2.2.3 Natürliche Semantik zur Interpretation von Ausdrücken

In der obigen Beschreibung der Semantik von Anweisungen ist die Interpretation von Ausdrücken, d.h. die Definition bzw. Berechnung von $I[\![e]\!]\,\sigma$ noch offen. Diese Beschreibung wollen wir nun nachholen. Da bei Ausdrücken die genaue Reihenfolge der Berechnung keine Rolle spielt, ist es nicht notwendig, eine schrittweise operationale Semantik zu definieren. Für Ausdrücke reicht es aus, festzulegen, welcher Wert ein Ausdruck hat. Hierfür kann man kompaktere Definitionen angeben. Zunächst benutzen wir wiederum ein Inferenzsystem, um die Bedeutung von Ausdrücken präzise zu definieren.
Die Struktur der Basissprache dieses Inferenzsystems ist etwas anders als bei Anweisungen, da wir hier nicht an den einzelnen Schritten interessiert sind, sondern nur an dem Wert eines Ausdrucks. Aus diesem Grund verwenden wir eine Form, die auch als „**natürliche Semantik**“ (natural semantics (Kahn, 1987)) bezeichnet wird. Eine Semantikbeschreibung in der Form der natürlichen Semantik basiert auf folgenden Ideen:

- Die Basissprache hat die Form $\sigma \vdash e : v$ („im Zustand σ hat e den Wert v“).
- Somit erfolgt eine direkte Zuordnung des vollständigen Ergebnisses der Berechnung (im Gegensatz zur Semantik von Anweisungen, wo die einzelnen Schritte zu einer Berechnungssequenz kombiniert werden müssen).
- Die Definition erfolgt über den syntaktischen Aufbau von e.

Wir betrachten die folgende Syntax für Ausdrücke. Im Gegensatz zur konkreten Syntax für Ausdrücke, wie dies in Abschnitt 2.1 beispielhaft gezeigt wurde, geben wir hier eine einfachere **abstrakten Syntax** an, bei der Mehrdeutigkeiten erlaubt sind, weil man davon ausgeht, dass hier immer die Ableitungsbäume bezüglich dieser Syntax vorliegen.

```
Exp ::= Number
      | Var
      | Exp Op Exp
```

Hierbei ist `Number` eine Zahl, `Var` eine Variable und `Op` $\in \{$`+`, `*`$\}$.

Die Definition der Semantik der Ausdrücke erfolgt durch ein Inferenzsystem. Die Basissprache dieses Inferenzsystems umfasst dabei alle Sätze der Form

$$\sigma \vdash e : v$$

wobei:

- e ist aus `Exp` ableitbar
- σ ist die textuelle Repräsentation eines Zustandes
- v ist ein Wert (eine Zahl)

Das Inferenzsystem besteht aus folgenden Axiomen und Regeln:

$$\sigma \vdash n : n \qquad \text{wobei } n \text{ eine Zahl ist}$$

$$\sigma \vdash x : \sigma(x) \qquad \text{wobei } x \text{ eine Variable ist}$$

$$\frac{\sigma \vdash e_1 : v_1 \quad \sigma \vdash e_2 : v_2}{\sigma \vdash e_1 \texttt{ + } e_2 : v_1 + v_2}$$

$$\frac{\sigma \vdash e_1 : v_1 \quad \sigma \vdash e_2 : v_2}{\sigma \vdash e_1 \texttt{ * } e_2 : v_1 \cdot v_2}$$

Hierbei ist zu beachten, dass in der Basissprache vor dem „:“ ein Ausdruck und dahinter eine Zahl steht. Dies bedeutet, dass in der Inferenzregel für die Addition bei $\sigma \vdash e_1 \texttt{ + } e_2 : v_1 + v_2$ das erste Plussymbol „+“ ein syntaktisches Symbol ist, während das zweite Plussymbol “+” für die mathematische Funktion steht, die an dieser Stelle ausgerechnet werden muss. Dies gilt ebenso für die Multiplikationsregel, wo zur Verdeutlichung unterschiedliche Symbole gewählt wurden. Zum Beispiel ist

$$\frac{\{\} \vdash \texttt{2 * 3} : 6 \quad \{\} \vdash \texttt{3} : 3}{\{\} \vdash \texttt{2 * 3 + 3} : 9}$$

eine Beispielinstanz der Multiplikationsregel. Wenn wir „6 + 3“ an Stelle der „9“ geschrieben hätten, wäre dies keine korrekte Regelinstanz, da der Ausdruck „6 + 3“ keine Zahl ist!

