PERPLEX: Logisches Programmieren an Hand von Beispielen

Michael Spenke und Christian Beilken, St. Augustin

Zusammenfassung

Es wird PERPLEX vorgestellt, eine Programmierumgebung für Endbenutzer. PERPLEX vereint die Konzepte der logischen Programmierung mit der interaktiven Benutzerschnittstelle von Tabellenkalkulationsprogrammen. Auf diese Weise wird einerseits eine interaktive, inkrementelle Form der logischen Programmierung ermöglicht und andererseits Funktionalität und Anwendungsbereich eines herkömmlichen Tabellenkalkulationsprogramms beachtlich erweitert. Berechnungen und Anfragen werden durch Constraints über mehreren Felder formuliert. Neue Prädikate werden durch Vorrechnen von Beispielen definiert, aus denen Regeln abgeleitet werden. Dadurch übernehmen konkrete Zwischenergebnisse die Rolle von abstrakten logischen Variablen. PERPLEX wurde im Rahmen des Verbund-Projektes WISDOM entwickelt und ist auf einer Symbolics Lisp Maschine verfügbar.

1. Einleitung

Hoch interaktive, graphische Benutzerschnittstellen haben den Umgang mit Standardanwendungen deutlich vereinfacht. Es besteht aber auch eine große, noch unbefriedigte Nachfrage nach Anwendungen, die speziell auf die Bedürfnisse einzelner Benutzer zugeschnitten sind (vgl. Rockart, 1983). Dieser Bedarf kann unmöglich durch eine wachsende Zahl von Programmierern gedeckt werden (vgl. Shu, 1985). Daher werden Werkzeuge benötigt, mit denen Endbenutzer ihre eigenen Anwendungen erstellen können.

PERPLEX bietet die Mächtigkeit logischer Programmierung unter der bewährten Benutzerschnittstelle von Tabellenkalkulationsprogrammen an. Inkrementelle Anfragen werden als natürliche Erweiterungen der Standardfunktionalität eines Tabellenkalkulationsprogrammes eingeführt. An Hand von Beispielen können benutzerdefinierte Funktionen und logische Regeln in einheitlicher Weise definiert werden, ohne daß der Benutzer eine neue formale Sprache lernen muß.

Ihren Erfolg verdanken **Tabellenkalkulationsprogramme** unter anderem der Tatsache, daß sich viele Probleme auf natürlicher Weise im Matrixformat darstellen lassen. Noch wichtiger ist, daß die strikte Trennung von Programmier- und Testphase aufgehoben ist. Da die Werte von Variablen ständig angezeigt werden, wird die Programmierung zu einer weniger abstrakten Aufgabe: Anstatt mit Variablen und deren vermuteten Werten kann der Benutzer mit konkreten Zwischenergebnissen arbeiten. Die meisten Fehler werden auf diese Art sofort entdeckt, und ein explorativer Programmierstil wird gefördert, da die Auswirkungen von Änderungen an den Eingabewerten oder den Formeln sofort sichtbar werden.

Ein konventionelles Tabellenkalkulationsprogramm stellt allerdings noch keine universelle Programmierumgebung dar, hauptsächlich weil Möglichkeiten fehlen, um neue Funktionen zu definieren und große Programme zu modularisieren (vgl. Arganbright, 1986). Zwar wird versucht, diese Lücke mit Makros zu schließen; dies stellt aber einen Schritt zurück zu traditionellen Programmiersprachen dar. Daher stellen Makros eine große Barriere für Endbenutzer dar: Während Tabellenkalkulationsprogramme sich großer Beliebtheit erfreuen, werden Makros kaum verwendet. PEPRLEX ermöglicht dagegen einen Übergang zum Programmieren, ohne das Paradigma zu wechseln.

Logische Programmiersprachen wie z.B. Prolog sind stärker problemorientiert und leichter erlernbar als prozedurale Sprachen. Außerdem sind sie aufgrund ihrer flexiblen Berechnungsrichtung ausdrucksstärker als funktionale Programmiersprachen (vgl. Reddy, 1986) und

können in natürlicher Weise als Datenbankabfragesprachen eingesetzt werden (vgl. Beyer, 1985).

Logische Programmiersprachen sind prinzipiell für die **interaktive** Nutzung geeignet. Dieses Potential bleibt jedoch weitgehend ungenutzt, solange eine einfache Zeilenschnittstelle Verwendung findet. Die interaktive, graphische Benutzungsoberfläche von PERPLEX ermöglicht dem Endbenutzer einen leichten Zugang zu den Möglichkeiten logischer Programmierung.

