University of Groningen **Faculty of Mathematics and Physical Sciences**

Department of Computing Science

Rehearsal incorporated

Annelies Nijdam

supervisors: Prof.dr. G.R. Renardel de Lavalette
Drs. N.A. Taatgen

august 1995

RuG

Contents

Acknowledgement

A number of people have contributed to this project. I thank them, for without them I would not have been able to present my graduate work in this way.

In the first place I thank both my supervisors, Niels Taatgen and Prof. Gerard Renardel. Niels has been the initiator of this project and has set out the lines for the implementation of the model. Apart from that, he was always willing to help me out and guide me through the back alleys of ACT-R. I am grateful to Prof. Renardel, for he provided me with comments that were very useful to set up and improve the report on the project. He was even ready to review the first two chapters during his holidays.

I also thank Constantijn who has helped me until late at night making the postscript pictures that illustrate this report. He also helped me making the sheets for my presentation and placed his work-room at my disposal, so I could continue working during the week-end . Great!

And then there are those that have contributed sideways to the project. I am indebted to Annemieke Beereboom, our coordinator of international student exchange, who has been a great help during my stay at the Université de Bourgogne when I had to organize the pursuit of my studies at the University of Dallas. I thank her for the encouraging support and creative solutions she offered to my urgent problems.

Of course I thank my parents, because they have facilitated me to study comfortably through their financial support. But I would especially thank my mother who has provided for board and lodging during the last few months of the project. I also thank my friends who, no doubt at times, have found it hard to accept my seclusion during my last months in the Netherlands. They have supported me through the hard times and without exception have lent a ready ear to hear me out.

Annelies

Preface

The origin of the project

To observe human behaviour on a planning task, Niels Taatgen (Ph.D. student at the Department of Psychology, University of Groningen) conducted experiments in which one of the groups had to do the planning overtly by heart. Examination of the verbal protocols showed that all subjects made extensive use of verbal rehearsal to keep track of their current results. They alternated between rehearsing the current solution and adding new pieces to it.

The cognitive architecture ACT-R is a tool to model human behaviour on problem solving. However, rehearsal is not incorporated in ACT-R. In this thesis, we show how ACT-R can be extended with rehearsal and claim that this improves ACT-R with respect to the modeling of problem solving behaviour in planning.

Plan of actions

The first thing to do is to find a theory on rehearsal. There are two criteria that this theory has to meet.

1. The nature of the theory has to fit in with that of ACT-R. That is, it should not violate the major assumptions in ACT-R, because it has to be inserted into ACT-R without disrupting the original structure of the program.

2. The theory has to be able to explain a number of phenomena that are associated with rehearsal.

Then decisions about the implementation of the theory on rehearsal have to be made. We aim at total independency of the ACT-R program towards the "rehearsal-code", leaving the original ACT-R program fully intact. How can, within these limitations, the various concepts of the theory be represented and what operations will manipulate them?

When rehearsal is incorporated, we have to consider to what extent this implementation offers the representative power to model and simulate rehearsal in problem solving. Experiments have been and will have to be conducted to reveal to what degree this implementation matches the data of real life experiments.

Outline of the contents

In the first chapter the study of literature in search of a suitable model of rehearsal is presented. I started from the principle that this chapter has to be worth reading for the average university graduate, so first some general concepts in psychology of memory are introduced. To get a clear idea about what rehearsal means and what effects it has on memory, specific theories of memory are outlined. The last section elaborates upon Baddeley's model of memory, as this theory embodies the most suitable mechanism to account for rehearsal: the Phonological Loop.

In the second chapter the ACT-R theory and program are discussed. The core notions of the architecture are outlined by means of the basic assumptions that are made in the theory. Next the implementation is more elaborately discussed, since from a computer science viewpoint we experienced a lack of documentation on the ACT-R program. First the ACT-R process is schematically described, after which the main features in the implementation are presented more elaborately. Then some general implementation details are provided. In the last section the files, of which the LISP code of the ACT-R program is made up, are set out in detail.

In Chapter 3 the arguments in favour of uniting rehearsal and ACT-R in one program are set out. Then some constraints on the model are discussed and subsequently the consequences of those constraints for the implementation on the model are set forth.

In Chapter 4 the code that implements the rehearsal process is discussed. Explained is how it works, why it is implemented in such a way and what the effects on the performance of ACT-R are.

Chapter 5 evaluates the outcome of the project on the basis of the constraints that were imposed on it in Chapter 3. It discusses some experiments, the strong and weak points of the implementation and suggests how it can be improved.

Finally in the last chapter, we draw some conclusions.

Chapter 1

Rehearsal in Human Memory

1.1 Introduction to some basic concepts

In everyday life, we have to perform cognitive tasks and when we do, we have to retain information for at least a short period of time. That is because the information vital to the solution of the task has to be available in memory when it is no longer present in our environment.

Memory does not simply supply us with the desired information. By experience we know that we have to exert ourselves to retain something. In order to assure the efficacy of memory, one has to spend energy on and devote attention to it.

To a great extent, research in psychology has been devoted to the psychology of memory. This has resulted in a large body of theories that is continuously evolving. Frequently theories compete with each other. As a consequence experiments are conducted to determine which one approaches the reality the best.

1.1.1 Chunks and Memory traces

A chunk is a semantic unit in memory: a cluster of memory cells that, as a group, is being associated with a concept. The presentation of information results in the formation of so-called *memory traces*. When, for example, the number 1 is presented, the chunk that is being associated with the concept 1 will be activated. This set of activated cells forms a knowledge structure and is called the memory trace. Of course this is a simplification, but at the moment the precise processing is of no importance.

1.1.2 Activation vs. durability

Most current theories of human memory agree on two important aspects:

- The activation of memory traces is assumed to *fade away* in about 1.5–2 seconds [Baddeley 1992].
  The things you currently think about, are presently active in memory. As knowledge with a low state of activation is not directly available, it costs more time to recall. Availability can thus be expressed in terms of activation, which decays over time.

- Certain recollections are very *durable*.
  Knowledge structures are interconnected by associations through which the concepts are tracked down. A strongly coded memory trace is highly connected with other knowledge stuctures. Because durable recollections are assumed to be strongly coded, durability is also called the long term strength of a memory trace. When associations are not used, they slowly fade. It is not a spontaneous decay, but a disruption caused by other associations that gain strength over time.

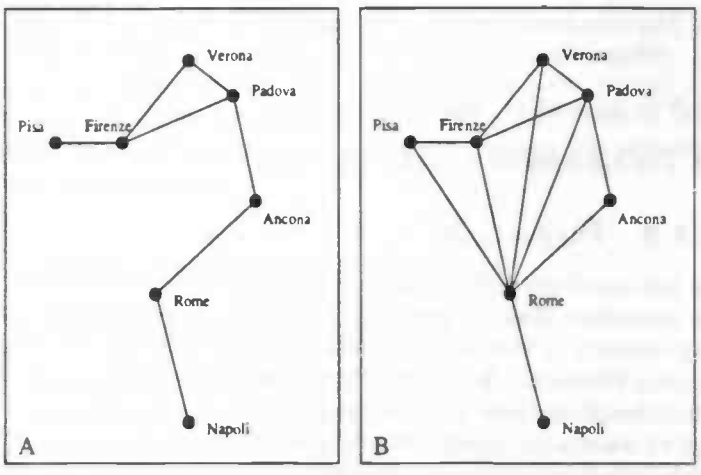


Figure 1.1: Metaphore associations—roads

The role of associations can be illustrated by a metaphore. Look upon memory traces as towns and consider associations to be roads that connect them. Figure 1.1 illustrates this metaphore: In situation A Rome is only weakly connected, for there are only two roads that lead to other towns. As a consequence one does not lightly get lost, for a small number of roads will not readily be confused.
In situation B Rome is highly connected to other towns. When you want to travel to Rome, you will reach it quickly and easily, for there are more roads

that lead to Rome. On the other hand, one will confuse the different roads more easily, because there are so many to choose from.

One can illustrate activation and durability of human memory with table 1.1 [Anderson, 1980, table 6-1] and an example:

Table 1.1: Activation vs.durability

	durable	not durable
high activation	A something familiar you are currently thinking of	B something you have just put put in memory
low activation	C knowledge you are not thinking of right now	D things you can not remember anymore

Situation: You want to make reservations for a movie.

A The name of the cinema.

B The telephone number of the cinema. You repeat it until you have completely dialed it.

C The director of "Apocalypse Now".

D The old telephone number of a friend.

1.1.3 Forgetting

In free-recall experiments several items are presented, after which subjects have to reproduce them in any order they like. A general observation in free-recall experiments is that subjects often covertly rehearse the items to prevent forgetting [Anderson & Milson, 1989]. This experimental situation corresponds to situation B in table 1.1. In 1958, Brown showed that even small amounts of information were rapidly forgotten when rehearsal was prevented [Brown, 1958]. This causes us to presume that rehearsal is a strategy to retain information for at least a short period of time.

When, in free-recall experiments, the capacity of immediate memory is exceeded, one can observe the serial position effect: not all items have the same probability to be recalled, it depends upon the position of the item in the order of presentation.

In figure 1.2 we see that recall for the first items *(a)* is better than for those in the middle section *(b)*. This is called the *primacy effect*. The items in the middle section *(b)* have the highest likelihood to be forgotten. The *recency effect* occurs at the end of the list: the last few items *(c)* are recalled best and

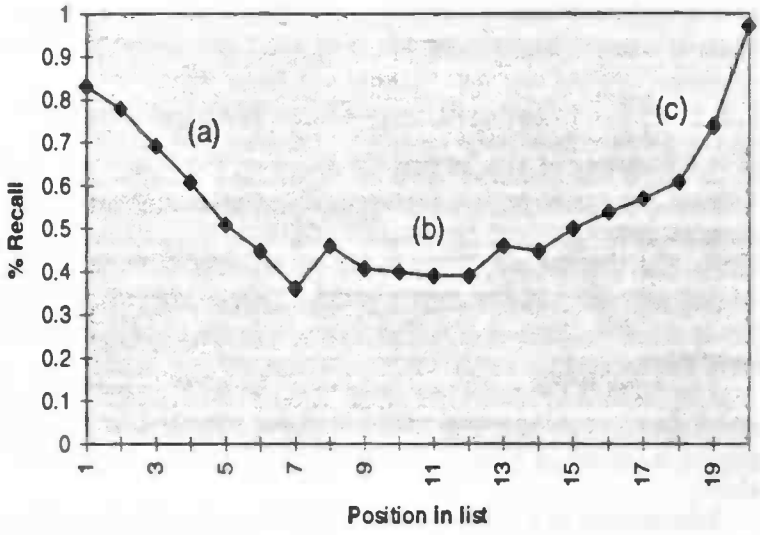


Figure 1.2: Recall depending on position in list

the probability of recall of the most recent item approaches unity. The serial position effect was discovered by Murdock [1962] and has proved to be very robust. The recency effect disappears when rehearsal is prevented [Postman & Philips, 1965]. This indicates that the rehearsal strategy has something to do with the serial position effect.

1.2 History of memory models

1.2.1 Miller's magical number seven

Miller [1956] observed in his immediate recall experiments that subjects could recall seven plus or minus two unrelated items in correct order. One can also observe that subjects reduce the amount of items that have to be memorized by a process that we call chunking: items are regrouped into other items in a convenient and logical way. For example, people who read fluently, separate sequences of letters into words and can, this way, memorize much more than seven letters. The 12-letter sequence HATPENDOGRUB will easily be remembered as the four items HAT PEN DOG RUB. The discovery of Miller's magical number seven was a revolutionary one, for he was the first to argue that the limitation of the memory span could not be measured in the amount of information (the number of bits) it contained [Baddeley, 1994, Shiffrin & Nosofsky, 1994], but partially depends upon the amount of processing that could reduce the number of concepts to be retained.

As a model for immediate memory, a buffer of seven items is far too simplistic. Therefore more elaborate models of memory are discussed subsequently.

1.2.2 Short-Term/Long-Term Memory model

In the majority of the models for memory, durability and activation reflect different aspects of human memory. This supports the idea of a dichotomy in memory [among others Brown, 1958, Broadbent, 1958, Peterson & Peterson, 1959]. On the one side there is the short term memory (STM), which has a limited capacity and fast access based on the activation of memory traces. On the other side there is the long term memory (LTM) that contains durable knowledge.

The STM/LTM-model assumes that the STM is based on memory traces, which decay spontaneously unless they are refreshed by active rehearsal. Forgetting is caused by memory traces which have faded below the point of retrievability.

Information in LTM is retained much longer without continuous effort. We do not have to keep repeating a well-known telephone number (like our own) to be able to recall it. This indicates a different cause of forgetting in LTM. Forgetting occurs in LTM by interference between new information and old habits. When you get a new telephone number it becomes harder to remember the old one. The amount of interference increases with the similarity of the competing traces.

