SEKI Working Paper

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany

CALDO - Dokumentation und Analyse von Softwaresystemen

Joachim Steinbach

SEKI Working Paper SWP-89-06

CALDO

-

Dokumentation und Analyse von Softwaresystemen

Harald Sohns & Joachim Steinbach

Diese Arbeit wurde unterstützt vom Sonderforschungsbereich 314, D4-Projekt

Zusammenfassung

CALDO (Computer Aided Lisp Documentation) ist ein Werkzeug zur Dokumentation und Analyse von - in Lisp geschriebenen - Softwaresystemen. Es können sowohl Systeme auf der Basis von Quelltexten, als auch im Lispsystem geladene Programme bearbeitet werden. CALDO besitzt u.a. folgende Fähigkeiten: Ermittlung fundamentaler Daten der Programmstruktur (vorhandene Objekte finden, Aufrufbäume erstellen, etc.), Zusammenfassung von Analyseergebnissen (in Form von Modulbeschreibungen), Zusammenarbeit mit dem Windowsystem (Verwaltung von Quelldateien). Der vorliegende Bericht ist als User-manual gedacht, der anhand vieler Beispiele einen Eindruck der Möglichkeiten von CALDO vermitteln soll.

Inhalt

1 Motivation

1.1 Das Warum

Ein bestehendes Softwaresystem in seiner Funktion und Struktur verstehen zu wollen, kann mit erheblichem Aufwand und einiger Mühe verbunden sein. Dokumentationen sind oft veraltet, das Zusammenwirken der einzelnen Funktionen ist nicht bekannt und in welchem Modul welche Komponente ihren Platz hat, ist oft nur mühsam herauszufinden. Aber auch die Dokumentation eigener Software erfordert Arbeit und wird oft aus Zeitgründen auf das Allernotwendigste beschränkt. Insbesondere werden Dokumentationen vielfach nicht fortlaufend auf dem neuesten Stand gehalten. Bei größeren Softwaresystemen erfordert es außerdem einige Anstrengung, auch nur den Überblick über das bereits Erstellte zu behalten oder wiederzugewinnen, besonders dann, wenn mehrere Personen an der Erstellung des Systems arbeiten, Revisionen oder Erweiterungen anstehen.

Um eine Reihe der oben beschriebenen Schwierigkeiten und Probleme anzugehen, entstand die Idee, dem Programmierer ein computerunterstütztes System zur Seite zu stellen. Dieses System -CALDO, Computer Aided Lisp Documentation- sollte

- die Dokumentation von Programmsystemen vereinfachen,
- die Erstellung und Modifikation von Programmen unterstützen,
- einen schnellen Überblick über ein bestehendes Programm, seine Vollständigkeit und innere Struktur ermöglichen.

Unsere Software entwickeln wir fast ausschließlich in Lisp. Das CALDO-System unterstützt daher die Dokumentation von Lisp-Programmen. Es ist selbst in Lisp geschrieben.

1.2 Erste Ideen

Eine wesentliche Grundidee war, Informationen über alle Objekte des zu dokumentierenden Programms zum einen automatisch zu extrahieren und zum anderen in Tabellen zu halten. Auf diese Weise sind die Informationen schnell verfügbar und der Overhead für neue, aufwendige Analyseprozesse wird gespart. Diese Tabellen wurden, etwas mißverständlich vielleicht, "Aufruftabellen" [engl. calltables] genannt. So heißen sie aber noch heute. Aufruftabellen erwiesen sich später allerdings als etwas schwerfällig und für manche Zwecke schlicht ungeeignet. Zum anderen konnte die Geschwindigkeit der Analyse zum Teil derart gesteigert werden, daß der entsprechende Overhead vernachläßigt werden kann.

Am Anfang war auch nur daran gedacht, Dateien zu analysieren, d.h. die Informationen aus den vorliegenden Quelltexten zu gewinnen. Die Analyse von Objekten im Lispsystem selbst, wie etwa die Funktion 'apropos' oder die Konstante 'call-arguments-limit', wurde erst später mit einbezogen.

So gab es also zunächst einmal ein System, das Lispquelldateien analysieren konnte. Als Ergebnis dieses Analyseprozesses erzeugte es eine oder mehrere Aufruftabellen. Diese Aufruftabellen konnten dann weiterverarbeitet werden und damit als Grundlage weitergehender Analysen dienen. Nun können andere Funktionen mit den Aufruftabellen arbeiten um z.B. Aufrufbäume zu erstellen oder Inhaltsverzeichnisse einzelner Quelldateien auszugeben, ohne daß eine neuerliche Analyse und damit ein neuerliches Einlesen der jeweiligen Datei[en] notwendig ist. Die Aufruftabellen stecken voll von Informationen über die, in den untersuchten Quelldateien, definierten Objekte. Einige der Informationen [direkt aus dem Quelltext extrahiert], die für jedes Objekt dort abgelegt werden, sind beispielsweise

- Name und Typ,
- Name der zugehörigen Quelldatei,
- eventuell vorhandene Dokumentationstexte und
- handelt es sich um eine Funktion [Makro], auch noch eine Liste der im Objekt direkt aufgerufenen Funktionen und Makros,
- sowie die Argumentliste des Objekts.

Um dem Benutzer die Arbeit zu erleichtern, überlegten wir uns eine Reihe kleiner Vereinfachungen. So gibt es beispielsweise eine besondere Aufruftabelle, die "Systemaufruftabelle". Sie ist zwar auch nur eine einfache Aufruftabelle, jedoch einige der Funktionen für weitergehende Analysen verwendet sie als 'Default'. Auf diese Art und Weise erübrigt es sich, beim Aufruf dieser Funktionen jeweils speziell eine Aufruftabelle angeben zu müssen, sofern die Informationen der Systemaufruftabelle verwendet werden. Außerdem kann der Benutzer eine Directory als 'Lispdirectory' auszeichnen und dann einzelne Quelldateien über Symbole ansprechen. Wird also z.B. "/sys/component/" als 'Lispdirectory' bestimmt, so bezeichnet 'myfile die Datei "/sys/component/myfile.lisp".

1.3 Das System wird komplexer

Zwar konnten mit diesem ersten Basissystem schon einige Aufgaben bearbeitet werden und es schien, als sei die Grundidee nicht gänzlich falsch. Mit der Zeit wurden jedoch eine Reihe von Lücken deutlich und es kamen neue Bedürfnisse hinzu. Störend war beispielsweise die Tatsache, daß sich Aufruftabellen nicht automatisch dem jeweils aktuellen Stand der analysierten Datei anpassen. Es ist daher notwendig, sie nach jeder Änderung neu zu erstellen. Zum anderen wurde die Tatsache, daß zwar Definitionen in Quelldateien, nicht jedoch aktuell existente Objekte im Lispsystem untersucht werden konnten, als Manko empfunden. Ein weiteres Problem war es, auch einem ungeübten Benutzer die Fähigkeiten des Systems zugänglich zu machen.

Dies alles führte zu einer Reihe von Erweiterungen. So entstand neben dem Hauptsystem zur Analyse von Quelldateien ein zweites zur Analyse von aktuell im Lispsystem existierenden Objekten (siehe Kapitel 3). Zum anderen wurde eine Menüschnittstelle für die wichtigsten Funktionen erstellt (siehe 1.4). Außerdem wurde für die inkrementelle Nachanalyse ein entsprechendes Werkzeug geschaffen.

Zu diesen grundsätzlichen Erweiterungen sind im Laufe der Zeit eine Reihe weiterer hinzu gekommen, die vor allem die Oberfläche von CALDO betreffen. So gibt es beispielsweise einen ersten Prototyp einer menügesteuerten Oberfläche für das On-line-Dokumentationssystem, den Teil von CALDO, der Objekte direkt im Lispsystem untersucht.

1.4 Die Menüschnittstelle

Um die Merk- und Schreibarbeit bei der Benutzung von CALDO zu reduzieren, haben wir uns eine einfache Menüschnittstelle überlegt. Diese Schnittstelle beruht auf einer losen Kopplung zwischen Lispsystem und Displaymanager und existiert daher auch nur auf den Apollo-Workstations.

Diese Menüschnittstelle erlaubt es, Argumente im Lispfenster auszuwählen und Funktionen bzw. Makros von CALDO auf sie anzuwenden. Argumente können Symbole, Pfadnamen und ähnliches sein (aber keine Listen).

Die wichtigsten Funktionen von CALDO werden in diesem Menü durch Abkürzungen identifiziert. Wir verweisen an den entsprechenden Stellen mit *'Menü:'* darauf. Zur Benutzung der Menüschnittstelle beachte man die folgenden Schritte.

Das Argument eines Kommandos befindet sich im Lisp-Ausgabefenster:

- Cursor über dem Argument positionieren
- TAB-Taste drücken
- im erscheinenden Menü den gewünschten Menüeintrag auswählen (mit Cursor)
- die rechte Maustaste betätigen

Der Ausdruck erscheint daraufhin im Lisp-Eingabefenster und entweder wird er automatisch ausgeführt oder der Benutzer kann noch Änderungen vornehmen (und danach mit return abschicken).

Das Argument befindet sich nicht im Lisp-Eingabefenster:

- Cursor ins Lispeingabefenster bringen
- Ein Blank und dann das Argument eingeben
- und weiter wie oben verfahren

Die Menüschnittstelle wird direkt aktiviert wenn das Dokumentationssystem auf einer Apollo geladen wird. Insbesondere ist zu beachten, daß solange das Menü zu sehen ist, die rechte Maustaste *generell nicht* ihre übliche Bedeutung hat, während die TAB-Taste *nur innerhalb* des Lispfensters in der beschriebenen Weise wirksam ist, d.h. das Dokumentationsmenü erscheinen läßt.

