INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The Use of Prior Knowledge in Learning from Examples

Stephen B. Blessing

Carnegie Mellon University

Committee:
John R. Anderson, chair
Jill H. Larkin
Herbert A. Simon

Submitted in partial fulfillment of the requirements of the degree of Ph.D.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 9701866

UMI Microform 9701866
Copyright 1996, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Carnegie-Mellon University

COLLEGE OF HUMANITIES & SOCIAL SCIENCES

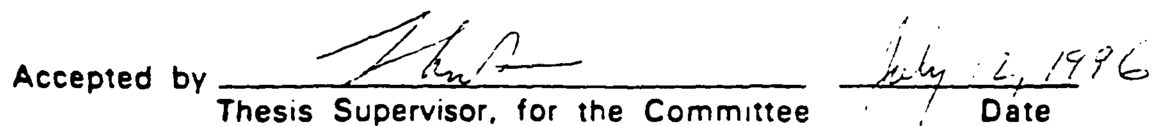
DISSERTATION

Submitted in partial fulfillment of the requirements

for the degree of Ph. D.

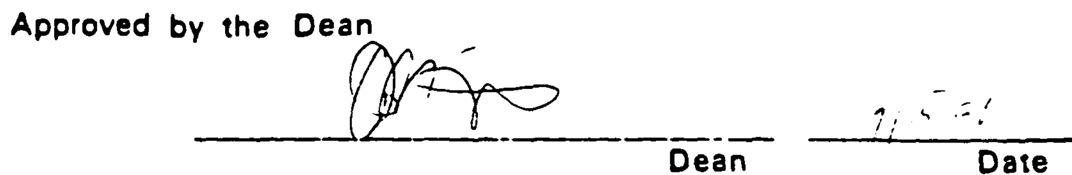
Title The Use of Prior Knowledge in Learning from Examples

Presented by Stephen B. Blessing



Accepted by ______________________ July 2, 1996
Thesis Supervisor, for the Committee **Date**

Approved by the Dean



Dean **Date**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

This dissertation examines the way people acquire procedures from examples, and provides a computational model of the results. In four experiments, people learned an analog of algebra. For each experiment, the initial knowledge that people had of the task was varied. In two experiments (Experiments 1 and 3), the syntactic knowledge that people had concerning the task was manipulated. The knowledge of syntax that participants had, particularly the ability to correctly parse the character string, was found to be a major determiner in the way participants acquired the rules. Experiment 2 explicitly manipulated participant's awareness as to how the task was related to their prior knowledge of algebra, with the finding that another major determiner of how the participants learned the task resting on how much of the task they can map to algebra. All three of these experiments examined the rule generalization behavior of the participants, with a fourth experiment specifically designed to examine this issue. The less syntactic and other declarative knowledge that participants had, the less general their rules. These findings, that people can learn from examples but that this learning is tempered by their additional declarative knowledge, are captured by an ACT–R model (Anderson, 1993).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgments

My sincere thanks go to the members of my committee. John has been a true mentor, and I have learned, by example, how to do research from him. The actual form of this dissertation has benefited greatly from Jill, and it is much better than it would have been without her suggestions. It was a honor to be a teaching assistant in Herb's class, and through his questions and insights, I have learned much. Thank you all.

What I will miss most about Pittsburgh is the friends I have made here. I have many fond memories, and you will be dearly missed. Also, to the friends made prior to Pittsburgh, thanks for being there, and I'm glad e–mail exists. Keep in touch, and you all have a place to stay when you come to Walt Disney World.

I thank my God for the many gifts He has given me. It is by His grace that I have made it as far as I have, and I pray that He will continue to guide and bless me through my life.

Finally, I dedicate this dissertation to my parents. They have allowed me to take my dreams as far as I could, even when it was against their better judgment. They have taught me (again, by example) that dedication, hard work, and perseverance will get you far in life, and these lessons I will have with me always. Thank you.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table of Contents

Abstract iii
Acknowledgments iv
Table of Contents v
Chapter 1: Introduction 1
- The Task
- Overview of ACT–SF
- Main Contributions of this Dissertation
- Dissertation Overview

Chapter 2: Literature Review 11
- Examples v. Procedures in Learning
- Schemata in Learning—Transfer
- Generalizations in Learning
- Previous Models of Learning by Example
- Summary

Chapter 3: Experiment 1—Syntactic Symbols 22
- Method
- Results
- Discussion

Chapter 4: Experiment 2—Algebraic Symbols 39
- Method
- Results
- Discussion

Chapter 5: The Model—ACT–SF 53
- Representation in ACT–SF
- Operation of ACT–SF
- ACT–SF Model Discussion
- Conclusion

Chapter 6: Experiment 3—Prefix Symbols 76
- Method
- Results
- Discussion

Chapter 7: Experiment 4—General Symbols 88
- Method
- Results
- Discussion

Chapter 8: Conclusions 101
- Implications

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

References 108
Appendix A: Additional Information 112
Appendix B: Annotated Examples 113
Appendix C: The ACT–SF Model 114
Appendix D: Model Run 127
Appendix E: Example Protocol 133
Appendix F: Additional Information II 145

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1
Introduction

How do people learn a new task, given the instructions and information available to them? How do they bring their existing knowledge, when appropriate, to bear in learning the new task? Furthermore, is there is a simple, underlying mechanism which can account for this learning? These are the questions which are at the heart of this dissertation. By examining people in-depth as they learn a new task, and by manipulating the amount and kind of knowledge that they have available with which to learn, answers can be given to such questions.

Anderson's ACT–R theory (1993) claims that all procedural knowledge (knowledge of how to do things) has its origins in declarative knowledge (knowledge of what things are). To be more concrete, and to use the terminology of Newell and Simon (1972), declarative knowledge can be thought of as the description of the problem states of a problem space, and procedural knowledge as the description of the transitions between these problem states. A similar distinction is made by Simon (1972). In the running system, ACT–R's syntax

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

makes this distinction apparent, with declarative memory realized as working memory elements, and procedural memory realized as production rules. Past researchers have made a similar claim concerning the transition of declarative to procedural knowledge and have created models of this process (e.g., Neves, 1981; Siklóssy, 1972; see the literature review in Chapter 2 for more information). The ACT–R theory posits a simple mechanism, called the analogy mechanism, by which declarative knowledge is proceduralized. This dissertation assumes this underlying claim and mechanism of the ACT–R theory. The model, described briefly in this chapter and more in–depth in Chapter 5, initially contains only declarative knowledge from which procedural knowledge is induced, via the analogy mechanism.

This chapter summarizes the task used in the experiments, the model developed within the ACT–R system, and the main contributions of this dissertation.

The Task

The task used in all the experiments of this dissertation is called Symbol Fun, and was used by Blessing and Anderson (1996) in their study of how people learn to skip steps. It is composed of different symbols which represent operators and operands, which are grouped together to form a character string. A sequence of two, three, or four such character strings form a problem and its solution, with legal steps in the sequence dictated by the application of particular rules. The task has its basis in algebra, and so the main manipulations involved are analogous to the algebraic manipulations of adding, changing, and deleting symbols from these character strings. However, the task is not a direct mapping of algebra, as can be seen in the sample problem displayed in Table 1.1. As in algebra, the goal is to follow syntactical rules to produce a final line in which the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 1.1
Sample of Problem in Symbol Fun

Step #	Symbol Fun	Corresponding Algebra
Given	♥℘♥Φ↔#Δ	– x – A = * C
1	♥℘♥Φ®Φ↔#Δ®Φ	– x – A + A = * C + A
2	♥℘↔#Δ®Φ	– x = * C + A
(Answer) 3	℘↔#Δ♥Φ	x = * C – A

variable, ℘, is alone on the left of the string divider, ↔. The beginning of Chapter 3 contains a more complete description of Symbol Fun's rules.

