Binärbäume und Listen

Obwohl ich Ihnen die Programmiersprache LISP mit einem numerischen Beispiel vorgestellt habe, darf man daraus nicht schließen, dass LISP für ähnliche Aufgaben entwickelt wurde wie beispielsweise FORTRAN. LISP wurde ganz speziell als Sprache für die Symbolverarbeitung entwickelt. Ein typisches Beispiel für Symbolverarbeitung ist die symbolische Differentiation. Wir betrachten ein Beispiel.

Die Zeichenfolge „$5x + 3$ SIN $(2x^3 - e^{4x})$“ sei als Argument der Differentiationsfunktion vorgegeben.
Die Zeichenfolge „$5 + 3 * (6x^2 - 4 e^{4x}) * $ COS $(2x^3 - e^{4x})$“ ist das Ergebnis.
Das Differenzieren geschieht nach ganz bestimmten festen Regeln: Summenregel, Produktregel, Quotientenregel, Kettenregel und ähnliche. Außerdem muss man die Ableitungen der transzendenten Funktionen SIN, COS, EXP usw. auswendig wissen.

Grundsätzlich besteht Symbolverarbeitung immer in der Berechnung von Funktionen, bei denen das Argument eine Zeichenfolge ist, zu dem als Ergebnis wieder eine Zeichenfolge bestimmt werden soll. Mit imperativen Sprachen sind solche Funktionen sehr schwierig zu realisieren. Hier eignet sich LISP sehr gut, weshalb LISP auch als eine Sprache der künstlichen Intelligenz (KI) charakterisiert wurde.

Es charakterisiert die Sprache LISP, dass nicht die alphanumerische Programmform das Primäre ist, sondern die Programmform in Binärbaumdarstellung. Damit wir auch Binärbäume mit großer Tiefe in einem übersichtlichen Layout darstellen können, führen wir eine orthogonale Darstellung ein: Ein linker Unterknoten hängt senkrecht unter seinem Oberknoten, und ein rechter Unterknoten hängt waagerecht rechts neben seinem Oberknoten.

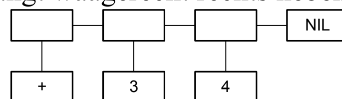


Bild 26 Beispiel für unser spezielles Layout zur Darstellung von LISP-Binärbäumen

Zu dem alphanumerischen Ausdruck (+ 3 4) gehört der Binärbaum in Bild 26. Es handelt sich hier um eine sog. Liste im Sinne von LISP. In einer Liste sind die waagerecht nebeneinander liegenden unbeschrifteten Knoten die Aufhänger für die Positionsbelegungen. Der mit NIL beschriftete Knoten dient dazu, die Kette abzuschließen. In dem betrachteten Beispiel in Bild 26 handelt es sich also um eine Liste mit 3 Positionen. Selbstverständlich dürfen die einzelnen Listenpositionen selbst wieder mit Listen belegt werden. In Bild 27 ist dies an einem Beispiel gezeigt.

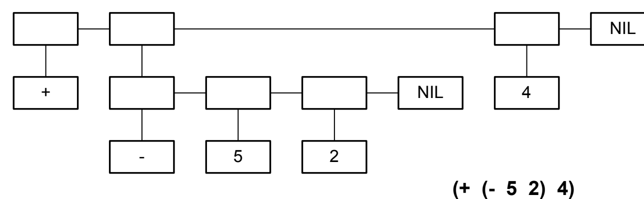


Bild 27 Beispiel einer Liste, die u.a. eine Liste enthält

Der Binärbaum in Bild 28 ist zwar auch alphanumerisch in LISP ausdrückbar, aber es handelt sich hier nicht um eine Liste. Die zugehörige alphanumerische Form benutzt den Punkt als Symbol zur jeweiligen Trennung zweier an einem Oberknoten hängenden Teile.

