**Christian-Albrechts-Universität zu Kiel**
Institut für Informatik
Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion

M.Sc. Finn Teegen, Jan Tikovsky, Prof. Dr. Michael Hanus

Institut für Informatik

# 1. Übung zur Vorlesung „Praktikum zu Fortgeschrittene Programmierung“

Prolog-Umgebung

Abgabe am Freitag, 09. März 2018 - 08:00

---

In dieser Praktikumsaufgabe soll unter Verwendung von Haskell als Programmiersprache ein Interpreter sowie eine interaktive Umgebung für eine vereinfachte Variante von Prolog programmiert werden.

### Unterstützter Sprachumfang

Die im Rahmen dieser Aufgabe zu implementierende Variante von Prolog unterscheidet sich in den folgenden Punkten von der aus der Vorlesung bekannten.

- Der Cut-Operator (`!`) wird nicht unterstützt und kann somit nicht verwendet werden.
- Es gibt (zunächst) keine vordefinierten Prädikate, sodass bspw. Dinge wie die Gleichheitsrelation (`=/2`) selbst definiert werden müssen.
- Prädikate können nicht infix notiert werden. Statt `X = Y` muss man also `=(X,Y)` schreiben.
- Die Disjunktion kann nur über mehrere Regeln, nicht aber über das Semikolon (`;`) ausgedrückt werden.
- Es werden keine Module unterstützt.

### Hinweise zur Bearbeitung

- Modularisieren Sie Ihr Programm auf sinnvolle Weise und halten Sie sich an den Haskell Style Guide.
- Dokumentieren Sie Ihr Programm ausführlich mit Typsignaturen und Kommentaren.
- Die Zusatzaufgaben müssen zum Bestehen des Praktikums nicht bearbeitet werden.
- Fragen Sie bei Unklarheiten rechtzeitig nach.

### Aufgabe 1 - Pretty Printing

0 Punkte

Das vorgegebene Modul Type.hs enthält Datentypdefinitionen zur Repräsentation von Variablen, Termen, Regeln, Programmen sowie Anfragen.

Machen Sie sich mit den Datentypen vertraut und definieren Sie eine Typklasse `Pretty`, die eine Methode `pretty :: a -> String` enthalten soll, um Daten lesbar, also in diesem Fall jeweils in gültiger Prolog-Syntax, darzustellen. Geben Sie dann eine Instanz für den vordefinierten Datentyp `Term` an. Dabei sollen Listenterme wie üblich dargestellt werden, was durch das folgende Beispiel verdeutlicht wird.

```
ghci> pretty (Comb "append" [ Var 0
                            , Comb "." [ Var 1
                                       , Var 2
                                       ]
                            , Comb "." [ Comb "1" []
                                       , Comb "." [ Comb "2" []
                                                  , Comb "[]" []
                                                  ]
                                       ]
                            ])
"append(A, [B|C], [1, 2])"
```

*Hinweis:* Sie können davon ausgehen, dass die Variablenindizes stets aus den natürlichen Zahlen einschließlich der Null sind. Sie müssen also keine negativen Variablenindizes berücksichtigen.

### Aufgabe 2 - Substitutionen

0 Punkte

Im nächsten Schritt sollen Substitutionen, d.h. Zuordnungen von Variablen zu Termen, realisiert werden.

1. Definieren Sie zunächst einen Datentyp `Subst` zur Repräsentation von Substitutionen.
2. Definieren Sie dann zwei Funktionen `empty :: Subst` und `single :: VarIndex -> Term -> Subst` zum Erstellen einer leeren Substitution bzw. einer Substitution, die lediglich eine einzelne Variable auf einen Term abbildet.
3. Implementieren Sie eine Funktion `apply :: Subst -> Term -> Term`, die eine Substitution auf einen Term anwendet.
4. Implementieren Sie außerdem eine Funktion `compose :: Subst -> Subst -> Subst`, die zwei Substitionen miteinander komponiert. Achten Sie darauf, dass für alle Terme `t` und Substitutionen `s1` und `s2` stets die Eigenschaft `apply (compose s2 s1) t == apply s2 (apply s1 t)` erfüllt ist.
5. Geben Sie schließlich für den Datentyp `Subst` eine Instanz der Typklasse `Pretty` an, um Substitutionen lesbar darzustellen. Solch eine lesbare Darstellung von Substitutionen kann bspw. wie folgt aussehen.