This task has two features which make it appropriate for examining how people use examples together with other knowledge to solve novel problems. First, because it is an artificial task, the information which participants have when starting to learn the task can be controlled. All participants in every experimental condition had the same set of examples to which to refer. However, some conditions in the different experiments were given additional information with which to learn the task. This additional information generally corresponded to syntactical information, such as which symbols are operators and which are operands, and also what makes a well-formed formula within the task. Second, even though the task is artificial, it did have its basis in algebra, and so some participants found it useful to use their knowledge of algebra in learning this new task. In one experiment (Experiment 2, "Algebraic Symbols"), participant's awareness as to how the task is related to algebra was explicitly manipulated.

Overview of ACT–SF

One of the main contributions of this dissertation is ACT–SF, an ACT–R implementation of people learning Symbol Fun. As stated above, the ACT–R theory claims that all knowledge begins in a declarative form, and that all procedural knowledge arises from this declarative knowledge. This transition is accomplished by the analogy mechanism. When ACT–R has a goal for which no

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

procedures apply, it will attempt to find a declarative example of the successful resolution of that goal, and then to infer the rule behind that resolution. It will next apply that rule to the current goal. ACT–SF uses this mechanism to learn the rules of Symbol Fun. The analogy mechanism of ACT–R was one of the least tested claims of the theory, and over the course of this dissertation, as well as through other research by different people, the mechanism has been refined.

Figure 1.1 provides a simple illustration of how the analogy mechanism works within ACT–SF. Panel A shows the current problem the system has, and for which no existing productions apply. Since no productions apply, ACT–SF must find an example which demonstrates what the proper rule to use is. Examples are chosen based on their similarity to the current goal and their activation. The model finds an example, such as in Panel B (Lines 1 and 2 from Table 1.1). Contained within that example is its "solution," or the next correct line in the solution sequence. ACT–R creates a new production rule which captures

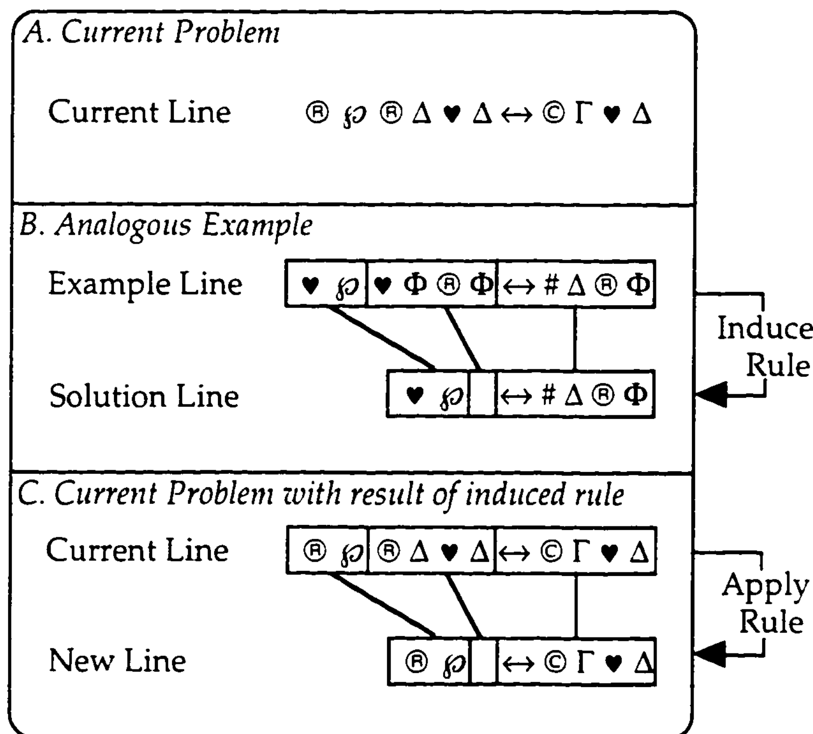


Figure 1.1: ACT–R's analogy mechanism

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the transformation from the example to its solution, basically by matching the symbols between the two lines, with a set way for variablizing or leaving as constants the various symbols. If the matching of symbols is not obvious, ACT-R can bring in other declarative knowledge with which to augment the rule, in order to create a potential candidate rule. The rule created from Panel B can be simple, such as "If you have a line that has all 11 symbols, then drop symbols 3 through 6 in the next line." The system next attempts to apply the new rule to the current line, as shown in Panel C. If it is successful, then it stores the rule for future use. If unsuccessful, it discards the rule and attempts to find another example to generate a different rule.

The full version of the model contains the necessary declarative representations, including a parsed, syntactically correct, hierarchic organization of each of the examples, to learn the correct procedural knowledge with minimal error. This model corresponds to participants given the most amount of information, before attempting to solve any problems. By removing pieces of that representation, the model mimics either participants early in the learning of the task who did not start out with the most information, or participants who were unsuccessful at learning the task. Chapter 5 further discusses this feature of the model.

Main Contributions of this Dissertation

This section outlines three main contributions of this dissertation. After each contribution is a short phrase in parentheses which will be used throughout the dissertation as an identifier for that contribution.

1) **In learning the rules of a task such as Symbol Fun, learners construct internal declarative representations of the examples presented to them. These declarative representations are influenced by knowledge of the task's syntax, as well as other**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

information particular to the task (e.g., knowledge of inverse operators). ***(Syntactic Knowledge)***

The experiments in this dissertation utilize examples as the main source of information people had to learn the task. A subset of participants had additional pieces of declarative information, about the task's syntax, with which to learn. By examining how people interact with these examples, and the extent to which they interact with them, a better understanding of how people incorporate examples in their learning of a new a task can be had. Furthermore, by investigating example use across the various informational conditions, the process by which people use this additional declarative information can be examined. The hypothesis is that the more relevant declarative information available at the time of learning, the more efficient the learning will be.

Experiment 1 (Chapter 3, "Syntactic Symbols") tested this claim by manipulating the amount of information participants had with which to learn the task. One group of participants only had some examples to which to refer, whereas two groups had, in addition to the examples, information regarding the task's syntax (e.g., a classification of the different symbols used, what makes a well–formed formula, etc.). Also, one of these two groups was also given a key piece of information (that two pairs of operators were related, or inverses, of one another) to aid in learning the task. Since this task has its origins in algebra, people may use their algebraic knowledge as a source for this syntactic information. If this is the case, then the effects of the syntactic knowledge in Experiment 1 will be attenuated. Experiment 2, discussed in the next contribution, was designed to manipulate people's awareness of how the task is related to algebra, and Experiment 3, discussed below, was designed to eliminate this attenuating factor.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Experiment 3 (Chapter 6, "Prefix Symbols") provided an even stronger test of this contribution by greatly reducing the similarity between the version of Symbol Fun used in Experiment 1 and algebra. The similarity was reduced by using a prefix notation instead of the standard infix notation. The reduction was necessary in order to provide a better picture of the benefit of syntactic knowledge, free of any extraneous knowledge, above just examples. The version of ACT–SF reported in Chapter 5 ("The ACT–SF Model"), as well as the ACT–R analogy mechanism in general, predicts that within a particular experimental condition (e.g., examples only or with syntax), learning across the two versions (infix or prefix notation) of the task should be equal.