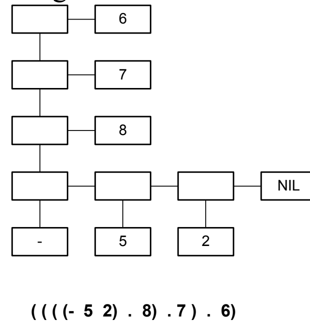


Bild 28 Beispiel eines Binärbaums, der keine Liste ist

Im Folgenden interessieren wir uns nur noch für sogenannte auswertbare Ausdrücke. In Bild 29 ist ein Klassifikationsschema für Binärbäume gezeigt. Von den allgemeinen Binärbäumen interessieren uns nur die, bei denen nur die Blätter beschriftet sind. Von diesen interessieren uns nur diejenigen, die als verschachtelte Listen gesehen werden können. Und von diesen interessieren uns nur die auswertbaren Ausdrücke. Bei den auswertbaren Ausdrücken kann man noch einmal unterscheiden zwischen solchen, für die eine Standardauswertung gilt, und andere, für die individuelle Sonderformen der Auswertung gelten. Als Beispiele für Ausdrücke, die bezüglich ihrer Interpretation als Sonderfälle zu behandeln sind, haben wir bisher kennengelernt PROGN, COND und DO. Nun soll aber die Standardform der Auswertung betrachtet werden.

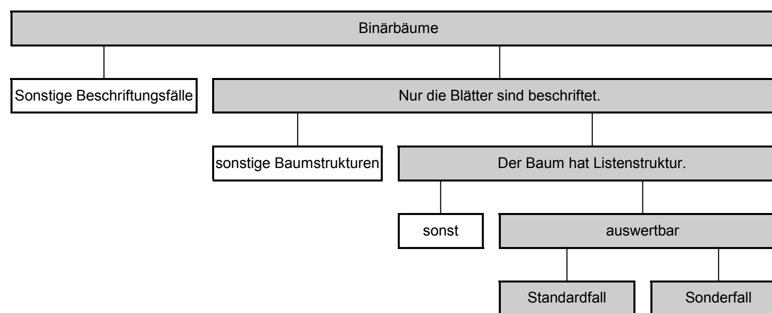


Bild 29 Klassifikation von Binärbäumen

Bei einer auswertbaren Liste steht auf der ersten Listenposition ein Ausdruck zur Identifikation einer Funktion, und auf den restlichen Positionen der Liste müssen auswertbare Ausdrücke stehen, deren Auswertung die Argumente der identifizierten Funktion sein sollen. Das in Bild 27 gezeigte Beispiel ist ein auswertbarer Ausdruck. Es handelt sich um eine Liste mit 3 Positionen, wobei auf der ersten Position das Symbol + steht, welches die Additionsfunktion identifiziert. Auf den beiden nachfolgenden Positionen stehen auswertbare Ausdrücke, wovon der erste selbst wieder eine Liste ist und der zweite eine Konstante. An der ersten Position der Argumentliste steht das Zeichen -, welches die Subtraktionsfunktion identifiziert.

Zur Kennzeichnung von Bäumen, die auswertbar sind, kennzeichne ich von nun an deren Wurzeln mit einem Doppelrahmen. Dem doppelumrandeten Knoten kann dann also jeweils gedanklich das Auswerteergebnis zugeordnet werden.

Definition von Funktionen

Der LISP-Interpreter kennt selbstverständlich nur ein kleines endliches Repertiore sogenannter Basisfunktionen, die durch Angabe eines Symbols oder eines Namens identifiziert werden können. Es ist aber auch erforderlich, Funktionen einzuführen, die der Interpreter noch nicht kennt. Hierfür gibt es in LISP zwei Möglichkeiten. Es gibt den Fall, dass man eine Funktion zum sofortigen Verbrauch einführt, und es gibt den anderen Fall, dass man eine Funktion zur beliebig langen Lagerung und mehrfachen Verwendung einführt. Im zweiten Fall muss die Funktion einen Namen bekommen, denn sonst könnte man sie nicht mehrfach verwenden. Im ersten Fall dagegen braucht man die Funktion nicht zu benennen, denn sie wird ja sofort auf ein Argumenttupel angewandt und ist dann wieder vergessen.

