Funktionale Programme

Bei einem rein funktionalen Programm gibt es innerhalb der Programmausführung keine vom Programmierer zu bedenkenden Zustände. Es gibt nur den Zustand PRE vor der Ausführung des gesamten Programms und den Zustand POST nach der Ausführung des gesamten Programms. Die Ausführung des Programms wird nicht als Folge von programmierten Schritten gedacht. Zwar werden selbstverständlich innerhalb des Rechners bei der Ausführung des Programms auch im Falle der funktionalen Programme einzelne Schritte ablaufen, aber diese Schritte werden nicht vom Programmierer ins Programm formuliert, sondern sie ergeben sich aus der Konstruktion des programmausführenden Akteurs.

Wir betrachten nun den Ausdruck $3+4$. Dies ist ein Beispiel für ein einfaches funktionales Programm. Man sollte diesen Ausdruck nicht als Anweisung deuten, denn es wird ja explizit nichts verlangt. Man ist zwar versucht, solche Ausdrücke doch als Anweisung zu deuten, denn man denkt sich immer die Anweisung „Berechnen Sie das Ergebnis des Ausdrucks $3 + 4$“ hinzu. Es ist aber viel angemessener, solche Ausdrücke als Identifikationen von Werten zu betrachten. In dieser Sichtweise identifiziert der Ausdruck $3+4$ den gleichen Wert wie die Ausdrücke $8-1$ und $\sqrt{49}$.

Grundsätzlich ist ein reines funktionales Programm nichts anderes als eine mehr oder weniger umfangreiche Umschreibung eines Wertes. Hierbei ist „Wert“ nicht notwendigerweise eine Zahl, sondern irgendein informationelles Individuum, für das eine sogenannte kanonische Form der Identifikation festgelegt wurde. Im betrachteten Beispiel ist das identifizierte informationelle Individuum die natürliche Zahl *sieben*, für die die Ziffer 7 als kanonische Form vereinbart ist. Die Zahl sieben selbst sollte man nicht mit der Ziffer 7 verwechseln. Die Ziffer 7 ist sichtbar, wogegen die natürliche Zahl *sieben* nur denkbar ist.

Wenn man ein funktionales Programm schreibt, umschreibt man also ein informationelles Individuum und erwartet, dass als Ergebnis der Programmausführung die kanonische Form zur Identifikation des Individuums ausgegeben wird.

Bei unserer Programmierübung mit der Sprache LISP arbeiten wir mit einem Interpreter und nicht mit einem Compiler. Im Falle von FORTRAN und C haben wir die Programme jeweils zuerst übersetzen müssen, bevor wir sie zur Ausführung bringen konnten. Im Falle des Einsatzes des Compilers haben wir also einen Aufbau vor Augen, wie er in Bild 21 gezeigt ist: Der Mensch editiert den programmiersprachlichen Quelltext. Diesen gibt er dem Compiler, der daraus einen ausführbaren Code erzeugt, den wir uns nicht anschauen, sondern den wir dem Abwickler zur Ausführung übergeben. Das Rechteck, welches den ausführbaren Code und den Programmabwickler einschließt, stellt den Akteur dar, den wir eigentlich realisieren wollten. Im Falle unseres Wurzelprogramms handelt es sich also um einen Wurzelzieher, der mit dem Anwender kommunizieren kann.