2) One of the strongest predictors of success for learning Symbol Fun was if the learner was able to access and use their knowledge of algebra. *(Prior Knowledge)*

Often a student attempts, or is told, to apply knowledge gained in learning an old task to the learning of a new task. The old knowledge will transfer to the new task. This issue of transfer has been studied by previous researchers (e.g., Singley & Anderson, 1989; Kieras & Bovair, 1984), but the manner and mechanism by which this prior knowledge interacts with a set of examples used to learn a new task has not been sufficiently examined within the context of the ACT–R theory. The hypothesis is that this prior information constrains the knowledge space the participant needs to search, and so learning will be more efficient when this transfer occurs, with the benefit being the proportion to which the old information can be mapped onto the new task.

Experiment 2 (Chapter 4, "Algebraic Symbols") explicitly manipulated participants' knowledge of how the task is related to algebra. Three levels of hints were given, with each level providing additional explicitness in suggesting the use of algebra as a source of task knowledge. One group of participants

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

received only the first level hint, another group the first and second level hints, and a third group received all three levels of hint. The more explicit the hint, the better the learning should be.

3) Lack of adequate syntactic knowledge causes the analogy mechanism to build over–specific rules from examples. ***(Over Specificity)***

When the procedural knowledge required to do a task is formed, that knowledge must be constrained to only apply in certain contexts. Furthermore, the procedural knowledge must encode the types of structures to which it pertains (i.e., it must be variablized in some way). When given only examples from which to learn, fewer generalizations can be formed than when additional information may be available (such as the fact that two pairs of operators are inverses). The hypothesis is that the generalizations of participants with more syntactic information will be less constrained than those of participants given only examples from which to learn. That is, the ability to bring in additional declarative information when the analogy mechanism constructs a rule results in more general rules.

The errors made in the various experimental conditions suggest how participants generalize their rules, particularly the sign elimination steps (eliminating the sign in front of the $\wp$, as in Line 2 to 3 in Table 1.1). By examining the way in which participants switched and inverted, or did not switch and invert, a line's symbols, inferences were made as to the way they variablized their analogized rules.

Experiment 4 (Chapter 7, "General Symbols") explicitly examined how participants variablized the rules they were learning and compared their processes to ACT–SF. Participants initially learned only a subset of Symbol Fun, just the sign elimination steps and simpler problems. They then transitioned to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

more complex problems, where a close examination of this generalization process was obtained. Participants were good at generalizing the position in which symbols appear and should change into other symbols, but were not good at generalizing to higher-order relations, like among the inverse operators (even if given the inverse operator pairs).

Dissertation Overview

The rest of this dissertation follows this format:

Chapter 2: Literature Review. Discusses the findings of past researchers that bear on the issues contained within this dissertation.

Chapter 3: Syntactic Symbols. Explains more fully the task used in this dissertation's experiments, and details the results of the first experiment, which tested the claims of the Syntactic Knowledge Contribution: the more relevant declarative, syntactic information available, the better the learning will be.

Chapter 4: Algebraic Symbols. Examines how people's knowledge of algebra aids in learning the task in relation to the Prior Knowledge Contribution: the more a new task can be mapped onto an old one, the better the learning will be.

Chapter 5: The ACT–SF Model. Contains a description and a discussion of the full version of the ACT–R model, and how by the removal of certain aspects of this model's representation that unsuccessful and beginning participants can be modeled.

Chapter 6: Prefix Symbols. Similar to the first experiment in that it tests the Syntactic Knowledge Contribution, but uses a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

modified version of the task in order to eliminate any outside knowledge that a participant could use. The experiment served as a strong test of the model, which predicts similar performance between this experiment and the corresponding groups of the first one.

Chapter 7: Generalized Symbols. Another strong test of the model, but one that specifically examines the issue raised by the Over Specificity Contribution, that of how the rules are generalized and variablized.

Chapter 8: Conclusions. Provides a summary of the experiments, the model, and the findings of this dissertation. It also discusses the implications of the findings for education.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2
Literature Review

A lot of learning, particularly of school–taught subjects, occurs by students examining worked–out examples (Reed & Bolstad, 1991). When given a homework assignment in math or physics, students will often forego actually reading the chapter, but instead will turn to the assigned problems, then flip through the chapter to find an analogous problem, and attempt to solve the homework problem by doing the same transformations found in the analogous worked–out example. Additional information is often provided with these worked–examples to enable the students to better interpret those examples. One of the main goals of this dissertation is to better understand how this additional information allows the learner to interpret such examples.

Several researchers have shown that people can learn a new task quite well with only examples, which they sometimes must generate themselves, to guide them (Zhu & Simon, 1987; Shrager & Klahr, 1986). Zhu and Simon (1987) had Chinese students learn factoring quadratics by studying a series of carefully chosen worked–out examples. The students performed quite well at the task,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sometimes outperforming students who were taught by more conventional means. These students who learned by examples understood the material, and did not just superficially learn the actions needed to solve problems. The students could state the rules of factoring, and moreover, could demonstrate their understanding by checking their factoring work by multiplying, an aid not directly taught them.

Shrager and Klahr (1986) had participants learn a complex device by not giving the participants any instructions, but rather by having them interact with the device. The goal that the participants had was to figure out the function of one particular key on the keypad. Participants could write simple programs using this keypad, and could watch as the device carried out its program. In a sense, the people were generating their own examples with which to learn, these combinations of programs and device actions. Most people learned the device adequately in about thirty minutes, honing the hypotheses they were developing as new evidence, in the form of these self–generated examples, was created.

Examples v. Procedures in Learning

As shown above, previous experiments have indicated the importance of examples in learning a new task, and the reliance that students place in them. In many of these experiments, however, learning from examples was pitted against other ways of learning. That is, in a typical experiment there are three groups, one where the people are given only examples to learn from, another where the people only have a set of procedures to learn from, and a third group which has both the examples and procedures with which to acquire a new skill (e.g., Sweller & Cooper, 1985; Reed & Bolstad, 1991). The examples usually take the form of worked–out problems, whereas the procedures are an abstract "recipe" for how to solve a certain class of problems. The general finding is that people learn best when both procedures and examples are given and a little worse when

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

they just have the examples available to learn from. People who are just given a list of procedures to learn from generally do not perform nearly as well as the other two groups. Perhaps non-intuitively, the examples enable the students to learn most of the "how-to" (procedural) knowledge, as opposed to the actual procedures.

In one study, Reed and Bolstad (1991) taught groups of participants a particular class of algebra word problem. Across two experiments the finding was as mentioned above—the group that had both examples and procedures performed best, followed closely by the group that only had the examples. The group that only had the procedures performed worst. In acknowledging the poor performance by the group who learned by procedures, they stated that procedures in may work better for some tasks than they do for others (cf. Cheng, Holyoak, Nisbett, & Oliver, 1986; Fong, Krantz, & Nisbett, 1986), and also that they may not have written the best set of procedures for learning these problems. The efficacy of examples needs to be more adequately explored, particularly what it is that people extract from examples with which to learn and how supporting declarative knowledge aids in that learning process. The Syntactic Knowledge Contribution from the first chapter addresses this issue.