Der Vorteil dieses Inferenzsystems ist die leichte Erweiterbarkeit auf komplexere Formen von Ausdrücken. Als Beispiel betrachten wir die Erweiterung um lokale Bindungen, z. B. in Form von `let`-Ausdrücken. Wir erweitern hierzu die Syntax um

```
Exp ::= ...
     |  let Var = Exp in Exp
```

wobei `Var` eine Variable ist.
Die Bedeutung von `let`-Ausdrücken kann durch eine weitere Inferenzregel festgelegt werden:

$$\frac{\sigma \vdash e_1 : v_1 \quad \sigma[x/v_1] \vdash e_2 : v_2}{\sigma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : v_2}$$

Diese Regel beschreibt, dass der Wert von x nur lokal gültig in e_2 ist. Zum Beispiel ist

$$\{\texttt{x} \mapsto 6\} \vdash \texttt{let x = 2 + 2 in x * x} : 16$$

ableitbar. Dagegen wäre eine Ableitung mit dem Ergebnis 36 (statt 16) nicht möglich.

Nun können wir die Interpretation von Ausdrücken wie folgt definieren:

$$\text{Es gilt } I[\![e]\!]\,\sigma = v \text{ genau dann wenn } \sigma \vdash e : v \text{ ableitbar ist.}$$

Wie wir gesehen haben, kann man zur Definition einer Programmiersprache auch mehrere Inferenzsysteme für verschiedene Aspekte oder Sprachanteile benutzen. In der Regel ist das jeweilige zu benutzende Inferenzsystem an der Form der abzuleitenden Sätze, d.h. der Basissprache, erkennbar. In unserem Beispiel haben wir das Inferenzsystem mit der Basissprache

$$\sigma \vdash e : v$$

zur Berechnung von Ausdrücken verwendet, während wir das SOS-Inferenzsystem mit der Basissprache

$$\langle S_1, \sigma_1 \rangle \to \langle S_2, \sigma_2 \rangle$$

für imperative Berechnungsschritte benutzt haben.

2.2.4 Implementierung von Inferenzsystemen

Wie oben schon erwähnt wurde, kann man durch die Implementierung einer operationalen Semantik einen Interpreter für die Programmiersprache realisieren. Wenn die operationale Semantik mittels Inferenzsystemen beschrieben wurde, kann man diese sehr leicht mit Programmmiersprachen implementieren, die einen regelorientierten Programmierstil zur Verfügung stellen. Hierzu bietet sich z.B. die Programmiersprache Prolog an, wie wir nachfolgend zeigen wollen.
Prolog-Programme basieren auf Regeln die eine große Ähnlichkeit zu Inferenzregeln aufweisen. Z.B. könnte eine Inferenzregel der Form

$$\frac{P_1 \ldots P_n}{P}$$

in Prolog in der Form

P :- P_1,...,P_n.

umgesetzt werden. Zusätzlich muss man eventuell noch Prädikate zur Berechnung von Werten hinzufügen, die in diesen Inferenzregeln vorkommen.
Als Beispiel zeigen wir die Implementierung der natürlichen Semantik von Ausdrücken in Prolog. Diese Semantik verwendet einen Zustand σ, d.h. eine Abbildung von Variablen in Werte. In Prolog können wir diesen Zustand als Liste von Elementen der Form x=v darstellen, wobei x ein Variablenname (ein Atom) und v der zugeordnete Wert ist. Z.B. könnte ein konkreter Zustand so aussehen:

```
[x=3, b1=true]
```

Zum Zugriff auf Zustände definieren wir ein Prädikat `lookup(S,X,V)`, welches erfüllt ist, wenn im Zustand `S` die Variable `X` den Wert `V` hat:

```
lookup([Y=W|E],X,V) :- X=Y -> V=W ; lookup(E,X,V).
```

Weiterhin definieren wir ein Prädikat `update(S,X,V,S1)`, welches erfüllt ist, wenn sich der Zustand `S1` aus dem Zustand `S` ergibt, indem die Variable `X` den Wert `V` erhält:

```
update([]     ,X,V,[X=V]).
update([Y=W|S],X,V,S1) :- X=Y -> S1=[X=V|S]
                               ; update(S,X,V,S2), S1=[Y=W|S2].
```