2 Das aktuelle System

2.1 Leistungen und Möglichkeiten

CALDO dient dazu, Softwaresysteme zu dokumentieren und zu analysieren. Dabei können sowohl Softwaresysteme auf der Basis von Quelltexten, als auch im Lispsystem geladene Systeme analysiert und dokumentiert werden.

Welche Elemente und Möglichkeiten beinhaltet nun die Analyse und die Dokumentation mittels CALDO ? Die folgende Tabelle bietet einen Überblick über die wichtigsten Möglichkeiten:

Ermittlung fundamentaler Daten der Programmstruktur :
- vorhandene Quelldateien
- vorhandene Funktionen, Makros, Variablen etc.
- in einer Programmeinheit aufgerufene Funktionen und Makros
- Argumentliste, Dokumentationstexte, Quelltextdatei und Typ von Objekten im Programm

Berechnen weiterführender Kennwerte :
- Aufrufbäume
- für eine Datei benötigte Funktionen
- benötigte, aber nicht definierte Funktionen
- indirekt aufgerufene Funktionen
- alle Funktionen [Makros], die eine bestimmte Funktion [Makro] direkt oder indirekt benötigen

Zusammenfassung der Analyseergebnisse [Reportwriting] :
- in Form einer Funktionendokumentation
- in Form einer ausführlichen Modulbeschreibung
- in Form einer Package-Beschreibung

Direkte Hilfen bei der Erstellung von Programmen :
- Auskünfte über die Quelltextdatei eines Objekts, seine Benutzung und seinen Zweck
- Suche nach Objekten mit bestimmten Eigenschaften
- Zusammenarbeit mit dem Windowsystem, beinhaltend z.B. die Verwaltung von, mit Quelldateien assoziierten Fenstern und die gezielte Bereitstellung von Quelltexten

Im weiteren Text wird noch erläutert, wie die einzelnen Möglichkeiten wahrgenommen werden können. Für fehlende Details sei schon an dieser Stelle auf die technische Dokumentation verwiesen, die als Help-Funktion zur Verfügung steht.

2.2 Installieren und Laden

Das Installieren von CALDO geschieht vergleichsweise einfach. Entweder wird die Datei

/madlener/rrl/user_data/common_lisp/startup.lisp

nach

~user_data/common_lisp/startup.lisp

kopiert oder es wird, falls eine solche Datei schon vorhanden ist, dort die Zeile (require "dok-sys-startup" "/madlener/rrl/user_data/common_lisp/startup") eingefügt. Damit ist das Dokumentationssystem auf der jeweiligen Benutzernummer installiert.

Es gibt bislang zwei verschiedene Möglichkeiten des Einstiegs in das System, die jedoch untereinander verträglich sind. Die beiden Einstiege sind über die Funktion 'main-menu' zu erreichen. Sie wird automatisch beim Lispstart geladen (falls das System installiert wurde).

Um mit dem Dokumentationssystem arbeiten zu können, muß zunächst Lisp gestartet werden. Dann kann das Dokumentationssystem geladen werden, indem das zentrale Menü von Lisp aus mit (main-menu) aufgerufen wird. Dieses Menü hat aktuell zwei Einträge :

0 Load CALDO (Vers. 5.1)
1 Load the On-Line-System (Vers. 1.0)

Ein Eintrag wird durch Angeben der vorangestellten Zahl ausgewählt. Die Auswahl des Eintrags 0 bewirkt das Laden aller für die Analyse von Lispquelldateien wesentlichen Komponenten ins Lispsystem. Die Auswahl des Eintrags 1 bewirkt, daß nur die Komponenten für die Analyse von Objekten im Lispsystem selbst geladen werden.

Während der Ladeoperation wird der Benutzer noch gefragt, ob das Dokumentationssystem mit einem erweiterbaren Aufrufanalysator geladen werden soll. Im Normalfall kann diese Frage verneint werden. Sie bietet aber die Möglichkeit, im Programm definierte, besondere Konstrukte, wie z.B. solche Makros, die sich wie 'special forms' verhalten, durch speziell definierte Methoden korrekt zu analysieren. Übliches Kennzeichen solcher besonderen Konstrukte ist, daß ihre Anwendung Ausdrücke beinhaltet, die wie Funktionsaufrufe aussehen, aber keine sind. Ein Beispiel aus Common-Lisp : (dolist (ai 9) (princ ai) (terpri)) ; (ai 9) sieht wie ein Funktionsaufruf aus, ist aber keiner. Enthält das Programmsystem keine solchen Formen, genügt die Standardanalyse (zur Aufrufanalyse s.a. techn. Doku. 7.4).

2.3 Initialisierung

Hat der Benutzer den Eintrag 0 gewählt, so wird er direkt nach dem Ende des Ladevorgangs gefragt, ob das Dokumentationssystem initialisiert werden soll. Um ein korrektes Funktionieren des Dokumentationssystems zu gewährleisten, muß eine solche Initialisierung erfolgen. Dies geschieht, indem die entsprechende Frage mit y [yes] beantwortet wird. Wird sie mit n [not] beantwortet, so kann das System später mit [init-dok-sys] initialisiert werden. Anschließend erscheint ein weiteres Menü. Dieses Menü hat 4 Einträge :

0 you want to document a new system
1 you have a calltable-file for your system
2 you have a system-specification-file
3 you don't want a system-calltable now

Durch Auswahl der einzelnen Möglichkeiten können nun 4 unterschiedliche Wege des weiteren Vorgehens beschritten werden.

System ohne Beschreibung neu analysieren
Der Benutzer möchte, daß ein neues, noch nicht analysiertes [oder ein wesentlich verändertes] System neu analysiert wird. Dazu wählt er den Eintrag mit der Nummer 0, womit eine neue Analyse initiiert wird. Näheres steht im nächsten Abschnitt.

Vorhandene Aufruftabelle nutzen [keine neue Analyse]
Der Benutzer hat schon früher einmal ein System analysiert und die entstandene Aufruftabelle auf einer Datei abgespeichert. Diese Aufruftabelle möchte er jetzt reaktivieren. In diesem Fall wählt er den Eintrag mit der Nummer 1.

System mit Beschreibung neu analysieren
Der Benutzer möchte ein System analysieren, für das er eine Beschreibung auf einer Datei erstellt hat [s. techn. Doku. 7.3]. Diese Beschreibung kann als Grundlage für die Erstellung der Systemaufruftabelle benutzt werden. Dazu wählt der Benutzer den Eintrag mit der Nummer 2.

Keine Analyse
Der Benutzer hat noch nicht die Absicht ein System zu analysieren oder ein Analyseergebnis zu reaktivieren. Dann kann er den Eintrag mit der Nummer 3 wählen.

2.4 Arbeiten mit CALDO

Was geschieht nun, nachdem der Benutzer eine der oben gezeigten Alternativen ausgewählt hat ? Betrachten wir zunächst die erste Alternative. Das Verhalten des Systems bei den anderen kann dann jeweils als Modifikation angesehen werden. Wir zeigen zur Erläuterung einen Systemlauf von Beginn an, den wir mit entsprechenden Kommentaren versehen haben. Um dem Benutzer am Rechner ein Nachvollziehen zu ermöglichen, wurde eine Directory mit Beispielquelldateien unter dem Namen "/madlener/rrl/dok-sys/mydir/" angelegt. Der Beispiellauf bezieht sich auf das Verhalten des Systems auf einer APOLLO-Workstation (zu Portierungen s. techn. Doku. 9.2). Zeilen mit Benutzereingaben sind mit '*' markiert.

Zunächst wird also Lisp gestartet :

```
;;; DOMAIN/CommonLISP, Development Environment Version 2.10, 21 May 1987
;;;
;;; Copyright (C) 1986 by Apollo Computer Inc. All Rights Reserved
;;;
;;; This software product contains confidential and trade secret information
;;; belonging to Apollo Computer. It may not be copied for any reason other
;;; than for archival and backup purposes.

;;; Loading source file
    "//node_5c4e/madlener/rrl/user_data/common_lisp/startup.lisp"
```

Als nächstes wird das System geladen:

```
* > (main-menu)
    MAIN MENU

      0  Load CALDO (Vers. 5.1)
      1  Load the On-Line-System (Vers. 1.0)

*     You choose number : 0
      ;;; Loading source file "//madlener/rrl/dok-sys/dokload3.lisp"
      Do you want to load the documentation system with the extendable calls-
      analyser ?
*     If you are not sure type n (Y or N) n
```

Dies ist die oben besprochene Frage, ob der erweiterbare Aufrufanalysator geladen werden soll. Die Frage wurde hier mit n (no) beantwortet. Das System wird nun geladen.

```
      loading files for the documentation system vers. 5.1 .....

          Welcome to the Lisp Documentation and Analyzing System
```

Es folgt die Frage, ob das System initialisiert werden soll.

* Do you want to initialize the system ? [Y or N] y

Wird die Frage mit y [yes] beantwortet, so wird eine Initialisierungsroutine aktiviert, die das oben besprochene Menü anbietet.

Initializing the documentation system : choose one of these possibilities

0 you want to document a new system
1 you have a calltable-file for your system
2 you have a system-specification-file
3 you don't want a system-calltable now

* You choose number : 0

Der Benutzer hat nun mit der Eingabe 0 die erste Alternative ausgewählt, d.h. er möchte eine Menge von Lispquelldateien analysieren lassen, die ein zusammenhängendes Lispprogrammsystem beschreiben.

