Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-92-23

Parsen und Generieren der Prolog-artigen Syntax von RELFUN

Michael Herfert

Oktober 1992

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, Philips, SEMA Group Systems, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Intelligent Communication Networks
- Intelligent Cooperative Systems.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Prof. Dr. Gerhard Barth
Director

Parsen und Generieren der Prolog-artigen Syntax von RELFUN

Michael Herfert

DFKI-D-92-23

Diese Arbeit wurde finanziell unterstützt durch das Bundesministerium für Forschung und Technologie (FKZ ITW-8902 C4).

Parsen und Generieren der Prolog–artigen Syntax von Relfun

Michael Herfert

2. Oktober 1992

Zusammenfassung

Ein Parser zum Erkennen und ein Pretty–Printer zum Erzeugen der Prolog–artigen Syntax von Relfun werden beschrieben. Ein weiteres Kapitel erläutert die Einbindung in das Relfun–System und die Auswirkungen auf seine Kommandos.

Projektarbeit

Betreuer: Dr. Harold Boley

Universität Kaiserslautern
Fachbereich Informatik

Inhaltsverzeichnis

Konkrete Syntax		Abstrakte Syntax
Lisp-artig	Prolog-artig	
`_abc`	`Abc`	`(vari abc)`

Tabelle 1: Abstrakte und konkrete Syntax

1 Die zwei Syntaxen von Relfun

Die Sprache Relfun [Bol92] besitzt sowohl funktionale als auch relationale Elemente. Sie verbindet die Vorteile beider Programmierstile. Um sowohl der Lisp-Welt als auch der Prolog-Welt einen einfachen Einstieg anzubieten, besitzt Relfun zwei Syntaxen. Die eine ist an Lisp angelehnt und aufgrund der Implementierung von Relfun in Lisp schon seit Anfang an in Gebrauch. Die zweite orientiert sich an Prolog. Sie ist zwar definiert, war aber bisher noch nicht implementiert. Dieser Text beschäftigt sich mit der Realisierung der Prolog-artigen Syntax.

1.1 Abstrakte und konkrete Syntax

Solange man sich nur als Anwender mit einer Sprache beschäftigt kommt man nur mit der konkreten Syntax in Kontakt. Das ist die Syntax, die im jeweiligen Sprachreport definiert wird. Im Compilerbau gibt es darüberhinaus den Begriff der *abstrakten* Syntax. Darunter versteht man die interne Darstellung von Strukturen, die in der konkreten Syntax definiert wurden. Je nach Implementierung wird die abstrakte Syntax von einem Interpreter ausgeführt oder von einem Compiler weiterverarbeitet.

Als Beispiel betrachte man die Darstellung von Variablen (Tab. 1). In der Lisp-artigen Syntax werden sie durch einen führenden Unterstrich gekennzeichnet. In der Prolog-artigen Syntax beginnen sie mit einem Großbuchstaben (genaueres im nächsten Kapitel). Beide Formen werden intern jedoch als Liste mit führendem `vari`-Tag dargestellt.

Das später beschriebene Modul Lisp2Pro hat die Aufgabe, die abstrakte Syntax in die Prolog-artige zu transformieren. Pro2Lisp realisiert die umgekehrte Richtung. Streng genommen ist die Bezeichnung „Lisp" in den Modulnamen unglücklich gewählt, denn die Lisp-artige Syntax wird in den Transformationen nicht explizit berücksichtigt. „Lisp2Pro" (statt „Abs2Pro") und „Pro2Lisp" (statt „Pro2Abs") sind allerdings vertretbar, wenn man bedenkt, daß Lisp-artige Syntax und abstrakte Syntax in vielen Bereichen identisch sind und von Relfuns Lisp-Implementation implizit ineinander überführt werden.

1.2 Die Prolog-artige Syntax

Abbildung 1 zeigt die formale Definition der Prolog-artigen Syntax. Dabei wurde die EBNF-Notation verwendet, wie sie in [Wir84] definiert ist. Insbesondere sind

```
clause          ::=  hornish_clause | footed_clause .
hornish_clause  ::=  hornish_fact | hornish_rule .
footed_clause   ::=  footed_factr | footed_rule .
hornish_fact    ::=  head "." .
hornish_rule    ::=  head ":-" expr* "." .
footed_factr    ::=  head ":-&" expr "." .
footed_rule     ::=  head ":-" expr* "&" expr "." .
head            ::=  term "(" term* ["|" variable] ")" .

expr*           ::=  [expr+] .
expr+           ::=  expr ["," expr+] .
expr            ::=    expr "(" expr* ["|" expr] ")"
                     | term
                     | primitive .
term*           ::=  [term+] .
term+           ::=  term ["," term+] .
term            ::=  constant | variable | list | structure .
constant        ::=  symbol | string | integer | real .
variable        ::=  big_char {char} | "_" | "_" (small_char | digit ) {char} .
list            ::=  "[" term* ["|" term] "]" .
structure       ::=  term "[" term* ["|" term] "]" .
primitive       ::=  term "is" expr | builtin "(" expr* [ "|" expr ] ")" .
builtin         ::=  lisp-function | lisp-predicate | lisp-extra | log-extra .

lisp-function   ::=  "+" | "-" | "*" ...   .
lisp-predicate  ::=  "atom" | ...   .
lisp-extra      ::=  "load" | ...   .
log-extra       ::=  "call" | "once" | "naf" | "tupof" .
symbol          ::=  small_char {char} .
string          ::=  "" {char} "" .
real            ::=  integer "." digit+ ["E" integer ] .
integer         ::=  ["+"| "-"] digit+ .
char            ::=  small_char | big_char | digit | "_" .
small_char      ::=  "a" | "b" | ...| "z" | "+" | "-" | "*" | "/" | "<" | "=" | ">" .
big_char        ::=  "A" | "B" | ...| "Z" .
digit           ::=  "0" | "1" | ...| "9" .
digit+          ::=  digit {digit} .
```

Abbildung 1: **Die Prolog–artige Syntax**

"+" und "*" keine Metazeichen. Als Kommentarzeichen dient „%". Es kennzeichnet den Rest der Zeile als Kommentar. Lisps Semikolon ist in dieser Funktion ungünstig, da es in Standard-Prolog die Disjunktion ausdrückt.

Vergleicht man die Prolog-artige Syntax von Relfun mit der Syntax von Standard-Prolog, so fällt auf, daß in Relfun einige Elemente, wie etwa das Ampersand „&" zur Kennzeichnung eines Rückgabewertes, hinzugekommen sind. Strukturen dagegen gibt es in beiden Sprachen, dennoch werden sie unterschiedlich formuliert. Während man in Prolog runde Klammern benutzt (z.B. `book(bible)`), verlangt Relfun nach eckigen (z.B. `book[bible]`). Diese Unterscheidung mag Prolog-Umsteigern die Einarbeitung in Relfun erschweren, ist jedoch nicht zu vermeiden, da Relfun runde Klammern für aktive Aufrufe reserviert.

Zu erwähnen ist weiterhin, daß arithmetische Operatoren in Relfun als Präfix geschrieben werden (z.B. `+(3, 4)`). Es ist dem Benutzer nicht möglich, eigene Infix-Operatoren zu definieren.

Für Relfun-Kenner ist sicherlich der Vergleich der Prolog-artigen mit der Lisp-artigen Syntax aufschlußreich. Tab. 2 stellt die charakteristischen Konstrukte gegenüber. Man beachte, daß die P-Syntax zwei Notationen für Variablen vorsieht, die intern nicht unterschieden werden.

Den Abschluß dieses Kapitels soll das Fakultätsbeispiel in drei Formulierungen bilden, wobei für die beiden Relfun-Formulierungen eine Funktionsdefinition gewählt wurde, während Prolog natürlich eine Relationsdefinition benutzt:

Relfun in Lisp-Stil

```
(ft (fac 0) 1)
(ft (fac _n)
    (> _n 0)
    (* _n
       (fac (- _n 1))))
```

Relfun in Prolog-Stil

```
fac(0):-& 1.
fac(N):-
  >(N, 0) &
  *(N,
    fac(-(N, 1))).
```

Prolog

```
fac(0,1).
fac(N, R):-
  N > 0,
  M is N-1,
  fac(M, F),
  R is N * F.
```

Im Anhang A wird beschrieben, wie man leicht weitere Beispiele zur Prolog-artigen Syntax gewinnen kann.

2 Lisp2Pro: Pretty-Printer für die Prolog-artige Syntax

Aufgabe des Moduls Lisp2Pro ist es, einen Ausdruck, der in der abstrakten Syntax vorliegt, in die Prolog-artige Syntax zu transformieren.

2.1 Die Schnittstelle

Die Beschreibung der exportierten Funktion ist in der Darstellungsform angelehnt an den Common-Lisp-Sprachreport:

Konstrukt	Konkrete Syntax		Abstrakte Syntax
	Prolog-artig	Lisp-artig	
Konstanten	`john`	`john`	`john`
Variablen	`Xyz` `_xyz`	`_xyz` `_xyz`	`(vari xyz)` `(vari xyz)`
Listen	`[a, b | X]`	`'(tup a b | _x)`	`(inst` `(tup a b | (vari x)))`
Strukturen	`book[bible]`	`'(book bible)`	`(inst (book bible))`
Aufrufe	`father(sue)`	`(father sue)`	`(father sue)`
Arithmetik	`+(3, 4)`	`(+ 3 4)`	`(+ 3 4)`
is	`X is 3`	`(is _x 3)`	`(is (vari x) 3)`
Konjunktionen	`X is 1,` `f(X)`	`(is _x 1)` `(f _x)`	`(is (vari x) 1)` `(f (vari x))`
Horn-Fakten	`on(hat, tom).`	`(hn (on hat tom))`	
Horn-Regeln	`p(X) :- q(X),` `r().`	`(hn (p _x)` `(q _x) (r))`	
Footed-Fakten	`on(tom) :-& hat.`	`(ft (on tom) hat)`	
Footed-Regeln	`p(X) :- q(X) &` `r().`	`(ft (p _x)` `(q _x) (r))`	
Kommentare	`X is 1 % Kom...`	`(is _x 1) ; Kom...`	`(is (vari x) 1)`

Tabelle 2: **Vergleich der Syntaxen** *Wenn in der rechten Spalte ein leeres Feld auftritt, unterscheiden sich abstrakte Syntax und Lisp-artige Syntax nur in der Darstellung von Variablen.*

`pro-print` *expr-or-clause* [Function]

Ihr Wert ist unbestimmt. Als Nebeneffekt wird jedoch der Ausdruck in P-Syntax ausgegeben. In Relfun übliche Umleitungen der Ausgabe, z.B. die parallele Erzeugung eines Scriptfiles, werden dabei korrekt behandelt. Die Zeilenbreite wird aus der Relfun-Variablen `*relfun-print-width*` gelesen.

2.2 Ausgabeformate

Lisp2Pro verwendet drei Ausgabeformate:

(1) linear

$functor(argument_1, \ldots, argument_n)$

(2) senkrecht

$functor(argument_1,$
$\vdots$
$argument_n)$

(3) senkrecht und eingerückt

$functor($
$argument_1,$
$\vdots$
$argument_n)$

Welches Format verwendet wird, hängt vom auszugebenden Ausdruck ab. Das primäre Ziel bei der Wahl des Formats is die Erzeugung eines leicht lesbaren Ausdrucks. Insbesondere wird gegebenenfalls auch dann eine neue Zeile angebrochen, wenn die aktuelle noch ausreichen würde. Zur Veranschaulichung betrachte man die folgenden Beispiele:

(1) `+(1, 2, 3, 4, 5, 6)`
Die Funktion `+` hat hier zwar sechs Argumente, jedoch bietet auch das lineare Format eine gute Lesbarkeit. Eine Aufspaltung in Einzelzeilen würde den Leser in den meisten Fällen stören, da durch die mangelhafte Platzausnutzung nur wenige Informationen auf den Schirm passen.