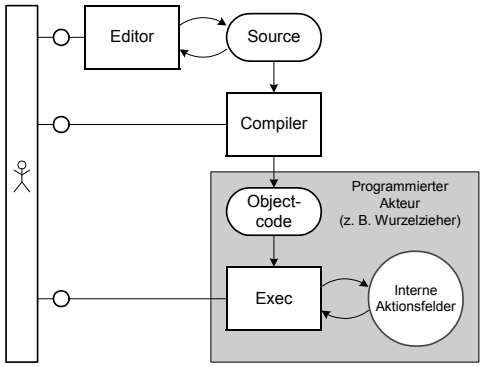

Bild 21 Der Compiler in der Sicht des Benutzers

Bild 22 zeigt einen Aufbau, wie wir ihn vor Augen haben sollten, wenn wir unter Verwendung eines Interpreters programmieren. Wir können direkt über die Tastatur dem Interpreter den auszuwertenden Ausdruck mitteilen, zu dem er dann die kanonische Form ausgibt. Man kann Ausdrücke eingeben, die ohne Zugriff auf andere Informationen vollständig auswertbar sind. Es kann aber auch sein, dass der Interpreter bei der Auswertung des eingegebenen Ausdrucks auf eine im externen Dateisystem gespeicherte Source und auf seinen internen Speicher zugreifen muss.

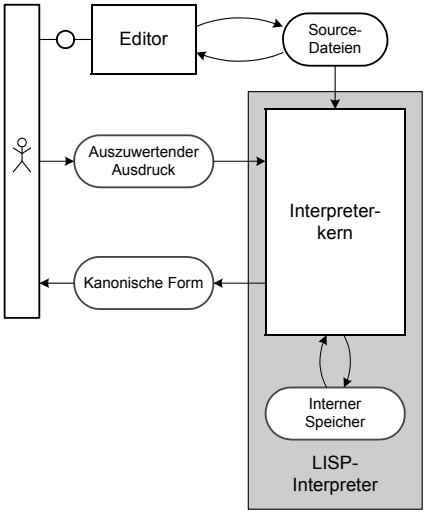

Bild 22 Der Interpreter in der Sicht des Benutzers

Wenn wir das zu identifizierende Individuum als Ergebnis einer Funktionsanwendung auf Argumente umschreiben, müssen wir in LISP die Präfixnotation verwenden. Wir können also nicht schreiben 3 + 4, sondern müssen schreiben (+ 3 4). Die Infixnotation ist zwar sehr angenehm zu lesen, sie ist aber auf Funktionen mit zwei Argumenten beschränkt. Dagegen erlaubt die Präfixnotation auch die Formulierung von Funktionen mit nur einem oder mit mehr als zwei Argumenten. Wenn also dem LISP-Interpreter ein Ausdruck übergeben wird, der mit einer öffnenden Klammer beginnt, wird der Interpreter versuchen, das Symbol nach der öffnenden Klammer als Funktionssymbol zu interpretieren. Eine Eingabe, die mit einer öffnen-

den Klammer beginnt, wird also vom Interpreter nie als kanonische Form angesehen werden, die er unverändert wieder ausgeben müsste. Beispiele für die Eingabe kanonischer Formen sind:

Eingabe: 7 Ausgabe: 7
Eingabe: „Hundefutter“ Ausgabe: „Hundefutter“

Eine Sprache, in der man nur identifizierende Ausdrücke, aber keine Anweisungen formulieren kann, wäre zu eingeschränkt, als dass man damit interessante Programme formulieren könnte. Man ist also gar nicht an rein funktionalen Sprachen interessiert. Nun ist aber in LISP festgelegt, dass jedes Statement ein identifizierender Ausdruck sein muss. Also muss man in die identifizierenden Ausdrücke irgendwie auf mehr oder weniger natürliche oder unnatürliche Weise einen imperativen Anteil hineinbringen.

Es gilt: Ein LISP-Ausdruck ist entweder nur identifizierend, oder er ist sowohl imperativ als auch identifizierend. Der Fall, dass ein LISP-Ausdruck nur imperativ ist, ist ausgeschlossen. Da die identifizierende Eigenschaft eines Ausdrucks immer vorhanden ist und die imperative Eigenschaft nur manchmal, spricht man bei dem imperativen Anteil von sogenannten „Seiteneffekten“ der Ausdrücke. Ein Seiteneffekt bedeutet also eine Zustandsveränderung im Inneren der Maschine oder auf den Ausgabegeräten. Diese Zustandsveränderungen werden erzeugt, bevor die kanonische Form des durch den Ausdruck identifizierten Wertes ausgegeben wird.

