3.4 Software-Praxis „The Adaptive Control of Thought-Rational"

Wie wir bereits in ► Abschn. 3.3.2 des Buches erfahren haben, handelt es sich bei ACT-R um eine kognitive Architektur. John Anderson definiert eine kognitive Architektur wie folgt:

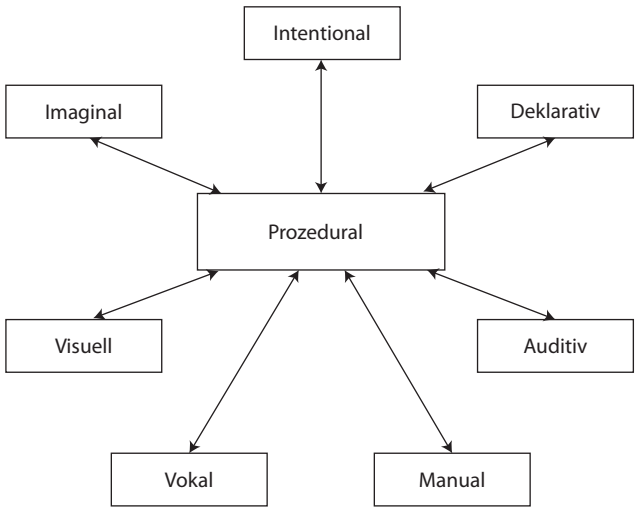
» A cognitive architecture is a specification of the structure of the brain at a level of abstraction that explains how it achieves the function of the mind. (Anderson, 2007, S. 7)

Das Besondere an dieser kognitiven Architektur ist, dass sie vollständig in eine Software implementiert ist. Spezifische kognitive Prozesse können in dieser Software modelliert werden und sogar zur Prädiktion von Parametern der menschlichen Kognition genutzt werden. Innerhalb der Forschung zur künstlichen Intelligenz ordnet Anderson diese den Produktionssystemen mit spezifischen Erweiterungen zu (Anderson, 1976, 2007). Aufgrund einer konsequenten Verbindung von physiologisch-psychologischer Forschung *(brain)* und traditioneller Kognitionsforschung *(mind)* ist ACT-R wohl als Prototyp einer artifiziellen Intelligenz zu betrachten. Die Software besteht aus unterschiedlichen Modulen (◘ Abb. 3.9), deren wesentlichsten Module bestimmten Hirnregionen zugeordnet werden können.

Es bietet sich an, in der Forschung zur künstlichen Intelligenz bei Systemen, die nicht den Anspruch der Modellierung menschlicher Intelligenz und ihrer Physiologie verfolgen, eher von *computationaler Intelligenz* (Kramer, 2009) zu sprechen, anstelle von *artifizieller Intelligenz*. In speziellen Bereichen sind Systeme computationaler Intelligenz der menschlichen Intelligenz bereits voraus. Die Modellierung der artifiziellen Intelligenz ist vielmehr an der Simulation der bestehenden Grenzen interessiert, anstatt diese zu übersteigen:

» However, suppose our goal is not to find the simplest model but rather the „true" model of the structures and processes in the human brain. How much credibility can we give to the claim that nature picked the simplest form? (Anderson, 1976, S. 5)

Wir wollen uns nun der Software ACT-R zuwenden. Die folgenden Seiten sollen zum Einstieg in die Software dienen und basieren hauptsächlich auf dem Tutorial zu ACT-R 6.1 und dessen Erweiterungen von Daniel Bothell und John R. Anderson im Jahr 2014.



◘ **Abb. 3.9** Es werden die wichtigsten Module von ACT-R (Version 6.0) und ihre Beziehungen untereinander aufgezeigt. Das prozedurale Modul ist die zentrale Instanz (Anderson, 2010, by permission of Oxford University Press, USA, www.oup.com)

3.4.1 Download und Installation

Es gibt unterschiedliche Möglichkeiten, ACT-R auf einem Computer zu installieren. Die Software ACT-R basiert auf der Programmiersprache LISP (List Processing). Wir verwenden aus pragmatischen Gründen die *Standalone-Version* von ACT-R, da in dieser Variante eine LISP-Version integriert ist und wir keine zusätzlichen Applikationen installieren müssen. Man findet die Software für Windows und Mac OS X 10.5 (oder höher) direkt auf der Website der ACT-R-Arbeitsgruppe an der Carnegie Mellon University (http://act-r.psy.cmu.edu/software/) oder auch auf der Begleitseite zu diesem Buch (www.lehrbuch-psychologie.de). Die komplette Software und das Tutorial finden sich in einem komprimierten Ordner, der an einem beliebigen Ort des Computers zu entpacken ist. Die Software wird über das Skript „Run ACT-R.command" aufgerufen. Wir können dies schon einmal vornehmen, aber auch noch mit dem Öffnen warten, denn zunächst wollen wir uns um den Kern von ACT-R kümmern, den Code eines ersten eigenen Modells. Wir haben nun zwei Möglichkeiten, dieses Kapitel durchzugehen: je nach Lerntyp hat man die freie Wahl, den Code von der Begleitwebsite zu diesem Buch herunterzuladen und in seiner Gänze während des Lesens zu betrachten oder man programmiert einfach direkt Zeile für Zeile mit. Letzteres ist natürlich spannender und lernwirksamer, daher empfohlen.

3.4.2 Ein erstes eigenes ACT-R-Modell programmieren

Programmieren Den Code eines ACT-R-Modells können wir zunächst in einem beliebigen Editor-Programm erstellen, z. B. im *TextEdit* des Mac-OS-Betriebssystems oder in *Notepad++*, das man kostenlos aus dem Internet herunterladen kann. Selbstverständlich gäbe es komplexere oder komfortablere Entwicklungsumgebungen, aber wir begnügen uns in dieser Einführung mit einem einfachen Editor. Eine Eigenschaft von LISP ist es, dass sich LISP-Funktionen immer in Klammern befinden müssen. Dies wird später im Code von ACT-R sehr auffällig sein. Wir öffnen daher eines unserer Editor-Programme aus dem Betriebssystem.

Modell Wir wollen in unserem ersten kleinen Modell den kognitiven Ablauf des Zählens von einer bestimmten Zahl zu einer höheren Zahl simulieren. Dies ist ein kognitiver Vorgang, der bei genügend Vorwissen hauptsächlich zwischen deklarativem und prozeduralem Wissen verläuft. Unser Modell wird daher nur die wichtigsten der ACT-R-Module belegen und z. B. nicht auf die visuelle Wahrnehmung oder die Motorik zurückgreifen; all dies wäre allerdings in ACT-R möglich. Die wesentlichen Komponenten eines ACT-R-Modells sind

- die Wissensrepräsentationen des deklarativen Wissens, das, was wir bewusst wissen, in Form von Chunks *(chunks)* und
- die Repräsentationen des prozeduralen Wissens, gewissermaßen unsere kognitiven Reflexe, in Form von Produktionsregeln *(productions)*.