(2) `append([a, b, c, [1, 2, 3]], [x, y, [z1, z2, z3], z])`
Hier hat `append` zwar nur zwei Argumente, und der gesamte Ausdruck paßt auf eine Zeile, jedoch erschweren verschachtelte Listen die Lesbarkeit. Hier sollte also das senkrechte Format gewählt werden:

```
append([a,
        b,
        c,
        [1, 2, 3]],
       [x,
        y,
        [z1, z2, z3],
        z])
```

Die Entscheidung über das Format läuft auf ein etwas diffiziles Problem hinaus. Zum einen benötigt man eine gewisse Erfahrung mit typischen Relfun-Ausgaben, zum anderen geht auch der persönliche Geschmack ein. Wer hier etwas ändern möchte, sei auf die Funktionen **pro-print-arg-list** und **get-arg-list-info** des Moduls Lisp2Pro verwiesen.

2.3 Der Maßbaum

Dieser Abschnitt behandelt ein wesentliches Merkmal der Implementierung.

Eine Frage, die sich im Laufe der Ausgabe wiederholt stellt, lautet: Paßt der Rest des Ausdrucks noch auf dieselbe Zeile oder muß ein Umbruch erfolgen? Um diese Frage für einen Ausdruck der Form $functor(arg_1, \ldots, arg_n)$ zu beantworten, könnte man zunächst $arg_1, \ldots, arg_n$ in P-Syntax in einem String zwischenspeichern. Stellt sich nun heraus, daß alles auf eine Zeile paßt, ergeben sich keine Nachteile. Ist dies jedoch nicht der Fall, so werden $arg_1, \ldots, arg_n$ auf separate Zeilen verteilt. Für jedes arg_i stellt sich nun wieder die gleiche Frage. Daher muß jedes nun zum zweiten Mal in einen String transformiert werden. Je schmaler die Zeilen, je tiefer die Verschachtelung, desto größer wird die Ineffizienz des Verfahrens.

Um das zu vermeiden, werden zwei Läufe über den auszugebenden Ausdruck gemacht. Der erste ergänzt Längeninformationen und wandelt alle atomaren Bestandteile (Zahlen, Symbole) in Strings um. Als Ergebnis entsteht ein Maßbaum (*measure-tree*). Der zweite Lauf entscheidet anhand dieser Informationen über das Format und führt die Ausgabe durch.

Ein Knoten des Baumes ist dabei durch folgenden Datentyp definiert:

```
(defstruct pro-expr
  type  ; dom(type) = {round-application, sqr-application,
        ;              argument-list, arg-list-tail, is,
        ;              variable-binding, atomic}
  len   ; dom(len) = integer
  value ; The value depends on the type-field
  )
```

Zwei einfache Beispiele sollen den Maßbaum verdeutlichen. Zunächst sei eine Konstante auszugeben:

```
> (construct-measure-tree 'son nil)
#S(pro-expr type atomic len 3 value "son")
```

Weil verschachtelte Strukturen durch den Lisp-Pretty-Printer nur schwer lesbar ausgegeben werden, enthält Lisp2Pro zu Debugging-Zwecken die Funktion **pps** (*pretty-print-structure*), die in diesem Fall die folgende Ausgabe erzeugt:

```
> (pps (construct-measure-tree 'son nil))
  (*pro-expr*)
Type: ATOMIC
Len:  3
Val:  son
```

Der Parameter `nil` in `construct-measure-tree` steuert die Ausgabe verschachtelter Strukturen und ist hier nicht von Interesse.

Nun soll ein Funktionsaufruf ausgegeben werden.

```
> (pps (construct-measure-tree '(likes eve adam) nil))
  (*pro-expr*)
Type: ROUND-APPLICATION
Len:  16
Val:  [
        (*pro-expr*)
      Type: ATOMIC
      Len:  5
      Val:  likes
        (*pro-expr*)
      Type: ARGUMENT-LIST
      Len:  11
      Val:  [
              (*pro-expr*)
            Type: ATOMIC
            Len:  3
            Val:  eve
              (*pro-expr*)
            Type: ATOMIC
            Len:  4
            Val:  adam
           ]
    ]
```

Wenn `type` mit `round-application` besetzt ist, enthält `value` eine Liste aus Funktor und Argumentliste. Analog ist `value` mit einer Liste aus Argumenten besetzt, falls in `type` `argument-list` steht. Die Beziehung zwischen diesen beiden Feldern ist im Quelltext ausführlich dokumentiert.

Die Länge 11 der Argumentliste (`eve, adam`) errechnet sich wie folgt:

$$11 = len("eve") + len("adam") + len("()") + len(",\sqcup")$$

Der Gesamtausdruck `likes(eve, adam)` hat damit die Länge 16.

3 Pro2Lisp: Reader für die Prolog–artige Syntax

Das Modul hat die Aufgabe, einen Ausdruck, der in Prolog–artiger Syntax als Folge von ASCII–Zeichen vorliegt, in die abstrakte Syntax zu transformieren.

3.1 Die Schnittstelle

Es werden vier Prozeduren exportiert.

`pro-read-data-base` *filename* [Function]

Nimmt an, daß die durch *filename* bezeichnete Datei Klauseln enthält. Liefert eine Liste der Klauseln in abstrakter Syntax. Sollte nur eine Klausel vorhanden sein, wird eine einelementige Liste zurückgegeben. Eine leere Datei erzeugt eine Fehlermeldung.

`pro-read-goal` *string* [Function]

Liest ein Ziel aus *string*. Das Ziel darf nicht mit einem Punkt enden. Liefert eine Liste, deren Elemente die Faktoren des Ziels sind.

`pro-read-clause` *string* [Function]

Liest eine Klausel aus *string*. Liefert die äquivalente Form in abstrakter Syntax. Das Ergebnis ist eine Liste, deren erstes Element ein Tag (`hn` oder `ft`) ist.

`pro-split-input` *string* [Function]

Liefert ein Paar. Über den Aufbau entscheidet das erste Token von *string*: Steht dort ein Symbol, so wird es an die erste Position des Paares geschrieben. Das zweite Element besteht aus dem Rest der Eingabe als String, beginnend beim ersten non–whitespace nach dem Symbol. Andernfalls wird die erste Position mit `nil` und die zweite mit dem leeren String besetzt. Die Funktion wird für die Behandlung von Relfun–Kommandos gebraucht.

Tritt in der Eingabe ein Fehler auf, so wird eine Fehlermeldung ausgegeben und der Wert `nil` zurückgeliefert.

3.2 Transformation der Grammatik

3.2.1 Anforderungen

Um den später beschriebenen Parser möglichst einfach und effizient zu halten, müssen Grammatikregeln, die eine Alternative der Form

$$A = \beta_1 | \ldots | \beta_n$$

ausdrücken, zwei Bedingungen erfüllen [Wir84]:

(1) Die First-Mengen müssen paarweise disjunkt sein:

$$first(\beta_i) \cap first(\beta_j) = \emptyset \qquad \forall i \neq j$$

Unter $first(\beta)$ versteht man dabei die Menge aller Terminalsymbole, mit denen eine Ableitung aus β beginnen kann.

(2) Nur eine der Alternativen darf mit einem Nichtterminalsymbol beginnen.

Die erste Regel erlaubt einen Verzicht auf Backtracking und die Begrenzung der Vorausschau des Parsers auf ein Symbol. Die zweite bewirkt, daß in der transformierten Grammatik keine First-Mengen berechnet werden müssen. Die entsprechenden Informationen sind direkt aus den Regeln ablesbar.

3.2.2 Regeln

Um die Grammatik gemäß den Anforderungen zu transformieren, benötigt man die nachfolgend beschriebenen Regeln [ASU88]. Dabei stehen Großbuchstaben für Nichtterminalsymbole, griechische Zeichen für beliebige EBNF-Ausdrücke.

(1) Einsetzen.
Eine Regelmenge der Form

$$\begin{aligned} A &= B_1\alpha_1 | B_2\alpha_2 \\ B_1 &= \beta_1 \\ B_2 &= \beta_2 \end{aligned}$$

transformiert man durch Einsetzen zu:

$$A = \beta_1\alpha_1 | \beta_2\alpha_2$$

(2) Faktorisieren.

$$A = \beta\alpha_1 | \beta\alpha_2$$

geht über in:

$$A = \beta(\alpha_1 | \alpha_2)$$

Man beachte, daß die runden Klammern zur EBNF-Notation gehören.

(3) Auflösen.
Die Linksrekursion

$$A = \beta_1 | A\beta_2$$

wird in eine iterative Form umgewandelt:

$$A = \beta_1\{\beta_2\}$$

3.2.3 Transformationen

Zu transformieren sind drei Regeln der Grammatik in Abb. 1:

clause = *hornish-clause* | *footed-clause*
= *hornish-fact* | *hornish-rule* | *footed-rule* | *footed-factr*
= *head* "."
| *head* ":-" *expr** "."
| *head* ":-&" *expr* "."
| *head* ":-" *expr** "&" *expr* "."
= *head*("."
| ":-&" *expr* "."
| ":-" *expr** ["&" *expr*] ".")

term = *constant* | *variable* | *list* | *structure*
= *constant*
| *variable*
| *list*
| *term list*
= (*constant* | *variable* | *list*) {*list*}

expr = *expr* "(" *expr** ["|" *expr*] ")"
| *term*
| *primitive*
= *expr* "(" *expr** ["|" *expr*] ")"
| *term*
| *term* "is" *expr*
| *builtin* "(" *expr** ["|" *expr*] ")"
= *expr* "(" *expr** ["|" *expr*] ")"
| *term* ["is" *expr*]
| *builtin* "(" *expr** ["|" *expr*] ")"
= (*term* ["is" *expr*]
| *builtin* "(" *expr** ["|" *expr*] ")")
{ "(" *expr** ["|" *expr*] ")" }

3.3 Scanner und Parser

Betrachtet man die Grammatikregel

constant = *symbol* | *string* | *integer* | *real*

so mag es verwundern, daß sie nicht transformiert wurde, obwohl jede Alternative mit einem Nichtterminal beginnt. Der Grund liegt darin, daß der Parser den Quelltext nicht zeichenweise liest, sondern in Form von lexikalisch zusammenhängenden Einheiten (*Tokens*). Ein Token besteht etwa aus allen Ziffern einer Zahl oder aus allen Zeichen eines Strings. Die Arbeit, Tokens zu bilden, ist Aufgabe des Scanners. Der Scanner liefert dazu bei jedem Aufruf ein Datum folgenden Typs:

```
(defstruct token
  type     ; dom = {number, constant, variable,
           ;        round-left, round-right, sqr-left, square-right
           ;        point, comma, ampersand,
           ;        implies, implies-amper,
           ;        is, empty}
  value    ; used with number, constant, variable
  x-pos    ; to report an error-position
  y-pos
)
```

Durch die Transformation der Grammatik wird der Parser sehr einfach. Er ist nach dem Prinzip des rekursiven Abstiegs konstruiert. Für jedes Nichtterminal wird eine Funktion definiert, deren Aufgabe es ist, die entsprechende Regel zu erkennen und Aktionen einzuleiten, um die abstrakte Syntax zu generieren. Diese Funktion wird nur aufgerufen, wenn feststeht, daß die korrespondierende Regel anwendbar ist.

Normalerweise gestaltet sich die Zusammenarbeit von Scanner und Parser so:

```
(defun scanner ()
    ;
    ; Lispcode zum Erkennen von Tokens
    ;
)

(defun parse-term ()
    ;
    ; Lispcode zur Verarbeitung der Regel term.
    ; Dazu wird der Scanner aufgerufen,
    ; um Tokens aus dem Eingabetext zu lesen.
    ;
)
```

Der Nachteil dieser Lösung ist die große Anzahl globaler Variablen. So benötigt man mindestens eine Variable `last-token`, die das zuletzt gelesene Token aufnimmt. Weiterhin werden Variablen für die aktuelle Zeilen- und Spaltennummer gebraucht.