Van Emden et al. (1985) verwenden ebenfalls eine Matrix, um die Lösungen einer inkrementellen Prolog-Anfrage anzuzeigen, bieten aber keine Möglichkeit zur Definition benutzerdefinierter Prädikate. In PERPLEX werden Datenbankanfragen in ähnlicher Weise formuliert, wie in Query-by-Example (vgl. Zloof, 1977); unser Konzept geht aber weit über Datenbankanfragen hinaus.

2. Das Berechnungsmodell

Auf den ersten Blick unterscheidet sich PERPLEX nicht von einem herkömmlichen Tabellenkalkulationsprogramm: Eine Tabelle besteht aus Feldern, die in Zeilen und Spalten angeordnet sind und mit A1, A2, ..., B1, B2, ... bezeichnet werden. Jedes Feld repräsentiert eine **Variable**, deren Wert — falls vorhanden — ständig angezeigt wird. Werte können Zahlen, Zeichenketten oder Listen von Werten sein.

2.1 Constraints anstelle von funktionalen Ausdrücken

In einem typischen Tabellenkalkulationsprogramm kann ein Feld entweder eine Konstante oder eine Formel enthalten. Dadurch werden die Felder in **Eingabefelder** (mit einer Konstanten) und **Ausgabefelder** (mit einer Formel) unterteilt. In PERPLEX sind Formeln jedoch nicht einem einzelnen Feld zugeordnet, sondern gelten für eine **Menge von Feldern**. Dementsprechend werden Formeln nicht als funktionale Ausdrücke angegeben, die einen Wert liefern, sondern als **Constraints**, die durch die Feldinhalte erfüllt werden sollen. Beispiele für solche Constraints sind:

- Der Wert von A1 soll doppelt so groß sein wie der von B1.
- Der Wert von A1 soll größer sein als der Wert von B1.
- Die Summe von A1, B1 und C1 soll 100 sein.
- Die Person in A1 soll der Vorgesetzte der Person in B1 sein.

Die Liste der vom Benutzer eingegebenen Constraints wird neben der Tabelle angezeigt. Während in herkömmlichen Tabellenkalkulationsprogrammen Ausdrücke mit Hilfe von **Funktionen** spezifiziert werden, werden Constraints mit Hilfe von **Prädikaten** formuliert, die keine feste Ein-/Ausgaberichtung vorschreiben. Die Standardsyntax für Constraints ist `P(`x_1`..`x_n`)`, wobei `P` ein Prädikat ist und x_1 .. x_n Feldnamen oder Konstanten sind. Die Liste der Constraints entspricht in etwa einer Prolog-Anfrage.

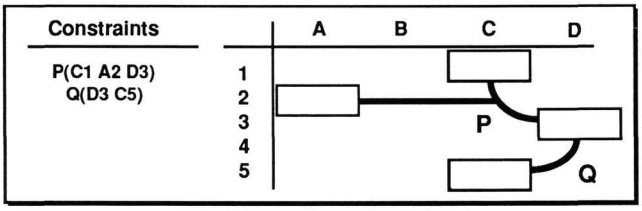

Bild 1: Eine Tabelle mit zwei Constraints

2.2 Evaluierung von Constraints

Aufgabe des Evaluierer ist es, eine **Belegung** für diejenigen Variablen zu finden, die in Constraints referenziert werden (und nicht durch eine Konstante gegeben sind), so daß alle Constraints erfüllt sind. Wenn eine Belegung gefunden wurde, wird der Wert jeder Variable in der Tabelle angezeigt. Jede Belegung stellt eine Lösung für die durch die Constraints definierte Anfrage dar. Im allgemeinen kann es mehrere Lösungen geben, von denen zunächst die erste

angezeigt wird. Der Benutzer kann dann durch die Menge der Lösungen blättern, wobei jeweils eine komplette Variablenbelegung angezeigt wird. Möglicherweise wird jedoch aufgrund widersprüchlicher Constraints überhaupt keine Lösung gefunden.

Bei der Verwendung von Constraints bleibt offen, ob ein Feld als Ein- oder Ausgabefeld dient. Daher kann man den ersten Beispiel-Constraint folgendermaßen verwenden:

- Wenn sowohl A1 als auch B1 als Konstanten gegeben sind, prüft der Evaluierer lediglich, ob die Bedingung erfüllt ist.
- Ist nur B1 gegeben, wird A1 durch Verdoppeln von B1 ermittelt.
- Ist nur A1 gegeben, wird B1 durch Halbieren von A1 ermittelt.
- Wenn weder A1 noch B1 gegeben sind, wird der Constraint ignoriert.