Wir betrachten das Programmstatement (PRINT „Hundefutter“). Wenn wir diesen Ausdruck dem Interpreter übergeben, wird er zweimal hintereinander die Zeichenfolge „Hundefutter“ ausgeben. Die erste Ausgabe ergibt sich als Seiteneffekt, weil PRINT eine Anweisung ist. Die zweite Ausgabe ist die Ausgabe der kanonischen Form des durch den Ausdruck identifizierten Wertes. Es wurde festgelegt, dass ein PRINT-Ausdruck immer das identifiziert, was er auch ausgibt.

Man sollte immer sorgfältig unterscheiden zwischen Funktionssymbolen und Anweisungen. In dem Ausdruck (+ 3 4) ist das Symbol + funktionssymbolisierend und stellt keine Anweisung dar. Deshalb hat dieser Ausdruck auch keine Seiteneffekte. Im Unterschied hierzu ist in der Anweisung (PRINT “Hundefutter“) das Wort PRINT eine Anweisung, die zu einem Seiteneffekt führt.

Die Zuweisung v:=47.5 äußert sich in LISP als der Ausdruck (SETQ v 47.5). In diesem Fall ist SETQ eine Anweisung. Der Seiteneffekt besteht darin, dass einem Speicherort v, der entweder vorher schon geschaffen wurde oder jetzt erst geschaffen wird, der Wert 47.5 zugewiesen wird. Da dieser Ausdruck aber nicht nur imperativ sein kann, sondern auch identifizierend sein muss, wird der Interpreter, dem man diesen Ausdruck gibt, die kanonische Form 47.5 ausgeben. Wenn wir anschließend dem Interpreter einfach den Ortsbezeichner v eingeben, wird ebenfalls die kanonische Form 47.5 ausgegeben. Die Eingabe eines Ortsbezeichners stellt eine Identifikation dar, und der identifizierte Wert ist der aktuelle Speicherinhalt.

Damit wir den Dialog mit dem Interpreter beenden können, muss es eine bestimmte Zeichenfolge geben, die der Interpreter als Wunsch, den Dialog abzubrechen, interpretiert. Diese Zeichenfolge lautet (QUIT) .

Prozedurale Anteile in LISP

Nachdem nun der imperative Anteil von LISP eingeführt ist, liegt die Frage nahe, wie man denn in LISP eine gewünschte Aufeinanderfolge von Anweisungen formulieren kann. Im Falle der imperativen Sprachen habe ich drei Strukturtypen für die sog. strukturierte Programmierung vorgestellt: Sequenz, Fallunterscheidung und Wiederholung. Diese drei elementaren Strukturtypen finden wir auch in LISP:

PROGN bzw. PROG1	für die Formulierung von Sequenzen
COND	für die Formulierung von Alternativen
DO	für die Formulierung von Wiederholungen

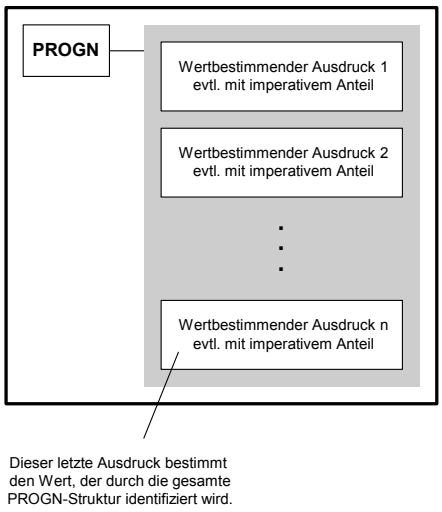

Bild 23 Sequentialisierung in LISP

Mit PROGN wird der sequentielle Ablauf, wie er aus der imperativen Programmierung bekannt ist, in die funktionale Sprache LISP eingebracht. Die Ausdrücke 1 bis n (s. Bild 23) werden sequentiell nacheinander interpretiert. Die identifizierende Eigenschaft der Ausdrücke 1 bis (n-1) wird dabei nicht wirksam. Diese Ausdrücke wirken in diesem Fall also nur durch ihren imperativen Anteil.