Rehearsal and Levels of Processing

Following the Levels of Processing Framework of Craik & Lockhart [1972], information is assumed to follow a sequence of stages from the peripheral sensory level to the deep semantic level. This can be illustrated by the process of reading: first we see a word and then we read it, after which it can be processed more deeply. Each stage of processing is assumed to leave a memory trace. The durability of a memory trace increases with the depth of processing, because the deeper a stimulus is processed, the stronger it is coded (see also the notion of durability at page 7). For example you process a word very shallowly when you have to determine whether it is written in capital letters. If you have to judge whether a word rhymes to another word, you process it more deeply. When you have to fit it into a sentence, the processing takes place at an even deeper level. Craik & Tulving [1975] showed that more deeply processed items are indeed recalled better.

The function of STM is seen in terms of the processing it carries out. The two principal processes distinguished are *maintenance rehearsal* and *elaborative rehearsal.*

Maintenance rehearsal simply prevents forgetting during a cognitive process. Information is being rehearsed without transforming it into a deeper, more

durable code. It does not lead to essential long term learning, for it only refreshes the already existing memory trace. The classic example of maintenance rehearsal is that of a new telephone number that you have to repeat overtly or covertly until you have written it down or dialed it.

Long term learning is assumed to depend upon **elaborative rehearsal**, in which each successive processing of an item increases the depth of encoding. This means that inter-item associations are strengthened or expanded. Long term learning is therefore also called associative learning. When you look for a mnemonic to help you remember a telephone number it is a case of elaborative rehearsal. In theory the fundamental difference between maintenance rehearsal and elaborative rehearsal is that maintenance rehearsal reactivates already existing knowledge structures, whereas elaborative rehearsal creates new associations or strengthens existing associations. In practice these processes interact and are difficult to separate. It is therefore better to talk about a continuum on which a process is described as having more maintenance or elaborative characteristics.

Definition(s) of rehearsal

There are, depending on the line of approach, several definitions of maintenance rehearsal. In these definitions there appears to be great conformity on the main features [Greene, 1987]:

- **Repetition** of the process, rather than extension:
  The processing of an item is being repeated at the same level, instead of being more deeply encoded.
- The **low-level quality** of the process:
  Processing of an item takes place at a shallow level.
- **Minimal quantity** of cognitive resources involved:
  minimal attention and capacity is spent on maintenance rehearsal.

Elaborative rehearsal also involves repetition, but this takes place at a higher level of processing and more cognitive resources are involved.

1.2.3 Working Memory

There are two approaches to STM/Working Memory: An empirical one, which started with Miller, and a theoretical/cognitive one, which embodies the need of a short term store for information. The first is often called STM, the latter Working Memory. Several memory models identify the emiprical concept STM with the theoretical concept Working Memory (WM) [among others Baddeley & Hitch, 1974]. When memory traces are highly activated, we are currently working with them and thus the WM is defined as the whole of the knowledge structures with a high state of activation. The STM is involved in a wide range of cognitive tasks and it is assumed to be able to employ several storage and

processing strategies, from which rehearsal is the most important. Short-term memory has two important features in the STM/LTM model:

1. It has a limited capacity
2. Information fades away quickly

Critique on the STM/LTM model: The STM/LTM model is naive, because it just assumes two memory systems, where information enters the LTM after it has resided in STM. Moreover, this model can not explain other aspects: STM seems to be a multicomponent system in which (at least) one of the subsystems depends on a phonological code.

Multicomponent System

When the memory span is almost completely used, you expect that the performance on tasks, like reasoning, learning, comprehension or recall will be severely affected. One can indeed observe an interaction effect, but it is by no means as dramatical as would be expected from an uniform STM. This and other experimental results have led to the conclusion that the STM consists of multiple subsystems [Baddeley, 1986], for multiplicity can explain the lack of interaction: the retrieval of an item from STM is the task of a subsystem other than the one responsible for the limited capacity.

Phonological Code

The STM, or at least a part of it, depends upon phonological coded memory traces of which the spontaneous decay can be arrested by subvocal rehearsal. This contrasts with the LTM which is assumed to be based on a semantic code.

The indications for a phonological code come from four phenomena which were observed in several immediate recall experiments:

1. **Phonological similarity effect**: items that are similar in sound tend to be mutually confused, leading to poorer immediate serial recall.
2. **Irrelevant speech effect**: immediate serial recall is disrupted by the presentation of irrelevant spoken material.
3. **Wordlength effect**: number of recalled items decreases with increasing spoken duration of the sequence of items.
4. **Articulatory suppression**: immediate memory performance is impaired when subvocal rehearsal is prevented.

Each model for the STM will have to deal with these phenomena.

1.2.4 Baddeley's Working Memory model

Throughout the years several models for the STM have been developed. The Working Memory model of Baddeley & Hitch [1974] appears to be the best candidate, because it not only incorporates the main features of the STM (multicomponent system based on phonological code, limited capacity), but it also plausibly explains the four aforenamed phenomena. Since the introduction of the model it has been adapted and extended [Baddeley, 1986].

Baddeley's system comprises three components, namely the *Central Executive* supported by two slave systems, viz. the *Phonological Loop* and the *Visuo-Spatial Sketchpad*. This is schematically illustrated in figure 1.3.

Figure 1.3: Baddeley's Working Memory model

The Phonological Loop is a rehearsal mechanism specialised in processing speech-based information and the Visuo-Spatial Sketchpad holds and manipulates visuo-spatial information. The Central Executive controls the two slave systems, which in turn place minimal demands on the Central Executive in their functioning.
The functioning of the Visuo-Spatial Sketchpad does not seem to be relevant to the rehearsal process, so I will now successively discuss the Phonological Loop and the Central Executive.

The Phonological Loop

The Phonological Loop consists of two subcomponents:

1. A *phonological store* holds phonologically coded information, from which the traces become irretrievable in 1.5–2 seconds.

2. The *articulatory control process* can reactivate traces by subvocal rehearsal, or transforms written information in phonological code and subsequently store it in the phonological store [Baddeley, 1992].

The functioning of the Phonological Loop will be clarified by showing how it offers a natural explanation for the four phenomena previously listed on page 12:

Phonological Similarity Effect Conrad [1964] was the first to observe that phonological similar items will more easily be confused in immediate recall tests.

Also lists with similar sounding items, like B-G-V-P-T, will not be as well retained as lists which consists of different sounds, for example Y-H-W-K-R [Conrad & Hull, 1964].
This is assumed to occur, because the store of the loop is based on a phonological code. Distinction between memory traces is essential to recall. As similar codes result in fewer discriminating features between traces, phonological similarity leads to to impaired retrieval and poorer recall.

Irrelevant Speech Effect Unattended speech (in native tongue, unfamiliar language or meaningless words) disrupts the retention of sequences of visually presented items [Colle & Welsh, 1976, Salamé & Baddeley, 1982]. Not every sound has this effect. Some sounds will not affect retention at all (arbitrary noise) and others will have a slight effect (like music[1]), compaired to the disruption caused by speech sounds. Music also deteriorates memory performance, especially in combination with singing, while arbitrary noise produces no effect [Salamé & Baddeley, 1989].
It is assumed that irrelevant speech gains obligatory access to the phonological store and is thus able to corrupt the memory traces there, which leads to impaired recall.

Wordlength Effect Baddeley, Thompson & Buchanan [1975] observed an inversed linear relation between memory span and the spoken duration of the sequence of words. The crucial feature is the duration, not the number of syllables [Baddeley, 1986]. Memory span (S) can thus be expressed in terms of reading- or pronuncation-rate (R): $S = c + kR$, in which c and k are constants. This can be explained as follows: At the presentation of an item, a memory trace is formed in memory, which decays with the passing time. Re-presentation of an item (by the experimenter, or through rehearsal by the subject self) will reactivate the trace and prevent it from fading away. The amount of information retained is therefore a joint function of decay rate and rehearsal rate. As the spoken duration of the sequence increases, so will the time needed to rehearse the entire sequence. At a certain moment the decay time for one item will be less than the time to rehearse the entire sequence and consequently errors will occur [Baddeley, 1986]. However, pauses up to $\pm$ 4 seconds do not cause rehearsal to fail. What happens exactly when the Phonological Loop is overloaded, is not clear. Probably the subjects swap to a visual or semantic strategy, when the performance drops below the efficiency limit [Salamé & Baddeley, 1986]. The speed of subvocal rehearsal limits the rate at which a memory trace can be refreshed and in this way the memory span.

Articulatory Suppression Immediate memory span is reduced when subjects suppress subvocal rehearsal by continiously uttering an irrelevant sound,

[1] It is worse in combination with singing

such as *"the"*. The wordlength effect is suppressed and when the items are presented visually, the irrelevant speech and phonological similarity effect are also removed.
This is interpreted as the suppression of subvocal rehearsal: the articulation of an irrelevant item dominates the articulatory control process and prevents it from being used to maintain material already in the phonological store or to convert visual material into phonological code [Baddeley, Lewis & Vallar, 1984].

A remark: One of the major functions of the Phonological Loop is the preservation of information about serial order, but little is said about how this is done. It is desirable that the Phonological Loop in Baddeley's model will be extended to incorporate a mechanism for storing both inter-item and position-item associations [Burgess & Hitch, 1992].

The Central Executive

The Central Executive has not been studied to the extent of which the two slave systems have been studied, because the Phonological Loop and the Visuo-Spatial Sketchpad were assumed to offer more tractable problems that are an easier way into the Working Memory. The major role of the Central Executive is to coordinate cognitive processes and their execution [Morris & Jones, 1990]. The Central Executive acts as a supervisor that directs the attention and controls the selection of the most beneficial strategies for integrating information from multiple sources, including the two slave systems. As such it is used for decision making and controlling the amount of resources that needs to be distributed among the ongoing information processes. [Baddeley, 1986].

The operation of the Central Executive is of interest, because the rehearsal process requires more than only the Phonological Loop. For example the memory updating is not performed by the Phonological Loop itself, but by the Central Executive, because the updating performance has proven to be independent of the memory load [Morris & Jones, 1990].
The Central Executive has no memory capacity of itself, according to an experiment done by Morris & Jones [1990].

In 1986 Baddeley proposes the *supervisory attentional system* (SAS) of Norman & Shallice [1980, 1986] as a preliminary model of the Central Executive. This system has been developed to model the attentional control of action. The SAS can be used to bias the selection of schemata through the application of extra activation or inhibition [Morris & Jones, 1990]. This way the SAS has to maintain long term goals and resist the distraction of stimuli that might otherwise trigger conflicting behaviour. When we have to swallow a badly tasting medicine, we are inclined to spit it right back out. The bad taste is an interfering stimulus that has to be suppressed by the long term wish to get well.
In addition to maintaining goals, the SAS is responsible for overriding existing

activities in order to achieve long term aims. This includes monitoring the outcome of current strategies and switching these when they cease to be fruitful [Baddeley, 1992].
Just like the Central Executive, SAS is assumed to have a limited capacity. This is one of the main reasons why the concept of the SAS is well suited as a provisional model for the Central Executive.

Critique on Baddeley's model: Baddeley's model is a somewhat disembodied theory, for it does not account for interaction with the rest of cognition. Moreover, the Central Executive is a weak point within the theory, as it has become a homunculus on which things that can not be explained within the slave systems are loaded. When a part of a theory acts as a homunculus it is hard to account for its falsifiability.

Remedy: The Phonological Loop seems to be a plausible subsystem of working memory. However, the problems with the Central Executive will not readily be solved, so we need a way to get round this system. This can be achieved by independently lifting the Phonological Loop out of Baddeley's theory and incorporating it in a cognitive architecture, like ACT-R.

1.3 Glossary

The terms chunk, item and concept represent essentially the same thing. It depends especially on the context in which it is used. The term **concept** is probably the most general. It is used to refer to *"something"*, ranging from, for example a table to an addition. In memory tests the term **item** is frequently used: it indicates a word that needs to be retained. The term **chunk** will rather be used on a more "physical" level to refer to the internal representation in ACT-R of a concept in memory.

Chunk a semantic unit in memory, representing a small, but complete piece of information

Free-recall experimental method in which the items are presented once, after which subjects have to reproduce them in any order they like

Knowledge structure cluster of memory cells that is being associated with a single concept

Memory span the capacity of short term memory

Memory trace activated structure in memory that represents a concept of knowledge

Serial position effect when a list of items needs to be retained, the chance that a particular item will be recalled depends upon its place in the list

Chapter 2

ACT-R

2.1 Introduction

2.1.1 In general

The ACT-R theory of J.R. Anderson is a relative complete proposal about the structure of human cognition. It can be called a cognitive architecture, because it accounts for a wide range of cognitive phenomena and is not restricted to one of them. The acronym ACT stands for *A*rchite*CT*ure and R stems from *R*ational).
The ACT-R program that comes along with the ACT-R theory is an implementation of this cognitive architecture with the aim of modeling and simulating human cognition, in particular human problem solving.

2.1.2 Documentation available

A theory is formulated on an abstract level and can therefore be implemented in several ways. At points where the theory is ambiguous, the implementation must make specific choices. The ACT-R theory is set forth in natural language in "Rules of the Mind" [Anderson, 1993], but as Anderson already had an implementation in mind, he sometimes interfused the theoretical enunciation with concepts on the implementation level.

