Rekursion

Wir kommen nun zu einem ganz neuen Thema, nämlich dem Prinzip der Rekursion. Viele Aufgaben, zu denen man einen lösenden Algorithmus sucht, lassen sich durch Rekursion lösen. Bevor wir aber in das Thema Rekursion einsteigen, möchte ich zuvor noch auf den Unterschied zwischen streng algorithmisch lösbaren Aufgaben und heuristisch zu lösenden Aufgaben hinweisen. Streng algorithmisch lösbare Aufgaben sind solche, bei denen auch der Mensch nach einem Algorithmus verfahren würde, wenn er die Lösung ohne Rechner finden müsste. Man denke hier an die Multiplikation mehrstelliger Zahlen. Heuristisch zu lösende Aufgaben sind solche, bei denen der Mensch die Aufgabe lösen kann ohne dass ihm bewusst wird, wie er die Aufgabe löst. Solche Aufgaben gehören in den Bereich der künstlichen Intelligenz. Ein typisches Beispiel hierfür ist die Erkennung von Sprechern an ihrer Stimme. Jeder Mensch ist in der Lage, sprechende Personen, die er nicht sieht, aber die er schon lange kennt, an ihrer Stimme zu erkennen. Man erhält aber keine brauchbare Antwort, wenn man den Menschen fragt, wie er das denn macht. Heuristik ist die experimentelle Lösungssuche. Man erfindet einen Lösungsansatz, von dem man vermutet, dass er möglicherweise eine mehr oder weniger brauchbare Lösung darstellt, und überprüft dann die Brauchbarkeit dieses Ansatzes. In den Bereich der Heuristik gehört auch das Programmieren von Computern für das Schachspiel. In diesen Fällen wird nicht das nachgebildet, was im Kopf eines menschlichen Schachspielers vorgeht.

Als erstes Beispiel, für das wir eine elegante rekursive Lösung angeben können, betrachten wir das Spiel „Türme von Hanoi“. In diesem Spiel besteht ein Turm aus aufeinanderliegenden runden Scheiben, bei denen die größte unten und die kleinste oben liegt. Die Scheiben haben alle ein Loch in der Mitte, welches dazu dient, die Scheiben jeweils an einer von drei durch Stifte definierten Positionen auf dem Spielbrett zu positionieren (siehe Bild 41). Zu Beginn des Spiels sitzt der Turm auf der linken Position. Die Aufgabe besteht darin, den Turm auf die rechte Position zu transportieren unter Beachtung der folgenden beiden Bedingungen:

1. Pro Schritt darf immer nur eine Scheibe bewegt werden.
2. Es darf nie eine größere Scheibe auf eine kleinere gelegt werden.

Für dieses Spiel gibt es eine optimale Schrittfolge mit $2^m - 1$ Schritten, wobei m die Anzahl der Scheiben ist. Mit dieser Minimalzahl von Schritten kommt man aus, wenn man im ganzen Spiel keinen einzigen falschen Schritt tut. Die Aufgabe besteht nun darin, diese Schrittfolge für eine variable Scheibenanzahl m zu bestimmen.

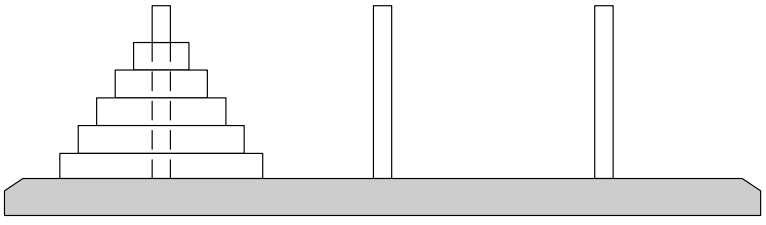


Bild 41 Türme von Hanoi

Bei der Suche nach einem Lösungsalgorithmus für irgendein Problem ist es immer zweckmäßig, sich zuerst einmal zu fragen, ob es nicht eine einfache rekursive Lösung gibt. Unabhängig von der aktuellen Aufgabe besteht eine rekursive Lösung immer aus den folgenden Bestimmungsstücken:

Es gibt immer mindestens einen Fall, wo die Lösung trivial ist und unmittelbar auf der Hand liegt. Im Beispiel der Türme von Hanoi ist dies das Spiel mit nur einer Scheibe. Dann muss man nur diese Scheibe von der Startposition auf die Zielposition legen und ist fertig.
Das zweite Bestimmungsstück einer rekursiven Lösung besteht in der sogenannten Argumentreduktion. Man sucht nach einer Konstruktion einer Lösung für die ursprünglich gegebene Aufgabe unter der Annahme, dass man den gleichen Aufgabentyp mit einem reduzierten Argument schon gelöst hätte. In unserem Beispiel heißt dies, dass wir die Lösung für m Scheiben konstruieren wollen unter der Annahme, wir wüssten schon, wie die Lösung für $m-1$ Scheiben aussieht. Die Argumentreduktion muss die Bedingung erfüllen, dass man durch endlich viele Reduktionsschritte von der ursprünglichen Aufgabenstellung zu der trivialen Aufgabenstellung gelangt, für die man die Lösung schon kennt. In unserem Falle gelangt man durch sukzessive Verringerung der Scheibenanzahl um 1 von jeder beliebigen Scheibenanzahl $m>1$ nach endlich vielen Schritten zur Scheibenanzahl 1.

Wir nehmen an, das Spiel solle mit 7 Scheiben gespielt werden. Dann lautet die rekursive Lösung:
„Transportiere den Turm aus 6 Scheiben von der linken Startposition auf die mittlere Hilfsposition. Anschließend bewege die siebte, also die größte Scheibe von der linken Startposition zur rechten Zielposition. Zuletzt transportiere den Turm aus 6 Scheiben von der Hilfsposition in der Mitte zur rechten Zielposition.“

Die allgemeine Formel für die rekursive Lösung lautet:

$$H(\,m \text{ Scheiben von } start \text{ über } hilf \text{ nach } ziel\,) = \begin{cases} (start \rightarrow ziel) & \text{falls } m=1 \\ H(m\text{-}1, start, ziel, hilf) \cdot (start \rightarrow ziel) \cdot H(m\text{-}1, hilf, start, ziel) & \end{cases}$$

In dieser Formel ist durch den dicken Punkt der sogenannte Konkatenationsoperator symbolisiert, der das Hintereinanderhängen von Folgen verlangt.

Wer noch nicht mit der Verwendung von Rekursion vertraut ist, hat anfänglich Schwierigkeiten, eine rekursiv formulierte Lösung überhaupt als Lösung zu akzeptieren. Denn wenn man nicht weiß, wie die Lösung mit 7 Scheiben aussieht, dann weiß man doch auch nicht, wie sie mit 6 Scheiben aussieht. Wir werden später aber sehen, dass es einen systematischen Weg gibt, die rekursiv formulierte Lösung in eine konkrete Schrittfolge umzusetzen.