Die Wirkung von PROG1 unterscheidet sich von PROGN nur dadurch, dass als wertbestimmender Ausdruck für die gesamte PROG-Struktur nicht der letzte, sondern der erste Ausdruck in der Sequenz genommen wird.

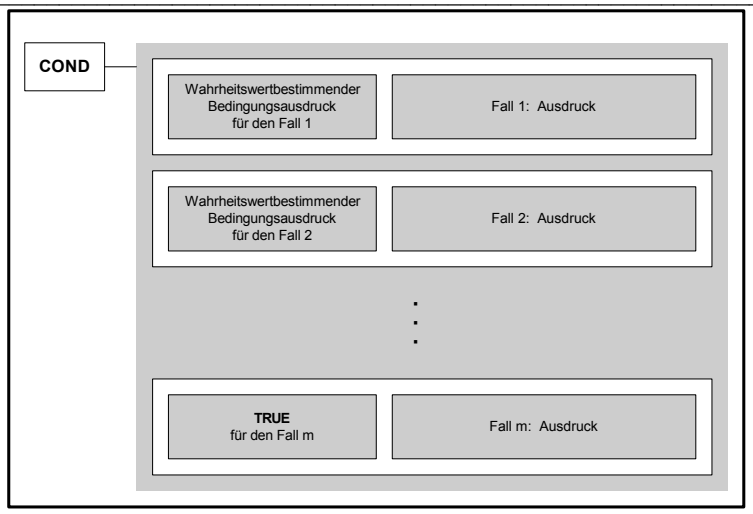

Bild 24 Fallunterscheidung in LISP

Bei der Interpretation einer COND-Struktur wird der zu interpretierende Fall-Ausdruck dadurch gefunden, dass vom Bedingungsausdruck 1 ausgehend sequentiell nacheinander die Bedingungsausdrücke interpretiert werden, bis ein Bedingungsausdruck gefunden ist, der den Wahrheitswert TRUE identifiziert. Dann wird der zugehörige Fall-Ausdruck interpretiert; dieser liefert den durch die gesamte COND-Struktur identifizierten Wert.

Damit es mindestens eine Bedingung gibt, die den Wert TRUE identifiziert, legt man als Wert der letzten Bedingung explizit den Wert TRUE fest.

Am Beispiel des COND-Ausdrucks soll auf ein Problem hingewiesen werden, welches häufig nicht bedacht wird. An Stellen im Programm, wo man eigentlich nur einen identifizierenden Ausdruck vermutet, könnte durchaus auch ein Ausdruck stehen, der nicht nur identifiziert, sondern der auch einen imperativen Anteil hat, welcher zu einer Zustandsveränderung führt. Die Bedingungsausdrücke in der COND-Struktur sollten eigentlich nur identifizierende Wirkung haben. Es soll jeweils ein Wahrheitswert aus dem Repertoire {FALSE, TRUE} identifiziert werden. Die Ausdrücke werden von oben nach unten ausgewertet, und in der Zeile, wo das erste Mal der Wert TRUE identifiziert wird, wird der daneben stehende Ausdruck ausgewertet, von dem man normalerweise eine zustandsverändernde Wirkung erwartet. Wenn man nun den Bedingungsausdrücken eine imperative Wirkung gibt, könnte man erreichen, dass auch alle Bedingungsausdrücke, die den Wert FALSE lieferten, eine bleibende Zustandsänderung bewirkt haben. Dies wäre aber kein guter Programmierstil, sondern müsste als üble Trickserei verurteilt werden.