The documentation of the ACT-R program is rather limited. The user's manual and primer are well suited to get started with ACT-R, but when more complicated problems arise, or more detailed information is necessary, they are clearly inadequate. Implementation details have to be inferred directly from the program code. Fortunately, the code is well structured and interspersed with useful and clear comment. It would also be desirable to know which choices the program designers have made, for it would reveal to a greater extent how different mechanisms in ACT-R co-operate. The documentation also fails to

indicate a direct and close relation between theory and program. This chapter supplements the existing documentation in that it connects more thightly the structure of the implementation and the different concepts in the theory. This way it attemps to guide you through the labyrinth of implementation details.

2.2 Core of the Architecture

2.2.1 Starting-points

The ACT-R theory has three important features:
- Cognitive skills can be realized by production rules.
- There are two memory systems
- Rational analysis

Production system

Production rules are *condition-action pairs*, in which the condition specifies the circumstances under which the corresponding action is to be carried out. The action can be executed, when the condition is satisfied. A production rule can be represented as follows:

`IF` condition `THEN` action.

A set of production rules is called a production system. Most expert systems are implemented as production systems. However, they have a fundamentally different character, as they address only one skill and do not pursue the modeling of human cognition in general. Complex cognitive skills can be accomplished by concatenating the execution of several production rules. This way a production system can model and simulate for example the addition of two numbers. Throughout this chapter the addition example will be more elaborately explained (for instance on page 20). An example of a production system that performes an addition is supplied in appendix A.

Two memory systems

There is a fundamental difference between *knowing how* and *knowing that*. You know how to add two numbers and you probably know that a cat is an animal. ACT-R proposes two memory systems that reflect this distinction: Production rules are stored in *procedural memory* and describe how to do something. Facts and things are represented by chunks in *declarative memory*. A chunk is an element of declarative memory.

Working Memory is the part of declarative memory in which the chunks that are highly activated. ACT-R does not embody a clear-cut Working Memory. ACT-R has no Working Memory. However, the role of Working Memory is taken care of by an activation mechanism, that controls the availability of a chunk in

declarative memory. Because of their high state of activation, the chunks in Working Memory will be used more readily than those in normal declarative memory (see also subsection 1.2.3).

Rational Analysis

The general principle of Rational Analysis is that a system adapts itself to its environment. That is, it adjusts itself to the demands the environment places on the system. For example, a problem can require a reasonable solution, instead of the best solution at higher costs.
The principle of ACT-R is incorporated in ACT-R: At a certain moment, there can be several productions ready to execute their actions. As only one is allowed to execute, a choice has to be made. Anderson's principle of Rational Analysis implies that on the basis of rational criteria, the best choice can be made. In ACT-R this results in comparing the nominees on the basis of their costs (= time) and expected gain. The exact mechanism is explained on page 27.

2.2.2 Goal-directed performance

In ACT-R a task will often be divided in several smaller subtasks. Every task has a goal: attaining the solution successfully. Subtasks have corresponding subgoals. While you are working on the solution of a subgoal, you have to keep track of previous, not yet attained (sub)goals. A goal can be represented by a chunk in declarative memory.

A multi-column addition illustrates the need to retain subgoals:

```
765     While you add two numbers in the second
340 +   column, you have to recall the original
-----   goal of performing the total addition.
... 5
```

The condition part of a production rule is satisfied when chunks from declarative memory have successfully matched it. Each production rule is focused on a goal. At one point in time, there is always one goal that needs to be attained. This happens when the chunk, representing the goal, has been matched by a production and consequently executes its action(s). To assure that a goal will be attained, it is put in the *focus of attention*. The chunk in the focus receives extra activation, through which it is the first chunk that will be used to match the condition patterns.

When a condition has been satisfied, the corresponding action(s) can be carried out. Usually this consists of adding a chunk to declarative memory or changing the focus of attention. After a production has executed, the matching process can start again.
The ACT-R process is illustrated in figure 2.1

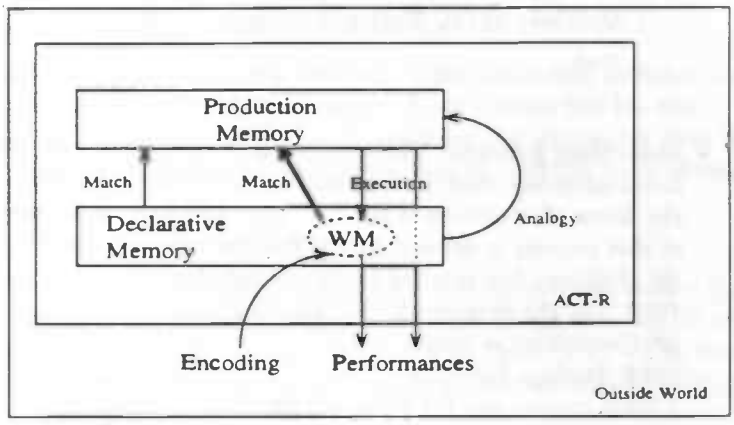


Figure 2.1: Actions and their destinations

2.3 Implementation

2.3.1 Process description

The ACT-R process can be separated into different levels (•-ed), which can have several subcomponents that will be executed sequentially (indicated by ;) or has some options (∗-ed) to choose from.
The **RUN-command** starts the ACT-R process that will try to perform the task. Optionally one can indicate the maximum number of cycles that will have to be carried out.

- **ACT-R process** can be defined with the statement:
  `WHILE NOT` *(Stop Execution)* `DO` *One Cycle*

- **Stop Execution**: the condition under which the ACT-R process has to be terminated. This is true when one of the following holds:
  - The number of cycles specified in the RUN- command has been carried out.
  - No production can be instantiated and no analogies can be formed (summarily explained on page 29).
  - The special action !stop! has been executed.
  - The GoalStack, which stores the goals that not yet have been attained, is empty (so there are no more goals).

  One Cycle consists of three phases:

; *Matching* UNTIL *Waited Enough*

; *Pick*

; *Fire*

- **Matching process**: The conditions of the production rules try to match simultaneously with the elements in declarative memory. The chunk in the focus of attention is the first that will have to be matched. The rest of this process is defined by the statements:
IF *Matching has resulted in an instantiation*
THEN *Add the instantiation to ConflictSet* ;
IF *ConflictSet is empty*
THEN *Analogy Learning*
This is carried out for every production.

  Waited Enough: The improvement you expect when you wait for new instantiations to be formed is computed. When it drops below a certain threshold, then one has Waited Enough.

  Pick process chooses the production instantiation from the ConflictSet that has the highest expected payoff.

  Fire: execute the action part of the chosen production. An action part consists of one or more actions that are executed in sequence. There are two kinds of actions, namely actions that create new chunks or modify already existing chunks and special actions that manipulate the focus of attention.

- **Analogy Learning**: mechanism to form new production rules that are applicable on the current element in the focus. It is based on a cause-effect relation inferred from information in both the declarative and procedural memory. The algorithm is described in short:

  ; *Determine the wmetype of the element that is in the focus of attention (that is the goalchunk)*

  ; *Search a different element in declarative memory that has the same wmetype*

  ; IF *Found*
  THEN BEGIN

  ; *Find out which production has effected the creation of this element*

  ; *Map the element found on the goalchunk*

  ; *Create a new production rule from the mapping*

  END
  ELSE *Failure to produce new production rules*

2.3.2 Knowledge representation

ACT-R has two memory systems that reflect the distinction between declarative (facts) and procedural (acts) knowledge. Each system has its own knowledge representation. The storage of knowledge does not guarantee that it can be retrieved later. This depends on the activation or strength parameters.

Working Memory element (wme)

Declarative knowledge is created either by encoding external events (just like a computer program sometimes has the possibility of interactive input) or through the actions of a production rule (illustrated in figure 2.1). The elements of declarative memory (Working Memory elements) represent the chunks of knowledge that one might have.

The abstract data structure of a wme is a record with two types of fields: the type field and the slot fields. Every wme has obligatorily one type field and zero or more slot fields. The type indicates the category to which the wme belongs and the slots contain the attributes that represent the specific characteristics of this particular wme.
This can be illustrated with the following wme, where the type slot (isa-slot) specifies that seven belongs to the category `number`:

```
(seven
isa   number
value 7)
```

With the statement: `(WMEType type slot-1 ... slot-n)` one can declare a wmetype. The first string after the keyword WMEType is the name of the type and the others are the names of the attributes.
The information about adding two numbers below ten can be represented by the wme-type addition-fact:

```
(WMEType addition-fact arg1 arg2 sum)
```

Wme's are in fact instantiations of wme-types. To create a new wme, you supply the wmetype and then specify the slot values. A wme can refer to another wme by storing the name of that wme in one of its attribute-slots.
In the addition-example a wme would look like this: `(fact2+5`

```
isa   addition-fact
arg1 two
arg2 five
sum   seven)
```

, in which `fact2+5` is the wme-identifier, the `isa`-slot denotes the wme-type and the other slots determine the values of the attributes. The value of the sum-attribute refers to the wme `seven`.

Activation Every wme has a corresponding activation value that depends on when it was used in the ACT-R process. The more often and/or the more recent a wme has been used, the higher its activation value will be.

The **baselevel activation** B_i of wme_i is an estimation of the logarithm of the odds that it will be used. It is calculated on the basis of past uses. When the wme is used t time units ago, the odds of being used now are at^{-d}, where a and d are constants and $d\epsilon[0,1]$ represents the decay rate. This reflects the argument that with the lapse of time the probability that a wme will be used again, decreases. If the wme is used more than once, for example n times, the odds should be added: $\sum_{j=1}^{n} at_j^{-d}$. The baselevel activation will be computed as follows:

$$B_i = log(\sum_{j=1}^{n} t_j^{-d}) + \log a \tag{2.1}$$

[formula's 4.1–4.3, Anderson, 1993].

When an **association** value has been defined among two wme's they can influence each other's activation values. The associations between wme's are established and adjusted during the execution of the ACT-R process and only when the flag for AssocativeLearning is turned on. This is somewhat similar to what happens between the cells of a neural network. An association value S_{ij} is supposed to estimate a log likelihood ratio measure of how much the presence of wme_i in the context increases the probability that wme_j is needed.

$$S_{ij} = \log(R_{ij}) \tag{2.2}$$

The calculation of R_{ij} requires the administration of the statistics of some events:

- $N_i \in [0,1,\ldots]$ represents the number of times that wme_i was needed.
- $C_j \in [0,1,\ldots]$ represents the number of times that wme_j has been in the context
- $F(N_i|C_j)$ is the number of times that wme_i was needed, given that wme_j is in the context

When a wme is created, it is assigned default values R_{ij}^{*}, that reflect the guesses as to what the likelihood ratio measure should be. The guess is made on the basis of the already existing connections between wme's (that is wme_i refers to wme_j, or vice versa). The association between wme_i and wme_j is in the implementation stored in a five element-list: (WME$_i$ S$_{ij}$ R$^{*}_{ij}$ F(N$_i$|C$_j$) Cycle),
where Cycle is the cycle in which S_{ij} was computed for the last time. R_{ij} is computed from these values with the following formula:

$$R_{ij} = \frac{R_{ij}^{*} + \frac{F(N_i|C_j)}{N_i/CycleNumber - CreationTimewme_i}}{1 + C_j} \tag{2.3}$$

N.B. This formula, differs from fromula (4.5) in "Rules of the Mind" [Anderson, 1993]. It is however not obvious whether this computation is approximative, or

just another way of calculating R_{ij}.

The total activation A_i of wme$_i$ can now be computed from the baselevel activation and the association values:

$$A_i = B_i + \sum_j W_j S_{ji} \tag{2.4}$$

where W_j indicates whether wme$_j$ is in the current goal context. That is, the activation of the current wme in the focus can be passed on to the slots that are defined with the `:activate` flag. If wme$_j$ is refered to in one of these slots and has been retrieved from declarative memory, then it is in the current goal context[1]. Formula 2.4 is formula (3.2) in the ACT-R theory [Anderson, 1993]. It predicts that memory performance will improve with practice and will deteriorate with the lapse of time.

Production Rules

In order to be able to carry out a task, it is necessary to know how you can reach the solution. The producedural knowledge that specifies this, is stored in procedural memory in the form of production rules. Usually several production rules have to be applied to reach the final solution. This set of production rules has to be supplied to the ACT-R system or can be created by a process of analogical matching.

Production rules have four significant features that capture the characteristics of ACT-R production systems:

- The **condition-action asymmetry** determines the direction of action: from condition to action. It is not possible to reverse this direction.
- **Production modularity** is the ability to add rules to and remove rules from the production system independent of any other production rule. This makes the system flexible.
- Production rules have an **abstract character**, so that they can be applied on several different occasions.
  We not only want a production rule to add two particular numbers, like 2 and 5, but we want it to produce the sum of any two arbitrary numbers below ten.
  Rules can be applied to any set of wme's that satisfies the condition part, provided that the goalchunk is the first chunk to be matched. The process is this general, because the production rules are schematic. One rule may have many instances, obtained by instantiating the variables occuring in

[1] In the ACT-R program version 2.1, the activation of the goal is automatically spread over the chunks that are in the context. If, for example there are three chunks in the context, they each receive $\frac{1}{3}$ of the goal activation

it. To instantiate a variable, it needs to be matched successfully with a wme in declarative memory.

