```
*****************************************************************************
*                                                                           *
*  Programm zur Diplomarbeit von Pok-Son Kim                                *
*                                                                           *
*  Thema der Diplomarbeit:                                                  *
*  Optimierung und Simplifikation von Anfragen an eine KL-ONE-Wissensbasis  *
*                                                                           *
*****************************************************************************

WICHTIG WICHTIG WICHTIG WICHTIG WICHTIG WICHTIG WICHTIG WICHTIG WICHTIG WICHTIG
---------------------------------------------------------------------------
Gofer ist eine  l a z y - e v a l u i e r e n d e  funktionale
Programmiersprache. Aus diesem Grund ist zu beachten, da"s im Falle der
zweimaligen Auswertung eines identischen Ausdrucks bei der zweiten
Auswertung eine gegen"uber der ersten Auswertung reduzierte Anzahl an
Reduktionsschritten ben"otigt wird.
Alle im Rahmen der Diplomarbeit oder dieses Programms erscheinende Beispiele
geben den bei der zweiten Auswertung eines Ausdrucks gegebenen Zustand wieder.
---------------------------------------------------------------------------


*******************************************
\/  Deklaration aller primitiver Konzepte  \/
*******************************************

> person :: [String]
> person = ["Stefan", "Gani", "Rainer", "Michael", "Hubert","Sima","Sia",
>           "Zeideh", "Axi", "Ralf", "Andreas", "Carina", "Rueck",
>           "Georg", "Frank", "Siville", "Alexandra", "Steffi", "Fifia",
>           "Madonna", "Maria", "Moshen", "Cosima", "Jan", "Tannja"]

> student :: [String]
> student = ["Gani","Michael","Frank"]

> maennlich :: [String]
> maennlich = ["Stefan", "Rainer", "Michael", "Hubert", "Sia","Moshen",
>         "Frank","Ralf","Axi","Georg","Jan", "Rueck", "Andreas"]

> weiblich :: [String]
> weiblich = ["Siville","Sima","Steffi","Gani","Alexandra","Fifia",
>         "Carina","Maria","Madonna","Cosima","Tannja", "Zeideh"]

> vegetarier :: [String]
> vegetarier = ["Siville", "Sima", "Gani", "Hubert"]

> bottom :: [String]
> bottom = []


*******************************************
\/  Deklaration aller primitiver Rollen  \/
*******************************************

> hat_Kind :: [(String, String)]
> hat_Kind = [("Stefan","Gani"),("Siville","Gani"),("Sima","Sia"),
>           ("Moshen", "Sia"),("Steffi","Michael"),("Rainer","Michael"),
>           ("Gani","Ralf"),("Sia","Ralf"),("Siville","Frank"),("Stefan",
>           "Frank"),("Alexsandra","Axi"),("Michael","Axi"),("Alexsandra",
>           "Georg"),("Michael","Georg"),("Fifia","Carina"),("Georg",
>           "Carina"),("Fifia","Maria"),("Georg","Maria"),
```

```
>          ("Fifia","Madonna"),("Georg","Madonna"),("Steffi","Cosima"),
>          ("Rainer", "Cosima"),("Jan","Stefan"),("Tannja","Steffi"),
>          ("Jan","Steffi")]

> ist_Kusin :: [(String, String)]
> ist_Kusin = [("Gani","Michael"),("Gani","Cosima"),("Frank","Michael"),
>              ("Frank","Cosima"),("Michael","Gani"),("Cosima","Gani"),
>              ("Michael","Frank"),("Cosima","Frank")]

> ist_verheiratet :: [(String, String)]
> ist_verheiratet = [("Siville","Stefan"),("Stefan", "Siville"),("Sima","Moshen"),
>         ("Moshen", "Sima"),("Steffi", "Rainer"),("Rainer", "Steffi"),
>         ("Gani","Sia"),("Sia", "Gani"),("Alexandra","Michael"),("Michael",
>         "Alexandra"),("Fifia","Georg"),("Georg", "Fifia"),("Tannja","Jan"),
>         ("Jan", "Tannja")]

> ist_Lehrer :: [(String, String)]
> ist_Lehrer = [("Gani", "Ralf"), ("Georg", "Maria"), ("Gani", "Carina")]


> r_bottom :: [(String, String)]
> r_bottom = []


*******************************************************************************
\/ Deklaration der Datenstruktur zur Darstellung zusammengesetzter Konzepte \/
*******************************************************************************

> data
>  Konzeptterm = Basisbottom|Person|Student|Maennlich|Weiblich|Vegetarier|Menge[String]|
>          And [Konzeptterm]|
>          Or [Konzeptterm]|
>          Not Konzeptterm|
>          Some Relationenterm Konzeptterm|
>          All Relationenterm Konzeptterm

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
------------------------------------------------------------------------------
 Mit Hilfe der Konstruktorfunktion "Menge [String]" k"onnen Konzeptterme
 dargestellt werden, die  b e l i e b i g e  Mengen von Personen beschreiben.

 Die folgende Beispielanfrage verdeutlicht, wie die Konstruktorfunktion
 "Menge [String]" verwendet werden kann:

       "finde_alle (Some HatKind (Menge ["Axi"]))"
------------------------------------------------------------------------------


*******************************************************************************
\/ Deklaration der Datenstruktur zur Darstellung zusammengesetzter Rollen \/
*******************************************************************************

> data
>  Relationenterm = Relationenbottom| HatKind| IstKusin|
>                   IstVerheiratet| IstLehrer|
>                   Relation [(String, String)]|
>                   RAnd [Relationenterm]|
>                   ROr [Relationenterm]|
>                   Compose [Relationenterm]|
>                   Inverse (Relationenterm)|
>                   Restrict (Relationenterm) (Konzeptterm)
```

```
>

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
------------------------------------------------------------------------------
 Mit Hilfe der Konstruktorfunktion "Relation [(String, String)]" k"onnen
 Relationenterme dargestellt werden, die  b e l i e b i g e  Relationen
 beschreiben.

 Die folgende Beispielanfrage verdeutlicht, wie die Konstruktorfunktion
 "Relation [(String, String)]"
 verwendet werden kann:

       "finde_alle (RAnd HatKind (Relation [("Stefan", "Axi")]))"
------------------------------------------------------------------------------


*******************************************************************************
\/ Vereinbarung des Datentyps Konzeptterm als Auspr"agung der Typklasse Eq \/
*******************************************************************************

> instance   Eq Konzeptterm  where
>    Basisbottom == Basisbottom = True
>    Person == Person  = True
>    Student == Student  = True
>    Maennlich == Maennlich = True
>    Weiblich == Weiblich  = True
>    Vegetarier == Vegetarier  = True

>    Menge as == (Menge bs) = as `subset` bs && (bs `subset` as)
>                        where as `subset` bs = all(`elem` bs)as
>    And as == (And bs) = as `subset` bs && (bs `subset` as)
>                        where as `subset` bs = all(`elem` bs)as
>    Or as == (Or bs) = as `subset` bs && (bs `subset` as)
>                        where as `subset` bs = all(`elem` bs)as
>    (Not a) == (Not b) = a == b
>    Some r a == (Some s b) = r == s && a == b
>    All r a == (All s b) = r == s && a == b
>    _  ==  _  = False


*******************************************************************************
\/ Vereinbarung des Datentyps Relationenterm als Auspr"agung der Typklasse Eq \/
*******************************************************************************

> instance   Eq Relationenterm     where
>    Relationenbottom == Relationenbottom = True
>    HatKind == HatKind  = True
>    IstKusin == IstKusin  = True
>    IstVerheiratet == IstVerheiratet = True
>    IstLehrer == IstLehrer = True
>    Relation rs == (Relation ts) = rs `subset` ts && ts `subset` rs
>                        where rs `subset` ts = all(`elem` ts)rs
>    Compose r == (Compose s) = r == s
>    Inverse r == (Inverse s) = r == s
>    ROr rs == (ROr ts) = rs `subset` ts && ts `subset` rs
>                        where rs `subset` ts = all(`elem` ts)rs
>    RAnd rs == (RAnd ts) = rs `subset` ts && ts `subset` rs
>                        where rs `subset` ts = all(`elem` ts)rs
>    Restrict r a == (Restrict s b) = r == s && a == b
>    _ == _   = False
```

```
**********************************************************
\/                                \/
\/ S i m p l i f i k a t i o n s f u n k t i o n e n \/
\/                                \/
**********************************************************
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
- K o n z e p t beschreibende Anfrageausdr"ucke werden unter Benutzung der
  Simplifikationsfunktion "simple_term" simplifiziert.

- R o l l e n beschreibende Anfrageausdr"ucke werden unter Benutzung der
  Simplifikationsfunktion "simple_r_term" simplifiziert.

In einigen Simplifikationen werden die Operatoren `Vereinigung', `Durchschnitt'
und `Composition' der universellen Identit"aten auf n-stellige Operatoren
ausgedehnt. (siehe auch Abschnitt 5.4.5 der Diplomarbeit)
---

```
**********************************************************************
\/ Innerhalb der Simplifikationsfunktionen `simple_term' und     \/
\/ `simple_r_term' benutzte Hilfsfunktion                        \/
**********************************************************************
```

```
> gleich_term :: Eq [a] => [a] -> [a] -> [a]

