aus

Oliver J. Bott, Peter Fricke, Uta Priss, Michael Striewe (Hrsg.)

Automatisierte Bewertung in der Programmierausbildung

Digitale Medien in der Hochschullehre Band 6
2017, 420 Seiten, br., 42,90 €, ISBN 978-3-8309-3606-0

Waxmann Verlag GmbH
www.waxmann.com info@waxmann.com

14 Der Grader VEA

Helmar Gust

Zusammenfassung
Vips ist ein virtuelles Prüfungssystem und dient zur Verwaltung, Durchführung und Auswertung von Online-Übungen und Online-Klausuren. Der Schwerpunkt der Vips-Entwicklung liegt auf der automatischen Bewertung von Aufgaben, um demjenigen, der die Aufgaben stellt und beurteilen muss, möglichst viel Arbeit abzunehmen. Hierfür wurde das Gradermodul VEA entwickelt, das Lösungen von Programmieraufgaben in Prolog, Lisp und ähnlichen Programmiersprachen bewerten kann [GW13]. Vips ermöglicht es, neben dem Prüfungsmodus Übungen und Klausuren für die Studierenden in einem Selbsttestmodus anzubieten. Das System unterstützt verschiedene Möglichkeiten zur Entwicklung, Pflege und Auswertung virtueller Aufgabenblätter und Prüfungen sowie die Verwaltung des gesamten Übungsbetriebs eines Kurses.

14.1 Einleitung

Vips wurde entwickelt, um einfach online Übungen und Klausuren durchführen zu können und die Lehrenden bei einer Reihe organisatorischer Aufgaben zu entlasten. Neben der Erstellung und Verwaltung von Übungsblättern und Klausuren umfasst Vips eine Arbeitsgruppenverwaltung, Punkte- und Notenübersichten für einzelne Teilnehmer und Arbeitsgruppen, sowie eine flexible Notenberechnung. Die Korrektur und Bewertung von Übungsaufgaben im Rahmen von Lehrveranstaltungen erfordert von den Lehrenden einen nicht unerheblichen Aufwand an Zeit. Daher haben sich in vielen Bereichen stark schematisierte Aufgabentypen durchgesetzt. Typische Beispiele dafür sind Multiple- und Single-Choice-Aufgaben, Ja/Nein-Fragen sowie Zuordnungsaufgaben. Auch für solche Aufgaben ist die händische Korrektur zeitaufwändig und lästig. Allerdings lässt sich die Auswertung dieser Aufgaben sehr leicht automatisieren. Vips führt für diese Aufgaben standardmäßig eine komplett automatische Bewertung durch. Allerdings

kann das Ergebnis auch bei diesen Aufgaben von Lehrenden nachträglich überschrieben werden.

Immer dann, wenn freie Texteingaben erwartet werden (und dies gilt bereits bei Lückentextaufgaben), ist für die Beurteilung aber ein gewisses Verständnis der Aufgabe notwendig, zumindest dann, wenn von der vorgegebenen Musterlösung abgewichen wird. Aber auch bei diesen Aufgaben lässt sich die Korrektur und Bewertung von automatischen Systemen zumindest unterstützen. So hilft bereits das Herausfiltern klarer Standardfälle, einen erheblichen Teil des Zeitaufwandes einzusparen, etwa wenn auf der einen Seite Musterlösungen erkannt und auf der anderen Seite eindeutige Fälle der Nichtbeantwortung herausgefiltert werden können. Programmieraufgaben nehmen eine Sonderstellung ein. Zum einen handelt es sich um Aufgaben, die sich weitgehend wie Freitextaufgaben verhalten. Zum andern gibt es aber starke formale Restriktionen an die eingegebenen Texte: Sie müssen als Programmcode fehlerfrei kompilierbar sein und sie müssen das geforderte Programmverhalten produzieren. Die Korrektur und Bewertung von Programmieraufgaben ist normalerweise noch erheblich aufwändiger als bei Freitextaufgaben, insbesondere dann, wenn sie Fehler enthalten. Eine Unterstützung bei diesen Aufgaben ist also sehr hilfreich, aber wegen der Komplexität solcher Aufgaben auch entsprechend schwierig. Genau hier liegt ein Schwerpunkt der Vips-Entwicklung: die automatische Auswertung von Programmieraufgaben, mit dem Ziel, demjenigen, der die Aufgaben stellt und beurteilen muss, möglichst viel Arbeit abzunehmen. Darüber hinaus ermöglicht Vips Übungen und Klausuren für die Studierenden in einem Selbsttestmodus anzubieten.

