UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Van sequence datalog naar prolog : ondersteuning van path expressions in prolog

Alessio Costa

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Jan VAN DEN BUSSCHE

2021
2022

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Van sequence datalog naar prolog : ondersteuning van path expressions in prolog

Alessio Costa

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Jan VAN DEN BUSSCHE

Dankwoord

Tijdens het academiejaar 2021-2022 heb ik gewerkt aan deze masterproef. Deze weg heb ik natuurlijk niet alleen afgelegd. Hiervoor ben ik veel samengekomen met mijn promotor en begeleider om de progressie te bespreken. Het stopt niet alleen hierbij want er is veel meer gedaan. Tijdens onze meetings is er niet zomaar feedback gegeven maar alles werd goed besproken en eventueel uitgelegd waar nodig zodat ik goed verder kon. Mijn vragen werden telkens goed onderzocht en er een betekenisvolle antwoord aan gegeven.

In eerste instantie zou ik mijn promotor Jan Van den Bussche willen bedanken. Hij heeft niet alleen zijn kennis gedeeld over het onderwerp maar ook bij andere zaken geholpen. Onderdelen hiervan zijn: het aanreiken van studiemateriaal, het geven van feedback van de gemaakte voorbeelden en de aanpak van deze paper. Ook heb ik meer inzicht gekregen om problemen te kunnen aanpakken door naar meerdere oplossing te kijken.

Vervolgens zou mijn begeleider Heba Mohamed willen bedanken. Tijdens dit proces heeft zij vooral haar kennis gedeeld over het onderwerp zelf. Hierdoor heb ik beter er beter inzicht in gekregen, niet alleen in het praktisch gedeelte zelf maar ook de theoretische achtergronden ervan. Hiervoor zijn we verschillende keren samengekomen om mijn voorbeelden over het onderwerp te bespreken.

Ten slotte wil ik vooral mijn ouders bedanken die mij hebben gesteund doorheen mijn studies. Zij hebben mij de mogelijkheid gegeven om zonder zorgen deze te kunnen afronden. Hiernaast zou ik ook graag mijn vrienden en familie bedanken voor de steun.

Samenvatting

In deze masterproef is de probleemstelling hoe we van een eerder theoretisch programmeermodel naar een praktische kunnen gaan. In ons geval is het theoretisch model Sequence Datalog en de praktische programmeertaal is Prolog. Dit situeert zich rondt het werken met sequentiële data en die kunnen manipuleren. Enkele voorbeelden van dit soort data is een logbestand waarbij elk event een tijd heeft, of genoomdata waarbij we een collectie hebben van RNA-strings. In dit geval gaat het vooral over het opvragen van dit soort data met behulp van eigen vraagstellingen of queries.

Het probleem is dat Sequence Datalog niet direct uitvoerbaar is in de praktijk omdat dit een theoretisch model is. Een belangrijke feature van Sequence Datalog zijn path expressions en wordt niet standaard ondersteund door Prolog. In een standaard toepassing in Prolog is de feature van path expressions bijna of helemaal niet zichtbaar. Omdat deze feature wel een belangrijk onderdeel is van Sequence Datalog en niet van Prolog is dit een probleem voor het implementeren van uitvoerbare toepassingen vanuit Sequence Datalog naar Prolog. Hiervoor ga ik een oplossing vinden en onderzoeken of deze uitvoerbaar is en goed werkt.

De doelstelling is om de ondersteuning van path expressions in Prolog, zoals die we hebben in Sequence Datalog, niet alleen op een simpele manier te kunnen toepassen maar ook onderzoeken of het effectief werkt in de praktijk. Hiervoor beschrijf ik eerst de situering zoals welk soort data alsook welke Prolog versie ik hiervoor gebruik. Deze twee programmeertalen zelf leg ik ook uit. Dan stel ik een oplossing voor m.b.v. voorbeelden en vergelijk ik telkens de drie versies: in Sequence Datalog, in Prolog op de standaard manier en in Prolog met de eigen implementatie van de feature. Hiervoor stel ik verschillende kleinere voorbeelden voor alsook twee grotere toepassingen.

Als resultaat heb ik een manier gevonden om Sequence Datalog te kunnen omzetten naar een Prolog-programma met ondersteuning van path expressions. Dit werkt maar uit de testen is er een probleem opgedoken: dit werkt niet goed bij een complexere query die veel diepgang heeft. Indien het zoeken naar een oplossing te diep in de recursie zit dat werkt dit niet. Ik heb hier de reden voor onderzocht, gevonden en beschreven.

Ik stel vast dat het mogelijk is om van Sequence Datalog naar Prolog te gaan maar niet zonder fouten. Bij een complexere vraagstelling dan kan het programma geen oplossing berekenen net door die diepgang. Dit betekent dat deze toepassing nog niet klaar is voor echt gebruik maar er meer onderzoek voor nodig is. Een voorbeeld hiervoor is kijken of eventueel andere programmeertalen dan Prolog of zelfs andere versies ervan dit wel kunnen realiseren.

Inhoudsopgave

1 Inleiding **1**
- 1.1 Sequentiële data . . . 1
- 1.2 Prolog, Datalog en Logic Programming . . . 2

2 Prolog **3**
- 2.1 Definities . . . 4
- 2.2 Unification . . . 5
- 2.3 Prolog voorbeelden met uitleg . . . 6
- 2.4 Lijsten . . . 9
- 2.5 Proof Search . . . 11
- 2.6 Negatie en cut . . . 12
- 2.7 Recursie . . . 13
- 2.8 Practical . . . 14

3 Sequence Datalog **17**
- 3.1 Syntax . . . 17
- 3.2 Concatenatie in Sequence Datalog . . . 19
- 3.3 Voorbeelden . . . 19

4 Implementatie van Sequence Datalog in Prolog **21**
- 4.1 Path Expressions als lijsten . . . 22
- 4.2 Path Expressions in Prolog - Introductie Expr-Predicaat . . . 23
  - 4.2.1 Van Sequence Datalog naar Prolog . . . 24
- 4.3 Bijkomende voorbeelden . . . 25
  - 4.3.1 a-exclusief . . . 25
  - 4.3.2 Invoer macht n . . . 27
  - 4.3.3 Omgekeerde lijst . . . 29
  - 4.3.4 Path Exists . . . 31

5 Toepassing 1: Workflow Data **39**
- 5.1 Van XES-bestand naar Prolog facts . . . 40
- 5.2 Query 1 . . . 42
- 5.3 Query 2 . . . 42
- 5.4 Query 3 . . . 43

6 Toepassing 2: Genome Data **45**
- 6.1 Genereren van data . . . 47
- 6.2 Van RNA naar proteïne - Sequence Datalog . . . 48
- 6.3 Van RNA naar proteïne - Prolog . . . 49
- 6.4 Van RNA naar proteïne - Expr-Predicaat . . . 52

6.5 Extra voorbeeld - data queryen 54
6.6 Grootte van data 55

7 Conclusies **59**

Hoofdstuk 1

Inleiding

Deze masterproef situeert zich in de wereld van databases, meer specifiek sequentiële databases. Met behulp van programma's in bepaalde programmeertalen kunnen we die sequentiële data manipuleren. Dit soort data leg ik specifiek uit in hoofdstuk 1.1 en verder refereer ik hiernaar met gewoon *data*. De bedoeling is van een programmeertaal die nog vooral theoretisch beschreven is over te gaan naar een praktische programmeertaal. Het theoretische programma is Sequence Datalog (of gewoon Datalog) en wordt specifiek uitgelegd in hoofdstuk 3. Het zogenaamd praktische programma die echt uitvoerbaar is op data is Prolog die ik ook als eerste bespreek in hoofdstuk 2. Omdat deze ook uitvoerbaar is en dus oefening op kan worden gemaakt ben ik hiermee begonnen. Dan is de overgang naar de theoretische Sequence Datalog die er erg op lijkt gemakkelijker. Die twee programmeer-talen heb ik ingestudeerd met behulp van een aantal bronnen: [LearnPrologNow], [Expressiveness within Sequence Datalog] en [Sequences Datalog en Transducers].

Een belangrijk onderdeel van deze studie zijn path expressions. Dit leg ik in detail uit in hoofdstuk 3.1. Essentieel is dit een manier om uw data te bewandelen door te omschrijven op welke manier die eruit ziet of hoe je verwacht die opgesteld moet worden. Path expressions worden niet standaard ondersteund in Prolog. De onderzoeksvraag van deze masterproef is of en hoe we path expressions kunnen gebruiken zoals in Sequence Datalog om naar Prolog te gaan. De link naar Prolog kan gelegd worden omdat we hiermee hetzelfde kunnen bereiken zoals in Sequence Datalog als we met lijsten werken. De sequenties worden dus voorgesteld als lijsten zoals ik bespreek in hoofdstuk 4.1.

1.1 Sequentiële data

Sequentiële data spreekt een beetje voor zichzelf. Volgens [Encyclopedia of Machine Learning] is het elke data die elementen bevat in de vorm van sequenties. De onderdelen van de sequenties zijn op een of andere manier gelinkt aan elkaar. Een goed voorbeeld hiervan is DNA-sequenties waarover ik een toepassing voor heb ik hoofdstuk 6. Een ander voorbeeld is als de sequentie verbonden zijn met een bepaald tijdstip en ook hierop geordend. Een toepassing hierover genaamd workflow data bespreek ik in hoofdstuk 5. Hierbij komen sequenties van data voor waarbij de elementen ervan verbonden zijn met een tijdstip. Sequentiële data heeft ook toepassingen in machine learning maar dat is voor deze masterproef niet relevant.

1.2 Prolog, Datalog en Logic Programming

De eerste stappen naar Prolog werden gemaakt in 1972, we zijn dus 50 jaar verder. Sinds die tijd zijn er vele versies van Prolog uitgekomen. Voorbeelden hiervan zijn SWI-Prolog en GNU-Prolog. Logic Programming speelt hier een grote rol bij. Volgens de paper [Fifty Years of Prolog and Beyond] is Prolog argumenteerbaar de meest prominente programmeertaal wanneer we spreken over Logic Programming. Een belangrijk onderdeel hiervan is het automatisch maken van oplossingen gebaseerd op een logisch programma zonder te baseren op functies. Het maken van deze oplossingen is een onderdeel van het stellen van vragen m.b.v. First-Order Logic waar ik voorbeelden voor geef in hoofdstuk 2.

De voordelen van Datalog is het declaratief programmeren. Hiervoor hoeven we enkel te bepalen wat en niet hoe iets gedaan moet worden. Logic Programming laat declaratief programming toe maar declaratief programming is breder. Een andere programmeertaal die declaratief programming toelaat is bijvoorbeeld SQL die ook eigenlijk declaratief is. Prolog is de gestandaardiseerde Logic Programming taal. Om de voordelen van Logic Programming ook te krijgen voor database queries is er daarvoor Sequence Datalog voor uitgevonden. Wat er in deze masterproef specifiek bijkomt is het onderdeel van sequenties. Door de relatie van Datalog met Logic Programming kan de link ook gelegd worden naar Prolog. In de paper [Fifty Years of Prolog and Beyond] verwijzen ze zelfs naar Datalog als een subset van Prolog.

Er zijn verschillende versies van Prolog zelf. Voor deze masterproef gebruik ik SWI-Prolog. Volgens de paper [Fifty Years of Prolog and Beyond] is dit een general-purpose systeem die dient voor echt applicaties waardoor multithreading hier ook een onderdeel van is. SWI-Prolog heeft de focus op robuustheid, schaalbaarheid en compatibiliteit met oudere versies van SWI-Prolog. Vooral door de voordelen van general-purpose en geschiktheid voor echte applicaties leek dit een goede programmeertaal voor te gebruiken in deze masterproef. Een andere versie van Prolog is bijvoorbeeld GNU Prolog.

Hoofdstuk 2

Prolog

De Prolog versie die ik gebruik in deze masterproef is SWI-Prolog. De reden hiervoor is beschreven in hoofdstuk 1.2. Het leren van Prolog was voor mij niet zoals het leren van een typische programmeertaal. Het schrijven van deze programma's vereist kennis van de verschillende componenten van de taal en de manier van interpretatie. Ik heb hiervoor de online website [LearnPrologNow] gebruikt. Prolog staat voor programming in logic. De componenten ervan zijn facts, rules and queries. Dit leg ik vervolgens uit.

```
% Facts
loves(vincent,mia).
loves(marsellus,mia).
loves(pumpkin,honey_bunny).
loves(honey_bunny,pumpkin).
```

Een **fact** beschrijft een **predicaat** over het domein. Een predicaat is toegepast op 0 of meerdere argumenten. In bovenstaand voorbeeld is "loves" een predicaat toegepast op 2 argumenten. Er kunnen ook meerdere facts beschreven worden. Elke fact eindigt op een punt. En even terzijde het procentteken is een notitie en is niet uitvoerbare code. Hier kan je ook namen zien zoals vincent en mia die atoms zijn. Beschouw ze voorlopig als gewoon namen, dit beschrijf ik verder in hoofdstuk 2.1.

```
% Rules
jealous(X,Y) :- loves(X,Z),
                loves(Y,Z).
```

Een **rule** ziet eruit als een fact maar heeft iets meer. Een fact beschrijft eerder een statement naar een rule gaat een relatie(s) opstellen vanuit de facts. En even terzijde facts kan je beschouwen als de eenvoudigste soort rules. Dit heeft twee onderdelen en wordt gesplitst door het teken : −. Merk op dat dit teken gewoon een voorstelling is van een pijltje naar links ←. De linkerzijde noemt de rule head (of simpelweg head) en bevat een enkele literal. Hier mag ook geen negatie in voorkomen. De rechterzijde noemt de body van de rule en kan 1 of meerdere regels bevatten, gescheiden door een komma die "en" wil zeggen. En even terzijde de puntkomma wil "of" zeggen.

Een rule wordt bekeken als *head als body*. Dit wil zeggen dat de head wordt geïmpliceerd als de body voldoet. Merk op dat in bovenstaande rule de hoofdletters X, Y en Z geïntroduceerd worden. Dit zijn variabelen en om dit beter te begrijpen refereer ik naar hoofdstuk 2.1. Voorlopig om dit nu te begrijpen volstaat het om te weten dat variabelen willekeurige waarden voorstellen, zolang de body voldaan is. X en Y zijn in bovenstaand

voorbeeld verschillende variabelen maar ze kunnen wel nog dezelfde waarden bevatten. Bovenstaande rule wil dus zeggen dat als er een X bestaat die van Z houdt, en er bestaat een Y die ook van dezelfde Z houdt, dan is X jaloers op Y!

```
wizard(ron).
hasWand(harry).
quidditchPlayer(harry).
wizard(X):- hasBroom(X), hasWand(X).
hasBroom(X):- quidditchPlayer(X).
```

De **knowledge-base** is de collectie van facts en rules samen. Prolog gaat in de rules proberen de body te voldaan m.b.v. de informatie die beschikbaar is in de knowledge-base. Indien voldaan dan wordt de head afgeleid. Bovenstaande facts zegt dus:

- Ron is een wizard.
- Harry heeft een wand.
- Harry is een quidditchplayer.

De rules stelt de relatie op:

- Als X een broom heeft en diezelfde X heeft ook een wand, dan is X een wizard.
- Als X een quidditchplayer is dan heeft diezelfde X een broom.

Hier merk je op dat je ook en de body van een rule een relatie kan opstellen die naar een andere rule verwijst. We kunnen met deze knowledge-base een aantal interessante vragen bij opstellen. De queries en bijbehorende antwoorden volgen:

```
% wizard(ron). true
% witch(ron). error
% wizard(hermione). false
% witch(hermione). error
% wizard(harry). true
% wizard(Y). Y = ron; Y = harry
% witch(Y). error
```

Merk op dat als je predicaten aan het programma vraagt die niet bestaan in de knowledge-base dan geeft het een error. Ook kan je zien in de voorlaatste query dat er meerdere antwoorden kunnen gegeven worden. Als er meerdere antwoorden zijn dan geeft het programma het eerste antwoord en wacht dan op gebruikers-input. Als je het puntkomma teken input, wat "of" betekent, dan geeft het de volgende oplossing. Als je een punt input dan stopt het programma ook al zijn er nog andere antwoorden.

Belangrijk te vermelden is dat Prolog gebruik maakt van First-order Logic. Inderdaad, First-Order Logic is eveneens opgebouwd uit statements die een predicaat toepast op 0 of meer argumenten. Dit wordt bijvoorbeeld geschreven zoals $apple(jonagold)$. Prolog gebruikt logische conjunctie en logische disjunctie. Logische conjunctie is een AND en wordt geschreven met een komma. Logische disjunctie is een OR en wordt geschreven met een puntkomma. Prolog maakt gebruik van depth-first search.

2.1 Definities

Een **atom** begint met een kleine letter, omgeven door single quotes (maar moet niet) of speciale karakters. Voorbeelden hiervan zijn $jonagold$, $'jonagold'$ en @. De eerste twee

zijn aan elkaar gelijk. De reden dat er single quotes zijn om de atom toch te laten beginnen met een hoofdletter zonder dat het een variabele wordt. Sommige atoms hebben een voorgedefinieerde betekenis zoals ; waarover later meer. Een **variable** begint met een hoofdletter of underscore. Dit is een placeholder voor informatie. Het programma zal proberen, van boven naar onder, om deze variable een betekenis te geven of te unifyen. In volgend hoofdstuk leg ik unification uit. Een **complex term** is een predicaat gevolgd door een sequentie van argumenten zoals $predicaat(argument1, argument2)$. De **arity** is de hoeveelheid argumenten een predicaat heeft.

2.2 Unification

Stel we hebben volgend knowledge-base:

```
fakeApple(pear).
apple(jonagold).
apple(golden).
```

Het programma probeert de goal te bereiken door de query te unifyen met alle heads (rules of head van facts) door van boven naar onder te gaan. Ook negeert het de heads die niet matchen met de query. De manier waarop het programma zoekt is **depth-first search**.