> gleich_term [] ys = []
> gleich_term xs [] = []
> gleich_term (x:xs) (ys)
>           | any(x==)ys = x: gleich_term xs ys
>           | otherwise = gleich_term xs ys
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Die polymorphe Hilfsfunktion `gleich_term' wird in der 4-ten und 9-ten
Simplifikation verwendet, um die Schnittmenge zweier Listen von Konzepten zu
bilden. Desweitern wird die Hilfsfunktion `gleich_term' in der 16-ten
Simplifikation zur Bildung des Schnitts zweier Listen von Rollen verwendet.
---

```
> simple_term :: Konzeptterm  ->  Konzeptterm
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
In der Definition der Funktion `simple_term' ist f"ur jede zum
Simplifizieren benutzte universelle Identit"at mindestens eine Gleichung
definiert, die die zur universellen Identit"at geh"orige Simplifikation
realisiert.
---

WARNUNG WARNUNG WARNUNG WARNUNG WARNUNG WARNUNG WARNUNG WARNUNG WARNUNG
---
Die Ordnung der Gleichungen innerhalb der Funktion `simple_term' ist teilweise
von gro"ser Bedeutung. Eine Ver"anderung der Ordnung der Gleichungen kann dazu
f"uhren, da"s eine zu den Gleichungen geh"orige Simplifikation unrealisierbar
wird.

Beispiel:
Das Vertauschen der 5-ten und 9-ten Gleichung hat zur Folge, da"s die 5-te
Simplifikation unrealisierbar wird.
---

```
 *-----------------------------------------------------*
| Simplifikation 1                                    |
| (Unabh"angig von Instanztermen)                     |
|                                                     |
| Universelle Identit"at: 0.a = a.0 = 0               |
 *-----------------------------------------------------*
```

\/ Ausdehnung auf den n-stelligen terminologischen Operator `and' \/

```
> simple_term (And as)
>           |any(Weiblich ==)as && any(Maennlich ==)as
>                     -- weiblich.maennlich = 0
>            || any((Not Weiblich) ==)as && any((Not Maennlich)==)as
>            = Basisbottom
>           | any (Basisbottom ==) as = Basisbottom
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
In Simplifikation 1 k"onnen Konzepterme verwendet werden, deren Interpretation
identisch der leeren Menge ist.
Beispiel: (and weiblich maennlich)
---

```
 *-----------------------------------------------------*
| Simplifikation 2                                    |
| (Unabh"angig von Instanztermen)                     |
|                                                     |
| Universelle Identit"at: a.1 = 1.a = a               |
 *-----------------------------------------------------*
```

\/ Ausdehnung auf den n-stelligen terminologischen Operator `and' \/

```
> simple_term (And as)
>           | any (Person ==) as
>             = simple_term (And (as \\ [Person]))
>           | any ((Not Basisbottom)==) as
>             = simple_term (And (as \\ [Not Basisbottom]))
```

```
 *-----------------------------------------------------*
| Simplifikation 3                                    |
| (Unabh"angig von Instanztermen)                     |
|                                                     |
| Universelle Identit"at:                             |
|   a.(a + b) = a.(b + a) = (a + b).a = (b + a).a = a |
 *-----------------------------------------------------*
```

\/ Ausdehnung auf den n-stelligen terminologischen Operator `or' \/

```
> simple_term (And [a, Or bs])
>           |any (a==)bs = simple_term a
> simple_term (And [(Or bs), a])
>           |any (a==)bs = simple_term a
```

```
*-------------------------------------------------------*
| Simplifikation 4                                      |
| (Unabh"angig von Instanztermen)                       |
|                                                       |
| Universelle Identit"at:                               |
| (a + b).(a + c) = (b + a).(a + b) = (a + b).(c + a)   |
|                 = (b + a).(c + a) = a + (b.c)         |
*-------------------------------------------------------*

> simple_term (And [Or [a, b], Or [c, d]])
>             | a == c = simple_term (Or [a, And [b, d]])
>             | a == d = simple_term (Or [a, And [b, c]])
>             | b == c = simple_term (Or [And [a, d], b])
>             | b == d = simple_term (Or [And [a, c], b])
```

\/ Ausdehnung auf den n-stelligen terminologischen Operator `or' \/

```
> simple_term (And [(Or as), (Or bs)])
>             | all (`elem` as)bs = simple_term (Or bs)
>             | all (`elem` bs)as = simple_term (Or as)
>             | not (null gterm)
>             = simple_term (Or (gterm ++
>                 [And (((as) \\ gterm)
>                     ++ ((bs) \\ gterm))]))
>              where gterm =  gleich_term (as) (bs)
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Die folgenden 3 Ausschnitte aus einer Gofer-Sitzung zeigen ein Beispiel f"ur
die Simplifikation von Ausdr"ucken, in denen terminologische Operatoren
vorkommen, die mehr als zweistellig sind:

```
 Simplifikationsanfrage:
 ? simple_term (And [(Or [Student, Maennlich, Vegetarier]),
                     (Or [Student, Maennlich, Weiblich])])
 Or [Student, Maennlich, And [Vegetarier, Weiblich]]
 (304 reductions, 525 cells)

 Anfrage f"ur den nicht simplifizierten Ausdruck:
 ? finde_alle (And [(Or [Student, Maennlich, Vegetarier]),
                    (Or [Student, Maennlich, Weiblich])])
 ["Andreas", "Axi", "Frank", "Gani", "Georg", "Hubert", "Jan", "Michael", "Moshen", "Rainer",
 "Ralf", "Rueck", "Sia", "Sima", "Siville", "Stefan"]
 (1282 reductions, 2384 cells)

 Anfrage mit  Simplifikation:
 ? finde_alle (Or [Student, Maennlich, And [Vegetarier, Weiblich]])
 ["Andreas", "Axi", "Frank", "Gani", "Georg", "Hubert", "Jan", "Michael", "Moshen", "Rainer",
 "Ralf", "Rueck", "Sia", "Sima", "Siville", "Stefan"]
 (583 reductions, 1180 cells)
```

---

```
*-------------------------------------------------------*
| Simplifikation 5                                      |
| (Unabh"angig von Instanztermen)                       |
|                                                       |
| Universelle Identit"at: a.a' = 0                      |
```

```
 *-----------------------------------------------------*

> simple_term (And [a, Not b])
>            | a == b = Basisbottom
> simple_term (And [Not a, b])
>            | a == b = Basisbottom




 *-----------------------------------------------------*
 | Simplifikation 6                                    |
 | (Unabh"angig von Instanztermen)                     |
 |                                                     |
 | Universelle Identit"at: a + 1 = 1 + a = 1           |
 *-----------------------------------------------------*

\/ Ausdehnung auf den n-stelligen terminologischen Operator `or' \/

> simple_term (Or as)
>     | any (Person ==) as = Person


 *-----------------------------------------------------*
 | Simplifikation 7                                    |
 | (Unabh"angig von Instanztermen)                     |
 |                                                     |
 | Universelle Identit"at:                             |
 | a + (a.b) = a + (b.a) = (a.b) + a = (b.a) + a = a   |
 *-----------------------------------------------------*

\/ Ausdehnung auf den n-stelligen terminologischen Operator `and' \/

> simple_term (Or [a, (And bs)])
>        |any (a==)bs = simple_term a
> simple_term (Or [(And bs), a])
>        |any (a==)bs = simple_term a


 *-------------------------------------------------------------------------------*
 | Simplifikation 8                                                              |
 | (Abh"angig von Instanztermen - Komplexit"atsabsch"atzungsfunktion angewendet)  |
 |                                                                               |
 | Universelle Identit"at:                                                       |
 |    a.b + a.c = b.a + a.c = b.a + c.a = a.b + c.a = a.(b + c)                  |
 *-------------------------------------------------------------------------------*

> simple_term (Or [And [a, b], And [c, d]])
>            | a == c && snd(komplexitaet_K (Or [And [a, b], And [c, d]])) >
>                      snd(komplexitaet_K (And [a, Or[b, d]]))
>              = simple_term (And [a, Or[b, d]])
>            | a == d && snd(komplexitaet_K (Or [And [a, b], And [c, d]])) >
>                      snd(komplexitaet_K (And [a, Or[b, c]]))
>              = simple_term (And [a, Or[b, c]])
>            | b == c && snd(komplexitaet_K (Or [And [a, b], And [c, d]])) >
>                      snd(komplexitaet_K (And [b, Or[a, d]]))
>              = simple_term (And [b, Or[a, d]])
>            | b == d && snd(komplexitaet_K (Or [And [a, b], And [c, d]])) >
>                      snd(komplexitaet_K (And [b, Or[a, c]]))
>              = simple_term (And [b, Or[a, c]])
```

\/ Ausdehnung auf den n-stelligen terminologischen Operator `and' \/

```
> simple_term (Or [(And as), (And bs)])
>           | all (`elem` bs)as
>             = simple_term (And as)
>           | all (`elem` as)bs
>             = simple_term (And bs)
```