Jedem Prädikat ist die Menge seiner erlaubten **Aufrufmodi** zugeordnet. Die Aufrufmodi geben an, welche Parameter gegeben sein müssen, wenn das Prädikat aufgerufen wird. Formal stellt jeder Modus eine Teilmenge der formalen Parameter des Prädikates dar. Falls alle Parameter von wenigstens einem Aufrufmodus gegeben sind, wird der Constraint **evaluierbar**, das heißt die Werte der übrigen Parameter können berechnet werden. Die Aufrufmodi werden verwendet, um die **Reihenfolge der Evaluierung** von Constraints zu bestimmen: Es wird stets der erste noch nicht ausgewertete, evaluierbare Constraint ausgewertet. Die Evaluierung eines Constraints kann weitere Variablen instantiieren, so daß zusätzliche Constraints evaluierbar werden. Wenn keine Constraints mehr evaluierbar sind, werden die übrigen Constraints vom Evaluierer ignoriert.

Ein Aufrufmodus eines Prädikates wird als **regulärer Ausdruck** über den Symbolen `in` und `out` beschrieben. Beispielsweise bedeutet der Aufrufmodus `(in in out)` des Prädikates `Times`, daß das Produkt (der letzte Parameter) berechnet werden kann, sofern beide Faktoren (die ersten beiden Parameter) gegeben sind. Es müssen nicht alle Aufrufmodi eines Prädikates explizit angegeben werden, da stets Konstanten für Ausgabeparameter vorgeben werden können.

2.3 Verschiedene Typen von Prädikaten

Bild 2 zeigt einige typische Prädikate mit ihren Aufrufmodi.

Prädikat	Aufrufmodi	Beispiele
Less	(in in)	Less(A4 5)
Times	(in in out)	Times(A3 2 A4)
	(in out in)	
	(out in in)	
And	(in in)	And(true true)
Member	(out in)	Member(A1 [1 2 3 4 5])
List	(in* out)	List(1 2 3 4 5 B1)
	(out* in)	List(A1 A2 A3 [1 2 3])
Concat	(in in out)	Concat("in" "put" A3)
	(out out in)	Concat(A1 A2 "output")
Substring	(in out out out)	Substring("core" 2 3 B4)
Employees	(out out out out)	Employees(A1 A2 A3 A4)
Copy File	(in in)	Copy File("file1" "file2")
Pie Chart	(in in in)	Pie Chart("Sales" A2 A4)

Bild 2: Einige typische Prädikate

Es gibt drei verschiedene Arten von Prädikaten, die in gleicher Weise angewendet werden und sich nur durch ihre zugrundeliegende Implementierung unterscheiden:

- **Eingebaute Prädikate**
  Ein eingebautes Prädikat ist durch eine Menge von Funktionen definiert — eine für jeden Aufrufmodus. Die Eingabeparameter jeder Funktion entsprechen dem zugehörigen Aufrufmodus. Jede Funktion liefert eine Lösungsmenge für die Ausgabeparameter. Für den Modus `(in in out)` bei dem Prädikat `Times` gibt es beispielsweise eine Funktion, die zwei Zahlen als Argumente hat und ihr Produkt liefert.

- **Datenbankrelationen**
  Datenbankrelationen werden durch eine Menge von Tupeln repräsentiert. Sie entsprechen in etwa den Fakten von Prolog.
- **Benutzerdefinierte Prädikate**
  Benutzerdefinierte Prädikate können mit Hilfe von eingebauten Prädikaten, Datenbankrelationen und anderen benutzerdefinierten Prädikaten definiert werden. Dies wird später genauer erläutert.

2.4 Rückwärtsberechnungen

Bild 3 zeigt, wie eine einfache Berechnung mit Hilfe von Constraints ausgeführt werden kann.

Constraints		A	B	C
Plus(B2 100 C2)	1	Basis:	200	
Divide(C2 100 C3)	2	Prozent:	14	114
Times(B1 C3 B3)	3	Gesamt:	228	1.14
	4			

Bild 3: Berechnung eines prozentualen Aufschlages mit expliziten Zwischenergebnissen

Sobald die drei Constraints definiert sind, kann man nicht nur mit verschiedenen Eingabewerten experimentieren, sondern auch **rückwärts** rechnen. Wird beispielsweise der Basisbetrag (Feld B1) und der Gesamtbetrag (B3) angegeben, so wird der Prozentsatz errechnet. Um das Resultat zu finden, muß der Evaluierer eine geeignete Evaluierungsreihenfolge für die drei Constraints bestimmen. Wie bereits oben erwähnt, wird stets der **erste evaluierbare Constraint** ausgewählt. In unserem Beispiel kann der dritte Constraint sofort evaluiert werden; dadurch wird der zweite Constraint evaluierbar und schließlich auch der erste.