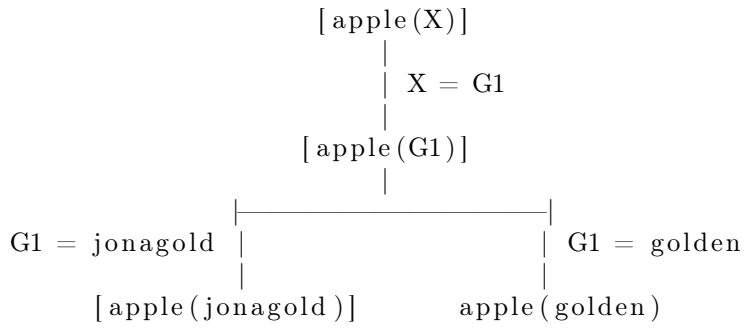
We stellen de query ***apple(X)***. De eerste rule in bovenstaande knowledge-base is

$$fakeApple(pear).$$

Deze rule komt natuurlijk niet overeen met onze query dus het programma gaat verder naar de volgende rule. De tweede rule $apple(jonagold)$ komt wel overeen. Het programma unificeert de variabele X met de atom jonagold dus $X = jonagold$. Als dit waar is dan komen we uit op de $apple(jonagold)$. Dit is een goal! Het programma gaat een opeenvolging van goals proberen te vinden dus we stoppen niet bij de eerste. De volgende rule is $apple(golden)$. Zoals ervoor wordt de variabele X unified met de atom $golden$, dus $X = golden$, en bekomen we volgens de rule $apple(golden)$. Dit is de tweede en laatste goal! Uiteindelijk ziet de query er zo uit in Prolog:

```
?- apple(X).
X = jonagold ;
X = golden.
```

Vervolgens ziet de (depth-first search) tree er zo uit:



Elke tekst tussen vierkante haakjes is eender een query of een goal. Vanboven beginnen we met alle variabelen om te zetten naar een nieuwe shared-variabele. In ons geval hebben we er maar 1, maar als er meerdere goals zijn dan worden ze gesplit met een komma

en elke goal krijgt een nieuwe variabele. Deze search-tree heeft twee bladeren en beiden zijn een oplossing voor dit programma. Het linkerblad geeft $G1$ de waarde jonagold en het rechterblad geeft deze golden. Zo kan je visueel zien dat het programma dus echt zoekt naar een of meerdere oplossingen vanuit de bestaande knowledge-base.

2.3 Prolog voorbeelden met uitleg

In Prolog kunnen we regels opstellen die bestaat uit een head en body. Merk op dat de body eventueel leeg kan zijn. Eerder heb ik besproken dat de head pas wordt afgeleid als de body voldaan is. Voorbeeld:

```
round(ball).
burger(X) :- round(X).
```

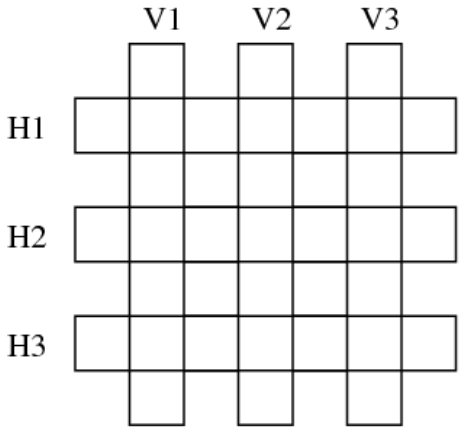
X hier is een variabele. De X in *burger* is exact hetzelfde als de X in *round*. We stellen de query **burger(Y)**. Het programma komt in het predicaat terecht en komt een onbekende X tegen. Van boven naar onder komen we *ball* tegen als candidaat. X wordt hiernaar geinstantieerd. De body is in deze stap dus $round(ball)$. Dit is true, omdat we hier een fact van hebben! Omdat nu een oplossing is gevonden voor de body, wordt de head inferred. Let hier op dat we X hebben unified met *ball* en dit blijft zo, zolang het programma niet back-trackt. Dit komt omdat, zoals je eerder had gezien met de search-tree, het programma kan teruggaan en een andere waarde toekennen aan de variabele. De head zegt dat $burger(ball)$ geldt. Dus onze query geeft terug dat $Y = ball$. Er zijn geen andere manieren om X te unifyen naar iets anders omdat het de enige atom is in het programma. Het programma stopt ook.

We kunnen meerdere regels met dezelfde naam hebben in een programma. Als we een query opstellen dan gaat het programma van boven naar onder de predicaten uitvoeren. Je kan je dus voorstellen dat er meerdere manieren zijn om een oplossing te vinden voor een query, door bijvoorbeeld eerst een snelle of naive oplossing te zoeken en indien dit faalt, goed gaan zoeken. Een voorbeeld hiervoor is een rekenintensieve query zoals machten.

Je moet weten dat we in Prolog tijdelijk resultaten kunnen opslaan in de interne database. Als we dus vele keren de macht van iets moeten berekenen kunnen we dit in de tussentijd opslaan. Bij herhaling van deze berekening hoeven we simpelweg de database te raadplegen i.p.v. opnieuw te rekenen. Hiervoor kunnen we twee predicaten opstellen. De eerste gaat eerst in de database kijken of er al een oplossing voor is opgeslagen in het verleden, voor de gegeven parameters. De tweede gaat de eigenlijke berekening doen. Het programma stopt bij het eerste of er een oplossing voor is gevonden. Om dit te laten werken moeten we wel in het tweede predicaat het resultaat gaan opslaan.

```
:- module(database, [pow/3]).
:- dynamic pow_result(X, Y, Result).

pow(X, Y, Result) :-
    pow_result(X, Y, Result),
    format("Result retrieved from database: ~w\n", [Result]),
    !.

pow(X, Y, Result) :-
    Result is X^Y,
    format("Result calculated: ~w\n", [Result]),
    assert(pow_result(X, Y, Result)).
```

Zie bovenstaande code voor een uitgewerkt voorbeeld. Het belangrijkste hier is de twee regels te kunnen onderscheiden. Het eerste gaat bij regel 5 het resultaat opzoeken. Als deze regel **true** is, betekent dat het bestaat in de huidige knowledge-base en gaat het programma verder. Regel 6 is gewoon een print. Regel 7 is een cut, het programma voert regel 9 niet uit. Een cut leg ik uit in hoofdstuk 2.6. Als regel 5 **false** is dan gaat het programma verder naar de volgende predicaat bij regel 9. Regel 10 is de eigenlijke berekening. Regel 11 is een print. Regel 12 is het resultaat opslaan in de database.



Figuur 2.1: Oefening uit [LearnPrologNow]: kruiswoordpuzzel

Een oefening uit [LearnPrologNow] (zie figuur 2.1) verwacht ons een kruiswoordpuzzel te maken die ons zegt hoe we de grid moeten opvullen. We krijgen er een woordenboek bij van 6 woorden die het programma moet proberen invullen in het kruiswoordraadsel. De zes woorden zijn in het engels en zijn: abalone, abandon, anagram, connect, elegant, enhance. Dit wordt voorgesteld in volgende knowledge-base:

```
word(abalone,a,b,a,l,o,n,e).
word(abandon,a,b,a,n,d,o,n).
word(enhance,e,n,h,a,n,c,e).
word(anagram,a,n,a,g,r,a,m).
word(connect,c,o,n,n,e,c,t).
word(elegant,e,l,e,g,a,n,t).
```

De opdracht is om nu een predicaat te maken genaamd *crosswd/6* (de 6 staat voor de arity of aantal argumenten). Even terzijde een laag streepje is een anonieme variabele wat wil zeggen dat het niet uitmaakt welke waarde hieraan wordt gegeven. Belangrijk

om te weten is dat elk van deze apart wordt beschouwd, als er meerdere zijn dan is er geen link met elkaar. De bijbehorende *crosswd/6* is als volgt:

```
crossword(H1, H2, H3, V1, V2, V3):-
    word(H1, _, _H12V12, _, _H14V22, _, _H16V32, _),
    word(H2, _, _H22V14, _, _H24V24, _, _H26V34, _),
    word(H3, _, _H32V16, _, _H34V26, _, _H36V36, _),
    word(V1, _, _H12V12, _, _H22V14, _, _H32V16, _),
    word(V2, _, _H14V22, _, _H24V24, _, _H34V26, _),
    word(V3, _, _H16V32, _, _H26V34, _, _H36V36, _).
```

Als we bovenstaande code samenzetten met figuur 2.1 dan kan je visueel meteen zien waarom de code er zo uitziet. De eerste regel bijvoorbeeld begint met H1, dit is een unieke variabele voor een uniek woord. Dit wordt uiteindelijk een van de woorden. Dan volgen de 7 letters. De eerste letter maakt niet uit wat het is, omdat het niet gedeeld wordt met eender wat. Dit geldt voor elke eerste letter dus die geven we telkens aan met een anonieme variabele. Voor ons maakt het dus niet uit welke letter dit is.

Vervolgens maakt de tweede letter wel uit. De tweede letter van H1 bijvoorbeeld moet hetzelfde zijn als die van V1 omdat die kruisen. Die krijgen in het programma ook exact dezelfde variabele-naam toegewezen. H1 krijgt als tweede letter dus _H12V12 en die is net hetzelfde als de tweede letter van V1. Hierdoor kunnen we regels maken zodat bepaalde letters hetzelfde moeten zijn. Niet alleen de tweede letter van H1 en V1 moeten hetzelfde zijn, maar ook elke andere letter die kruist. Op die manier kan je dus de hele kruiswoordpuzzel opstellen.

```
?- crossword(H1,H2,H3,V1,V2,V3).
H1 = abalone,
H2 = anagram,
H3 = connect,
V1 = abandon,
V2 = elegant,
V3 = enhance ;

H1 = abandon,
H2 = elegant,
H3 = enhance,
V1 = abalone,
V2 = anagram,
V3 = connect ;

false.
```

Als resultaat krijgen we bovenstaande query en output. Er zijn dus twee combinaties waarmee we de kruiswoordpuzzel kunnen oplossen. Je kan zien dat de regels die werden opgesteld werken. De eerste letter maakt totaal niet uit voor eender welke regel. De tweede letter van bijvoorbeeld H1 komt overeen met de tweede letter van V1 zoals eerder aangegeven. Uiteindelijk bekomen we in dit voorbeeld twee resultaten.

2.4 Lijsten

```
[a,b,c]
```

Lijsten in Prolog worden gerepresenteerd met vierkante haakjes. Een lege lijst wordt genoteerd als []. Een lijst kan van alles bevatten. Het kan atoms, variabelen, termen of zelfs andere lijsten[1]. Een belangrijk concept voor lijsten in Prolog is de head en de tail. De head is de eerste waarde uit te lijst. Zoals eerder besproken kan de waarde van de head dus van alles zijn. De tail is simpelweg de rest van lijst. In bovenstaand voorbeeld is a de head en $[b, c]$ de tail. Merk op dat a een constante variabele is en de rest blijft een lijst.

```
?- [H|T] = [a,b,c]
H = a,
T = [b,c]
```

Bovenstaand voorbeeld is de syntax van een lijst met een head en tail. De variabele H staat voor de head van de lijst en is de constante variabele a. T is de tail van de lijst en is de lijst $[b, c]$. Er zijn ook meerdere operaties mogelijk als we meer willen weten over de lijst waar we mee werken. Behalve alleen met de head en tail te werken kunnen we ook specifieke plaatsen in de lijst aanspreken.

```
?- [X,Y,Z|E] = [a,b,c,d,e].
X = a,
Y = b,
Z = c,
E = [d, e].
```

In bovenstaande code kan je zien dat je in de head van de lijst meer kan doen dan alleen het eerste element uit te halen. Merk op dat dit alleen in de head kan gedaan worden. De tail verwacht geen komma's want dit is letterlijk de rest van de lijst. In dit voorbeeld krijgt elke variabele in de head 1 element voor zich. We kunnen ook zoals in onderstaande code een bepaalde element opvragen. Onderstaande code laat hetzelfde zien maar dan met een lijst in genesteld:

```
?- [X,Y,Z|E] = [[a,x,z],b,c,d,e].
X = [a, x, z],
Y = b,
Z = c,
E = [d, e].
```

[1] Een lijst is eigenlijk zelf een term. Voorbeeld: $[H|T]$ is zoals $lijst(H, T)$, of $[a, b, c]$ is zoals $lijst(a, lijst(b, lijst(c, [])))$

```
?- X = [a,b,c], nth0(2, X, A).
X = [a, b, c],
A = c.

% append(Lijst1, Lijst2, Lijst1enLijst2)

?- append([a,b,c], [d,e,f], X).
X = [a, b, c, d, e, f].

?- append([a,b,c], X, [a,b,c,d]).
X = [d].

?- append(X, [d], [a,a,a,a,d]).
X = [a, a, a, a] ;
false.

?- append(X, [d], [a,a,a,a,a]).
false.
```

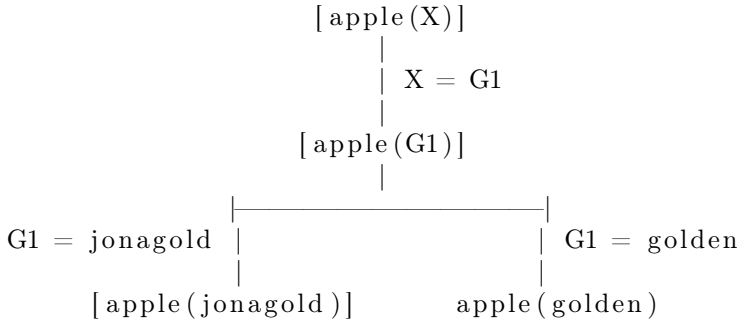
Verder is een krachtige manipulatie voor lijsten die ik veel heb gebruikt een predicaat in Prolog genaamd *append*/3[2]. Waarom we dit nodig gaan hebben komt te pas in hoofdstuk 4.1. Het programma probeert alle drie de parameters te unificeren of waarden te geven zodat de concatenatie van lijsten klopt. Het moet kloppen zodat de derde parameter de uitkomst is van de concatenatie van de eerste parameter met de tweede parameter. We zijn ook niet verplicht om alleen bepaalde parameters in te vullen. Het programma zoekt zelf de goals voor alle mogelijke combinaties die we geven.

In bovenstaand voorbeeld is het procent-teken een commentaar die de syntax laat zien. Daaronder volgt een simpele append. Je kan je inbeelden dat je hier krachtige logische connecties mee kan maken. Deze functie is simpel maar krachtig en gebruik ik veel in de oefeningen die ik later bespreek. De append-functie is ook een onderdeel van de implementatie van voor het ondersteunen van path expressions in Prolog.

[2]De 3 staat voor de arity. Dus het predicaat append heeft 3 argumenten.

2.5 Proof Search

Proof Search wil zeggen dat Prolog keuzepunten heeft waar die keuzes maakt voor een variabele. Prolog zal deze keuzepunten onthouden wanneer die een variabele probeert te unifyen. Het programma zal backtracken wanneer er geen oplossing is naar een vorige keuzepunt. Als je niet dit standaardgedrag wil kan je een keuzepunt committen m.b.v. het uitroepteken (!). Dit is een cut. Een cut leg ik uit in hoofdstuk 2.6.



Voor een programma kan je een keuzeboom of search-tree uittekenen zoals die in hoofdstuk 2.1 die ik hier bovenaan nog even toon. Ik neem nu hetzelfde voorbeeld om een cut uit te leggen. De knowledge-base is als volgt:

```
fakeApple(pear).
apple(jonagold).
apple(golden).
```

De query:

```
?- apple(X), !.
X = jonagold.
```

De query die ik eerder gebruikte was $apple(X)$. Die verander ik nu naar $apple(X), !$. De cut zorgt ervoor dat na de eerste goal de variabelen die we hebben unificeert die committen. Met andere woorden ze mogen niet meer veranderen en als resultaat gaat het programma niet backtracken om deze te veranderen. In bovenstaande seach-tree wil dit zeggen dat na het linkerblad te hebben bewandelt het programma stopt. De variabele $G1$ behoudt de waarde jonagold! Hierdoor stopt het programma na de eerste goal te hebben bereikt.

2.6 Negatie en cut

Negatie of de Engelse term "negation as failure" is zogenaamd een manier in Prolog om negaties voor te stellen gebruikmakend van een **cut**. Omdat Prolog niet rechtstreeks negaties kan bewijzen gaat die eerst een goal proberen te volbrengen zonder de negatie. Indien deze goal faalt dan is de negatie geslaagd. Ook vice versa indien de goal slaagt dan is de negatie gefaald. In Prolog is de notatie het uitroepteken voor een cut. Daarnaast kan je een negatie voorstellen met een cut gevolgd door een letterlijke **fail**.

Prolog - negation as failure:

```
burger(b).
raw_burger(r).
enjoys(vincent, X) :- raw_burger(X), !, fail.
enjoys(vincent, X) :- burger(X).
```

In bovenstaand voorbeeld houdt Vincent niet van een rauwe burger (regel 3), dit is dus een negatie. Zoals je kan zien schrijven we dit gewoon als een rule net zoals Vincent wél van een rauwe burger houdt. Maar dit wordt opgevolgd door een cut en een fail die de negatie wil voorstellen. Merk op dat deze negatie wordt toegepast op het hele predicaat. Een voorbeeld van een query op deze knowledge-base volgt.

Prolog - voorbeeld uitvoer met cut-fail:

```
?- enjoys(vincent, b).
true.

?- enjoys(vincent, r).
false.
```

Hetzelfde programma kan je korter schrijven door de twee predicaten genaamd *enjoys* samen te nemen:

Prolog - negation as failure met /+:

```
burger(b).
raw_burger(r).
enjoys(vincent, X) :- burger(X), \+ raw_burger(X).
```

De backslash in Prolog is een teken om negatie voor te stellen. Deze gevolgd door een plus-teken wil zeggen dat het predicaat *true* is als het argument faalt en vice versa. Aanvullend kunnen we de cut apart gebruiken om variabelen vast te leggen. Hier had ik al een voorbeeld voor gegeven in hoofdstuk 2.5 dus ga ik hier niet dieper op.