Schemata in Learning—Transfer

People often try to understand a new domain in terms of previously learned knowledge, and studies have shown that it is often advantageous to do so (Singley & Anderson, 1989). A common way of characterizing such knowledge is in terms of schemata (Bartlett, 1932; Rumelhart & Ortony, 1977). Schemata are knowledge structures that contain related information about a particular topic. For example, a person may have a schema for a type of physics problem that involves an inclined plane. This schema might contain information regarding the typical diagram that is associated with such problems, as well as the formulae

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

usually used to solve that type of problem. Schemata help problem solvers to organize the knowledge they possess about a particular topic for easy and quick access. Furthermore, schemata allow people to make inferences about unknown aspects of a situation, by providing default assumptions about it. Other researchers have developed different conceptualizations of schemata (e.g., the scripts of Schank & Abelson, 1977), but they all share the common framework of related knowledge elements within a single memory structure. The Prior Knowledge Contribution claims that a schema for an old domain can help a learner interpret examples for a new domain. In ACT–R schemata can contain both declarative and procedural knowledge, with the potential for both to transfer, depending on the closeness of the target domain. In the model discussed in Chapter 5, the transfer of procedural knowledge is not modeled.

Students are often told that a new concept that they are about to learn is similar to a concept that they already know, and thus for which they already possess a schema. For example, when learning about electricity, students are often told to think of it as water running down a pipe, or when learning about atoms, students are told they are similar to planets rotating around the sun in our solar system. The students are then expected to interpret the new knowledge in terms of their old knowledge, stored in a schema. How useful is this information? Do students learn more or learn faster when they are told that new information will be similar to previously acquired information, or are they better off learning from scratch, as it were? One of the goals of this dissertation was to examine these questions closely, particularly as it pertains to learning procedural information from prior, declarative knowledge.

Researchers have shown that schemata can be used in order to more easily learn and remember new, declarative material. By being able to place incoming information within an existing schema aids the learning process. Bransford and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Johnson (1972) gave people a passage of text to memorize. The group of participants who knew that the passage referred to doing laundry recalled more of the text than the participants who did not know what the passage was describing. People were able to use their knowledge about doing laundry, stored in a schema, in order to help them remember the passage.

A few studies have shown that people can also use previously acquired knowledge in order to help them learn new procedural skills. Kieras and Bovair (1984) gave people an electrical device that they had to learn to operate. One group of participants was instructed on how to use the device as if it were the weapon system on a spaceship from Star Trek. The other group of participants was shown how to use the device without reference to phasers, accumlators, and other science fiction elements. The group who received the Star Trek–like training learned to use the device in the same amount of training time, but remembered the procedures more accurately, used more efficient procedures, and executed them faster. Obviously, participants did not have a schema for how to use a phaser weapon system, and probably not all participants were even familiar with Star Trek and other science fiction works. However, the information could be tied together with a simple schema for how electrical systems should work ("shipboard power," "energy source selector," etc.), and so was able to aid participants in learning about the system. While Kieras and Bovair did not offer a mechanism to account for their finding, one explanation could be that the Star Trek information elaborated and built redundancy into their declarative knowledge of the system. These elaborations and redundancies allow easier access to the necessary knowledge.

However, the different kinds of information given to a problem solver as they are learning the task will not all be equally effective. Therefore it should not be interpreted that providing additional, even apparently relevant, information

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

will always lead to better learning. In one experiment of their study, Kieras and Bovair gave different groups of participants different information, all of it related to either a Star Trek–theme or to electronics. They found that the given information was most effective when it contained useful, lower–level knowledge (i.e., specific descriptions of the parts and knowledge of what parts were connected to one another) about the internal workings of the system that allowed the learner to infer exactly how to operate the device. Information that was overly general—that did not talk about the system in particular—was of no use. However, the lower–level knowledge did not have to be complete or set in a fantasy setting in order to be useful.

Generalizations in Learning

In developing a theory of how task instructions and prior knowledge are used in learning a new task, it is important to also examine how such knowledge either generalizes or constrains the rules that are being learned to do the new task. For instance, when learning by example, how does one decide which aspects of the problem are essential for solving it, and which aspects can be glossed over or variablized?

Many researchers have demonstrated that people just learning a task or domain often pay much attention to the superficial aspects of the problem (Chi, Feltovich, & Glaser, 1981; Novick, 1988; Holyoak & Koh, 1987; Ross, 1984). Instead of depending on how the problem is actually solved, they will often use a problem's content in determining its solution. For example, people will describe problems in terms of their typical contents (e.g., "riverboat" problems in algebra, or "spring" problems in physics), and will base their initial categorizations on the presence of such contents (Hinsley, Hayes, & Simon, 1977). It is only as they become more expert in the domain that they begin to focus more on the structural aspects of a problem (Cummins, 1992), such as its underlying

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

equations. However, even experts place at least some importance on content (Blessing & Ross, 1996; Hardiman, Dufresne, & Mestre, 1989), since content is often predictive of how the problem is solved.

This reliance on superficial content features make people very conservative in the generalizations they make while learning a new skill (Ross & Kennedy, 1991). Research by Bassok and Holyoak (1989; Bassok, 1990) investigated people learning physics. When tested for transfer on analogous problems in algebra, they performed poorly, since the original physics problems, as is typical for such problems, were presented in a very content–dependent manner. People originally taught algebra, on the other hand, did exhibit transfer to the physics problems. Bassok (1990) further examined this finding, and found that participants are sensitive to the type of variables (e.g., intensive vs. extensive) used to solve the problems. Ross (1989) has also demonstrated in his work with probability problems that people will generalize to categories of animate objects and inanimate objects, but when the current problem requires that an inanimate object take the role of an animate object in a previous problem, they are hesitant to do so. In recent work, however, Bassok, Wu, and Olseth (1995) found evidence that suggests people generalize by inducing semantic knowledge from the problems and creating "interpreted structures" that encode the relation between the objects in the problems. Lastly, Bernardo (1994) found that people tend to keep around problem–specific information in their schemata. He argues that this problem–specific information affords access to more abstract information during transfer.

In many respects, then, the problem of forming generalizations in the service of creating, or perhaps modifying, rules for a new task can be thought of as trying to decide which example to refer to, or what the applicable instructions are, and then deciding which aspects of the example or instruction is relevant to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the current situation. Once that determination is made, the solver must decide what is the proper level of generalization. Each experiment in this dissertation examined how people generalize the rules they are learning given their prior instructions, with one experiment (Experiment 4, "General Symbols") specifically designed to examine this issue (the Over Specificity Contribution of the last chapter: *Lack of adequate syntactic knowledge causes the analogy mechanism to build over–specific rules from examples*).

Previous Models of Learning by Example

Several past researchers have put forward the idea of learning procedural knowledge by declarative instruction with some computer simulations having been implemented (e.g., the UNDERSTAND program of Hayes & Simon, 1974; the Aptitude Test Taker of Williams, 1972). Perhaps the most ambitious effort, and the one most similar to the model presented in this dissertation, was a simulation by Neves (1978, 1980), who developed a computer model, called Alex, that learned simple linear algebra by having available only examples. Alex learned by examining pairs of lines for similarities and differences, and then constructing a rule that would account for the change. His system started with knowledge of arithmetic and a representation of algebraic structure, and then learns the rules of algebraic manipulations. It is remarkable in that it is still one of the few computer models that takes as its goal to account for learning by example essentially the whole of a real domain, but Neves does not present any empirical work to check if the processes used by Alex resembled the processes used by humans to learn the same material.