```
*-----------------------------------------------*
| Simplifizierung in umgekehrter Richtung      |
|                                               |
| a.(b + c) = a.b + a.c  (a + b).c = a.c + b.c |
*-----------------------------------------------*
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Es gibt Anfrageausdr"ucke in der Form a.(b + c), bei denen der "aquivalente Term
`a.b + a.c' effizienter auswertbar ist.
---

```
> simple_term (And [a, Or [b, c]])
>           | snd(komplexitaet_K (And [a, Or [b, c]])) >
>             snd(komplexitaet_K (Or [And [a, b], And [a, c]]))
>             = Or [And [a, b], And [a, c]]
> simple_term (And [Or [a, b], c])
>           | snd(komplexitaet_K (And [Or [a, b], c])) >
>             snd(komplexitaet_K (Or [And [a, c], And [b, c]]))
>             = Or [And [a, c], And [b, c]]
```

```
 *-----------------------------------------------------*
| Simplifikation 9                                    |
| (Unabh"angig von Instanztermen)                     |
|                                                     |
| Universelle Identit"at: a.a = a                     |
 *-----------------------------------------------------*
```

\/ Ausdehnung auf den n-stelligen terminologischen Operator `and' \/

```
> simple_term (And as) = And (nub as)
```

```
 *-------------------------------------------------------------------------------*
| Simplifikation 10                                                             |
| (Abh"angig von Instanztermen - Komplexit"atsabsch"atzungsfunktion angewendet)  |
|                                                                               |
| Universelle Identit"aten:                                                     |
|      r : a + s : a = (r + s) : a     r : a + r : b = r : (a + b)              |
 *-------------------------------------------------------------------------------*
```

```
> simple_term (Or [(Some r a), (Some s b)])
>           | r == s && a == b = Some r a
>           | a == b &&
>             snd(komplexitaet_K (Or [(Some r a), (Some s b)])) >
>             snd(komplexitaet_K (Some (ROr [r, s]) a))
>               = Some (ROr [r, s]) a
>           | r == s &&
>             snd(komplexitaet_K (Or [(Some r a), (Some s b)])) >
>             snd(komplexitaet_K (Some r (Or [a, b])))
```

```
>               = Some r (Or [a, b])
>           | otherwise = Or [(Some r a), (Some s b)]
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Die folgenden 3 Ausschnitte aus einer Gofer-Sitzung zeigen ein Beispiel f"ur
die Verwendung der 10ten Simplifikation:

```
 Simplifikationsanfrage:
 ? simple_term (Or [Some HatKind Student, Some HatKind Vegetarier])
 Some HatKind (Or [Student, Vegetarier])
 (317 reductions, 899 cells)

 Anfrage f"ur den nicht simplifizierten Ausdruck:
 ? finde_alle (Or [Some HatKind Student, Some HatKind Vegetarier])
  ["Rainer", "Siville", "Stefan", "Steffi"]
 (2769 reductions, 4886 cells)

 Anfrage f"ur den simplifizierten Ausdruck:
 ? finde_alle (Some HatKind (Or [Student, Vegetarier]))
 ["Rainer", "Siville", "Stefan", "Steffi"]
 (2071 reductions, 3622 cells)
```

---

```
*-----------------------------------------------------------*
| Simplifizierung in umgekehrter Richtung                   |
|                                                           |
|  (r + s) : a = r : a + s : a   r : (a + b) = r : a + r : b |
*-----------------------------------------------------------*

> simple_term (Some (ROr [r, s]) a)
>   | snd(komplexitaet_K (Some (ROr [r, s]) a)) >
>     snd(komplexitaet_K (Or [(Some r a), (Some s a)]))
>       = Or [(Some r a), (Some s a)]
> simple_term (Some r (Or [a, b]))
>   | snd(komplexitaet_K (Some r (Or [a, b]))) >
>     snd(komplexitaet_K (Or [(Some r a), (Some r b)]))
>       = Or [(Some r a), (Some r b)]
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Die folgenden 3 Ausschnitte aus einer Gofer-Sitzung zeigen ein Beispiel f"ur
die 11te Simplifikation:

```
 Simplifikationsanfrage:
 ? simple_term (Some (ROr[HatKind, IstVerheiratet]) Student)
 Or [Some HatKind Student, Some IstVerheiratet Student]
 (210 reductions, 682 cells)

 Anfrage f"ur den nicht simplifizierten Ausdruck:
 ? finde_alle (Some (ROr[HatKind, IstVerheiratet]) Student)
 ["Alexandra", "Rainer", "Sia", "Siville", "Stefan", "Steffi"]
 (5186 reductions, 8747 cells)

 Anfrage f"ur den simplifizierten Ausdruck:
 ? finde_alle (Or [Some HatKind Student, Some IstVerheiratet Student])
 ["Alexandra", "Rainer", "Sia", "Siville", "Stefan", "Steffi"]
 (2323 reductions, 4082 cells)
```

---

```
 *-------------------------------------------------*
| Simplifikation 11                               |
| (Unabh"angig von Instanztermen)                 |
|                                                 |
| Universelle Identit"at a + a = a                |
 *-------------------------------------------------*
```

\/ Ausdehnung auf den n-stelligen terminologischen Operator `or' \/

```
> simple_term (Or as) = Or (nub as)
```

```
 *-------------------------------------------------*
| Simplifikation 12                               |
| (Unabh"angig von Instanztermen)                 |
|                                                 |
| Universelle Identit"at:  0' = 1   1' = 0        |
 *-------------------------------------------------*
```

```
> simple_term (Not Basisbottom) = Person
> simple_term (Not Person) = Basisbottom
> simple_term (Not (And as))
>   |any(Weiblich ==)as && any(Maennlich ==)as
>         = Person
```

```
 *-------------------------------------------------*
| Simplifikation 13                               |
| (Unabh"angig von Instanztermen)                 |
|                                                 |
| Universelle Identit"at:   a" = a                |
 *-------------------------------------------------*
```

```
> simple_term (Not (Not a)) = a
```

```
> simple_term a = a
```

```
> simple_r_term :: Eq Relationenterm =>
>                  Relationenterm -> Relationenterm
```

```
 *-------------------------------------------------*
| Simplifikation 14                               |
| (Unabh"angig von Instanztermen)                 |
|                                                 |
| Universelle Identit"at:   r.0 = 0.r = 0         |
 *-------------------------------------------------*
```

```
> simple_r_term (RAnd as)
>     | any (HatKind ==) as &&
>       any (IstVerheiratet ==) as  -- "HatKind . IstVerheiratet = {}"
>           = Relationenbottom
>     | any (HatKind ==) as &&
>       any (IstKusin==) as       -- "HatKind . IstKusin = {}"
>           = Relationenbottom
```

```
>


 *-----------------------------------------------------------*
| Simplifikation 15                                         |
| (Unabh"angig von Instanztermen)                           |
|                                                           |
| Universelle Identit"at:                                   |
|  r . (r + s) = r . (s + r) = (r + s) . r = (s + r) . r = r |
 *-----------------------------------------------------------*

\/ Ausdehnung auf den n-stelligen terminologischen Operator `or' \/

> simple_r_term (RAnd [r, ROr rs])
>       | any (r==) rs = simple_r_term r
> simple_r_term (RAnd [(ROr rs), r])
>       | any (r==) rs = simple_r_term r


 *-------------------------------------------------------*
| Simplifikation 16                                     |
| (Unabh"angig von Instanztermen)                       |
|                                                       |
| Universelle Identit"at:                               |
|  (r + s).(r + t) = (s + r).(r + t) = (r + s).(t + r)  |
|                = (s + r).(t + r) = r + (s.t)          |
 *-------------------------------------------------------*

> simple_r_term (RAnd [ROr [r, s], ROr [t, u]])
>             | r == t = simple_r_term (ROr [r, RAnd [s, u]])
>             | r == u = simple_r_term (ROr [r, RAnd [s, t]])
>             | s == t = simple_r_term (ROr [RAnd [r, u], s])
>             | s == u = simple_r_term (ROr [RAnd [r, t], s])

\/ Ausdehnung auf den n-stelligen terminologischen Operator `or' \/

> simple_r_term (RAnd [(ROr rs), (ROr ts)])
>             | all (`elem` rs)ts = simple_r_term (ROr ts)
>             | all (`elem` ts)rs = simple_r_term (ROr rs)
>             | not (null glrterm)
>             = simple_r_term (ROr (glrterm ++
>               [RAnd (((rs) \\ glrterm)
>                   ++ ((ts) \\ glrterm))]))
>               where glrterm = gleich_term rs ts


 *-------------------------------------------------------*
| Simplifikation 17                                     |
| (Unabh"angig von Instanztermen)                       |
|                                                       |
| Universelle Identit"at:  r~ . s~ = (r . s)~           |
 *-------------------------------------------------------*

> simple_r_term (RAnd [Inverse r1, Inverse r2])
>     = simple_r_term (Inverse (RAnd [r1, r2]))

\/ Ausdehnung auf den n-stelligen terminologischen Operator `and' \/

> simple_r_term (RAnd [Inverse r1, Inverse r2, Inverse r3])
>     = simple_r_term (Inverse (RAnd [r1, r2, r3]))
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Der obigen  Ausdehnung auf den n-stelligen Operator `and' liegt folgende
Herleitung zugrunde:

```
    (r1~.r2~.r3~...rn~) = (...((r1~.r2~).r3~)...rn~) (Assoziativitaet der
                                        Booleschen Algebra)
                        = (...((r1.r2)~.r3~)...rn~)
                        = (...(r1.r2.r3)~...rn~)
                        ....
                        = (r1.r2.r3...rn)~
```

---