   ```
   ghci> pretty (compose (single 1 (Var 2))
                         (single 0 (Comb "f" [Var 1, Comb "true" []])))
   "{A -> f(C, true), B -> C}"
   ```

### Aufgabe 3 - Unifikation

0 Punkte

Aufbauend auf Substitutionen soll nun die zentrale Operation der Logikprogrammierung realisiert werden: die Unifikation.

1. Definieren Sie als erstes eine Funktion `ds :: Term -> Term -> Maybe (Term, Term)`, die die Unstimmigkeitsmenge zweier Terme berechnet und sie als Paar zurückgibt. Sollte die Unstimmigkeitsmenge leer sein, soll die Funktion stattdessen `Nothing` zurückgeben.

2. Definieren Sie anschließend eine Funktion `unify :: Term -> Term -> Maybe Subst`, die aufbauend auf der Funktion `ds` den allgemeinsten Unifikator für zwei Terme bestimmt, sofern die beiden Terme unifizierbar sind. Ansonsten soll die Funktion `Nothing` zurückgeben.

### Aufgabe 4 - SLD-Bäume

0 Punkte

Zur Auswertung von Anfragen wird eine geeignete Repräsentation von SLD-Bäumen benötigt.

1. Definieren Sie einen Datentyp `SLDTree`, der SLD-Bäume repräsentiert. Dabei sollen die Kanten der SLD-Bäume mit dem jeweiligen Unifikator beschriftet sein, der in dem entsprechenden Resolutionsschritt berechnet wurde.
2. Definieren Sie im Anschluss eine Funktion `sld :: Prog -> Goal -> SLDTree`, die den SLD-Baum zu einem Programm und einer Anfrage konstruiert.

   *Hinweis*: Verwenden Sie bei der Konstruktion des SLD-Baums die Selektionsstrategie FIRST, d.h. wählen Sie immer das am weitesten links stehende Literal zum Beweisen aus.

### Aufgabe 5 - Suchstrategien

0 Punkte

Ein SLD-Baum kann dann auf verschiedene Weisen durchlaufen werden, um alle Lösungen für eine Anfrage zu finden. Eine Lösung ist durch die Belegung der freien Variablen der ursprünglichen Anfrage gegeben. Suchstrategien können mithilfe des Typs

```
type Strategy = SLDTree -> [Subst]
```

ausgedrückt werden.

1. Definieren Sie eine Funktion `dfs :: Strategy`, die einen SLD-Baum mittels Tiefensuche durchläuft und dabei alle Lösungen zurückgibt.
2. Definieren Sie außerdem eine Funktion `bfs :: Strategy`, die einen SLD-Baum mittels Breitensuche durchläuft und dabei ebenfalls alle Lösungen zurückgibt.
3. Implementieren Sie schließlich eine Funktion `solve :: Strategy -> Prog -> Goal -> [Subst]`, die unter Verwendung einer gegebenen Suchstrategie zu einem Programm und einer Anfrage alle Lösungen berechnet und zurückgibt.

### Aufgabe 6 - Interaktive Umgebung

0 Punkte

Abschließend soll nun die interaktive Umgebung implementiert werden. Beachten Sie bei Ihrer Umsetzung die folgenden Punkte.