Damit haben wir alle Hilfsmittel zur Verfügung, um ein Prädikat `eval(`σ`,`e`,`v`)` zu definieren, welches erfüllt ist, wenn $\sigma \vdash e : v$ aus den Regeln der natürlichen Semantik ableitbar ist. Dies können wir einfach durch Umsetzung der obigen Regeln realisieren:

```
eval(_,N,N) :- number(N).
eval(S,X,V) :- atom(X), lookup(S,X,V).
eval(S,E1+E2,V) :- eval(S,E1,V1), eval(S,E2,V2), V is V1+V2.
eval(S,E1*E2,V) :- eval(S,E1,V1), eval(S,E2,V2), V is V1*V2.
eval(S,E1>E2,V) :- eval(S,E1,V1), eval(S,E2,V2),
                   (V1>V2 -> V=true ; V=false).
```

... weitere Regeln...

In ähnlicher Weise können wir die Regel der strukturierten operationalen Semantik umsetzen. Hierzu definieren wir ein Prädikat `step(`S_1`,`σ_1`,`S_2`,`σ_2`)`, das erfüllt ist, wenn $\langle S_1, \sigma_1 \rangle \to \langle S_2, \sigma_2 \rangle$ ableitbar ist:

```
step((X = E)   ,Sigma,skip,SigmaV) :- eval(Sigma,E,V),
                                      update(Sigma,X,V,SigmaV).
step(if(E,S,_) ,Sigma,S,Sigma)     :- eval(Sigma,E,true).
step(if(E,_,S) ,Sigma,S,Sigma)     :- eval(Sigma,E,false).
step(while(E,_),Sigma,skip,Sigma)  :- eval(Sigma,E,false).
step(while(E,S),Sigma,(S ; while(E,S)),Sigma) :- eval(Sigma,E,true).
step((skip ; S),Sigma,S,Sigma).
step((S1 ; S)  ,Sigma,(S2 ; S),Sigma1) :- step(S1,Sigma,S2,Sigma1).
```

Wie man sieht, folgt die Umsetzung exakt den SOS-Regeln. Schließlich können wir nun einen *Interpreter für imperative Programme* implementieren, indem wir einfach Berechnungsschritte so lange durchführen, bis wir in einen Endzustand kommen:

```
exec(skip,Sigma,Sigma) :- !.   % final state reached
exec(S1,Sigma1,R) :- step(S1,Sigma1,S2,Sigma2), exec(S2,Sigma2,R).
```

Damit können wir z.B. eine einfache Schleife zur Berechnung der Fakultät einer Zahl `n`, deren Wert sich in dem Zustand befindet, abarbeiten:

```
?- exec((p=1; while(n>0, (p=p*n ; n=n-1))), [n=6], R).
R = [n=0, p=720]
```

2.2.5 Denotationelle Semantik zur Interpretation von Ausdrücken

Wie schon erwähnt, gibt es verschiedene Methoden, um die Bedeutung von Sprachkonstrukten festzulegen. Als kurze Demonstration zeigen wir in diesem Abschnitt, wie man die Bedeutung von Ausdrücken, d.h. $I[\![e]\!]\,\sigma$, auch durch eine denotationelle Semantik beschreiben kann.
Wie oben erwähnt basiert die **denotationelle Semantik** auf dem Prinzip, jedem syntaktischen Konstrukt eine Bedeutung, d.h. ein semantisches Objekt, zuzuordnen. Diese klare Trennung von Syntax und Semantik ist bei der operationalen Semantik nicht gegeben. Z.B. hatten wir die Bedeutung einer `while`-Schleife dadurch definiert, dass wir in dem Fall, dass die Bedingung erfüllt ist, nach Abarbeitung des Rumpfes die Schleife wieder selbst verwenden:

$$\langle \text{while } b \text{ do } S, \sigma\rangle \to \langle S; \text{while } b \text{ do } S, \sigma\rangle \qquad \text{falls } I[\![b]\!]\,\sigma = true$$

Somit wird bei der Abarbeitung die Syntax selbst manipuliert. Solche Syntaxmanipulation sind bei der denotationellen Semantik nicht erlaubt, da hier jedem Syntaxkonstrukt ein Semantikobjekt zugeordnet wird. Die Zuordnung ist dabei über den Aufbau der Syntax definiert, d.h. wenn ein komplexeres syntaktische Konstrukt S mehrere einfachen Konstrukte $S_1, \ldots, S_n$ enthält, dann kann man bei der Definition der Semantik von S die Semantik von $S_1, \ldots, S_n$ verwenden (aber nicht deren Syntax!). Diese Idee können wir sehr einfach am Beispiel von Ausdrücken zeigen.