2.5 Syntaktischer Zucker

Da es sehr ungewohnt ist, arithmetische Ausdrücke in Prädikatnotation anzugeben, ist auch die übliche Infixnotation für Constraints zugelassen. Statt der drei Constraints in Bild 3 kann man also auch kürzer schreiben:

```
B3 = B1 * (B2 + 100) / 100
```

Die interne Repräsentation basiert aber nach wie vor auf Constraints (mit Hilfsvariablen), so daß Rückwärtsberechnungen weiterhin möglich sind. Man kann Prädikate auch in geschachtelten Ausdrücken wie Funktionen benutzen. Dabei wird der letzte Parameter weggelassen und als Ergebnis des Funktionsaufrufes geliefert. Zum Beispiel ist

```
A3 = Sinus(A1) + A2
```

äquivalent zu

```
Sinus(A1 AUX1)
Plus(AUX1 A2 AUX2)
Equal(AUX2 A3)
```

Weiterhin können Operatoren wie `>`, `<`, `and`, `or` und `not` benutzt werden, um entweder Bool'sche Ausdrücke (die `true` oder `false` liefern) oder Constraints (die erfüllt sein müssen) zu definieren. Beispiel: Der Constraint

```
A1>5 and A2<10 and not A3
```

ist nur erfüllt, wenn alle drei Bedingungen erfüllt sind. Dagegen wird der Constraint

```
A4 = (A1>5 and A2<10 and not A3)
```

A4 entweder `true` oder `false` zuweisen.

2.6 Mehrfachlösungen

Mehrfachlösungen erhält man typischerweise, wenn Datenbankrelationen verwendet werden. Datenbankanfragen können ebenfalls mit Hilfe der bisher beschriebenen Mechanismen

formuliert werden. Die folgenden Beispiele basieren auf einer kleinen relationalen Datenbank mit 10 Angestellten (eingeführt durch Zloof, 1977; siehe Bild 4).

Name	Gehalt	Vorgesetzter	Abteilung
anderson	6000	murphy	toy
henry	9000	smith	toy
hoffman	16000	morgan	cosmetics
jones	8000	smith	household
lewis	12000	long	stationery
long	7000	morgan	cosmetics
morgan	10000	lee	cosmetics
murphy	8000	smith	household
nelson	6000	murphy	toy
smith	12000	hoffman	stationery

Bild 4: Die Datenbank `Employees`

Wenn man den Constraint **`Employees(B1 B2 B3 B4)`** eingibt, liefert der Evaluierer alle Tupel der Datenbank als Lösung. Die Angaben zu dem ersten Angestellten werden in den Feldern B1 .. B4 angezeigt und die Gesamtzahl der Lösungen erscheint oberhalb der Tabelle. Die anderen Lösungen werden angezeigt, wenn mit Hilfe der Pfeile geblättert wird.

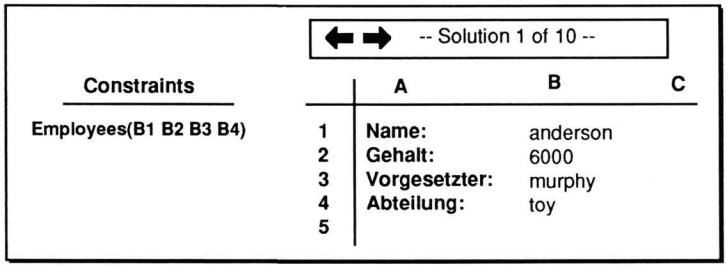

Bild 5: Eine einfache Datenbankanfrage

Speziellere Anfragen können gestellt werden, indem man Konstanten vorgibt. Wenn z.B. **`jones`** in B1 eingetragen wird, ergibt sich eine eindeutige Lösung: Nur die Daten von Jones werden angezeigt. Wenn man stattdessen **`6000`** in B2 einträgt, bleiben 2 Lösungen übrig, da es zwei Angestellte gibt, die 6000 $ verdienen.

2.7 Kombinierte Datenbankanfragen

Um komplexe Datenbankanfragen zu formulieren, kann man zusätzliche Constraints angeben. Um beispielsweise alle Angestellten zu ermitteln, die mehr als 8000 $ verdienen, fügt man den Constraint **`B2>8000`** hinzu und erhält 5 Lösungen.