2.7 Recursie

Een feature dat Prolog zeer krachtig maakt is recursie. Recursie in Prolog gebeurt automatisch omdat het gebruikt maakt van depth-first search, dit leg ik iets verder uit met een voorbeeld. Een simpel voorbeeld is het voorstellen van nakomelingen. In de knowledge-base hebben we rules die zeggen dat bepaalde personen kinderen zijn van iemand. Deze regels worden voorgesteld door $child(john, brady)$. Dit is het enige dat we weten. Apart van dit willen we een predicaat die onderzoekt als X een descendant is van Y. Zonder recursie zouden we alleen weten als X een descendant is van Y. Maar we willen dieper gaan kijken, omdat Y ook een grootouder kan zijn van X. Zonder recursie ziet de functie er zo uit:

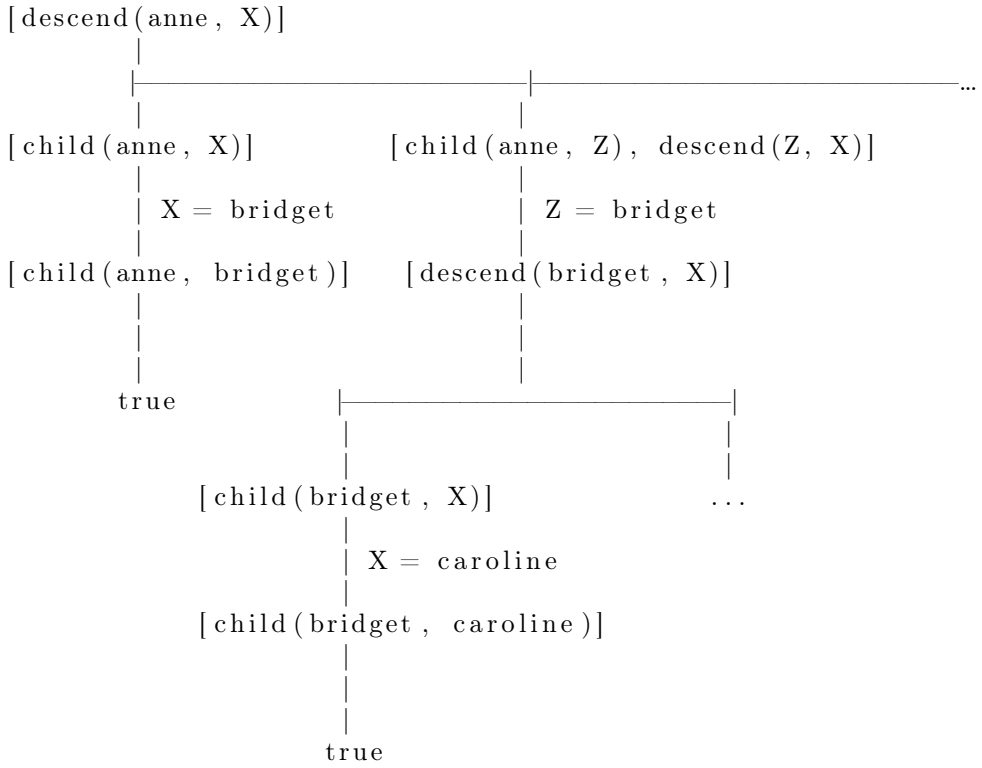
```
descend(X, Y) :- child(X, Y).
```

Je kan meteen zien dat dit alleen iets oplevert als X een direct kind is van Y, zonder verder te kijken.

```
1. child(anne, bridget).
2. child(bridget, caroline).
3. child(caroline, donna).
4. child(donna, emily).
5. descend(X,Y) :- child(X,Y).
6. descend(X,Y) :-
       child(X,Z),
       descend(Z,Y).
```

In Prolog werkt recursie met een base-case en een recursie. Als het eerste faalt dan probeert het programma het tweede uit te voeren. In het voorbeeld van de descendants is de base-case dat X het kind is van Y. Als dit niet het geval is moeten we een Z introduceren, een tussenpersoon. Stel dat X een kind is van Y, maar Y is niet de ouder (X->Z->Y). Dan kan eventueel Z de ouder zijn van X, en Y de ouder van Z. We zouden dit manueel kunnen beschrijven en veel regels toevoegen voor elke niveau van diepte maar dat is natuurlijk niet de bedoeling. In bovenstaande code zie je op regel 5 de base-case, die voor de recursie komt. Het programma probeert regel 5 te voldaan. Indien het niet mogelijk is, gaat het verder naar regel 6. Merk op dat we opnieuw hetzelfde predicaat oproepen voor de tussenpersoon. En als die tussenpersoon nog een tussenpersoon heeft dan gaat de recursie nog een niveau dieper. Het programma gaat alle mogelijke combinates proberen die voldoet aan deze regels. Als we uiteindelijk de query $descend(anne, emily)$ stellen krijgen we het resultaat *true*.

Merk op dat depth-first search in Prolog automatisch met recursie werkt. Zie volgende search-tree:



Deze search-tree heb ik heel hard ge-simplificeert omdat het anders te groot gaat worden. De bedoeling is om aan te tonen dat het programma automatisch recursief is door de depth-first search van Prolog. Visueel kan je m.b.v. de search-tree zien dat het programma meerdere goals probeert te bereiken en telkens teruggaat om andere resultaten te berekenen. Het programma gaat ook bepaalde niveau's diep om deze goals te bereiken.

2.8 Practical

De practical van Prolog heb ik gemaakt om zo veel mogelijk Prolog features in 1 programma te gebruiken om de taal goed onder te knie te krijgen. Hiervoor ben ik vertrokken van de basis van een van de kleinere voorbeelden. Het gaat over het voorbeeld waarbij ik een macht doe van iets en het resultaat tussentijds opsla voor eventueel later gebruik (hoofdstuk 2.3). Dit wordt gedaan om niet telkens opnieuw dezelfde berekening te hoeven doen. Zo kan het programma eerst kijken of het resultaat eerder werd berekend en dan pas zelf berekenen.

Prolog - practical

```
:- use_module(databases).
:- use_module(files).

powAndStore(X, Y, Result, File) :-
    pow(X, Y, Result),
    atomics_to_string([X,Y,Result], ", ", FormattedResult),
    appendInFile(FormattedResult, File).
```

```
% This overrides pow/3 in module databases
pow(X, Y, Result) :-
    open('results.txt', read, Stream),
    repeat,
    (
        read_line_to_string(Stream, Line0),
        Line0 \== end_of_file ->
            format("~w\n", [start]),
            Line0 = Line,
            % Split the 3 numbers from the read line
            split_string(Line, ", ", ", ", SubStrings),
            % Seperate the 3 numbers
            first(SubStrings, First),
            second(SubStrings, Second),
            third(SubStrings, Third),
            format("~w\n", [First == X]), format("~w\n", [Second == Y]),

            number_string(FirstNumber, First),
            number_string(SecondNumber, Second),
            number_string(ThirdNumber, Third),

            ((FirstNumber == X, SecondNumber == Y) ->
                Result is ThirdNumber, format("~w\n", [result]);
                format("~w\n", [else]),!)
        ; format("~w\n", [else2]), !, fail

    ),
    close(Stream).
    % format("Result from file: ~w\n", [Result]),

% Retrieve first/second/third value in a 3N list
first([E,_,_], E).
second([_,E,_], E).
third([_,_,E], E).
```

In deze practical gebruik ik meerdere Prolog features:

- Modules.
- Asserts.
- Bestanden lezen en schrijven.
- Wiskundige operaties.
- Manipulatie van lijsten.

De bedoeling hiervan is om het voorbeeld van het berekenen van machten uit hoofdstuk 2.3 uit te breiden zodat die het tussenresultaat niet alleen intern opslaat maar ook in een bestand. Dit doe ik omdat het interne geheugen niet permanent is. Bij een *assert* gaat het programma dynamisch een regel in het interne geheugen opslaan maar dit gaat verloren als we uit de Prolog-omgeving gaan. Dit is in tegenstelling tot facts die vast in een Prolog-bestand geschreven zijn.

Prolog module database:

```
:- module(database, [pow/3]).
:- dynamic pow_result(X, Y, Result).

pow(X, Y, Result) :-
    pow_result(X, Y, Result),
    format("Result retrieved from database: ~w\n", [Result]),
    !.
```

```
pow(X, Y, Result) :-
    Result is X^Y,
    format("Result calculated: ~w\n", [Result]),
    assert(pow_result(X, Y, Result)).
```

Hiervoor gebruik ik eerst en vooral de modules genaamd *databases* en *files*. Het doel is deze samen te gebruiken en er een uitbreiding van te maken. Vervolgens overschrijf ik (of de Engelse term "override") het predicaat genaamd *pow* zodat die een bestand gaat openen en hier tussentijdse resultaten gaat ophalen. Dit bestand bevat regels met telkens 3 argumenten gescheiden door een komma: het getal, de macht en de uitkomst. Dan gaat het programma lijn per lijn lezen en die verwerken. Als doel gaat het programma de uitkomst vinden in het bestand op basis van de input die we geven.

Prolog module files:

```
:- module(files, [appendInFile/2]).

appendInFile(Text, File) :-
    open(File, append, Stream),
    write(Stream, Text),
    nl(Stream),
    close(Stream).
```

Aanvullend is er nog het *powAndStore*/4 predicaat. Merk op dat de eerste regel hiervan het *pow*/3 predicaat gebruikt uit de module *database*. Dit is om de normale werking ervan te behouden. Als extra gaan we die dus nog wegschrijven in een bestand.

results.txt:

```
2, 11, 2048
2, 15, 32768
```

De moeilijkheid in deze practical lag bij het werken met verschillende types van data. Het bestand dat ik inlees is een string. De nummers in dit bestand moeten omgezet worden zodat er mee gerekend kan worden. Als oplossing hiervoor heb ik het $number_string/2$ predicaat gebruikt die ingebouwd is in Prolog. Deze doet voornamelijk wat het zegt en kan een string omzetten naar een nummer waarmee we kunnen rekenen en vice versa.

Prolog - uitvoer:

```
?- pow(2,11,X).
X = 2048
```

Als uitbreiding van deze practical is het eventueel mogelijk om de tussenresultaten in een Prolog-bestand te schrijven i.p.v. een gewoon tekstbestand. Op die manier zouden we dit bestand rechtstreeks in Prolog kunnen inladen en hoeven we niet constant met strings te werken en de omzetting ervan naar nummers.

Hoofdstuk 3

Sequence Datalog

Sequence Datalog is een querytaal voor sequentie-databases. We kunnen data extracten en manipuleren op bepaalde relaties. Sequenties worden opgebouwd om dit te verwezenlijken. Dit is eerder een theoretische taal en niet echt uitvoerbaar in tegenstelling tot bijvoorbeeld Prolog. De papers die ik heb ingestudeerd zijn [Expressiveness within Sequence Datalog] met de bijbehorende presentatie [Presentation Expressiveness within Sequence Datalog], door Heba Aamer, Jan Hidders, Jan Paredaens en Jan Van den Bussche. Ook heb ik [Sequences Datalog en Transducers] bekeken door Bonner en Mecca.

Om Sequence Datalog te kunnen laten werken met sequentiële data is er een concatenatie-operator nodig om termen te kunnen opstellen die sequentiële variabelen te kunnen voorstellen als een sequentie. Deze termen zijn de zogenaamde path expressions en de sequentiële variabelen zijn de path variables. Dit leg ik uit in hoofdstuk 3.1.

3.1 Syntax

In Datalog worden database relaties bekeken als predicaten. Bijvoorbeeld $R(X)$ beschrijft een variabele X die in relatie R zit. Vervolgens kunnen we atoms of paden concateneren m.b.v. het dot (.), zoals het dot-product. Bijvoorbeeld $S(X.Y) \leftarrow R(X), R(Y)$ heeft twee variabelen X en Y die zich in het R-domein bevinden. Als oplossing wordt de concatenatie van die twee gegeven.

Path expressions in Sequence Datalog zijn terms, of termen, die geconcateneerd worden. De concatenatie-operator is een dot. Deze concatenatie is een path expression en kan voorgesteld worden in Datalog. Het probleem is dat we dit ook willen in Prolog. De reden hiervoor is omdat we dan problemen op een andere manier kunnen oplossen, dus met behulp van path expressions. Niet alleen hiervoor, maar ook om een implementatie te maken van Sequence Datalog. In prolog bestaat er geen directe manier.

$$R(e_1, e_2, ..., e_n)$$

Bovenstaand syntax-voorbeeld is een predicaat in de vorm van een expression in relatie R, met daarin path expressions.

Path variables, genoteerd door $\$x$, zijn de sequentiële variabelen die kunnen deel uitmaken van een path expression. Er is dus een concatenatie-operator nodig die deze path variables kunnen verbinden om path expressions op te stellen. Dit is nodig in

Sequence Datalog om met sequentiële data te kunnen werken. De prefix van een path variable is een dollar-teken.

Atomic variables, genoteerd door $@x$, is een enkele atom of teken van lengte 1. In Prolog kan dit een atom, character, SWI-Prolog string of een number zijn. In Sequence Datalog volgens de paper [Expressiveness within Sequence Datalog] is een atomic value data elementen en elke atomic value is een value.

Packed values of packing, genoteerd door $< a >$, werd geintroduceerd in [J-Logic] om subsequenties als atomic values te beschouwen. Een packed-value kan uiteindelijk opengemaakt worden.

Constant values is simpelweg het letterlijke teken zoals x. Het verschil met een atomic value is dat de atomic value $@x$ eender welk teken kan zijn en het constant value x is een x.

De syntax wordt als volgt beschreven volgens [Expressiveness within Sequence Datalog]:

- Elke atomic value is een path expression.
- Elke variable is een path expression.
- Als e een path expression is dan is $< e >$ een path expression.
- Elk eindige sequentie van path expressions is een path expression gescheiden door dots.

Volgens [Expressiveness within Sequence Datalog] beschrijven volgende regels het Sequence Datalog data model:

- Elke atomic value is een value.
- Elk eindige sequentie van values is een path. De lege path is genoteerd door ε. Paths worden uit elkaar gehaald met het dot-teken, wat ook de concatenatie ervan betekent. Dit is associatief.
- Als p een path is dan is $< p >$ een packed value.
- Elke packed value is een value.

Als voorbeeld als a, b en c atomic values zijn dan is $a.b.a$ een path. $< a.b.a >$ is een packed value en $c. < a.b.a >$ is terug een path.

3.2 Concatenatie in Sequence Datalog

De concatenatie zoals getoond in sectie 4.1 gaat een grote impact spelen op hoe we dit gaan implementeren. De concatenatie in Sequence Datalog is genoteerd met een dot. De dot verbindt het linkerdeel met het rechterdeel. Dit is vergelijkbaar met een append in andere programmeertalen. Merk op dat in Sequence Datalog dit niet alleen gebruikt wordt voor concatenatie maar ook voor data extractie. De termen die we verbinden noemen path expressions. Voor meer informatie hierover zie hoofdstuk 3.1. Voorbeelden volgen.

```
$x . $y
```

Bovenstaande code is de concatenatie van in dit geval 2 path variables. Op zich laat het niet veel zien, behalve de schrijfwijze ervan, omdat we geen input hebben. Stel dat de input $[a, b, c]$ is en we passen bovenstaande path expression toe. Het programma gaat zoals altijd alle mogelijke oplossingen proberen te vinden. In dit geval zijn er 4 mogelijke oplossingen. Merk op dat 1 van de path variables leeg kan zijn en de andere de hele input! Alle mogelijke oplossingen volgen:

```
X = [] ,
Y = [a, b, c] ;
X = [a] ,
Y = [b, c] ;
X = [a, b] ,
Y = [c] ;
X = [a, b, c] ,
Y = [] .
```

Vervolgens kunnen we ook aan data extractie doen. Behalve path variables kunnen we ook atomic variables, constante waarden of packed values gebruiken.

```
@z . $x
```

Stel dat we bovenstaande path expression hebben en we gebruiken dezelfde input als ervoor: $[a, b, c]$. De atomic variable $@z$ krijgt als waarde a en de path variable $\$x$ krijgt als waarde $[b, c]$. Dit is dan ook de enige mogelijke combinatie, dus ook de enige oplossing. Dit is handig omdat we op die manier een atomic variable uit de originele input kunnen halen, daar iets mee doen, en dan de rest van de path kunnen gebruiken in eventuele recursie. Dit gaat blijken uit verdere voorbeelden in hoofdstuk 3.3.

3.3 Voorbeelden

Sequence Datalog: voorbeeld omgekeerd pad:

```
1. T($x, e) <- R($x).
2. T($x, $y.@u) <- T($x.@u, $y).
3. S($x) <- T(e, $x)
```

Bovenstaand voorbeeld komt uit de paper [Expressiveness within Sequence Datalog]. Dit Sequence Datalog programma neemt een path $\$x$ en gaat die omkeren. Merk op dat deze path in relatie met R is en als uitkomst bekomen we een relatie met S. We kijken telkens eerst naar de body voordat we de head gaan toepassen. De eerste regel neemt een path $\$x$ uit relatie R. In de head gaan dan een nieuwe predicaat toepassen in relatie T die twee argumenten bevat. Het eerste argument is $\$x$ en is dus gewoon de

input die we hadden verkregen in de body. Het tweede argument is e dat staat voor een lege lijst, waarbij de letter e voor empty staat. In de body van regel 2, in het eerste argument, gaat het programma $\$x$ concateneren met $@u$. Dit wil zeggen dat we een oplossing zoeken zodat de input, in ons geval de initiele path $\$x$, matcht met deze path expression. De enige manier waarop dit matcht is als $\$x$ de waarde krijgt van de initiële $\$x$ zonder het laatste element. De reden hiervoor is omdat het laatste element de path variable $@u$ moet zijn om dit te laten kloppen. Het tweede argument is y, hier gaan we in de head meer mee doen. Een voorbeeld met de input als $[a, b, c]$:

```
regel 1 body: R($x) = R([a,b,c])
regel 1 head: T($x, e) = T([a,b,c], [])
regel 2 body: T($x.@u, $y) = T([a,b].[c], [])
...
```

Nu begint het omdraaien van de lijst. In de head van regel 2 is het eerste argument $\$x$. Merk op dat deze in de body veranderd werd, want we hadden hier het laatste element uitgehaald. Dit specifiek argument zorgt er ook voor dat het programma eventueel gaat stoppen. Want als je kijkt naar regel 3, de body ervan verwacht een lege lijst in het eerste argument. Dit gaat uiteindelijk ook gebeuren. Als we nu teruggaan naar regel 2, het tweede argument gaat die laatste element die we er eerder hadden uitgehaald concateneren met de initieel lege lijst. Het programma gaat dit ook blijven doen waardoor uiteindelijk de lijst omgedraaid wordt. Merk op dat de $\$x$ uit regel 3 een beetje verwarrend is omdat dit eigenlijk de $\$y$ is uit de vorige stappen, maar die heeft gewoon een andere naam gekregen. Het eerdere voorbeeld uitgewerkt:

```
regel 1 body: R($x) = R([a,b,c])
regel 1 head: T($x, e) = T([a,b,c], [])
regel 2 body: T($x.@u, $y) = T([a,b].[c], [])
regel 2 head: T($x, $y.@u) = T([a,b], [].[c])
regel 2 body: T($x.@u, $y) = T([a].[b], [])
regel 2 head: T($x, $y.@u) = T([a], [c].[b])
regel 2 body: T($x.@u, $y) = T([].[a], [c,b])
regel 2 head: T($x, $y.@u) = T([], [c,b].[a])
regel 3 body: T(e, $x) = T([], [c,b,a])
regel 3 head: S($x) = S([c,b,a])
```

We komen dus uit op een predicaat in relatie S met als inhoud 1 argument waarbij de initiële lijst omgekeerd is. Dit voorbeeld ga ik in hoofdstuk 4.3.3 omzetten in Prolog gebruikmakend van de eigen implementatie van path expressions.

Hoofdstuk 4

Implementatie van Sequence Datalog in Prolog

Het probleem dat we willen bekijken in Prolog is een implementatie van Sequence Datalog zodat het uitvoerbaar wordt. Ideaal zou ik een compiler van Sequence Datalog naar een uitvoerbare taal kunnen maken. Gezien de affiniteit tussen Prolog en Sequence Datalog is Prolog de voor de hand liggende keuze om Sequence Datalog naar te compileren. Prolog is ook geschikt omdat die lijsten ondersteund. Dit gebruik ik om de sequenties mee te implementeren.

Bij het ontwikkelen van zulke compiler zal de hoofdmoeilijkheid zijn hoe we rules met path expressions zouden vertalen naar Prolog-rules. Daarbuiten gebruiken we enkel standaard compilatietechnieken zoals lexicale analyse, parsing enzovoort. Ik wil mij focussen op de hoofdmoeilijkheid. Eerst en vooral moeten we de methode opstellen. Eens die bekend is door de programmeurs kan die rechtstreeks Sequence Datalog schrijven in Prolog. Een compiler is hier zelfs niet voor nodig.