Jetzt soll aber erst noch gezeigt werden, dass es für die „Türme von Hanoi“ auch eine nicht rekursive Lösung gibt. Diese ist aber nicht so leicht zu finden wie die rekursive Lösung.

Bei einer Anlayse der optimalen Spielschrittfolge, die sich aus der rekursiven Lösung ergibt, erkennt man die folgenden Zusammenhänge:
Erste Erkenntnis:
Die Scheiben lassen sich in zwei Gruppen einteilen, die durch ihren Wegetyp gekennzeichnet sind (siehe Bild 68). Bei einer Analyse des optimalen Spiels stellt man nämlich fest, dass alle schattierten Scheiben einen gemeinsamen Weg gehen und alle unschattierten Scheiben einen anderen gemeinsamen Weg gehen.

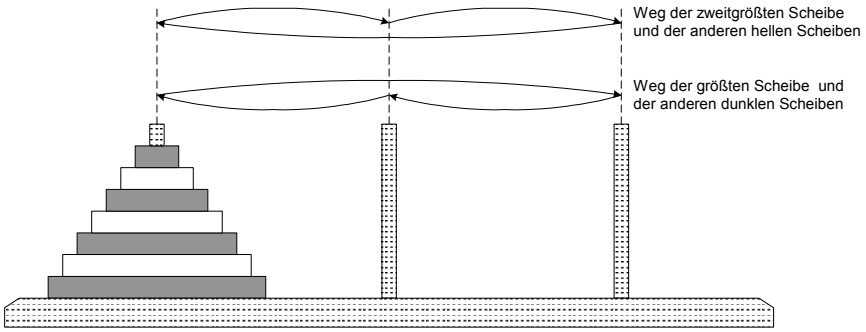


Bild 68 Einteilung der Hanoischeiben nach ihrem Wegetyp

Eine geschickte Codierung von Scheibenpositionen und Wegetypen sieht wie folgt aus:
- Positionen: {LINKS, MITTE, RECHTS} ⇔ {0, 1, 2}
- Wegetypen: {wieGrößteScheibe, wieZweitgrößteScheibe} ⇔ {-1, +1}

Denn dann kann die nächste Position aus der aktuellen Position einfach wie folgt berechnet werden:

$$\text{nächstePosition} = (\text{aktuellePosition} + \text{Wegetyp}) \bmod 3$$

Zweite Erkenntnis:
Im ersten und in allen weiteren Schritten mit ungerader Ordnungsnummer wird die kleinste Scheibe bewegt. Da kann man auf die Idee kommen, die Ordnungsnummern der Spielsituationen als Dualzahlen zu codieren und zu schauen, ob man daraus die Information gewinnen kann, in welcher Reihenfolge die Scheiben bewegt werden müssen.

In Bild 68a wird das Spiel mit 4 Scheiben betrachtet, bei dem es 16 Spielsituationen und 15 Spielschritte gibt. Rechts außen sind die Spielschritte aufgeführt, die man aus der rekursiven Lösung kennt. Man erkennt schnell, dass überall dort, wo in einer Scheibenspalte ein Übergang von 0 nach 1 vorkommt, die zugehörige Scheibe versetzt werden muss. Welchen Weg sie dabei nimmt, ergibt sich aus ihrem Wegetyp nach Bild 68.

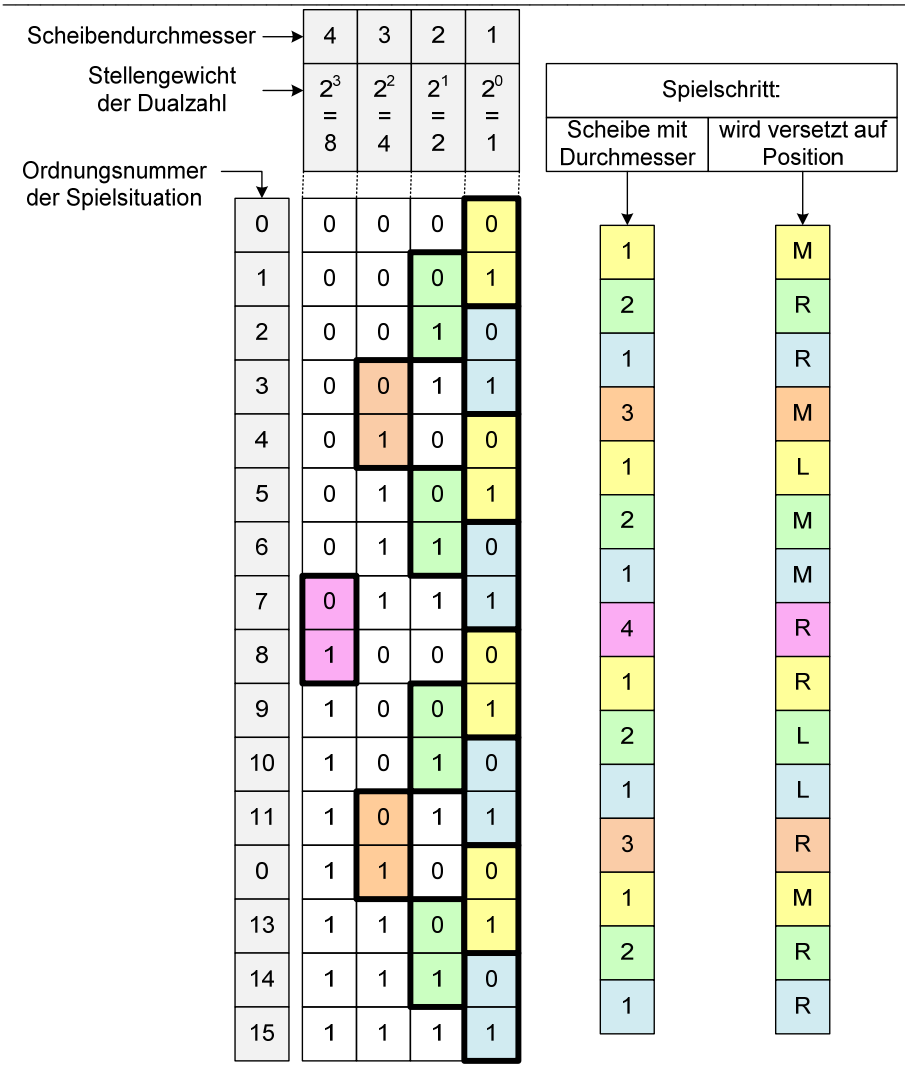


Bild 68a Die optimale Folge der Spielschritte bei einem Spiel mit 4 Scheiben

---

Als zweites Beispiel einer rekursiven Formulierung betrachten wir die Fakultätsfunktion

$$0! = 1 \qquad \text{und} \qquad n! = 1 * 2 * 3 * \ldots * (n-1) * n \qquad \text{für } n>0$$

Im Falle der Fakultät gibt es keinen Grund, nach einer rekursiven Formulierung zu suchen, denn die Berechnung der Fakultät nach dem gegebenen iterativen Schema ist ja sehr einleuchtend. Man kann aber feststellen, dass Mathematiker die Rekursion so sehr lieben, dass sie sie auch in Fällen anwenden, wo man sie nicht braucht. Die rekursive Definition der Fakultätsfunktion lautet:

$$n! = \begin{cases} 1 & \text{falls } n=0 \\ n * (n-1)! & \text{falls } n>0 \end{cases}$$

Auch in diesem Fall finden wir wieder den trivialen Fall, für den die Lösung unmittelbar gegeben ist. Daneben finden wir wieder die Argumentreduktion, die auch in diesem Fall durch Subtraktion von 1 gegeben ist. Auch diese Subtraktion von 1 führt in endlich vielen Schritten zu dem trivialen Fall.