Als nächstes fragt das System nun nach Modulnamen, die in der Form einer Lispliste angegeben werden sollen. Modulnamen können Pfadnamen, Strings oder Symbole sein. Sie kennzeichnen die Namen der zu analysierenden Lispquelldateien. Symbole werden in diesem Zusammenhang oft dazu verwendet, die Namen von Dateien in der sogenannten Lisp-Directory abzukürzen. In unserem Beispiellauf werden wir diese Möglichkeit verwenden. Anstatt einer Liste vollständiger Pfadnamen, geben wir also eine Liste von Symbolen an. Aber auch eine beliebige Mischung von Pfadnamen und Symbolen wäre zuläßig. Wichtig ist, daß sich die Dateien, deren Namen mit Symbolen abgekürzt werden, alle in der Lisp-Directory befinden. Lispquelldateien in dieser Directory können dann innerhalb von CALDO mit entsprechenden Symbolen bezeichnet werden [z.B. die Datei /madlener/rrl/dok-sys/mydir/module1.lisp mit 'module1].

* Your modulenames as a lisp-list : (module1 module2 module3)

Die Liste (module1 module2 module3) beschreibt die Menge der Dateien, die zu unserem Programmsystem gehören. Es kann eine Beschreibung der zugehörigen Dateien in einer sogenannten Systembeschreibung abgelegt werden [s. hierzu techn. Doku. 7.3]. Dies erspart die Eingabe der Namensliste der Dateien. Stattdessen kann man Alternative 2 wählen und gibt dort dann den Namen der Beschreibungsdatei an. Doch zurück zum aktuellen Systemlauf. Als nächstes folgt die Frage nach der Lisp-Directory.

```
Lisp-directory [mit abschließendem /] : /madlener/rrl/dok-sys/mydir/
```

Wir haben in diesem Fall den Directory-Namen als Symbol /madlener/rrl/dok-sys/mydir/ angegeben. Genausogut hätten wir aber auch den String "/madlener/rrl/dok-sys/mydir/" oder den Pfadnamen ¤P"/madlener/rrl/dok-sys/mydir/" verwenden können. Das Dokumentationssystem ist so konstruiert, daß es sich im allgemeinen gegenüber diesen verschiedenen Bezeichnungsmodi bei Directory- und Dateinamen tolerant verhält. Bei Symbolen ist allerdings Vorsicht angebracht, da sie - s. oben - als Dateinamen eine besondere Bedeutung haben. Bei Directory-Namen sollte auch, soweit es das Dokumentationssystem betrifft, darauf geachtet werden, daß der Name mit einem '/' abgeschlossen wird, also /madlener/rrl/dok-sys/mydir/ statt /madlener/rrl/dok-sys/mydir. Das System extrahiert nun aus dem Quelltext Daten, mit denen die Systemaufruftabelle berechnet wird und kehrt im Anschluß an diese Berechnung auf Toplevel zurück.

```
The calculation may take some time...
¤P"/madlener/rrl/dok-sys/dokload3.lisp"
>
```

Zur Erinnerung: Die Systemaufruftabelle ist auch nur eine Aufruftabelle (s. 1.2 letzter Abschnitt). Sie wird aber von einer Reihe von Funktionen bzw. Makros, die wir noch kennenlernen werden, als Default verwendet. Daraus ergibt sich ihre besondere Stellung.

Mit der Berechnung der Systemaufruftabelle ist die Basisanalyse des zu Grunde gelegten Softwaresystems beendet. In der Systemaufruftabelle liegen nun die wesentlichen Daten des Softwaresystems vor. Der Benutzer kann auf diesen Daten aufbauende Analysen anfordern. Änderungen, die nach Erstellung der Systemaufruftabelle im zu Grunde gelegten Softwaresystem erfolgen, finden ohne erneute Analyse allerdings keine Widerspiegelung in der Systemaufruftabelle. Es ist aber eine partielle Neuanalyse möglich (s. hierzu techn. Doku. 7.1.4.2).

Die Alternative 2 unterscheidet sich von Alternative 0 dadurch, daß statt einer Dateinamensliste, eine Spezifikationsdatei angegeben wird, aus der das Dokumentationssystem die Dateinamen extrahiert. Bei der Alternative 1 wird eine zuvor abgespeicherte Aufruftabelle direkt als Systemaufruftabelle geladen. Bei der Alternative 3 erfolgt lediglich eine Basisinitialisierung von CALDO, weiter geschieht nichts.

2.5 Einfache Analysen

Basierend auf den Daten der Systemaufruftabelle können verschiedene Analysen des Sofwaresystems erfolgen. Die folgenden Beispiele sollen einige der typischen Analysen demonstrieren.

1. Zeige alle dem Softwaresystem zugehörigen Dateien

```
> (*filesof)
(#P"//node-lab59/data/rrl/dok-sys/mydir/module1.lisp"
#P"//node-lab59/data/rrl/dok-sys/mydir/module2.lisp"
#P"//node-lab59/data/rrl/dok-sys/mydir/module3.lisp")
```

Menü: Aufruftabellen/*zugehörige Objekte

2. Zeige alle im Softwaresystem definierten Objekte

```
> (*tabledomain)
(FN7 FN8 FN3 FN4 FN5 FN6 FN1 FN2 MKR1)
```

Menü: Aufruftabellen/*zugehörige Dateien

3. Zeige im Softwaresystem definierte Objekte eines bestimmten Typs

3.1 Makros und Funktionen

```
> (typed-tabledomain *systbl :types 'mf)
(MKR1 FN2 FN1 FN6 FN5 FN4 FN3 FN8 FN7)
```

3.2 Variablen

```
> (typed-tabledomain *systbl :types 'var)
NIL
```

Menü: Aufruftabellen/*. . . mit bestimmtem Typ

Objekte anderer Typen können über entsprechende Typoptionen angesprochen werden (s. hierzu techn. Doku. 0.2/Tableautyp).

4. Zeige einzelne Informationen zu Funktionen (Makros)

4.1 Gib Informationen zu Benutzung und Zweck einer Funktion (Makro)

```
> (*dok 'fn1)
function FN1
FN1 dient zum Testen des Systems.
usage : (FN1 A B C)
```

Menü: andere Infos/*Kurzinfo

4.2 Zeige nur das Aufrufformat einer Funktion (eines Makros) im Softwaresystem

```
> (*fn-use 'fn1)
(FN1 A B C)
```

Menü: andere Infos/*Benutzung einer Funktion

'fn1 steht für den Namen des Objekts.

5. Zeige den Namen der Quelltextdatei zu einem Objekt oder falls eine solche aus der Aufruftabelle heraus nicht zu ermitteln ist, das Package dem das Objekt zugehört

```
> (*belongsto 'fn1)
#P"//node-1ab59/data/rr1/dok-sys/mydir/module1.lisp"
> (*belongsto 'car)
#<Package "LISP" D90813>
> (*belongsto '*belongsto)
#<Package "USER" D90CCB>
> (*belongsto 'unknown)
UNKNOWN is not bound
```

Menü: andere Infos/*gehört zu?

6. Zeige den Typ eines Objekts

```
> (fntype (cltbl-assoc1 'fn2 *systbl))
FUNCTION
> (fntype (cltbl-assoc1 'mkr1 *systbl))
MACRO
```

Menü: andere Infos/*Objekttyp

7. Zeige alle von einer Funktion (Makro) aufgerufenen Funktionen und Makros

```
> (*directcalls 'fn8)
(FN4 FN3 FN2 FN1 DOLIST LET)
```

Menü: Aufrufe/*direkte Aufrufe

8. Zeige alle in einer Quelldatei aufgerufenen Funktionen und Makros

```
> (*neededfiles&funs3 'module3)
(#<Package "LISP" D90813> DOLIST LET)
(#P"//node-1ab59/data/rrl/dok-sys/mydir/module1.lisp" FN2 FN1)
(#P"//node-1ab59/data/rrl/dok-sys/mydir/module2.lisp" FN3 FN4)
```

Menü: Dateien/Module/*benötigte Objekte

Es wird hier keine einfache Namensliste ausgegeben, sondern die Funktionen und Makros sind nach der Datei oder dem Package, in dem sie definiert wurden, sortiert. Ist eine Funktion oder ein Makro undefiniert, wird es unter NIL eingeordnet. Im Beispiel steht 'module3 für den jeweiligen Datei- bzw. Modulnamen.

9. Zeige alle einer Quelldatei zugehörigen Objekte

```
> (obj-of 'module2 :cltbl *systbl :types 'all)
(FN6 FN5 FN4 FN3)
```

Menü: Dateien/Module/*enthaltene Objekte

'module2 steht für den Dateinamen

10. Zeige alle aufrufbaren Objekte in einer Quelldatei

```
> (*funsof 'module2)
(FN6 FN5 FN4 FN3)
```

Menü: Dateien/Module/*enthaltene Funktionen

'module2 steht für den Dateinamen

11. Zeige alle in einer Quelldatei definierten Objekte eines bestimmten Typs

11.1 Makros und Funktionen

```
> (obj-of 'module2 :cltbl *systbl :types 'mf)
(FN6 FN5 FN4 FN3)
```

11.2 Variablen

```
> (obj-of 'module2 :cltbl *systbl :types 'var)
NIL
```

Menü: Dateien/Module/*enhaltene Objekte

'module2 steht für den Dateinamen. Objekte anderer Typen können über entsprechende Typoptionen angesprochen werden (s. hierzu techn. Doku. 0.2/Tableautyp).

Es sei nochmals darauf hingewiesen, daß die oben demonstrierten Analysen auf den Daten in der Systemaufruftabelle beruhen. Wurde keine Systemaufruftabelle erstellt, so sind die Analysen unmöglich, d.h. sie brechen mit Fehler ab. Auf die Möglichkeit von Analysen ohne Erstellung von Aufruftabellen wird im Rahmen der Präsentation des sogenannten On-Line-Dokumentationssystems eingegangen (zu einfachen Analysen s.a. techn. Doku. 1.).

2.6 Komplexe Analysen

1. Aufrufbäume

Aufrufbäume können für einzelne Funktionen und Makros aber auch für ein ganzes Softwaresystem erstellt werden.