Dieser Nachteil läßt sich vermeiden, indem man `last-token` und Verwandte zu privaten Variablen des Scanners macht. Aus der Sicht der objektorientierten Programmierung wird der Scanner dadurch zu einem Objekt. Man kann ihm nun eine

Nachricht (z.B. `next-token`) senden, um eine seiner Dienstleistungen auszuwählen. Implementierungstechnisch erreicht man dies dadurch, daß für jeden zu lesenden Quelltext eine Funktion `scanner` zur Laufzeit des Programms erzeugt wird [ASS87]:

```
(defun gen-scanner (input-stream)
  (let ((last-token init-value)
        (x-pos      init-value)
    ;
    ; weitere Variablen, die vormals global waren
    ;
      )
   #'(lambda (message)
       (case message
         (last-token last-token)
         (next-token ...   ) ;Code, um das nächste Token zu liefern.
         ;
         ; weitere Nachrichten
         ;
       ))))

(defun parse-term (scanner)
  ;  Lispcode, um term zu verarbeiten.
  ;  Dabei wird der Scanner in der Form
  ;  (funcall scanner 'next-token)
  ;  aufgerufen
)
```

Allen Parserfunktionen wird also der Scanner als Argument übergeben. Der objekt-orientierte Eindruck wird syntaktisch leider durch die Verwendung von `funcall` etwas getrübt. [1]

4 Einbindung in das Relfun-System

Um die neue Syntax in das Relfun-System einzubinden, sind zunächst die Moduln Lisp2Pro und Pro2Lisp hinzuzufügen. Außerdem werden System-Funktionen modifiziert sowie einige neue ergänzt. Wesentlich ist die Einführung der globalen Variablen `*style*` mit dem Wertebereich {`lisp`, `prolog`}. Sie entscheidet über die Syntax bei Ein-/Ausgabeoperationen.

[1] Common-Lisp schreibt die Benutzung von `funcall` vor. Der Grund ist vermutlich darin zu sehen, daß das Common-Lisp-Komitee die Doppelwertigkeit eines Symbols aufrechterhalten wollte. Ein Symbol kann zu einer Zeit sowohl einen mittels `defun` zugewiesenen Wert haben, als auch einen, der mit `setq` eingerichtet wurde. Im Scheme-Dialekt von Lisp wurde diese Ambivalenz fallengelassen. Das führt zu einer klaren Semantik und erlaubt den Verzicht auf `funcall`.

4.1 Ein-/Ausgabefunktionen

Beabsichtigt das Relfun-System eine Ausgabe, so wird `pp` oder `pp-clause` aufgerufen. Dort wird nun die Variable `*style*` abgefragt und gegebenenfalls `pro-print` aus dem Modul Lisp2Pro aufgerufen.

Im Falle einer Eingabe ruft Relfun die Prozedur `readl` auf. Die Eingabe wird gelesen und in Abhängigkeit von `*style*` in die abstrakte Syntax transformiert. Der Aufrufer von `readl` merkt also gar nicht, daß möglicherweise die Prolog-artige Syntax aktiviert ist.

Ein ähnlicher Eingriff ist in `read-db-from-file` nötig. Schließlich müssen noch einige syntax-spezifische System-Kommandos (z.B. `a0`) abgefangen werden. Diese Arbeit erledigt `handle-rfi-cmd`.

4.2 System-Kommandos

Es gibt ein neues System-Kommando:

style:
Format: `style` *io-style*
Optionen: keine.
Effekt: Legt die Syntax für Ein-/Ausgabeoperationen fest. Erlaubte Werte für *io-style* sind `lisp` und `prolog`. Der aktuelle Stil wird im Prompt angezeigt.

Darüberhinaus werden einige bestehende Kommandos beeinflußt, falls die P-Syntax aktiviert ist.

- `a0`, `az`, `rx`
  erwarten ihr Argument in Prolog-artiger Syntax.
- `consult`, `replace` und `tell`
  Verwenden als Standard-Filename-Extension "`.rfp`".

Literatur

[ASS87] Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1987.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau, Teil I.* Addison-Wesley, 1988.

[Bol92] Harold Boley. Extended logic-plus-functional programming. In *Workshop on Extensions of Logic Programming, ELP '91, Stockholm 1991*, LNAI. Springer, 1992.

[Wir84] Niklaus Wirth. *Compilerbau.* Teubner, 1984.

A Beispieldialog

Das folgende Scriptfile zeigt eine Beispielsitzung. Man beachte, daß man mittels

```
style lisp
consult filename-without-proper-extension        ; consult ergänzt ".rf"
style prolog
tell filename-without-proper-extension           % tell ergänzt ".rfp"
```

eine Datei von der Lisp–artigen in die Prolog–artige Syntax transformieren kann. Dabei gehen Kommentare verloren.

```
rfi-l> consult higher-order
; Reading file "higher-order.rf"
rfi-l> listing
(ft (twice1 _f)
    '(compose _f _f) )
(ft ((twice2 _f) _a)
    ('(compose _f _f) _a) )
(ft ((compose _f _g) | _a)
    (_f (_g | _a)) )
(ft (adofad1)
    (twice1 1+) )
(ft (adofad2)
    '(twice2 1+) )
(ft (add2 _x)
    ((twice1 1+) _x) )
(ft (goal1)
    ('(compose 1+ *) 2 3) )
(ft (goal2)
    (twice1 1+) )
(ft (goal3)
    ((twice1 1+) 0) )
(ft (goal4)
    ((adofad1) 0) )
(ft (goal5)
    (add2 0) )
(ft (goal6)
    (twice1 twice1) )
(ft (goal7)
    (((twice1 twice1) 1+) 0) )
rfi-l> style prolog
rfi-p> listing
twice1(F) :-& compose[F, F].
```

```
twice2[F](A) :-& compose[F, F](A).
compose[F, G](| A) :-& F(G(| A)).
adofad1() :-& twice1(1+).
adofad2() :-& twice2[1+].
add2(X) :-& twice1(1+)(X).
goal1() :-& compose[1+, *](2, 3).
goal2() :-& twice1(1+).
goal3() :-& twice1(1+)(0).
goal4() :-& adofad1()(0).
goal5() :-& add2(0).
goal6() :-& twice1(twice1).
goal7() :-& twice1(twice1)(1+)(0).
rfi-p> tell higher-order  % erzeugt die Datei higher-order.rfp
rfi-p>goal1()
7
rfi-p> goal2()
compose[1+, 1+]
rfi-p> goal3()
2
rfi-p> goal4()
2
rfi-p> goal5()
2
rfi-p> goal6()
compose[twice1, twice1]
rfi-p> goal7()
4
```

B Beispiele in zwei Syntaxen

Zur Veranschaulichung folgt nun eine Reihe von Beispielprogrammen. Jedes wird in L-Syntax und in P-Syntax aufgeführt. Die Originale in Lisp–artiger Syntax wurden zum Testen der Beispiele in [Bol92] erstellt.

Im Prinzip handelt es sich um ein Script des folgenden Batchfiles. Lediglich die Zwischenüberschriften wurden von Hand ergänzt.

```
style lisp
destroy
consult "/home/rfm/RFM/extensions/attval.rf"
listing
style prolog
listing

style lisp
destroy
consult "/home/rfm/RFM/extensions/genints.rf"
listing
style prolog
listing

style lisp
destroy
consult "/home/rfm/RFM/extensions/inv.rf"
listing
style prolog
listing

style lisp
destroy
consult "/home/rfm/RFM/extensions/revise.rf"
listing
style prolog
listing

style lisp
destroy
consult "/home/rfm/RFM/extensions/serialisef.rf"
listing
style prolog
listing

style lisp
destroy
```

```
consult "/home/rfm/RFM/extensions/serialiser.rf"
listing
style prolog
listing

style lisp
destroy
consult "/home/rfm/RFM/extensions/signum.rf"
listing
style prolog
listing
```

B.1 attval

```
rfi-l>
style lisp
rfi-l> destroy
rfi-l> consult "/home/rfm/RFM/extensions/attval.rf"
; Reading file "/home/rfm/RFM/extensions/attval.rf" ..
rfi-l> listing
(ft (attval _obj _valfilter)
    (tup _f (_valfilter (_f _obj))) )
(ft (numfilter _x)
    (numberp _x)
    _x )
(ft (area china)
    3380 )
(ft (pop china)
    825 )
rfi-l> style prolog
rfi-p> listing
attval(Obj, Valfilter) :-& tup(F, Valfilter(F(Obj))).
numfilter(X) :- numberp(X) & X.
area(china) :-& 3380.
pop(china) :-& 825.
```

B.2 genints

```
rfi-p>
style lisp
rfi-l> destroy
rfi-l> consult "/home/rfm/RFM/extensions/genints.rf"
; Reading file "/home/rfm/RFM/extensions/genints.rf" .
rfi-l> listing
(ft (genints)
```

```
    0 )
(ft (genints)
    (genints '(_sign 0)) )
(ft (genints (_sign _n))
    (is _sign s)
    '(_sign _n) )
(ft (genints (_sign _n))
    (is _sign p)
    '(_sign _n) )
(ft (genints (_sign _n))
    (genints '(_sign (_sign _n))) )
rfi-l> style prolog
rfi-p> listing
genints() :-& 0.
genints() :-& genints(Sign[0]).
genints(Sign[N]) :- Sign is s & Sign[N].
genints(Sign[N]) :- Sign is p & Sign[N].
genints(Sign[N]) :-& genints(Sign[Sign[N]]).
```

B.3 inv

```
rfi-p>
style lisp
rfi-l> destroy
rfi-l> consult "/home/rfm/RFM/extensions/inv.rf"
; Reading file "/home/rfm/RFM/extensions/inv.rf"
rfi-l> listing
(ft ((inv _f) _v)
    (is _v (_f | _x))
    _x )
(ft (area india)
    1139 )
(ft (f _i _j | _r)
    _j )
rfi-l> style prolog
rfi-p> listing
inv[F](V) :- V is F(| X) & X.
area(india) :-& 1139.
f(I, J | R) :-& J.
```

B.4 revise

```
rfi-p>
style lisp
rfi-l> destroy
```

```
rfi-l> consult "/home/rfm/RFM/extensions/revise.rf"
; Reading file "/home/rfm/RFM/extensions/revise.rf" ..
rfi-l> listing
(ft (revise _f _n (tup))
    (tup) )
(ft (revise _f 1 (tup _h | _t))
    (tup (_f _h) | _t) )
(ft (revise _f _n (tup _h | _t))
    (tup _h | (revise _f (1- _n) _t)) )
rfi-l> style prolog
rfi-p> listing
revise(F, N, []) :-& tup().
revise(F, 1, [H | T]) :-& tup(F(H) | T).
revise(F, N, [H | T]) :-& tup(H | revise(F, 1-(N), T)).
```

B.5 serialisef

```
rfi-p>
style lisp
rfi-l> destroy
rfi-l> consult "/home/rfm/RFM/extensions/serialisef.rf"
; Reading file "/home/rfm/RFM/extensions/serialisef.rf" ..
rfi-l> listing
(ft (serialise _l)
    (numbered (arrange (pairlists _l _r)) 1)
    _r )
(ft (pairlists (tup _x | _l) (tup _y | _r))
    (tup
     '(pair _x _y)
     |
     (pairlists _l _r) ) )
(ft (pairlists (tup) (tup))
    (tup) )
(ft (arrange (tup _x | _l))
    (partition _l _x _l1 _l2)
    (is _t1 (arrange _l1))
    (is _t2 (arrange _l2))
    '(tree _t1 _x _t2) )
(ft (arrange (tup))
    void )
(hn (partition (tup _x | _l) _x _l1 _l2)
    (partition _l _x _l1 _l2) )
(hn (partition
     (tup _x | _l)
     _y
```

```
     (tup _x | _l1)
     _l2 )
    (before _x _y)
    (partition _l _y _l1 _l2) )
(hn (partition
     (tup _x | _l)
     _y
     _l1
     (tup _x | _l2) )
    (before _y _x)
    (partition _l _y _l1 _l2) )
(hn (partition (tup) _y (tup) (tup)))
(hn (before (pair _x1 _y1) (pair _x2 _y2))
    (string< _x1 _x2) )
(ft (numbered
     (tree _t1 (pair _x _n1) _t2)
     _n0 )
    (numbered _t2 (1+ (is _n1 (numbered _t1 _n0)))) )
(ft (numbered void _n)
    _n )
rfi-l> style prolog
rfi-p> listing
serialise(L) :- numbered(arrange(pairlists(L, R)), 1) & R.
pairlists([X | L], [Y | R]) :-& tup(pair[X, Y] | pairlists(L, R)).
pairlists([], []) :-& tup().
arrange([X | L]) :- partition(L, X, L1, L2),
                    T1 is arrange(L1),
                    T2 is arrange(L2) &
                    tree[T1, X, T2].
arrange([]) :-& void.
partition([X | L], X, L1, L2) :- partition(L, X, L1, L2).
partition([X | L],
          Y,
          [X | L1],
          L2) :- before(X, Y), partition(L, Y, L1, L2).
partition([X | L],
          Y,
          L1,
          [X | L2]) :- before(Y, X), partition(L, Y, L1, L2).
partition([], Y, [], []).
before(pair[X1, Y1], pair[X2, Y2]) :- string<(X1, X2).
numbered(tree[T1, pair[X, N1], T2], N0) :-&
  numbered(T2, 1+(N1 is numbered(T1, N0))).
numbered(void, N) :-& N.
```