Der Begriff der Rekursion wurde nicht erst im Zusammenhang mit Computern und deren Programmierung erfunden. Rekursion ist ein lange bekannter mathematischer Begriff. Es handelt sich letztlich nur um eine spezielle Variante der sogenannten „vollständigen Induktion“, die ein bestimmtes Prinzip zum Beweis von Allaussagen ist. Mit dem Prinzip der vollständigen Induktion beweist man Sätze der Form
„Für alle Elemente x_i einer diskreten geordneten unendlichen Folge gilt das Prädikat $P(x_i)$.“
Der Beweis besteht aus zwei Teilen:

(1) Man zeigt, dass $P(x_1)$ gilt.
(2) Man zeigt, dass aus der angenommenen Gültigkeit von $P(x_n)$ für beliebige Werte der natürlichen Zahl n die Gültigkeit von $P(x_{n+1})$ folgt.

Auf die Rekursion übertragen entspricht der Teil (1) der bekannten Lösung für das Trivialargument, und der Teil (2) entspricht der Konstruktion der Lösung für das ursprüngliche Argument unter Verwendung der angenommenen Bekanntheit der Lösung für das reduzierte Argument.

Eine rekursive Lösung ist oft die erste, die man zu einer gegebenen Aufgabe findet. In diesem Fall ist die formulierte Lösung gut verständlich wie im Beispiel der Türme von Hanoi. Anders liegt der Fall im Beispiel der Fakultätsfunktion, wo die iterative Formulierung sehr viel leichter verständlich ist als die rekursive. In diesem Fall kann man die rekursive Formulierung vermutlich nur dann leicht verstehen, wenn man die iterative Formulierung schon kennt.

Häufig ist eine rekursiv formulierte Lösung nicht optimal in Hinblick auf Rechenzeit und Speicherplatz. Deshalb ist es immer angebracht zu fragen, ob es zu der Aufgabe, die man rekursiv gelöst hat, nicht doch noch eine für die Berechnung aufwandsgünstigere nichtrekursive Lösung gibt.

Die 5. Übungsaufgabe als Beispiel für Rekursion

Die erste Aufgabe, wo Sie eine rekursive Lösung formulieren sollen, ist die 5. Übungsaufgabe. In diesem Beispiel kommen zwei Rekursionen vor, von denen die eine im Programm bereits ausformuliert ist. Sie müssen also nur die zweite Teilaufgabe lösen, die auch eine Rekursion erfordert.

Bei dieser Aufgabe geht es darum, dass zuerst per Interaktion zwischen dem Benutzer und dem Rechner ein Binärbaum eingegeben wird, zu dem dann anschließend ein Zahlenwert berechnet werden soll. Die einzugebenden Bäume sind sogenannte arithmetische Binärbäume,

bei denen in den Blättern Zahlen stehen und in den Nichtblättern arithmetische Operatoren. Bild 42 zeigt ein Beispiel.

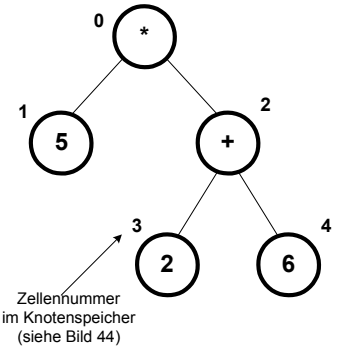


Bild 42 Beispiel eines arithmetischen Binärbaums

Jedem Knoten in einem solchen Baum ist ein Zahlenwert zuordenbar, den Blättern der hineingeschriebene Zahlenwert und den Operatoren das Ergebnis, welches man durch arithmetische Verknüpfung aus den beiden am Operator hängenden Teilbaumwerten erhält.

Bild 43 zeigt das Schichtungsbild für das Programm der 5. Übungsaufgabe. Die Prozedur `wert` ist diejenige, die von Ihnen programmiert werden soll. In dem Schichtungsplan ist erkennbar, dass die Prozeduren `wert` und `grow` rekursiv aufgerufen werden, denn es gibt für beide Prozeduren ein Aufrufkringel, zu dem es einen Weg gibt, der von unten nach oben führt. Im Falle der Rekursion ist also das eigentliche Schichtungsprinzip durchbrochen, welches nur einen eindeutigen Abstieg von oben nach unten erlaubt. Es lässt sich aber auch im Falle der Rekursion immer feststellen, woher der erste Aufruf einer Prozedur kommt. Bevor die Prozedur `wert` sich selbst aufrufen kann, muss sie zuerst einmal von `main` aufgerufen worden sein. Entsprechendes gilt für die Prozedur `grow`.

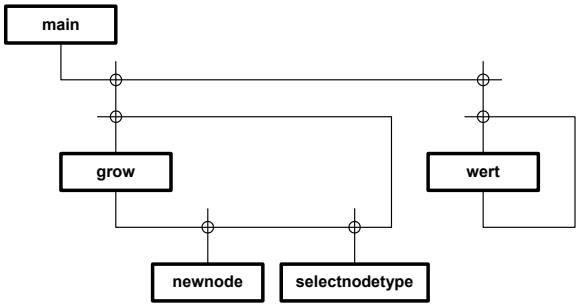
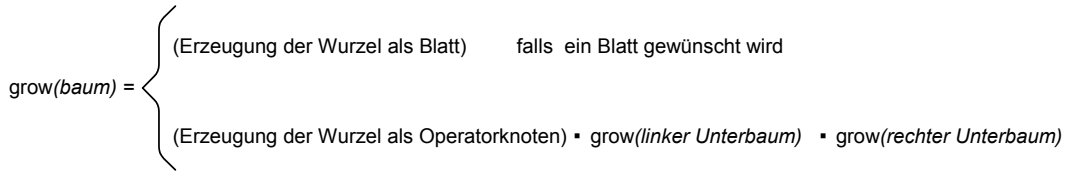