Vips stellt für Programmieraufgaben eine Runtime-Umgebung mit einer einfachen GUI zur Verfügung. Dies ermöglicht den Kursteilnehmern, die Aufgaben ohne lokale Installationen der Programmiersprachen zu lösen. Allerdings stehen interaktive Debug-Möglichkeiten in einer solchen Umgebung nicht zur Verfügung. Es kann daher auf eine lokale Installation der Programmiersprachen vor allem bei komplexen Aufgaben nicht immer verzichtet werden. Um lokale Installationen zum Aufgabenlösen benutzen zu können, gibt es Up- und Download-Möglichkeiten für die Aufgaben. Die Runtime-Umgebung steht auch für die Aufgabenkorrektur zur Verfügung. Zusammen mit einem vorgegebenen Default-Aufruf der Hauptfunktion eines Programms ermöglicht dies sowohl dem Kursteilnehmer als auch dem Korrekteur sich mit einem Mausklick einen ersten Überblick über die Lauffähigkeit der Lösung zu verschaffen.

Zur Unterstützung der Bewertung dieser Programmieraufgaben gibt es ein eigenes Servermodul VEA (Vips-Evaluation-Assistent), das auf einem vom StudIP/Vips-System getrennten Rechner läuft. Die Kommunikation zwischen StudIP/Vips und dem Servermodul VEA geschieht über HTTP. Mit einem Browser kann diese

Schnittstelle auch direkt benutzt werden[1], um VEA zu testen und zu konfigurieren. Vips entstand aus Ansätzen, die im Vorfeld und im Rahmen eines Projektes am Institut für Kognitionswissenschaft entwickelt wurden [Pey+00]. Vips ist als Plugin für Stud.IP[2] konzipiert und daher zurzeit nur in Kombination mit Stud.IP verfügbar. Einen Überblick über die Architektur gibt Abbildung 14.1.

14.2 Der Vips-Evaluations-Assistent VEA

Zur Unterstützung der Auswertung von Programmieraufgaben gibt es ein Bewertungsmodul (Grader Module) VEA (Vips-Evaluation-Assistent), das als Evaluationsserver auf einem vom StudIP-System getrennten Rechner läuft. Sinnvoll ist hier ein Rechner, der nur für diese Aufgabe zur Verfügung steht, da die Ausführung von fremdem Programmcode immer auch mit Sicherheitsrisiken verbunden ist. VEA stellt Runtime-Umgebungen für Programmieraufgaben in ausgewählten Programmiersprachen zur Verfügung. Diese Umgebungen können zur Entwicklung der Lösungen und zur Bewertung dieser Lösungen benutzt werden. Vips/VEA wurde ursprünglich für Scriptsprachen ausgelegt, bei denen nach dem Laden von Runtime-Umgebung und Programm eine (interaktive) Oberfläche zur Verfügung steht, in der einzelne Codesegmente oder Prozeduren und Funktionen