Siklóssy (1972) also developed a computer model, referred to as ZBIE, that learned natural language by being presented with sentences in the target language along with representations of those sentences (e.g., a picture which is described by the sentence). By comparing across these representations and then

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to the paired sentences, ZBIE learned the language's lexicon and syntax. Like Neves, however, Siklóssy did not report any empirical evidence to check if the processes ZBIE used to learn a language this way was similar to the way a human would do it. Indeed, Siklóssy anecdotally stated that he himself had difficulty learning a language through this method (a picture book series called *Language through Pictures*), more so than ZBIE would predict. Other cognitive architectures have also addressed language learning by example (e.g., Anderson, 1983; Rumelhart & McClelland, 1986)

In modeling how people supposedly generalized rules while learning, both Alex and ZBIE would sometimes create rules that would be either over– or under–specified. Over the course of learning, these rules would be replaced by more correct versions. Both systems had their own method of dealing with how that process occurred. A few computer models have examined explicitly how generalizations are formed while learning from specific examples. Hofstadter, Mitchell, and French (1987) have developed a computer system, called Copycat, that attempts to find generalizations from a given pair of letter strings. Copycat has limited knowledge of the Roman alphabet (e.g., what comes before and after each letter) and the idea of sameness. When given a string transformation pair like *abc* —> *abd* and asked what *ijk* should be transformed into, it will probably respond (it is non–deterministic) with *ijl*. It develops its rule by noticing in the given pair what letters are the same, or proceed or succeed one another. When given a more challenging transformation, like *ssskkoooo* —> *oopokkkss*, it can use the notions of rightmost or left–neighbor in order to produce a generalization that is "robust"—a rule that takes structural features into account. Little empirical work has been done to see if the transformations that Copycat tends to produce are similar to the rules that humans would produce.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Lewis (1988), however, did examine empirical evidence in order to validate the kinds of generalizations that his computer model, EXPL, made. Lewis described a handful of heuristics that aided people in making their generalizations. Two of these were the identity heuristic and the loose–ends heuristic. The identity heuristic asserts that when a component of a system response has occurred earlier in a user action, that user action specified that component of the system response. For example, if clicking a mouse on an object is followed by the disappearance of that object, then the identity heuristic would lead one to conclude that it was the clicking on the object that led to its disappearance. The loose–ends heuristic states that if an unexplainable response occurs in the presence of an action for which it cannot account, then that action is linked to the unexplained response.

Lewis performed an experiment in which he presented participants several scenes of a person interacting with a computer. Lewis asked the participants several questions concerning this interaction. For example, one scene has the words "alpha," "beta," "gamma," and "epsilon" in a bar at the top of the screen, and a star in the lower part of it. The user touches the star, then touches beta, and then touches the left side of the screen. The star then moves to the left part of the screen. For this scene, Lewis asked the question: "If a person tried to move the star to the bottom of the screen this way: 1) Touch "beta", 2) touch the star, 3) touch a place near the bottom of the screen, would it work. If not, why not?" For this particular item, most people (67%) replied that the attempt would not work, since the order was wrong. From an analysis of such responses across similar stimuli, Lewis found support for the identity (the one illustrated by the example) and loose–ends heuristics.

He further analyzed how people generalized from the given scenes, and characterized the generalizations as either as rational or superstitious. A

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

superstitious generalization will normally preserve the order of steps, and will also leave unchanged any unexplained steps. A rational generalization, on the other hand, will accept step reorderings, assuming that no logical constraint, such as removing a floppy disk before it is ejected, is violated in the reordering, and will get rid of any unexplained step. Lewis found that people make both types of generalizations, but tend to make more superstitious than rational generalizations. However, it is possible for the same person to make a superstitious generalization in one instance and then a rational generalization in another. It is still an open question as to what influences a person to make either a rationalistic or superstitious generalization in a particular instance, and what the role of prior knowledge may be in making these sorts of generalizations.

Summary

Previous researchers have shown the importance of examples in learning a new task. However, while models of the mechanisms by which the examples and other supporting declarative information are used to infer rules have been developed, their relation to the processes by which humans do it is not clear. The goal of this dissertation is to closely study this process empirically, and to model the results, including how people generalize the rules, within an existing cognitive architecture.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3
Experiment 1—Syntactic Symbols

The initial experiment tested how crucial examples are in the learning process, and to see the benefit and importance of various pieces of declarative knowledge in interpreting those examples, such as the task's syntax and how the operators are related to one another. This is in accordance with the first main contribution of this dissertation:

1) **In learning the rules of a task such as Symbol Fun, learners construct internal declarative representations of the examples presented to them. These declarative representations are influenced by knowledge of the task's syntax, as well as other information particular to the task (e.g., knowledge of inverse operators).**

The more relevant declarative knowledge that can be brought to bear in interpreting the examples, the more efficient the learning will be. As stated previously, the ACT–R theory claims that through these interpreted examples new procedural knowledge arises, through a process dictated by the analogy

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 3.1

How the Symbols Used in this Task Map onto Algebraic Symbols (All Experiments)

Algebraic Symbol	+	–	*	/	Operands	x	=
Symbol in Task	®	♥	#	©	Δ, Γ, Φ, Ω	℘	↔

mechanism. By varying the amount and kind of information available to people as they try to the task, some measure of the contribution of the various pieces of declarative knowledge can be assessed and modeled.

As mentioned earlier, the task used in this dissertation, called "Symbol Fun," was designed to be an analog of algebra. In place of the standard four operators and Roman letters, Greek and various other symbols were used in order to mask the similarity to algebra. Table 3.1 lists the symbols used, and how they map onto the standard algebraic symbols. In most of the examples to be presented in this dissertation, the standard algebraic symbols were used, so that the reader may use previous knowledge in order to decode parts of the task.

The manipulations used in the task correspond to the algebraic manipulations of adding the same thing to both side of the character string, canceling symbols, and eliminating signs in front of the ℘. All of these manipulations make use of the fact that there are two pairs of inverse operators. Table 3.2 contains an example of one of the hardest problems, with all of the steps needed to solve the problem made explicit. The first step in solving this problem is to add ®Φ to both sides of the character string (the ↔ divides the

Table 3.2

Sample of Problem in Symbol Fun

Step #	Symbol Fun	Corresponding Algebra
Given	♥℘♥Φ↔#Δ	– x – A = * C
1	♥℘♥Φ®Φ↔#Δ®Φ	– x – A + A = * C + A
2	♥℘↔#Δ®Φ	– x = * C + A
(Answer) 3	℘↔#Δ♥Φ	x = * C – A

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

string into left and right halves). For the second step, the ♥Φ®Φ is canceled from the left hand side. For the final step, a rule is applied in order to eliminate the ♥ from in front of the ℘. It should be noted that the underlying rules were constructed such that each problem only had one solution path—there is no branching.

Method

Participants. Forty-nine Carnegie Mellon University undergraduates participated in this experiment for partial course credit and pay.

Materials. I constructed an algebra analog for this experiment. Differences existed between this task and algebra, and so the mapping was not perfect. For example, the division/multiplication operator pair acted more like the addition/subtraction operator pair than in standard algebra. Also, this task had a more limited order of operations. Parentheses were not used, and some of the allowable manipulations would look strange in algebra. Also, any operator was allowed in front of x, so it was possible to end up with an equation which looked like $* x = * A + B$. The order of operations was constrained so that at each step in any problem, only one rule was applicable. That is, at any intermediate step in solving a problem, only one operator can be used to achieve the next step in the problem. There was never a choice between operators.