Als Beispiel einer Funktion, die der LISP-Interpreter nicht kennt und die wir ihm bekannt machen wollen, betrachten wir die Funktion

$$f(x) = 2x + 1$$

Zu einer Anwendung dieser Funktion f auf das Argument 5 muss der Baum eine Struktur haben, wie sie in Bild 30 gezeigt ist.

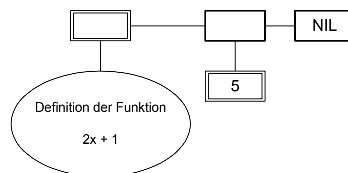


Bild 30 Prinzipielle Baumstruktur für Funktionsdefinition und Funktionsanwendung

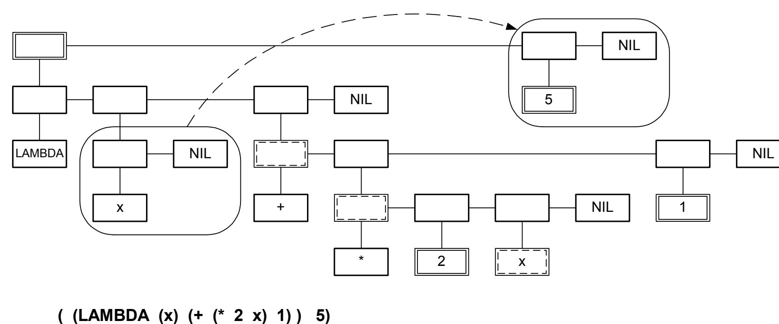


Bild 31 Definition einer Funktion „für den sofortigen Verbrauch“

Wir betrachten nun den Fall, dass wir die Funktion f zum sofortigen Verbrauch definieren, d.h. dass außerhalb des Baumes in Bild 30 die Funktionsdefinition keine Rolle spielen soll. Dann sieht der Baum so aus, wie er in Bild 31 gezeigt ist. Die Funktionsdefinition ist eine Liste mit 3 Positionen. An der ersten Position steht das Wort `LAMBDA`. Der griechische Buchstabe λ bzw. Λ wird häufig verwendet, wenn man darauf hinweisen will, dass etwas leer oder irrelevant sei. `LAMBDA` ist kein Funktionsname, sondern nur der Hinweis auf den Sachverhalt, dass hier eine Funktion definiert wird, die zum sofortigen Verbrauch bestimmt ist.

An der zweiten Position der funktionsdefinierenden Liste steht die Liste der Argumentvariablen dieser Funktion. Im konkreten Fall enthält diese Liste nur ein einziges Element, nämlich die Argumentvariable x. Auf der dritten Position steht die konkrete Funktionsumschreibung. Es handelt sich hierbei um die Form einer auswertbaren Liste, deren Auswertbarkeit aber davon abhängt, dass der Variablen x ein Wert zugewiesen wurde. Diese Zuweisung ist aber im gegebenen Fall vorhanden, denn die Variablenliste in der Funktionsdefinition ist strukturgleich im Baum rechts oben noch einmal vorhanden, nur dass dort nicht mehr der Variablenname x, sondern der aktuell gewünschte Wert 5 steht.

In Bild 32 ist der Fall gezeigt, dass die zu definierende Funktion beliebig oft verwendbar bleiben soll, d.h. dass sie nicht nach der ersten Verwendung schon wieder vergessen wird. Das Wort `DEFUN` weist darauf hin, dass es hier um die Definition einer Funktion geht. Man sieht, dass der `DEFUN`-Ausdruck große Ähnlichkeit mit dem `LAMBDA`-Ausdruck aus Bild 31 hat. Das konkrete Argument, welches immer unmittelbar auf eine `LAMBDA`-Definition folgen muss, fehlt aber hier bei der `DEFUN`-Definition. Die Funktionsanwendungen kommen ja erst später.