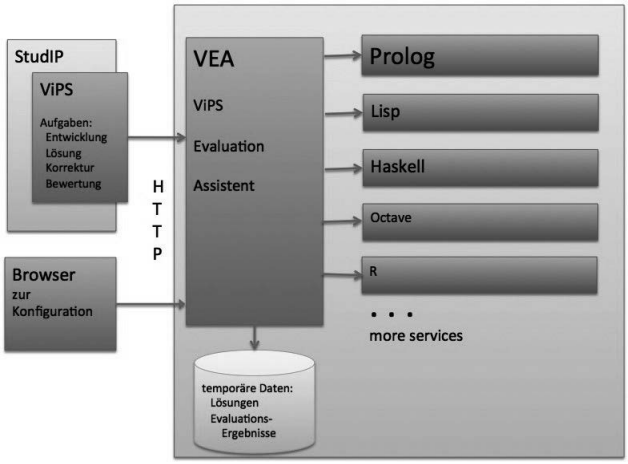

Abbildung 14.1: ViPS/VEA-Architektur

1 https://mvc.ikw.uni-osnabrueck.de/vips/vips.php
2 www.studip.de

ausgeführt werden können, ohne dass ein expliziter Compilationsschritt des gesamten Programms notwenig ist. Es läßt sich aber auch auf Compilersprachen erweitern. Nicht vorgesehen sind allerdings komplexe Programmstrukturen aus vielen interagierenden Modulen. Im Gegensatz zu klassischen Unit-Tests basieren die Tests auf Musterlösungen, die die korrekte Input/Output-Relation für die zu testende Funktion spezifizieren:

- Ein Generator erzeugt Input-Parameter für die zu testende Funktion.
- Die Musterlösung berechnet die zugehörigen Ausgabewerte.
- Diese Ausgabewerte werden mit den Ausgabewerten der abgegebenen Lösung verglichen.

Da die Kommunikation zwischen StudIP/Vips und VEA über HTTP geschieht, kann diese Schnittstelle auch mit einem normalen Browser direkt benutzt werden. Einen Eindruck von der Browseroberfläche zum Testen und Konfigurieren von VEA zeigt Abbildung 14.2.

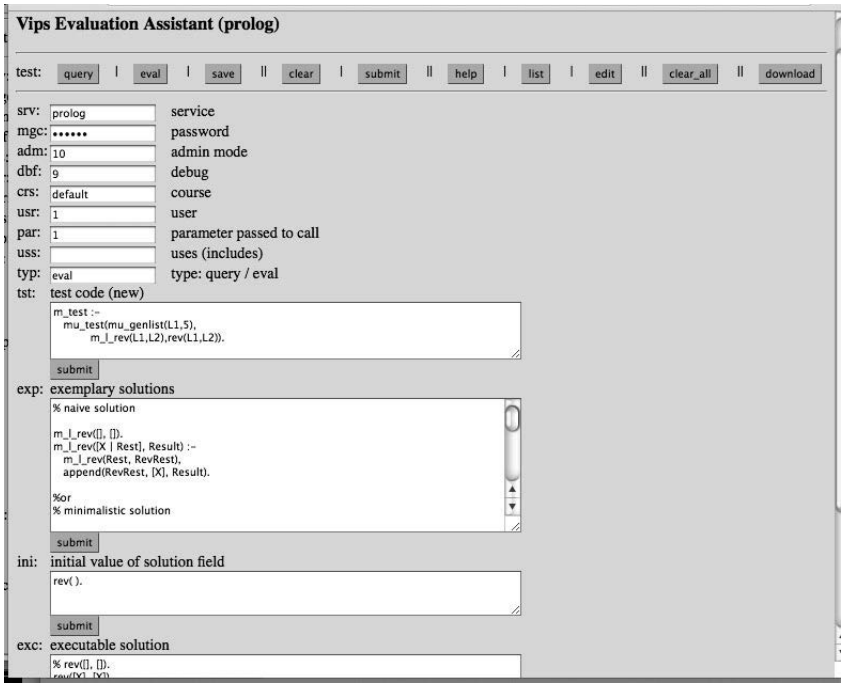

Abbildung 14.2: Browseroberfläche VEA

Dieses Interface gestattet neben der Konfiguration auch den Test aller Funktionen, sowie die Erweiterung des Systems um weitere Programmiersprachen.

14.2.1 Kommunikation mit Vips

Vips stellt für eine Programmierübungsaufgabe neben der Aufgabenstellung eine Reihe von Informationsfeldern zur Verfügung: ein Textfeld für den zu entwickelnden Code, Musterlösungen und den Aufruf der Hauptfunktion. Diese Felder werden zusammen mit einem Auswertungs-Modus an den VEA-Server übertragen.

Zurzeit gibt es zwei Modi für die Kommunikation zwischen Vips und VEA:

1. Alle Datenfelder werden als eigene POST-Felder übertragen (wird zurzeit von Vips benutzt).
2. Die Aufgabenspezifikation wird als XML-Strutur übertragen und nur die abgegebene Lösung wird als eigenes Feld übertragen. Die XML-Struktur entspricht dem Aufgabenaustauschformat, das im Rahmen von eCULT[3] entwickelt wurde [Str+15].