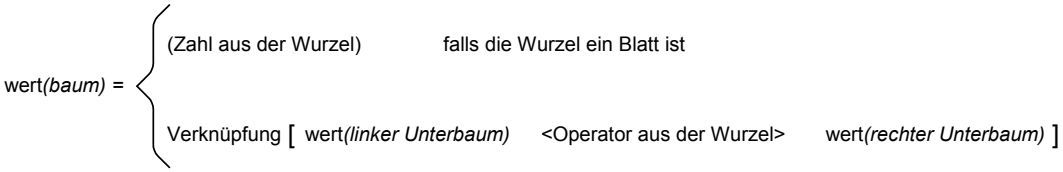
Bild 43 Schichtungsplan eines Programms zur Eingabe und Wertbestimmung eines arithmetischen Binärbaums

Für die Prozedur `grow` gilt das folgende rekursive Konzept:



An dieser Stelle weise ich darauf hin, dass das von mir vorgegebene Programm `grow` nicht robust ist gegen Eingabefehler. Es wurde einfach angenommen, dass Sie bei der Eingabe für die Inhalte von Blattknoten tatsächlich nur Zahlen eingeben und für die Inhalte von Operatorknoten nur Operatoren. Da es sich nicht verhindern lässt, dass sich ein Benutzer bei der Eingabe nicht an die Vereinbarungen hält, darf man ein solches Programm nie tatsächlich als Produktprogramm ausliefern. In unserem Falle aber, wo wir uns auf die Rekursion konzentrieren wollen, erschien es mir besser, die Forderung nach Robustheit nicht zu erfüllen, weil sonst zuviel Programmcode entstanden wäre, der das Rekursionsprinzip überdeckt hätte.

Für die von Ihnen zu gestaltende Prozedur `wert` gilt das folgende rekursive Schema:



****

Speicherstruktur für das Übungsbeispiel

In unserer Aufgabe müssen wir die offene Datenstruktur „Binärbaum“ realisieren, denn wir können zum Zeitpunkt der Programmierung nicht wissen, wie groß ein Baum werden wird. Es liegt ja in der Entscheidung des eingebenden Benutzers, wie groß der Baum wird.

Zur Realisierung offener Datenstrukturen benötigt man eine Speicherverwaltung, bei der man Speicherzellen anfordern kann und die dem Anfordernden einen Pointer zur Identifikation der bereitgestellten Speicherzelle zurückliefert. In vielen höheren Programmiersprachen ist das Ansprechen der Speicherverwaltung durch verhältnismäßig komfortable Sprachkonstrukte möglich. Im Falle von C muss man aber die Speicherverwaltung auf sehr tiefer maschinennaher Ebene ansprechen, denn C ist keine reine höhere Programmiersprache, sondern enthält durchaus Möglichkeiten, wie sie ein Assembler bietet. C ist eigentlich eine Sprache für die Systemprogrammierung und nicht für die Anwendungsprogrammierung.
Damit Sie in C nicht eine Speicherverwaltung auf sehr tiefer Ebene benutzen müssen, habe ich Ihnen eine übersichtliche Speicherstrukturierung vorgegeben. Bild 44 zeigt meine vorgegebene Struktur.

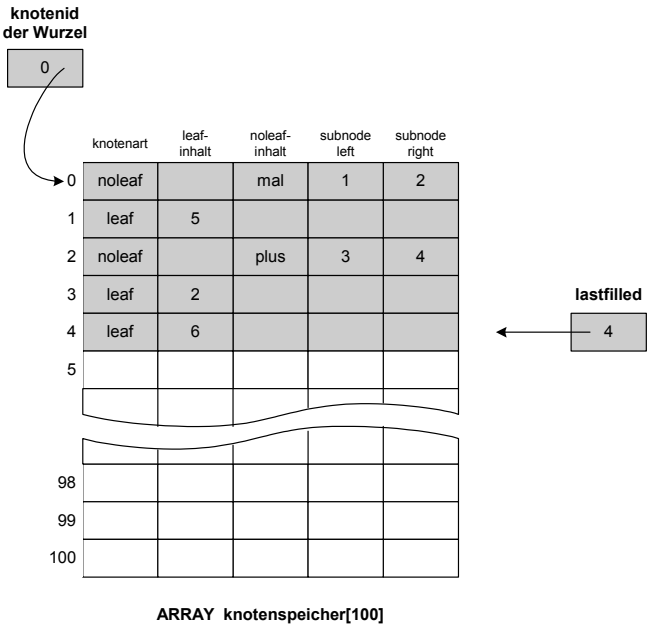


Bild 44 Speicherung des arithmetischen Binärbaums aus Bild 42 in einem Array

Die Reihenfolge der Einträge ergibt sich aus der sogenannten Linksabwicklung des Baumes. Eine Linksabwicklung ist gegeben, wenn man beim Abstieg von einem Knoten zu seinen Teilknoten immer zuerst den linken Weg beschreitet und erst später den rechten Weg (siehe die Ordnungsnummern in Bild 42).

Von der Rekursion zur Wiederholung

Nun behandeln wir die Methode, die es ermöglicht, eine rekursiv formulierte Lösung in eine Schrittfolge zu überführen. Im Zentrum dieser Methode steht das sogenannte Stack-Prinzip. Stack ist das englische Wort für Stapel; man denke an einen Stapel von Akten oder von Tellern. Für die Nutzung des Stack-Prinzips ist es irrelevant, ob der Stack mit einem festen Boden oder mit einer festen Oberkante realisiert wird. Das anschauliche Beispiel für einen Stack mit festem Boden ist ein Stapel von Büchern oder Tellern auf einem Tisch. Wenn ein Element von diesem Stapel entnommen wird oder ein weiteres Element auf diesen Stapel gelegt wird, verschiebt sich die Oberkante des Stapels, während der Boden, auf dem der Stapel steht, seine Lage beibehält. Es gibt aber auch Stapel, insbesondere Stapel von Tellern oder Tabletts in Kantinen, bei denen die Oberkante ihre Lage beim Entnehmen oder beim Hinzufügen von Elementen ihre Lage nicht ändert, weil hier der Boden beweglich ist. Der Boden ist in diesem Fall gefedert in einem Gehäuse befestigt; wenn ein Element auf den Stapel gelegt wird, erhöht sich das Gewicht des Stapels und drückt den Boden genau um so viel nach unten, wie das neue Element an Höhe hinzubringt.

Im Bereich der Programmierung wird der Stack immer dann verwendet, wenn es darum geht, eine aktuell bearbeitete Aufgabe zu Gunsten einer neu gestellten dringlicheren Aufgabe vorläufig zurückzustellen. Man stelle sich vor, man sitze am Schreibtisch und bearbeite gerade

ein bestimmtes Schriftstück. Es kann nun vorkommen, dass man gebeten wird, die Arbeit zu unterbrechen, um eine dringlichere Aufgabe zu beginnen; in diesem Fall wird man das Schriftstück zur Seite legen und mit einem leeren Papier die neue Aufgabe in Angriff nehmen. Auch dabei kann man wieder unterbrochen werden und wird dann auch dieses Papier wieder auf den Stapel legen, der auf diese Weise immer höher werden kann. Erst wenn die dringlichste Aufgabe erledigt ist, wird man sich vom Stapel die Unterlagen herunternehmen, die ganz oben liegen. Denn im Stapel sind die Unterlagen gemäß ihrer Dringlichkeit geordnet: Ganz unten liegt die am wenigsten dringliche Aufgabe, und ganz oben liegt die dringlichste Aufgabe, die fortgesetzt werden kann, wenn die aktuelle Aufgabe erledigt ist.