1.1 Aufrufbäume für einzelne Funktionen und Makros

```
> (*calltreeof 'fn2)
(FN2 (LIST))
> (*ppcalltree 'fn2)
FN2
 LIST
NIL
> (*ppcalltree 'fn8)
FN8
 LET
 DOLIST
 FN1
 FN2
  LIST
 FN3
  FN1
  MKR1
 FN4
  FN3
   FN1
   MKR1
NIL
```

Interpretiert ungefähr : fn8 ruft let, dolist, fn1, fn2, fn3 und fn4. fn3 ruft wiederum fn1 und mkr1. fn4 ruft fn3.

Menü: Aufrufe/*Aufrufbaum

1.2 Aufrufbäume eines ganzen Softwaresystems

Ausgegeben werden nur diejenigen Aufrufbäume, die nicht vollständig in einem anderen Aufrufbaum des Softwaresystems enthalten sind.

```
> (*general-call-trees)
FN7
  FN3
    FN1
    MKR1
```

```
  FN4
    FN3
      FN1
      MKR1
FN8
  LET
  DOLIST
  FN1
  FN2
  FN3
    FN1
    MKR1
  FN4
    FN3
      FN1
      MKR1
NIL
```

Menü: Aufrufe/*sys-Aufrufbaum

2. Ausgabe aller für eine Datei benötigten Dateien

Als für eine Datei nötig werden alle Dateien angesehen, die Objekte definieren, die in der Datei aufgerufen werden. Dabei können auch mittelbar, d.h. über andere Funktionen oder Makros aufgerufene Objekte berücksichtigt werden. Wichtig ist, daß Abhängigkeiten, die über die Benutzung von gemeinsamen Variablen entstehen, nicht berücksichtigt werden.

2.1 Direkt benötigte Dateien

```
> (*neededfiles&funs3 'module2)
(#P"//node-1ab59/data/rrl/dok-sys/mydir/module2.lisp" FN5 FN6 FN3)
(#P"//node-1ab59/data/rrl/dok-sys/mydir/module1.lisp" MKR1 FN1)
(#<Package "LISP" D90813> IF * 1- ZEROP 1+)
NIL
```

Ausgegeben wird hier nicht eine einfache Liste von Dateinamen, sondern es werden die benötigten Objekte nach Quelldatei bzw. Package sortiert ausgegeben.

2.2 Direkt und indirekt benötigte Dateien

```
> (*neededfiles&funs3 'module3)   ;; - direkt benötigte Dateien
(#<Package "LISP" D90813> DOLIST LET)
(#P"//node-1ab59/data/rrl/dok-sys/mydir/module1.lisp" FN2 FN1)
(#P"//node-1ab59/data/rrl/dok-sys/mydir/module2.lisp" FN3 FN4)
NIL
> (*neededfiles&funs3 'module3 :direct-fl nil)   ;; - direkt und indirekt
                                                 ;; benötigte Dateien
(#<Package "LISP" D90813> LIST DOLIST LET)
(#P"//node-1ab59/data/rrl/dok-sys/mydir/module1.lisp" MKR1 FN1 FN2)
(#P"//node-1ab59/data/rrl/dok-sys/mydir/module2.lisp" FN4 FN3)
NIL
```

Menü: Dateien/Module/*benötigte Objekte

3. Ausgabe aller für ein Softwaresystem direkt benötigten aber nicht im Softwaresystem definierten Objekte

```
> (*foreign-calls)
(LIST * IF 1+ ZEROP 1- LET DOLIST)
> (*foreign-calls t)
NIL
```

Menü: Aufruftabellen/*fremde Aufrufe

Die Angabe von t als Parameter bewirkt, daß Lispstandardfunktionen und -makros als definiert betrachtet werden.

4. Ausgabe aller für eine Datei im Softwaresystem direkt benötigten aber nicht im Softwaresystem definierten Objekte

```
> (mapcan #'(lambda (ai) (*fun-foreign-calls ai)) (*funsof 'module2))
(ZEROP 1- x IF 1- ZEROP 1+ IF)
> (mapcan #'(lambda (ai) (*fun-foreign-calls ai t)) (*funsof 'module2))
NIL
```

Die Angabe von t als Parameter bewirkt, wie bei 3., daß Lispstandard-funktionen und -makros als definiert betrachtet werden.

5. Ausgabe aller direkt oder mittelbar von einer Funktion (Makro) aufgerufenen Funktionen und Makros

```
> (*usedfuns 'fn8)
(FN3 FN4 FN2 DOLIST LET FN1 MKR1 LIST)
```

Menü: Aufrufe/*und indirekte

Vgl. 1.2 in diesem und 8 im vorigen Abschnitt

6. Ausgabe aller Funktionen und Makros im Softwaresystem, die eine bestimmte Funktion (Makro) direkt oder indirekt aufrufen

6.1 Die eine bestimmte Funktion (Makro) direkt aufrufen

```
> (*whocalls 'fn1)
(FN3 FN8)
```

Menü: aufrufende fns/*wer ruft Objekt?

6.2 Die eine bestimmte Funktion (Makro) direkt oder indirekt aufrufen

```
> (*whoneeds 'fn1)
(FN4 FN7 FN3 FN8)
```

Menü: aufrufende fns/*wer braucht Objekt?

Vgl. 1.2 in diesem Abschnitt

7. Ausgabe der aus einer Datei im Softwaresystem in andere Dateien des Softwaresystems exportierten Objekte

```
> (*calc-export-list 'module1)
([#P"//node-1ab59/data/rrl/dok-sys/mydir/module2.lisp" FN1 MKR1)
(#P"//node-1ab59/data/rrl/dok-sys/mydir/module3.lisp" FN2 FN1)]
```

Menü: Dateien/Module/*exportierte Objekte

Auch für die komplexeren Analysen gilt das bereits zu den einfacheren Analysen Gesagte, nämlich die Abhängigkeit von der Korrektheit und Vollständigkeit der Daten in der Systemaufruftabelle (zu den komplexeren Analysen s.a. techn. Doku. 2.).

2.7 Erstellen von Dokumentationen

Reports und Dokumentationen fassen Informationen und Analysen in festen Formaten zusammen. In unserem Fall dienen sie vor allem der Ergänzung einer technischen Dokumentation und als Grundlage oder als Referenz für ein Benutzermanual. Insbesondere sollen die vorgestellten Werkzeuge es ermöglichen, technische Dokumentationen auf einfache Art und Weise fortlaufend auf dem neuesten Stand zu halten.

1. Erstellen einer Dokumentation zu einer Menge von Objekten

```
> (*dok '(fn7 fn8) :all-information t)
function FN7
FN7 benutzt indirekt Funktionen aus module1 und module2.
usage : (FN7 A B C &OPTIONAL (D (FN3 1 2 3)))
FN7 belongs to #P"//node-1ab59/data/rrl/dok-sys/mydir/module3.lisp".
It calls :  FN4  FN3 .

function FN8
FN8 benutzt direkt Funktionen aus module1 und module2.
usage : (FN8 A B C &OPTIONAL (D (FN3 1 2 3)))
FN8 belongs to #P"//node-1ab59/data/rrl/dok-sys/mydir/module3.lisp".
It calls :  FN4  FN3  FN2  FN1  DOLIST  LET .
```

'(fn7 fn8) steht für die einzusetzende Objektliste

2. Erstellen einer Dokumentation oder Beschreibung zu einer Quelldatei

2.1 Moduldokumentation

2.1.1 Informationen aus der Systemaufruftabelle

```
> (make-documentation :modulename 'module3)
  DOCUMENTATION FOR #P"/madlener/rrl/dok-sys/mydir/module3.lisp"
  ************************************************************************

  FN7                                                          function

      CALLS : FN4 FN3

      IS CALLED BY :

      benutzt indirekt Funktionen aus module1 und module2.
```

FN7 : [A B C &OPTIONAL [D [FN3 1 2 3]]] →

FN8 function

CALLS : FN4 FN3 FN2 FN1 DOLIST LET

IS CALLED BY :

benutzt direkt Funktionen aus module1 und module2.

FN8 : [A B C &OPTIONAL [D [FN3 1 2 3]]] →

Menü: Dateien/Module/*make documentation

2.1.2 Informationen direkt aus der Datei

```
> [*make-documentation : parobj
                         [get-lisp-filename-from-modulename 'module3]]
DOCUMENTATION FOR #P"/madlener/rrl/dok-sys/mydir/module3.lisp"
**************************************************************************
```

FN7 function

CALLS : FN4 FN3

IS CALLED BY :

benutzt indirekt Funktionen aus module1 und module2.

FN7 : [A B C &OPTIONAL [D [FN3 1 2 3]]] →

FN8 function

CALLS : FN4 FN3 FN2 FN1 DOLIST LET

IS CALLED BY :

benutzt direkt Funktionen aus module1 und module2.

FN8 : [A B C &OPTIONAL [D [FN3 1 2 3]]] --»

Wird der key-Parameter 'alphabetic' auf t gesetzt, so werden die Objekte in alphabetischer Reihenfolge dokumentiert.