- Die interaktive Umgebung soll solange aktiv sein, bis Sie explizit vom Benutzer beendet wird. Die Beendigung kann bspw. durch die Eingabe eines speziellen Kommandos erfolgen.
- Es soll ein Programm geladen werden können. Diese Programm soll solange geladen bleiben, bis der Benutzer ein neues lädt.
- Der Benutzer kann Anfragen eingeben, deren Lösungen nacheinander angefordert werden können. Insbesondere soll es auch bei unendlich vielen Lösungen möglich sein, sich so viele Lösungen wie gewünscht einzeln anzeigen zu lassen. Die Antwortsubstitutionen sollen sich auf Belegungen für die freien Variablen der Anfrage beschränken, wobei anonyme Variablen (`_`) unterdrückt werden sollen.
- Der Benutzer soll zwischen Tiefen- und Breitensuche für die Auswertung wählen können. Die gewählte Strategie soll für alle Anfragen beibehalten werden, bis eine andere ausgewählt wird.
- Es soll eine Hilfe angezeigt werden können, die alle Funktionen Ihrer interaktiven Umgebung auflistet.
- Es soll weiterhin möglich sein, sich alle verfügbaren Prädikate in alphabetischer Reihenfolge mit ihrer Stelligkeit anzeigen zu lassen.

- Die interaktive Umgebung soll stets sinnvolle Rückmeldungen ausgeben und robust gegenüber Eingabefehlern sein.

Nachfolgend ist ein beispielhafter Ablauf für eine Sitzung in der interaktiven Umgebung gegeben, an dem Sie sich für Ihre Implementierung orientieren können.

```
Welcome to Simple Prolog!
Type ":help" for help.
?- :help
Commands available from the prompt:
  <goal>        Solves/proves the specified goal.
  :help         Shows this help message.
  :info         Shows all available predicates.
  :load <file>  Loads the specified file.
  :quit         Exits the interactive environment.
  :set <strat>  Sets the specified search strategy
                where <strat> is one of 'dfs' or 'bfs'.
?- :load examples/append.pl
Loaded.
?- :info
Available predicates:
append/3
?- :set bfs
Strategy set to breadth-first search.
?- append(A, [B|_], [1, 2]).
{A -> [], B -> 1} n
{A -> [1], B -> 2} ;
No more solutions.
?- :quit
Halt!
```

*Hinweis:* Zum Einlesen von Programmen und Anfragen wird das Modul Parser.hs bereitgestellt, das eine Typklasse `Parse` mit der Methode `parse :: String -> Either String a` sowie Instanzen für die vordefinierten Datentypen `Prog` und `Goal` enthält. Der Rückgabewert dieser Methode ist entweder eine Fehlermeldung oder der erfolgreich gelesene Wert. Weiterhin gibt es in dem Modul eine Funktion `parseFile :: Parse a => FilePath -> IO (Either String a)`, mithilfe derer der Inhalt einer Datei eingelesen werden kann. Falls die Datei nicht gelesen werden konnte, wird ebenfalls eine entsprechende Fehlermeldung zurückgegeben.

### Aufgabe 7 - Verbessertes Pretty Printing

0 Punkte

Es kann u.U. vorkommen, dass sich die Variablennamen der Antwortsubstitutionen bei der Ausgabe von denen, die in der ursprünglichen Anfrage verwendet wurden, unterscheiden. Diese Tatsache ist darauf zurückzuführen, dass die Zuordnung von Variablenindizes und -namen beim Parsen verloren geht und bei der Ausgabe somit nicht mehr bekannt ist.

Damit die Zuordnung erhalten bleiben kann, enthält die vorgegebene Typklasse `Parse` neben der Methode `parse :: String -> Either String a` noch eine weitere Methode, die zusätzlich zu dem eingelesenen Wert jene Zuordnung zurückgibt: `parseWithVars :: String -> Either String (a, [(VarIndex, String)])`.