Chunks

Chunk-Typen Ein Chunk ist definiert durch seine Typenbezeichnung *(chunk-type)*, die sehr verwandt sind mit Kategorien, und durch seine Fähigkeit, eine Anzahl an Attributen zu tragen *(slots)*, wie etwa die Attribute Farbe oder Größe. Zudem werden Chunks in ACT über eigene Namen deklariert. Bevor man also einen Chunk anlegt, muss man zunächst Chunk-Typen erstellen, die einem Chunk Attribute zuweisen und eigene Namen tragen.

#1 Für unser Zähl-Modell legen wir die erste Zeile Code an, in der wir einen benötigen Chunk-Typen spezifizieren:

```
(chunk-type zaehl-anordnung erstens zweitens)
```

Der Chunk-Typ trägt die Bezeichnung `zaehl-anordnung` und wird für Chunks gebraucht, die eine Ordnung von Zahlen beinhaltet, dass also z. B. die Zahl 4 auf die Zahl 3 folgt.

#2 Wir legen einen zweiten Chunk-Typen an:

```
(chunk-type zaehle-von start ende zahl)
```

Dieser Chunk enthält drei Slots, um die Startzahl, die Endzahl und die aktuelle Zahl zu enkodieren.

Chunks Einen Chunk legt man an, indem man ihm 1.) einen Namen gibt, 2.) den Chunk-Typen bestimmt und 3) den Attributen, also den *slots*, entsprechende Werte zuweist, indem man sich am Raster des Chunk-Typen orientiert.

#3 Ein erster Chunk wird wie folgt angelegt:

```
(b ISA zaehl-anordnung erstens 1 zweitens 2)
```

Das *b* steht für den Namen dieses Chunks, der beliebig gewählt werden kann. Wenn man keinen speziellen Namen für einen Chunk vergibt, wird dies vom System später automatisch vorgenommen. Wir sehen, dass die slot-Namen *erstens* und *zweitens* angegeben werden, um ihnen spezifische Werte zuzuordnen, in diesem Fall die Werte *1* und *2*. Werden keine Werte eingetragen, so weist ACT-R den freien slots den Wert *nil* zu. Der Ausdruck *ISA* ist die wohl wesentlichste Komponente eines Chunks. Er füllt ebenfalls einen *slot* aus und weist jedem Chunk seinen Typen zu. Während die slots zu den Attributen später geändert werden können, ist die Zuordnung des Typen zu keinem späteren Zeitpunkt mehr variabel. Wir wollen unserem Modell aber noch ein paar weitere Fakten bzw. deklaratives Wissen über das Zählen hinzufügen.

#4 Weitere Chunks vergrößern das deklarative Wissen über die Zahlenfolge:

```
(c ISA count-order erstens 2 zweitens 3)
(d ISA count-order erstens 3 zweitens 4)
(e ISA count-order erstens 4 zweitens 5)
(f ISA count-order erstens 5 zweitens 6)
(g ISA count-order erstens 7 zweitens 8)
(h ISA count-order erstens 8 zweitens 9)
(i ISA count-order erstens 9 zweitens 10)
```

Unsere ersten 8 Chunks bilden im Modell das deklarative Wissen über die Ordination der Zahlen 1–10. Dieses Wissen muss nun noch über einen Befehl dem deklarativen Gedächtnis zugefügt werden.

Deklaratives Gedächtnis Das deklarative Gedächtnis ist in ACT-R ein eigenes Modul, so wie es physiologisch hauptsächlich dem Temporallappen und dem Hippokampus zugeordnet werden kann (Anderson et. al., 2004; Pritzel et. al. 2003; Kandel, 2009).

#5 Mit dem Befehlt add-dm fügt man dem deklarativen Gedächtnis Chunks zu:

```
(add-dm
(b ISA zaehl-anordnung erstens 1 zweitens 2)
(c ISA zaehl-anordnung erstens 2 zweitens 3)
(d ISA zaehl-anordnung erstens 3 zweitens 4)
(e ISA zaehl-anordnung erstens 4 zweitens 5)
(f ISA zaehl-anordnung erstens 5 zweitens 6)
(g ISA zaehl-anordnung erstens 7 zweitens 8)
(h ISA zaehl-anordnung erstens 8 zweitens 9)
(i ISA zaehl-anordnung erstens 9 zweitens 10))
```

Wir erkennen, dass das Kommando *add-dm* den Chunks vorangestellt ist und sich diese in den Klammern des Befehls befinden. Jeder Chunk ist ein Argument des Befehls und es können beliebig viele Chunks dem deklarativen Gedächtnis hinzugefügt werden.

Intention Wir haben bereits in ► Abschn. 3.3.2 erfahren, dass eine zielgesteuerte Informationsverarbeitung in ACT-R als primär angesehen wird. Diese Informationsverarbeitung findet zwar in einem eigenen *Modul zur Intention* statt, allerdings ist bewusstes Wissen um ein Ziel immer deklarierbar und befindet sich somit auch im deklarativen Gedächtnis. Im Falle des ersten Modells fügen wir ein Ziel des Zählens hinzu: es soll von 2–7 gezählt werden.

#5 Dem deklarativen Gedächtnis wird ein Chunk mit deklarativem Wissen zum Ziel hinzugefügt:

```
(add-dm
(b ISA zaehl-anordnung erstens 1 zweitens 2)
(c ISA zaehl-anordnung erstens 2 zweitens 3)
(d ISA zaehl-anordnung erstens 3 zweitens 4)
(e ISA zaehl-anordnung erstens 4 zweitens 5)
(f ISA zaehl-anordnung erstens 5 zweitens 6)
(g ISA zaehl-anordnung erstens 7 zweitens 8)
(h ISA zaehl-anordnung erstens 8 zweitens 9)
(i ISA zaehl-anordnung erstens 9 zweitens 10)
(erstes-ziel ISA zaehle-von start 2 ende 7))
```

Der Chunk mit Informationen zum Ziel hat den Namen *erstes-ziel* erhalten und es wird deklariert, dass er zum Chunk-Typen *zaehle-von* gehört. Als Attribute werden ihm eine Startwert von *2* und ein Endwert von *7* zugeordnet. Wir können erkennen, dass das dritte Attribut *zahl* dieses Chunk-Typen nicht spezifiziert wird. Dieser *slot* bleibt also leer und wird vorerst den Wert *nil* vom System erhalten. Für unser erstes kleines Modell reicht das hier eingefügte deklarative Wissen vorerst aus.