Es können natürlich nicht nur zusätzliche Restriktionen angegeben werden, sondern man kann auch abgeleitete Informationen berechnen: Um beispielsweise das Jahresgehalt jedes Angestellten zu berechnen, gibt man den Constraint **`C2=B2*12`** ein.

Es ist auch möglich zwei Datenbankzugriffe in einer Anfrage zu kombinieren: Angenommen, es sollen alle Angestellten mit demselben Vorgesetzten ermittelt werden. Man beginnt wiederum mit dem Constraint **`Employees(B1 B2 B3 B4)`** und trägt **`jones`** in B1 ein, um zunächst mit einem konkreten Beispiel zu arbeiten. Dadurch wird **`smith`** als der Vorgesetzte von Jones angezeigt und durch die Constraints **`Employees(C1 C2 C3 C4)`** und **`B3=C3`** können alle Angestellten ermittelt werden, die ebenfalls Smith als Vorgesetzten haben. Es ergeben sich 3 Lösungen und die Kollegen von Jones werden in C1 angezeigt. Beim Blättern durch die Lösungen fällt auf, daß sich auch Jones noch unter diesen Kollegen befindet. Um diesen Mangel zu beseitigen, fügt man noch den Constraint **`B1≠C1`** hinzu (Bild 6). Zum Schluß kann

man noch das Beispiel **jones** aus B1 löschen und erhält 10 Paare von Angestellten jeweils mit demselben Vorgesetzten.

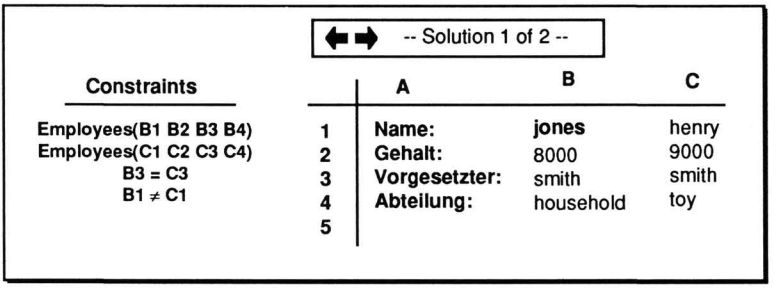

Bild 6: "Wer hat denselben Vorgesetzten wie Jones?"

2.8 Inkrementelles Problemlösen

Wie das letzte Beispiel gezeigt hat, liegt die Stärke von PERPLEX darin, daß eine **interaktive, explorative** Vorgehensweise beim Problemlösen unterstützt wird. Der Benutzer muß nicht sofort eine komplette Anfrage formulieren, sondern kann sich der Lösung schrittweise nähern und dabei auf konkrete Zwischenergebnissse zurückgreifen. Abhängig von den angezeigten Zwischenergebnisssen kann er weitere Constraints hinzufügen, um unerwünschte Lösungen auszuschließen oder existierende Constraints modifizieren falls sie zu restriktiv sind. Die Möglichkeit, auf konkrete Zwischenergebnisse mit der Maus zu zeigen, macht die Verwendung von abstrakten Variablen, dessen Werte man sich vorstellen muß, überflüssig.

3. Programmieren mit Beispielen

In den bisherigen Beispielen wurde gezeigt, wie man existierende Prädikate zum inkrementellen Problemlösen einsetzt. Neue, benutzerdefinierte Prädikate können an Hand von Beispielen definiert werden (**Programming-by-example**).

Die Grundidee ist hier, zunächst die Lösung für ein Problem an Hand eines typischen Beispiels exemplarisch in dem bereits beschriebenen explorativen Stil zu erarbeiten und anschließend die relevanten Parameter des Problems zu identifizieren. Durch die während des Problemlösungsprozesses festgelegten Constraints wird eine bestimmte Relation zwischen den Eingabe- und Ausgabeparametern definiert. Falls diese Relation von genereller Bedeutung ist und voraussichtlich auch in anderen Zusammenhängen gebraucht wird, kann man ein **neues, benutzerdefiniertes Prädikat** einführen, indem man einen Namen für die Relation vergibt und die relevanten Parameter bezeichnet.

Fallunterscheidungen können spezifiziert werden, indem mehrere Beispiele zu demselben Prädikat gegeben werden, aus denen dann je eine **Regel** abgeleitet wird. Dadurch ist es sogar möglich, rekursive Prädikate zu definieren.