Dass das Stack-Prinzip genau das richtige Prinzip für die Realisierung rekursiv formulierter Lösungen ist, erkennt man leicht, indem man das Beispiel der Fakultätsberechnung betrachtet. Die ursprüngliche Aufgabe bestehe darin, die Fakultät zum Argument 4 zu berechnen. Die rekursive Definition sagt nun,

dass die Berechnung von 3! dringlicher ist als die Berechnung von 4!, und
dass die Berechnung von 2! dringlicher ist als die Berechnung von 3!, und
dass die Berechnung von 1! dringlicher ist als die Berechnung von 2! und
dass die Berechnung von 0! dringlicher ist als die Berechnung von 1!.

Es ergibt sich somit die Folge von Belegungen des Aufgaben-Stack, die in Bild 45 dargestellt ist.

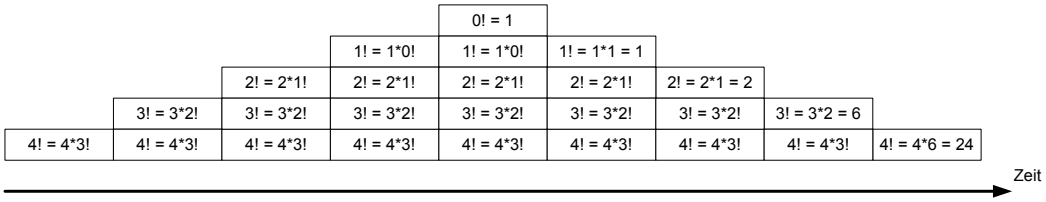


Bild 45 Folge der Belegungen des Stacks bei der Berechnung der rekursiv definierten Fakultätsfunktion für das Argument 4

Zur Benennung von Operationen bzw. Funktionen bezüglich eines Stacks sind die folgenden Bezeichner international üblich:

PUSH (e)	Lege das Element e oben auf den Stack.
TOP	Sage mir, welches Element aktuell oben auf dem Stack liegt.
POP	Entferne das oberste Element vom Stack.
HEIGHT bzw. DEPTH	Sage mir wieviele Elemente aktuell im Stack liegen.

Die Funktion TOP und die Operation POP sind nur bei einem nichtleeren Stack definiert.

Der Stack wird häufig als sogenannter LIFO-Speicher bezeichnet. LIFO ist die Abkürzung für „last in, first out“. Dadurch wird zum Ausdruck gebracht, dass das letzte Element, welches auf den Stack gelegt wird, das erste sein wird, welches wieder herunter genommen werden kann. Es gibt nicht nur die Vorstellung des vertikalen Stacks, sondern man kann sich auch ei-

nen horizontalen Stack vorstellen. Denken Sie an einen sehr schmalen, aber tiefen Fahrstuhl, in den nacheinander Personen eintreten. Die erste Person wird, wenn der Fahrstuhl anfährt, ganz hinten an der Wand stehen, während die Person, die zuletzt eingetreten ist, ganz vorne an der Türe steht. Wenn der Fahrstuhl dann seine Zielposition erreicht hat und die Türe wieder aufgeht, wird die Person, die als letzte in den Fahrstuhl eingetreten ist, als erste den Fahrstuhl wieder verlassen.

Wenn wir danach fragen, wie eine rekursiv definierte Funktionsberechnung in eine Schrittfolge überführt wird, fragen wir eigentlich danach, was der Compiler aus dem in höherer Sprache formulierten Programm macht. Wir betrachten hierzu die folgende Kombination aus einem Hauptprogramm und einem Funktionsprogramm:

Hauptprogramm:

```
{
 n:=READ();
 PRINT(fak(n));
}
```

Funktionsprozedur:

```
DEF INTEGER fak(INTEGER m)
{
 IF m=0 THEN RETURN 1
 ELSE RETURN m*fak(m-1);
}
```

Der Compilerbauer muss festlegen, in welcher konkreten Form er den Stack verwenden will. Das bedeutet, dass er festlegen muss, wie die sogenannte PRE-Belegung und die sogenannte POST-Belegung des Stacks aussehen sollen. Mit PRE-Belegung bezeichnen wir die Stack-Situation, die der Auftragnehmer vom Auftraggeber übergeben bekommt. Mit POST-Belegung bezeichnen wir die Stack-Situation, die der Auftraggeber als Ergebnis vom Auftragnehmer zurück bekommt. Im Beispiel der Fakultätsberechnung ist es zweckmäßig, die PRE- und POST-Belegungen so festzulegen, wie es das Bild 46 zeigt: In der PRE-Belegung liegt als oberstes Element das Argument n auf dem Stack, und in der POST-Belegung liegt anstelle des Arguments das Ergebnis n! als oberstes Element auf dem Stack. Der PRE-Stack und der POST-Stack haben also die gleiche Höhe und sie unterscheiden sich nur bezüglich des obersten Elements.

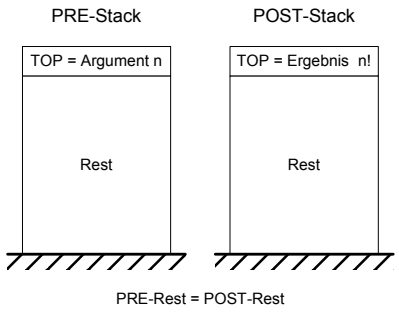


Bild 46 PRE- und POST-Stack für die rekursiv definierte Fakultätsberechnung

Welche Stack-Belegungen bei der Berechnung von 4! aufeinander folgen, und wie sie in PRE- und POST-Belegungen unterteilt sind, zeigt das Bild 47.

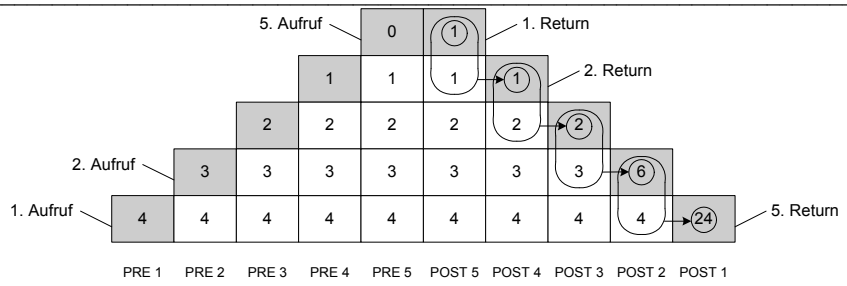


Bild 47 Folge der Stack-Belegungen bei der Berechnung von 4!

