expressie`.



Implementaties van Vergelijking kunnen enkel verkregen worden via de `VergelijkerFactory`. Deze factory scant eenmalig naar implementaties in de package `nl.bzk.brp.domain.expressie.vergelijker`.

Voorwaarde is dat implementaties geannoteerd zijn met de `@Component` en de `VergelijkerConfig` annotaties. De VergelijkerConfig annotatie geef aan welke ExpressieType operands hebben en welk type vergelijking ondersteund wordt.

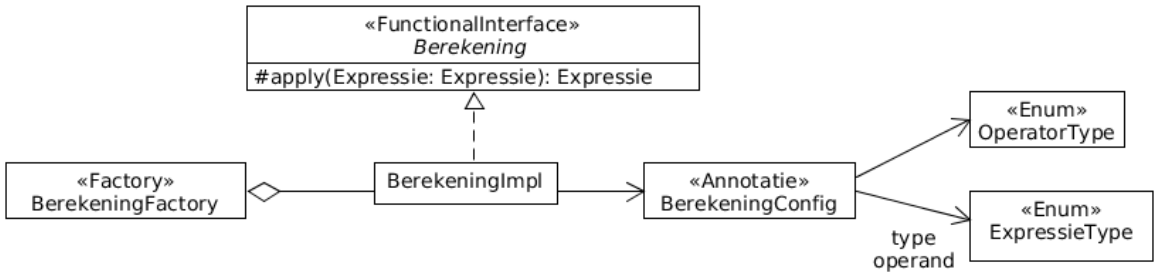
Om een `Vergelijking` op te halen met de `VergelijkerFactory` dient opgegeven te worden welk type vergelijking gedaan moet worden en wat de type operands zijn. (combinatie `ExpressieType` en `OperatorType`).

Onderstaand een voorbeeld waar een boolean gelijk vergelijking gedaan wordt:

```
@Component
@Vergelijking(operator = OperatorType.GELIJK, typeSupport = ExpressieType.BOOLEAN)
final class BooleanGelijkVergelijker implements Vergelijker<BooleanLiteral,
BooleanLiteral> {
  public Expressie apply(final BooleanLiteral l, final BooleanLiteral r) {
    return BooleanLiteral.valueOf(l.equals(r));
  }
}
```

4.5 Berekeningen

Analoog aan vergelijkingen zijn berekeningen als plus, min, losgetrokken van de operatoren en van de objecten waarmee gerekend wordt. Voor het uitvoeren van een berekening moet er gebruik gemaakt worden van de speciale `Berekening` interface, het supertype voor alle berekeningen.



Een Berekening is een `BiFunctie` met als parameter en return type `Expressie`. De `BerekeningFactory` scant eenmalig naar implementaties in de package nl.bzk.brp.domain.expressie.berekening middels de `@Component` annotatie en de `BerekeningConfig` annotatie. De `BerekeningConfig` annotatie geef aan welke `ExpressieType` operands hebben en welke reken operator ondersteund wordt.

Operatoren of functies welke een berekening willen doen moeten de `BerekeningFactory` vragen om een instantie van Berekening welke in staat is een bepaald type berekening te doen (combinatie `ExpressieType` en `OperatorType`). De `BerekeningFactory` is verantwoordelijk voor het zoeken van de correcte Berekening implementaties.

Onderstaand een voorbeeld waar een getallen opgeteld worden:

@Component

```
@BerekeningConfig(operator = OperatorType.PLUS, typeLinks = ExpressieType.GETAL,
typeRechts = ExpressieType.GETAL)
final class GetalPlusGetal implements Berekening<GetalLiteral, GetalLiteral> {
@Override
public Expressie apply(final GetalLiteral getalLiteral,final GetalLiteral
getalLiteral2) {
  return new GetalLiteral(getalLiteral.alsInteger()
    + getalLiteral2.alsInteger());
  }
}
```

5 Foutafhandeling

Fouten in expressies kunnen op verschillende momenten gedetecteerd worden en effect hebben. Veel fouten worden tijdens het vertalen van een expressie al gevonden. Een expressie "`persoon.gb`**`oorte`**`.datum`" zal bijvoorbeeld niet geparsed kunnen worden.

De expressietaal probeert zo snel mogelijk fouten in de expressies te rapporteren. De afweging om fouten vroeg of laat te constateren is vaak gedaan op basis van complexiteit. Het is vaak erg lastig is om controles op een vroeg moment te doen, en vele malen makkelijker op een later moment. Er zijn drie momenten wanneer er fouten kunnen optreden, syntax, parse-tijd en evaluatietijd. In elk van deze gevallen wordt een `ExpressieException` gegooid met daarin een beschrijving van de fout die optreedt.

