Letzte/Diese Vorlesung

Mehr über Makros	Letzte Woche
Mehr über Debugging	
Streams	Diese Woche

Zugriff (auch) auf Dateien: Streams

Mit **Streams** wird der Zugriff (also Input/Output-Operationen) auf Dateien realisiert. Streams sind ein neuer LISP-Datentyp. Es gibt also Konstruktor- und Zugriffsfunktionen.

Vordefinierte Streams:

`*standard-input*` Terminaleingabe

`*standard-output*` Terminalausgabe

`*error-output*` Ausgabestream für Fehlermeldungen (meist identisch mit Terminalausgabe)

streams als Datentyp

Operationen auf streams:

Öffnen und Schließen: `open, close, with-open-file`
Tests: `streamp, open-stream-p, input-stream-p, output-stream-p`
Schreiben auf Ausgabestreams: `format`
Lesen von Eingabestreams: `read, read-char, read-line, listen`

with-open-file: Datei und streams verbinden

Eingabedatei öffnen:
`(with-open-file (`<*stream-variable*> <*pathname*> `:direction :input)`
`<` *body* `>)`

Ausgabedatei öffnen:
`(with-open-file (`<*stream-variable*> <*pathname*> `:direction :output`
`:if-exists { :supersede | :overwrite | :append | :error })`
`<` *body* `>)`

Im <*body*>, der als implizites `progn` ausgewertet wird, ist die Variable <*stream-variable*> an einen stream gebunden, den man an Ein- bzw. Ausgabefunktionen übergeben kann.

format: Formatierte Ausgabe auf streams

Allgemein:
`(format` <*stream*> <*format-string*> `&rest` <`format-args`>`)`

Der <*format-string*> kann Formatanweisungen enthalten.
Formatanweiseungen beginnen mit dem Zeichen "~".

```
(format t "foo")                              -->   foo
(format t "Der Wert ist ~d~%Nicht Null" 4)    -->   Der Wert ist 4
                                                    Nicht Null
(format t "Der Wert ist ~3,'0d~%" 7)          -->   Der Wert ist 007
(format t "Der Wert ist ~,4f~%" 4.123456)     -->   Der Wert ist 4.1234
(format t "Der Wert ist ~s~%" "ein String")   -->   Der Wert ist "ein String"
(format t "Der Wert ist ~a~%" "ein String")   -->   Der Wert ist ein String
```

Einige Formatierungsanweisungen für format

Beachte: Es müssen mindestens soviele Argumente wie Formatierungsanweisungen im Formatstring sein. Zusätzliche Argumente werden ignoriert

`~A`	S-Expression als ASCII-Darstellung
`~S`	S-Expression so, daß sie von `read` akzeptiert wird
`~D`	Dezimalzahl. Das Argument muß vom Type `number` sein
`~F`	Floatingpoint-Zahl. Das Argument muß vom Type `number` sein
`~C`	Character (ein Zeichen). Das Argument muß vom Type `character` sein
`~%`	Zeilenumbruch
`~&`	Zeilenumbruch falls nicht schon am Zeilenanfang
`~~`	Drucke ein `~`

Ausgabe auf streams mit format

- `format` arbeitet unabhängig vom verwendeten Stream.
- Stream kann sein:
  - `nil` Ausgabe in String.
  - `t` Ausgabe auf `*standard-output*`.
  - `stream` Ausgabe auf stream: Datei, Socket, `*standard-output*`, usw.

Einlesen von Daten:

Die Kernfunktion für das Einlesen von Daten ist `read`.

```
? (setq eingabe (read))
42
--> 42
? (setq eingabe (read))
42 - 17
-->42
-->-
-->17
? eingabe
--> 42
```

Einlesen von Streams:

`read` nimmt als optionales Argument einen Stream.

```
? (with-open-file (file-stream "input-datei.txt"
                   :direction :input)
    (setq ergebnis (read file-stream)))
--> 42
```

D.h. `(read)` ist das gleiche wie `(read *standard-input*)`.

Einlesen von Streams:

...und was passiert, wenn die Datei leer ist?

- Ein Stream kann zwar nicht „leer“ sein, aber „am Ende“ sein.
- Man sagt trotzdem: „Das Ende der *Datei* ist erreicht.“

```
USER(28): (with-open-file (stream "leere-datei" :direction :input)
              (read stream))
Error: eof encountered on stream
       #<EXCL::CHARACTER-INPUT-FILE-STREAM #"leere-datei" pos 0 @ #x22e971a>
  [condition type: END-OF-FILE]
[1] USER(35):
```

Dafür hat `read` weitere Parameter:

Am Ende des Streams:

`read` nimmt als weitere optionale Parameter `eof-errorp` und `eof-value`.

- `eof-errorp` entscheidet, ob ein `error` signalisiert wird.
- `eof-value` gibt an, was der Rückgabewert von `read` sein soll, wenn das Ende des Streams erreicht wird.
- Die typische Verwendung ist:
- `(read stream nil 'eof)`
- Damit wird eine Lisp-Fehlermeldung vermieden, und es kann getestet werden, ob der Stream „am Ende“ ist.

`read` nimmt als weitere optionale Parameter `eof-errorp` und `eof-value`.

- Eine typische Verwendung ist:

```
(let ((eingabe nil))
    ...
   (setq eingabe (read stream nil 'am-ende))
   (unless (eq eingabe 'am-ende)
    ...))
```

- Oder oft verkürzt als:

```
(unless (eq (setq eingabe (read stream nil 'am-ende)) 'am-ende)
```

Was genau wird denn von `read` eingelesen?

- `read` liest ein Lisp-Objekt ein, also:
- einen String oder
- eine Zahl oder
- einen Character oder
- eine S-Expression

D.h., mit einem Befehl `read` kann man über viele Zeilen hinweg einen „korrekt geklammerten“ Ausdruck einlesen!

Einlesen von S-Expressions:

Mit `read` können ganze Programme leicht eingelesen werden!

```
(with-open-file (in-stream "programm.lisp")
  (loop for input-expression = (read in-stream nil :dateiende)
        until (eq input-expression :dateiende)
        do (eval input-expression)))
```

Dafür gibt es auch eine Lisp-Funktion: `load`:

```
(load "programm.lisp")
```

Weitere Einlesefunktionen:

- `read-from-string` liest statt von einem Stream von einem String:
  `(read-from-string "   42   ") --> 42`
- Kann auch mit `with-input-from-string` gemacht werden.
- `read-line` liest nur eine einzige Zeile und liefert diese als String.
- Was ist der Unterschied zwischen `(read)` und `(read-from-string (read-line))`?
- `read-char` liest einen einzelnen Character ein

Kopieren von Lisp-Objekten:

Was macht:

```
(setq foobar (list 'a 'b 'c))
(setq barfoo (read-from-string (format nil "~S" foobar)))

(eq barfoo foobar)
--> NIL
(equal barfoo foobar)
--> T
```