Thirteen rules are sufficient to do all problems (see the model in Chapter 5). These rules corresponded to adding the same symbols to both sides of the character string, canceling symbols when appropriate on one side of the equation, and eliminating the sign in front of the ℘ when one occurred. However, these rules were never presented to the participants. Instead, participants had to infer the rules from the information that was available to them and by interacting with the task. Depending upon condition, the initial

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 3.3

Examples available to all participants in Experiments 1 and 2

Example 1

℘®Φ↔®Δ

℘®Φ♥Φ↔®Δ♥Φ

℘↔®Δ♥Φ

Example 2

♥℘#Γ↔♥Φ

♥℘#Γ©Γ↔♥Φ©Γ

♥℘↔♥Φ©Γ

℘↔®Φ©Γ

Example 3

©℘↔#Γ®Δ

℘↔#Δ®Γ

Example 4

℘♥Γ↔♥Φ

℘♥Γ®Γ↔♥Φ®Γ

℘↔♥Φ®Γ

Example 5

#℘↔#Γ©Δ

℘↔©Γ#Δ

Example 6

®℘®Δ↔©Γ

®℘®Δ♥Δ↔©Γ♥Δ

®℘↔©Γ♥Δ

℘↔©Γ♥Δ

Example 7

℘#Γ↔©Δ

℘#Γ©Γ↔©Δ©Γ

℘↔©Δ©Γ

Example 8

©℘®Γ↔®Ω

©℘®Γ♥Γ↔®Ω♥Γ

©℘↔®Ω♥Γ

℘↔®Γ♥Ω

information available to the participants differed. All participants had a screen of eight completely worked–out examples available to them, as presented in Table 3.3. They could refer to this screen at any point as they tried to solve problems. In picking this set of examples, the only rubric used was that each underlying rule had to be represented at least once. Some of the conditions received additional information, to be described shortly.

Procedure. The task was implemented as a Hypercard 2.2.1 stack (Apple Computer, Inc., 1994) which was run on an accelerated Apple Macintosh IIci computer connected to a two–page monitor. All participants initially saw two screens that contained some introductory comments about the experiment and instructions on the task's interface. After this point, the information that participants subsequently received depended on what condition they were placed (Appendix A contains the information that the two syntax groups had):

Examples: This group only saw the screen with the eight examples (shown in Table 3.3)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Syntax(No Hint): Before seeing the examples screen, both syntax groups (Syntax(Hint) and Syntax(No Hint)) received information concerning the task's syntax and goal structure. The syntax information classified the symbols used in this task as either "object" or "connector" symbols, roughly corresponding to constants and operators in algebra, and also explained what constituted a well–formed formula in the task. The goal structure simply indicated that the goal for each problem was to "isolate" (i.e., solve for) the script–p character, that a set of rules existed for solving the problems, and that only one rule was applicable at any step in solving the problem.

Syntax(Hint): Between seeing the syntax information screens and the example screen, this group received a hint for learning the task. This hint told the participants that two pairs of operators were "related" to one another. In algebra, this would correspond to the fact that plus and minus, and times and divide, are inverses.

Once participants started to solve problems, they could refer back to any of the information they had already seen by clicking on–screen buttons. It should be emphasized that for this experiment no mention of algebra was made to the participants, and the terminology used tried to distance the task as much as possible from algebra (e.g., using "isolate" instead of "solve for").

Each problem was presented in a box near the top of the screen. The participant then used an on–screen keypad which contained all the symbols used in the task to click out, with the mouse, the next correct step which would follow from either the problem, or from one of the lines the participant had already

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

clicked. A delete key was available to erase any character they had clicked. The participant's lines appeared in a box below the problem. Once the participant had clicked out a step, he or she clicked a special button to have the computer check the answer. If the step they had clicked out was the next correct one, the computer would respond, "Good," and the participant could continue with the problem. If the line clicked out was the problem's solution, then the computer would respond, "Excellent," the box containing the participant's lines would clear and a new problem would appear. If the line was correct, but the participant had skipped a step (possible on the two– and three–step problems), a dialog box would appear stating that step skipping was not allowed, their line would be erased, and they would be given another chance to click out a line. If, however, the line was incorrect, the computer would respond, "Try again," the participant's line would be erased from the box below the problem and moved to a different location, and the participant would then have another chance to click out a correct line. If the second attempt was not correct, the computer would respond, "Here's the correct line" and the next correct step (following from the last correct line) would appear.

Each participant was asked to solve 32 of each of the three types of problems (one–, two–, and three–step problems) for a total of 96 problems. Each participant had 2 hr with which to solve all 96 problems. There were 12 participants in the Syntax(Hint) group, 14 in the Syntax(No Hint) group, and 23 participants in the Examples group.

Results

Background and General Results

Table 3.4 contains summary information about the performance of participants in this experiment for easy reference. Participants reported their math SAT scores on a voluntary basis (out of all the experiments in this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 3.4
Syntactic Symbols At–a–Glance

	Syntax(Hint)	Syntax(No Hint)	Examples Only
Self–reported math SATs	662[a]	656[a]	655[a]
Reading Instructions (min)	5.11[a]	4.34[a]	2.94[b]
Examining Examples (min)	2.00[a]	1.58[a]	0.79[b]
Successful Participants	12 of 12[a]	12 of 14[a]	12 of 23[b]
Self–reported math SATs	662[a]	673[a]	683[a]
Example References	23.72[a]	52.41[b]	70.83[b]
Total Time (min)	64.09[a]	79.42[b]	81.54[b]
First Block (12 problems)	20.10[a]	24.12[a]	33.09[b]

dissertation, only 6 participants reported that they did not remember their score, or that they did not wish to divulge it). No difference is detected between the SAT scores of the participants in the three groups ($F < 1$), either when examining the groups as a whole, or just looking at those participants who completed all 96 problems (the "successful" participants, to be discussed shortly).

Preparation times. Not surprisingly, participants in the three groups spent different amounts of time reading the initial information ($F(2,46) = 9.96$, $MSE = 2.08$, $p < .001$), with the Examples group taking less time (2.94 min on average) than the other two groups (5.11 min for the Syntax(Hint) group and 4.34 min for the Syntax(No Hint) group), as shown by a Newman Keuls post–hoc test, $p < .05$. Participants in the three groups also differed in the amount of time initially examining the screen of examples ($F(2,46) = 6.16$, $MSE = 1.07$, $p < .01$). Again, a Newman Keuls post–hoc test shows that the Examples group spent less time (0.79 min, on average) than the other two groups (2.00 min for the Syntax(Hint) group and 1.58 min for the Syntax(No Hint) group), which did not differ from each other.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Successful and unsuccessful participants. At this points it is important to make a distinction between two types of participants within each group: those participants who completed all 96 problems in the allotted two hours and those who did not. Twelve participants in each group completed the entire set of 96 problems. Everyone in the Syntax(Hint) group finished, but 2 people in the Syntax(No Hint) group did not, and 11 people in the Examples group did not complete the task. The 2 people who did not complete the task in the Syntax(No Hint) condition solved 56 problems in one case and 52 problems in the other, and the 11 people who did not finish in the Examples group made it to problem 23.4 on average. Significantly fewer people ($p < .05$) finished in the Examples group. Looking at the initial instruction time measures examined in the previous paragraph, the people who did not finish the task did not differ on those measures from the people who did finish. Unless specifically mentioned, the analyses discussed for the rest of this experiment, and also for the other experiments, will be based just on those participants who completed the task.

Remindings. At the end of the experiment, every participant was asked if the task they just learned (or attempted to learn) reminded them of anything—any other task or domain that they knew about. In both the Syntax(Hint) and Syntax(No Hint) groups, 9 of the 12 participants who finished the task reported that the task reminded them of algebra. In the Examples group, 11 of the 12 people who learned the task said the task was similar to algebra. However, of the 11 people who did not learn the task in the Examples group, only 1 participant reported the task's similarity to algebra. The two people who did not finish in the Syntax(No Hint) group, one reported being reminded of algebra, the other one did not. For those who did not say algebra, the most common answers were either that they were reminded of nothing or they were reminded of some sort of logic task. Clearly, for those people who learned the task, drawing upon

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

algebraic knowledge was beneficial—particularly for those who made that connection in the Examples group. This relation between being reminded of algebra and learning the task was examined in depth in the second experiment.