Wir wenden uns nun der zweiten großen Form der Wissensrepräsentation zu, dem prozeduralen Wissen.

Produktionen

Prozedurales Wissen, das Handlungswissen, besteht in ACT-R aus sog. Produktionen *(productions)*. Das Modul wird physiologisch hauptsächlich mit den Basalganglien und dem Thalamus in Verbindung gebracht (Anderson et. al. 2004, 2007). Eine Produktion kann als Kontrolle des Verhaltens verstanden werden, sie führt also zu einer Handlung. Hierzu besteht eine Produktion aus einem Bedingungsteil und einer bestimmten Aktion. Sie folgt immer einer Wenn-dann-Regel. Der Bedingungsteil kann aus unterschiedlichen Bedingungen bestehen, die in ihrer Gesamtheit einen logischen Wahrheitswert *(true)* erhalten müssen, um im ann-Teil eine Aktion hervorzurufen. Eine solche Aktion ist ein konkreter Operator. In der Sprache von ACT-R wird der Bedingungsteil auch als *left-hand side (LHS)* bezeichnet und der Aktions-Teil als *right-hand side (RHS)*. Das Modul des prozeduralen Wissens kann als Herzstück der ACT-R-Architektur betrachtet werden. Über dieses Modul stehen alle anderen Module miteinander in Verbindung (◘ Abb. 3.9). Eine wichtige Struktur ist allerdings zwischen die weiteren Module und das prozedurale Modul geschaltet, die sog. *Buffer*. Eine Produktion agiert immer mit einem Buffer, sowohl auf der LHS als auch auf der RHS. Schauen wir uns hierzu einfach einmal eine erste Produktion an.

#6 Die erste Produktion wird dem Modell hinzugefügt:

```
(p anfang
   =goal>
      ISA         zaehle-von
      start       =num1
      zahl        nil
==>
   =goal>
      ISA         zaehle-von
      zahl        =num1
   +retrieval>
      ISA         zaehl-anordnung
      erstens     =num1
)
```

Betrachten wir in Ruhe unsere erste Produktion: Sie befindet sich wieder in eigenen Klammern und beginnt mit einem einfachen Befehl, dem *p*. Diesem Befehl für das Anlegen einer Produktion folgt der Name der Produktion, in unserem Fall der Name *anfang*. Wir erinnern uns, dass die Produktion eine Wenn-dann-Regel bildet. Der Bedingungsteil „wenn" ist durch eine eindeutige Markierung vom Dann-Teil getrennt. Diese Trennung finden wir in der Notation ==>, die wir

als das „dann" lesen dürfen. Wir können also *LHS* und *RHS* getrennt betrachten. Das, was wir in beiden Teilen finden, sind die Zugriffe auf die Buffer.

Buffer

Wir haben bereits die wichtigsten Informationen zu den Buffern erhalten. Diese liegen als Schnittstellen zwischen dem prozeduralen Gedächtnis und den weiteren Modulen. So wird der Buffer zum deklarativen Gedächtnis als *retrieval-Buffer* bezeichnet oder zum intentional Modul als *goal-Buffer*. Jeder Buffer kann immer nur einen Chunk zu einem Zeitpunkt enthalten. Die Aktionen der RHS wirken auf die Chunks in den Buffern ein, indem Chunks kreiert, gespeichert oder modifiziert werden. Dies werden wir im Unterkapitel zur RHS vertiefen. Zuvor schauen wir uns nun den Bedingungsteil an.

LHS

Unser Wenn-Teil greift auf den goal-Buffer zu, dies erkennen wir an der Notation =goal>. Weitere Zugriffe auf andere Buffer finden wir in LHS nicht; es wird also nur ein Buffer auf bestimmte Bedingungen hin getestet:

```
=goal>
       ISA          zaehle-von
       start        =num1
       zahl         nil
```

Es wird abgefragt, ob im goal-Buffer ein *zaehle-von*-Chunk vorliegt und ob in dessen *slot zahl* kein Wert *(nil)* vorhanden ist.

Variablen Die Notation *=num1* bindet den Wert, der sich im *slot start* des *zaehle-von*-Chunks befindet an die Variable *num1* innerhalb der Produktion. Solch eine angelegte Variable ist nur innerhalb einer spezifischen Produktion gültig. Die Variable *num1* wirkt also nicht über die Produktion *anfang* hinaus. Wenn alle Bedingungen der LHS getestet wurden und erfüllt sind, dann wird der Aktions-Teil der Produktion aktiviert. Man sagt auch, dass die Produktion feuert.

RHS

Der Aktionsteil bezieht sich auf zwei Buffer, ebenfalls auf den goal-Buffer und zusätzlich auf den *retrieval-Buffer*:

```
=goal>
   ISA          zaehle-von
   zahl         =num1
+retrieval>
   ISA          zaehl-anordnung
   erstens      =num1
```

Wir erkennen beim Zugriff auf diese Buffer unterschiedliche Charaktere, die den Buffernamen voranstehen. Diese stehen jeweils für einen bestimmten Typen von Aktion. Insgesamt können drei unterschiedlichen Aktionen mit einem Buffer auf der RHS durchgeführt werden:

1. eine Modifikation,
2. ein Abruf und
3. ein Clearing.

Modifikation Der goal-Buffer wird mit *=goal>* angesprochen, was im Aktionsteil eine Modifikation *(=)* bedeutet. Im goal-Buffer soll der *zahl slot* des *zaehle-von*-Chunks verändert werden auf den Wert der Variablen *num1*.

Abruf Der retrieval-Buffer wird mit *+retrieval>* angesprochen, was einen Abruf eines Chunks, in diesem Fall einen *zaehl-anordnung*-Chunk, auslösen soll. Dieser Chunk soll den Wert der Variablen *num1* im slot *erstens* enthalten. Ein Abruf löst meistens den Austausch eines Chunks hervor, kann aber auch wie bei der Modifikation zu einer Veränderung einzelner slots eines schon vorhandenen Chunks führen. In Bezug auf den retrieval-Buffer ist es immer eine Anfrage an das deklarative Gedächtnis, den passenden Chunk in den retrieval-Buffer abzurufen, also einen Austausch vorzunehmen.