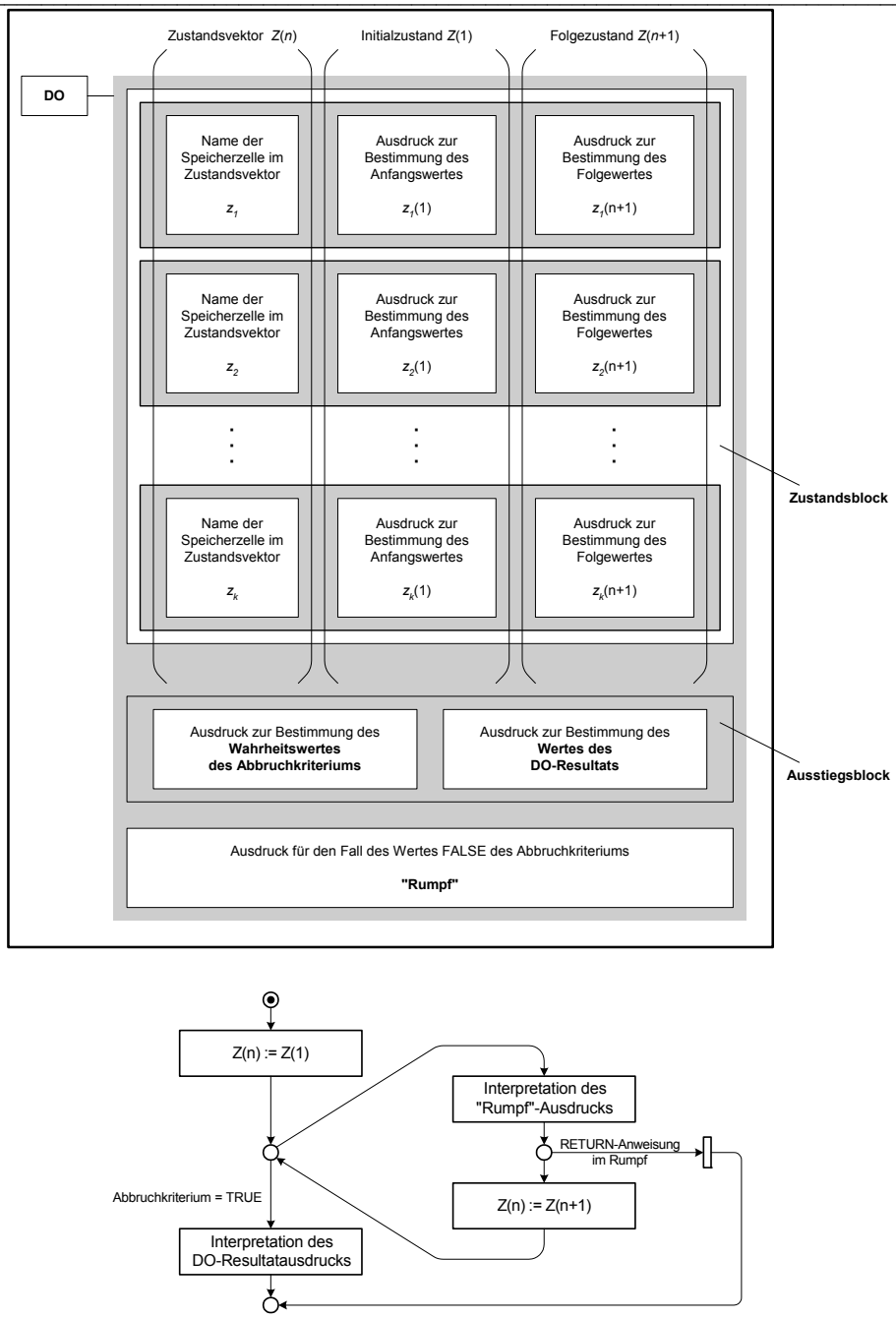

Bild 25 Wiederholung in LISP

Die Interpretation der DO-Struktur ist verhältnismäßig einfach, wenn man sie als Formulierung eines Ablaufs ansieht, den man in Form eines Petrinetzes darstellen kann. Das zur DO-Struktur gehörende Petrinetz ist in Bild 25 dargestellt.