Die wichtigsten Felder, die VEA erwartet, sind in Tabelle 14.1 aufgeführt. Im Fall, dass eine XML-Struktur übertragen wird, werden diese Felder teilweise aus der XML-Struktur gefüllt.

Feldname	Beschreibung	Werte
adm	Administrationsebenen	0|1|2|3|4|5|6
typ	Auswertungsmodus	query | eval
mgc	Password	
srv	Service	Prolog | Lisp ....
qry	Query / Call	
exp	Musterlösungen	
ini	Lösungs-Template	
exc	ausführbare Lösung	
xml	XML-Spezifikation einer Aufgabe	

Tabelle 14.1: Liste der wichtigsten Felder

14.2.2 Implementierte Funktionen

Die Browserschnittstelle erlaubt neben der Konfiguration und der Ausführung von Programmen eine Reihe von Funktionen: *list*, *hash*, *query* und *eval*.

3 http://www.ecult-niedersachsen.de/

list Es wird eine Liste der verfügbaren Services zurückgegeben. Zurzeit ist nur der Service Prolog und Lisp komplett realisiert. Haskel, Oktave und R sind nur rudimentär vorhanden.

hash Es werden vier MD5-Hash-Werte zurückgegeben, die den vier Codenormalisierungen (s. u.) entsprechen. Diese Werte können in Vips benutzt werden, um Ähnlichkeiten zwischen den Lösungen unterschiedlicher Teilnehmer zu erkennen. So kann z. B. festgestellt werden, ob eine Lösung schon als Lösung eines anderen Teilnehmers korrigiert wurde.

query Der Auswertungsmodus $query$ wertet das qry-Feld relativ zum Inhalt des Lösungsfeldes aus. In Prolog enthält qry ein $Ziel$ ($Goal$), d. h. einen logischen Ausdruck, der bewiesen werden soll. Bei funktionalen Sprachen wie Lisp enthält qry einen Funktionsaufruf. Der Ergebnistext (z. B. das Compiler-Protokoll und das Ergebnis der Ausführung, falls das Programm syntaktisch fehlerfrei war) wird an Vips übertragen und ausgegeben. Im Falle einer Fehlermeldung oder einer Warnung wird zusätzlich der mit Zeilennummern versehene Code ausgegeben, um die Fehlersuche zu erleichtern. Damit wird eine rudimentäre Entwicklungsumgebung realisiert, in der Programmentwürfe und Lösungsansätze getestet werden können.

eval Der Auswertungsmodus $eval$ testet die abgegebene Lösung und versucht, eine automatische Bewertung durchzuführen. In diesem Modus liefert VEA im Wesentlichen zwei Zahlen s (score) und v (validity) zwischen 0 und 1 zurück. $v = 1$ bedeutet, dass eine valide Bewertung vorgenommen werden konnte. $s = 0$ bedeutet in diesem Fall, dass die Lösung vollkommen falsch ist und $s = 1$ entsprechend, dass sie vollkommen richtig ist (z. B. wenn sie mit einer Musterlösung übereinstimmt). Eine Überprüfung durch einen Korrekteur sollte in diesem Fall nicht notwendig sein.

Werte von v kleiner 1 deuten darauf hin, dass keine sichere Bewertung vorgenommen werden konnte, der Score s also nur ein Anhaltspunkt für den Korrekteur sein sollte. Es stehen mehrere Testverfahren zur Verfügung. VEA iteriert die beiden folgenden Verfahren jeweils über alle Musterlösungen m.