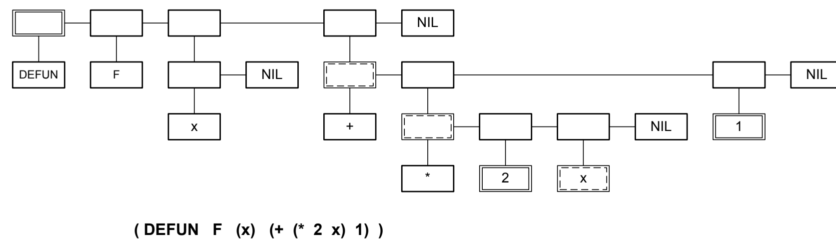


Bild 32 Definition einer Funktion für die spätere mehrfache Benutzung

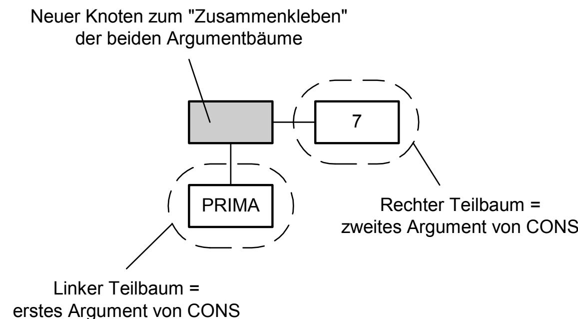
Für `DEFUN`-Ausdrücke gilt nicht die Standardform der Auswerrtung, sie sind Sonderfälle für die Auswertung im Sinne von Bild 29. Aber auch für diese gilt wie für alle auswertbaren Formen, dass sie möglicherweise einen imperativen Anteil haben und in jedem Falle einen identifizierenden. Der imperative Anteil von `DEFUN` besteht in der internen Abspeicherung der Funktionsdefinition, so dass diese später noch beliebig oft benutzt werden kann. Der identifizierende Anteil von `DEFUN` besteht darin, dass der Funktionsname identifiziert wird.

Zwischen `DEFUN` und `SETQ` besteht eine große Verwandtschaft: `DEFUN` dient dem Abspeichern von Funktionsdefinitionen zum Zwecke ihrer späteren Nutzung, wogegen `SETQ` der Abspeicherung von Werten in benannten Speicherzellen dient, damit diese Werte anschließend beliebig oft benutzt werden können.

Verknüpfung von Binärbäumen

Ich habe schon früher darauf hingewiesen, dass die Programmiersprache LISP entwickelt wurde zur Verarbeitung von Symbolstrukturen. Die Möglichkeit der effizienten Symbolverarbeitung beruht auf 3 Basisfunktionen, bei denen sowohl jedes Argument als auch das jeweilige Ergebnis jeweils ein Binärbaum ist. Die Funktion CONS (Abkürzung von Construction) verknüpft 2 Bäume, die als Argumente der Funktion vorgegeben werden, zu einem neuen Baum. In dem Ergebnisbaum kommen die beiden Argumentbäume im Original wieder vor und zusätzlich noch ein weiterer Knoten, der die Wurzel des Ergebnisbaumes ist. Als linker Teilbaum, der an dieser Wurzel hängt, wird das erste Argument von CONS genommen, und

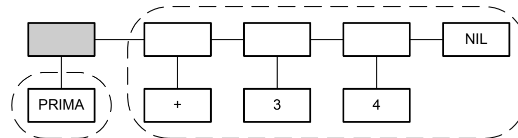
als rechter Teilbaum das zweite Argument. Bild 33 veranschaulicht dies. Hier ist das zweite Argument der CONS-Funktion ein auswertbarer Ausdruck, der auch tatsächlich ausgewertet wird. Was also als rechter Teilbaum im Ergebnis auftaucht, ist nicht die Liste (+ 3 4) , sondern das Ergebnis 7.