B.6 serialiser

```
rfi-p>
style lisp
rfi-l> destroy
rfi-l> consult "/home/rfm/RFM/extensions/serialiser.rf"
; Reading file "/home/rfm/RFM/extensions/serialiser.rf" ..
rfi-l> listing
(hn (serialise _l _r)
    (pairlists _l _r _a)
    (arrange _a _t)
    (numbered _t 1 _n) )
(hn (pairlists
     (tup _x | _l)
     (tup _y | _r)
     (tup (pair _x _y) | _a) )
    (pairlists _l _r _a) )
(hn (pairlists (tup) (tup) (tup)))
(hn (arrange
     (tup _x | _l)
     (tree _t1 _x _t2) )
    (partition _l _x _l1 _l2)
    (arrange _l1 _t1)
    (arrange _l2 _t2) )
(hn (arrange (tup) void))
(hn (partition (tup _x | _l) _x _l1 _l2)
    (partition _l _x _l1 _l2) )
(hn (partition
     (tup _x | _l)
     _y
     (tup _x | _l1)
     _l2 )
    (before _x _y)
    (partition _l _y _l1 _l2) )
(hn (partition
     (tup _x | _l)
     _y
     _l1
     (tup _x | _l2) )
    (before _y _x)
    (partition _l _y _l1 _l2) )
(hn (partition (tup) _y (tup) (tup)))
(hn (before (pair _x1 _y1) (pair _x2 _y2))
    (string< _x1 _x2) )
(hn (numbered
```

```
     (tree _t1 (pair _x _n1) _t2)
     _n0
     _n )
    (numbered _t1 _n0 _n1)
    (is _n2 (+ _n1 1))
    (numbered _t2 _n2 _n) )
(hn (numbered void _n _n))
rfi-l> style prolog
rfi-p> listing
serialise(L, R) :- pairlists(L, R, A), arrange(A, T), numbered(T, 1, N).
pairlists([X | L], [Y | R], [pair[X, Y] | A]) :- pairlists(L, R, A).
pairlists([], [], []).
arrange([X | L], tree[T1, X, T2]) :- partition(L, X, L1, L2),
                                     arrange(L1, T1),
                                     arrange(L2, T2).
arrange([], void).
partition([X | L], X, L1, L2) :- partition(L, X, L1, L2).
partition([X | L],
          Y,
          [X | L1],
          L2) :- before(X, Y), partition(L, Y, L1, L2).
partition([X | L],
          Y,
          L1,
          [X | L2]) :- before(Y, X), partition(L, Y, L1, L2).
partition([], Y, [], []).
before(pair[X1, Y1], pair[X2, Y2]) :- string<(X1, X2).
numbered(tree[T1, pair[X, N1], T2], N0, N) :- numbered(T1, N0, N1),
                                              N2 is +(N1, 1),
                                              numbered(T2, N2, N).
numbered(void, N, N).
```

B.7 signum

```
rfi-p>
style lisp
rfi-l> destroy
rfi-l> consult "/home/rfm/RFM/extensions/signum.rf"
; Reading file "/home/rfm/RFM/extensions/signum.rf" ..
rfi-l> listing
(ft (signum _x)
    (< _x 0)
    -1 )
(ft (signum 0)
    0 )
```

```
(ft (signum _x)
    (> _x 0)
    1 )
rfi-l> style prolog
rfi-p> listing
signum(X) :- <(X, 0) & -1.
signum(0) :-& 0.
signum(X) :- >(X, 0) & 1.
```