3.1 Ein Beispiel für ein Beispiel

Angenommen, wir haben wiederholt das Problem, einen Teil eines Wortes durch eine andere Zeichenkette zu ersetzen. Als Demonstrationsbeispiel versuchen wir "act" in "interaction" durch "sect" zu ersetzen. Die Grundidee ist, das Wort "interaction" in "inter", "act" und "ion" zu zerlegen und dann den Präfix, den neuen Infix und den Postfix zu konkatenieren.

Zunächst tragen wir die Beispielkonstanten ein und beschriften die Felder entsprechend ihrer Bedeutung (Bild 7). Sodann verwenden wir das Prädikat **Substring**, um Anfangs- und Endposition von "act" in "interaction" zu bestimmen und erhalten 6 und 8. Als nächstes versuchen wir, den gesuchten Präfix als Anfangsstück von "interaction" von 1 bis 6 zu erhalten. Sobald wir aber das Resultat "intera" sehen, stellen wir fest, daß uns ein Fehler unterlaufen ist: Die Endposition von "inter" ist um 1 kleiner als Anfangsposition von "act" und der letzte Constraint muß dementsprechend modifiziert werden. Um den Postfix "ion" bestimmen zu können, benötigen wir zunächst die Gesamtlänge von "interaction", die mit Hilfe des Prädikates

String-length ermittelt werden kann. Schließlich konkatenieren wir die 3 Teilstücke zu dem Wort "intersection".

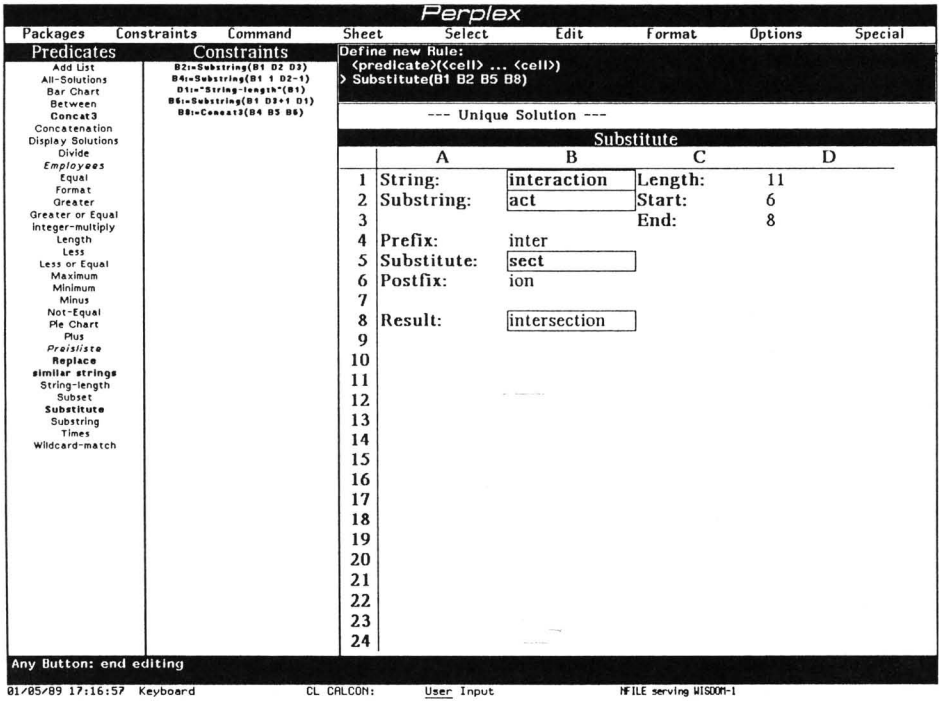

Bild 7: Exemplarische Lösung des Ersetzungsproblems (Bildschirmabzug)

Wir können nun mit anderen Wörtern experimentieren, um festzustellen, ob unsere Lösung allgemeingültig ist. Sobald wir uns davon überzeugt haben, definieren wir ein neues Prädikat **Substitute** mit den Parametern **String**, **Substring**, **Substitute** und **Result**, so daß **Substitute("interaction" "act" "sect" "intersection")** gilt. Dazu wählen wir das Kommando **Define Rule**, tippen den Namen des neuen Prädikates und klicken mit der Maus auf die relevanten Parameterfelder. Auf diese Weise haben wir dem System mitgeteilt, daß die Parameter des neu definierten Prädikates **Substitute** in der gleichen Relation stehen sollen wie die bezeichneten Felder.