Wir haben imperative Programme, soweit sie noch keine Rekursion enthielten, stets in Form eines Petrinetzes dargestellt. Es stellt sich nun die Frage, ob es nicht auch möglich ist, das Konzept der Rekursion mit dem Konzept des Petrinetzes zu verbinden. Diese Frage habe ich mir bereits im Jahre 1975 gestellt, und meine Lösung habe ich in einem Aufsatz mit dem Titel „Modified Petri Nets as Flowcharts for Recursive Programs“ veröffentlicht.

Wenn wir nach einer neuen Darstellungsmöglichkeit für einen zu beschreibenden Sachverhalt suchen, sollte diese Bemühung immer durch unsere Unzufriedenheit mit den aktuellen Beschreibungsformen motiviert sein. Selbstverständlich gibt es kein strenges Maß zur Beurteilung von Beschreibungen, aber es ist durchaus möglich, beim Vergleich zweier Beschreibungen des gleichen Sachverhalts eine Qualitätswertung vorzunehmen. Man fragt sich einfach, wie lange man braucht, um aus einer gegebenen Beschreibung zu einem befriedigenden Verständnis des Beschriebenen zu kommen. Wenn man bei einer gegebenen Beschreibung einen ganzen Tag benötigt, um zu einem befriedigenden Verständnis zu kommen, kann dies zum Anlass werden, nach einer deutlich besseren Beschreibung zu suchen. Nachdem man den Sachverhalt schließlich verstanden hat, kann man nämlich durchaus zu der Überzeugung gelangt sein, dass dieser Sachverhalt gar nicht so kompliziert ist, dass man unbedingt einen ganzen Tag benötigt, um ihn zu verstehen. Man kann durchaus intuitiv das Gefühl haben, dass eine deutlich bessere Beschreibung möglich sein müsste, aus der man das gleiche Verständnis in einer Stunde gewinnen kann. In einer ähnlichen Situation befand ich mich vor über 25 Jahren, als mir eine Beschreibung eines rekursiven Programms vorlag, die ich auch nach einem ganzen Tag noch nicht befriedigend verstanden hatte. Meine Überlegungen, die zu einer neuen Form der Beschreibung führte, möchte ich Ihnen nun vorstellen.

In Bild 48 sind die Abläufe des Hauptprogramms und der Funktionsprozedur zur Fakultätsberechnung in Form von Petrinetzen dargestellt. Jedes dieser beiden Petrinetze ist durch eine vertikale Linie in zwei Teile zerschnitten: Links befindet sich der leicht verständliche Teil, der die Schritte des jeweiligen Auftraggebers beschreibt, und rechts von der Schnittlinie befindet sich jeweils die schattierte Transition, die aus der Sicht des Auftraggebers in die Zuständigkeit eines Auftragnehmers fällt. Die Aufgabe des jeweiligen Auftragnehmers besteht darin, die PRE-Belegung des Stacks in die POST-Belegung zu überführen. Aus der Sicht des Hauptprogramms ist der Auftragnehmer durch die Funktionsprozedur definiert, und aus der Sicht der Funktionsprozedur ist der Auftragnehmer auch durch die Funktionsprozedur definiert. Man müsste also die schattierten Transitionen durch den Ablauf der Funktionsprozedur ersetzen. Ein erster Versuch einer solchen Ersetzung ergibt die Struktur in Bild 49. Diese Ablaufstruktur kann aber noch nicht die endgültige Lösung sein, denn hier gibt es eine nicht entschiedene Verzweigung des Markenweges, der von der mit R bezeichneten Stelle ausgeht; R steht hier für Return bzw. Rückkehr zum Auftraggeber.

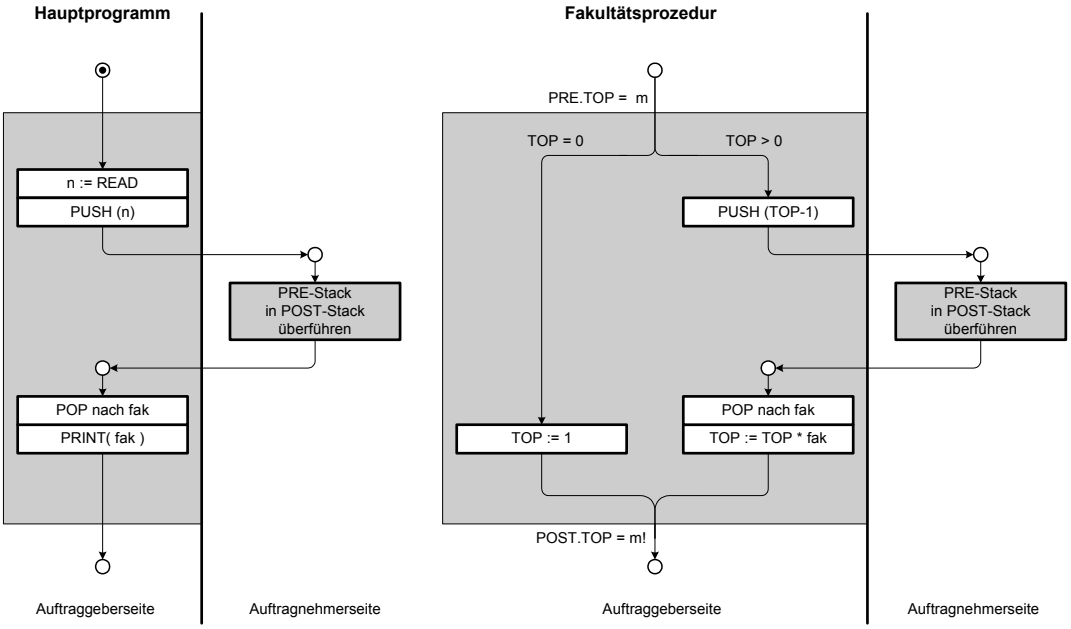


Bild 48 Erster Ansatz zur Darstellung des Ablaufs der rekursiven Fakultätsberechnung

Es ist leicht einzusehen, entlang welcher Wege die Marke im Falle der Berechnung von 4! durch das Petrinetz in Bild 49 fließen muss. Zuerst erreicht die Marke von links kommend die Eintrittsstelle in die Prozedur. Danach wird viermal die Stack-Aufbauschleife durchlaufen. Anschließend wird viermal die Stack-Abbauschleife durchlaufen, und zum Schluss wandert die Marke von der R-Stelle nach links ins Hauptprogramm zurück.