C Listing: Lisp2Pro

```
;-----------------------------------------------------------------------
; Module Lisp2Pro.Lsp:      Transforms an expression given in the Lisp-like
;                           syntax into the Prolog-like syntax.
;
;
; Michael Herfert.
; 2/92: First Version
; 7/92: Fixed problem with " |"
;
;
; Exported Item:
;
;   pro-print      expr-or-clause-in-lisp-syntax                    [Function]
;
;       Prints the argument in a pretty-print format in Prolog-like syntax.
;

(defun inst-tup-t (x)
  (and (inst-t x)
       (tup-t (second x)) ))

(defun inst-vari-t (x)
  (and (inst-t x)
       (vari-t (second x)) ))

(defun variable-binding-t (x)
  (and (consp x)
       (= 3 (length x))
       ;(vari-t (first x))
       (eq '= (second x)) ))

(defconstant the-non-printing-char (code-char 0))

;; The following constants are parameters of the pretty printer.
;; They can be changed here without side effects:

;; Small number ==> use many lines when printing:
(defconstant max-len-of-a-simple-arg 7)

;; Small number ==> use many lines when printing:
(defconstant max-args-per-line 3)

(defun internal-error (&rest msgs)
  (mapcar #'rf-print msgs)
  (error "Internal-Error.") )
```

```
; The type pro-expr describes the Prolog-like representation of a RelFun-
; expression.
; The field len describes the space to print the whole expression
; including commas, spaces and parentheses resp. brackets.
; The contents of value depends on the value of type:
; type = atomic      (that means constant or variable)
;    value: the string-form of the atom.
; type = argument-list
;    value: a list of the arguments.
; type = arg-list-tail (e.g. " | (vari x)")
;    value: a pro-expr, normaly containing a variable
; type = round-application (e.g. f(x, y))
;    value: a list.        First element:      the functor,
;                          Second element:     the argument-list.
; type = sqr-application (e.g. g[a,b])
;    value: a list.        First element:      the functor or nil if it is a tupel.
;                          Second element:     the argument-list.
; type = is (e.g. X is 5)
;    value: a list.        First element:      left side of is
;                          Second element:     right side of is
; type = variable-binding (e.g. (X = 10))
;    value: a list.        First element:      the variable
;                          Second element:     the binding-value

(defstruct pro-expr
  type   ; dom(type) =    {round-application, sqr-application,
         ;                 argument-list, arg-list-tail, is,
         ;                 variable-binding, atomic}
  len    ; dom(len) = integer
  value  ; dom(value) = STRING u pro-expr.
  )


(defun newline-indent (indent)
  (rf-terpri)
  (dotimes (i indent)
           (rf-princ-like-lisp " ") ))


(defun pps (s &optional (indent 0) &key rem)
  ;; pretty-print-structure useful for debugging.
  (if rem (progn (rf-terpri) (rf-princ-like-lisp rem) (rf-terpri)))
  (cond ((pro-expr-p s)
         (rf-princ-like-lisp "  <*pro-expr*>")
         (newline-indent indent)
         (rf-princ-like-lisp "Type: ")
         (rf-princ-like-lisp (pro-expr-type s))
         (newline-indent indent)
         (rf-princ-like-lisp "Len:  ")
         (rf-princ-like-lisp (pro-expr-len s))
         (newline-indent indent)
         (rf-princ-like-lisp "Val:  ")
         (pps (pro-expr-value s) (+ indent 6)) )
        ((consp s)
         (rf-princ-like-lisp "[")
```

```
        (mapcar #'(lambda (x)
                    (newline-indent (1+ indent))
                    (pps x (1+ indent))
                    )
                s)
        (newline-indent indent)
        (rf-princ-like-lisp "]"))
       (t
        (rf-princ-like-lisp s) ))
  s)

(defun construct-measure-tree (expr must-be-sqr-p &aux pair)
  ; Returns a value of type pro-expr.
  ; The parameter must-be-sqr-p signals that all round applications
  ; have to be transformed to sqr-applications, because the call came
  ; from inside a sqr argument list.
  (cond ((stringp expr)
         (make-pro-expr :type 'atomic
                        :len  (flatsize expr)
                        :value (prin1-to-string expr) ))
        ((eq 'id expr)
         (make-pro-expr :type 'atomic
                        :len  1
                        :value "_" ))
        ((atom expr)
         (make-pro-expr :type 'atomic
                        :len  (flatc expr)
                        :value (string-downcase (princ-to-string expr)) ))
        ((vari-t expr)
         ; expr = (VARI symbol)
         (let ((name-conc-level (pro-get-variable expr)))
           (make-pro-expr :type 'atomic
                          :len (length name-conc-level)
                          :value name-conc-level )))
        ((variable-binding-t expr)
         ; expr = (variable = binding-value)
         ; If the interpreter is active, then variable is a symbol with
         ; leading "_"      (e.g. _xyz).
         ; If the emulator is active then, variable is a taged list
         ; (e.g. (vari x))
         (let ((var-as-str
                (if (atom (first expr))
                    (lisp-var-sym->pro-var-sym (first expr))
                    (pro-get-variable (car expr)) )))
           (setq pair (list (make-pro-expr
                             :type 'atomic
                             :len (length var-as-str)
                             :value var-as-str )
                            (construct-measure-tree (third expr)
                                                    must-be-sqr-p ))))
         (make-pro-expr
          :type 'variable-binding
          :len  (+ 3                              ; 3 = length(" = ")
```

```
                    (pro-expr-len (first pair))
                    (pro-expr-len (second pair)) )
           :value pair ))
((is-t expr)
          ; expr = (IS left right)
          ; impl. INST on the left side
          (setq pair (list (construct-measure-tree (second expr)
                                                   t )  ;left
                           (construct-measure-tree (third expr)
                                                   must-be-sqr-p ) ))  ;right
          (make-pro-expr
           :type 'is
           :len  (+ 4                              ; 4 = length(" is ")
                    (pro-expr-len (first pair))
                    (pro-expr-len (second pair)) )
           :value pair ))
((and must-be-sqr-p (tup-t expr))
 ;; expr = (tup arg-1 .. arg-n)
 ;; inst outside -> suppress the tup-constructor
  (setq pair (list                ; constructor & argument-list
                           nil
                           (construct-measure-tree-of-arg-list (cdr expr)
                                                               t ) ))
 (make-pro-expr
           :type 'sqr-application
           :len (pro-expr-len (second pair))
           :value pair ))
(must-be-sqr-p
 ;; expr = (constructor arg-1 .. arg-n)
 ;; inst outside -> handle inst like any other constructor
  (setq pair (list     ; constructor & argument-list
                           (construct-measure-tree (first expr) t)
                           (construct-measure-tree-of-arg-list (cdr expr) t) ))
          (make-pro-expr
           :type 'sqr-application
           :len  (+ (pro-expr-len (first pair))
                    (pro-expr-len (second pair)) )
           :value pair ))
;; there was no inst outside:
;; By the following case (inst (vari x)) is printed as X if there
;; was no inst outside.
;; Remove this sexpression and (inst (vari x)) is printed as inst(X).
((inst-vari-t expr)
 (construct-measure-tree (second expr) nil) )
((and (inst-t expr) (not (vari-t (second expr))))
 ;; expr = (INST (constructor arg-1 .. arg-n))
 (construct-measure-tree (second expr) t) )
(t
 ;; expr = (functor arg-1 .. arg-n)
          (setq pair (list
        (construct-measure-tree (first expr) nil)
        (construct-measure-tree-of-arg-list (cdr expr) nil)))
 (make-pro-expr
   :type 'round-application
```

```
   :len (+ (pro-expr-len (first pair))
 (pro-expr-len (second pair)) )
   :value pair ))))


(defun construct-measure-tree-of-arg-list (arg-list must-be-sqr-p)
  ; Returns a value of type pro-expr.
  ; The len-field contains the whole length included parentheses (or brackets),
  ; commas and spaces.
  (do* ((arg-list-tail-p nil)
        (l arg-list (cdr l))
        (arg (car l) (car l))
        (tree nil)
        (result nil)
        (first-arg-p t nil)
        (total-length 2)          ; 2 = length("()")
        )
       ((null l) (make-pro-expr :type 'argument-list
                                :len total-length
                                :value result ))
       (if (eq '\| arg)
           (progn (if (not first-arg-p)
                      ; bar not at the beginning:
                      ; 1 = length(" "), space before the bar
                      (setq total-length (1+ total-length)) )
                  (setq arg-list-tail-p t) )
           (progn (setq tree (construct-measure-tree arg must-be-sqr-p))
                  (if arg-list-tail-p
                      ; last argument was a bar:
                      (setq tree (make-pro-expr
                                  :type 'arg-list-tail
                                  :len  (pro-expr-len tree)
                                  :value tree )) )
    (if (not first-arg-p)
        ;; 2 = length(", ") resp. length("| ")
        ;; seperator before this arg.
        (setq total-length (+ 2 total-length)) )
                  (setq total-length (+ total-length
                                        (pro-expr-len tree) ))
                  (setq result (append result (list tree))) ))))


(defun construct-measure-tree-of-head (head)
  (if (consp head)
      ; head = (functor arg-1 arg-2 .. )
      (let ((pair (list
                   ; functor & argument-list
                   (construct-measure-tree (first head) t)
                   (construct-measure-tree-of-arg-list (cdr head) t) )))
        (make-pro-expr
         :type 'round-application
         :len  (+ (pro-expr-len (first pair))
                  (pro-expr-len (second pair)) )
         :value pair ))
```

```
      ; head = ATOM
      (construct-measure-tree head nil) ))

(defun construct-measure-tree-of-footed-rule (arg-list)
  (let ((r (construct-measure-tree-of-arg-list arg-list nil)))
    (setf (pro-expr-len r) (1+ (pro-expr-len r)))
    ; 1 = length(" &") - length(",")
    r ))

(defun pro-print (expr &optional (x-cursor 1))
  ; Prints expr in the Prolog-style syntax.
  ; Starts at x-cursor.
  ; Returns no value
  (if (consp expr)
      (case (car expr)
            (hn (if (= 2
                       (length expr))
                    (pro-print-clause
                     (construct-measure-tree-of-head (second expr))
                     nil
                     'horn-fact )
                    (pro-print-clause
                     (construct-measure-tree-of-head (second expr))
                     (construct-measure-tree-of-arg-list (cddr expr) nil)
                     'horn-rule )))
            (ft (if (= 3
                       (length expr))
                    (pro-print-clause
                     (construct-measure-tree-of-head (second expr))
                     (construct-measure-tree-of-arg-list (cddr expr) nil)
                     'footed-fact )
                    (pro-print-clause
                     (construct-measure-tree-of-head (second expr))
                     (construct-measure-tree-of-footed-rule (cddr expr))
                     'footed-rule )))
            (t (pro-print-tree (construct-measure-tree expr nil) x-cursor)) )
      (pro-print-tree (construct-measure-tree expr nil) x-cursor) ))

;; All functions named pro-print-... return the position of the
;; cursor after printing the expression.

(defun pro-print-clause (left right type &optional (x-cursor 1))
  ; dom(type) = {horn-fact, horn-rule, footed-fact, footed-rule}
  (let ((middle nil)
        (sep-before-last #\,)
        (same-line-p nil)    ; <==> left middle are on the same line
        (x x-cursor) )
    (case type
          (horn-fact (setq middle nil))
          (horn-rule (setq middle " :- "))
          (footed-fact (setq middle " :-& "))
          (footed-rule (setq middle " :- ")
```

```
                    (setq sep-before-last " &") )
          (t (internal-error "unknown type in <pro-print-clause>: " type)))
    (setq x (pro-print-tree left x-cursor))
    (if middle
        (progn
         (setq same-line-p (space-enough-p (length middle) x))
         (setq x (pro-print-string middle x)) ))
    (if right
        (if same-line-p
            (setq x (pro-print-arg-list right x (+ x-cursor 2)
                                        the-non-printing-char
                                        the-non-printing-char
                                        sep-before-last ))
            (setq x (pro-print-arg-list right x x
                                        the-non-printing-char
                                        the-non-printing-char
                                        sep-before-last )
                  )))
    (pro-print-char #\. x) ))

(defun pro-print-tree (tree x-cursor)
  (case (pro-expr-type tree)
        (atomic
         (pro-print-atom tree x-cursor) )
        ((round-application sqr-application)
         (pro-print-application tree x-cursor) )
        (arg-list-tail
 ;; the bar was printed by pro-print-arglist-elements.
 ;; Now print the tail itself:
         (pro-print-tree (pro-expr-value tree) x-cursor) )
        (is
         (pro-print-is tree x-cursor) )
        (variable-binding
         (pro-print-variable-binding tree x-cursor) )
        (t
         (internal-error "Unknown tree-type in <pro-print-tree>"
                         (pro-expr-type tree) ))))

(defun pro-print-atom (tree x-cursor)
  (if (space-enough-p tree x-cursor)
      (progn (rf-princ-like-lisp (pro-expr-value tree))
             (+ x-cursor
                (pro-expr-len tree) ))
      (progn (rf-terpri)
             (rf-princ-like-lisp (pro-expr-value tree))
             (pro-expr-len tree)  )))


(defun pro-print-application (tree x-cursor)
  ; type    = round-application or sqr-application
  ; value = (functor arg-list)
  (let ((functor  (first (pro-expr-value tree)))
        (arg-list (second (pro-expr-value tree)))
```

```
        (x        x-cursor)
        (xx nil)
        (open-char nil)
        (close-char nil) )
    (case (pro-expr-type tree)
          (round-application (setq open-char round-left)
                             (setq close-char round-right) )
          (sqr-application (setq open-char sqr-left)
                           (setq close-char sqr-right) )
          (t (error "in pro-print-application.")) )
    (if functor
        ; it's not a list:
        (progn (if (space-enough-p functor x)
                   ; never break a functor
                   (setq x x-cursor)
                   (setq x (pro-newline-and-indent 1)) )
               (setq xx (pro-print-tree functor x)) )
        ; it's a list:
        (setq xx x) )
    (pro-print-arg-list arg-list
                        xx
                        (+ x 2)
                        open-char
                        close-char
                        #\, )))

(defun pro-print-arg-list (arg-list x-cursor x-cursor-left
                                    open-char close-char sep-before-last )
  (let* ((elements (pro-expr-value arg-list))
         (arg-list-info (get-arg-list-info elements))
         (nr-of-el (first arg-list-info))
         (max-len  (second arg-list-info))
         (simple-args-p (third arg-list-info))
         )
    (cond ((and (or (<= nr-of-el max-args-per-line)
                    simple-args-p )
                (space-enough-p (pro-expr-len arg-list) x-cursor) )
           ;; functor(arg-1, arg2)
           (pro-print-char close-char
                           (pro-print-arg-list-elements
                            elements
                            (pro-print-char open-char x-cursor)
                            nil   ; no extra-lines
                            sep-before-last )))
          ((or (space-enough-p max-len x-cursor 2)    ; 2 = length(sqr-left",")
               (<= x-cursor x-cursor-left) )
           ;; functor(      arg-1,
           ;;               arg-2 )
           (pro-print-char close-char
                           (pro-print-arg-list-elements
                            elements
                            (pro-print-char open-char x-cursor)
                            t   ; print on separated lines
```

```
                          sep-before-last )))
          (t
           ;; functor(
           ;;     arg-1,
           ;;     arg-2)
           (pro-print-char open-char x-cursor)
           (pro-print-char close-char
                          (pro-print-arg-list-elements
                           elements
                           (pro-newline-and-indent x-cursor-left)
                           t   ; print on separated lines
                           sep-before-last ))))))

(defun pro-print-arg-list-elements (elements x-cursor
                                             extra-lines sep-before-last)
  (do* ((rest-elements elements (cdr rest-elements))
(akt-element (first rest-elements) (first rest-elements))
(next-element (second rest-elements) (second rest-elements))
(first-arg-p t nil)
(x x-cursor) )
       ((null rest-elements)
x )
       ;; at least one left to print:
       (if (and first-arg-p
(eq 'arg-list-tail (pro-expr-type akt-element)) )
    ;; tail at the front, no space before the bar:
    (return (pro-print-tree akt-element
       (pro-print-string "| " x) )) )
       ;; no tail at the begin of the arg-list:
       (setq x (pro-print-tree akt-element x))
       ;; the akt. argument has been printed.
       ;; How many args are left ?
       (cond ((null (cdr rest-elements))
       ;; no args left:
       (return x) )
      ((not (null (cddr rest-elements)))
       ;; more then one argument left to print:
       (setq x (pro-print-char comma x)) )
      ;; exact one argument left to print:
      ((eq 'arg-list-tail
    (pro-expr-type next-element) )
       ;; to print: " |"<tail>
       (setq x (pro-print-string " |" x)) )
      ;; last element is not of type arg-list-tail:
      ((stringp sep-before-last)
       (setq x (pro-print-string sep-before-last x)) )
      (t
       (setq x (pro-print-char sep-before-last x)) ))
       (if extra-lines
    (setq x (pro-newline-and-indent x-cursor))
    (setq x (pro-print-char #\space x)) )))

(defun pro-print-is (tree x-cursor)
```

```
  ; type = is
  ; value = (left-side right-side)
  (let ((x x-cursor)
        (left-side (first (pro-expr-value tree)))
        (right-side (second (pro-expr-value tree))) )
    (setq x (pro-print-tree left-side x))
    (pro-print-tree right-side
                    (pro-print-string " is "
                                      x
                                      x-cursor ))))

(defun pro-print-variable-binding (tree x-cursor)
  ; type = variable-binding
  ; value = (variable       binding-value)
  ; Assumes that x-cursor = 1.
  (let ((variable (first (pro-expr-value tree)))
        (binding-value (second (pro-expr-value tree))) )
    (pro-print-tree
     binding-value
     (pro-print-string " = "
                       (pro-print-tree variable x-cursor)
                       x-cursor ))) )

(defun pro-print-char (char x-cursor)
  (if (char= char the-non-printing-char)
      x-cursor
      (case char
            ((#\. #\, #\space)
             (rf-princ-like-lisp char)
             (1+ x-cursor) )
            (t
             (if (space-enough-p 1 x-cursor)
                 (progn (rf-princ-like-lisp char)
                        (1+ x-cursor) )
                 (progn (pro-newline-and-indent 1)
                        (rf-princ-like-lisp char)
                        2 ))))))        ; 2 = Cursor column

(defun pro-print-string (str x-cursor &optional (x-cursor-left 1))
  ; Tries to print str on the current line.
  ; If not possible print on the next line at column x-cursor-left.
  (let ((str-wo-blank ""))
    (cond ((space-enough-p (length str) x-cursor)
           ; fits on line
           (rf-princ-like-lisp str)
           (+ x-cursor
              (length str) ))
          ((space-enough-p (length str) x-cursor-left)
           ; next line with indentation
           (pro-newline-and-indent x-cursor-left)
           (if (char= #\Space
                      (char str 0))
```

```
            (setq str-wo-blank (subseq str 1))
            (setq str-wo-blank str) )
          (rf-princ-like-lisp str-wo-blank)
          (+ x-cursor-left
             (length str-wo-blank) ))
         (t
          ; next line without indentation
          (pro-newline-and-indent 1)
          (if (char= #\Space
                     (char str 0))
            (setq str-wo-blank (subseq str 1))
            (setq str-wo-blank str) )
          (rf-princ-like-lisp str-wo-blank)
          (+ x-cursor-left
             (length str-wo-blank) )))))

(defun pro-newline-and-indent (x)
  (rf-terpri)
  (dotimes (i (1- x) x) (rf-princ-like-lisp " ")) )

(defun space-enough-p (tree-or-number x-cursor &optional (extra-space 0))
  ; Note:   the last column is reserved for special characters, which
  ;         never should be printed at the begin of a line (e.g comma,
  ;         point).
  (>= (- *rf-print-width* x-cursor extra-space)
      (if (pro-expr-p tree-or-number)
          (pro-expr-len tree-or-number)
          tree-or-number )))

(defun pro-get-variable (var)
  ; var = (VARI name level)
  ; where the level-field is optional.
  ; Example: var = (VARI time 127) ==> "Time:127"
  (let ((level (level-of var))
        (name (string-upcase
                (string-downcase (princ-to-string
                                   (second var)) )
                :start 0
                :end 1 )))
    (if (not (null level))
        (setq name (strcat name ":" (princ-to-string level))) )
    (if (digit-char-p (character (subseq name 0 1)))
        (strcat "_" name)
        name )))

(defun lisp-var-sym->pro-var-sym (lisp-var-sym)
  ; Example:   input:     _xyz
  ;            output:    Xyz
  (let ((pro-name
         (nstring-upcase (nstring-downcase
                           (subseq (princ-to-string lisp-var-sym) 1))
                         :start 0
```

```
                        :end 1 )))
    (if (digit-char-p (character (subseq pro-name 0 1)))
        (strcat "_" pro-name)
        pro-name )))

(defun get-arg-list-info (elements)
  ;; Returns a list.        1.: No. of elements in elements.
  ;;                        2.: Length of biggest element.
  ;;                        3.: simple-arg-p
  (do* ((n 0 (1+ n))
        (maxi 0 (max maxi
                     (pro-expr-len element) ))
        (simple-arg-p t (and simple-arg-p
                             (or (eq 'atomic
                                     (pro-expr-type element) )
                                 (<= (pro-expr-len element)
                                     max-len-of-a-simple-arg))))
        (l elements (cdr l))
        (element (car l) (car l) )
        )
       ((null l)
        (list n maxi simple-arg-p) )))
```

```
;;; End of Module Lisp2Pro.Lsp
```