- Vergleich der Ergebnisse der eingereichten Lösung und der Musterlösung bei verschiedenen Eingaben („black box“-Test). Der Auswertungsmodus $eval$ integriert den Lösungsfeldinhalt und die Musterlösung zusammen mit einem Stückchen Testcode in ein Programm, das die Ergebnisse des Lösungsvorschlags und der Musterlösungen vergleicht. Die Werte für s und v für diesen Testteil werden folgendermaßen berechnet:

$$s_{comp} = \frac{1}{1 + \#compiler\_fehler} \tag{14.1}$$

$$s^m_{eval} = \frac{0.5 * \#pos}{\#pos + \#neg} + 0.5 * s_{comp} \quad (14.2)$$

$$v^m_{eval} = s^m_{eval} \quad (14.3)$$

$\#pos$ ist dabei die Anzahl der Testfälle, für die korrekte Ergebnisse geliefert wurden, und $\#neg$ entsprechend die Fälle, für die falsche Ergebnisse geliefert wurden. Die letzte Gleichung ist in sofern plausibel, als dass ein niedriger Score-Wert nicht bedeuten muss, dass das Programm komplett falsch ist: Ein kleiner Fehler kann etwa eine Reihe von Compiler-Fehlermeldungen evozieren. In diesem Fall sollte also auch ein entsprechend niedriger Validitätswert angesetzt werden. Auf der anderen Seite bedeutet ein hoher Score-Wert, dass das Programmverhalten korrekt ist, was ein relativ sicherer Hinweis darauf sein sollte, dass auch das Programm korrekt ist, solange genügend viele Beispiele getestet wurden.

- Textueller Vergleich des Lösungsvorschlags mit der Musterlösung und der Vorbelegung des Lösungsfeldes. Auf der Basis eines Ähnlichkeitsmaßes[4] sim werden ein Score s_{sim} und ein Validitätswert v_{sim} nach folgender Formel berechnet

$$s' = sim(EXC, EXP_m) * \frac{1 - sim(INI, EXC)}{1 - sim(INI, EXP_m)} \quad (14.4)$$

$$s^m_{sim} = s'^2 * s^m_{max} \quad (14.5)$$

$$v^m_{sim} = max(1 - s', s^m_{sim})^2 \quad (14.6)$$

EXC und INI referieren auf die entsprechenden Feldinhalte (Lösungsvorschlag und Vorbelegung) und EXP_m auf die m-te Musterlösung. s' liefert also 0, falls die Vorbelegung nicht geändert wurde, und 1, falls eine der Musterlösungen eingegeben wurde. Formel 14.5 realisiert eine pessimistische Sichtweise für den Score-Wert und Formel 14.6 entsprechend für den Validitätswert. s^m_{max} ist der vorgegebene maximale Score für die m-te Musterlösung[5]. Kritisch ist natürlich die Wahl der Funktion sim. Das mögliche Spektrum reicht von spezifischen Funktionen für die einzelnen Programmiersprachen, die die syntaktische Struktur berücksichtigen können, bis zu robusten Textvergleichsmethoden unabhängig von der konkreten Program-

4 Für ein Ähnlichkeitsmaß sim muss gelten $0 \leq sim(x, y) \leq sim(x, x) = 1$.

5 Musterlösungen können unterschiedliche Güte haben. Für die erste Musterlösung muss immer gelten $s^m_{max} = 1$.

miersprache. In der gegenwärtigen VEA-Version wird die zweite Möglichkeit benutzt:

$$sim(t_1, t_2) \quad = \quad \frac{2 * similar\_text(t_1, t_2)}{|t_1| + |t_2|} \tag{14.7}$$

Dabei ist $similar\_text$ eine PHP-Funktion, die die übereinstimmenden Zeichen zählt.

- Textueller Vergleich des Lösungsvorschlags und der Musterlösungen auf der Basis verschieden starker Normalisierungsstufen. Diese Stufen sollen an folgendem Beispiel veranschaulicht werden:

```
% Aufgabe: invertieren einer Liste
% naive reverse
rev([], []). % Rekursionsabbruch
rev([X | Rest], Result) :-
       % rekursiver Aufruf für den Rest
       rev(Rest, RevRest),
       % erstes Element hinten anhängen
       append(RevRest, [X], Result).
% fertig
```

  Die wesentlichen Stufen sind folgende:

  – In der Standardstufe (0) (wird grundsätzlich angewendet) werden nur Kommentare am Anfang und Ende[6] der Lösung sowie doppelte Zeilenumbrüche und Blanks am Zeilenende entfernt.[7] Beispiel:

```
rev([], []). % Rekursionsabbruch
rev([X | Rest], Result) :-
       % rekursiver Aufruf für den Rest
       rev(Rest, RevRest),
       % erstes Element hinten anhängen
       append(RevRest, [X], Result).
```