Die Syntax von Ausdrücken hatten wir vorher schon beschrieben:

```
Exp ::= Number
      | Var
      | Exp Op Exp
```

Wie wir bei der natürlichen Semantik von Ausdrücken gesehen hatten, ist die Bedeutung eines Ausdrucks ein Wert bzgl. eines Zustandes. Somit kann die *Semantik eines Ausdrucks* als Funktion von Zuständen in Werte definiert werden. Wenn also *Value* die Menge aller Werte

$$Int \cup Bool \cup \ldots$$

und *State* die Menge aller Zustände, d.h. Abbildungen

$$\sigma : Var \to Value$$

bezeichnet, dann ordnet eine denotationelle Semantik jedem Ausdruck e (Syntax!) eine Funktion

$$State \to Value$$

zu. Somit hat die Semantikfunktion I den Typ

$$I : \texttt{Exp} \to (State \to Value)$$

Wie bei der strukturierten operationalen und der natürlichen Semantik definiert man eine denotationelle Semantik über die syntaktische Struktur. Um die Trennung von Syntax

und Semantik deutlich zu machen, wird bei der Semantikfunktion (hier: I) das Syntaxargument in eckige Klammern gesetzt (so haben wir dies oben auch schon gemacht). Dies führt dann zu folgender Definition der Semantikfunktion I:

$$\begin{aligned} I[\![n]\!]\,\sigma &= n && (n \text{ Zahl}) \\ I[\![x]\!]\,\sigma &= \sigma(x) && (x \text{ Variable}) \\ I[\![e_1 \texttt{ + } e_2]\!]\,\sigma &= I[\![e_1]\!]\,\sigma + I[\![e_2]\!]\,\sigma \\ I[\![e_1 \texttt{ * } e_2]\!]\,\sigma &= I[\![e_1]\!]\,\sigma \cdot I[\![e_2]\!]\,\sigma \end{aligned}$$

Ebenso wie bei der natürlichen Semantik kann man auch die denotationelle Semantik einfach um neue Sprachkonstrukte wie `let`-Ausdrücke erweitern ($\rightsquigarrow$ Übung).

Da die denotationelle Semantik nur mit Funktionen arbeitet, können wir sie z.B. direkt in einer funktionalen Sprache wie Haskell implementieren. Wenn wir Ausdrücke zunächst auf ganze Zahlen, Addition und Multiplikation einschränken, können wir Ausdrücke durch folgenden Datentyp in Haskell beschreiben:

```
data Exp = Num Int
         | Var String
         | Add Exp Exp
         | Mul Exp Exp
```

Z.B. würde in dieser Darstellung der Ausdruck 3+4∗5 folgendem Datenterm entsprechen:

```
Add (Num 3) (Mul (Num 4) (Num 5))
```

Um eine Programmiersprache zu implementieren, müsste man noch einen Parser von der konkreten Syntax in die „abstrakte Syntax“ (Datentyp `Exp`) implementieren. Dies ist eine klassische Aufgabe des Compilerbaus (und die Vorlesung „Übersetzerbau“ zeigt konkrete Methoden dafür).