```
*-----------------------------------------------------*
| Simplifikation 18                                   |
| (Unabh"angig von Instanztermen)                     |
|                                                     |
| Universelle Identit"at:   r . r = r                 |
*-----------------------------------------------------*
```

\/ Ausdehnung auf den n-stelligen terminologischen Operator `and' \/

```
> simple_r_term (RAnd rs) = (RAnd (nub rs))
```

```
*-----------------------------------------------------*
| Simplifikation 19                                   |
| (Unabh"angig von Instanztermen)                     |
|                                                     |
| Universelle Identit"at:   r~~ = r                   |
*-----------------------------------------------------*
```

```
> simple_r_term (Inverse (Inverse r)) = simple_r_term r
```

```
*-----------------------------------------------------*
| Simplifikation 20                                   |
| (Unabh"angig von Instanztermen)                     |
|                                                     |
| Universelle Identit"at: r + (r . s) = r             |
*-----------------------------------------------------*
```

\/ Ausdehnung auf den n-stelligen terminologischen Operator `and' \/

```
> simple_r_term (ROr [r, RAnd rs])
>       | any (r==) rs = simple_r_term r
> simple_r_term (ROr [RAnd rs, r])
>       | any (r==) rs = simple_r_term r
```

```
*-----------------------------------------------------*
| Simplifikation 21                                   |
| (Abh"angig von Instanztermen -                      |
|  Komplexit"atsabsch"atzungsfunktion angewendet)     |
|                                                     |
| Universelle Identit"at:  r~ + s~ = (r + s)~         |
*-----------------------------------------------------*
```

```
> simple_r_term (ROr [Inverse r1, Inverse r2])
>       | snd (komplexitaet_R (ROr [Inverse r1, Inverse r2])) >
```

```
>           snd (komplexitaet_R (Inverse (ROr [r1, r2])))
>         = (Inverse (ROr [r1, r2]))

\/ Ausdehnung auf den n-stelligen terminologischen Operator `or' \/

> simple_r_term (ROr [Inverse r1, Inverse r2, Inverse r3])
>       | snd (komplexitaet_R (ROr [Inverse r1, Inverse r2, Inverse r3])) >
>           snd (komplexitaet_R (Inverse (ROr [r1, r2, r3])))
>       = (Inverse (ROr [r1,r2,r3]))

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
--------------------------------------------------------------------------------
Der obigen  Ausdehnung auf n-stellige Operationen liegt folgende Herleitung
zugrunde:

 (r1+r2+r3+...+rn)~ = {r1+(r2+r3+...+rn)}~ (Assoziativitaet der
                                    Booleschen Algebra)
                    = r1~+(r2+r3+...+rn)~
                    ...
                    = r1~+r2~+...+rn~
--------------------------------------------------------------------------------

*-----------------------------------------*
| Simplifizierung in umgekehrter Richtung  |
|                                          |
| (r + s)~ = r~ + s~                       |
*-----------------------------------------*

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
--------------------------------------------------------------------------------
Es gibt Anfrageausdr"ucke in der Form (Inverse (ROr [r1, r2])) bzw.
(Inverse (ROr [r1,r2,r3])), die in einen Ausdruck der Form
(ROr [Inverse r1, Inverse r2]) bzw. (ROr [Inverse r1, Inverse r2, Inverse r3])
"uberf"uhrt werden k"onnen.
--------------------------------------------------------------------------------



> simple_r_term (Inverse (ROr [r1, r2]))
>     | snd(komplexitaet_R (Inverse (ROr [r1, r2]))) >
>       snd(komplexitaet_R (ROr [Inverse r1, Inverse r2]))
>      = ROr [Inverse r1, Inverse r2]

> simple_r_term (Inverse (ROr [r1,r2,r3]))
>     | snd(komplexitaet_R (Inverse (ROr [r1,r2,r3]))) >
>       snd(komplexitaet_R (ROr [Inverse r1, Inverse r2, Inverse r3]))
>      = ROr [Inverse r1, Inverse r2, Inverse r3]




 *-------------------------------------------------------------------------------*
| Simplifikation 22                                                              |
| (Abh"angig von Instanztermen - Komplexit"atsabsch"atzungsfunktion angewendet) |
|                                                                                |
| Universelle Identit"aten:  r;t + s;t = (r+s);t   r;s + r;t = r;(s+t)           |
 *-------------------------------------------------------------------------------*

> simple_r_term (ROr [Compose [r, t], Compose [s, p]])
>             | t == p &&
>                 snd(komplexitaet_R (ROr [Compose [r, t], Compose [s, p]])) >
```

```
>               snd(komplexitaet_R (Compose [ROr [r, s], t]))
>             = Compose [ROr [r, s], t]
>           | r == s &&
>               snd(komplexitaet_R (ROr [Compose [r, t], Compose [s, p]])) >
>               snd(komplexitaet_R (Compose [r, ROr [t, p]]))
>             = Compose [r, ROr [t, p]]
>           | r == s && t == p = (Compose [r, t])
>           | otherwise = ROr [Compose [r, t], Compose [s, p]]
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Die Bedeutung dieser Simplifikation zeigt die folgende Beispielanfrage:

```
 Simplifikationsanfrage:
 ? simple_r_term (ROr [Compose [HatKind, IstVerheiratet ], Compose [HatKind,
 IstKusin]])
 ROr [Compose [HatKind, IstVerheiratet], Compose [HatKind, IstKusin]]
 (488 reductions, 1158 cells)

? finde_Relation (ROr [Compose [HatKind, IstVerheiratet],
                       Compose [HatKind, IstKusin]])
[("Alexsandra","Fifia"), ("Jan","Rainer"), ("Jan","Siville"),
("Michael","Fifia"), ("Moshen","Gani"), ("Rainer","Alexandra"),
("Rainer","Frank"), ("Rainer","Gani"), ("Sima","Gani"), ("Siville","Cosima"),
("Siville","Michael"), ("Siville","Sia"), ("Stefan","Cosima"),
("Stefan","Michael"), ("Stefan","Sia"), ("Steffi","Alexandra"), ("Steffi","Frank"),
("Steffi","Gani"), ("Tannja","Rainer")]
(7789 reductions, 12027 cells)

? finde_Relation (Compose [HatKind, ROr [IstVerheiratet, IstKusin]])
[("Alexsandra","Fifia"), ("Jan","Rainer"), ("Jan","Siville"),
("Michael","Fifia"), ("Moshen","Gani"), ("Rainer","Alexandra"),
("Rainer","Frank"), ("Rainer","Gani"), ("Sima","Gani"), ("Siville","Cosima"),
("Siville","Michael"), ("Siville","Sia"), ("Stefan","Cosima"),
("Stefan","Michael"), ("Stefan","Sia"), ("Steffi","Alexandra"), ("Steffi","Frank"),
("Steffi","Gani"), ("Tannja","Rainer")]
(8124 reductions, 12460 cells)
?
```

---

```
*-----------------------------------------*
| Simplifizierung in umgekehrter Richtung  |
|                                          |
| (r+s);t = r;t + s;t   r;(s+t) = r;s + r;t|                  |
*-----------------------------------------*
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Es gibt Anfrageausdr"ucke in der Form `(r+s);t' bzw. `r;(s+t)', bei denen der
"aquivalente Term `r;t + s;t' bzw. `r;s + r;t' effizienter auswertbar ist.
---

```
> simple_r_term (Compose [ROr [r, s], t])
>     | snd(komplexitaet_R (Compose [ROr [r, s], t])) >
>       snd(komplexitaet_R (ROr [Compose [r, t], Compose [s, t]]))
```

```
>      = ROr [Compose [r, t], Compose [s, t]]
> simple_r_term (Compose [r, ROr [t, p]])
>     | snd(komplexitaet_R (Compose [r, ROr [t, p]])) >
>       snd(komplexitaet_R (ROr [Compose [r, t], Compose [r, p]]))
>      = ROr [Compose [r, t], Compose [r, p]]
```

```
 *-------------------------------------------------------*
 | Simplifikation 23                                     |
 | (Unabh"angig von Instanztermen)                       |
 |                                                       |
 | Universelle Identit"at:  r + r = r                    |
 *-------------------------------------------------------*
```

```
> simple_r_term (ROr rs) = (ROr (nub rs))
```

```
 *-------------------------------------------------------*
 | Simplifikation 24                                     |
 | (Abh"angig von Instanztermen -                        |
 |  Komplexit"atsabsch"atzungsfunktion angewendet)       |
 |                                                       |
 | Universelle Identit"at:  s~ ; r~ = (r;s)~             |
 *-------------------------------------------------------*
```

```
> simple_r_term (Compose [(Inverse r1), (Inverse r2)])
>     | snd(komplexitaet_R (Compose [(Inverse r1), (Inverse r2)])) >
>       snd(komplexitaet_R (Inverse (Compose [r2, r1])))
>     = Inverse (Compose [r2, r1])
> simple_r_term (Compose [(Inverse r1), (Inverse r2),
>                         (Inverse r3)])
>     | snd(komplexitaet_R (Compose [(Inverse r1), (Inverse r2),
>                         (Inverse r3)])) >
>       snd(komplexitaet_R (Inverse (Compose [r3, r2, r1])))
>     = Inverse (Compose [r3, r2, r1])
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS

---

Der obigen  Ausdehnung auf die n-stellige Operation "compose" liegt folgende
Herleitung zugrunde:

```
      (r1;r2;r3;...;rn)~ = {(r1;r2;...;r(n-1));rn}~ (Assoziativit"at der
                                                     Relationenalgebra)
                         = rn~;(r1;r2;...;r(n-1))~ (Distributivit"at "uber `;'
                                                    und Umkehrung der Ordnung
                                                    der Relationenalgebra)
                         = rn~;{(r1;r2;...;r(n-2));r(n-1)}~
                         = rn~;r(n-1)~;(r1;r2;...;r(n-2))~
                         ...
                         = rn~;r(n-1)~;...;r1~
```

---

```
*---------------------------------------*
| Simplifizierung in umgekehrter Richtung |
|                                       |
```

```
|    (r;s)~ = s~ ; r~                |
*-------------------------------------*

> simple_r_term (Inverse (Compose [r, s]))
>             | komplexitaet_R (Inverse (Compose [r, s])) >
>               komplexitaet_R (Compose [Inverse s, Inverse r])
>               = Compose [Inverse s, Inverse r]



> simple_r_term r = r



***********************************************
\/                            \/
\/ E v a l u a t i o n s f u n k t i o n e n \/
\/                            \/
***********************************************

*****************************************************************
* T e i l  1 : Evaluationsfunktionen f"ur  K o n z e p t e *
*****************************************************************

> s_person = sort_nub person
> s_student = sort_nub student
> s_weiblich = sort_nub weiblich
> s_maennlich = sort_nub maennlich
> s_vegetarier = sort_nub vegetarier

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
-------------------------------------------------------------------------
Wegen der Existenz von lazy-evaluation in Gofer werden die sortierten Listen
von Konzepten (primitive Konzepte der Wissensbasis) in Variablen festgehalten.
-------------------------------------------------------------------------

> evaluiere_K :: Konzeptterm -> [String]

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
-------------------------------------------------------------------------
Mittels der Funktion `evaluiere_K' kann die Instanzenmenge eines Konzeptes
ermittelt werden. (Zu Aufbau und Funktionsweise von `evaluiere_K' siehe
Abschnitt 5.3 der Diplomarbeit)
-------------------------------------------------------------------------

> evaluiere_K (Person) = s_person
> evaluiere_K (Student) = s_student
> evaluiere_K (Weiblich) = s_weiblich
> evaluiere_K (Maennlich) = s_maennlich
> evaluiere_K (Vegetarier) = s_vegetarier
> evaluiere_K (Basisbottom) = bottom
> evaluiere_K (Menge a) = sort_nub a

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
-------------------------------------------------------------------------
Die Argumentliste `a' der Konstruktorfunktion `Menge' wird
- sortiert und
- gleichzeitig werden alle Duplikate aus der Liste entfernt.

(Definition und Kommentar zur Funktion `sort_nub' befinden sich im Definitionsteil
 der ,,Subfunktion `compose'``)
```

---

```
> evaluiere_K (And (a:as)) = und ((evaluiere_K a)
>                               : (evaluiere_K_z as)) where
>           evaluiere_K_z as = case as of
>               [] -> []
>               x:xs -> (evaluiere_K x):(evaluiere_K_z xs)
> evaluiere_K (Or (a:as)) = oder ((evaluiere_K a)
>                                : (evaluiere_K_z as)) where
>            evaluiere_K_z as = case as of
>               [] -> []
>               x:xs -> (evaluiere_K x): (evaluiere_K_z xs)
> evaluiere_K (Not a) = nicht (evaluiere_K a)
> evaluiere_K (Some b1 b2) = some (evaluiere_R b1) (evaluiere_K b2)
> evaluiere_K (All b1 b2) = f_all (evaluiere_R b1) (evaluiere_K b2)
```

**********
\/ Subfunktionen von `evaluiere_K' \/
**********

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Weitere Erl"auterungen zu den Subfunktionen k"onnen Abschnitt 5.3 der
Diplomarbeit entnommen werden.
---

*----------*
| Subfunktion `und' |
*----------*

```
> und :: [[String]] -> [String]

> und [a] = a
> und (a:b:as) = foldl merge_und (merge_und a b) as

> -- \/ Hilfsfunktion von `und' \/
> merge_und :: [String] -> [String] -> [String]
> merge_und [] ys = []
> merge_und xs [] = []
> merge_und (x:xs) (y:ys)
>           | x == y = x : merge_und xs ys
>           | x > y = merge_und (x:xs) ys
>           | otherwise = merge_und xs (y:ys)
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
- `und' bildet die  S c h n i t t m e n g e  der als Argumente "ubergebenen
  s o r t i e r t e n  Mengen.

- `merge_und xs ys' bildet die  S c h n i t t m e n g e  der beiden als
  Argumente "ubergebenen  s o r t i e r t e n  Listen.
---

*----------*
| Subfunktion `oder' |
*----------*

```
> oder :: [[String]] -> [String]

> oder [a] = a
> oder (a:b:as) = foldl merge_oder (merge_oder a b) as

> -- \/ Hilfsfunktion von `oder' \/
> merge_oder :: [String] -> [String] -> [String]
> merge_oder [] ys = ys
> merge_oder xs [] = xs
> merge_oder (x:xs) (y:ys)
>           | x == y = x: merge_oder xs ys
>           | x > y = y: merge_oder (x:xs) ys
>           | otherwise = x: merge_oder xs (y:ys)



HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
------------------------------------------------------------------------------
- `oder' bildet die  V e r e i n i g u n g s m e n g e  der als Argumente
   "ubergebenen s o r t i e r t e n  Mengen.

- `merge_oder xs ys' bildet die  V e r e i n i g u n g s m e n g e  der beiden
   als Argumente "ubergebenen  s o r t i e r t e n  Listen.
------------------------------------------------------------------------------



*--------------------*
| Subfunktion `some' |
*--------------------*

> some :: [(String, String)] -> [String] -> [String]

> some [] bs = []
> some as [] = []
> some (a:as) bs = nub_x ((merge_some a bs) ++ (some as bs))


> -- \/ Hilfsfunktionen von `some' \/
> merge_some :: (String, String) -> [String] -> [String]
> merge_some a [] = []
> merge_some (a, b) (c:cs)
>           | b == c = [a]
>           | b > c = merge_some (a, b) cs
>           | otherwise = []

> nub_x :: Eq a => [a] -> [a]
> nub_x [] = []
> nub_x (x : xs) = x: nub_x (filter_x (x /=) xs)

> -- \/ Hilfsfunktion von `nub_x' \/
> filter_x :: (a -> Bool) -> [a] -> [a]
> filter_x _ [] = []
> filter_x p (x : xs)
>       | p x = x : xs
>       | otherwise = xs'
>           where xs' = filter_x p xs



HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
------------------------------------------------------------------------------
```

- `merge_some' "uberpr"uft, ob die zweite Komponente des als erstes Argument
  "ubergebenen Tupels ein Element der als zweites Argument "ubergebenen
  sortierten Liste ist. Wird die zweite Komponente des Tupels mit einem Element
  der sortierten Liste verglichen, das gr"o"ser oder gleich ist, so wird
  die "Uberpr"ufung abgebrochen.

- `nub_x' entfernt alle Duplikate aus der als Argument "ubergebenen
  s o r t i e r t e n Liste.

- `filter_x p xs' liefert eine Subliste mit allen Elementen aus der sortierten
  Liste `xs', die das Pr"adikat `p' erf"ullen.
---

*----------*
| Subfunktion `f_all' |
*----------*

```
> f_all :: [(String, String)] -> [String] -> [String]

> f_all as [] = []
> f_all [] bs = []
> f_all as bs = differenz_ls (some as bs) (f_all_z as bs)

> -- \/ Hilfsfunktionen von `f_all' \/
> f_all_z :: [(String, String)] -> [String] -> [String]
> f_all_z [] bs = []
> f_all_z (a:as) bs = nub_x ((merge_all a bs) ++ (f_all_z as bs))

> -- \/ Hilfsfunktionen von `f_all_z' \/
> merge_all (a, c) [] = [a]
> merge_all (a, c) (b:bs)
>          | c == b = []
>          | c < b = [a]
>          | otherwise = merge_all (a, c) bs

> differenz_ls :: [String] -> [String] -> [String]
> differenz_ls [] bs = []
> differenz_ls as [] = as
> differenz_ls (a:as) (b:bs)
>            | a == b = differenz_ls as bs
>            | a > b = differenz_ls (a:as) (bs)
>            | otherwise = a:(differenz_ls as (b:bs))
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---

- `merge_all' "uberpr"uft, ob die zweite Komponente des als erstes Argument
  "ubergebenen Tupels nicht in der Menge enthalten ist, die die als zweites
  Argument "ubergebenen s o r t i e r t e Liste repr"asentiert.

- Die Aufgabe der Funktion `f_all_z' verdeutlicht folgendes Beispiel:
  `f_all_z hat_Kind student' liefert als Ergebnis die Menge aller Eltern, die
  mindestens ein Kind haben, das nicht Student ist.