Clearing Die dritte Form einer Aktion auf der RHS ist das *Clearing*. Wir finden diesen Operator der Bereinigung zwar noch nicht in der aktuellen Produktion, er sei aber schon der Vollständigkeit halber genannt. In der nächsten Produktion wird er uns begegnen. Durch das Voranstellen des Minuszeichens *(–)* an den Buffernamen, wird der Chunk aus dem Buffer entfernt. Wir erinnern uns, dass zu einem spezifischen Zeitpunkt immer nur ein Chunk in einem Buffer vorhanden sein kann. Es ist daher ausreichend, den Buffer lediglich mit dem Minuszeichen anzusprechen, also z. B. *–retrieval>* um den retrieval-Buffer zu räumen.

Sind alle Bedingungen der LHS erfüllt, so werden die Aktionen der RHS durchgeführt. Es ist wichtig anzumerken, dass durchaus Situationen entstehen können, in denen der Wenn-Teil mehrerer Produktionen gleichzeitig erfüllt sein kann. Dennoch feuert immer nur eine Produktion. Hierzu gibt es in ACT-R Funktionen und Spezifikationen, die in späteren Teilen des Original-Tutorials erläutert werden. Wir werden es an dieser Stelle nicht weiter vertiefen. Ein fester Parameter ist die Dauer, die belegt ist, vom Zeitpunkt der Wahl einer Produktion bis diese feuert, dies sind bei jeder Produktion 50 Millisekunden.

Vervollständigung des Modells

Betrachten wir die LHS unserer ersten Produktion, so kann ihr Bedingungsteil in unserem bisherigen Code nicht erfüllt werden.

Goal-Buffer

Wir haben zwar bislang einen Chunk zum ersten Ziel im deklarativen Gedächtnis abgelegt, dieser befindet sich aber noch nicht im eigentlichen Modul zur Intention bzw. in dessen Buffer, dem goal-Buffer.

#7 Damit eine Intention im System vorliegt, legen wir den Chunk *erstes-Ziel* im goal-Buffer ab:

```
(goal-focus erstes-ziel)
```

Der Befehl *goal-focus* legt einen Chunk im goal-Buffer ab. Diese Codezeile ist kein Teil einer Produktion, sondern verursacht eine Intention im Modell. Unsere erste Produktion fragt ab, ob ein solches Ziel im goal-Buffer vorliegt. Erst jetzt wäre der Bedingungsteil der Produktion *start* erfüllt und die Produktion würden mit dem Feuern beginnen.

Produktion Inkrement

Unser Anliegen ist das Zählen von einer Startzahl zu einer Endzahl, wir hatten dazu die Zwei und die Sieben gewählt. Unter Zählen verstehen wir ein Inkrement oder ein Dekrement, eine schrittweise Erhöhung oder Erniedrigung von Zahlen einer bestimmten Ordnung. Diese Ordnung liegt im deklarativen Wissen vor. Im prozeduralen Wissen finden wir allerdings noch keine Produktion, die dies aktiv vornimmt.

#8 Wir fügen unserem Modell eine Produktion zur Operation Inkrement hinzu:

```
   (P inkrement
      =goal>
         ISA          zaehle-von
         zahl        =num1
       - ende        =num1
      =retrieval>
         ISA          zaehl-anordnung
         erstens        =num1
         zweitens       =num2
    ==>
      =goal>
         ISA          zaehle-von
         zahl         =num2
      +retrieval>
         ISA          zaehl-anordnung
         erstens        =num2
      !output!         (=num1)
)
```

Zunächst verschaffen wir uns wieder einen globalen Überblick über die Produktion und erkennen, dass sowohl die LHS als auch die RHS auf den goal-Buffer und den retrieval-Buffer zugreifen.

LHS Zunächst wird in der LHS geprüft, ob das Ziel bzw. der Chunk zum Ziel *zaehle-von* vorliegt:

```
    =goal>
       ISA         zaehle-von
       zahl       =num1
 - ende       =num1
```

Im nächsten Schritt taucht ein spezieller Vergleich auf: Es wird getestet, ob die Variable im slot *zahl* nicht der Variablen im slot *ende* entspricht. Das Nicht verbirgt sich im Minuszeichen vor dem Namen des slots. Ein weiterer Test bezieht sich auf den Buffer zum deklarativen Gedächtnis und prüft, ob im retrieval-Buffer der Chunk *zaehl-anordnung* vorliegt:

```
    =retrieval>
       ISA         zaehl-anordnung
       erstens       =num1
zweitens      =num2
```

Es wird getestet, ob der slot *erstens* der Variablen *num1* entspricht und somit auch dem slot *zahl* im goal-Buffer. Der Wert des slots *zweitens* wird an die Variable *num2* gebunden. Wir können also schon jetzt davon ausgehen, dass dieser Wert in der RHS eine Rolle spielen wird.

RHS In der RHS finden wir zunächst eine Modifikation des goal-Buffers:

```
  =goal>
     ISA         zaehle-von
zahl        =num2
```

Der slot *zahl* des Chunk *zaehle-von* wird modifiziert mit dem Wert der Variablen *num2*, die zuvor im retrieval-Buffer dem slot *zweitens* des Chunks *zaehl-anordnung* entnommen worden ist. Im nächsten Schritt wird ein Chunk-Abruf aus dem deklarativen Gedächtnis in den retrieval-Buffer vorgenommen, wobei der slot *erstens* des Chunks dem Wert der Variablen *num2* entsprechen soll:

```
  +retrieval>
     ISA         zaehl-anordnung
     erstens     =num2
```

Die dritte Aktion der RHS ist ein spezielles Kommando:

```
  !output!        (=num1)
```

Das Kommando *!output!* wird dazu genutzt, um bestimmte Informationen später im Modellablauf gesondert anzeigen zu lassen. In unserem Fall wird die gesonderte Anzeige dem Wert der Variablen *num1* entsprechen. Man wird also den Zählvorgang am Monitor beobachten können.

Produktion Stop

Eine letzte Produktion ist notwendig, um unser erstes ACT-R-Modell zu vollenden, die Überwachung des Erreichens eines Zielzustandes, und damit die Beendigung des Modells.