  – In der schwachen Stufe (1) werden zusätzlich Blanks vor und nach Operatoren und Trennzeichen sowie mehrfache Blanks und Kommentare entfernt. Beispiel:

```
rev([],[]).
rev([X|Rest],Result) :-
       rev(Rest,RevRest),
       append(RevRest,[X],Result).
```

6 Meist bezieht sich ein Kommentar vor dem Code nicht auf die Programmstruktur, sondern beschreibt das Problem. Ob ähnliche Überlegungen für Endkommentare gelten, ist unklar.

7 Wie Kommentare zu erkennen sind und welche gelöscht werden, kann konfiguriert werden.

– In der mittleren Stufe (2) werden zusätzlich alle Blanks und Zeilenumbrüche entfernt. Beispiel:

```
rev([],[]).rev([X|Rest],Result):-rev(Rest,RevRest),
                                 append(RevRest,[X],Result).
```

– In der starken Stufe (3) werden zusätzlich alle kleingeschriebenen Symbole auf „a“ und alle großgeschriebenen Symbole auf „V“ reduziert und alle Symbole durch Blanks getrennt, so dass nur die Programmstruktur selbst erhalten bleibt. [8] Beispiel:

```
a ( [ ] , [ ] ) . a ( [ V | V ] , V ) : - a ( V , V ) ,
                                         a ( V , [ V ] , V ) .
```

Für die ersten drei Stufen wird auf Identität der normalisierten Texte getestet. Für die stärkste Stufe wird ein toleranter Textmatch, der maximale gemeinsame Teilstrings berücksichtigt, verwendet. Diese Vergleiche werden u. a. dazu benutzt, um ein textuelles Ergebnis ausgeben zu können:

literally same (Gleichheit auf Stufe 0) Die eingereichte Lösung entspricht wörtlich einer Musterlösung inklusive der Kommentare.

same with similar layout (Gleichheit auf Stufe 1) Die eingereichte Lösung entspricht wörtlich einer Musterlösung inklusive des Layouts. Kommentare unterscheiden sich.

same up to layout (Gleichheit auf Stufe 2) Die eingereichte Lösung entspricht wörtlich einer Musterlösung. Layout und Kommentare unterscheiden sich.

stucturally similar (ähnlich auf Stufe 3) Die eingereichte Lösung entspricht strukturell einer Musterlösung. Funktionsnamen und Variablennamen können sich unterscheiden. Die Abfolge von Definitionen kann sich unterscheiden.

Für diese Fälle können auch feste vorgegebene Werte für den Score s und den Validitätswert v vergeben werden. In diesem Fall überschreiben diese die Werte aus den vorigen Verfahren.

Aus den Werten für die verschiedenen Musterlösungen berechnen sich die endgültigen Werte (v_{sim}, s_{sim}) als

$$(v_{sim}, s_{sim}) \quad = \quad max_m((v^m_{sim}, s^m_{sim})) \tag{14.8}$$

8 Insbesondere die stärkste Normalisierungsstufe sollte abhängig von der Programmiersprache sein. Es ist daran gedacht, für alle Stufen in der Konfiguration parametrisierbare Pattern zu erlauben.

$$(v_{eval}, s_{eval}) \quad = \quad max_m((v^m_{eval}, s^m_{eval})) \tag{14.9}$$

bezüglich der lexikalischen Ordnung.

Für jede Aufgabe kann festgelegt werden, ob nur der Lösungsmengenvergleich, nur der Textvergleich oder eine Kombination von beiden (Default) in die endgültigen Werte von s und v eingehen, die zur Berechnung eines Bewertungsvorschlages benutzt werden. Diese relativ groben heuristischen Verfahren zur Berechnung der Werte s und v haben sich insbesondere für Prolog als durchaus brauchbar erwiesen.

Es ergibt sich folgende grobe Interpretation der Werte s und v:

$v = 1$ Die Lösung konnte sicher bewertet werden. s kann als Faktor zur Berechnung der Punkte für die eingereichte Lösung verwendet werden. Es könnte sich aber um ein Plagiat halten, falls z. B. eine der Musterlösungen eingereicht wurde.