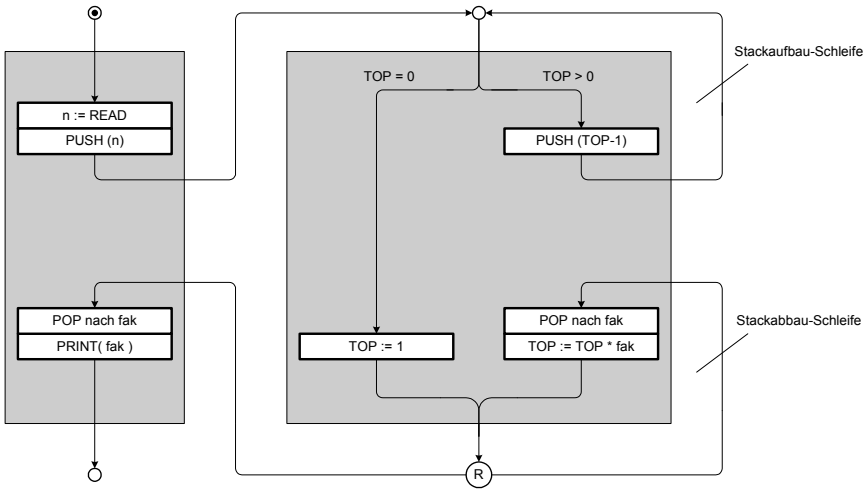


Bild 49 Zwischenergebnis auf der Suche nach einer Darstellung

Die Entscheidung, wohin eine auf R liegende Marke fließen muss, ist anhand der Frage zu entscheiden, woher denn der letzte Auftrag stammt, d.h. von welcher Stelle zuletzt der Prozedureingangsplatz belegt wurde. Auch hier gilt ja das Stack-Prinzip, dass der zuletzt erteilte Auftrag derjenige ist, der als erster fertig wird. Deswegen reichern wir nun das Petrinetz aus Bild 49 mit sogenannten Stack-Plätzen an, wie es in Bild 50 gezeigt ist. Ein mit S bezeichne-

ter Stack-Platz wird immer genau dann belegt, wenn eine Auftragserteilung erfolgt. Also muss jedes Mal, wenn eine Marke vom R-Platz abfließen muss, diese Marke zu einer Transition gehen, vor der eine S-Marke liegt. Um die Reihenfolge der Auftragserteilungen berücksichtigen zu können, stellen wir uns einfach vor, auf den Stack-Marken wäre jeweils der Zeitpunkt ihrer Entstehung vermerkt. Dann fällt die Verzweigungsentscheidung bei R derart, dass die Marke von R dorthin fließt, wo die jüngste S-Marke liegt.

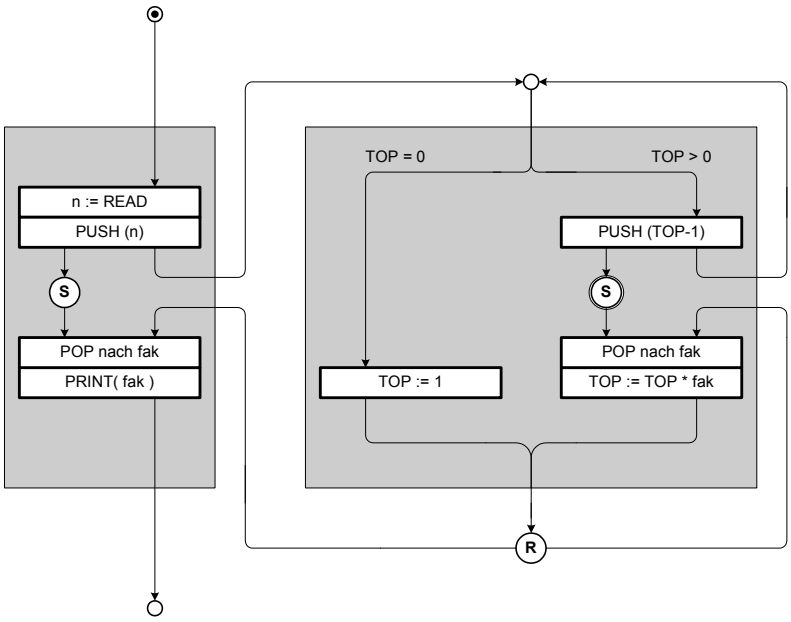


Bild 50 Endfassung der Darstellung des Ablaufs der rekursiven Fakultätsberechnung

Der S-Platz im Hauptprogramm ist nur mit einem einfachen Kreis berandet, womit symbolisiert wird, dass hier maximal eine Marke liegen kann. Dagegen ist der S-Platz in der Funktionsprozedur doppelt berandet, was darauf hinweisen soll, dass hier mehrere Marken gleichzeitig liegen dürfen. Diese mehreren Marken unterscheiden sich aber bezüglich ihres Alters. Wenn sowohl auf dem S-Platz im Hauptprogramm als auch auf dem S-Platz in der Prozedur Marken liegen, dann ist garantiert die Marke im Hauptprogramm die älteste. Deshalb erfolgt die Rückkehr vom R-Platz ins Hauptprogramm erst ganz zum Schluss.

Wir haben nun also zwei unterschiedliche Stacks eingeführt. Der sogenannte Datenstack äußert sich in den expliziten PUSH- und POP-Operationen, die in den Transitionen stehen. Der Stack für den Steuerfluss äußert sich in den S-Plätzen des Petrinetzes.

Nun betrachten wir ein etwas komplizierteres Beispiel, nämlich die Türme von Hanoi. Auch hier müssen wir wieder festlegen, wie die PRE- und POST-Belegung des Stacks aussehen soll. Diese Festlegung geht aus dem zugehörigen Petrinetz in Bild 51 hervor.

Nicht nur in den hier betrachteten Beispielen, sondern ganz allgemein gilt die Regel, dass man Petrinetze, die rekursive Vorgänge veranschaulichen, nicht lesen sollte, indem man die Schleifen verfolgt. Es genügt vollkommen, die jeweiligen Abläufe zu verstehen, die sich unter der Annahme ergeben, dass es jeweils einen Auftragnehmer gibt, der garantiert die PRE-Belegung des Stacks in die geforderte POST-Belegung überführen wird (s. Bild 48). Man muss also stets nur darauf achten, dass die PUSH/ POP- Bilanz in den Abläufen ausgeglichen

ist. Am Beispiel des Bildes 51 heißt dies, dass es genügt, die in den beiden großen Rechtecken enthaltenen Ablaufteile zu verstehen. Der Steuermechanismus auf der Grundlage des Stack-Konzepts bringt die Garantie, dass zu jeder Auftragserteilung die zugehörige Ergebnisrückkehr geliefert wird.