De cliënt code (de code welke gebruik maakt van de expressietaal) maakt altijd gebruikt van de klasse `BRPExpressies`. Fouten die optreden bij het parsen of evalueren van de expressies wordt altijd vertaald naar de **checked** `ExpressieException`. Intern wordt er gebruik gemaakt van een aantal runtime excepties (`ExpressieParseException`), welke naar de 'buitenwereld' weer vertaald worden naar een `ExpressieException`.

5.1 Syntaxfouten

Syntaxfouten worden gevonden door ANTLR en gerapporteerd middels een errorlistener welke gezet wordt op de `BRPExpressietaalParser` en `BRPExpressietaalLexer`.
De klasse `ParserErrorListener` is de BRP implementatie van `BaseErrorListener` en vertaalt de syntaxfout naar een `ExpressieParseException`.

5.2 Parse-tijd fouten

De syntax van de taal is grotendeels weakly typed. Dit heeft tot gevolg dat ANTLR niet kan rapporteren over constructies waarbij typen niet overeenkomen met wat verwacht wordt. In deze gevallen is het aan de implementatie om dit te constateren en hierover te rapporteren.

Een functie-implementatie is bijvoorbeeld afhankelijk van een aantal argumenten van een gegeven type. De syntax van de taal zegt helemaal niets over het verwachte aantal argumenten en het verwachte type. Het controleren van deze argumenten moet daarom zelf gecontroleerd worden (middels een `Signatuur` controle). Hierbij wordt gekeken of de argumenten evaluatie-tijd het verwachte resultaattype gaan teruggeven. Bijvoorbeeld: `AANTAL(10);` het verwachte type is `LIJST` en niet `GETAL`.

De expressie `WAAR EN 1` geeft de fout "*Parsefout: Boolean expressie verwacht*". EN is een logisch expressie en de linker en rechter operand moeten van het type `BOOLEAN` zijn. De rechter operand is een `GetalLiteral`.

5.3 Evaluatie-tijd fouten

Een aantal controles zijn niet eerder mogelijk dan tijdens evaluatie-tijd. De persoonslijst is bijvoorbeeld pas beschikbaar tijdens evaluatie-tijd, controleren m.b.t. het wel / niet aanwezig zijn van attributen, groepen of objecten kunnen hier pas gedaan worden.

In het kader van robuustheid wordt met onverwachte situaties soepel omgegaan. Dit houdt in dat er géén exceptie gegooid wordt maar dat er vaak NULL of ONWAAR geretourneerd wordt in een onverwachte situatie. Er is getracht elke operator en functie zo robuust te maken dat het kan omgaan met dit soort situaties.

6 Expressies in de BRP

6.1 BRPExpressies API

Modules die gebruik willen maken van de expressietaal dienen dat nu te doen middels de klasse `nl.bzk.brp.domain.expressie.api.BRPExpressies`. Deze klasse biedt een façade voor het parsen en evalueren van expressies. Het draagt zorgt voor een correcte vertaling van excepties naar de checked `ExpressieException`.

6.2 Performance

Voor elke handeling die verwerkt wordt in mutatielevering worden één of meerdere expressies uitgevoerd per autorisatie. Potentieel bevat de database duizenden autorisaties, en moeten er meerdere administratieve handelingen per seconde verwerkt worden. Het is daarom erg belangrijk dat het evalueren van expressies goed performt.

De grootste performance optimalisatie die gedaan is in de BRP is het éénmalig parsen van de expressies tijdens het inlezen van de autorisaties. De geparsde expressie wordt vervolgend gecached.

Het parsen van expressies is in vergelijking tot het evalueren een relatief duur proces (ordegrootte 100 tot 1000 keer duurder). De evaluatieperformance hangt sterk af van een aantal factoren; aantal sub-expressies (méér is langzamer), functie gebruik (sommige functies zijn minder efficiënt, bijvoorbeeld hele grote IN clausules), de persoon zelf (bijvoorbeeld oud/nieuw) vergelijkingen op grote persoonslijsten.

Al met al is de evaluatie performance nu geen beperkende factor mbt de verwerkingssnelheid in mutatielevering. Het is daarom niet verder geoptimaliseerd. E.e.a. is aangetoond met een aantal performance metingen welke te vinden zijn in het BRP Benchmark project.

6.3 Verbetermogelijkheden

Er zijn altijd verbetermogelijkheden te bedenken, onderstaand een overzicht:

- De client-API is niet helemaal strak, evaluatie kan via `nl.bzk.brp.domain.expressie.api.BRPExpressies`, of direct op via Expressie object. In het laatste geval moet het Context object zelf goed gevuld worden wat natuurlijk foutgevoelig is. Wellicht is het beter om de context algeheel te verbergen in de client-API en dus ook een andere interface terug te geven aan de cliënt dan de Expressie interface.
- De DOT-notatie regex voor persoonsgegevens expressies kan versimpeld worden.