1. Erweitern Sie die in der ersten Aufgabe eingeführte Typklasse `Pretty` um eine zusätzliche Methode `prettyWithVars :: [(VarIndex, String)] -> a -> String`, die solch eine Zuordnung von Variablenindizes und -namen bei der Generierung der lesbaren Darstellung berücksichtigt, und führen Sie die zuvor definierte Methode `pretty` auf die neue Methode `prettyWithVars` zurück. Nutzen Sie hierfür

eine vordefinierte Definition innerhalb der Klasse. Passen Sie auch alle bisher angegebenen Instanzen entsprechend an.

2. Modifizieren Sie die interaktive Umgebung schließlich so, dass bei der Ausgabe der Antwortsubstitutionen die Variablennamen aus der ursprünglichen Anfrage verwendet werden.

### Zusatzaufgabe 8 - Prädikate höherer Ordnung

0 Punkte

Im Rahmen der Vorlesung wurden u.a. Prädikate höherer Ordnung behandelt: Mittels der Familie vordefinierter `call`-Prädikate können Prädikate oder Prädikataufrufe als Argument an andere Prädikate übergeben und dann als Anfrage interpretiert werden. Erweitern Sie Ihre Implementierung entsprechend um die Unterstützung für Prädikate höherer Ordnung.

### Zusatzaufgabe 9 - Arithmetik

0 Punkte

Bisher fehlt die Unterstützung für das Rechnen mit arithmetischen Ausdrücken, wie es in anderen Prolog-Implementierungen möglich ist. Diese Unterstützung sollen Sie nun ergänzen.

1. Implementieren Sie eine Funktion `eval :: Term -> Maybe Int`, die den Wert eines arithmetischen Ausdrucks berechnet. Falls der arithmetische Ausdruck nicht wohlgeformt ist oder noch Variablen enthalten sollte, soll die Funktion `Nothing` zurückgeben.

   *Hinweis:* Für diese Aufgabenstellung sollen ausschließlich arithmetische Ausdrücke betrachtet werden, die aus ganzen Zahlen sowie den binären Funktoren `+`, `-`, `*`, `div` sowie `mod` aufgebaut sind. Ihr Interpreter muss also keine Gleitkommazahlen unterstützen.

2. Erweitern Sie den Interpreter auf Basis der Funktion `eval` um das vordefinierte Prädikat `is`.

3. Erweitern Sie den Interpreter auf Basis der Funktion `eval` zudem um die vordefinierten Vergleichsprädikate `=:=`, `=\=`, `<`, `>`, `>=` sowie `=<`.

### Zusatzaufgabe 10 - Negation

0 Punkte

Aus der Vorlesung ist ebenfalls die Negation als Fehlschlag bekannt, die in dieser Aufgabe implementiert werden soll. Erweitern Sie Ihren Interpreter zu diesem Zweck um die vordefinierten, einstelligen Prädikate `\+` und `not`. Berücksichtigen Sie dabei auch die verschiedenen Suchstrategien.

*Hinweis:* Die Berücksichtigung verschiedener Suchstrategien macht es ggf. erforderlich, die zuvor definierte Funktion `sld` um einen entsprechenden Parameter zu erweitern.

### Zusatzaufgabe 11 - Kapselung

0 Punkte

Zu guter Letzt sollen Sie Ihre Implementierung um die Unterstützung für die Kapselung von Nichtdeterminismus erweitern. Ergänzen Sie Ihren Interpreter hierfür um das vordefinierte, dreistellige Prädikat `findall` und stellen Sie alle Aspekte seines Originalverhaltens nach. Berücksichtigen Sie dabei auch die verschiedenen Suchstrategien.

*Hinweis:* Untersuchen Sie insbesondere, wie sich das `findall`-Prädikat anderer Prolog-Implementierungen verhält, wenn das erste Argument freie, nicht im zweiten Argument vorkommende Variablen enthält. Die Berücksichtigung verschiedener Suchstrategien macht es ggf. erforderlich, die zuvor definierte Funktion `sld` um einen entsprechenden Parameter zu erweitern.