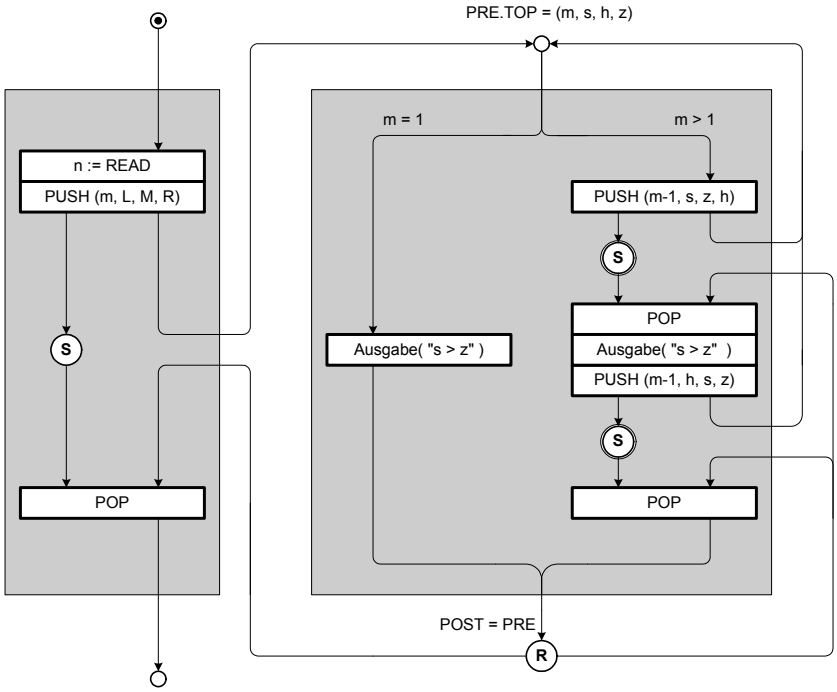


Bild 51 Ablauf für das Spiel „Die Türme von Hanoi“

Ich habe schon früher darauf hingewiesen, dass in den Fällen, wo eine rekursive Lösung besonders plausibel ist, man immer die Frage stellen sollte, ob diese Lösung denn auch hinsichtlich des Zeit- und Speicheraufwands akzeptabel ist, oder ob man nicht doch nach einer in dieser Hinsicht günstigeren Lösung suchen sollte. Wir stellen nun also die Frage, ob die in Bild 51 dargestellte Lösung im Hinblick auf den Aufwand verbesserungsbedürftig ist.

Es sind zwei Fragen, die wir durch Analyse der Ablaufstruktur in Bild 51 beantworten müssen:

(1) Wie oft werden die einzelnen Abschnitte des Petrinetzes durchlaufen?
(2) Wie hoch wird der Stack im äußersten Falle?

Da in jedem der beiden Zweige der Fallunterscheidung im Ablauf jeweils eine Ausgabeanweisung steht, wird die Prozedur genauso oft aufgerufen, wie es Spielschritte in der optimalen Folge gibt. Dies sind 2^n-1 Schritte. Der linke Pfad der Fallunterscheidung, der zu der Bewegung der kleinsten Scheibe gehört, wird 2^{n-1} mal durchlaufen. Die kleinste Scheibe wird in jedem zweiten Spielschritt bewegt. Der mittlere Abschnitt des rechten Pfades wird 2^{n-1}-1 mal durchlaufen. Der Stack kann nie mehr als n Elemente enthalten, wenn n die Anzahl der Scheiben im Spiel ist.

Diese Analyseergebnisse geben keinen Anlass zu der Befürchtung, dass der rekursive Algorithmus einen unzulässig hohen Laufzeit- oder Speicheraufwand erfordert. Der Speicheraufwand

wand wächst nur linear mit der Scheibenanzahl n. Der Laufzeitaufwand wächst zwar exponentiell mit der Scheibenanzahl, aber dieser Aufwand kann auch durch andere Algorithmen nicht reduziert werden, denn die optimale Schrittfolge erfordert 2^n-1 Schritte.

Zeitvarianter Systemaufbau

Bei der Frage, wie wir rekursive Abläufe realisieren können, ergab sich als Lösung das entsprechend modifizierte Petrinetz, worin es neben dem Daten-Stack noch den Marken-Stack gibt. Ich hätte aber auch eine andere Sicht vorstellen können, die ich jetzt kurz erwähnen will. Mit jedem Prozeduraufruf, egal ob es sich um einen rekursiven oder um einen nichtrekursiven Aufruf handelt, können wir die Vorstellung verbinden, dass eine neue Systemkomponente geschaffen wird, deren Verhalten durch durch die Prozedur beschrieben wird. Diese Vorstellung ist eine Konsequenz unserer strikten Trennung der Systembeschreibung vom beschriebenen System. Wenn ich eine Zeichnung anfertige, welche die Struktur einer Hundehütte zeigt, wird niemand diese Zeichnung mit einer konkreten Hundehütte verwechseln. Ich kann nun gemäß dieser Zeichnung mehrere Hundehütten herstellen, die alle das gemeinsame Merkmal haben, dass sie durch die Zeichnung korrekt beschrieben sind. Der Programmtext ist in jedem Falle Beschreibung. Daneben müssen wir uns aber immer das beschriebene dynamische System vorstellen. In diesem System sehen wir agierende Akteure, deren Agieren sich auf sogenannten Aktionsfeldern äußert. Wenn ich auf die Tafel schreibe, bin ich der Akteur und die Tafel ist das Aktionsfeld. Wenn ich zu Ihnen spreche, bin ich der Akteur und die Luft zwischen Ihnen und mir ist das Aktionsfeld.

Die Menge der Akteure in einem System muss nicht notwendigerweise konstant sein. Wir können durchaus die Vorstellung haben, dass ein Akteur einen weiteren Akteur erzeugt oder einen bestehenden Akteur eliminiert. Dies entspricht der Vorstellung, dass ein Mensch geboren wird oder stirbt. Rekursive Abläufe sind besonders geeignet für die Vorstellung des Entstehens und Verschwindens von Akteuren.
In Bild 52 sind die Akteure gezeigt, die am Vorgang der Berechnung von 4! beteiligt sind. Der links außen liegende Akteur ist der auftraggebende Mensch, der den Auftrag an den ersten Fakultätsakteur vergibt. Dieser kann die Aufgabe nicht erledigen, ohne einen weiteren Akteur zu erzeugen, dem er die Aufgabe gibt, 3! zu berechnen. Auch dieser erzeugt wieder einen Akteur usw. Erst der ganz rechts außen liegende Akteur, der 0! berechnen soll, braucht keinen weiteren Helfer zu erzeugen, sondern kann das Ergebnis liefern. Nachdem das jeweilige Ergebnis an den jeweils links liegenden Auftraggeber abgeliefert wurde, kann der ergebnisliefernde Akteur verschwinden, denn nun wird er nicht mehr gebraucht. Man kann die Vorstellung haben, dass er Selbstmord begeht.

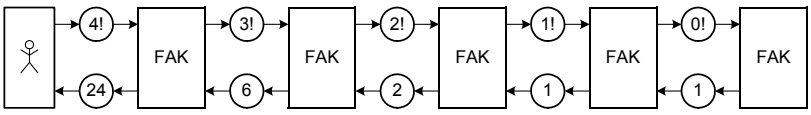


Bild 52 Maximale Akteurskette bei der Berechnung von 4!