D Listing: Pro2Lisp

```
;-----------------------------------------------------------------------
;
; Module Pro2Lisp.Lsp:      Contains functions to read goals und clauses
;                           given in the Prolog-like syntax of Relfun.
;                           The input can be read from a string or from a
;                           file of chararacters.
;                           The output contains the equivalent form in
;                           the Lisp-like syntax as a symbolic expression.
;
;                           If an error is detected a message is displayed
;                           an the value NIL is returned.
;
;
; Michael Herfert, 2/92
;
;
; Exported items:
;
;
;   pro-read-data-base      filename                                [Function]
;
;       Assumes that the file contains clauses.
;       Returns a list of the clauses in Lisp-like syntax.
;
;
;   pro-read-goal       string                                      [Function]
;
;       Reads in a single goal from a string.
;       Note that there is no point at the end of a goal.
;
;
;   pro-read-clause     string                                      [Function]
;
;       Reads in a single clause from the string.
;
;
;   pro-split-input     string                                      [Function]
;
;       Splits the given string in the first symbol and the rest
;       of it. Useful for recognizing system-commands.
;       Returns a pair.
;       Examples:
;                   (pro-split-input "consult my-base.rfp")
;           ==>     (consult . "my-base.rfp")
;
;                   (pro-split-input "[a,b,c]")
;           ==>     (nil . "")


(defun set-syntax-pro (a-readtable)
  ; reading lists:
```

```
  (set-macro-character #\| #'single-char-reader  nil a-readtable)
  (set-macro-character #\, #'single-char-reader  nil a-readtable)
  (set-macro-character #\' #'single-char-reader  nil a-readtable)
  (set-macro-character #\` #'single-char-reader  nil a-readtable)
  (set-syntax-from-char #\_ #\A a-readtable nil)
  a-readtable )

(defun single-char-reader (stream char)
  (declare (ignore stream))
  char )

(defvar *rfi-readtable-pro*
  ; this is a read-only variable
  (set-syntax-pro (copy-readtable nil)) )   ; nil -> copy of std. readtable

; Easy life for the editor:
(defconstant sqr-left #\[)
(defconstant sqr-right #\])
(defconstant round-left #\()
(defconstant round-right #\))
(defconstant comma #\,)
(defconstant bar #\|)
(defconstant eof-char (code-char 4))

(defun signal-error (scanner &rest msg-s)
  ; Prints all arguments and the error-position via the scanner.
  (pro-print-error scanner msg-s)
  (throw :pro-read-error-tag nil) )    ; nil signals an error

(defun pro-print-error (scanner msg-s)
  ; msg-s = (msg-1 msg-2 ..)
  (rf-terpri)
  (let ((x (token-x-pos (funcall scanner 'last-token)))
        (y (token-y-pos (funcall scanner 'last-token)))
        (last-line (funcall scanner 'last-line))
        (act-line (funcall scanner 'act-line))
        )
    (rf-princ-like-lisp (format nil
                                "Error near line ~A, column ~A.~%"
                                y
                                x ))
    (mapcar #'(lambda (msg)
                (if (token-p msg)
                    (if (token-value msg)
                        (rf-princ-like-lisp (token-value msg))
                        (rf-princ-like-lisp (tok-type->string
                                             (token-type msg) )))
                    (rf-princ-like-lisp msg) ))
            msg-s )
    (rf-terpri)
    (if (> y 1)
        (rf-princ-like-lisp last-line) )
```

```
    (rf-terpri)
    (rf-princ-like-lisp act-line)
    (rf-terpri)
    (rf-terpri) ))

(defun tok-type->string (tok-type)
  (case tok-type
        (round-left (string round-left))
        (round-right (string round-right))
        (sqr-left (string sqr-left))
        (sqr-right (string sqr-right))
        (comma (string comma))
        (bar (string bar))
        (ampersand "&")
        (implies ":-")
        (point ".")
        (is "is")
        (empty "End of input")
        (t (princ-to-string tok-type)) ))


;; The scanner returns a value of this type:
(defstruct token
  type     ; dom = {constant, variable,
           ;          round-left, round-right, sqr-left, square-right
           ;          point, comma, ampersand, implies,
           ;          is, empty}
  value    ; used with number, constant, variable.
  x-pos    ; to report an error-position
  y-pos
)

(defun neq (x y)
  (not (eq x y)) )

(defmacro accept-token-type (tok-type)
  ;Signals an error, if the wrong token is found,
  ;otherwise it reads in the next token.
  `(if (eq (funcall scanner 'last-token-type)
           ,tok-type)
       (funcall scanner)
       (signal-error scanner
                     (format nil "Expected: ~A~%Found:    "
                             (tok-type->string ,tok-type) )
                     (funcall scanner 'last-token) )))

(defmacro strcat (&rest strings)
  `(concatenate 'string
                ,@strings ))

(defun gen-scanner (the-input-stream)
  ; Returns a scanner-function.
```

```
; The scanner could be seen as an object in the view of OOP.
; A very big function, but there are many local functions defined by LABELS.
(let ((x-pos 0)
      (y-pos 1)
      (char-pos -1)
      (last-ch #\Space)
      (last-token (make-token :x-pos 1 :y-pos 1 :type 'empty))
      (last-line "")
      (act-line "1:  ")
      ;; terminating characters:
      ;; (cons <char> <type>)
      (terminating-chars (list (cons sqr-left 'sqr-left)
                               (cons sqr-right 'sqr-right)
                               (cons round-left 'round-left)
                               (cons round-right 'round-right)
                               (cons #\. 'point)
                               (cons #\, 'comma)
                               (cons #\| 'bar)
                               (cons #\& 'ampersand) )))
  (labels
   ((get-ch ()
            (if (char/= last-ch eof-char)
                (progn
                 (setq last-ch (read-char the-input-stream
                                          nil
                                          eof-char ))
                 (setq char-pos (1+ char-pos))
                 (cond ((char= last-ch #\Newline)
                        (setq x-pos 1)
                        (setq y-pos (1+ y-pos))
                        (setq last-line act-line)
                        (setq act-line (strcat (princ-to-string y-pos)
                                               ":  " )))
                       (t
                        (setq act-line (strcat act-line (string last-ch)))
                        (setq x-pos (1+ x-pos))) )))
            last-ch )
    (scan-item (continue-p)
               (let ((r ""))
                 (loop (if (and (char/= eof-char last-ch)
                                (funcall continue-p last-ch) )
                           (setq r (strcat r (string last-ch)))
                           (return r) )
                       (get-ch) )))
    (id-char-p (ch)
               (cond ((both-case-p ch) t)
                     ((digit-char-p ch) t)
                     (t (member ch '(#\+ #\- #\* #\/ #\_ #\! #\< #\>
                                     #\=
                                     )
                                :test #'char=)) ))
    (special-char-p (ch)
                    (cond ((assoc ch terminating-chars :test #'char=)
                           nil )
```

```
                    ((member ch '(#\; #\' #\`) :test #'char=)
                     nil )
                    ((char<= ch #\Space) nil)
                    ((char> ch (int-char 127)) nil)
                    ((digit-char-p ch) nil)
                    ((both-case-p ch) nil)
                    (t t) ))
(string-constituent-p (ch)
                    (and (not (char= #\" ch))
                         (not (char= #\Newline ch)) ))
(scan-whitespace ()
                (do ((abort nil))
                    (abort)
                    (case last-ch
                          (#\Space (get-ch))
                          (#\Newline (get-ch))
                          (#\Tab (get-ch))
                          (#\%  (do ((abort nil)) ; comment
                                    (abort)
                                    (setq abort (char=
                                                 last-ch
                                                 #\Newline))
                                    (get-ch) ))
                          (t (setq abort t)) )))
(scan-digits ()
             ;; digits := digit {digit}
             ;; returns a string of digits
             (if (digit-char-p last-ch)
                 (scan-item #'digit-char-p)
                 (signal-error #'scanner
                               (format nil
                                       "Expected:  digit~%Found:     ~C"
                                       last-ch ))))
(scan-integer ()
              ;; integer := ["+" | "-"] digits
              ;; returns a string
              (let ((sign (case last-ch
                                (#\+ (get-ch)
                                 "+")
                                (#\- (get-ch)
                                 "-")
                                (t "" ))))
                (strcat sign (scan-digits)) ))
(scan-real ()
           ;; real :=     integer ["." digits]
           ;;             ["E" integer]
           ;; returns a number
           (let* ((left-side (scan-integer))
                  (right-side (if (and (char= #\. last-ch)
                                       (digit-char-p (peek-char
                                                      nil
                                                      the-input-stream
                                                      nil
                                                      eof-char)))
```

```
                                    (progn (get-ch)
                                           (strcat "." (scan-integer)) )
                                  "" ))
                        (exponent (case last-ch
                                    ((#\E)
                                     (get-ch)
                                     (strcat "E" (scan-integer)) )
                                    (t "") )) )
                    (read-from-string (strcat left-side
                                              right-side
                                              exponent ))))
      (next-token ()
                  (let ((r nil)
(*readtable* *rfi-readtable-pro*) ) ; dynamic variable
                    (scan-whitespace)
                    (setq r (make-token :x-pos x-pos :y-pos y-pos))
                    (cond  ((digit-char-p last-ch)
                            (setf (token-type r) 'constant)
                            (if (char= #\1 last-ch)
                                ;; check for functors 1- and 1+
                                (case (peek-char nil the-input-stream
                                                 nil eof-char)
                                      (#\+ (setf (token-value r) '1+)
                                           (get-ch) (get-ch) )
                                      (#\- (setf (token-value r) '1-)
                                           (get-ch) (get-ch) )
                                      (t   (setf (token-value r)
                                                 (scan-real) )))
                              (setf (token-value r) (scan-real))))
                           ((lower-case-p last-ch)
                            (setf (token-value r)
                                  (read-from-string (scan-item
                                                     #'id-char-p)) )
                            (if (eq 'is (token-value r))
                                (setf (token-type r) 'is
                                      (token-value r) nil )
                              (setf (token-type r) 'constant) ))
                           ((upper-case-p last-ch)
                            (setf (token-type r) 'variable)
                            (setf (token-value r)
                                  (read-from-string (scan-item
                                                     #'id-char-p)) ))
                           ((char= #\_ last-ch)
                            (get-ch) ; read the underscore
                            (cond ((digit-char-p last-ch)
                                   (setf (token-type r)
                                         'variable
                                         (token-value r)
                                         (intern (scan-digits))))
                                  ((lower-case-p last-ch)
                                   (setf (token-type r)
                                         'variable
                                         (token-value r)
                                         (read-from-string (scan-item
```

```
                                        #'id-char-p ))))
        ((upper-case-p last-ch)
         (signal-error
          #'scanner
          "Uppercase-letter not allowed after _"))
        (t ; anonymous variable
         (setf (token-type r)
               'variable
               (token-value r)
               '_ ))))
((char= #\" last-ch)
 ;;(setf (token-type r) 'string)
 (setf (token-type r) 'constant)
 (get-ch)    ; read opening quote
 (setf (token-value r)
       (scan-item #'string-constituent-p) )
 (if (char= #\" last-ch)
     (get-ch)     ; read closing quote
     (signal-error #'scanner
                   (format nil
                           "Expected: \"~%Found: ~C"
                           last-ch ))))
((or (char= #\+ last-ch)
     (char= #\- last-ch) )
 (cond ((digit-char-p (peek-char
                       nil the-input-stream
                       nil eof-char))
        ;;(setf (token-type r) 'number)
        (setf (token-type r) 'constant
              (token-value r) (scan-real) ))
       (t
        (setf (token-type r) 'constant