2.2 Modulbeschreibung

```
> (*describe-module 'module2)
Software Identification

    KIND : module
    SHORT TITLE : MODULE2
    ACTUAL SOURCE FILE : #P"/madlener/rrl/dok-sys/mydir/module2.lisp"
    PREFIXES/NICKNAMES :
        FN6 is not bound
        FN5 is not bound
        FN4 is not bound
        FN3 is not bound
    PACKAGE :  NIL

    Last version is from 15.12.1988 (15:36)
    Actual length is 405 bytes
    includes  4 functions

    SUPERIORS : #<Package "LISP" D90813> MODULE1
    INFERIORS : MODULE3
    OTHER DEPENDENCIES :

    IMPORTS FROM #<Package LISP D90813> : 1- ZEROP 1+ IF *

    IMPORTS FROM MODULE1 : FN1 MKR1

    EXPORTS TO MODULE3 : FN3 FN4


    DATASTRUCTURES :

    TASK :

    CONDITIONS :

    REFERENCES :

    DESIGNER : NIL
```

Menü: Dateien/Module/*beschreibe Modul

3. Erstellen einer Dokumentation zu einem Package

```
> (make-documentation 'user)
DOCUMENTATION FOR package USER
**************************************
```

The package USER
There are 1072 symbols present of which
0 are external symbols, and 0 are shadowing symbols;
Superiors: ("LISP" "FLAVORS" "SYSTEM")
Inferiors: NIL

ADD-ITEM function

CALLS : NCONC PUSH LIST MAKE-ACTUAL-MENU CAR IF

ADD-ITEM : (ITEM) →

CALLTABLE-APPEND function

CALLS : APPEND FUNCTION APPLY

setzt mehrere DOK::Aufruftabellen zu einer zusammen (nicht destruktiv)

CALLTABLE-APPEND : (&REST CTABLELIST) →

PBUFS—$CREATE function

CALLS :

PBUFS—$CREATE : (BUFFER-NAME BUFFER-TYPE STREAM-ID STATUS) →

.
.
.

Menü: Dateien/Module/*make documentation

4. Erstellen einer Dokumentation zu einem Softwaresystem

```
> (*make-documentation)
DOCUMENTATION FOR #P"/madlener/rrl/dok-sys/mydir/module3.lisp"
**************************************************************************

FN7                                                          function
===

    CALLS : FN4 FN3

    IS CALLED BY :

    benutzt indirekt Funktionen aus module1 und module2.

    FN7 : (A B C &OPTIONAL (D (FN3 1 2 3))) →


FN8                                                          function
===

    CALLS : FN4 FN3 FN2 FN1 DOLIST LET

    IS CALLED BY :

    benutzt direkt Funktionen aus module1 und module2.

    FN8 : (A B C &OPTIONAL (D (FN3 1 2 3))) →


DOCUMENTATION FOR #P"/madlener/rrl/dok-sys/mydir/module2.lisp"
**************************************************************************

FN3                                                          function
===

    CALLS : FN1 MKR1

    IS CALLED BY : FN4 FN8 FN7

    benutzt Funktionen in module1.

    FN3 : (A B C) --»
```

Menü: Aufruftabellen/*dokumentiere das System

[Zum Erstellen von Reports und Dokumentationen s.a. techn. Doku. 3. und 4.5].

Noch ein Hinweis : Die in den drei vorhergehenden Abschnitten demonstrierten Analysen und Dokumentationsmöglichkeiten können auch auf der Basis beliebiger Aufruftabellen erfolgen und müssen nicht notwendigerweise die Systemaufruftabelle benutzen. Für die Details sei auf die angegebenen Abschnitte in der technischen Dokumentation verwiesen.

Eine Aufruftabelle kann im übrigen nicht nur über das Eingangsmenü, sondern auch über eine Reihe von Dienstfunktionen aus verschiedenen Objekten erstellt oder geladen werden [s. hierzu techn. Doku. 7.1.3 und 7.1.5].

3 Das On-Line-Dokumentationssystem

Das On-Line-Dokumentationssystem dient zur Analyse und Dokumentation aktuell im Lispsystem existenter Objekte. Dabei ist es für alle einfachen und die hier gezeigten komplexeren Analysen nicht notwendig, eine Aufruftabelle zu erstellen.

Ist ein Softwaresystem in einer Quelltextversion geladen, so kann es z.T. auch mit dem On-Line-Dokumentationssystem analysiert werden (aus dem Binärcode können bislang noch keine Informationen gewonnen werden). Ein wesentlicher Vorteil gegenüber der zuvor vorgestellten Methode liegt darin, daß die Analyseergebnisse den jeweils aktuellen Zustand des geladenen Systems reflektieren, sich also quasi selbsttätig auf dem Laufenden halten.

3.1 Laden des On-Line-Systems

Um mit CALDO arbeiten zu können, muß es geladen werden. Ist es bereits vollständig geladen, ist keine erneute Ladeaktion notwendig, da CALDO das On-Line-Dokumentationssystem enthält.

Das On-Line-Dokumentationssystem wird geladen, indem das bereits bekannte zentrale Menü (s. 2.2) von Lisp aus mit (main-menu) aktiviert wird. Dieses Menü hat, wie schon besprochen, zwei Einträge :

0 Load CALDO (Vers. 5.1)
1 Load the On-Line-System (Vers. 1.0)

Das On-Line-Dokumentationssystem wird durch Eingabe der Zahl 1 geladen.

Direkt nach dem Laden sind die meisten der nachfolgenden Analysen möglich. Einige sind sogar unabhängig vom On-Line-Dokumentationssystem möglich, da schon in Standard-Common-Lisp vorgesehen, andere hingegen benötigen das gesamte Dokumentationssystem. Im Text wird jeweils auf entsprechende Voraussetzungen hingewiesen.

3.2 Einfache Analysen

1. Geladene Dateien und Module, vorhandene Packages

1.1 Zeige die Pfadnamen der den aktuell geladenen Dateien zugehörigen Quelldateien (nur Apollo)

```
> (loaded-files)
(#P"//node-lab59/data/rrl/user-data/common-lisp/startup.lisp" ...
#P"//node-lab59/data/rrl/dok-sys/glisp2.lisp")
```

Menü: Dateien/Module/geladene Dateien

1.2 Zeige alle aktuell vorhandenen Packages (Common-Lisp)

```
> (list-all-packages)
(#<Package "USER" D90CCB> #<Package "LISP" D90813> #<Package
"FLAVORS" D9090B> #<Package "KEYWORD" D90E13> #<Package "LUCID"
D90003> #<Package "SYSTEM" D90843> #<Package "FLAVOR-INTERNALS"
D90953>)
```

1.3 Zeige die aktuell geladenen Module (COMMON-Lisp)

```
> *modules*
("new-calls-analyser-info" "new-calls-analyser-ext" "doky" "dok5" "dok4"
"dok3" "lisp-wpos" "doku2" "window-interface2" "window-interface"
"calltable" "lsp-modules" "dok-service-routines" "key-def-manager"
"p-buffer" "printing" "dm-interface" "strings" "calls-analyser"
"modules-and-files" "dok-sys-parameter" "dok-types-and-options" "glisp2"
"on-line-dok" "interface1")
```

2. Definierte und erreichbare Objekte

2.1 Zeige alle aktuell in einem Package definierten Symbole

Dafür kann der Ausdruck

```
(let (erg (pck «package»))
    (do-symbols (ai pck erg)
                (if (eq (symbol-package ai) pck) (push ai erg))))
```

benutzt werden (Common-Lisp).

2.2 Zeige alle in einem Package momentan erreichbaren Symbole

Dafür kann der Ausdruck

```
(let (erg (pck <package>))
    (do-symbols (ai pck erg) (push ai erg)))
```

benutzt werden (Common-Lisp).

2.3 Zeige die Namen aller in einem Package definierten Objekte

D.h. zeige all jene Symbole, denen ein typisierbares Objekt zugeordnet werden konnte. Voraussetzung dafür ist, daß das Dokumentationssystem geladen ist.

```
> (capropos-list nil :t-p 'user :all-syms nil)
(+INTERN-TABLEAU-TYPES CALLTABLE PAD-$DEF-PFK PAD-DM-CMD ...
SYMBOLLP CSTRING FN-TABLEAU-TYPE *DOMAIN-COUNT)
```

'user steht für ein beliebiges Package

3. Zeige in einem Package definierte Objekte eines bestimmten Typs

3.1 Makros und Funktionen

Voraussetzung : Dokumentationssystem ist geladen

```
> (tbleorpckdomain 'user :types '(macro function) :lispflg nil)
(PAD-$DEF-PFK PAD-DM-CMD POSITIVE GET-KEY-DEF STD-TYPE-TEST-
FUNCTION ... SYMBOLLP CSTRING FN-TABLEAU-TYPE *DOMAIN-COUNT)
```

'user steht für ein beliebiges Package

3.2 Variablen

Voraussetzung : Dokumentationssystem ist geladen

```
> (tbleorpckdomain 'user :types 'var :lispflg nil)
(*ADDITIONAL-GENERAL-DOK-OPTIONS ... *SYSTBL)
```

'user steht für ein beliebiges package

Objekte anderer Typen können über entsprechende Typoptionen angesprochen werden (s. hierzu techn. Doku. 0.2/Tableautyp).
(s. a. Suche nach Objekten mit bestimmten Eigenschaften)

4. Zeige die Argumentliste einer Funktion oder eines Makros (Common-LISP)

```
> (arglist 'belongsto)
(OBJ &OPTIONAL (CALLTABLE NIL) (PCKFLG NIL))
```

'belongsto steht für den Namen des Makros oder der Funktion

5. Zeige den Dokumentationstext zu einem Objekt und gib Informationen zur Benutzung des Objekts

```
> (idok 'belongsto)
function BELONGSTO liefert die Datei oder das Package zu dem obj gehört.
usage : (BELONGSTO OBJ &OPTIONAL (CALLTABLE NIL) (PCKFLG NIL))
```

Menü: andere Infos/Kurzinfo

6. Zeige den Namen der Quelltextdatei zu einem Objekt

```
> (idok 'belongsto :opt 'from)
loaded from #P"//node-1ab59/data/rrl/dok-sys/doku2.lisp"

> (idok 'test-fn :opt 'from)
defined at top-level

> (idok 'car :opt 'from :types 'function)
no source file recorded

> (idok 'unknown :opt 'from)
NIL
```

Menü: andere Infos/gehört zu?