- **Goal structuring**: Production rules have to match goals. There is, at any point in the ACT-R process, exactly one active goal, which is the wme in the focus of attention. The condition part of the production rules always specifies a goal condition: the wme at the head of the condition, which is the first that has to be matched.

A production rule has a name, a condition part and an action part and is specified as follows:
(p *name*
goalmatch
zero or more memory retrievals
==>
actions to be executed)
Each production rule has five parameters, from which the last four are used to evaluate the costs and payoff of its execution (they are used in conflict resolution, which will be discussed later):

- **Production strength** (S_p) is to a production as baselevel activation to a wme (compare equation (2.1)). It corresponds to how often and how recently a production has been used. The weaker the production strength, the slower its condition will be matched.

$$S_p = \log(\sum_{j=1}^{n} t_j^{-d}) + B \tag{2.5}$$

  where t_j is the j^{th} point in time that the production has executed and B and $d \in [0, 1]$ constants.

- a: the costs associated with performing this production, representing the cognitive effort.
- b^*: mean costs of further actions to reach the goal after this production.
- q: prior probability that the production will successfully execute.
- r^*: mean future probability that a successful execution will be followed by achieving the goal.

2.3.3 Specific subprocesses

Pattern Matching

The condition in a production is a list of variable wme's, which can in turn refer to (a list of) other variable wme's or have constant values in their attribute-slots. One can satisfy a condition by successfully matching wme's in declarative

memory with the variable wme's in the condition. To match a variable wme, one has to know what wmetype has to be searched for in declarative memory. Therefore the isa-slot, containing the wmetype, always has to be concretely specified and is never variable.

The time needed to match a wme depends on its level of activation and on the production strength of the production it appears in. The lower the level of activation A_i and production strength S_p of production p are, the longer it takes to match that wme. This can be expressed in a formula:

$$T_{ip} = Be^{-b(A_i+S_p)} \tag{2.6}$$

where T_{ip} expresses the time to match wme$_i$ in p, b and B (in implementation resp. LatencyFactor and LatencyExponent) are constants [formula 3.3, Anderson, 1993]. Inside a production matching is a sequential process, so the total matching time of a production will be the sum of the matching times of the individual wme's in its condition part.

The first wme that has to be matched is the wme at the head of the condition part (goalwme). When there is no wme in declarative memory that has a sufficient level of activation to be matched (in the implementation this is solely reserved for the wme in the focus) or the production strength is too weak, the condition will not be matched and consequently the production will not be executed.

In the pattern matching process, three auxiliary symbols are being used: =, $, - and >. The = as first symbol of an identifier indicates that this is a variable, the $-sign[2] represents an arbitrary number of elements, the - denotes a negation and the > separates the wme-identifier from its slots.
Example:

```
=fact>        matches: fact5+0        but not: fact2+5
isa addition-fact      isa addition-fact      isa addition fact
arg1 =num1             arg1 5                 arg1 2
arg2 =num2             arg2 0                 arg2 5
sum  =num1             sum  5                 sum  7
```

Conflict Resolution by Rational Analysis

All productions are matched in parallel, so at a point in time it is possible that more than one production has been matched and is ready to execute its actions. However, only one will be privileged to do so. This means that a choice has to be made between the different candidates. The whole of matching, chosing one production instantiation and subsequently executing it, is called a *cycle*.

Not all productions match equally fast, but this does not imply that the first candidate for execution is imperatively the best. Therefore a ConflictSet, which

[2]The $, which allows the use of lists in the slots of wme's, is abolished in version 2.1 of ACT-R

contains all production instantiations formed so far is kept. The candidates in the ConflictSet are compared on the basis of some criteria. However, we do want the (relatively) best production to execute as soon as possible: time and quality are concurrent aims in the process of selecting an appropriate instantiation. This is an example of what Anderson [1993] calls Rational Analysis: every time a production has successfully been matched, an evaluation value is being assigned to it. The evaluation value represents the expected payoff of the instantiation, which can be calculated by $PG - C$, where P is the probability that this production will lead to the goal, G the value that corresponds to the importance of reaching the goal and C the expected costs (= time) of achieving the goal via this production.
In the implementation P and C are computed by resp. the functions EstimateP and EstimateC. They use the a, b^*, q and r^* parameters of the production, the effort spent so far (in the implementation this is the variable *TotalEffort*) and the current distance from the goal.
The expected improvement I, that is calculated when one waits for a new production to be matched, is based on the best evaluation found so far and the maximum payoff G. In the implementation this is calculated by the function I[n]. When I drops below a certain threshold, the matching process is aborted and the instantiation with the highest evaluation value in the ConflictSet will be executed.

Actions in productions and the Goal-stacking mechanism

In the action part of productions two types of actions can occur. One changes already existing wme's (they will have to have been matched in the condition part) or adds new wme's to declarative memory. The other type of action is the *special action*, which is always written between exclamation marks.
ACT-R has a goal stack that keeps track of the wme's (that were) in focus. The top of the stack contains the wme that is presently in focus. Most special actions are used to manipulate the focus.

To perform a task it often has to be divided into smaller subtasks. The goal-stacking mechanism changes the focus towards the subgoals that correspond with the subtasks. On the stack (sub)goals are being stored while working on another subgoal. The basis of the mechanism is the focus: all productions try to match the wme in the focus. A former (sub)goal is re- enters the focus, when a subgoal has been popped from the stack. This way the production system controls through subgoals the flow of actions towards the endgoal.

Some characteristic special actions are:

```
!push!  subgoal          : put subgoal on the stack
!pop!                    : restates the former focus
!output!  ("string")     : print the string string on the screen
```

For a complete catalogue of special actions, the reader is referred to the ACT-R manual, section 4.

Learning mechanisms

Each of the two memory systems in ACT-R has its own learning mechanisms. In declarative memory the baselevel activation and the associations between wme's is being adjusted through management of parameters discussed before. In procedural memory the parameters of production rules are also kept. There is however another learning mechanism in procedural memory. It is called Analogy Learning. Through this mechanism new production rules can be formed on the basis of a history of productions that have created new wme's which now serve as examples.

2.4 Implementation specifics

The ACT-R program was originally written in MacIntosh Common LISP version 2.0. After some small changes by Niels Taatgen it was fit to run on Allegro Common LISP version 4.2 that is available on the UNIX system of the TCW-network. The ACT-R program comprises about 180kB of LISP code.

The processes mentioned in the previous section naturally return in the ACT-R program, but not always exactly the way it was set out in the theory. In the ACT-R theory analogy learning competes with the normal flow of control, but in the implementation it will only start when no productions can be executed. Sometimes the computation of formula is approximative. When, for example, time is incorporated, cycle numbers are used instead of the real run latency, which is measured in seconds[3]. When the option OptimizedLearning is *on* the notion of time is reduced to "how often" compared to the age of the chunk instead of a list of cycle numbers.
Some theoretical entities of ACT-R have a different name in the implementation. What Anderson calls declarative memory is in the implementation being referred to as Working Memory. This conflicts with the psychological convention that Working Memory is only that part of the memory that contains the highly activated elements. The theoretical concept chunk, is in the implementation a Working Memory element (wme).

2.4.1 Interface

When you start the LISP program with the command `lisp42`, an emacs window will be opened. The window is a user-interface to the LISP interpreter. From this window the ACT-R program can be loaded in the interpreter with the command `:ld actload`. Now procedural and declarative knowledge can be fed into the cognitive architecture provided by ACT-R, and one can work with the architecture:

[3] In version 2.1 of ACT-R (July 1995) this is fixed: real time, measured in seconds, is used and analogy competes with the other processes.

- With about thirty commands, the production system and declarative memory can be controled and manipulated, and the ACT-R process can be forced to start or continue its course of action. The commands and their use are catalogued in Section 1 of the ACT-R User's Manual.
- With the command (`menu`), a menu will be accessed. With this menu one can set and adjust ACT-R options and parameters. Every option and parameter has a corresponding LISP name and can therefore also be set using the LISP `setq` or `setf` command. Some of the toggleable options offer the possibility to trace the course of action of the ACT-R process. Each option focuses on a different aspect. The options are described in Section 7 of the ACT-R User Manual.

2.4.2 Inputfile

An inputfile, specifying the task that ACT-R has to perform, can be loaded from the emacs window into the ACT-R system with the LISP command `:ld` *filename*. In this file the production system has to be specified, along with the (for this task) specific wmetypes and wme's in declarative memory. The addition task in appendix A is a simple example to gain insight into the way a simple inputfile needs to be specified.
An inputfile can have six different components.

1. [Optional] LISP commands that set the ACT-R options and/or parameters.
2. [Optional] Auxiliary LISP functions and variables can be defined to simplify the code in the production rules.
3. Declaration of wmetypes, specific for this (sort of) task.
4. Contents of procedural memory: a production system that has to perform the task. Optionally parameters that estimate the costs and the probability of success can be specified for each production rule.
5. Initialisation of declarative memory with wme's of the earlier specified wmetypes.
6. Initially value of the focus has to be the wme that represents the goal of the task. The focus of attention is instantiated with the ACT-R command (`WMFocus` *goal*), where *goal* is the goal of the task.

The order between the components is arbitrary, as long as the specifications of wmetypes and auxiliary LISP functions preceed their use in the production rules and declarative memory.

2.5 LISP Code

The LISP code of the ACT-R program is divided in several files in a logic manner. The files are not organized in LISP packages, but they are loaded one after another in such an order that no conflicts arise. That is, the specifications of functions, variables, etc. are loaded before the files in which they are used. This order is specified in the file *actload.lisp*.

A description of the files that form the ACT-R program:

Analogy.lisp The analogy algorithm is implemented by a number of functions, that are helped by some functions that operate on Working Memory. The main analogy function is `solve-by-analogy`.

Commands.lisp These macro's and functions implement the ACT-R commands with which one can manipulate and control the cognitive architecture, listed in Section 1 of the User's Manual. The only exeptions are the command `PMatches`, which is implemented in the file PrettyPrinter.lisp and the macro `WMEs` *(type)* which is not listed as an ACT-R command.

Definitions.lisp This file is subdivided in six sections that each have a specialized application:

- Toggleable options: The variables that represent the ACT-R options and parameters are declared and initialized with their default value.
- Recognize-Act cycle definitions: Data structures like `*Instantiation*` and `*GoalStackFrame*` are defined and variables like `*TotalEffort*`, `*ConflictSet*`, `*CycleNumber*` and `*GoalStack*` are declared.
- Production definitions: The data structure `Prod` is defined. It stores information of a production rule, like its name, creation time and parameters. Several variables concerning productions are declared. Variables `*Productions*` and `*DisabledProductions*` respectively contain the list of currently enabled and disabled productions.
- Working Memory definitions: The initial size of the WM is declared and `*WorkingMemory*` is a hash table that contains WMNodes. Some Working Memory concepts are represented by data structures: `WMNode` stores information of one wme, `WMETypeDef` stores type information and `Link` stores the information of an association between two WMNodes. The variable `*WMFocus*` (default value `Goal`) is the root WMNode for all left-hand side matches.

- Instantiation Chooser definitions: constants (like *DefaultG*) and variables (like *G* and *Z[n-1]*), are used when the ACT-R process chooses the best production instantiation. The variable *LatencyHook* represents the latency of one cycle, whereas *run-latency* represents the simulated latency of one run.
- Discrimination Net definitions: Variables that are used in the WM hash table, which is also reffered to as the WM net.

Est.lisp Past effort and the current distance to the goal are used in functions that compute the costs (EstimateC) and the probability of success (EstimateP).

InstantiationChooser.lisp Implements the rational analysis mechanisms of production selection. When the left-hand side of a production is matched, an instatiation has been formed. The functions C[i] and P[i] respectively compute the costs and probability of success of an instantiation. These are used to compute its ExpectedGain. The function CalcInstantiationActivation returns the activation of an instantiation. It is computed on the basis of the sum of the activation of the wme's that have matched the variables on the left-hand side, plus the strength of the production (and some noise). Conflict resolution is enabled when the option *EnableRationalanalysis* is non-nil, otherwise an instatiation is picked at random.
The function ChooseInstantiation chooses an instantiation from the *ConflictSet*. The function I[n] returns the expected improvement when one waits for a new instantiation to be formed. The variable *LatencyHook* is bound to the function CutoffMatchingLatency, which the latency of the selecting process, that is the time when rational analysis cuts of further evaluation.

Learning.lisp Functions that implement the learning mechanisms [Anderson, 1993, Chapter 4]. Function exact-learning-3-5-6 adjusts the References, NtimesInContext and NtimesNeeded parameters of the wme's that have matched and the References parameter of the production rule. From the last mentioned parameter the strength of a production rule is computed in function GetStrength. Function exact-learning-7-8 adjusts the production parameters q, a, r* and b* on the basis of equqtions 4.7 and 4.8.