(CONS "PRIMA" (+ 3 4))

Bild 33 Beispiel zur CONS-Funktion

Wenn wir nun aber als CONS-Ergebnis den Baum haben wollen, der in Bild 34 gezeigt ist, dann dürfen wir das zweite Argument von CONS nicht als Liste (+ 3 4) umschreiben. Mit den bisher bekannten sprachlichen Mitteln können wir diese Liste durch dreifache Anwendung der CONS-Funktion umschreiben:



(CONS "PRIMA" (CONS "+" (CONS 3 (CONS 4 NIL))))

Bild 34 Beispiel zur CONS-Funktion

Nun ist es aber auch wünschenswert, den Baum in Bild 34 durch einen Ausdruck zu umschreiben, worin die CONS-Funktion nur einmal vorkommt, bei der als erstes Argument das Wort „PRIMA“ und als zweites Argument die Liste (+ 3 4) explizit vorkommt. Dies erfordert eine Möglichkeit, auszudrücken, dass die auswertbare Liste
(+ 3 4) nicht ausgewertet werden soll. Hierzu wurde die Quotierungsmöglichkeit in LISP eingeführt, für die es die folgenden beiden alternativen Schreibweisen gibt:

(QUOTE (+ 3 4))   ’(+ 3 4)

Neben der Möglichkeit, Bäume zu einem größeren Baum zusammen zu kleben, gibt es auch die Möglichkeit; von einem gegebenen Baum etwas abzuschneiden und dieses Abgeschnittene zum Ergebnis zu erklären. So kann man aus einem zusammengeklebten Baum wieder die Teile zurückgewinnen. Es gilt

(CAR (CONS „LINKS“ „RECHTS“)) = “LINKS”

(CDR (CONS „LINKS“ „RECHTS“)) = “RECHTS”

Die Wahl der seltsamen Funktionsbezeichner CAR und CDR ist historisch bedingt. Der erste LISP-Interpreter wurde in Assembler-Sprache auf einer IBM-Maschine realisiert. Dabei wurde die Binärbaumstruktur dadurch realisiert, dass man die Wurzel des linken Teilbaumes unter Verwendung des Adressregisters und die Wurzel des rechten Teilbaumes unter Verwendung des Datenregisters realisierte. CAR bedeutet *Content of Address Register*, und CDR bedeutet *Content of Data Register*. Es ist eigentlich unverständlich, dass diese Bezeichnungen aus der Anfangszeit von LISP beibehalten wurden und nicht durch Bezeichnungen ersetzt wurden, die Hinweise auf links und rechts oder auf vorne und hinten geben.

Ein ganz einfacher Fall, bei dem man diese Klebe- bzw. Zerschneidefunktionen braucht, liegt vor, wenn ein Ergebnis einer Funktion eine Menge mehrerer Teile ist, die anschließend auch einzeln verwendet werden sollen. Eine Funktion kann zwar eine Menge als Ergebnis liefern, aber dann ist diese Menge doch ein einzelnes Individuum. Die Elemente der Menge müssen also zu diesem Individuum zusammengefügt werden, und genau dies macht man mit Hilfe der CONS-Funktion. Anschließend kann man aus der Menge einzelne Elemente herauslösen durch die Zerschneidefunktionen.

Da es verhältnismäßig oft vorkommt, dass man die Funktionen CAR und CDR verkettet mehrfach anwenden muss, hat man hierfür eine Abkürzungsmöglichkeit vorgesehen. Bild 35 veranschaulicht dies an Hand von Beispielen.