              (token-value r) (read-from-string
                               (scan-item
                                #'id-char-p))))))
((special-char-p last-ch)
 ;; read in a max. sequence of special-chars
 (let ((str (scan-item #'special-char-p)))
   (if (string= ":-" str)
       (setf (token-type r) 'implies)
     (setf (token-type r) 'constant
           (token-value r) (read-from-string str)
           ))))
((char= eof-char last-ch)
 (setf (token-type r) 'empty) )
(t (let ((pair-of-char-and-type
          (assoc last-ch terminating-chars) ))
     (if pair-of-char-and-type
         (progn
           (setf (token-type r)
                 (cdr pair-of-char-and-type))
           (get-ch) )
       (signal-error #'scanner
                     (format nil
```

```
                                                    "Illegal character: ~C"
                                                    last-ch ))))))
                    (setq last-token r)
                    r ))
      (continue-after-error ()
                          ;; searches for the next token after a point.
                          (loop
                           (case (token-type last-token)
                                 (point (return (next-token)))
                                 (empty (return last-token))
                                 (t (next-token)) )))
      (scanner (&optional (message 'next-token))
               (case message
                     (next-token (next-token))
                     (last-token last-token)
                     (last-token-type (token-type last-token))
                     (last-token-value (token-value last-token))
                     (x-pos x-pos)
                     (y-pos y-pos)
                     (last-line last-line)
                     (act-line  act-line)
                     (continue-after-error (continue-after-error))
                     (pos-of-first-non-white (scan-whitespace)
                                             char-pos )
                     (t  (internal-error "unknown message in scanner"
                                         message)) )))
      ; init variables:
      (get-ch)
      (next-token)
      #'scanner   ; value of gen-scanner-from-fct
      )))

(defun builtin-sym-p (tok)
  ;; for future extensions
  (declare (ignore tok))
  nil
  )

; Following two meta-rules to obtain a clear structure:

(defun parse-general-loop ( scanner syntax-rule
                            first-of-syntax-rule
                            &optional auto-inst-p )
  ; Parses a construct of the form:
  ;    syntax-rule {syntax-rule}
  ; where   first(syntax-rule) = first-of-syntax-rule.
  ; syntax-rule is a parser-function, first-of-syntax-rule is a token-type.
  ; Returns a list of the results
  (do ((l (list (funcall syntax-rule scanner auto-inst-p))
          (append l (list (funcall syntax-rule scanner auto-inst-p))) ))
      ((neq first-of-syntax-rule
            (funcall scanner 'last-token-type) )
       l) ))
```

```
(defun parse-general-enumeration (scanner syntax-rule &optional auto-inst-p)
  ; Parses a construct of the form:
  ;    syntax-rule {"," syntax-rule}
  ; where syntax-rule is a parser-function.
  ; Returns a list of the results.
  (do ((l (list (funcall syntax-rule scanner auto-inst-p))
          (append l (list (funcall syntax-rule scanner auto-inst-p))) ))
      ((neq (token-type (funcall scanner 'last-token))
            'comma)
       l )
      (accept-token-type 'comma) ))


(defun parse-clause (scanner)
  ; clause ::= head      (    "."
  ;                       |   ":-" [body] "."
  ;                      )
  (let* ((head      (parse-head scanner))
         (tok-type  (funcall scanner 'last-token-type)) )
    (funcall scanner)  ; accept the symbol between head and body
    (case tok-type
          (point     (list 'hn head))                                  ; horn-fact
          (implies   (if (eq 'point (funcall scanner 'last-token-type)) ;rule
                         ; empty body:
                         (progn (accept-token-type 'point)
                                (list 'hd head) )
                         (prog1 (parse-the-body scanner head)
                                (accept-token-type 'point) )))
          (t       (signal-error scanner
                                 "Unknown symbol between head and body: "
                                 tok-type )))))


(defun parse-head (scanner)
  ; head ::= term round-list-with-terms
  (cons (parse-term scanner t)    ; t means auto-inst-p
        (parse-round-list-with-terms scanner) ))


(defun parse-the-body (scanner head &aux body)
  ; body ::=     ( expr+ ["&" expr] )
  ;           | ( "&" expr )
  ; Returns a horn- or a footed-rule.
  (if (neq 'ampersand (funcall scanner 'last-token-type))
      (setq body (parse-expr+ scanner)) )
  (if (eq 'ampersand (funcall scanner 'last-token-type))
      (progn (accept-token-type 'ampersand)
             `(ft ,head ,@body ,(parse-expr scanner)) )
      `(hn ,head ,@body) ))


(defun parse-expr (scanner &optional auto-inst-p)
  ; expr ::=    (   term [    {round-list}
  ;                       |  IS expr
  ;                      ]
```

```
  ;             | builtin
  ;           )
  ; auto-inst-p is not used in this function.
  (declare (ignore auto-inst-p))
  (if (builtin-sym-p (funcall scanner 'last-token))
      (error "Noch zu ergaenzen")
      (let ((r (parse-term scanner)))
        (case (funcall scanner 'last-token-type)
              (is (accept-token-type 'is)
    (list 'is (remove-non-var-inst r) (parse-expr scanner)))
       (round-left (construct-application
                           r
                           (parse-general-loop scanner
                                               #'parse-round-list
                                               'round-left )))
              (t    r) ))))

(defun parse-term (scanner &optional auto-inst-p)
  ; term ::=    (   CONSTANT
  ;              | VARIABLE
  ;              | sqr-list
  ;            )
  ;            {sqr-list}
  ; If auto-inst-p is t then no explicit inst-tag is generated.
  ; Needed for terms in the head of a rule and for nested lists.
  (let ((tok (funcall scanner 'last-token))
        (l nil)
        (sqr-left-p nil)
        )
    (case (token-type tok)
          (constant (setq l (token-value tok))
                    (accept-token-type 'constant))
          (variable (setq l (parse-variable scanner)))
          (sqr-left (setq l (cons 'tup (parse-sqr-list scanner t))
                          sqr-left-p t ))
          (t        (signal-error scanner
                                  (format nil
                                          (strcat
                                           "Constant, variable or [ expected."
                                           "~%Found: " ))
                                  tok )))
    (if (eq 'sqr-left (funcall scanner 'last-token-type))
        (setq l (construct-application
                 l
                 (parse-general-loop scanner
                                     #'parse-sqr-list
                                     'sqr-left
                                     t ) )
              sqr-left-p t ))
    (if (and (not auto-inst-p)
             sqr-left-p )
        (setq l (list 'inst l)) )
    l ))
```

```
(defun parse-round-list (scanner &optional auto-inst-p &aux l)
  ; round-list ::= "(" [expr+] ["|" expr] ")"
  (accept-token-type 'round-left)
  (case (funcall scanner 'last-token-type)
        (bar )
        (round-right )
        (t (setq l (parse-expr+ scanner))) )
  (if (eq 'bar (funcall scanner 'last-token-type))
      (progn (accept-token-type 'bar)
             (setq l (append l
                             (list '\| (parse-expr scanner auto-inst-p)) ))))
  (accept-token-type 'round-right)
  l )

(defun parse-sqr-list (scanner &optional auto-inst-p &aux l)
  ; sqr-list ::= "[" [term+] ["|" term] "]"
  (accept-token-type 'sqr-left)
  (case (funcall scanner 'last-token-type)
        (bar )
        (sqr-right )
        (t (setq l (parse-term+ scanner auto-inst-p))) )
  (if (eq 'bar (funcall scanner 'last-token-type))
      (progn (accept-token-type 'bar)
             (setq l (append l
                             (list '\| (parse-term scanner auto-inst-p)) ))))
  (accept-token-type 'sqr-right)
  l )

(defun parse-round-list-with-terms (scanner &aux l)
  ; round-list-with-terms ::= "(" [term+] ["|" VARIABLE] ")"
  (accept-token-type 'round-left)
  (case (funcall scanner 'last-token-type)
        (bar )
        (round-right )
        (t (setq l (parse-term+ scanner t))) )
  (if (eq 'bar (funcall scanner 'last-token-type))
      (progn (accept-token-type 'bar)
             (setq l (append l
                             (list '\| (parse-variable scanner)) ))))
  (accept-token-type 'round-right)
  l )

(defun parse-expr+ (scanner &optional auto-inst-p)
  ; expr+ ::= expr {"," expr}
  ; Returns a list of results.
  ; auto-inst-p is not used in this function.
  (declare (ignore auto-inst-p))
  (parse-general-enumeration scanner #'parse-expr) )

(defun parse-term+ (scanner &optional auto-inst-p)
  ;; term+ ::= term {"," term}
```

```
  ; Returns a list of results.
  (parse-general-enumeration scanner #'parse-term auto-inst-p) )

(defun parse-variable (scanner)
  (let ((var-name (funcall scanner 'last-token-value)))
    (accept-token-type 'variable)
    (if (eq '_ var-name)
        'id                          ; anonymous variable
      (list 'vari var-name) )))

(defun construct-application (functor list-of-arg-lists)
  ; Example:   functor = f
  ;            list-of-args-lists = ((a b) (x y))
  ;            Result: ((f a b) x y)
  (if (null list-of-arg-lists)
      functor
      (construct-application (cons functor
                                   (car list-of-arg-lists) )
                             (cdr list-of-arg-lists) )))

(defun remove-non-var-inst (term)
  ;; removes a possibly existing INST-tag if is not followed by a variable.
  (if (and (consp term)
    (inst-t term)
    (not (vari-t (second term))) )
      (second term) ; remove inst
      term ))

(defun pro-parse-head (head-as-string)
  (with-input-from-string (the-input-stream head-as-string)
                          (catch :pro-read-error-tag
                            (parse-head (gen-scanner the-input-stream)) )))

; ————————————        Exported functions:        ————————————————————

(defun pro-read-data-base (filename)
  (catch :pro-read-error-tag
    (with-open-file
     (the-input-stream filename :direction :input)
     (let ((scanner (gen-scanner the-input-stream))
           (clause nil)
           (data-base nil)
           (error-p nil) )
       (loop
        (if (eq 'empty (funcall scanner 'last-token-type))
            (if error-p
                (return nil)
              (return (reverse data-base)) ))
        (setq clause
              (catch :pro-read-error-tag
                (parse-clause scanner) ))
```

```
        (cond ((null clause)
               ;; Error
               (setq error-p t)
               (rf-terpri)
               (rf-princ-like-lisp "Continue reading to find more")
               (rf-princ-like-lisp " errors in line ")
               (funcall scanner 'continue-after-error)
               (rf-princ-like-lisp (token-y-pos (funcall scanner
                                                         'last-token )))
               (rf-princ-like-lisp ".")
               (rf-terpri) )
              (error-p
               ;; Error in previous clause -> don't construct database
               )
              (t
               ; No error
               (setq data-base (cons clause data-base)) )))))))

(defun pro-read-clause (str)
  (catch :pro-read-error-tag
    (with-input-from-string
     (the-input-stream str)
     (let* ((scanner (gen-scanner the-input-stream))
            (clause (parse-clause scanner)) )
       (cond ((null clause)
              ;; Error
              nil )
             ((eq 'empty (funcall scanner 'last-token-type))
              ;; all of the input is o.k.
              clause )
             (t
              (pro-print-error
               scanner
               (list "Only the first part of the input is correct."))
              nil ))))))

(defun pro-read-goal (str)
  ; goal ::= expr {"," expr}
  (catch :pro-read-error-tag
    (with-input-from-string
     (the-input-stream str)
     (let* ((scanner (gen-scanner the-input-stream))
            (goal (parse-general-enumeration scanner #'parse-expr)) )
       (cond ((null goal)
              ;; Error
              nil )
             ((eq 'empty (funcall scanner 'last-token-type))
              ;; all of the input is o.k.
              goal )
             ((eq 'point (funcall scanner 'last-token-type))
              (pro-print-error
               scanner
               (list "Point not allowed at the end of a goal.") )
```

```
              nil )
             (t
              (pro-print-error
               scanner
               (list "Only the first part of the input is correct.") )
              nil ))))))

(defun pro-split-input (input-line)
  ; Examples:
  ;
  ;         (pro-split-input "consult my-base.rfp")
  ;   ==>   (consult . "my-base.rfp")
  ;
  ;         (pro-split-input "[a,b,c]")
  ;   ==>   (nil . "")
  ;
  (catch :pro-read-error-tag
    (with-input-from-string
     (the-input-stream input-line)
     (let* ((scanner (gen-scanner the-input-stream))
            (tok     (funcall scanner 'last-token)) )
       (if (and (eq 'constant
                    (token-type tok) )
                (symbolp (token-value tok)) )
           (cons (token-value tok)
                 (subseq input-line
                         (funcall scanner
                                  'pos-of-first-non-white) ))
         (cons nil "") )))))

;-------------------------------------------------------------------------
; End of module Pro2Lisp.Lsp
;-------------------------------------------------------------------------
```