#9 Wir legen eine Produktion an, die überwacht, ob das Ziel erreicht worden ist:

```
(P fertig
   =goal>
     ISA         zaehle-von
     zahl        =num
     ende        =num
 ==>
   -goal>
   !output!     (=num)
 )
```

Da nun unser Blick schon etwas geübt ist, können wir unmittelbar erkennen, dass sich LHS und RHS lediglich auf den goal-Buffer beziehen. In der LHS wird getestet, ob der aktuelle Wert des slots *zahl* dem Wert des slots *ende* entspricht. Hierzu wird die Variable *num* verwendet. Zeigt sich nach der Prüfung ein Wahrheitswert, so wird der goal-Buffer bereinigt, da das aktuelle Ziel erreicht ist. Die aktuelle Zahl wird auf dem Bildschirm ausgegeben. Beim Feuern dieser Produktion wird also das Modell beendet.

Fertigstellen des Modell-Codes

Wir haben in den vorherigen Kapiteln die zentralen Elemente eines ACT-R Modells kennenlernen dürfen. In diesem Unterkapitel beziehen wir uns noch auf die letzten Kommandos und Zeilen, die das Modell zur Lauffähigkeit führen werden.

Clear-all Wir werden unser Modell zum Zählen in nächsten Abschnitt in der ACT-R-Umgebung zum Einsatz bringen. Die ACT-R-Umgebung kann sehr viele unterschiedliche Modelle aufnehmen und verarbeiten, die Modelle können also auch parallel laufen. Für uns ist es zunächst wichtig, dass sich nur unser erstes kleines Modell in ACT-R befindet und sonst nichts. Um alle Einstellungen der Umgebung auf ihre Grundwerte zu setzen und die ACT-R-Umgebung von möglichen anderen Modellen zu lösen, benutzen wir eine erstes wichtiges Kommando clear-all.

#10 Wir setzen in die allererste Zeile unseres Modells das neue Kommando:

```
(clear-all)
```

ACT-R wird in gewisser Weise mit diesem Kommando für unser Modell startklar gemacht. Dieses Kommando steht in der ersten Zeile vor dem Code zu unserem ersten Modell. Unser Modell starte mit einem anderen Befehl in der nächsten Zeile; wir werden es definieren müssen.

Definition Ein Modell wird eingeleitet mit einem einzigen Befehl, dem Kommando *define-model*. Diesem Befehl folgt der Name des Modells. Dieser sollte sehr spezifisch für dieses eine Modell gewählt werden. Wir haben bereits erfahren, dass Modelle in ACT-R parallel laufen können; durch den Namen sollte ihre Einzigartigkeit gekennzeichnet werden. Alle weiteren Befehle zum Anlegen von Chunk-Typen, Chunks oder Produktionen folgen erst unter diesem Befehl.

#11 Wir legen unser Modell an, indem wir in die zweite Zeile folgenden Code schreiben

```
(define-model zaehlen
```

Das wohl wichtigste Detail an dieser Codezeile ist die fehlende Klammer hinter dem Befehl. Diese abschließende Klammer setzen wir erst viele Zeilen später am Ende unseres Modells, sie befindet sich praktisch in der letzten Zeile unseres Modell-Codes. Der Name des Modells steht direkt hinter dem Kommando. Unser erstes Modell trägt den Titel *zaehlen*.

SGP Wie wir bereits gesehen haben, besteht ACT-R aus unterschiedlichen Modulen. Zu allen Modulen gibt es in ACT-R eine Menge an Parametern. Mit dem Befehl *sgp* (es steht für set/show general parameters) greift man auf diese Parameter zu. Wir werden mit einer letzten Zeile Code diese Parameter etwas spezifizieren.

#12 mit dem Befehl sgp spezifizieren wir einige wenige der Parameter in ACT-R:

```
(sgp:esc t:lf.05:trace-detail high)
```

Anhand der Syntax können wir erkennen, dass Parameter in ACT-R mit einem: angesprochen werden. Wir wollen diesen Aspekt nicht sehr vertiefen. Die ersten beiden Parameter beziehen sich auf den retrieval-Buffer. Interessant ist die dritte Spezifikation zu trace-detail, die uns praktisch in das nächste Kapitel führt. Als trace bezeichnen wir die spätere Möglichkeit, in der ACT-R Umgebung den Modellablauf zu verfolgen. Die Voreinstellung lautet in ACT-R *medium*, dies wird nun verändert auf *high*. ACT-R wird uns also die größtmögliche Information zum Ablauf des Modells liefern. Unser gesamter Modell-Code müsste wie folgt aussehen:

```
(clear-all)

(define-model zaehlen

(sgp :esc t :lf.05 :trace-detail high)

(chunk-type zaehl-anordnung erstens zweitens)
(chunk-type zaehle-von start ende zahl)

(add-dm
```

```
   (b ISA zaehl-anordnung erstens 1 zweitens 2)
   (c ISA zaehl-anordnung erstens 2 zweitens 3)
   (d ISA zaehl-anordnung erstens 3 zweitens 4)
   (e ISA zaehl-anordnung erstens 4 zweitens 5)
   (f ISA zaehl-anordnung erstens 5 zweitens 6)
   (g ISA zaehl-anordnung erstens 7 zweitens 8)
   (h ISA zaehl-anordnung erstens 8 zweitens 9)
   (i ISA zaehl-anordnung erstens 9 zweitens 10)

   (erstes-ziel ISA zaehle-von start 2 ende 7))

(p anfang
   =goal>
      ISA         zaehle-von
      start       =num1
      zahl        nil
  ==>
   =goal>
      ISA         zaehle-von
      zahl       =num1
   +retrieval>
      ISA         zaehl-anordnung
      erstens      =num1
)

(P inkrement
   =goal>
      ISA         zaehle-von
      zahl       =num1
    - ende       =num1
   =retrieval>
      ISA         zaehl-anordnung
      erstens       =num1
      zweitens      =num2
 ==>
   =goal>
      ISA         zaehle-von
      zahl        =num2
   +retrieval>
      ISA         zaehl-anordnung
      erstens       =num2
   !output!        (=num1)
)

(P fertig
   =goal>
      ISA         zaehle-von
      zahl        =num
      ende        =num
 ==>
   -goal>
   !output!       (=num)
)

(goal-focus erstes-ziel)

)
```

Sofern wir noch nicht zwischengespeichert haben, speichern wir nun unsere Datei unter dem Namen „zaehlen.lisp" an einem beliebigen Ort auf unserem Computer ab, um sie im anschließenden Abschnitt aus der ACT-R-Umgebung heraus laden zu können.