- `differenz_ls' liefert die Differenzmenge der beiden als Argumente
  "ubergebenen s o r t i e r t e n Listen.
---

```
*---------------------*
| Subfunktion `nicht' |
*---------------------*

> nicht :: [String] -> [String]

> nicht as = differenz_ls s_person as


HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
------------------------------------------------------------------------------
- Erl"auterungen zur Funktion `differenz_ls' befinden sich im Teil
 ,,Hilfsfunktion von `f_all'``.
------------------------------------------------------------------------------


**********************************************************
* T e i l  2 : Evaluationsfunktionen f"ur  R o l l e n *
**********************************************************

> s_hat_Kind = sort_nub hat_Kind
> s_ist_verheiratet = sort_nub ist_verheiratet
> s_ist_Kusin = sort_nub ist_Kusin
> s_ist_Lehrer = sort_nub ist_Lehrer

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
------------------------------------------------------------------------------
Wegen der Existenz von lazy-evaluation in Gofer werden die sortierten Listen
von Rollen (primitive Rollen der Wissensbasis) in Variablen festgehalten.
------------------------------------------------------------------------------


> evaluiere_R :: Relationenterm -> [(String,String)]

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
------------------------------------------------------------------------------
Mittels der Funktion `evaluiere_R' kann die Instanzenmenge einer Rolle
ermittelt werden. (Zu Aufbau und Funktionsweise von `evaluiere_R' siehe
Abschnitt 5.3 der Diplomarbeit)
------------------------------------------------------------------------------

> evaluiere_R (Relationenbottom) = r_bottom
> evaluiere_R (HatKind) = s_hat_Kind
> evaluiere_R (IstKusin) = s_ist_Kusin
> evaluiere_R (IstVerheiratet) = s_ist_verheiratet
> evaluiere_R (IstLehrer) = s_ist_Lehrer
> evaluiere_R (Relation a) = sort_nub a

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
------------------------------------------------------------------------------
Die Argumentliste `a' der Konstruktorfunktion `Relation' wird
- sortiert und
- gleichzeitig werden alle Duplikate aus der Liste entfernt.

(Definition und Kommentar zur Funktion `sort_nub' befinden sich im Definitionsteil
 der ,,Subfunktion `compose'``)
------------------------------------------------------------------------------

> evaluiere_R (ROr (a:as)) = oder_r ((evaluiere_R a):(evaluiere_R_z as))
>             where evaluiere_R_z as = case as of
```

```
>             [] -> []
>             x:xs -> (evaluiere_R x):(evaluiere_R_z xs)
> evaluiere_R (RAnd (a:as)) = und_r ((evaluiere_R a): (evaluiere_R_z as))
>           where evaluiere_R_z as = case as of
>             [] -> []
>             x:xs -> (evaluiere_R x):(evaluiere_R_z xs)
> evaluiere_R (Inverse a) = inverse (evaluiere_R a)
> evaluiere_R (Compose (a:as))
>       = compose ((evaluiere_R a):(evaluiere_R_z as)) where
>         evaluiere_R_z as = case as of
>               [] -> []
>               x:xs -> (evaluiere_R x):(evaluiere_R_z xs)
> evaluiere_R (Restrict b1 b2)
>                   = range_restrict (evaluiere_R b1) (evaluiere_K b2)


************************************
\/ Subfunktionen von `evaluiere_R' \/
************************************

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
--------------------------------------------------------------------------------
Weitere Erl"auterungen zu den Subfunktionen k"onnen Abschnitt 5.3 der
Diplomarbeit entnommen werden.
--------------------------------------------------------------------------------

*--------------------*
| Subfunktion `und_r' |
*--------------------*

> und_r :: [[(String, String)]] -> [(String, String)]

> und_r [a] = a
> und_r (a:b:as) = foldl merge_r_und (merge_r_und a b) as

> -- \/ **** Hilfsfunktion von `und_r' **** \/
> merge_r_und :: [(String, String)] -> [(String, String)] -> [(String, String)]
> merge_r_und [] ys = []
> merge_r_und xs [] = []
> merge_r_und (x:xs) (y:ys)
>             | x == y = x: merge_r_und xs ys
>             | x > y = merge_r_und (x:xs) ys
>             | otherwise = merge_r_und xs (y:ys)


HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
--------------------------------------------------------------------------------
- `und_r as' bildet die  S c h n i t t m e n g e  aller  s o r t i e r t e n
  Relationen, die in der als Argument "ubergebenen Liste enthalten sind.
- `merge_r_und xs ys' bildet die  S c h n i t t m e n g e  der  b e i d e n  als
   Argumente "ubergebenen  s o r t i e r t e n  Relationen.
--------------------------------------------------------------------------------

*----------------------*
| Subfunktion `oder_r' |
*----------------------*

> oder_r :: [[(String, String)]] -> [(String, String)]

> oder_r [a] = a
> oder_r (a:b:as) = foldl merge_r_oder (merge_r_oder a b) as
```

```
> -- \/ **** Hilfsfunktion von `oder_r' **** \/

> merge_r_oder :: [(String, String)] -> [(String, String)] -> [(String, String)]
> merge_r_oder [] ys = ys
> merge_r_oder xs [] = xs
> merge_r_oder (x:xs) (y:ys)
>             | x == y = x: merge_r_oder xs ys
>             | x > y = y: merge_r_oder (x:xs) ys
>             | otherwise = x: merge_r_oder xs (y:ys)
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
- `oder_r as' bildet die V e r e i n i g u n g s m e n g e aller
  s o r t i e r t e n Relationen, die in der als Argument "ubergebenen Liste
  enthalten sind.
- `merge_r_oder xs ys' bildet die V e r e i n i g u n g s m e n g e der beiden
  als Argumente "ubergebenen s o r t i e r t e n Relationen.

---

```
*----------------------*
| Subfunktion `compose' |
*----------------------*

> compose :: [[(String, String)]] -> [(String, String)]

> compose [a] = a
> compose (a:b:as) = foldl m_composition (m_composition a b) as


> -- \/ **** Hilfsfunktion von `compose' **** \/
> m_composition :: [(String, String)] -> [(String, String)] -> [(String, String)]
> m_composition as bs = sort_nub[(x, w)|(x, y)<-as, (v, w)<-bs, y==v]


> -- \/ **** Hilfsfunktion von `m_composition' **** \/
> sort_nub :: Ord a => [a] -> [a]
> sort_nub = foldr insert_x []

> -- \/ **** Hilfsfunktion von `sort_nub' **** \/
> insert_x :: Ord a => a -> [a] -> [a]
> insert_x x [] = [x]
> insert_x x (y:ys)
>         | x == y = y:ys
>         | x < y = x:y:ys
>         | otherwise = y:insert_x x ys
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
- Die Aufgabe der Funktion `m_composition' kann an dem folgenden Beispiel
  abgelesen werden:
  `m_composition hat_Kind ist_verheiratet' liefert als Ergebnis die Relation,
  die die Beziehung zwischen Eltern und ihren Schwiegers"ohnen bzw. zwischen
  Eltern und ihren Schwiegert"ochtern beschreibt.
  (`m_composition' realisiert die Interpretation des terminologischen Operators
   `compose'.)

- `sort_nub' sortiert eine Liste und entfernt gleichzeitig alle Duplikate.

- `insert_x' f"ugt ein Element zu einer s o r t i e r t e n Liste hinzu, wobei
  die Entstehung von Duplikaten verhindert wird.
---

```
*------------------------*
| Subfunktion `inverse' |
*------------------------*

> inverse :: [(String, String)] -> [(String, String)]

> inverse as = sort ([(y,x)|(x,y)<-as])


*------------------------------*
| Subfunktion `range_restrict' |
*------------------------------*

> range_restrict :: [(String, String)] -> [String] -> [(String, String)]

> range_restrict [] bs = []
> range_restrict as [] = []
> range_restrict (a:as) bs = (merge_restrict a bs) ++ (range_restrict as bs)

> -- \/ **** Hilfsfunktion von `range_restrict' **** \/
> merge_restrict :: (String, String) -> [String] -> [(String, String)]

> merge_restrict a [] = []
> merge_restrict (a, b) (d:bs)
>           | b == d = [(a, b)]
>           | b < d = []
>           | b > d = merge_restrict (a, b) bs


*****************************************************************************
\/                                                  \/
\/  K o m p l e x i t "a t s a b s c h "a t z u n g s f u n k t i o n e n  \/
\/                                                  \/
*****************************************************************************

*********************************************************************
* T e i l  1 : Komplexit"atsabsch"atzungsfunktionen f"ur Konzepte *
*********************************************************************

> l_person = length_Float person
> l_student = length_Float student
> l_weiblich = length_Float weiblich
> l_maennlich = length_Float maennlich
> l_vegetarier = length_Float vegetarier

> length_Float :: [a] -> Float

> length_Float    = foldl' (\n _ -> n + 1.0) 0.0


HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
------------------------------------------------------------------------------
Wegen der Existenz von lazy-evaluation in Gofer werden die L"angen der
konzept-repr"asentierenden Listen (primitive Konzepte der Wissensbasis) in
```

Variablen festgehalten.