Mit diesem Datentyp modellieren wir Variablen als Strings, sodass ein *Zustand* einfach eine Funktion von Strings nach ganzen Zahlen ist:

```
type State = String → Int
```

Somit kann die denotationelle Semantik von Ausdrücken durch folgende Funktionsdefinition implementiert werden:

```
semE :: Exp → State → Int
semE (Num n)     s = n
semE (Var x)     s = s x
semE (Add e1 e2) s = semE e1 s + semE e2 s
semE (Mul e1 e2) s = semE e1 s * semE e2 s
```

Hier zeigt sich also der Vorteil einer Semantikbeschreibung für eine Programmiersprache: Wir haben damit nicht nur die Bedeutung der Programme präzise festgelegt, sondern

gleichzeitig auch schon eine erste prototypische Implementierung zur Verfügung! Z.B. können wir den Wert des obigen Ausdrucks mit diesem Haskell-Programm wie folgt ausrechnen:

```
> semE (Add (Num 3) (Mul (Num 4) (Num 5))) (\_ → 0)
23
```

Soweit sieht die Verwendung der denotationellen Semantik recht einfach aus. Eine Schwierigkeit ergibt sich aber, wenn man Wiederholungen hat, wie z.B. eine `while`-Schleife. Eine `while`-Schleife wird ja abgearbeitet, in dem man (bei positiver Bedingung) den Rumpf und dann wieder die komplette `while`-Schleife abarbeitet. Wie können wir also direkt einer Schleife eine Semantik zuordnen, ohne deren gesamte Syntax zu verwenden?
Letztendlich müssen wir also ein Konstrukt durch sich selbst definieren. In der Informatik kennen wir dies als Rekursion. Wenn man dies aber mathematisch fundiert definieren will, stellt sich die Frage, ob durch rekursive Definitionen überhaupt etwas Vernünftiges definiert wird. Genau damit beschäftigen sich die Grundlagen der denotationellen Semantik. Das Rekursionsproblem wird dadurch gelöst, dass die semantischen Bereiche keine einfachen Mengen sind, sondern bestimmte Strukturen (Verbände), auf denen Fixpunktberechnungen möglich sind, d.h. z.B. kleinste Fixpunkte für Funktionen existieren. Damit kann dann die Bedeutung einer (rekursiven) semantischen Funktion als deren kleinster Fixpunkt definiert werden. Dies verlangt allerdings einiges an mathematischer Theorie, was wir aber in dieser praktisch orientierten Vorlesung nicht betrachten wollen. Um dennoch zu skizzieren, wie man denotationelle Semantik verwenden kann, um Programmiersprachen zu beschreiben und zu implementieren, nehmen wir im folgenden einfach an, dass wir rekursive Funktionen (wie in Programmiersprachen) verwenden können und diese eine wohldefinierte Bedeutung haben.

Betrachten wir also die in Kapitel 2.2.1 eingeführte imperative Sprache:

```
Stm ::= skip
      | Stm ; Stm
      | Var := Exp
      | if Exp then Stm else Stm
      | while Exp do Stm
```

Um jeder Anweisung eine Bedeutung zuzuordnen, müssen wir uns überlegen, was eigentlich die Bedeutung einer Anweisung ist, d.h. was diese eigentlich „macht“. Wie schon erwähnt, manipulieren Anweisungen Zustände, indem eine Zuweisung den Wert einer Variablen ändert. Daher ist die Bedeutung einer Anweisung eine *Zustandsänderung*, d.h. eine Funktion

$$State \to State$$

Damit hat die Semantikfunktion S für Anweisungen den Typ

$$S : \texttt{Stm} \to (State \to State)$$

und kann wie folgt definiert werden (um die Einführung eines Datentyps für Wahrheitswerte zu vermeiden, interpretieren wir 0 als „falsch“ und andere Zahlen als „wahr“):

$$
\begin{aligned}
S[\![\texttt{skip}]\!]\,\sigma &= \sigma \\
S[\![s_1\ \texttt{;}\ s_2]\!]\,\sigma &= S[\![s_2]\!]\,(S[\![s_1]\!]\,\sigma) \\
S[\![v\ \texttt{:=}\ e]\!]\,\sigma &= \sigma[v/(I[\![e]\!]\,\sigma)] \\
S[\![\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]\,\sigma &= \begin{cases} S[\![s_1]\!]\,\sigma & \text{falls } I[\![e]\!]\,\sigma \neq 0 \\ S[\![s_2]\!]\,\sigma & \text{sonst} \end{cases} \\
S[\![\texttt{while}\ e\ \texttt{do}\ s]\!]\,\sigma &= \mathit{loop}(I[\![e]\!], S[\![s]\!], \sigma)
\end{aligned}
$$

Hierbei ist *loop* eine rekursive Funktion, die die Semantik eines Ausdrucks, einer Anweisung und einen Zustand als Parameter hat und einen neuen Zustand wie folgt berechnet:

$$
\mathit{loop}(E, S, \sigma) = \begin{cases} \sigma & \text{falls } E(\sigma) = 0 \\ \mathit{loop}(E, S, S(\sigma)) & \text{sonst} \end{cases}
$$

Unter der Voraussetzung, dass alle beteiligten Mengen (wie z.B. *Value*, *State*) Verbandsstrukturen sind, d.h. geordnet und kleinste Elemente („undefiniert“) enthalten, dann hat *loop* einen kleinsten Fixpunkt, was dann als Semantik von *loop* interpretiert wird.