Main.lisp This file contains some top-level functions.
The function AddToConflictSet adds a new instantiation at an appropriate point in *ConlictSet*, which is sorted by expected gain. DoOneCycle performs one cycle, that is it executes the matching process, subsequently picks an instantiation, executes it and increments variables as *TotalEffort* and *CycleNumber*. Fire executes the actions in the right-hand side of the production instantiation. SetWMFocus set the focus and updates the *ActivationSources*,

which is the list of wme's out of which flows activation. PushGoal and PopGoal implement the !push! and !pop! actions.

MatchCoder.lisp Code for pattern matching needed by the WM net and production compilers.

NetCompiler.lisp Assembles and compiles a discrimination net of productions.

PrettyPrinter.lisp Four functions that legibly print some ACT-R entities: PPInstantiation, PPrintValue, PPrintWM and PMatches. The last is an ACT-R command that prints all productions that match against the current state of Working Memory.

ProductionCompiler.lisp This file provides code that compiles production declarations into LISP code. Productions are compiled when they are defined. This makes loading slow, but execution fast. The macro p operates at the top-level and separates production declarations in left- and right-hand sides, which are in turn handled by ComposeLHS, ComposeLHSLambda and ComposeRHS, ComposeRHSLambda.

Stat-Functions.lisp Some general purpose statistical functions, like Sum, Mean, Variance and StDev used by the InstantiationChooser, or not at all.

Syntax.lisp Some functions that return a boolean, which indicates if the argument is of the specified type, for instance Variabelep, Productionp, !keyword!p and other functions that have to do with syntactic matters.

WorkingMemory.lisp "Low-level" code to create WMNodes, access and set slot values.
The wme's are stored in a hash table. The macro WMHash returns the WMNode that corresponds to the wme that was supplied in its argument.

- Association Link stuff: Code to create (new-link), compute (ComputeIA) and manipulate (IA, ClearIA, SetIA, IncIA, and DecIA) association strength between wme's.
- Wme construction functions: In order to create (CreateWME), add (AddWME) and delete (DeleteWME) wme's from Working Memory. They are assisted by some low-level functions, like for instance BuildConstructor and GetSlotValue.

- Auxiliary command functions: used by ACT-R commands relevent to WM, e.g. `ResetWMFocus` and `ResetWM`.

- WM Activation functions: Activation can flow from a wme to its slots. Every wmetype stores a list that contains the wme's that have to be activated when the parent is. The variable `*ActivationSources*`, which is the list of current active wme's, is declared. When a wme is activated, the wme's in its slots are added to the *ActivationSources* by `activate-wme`. With `deactivate-wme` these wme's are deleted from *ActivationSources*. The functions are assisted by `AddActivationSource` and `RemoveActivationSource`. The function `GetActivation` computes the activation of a wme from the sources on the basis of its associations (equation 4.5) and baselevel activation (equation 4.3) [Anderson, 1993, Chapter 4]. `act->lat` and `lat->act` compute the latency from the activation and vice versa.

- Undo-production-firing function `ComplementSaveStateChanges` sets StateChanges* to nil and *SaveStateChanges* to its complement.

- Stuff to maintain WME connectivity: Every time one wme mentions another, a bi-directionally connection is established between them. `NoteWMConnections` is used in the function WMCreate. When a wme has just been added and doesn't have any slots yet, this function goes through all of the WM and adds a connection every time the new wme is mentionned by an already existing wme. The function `UpdateWMConnections` is used every time a slot value is changed. `AddConnection` and `RemoveConnection` are low-level functions to add and remove connections, used in the previous described functions.

2.6 Glossary

Chunk/wme element of declarative memory that represents a fact or an object

Cognitive architecture theory about the structure of human cognition, with the object to provide a complete specification.

Cycle The whole process of pattern matching,piking a production instantiation and subsequently executing it

Declarative memory Memory that stores factual knowledge

Focus Chunk with the highest activation value in declarative memory and the first that will have to be matched in a production rule

Instantiation Every way a production rule has been matched

Latency the time it takes to complete something, like for instance a run: run-latency

Production rule A condition-action pair; the action is executed when the condition is satisfied

Production system A set of production rules

Wme Working Memory element, see also chunk

Wmetype The type of a wme, always stored in the isa-slot

Chapter 3

Rehearsal in ACT-R

3.1 Reasons

Modelling rehearsal in ACT-R has two goals. The most obvious one is the improvement of ACT-R as a cognitive architecture. The second goal is to set rehearsal in the wider context of a cognitive system. We discuss these goals now in inverted order.

Rehearsal in a wider context

A lot of researche has been made into rehearsal: a lot of experiments have been conducted, conclusions have been drawn and models have been made. However, research has been limited to the field of psychology of memory. Models, not to mention aspects, of rehearsal have not been integrated satisfactorily into a larger system, whereas this seems a logical step to make. Rehearsal is not an independent phenomenon in human cognition. On the contrary, it supports the functioning of the memory. Implementation of rehearsal in the cognitive architecture ACT-R has the added value of putting rehearsal at the service of memory performance. This point of view might answer some questions and raise interesting new research topics.

Baddeley's theoretical Working Memory model in particular, incorporates the Phonological Loop as rehearsal subsystem in memory. However, the Central Executive, which supervises the Phonological Loop and is supposed to coordinate and execute cognitive processes, is the weakest part of his theory. It is unclear what representation it uses and when and why it puts something in the Phonological Loop. Actually the Central Executive is a substitute for the "rest" of cognition, for everything that is not directly related to processes inside the subsystems is moved to the Central Executive. A way out of this vagueness is to lift the Phonological Loop out of Baddeley's Working Memory model and tie it to a ready-made cognitive architecture like ACT-R. In fact ACT-R can be seen

as the Central Executive. We then can not only predict facts about working memory, but also study the interaction with the rest of cognition, like long term memory and problem solving.

ACT-R improved

There are some advantages that an implementation of rehearsal has for ACT-R as a whole.
As ACT-R is a cognitive architecture, it aims at completeness. In practice, rehearsal has proved to be an essential component in retaining intermediate solutions when one carries out cognitive tasks. The implementation of rehearsal is useful for ACT-R, because this way ACT-R approximates human cognition more closely.
There are some architectural choices in the current ACT-R version that are cognitively implausible. Anderson recommends to limit the number of slots to three or four a wme, but there is no such limit built in the implementation. Apart from this you can store lists of unlimited length in these slots, which makes the matching process far too powerful compaired to normal human cognition. If we choose not use this capability, we need another way to represent serial information. A long-term memory solution is to link chunks in lists, something that occurs naturally in the Phonological Loop. These lists are more difficult to match and this fits much better the human capabilities for lists.
The activation value of wme's is controlled by autonomous mechanisms and can not be manipulated directly. When you deliberately want to raise the activation value of a wme, you have to access it repeatedly: a function for which the Phonological Loop is ideally suited. The Phonological Loop can be used to learn both base-level and association strengths between wme's.
To summarize, the incorporation of the Phonological Loop in ACT-R has at least two advantages for the architecture: it gives a cognitively plausible mechanism to represent serial information, and it allows deliberate control over activation levels and association strengths. The disadvantage is that adding more elements to the system weakens the theory, as it creates more ways to explain something. The more possibilities one has to account for something, the less falsifiable a theory is. However, we believe that the advantages dominate.

3.2 Desired properties of the implementation

There are some properties that the model ideally should have.

1. Theoretical constraints that are inferred from the expectations described above.

    (a) The rehearsal process is a deliberate act. That is, the Phonological Loop is part of the problem solving strategy and rehearsal actions

have to be planned and compete with actions that bring us closer to the goal. When the rehearsal actions give too much way to goal-directed actions, the activation of the elements in the Loop will decay to a point that they will become irretrievable.

(b) The Phonological Loop is based on a verbal code and can therefore not be part of declarative memory, which means that we must create a separate store. The elements that are (temporarily) stored in the Loop must have a phonological representation and the length of the representation determines the amount of ressources needed in the Loop.

(c) The distinction between maintenance rehearsal and elaborative rehearsal that we came across in chapter 1, should both be possible in the Phonological Loop.

(d) The goal of rehearsal is to keep chunks active. Each rehearsal will have to revive the activation level of the chunk, and this can be achieved by increasing the base-level activation of the element. Because of the serial nature of the rehearsal process, association strengths between successive elements in the Loop could also be built up.

(e) To control the rehearsal process we need a way to manipulate the Loop. The basic manipulations are putting something in the Loop and reiterate the contents of the Loop.

2. Empircal predictions: in memory experiments some phenomena occur consistently. In a plausible model (most of) these phenomena must also figure. The phenomena in question are:

(a) Immediate recall will deteriorate with increasing list length. This is illustrated by figure 3.1 [Burgess & Hitch, 1992]. The limit on

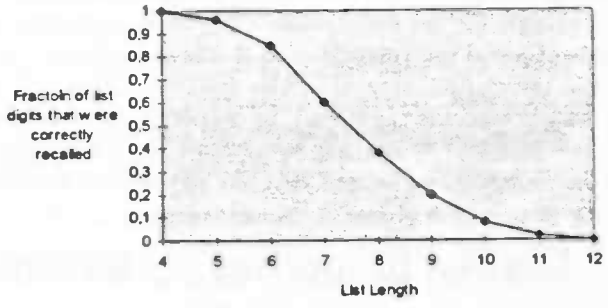


Figure 3.1: Fraction of list of digits that is correctly recalled versus list length

memory span of seven items that Miller [1956] discovered, is relatively crude. It is clear that up to five digits are almost perfectly recalled and after that the correct proportion drops dramatically.

(b) Phonological similarity, irrelevant speech, increasing wordlength and articulatory suppression (see also chapter 1, page 13 and further) deteriorate the performance of short term memory.

(c) The serial position effect: The probability to recall an item depends upon its position in the list. This is also known as the primacy/recency effect. (see also chapter 1, page 8)

(d) Errors that occur fall into two categories:

Order error occurs when an item of the list, is recalled at the wrong place. In most of the cases two items have swapped places.

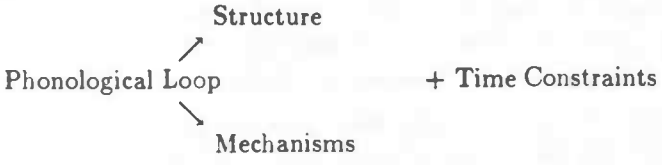
Item errors occur when an item is forgotten or reported when it was not in the presented list.

A lot of errors are caused by phonemic confusion. The majority of the errors that occur are order errors.

Chapter 4

Rehearsal implemented

The implementation of the Phonological Loop can roughly be broken into three parts.



4.1 Preview of the changes in the architecture

1. Addition of a new variable `*PLoop*` to ACT-R and a new datastructure `PLelt`. The datastructure declares the format of the elements that enter the loop. The variabele represents the Phonological Loop and contains a list that consists of PLelt-elements.

2. To be able to rehearse, new functions have been added. The two most important are `add_to_loop` and `iteration`, which respectively adds an element to the loop and iterates one element in the loop. These functions incorporate the time constraints.

3. As the top level of the rehearsal process is deliberately planned, special, standard production rules, working memory definitions and a rehearsal subgoal are added in order to implement the competition between rehearsal and the completion of the task.

4. An element that is pushed into the loop, refers to an inter-chunk, which in turn refers to the wme that we want to rehearse.

5. After having rehearsed an element in the Phonological Loop, the rehearse process allows for elaborative rehearsal and associative learning.

4.2 Structure

4.2.1 WME's

When you want to use rehearsal, you have to be able to deduce phonological characteristics from the wme's that you want to put into the Loop. It means that those wme's have to be *pronounceable*. Consequently, unpronounceable concepts can not enter the loop.

As finding and incorporating a phonological representation for concepts in memory is a problem in itself and we do not want to make things any more difficult than they already are, we decided not to search and implement a phonological representation for wmes. The phonological aspect will be reduced to phonological length, measured in seconds. This simplification affects performance of the Phonological Loop and will be discussed in Chapter 5, on page 49.

When we consider the external behaviour of the Phonological Loop it is of no importance whether the phonological length is stored in every wme in declarative memory, or can be deduced or looked up. I have chosen the most simple solution: for every wme that enters the loop phonological length (> 0) has to be specified. For reasons of simplicity, this implementation demands the phonological length to be stored in a standard slot, that is called *duration*. When the phonological length is unknown or irrelevant, no duration is present in the WMEType. As long as it is not used in a phonological context, it is not imperative to specify this slot for every wme. An arbitrary wme, let's call it wme_i, that does specify the phonological length, should be declared in the inputfile (see Chapter 2, page 30) as follows:

```
(WMEType wmei $ duration $),
```

in which the $-signs denote one or more other slots.

4.2.2 Phonological Loop

The Phonological Loop consists of a list of elements (stored in the variable *PLoop*) and a pointer. The pointer, contained in the variable `*PPointer`, is implemented by means of a list that contains the "rest" of the *PLoop*, that is, the part of the loop that still needs to be rehearsed. The element that is the next to be iterated is the element in head of *PPointer*.