-`length_Float' berechnet die L"ange einer Liste und hat den Ergebnistyp `Float'.
---

> komplexitaet_K :: Konzeptterm -> (Float, Float)

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Die beiden Komponenten des Ergebnistupels der Komplexit"atsabsch"atzungsfunktion
`komplexitaet_K' haben folgende Bedeutung:

1. Komponente : Abgesch"atzte Gr"o"se der resultierenden Menge
2. Komponente : Abgesch"atzte Berechnungskomplexit"at (Anzahl der Reduktionsschritte) der resultierenden Menge

Weitere Erl"auterungen befinden sich in Abschnitt 5.4.3 der Diplomarbeit.
---

```
> komplexitaet_K (Basisbottom) = (0.0, 0.0)
> komplexitaet_K (Person) = (l_person, 0.0)
> komplexitaet_K (Student) = (l_student, 0.0)
> komplexitaet_K (Weiblich) = (l_weiblich, 0.0)
> komplexitaet_K (Maennlich) = (l_maennlich, 0.0)
> komplexitaet_K (Vegetarier) = (l_vegetarier, 0.0)
> komplexitaet_K (Menge as) = (v_lgth, (v_lgth * v_lgth - 1.0)/2.0)
>                 where v_lgth = length_Float as
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Matcht das Argument die Konstruktorfunktion `Menge', wird die im Rahmen des
Arguments "ubergebene Liste `as' bei der Evaluierung sortiert.

Die zweite Komponente des Ergebnistupels beinhaltet ein Sch"atzma"s f"ur den
Berechnungsaufwand, der zur Sortierung der Liste `as' erforderlich ist.
---

```
> komplexitaet_K (And (a:as)) = k_und ((komplexitaet_K a)
>           :(komplexitaet_K_z as)) where
>       komplexitaet_K_z as = case as of
>                 [] -> []
>                 x:xs -> (komplexitaet_K x):(komplexitaet_K_z xs)
>
> komplexitaet_K (Or (a:as)) = k_oder ((komplexitaet_K a)
>           :(komplexitaet_K_z as)) where
>       komplexitaet_K_z as = case as of
>                 [] -> []
>                 x:xs -> (komplexitaet_K x):(komplexitaet_K_z xs)
>
> komplexitaet_K (Not a) = k_nicht (komplexitaet_K a)
> komplexitaet_K (Some r a)
>               = k_some (komplexitaet_R r) (komplexitaet_K a)
> komplexitaet_K (All r a)
>               = k_all (komplexitaet_R r) (komplexitaet_K a)
```

**********
\/ Subfunktionen von `komplexitaet_K' \/
**********

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Die Bedeutung der Subfunktionen, sowie ihre Arbeitsweise und zugrundeliegende
Berechnungen k"onnen Abschnitt 5.4.3 der Diplomarbeit entnommen werden.
---

```
*--------------------*
| Subfunktion `k_und' |
*--------------------*

> k_und :: [(Float, Float)] -> (Float, Float)

> k_und [a] = a
> k_und [(l1, k1), (l2, k2)]
>     | l1 == 0.0 || l2 == 0.0  = (0.0, k1 + k2)
>     | otherwise = (var, k1 + k2
>               + (((2.0 * l1 + 2.0 * l2 - 2.0 + 2.0 * min)/3.0) * g_vgl_Konzept)
>                           + (var * g_add_Konzept))
>           where min =  minimum [l1, l2]
>                 var =  (minimum [l1, l2])/2.0

> k_und (a:b:as) = k_und (k_und [a, b]: as)

> g_vgl_Konzept = 8.8
> g_add_Konzept = 6.3
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Der obigen Funktionsdefinition liegen folgende Berechnungen und Konstanten
zugrunde:
- (2.0 * l1 + 2.0 * l2 - 2.0) : Vergleichsanzahl im worst-case
- min =  minimum [l1, l2]     : Vergleichsanzahl im best-case
- ((2.0 * l1 + 2.0 * l2 - 2.0
          + 2.0 * min)/3.0 : abgesch"atzte Anzahl der Vergleiche, die bei
                    einem Aufruf der Funktion `merge_und'
                    durchgef"uhrt werden.
- g_vgl_Konzept           : Absch"atzung f"ur die Anzahl der
                    Reduktionsschritte, die f"ur die Durchf"uhrung
                    eines Vergleichs erforderlich sind.
- g_add_Konzept           : Absch"atzung f"ur die Anzahl der
                    Reduktionsschritte, die f"ur die Hinzuf"ugung
                    eines Elements zu einer Liste mittels des
                    Operators (:) ben"otigt werden.
Die Konstanten `g_vgl_Konzept' und `g_add_Konzept' sind im Bezug auf meine
Wissensbasis ermittelt und m"ussen bei einer Ver"anderung der Wissensbasis
adaptiert werden.
---

```
*---------------------*
| Subfunktion `k_oder' |
*---------------------*

> k_oder :: [(Float, Float)] -> (Float, Float)

> k_oder [a] = a
> k_oder [(l1, k1), (l2, k2)]
>     | l1 == 0.0 = (l2, k1 + k2)
>     | l2 == 0.0 = (l1, k1 + k2)
>     | otherwise = (val2 + (val1/2.0), k1 + k2
>           + ((2.0 * (l1 + l2 -1.0)  + val1)/2.0) * g_vgl_Konzept
```

```
>                       + (val2 + (val1/2.0)) * g_add_Konzept)
>               where val1 = minimum [l1, l2]
>                     val2 = maximum [l1, l2]
> k_oder (a:b:as) = k_oder (k_oder [a, b]:as)
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Der obigen Funktionsdefinition liegen folgende Berechnungen zugrunde:
- (2.0 * (l1 + l2 -1.0))  : Vergleichsanzahl im worst-case
- minimum [l1, l2]        : Vergleichsanzahl im best-case
- ((2.0 * (l1 + l2 -1.0)
  + val1)/2.0)  : abgesch"atzte Anzahl der Vergleiche, die bei einem
  Aufruf der Funktion `merge_oder' durchgef"uhrt
  werden.

---

*----------*
| Subfunktion `k_nicht' |
*----------*

```
> k_nicht :: (Float, Float) -> (Float, Float)

> k_nicht (l1, k1) = ((l_person - l1), k1 + (l1 * g_glh_test)
>                           + ((l_person - l1) * g_uglh_test))

> g_glh_test = 3.92
> g_uglh_test = 14.0769
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Der obigen Funktionsdefinition liegen folgende Konstanten zugrunde:
- g_glh_test  : Absch"atzung der Anzahl der Reduktionsschritte, die zur
  Durchf"uhrung eines Gleichheitstests (`==') erforderlich sind.
- g_uglh_test : Absch"atzung der Anzahl der Reduktionsschritte, die f"ur
  eine einmalige Durchf"uhrung der Vergleichsoperation `>' oder
  `<' erforderlich sind.

Die Konstanten `g_glh_test' und `g_uglh_test' sind im Bezug auf meine
Wissensbasis ermittelt und m"ussen bei einer Ver"anderung der Wissensbasis
adaptiert werden.

---

*----------*
| Subfunktion `k_some' |
*----------*

```
> k_some :: (Float, Float) -> (Float, Float) -> (Float, Float)

> k_some (l1, k1) (l2, k2)
>       | l1 == 0.0 || l2 == 0.0 = (0.0, k1 + k2)
>       | otherwise = (var, k1 + k2 + (((l1 * l2 + l1)/2.0) * g_vgl_Konzept) + l1 * 2.0
>                   + (var * ((2.0 * (g_add_Konzept + 1.0)) + (g_glh_test + 1.0))))
>           where  var = (l1+l2)/4.0
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Der obigen Funktionsdefinition liegt folgende Berechnung zugrunde:

- (l1 * l2 + l1)/2.0 : abgesch"atzte Anzahl der Vergleiche, die bei
einem Aufruf der Funktion `merge_some'
durchgef"uhrt werden.
---

*----------*
| Subfunktion `k_all' |
*----------*

```
> k_all :: (Float, Float) -> (Float, Float) -> (Float, Float)

> k_all (l1, k1) (l2, k2)
>         | l1 == 0.0 || l2 == 0.0 = (0.0, k1 + k2)
>         | otherwise = (var2,
>                 k1 + k2 + (((var3 + l1)/2.0) * g_vgl_Konzept) + l1 * 3.0
>                     + (var1 * ((2.0 * (g_add_Konzept + 1.0)) + (g_glh_test + 1.0)))
>                     + ((2.0*var3 + l1)/2.0)*g_vgl_Konzept + var2*g_add_Konzept
>                     + l1*3.0
>                     + var2*(g_glh_test + 2.0)
>                     + (var1 + var2 - 1.0 + var2/2.0)*g_vgl_Konzept
>                     + var2*g_add_Konzept)
>             where  var1 = (l1+l2)/4.0
>                    var2 = var1/2.0
>                    var3 = l1 * l2
```

**********
* T e i l  2 : Komplexit"atsabsch"atzungsfunktionen f"ur Rollen *
**********

```
> l_hat_Kind = length_Float hat_Kind
> l_ist_verheiratet = length_Float ist_verheiratet
> l_ist_Kusin = length_Float ist_Kusin
> l_ist_Lehrer = length_Float ist_Lehrer
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Wegen der Existenz von lazy-evaluation in Gofer werden die L"angen der
rollen-repr"asentierenden Listen (primitive Rollen der Wissensbasis) in
Variablen festgehalten.
---

```
> komplexitaet_R :: Relationenterm -> (Float, Float)
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Die beiden Komponenten des Ergebnistupels der Funktion `komplexitaet_R' haben
folgende Bedeutung:
1. Komponente : Abgesch"atzte Gr"o"se der resultierenden Relation
2. Komponente : Abgesch"atzte Berechnungskomplexit"at (Anzahl der
Reduktionsschritte) der resultierenden Relation

Weitere Erl"auterungen befinden sich in Abschnitt 5.4.3 der Diplomarbeit.
---

```
> komplexitaet_R (Relationenbottom) = (0.0, 0.0)
> komplexitaet_R (HatKind) = (l_hat_Kind, 0.0)
> komplexitaet_R (IstKusin) = (l_ist_Kusin, 0.0)
> komplexitaet_R (IstVerheiratet) = (l_ist_verheiratet, 0.0)
```

```
> komplexitaet_R (IstLehrer) = (l_ist_Lehrer, 0.0)
> komplexitaet_R (Relation rs) = (v_lgth, (v_lgth * v_lgth - 1.0) / 2.0)
>                         where v_lgth = length_Float rs
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Matcht das Argument die Konstruktorfunktion `Relation', wird die im Rahmen des
Arguments "ubergebene Liste `rs' bei der Evaluierung sortiert.