7. Zeige den Typ eines Objekts

```
> (idok 'belongsto :opt 'type)
function BELONGSTO
> (idok 'car :opt 'type)
function CAR
compiler-macro CAR
> (idok 'unknown :opt 'type)
NIL
```

8. Zeige alle von einer Funktion (Makro) aufgerufenen Funktionen und Makros

Voraussetzung : Quelltext der Funktion oder des Makros ist im Lispsystem verfügbar. Falls das gesamte Dokumentationssystem geladen ist :

```
> (directcalls 'belongsto)
(BOUNDP OR SYMBOL-PACKAGE AND FORMAT FNFILE CLTBL-ASSOC1
NOT LET FBOUNDP IF)
```

sonst :

```
> (idok 'belongsto :opt 'calls)
The function calls : BOUNDP OR SYMBOL-PACKAGE AND FORMAT FNFILE
CLTBL-ASSOC1 NOT  LET  FBOUNDP  IF .
```

Menü: Aufrufe/direkte Aufrufe

9. Zeige den internen Quelltext eines Objekts

```
> (idok 'cstring :opt 'source-code)
(NAMED-LAMBDA CSTRING
            (OBJECT)
            (BLOCK CSTRING
              (TYPECASE OBJECT
                (STRING OBJECT)
                (NULL "")
                (SYMBOL (STRING OBJECT))
                (T (WRITE-TO-STRING OBJECT)))))
```

10. Zeige das aktuelle Package eines Objekts

```
> (idok '+ :opt 'in)
actual package is #<Package LISP D90813>.
```

oder (COMMON-Lisp) :

```
> (symbol-package 'car)
#<Package "LISP" D90813>
```

11. Analysiere den Wert eines Objekts

```
> (setq test-var ¤(a b c))
¤<Simple-Vector T 3 12EC61B>
> (idok 'test-var :opt '(def val-type ext-val) :types 'var)
actual value is : ¤<Simple-Vector T 3 12EC61B>
an extended value-representation is : ¤( A B C )
value-type is : SIMPLE-VECTOR
```

Spezielle Analysemethoden sind bisher u.a. für hash-tables, Vektoren und sonstige Arrays implementiert.

12. Ermittle Subtypes und Supertypes eines Typs

```
> (idok 'integer :opt 'types :types 'type)
subtypes : SIGNED-BYTE UNSIGNED-BYTE MOD BIT BIGNUM INTEGER
FIXNUM NIL
supertypes : SIGNED-BYTE COMMON RATIONAL ATOM NUMBER INTEGER T
These types may be supertypes : SEQUENCE LIST
```

13. Analysiere einen Struct-type

```
> (idok 'random-state :opt 'all :types 'struct-type)
struct-type RANDOM-STATE
usage : ¤S( RANDOM-STATE ... )
actual package is ¤<Package LISP D90813>.
no source file recorded
constructor-function is LUCID::MAKE-RANDOM-OBJECT.
selector-function for slot nr. 0 (LUCID::J) is LUCID::RANDOM-STATE-J.
selector-function for slot nr. 1 (LUCID::K) is LUCID::RANDOM-STATE-K.
selector-function for slot nr. 2 (LUCID::SEED) is LUCID::RANDOM-STATE-SEED.
Test-function : ¤'(lambda (ai aj) (typep ai 'RANDOM-STATE))
```

Zu den einfachen Analysen, insbesondere zur Funktion idok, s.a. Dokumentationsoptionen im Abschnitt 0.2 der technischen Dokumentation, sowie die Abschnitte 1.3.1, 1.3.4 bis 1.3.7 und 4.3.1 .

Die oben gezeigten Dokumentationsoptionen (Parameter opt von idok) sind in Optionslisten kombinierbar (z.B. '(type from)).

3.3 Komplexe Analysen

1. Aufrufbäume für einzelne Funktionen und Makros

Voraussetzung : Quelltexte der beteiligten Funktionen und Makros sind im Lispsystem verfügbar und das Dokumentationssystem ist geladen.

```
> (ppcalltree 'belongsto)
; mit Lispstandardfunktionen und -makros
BELONGSTO
  IF
  FBOUNDP
  LET
  NOT
  CLTBL-ASSOC1
    IF
    ATOM
    CAR
    DECLARE
    WHEN
    NULL
    GO
    SETQ
    LUCID::ARGERR
    LET*
    LOCALLY
    ASSOC
    CDR
  FNFILE
    IF
    ATOM
    CAR
    DECLARE
    WHEN
    NULL
    SECOND
    GO
    SETQ
    LUCID::ARGERR
    LET*
    LOCALLY
    CDR
  FORMAT
  AND
  SYMBOL-PACKAGE
  OR
  BOUNDP
```

```
> (ppcalltree 'belongsto nil t)
; ohne Lispstandardfunktionen und -makros
BELONGSTO
  FNFILE
    LUCID::ARGERR
  CLTBL-ASSOC1
    LUCID::ARGERR
```

Menü: Aufrufe/Aufrufbaum

2. Abhängige Packages

2.1 Ausgabe einer Liste der von einem Package direkt benutzten Packages (Common-Lisp)

```
> (package-use-list 'user)
(#<Package "LISP" D90813>                #<Package "FLAVORS" D9090B>
#<Package "SYSTEM" D90843>)
```

2.2 Ausgabe einer Liste der Packages, die ein bestimmtes Package direkt benutzen (Common-Lisp)

```
> (package-used-by-list 'lisp)
(#<Package "FLAVORS" D9090B> #<Package "FLAVOR-INTERNALS" D90953>
#<Package "LUCID" D90003>                #<Package "USER" D90CCB>
#<Package "SYSTEM" D90843>)
```

3. Ausgabe aller in einem Package direkt oder indirekt aufgerufenen Objekte

Voraussetzung : Quelltexte der im Package geladenen Funktionen und Makros sind im Lispsystem verfügbar und das Dokumentationssystem ist geladen.

3.1 Direkt aufgerufene Objekte

```
(mapcan #'cdr (neededfiles&funs1 <package>))
```

Diese Art der Berechnung ist allerdings aktuell noch sehr aufwendig und zeitintensiv. Hier ist es wohl meistens zweckmäßiger mit Aufruftabellen zu arbeiten.

3.2 Direkt und indirekt benötigte Objekte

```
(mapcan #'cdr (neededfiles&funs1 <package> :direct-fl nil))
```

Hier gilt um so mehr das unter 3.1 Gesagte.

4. Ausgabe gerufener Objekte, die nicht im gleichen Package oder im Lispsystem definiert sind bzw. keine Lispstandardfunktionen und -makros darstellen

Voraussetzung : Quelltexte der beteiligten Funktionen und Makros sind im Lispsystem verfügbar und das Dokumentationssystem ist geladen.

4.1 Nicht im gleichen Package als aufrufbar definiert

(foreign-calls <package> nil)

4.2 Nicht im gleichen Package als aufrufbar definiert und keine Lispstandard-funktion (-makro)

(foreign-calls <package>)

4.3 Im Package gerufen und nicht im Lispsystem als aufrufbar definiert

(foreign-calls <package> t nil t)

4.4 Im Lispsystem laut Quelltext gerufen aber dort nicht als aufrufbar definiert

(foreign-calls nil t t t)

5. Ausgabe aller direkt oder mittelbar von einer Funktion (Makro) aufgerufenen Funktionen und Makros

Voraussetzung : Quelltexte der beteiligten Funktionen und Makros sind im Lispsystem verfügbar und das Dokumentationssystem ist geladen.

```
> (usedfuns 'directcalls)
; mit Lispfunktionen und -makros
;;; Warning: no source-code available for LUCID::ARGERR
(THIRD EQ UNBQ CADR ... SETQ GO)

(rem-lspfns (usedfuns 'directcalls))
; ohne Lispfunktionen und -makros
;;; Warning: no source-code available for LUCID::ARGERR
(LUCID::ARGERR HCALLS MARK-CALL ... UNBQ)
```

Menü: Aufrufe/. . . und indirekte

6. Ausgabe aller Funktionen und Makros, die eine bestimmte Funktion (Makro) direkt oder indirekt aufrufen

Voraussetzung : Quelltexte der beteiligten Funktionen und Makros sind im Lispsystem verfügbar und das Dokumentationssystem ist geladen.

6.1 Direkt aufrufende Funktionen und Makros

- Im gesamten Lispsystem

```
> (whocalls 'belongsto)
;;; Warning: whocallsdirect>> This function may run a long time and
;;; consume a lot of memory
(SHOW-FUNCTION-WINDOW *BELONGSTO ... FILESOF)
```

- In einem bestimmten Package

```
> (whocalls 'belongsto 'user)
;;; Warning: whocallsdirect>> This function may run a long time and
;;; consume a lot of memory
(SHOW-FUNCTION-WINDOW *BELONGSTO ... FILESOF)
```

Menü: aufrufende fns/wer ruft Objekt?

6.2 Direkt oder indirekt aufrufende Funktionen und Makros

- Im gesamten Lispsystem

```
> (whoneeds 'belongsto)
;;; Warning: whoneeds>> This function may run a long time and consume
;;; a lot of memory
(*CALC-IMPORTER-LIST *NEEDEDFILES&FUNS3 ... FILESOF)
```

- In einem bestimmten Package

```
> (whoneeds 'belongsto 'user)
;;; Warning: whoneeds>> This function may run a long time and consume
;;; a lot of memory
(*CALC-IMPORTER-LIST *NEEDEDFILES&FUNS3 ... FILESOF)
```

Menü: aufrufende fns/wer braucht Objekt?

Die Analysen unter 6 sind recht aufwendig und speicherplatzintensiv. Dies gilt insbesondere für 6.2. Daher ist es für diese Art von Analysen oft zweckmäßig eine Aufruftabelle zu benutzen, insbesondere wenn sie häufiger durchgeführt werden und das System nicht ständig modifiziert wird.