De voorstelling van een Sequence Datalog programma en manipulatie van sequenties moet ik kunnen omschrijven in deze methode. Omdat Prolog niet direct path expressions ondersteunt wil ik een manier vinden om dit toch te doen. De bedoeling is ook om er een elegante manier voor te vinden, dus niet zomaar een oplossing. Ook is het doel dat dit goed werkt en conclusies uit gemaakt kan worden en of het niet goed werkt ook weten waarom. Daarom zijn hier ook, behalve kleine oefeningen, ook grotere toepassingen voor nodig die ik later bespreek in hoofdstuk 6 voor genoom-data en in hoofdstuk 5 voor workflow data.

Sequence Datalog werkt met sequenties en Prolog werkt met lijsten. Voor de implementatie om Sequence Datalog programma's te simuleren in Prolog moeten we kijken hoe we de sequenties, zoals die gekend zijn in Sequence Datalog, hoe we die in Prolog kunnen voorstellen. Voor dit te realiseren heb ik voornamelijk gewerkt met lijsten. Sequenties worden dus voorgesteld als lijsten. Voorbeelden van dit zijn eerder terug te vinden in sectie 4.1.

Vooraleer direct over te gaan naar de implementatie hiervan in Prolog heb ik eerst een aantal programma's in Sequence Datalog gemaakt in standaard Prolog. De bedoeling hiervan is om aan te tonen dat Sequence Datalog elegantere programma's toelaat met behulp van path expressions. De path expressions in Prolog zijn verdwenen of onherkenbaar in de programmacode. De vraag die we moeten beantwoorden is dus of we deze

path expressions direct kunnen ondersteunen in Prolog? Daarom heb ik vervolgens nagedacht over een mogelijke implementatie om path expressions directer te ondersteunen in Polog. Hiervoor heb ik dus eerst Prolog en Sequence Datalog apart moeten studeren zoals eerder besproken in hoofdstukken 2 en 3.

4.1 Path Expressions als lijsten

De bedoeling is om te kijken hoe we path expressions kunnen voorstellen in Prolog. Path expressions zoals $@a.\$x$ is fijn maar dit bestaat niet standaard in Prolog. We zouden dus een manier moeten vinden om dit te simuleren met lijsten, of we zouden op een of andere manier een predicaat hiervoor kunnen gebruiken. De reden dat we lijsten hiervoor nodig hebben is omdat we de syntax van path expressions moeten kunnen voorstellen. Dit kunnen we bekomen door te werken met lijsten. De reden hiervoor is omdat de features die Prolog biedt voor het werken met lijsten iets heeft wat we kunnen gebruiken. Hier spreek ik over de concatenatie in Sequence Datalog die we kunnen bereiken met behulp van de concatenatie van lijsten in Prolog. In Prolog is er predicaat genaamd $append/3$[1]. De werking van deze predicaat heb ik eerder uitgelegd in hoofdstuk 2.4. Deze speelt een belangrijke rol in de implementatie, uitgelegd in hoofdstuk 4.2.

Path Expr e	**Lijst rep(e)**
a	$[a]$
$\$x$	$[pathV(X)]$
$@x$	$[atomicV(X)]$
$e_1 \cdot e_2$	$rep(e_1) \cdot rep(e_2)$
$< e >$	$[pack(rep(e))]$

Voorbeeld: <$x>.$y -> [pack([pathV(X)]), pathV(Y)]

Path expressions kunnen we voorstellen als lijsten op bovenstaande manier. De linkerkolom is hoe we dit in Sequence Datalog zouden schrijven en de rechterkolom hoe we dit kunnen vertalen in de vorm van een lijst. Waarom lijsten? Omdat we dan path expressions kunnen gebruiken in Prolog. Rep wil zeggen dat het de representatie is van e. Dit is een conceptuele representatie. De concatenatie wordt voorgesteld met de dot, zoals $e_1.e_2$ Dit volgt in de volgende sectie. Deze manier van schrijven van path expressions ga je ook terugzien in de implementatie van path expressions in Prolog op een soortgelijke manier. Hiervoor refereer ik naar hoofdstuk 4.2 waar ik de implementatie introduceer.

[1] De 3 staat voor de arity. Dus het predicaat append heeft 3 argumenten.

4.2 Path Expressions in Prolog - Introductie Expr-Predicaat

```
1.  expr([], []).

    % path value
2.  expr(L, [pathV(A1)|ERest]) :-
3.      append(A1, LRest, L),
4.      expr(LRest, ERest).

    % packed value
5.  expr(L, [pack(X) | ERest]) :-
6.      L = [H|LRest],
7.      H = pack(L1),
8.      expr(L1, X),
9.      expr(LRest, ERest).

    % atomic variable
10. expr(L, [atomicV(A1) | ERest]) :-
11.     L = [A1 | LRest],
12.     atom(A1),
13.     expr(LRest, ERest).

    % constant value
14. expr(L, [A1 | ERest]) :-
15.     atom(A1),
16.     L = [A1 | LRest],
17.     expr(LRest, ERest).
```

Bovenstaande code is de implementatie van path expressions in Prolog genaamd het expr-predicaat. Hiermee kunnen we path expressions voorstellen zoals in Sequence Datalog. Dit neemt 2 argumenten, een lijst dat de data representeert en de opgebouwde path expression. De bedoeling is dat we deze path expression op kunnen bouwen op ons eigen manier! Het programma genereert alle mogelijke resultaten op basis van de meegegeven argumenten.

Regel 1 is de base-case en zorgt ervoor dat het programma stopt in de recursie als de lijsten leeg zijn. De code is ingedeeld in vier predicaten (regel 2, 5, 10 & 14). In de head van elk van deze wordt gematcht met welke soort data we te maken hebben. Als je kijkt naar het tweede argument van elk van deze vier predicaten dan merk je op dat de lijst wordt opgedeeld in de head van de lijst (of het eerste element), gevolgd door de rest van de lijst. Het type van dit element bepaalt dus met welke regel een match wordt gemaakt.

Vanaf regel 3 is het geval van een **path value**. Path values worden in Sequence Datalog voorgesteld door $\$x$. In deze implementatie wordt dit voorgesteld als $pathV(X)$. Merk op dat in het tweede geval we een hoofdletter schrijven omdat dit de schrijfwijze is voor een variabele in Prolog. Bij een path value verwachten we de inhoud in de vorm van een lijst. In regel 3 gaan we dus die lijst concateneren met een nog onbekende variabele *LRest* zodat we L uitkomen, onze input. De *LRest* variabele krijgt dus de waarde die $A1$ en *LRest* verbindt om L te bekomen. Ten slotte gaat het programma verder met de recursie in regel 4 waarbij de *LRest* en de *ERest* wordt meegenomen, dus exclusief de path value $A1$ die we net hebben verwerkt. Als we bijvoorbeeld $[a, b]$ als input hebben,

dus $L = [a, b]$, en de path expression als tweede argument is $[pathV(X), pathV(Y)]$, dan genereert dit drie mogelijke resultaten. Twee resultaten zijn al meteen duidelijk als X een lege lijst is ($[]$) dan is Y de hele oplossing $[a, b]$. En visa-versa in het geval dat Y leeg is. Als laatste oplossing dat $X = [a]$ en $Y = [b]$.

4.2.1 Van Sequence Datalog naar Prolog

Sequence Datalog:

```
1. T(e, $x, $x) <- R($x).
2. T($y.$x, $x, $z) <- T($y, $x, a.$z).
3. S($y) <- T($y, $x, e).
```

In dit hoofdstuk ga ik stap voor stap uitleggen hoe van een Sequence Datalog programma naar een Prolog programma te gaan m.b.v. het expr-predicaat. Voor dit gebruik ik het voorbeeld dat van "invoer macht n" die ik later in meer detail bespreek in hoofdstuk 4.3.2. Voorlopig is het niet belangrijk om het programma te begrijpen zodat we puur focussen op het implementeren naar Prolog m.b.v. het expr-predicaat zonder te veel bij de werking van het programma stil te staan. Ten eerste verwacht het programma dat we een path-variable hebben in relatie met R. We hebben dus eerst een of meerdere regels nodig die een lijst in relatie met R heeft als volgt:

```
r([a,a]).
r([a,a,a]).
r([a,a,a,a]).
```

Dan beginnen we met de eigenlijke omzetten van regel 1. In Prolog is de path-variable $\$x$ voorgesteld als een lijst. We vragen dit simpelweg op in prolog met $r(X)$. Merk op dat we in prolog relaties met kleine letter schrijven en variabelen met hoofdletter. De head van regel 1 heeft 3 parameters en is simpelweg een lege lijst en de path-variable X twee keer. Hier is geen expr-predicaat nodig omdat we nergens een concatenatie of dergelijke doen. Regel 1 ziet er dus als volgt uit:

```
t([], X, X) :- r(X).
```

Regel 2 is iets complexer, we beginnen met de body of het rechter gedeelte. De body verwacht drie parameters in relatie met T. Dit definiëren we als eerst. Merk op dat de derde parameter iets speciaals is dat we niet standaard in Prolog kunnen doen. Hier komt dus het expr-predicaat aan te pas. We beginnen met simpelweg de variabelen te definiëren met $t(Y, X, T1)$. We gaan $T1$ zelf moeten definiëren m.b.v. path expressions. $T1$ representeert dus $a.\$z$. In Prolog m.b.v. het expr-predicaat is dit als volgt: $expr(T1, [a, pathV(Z)])$. Hier verwachten we dus dat $T1$ een path expression is met een constante a gevolgd door een path-variable Z. We bekomen het volgende:

```
t([], X, X) :- r(X).
... :-
    t(Y, X, T1),
    expr(T1, [a, pathV(Z)]),
```

We zijn hier nog niet klaar mee want de head of het linkergedeelte moet nog gedefinieerd worden. Net zoals de body iets speciaals had, heeft de head dit ook. We gaan op soortgelijke manier te werk. De eerste parameter moeten we nog definiëren en noemen we $T2$. Deze stap kunnen we in de body doen. De andere variabelen kennen we al en kunnen we gewoon overnemen. De head wordt dus $t(T2, X, Z)$. $T2$ definiëren doen we

op dezelfde manier zoals ervoor. $T2$ die gerepresenteerd is als $y.\$x$ schrijven we als volgt: $expr(T2, [pathV(Y), pathV(X)])$. We bekomen:

```
t([], X, X) :- r(X).
t(T2, X, Z) :-
    t(Y, X, T1),
    expr(T1, [a, pathV(Z)]),
    expr(T2, [pathV(Y), pathV(X)]).
```

Als laatste moeten we regel 3 nog schrijven, hiervoor hebben we het expr-predicaat niet voor nodig. Deze regel matcht als de derde parameter een lege lijst is. Dit definiëren we in de body of rechter gedeelte als volgt: $t(Y, X, [])$. Het hele programma is als volgt:

De Prolog versie ondersteund door het expr-predicaat:

```
r([a,a]).
r([a,a,a]).
r([a,a,a,a]).

t([], X, X) :- r(X).
t(T2, X, Z) :-
    t(Y, X, T1),
    expr(T1, [a, pathV(Z)]),
    expr(T2, [pathV(Y), pathV(X)]).
success(Y) :-
    t(Y, X, []).
```

4.3 Bijkomende voorbeelden

4.3.1 a-exclusief

De opgave van a-exclusief is een slimme manier om met behulp van vergelijkingen te kijken of alle elementen in de sequentie alleen maar uit a's bestaan. Deze opgave heb ik eerst in normaal Prolog opgelost en ten slotte de Sequence Datalog versie gesimuleerd naar Prolog met behulp van het expr-predicaat.

Sequence Datalog:

```
S($x) <- R($x), a.$x = $x.a
```

Standaard Prolog versie:

```
excl([]).
excl(X) :-
    head(X, XHead),
    last(X, XLast),
    without_first(X, XWithoutFirst),
    without_last(X, XWithoutLast),
    without_last(X, [_|XWithoutLastOrFirst]),
    XHead = a, XLast = a, XWithoutFirst = XWithoutLast,
    excl(XWithoutLastOrFirst).
```

Expr-predicaat met path expressions:

```
R = [a,a],
expr(R, [pathV(X)]),
expr(L, [a, pathV(X)]),
expr(L, [pathV(X), a]).
```

De input voor dit programma is een sequentie van letters. In het linkergedeelte neemt het programma de eerste letter gevolgd door de input zelf. Een aantal outputs volgen:

```
?- R = [a,a,a,a,a,a],
expr(R, [pathV(X)]),
expr(L, [a, pathV(X)]), expr(L, [pathV(X), a]).
R = X, X = [a, a, a, a, a, a],
L = [a, a, a, a, a, a, a] ;

?- R = [a,a,a,b,a,a,a],
expr(R, [pathV(X)]),
expr(L, [a, pathV(X)]), expr(L, [pathV(X), a]).
false.
```

Merk op dat de uitvoer niet gewoon *true* of *false* is. Dat komt omdat het programma teruggeeft wat de onbekende variabelen zijn voor de bijbehorende oplossing.

4.3.2 Invoer macht n

```
input: [a,a,a], output: [a,a,a,a,a,a,a,a,a]
```

Sequence Datalog:

```
1. T(e, $x, $x) <- R($x).
2. T($y.$x, $x, $z) <- T($y, $x, @a.$z).
3. S($y) <- T($y, $x, e).
```

Deze oefening neemt een input in de vorm van een path-variable. De macht van een sequentie wil zeggen de sequentie zoveel keer herhalen. Het programma gaat in het algemeen $\$x^n$ doen met $\$x$ de sequentie en n de lengte van de sequentie. Dit kunnen we ook zien als $n.l$ met l de lengte van de sequentie $\$x$ waarbij $n = l$. Om het direct duidelijk te maken geef ik hier een aantal uitvoeringen:

Uitvoer van "invoer macht n" op verschillende inputs waarbij X de input is en Y de output:

```
?- a_macht_n(X).
Y: [a,a,a,a]
X = [a, a] ;
Y: [a,a,a,a,a,a,a,a,a]
X = [a, a, a] ;
Y: [a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a]
X = [a, a, a, a].
```

X is hier telkens de input en Y is de bijbehorende output. Elke output is gescheiden door een puntkomma. De werking van het Sequence Datalog programma is als volgt. Regel 1 neemt een path-variable in relatie met R. Vervolgens in de head nemen we e, een lege lijst, gevolgd door de lijst twee keer.

Dan komen we terecht in de body dus rechterzijde van regel 2. De derde parameter haalt een constante waarde a uit de oorspronkelijk lijst $\$x$. De variabele $\$z$ is dus de originele input zonder de eerste letter. In de head wordt de $\$z$ alleen meegeven, dus exclusief de eerste letter. De eerste parameter was oorspronkelijk een lege lijst en in de body van regel 2 verandert hier nog niks aan en krijgt het de variabele $\$y$.

In de head concateneren we dit met de hele input. De lege lijst wordt dus in eerste instantie de hele input. De tweede parameter blijft simpelweg de originele input en we behouden dezelfde naam ervoor. Dit blijven we constant herhalen tot de body van regel 3 geldt. Dit kunnen we zien aan de derde parameter van regel 3, die verwacht dat de laatste parameter leeg is! Dus het programma neemt telkens de eerste letter van de input weg (derde parameter) tot die leeg is, en concateneert telkens de hele input (eerste parameter) aan elkaar.

Uiteindelijk geraakt de derde parameter leeg en komen we aan regel 3. Regel 3 verwacht dat de derde parameter leeg is terwijl in de head we de eerste parameter meegeven. Dit is dan ook het resultaat. Want we hebben het aantal keer dat we de a hebben weggehaald er de hele input aan toegevoegd. In standaard Prolog zou deze implementatie er zo kunnen uitzien:

Implementatie sequentie macht n in standaard Prolog:

```
ex5_1(X) :-
    r(X),
    ex5_1([], X, X).
ex5_1(A, X, []) :-
    format("Y: ~w\n", [A]), !.
ex5_1(A, X, Z) :-
    [_|T] = Z,
    append(A, X, YappendedX),
    ex5_1(YappendedX,X,T).
```

Je kan al opmerken dat de implementatie niet erg lijkt op die van Sequence Datalog. De path expressions zijn hier niet zichtbaar of gaan zelfs verloren. Hier gebruiken we standaard Prolog implementaties om de head van de lijst uit te halen en te matchen met een a. Vervolgens doen we een append om die nieuwe variabelen op te stellen. Dit is nog een redelijk kleine oefening maar het is duidelijk dat zelfs hier de originele Sequence Datalog path expressions niet meer zichtbaar zijn. Deze oefening zet ik om in Prolog m.b.v. het expr-predicaat en leg ik stap voor stap uit in hoofdstuk 4.2.1. De implementatie ervan volgt:

Implementatie sequentie macht n in Prolog m.b.v. path expressions:

```
t([], X, X) :- r(X).
t(T2, X, Z) :-
    t(Y, X, T1),
    expr(T1, [atomicV(A), pathV(Z)]),
    expr(T2, [pathV(Y), pathV(X)]).
success(Y) :-
    t(Y, X, []).
```

Als je bovenstaande code naast de Sequence Datalog versie zet kan je meteen zien dat ze een beetje overeenkomen. Ook zijn de path expressions nog volledig zichtbaar. De ondersteuning van path expressions in Prolog zorgt er niet alleen voor dat de path expressions nog zichtbaar zijn, maar ook om de vertaling van een Sequence Datalog programma te vergemakkelijken. Ten slotte ziet dit ziet er properder uit dan de implementatie in standaard Prolog. Later leg ik twee toepassingen uit en gaat dit nog duidelijker worden met grotere implementaties (zie hoofdstuk 5 en hoofdstuk 6).