The elements in the Phonological Loop each contain a reference to an inter-wme and phonological information: a slot with phonological length and a code-slot. The latter is not very useful right now, but might be used later to incorporate a phonological code. At the time being, it just stores the identifier of the wme that will be rehearsed. The structure of the elements in the Phonological Loop in LISP is as follows:

```
(defstruct PLelt
  (PCode nil)
  (PLength nil)
  (Iwme nil)  )
```

PCode stands for the phonological code, PLength contains the phonological length and Iwme the reference to the inter-chunk.

4.2.3 Inter-chunk

When an element is added to the Phonological Loop, an inter-chunk is created and added to declarative memory. This inter-chunk is the link between the wme in declarative memory that we want to rehearse and the element in the loop, representing it.

If we only wanted to model maintenance rehearsal, it would easily be implemented by increasing the baselevel activation, each time a wme is rehearsed. However, we also want to handle elaborative rehearsal and be able to learn association stengths between the wme's that are rehearsed, so a simple increment of baselevel activation does not satify this demand. The purpose of this intermediate layer of wme's thus appears to be twofold:

- Association learning takes place between the wme in the focus and the wme's in the context (that is, the wme's that the goalwme refers to in its slots). However, direct associations between the wme's that represent the rehearsed concepts does not seem plausible. For example, it does not seem realistic to assume that memorizing a telephone number is achieved by strenghtening the associations between the digits. A digit will appear in more than one telephone number and might appear more than once in a telephone number. There would be an enormous amount of interference, that would dramatically disrupt memory performance on telephone numbers. Learning an intermediate representation offers the solution to this problem: associations are learned between inter-chunks. One could talk about a hierarchy of knowledge: a cluster of strongly connected wme's represent through their associations a piece of knowledge that consists of more than one concept, like a telephone number.

- Elaborative rehearsal is implemented by pushing the inter-chunk on the goalstack, just after it has been rehearsed. This allows production rules to match and subsequently manipulate it. Elaborative rehearsal production rules compete with a production rule that effects in returning to the normal rehearsal process.

The inter-chunks have a standard WMEType, that consists of two slots:
- The concept-slot, that refers to the wme that entered the loop.
- The supergoal-slot, that indicates from which (sub)goal the wme has entered the loop.

```
(WMEType interchunk (concept :activate) supergoal)
```

The supergoal indicates a sort of context in which the inter-chunk has been created. This can prove to be useful information in the case of elaborative rehearsal.

4.3 Mechanisms

Rehearsal is a deliberate act. To use it, we need some functions that perform standard actions, as adding and rehearsing an element, on the Phonological Loop. The functions can be used in an eval-statement in the action part of the production rules. This looks as follows:

`!eval!` *expression*,

in which *expression* is a piece of LISP-code, that will be evaluated by the LISP interpreter. In this LISP code, one can use the functions that will be discussed in the following subsections. See the appendix with the inputfile for rehearsal for more detailed examples.

4.3.1 Add an element to the Loop

The function `add_to_loop` implements the adding of a wme to the Phonological Loop. The argument for the function is a pair that consists of a newly created inter-chunk and the wme that has to be added.
Precondition: in the production rule that calls the add_to_loop function, the interchunk has to be created before the function is called.
Adding an element to the loop, comprises:

- Check if you are allowed to use the Phonological Loop.
  IF you are not allowed to use it, THEN you exit ACT-R with an error-message, ELSE continue.
- IF the wme that is supposed to enter the loop has no phonological lenght, THEN you exit ACT-R with an appropriate error-message,
  ELSE continue.
- Create the PLelt-element that is going to enter the loop.
- Assign values to the concept- and supergoal-slots of the inter-chunk.
- Assign the appropriate values to the attributes of the PLelt-element.
- Raise the baselevel activation value of the wme that is about to be added to the Phonological Loop.

The precondition can be guaranteed, with the use of the function `test_loop` in the condition part of the production rule. This function checks if the loop may be used and still has elements that need to be rehearsed in this re-iteration cycle.

The variable `*ActivationThreshold*` contains the threshold value that determines whether a wme is active enough to be retrieved from declarative memory.

Latency Adjustment

The function `LatencyAdjustment` adapts the latency of a rehearsal-production-cycle. If we are engaged in rehearsing, it sets the production-latency (stored in the variable `*LatencyHook*` to the maximum of the normal latency of the rehearse production and the phonological length of the item that was last rehearsed. A phonological length of 0 indicates that there was no iteration in this production cycle.

4.3.3 Rehearsal production rules

To implement the rehearsal process some extra production rules and a rehearsal-subgoal will be added.

```
(WMEType rehearsal)
```

A reiteration-cycle takes the following form: if the decision to rehearse has been taken, a rehearsal-subgoal is created and pushed on the stack. Then one by one the elements in the Phonological Loop are re-iterated (the wme's that correspond to the elements are re-activated) or deleted from the loop, if the wme's could not be retrieved from memory. To allow for elaborative rehearsal, the corresponding inter-chunk is temporarily put in the focus of attention after each successful re-iteration. As a result two sorts of productions that match these inter-chunks may fire: rules that elaborate the inter-chunk and a rule that results in returning to the normal rehearsal process. If the elaborative production rules take too much time, this might cause the elements in the Phonological Loop to become irretrievable and thus cause the rehearsal process to break down.

The production rule `reh_step` implements the iteration of the next element in the Phonological Loop. The condition part will not be satisfied, if the function test_loop evaluates to nil. This way it can be guaranteed that the loop will only be rehearsed once during a rehearsal-subgoal.
In the action part the inter-chunk that corresponds to the element that has just been re-iterated is pushed on the goalstack.

```
(p reh_step
   =goal>
      isa rehearsal
```

```
   !eval!  (test_loop)
==>
   !bind!  =interc (!eval!  (iteration))
   !eval!  (if =interc
             (PushGoal =interc))
)
```

The production rule call_and_exit implements the return from elaborative rehearsal to the normal rehearsal process. The inter-chunk is simply popped from the stack and then the rehearsal subgoal is being restored. This production competes with specific productions for elaborative rehearsal. To make sure this production will only fire when there are no others, it gets a low r-value.

```
(p call_and_exit
   =goal>
      isa interchunk
      concept =wme
      supergoal =sg
==>
   !output!  ("Ehr...  ~A ~%" =wme)
   !pop!  )
```

The production rule reh_done effects in the exit from the re-iteration process. It pops the rehearsal-subgoal from the stack, if every element in the loop proves to have been rehearsed.

```
(p reh_done
   =goal>
      isa rehearsal
   !eval!  (not (test_loop))
==>
   !pop!
   !delete!  =goal )
```

Time constraints

The notion of time has been taken into account in three different ways:
- The loop can store elements up to a maximum capacity. This limitation is accomplished in the function add_to_loop, where the current contents is checked and compared to the length of the element that will be added. The variable *TimeContentsPLoop* contains the value of the maximum capacity. It can be manipulated, but it is supposed to be 1.5–2 seconds (see also page 7).
- If the activation value of a wme is too low, it can not be retrieved from declarative memory. Consequently, wme's that engage in rehearsal might prove to be irretrievable and are deleted from the loop. To achieve this, a strict threshold will determine whether the activation of a wme is sufficiently high. The variable *ActivationThreshold* contains the value of the threshold. There

is no hard and fast rule that helps determining the appropriate value. One has to keep in mind that a high threshold will discard elements from the loop very quickly. On the other side, a low threshold will be very reluctant to delete elements from the loop. One just has to experiment with different values to fix the best.
- The latency of a reheasal production rule has to be at least the phonological duration of the element that is re-iterated. The function LatencyAdjustment is responsible for satisfying this condition (see page 45).

Competition between rehearsal and completion

If you want rehearsal to compete with production rules that work towards the completion of the task, you will have to model this in the task itself. At points where you want rehearsal to be able to occur, you will have to have two competing production rules: one that enters the rehearsal process and the other that results in continuing the task.

4.4 Associative learning

Inter-chunks can also be used for associative learning. If we want associations between the interchunks of the elements in the loop to be built up during rehearsal, we have to explicitly program this in the inputfile. Associative learning occurs exclusively between the wme in the focus and those in the context. For associations between inter-chunks to be formed, we would have to place one of them in the focus (as happens when we push it on the goalstack, just after an iteration has successfully been done), and have the other(s) in the context. A special production rule could account for this. Unfortunately we have not had enough time to look into this aspect more deeply, so no tests or examples are ready for explanation and illustration.

Chapter 5

Evaluation

Now that rehearsal is incorporated, does it satisfy the constraints that we have outlined in section 3.2?

5.1 Theoretical Constraints

As for the theoretical constraints: the implementation has been made to comply to these demands.

1. (a) The rehearsal process is deliberate, because it has to be planned explicitly in the flow of actions.
   (b) The Phonological Loop is not part of declarative memory and uses a different representation for its elements. This store depends largely on the phonological length of the wme's and not, as in the ideal case, on a complete phonological representation. This has some important effects on the performance of the process. This will be discussed later on page 49, because it merely affects performance on the level of the empirical constraints.
   (c) Maintenance rehearsal can be regarded as raising the baselevel activation of a wme. Elaborative rehearsal occurs when a special, for this purpose designed, production rule matches and subsequently manipulates the inter-chunk.
   (d) Wme's that engage in the rehearsal process will be re-activated, because a successful re-iteration action will raise its baselevel activation. Association strengths can also be established and adjusted, but this needs to be modelled within a task (see section 4.4).
   (e) The basic manipulations of the loop are adding an element to the loop and re-iteration an element in the loop. These functions are sufficient for the things we want to do with the loop.

5.2 Empirical predictions

Some empirical predictions will have to come out in testing the performance of the ACT-R process. In some cases however, it can be argued that ACT-R has the desired behaviour.

2 (a) It is clear that immediate recall will deteriorate with increasing list length, when one has implemented a loop that has a maximum capacity. Experiments with a list of words that were rehearsed were conducted (adaption from the "twenty words" experiment that will be discussed later on page 50). The words have equal phonological length and the list lengths varied from 2 to 12 elements. The baselevel activation is used as indication for the probability of recall of a wme (the higher its activation, the better it will be recalled). The mean value of the baselevel activations of the elements in the recall-list is taken over 100 runs. An *ActivationThreshold* of -1.0 and a *TimeContentsPLoop* if 2 seconds were used. The results are shown in figure 5.1. Altough the shape of the curve is not similar to the one in figure 3.1, the probability of recall will have some normal distribution around -1, which will return the S-shape.

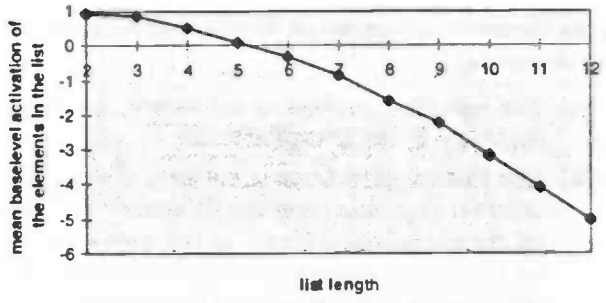


Figure 5.1: Probability of recall in of rehearsed wme's in ACT-R vs. list length

(b) - The **phonological similarity** effect will not occur, because the phonological code has been reduced to phonological length. With only phonological length, one can not deduce whether wme's are similar in pronounciation. This is the major weak spot of this implementation of rehearsal, for the basis of Baddeley's Phonological Loop is neglected: the elements that reside in the Phonological Loop depend upon a phonological code. From this code several characteristics, like phonological length and phonological similarity, can be infered. When a satisfactory phonological representation for wme's is found, the implementation can be extended to incorporate all features.

- **Irrelevant speech** is assumed to gain access to the phonological store of the loop and this way disrupting the contents of the store. When in the implemented rehearsal process irrelevant items are fed into the loop, clearly memory performance on the elements that were fed into the loop deliberately will deteriorate.

- **Word length**: It may be clear that in a loop with 6 elements of small phonological length, elements will more easily be recalled, than in a loop with 6 phonologically long elements.

- **Articulatory suppression** can be considered as being unable to use the rehearsal process. If one can not rehearse, elements will not be re-activated and more likely to become irretrievable.