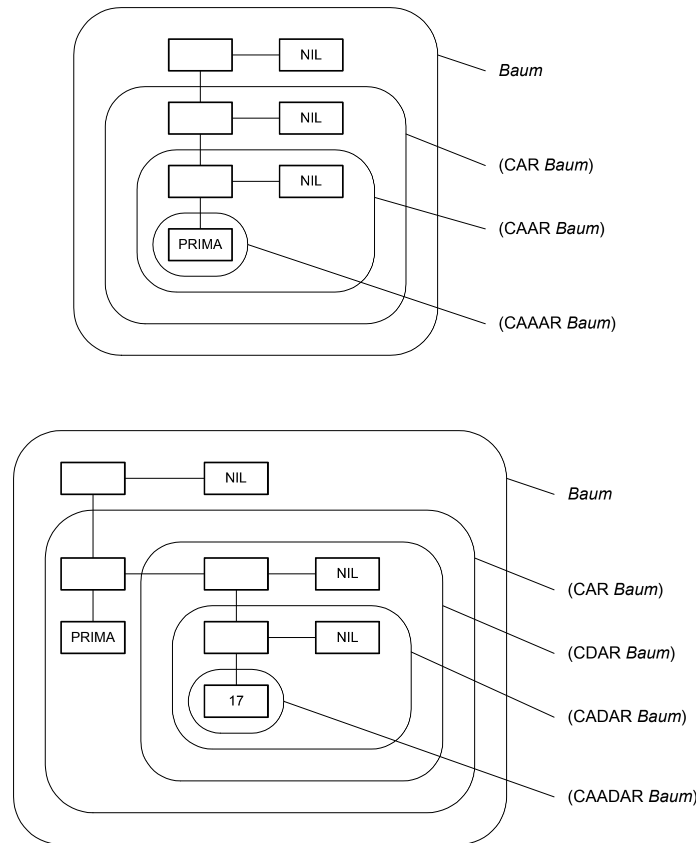


Bild 35 Beispiele für die Verkettung von CAR und CDR

Wenn man eine bestimmte Funktion kennen gelernt hat, sollte man auch die Frage stellen, ob es nicht dazu eine Umkehrfunktion gibt. So gibt es beispielsweise zu der Zusammenbaufunktion CONS auch die Zerschneidefunktionen CAR und CDR.
Die Umkehrfunktion zu QUOTE gibt es auch. Es ist die Funktion EVAL (Abkürzung von Evaluation). Die Funktion QUOTE verhindert die üblicherweise automatisch stattfindende Auswertung eines Argumentbaumes. Die Funktion EVAL dagegen erzwingt die Auswertung eines Baumes, der sonst nicht ausgewertet werden würde. Es gilt also der Zusammenhang

Wert von [(EVAL (QUOTE *Baum*))] = Wert von [*Baum*]

Wir betrachten hierzu ein Beispiel:

(EVAL (CONS ‘+ ’(3 4)))

In diesem Beispiel wird durch CONS ein auswertbarer Baum zusammengebaut, der durch die Liste (+ 3 4) darstellbar ist. Durch EVAL wird die Auswertung dieses Baumes verlangt; das Ergebnis ist 7. Wenn man das Zeichen + , das als erstes Argument von CONS vorkommt, ohne Quotierung geschrieben hätte, würde es vom Interpreter als Symbol mit einer hier nicht interessierenden Sonderbedeutung angesehen.

Es gilt allgemein, dass die Funktion EVAL dazu dient, Bäume auszuwerten, die aus Teilen zusammengebaut wurden. Das typische Beispiel hierfür ist die Differentiation. Die zu

differentierende Funktion ist in der alphanumerischen Form ein grammatikalisch strukturierter Text, dessen Struktur in Baumform erfasst werden kann. Dieser Baum wird in seine Einzelteile zerlegt, denen dann über die Differentiationsregeln neue Teilbäume zugeordnet werden, aus denen der Ergebnisbaum zusammengebaut wird. Wenn man dann noch will, dass die Ergebnisfunktion auf ein bestimmtes Argument angewandt wird, muss man auch noch das Argument mit dem funktionsbeschreibenden Baum zusammenfügen und auf diesen Baum die EVAL-Funktion anwenden.

Damit endet unsere Betrachtung der funktionalen Programmierung.

****