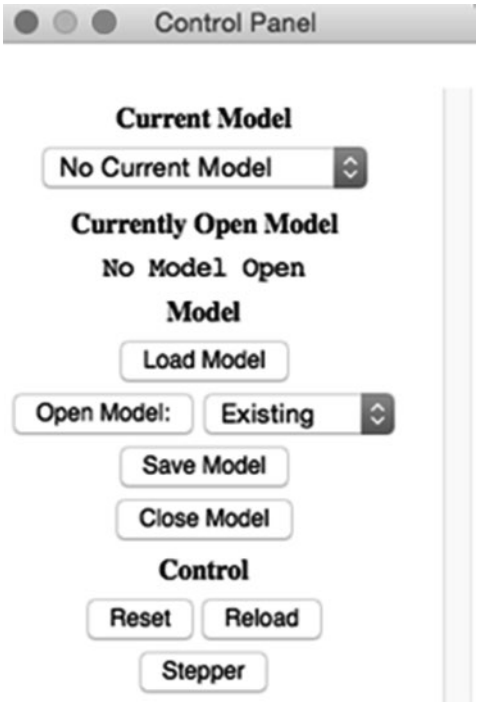
3.4.3 Das erste eigene ACT-R-Modell laufen lassen

Laden und Starten

In diesem Abschnitt werden wir unser erstes eigenes Modell nun in der ACT-R-Umgebung laufen lassen. Wir starten diese Umgebung, sofern wir dies noch nicht getan haben, aus dem ACT-R-Ordner heraus mit dem Skript „Run ACT-R.command". Zentrales Element der ACT-R GUI (Graphical User Interface) ist das sog. Control Panel, das uns eine Menge Bedienungskomfort im Umgang mit den Modellen gibt. Die Buttons weisen eine Vielzahl von Funktionen auf, wobei in unserer Einführung zunächst der obere Bereich mit den Überschriften *Current Model, Currently Open Model, Model* und *Control* ausreichen wird (◘ Abb. 3.10).

Zudem öffnet sich ein Fenster mit der Bezeichnung *ACT-R Listener*, in dieses Terminal können von uns weitere Interaktionen an dem Modell vorgenommen werden und wir interagieren auch mit der ACT-R-Umgebung über dieses Fenster.

Load Wir befinden uns nun in der Situation, ein eigenes Modell in die Umgebung laden zu können. Hierzu verwenden wir den Button *Load* und wählen den Speicherplatz unseres Modells „zaehlen.lisp". Wir erhalten eine Bestätigung, dass das Modell erfolgreich in unsere Umgebung



◘ **Abb. 3.10** Das *Control Panel* in ACT-R weist eine Menge an Funktionen auf. Für unsere Zwecke reicht zunächst das obere Drittel

Abb. 3.11 Nach erfolgreichem Laden eines Modells erhält der User eine Bestätigung

geladen worden ist (Abb. 3.11). Anschließend wird der Name unseres Modells unter *Current Model* angezeigt. Wir konnten während des Vorganges auch beobachten, dass eine Menge Informationen in unserem *ACT-R Listener*-Fenster erschienen sind.

Run Nach dem Ladevorgang ist unser Modell nun einsatzbereit. Wir haben versucht, ACT-R dazu vorzubereiten, von 2–7 zählen zu können. Ob es dazu nun in der Lage ist, werden wir gleich beobachten. Hierzu ist nur noch ein Befehl notwendig: run. Dem Befehl run für das Starten und Laufen des Modells ist immer ein Parameter hinzuzufügen, der die Laufzeit der Simulation in Sekunden angibt.

#13 Wir geben den Befehl run und den Wert 10 für eine 10-sekündige Modell-Simulation in das Terminal Fenster, den ACT-R-Listener, ein:

```
(run 10)
```

Das, was wir anschließend im Fenster beobachten können, ist der detaillierte Modellablauf. Wir erinnern uns, dass wir dies im obigen Abschnitt „Produktion Stop" mit dem Befehl: *trace-detail high* selbst eingefordert hatten. Nun können wir diese Spur unseres laufenden Modells zum ersten Mal nachverfolgen.

Den Modellablauf verfolgen

Im ACT-R-Listener-Fenster erscheint der gesamte beobachtbare Modellablauf für eine Simulation von 10 Sekunden. Schauen wir uns die Informationen etwas genauer an und starten mit der ersten Zeile:

```
     0.000   GOAL                    SET-BUFFER-CHUNK GOAL ERSTES-ZIEL
REQUESTED NIL (1)
```

GOAL Die Ausgabe besteht pro Zeile aus jeweils drei Teilen. Die erste Spalte zeigt zunächst die Zeit in Sekunden an. Da sich die erste Zeile auf den Start der Simulation bezieht, geht es mit der Sekunde *0.000* los. Die folgende Information zeigt an, welches Modul an dem gezeigten Ereignis

beteiligt ist. Es startet das goal-Modul. Die dritte Spalte gibt nun Informationen zu den Details des Ereignisses zu diesem Modul heraus: es ist herauszulesen, dass der Chunk *ERSTES-ZIEL* in den Buffer des goal-Moduls gesetzt worden ist. In der Information *NIL* zeigt sich zudem, dass diese Aktion nicht durch eine Produktion erfolgt ist. Wir erinnern uns, dass wir diese Aktion im letzten Teil unseres Modellcodes direkt in den goal-Buffer gelegt haben. Ein Detail, das für spätere Modelle wichtig ist.

3.4.3.2.1 Produktion Anfang

PROCEDURAL In der folgenden Zeile sehen wir, dass das prozedurale Modul aktiv ist. Hinter der Aktion *CONFLICT-RESOLUTION* verbirgt sich, dass die aktuellen Inhalte aller Buffer durchsucht werden und getestet wird, ob und welcher Bedingungsteil einer Produktion auf den aktuellen Zustand der Buffer passt:

```
    0.000   PROCEDURAL            CONFLICT-RESOLUTION                (2)
```

Darauf zeigt sich, dass die Bedingungen der LHS in der Produktion ANFANG erfüllt sind und diese Produktion ausgewählt wird:

```
0.000  PROCEDURAL       PRODUCTION-SELECTED ANFANG                  (3)
0.000  PROCEDURAL       BUFFER-READ-ACTION G ACT-R Trace-Schrift?OAL      (4)
```

Bei jeder Produktion, die ausgewählt wird, vergehen 50 Millisekunden zwischen dem Test der LSH und der Aktion der RHS. Es sind also 50 Millisekunden vergangen, bevor eine Produktion ihre Aktionen durchführt. Dieses Feuern wird in Zeile 5 bestätigt:

```
0.050  PROCEDURAL       PRODUCTION-FIRED ANFANG          (5)
0.050  PROCEDURAL       MOD-BUFFER-CHUNK GOAL            (6)
0.050  PROCEDURAL       MODULE-REQUEST RETRIEVAL         (7)
   0.050   PROCEDURAL            CLEAR-BUFFER RETRIEVAL             (8)
```

Wir erinnern uns an die RHS der Produktion Anfang und können nun an der Ausgabe (6) mitverfolgen, dass der goal-Buffer modifiziert werden sollte. Gehen wir in den Code unseres Modells zurück, so sehen wir, dass wir den Wert im slot *zahl* des *zaehle-von*-Chunks aktualisieren wollten. Die siebte Zeile bestätigt, dass der retrieval-Buffer, also das deklarative Modul, eine Abfrage erhält. Eine solche Abfrage führt zunächst dazu, dass der retrieval-Buffer bereinigt wird. Diese Form der Bereinigung wird auch als *implizites Clearing* bezeichnet und in Zeile acht bestätigt.