(c) To examine the serial position effect, the so-called "twenty words" free-recall experiment was modelled. The inputfile of this test is included in an appendix. Twenty words are presented at a presentation rate of one word every 2 seconds and the subject (in our case ACT-R) is supposed to recall as much as possible in any order it likes. As twenty words exeed the immediate memory span, the subject tries to make the best of it by rehearsing as much as possible words in the presentation intervals. The simulation has been run 100 times. Figure 5.2 shows the results. The left Y-axis plots the mean activation levels of the elements over 100 runs on the serial position of the elements. The Y-axis at the right plots the probability of recall, drawn from real experimental data on serial position. [1]

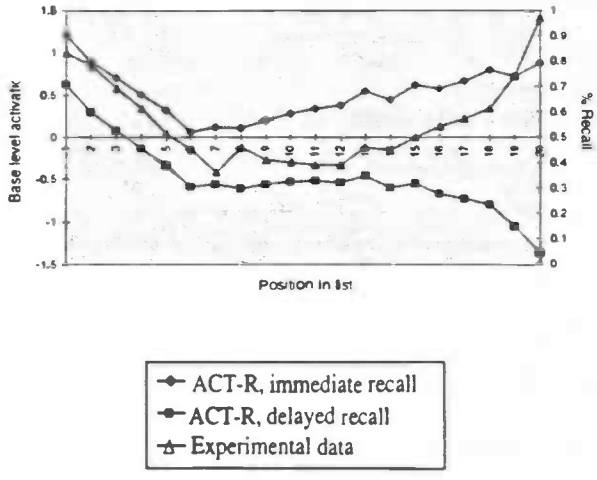


Figure 5.2: Probability of recall vs. serial position

[1]The ACT-R runs are executed on the latest version of ACT-R (July 1995)

The middle line with triangle points is the real experimental data on the twenty words test. The upper line, the one with diamond points, is the data from the ACT-R performances in immediate recall. A distinct primacy and recency is noted. By contrast with the real data, the difference between the middle section and the primacy and recency sections is not large, yet clearly visible. The lower line with the square points are the activation values of the elements, after a break of 60 seconds. Here a marked absence of the recency effect is noted. This appears to be a general observation in delayed recall experiments [Sato, 1990]. This data shows that the ACT-R simulation approximates behaviour on rehearsal very well.

(d) As noted before, phonological similarity does not influence the performance of ACT-R. Therefore no errors due to phonemic confusion will occur. Item errors can occur in the simulation when a wme has become irretrievable and is discarded from the loop, because this wme will also be irretrievable at the moment of recall.
Pure order errors are not examined yet, as we have not looked detailed into an example of learning association strengths.

Chapter 6

Conclusion

The implementation of rehearsal in ACT-R and the results obtained with the use of maintenance rehearsal are quite satisfactory. In experiments a marked primacy and recency effect is reproduced. Experimental results also show the absence of the recency effect in delayed recall, whereas previously we did not know of the existance of this phenomenon.

The facilities that are now available to model an simulate rehearsal in the performance of a task are quite elaborate. I've experienced, that—with enough ingenuity—they offer a solid tool to mix rehearsal in the flow of actions that perform the task, just the way you want it.

The limitation of this implementation of rehearsal is, above all, the lack of a decent phonological representation that could account for phenomena that are related to phonological similarity.

The final success of the incorporation of rehearsal in ACT-R still depends on the results that will be obtained with elaborative rehearsal and associative learning. These are aspects in the rehearsal process that have not been modeled and tested in this project yet, but will be in the future.

Appendix A

Simple example of an inputfile : Addition

```
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                                ;Parameters
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(clearall)

(setf *EnableRationalAnalysis* nil
UseDefaultIA* nil
EnableIndirectMatches* t)


    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;Working Memory types for this problem
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Addition-problem is a array of numbers.
(WMEType addition-problem object)

;; Numberarray consists of a list of columns, in left-to-right order
(WMEType numberarray columns)

;; A column has a top row, a bottom row, an answer row
;; and a plane to note the carry.
(WMEType column toprow bottomrow answerrow note)

;; A number* has an integer value
(WMEType number* value)
```

```
;; An addition-fact relates two numbers to their sum.
(WMEType addition-fact addend1 addend2 sum)

;; Write-answer is solved by writing an object in the solution
;; row of a column.
(WMEType write-answer column object)


    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;Productions to solve the problem
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Focuses on the next column

(p next-column
       =goal>
         isa addition-problem
         object =array
       =array>
         isa numberarray
         columns ($ =column2 $)
       =column2>
         isa column
     -   bottomrow +
         answerrow blank
     - (=array>
         columns ($ =column2 =column1 $)
      =column1>
         isa column
         answerrow blank)
==>
       =subgoal>
          isa write-answer
          column =column2
       !output!  ("Focusing on the next column ~%~%")
       !push!  =subgoal)

;; Adds a column
```

```
(p process-column
      =goal>
         isa write-answer
         column =column
         object nil
      =column>
          isa column
          toprow =num1
          bottomrow =num2
      =fact>
          isa addition-fact
          addend1 =num1
          addend2 =num2
          sum =sum
==>
      =goal>
         object =sum
     !output!  ("Adding ~S and ~S to get ~S. ~%~%"
               =num1 =num2 =sum))

;; Adds the carry
(p write-answer-carry
     =goal>
        isa write-answer
        column =column
        object =num
     =column>
        isa column
        note carry
     =fact>
        isa addition-fact
        addend1 =num
        addend2 one
        sum =new
==>
     =goal>
        object =new
     =column>
        note nil
    !output!  ("Adding 1 for the carry to ~S to get ~S. ~%~%" =num
=new))

;; If answer is 10 or more, writes a carry in the next column
```

```
(p write-answer-greater-than-nine
      =goal>
        isa write-answer
        column =column
        object =num
      =fact>
        isa addition-fact
        addend1 ten
        addend2 =new
        sum =num
      =column>
        isa column
      - note carry
      =problem>
         isa numberarray
         columns ($ =column1 =column $)
      =column1>
         isa column
==>
      =column1>
         note carry
      =goal>
         object =new
      =column>
          answerrow =new
      !pop!
      !output!  ("Setting a carry in the next column.")
      !output!  ("Writing out ~S and going to the next column.~% %"
=new))

;; If answer is less than 10, writes it down in the column
```

```
(p write-answer-less-than-ten
      =goal>
        isa write-answer
        column =column
        object =num
      =num>
        isa number*
      =column>
        isa column
      - note carry
      - (=fact>
          isa addition-fact
          addend1 ten
          addend2 =new
          sum =num)
==>
      =column>
        answerrow =num
      !pop!
      !output!  ("Writing out ~S and going to the next column.~% %"
=num))


    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          ;Initializes the problem and tables in memory
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(SetWM
(goal
    isa addition-problem
    object problem1)
(problem1
    isa numberarray
    columns (column0 column1 column2 column3))
(column0
    isa column
    toprow blank
    bottomrow +
    answerrow blank)
(column1
    isa column
    toprow two
    bottomrow seven
    answerrow blank)
```

```
(column2
    isa column
    toprow six
    bottomrow one
    answerrow blank)
(column3
    isa column
    toprow four
    bottomrow six
    answerrow blank)
(blank
    isa number*)
(zero
    isa number*
    value 0)
(one
    isa number*
    value 1)
(two
    isa number*
    value 2)
(three
    isa number*
    value 3)
(four
    isa number*
    value 4)
        ⋮
(nineteen
    isa number*
    value 19)
(fact00
    isa addition-fact
    addend1 zero
    addend2 zero
    sum zero)
(fact01
    isa addition-fact
    addend1 zero
    addend2 one
    sum one)
        ⋮
```

```
(fact09
    isa addition-fact
    addend1 zero
    addend2 nine
    sum nine)
(fact10
    isa addition-fact
    addend1 one
    addend2 zero
    sum one)
(fact11
    isa addition-fact
    addend1 one
    addend2 one
    sum two)
          :
(fact109
    isa addition-fact
    addend1 ten
    addend2 nine
    sum nineteen)
)

(WMFocus goal)
```

;;
;Sample run
;;

#—
? (run)
Focusing on the next column

Adding FOUR and SIX to get TEN.

Setting a carry in the next column.
Writing out ZERO and going to the next column.

Focusing on the next column

Adding SIX and ONE to get SEVEN.

Adding 1 for the carry to SEVEN to get EIGHT.

Writing out EIGHT and going to the next column.

Focusing on the next column

Adding TWO and SEVEN to get NINE.

Writing out NINE and going to the next column.

"No productions can be instantiated"
?
—#

Appendix B

LISP Code that implements the rehearsal process

```
(WMEType rehearsal)        ;rehearsal-subgoal
(WMEType interchunk (concept :activate) supergoal)   ;interchunk-type


(p reh_step
    =goal>
      isa rehearsal
    !eval!  (test_loop)
==>
    !bind!  =interc (!eval!  (iteration))
    !eval!  (if =interc
            (PushGoal =interc))
)
(parameters reh_step
           :a 0.05)


(p call_and_exit
     =goal>
      isa interchunk
      concept =wme
      supergoal =sg
```

```
==>
    !output!  ("Ehr...  ^A^%" =wme)
    !pop!
)
(parameters call_and_exit
            :a 0.25
            :r 0.25)


(p reh_done
    =goal>
      isa rehearsal
    !eval!  (not (test_loop))
==>
    !pop!
    !delete!  =goal
)
(parameters reh_done
            :a 0.05)


(defvar *PLoop* () ) ;List of Phonological Loop elements
(defvar *PPointer* *PLoop*)
; the pointer is implemented by means of a list that contains
; the rest of the PLoop, that is the part of the loop that still
; needs to be rehearsed.  The elt at the head of the list is the
; element that will be the first to be iterated.


(defstruct PLelt
(PCode nil)
(PLength nil)
(Iwme nil)  )
(defvar *EnablePhonologicalLoop* t)
; toggleable option to indicate whether the PLoop can be used
(defvar *TimeContentsPLoop* 2.0)
; the maximum contents of the PLoop, measured in seconds
```

```
(defun current_loop_duration ()
    (let ( (sum 0.0)
           (hdloop (ldiff *PLoop* *PPointer*)) )
     (dolist
        (e hdloop sum)
        (setf sum (+ sum (PLelt-PLength e))) )
    )
)


(defun kick_out_random_element ()
"Kick out a random element from the loop to make room for a new one"
   (let ( (doei (nth (floor (random (length *PLOOP*))) *ploop*)) )
      (print (list 'Loop 'overflow
                   (PLelt-PCode doei) 'was 'kicked 'out))
      (setf *ploop*
        (remove doei *ploop*)) ) )


(defun add_elt (elt)
; adds an element to the Phonological Loop and takes care that the
; loop contents never exeeds 2 seconds.  If necessary one or more
; elements are kicked out at random to make room for the element
; that needs to be added.

   (progn
     (if (eql *PLoop* '())
         (setf *PLoop* (list elt)
               *PPointer* '())
         (if (<= (+ (current_loop_duration) (PLelt-PLength elt))
              *TimeContentsPLoop*)
           ; *PPointer* is the tail of the loop that has not been
           ; iterated yet in an re-iteration cycle.  It is thus a
           ; sublist of the loop.
           ; The element is added at the end of the difference between
           ; the loop and the pointer and the tail is discarded.
           (setf *PLoop* (append (ldiff *PLoop* *PPointer*)
                                 (list elt))
                 *PPointer* '() )
           (progn (kick_out_random_element)
                  (add_elt elt) )
         ) )
         *PLoop*) )
```

```
; Precondition:  in the production an inter-chunk has been
; created
(defun add_to_loop (ichunk pwme)
    (if (not *EnablePhonologicalLoop*)
     (error "You are not allowed to use the Phonological Loop.
             If you wish to use it, set the parameter
             *EnablePhonologicalLoop* to true.")
     (let* ( (pnode (WMHash pwme))
             ; get the length of the wme that will be put into thePLoop
             (pnodelength (let ((type (type-of (WMNode-Slots pnode))))
                            (GetSlotValue pwme 'duration type)))
           )
           (if (eq pnodelength '())
              ; if the chunk has a phonological representation, then
              ; the duration-slot that contains the phonological length
              ; should not contain NIL
              (error "The WME  S has no phonological characteristics."
                     pwme) )
           (setf *PLoop*
                 (progn
                    ; construct the PLoop-element that will be added
                    (setf addelt (make-PLelt))
                    (setf (PLelt-PCode addelt) pwme)
                    (setf (PLelt-PLength addelt) pnodelength)
                    (PutSlotValue ichunk 'supergoal 'interchunk
                                                    *WMFocus*)
                    (PutSlotValue ichunk 'concept 'interchunk pwme)
                    (setf (PLelt-Iwme addelt) ichunk)
                    ;ophogen baselevel activation
                    (if *OptimizedLearning*
                        (incf (WMNode-References pnode))
                        (push *CycleNumber* (WMNode-References pnode))
)
                    ; the newly created addelt must be "shorter"
                    ; than 2 seconds if it is, then it can be added
```

```
to
                    ; the Phonological Loop
                    (if (< pnodelength 2.0)
                        (add_elt addelt)
                        (error "The element  S doesn't fit in the
                                             Phonological Loop" pwme)
                    ) )
          ) )
    ) )


(defvar *ActivationThreshold* -1.0)
; when an element in the PLoop has an activation that is insuffcient,
; it will be deleted from the PLoop
(defun set_pointer ()
; resets the variable *PPointer*
    (setq *PPointer* *PLoop*))
(defun test_loop ()
; checks if the Loop may be used and has contents
    (if *EnablePhonologicalLoop*
       (if (endp *PPointer*)
          '()
          t)
    (error "You are not allowed to use the Phonological Loop") )
)
```

```
(defun iteration ()
    (progn
         ;get the element that has to be rehearsed
         (setq pelt (car *PPointer*)
                 *PPointer* (cdr *PPointer*)
                 ichunk (PLelt-Iwme pelt)
                 ;determine which wme corresponds with the PLoop-element
                 pwme (GetSlotValue ichunk 'concept 'interchunk)
                 pnode (WMHash pwme))
         (if (< (GetActivation pwme) *ActivationThreshold*)
                 ; delete this element from the Loop
             (progn
                 (print (list 'DELETED pwme))
                 (setf *PLoop* (remove pelt *PLoop* ))
                 nil)
                 ; update the references, as they are used in calculating
                 ; the baselevel activation
             (progn
                 (if *OptimizedLearning*
                    (incf (WMNode-References pnode))
                    (push *CycleNumber* (WMNode-References pnode))
)
                 ichunk) )
    )
)


(defun LatencyAdjustment (chosen sorted cset)
```

```
; the latency of a rehearse-productioncycle is adjusted
   (let* ( (latency (CutoffMatchingLatency chosen sorted cset))
      (type (type-of (WMNode-Slots (WMHash *WMFocus*))))
      (reh_length 0) )
     ; check if this is an rehearse production
     (if (eql type 'rehearsal)
        (progn
          ; check if the latency of the rehearse production is longer
          ; than the phonological length of the item that was last
          ; rehearsed a phonological length of 0 indicates that there
          ; was no iteration in this production.
          (if *PPointer* (setf reh_length (PLelt-PLength (car *PPointer*))))
             (if (< latency reh_length)
                  reh_length
                  latency) )
             latency)
     )
)
(setf *LatencyHook* #'LatencyAdjustment)
; the latency of a productioncycle is computed.  When there was an
; iteration, it has to be at least the phonological length of the item
; that was iterated
```