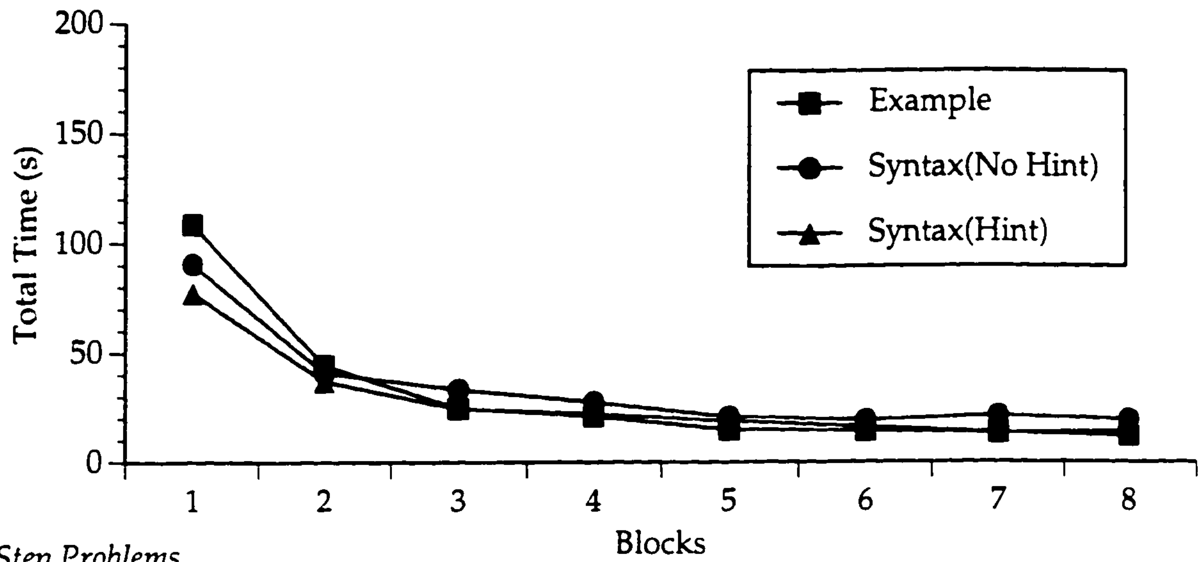
Learning

Accessing information. The most common piece of information that participants referred back to while solving the problems was the screen of examples (indeed, that was all the Examples group had to refer back), and significant differences were detected between the number of times participants returned to that page ($F(2,33) = 5.54$, $MSE = 6759$, $p < .01$). The Syntax(Hint) group turned back to that page a mean of 23.72 times, the Syntax(No Hint) group 52.41 times, and the Examples group 70.83 times. A Newman Keuls post–hoc test showed that the Syntax(Hint) group was significantly lower than the other two groups, but the Syntax(No Hint) group did not differ from the Examples group. The Syntax(Hint) group and the Syntax(No Hint) group did not refer back to the screens of syntax or goal information often (on average only twice for the syntax screen, and less than once for the goal screen). There were no differences between these two groups on those references (for both, $F < 1$). No one in the Syntax(Hint) group referred back to the hint screen. For the groups that did receive the additional information, that extra information just needed to be viewed once, and that was sufficient to help them in learning the task. Furthermore, despite the fact that the additional information just needed to be examined once, it allowed those people to learn the task with fewer references back to the example screen. The 11 people who did not finish in the Examples group referred back to the Examples page 93.52 times. Even though they made it through roughly 24 problems on average, they referred back to the examples screen a lot.

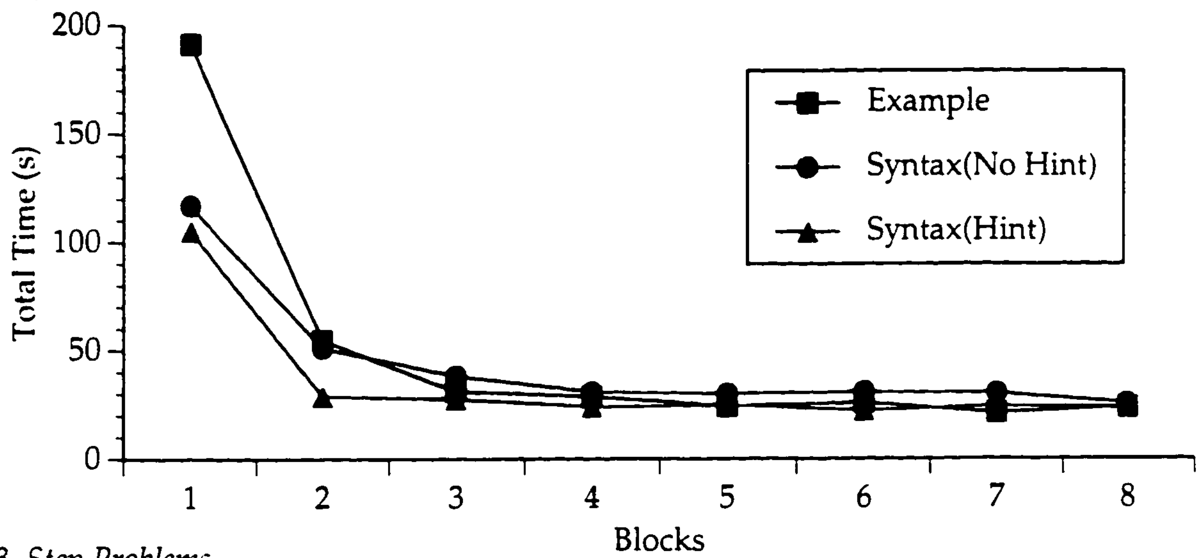
Completion time. The three groups differed significantly in the mean total time it took participants to solve all 96 problems, $F(2,33) = 3.50$, $MSE = 310.57$, $p <$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1–Step Problems



2–Step Problems



3–Step Problems

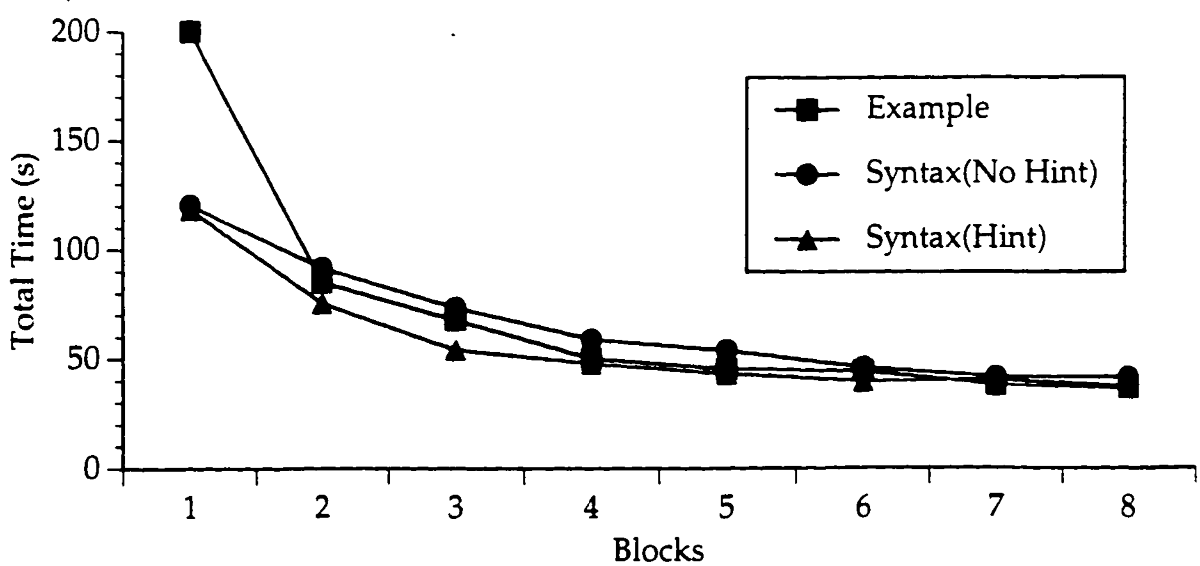


Figure 3.1: Overall time by block for each problem type (Experiment 1)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