$v \geq 0.7, s \geq 0.5$ Die Lösung scheint partiell richtig zu sein.

$v = 0.5, s = 0.5$ Die Lösung konnte nicht bewertet werden. Vermutlich ist sie falsch.

Dieses Schema ist natürlich sehr grob und muss noch verfeinert werden.

14.2.3 Konfiguration

Zur Konfiguration eines Services sind zwei Dateien notwendig:

config enthält eine Reihe von Attribut-Wert-Paaren zur Konfiguration des Systems. Hier werden u. a. die Auswertungsparameter eingestellt, die Kommentarspezifikation der Programmiersprache festgelegt sowie die erlaubten Funktionen und Symbole aufgelistet.

Eine typische Konfiguration für die Programmiersprache Prolog sieht folgendermaßen aus:

```
# Debug-Flag
dbf=0

# Lösungen, die gleich sind bezüglich dieser
# Normalisierungsebene werden nicht neu evaluiert
hash_result=1

# Comments (line comments, multi line comments
line_comment=%
multi_line_comment=\/\*.*?\*\/
```

```
# Eine einfache Methode Missbrauch zu erschweren
# Alle entsprechenden Symbole werden mit einem Präfix
# versehen
save_pattern=\b[a-z]\w*\b
save_prefix=s__

# check for non-meaningful input
empty_pattern=[^()[\].:]*

# Nur diese Symbole können in Lösungen benutzt werden.
# Für diese Symbole wird der Präfix unterdrückt
allow="""
   member,xfx,
   fx,yfx,xfy,fy,xf,yf,op,current_op, display,
   not,true,fail, trace,
   findall, forall,
   length, nth0, random, mod, max, number, is_list,
   time, use_module,
   dynamic, assert, asserta, assertz,
   retract, abolish, sort, maplist, reverse, permutation
"""
```

exec ist ein Shell-Script und implementiert für die Funktionen $query$ und $eval$ den Compiler-Aufruf. Hier ist darauf zu achten, dass der Ressourcenverbrauch des Prozesses limitiert und die Priorität niedrig eingestellt wird, damit der Prozess die Maschine nicht blockiert. Zudem sind Sicherheitsmaßnahmen gegen den Missbrauch dieser Schnittstelle zu bedenken.

Darüber hinaus muss für die Funktion $eval$ ein Programm zur Verfügung gestellt werden, das die IO-Relation des Lösungsvorschlags überprüft. In den implementierten Services wird dazu die erste Musterlösung benutzt und für eine Reihe von Testfällen die Ergebnisse beider Programme verglichen.

14.3 Ein Beispiel

Als typisches kleines Beispiel für eine Prologaufgabe wählen wir wieder das Problem der Invertierung der Reihenfolge der Listenelemente.

Beispiel: Invertieren einer Liste

Die Aufgabenstellung

> Definiere ein Prädikat $rev(L1, L2)$ das wahr ist falls $L2$ die Elemente von $L1$ in inverser Reihenfolge enthält.

Beispielaufruf

```
rev([1,2,3,4,5,6,7,8,9,0],Result)
```

liefert

```
Result=[0,9,8,7,6,5,4,3,2,1]
```

Drei Musterlösungen

- Naive Variante (benutzt *append*)

```
% naive reverse
m_l_rev([], []).
m_l_rev([X | Rest], Result) :-
   m_l_rev(Rest, RevRest),
   append(RevRest, [X], Result).
```

- Minimalistische Variante (keine Hilfsprädikate oder zusätzliche Argumente)

```
% minimalistic solution
m_l_rev([], []).
m_l_rev([X], [X]).
m_l_rev([F | R1], [L | R4]) :-
   m_l_rev(R1, [L | R2]),
   m_l_rev(R2, R3),
   m_l_rev([F | R3], R4).
```

- Optimale Variante (benutzt ein Akkumulator-Argument)

```
% optimal solution
m_l_rev(L1, L2) :-
   m_l_rev(L1, [], L2).
m_l_rev([], AC, AC).
m_l_rev([X |R], AC, Result) :-
   m_l_rev(R, [X | AC], Result).
```

Testprogramm

```
m_test :-
   mu_test(mu_genlist(L1,5),
           m_l_rev(L1,L2),rev(L1,L2)).
```

mu_test erwartet im ersten Argument einen Testdatengenerator (hier werden durch $mu\_genlist$ Listen generiert) und im zweiten und dritten Argument die zu vergleichenden Lösungen.