Der erste deklarative Chunk

Die Anfrage der Produktion an das deklarative Modul ist in Zeile 7 in der *0,050* Sekunde erfolgt. Zeile 8 bestätigt, dass das deklarative Modul die Anfrage erhalten hat und deren Prozess bereits in Sekunde *0,050* einleitet. Zeile 10 zeigt, dass das prozedurale Modul schon wieder im Modus

der aktiven Suche nach zu erfüllenden LHS-Bedingungen in den Buffern ist. Allerdings folgt dieser Tätigkeit keine Aktivierung einer Produktion; es scheinen noch keine Bedingungen erfüllt zu sein.


```
0.050  DECLARATIVE      START-RETRIEVAL                (9)
0.050  PROCEDURAL       CONFLICT-RESOLUTION            (10)
0.100  DECLARATIVE      RETRIEVED-CHUNK C              (11)
0.100  DECLARATIVE      SET-BUFFER-CHUNK RETRIEVAL C   (12)
```

In Sekunde *0,100* wird bestätigt, dass der retrieval-Buffer Chunk *C* erhalten hat. Dieser Chunk erfüllt folglich die Eigenschaften der in der Produktion *Anfang* gestarteten Nachfrage. Schauen wir in unseren Code, so sehen wir, dass Chunk *C* in seinem Slot *erstens* die 2 enthält. Wir wissen auch, dass unser System vor dem Ziel steht von 2–7 zu zählen. Es musste also mit der 2 gestartet werden. Zeile 12 zeigt an, dass der in Zeile 8 bereinigte retrieval-Buffer nun mit Chunk *C* besetzt wird.

Die zweite Produktion und ihre Folgen

Es folgt in Sekunde *0,100* eine weitere Abfrage des prozeduralen Moduls aller weiteren Buffer. Da sich der retrieval-Buffer nun aktualisiert hat, scheint die LHS einer Produktion erfüllt zu sein; diesmal ist es aber nicht die Produktion *Anfang*:

```
0.100  PROCEDURAL       CONFLICT-RESOLUTION            (13)
0.100  PROCEDURAL       PRODUCTION-SELECTED INKREMENT  (14)
0.100  PROCEDURAL       BUFFER-READ-ACTION GOAL        (15)
0.100  PROCEDURAL       BUFFER-READ-ACTION RETRIEVAL   (16)
0.150  PROCEDURAL       PRODUCTION-FIRED INKREMENT     (17)
2
0.150  PROCEDURAL       MOD-BUFFER-CHUNK GOAL          (18)
0.150  PROCEDURAL       MODULE-REQUEST RETRIEVAL       (19)
0.150  PROCEDURAL       CLEAR-BUFFER RETRIEVAL         (20)
```

Zeile 14 bestätigt, dass die Produktion *Inkrement* gewählt wurde, diese enthält auf der LHS Bedingungen zu zwei Buffern. Die Tests dieser Buffer werden in den Zeilen 15 und 16 angezeigt; es ist der goal-Buffer und der retrieval-Buffer. Da beide Tests positiv ausfallen, feuert die Produktion *Inkrement* in der Sekunde 0,150. Hierbei modifiziert sie den Chunk im goal-Buffer (Zeile 18) und sendet eine Anfrage an das deklarative Gedächtnis (Zeile 19). Vergleichen wir in unserem Code die RHS unserer Produktionen *Anfang* und *Inkrement*, so sehen wir, dass diese bis auf einen Befehl identisch sind. Dieser Befehl lautet *!output! (=num1)* und bewirkt, dass wir in unserer Anzeige des *ACT-R-Listeners* nicht nur die Reaktionen in den Modulen verfolgen können, sondern auch den aktuellen Wert des Slots *zahl* im Chunk-Typen *zaehle-von* erhalten. Wir können also den Vorgang des Zählens beobachten und sehen den aktuellen Stand: das System ist mit der 2 gestartet. Die Produktion Inkrement soll eine schrittweise Annäherung an den Zielzustand, in unserem Fall die 7, verursachen. Wir wollen also in unserem Output jede Zahl auf dem Weg von der 2 bis zur 7 sehen. Zeile 20 zeigt wieder ein implizites Cleaning durch eine Anfrage an das deklarative Gedächtnis. Zeile 21 ähnelt wieder der Situation in Zeile 10, das prozedurale Modul scannt in der Sekunde *0,150* wieder alle Buffer nach Bedingungen in der LHS, es folgt aber keine passende Produktion.

```
   0.150  PROCEDURAL      CONFLICT-RESOLUTION                    (21)
   0.200  DECLARATIVE     RETRIEVED-CHUNK D                      (22)
   0.200  DECLARATIVE     SET-BUFFER-CHUNK RETRIEVAL D           (23)
   0.200  PROCEDURAL      CONFLICT-RESOLUTION                    (24)
   0.200  PROCEDURAL      PRODUCTION-SELECTED INKREMENT          (25)
   0.200  PROCEDURAL      BUFFER-READ-ACTION GOAL                (26)
   0.200  PROCEDURAL      BUFFER-READ-ACTION RETRIEVAL           (27)
   0.250  PROCEDURAL      PRODUCTION-FIRED INKREMENT             (28)
3
   0.250  PROCEDURAL      MOD-BUFFER-CHUNK GOAL                  (29)
   0.250  PROCEDURAL      MODULE-REQUEST RETRIEVAL               (30)
   0.250  PROCEDURAL      CLEAR-BUFFER RETRIEVAL                 (31)
```