Zu den komplexeren Analysen s. die Abschnitte 2.3.1 (zu 1), 2.1.3 (zu 3), 2.1.2 und 2.3.5 (zu 4), 2.3.2 (zu 5), 2.3.3 und 2.3.4 (zu 6) in der techn. Dokumentation.

4 Suche nach Objekten mit bestimmten Eigenschaften

Unter den Eigenschaften eines Objekts verstehen wir beispielsweise

- Typ,
- zugehörige Quelldatei,
- zugehöriger Dokumentationstext,
- Struktur des Objekts und
- aktuelles Package.

CALDO erlaubt es nun, nach Objekten mit einer spezifischen Kombination solcher Eigenschaften im Lispsystem oder einer Aufruftabelle zu suchen. Die dafür verwendeten Funktionen sind capropos und capropos-list. Capropos-list gibt die gefundenen Objekte als Liste zurück, während capropos selbst Informationen zu den gefundenen Objekten ausgibt und als Funktionsergebnis nil zurückliefert.

Capropos bietet eine große Anzahl von Kombinationsmöglichkeiten für die Suche und Aufbereitung von Ergebnissen. Diese Vielfalt kann am Anfang verwirrend sein. Im weiteren werden die wesentlichsten Optionen daher allmählich eingeführt.

Zunächst ist es vielleicht hilfreich zu wissen, daß capropos sich ähnlich wie apropos in Common-Lisp verhält, also zur Suche nach Symbolen benutzt werden kann, deren Namen einen bestimmten String als Teilstring enthalten. Capropos ist aber schon dadurch vielseitiger, daß nicht nur in Packages gesucht werden kann, sondern auch in Aufruftabellen und daß es möglich ist, mehrere Suchstrings zu kombinieren.

Wie geschieht nun die Kombination? Capropos kennt drei verschiedene Arten von Kombinationen von keys (wie die Suchstrings oder -symbole im weiteren genannt werden): and-keys, or-keys und not-keys. Diese Kombinationen können auch als keysets aufgefaßt werden. Alle keys in and-keys sollen im Namen des gesuchten Objekts erscheinen, ist or-keys nicht leer, dann auch mindestens einer der keys in or-keys und es soll kein key aus not-keys erscheinen. Damit ist es nun z.B. möglich mit zwei keys gleichzeitig zu suchen oder nur Symbole zuzulassen, die beide Suchkeys enthalten etc. .

Bsp. :

```
> (capropos '("PACKAGE" "USE"))
PACKAGE-USE-LIST, function (PACKAGE)
PACKAGE-USED-BY-LIST, function (PACKAGE)
USE-PACKAGE, function (PACKAGES-TO-USE
                &OPTIONAL (PACKAGE (GET-DEFAULT-PACKAGE)))
UNUSE-PACKAGE, function (PACKAGES-TO-UNUSE &OPTIONAL ...)
> (capropos nil :or-keys '("VALUE-BIND" "FUNCALL"))
*REM-FUNCALLS, macro (&OPTIONAL (FUNS (QUOTE *FUNS))
                  (EXTCALLTABLE (QUOTE *SYSTBL)))
REM-FUNCALLS, function (EXTCALLTABLE &OPTIONAL (FUNS *FUNS))
FUNCALL, function NIL
MULTIPLE-VALUE-BIND, macro (NAMES FORM &BODY BODY ...)
```

Mit den not-keys ist es sehr einfach, bestimmte Symbole auszuschließen :

```
;; Suche nach Symbolen ohne Vokale
> (capropos nil :not-keys '("A" "E" "U" "O" "I"))
*SYSTBL, global
CLTBL?, function (ARG)
TR-G-TT-B-TT, function (G-T-TYPE)
///, global
GCD, function (&OPTIONAL (INTEGER1 0) (INTEGER2 0 I2P)
      &REST MORE-INTEGERS)
**, global
.
.
.
WD, function (&OPTIONAL DIRECTORY-PATHNAME)
PWD, function NIL
ND, function (&OPTIONAL DIRECTORY-PATHNAME)
```

Der Typ eines Objekts bietet eine weitere Möglichkeit der Restriktion :

```
;; Suche nach Variabeln mit Teilstring tbl oder table
> (capropos nil :or-keys '("TBL" "TABLE") :type 'var)
*SYSTBL, global
*SYSTBL, parameter
*VALUE-DOC-METHOD-TBL, global
*VALUE-DOC-METHOD-TBL, parameter
.
.
.
*READTABLE*, global
*READTABLE*, parameter
```

Als Typ kann jeder sogenannte Tableautyp benutzt werden (also z.B. macro, function, constant) (zu Tableautypen s. 0.2 in der techn. Dokumentation). Statt die Objekte auszugeben, die dem angegebenen Typ zugehören, können auch gerade die Objekte ausgegeben werden, für die das nicht der Fall ist.

```
;; Suche nach Objekten, die keine Funktionen oder Makros sind und deren
;; Namen den Teilstring "ADD" enthalten
> (capropos "ADD" :in nil :type 'mf)
*ADDITIONAL-TABLEAU-TYPES-TABLE, global
*ADDITIONAL-GENERAL-DOK-OPTIONS, global
```

Objekte, denen kein Tableautyp zugeordnet werden konnte, werden normalerweise nicht aufgeführt. Sie können aber berücksichtigt werden, wenn der Parameter other-syms auf t gesetzt wird.

```
;; Suche nach allen Objekten, denen kein Typ zugeordnet werden kann
;; und die den Teilstring "AB" enthalten
> (capropos "AB" :type nil :other-syms t)
TABLE
INTPR-CALLABLE
.
.
.
ABST
VARIABLE
```

Wird nichts anderes angegeben, dann sucht capropos im Package 'user. Es können aber auch ein oder mehrere Packages oder Aufruftabellen angegeben werden, in denen gesucht werden soll. Dies geschieht über den Parameter t-p. Wird in t-p eine Liste übergeben, muß t-p-list auf t gesetzt werden.

```
;; Suche in den Packages system und flavors nach Symbolen mit dem
;; Teilstring "DOC"
> (capropos "DOC" :t-p '(system flavors) :t-p-list t)
SET-DOCUMENTATION, function (SYMBOL DOC-TYPE DOC-STRING)
DOCUMENTATION, function (SYMBOL DOC-TYPE)
DOCUMENTATION, function (SYMBOL DOC-TYPE)
.
.
.
;; DOCUMENTATION erscheint mehrfach, da es sowohl vom Package
;; flavors als auch vom Package system aus erreichbar ist.
```

Wie im Beispiel zu sehen ist, kann es zu Doppelnennungen kommen, wenn in mehreren Packages gesucht wird, da normalerweise alle erreichbaren Symbole untersucht werden. Oft ist es erwünscht, nur die in einem Package definierten Symbole zu betrachten. Dazu wird der Parameter all-syms auf nil gesetzt.

```
;; Suche nach Symbolen mit "CDR" als Teilstring, die in Package user selbst
;; definiert sind
> (capropos "CDR" :all-syms nil)
NIL
;; kein Symbol mit Teilstring "CDR" in user selbst definiert
```

Eine andere Möglichkeit die verwendeten Werte einzuschränken, ist, nur solche Symbole zu betrachten, deren Definitionen aus einer bestimmten Datei geladen wurden.

```
;; Zeige alle im Package user selbst definierten Symbole mit "FOREIGN",
;; die aus /madlener/rrl/dok-sys/dok4.lisp geladen wurden.
> (capropos "FOREIGN" :all-syms nil :mod-obj #P"/madlener/rrl/dok-sys/
  dok4.lisp")
*FUN-FOREIGN-CALLS, macro (FUN &OPTIONAL (LISPFLG NIL)
                                         (CTBLEORPCK (QUOTE *SYSTBL)))
*FUNS-FUNS-FOREIGN-CALLS, macro (FUNS FUNS2 &OPTIONAL
                                          (CALLTEBLE (QUOTE *SYSTBL)))
*FOREIGN-CALLS, macro (&OPTIONAL LSPOPT
                                         (CALLTABLE (QUOTE *SYSTBL)))
```

Vorsicht : Wird im Lispsystem selbst auf diese Art und Weise gesucht, ist zu beachten, daß das System nicht jede Art von Definition automatisch einer Datei zuordnet, so geschieht dies normalerweise nicht für Variablen- und Konstantendefinitionen.

Neben der Suche mit keys im Namen der Symbole kann auch in zugehörigen Dokumentationsstrings gesucht werden. Dazu wird der Parameter in-dok auf t gesetzt.

```
;; Suche nach Objekten mit "source" im Dokumentationsstring
> (capropos "source" :in-dok t)
GETD1, function (ARG &OPTIONAL (PRFLG T))
.
.
.
HSHOW-OBJ-WINDOW, function (AI HV)
SHOW-MACRO-WINDOW, function (FUN CALLTABLE)
GETD, macro (ARG &OPTIONAL (PRFLG T))
SHOW-FUNCTION-WINDOW, function (FUN CALLTABLE)
SHOW-EXPR-WINDOW, function (KEYWORD-STR FUN WINDOW-NAME)
```

Vorsicht : Die Suche in den Dokumentationsstrings ist erheblich aufwendiger als die Suche in den Symbolnamen.

Sind die Objekte, nach denen gesucht wird, mit den obigen Parametern bestimmt, so kann auch noch über Optionen die Ausgabe von Zusatzinformationen und eine Weiterverarbeitung der Objekte veranlaßt werden.