Lösungsbeispiele und ihre Bewertung

Werden Musterlösungen eingegeben, werden diese erkannt und entsprechend bewertet. Daher werden im Folgenden nur Fälle diskutiert, die sich von allen Musterlösungen unterscheiden.

- Korrekt, aber geänderte Klauselreihenfolge

```
rev([X], [X]).
rev([F | R1], [L | R4]) :-
   rev(R1, [L | R2]), rev(R2, R3),
   rev([F | R3], R4).
rev([], []).
```

  Bewertung:

  - Strukturell ähnlich zu Musterlösung 2
  - korrekte Ein/Ausgabe-Relation
  - validity=1, score=1

- Korrekte Programmstruktur, Variablennamen falsch

```
rev([X], [X]).
rev([F | R1], [L | R4]) :-
   rev(R1, [L | R3]), rev(R4, R3),
   rev([F | R3], R4).
rev([], []).
```

  Bewertung:

  - Strukturell ähnlich zu Musterlösung 2
  - Gegenbeispiel:

    ```
    rev([5,4],[4,5])
    ```

    sollte wahr sein, ist aber falsch
  - validity=0.7, score=0.7

- Inkorrekt: falsche Definition

```
rev(X, [X|_]).
rev(X, [Y|R]) :-
   rev(X, R).
```

  Bewertung:

  - Keine Ähnlichkeit zu einer Musterlösung
  - Gegenbeispiel:

    ```
    rev([],[[]|B])
    ```

    ist wahr, sollte aber falsch sein.
  - validity=0.5, score=0.5

- Korrekt: andere Reihenfolge der Prädikatsdefinitionen, andere Prädikatsnamen, andere Variablennamen und andere Verwendung der Argumente.

```
rev_a([], Accu, Accu).
rev_a([X | Rest], Result, Accu) :-
   rev_a(Rest, Result, [X | Accu]).

rev(Liste1, Liste2) :- rev(Liste1, Liste2, []).
```

  Bewertung:

  - Strukturell ähnlich zu Musterlösung 3
  - korrekte Ein/Ausgabe-Relation
  - validity=1, score=1

- Lösungstemplate

```
rev( ).
```

  Bewertung:

  - leere Eingabe
  - validity=1, score=0

- Unsinnseingabe

```
dazu habe ich keine Lust!!!!!!!
```

  Bewertung:

  - leere Eingabe
  - validity=1, score=0

14.4 Zusammenfassung und Ausblick

Das Modul Vips (VIrtuelles PrüfungsSystem) übernimmt in StudIP die komplette Verwaltung des Übungsbetriebs einer Veranstaltung. Vips wurde um eine Komponente VEA (Vips-Evaluation-Assistent) zur automatischen Bewertung von Programmieraufgaben erweitert. VEA stellt sowohl semantische (Programmverhalten) als auch syntaktische (Beurteilung des Programmcodes) Verfahren zur Bewertung von Programmieraufgaben zur Verfügung. VEA ist immer noch in einem experimentellen Stadium. Sowohl die Funktionalität, als auch die Kommunikation zwischen Vips und VEA sollen weiterentwickelt werden.

Literatur für dieses Kapitel

[GW13] Helmar Gust und Nadine Werner. „Automatische Bewertung von Übungsaufgaben in VIPS.“ In: *Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2013)*. Bd. 1067. CEUR Workshop Proceedings. 2013.

[Pey+00] Christoph Peylo u. a. „A Web-based intelligent educational system for PROLOG“. In: *Proceedings of the International Workshop on Adaptive and Intelligent Web-Based Education Systems held in conjunction with ITS 2000 Montreal, Canada.* 2000, S. 85–96.

[Str+15] Sven Strickroth u. a. „ProFormA: An XML-based exchange format for programming tasks“. In: *eleed* 11.1 (2015). URL: https://eleed.campussource.de/archive/11/4138.