Die zweite Komponente des Ergebnistupels beinhaltet ein Sch"atzma"s f"ur den
Berechnungsaufwand, der zur Sortierung der Liste `rs' erforderlich ist.
---

```
> komplexitaet_R (ROr (r:rs)) = k_r_oder ((komplexitaet_R r)
>                 : (komplexitaet_R_z rs))  where
>         komplexitaet_R_z rs = case rs of
>               [] -> []
>               x:xs -> (komplexitaet_R x):(komplexitaet_R_z xs)
> komplexitaet_R (RAnd (r:rs)) = k_r_und ((komplexitaet_R r)
>                 : (komplexitaet_R_z rs)) where
>         komplexitaet_R_z rs = case rs of
>               [] -> []
>               x:xs -> (komplexitaet_R x):(komplexitaet_R_z xs)
>
> komplexitaet_R (Inverse r) = k_converse (komplexitaet_R r)
>
> komplexitaet_R (Compose (r:rs))
>    = k_composition ((komplexitaet_R r):(komplexitaet_R_z rs)) where
>         komplexitaet_R_z rs = case rs of
>               [] -> []
>               x:xs -> (komplexitaet_R x):(komplexitaet_R_z xs)
>
> komplexitaet_R (Restrict r a)
>        = k_range_restrict (komplexitaet_R r) (komplexitaet_K a)
```

**********
\/ Subfunktionen von `komplexitaet_R' \/
**********

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Die Bedeutung der Subfunktionen sowie ihre Arbeitsweise und zugrundeliegende
Berechnungen k"onnen Abschnitt 5.4.3 der Diplomarbeit entnommen werden.
---

```
*--------------------*
| Subfunktion `k_r_und' |
*--------------------*

> k_r_und :: [(Float, Float)] -> (Float, Float)

> k_r_und [a] = a
> k_r_und [(l1, k1), (l2, k2)]
>     | l1 == 0.0 || l2 == 0.0  = (0.0, k1 + k2)
>     | otherwise = (var, k1 + k2
>               + (((2.0 * l1 + 2.0 * l2 - 2.0 + 2.0 * min)/3.0) * g_vgl_Rolle)
>                           + (var * g_add_Rolle))
>           where min = minimum [l1, l2]
>                 var = (minimum [l1, l2])/2.0
```

```
> k_r_und (a:b:as) = k_r_und (k_r_und [a, b]: as)
>
> g_vgl_Rolle = 21.21
> g_add_Rolle = 14.44
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Der obigen Funktionsdefinition liegen folgende Berechnungen und Konstanten
zugrunde:
- (2.0 * l1 + 2.0 * l2 - 2.0) : Vergleichsanzahl im worst-case
- min = minimum [l1, l2] : Vergleichsanzahl im best-case
- ((2.0 * l1 + 2.0 * l2 - 2.0
+ 2.0 * min)/3.0 : abgesch"atzte Anzahl der Vergleiche, die bei
einem Aufruf der Funktion `merge_r_und'
durchgef"uhrt werden.
- g_vgl_Rolle : Absch"atzung f"ur die Anzahl der
Reduktionsschritte, die f"ur die Durchf"uhrung
eines Vergleichs zweier Tupel ben"otigt werden.
- g_add_Rolle : Absch"atzung f"ur die Anzahl der
Reduktionsschritte, die f"ur die Hinzuf"ugung
eines Tupels zu einer Liste mittels des
Operators (:) ben"otigt werden.
Die Konstanten `g_vgl_Rolle' und `g_add_Rolle' sind im Bezug auf meine
Wissensbasis ermittelt und m"ussen bei einer Ver"anderung der Wissensbasis
adaptiert werden.
---

*----------*
| Subfunktion `k_r_oder' |
*----------*

```
> k_r_oder :: [(Float, Float)] -> (Float, Float)

> k_r_oder [a] = a
> k_r_oder [(l1, k1), (l2, k2)]
>       | l1 == 0.0 = (l2, k1 + k2)
>       | l2 == 0.0 = (l1, k1 + k2)
>       | otherwise = ((l1+l2)/2.0, k1 + k2
>                         + ((2.0 * (l1 + l2 -1.0)  + minimum [l1, l2])/2.0) * g_vgl_Rolle
>                         + ((l1+l2)/2.0) * g_add_Rolle)
> k_r_oder (a:b:as) = k_r_oder (k_r_oder [a, b]:as)
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Der obigen Funktionsdefinition liegen folgende Berechnungen zugrunde:
- (2.0 * (l1 + l2 -1.0)) : Vergleichsanzahl im worst-case
- minimum [l1, l2] : Vergleichsanzahl im best-case
- ((2.0 * (l1 + l2 -1.0)
+ minimum [l1, l2])/2.0) : abgesch"atzte Anzahl der Vergleiche, die bei einem
Aufruf der Funktion `merge_r_oder' durchgef"uhrt
werden.
---

*----------*
| Subfunktion `k_converse' |
*----------*

```
> k_converse :: (Float, Float) -> (Float, Float)

> k_converse (l1, k1) = (l1, k1 + (l1 * g_l_comp) + (((l1*l1 + l1 -2.0)/4.0) * g_v_sort))

> g_l_comp = 17.84
> g_v_sort = 15.6407
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS

---

Der obigen Funktionsdefinition liegen folgende Berechnungen und Konstanten zugrunde:

- (l1 * g_l_comp) : Absch"atzung f"ur die Komplexit"at der Berechnung der ,,list-comprehension`` [(y, x)|(x, y) <- as].
- ((l1*l1 + l1 -2.0)/4.0) : abgesch"atzte Anzahl der Vergleiche, die bei einem Aufruf der Funktion `sort' durchgef"uhrt werden.
- g_v_sort : Absch"atzung f"ur die Anzahl der Reduktionsschritte, die f"ur eine einmalige Durchf"uhrung der Vergleichsoperation `<' oder `==' erforderlich sind. (siehe Definition der Funktion `sort' im Standard-prelude von Gofer)

Die Konstanten `g_l_comp' und `g_v_sort' sind im Bezug auf meine Wissensbasis ermittelt und m"ussen bei einer Ver"anderung der Wissensbasis adaptiert werden.

---

```
*----------------------------*
| Subfunktion `k_composition' |
*----------------------------*

> k_composition :: [(Float, Float)] -> (Float, Float)

> k_composition [a] = a
> k_composition [(l1, k1), (l2, k2)]
>         | l1 == 0.0 || l2 == 0.0 = (0.0, k1 + k2)
>         | otherwise = (v2, k1 + k2 + (l1 * l2 * g_glh_test)
>                                    + v1
>                                    + v2 * 2.0 * g_v_sort
>                                    + v1)
>                                      where v1 = v2 * g_add_Rolle
>                                            v2 = (l1 + l2) / 4.5
>
> k_composition (a:b:as) = k_composition (k_composition [a, b]:as)


*-------------------------------*
| Subfunktion `k_range_restrict' |
*-------------------------------*

> k_range_restrict :: (Float, Float) -> (Float, Float) -> (Float, Float)

> k_range_restrict (l1, k1) (l2, k2)
>               | l1 == 0.0 || l2 == 0.0 = (0.0, k1 + k2)
>               | otherwise = ((l1 + l2) / 5.8,  k1 + k2
>                                  + ((l1 * l2 + l1/2.0) * g_vgl_Konzept)
>                                        + l1 * 3.0
>                                  + ((l1 + l2) / 5.8) * g_add_Rolle)
```

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS

---
Der obigen Funktionsdefinition liegen folgende Berechnungen zugrunde:
- 2.0 * l1 * l2 : Vergleichsanzahl im worst-case
- l1 : Vergleichsanzahl im best-case
- (l1 * l2 + l1/2.0) : abgesch"atzte Anzahl der Vergleiche, die bei einem Aufruf der Funktion `range_restrict' durchgef"uhrt werden.

---

**********
\/ \/
\/ A n f r a g e f u n k t i o n e n \/
\/ \/
**********

> finde_alle :: Konzeptterm -> [String]

> finde_alle a = evaluiere_K a

> finde_Relation :: Relationenterm -> [(String,String)]

> finde_Relation a = evaluiere_R a

HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS HINWEIS
---
Die beiden Anfragefunktionen `finde_alle' und `finde_Relation' dienen dazu, die Instanzenmenge des als Argument "ubergebenen Konzepts bzw. der als Argument "ubergebenen Rolle zu ermitteln.

---