Mit dem neuen Prädikat können nun zu dem Demonstrationsbeispiel analoge Ersetzungsprobleme gelöst werden: Zu gegebenem **String**, **Substring** und **Substitute** wird der Ausgabeparameter **Result** geliefert. Darüberhinaus sind aber noch weitere Verwendungen, insbesondere andere Aufrufmodi, für das neue Prädikat möglich:

- **Substitute("interaction" "i" "o" A4)** liefert die zwei Lösungen "onteraction" und "interactoon".
- **Substitute("interaction" A2 "*" A4)** ersetzt jedes der 78 möglichen Teilworte von "interaction" durch "*".
- **Substitute("interaction" A2 "sect" "intersection")** findet heraus, daß "act" ersetzt werden muß.
- **Substitute("interaction" A2 A3 "intersection")** liefert alle 36 möglichen Umformungen von "interaction" nach "intersection".

3.2 Vorteile des Programmierens mit Beispielen

Das obige Beispiel zeigt die wichtigsten Vorteile unseres Ansatzes:

- Es muß keine neue Sprache erlernt werden, um neue Prädikate definieren zu können. Das Programmieren allgemeiner Lösungen für eine Problemklasse ist kaum schwieriger als das Lösen eines einzelnen, konkreten Problems. Der Benutzer muß nicht einmal von Anfang an planen, ein neues Prädikat zu definieren. Wenn er eine allgemeingültige Lösung für ein Problem erarbeitet hat, kann er sich noch nachträglich entscheiden, ein neues Prädikat zu definieren, indem er einfach die relevanten Parameter identifiziert. Auf diese Weise kann der von der Tabellenkalkulation gewohnte explorative Arbeitsstil auch bei der Programmierung beibehalten werden.
- Die zulässigen Aufrufmodi eines neuen Prädikates werden **automatisch** errechnet. So kann der Benutzer sofort feststellen, ob sein Beispiel im gewünschten Umfang **generalisiert** werden konnte. Oftmals kann man mit einem sehr einfachen Beispiel arbeiten, wo mehr Parameter vorgegeben sind als nötig und sich daher eindeutige Lösungen ergeben und dennoch den allgemeinen Fall implementieren, wo weniger Parameter vorgegeben sind und Mehrfachlösungen auftreten. Ebenso kann oft eine Vorwärtsberechnung demonstriert werden, und das System entdeckt die Möglichkeit, mit dem neuen Prädikat auch **rückwärts** zu rechnen. Auf diese Weise stellt sich die logische Programmierung als natürliche Erweiterung der funktionalen Programmierung dar.
- Da zu jeder Regel die Tabelle mit dem Originalbeispiel abgespeichert wird, können existierende Regeln in natürlicher Weise **editiert** werden: Die Tabelle mit dem Beispiel kann mit den ganz normalen Operationen modifiziert werden.

4. Zusammenfassung

Es wurde gezeigt, daß die Konzepte der logische Programmierung in natürlicher Weise mit der populären Benutzerschnittstelle von Tabellenkalkulationsprogrammen kombiniert werden können. Auf diese Weise kann der explorative Arbeitsstil, der durch die direkte Anzeige von Zwischenergebnissen in einer Tabelle gefördert wird, auch bei inkrementellen Anfragen und sogar bei der Definition von neuen Prädikaten an Hand von Beispielen beibehalten werden.

PERPLEX wurde erfolgreich auf einer Symbolics Lisp Maschine implementiert (siehe Spenke und Beilken, 1988). Erste Resultate einer Interaktionsanalyse (vgl. Grunst, 1988) von PERPLEX im Vergleich zu EXCEL (Campbell, 1987) haben gezeigt, daß das Programmierparadigma zwar für den Endbenutzer geeignet ist, aber die Benutzer Schwierigkeiten mit der Schnittstelle der Lisp Maschine haben. Zur Zeit wird PERPLEX auf einem Macintosh reimplementiert.

Literatur

Arganbright, Deane E. (1986): **Mathematical Modeling with Spreadsheets**, ABACUS, 3(4), S. 18-31.

Attardi, Giuseppe; Simi, Maria (1982): **Extending the Power of Programming by Example**, in: J.O. Limb (Hrsg.): SIGOA Conference on Office Information Systems, Philadelphia, PA, S. 52-66.

Bauer, Michael A. (1979): **Programming by Examples**, Artificial Intelligence 12, S. 1-21.

Beyer, Rudolf (1985): **Database Technology for Expert Systems**, in: Internationaler GI-Kongreß 85: "Wissensbasierte Systeme", München, GI Tagungsband, Informatik Fachberichte 112, Springer-Verlag, S. 1-16.

Borning, Alan (1981): **The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory**, ACM TOPLAS, 3(4), S. 353-387.