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
D-6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all published papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-91-28
Rolf Backofen, Harald Trost, Hans Uszkoreit:
Linking Typed Feature Formalisms and Terminological Knowledge Representation Languages in Natural Language Front-Ends
11 pages

RR-91-29
Hans Uszkoreit: Strategies for Adding Control Information to Declarative Grammars
17 pages

RR-91-30
Dan Flickinger, John Nerbonne:
Inheritance and Complementation: A Case Study of *Easy* Adjectives and Related Nouns
39 pages

RR-91-31
H.-U. Krieger, J. Nerbonne:
Feature-Based Inheritance Networks for Computational Lexicons
11 pages

RR-91-32
Rolf Backofen, Lutz Euler, Günther Görz:
Towards the Integration of Functions, Relations and Types in an AI Programming Language
14 pages

RR-91-33
Franz Baader, Klaus Schulz:
Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures
33 pages

RR-91-34
Bernhard Nebel, Christer Bäckström:
On the Computational Complexity of Temporal Projection and some related Problems
35 pages

RR-91-35
Winfried Graf, Wolfgang Maaß: Constraint-basierte Verarbeitung graphischen Wissens
14 Seiten

RR-92-01
Werner Nutt: Unification in Monoidal Theories is Solving Linear Equations over Semirings
57 pages

RR-92-02
Andreas Dengel, Rainer Bleisinger, Rainer Hoch, Frank Hönes, Frank Fein, Michael Malburg:
Π_{ODA}: The Paper Interface to ODA
53 pages

RR-92-03
Harold Boley:
Extended Logic-plus-Functional Programming
28 pages

RR-92-04
John Nerbonne: Feature-Based Lexicons: An Example and a Comparison to DATR
15 pages

RR-92-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner, Michael Schulte, Rainer Stark:
Feature based Integration of CAD and CAPP
19 pages

RR-92-06
Achim Schupetea: Main Topics of DAI: A Review
38 pages

RR-92-07
Michael Beetz:
Decision-theoretic Transformational Planning
22 pages

RR-92-08
Gabriele Merziger: Approaches to Abductive Reasoning - An Overview -
46 pages

RR-92-09
Winfried Graf, Markus A. Thies:
Perspektiven zur Kombination von automatischem Animationsdesign und planbasierter Hilfe
15 Seiten

RR-92-10
M. Bauer: An Interval-based Temporal Logic in a Multivalued Setting
17 pages

RR-92-11
Susane Biundo, Dietmar Dengler, Jana Koehler:
Deductive Planning and Plan Reuse in a Command Language Environment
13 pages

RR-92-13
Markus A. Thies, Frank Berger:
Planbasierte graphische Hilfe in objektorientierten Benutzungsoberflächen
13 Seiten

RR-92-14
Intelligent User Support in Graphical User Interfaces:
1. InCome: A System to Navigate through Interactions and Plans
   Thomas Fehrle, Markus A. Thies
2. Plan-Based Graphical Help in Object-Oriented User Interfaces
   Markus A. Thies, Frank Berger

22 pages

RR-92-15
Winfried Graf: Constraint-Based Graphical Layout of Multimodal Presentations
23 pages

RR-92-16
Jochen Heinsohn, Daniel Kudenko, Berhard Nebel, Hans-Jürgen Profitlich: An Empirical Analysis of Terminological Representation Systems
38 pages

RR-92-17
Hassan Aït-Kaci, Andreas Podelski, Gert Smolka:
A Feature-based Constraint System for Logic Programming with Entailment
23 pages

RR-92-18
John Nerbonne: Constraint-Based Semantics
21 pages

RR-92-19
Ralf Legleitner, Ansgar Bernardi, Christoph Klauck
PIM: Planning In Manufacturing using Skeletal Plans and Features
17 pages

RR-92-20
John Nerbonne: Representing Grammar, Meaning and Knowledge
18 pages

RR-92-21
Jörg-Peter Mohren, Jürgen Müller
Representing Spatial Relations (Part II) -The Geometrical Approach
25 pages

RR-92-22
Jörg Würtz: Unifying Cycles
24 pages

RR-92-23
Gert Smolka, Ralf Treinen:
Records for Logic Programming
38 pages

RR-92-24
Gabriele Schmidt: Knowledge Acquisition from Text in a Complex Domain
20 pages

RR-92-25
Franz Schmalhofer, Ralf Bergmann, Otto Kühn, Gabriele Schmidt: Using integrated knowledge acquisition to prepare sophisticated expert plans for their re-use in novel situations
12 pages

RR-92-26
Franz Schmalhofer, Thomas Reinartz, Bidjan Tschaitschian: Intelligent documentation as a catalyst for developing cooperative knowledge-based systems
16 pages

RR-92-27
Franz Schmalhofer, Jörg Thoben: The model-based construction of a case-oriented expert system
18 pages

RR-92-29
Zhaohur Wu, Ansgar Bernardi, Christoph Klauck:
Skeletel Plans Reuse: A Restricted Conceptual Graph Classification Approach
13 pages

RR-92-33
Franz Baader:
Unification Theory
22 pages

RR-92-34
Philipp Hanschke:
Terminological Reasoning and Partial Inductive Definitions
23 pages

RR-92-35
Manfred Meyer:
Using Hierarchical Constraint Satisfaction for Lathe-Tool Selection in a CIM Environment
18 pages

RR-92-36
Franz Baader, Philipp Hanschke:
Extensions of Concept Languages for a Mechanical Engineering Application
15 pages

RR-92-37
Philipp Hanschke:
Specifying Role Interaction in Concept Languages
26 pages

RR-92-38
Philipp Hanschke, Manfred Meyer:
An Alternative to Θ-Subsumption Based on Terminological Reasoning
9 pages

RR-92-43
Christoph Klauck, Jakob Mauss: A Heuristic driven Parser for Attributed Node Labeled Graph Grammars and its Application to Feature Recognition in CIM
17 pages

RR-92-44
Thomas Rist, Elisabeth André: Incorporating Graphics Design and Realization into the Multimodal Presentation System WIP
15 pages

RR-92-45
Elisabeth André, Thomas Rist: The Design of Illustrated Documents as a Planning Task
21 pages

RR-92-46
Elisabeth André, Wolfgang Finkler, Winfried Graf, Thomas Rist, Anne Schauder, Wolfgang Wahlster:
WIP: The Automatic Synthesis of Multimodal Presentations
19 pages

DFKI Technical Memos

TM-91-11
Peter Wazinski: Generating Spatial Descriptions for Cross-modal References
21 pages

TM-91-12
Klaus Becker, Christoph Klauck, Johannes Schwagereit: FEAT-PATR: Eine Erweiterung des D-PATR zur Feature-Erkennung in CAD/CAM
33 Seiten

TM-91-13
Knut Hinkelmann:
Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

TM-91-14
Rainer Bleisinger, Rainer Hoch, Andreas Dengel:
ODA-based modeling for document analysis
14 pages

TM-91-15
Stefan Bussmann: Prototypical Concept Formation
An Alternative Approach to Knowledge Representation
28 pages

TM-92-01
Lijuan Zhang:
Entwurf und Implementierung eines Compilers zur Transformation von Werkstückrepräsentationen
34 Seiten

TM-92-02
Achim Schupeta: Organizing Communication and Introspection in a Multi-Agent Blocksworld
32 pages

TM-92-03
Mona Singh
A Cognitiv Analysis of Event Structure
21 pages

TM-92-04
Jürgen Müller, Jörg Müller, Markus Pischel, Ralf Scheidhauer:
On the Representation of Temporal Knowledge
61 pages

TM-92-05
Franz Schmalhofer, Christoph Globig, Jörg Thoben
The refitting of plans by a human expert
10 pages

TM-92-06
Otto Kühn, Franz Schmalhofer: Hierarchical skeletal plan refinement: Task- and inference structures
14 pages

TM-92-08
Anne Kilger: Realization of Tree Adjoining Grammars with Unification
27pages

DFKI Documents

D-92-03
Wolfgan Maaß, Thomas Schiffmann, Dudung Soetopo, Winfried Graf: LAYLAB: Ein System zur automatischen Plazierung von Text-Bild-Kombinationen in multimodalen Dokumenten
41 Seiten

D-92-04
Judith Klein, Ludwig Dickmann: DiTo-Datenbank - Datendokumentation zu Verbrektion und Koordination
55 Seiten

D-92-06
Hans Werner Höper: Systematik zur Beschreibung von Werkstücken in der Terminologie der Featuresprache
392 Seiten

D-92-07
Susanne Biundo, Franz Schmalhofer (Eds.): Proceedings of the DFKI Workshop on Planning
65 pages

D-92-08
Jochen Heinsohn, Bernhard Hollunder (Eds.): DFKI Workshop on Taxonomic Reasoning Proceedings
56 pages

D-92-09
Gernod P. Laufkötter: Implementierungsmöglichkeiten der integrativen Wissensakquisitionsmethode des ARC-TEC-Projektes
86 Seiten

D-92-10
Jakob Mauss: Ein heuristisch gesteuerter Chart-Parser für attributierte Graph-Grammatiken
87 Seiten

D-92-11
Kerstin Becker: Möglichkeiten der Wissensmodellierung für technische Diagnose-Expertensysteme
92 Seiten

D-92-12
Otto Kühn, Franz Schmalhofer, Gabriele Schmidt: Integrated Knowledge Acquisition for Lathe Production Planning: a Picture Gallery (Integrierte Wissensakquisition zur Fertigungsplanung für Drehteile: eine Bildergalerie)
27 pages

D-92-13
Holger Peine: An Investigation of the Applicability of Terminological Reasoning to Application-Independent Software-Analysis
55 pages

D-92-14
Johannes Schwagereit: Integration von Graph-Grammatiken und Taxonomien zur Repräsentation von Features in CIM
98 Seiten

D-92-15
DFKI Wissenschaftlich-Technischer Jahresbericht 1991
130 Seiten

D-92-16
Judith Engelkamp (Hrsg.): Verzeichnis von Softwarekomponenten für natürlichsprachliche Systeme
189 Seiten

D-92-17
Elisabeth André, Robin Cohen, Winfried Graf, Bob Kass, Cécile Paris, Wolfgang Wahlster (Eds.): UM92: Third International Workshop on User Modeling, Proceedings
254 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-92-18
Klaus Becker: Verfahren der automatisierten Diagnose technischer Systeme
109 Seiten

D-92-19
Stefan Dittrich, Rainer Hoch: Automatische, Deskriptor-basierte Unterstützung der Dokumentanalyse zur Fokussierung und Klassifizierung von Geschäftsbriefen
107 Seiten

D-92-21
Anne Schauder: Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars
57 pages

D-92-23
Michael Herfert: Parsen und Generieren der Prolog-artigen Syntax von RELFUN
51 Seiten

D-92-25
Martin Buchheit: Klassische Kommunikations- und Koordinationsmodelle
31 Seiten

D-92-26
Enno Tolzmann:
Realisierung eines Werkzeugauswahlmoduls mit Hilfe des Constraint-Systems CONTAX
28 Seiten

D-92-27
Martin Harm, Knut Hinkelmann, Thomas Labisch: Integrating Top-down and Bottom-up Reasoning in COLAB
40 pages

Parsen und Generieren der Prolog-artigen Syntax von RELFUN

Michael Herfert

D-92-23
Document