4.3.3 Omgekeerde lijst

Sequence Datalog omgekeerde lijst:

```
1. T($x, e) <- R($x).
2. T($x, $y.@u) <- T($x.@u, $y).
3. S($x) <- T(e, $x)
```

Het voorbeeld van omgekeerde lijst werd eerder in detail besproken in hoofdstuk 3.3. Bovenstaand heb ik die opnieuw toegevoegd. Dit voorbeeld heb ik, toen ik nog Prolog aan het leren was, in standaard Prolog gemaakt dus nog zonder de implementatie van het expr-predicaat. Dit was een uitdaging omdat je niet alleen het Sequence Datalog programma moest begrijpen maar ook goed overweg moet kunnen met Prolog om dit om te zetten. Zoals eerder besproken zijn path expressions niet standaard inbegrepen in Prolog waardoor dit iets moeilijker maakt. Uiteindelijk is de code wel kort maar het had enige tijd geduurd vooraleer dit tot stand is gekomen. De code volgt:

Standaard Prolog omgekeerde lijst:

```
rev([], A).
rev(X, A) :-
    last(X, XLast),
    append(A, [XLast], L),
    without_last(X, XWithoutLast),
    rev(XWithoutLast, L).
```

Het is zeker mogelijk dat dit niet de perfecte implementatie hiervoor is. Er zijn veel verschillende manieren om dit op te lossen in standaard Prolog. Voor deze implementatie heb ik een paar helper-predicaten gebruikt. Het predicaat *last*/2 is vanzelfsprekend en gaat het tweede argument unificeren met de laatste waarde uit de lijst van het eerste argument. Het predicaat *without_last* gaat het tweede argument unificeren met alle elementen uit het eerste argument, behalve het laatste. Het programma stopt omdat we telkens een waarde uit de initiële input halen en er recursie mee gaan doen. Uiteindelijk wordt deze lijst leeg en wordt de eerste regel voldaan waardoor het programma stopt. Om dit programma stap voor stap aan het werk te zien heb ik een voorbeeld uitvoer van drie stappen toegevoegd:

```
?- X = [a,b,c], A = [], last(X, XLast),
append(A, [XLast], L), without_last(X, XWithoutLast).
X = [a, b, c],
A = [],
XLast = c,
L = [c],
XWithoutLast = [a, b] ;
false.

?- X = [a,b], A = [c], last(X, XLast),
append(A, [XLast], L), without_last(X, XWithoutLast).
X = [a, b],
A = [c],
XLast = b,
L = [c, b],
XWithoutLast = [a] ;
false.
```

```
?- X = [a], A = [c,b], last(X, XLast),
append(A, [XLast], L), without_last(X, XWithoutLast).
X = [a],
A = [c, b],
XLast = a,
L = [c, b, a],
XWithoutLast = [] ;
false.
```

In bovenstaande code toon ik een voorbeeld die een input van de lijst $[a, b, c]$ stap voor stap omkeert. Ik heb drie queries uitgevoerd met telkens de veranderde waarden zoals ze in de recursie zouden voorkomen. Telkens eindigen we met een $false$ om te laten zien dat er maar 1 oplossing is voor elke stap die we doen. De variabele X wordt telkens verkort met de laatste waarde. De variabele A heb ik zo genoemd omdat het staat voor de accumulator. De variabele L is de uiteindelijke oplossing. Na deze derde stap gaat het programma het predicaat rev nog 1 keer uitvoeren. Omdat de eerste variabele $XWithoutLast$ leeg is gaat het programma matchen met de eerste regel waardoor die ook stopt. Als uiteindelijke uitvoer krijgen we dus $[c, b, a]$.

Sequence Datalog omgekeerde lijst:

```
1. T($x, e) <- R($x).
2. T($x, $y.@u) <- T($x.@u, $y).
3. S($x) <- T(e, $x)
```

Prolog - Expr-predicaat met path expressions:

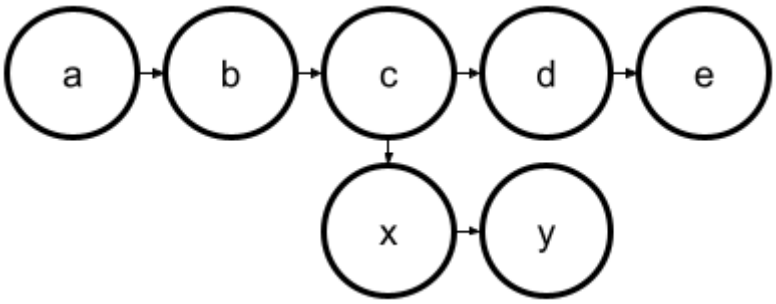
```
1. r([a,b,c]).

2. t(X, []) :- r(X).
3. t(X, T2) :- t(T1, Y),
4.       expr(T1, [pathV(X), atomicV(U)]),
5.       expr(T2, [pathV(Y), atomicV(U)]).
6. s(X) :- t([], X).
```

Nu volgt het omzetten van dit programma met behulp van het expr-predicaat en path expressions. Bovenstaande code is de implementatie ervan. Als je dit programma vergelijkt met de originele Sequence Datalog versie kan je meteen opmerken dat de path expressions zichtbaar blijven in deze versie van het Prolog programma. In hoofdstuk 4.2.1 leg ik in detail uit hoe je een Sequence Datalog programma omzet m.b.v. het expr-predicaat dus ik ga hier niet te veel in detail treden. Even kort voor dit voorbeeld is het duidelijk dat we twee path expressions moeten definiëren. Een ervan is voor de head van de tweede regel, het tweede argument, in de Sequence Datalog versie. Het andere zit in de body van dezelfde regel, het eerste argument. Die heb ik variabelen gegeven genaamd $T1$ en $T2$. Hoe die eruit moeten zien definieer ik in het expr-predicaat in bovenstaande regels 4 en 5. Als we dit uitvoeren komen we uit op volgende output:

Prolog - Output van s(X).:

```
?- s(X).
X = [c, b, a] .
```

4.3.4 Path Exists



Figuur 4.1: Directe Graaf

Het path-exists voorbeeld begint vanuit een direct graaf. In afbeelding 4.1 kan je zo'n directe graaf zien. In prolog wordt deze gedefinieerd met een aantal facts die elk twee argumenten hebben, namelijk de node waar de pijl vertrekt en de node waar die aankomt. De facts in Prolog voor de directe graaf zoals in afbeelding 4.1 is als volgt:

```
r([a, b]).
r([b, c]).
r([c, d]).
r([d, e]).
r([c, x]).
r([x, y]).
```

De reden dat die zich in lijsten bevinden is belangrijk om te weten. Dat komt omdat we zoals eerder besproken met lijsten werken in Prolog om path expressions te kunnen simuleren. Omdat deze in lijsten zijn kunnen we gebruik maken van het expr-predicaat. Deze opgave heb ik eerst in normaal Prolog opgelost en ten slotte de Sequence Datalog versie gesimuleerd naar Prolog met behulp van het expr-predicaat.

Sequence Datalog:

```
1. T(@x.@y) <- R(@x.@y).
2. T(@x.@z) <- T(@x.@y), R(@y.@z).
3. S <- T(a.y).
```

Standaard Prolog versie:

```
1. t([X,Y]) :- r([X,Y]).
2. t([X,Y]) :- r([X,Z]), t([Z,Y]).
```

Prolog - Expr-predicaat met path expressions:

```
1. t(R1) :- r(R1), expr(R1, [X, Y]).
2. t(T2) :- t(T1), r(R),
        expr(T1, [X, Y]), expr(R, [Y,Z]),
        expr(T2, [X,Z]).
3. success :- t([a,y]).
```

Prolog - Expr-predicaat met path expressions en tail recursion:

```
1. t(R1) :- r(R1), expr(R1, [X, Y]).
2. t(T2) :- r(R1), t(T),
        expr(R1, [X, Y]), expr(T, [Y,Z]),
        expr(T2, [X,Z]).
3. success :- t([a,y]).
```

Ik geef een korte omschrijving van programma met behulp van de Sequence Datalog versie. Om te bepalen of er een pad bestaat van x naar y kijkt de eerste regel eerst als er een directe pad bestaat van x naar y. Indien dat het geval is dan is de oplossing *true*. Anders gaat het programma verder naar regel 2. Regel 2 is recursief en die kijkt of er een pad bestaat van x naar y alsook van y naar z. Met andere woorden die kijkt of er een tussenpad bestaat die x en y verbindt. Deze recursie-stap leg ik later in meer detail uit.

De normale Prolog versie is redelijk simpel. Deze oefening dient dan ook niet om de vertaling te laten zien maar eerder een probleem dat is opgedoken met het expr-predicaat die ik nu ga bespreken. Daarom heb ik hier ook twee versies van, zonder en met tail-recursion, respectievelijk.

Om hier mee te beginnen zal ik direct de output laten zien van beide versies.

Output Path Exists:

```
?- success().
ERROR: Out of local stack
```

Output Path Exists met tail-recursion

```
?- success().
true
```

Probleem expr-predicaat

Hier duikt dus een probleem op met de recursie van het expr-predicaat in Prolog. Dit is de eerste keer dat dit probleem is opgedoken en ben hier ook dieper op ingegaan om

uit te zoeken waarom precies dit probleem optreedt. Er volgt notities om te laten zien wat de reden is dat het programma niet termineert.

Notities stap per stap voor een niet-terminered programma:

```
t(R) :- r(R), expr(R,[X,Y]), assert(my_t(R)).
t(H) :- r(R), expr(R,[X,Y]), my_t(T), expr(T,[Y,Z]), expr(H,[X,Z]),
            assert(my_t(H)). wrong (not recursive)

r([a,b]).
r([b,c]).

t([c,a])
first rule r([c,a]) --> false
second rule:
r(R), expr(R,[X,Y]), t(T), expr(T,[Y,Z]), expr([c,a],[X,Z]).
R=[a,b], X=a, Y=b
        first rule r(T), expr(T,[X1,Y1]), expr(T,[Y,Z]), expr([c,a],[X,Z]).
        T=[a,b], X1=a, Y1=b, Y=a, Z=b, expr([c,a],[a,b]) --> fail
        T=[b,c], X1=b, Y1=c, Y=b, Z=c, expr([c,a],[a,c]) --> fail
        second rule:
        r(R1), expr(R1,[X2,Y2]), t(T2), expr(T2,[Y2,Z2]),
                        expr(T,[X2,Z2]), expr(T,[Y,Z]), expr([c,a],[X,Z]).
        R1=[a,b]
        ...
```

De eerste regel faalt als er geen pad is waardoor er telkens een nieuwe T wordt geïntroduceerd in de tweede regel. In bovenstaande notities is te zien waarom het programma niet eindigt in dit voorbeeld. De vraag is ook waarom dit niet optreed in het normale Prolog-programma. De reden hiervoor is omdat de Z wordt verbonden met de huidige waarde met relatie t. In tegenstelling tot bovenstaande notities wordt dit gedaan met H.

Conclusie is dat het expr-predicaat recursief niet altijd werkt en we hier rekening mee moeten houden. Nu volgt een volledige en gedetailleerde uitwerking van de uitvoering:

Uitwerking zonder tail recursion, stap voor stap (zie onderstaande uitleg):

```
regel 1:
    r(R), expr(R, [X, Y])
R = [a,y]
fail

regel 2:
    r(R), expr(R, [X, Y]), t(T),
    expr(T, [Y, Z]), expr(H, [X, Z])
H = [a,y],
X=_G1,
Y=_G2,
Z=_G3,
T=_G4,
R = _G5

        r(_G5), expr(_G5, [_G1, _G2]), t(_G4),
```

```
          expr(_G4, [_G2, _G3]), expr(H, [_G2, _G3])

_G5 = [a,b]

          expr([a,y], [_G1, _G2]), t(_G4),
          expr(_G4, [_G2, _G3]), expr(H, [_G2, _G3])

_G1 = pathV([])
_G2 = pathV([a,y])

          t(_G4), expr(_G4, [_G2, _G3]),
          expr(H, [pathV(a,y), _G3])

_G4 = [a,b]
recursive call

          t(R), expr(R, [X, Y]), r(T),
          expr(T, [Y, Z]), expr(H, [X, Z])

H = _G4 = [a,b], X=_G6, Y=_G7,Z=_G8, T=_G9, R = _G10

          r(_G10), expr(_G10, [_G6, _G7]), t(_G9),
          expr(_G9, [_G7, _G8]), expr(H, [_G7, _G8])

_G10 = [a,b]

          expr([a,b], [_G6, _G7]), t(_G9),
          expr(_G9, [_G7, _G8]), expr(H, [_G7, _G8])

_G6 = pathV([]), _G7 = pathV([a,y])

          t(_G9), expr(_G9, [_G7, _G8]), expr(H, [_G7, _G8])

_G9 = [a,b]
recursive call

    infinite loop
```

Bovenstaande uitwerking laat zien hoe het programma stap voor stap elk volgende rule neemt en dan nieuwe variabelen introduceert als het nodig is. Regel 1 faalt meteen als we $[a, y]$ als input hebben omdat er niet rechtstreeks een pad voor bestaat, dus we gaan door naar regel 2. Als regel 2 begint dan hebben we een heel arsenaal van nieuwe variabelen omdat we een lange regel hebben van allemaal rules. Vervolgens vervangen we die variabelen met de nieuwe interne variabelen in de vorm van $\_G1$. Dan krijgt als eerste $\_G5$ een waarde $[a, b]$ en we gaan zo verder.

Belangrijk hier om te zien is op het einde dat voor t(T) we telkens nieuwe variabelen aanmaken. Kijk naar $\_G4$ en naar $\_G9$ die krijgen een nieuwe interne variabele maar die gaan telkens hetzelfde doen. Als gevolg gaat het programma nooit eindigen, die blijft namelijk nieuwe variabelen aanmaken. Omdat die nieuwe variabelen telkens dezelfde waarden toegewezen krijgen stopt het programma niet.

Zonder tail recursion, output recursie t(T):

```
?- t(T).
T = [a, b] ;
T = [a, b] ;
T = [a, b] ;
T = [b, c] ;
T = [b, c] ;
T = [b, c] ;
T = [c, d] ;
T = [c, d] ;
T = [c, d] ;
T = [d, e] ;
T = [d, e] ;
T = [d, e] ;
T = [c, x] ;
T = [c, x] ;
T = [c, x] ;
T = [x, y] ;
T = [x, y] ;
T = [x, y] ;
T = [] ;
T = [] ;
T = [] ;
out of local stack
```

Voor dit beter zichtbaar te maken voer ik t(T) uit en komen we uit op bovenstaande output. Nu volgt de uitwerking stap voor stap voor de versie met tail recursion. Merk op dat de recursieve calls allemaal eerst komen en we komen rechtstreeks in de tail van de recursie terecht. Dit was niet het geval in de versie zonder de tail recursion.

Uitwerking tail recursion, stap voor stap (zie onderstaande uitleg):

```
regel 1:
    r(R), expr(R, [X, Y])
R = [a,y]
fail

regel 2:
        t(R), expr(R, [X, Y]), r(T),
        expr(T, [Y, Z]), expr(H, [X, Z])
H = [a,y],
X=_G1,
Y=_G2,
Z=_G3,
T=_G4,
R = _G5

    t(_G5), expr(_G5, [_G1, _G2]), r(_G4),
    expr(_G4, [_G2, _G3]), expr(H, [_G2, _G3])

_G5 = [a,b]
recursive call

    t(_G10), expr(_G15, [_G11, _G12]),
```

```
    r(_G14), expr(_G14, [_G12, _G13]),
    expr(H, [_G12, _G13])

_G10 = [a,b]
recursive call

    t(_G15), expr(_G20, [_G16, _G17]),
    r(_G19), expr(_G19, [_G17, _G18]),
    expr(H, [_G17, _G18])

_G15 = [a,b]
recursive call

    t(_G20), expr(_G25, [_G21, _G22]),
    r(_G24), expr(_G24, [_G22, _G23]),
    expr(H, [_G22, _G23])

_G20 = [b,c]
recursive call

    expr(_G250, [_G210, _G220]), r(_G240),
    expr(_G240, [_G220, _G230]),
    expr(H, [_G220, _G230])

_G210 = pathV([])
_G220 = pathV([a,y])

    r(_G240), expr(_G240, [_G220, _G230]),
    expr(H, [_G220, _G230])

_G240 = [a,b]
recursive call

    expr(_G240, [_G220, _G230]),
    expr(H, [_G220, _G230])

...
```

Zonder tail recursion, output recursie t(T):

```
?- t(R).
R = [a, b] ;
R = [a, b] ;
R = [a, b] ;
R = [b, c] ;
R = [b, c] ;
R = [b, c] ;
R = [c, d] ;
R = [c, d] ;
R = [c, d] ;
R = [d, e] ;
R = [d, e] ;
R = [d, e] ;
```

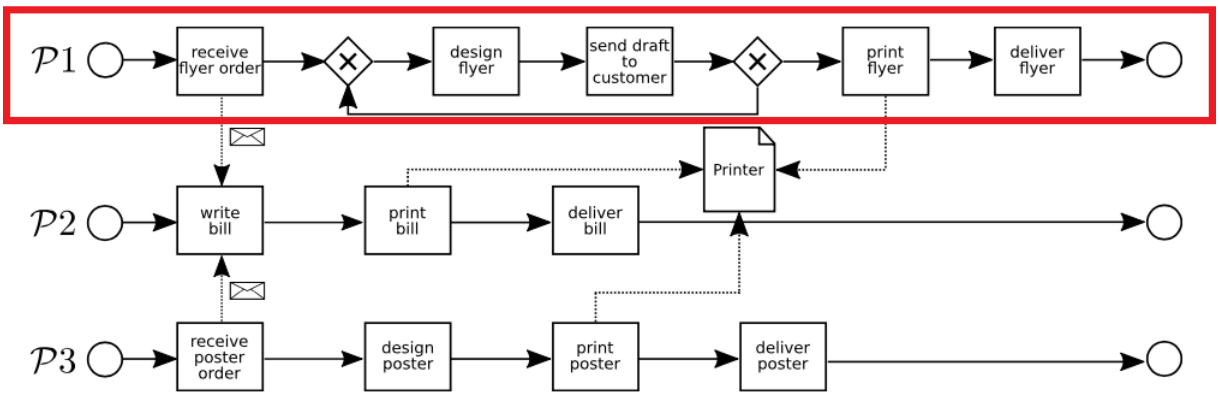
```
R = [c, x] ;
R = [c, x] ;
R = [c, x] ;
R = [x, y] ;
R = [x, y] ;
R = [x, y] ;
R = [] ;
R = [a, c] ;
R = [a, b, a, b] ;
R = [a, b, b, c] ;
R = [a, b, c, d] ;
R = [a, b, d, e] ;
R = [a, b, c, x] ;
R = [a, b, x, y] ;
R = [] ;
R = [a, c] ;
R = [a, b, a, b] ;
R = [a, b, b, c] ;
R = [a, b, c, d] ;
R = [a, b, d, e] ;
R = [a, b, c, x] ;
R = [a, b, x, y] ;
...
```

Bovenstaande uitvoer blijft nog heel lang doorgaan en geraakt niet “out of local stack”.

Hoofdstuk 5

Toepassing 1: Workflow Data

Voor deze toepassing refereer ik naar de workflow paper: [Workflow]. Een workflow beschrijft bepaalde processen van begin tot einde. Ook beschrijft het de volgorde tussen de onderlinge processen en eventuele relaties ervan. Workflows kan worden toegepast in ondernemingen en wordt meerdere keren of zelfs dagelijks uitgevoerd. De uitvoering van zo'n workflow worden *traces* genoemd. Elk van deze trace heeft typisch ook een timestamp. Hierdoor kunnen deze traces als logs worden opgeslagen op een sequentiële manier en is dit een goede toepassing van sequentiële data.



Figuur 5.1: Printing processen

Als voorbeeld gebruik ik de workflow uit paper [Workflow] die printing processen beschrijft. In figuur 5.1 kan je de workflow zien van de processen. Een volledig proces van begin tot einde noemt een trace. Elke trace heeft daarin events. De events (of activiteit) worden voorgesteld met de vierkanten die je kan zien in figuur 5.1. In mijn geval focus ik mij enkel op P1.