Der Parameter from, auf :file gesetzt, bietet beispielsweise die Möglichkeit, die Ladedatei mit anzuzeigen, wenn sie bekannnt ist.

```
;; Zeige alle im Package user selbst definierten Symbole mit den
;; Teilstrings "FUN" und "CALLS" mit ihrer Ladedatei, falls sie bekannt ist.
> (capropos '("FUN" "CALLS") :all-syms nil :from :file)
*REM-FUNCALLS, macro (&OPTIONAL (FUNS (QUOTE *FUNS)) ...)
loaded from #P"//node-1ab59/data/rrl/dok-sys/dok4.lisp"
FUN-FOREIGN-CALLS, function (FUN &OPTIONAL ...)
loaded from #P"//node-1ab59/data/rrl/dok-sys/dok3.lisp"
*FUN-FOREIGN-CALLS, macro (FUN &OPTIONAL (LISPFLG NIL) ...)
loaded from #P"//node-1ab59/data/rrl/dok-sys/dok4.lisp"
.
.
.
```

Es kann aber auch jeweils das aktuelle Package oder je nach dem ob die Ladedatei bekannt ist, diese oder das Package ausgegeben werden (s. hierzu die techn. Dokumentation 4.4.2)

```
;; mit aktuellem Package
> (capropos "APROPOS" :from :package)
*CAPROPOS, macro (IN-KEYS &KEY (T-P (QUOTE *SYSTBL)) FROM (TYPE
(QUOTE (QUOTE ALL))) (TR T) (IN T) NOT-KEYS OR-KEYS (ALL-SYMS T)
(STREAM T) LIST OTHER-SYMS COUNT (COUNT-STREAM T) MOD-OBJ TEST
IN-DOK) #<Package USER D90CCB>
.
.
.
APROPOS, function (NAME-STRING &OPTIONAL PACKAGE-NAME)
#<Package LISP D90813>
APROPOS-LIST, function (NAME-STRING &OPTIONAL PACKAGE-NAME)
#<Package LISP D90813>
CASE-INSENSITIVE-APROPOS, function (NAME-STRING &OPTIONAL
PACKAGE-NAME) #<Package SYSTEM D90843>

;; je nach dem mit Package oder Ladedatei
> (capropos "APROPOS-LIST" :from t)
*CAPROPOS-LIST, macro (IN-KEYS &KEY (T-P (QUOTE *SYSTBL)) ...)
loaded from #P"//node-1ab59/data/rrl/dok-sys/dok4.lisp"
APROPOS-LIST, function (NAME-STRING &OPTIONAL PACKAGE-NAME)
#>Package LISP D90813>
```

```
CASE-INSENSITIVE-APROPOS-LIST, function (NAME-STRING &OPTIONAL
PACKAGE-NAME) #<Package SYSTEM D90843>
```

Eine eventuelle Weiterverarbeitung wird über den Parameter 'count' gesteuert. Dieser Parameter erlaubt es, die Objekte -nach Typ getrennt- zu zählen oder zu sortieren. Dafür wird in count eine Liste übergeben, die ein oder mehrere der Symbole count, print und erg enthält. Ist count in der Liste, werden die Vorkommen der einzelnen Tableautypen gezählt, ansonsten erfolgt eine Zuordnung von Objektnamen zu Tableautypen. Ist print in der Liste, wird ein Ergebnis ausgedruckt. Ist schließlich erg in der Liste, so wird das Ergebnis der Zählung oder Zuordnung als Funktionswert zurückgegeben.

```
;; Anzahl der im Package user selbst definierten Makros
> (capropos nil :type 'macro :count '(count print) :stream nil :all-syms nil)
includes  94 macros
```

stream = nil diente hier der Unterdrückung der normalen Ausgabe von capropos. Üblicherweise legt dieser Parameter den Ausgabestream für die normalen Printausgaben von capropos fest. Die Ausgaben der count-Verarbeitung können über den Parameter count-stream umgelenkt werden.

```
;; alle Symbole mit erkennbarem Typ, die den Teilstring "CH" enthalten
;; und im Package user erreichbar sind, nach Typ sortiert ausgeben
> (capropos "CH" :count '(print) :stream nil)

(BUILD-IN-TYPE CHARACTER STANDARD-CHAR STRING-CHAR)
(SETF SCHAR CHAR CHAR-BIT)
(COMPILER-MACRO SET-SCHAR CHAR-INT CHAR<= SCHAR CHARACTERP
 MAKE-CHAR CHAR-FONT CHAR-CODE CHAR>= CHAR-BITS CHAR/=
 CHAR> CHAR= CHAR<)
(CONSTANT CHAR-HYPER-BIT CHAR-CONTROL-BIT CHAR-SUPER-BIT
 CHAR-META-BIT CHAR-FONT-LIMIT CHAR-CODE-LIMIT CHAR-BITS-LIMIT)
(MACRO CATCH CHECK-TYPE CHECK-NAME-AND-TYPE-OF-ENTRY)
(FUNCTION FUNCTION-CHECK-RUNTIME ... CHANGE-WP-WORKAREA)
```

Neben capropos gibt es noch einige abgeleitete Funktionen. So druckt capropos-list nichts (außer auf einen eventuellen count-stream) und liefert eine Liste der gefundenen Symbole als Funktionswert.

*capropos sucht statt im Package user in der Systemaufruftabelle und *capropos-list verhält sich zu *capropos wie capropos-list zu capropos.

Zur Suche nach Objekten mit bestimmten Eigenschaften und capropos s. Abschnitt 4.4.2 in der techn. Dokumentation.

5 Zusammenarbeit mit dem Apollo-Windowsystem

CALDO bietet auf Apollo-Workstations eine reichhaltige Schnittstelle zum Windowsystem. Wir werden auf diese Schnittstelle hier aber nicht näher eingehen, sondern uns nur auf die Möglichkeit des gezielten Bereitstellens von Quelltexten beschränken. Wer mehr über die Zusammenarbeit mit dem Windowsystem auf Apollo wissen will, findet Informationen in der technischen Dokumentation im Kapitel 6.

Oft ist es bei der Entwicklung oder der Korrektur von Programmen erwünscht schnell die Definition eines bestimmten Objekts editieren zu können.

Insbesondere wenn man mit dem Softwaresystem noch nicht sehr vertraut ist, ergibt sich die Schwierigkeit, die zugehörige Quelldatei zu finden. Aber auch wenn sie gefunden ist oder bekannt war, muß sie erst noch in einem Fenster sichtbar gemacht werden. In diesem muß man dann noch die Stelle suchen, an der sich die gewünschte Definition befindet.

Alle diese Aufgaben kann das Dokumentationssystem (in gewissem Rahmen) übernehmen. Ist bspw. eine Systemaufruftabelle definiert und soll der Quelltext einer Funktion 'fn1 gefunden werden, ist lediglich (show-source-code1 'fn1) einzugeben. Ist ein mit der Quelldatei assoziiertes Fenster vorhanden, wird es in den Vordergrund geholt und der Cursor wird an den Beginn der gesuchten Definition gesetzt. Ist noch kein Fenster mit der zugehörigen Quelldatei offen, wird ein solches erzeugt. Wenn bereits bekannt ist, daß ein Fenster der Quelldatei geöffnet ist, genügt auch der Ausdruck (show-source-code 'fn1).

Ist keine Systemaufruftabelle definiert oder kann das Objekt dort nicht gefunden werden oder wird der key-Parameter explizit auf nil gesetzt, so wird nach einem entsprechenden Objekt im Lispsystem gesucht. Falls die zugehörige Quelltextdatei ermittelt werden kann, wird dort nach der Definition gesucht.

Die Menüschnittstelle enthält die Einträge 'andere Infos/Quelltext im Fenster' und 'andere Infos/... in vorhandenen Fenster' zum Auffinden von Fenstern mit gewissem Inhalt.

Über den key-Parameter 'type' [Tableautyp] kann die Spezifikation des gesuchten Objekts eingeschränkt werden. Statt eines Symbols kann auch eine Symbolliste angegeben werden.

Speziell auf die Systemaufruftabelle zugeschnitten ist der Makro *show-system-sources, der mit einer Liste von Objektnamen aufgerufen wird und ebenfalls einen key-Parameter type kennt.

Detailiertere Informationen zum gezielten Anzeigen von Quelltexten finden sich in der techn. Dokumentation in den Abschnitten 6.2.2.6, 1.3.6, 6.2.2.2 und in 0.2 unter Dokumentationsoptionen/source.

Anhang: Tableautypen

Jedem, in einer Aufruftabelle gespeicherten Modulobjekt wird ein Typ zugeordnet. Dieser Typ wird als Tableau-Typ bezeichnet. Momentan sind dem System im wesentlichen die folgenden einfachen Tableautypen bekannt:

advice	- advice mit defadvice
compiler-macro	- Makro mit def-compiler-macro
constant	- Konstante mit defconstant
flavor-construct	- Flavor-Konstrukte
foreign-function	- externe Funktion mit define-foreign-function
foreign-type	- externer Datentyp mit def-foreign-type
function	- normale Lispfunktion mit defun
global	- globale variable mit defvar
inline-function	- Inline-Lispfunktion mit defsubst
macro	- Makro mit defmacro
modify-macro	- Makro mit define-modify-macro
parameter	- Parameter mit defparameter
setf	- setf-Methode mit defsetf oder define-setf-method
struct-type	- Struct-Typ mit defstruct
top-level-expression	- Ausdruck auf top-level
user-type	- Lisptyp mit deftype

Außer diesen einfachen Tableautypen gibt es noch sogenannte zusammengesetzte Tableautypen, die eine Zusammenfassung ein oder mehrerer Tableautypen darstellen. Eine Liste dieser Typen findet man in der technischen Dokumentation unter 0.2.

An dieser Stelle möchten wir Roland Fettig, Manuela Gaß, Rita Kohl und Michael Zehnter danken, die durch ihre Unterstützung und konstruktive Kritik wesentlich zum Gelingen dieses Tutorials beigetragen haben.

Index