Wie man an der Definition von S sehen kann, ist dies eine denotationelle Semantik: jedem syntaktischen Konstrukt wird eine Bedeutung zugeordnet, indem die Bedeutungen der Teilkonstrukte geeignet kombiniert werden. Außerdem können wir diese Semantik in jeder Programmiersprache, die rekursive Funktionen unterstützt (also eigentlich alle modernen Sprachen), implementieren. Als Beispiel zeigen wir eine Implementierung in Haskell, die auf der obigen Implementierung der denotationellen Semantik von Ausdrücken aufbaut. Zunächst definieren wir einen Datentyp für Anweisungen:

```
data Stm = Skip
         | Seq    Stm Stm
         | Assign String Exp
         | If     Exp Stm Stm
         | While  Exp Stm
```

Zur Definition der Zuweisung definieren wir eine Hilfsfunktion zur Abänderung einer Funktion an einer Stelle, d.h. `updF` $f\ x\ v$ entspricht $f[x/v]$:

```
updF f x v = \z → if z==x then v
                          else f z
```

Damit können wir die obige Definition der denotationellen Semantik für Anweisungen direkt in Haskell implementieren:

```
semS :: Stm → State → State
semS Skip           s = s
```

```
semS (Seq st1 st2)  s = semS st2 (semS st1 s)
semS (Assign v e)   s = updF s v (semE e s)
semS (If e st1 st2) s = if semE e s /= 0 then semS st1 s else semS st2 s
semS (While e st)   s = loop (semE e) (semS st) s
 where
  loop seme sems s = if seme s == 0 then s
                                    else loop seme sems (sems s)
```

Als Beispiel betrachten wir ein einfaches Programm zur Berechnung der Fakultät von 6, das in der konkreten Syntax z.B. so aussieht:

```
x := 6 ; f := 1 ; while (x) { f := x * f ; x := x - 1 }
```

In der abstrakten Syntax kann dies als folgender Datenterm dargestellt werden:

```
facprog = Assign "x" (Num 6) `Seq`
          (Assign "f" (Num 1)) `Seq`
           (While (Var "x")
              (Assign "f" (Mul (Var "x") (Var "f")) `Seq`
               Assign "x" (Add (Var "x") (Num (-1)))))
```

Dieses Programm können wir dann durch folgenden Aufruf ablaufen lassen:

```
> semS facprog (\_ → 0) "f"
720
```

Die denotationelle Semantik liefert also eine Möglichkeit, Programmiersprachen zu implementieren, was in manchen Compilergeneratoren verwendet wird: man gibt eine EBNF-Grammatik für die konkrete Syntax an, woraus automatisch ein Parser generiert wird. Wenn man dann noch für jedes abstrakte Syntaxkonstrukt eine Regel zur Zuordnung eines semantischen Konstrukts angibt, welches z.B. in einer funktionalen Programmiersprache implementiert ist, hat man ohne viel Aufwand eine Sprachimplementierung zur Verfügung. Dies kann man praktisch z.B. zur Implementierung von DSLs (domänenspezifische Sprachen) benutzen.

Wir können also festhalten, dass eine Semantikbeschreibung nicht nur dazu dient, eine Programmiersprache präzise zu beschreiben, sondern sie kann auch eine Basis für die Implementierung einer Programmiersprache sein. Hierbei führt die Implementierung einer operationale Semantik zu einem Interpreter, während die Implementierung einer denotationellen Semantik eher einem Übersetzer entspricht, da nach der Zuordnung der Semantik die Syntax nicht mehr betrachtet wird. Auf dieser Idee aufbauend gibt es durchaus komplexe Systeme, mit denen Programmiersprachen implementiert werden können (z.B. "K framework"[1]).

[1] `http://www.kframework.org/`