Een voorbeeld van een trace kan zijn: *receive flyer order -> design flyer -> send draft to customer -> design flyer -> send draft to customer -> print flyer -> deliver flyer.*

Merk op dat na het event *send draft to customer* er een beslissing moet worden gemaakt. Er kan hier ook een wachttijd in voorkomen, deze beslissing is natuurlijk niet instant want die komt gebruikelijk van de klant. De beslissing is ofwel een OK ofwel een NOK (not okay). Indien de beslissing NOK is gaan we verder naar het event *design flyer*, anders gaan we verder naar event *print flyer*.

In deze toepassing is het de bedoeling om in Prolog sequenties te maken van deze trace logs. Naast de traces met events ga ik daar ook extra data in mee opslaan. Sommige events hebben extra data, zoals de *send draft to customer* die een OK of NOK heeft, of *print flyer* waarbij we de naam van de printer weten. Deze informatie moet ik zien te bundelen en daar interessante queries op vinden met behulp van het expr-predicaat.

De bundeling van deze data heeft een aantal stappen nodig gehad. We kunnen ofwel het XES-bestand ofwel het CSV-bestand gebruiken. Het is mogelijk om het CSV-bestand te gebruiken maar die is niet gesorteerd op activiteiten. Een optie is om die eventueel te sorteren op de timestamps maar dat zou eventueel veel werk vergen. Ik heb gekozen voor met het XES-formaat te werken. Hiervoor gebruik ik de library Process Mining in Python [pm4py]. De traces kunnen hiermee één voor één ingelezen worden waardoor we echt trace per trace kunnen werken.

De werkwijze van deze toepassing ziet er dus als volgt uit: XES-formaat -> Python -> Prolog facts.

5.1 Van XES-bestand naar Prolog facts

Om het concept beter uit te leggen laat ik hier een volledige trace zien in XES-formaat. Onder deze trace leg ik het in meer detail uit.

```
<trace>
    <string key="concept:name" value="CustomerI4M3_flyer"/>
    <event>
      <string key="concept:instance" value="CustomerI4M3_flyer"/>
      <string key="concept:name" value="receive flyer order"/>
      <string key="lifecycle:transition" value="start"/>
      <string key="knr" value="CustomerI4M3"/>
      <date key="time:timestamp" value="2017-03-01T11:55:09.868+01:00"/>
    </event>
    <event>
      <string key="concept:instance" value="CustomerI4M3_flyer"/>
      <string key="concept:name" value="design flyer"/>
      <string key="lifecycle:transition" value="start"/>
      <string key="knr" value="CustomerI4M3"/>
      <date key="time:timestamp" value="2017-03-01T13:03:09.868+01:00"/>
    </event>
    <event>
      <string key="concept:instance" value="CustomerI4M3_flyer"/>
      <string key="concept:name" value="send draft to customer"/>
      <string key="lifecycle:transition" value="start"/>
      <string key="knr" value="CustomerI4M3"/>
      <string key="decision" value="nok"/>
      <date key="time:timestamp" value="2017-03-02T13:14:09.868+01:00"/>
    </event>
    <event>
      <string key="concept:instance" value="CustomerI4M3_flyer"/>
      <string key="concept:name" value="design flyer"/>
      <string key="lifecycle:transition" value="start"/>
      <string key="knr" value="CustomerI4M3"/>
      <date key="time:timestamp" value="2017-03-03T06:14:09.868+01:00"/>
```

```
    </event>
    <event>
      <string key="concept:instance" value="CustomerI4M3_flyer"/>
      <string key="concept:name" value="send draft to customer"/>
      <string key="lifecycle:transition" value="start"/>
      <string key="knr" value="CustomerI4M3"/>
      <string key="decision" value="ok"/>
      <date key="time:timestamp" value="2017-03-04T08:32:09.868+01:00"/>
    </event>
    <event>
      <string key="concept:instance" value="CustomerI4M3_flyer"/>
      <string key="concept:name" value="print flyer"/>
      <string key="lifecycle:transition" value="start"/>
      <string key="knr" value="CustomerI4M3"/>
      <string key="format" value="a4"/>
      <string key="org:resource" value="printer1"/>
      <int key="filling amount" value="18"/>
      <date key="time:timestamp" value="2017-03-06T08:21:12.000+01:00"/>
    </event>
    <event>
      <string key="concept:instance" value="CustomerI4M3_flyer"/>
      <string key="concept:name" value="deliver flyer"/>
      <string key="lifecycle:transition" value="start"/>
      <string key="knr" value="CustomerI4M3"/>
      <date key="time:timestamp" value="2017-03-06T16:00:00.000+01:00"/>
    </event>
</trace>
```

Bovenstaande trace neem ik als voorbeeld en heeft volgende workflow: *receive flyer order, design flyer, send draft to customer, design flyer, send draft to customer, print flyer, deliver flyer.* Er wordt dus één keer een NOK gesignaleerd waarbij we dus terugkeren naar het event *design flyer.*

concept:instance is de naam van flyer. *concept:name* is de naam van het event. *lifecycle:transition* is ofwel *start* ofwel *complete.* Na elke *start* wordt hetzelfde event herhaalt met een *complete*, maar die heb ik er uitgelaten. *knr* is klantennummer. *decision* komt voor in het *send draft to customer* event en is de eerder besproken OK of NOK en die ga ik ook gebruiken. *org:resource* komt voor in het *print flyer* event en gebruik ik ook. Om een beter beeld hiervan te geven volgt hier de bijbehorende trace die gegenereerd is door Python in een Prolog-bestand:

```
trace_path(2, [pack([a,"2017-03-01 12:03:09.868000"]),
               pack([b]),pack([c,nok]),
               pack([b]),pack([c,ok]),
               pack([d,printer1]),pack([e])]).
```

Merk op dat deze fact allemaal één regel is. Het eerste argument is de identifier. In dit geval heeft deze index twee en is dus de derde trace. Het tweede argument is de lijst van alle events in de trace. Ze zitten in packs omdat er bij sommige events nog extra data is aan toegevoegd en we moeten elke event apart beschouwen, ook al zien ze er so soms identiek uit. Elk event is gerepresenteerd met een letter, die ook overeenkomt met de volgorde van de workflow (zie figuur 5.1). Omdat deze Prolog-fact een lijst is kunnen we hier queries op uitvoeren met behulp van het expr-predicaat.

We kunnen hier interessante queries op uitvoeren. Er volgen twee secties met elk de query in Sequence Datalog en daarna vertaald in Prolog met behulp van het expr-predicaat.

5.2 Query 1

Query: is er een order waarbij de klant heeft toegestaan na exact drie versies?

Sequence Datalog:

```
s(@trace_id) :- $pattern=<B>.<C.nok>,
trace(@trace_id, $x.$pattern.$pattern.<B>.<C.ok>.$y).
```

Prolog met het expr-predicaat:

```
solution(Trace_Id) :-  trace_path(Trace_Id, Body),
expr(Body, [pathV(X), pack([b]), pack([c,nok]), pack([b]),
pack([c,nok]), pack([b]), pack([c,ok]), pathV(Y)]).
```

Voor deze query maak ik een pattern die moet matchen met de trace. De pattern die moet voorkomen is dus een NOK gevolgd door een NOK gevolgd door een OK. De body van de simulatie bevat dus eerst de trace path gevolgd door de expression waaraan de Body-variabele moet voldaan. Als resultaat krijgen we de identificatie van de trace(s) waaraan deze query voldoet.

5.3 Query 2

Query: is er een printer die is gebruikt in ten minste drie traces?

Eerst stel ik de Sequence Database versie op en daarna de vertaling.

Sequence Datalog:

```
printing_instances(@trace_id, @printer) :-
                trace(@trace_id, $x.<D.@printer>.$y).
s(@printer) :-  printing_instances(@id1, @printer),
                printing_instances(@id2, @printer),
                printing_instances(@id3, @printer),
                @id1 \= @id2, @id2 \= @id3, @id1 \= @id3.
```

Prolog met het expr-predicaat:

```
printing_instances(Trace_Id, Printer) :-
                trace_path(Trace_Id, Body),
                expr(Body,[pathV(X),
                    pack([d, atomicV(Printer)]),pathV(Y)]).
solution2(Printer) :-
                printing_instances(Id1, Printer),
                printing_instances(Id2, Printer),
                printing_instances(Id3, Printer),
                Id1 \= Id2, Id2 \= Id3, Id1 \= Id3.
```

Je kan hier al opmerken dat de query bijna hetzelfde eruitziet. Bij de simulatie gebruik ik het expr-predicaat om de path-expression van de sequentie te definiëren. In dit geval

zoeken we dus een pack met event d waarbij we ook informatie over de printer in een variabele *Printer* zetten. Zowel de identificatie van de trace en de printer wordt meegegeven in de head van de query. In solution gaan we die dus drie keer herhalen, waarbij we drie verschillende identificaties hebben die niet mogen overeenkomen. Merk op dat de printer-variabele telkens hergebruikt wordt, wat wil zeggen dat die niet verandert en dus dezelfde printer is. Deze query werkt met de trace identificaties en we vergelijken deze ook. Er bestaat een betere oplossing door niet met deze trace identificaties te werken maar wel meer gebruik te maken van de features van packing. Packing zorgt ervoor dat we een uniek resultaat krijgen en als we daar gebruik van maken dan hoeven we de trace identificaties helemaal niet te gebruiken. Dit doe ik in de volgende query in hoofdstuk 5.4.

Als resultaat krijgen we de naam van de printer(s) waaraan deze query voldoet.

5.4 Query 3

Query: veralgemening van query 2. Is er een subtrace van 5 events die minstens 3 keer voorkomt?

Prolog met het expr-predicaat:

```
printing_instance(H, Printer) :-
    trace_path(Trace_Id, Body),
    expr(Body, [pathV(U),
        pack([d, atomicV(Printer)]), pathV(V)]),
    % H is highlight
    expr(H, [pathV(U),
        pack([pack([d, atomicV(Printer)])]), pathV(V)]).
solution3(Printer) :-
    printing_instance(H1, Printer),
    printing_instance(H2, Printer),
    printing_instance(H3, Printer),
    H1 \= H2, H2 \= H3, H1 \= H3.
```

Deze query is een stap naar de uitbreiding van de query uit hoofdstuk 5.3. In de plaats van 1 printer die 3 keer voorkomt gaan we kijken naar een subtrace van 5 events die minstens 3 keer voorkomt. Vooraleer we dit bekomen generaliseer ik het probleem naar een subtrace die 3 of meerdere keren voorkomt. Hier verandert nog niet veel tegenover de vorige query maar het is wel een belangrijke stap om de verandering te begrijpen. In bovenstaande code, net onder de commentaar dat H de highlight is, is er een nieuwe regel geïntroduceerd. De bedoeling is om de trace die we willen vinden te highlighten door de packing feature te gebruiken. Dit doen we zodat we later de trace identificaties niet meer nodig gaan hebben. Een voorbereiding hiervoor is deze nieuwe regel waarbij we de subtrace gaan highlighten of opnieuw gaan packen om een uniek resultaat te bekomen. Door dit uniek resultaat hoeven we de trace identificatie niet meer mee te nemen en veralgemenen we de query naar 3 of meer subtraces, in de plaats van 3 of meer traces waar er dezelfde printer wordt gebruikt zoals in hoofdstuk 5.3.

Query: veralgemening van query 2. Is er een subtrace van 5 events die minstens 3 keer voorkomt? Prolog met het expr-predicaat:

```
printing_instance(H, X1, X2, X3, X4, X5) :-
    trace_path(Trace_Id, Body),
    expr(Body, [pathV(U), pack([X1]), pack([X2]),
```

```
        pack([X3]), pack([X4]), pack([X5]), pathV(V)]),
    expr(H, [pathV(U), pack([pack([X1]), pack([X2]),
        pack([X3]), pack([X4]), pack([X5])]), pathV(V)]).
solution4(H1, H2, H3) :-
    printing_instance(H1, X1, X2, X3, X4, X5),
    printing_instance(H2, X1, X2, X3, X4, X5),
    printing_instance(H3, X1, X2, X3, X4, X5),
    H1 \= H2, H2 \= H3, H1 \= H3.
```

Bovenstaande code is de uiteindelijke versie van de veralgemening van query 2. Merk op dat we met de $Trace_Id$ verder niets mee doen. Dit was niet het geval in query 2 (hoofdstuk 5.3) waar we die nog gaan vergelijken. Nu hebben we puur unieke subtraces door de dubbele packing of highlighting die ik eerder heb besproken. Als we bovenstaande *solution*4 gaan uitvoeren krijgen we de desbetreffende highlights als resultaat:

```
?- solution4(H1, H2, H3).
H1 = [pack([a, "2017-03-01 09:30:19.523000"]), pack([pack([b]),
      pack([c, nok]), pack([b]), pack([c|...]), pack([...])]),
      pack([c, ok]), pack([d, printer1]), pack([e])],
H2 = [pack([a, "2017-03-01 11:19:06.466000"]), pack([pack([b]),
      pack([c, nok]), pack([b]), pack([c|...]), pack([...])]),
      pack([c, ok]), pack([d, printer1]), pack([e])],
H3 = [pack([a, "2017-03-01 10:51:19.566000"]), pack([pack([b]),
      pack([c, nok]), pack([b]), pack([c|...]), pack([...])]),
      pack([c, ok]), pack([d, printer1]), pack([e])] .
```

We kunnen de uitvoer ook veranderen door de onderdelen van de highlight op te vragen op volgende manier:

```
?- solution4(X1, X2, X3, X4, X5).
X1 = X3, X3 = X5, X5 = pathV([b]),
X2 = X4, X4 = pathV([c, nok]) ;
X1 = X3, X3 = X5, X5 = pathV([b]),
X2 = X4, X4 = pathV([c, nok]) ;
X1 = X3, X3 = X5, X5 = pathV([b]),
X2 = X4, X4 = pathV([c, nok]) ;
X1 = X3, X3 = X5, X5 = pathV([b]),
X2 = X4, X4 = pathV([c, nok]) ;
X1 = X3, X3 = X5, X5 = pathV([b]),
X2 = X4, X4 = pathV([c, nok]) ;
```

Hoofdstuk 6

Toepassing 2: Genome Data

Het belang van sequentiële data in bio-informatica gaat duidelijk worden door onder andere het voorbeeld van genoom-data. In de wereld van genetische data wordt er gebruikgemaakt van strings van data. Deze strings van genetische informatie kan je sequenties noemen. Die worden een sequentiële manier opgeslagen worden en is dus sequentiële data. Hierdoor is Sequence Datalog geschikt voor toepassingen hierover. In de paper [Sequences Datalog en Transducers] bespreekt men dat je met dit soort data eventueel aan pattern extraction kan doen. Een nadeel is dat het een belangrijke feature zoals de reconstructie van sequenties mist. Daarom kan Sequence Datalog een geschikte kandidaat zijn voor het werken met genoom-data.

Prolog - RNA sequenties:

```
rna_sequences([a,g,g,a,c,c]).
rna_sequences([a,c,a,g,a,c]).
rna_sequences([g,c,u,g,u,g]).
```

Voor de toepassing van genoom-data gebruik ik de vertaling van een RNA-string naar een proteïne-string. RNA ziet eruit als een string van 3 waarden die uit volgende letters bestaan: A, C, G, U. Deze letters staan voor adenine, cytosine, guanine en uracil. Bovenstaande code laat een aantal sequenties zien van RNA-data. Merk op dat we deze data niet specifiek opdelen in stukken van 3. Deze opdeling wordt gedaan tijdens de vertaling naar proteïnen. Proteïnen ziet eruit als simpelweg 1 letter. Je kan dus een tabel opstellen om de vertaling van RNA naar proteïne te doen. De code volgt:

Prolog - vertaling tabel:

```
from_RNA_to_protein_table([g,c,a], a).
from_RNA_to_protein_table([g,c,c], a).
from_RNA_to_protein_table([g,c,g], a).
from_RNA_to_protein_table([g,c,u], a).
from_RNA_to_protein_table([a,g,a], r).
from_RNA_to_protein_table([a,g,g], r).
from_RNA_to_protein_table([c,g,a], r).
from_RNA_to_protein_table([c,g,c], r).
from_RNA_to_protein_table([c,g,u], r).
from_RNA_to_protein_table([g,a,c], d).
from_RNA_to_protein_table([g,a,u], d).
from_RNA_to_protein_table([a,a,c], n).
```

```
from_RNA_to_protein_table([a,a,u], n).
from_RNA_to_protein_table([u,g,c], c).
from_RNA_to_protein_table([u,g,u], c).
from_RNA_to_protein_table([g,a,a], e).
from_RNA_to_protein_table([g,a,g], e).
from_RNA_to_protein_table([c,a,a], q).
from_RNA_to_protein_table([c,a,g], q).
from_RNA_to_protein_table([g,g,a], g).
from_RNA_to_protein_table([g,g,c], g).
from_RNA_to_protein_table([g,g,g], g).
from_RNA_to_protein_table([g,g,u], g).
from_RNA_to_protein_table([c,a,c], h).
from_RNA_to_protein_table([c,a,u], h).
from_RNA_to_protein_table([a,u,a], i).
from_RNA_to_protein_table([a,u,c], i).
from_RNA_to_protein_table([a,u,u], i).
from_RNA_to_protein_table([u,u,a], l).
from_RNA_to_protein_table([u,u,g], l).
from_RNA_to_protein_table([c,u,a], l).
from_RNA_to_protein_table([c,u,c], l).
from_RNA_to_protein_table([c,u,g], l).
from_RNA_to_protein_table([c,u,u], l).
from_RNA_to_protein_table([a,a,a], k).
from_RNA_to_protein_table([a,a,g], k).
from_RNA_to_protein_table([a,u,g], m).
from_RNA_to_protein_table([u,u,c], f).
from_RNA_to_protein_table([u,u,u], f).
from_RNA_to_protein_table([c,c,a], p).
from_RNA_to_protein_table([c,c,c], p).
from_RNA_to_protein_table([c,c,g], p).
from_RNA_to_protein_table([c,c,u], p).
from_RNA_to_protein_table([a,g,c], s).
from_RNA_to_protein_table([a,g,u], s).
from_RNA_to_protein_table([u,c,a], s).
from_RNA_to_protein_table([u,c,c], s).
from_RNA_to_protein_table([u,c,g], s).
from_RNA_to_protein_table([u,c,u], s).
from_RNA_to_protein_table([a,c,a], t).
from_RNA_to_protein_table([a,c,c], t).
from_RNA_to_protein_table([a,c,g], t).
from_RNA_to_protein_table([a,c,u], t).
from_RNA_to_protein_table([u,g,g], w).
from_RNA_to_protein_table([u,a,c], y).
from_RNA_to_protein_table([u,a,u], y).
from_RNA_to_protein_table([g,u,a], v).
from_RNA_to_protein_table([g,u,c], v).
from_RNA_to_protein_table([g,u,g], v).
from_RNA_to_protein_table([g,u,u], v).
from_RNA_to_protein_table([u,a,a], stop).
from_RNA_to_protein_table([u,a,g], stop).
from_RNA_to_protein_table([u,g,a], stop).
```

Bovenstaande code laat alle facts zien die de tabel vormen voor de vertaling van een RNA-string naar de bijbehorende proteïne-string. De volgorde is op die manier zodat de proteïnes gegroepeerd zijn. Merk op dat verschillende RNA-strings kunnen vertalen naar dezelfde proteïne-string. De bedoeling van deze toepassing is om deze sequenties te genereren in de groottes of lengtes die we willen. Vervolgens gaan we een manier vinden om deze sequenties te vertalen naar sequenties van proteïnen in Sequence Datalog. Ten slotte gaan we die omzetten naar een programma in Prolog door het expr-predicaat toe te passen zodat we path expressions rechtstreeks kunnen gebruiken. Ter vergelijking heb ik het programma ook in standaard Prolog gemaakt zonder het expr-predicaat. Ik vergelijk ik de normale Prolog implementatie met de simulatie van de Sequence Datalog implementatie die beide in Prolog zijn gemaakt. Als extra, om nog met deze data te werken met behulp van het expr-predicaat, is er nog een query opgesteld waarbij alle maximale substrings wordt opgevraagd van minimum lengte 3 die uitsluitend uit A, C, G of T bestaan. Hier ga ik dieper op in in sectie 6.5.