Campbell, M. (1987): **Excel Macro Treasury**, Macworld 11/87, S. 122-125.

Colmerauer, Alain (1987): **Opening the Prolog III Universe**, BYTE, 8/87, S. 177-158.

Eisenstadt, Marc; Hasemer, Tony; Kriwaczek, Frank (1985): **An Improved User Interface for PROLOG**, In: B. Shackel (Hrsg.): Human-Computer Interaction—INTERACT '84, Elsevier Science Publishers B.V., S. 385-389.

van Emden, M.H.; Ohki, M.; Takeuchi, A. (1985): **Spreadsheets with Incremental Queries as a User Interface for Logic Programming**, ICOT Technical Report, TR-144, 10/85.

Fischer, Gerhard; Rathke, Chrsitian (1988): **Knowledge-Based Spreadsheets**, Proceedings of AAAI-88, 7th National Conference on Artificial Intelligence, St. Paul, MI, 8/88.

Grunst, Gernoth G. (1988): **Reconstructing Cooperative Advising—Interaction Analysis as a Research Tool for the Development of Intelligent Assistants**, interner Bericht, GMD.

Halbert, Daniel C. (1984): **Programming by Example**, PhD Thesis, Computer Science Division, Dept. of EE&CS, University of California, Berkeley.

Kriwaczek, Frank (1986): **LogiCalc—A PROLOG Spreadsheet**, in: Bob Kowalski, F. Kriwaczek: Logic Programming, Addison-Wesley, S. 105-117.

Lassez, C. (1987): **Constraint Logic Programming**, BYTE 8/87, S. 171-176.

Leler, Wm (1988): **Constraint Programming Languages, Their Specification and Generation**, Addison-Wesley.

Lieberman, Henry; Lieberman, Carl (1980): **A Session With TINKER: Interleaving Program Testing With Program Design**, Conference Record of the 1980 LISP Conference, Stanford University, S. 90-99.

Myers, Brad A. (1986): **Visual Programming, Programming by Example, and Program Visualization: A Taxonomy**, in: Marilyn Mantei, Peter Orbeton (Hrsg.): Human Factors in Computing Systems—III, Proceedings of the CHI '86 Conference, Boston, MA, S. 59-66.

Ohki, M.; Takeuchi, A.; Furukawa, K. (1986): **A Framework for Interactive Problem Solving based on Interactive Query Revision**, ICOT Technical Report, TR-188.

Piersol, Kurt W. (1986): **Object oriented Spreadsheets: The Analytic Spreadsheet Package**, ACM OOPSLA Proceedings, S. 385-390.

Reddy, Uday S. (1986): **On the Relationship between Logic and Functional Languages**, in: Doug De Groot, Gary Lindstrom (Hrsg.): Logic Programming, Functions, Relations and Equations, Prentice Hall, S. 3-35.

Rubin, Robert V.; Golin, Eric J.; Reiss, Steven P. (1985): **ThinkPad: A Graphical System for Programming by Demonstration**, IEEE Software, 3/85, S. 73-79.

Rockart, John F.; Flannery, Lauren S. (1983): **The Management of End User Computing**, Communications of the ACM, 26(10), 10/83, S. 776-784.

Shu, Nan C. (1985): **FORMAL: A Forms-Oriented, Visual-Directed Application Development System**, IEEE Computer, 8/85, S. 38-49.

Spenke, Michael; Beilken, Christian (1988): **The Implementation of PERPLEX: A Spreadsheet Interface for Logic Programming**, WISDOM Forschungsbericht FB-GMD-88-29, GMD, 11/88, 86 Seiten.

Sussman, Gerald Jay; Steele, Guy Lewis Jr (1980).: **CONSTRAINTS—A Language for Expressing Almost-Hierarchical Descriptions**, Artificial Intelligence 14, S. 1-39.

Zloof, Moshé M.; de Jong, S. Peter (1977): **The System for Business Automation (SBA): Programming Language**, Communications of the ACM 20(6), S. 385-396.

Zloof, Moshé M. (1977): **Query-by-Example: a data base language**, IBM System Journal 16(4), S. 324-343.

Zloof, Moshé M. (1981): **QBE/OBE: A Language for Office and Business Automation**, IEEE Computer 14(5), S. 13-22.

Michael Spenke und Christian Beilken
Gesellschaft für Mathematik und Datenverarbeitung mbH
Postfach 1240
5205 St. Augustin 1
Telefon: 02241 14-2642
email: spenke@gmdzi.uucp bzw. cici@gmdzi.uucp