Appendix C

Example of an inputfile that uses rehearsal

This file contains a task that has three subtasks:
First a list of digits is read and put into the Phonological Loop.
Subsequently ACT-R has to "wait" and rehearse.
Finally the contents of the Loop has to be reproduced.
In real life this could be a memory experiment in which the subjects are asked to read a list with digits which they are supposed to recall. Then they are told to put the list away, wait and rehearse the digits. After a certain period of time the subjects are asked to reproduce as much digits as possible.

```
(load "/usr/local/act/oldCode/WMpatch.lisp")


(setf *PLoop* '())
(setf *EnableRationalanalysis* t
      *BaseLevelLearning* 0.5
      *OptimizedLearning* nil
      *ExtendedContext* t
)


(WMEType topgoal tasklist)
(WMEType number* value duration)
(WMEType readgoal readlist)
(WMEType waitgoal duration)
(WMEType reproducegoal busy)
```

```
(p do-tops        ;three tasks are stored in a tasklist, which will
     =goall>              ;be in the focus of attention successively.
       isa topgoal
       tasklist (=subgoal $rest)
==>
     =goal>
       tasklist $rest
     !push!  =subgoal
     !output!  ("Pushing subgoal ~A ~%" =subgoal)
)


(p read-word        ;read the following digit in the list
     =goal>          ;and put it in the Phonological Loop
       isa readgoal
       readlist (=number $rest)
     =number>
       isa number*
       value =val
       duration =d
==>
     =goal>
       readlist $rest
     =inter>
       isa interchunk
     !eval!(add_to_loop =inter =number)
     !output!  ("Reading ~A...~%" =number)
)


(p reading_done      ;pop the readgoal from the goalstack
     =goal>          ;when all the digits in the list have been read
       isa readgoal
       readlist ()
==>
     !pop!  !output!  ("Finished reading. ~%")
)
```

```
(p wait_some_time      ;wait as many times as there are elements
     =goal>            ;in the waitgoal duration-slot
       isa waitgoal
       duration (=d $rest)     ;NB this slot has nothing to
==>                                    ;do with its phonological length!
     =goal>
       duration $rest
     !output!  ("Waiting..~A ~%" $rest)
     =subgoal>
       isa rehearsal
     !push!  =subgoal    ;every time the waiting goal has been matched
     !eval!  (set_pointer)     ;the rehearsal process is started.
     !output!  ("Start rehearsing ~%")
)


(p waiting_is_over      ;stop the waiting to go to the next task
     =goal>
       isa waitgoal
       duration ()
==>
       !output!  ("Waiting is over ~%")
       !pop!
)


(p reproduce_start      ;start reproducing the elements in the Loop
       =goal>
       isa reproducegoal
       busy nil
==>
     =goal>
       busy t
     =subgoal>
       isa rehearsal
     !eval!  (set_pointer)
     !output!  ("Starting reproduction ~%")
     !push!  =subgoal
)
```

```
(p reproduction_done    ;the whole Loop has been run through
     =goal>
       isa reproducegoal
       busy t
     !eval!  (not (test_loop))
==>
     !pop!
     !output!  ("These were all the numbers I could recall ~%")
)


(p reproduce_interchunk  ;reproduce the wme that is attached to the
     =goal>              ;inter-chunk that is contained in the Loop
       isa interchunk
       concept =con
       supergoal =sg
==>
     !output!  ("Reproducing ~S ~%" =con)
)
```

```
(SetWM
     (top
       isa topgoal
       tasklist (goal1 goal2 goal3))
     (goal1
       isa readgoal                          ;the list of digits that
     readlist (nine three three one one seven two))   ;has to be recalled
     (goal2
       isa waitgoal
       duration (x x x x x))   ;rehearse five times during the waiting
     (goal3
       isa reproducegoal
       busy nil)
     (one
       isa number*
       value 1
       duration 0.1)
     (two
       isa number*
       value 2
       duration 0.2)
     (three
       isa number*
       value 3
       duration 0.3)
     (seven
       isa number*
       value 7
       duration 0.5)
     (nine
       isa number*
       value 9
       duration 0.4)
)
(WMFocus top)
```

Appendix D

Inputfile for the "twenty words" memory test

```
(setf *PLoop* '())
(setf *EnableRationalanalysis* t
     *BaseLevelLearning* 0.5
     *ActivationThreshold* -1.0
     *OptimizedLearning* nil
)


(WMEType item duration)


(WMEType store-goal items)
```

```
(SetWM
     (Book1 isa item duration 0.3)
     (apple2 isa item duration 0.3)
     (happy3 isa item duration 0.3)
     (chair4 isa item duration 0.3)
     (hair5 isa item duration 0.3)
     (blue6 isa item duration 0.3)
     (honey7 isa item duration 0.3)
     (butterfly8 isa item duration 0.3)
     (coat9 isa item duration 0.3)
     (eagle10 isa item duration 0.3)
     (map11 isa item duration 0.3)
     (popcorn12 isa item duration 0.3)
     (seven13 isa item duration 0.3)
     (keyboard14 isa item duration 0.3)
     (lock15 isa item duration 0.3)
     (cracker16 isa item duration 0.3)
     (paper17 isa item duration 0.3)
     (fish18 isa item duration 0.3)
     (dead19 isa item duration 0.3)
     (band20 isa item duration 0.3)
     (goal isa store-goal items (book1 x x x apple2 x x happy3 x
                        chair4 hair5 blue6 honey7 butterfly8 coat9
                         eagle10 map11 popcorn12 seven13 keyboard14
                          lock15 cracker16 paper17 fish18 dead19 band20)))


(p do-an-extra-rehearsal
     =goal>
       isa store-goal
       items (x $rest)
==>
     =goal>
       items $rest
     =subgoal>
       isa rehearsal
     !eval!(set_pointer)
     !push!  =subgoal)

(parameters do-an-extra-rehearsal
           :a 0.05)
```

```
(p store-a-word
     =goal>
       isa store-goal
       - items nil
       items (=item $rest)
==>
     =goal>
       items $rest
     =inter>
       isa interchunk
     !eval!(add_ to_ loop =inter =item)
     !output!  ("Reading S" =item)
     !eval!(set_ pointer)
     =subgoal>
       isa rehearsal
     !push!  =subgoal)

(parameters store-a-word
           :a 0.05
           :r 0.8)


(p storing-done
     =goal>
       isa store-goal
       items nil
==>
     !pop!)

(parameters storing-done
           :a 0.05)
```

Bibliography

[1] Anderson, J.R. (1980). *Cognitive Psychology and its implications.* third edition, New York: W.H. Freeman and Company.

[2] Anderson, J.R. (1993). *Rules of the mind.* Hillsdale, New Jersey: Lawrence Erlbaum Associates Ltd.

[3] Anderson, J.R., Milson, R. (1989). Human memory: An adaptive perspective. *Psychological Review*, **2(6)** , 396–408.

[4] Baddeley, A.D. (1986). *Working memory.* Oxford Univ. Press (Clarendon).

[5] Baddeley, A.D. (1990). *Human memory : Theory and practice.* Allyn and Bacon

[6] Baddeley, A.D. (1992). Is Working Memory Working? *Quarterly Journal of Experimental Psychology*, **44A(1)**, 1–31.

[7] Baddeley, A.D. (1994). The Magical Number Seven: Still Magic After All These Years? *Psychological Review*, **101(2)**, 353–356.

[8] Baddeley, A.D., Hitch, G.J. (1994). Working Memory. In G. H. Bower (Ed.), *Recent advances in the psychology of learning and motivation*, **VIII**, 47–90. New York: Academic Press.

[9] Baddeley, A.D., Hitch, G.J. (1994). Developments in the Concept of Working Memory. *Neuropsychology*, **8(4)**, 485–493.

[10] Baddeley, A.D., Thomson, N., Buchanan, M. (1975). Wordlength and structure of short-term memory. *Journal of Verbal Learning and Verbal Behaviour*, **14**, 575–589

[11] Boneau, C.A. (1990). Short-term recognition memory under rehearsal instruction and imaging instructions. *Bulletin of the Psychonomic Society*, **28(4)**, 297–299.

[12] Broadbent, D.E. (1958). Perception and communication. London: Pergamon Press.

[13] Brown, J. (1958). Some tests of the decay theory of immediate memory. *Quarterly Journal of Experimental Psychology*, **10**, 12–21.

[14] Burgess, N., Hitch, G.J. (1992). Toward a Network Model of the Articulatory Loop. *Journal of Memory and Language*, **31**, 429–460.

[15] Colle, H.A., Welsh A. (1976). Acoustic masking in primary memory. *Journal of Verbal Learning and Verbal Behaviour*, **15**, 17–32.

[16] Conrad, R. (1964). Acoustic confusion in immediate memory. *British Journal of Psychology*, **55**, 429–432.

[17] Conrad, R., Hull, A.J. (1964). Information, acoustic confusion and memory span. *British Journal of Psychology*, **55**, 75–84.

[18] Craik, F.I.M., Lockhart, R.S. (1972). Levels of processing: a framework for memory research. *Journal of Verbal Learning and Verbal Behaviour*, **11**, 671–684.

[19] Craik, F.I.M., Tulving, E. (1975). Depth of processing and the retention of words in episodic memory. *Journal of Experimental Psychology: General*, **104**, 268–294.

[20] Greene, R.L. (1987). Effects of Maintenance Rehearsal on Human Memory. *Psychological Bulletin*, **102(3)**, 403–413.

[21] Lebière, C., Kushmerick, N. *ACT-R User's Manual*. Carnagie Mellon University.

[22] Lehrer, R. (1993). *ACT-R Primer*. University of Wisconsin-Madison.

[23] Miller, G.A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, **63**, 81–97.

[24] Morris, N., Jones, D.M. (1990). Memory updating in working memory : The role of the central executive. *British Journal of Psychology*, **81** , 111–121.

[25] Murdock, B.B. (1962). The serual position effect in free recall. *Journal of Experimental Psychology*, **64**, 482–488.

[26] Peterson, L.R., Peterson, M.J. (1959). Short-term retention of individual verbal items. *Journal of Experimental Psychology*, **58**, 193–198.

[27] Postman, L., Philips, L.W. (1965). Short-term temporal changes in free recall. *Quarterly Journal of Experimental Psychology*, **17**, 132–138.

[28] Ratcliff, R., Clark, S.E., Shiffrin, R.M. (1990). List-Strenght Effect: I. Data and Discussion. *Journal of Experimental Psychology: Learning, Memory and Cognition*, **16(2)**, 163–178.

[29] Salame, P., Baddeley, A.D. (1986). Phonological factors in STM: Similarity and the unattended speech effect. *Bulletin of the Psychonomic Society*, **24(4)** , 263–265.

[30] Sato, Koichi (1990). Recency effects and temporal distinctiveness: dissociation between free recall and memory for position/order. *Perceptual and Motor Skills*, **71**, 1339–1351.

[31] Shiffrin, R.M., Nosofsky, R.M. (1994). Seven Plus or Minus Two: A Commentary On Capacity Limitations. *Psychological Review*, **101(2)**, 357–361.

[32] Shimizu, H. (1987). The relationship between memory performance and the number of rehearsals in free recall. *Memory & Cognition*, **15(2)**, 141–147.

[33] Steele, G.L, Jr. (1990) *Common LISP, The Language*. second edition, Digital Press.

[34] Turner, M.L., Engle, R.W. (1989). Is working memory capacity task-dependent? *Journal of Memory and Language*, **28**, 127–154.

[35] Winston, P.H., Horn, B.K.P. (1989). *LISP*. third edition, Addison-Wesley.

[36] Vallar. G., Baddeley, A.D. (1982). Short-term Forgetting and the Articulatory Loop. *Quarterly Journal of Experimental Psychology*, **34A**, 53–60.