6.1 Genereren van data

Python - genereren van RNA:

```
import random

_amount_of_triples = 2
_amount_of_lines = 3

# 60 possibilities
RNA_List = ["gca","gcc","gcg","gcu","aga","agg","cga", ...]


f = open("rna.txt", "w")
for y in range(_amount_of_lines):
    for x in range(_amount_of_triples):
        _triple = RNA_List[random.randint(0, 59)]
        _triple_with_commas = "" + _triple[0:1] + ','
                              + _triple[1:2] + ',' + _triple[2:3]
        if x is not _amount_of_triples-1:
            _triple_with_commas += ','
        f.write(_triple_with_commas)
        # f.write(_triple) # old way without commas
    f.write('\n')
f.close()
```

Voor het genereren van de RNA-strings heb ik een tweedelige Python-programma gemaakt. Alle mogelijkheden van RNA-strings zijn hierin gedefinieerd en worden willekeurig in een tekstbestand weggeschreven. Het is mogelijk om zelf te bepalen hoeveel triples en facts het programma moet genereren. Bovenstaand is de code ervan. Als tweede deel wordt de uitvoer hiervan (strings van RNA-data) ingelezen en omgevormd naar een Prolog-bestand. De RNA-strings worden in deze stap omgezet naar facts in Prolog.

Python - inladen van RNA naar Prolog:

```
f = open("rna.txt", "r")
_string_to_write = ""
```

```
_line = f.readline()
while _line:
    if _line is not "":
        _len = len(_line)
        _string = _line[0:_len-1] # remove the last \n
        _string_to_write += "rna_sequences([" + _string + "]).\n"
    _line = f.readline()
f.close()

f = open("rna.pl", "w")
f.write(_string_to_write)
f.close()
```

De uitvoer van bovenstaande code ziet eruit als volgt:

Prolog - RNA sequenties:

```
rna_sequences([a,g,g,a,c,c]).
rna_sequences([a,c,a,g,a,c]).
rna_sequences([g,c,u,g,u,g]).
```

Merk op dat dit opgeslagen wordt in Prolog-formaat zodat we deze sequenties rechtstreeks kunnen inladen. De reden voor een separatie voor het genereren en omvormen van data is om eventueel externe databases met strings van RNA-sequenties te kunnen inlezen.

6.2 Van RNA naar proteïne - Sequence Datalog

Sequence Datalog:

```
1. T(b.$seq, e) <- RNA($seq)
2. T($u.@x1.@x2.@x3.b.$r, $v.@p) <- T($u.b.@x1.@x2.@x3.$r, $v),
                                    translate(@x1, @x2, @x3, @p).
3. R($v) <- T($seq.b, $v).
```

Dit is de code in Sequence Datalog om van een RNA-string naar een proteïne-string te gaan. De input van het programma is een sequentie van een RNA-string van onbepaalde lengte. Ik introduceer een pointer genaamd b en dit is een letter die niet voorkomt in eender welke RNA-string. Deze pointer zorgt ervoor dat het programma weet op welke plaats in de RNA-string het zich bevindt. Dit wordt duidelijker in het voorbeeld dat ik geef vanaf de volgende paragraaf. Vervolgens gaat het programma per triple de hele sequentie afgaan, die telkens vertalen en tussentijds opslaan m.b.v. concatenatie op een initieel lege path-variable $\$v$. Als uiteindelijk de pointer zich op het einde bevindt dan stop het programma en geeft het de proteïne-string terug. Als volgende bespreek ik een voorbeeld die ik stap per stap doorloop om zo de werking van het programma in detail te bespreken.

Voorbeeld: stel dat we een RNA-sequentie hebben van twee triples zoals:

$$RNA([a, g, g, a, c, c]).$$

Regel 1 wordt uitgevoerd. De body doet niets speciaal. De head concateneert een b met de sequentie. De b komt voor in geen enkele RNA-string en gebruiken we als een pointer. Het resultaat van regel 1 is $T([b, a, g, g, a, c, c], [])$. Ter verduidelijking, het tweede argument e in het programma staat voor een lege lijst.

Als volgende voeren we regel 2 uit. Dit blijven we doen tot pointer b zich aan het einde van de sequentie bevindt. De reden hiervoor is omdat de eerste regel niet voldoet (we werken nu met $T()$ en niet met $RNA()$) en regel 4 voldoet niet omdat het eerste argument van de body verwacht dat pointer b aan het einde van de sequentie is.

De eerste iteratie van regel 2 ziet er als volgt uit:

```
body:
$u = []
x1 = a
x2 = g
x3 = g
$r = [a,c,c]
$v = []
T([b,a,g,g,a,c,c], [])
head:
$v = [r]
T([a,g,g,b,a,c,c], [], r) # translate([x1,x2,x3], @p)
```

$[a, g, g]$ wordt vertaald naar r.

De tweede iteratie van regel 2 ziet er als volgt uit:

```
body:
$u = [a,g,g]
x1 = a
x2 = c
x3 = c
$r = []
$v = [r]
T([a,g,g,b,a,c,c], [])
head:
$v = [r,t]
T([a,g,g,a,c,c,b], [r, t], t) # translate([x1,x2,x3], @p)
```

Merk op dat we nu eindigen met een sequentie die eindigt met pointer b. Dit is wat regel 3 van het programma verwacht. Dit ziet er als volgt uit:

```
$seq = [a,g,g,a,c,c]
$v = [r, t]
R([r, t])
```

We bekomen $[r, t]$ als resultaat.

6.3 Van RNA naar proteïne - Prolog

De standaard Prolog-versie voor de vertaling van RNA naar proteïne, in vergelijking met de Sequence Datalog versie in hoofdstuk 6.2, heeft veel stappen nodig gehad. Ik ben begonnen met het interpreteren van deze RNA-sequenties door ze voor te stellen als strings. Vervolgens ging ik deze sequenties, in de vorm van strings, manipuleren door features van manipulatie van strings in Prolog te gebruiken. Dit heeft veel werk gevergd en is niet de uiteindelijke oplossing omdat we uiteindelijk van strings zijn overgegaan naar werken met lijsten. Om het verschil te laten zien in complexiteit tegenover de Sequence Datalog versie (hoofdstuk 6.2) voeg ik deze code toch toe.

Prolog - vertaling van RNA naar proteïne met strings:

```
% STANDARD WAY IN PROLOG
% +RNA_String is a string of unknown length (assuming it's %3).
% This picks the first three, asserts the RNA, translates to protein,
% asserts this too.
save_RNA_string("").
save_RNA_string(RNA_String) :-
    substring(RNA_String, 1, 3, RNA_String_Single),
    rna_String_To_List(RNA_String_Single, RNA_List),
    !, % Commit to prevent backtrack, otherwise it saves two times
    assert(rna(RNA_List)),
    string_length(RNA_String,StrLen),
    StrLenMin3 is StrLen - 3,
    substring(RNA_String, 4, StrLenMin3, Rest),
     % translate RNA to Protein
    from_RNA_to_protein_table(RNA_List, Protein),
    assert(protein(Protein)),
    save_RNA_string(Rest).

% +RNA_String_Single is a string of length 3, -RNA_List is a list
% of length 3 with single atoms
% "abc" -> [a,b,c]
rna_String_To_List(RNA_String_Single, RNA_List) :-
    substring(RNA_String_Single, 1, 1, First),
    substring(RNA_String_Single, 2, 1, Second),
    substring(RNA_String_Single, 3, 1, Third),
    atom_string(FirstAtom, First),
    atom_string(SecondAtom, Second),
    atom_string(ThirdAtom, Third),
    RNA_List = [FirstAtom, SecondAtom, ThirdAtom].
```

Bovenstaande code leg ik uit in woorden en ga ik niet heel in detail op in omdat dit een oudere versie is, maar wel belangrijk is geweest in het proces. Als input hebben we dus een string die de RNA-sequentie representeert. We nemen een substring: in dit geval de eerste 3 letters. In regel genaamd $rna_String_To_List$ wordt die omgezet in een lijst. We bekomen een lijst met 3 letters. Vervolgens berekent het de lengte van de hele string exclusief de eerste 3 tekens. Dit doen we om de substring te nemen exclusief de eerste 3 voor de volgende stappen. De lijst met 3 letters wordt dan vertaald naar de bijbehorende proteïne en opgeslagen per proteïne. Merk op dat dit een hele stuk code is voor zo iets simpels. In Sequence Datalog in hoofdstuk 6.2 ziet het er veel eleganter uit!

Prolog - vertaling van RNA naar proteïne in standaard Prolog met lijsten:

```
translate_in_prolog() :-
    rna_sequences(RNA),
    start_translate(RNA, []).
start_translate([], P) :-
    format("~w\n", [P]).
start_translate(L, P) :-
    [First, Second, Third | R] = L,
    from_RNA_to_protein_table([First, Second, Third], Protein),
```

```
    append(P, [Protein], NewProtSeq),
    start_translate(R, NewProtSeq).
```

Als volgende laat ik de verbetering zien die met lijsten werkt in de plaats van strings. Bovenstaande code is de implementatie ervan. Deze kan je naast de Sequence Datalog versie kan plaatsen (hoofdstuk 6.2) en vergelijken. Dit programma begint met het uitvoeren van $translate_in_prolog()$. Het neemt een volledige lijst in relatie $rna_sequences$ die een RNA-lijst bevat. In de tweede $start_translate$, de eerste regel, kan je zien dat we geen pointer gebruiken maar gewoon de features van Prolog om met lijsten te werken. Hier maken we een separatie van de eerste 3 letters, die we apart nemen, en de rest van de sequentie. Vervolgens gaan we de enkele RNA vertalen m.b.v. de vertalingstabel die ik heb besproken in de inleiding. Dan gaan we een initieel lege P concateneren met de bijbehorende proteïne zodat we uiteindelijk in de recursie hier de hele sequentie van proteïnen bekomen. De recursie-stap volgt met als invoer de rest van de sequentie en de nieuwe proteïne-sequentie. Een voorbeeld van de uitvoer van dit programma volgt.

Prolog - uitvoer van vertaling in standaard Prolog:

```
rna_sequences([a,g,g,a,c,c]).
rna_sequences([a,c,a,g,a,c]).
rna_sequences([g,c,u,g,u,g]).

?- translate_in_prolog().
[r,t]
true ;
[t,d]
true ;
[a,v]
true ;
false.
```

Merk op dat bovenstaande sequenties zijn gemaakt m.b.v. een generator die ik in Python heb geschreven zoals besproken in hoofdstuk 6.1. Die worden in een Prolog-bestand opgeslagen waardoor we die rechtstreeks kunnen inladen in de Prolog-omgeving. Om consistentie te bewaren laat ik ook nog een aantal andere inputs zien die ik in hoofdstuk 6.4 hergebruik bij de versie van het expr-predicaat met path expressions.

Prolog - extra voorbeelden uitvoer van vertaling in standaard Prolog:

```
rna_sequences([a,g,g,a,c,c]).
rna_sequences([u,c,g,a,a,a,a,c,u,u,u,g,u,c,g]).
rna_sequences([a,u,u,a,a,c,c,a,u,a,g,u,c,c,g,g,
               u,g,a,g,u,c,c,c,c,u,u,u,c,a]).

?- translate_in_prolog().
[r,t]
true ;
[s,k,t,l,s]
true ;
[i,n,h,s,p,v,s,p,l,s]
true
```

6.4 Van RNA naar proteïne - Expr-Predicaat

Prolog - vertaling van RNA naar proteïne in Prolog met het expr-predicaat en path expressions:

```
t(T1, []) :- rna_sequences(Seq),
             expr(T1, [b, pathV(Seq)]).
t(T2, V2) :- t(T1, V),
             expr(T1, [pathV(U), b, atomicV(X1), atomicV(X2),
                 atomicV(X3), pathV(R)]),
             expr(T2, [pathV(U), atomicV(X1), atomicV(X2),
                 atomicV(X3), b, pathV(R)]),
             from_RNA_to_protein_table([X1,X2,X3],Protein),
             expr(V2, [pathV(V), atomicV(Protein)]).
result(V) :- t(T3, V),
             expr(T3, [pathV(Seq), b]).
```

Bovenstaande code is de implementatie van de Sequence Datalog versie in Prolog met path expressions. Hier heb ik het expr-predicaat voor gebruikt. Voor te vergelijken voeg ik nog eens de Sequence Datalog versie toe:

Sequence Datalog:

```
1. T(b.$seq, e) <- RNA($seq)
2. T($u.@x1.@x2.@x3.b.$r, $v.@p) <- T($u.b.@x1.@x2.@x3.$r, $v),
                                    translate(@x1, @x2, @x3, @p).
3. R($v) <- T($seq.b, $v).
```

De path expressions zijn duidelijk nog zichtbaar in de implementatie in Prolog. Als je deze versies nu vergelijkt met die van de standaard Prolog implementatie m.b.v. lijsten dan is het verschil groot. In het laatste zijn de path expressions niet zichtbaar!

De uitvoer van dit programma werkt helaas wel niet naar behoren bij grotere inputs. We wisten al dat het expr-predicaat niet goed werkt met recursie. Dit heb ik al besproken in hoofdstuk 4.3.4. De uitvoering van dit programma werkt wel met recursie maar het duurt heel lang bij een grote hoeveelheid data. Dit bespreek ik verder in hoofdstuk 6.6. Hier volgt een aantal uitvoeringen met verschillende inputs:

Prolog - voorbeelden uitvoer vertaling van RNA naar proteine met het expr-predicaat:

```
rna_sequences([a,g,g,a,c,c]).
?- result(X).
X = [r, t] .

rna_sequences([u,c,g,a,a,a,a,c,u,u,u,g,u,c,g]).
?- result(X).
X = [s, k, t, l, s] .

rna_sequences([a,u,u,a,a,c,c,a,u,a,g,u,c,c,g,g,u,g,a,g,u,c,c,c,
    c,u,u,u,c,a]).
?- result(X).
X = [i, n, h, s, p, v, s, p, l, s] .
```

Vervolgens laat ik een soort debug-uitvoer laat zien om te tonen in detail hoe het programma werkt en waarom het ook stopt. Dit doe ik door elke regel stap per stap uit te

voeren en de tussenresultaten te tonen met als invoer $[a, g, g, a, c, c]$ die als uitvoer $[r, t]$ zou moeten geven:

Prolog - stap per stap vertaling van RNA naar proteïne met het expr-predicaat:

```
?- expr(T1, [b, pathV([a,g,g,a,c,c])]).
T1 = [b, a, g, g, a, c, c] ;
false.

?- expr([b,a,g,g,a,c,c], [pathV(U), b, atomicV(X1), atomicV(X2),
    atomicV(X3), pathV(R)]).
U = [],
X1 = a,
X2 = X3, X3 = g,
R = [a, c, c] ;
false.

?- expr(T2, [pathV([]), atomicV(a), atomicV(g), atomicV(g), b,
    pathV([a,c,c])]).
T2 = [a, g, g, b, a, c, c] ;
false.

?- from_RNA_to_protein_table([a,g,g],Protein).
Protein = r ;
false.

?- expr(V, [pathV([]), atomicV(r)]).
V = [r] ;
false.

?- expr([a,g,g,b,a,c,c], [pathV(U), b, atomicV(X1), atomicV(X2),
    atomicV(X3), pathV(R)]).
U = [a, g, g],
X1 = a,
X2 = X3, X3 = c,
R = [] ;
false.

?- expr(T2, [pathV([a,g,g]), atomicV(a), atomicV(c), atomicV(c),
    b, pathV([])]).
T2 = [a, g, g, a, c, c, b] ;
false.

?- from_RNA_to_protein_table([a,c,c],Protein).
Protein = t ;
false.

?- expr(V, [pathV([r]), atomicV(t)]).
V = [r, t] ;
false.
```

Ik laat hierboven telkens het tussenresultaat zien en ook eventueel andere resultaten met

de puntkomma, maar er is in dit geval telkens maar 1 resultaat. Het programma stopt omdat in het tweede predicaat in de Prolog-versie de path expression van $T1$ verwacht dat de pointer b niet op het einde is. Anders kan hier geen oplossing voor gevonden worden en voert het predicaat 2 niet uit maar gaat het verder naar het derde predicaat. Hier een voorbeeld waarom predicaat 2 niet wordt uitgevoerd na het verwerken van de hele invoer:

```
?- expr([a, g, g, a, c, c, b], [pathV(U), b, atomicV(X1),
    atomicV(X2), atomicV(X3), pathV(R)]).
false.
```

Hier wordt geen enkele goal voldaan dus het programma voert predicaat 3 uit. Deze geeft dus het resultaat van de waarde V terug die de lijst van de vertaalde proteïnen bevat.