Die Anfrage aus Zeile 19 an das deklarative Gedächtnis erfüllt sich in Sekunde *0,200* durch den Empfang von Chunk *D* in den retrieval-Buffer. Chunk *D* erfüllt also alle Bedingungen, wie es zuvor bei Chunk *C* gewesen ist. Die Vorgänge in den folgenden Zeilen sind nun analog zu den vorherigen Ereignissen und die Produktion Inkrement läuft erfolgreich vorwärts bis die Zahl 3 in Sekunde *0,250* gezählt wird. Wir können erkennen, dass die Produktion *Inkrement* nun stabil in einer Schleife vorwärts läuft:

```
   0.250  DECLARATIVE     START-RETRIEVAL                        (32)
   0.250  PROCEDURAL      CONFLICT-RESOLUTION                    (33)
   0.300  DECLARATIVE     RETRIEVED-CHUNK E                      (34)
   0.300  DECLARATIVE     SET-BUFFER-CHUNK RETRIEVAL E           (35)
   0.300  PROCEDURAL      CONFLICT-RESOLUTION                    (36)
   0.300  PROCEDURAL      PRODUCTION-SELECTED INKREMENT          (37)
   0.300  PROCEDURAL      BUFFER-READ-ACTION GOAL                (38)
   0.300  PROCEDURAL      BUFFER-READ-ACTION RETRIEVAL           (39)
   0.350  PROCEDURAL      PRODUCTION-FIRED INKREMENT             (40)
4
   0.350  PROCEDURAL      MOD-BUFFER-CHUNK GOAL                  (41)
   0.350  PROCEDURAL      MODULE-REQUEST RETRIEVAL               (42)
   0.350  PROCEDURAL      CLEAR-BUFFER RETRIEVAL                 (43)
   0.350  DECLARATIVE     START-RETRIEVAL                        (44)
   0.350  PROCEDURAL      CONFLICT-RESOLUTION                    (45)
   0.400  DECLARATIVE     RETRIEVED-CHUNK F                      (46)
   0.400  DECLARATIVE     SET-BUFFER-CHUNK RETRIEVAL F           (47)
   0.400  PROCEDURAL      CONFLICT-RESOLUTION                    (48)
   0.400  PROCEDURAL      PRODUCTION-SELECTED INKREMENT          (49)
   0.400  PROCEDURAL      BUFFER-READ-ACTION GOAL                (50)
   0.400  PROCEDURAL      BUFFER-READ-ACTION RETRIEVAL           (51)
   0.450  PROCEDURAL      PRODUCTION-FIRED INKREMENT             (52)
5
   0.450  PROCEDURAL      MOD-BUFFER-CHUNK GOAL                  (53)
   0.450  PROCEDURAL      MODULE-REQUEST RETRIEVAL               (54)
   0.450  PROCEDURAL      CLEAR-BUFFER RETRIEVAL                 (55)
   0.450  DECLARATIVE     START-RETRIEVAL                        (56)
   0.450  PROCEDURAL      CONFLICT-RESOLUTION                    (57)
   0.500  DECLARATIVE     RETRIEVAL-FAILURE                      (58)
   0.500  PROCEDURAL      CONFLICT-RESOLUTION                    (59)
   0.500  ..............  Stopped because no events left to process (60)
```

Unser Modell zählt erfolgreich weiter bis zur Zahl 5. Bis hierher einen herzlichen Glückwunsch! Sie haben selbstständig einen ersten ACT-R-Code erstellt und dessen Lauffähigkeit ausführlich innerhalb von ACT-R beobachten können. ACT-R interagiert in unserem Beispiel nur mit seinem eigenen Code. Durch seine lange Geschichte seit 1976 ist ACT ein sehr mächtiges Produktionssystem. Traditionelle Forschungsparadigmen und Experimente der Psychologie, ob zum Stroop-Effekt, zur Aufmerksamkeit nach Sperling oder zum Turm von Hanoi, sind in den letzten Jahren in ACT-R simuliert und repliziert worden. Wirklich spannend ist natürlich, dass ACT-R mit allen möglichen virtuellen Welten interagieren kann und noch spannender ist die Tatsache, dass es mit der realen Welt in Interaktion treten kann.

3.4.4 Problemlöseorientierter Abschluss

Wollten wir das Modell nicht bis zur Zahl 7 zählen lassen? Es hat allerdings nur bis zur Zahl 5 gezählt. Woran kann das liegen? Vielleicht ist Ihnen das Problem schon während unseres Weges durch den Code aufgefallen. Ansonsten versuchen Sie auf die Suche zu gehen und das Problem zu lösen! Wie können Sie dabei vorgehen? Versuchen Sie, sich beim Problemlöseprozess zu beobachten! Mit welcher Systematik gehen Sie bei der Suche vor? Verläuft Ihre Suche erfolgreich oder kommen bei fehlendem Erfolg unerwünschte Gefühle auf? Damit wir Ihre Frusttoleranz nicht zu sehr auf die Probestellen, befindet sich selbstverständlich eine Lösung des Problems auf der begleitenden Internetseite. Aber versuchen Sie möglichst lange, ohne diese zusätzlichen Informationen das Problem zu lösen! Ein Modell zum Zusammenhang zwischen Problemlösen und Emotion wird uns in Kap. 4 des Buches auch noch begegnen. An dieser Stelle haben wir vorerst die Möglichkeit einer Selbstbeobachtung.

3.4.5 Literaturempfehlungen

Wir konnten anhand eines kleinen Modells einen ersten Eindruck zur Softwareumsetzung von ACT-R gewinnen. Es lohnt sich, das Tutorial vertiefend abzuarbeiten. Zusätzlich ist natürlich ein historischer Blick (Anderson, 1976) ebenso interessant wie eine aktuelle und umfassende Übersicht zu ACT-R (Anderson et. al. 2004; Anderson 2007). Physiologische Aspekte finden sich ausführlich bei Anderson (2010). Eine übersichtliche Einführung in diese Forschungstradition bieten Kandel u. Squire (2009), eine Ausfächerung der Teilgebiete findet sich bei Pritzel et al. (2003). Da ACT-R auf der Programmiersprache LISP basiert, sei die systematische Einführung in diese Hochsprache von Seibel (2005) empfohlen. Eine praktische Auseinandersetzung mit Paradigmen der künstlichen Intelligenzforschung unter Verwendung von LISP bietet Norvig (1992).

Literatur

Anderson, J. R. (1976). *Language, memory and thought*. New Jersey: Erlbaum.

Anderson, J. R. (2007). *How can the human mind occur in the physical universe?* Oxford: University Press.

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological Review*, 111(4),1036–1060.

Kramer, O. (2009). *Computational Intelligence*. Heidelberg: Springer.