6.5 Extra voorbeeld - data queryen

```
same_subsequences(X) :- % pathV(X) is a substring in the middle
    rna_sequences(RNA),
    expr(RNA, [pathV(L), atomicV(B), pathV(X), atomicV(C), pathV(R)]),
    % at least three where the first three are the same @z letters
    expr(X, [atomicV(Z), atomicV(Z), atomicV(Z), pathV(Y)]),
    Z \= B, Z \= C, % @a \= @b, @a \= @c
    % next two lines, check if they only consists of a's
    expr(K, [atomicV(Z), pathV(X)]),
    expr(K, [pathV(X), atomicV(Z)]).

same_subsequences(X) :- % pathV(X) is a prefix
    rna_sequences(RNA),
    expr(RNA, [pathV(X), atomicV(C), pathV(R)]),
    % at least three where the first three are the same @z letters
    expr(X, [atomicV(Z), atomicV(Z), atomicV(Z), pathV(Y)]),
    Z \= C, % @a \= @c
    % next two lines, check if they only consists of a's
    expr(K, [atomicV(Z), pathV(X)]),
    expr(K, [pathV(X), atomicV(Z)]).

same_subsequences(X) :- % pathV(X) is a suffix
    rna_sequences(RNA),
    expr(RNA, [pathV(L), atomicV(B), pathV(X)]),
    % at least three where the first three are the same @z letters
    expr(X, [atomicV(Z), atomicV(Z), atomicV(Z), pathV(Y)]),
    Z \= B, % @a \= @b
    % next two lines, check if they only consists of a's
    expr(K, [atomicV(Z), pathV(X)]),
    expr(K, [pathV(X), atomicV(Z)]).

same_subsequences(X) :- % pathV(X) is a prexix and a suffix
    rna_sequences(RNA),
    expr(RNA, [pathV(X)]),
    % at least three where the first three are the same @z letters
    expr(X, [atomicV(Z), atomicV(Z), atomicV(Z), pathV(Y)]),
    % next two lines, check if they only consists of a's
    expr(K, [atomicV(Z), pathV(X)]),
    expr(K, [pathV(X), atomicV(Z)]).
```

Zoals eerdere aan gegeven in de inleiding van dit hoofdstuk gaan we de RNA-sequenties nog queryen met behulp van het expr-predicaat. De query zoekt alle maximale substrings van minimum lengte 3 die uitsluitend uit A, C, G of T bestaan. Er zijn verschillende regels voor gemaakt. Het gezochte sub-string kan zich op meerdere plaatsen bevinden. Het kan voorkomen als prefix, als suffix, als beiden of in the midden. Voor al deze scenario's zijn dus regels voor geschreven.

Een beschrijving van het programma volgt. We gaan eerst de RNA-sequentie opvragen. Vervolgens volgen de expressies om deze te matchen aan een bepaalde pattern (suffix/prefix/beide/midden). Dan maken we een aantal ongelijkheden om ervoor te zorgen dat het pad dat de omliggende atoms van het pad dat we zoeken niet gelijk zijn aan elkaar. Bijvoorbeeld in het geval van a, c, c, c, b is dit het geval. Deze ongelijkheid is nodig omdat we het minimum substring opvragen en er kunnen er meer dan 3 zijn, dus de omliggende letters mogen niet overeenkomen met de letters van de substring.

Vervolgens laat ik een aantal uitvoeringen zien met voorbeelden.

Prolog - voorbeeld uitvoeringen:

```
rna_sequences([a,g,g,a,c,c]).
?- same_subsequences(X).
false.

rna_sequences([u,c,g,a,a,a,a,c,u,u,u,g,u,c,g]).
?- same_subsequences(X).
X = [a, a, a, a] ;
X = [u, u, u] ;
false.

rna_sequences([a,u,u,a,a,c,c,a,u,a,g,u,c,c,g,g,u,g,a,g,u,c,c,c,
    c,u,u,u,c,a]).
?- same_subsequences(X).
X = [c, c, c, c] ;
X = [u, u, u] ;
false.
```

6.6 Grootte van data

Voor deze toepassing heb ik gekeken hoe ver we kunnen gaan met de grootte van data die we als invoer geven. Hiervoor heb ik de generator gebruikt die ik heb geschreven in Python (hoofdstuk 6.1). Hiervoor kan ik specifiek meegeven hoeveel RNA-triples ik wil genereren en hoeveel facts. De query die ik gebruik voor deze tests is de vertaling van RNA naar proteïne. De reden hiervoor is omdat ik dan de implementatie met het expr-predicaat kan vergelijken met de implementatie in standaard Prolog. De code en uitleg van deze twee implementaties is terug te vinden in hoofdstuk 6.3 en 6.4.

Ik ben begonnen met **1000 triples** te genereren van zo'n RNA-sequentie. Dit zijn 3000 variabelen. Vervolgens heb ik de vertaling uitgevoerd met de versie met het expr-predicaat en path expressions. Op mijn computer heeft het maar liefst 9 minuten geduurd aan berekeningen! Om te verifiëren heb ik dit een paar keren uitgevoerd. Hierbij kreeg ik telkens een consistente meting.

De hele uitvoer is als volgt:

Prolog - uitvoer van de vertaling:

```
?- result(X).
X = [t, s, t, k, v, p, s, i, k|...] [write]
X = [t, s, t, k, v, p, s, i, k, g, r, v, s, v, v, m, a, s, r, a, a, p,
l, v, c, h, s, i, l, i, s, i, g, s, c, a, y, l, f, f, c, q, t, m, t,
q, s, y, p, g, y, v, n, e, e, r, l, s, l, v, c, e, v, y, e, q, i, t,
i, y, q, a, a, f, s, g, d, s, p, d, a, w, f, s, q, g, t, g, y, p, l,
h, c, w, r, c, m, v, k, l, r, t, g, a, p, e, s, l, d, a, t, r, r, t,
a, f, p, s, s, k, k, f, l, t, l, s, q, r, p, d, l, s, g, d, h, q, s,
n, y, v, p, s, l, g, g, g, s, l, r, n, l, t, d, i, p, v, h, i, p, g,
h, q, e, w, p, l, y, v, i, i, a, w, p, v, l, s, r, h, g, r, r, q, a,
s, s, t, y, p, y, r, a, e, t, l, n, w, t, r, d, i, a, l, g, s, g, p,
p, p, p, r, k, t, s, g, l, p, e, r, a, f, q, q, p, f, i, s, t, k, m,
d, g, i, s, p, t, a, n, r, l, m, a, k, f, t, s, h, q, r, l, t, y, s,
g, f, m, v, s, r, r, p, v, t, r, v, l, w, e, r, v, i, l, i, k, s, r,
t, s, g, l, d, r, f, t, v, s, p, d, s, l, r, g, n, l, k, g, g, r, e,
l, r, t, a, v, r, n, r, m, q, g, e, l, g, t, c, t, k, s, a, h, l, t,
r, a, v, e, f, c, q, r, s, t, s, c, l, p, n, r, p, c, l, h, l, a, p,
v, a, r, w, s, t, d, l, f, l, v, k, i, m, a, l, e, a, r, f, v, a, l,
g, l, v, p, l, w, v, h, a, p, i, p, a, m, s, l, v, l, d, e, i, v, f,
r, n, r, m, n, k, s, s, p, p, l, c, s, p, d, e, a, s, n, l, g, m, y,
r, l, g, s, t, h, g, d, v, s, r, a, r, p, a, l, t, s, r, r, n, m, c,
g, h, d, y, a, p, s, l, t, f, h, d, r, p, k, a, w, h, c, r, t, l, p,
r, q, t, n, v, p, q, t, h, t, f, t, s, q, n, t, s, s, i, y, r, t, v,
r, r, q, a, v, m, c, c, l, f, d, v, s, s, v, a, g, n, g, r, n, l, q,
t, g, c, g, t, i, e, e, g, l, s, t, g, h, v, p, t, s, c, r, r, l, n,
s, g, l, a, g, l, f, l, e, n, f, h, v, g, e, i, a, e, v, a, s, g, e,
v, f, l, a, v, t, q, i, s, s, p, v, l, d, i, p, d, g, g, t, f, l, n,
v, s, h, e, y, a, g, t, e, l, c, m, l, l, g, y, g, t, s, l, g, a, w,
t, v, i, p, k, s, g, s, w, t, i, l, y, y, g, v, h, p, c, k, g, s, g,
m, a, h, y, t, t, d, r, q, s, s, v, v, l, r, y, l, t, h, f, r, i, s,
a, p, r, i, r, w, k, a, g, s, v, r, r, r, a, l, s, n, i, c, a, a, g,
h, g, s, p, q, l, s, r, k, h, r, g, r, g, p, d, p, f, l, s, g, n, l,
g, k, d, p, k, f, i, r, r, g, g, s, m, d, c, v, g, t, c, w, i, g, g,
a, h, e, r, l, a, p, y, v, r, g, h, f, i, r, p, r, d, v, a, s, i, w,
i, s, t, k, l, y, r, y, l, f, t, t, p, s, r, r, e, s, s, q, s, i, p,
t, w, l, e, t, i, m, n, g, g, s, f, i, n, y, e, y, l, n, i, s, l, s,
k, p, r, n, e, w, y, c, s, l, a, v, a, p, h, y, f, r, h, s, s, l, i,
t, v, v, g, r, y, y, t, a, g, t, m, c, d, e, s, s, n, a, t, i, e, g,
l, n, g, e, t, r, i, d, p, v, l, a, r, y, g, n, d, s, g, y, n, g, t,
r, r, t, t, s, y, i, i, p, l, p, y, h, k, k, s, i, p, t, r, g, y, y,
c, k, s, g, p, s, l, h, p, l, l, c, a, c, n, n, s, d, g, v, g, r, v,
g, y, v, l, m, c, r, l, d, k, l, v, s, h, c, t, k, s, r, r, q, l, q,
s, c, a, q, r, w, f, g, v, k, t, k, r, r, a, v, g, l, l, i, i, k, v,
q, r, q, c, n, s, f, s, l, v, l, d, s, k, i, d, t, c, f, t, s, e, k,
e, s, a, c, s, v, p, g, l, w, l, f, v, v, t, l, l, t, p, p, l, g, l,
f, a, t, l, l, s, g, t, a, t, c, g]
```

Om bovenstaande uitkomst te verifiëren heb ik manueel de eerste paar triples nagekeken. Die zijn: $[a, c, a, a, g, c, a, c, g, a, a, g, ...]$. Die zijn inderdaad correct. De tabel voor de vertaling van deze specifieke triples volgt zodat je deze zelf kan verifiëren. De volledige tabel kan je trouwens terugvinden in hoofdstuk 6.

```
from_RNA_to_protein_table([a,c,a], t).
from_RNA_to_protein_table([a,g,c], s).
from_RNA_to_protein_table([a,c,g], t).
from_RNA_to_protein_table([a,a,g], k).
```

Vervolgens doe ik hetzelfde maar dan met de implementatie van de standaard Prolog-versie. Verrassend genoeg krijgen we hier zo goed als instant een resultaat. Met een diff-checker heb ik gekeken als beide versies dezelfde resultaat geven. Dit is wel het geval. Voor de consistentie te bewaren heb ik deze testen meerdere keren uitgevoerd. Hiervoor kwam ik telkens op hetzelfde resultaat uit.

Bovenstaande test is een input met 1000 triples. Ik ben vervolgens stapsgewijs omlaaggegaan wat de grootte van data betreft. Nu volgt een tabel met de metingen die ik heb uitgevoerd. Merk op dat dit alleen resultaten zijn voor de versie met het expr-predicaat en path expressions. Dit komt omdat telkens de standaard Prolog-versie zo goed als instant een oplossing gaf. Deze tests heb ik telkens een paar keer uitgevoerd en de gemiddelde tijd genomen.

Aantal triples	**Uitvoer (min:sec)**
1000	9:04
750	3:52
500	2:18
250	0:8

De conclusie is dat het mogelijk is om met grote hoeveelheid data te werken. Voor sommige implementaties werkt het wel correct, ook al heb ik eerder gevonden dat het bij recursie niet altijd goed werkt door oneindige loops (hoofdstuk 4.3.4). In dit geval treedt er geen oneindige loop op. Het valt wel op dat de berekeningen heel lang duren bij grote hoeveelheid data.

Hoofdstuk 7

Conclusies

Deze masterproef is begonnen met het leren van Prolog. Ik heb hiervoor het online handboek gebruikt van *Learn Prolog Now!* [LearnPrologNow]. Van elk bekeken hoofdstuk heb ik een aantal oefeningen van gemaakt om de leerstof te begrijpen. Deze oefeningen zijn vooral praktisch-gericht die de basis van Prolog dekt. Deze oefeningen en ook mijn begrip ervan werden voorgelegd aan mijn promotors waarbij ik ook feedback op heb gekregen. Op het einde van deze cursus heb ik ook een practica gemaakt, een overdekking van meerdere hoofdstukken. Voor dit deel had ik verwacht om met een typische programmeertaal te werken. Omdat ik een aantal programmeertalen kende verwachtte ik dat het leren van Prolog heel gemakkelijk op te vangen zou zijn. In tegendeel, dit vergt meer logisch denkwerk.

Met een klein aantal programma-regels waar goed over is nagedacht kan je al veel mee doen. Hierdoor moet de basis van Prolog ook goed gekend zijn. In vergelijking met andere programmeertalen die ik al kende was dit niet hetzelfde en moest ik een andere manier van denken hanteren. Een simpel voorbeeld, het is niet mogelijk om variabelen zelf te declareren. Je moet weten hoe het Prolog-programma de variabelen voorstelt en hoe die in de loop van het programma zich aanpast door matches te vinden. Al bij al vond ik het wel interessant om met Prolog gewerkt te hebben omdat deze denkwijze zeker in andere velden ook gebruikt wordt. Dat heb ik ook gezien in deze masterproef zelf waarbij ik refereer naar Sequence Datalog. Het moeilijkste van dit onderdeel was het begrijpen van recursie. Hiervoor hielp het om programma-bomen (of search-trees) op te stellen om de beslissingen en het verloop van het programma visueel voor te stellen.

Na het leren van Prolog ben ik overgegaan met het leren van Sequence Datalog. Hiervoor heb ik een aantal papers gekregen om het in te studeren. De overstap van Prolog naar Sequence Datalog vond ik redelijk makkelijk omdat het dezelfde denkwijze hanteert. Hiervoor ben ik minder praktisch te werk gegaan in tegenstelling tot het leren van Prolog, maar meer gefocust op het begrijpen van programma's in Sequence Datalog. Het begrijpen ervan was belangrijk omdat ik die later moest vertalen naar, of simuleren in Prolog.

Als voorbereiding heb ik hier een deel oefeningen van gemaakt in Prolog zelf, dus nog niet met het expr-predicaat, om die daarna om te zetten en te vergelijken met de path-expressions in Prolog. Hier begon echt het maken van oefeningen en er goed over nadenken. Het maken van deze oefeningen had soms meerdere iteraties nodig en feedback van mijn begeleiders. Ik denk dat het probleem eerder lag dat het moeilijk te debuggen

is, maar ik had ondertussen geleerd hoe je een programma stap voor stap uitvoert en tussenresultaten kan berekenen en bekijken.

Vervolgens werd het expr-predicaat opgesteld om path-expressions zoals in Sequence Datalog voor te stellen in Prolog. Het opstellen ervan was in samenwerking met mijn begeleiders. Het gebruik ervan was niet zo simpel naar mijn mening. Ik had toch enige tijd en meerdere oefeningen nodig om vlot deze vertaling te maken van Sequence Datalog naar Prolog met behulp van path expressions. Eenmaal ik de vergelijking kon maken met normaal Prolog, de versie met path-expressions én de Sequence Datalog begon ik er meer in te komen.

Wat we nu weten is dat het mogelijk is om path expressions te ondersteunen in Prolog vanuit Sequence Datalog. Sequence Datalog, die nog eerder een theoretische en niet uitvoerbare taal is. We weten dat het werkt op algemene toepassingen. Ook dat het niet zo goed werkt in toepassing met recursie. Hiervoor weten we ook de reden ervan. Wat we nog niet weten zijn eventuele oplossingen voor het probleem dat het niet goed werkt op toepassingen met recursie. Toekomstig werk kan zijn om deze implementatie te testen op andere Prolog-omgevingen in de plaats van SWI-Prolog zoals GNU-Prolog.

Tijdens deze masterproef heb ik geleerd om met een programmeertaal om te gaan waar niet al te veel documentatie over te vinden is. Ik ben hiermee omgegaan door de basis en theorie goed te begrijpen vooraleer verder te gaan met het volgende. Door een goede basis heb ik minder last gehad van later niet weten hoe iets gedaan moest worden. Ook kon ik gemakkelijk teruggaan naar mijn eigen notities die ik gemaakt heb. Door deze notities had ik een centrale plaats waar ik naartoe kon gaan als ik niet meer wist wat ik misschien even geleden al geleerd had.

De moeilijkheid in deze masterproef was het schrijven van de paper zelf. Het is heel moeilijk voor mij om uit te drukken in woorden en wat ik denk. In de loop van de masterproef heb ik dan ook geleerd om hier beter mee om te gaan. Ik heb vaak de tendens om direct aan de slag te gaan met iets en er volop aan te werken, maar soms is het goed om een stap terug te zetten en na te denken. Dit gebeurt niet alleen in het schrijven van woorden maar ook tijdens het programmeren voor mij. Ik wist dit al van mezelf door eerdere ervaringen zoals een stage die ik een aantal jaren geleden heb gelopen en een vakantiewerk die ik heb doorlopen.

Wat goed ging in deze masterproef is eigenlijk de progressie op het einde. Voor mij persoonlijk ga ik traag op gang met iets in het begin. De reden hiervoor is omdat ik graag alles wil uitzoeken. Ik wil vaak niet alleen stoppen bij het weten hoe iets werkt, maar ik vind het belangrijk om ook te weten waarom iets werkt. Dat is ook een van de redenen waarom ik aan het schakelprogramma ben begonnen om door te stromen naar de opleiding master in de informatica, komende van een toegepaste bachelor in de informatica.

Bibliografie

[Workflow] Karolin Winter, Florian Stertz, Stefanie Rinderle-Ma, Discovering instance and process spanning constraints from process execution logs, Information Systems, Volume 89, 2020, 101484, ISSN 0306-4379, https://doi.org/10.1016/j.is.2019.101484 (https://www.sciencedirect.com/science/article/pii/S0306437919305368)

[pm4py] pm4py: Process Mining in Python. https://pm4py.fit.fraunhofer.de/

[LearnPrologNow] Learn Prolog Now. Patrick Blackburn, Johan Bos, and Kristina Striegnitz. https://lpn.swi-prolog.org/lpnpage.php?pageid=online

[Expressiveness within Sequence Datalog] Expressiveness within Sequence Datalog Heba Aamer, Jan Hidders, Jan Paredaens, Jan Van den Bussche. PODS'21: Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, June 2021, Pages 70–81, https://doi.org/10.1145/3452021.3458327

[J-Logic] J. Hidders, J. Paredaens, and J. Van den Bussche. 2017. J-Logic: Logical foundations for JSON querying. In Proceedings 36th ACM Symposium on Principles of Databases. ACM, 137–149.

[Presentation Expressiveness within Sequence Datalog] Presentation - Expressiveness within Sequence Datalog.
https://dl.acm.org/doi/10.1145/3452021.3458327

[Sequences Datalog en Transducers] A.J. Bonner and G. Mecca. 2000. Querying sequence databases with transducers. Acta Informatica 36 (2000), 511–544.

[Encyclopedia of Machine Learning] (2011). Sequential Data. In: Sammut, C., Webb, G.I. (eds) Encyclopedia of Machine Learning. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-30164-8_754

[Fifty Years of Prolog and Beyond] Körner P., Leuschel M., Barbosa J., Costa V., Dahl V., Hermenegildo M., Ciatto G. (2022). Fifty Years of Prolog and Beyond. Theory and Practice of Logic Programming, 1-83. doi:10.1017/S1471068422000102