.05. The Syntax(Hint) group took a mean of 64.09 min to solve all the problem, the Syntax(No Hint) group spent 79.42 min, and the Examples group took 81.54 min. A Newman Keuls test revealed that the Syntax(Hint) group took significantly less time than both the other groups ($p < .05$), but that the other two groups did not differ from one another. Figure 3.1 plots the average time participants spent solving the problems (broken up between problems of differing lengths). Note that trials have been blocked in these graphs, and in the graphs to follow, to aid readability. As can be seen, the groups differed substantially in the first block of trials, less so in the second block, and by the third block the groups were performing almost equally, and continued to do so throughout the rest of the experiment.

Since the most difference is seen in the first block of trials, a separate analysis was done on it. This block contains the first 12 problems, with each type of problem being represented 4 times. The same set of problems was used for each participant, and the problems were presented in the same order. The results of this analysis show a significant difference ($F(2,33) = 3.50$, $MSE = 310.57$, $p < .05$), with a Newman Keuls test showing that the Examples group took significantly longer than the other two groups (on average, 33.09 min to complete these first 12 problems), but no difference between the Syntax(Hint) and Syntax(No Hint) groups (20.10 min and 24.12 min, respectively).

Errors

Error types. The three groups differed on the number and kind of errors they produced while learning the task. Table 3.5 provides a breakdown of those errors by group. Syntax errors refer to lines that participants type that are not well-formed. That is, these lines could in no way exist within the task's syntax. Semantic errors are all other errors—generally they are the use of the wrong operator. The table rows for each group refer to the step that the error occurred.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 3.5

Experiment 1: Average Errors per Participant

Examples Only

	Syntax	Semantics	Total
Addition	2.75 (8%)	9.42 (28%)	12.17 (36%)
Cancellation	0.50 (2%)	1.5 (4%)	2.00 (6%)
Sign Elimination	3.17 (9%)	16.50 (49%)	19.67 (58%)
Total	6.42 (19%)	27.42 (81%)	33.83

Syntax(No Hint)

	Syntax	Semantics	Total
Addition	2.75 (4%)	17.00 (26%)	19.75 (30%)
Cancellation	0.67 (1%)	3.17 (5%)	3.83 (6%)
Sign Elimination	5.67 (9%)	37.17 (56%)	42.83 (65%)
Total	9.08 (14%)	57.33 (86%)	66.42

Syntax(Hint)

	Syntax	Semantics	Total
Addition	1.08 (4%)	4.25 (14%)	5.33 (17%)
Cancellation	0.17 (1%)	2.67 (9%)	2.83 (9%)
Sign Elimination	1.42 (5%)	21.42 (70%)	22.83 (74%)
Total	2.67 (9%)	28.33 (91%)	31.00

Addition errors occurred on the first step of one– and two–step problems, where the proper thing to do was to add an operator/operand pair to both sides of the character string. Cancellation errors occurred on the second step of one– and two–step problems, in which participants needed to cancel symbols on the left-hand side. Finally, sign elimination errors happened on the last step of a three-step problem or the only step of a one–step problem. These steps involved the removal of the sign in front of the script–p, and generally involved some manipulation to the symbols on the right–hand side.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In terms of total errors the Syntax(No Hint) group made significantly more than the other two groups ($F(2,33) = 3.82$ $MSE = 1281$, $p < .05$). This can be attributed to two reasons. First, the Syntax(No Hint) group knew what made a well-formed expression, but did not initially have the knowledge that two pairs of operators were related to one another. This additional information that the Syntax(Hint) group had enabled them to learn the task while making significantly fewer errors. A lot of the errors made by the Syntax(No Hint) group were at the beginning, trying to figure out the proper operator to add for the addition step (12.10 errors per participant, of the 17.00 erorrs, could be attributed to participants knowing that the same thing needed to be added to both sides of the character string, but no knowing which operator), or how the operators affected one another during the sign elimination step (essentially all of the semantic sign elimination errors). Second, there is a selection bias in the Examples group, in that Table 3.5 lists the statistics for 12 of 23 people in the Examples group and 12 of 14 people in the Syntax(No Hint) group. The Examples group contains only participants fairly proficient at learning the task. Examining in more detail the 11 participants who did not master the task in the Examples group (and who made it to a mean of 23.5 problems), it is found that they made 516 total errors, 221 (43%) of them being syntactic in nature. However, looking at only the top 50% of participants in each group (i.e., 12 of 23 participants in the Examples group, 7 of 14 in the Syntax(No Hint) group, and 6 of 12 in the Syntax(Hint) group), in terms of least number of total errors, one does not find a difference ($F(2,22) = 1.96$, $MSE = 383.2$, $p > .1$).

Examining the percentages of errors in Table 3.5, one sees that the profile of errors in the Syntax(No Hint) group is much more similar to that of the Examples group; the correlation between the percentages of these two groups is .98 (the correlation between the Syntax(No Hint) and Syntax(Hint) groups is .92,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and .85 between the Examples and Syntax(Hint) group). The Syntax(No Hint) and Examples groups made a much higher percentage of semantic errors on the addition step than did the Syntax(Hint) group. The addition step, once one knows that two pairs of operators are related, is relatively simple to learn. No one had much difficulty with the cancellation step. The proper rule for that step is that if the pattern `((operator) (constant) (operator inverse) (same constant))` appears on one side of the equation, those four symbols can be eliminated. At the beginning, however, most participants learned it as just dropping the four right–most symbols on the left side of the character string. This is evident in verbal protocols, to be discussed in conjunction with the model in Chapter 5. The sign elimination step was difficult for participants to master, and this is where most errors occurred for all participants, but particularly so for those in the Syntax(Hint) group, who had the information available to quickly master the addition steps (e.g., knowledge of inverse operators).

Sign–elimination errors. The Syntax(Hint) group tended to make errors that made the rule set more parsimonious. The rule for eliminating a # in front of the script–p was similar to the rule for eliminating a ♥ (that is, inverting the related symbols on the right–hand side). However, the rule for eliminating the ® (do nothing to the right–hand side) was quite different than the rule for eliminating the © (switch the two constants on the right–hand side). People in the Syntax(Hint) group attempted to apply the ® elimination rule when eliminating a © and vice versa 54 times (42% of all errors on ® and © elimination steps), whereas participants in the Examples group did so only 17 times (14% of errors on those steps). The Syntax(No Hint) group was more similar to the Syntax(Hint) group, making those errors 54 times (30% of applicable errors). Thus participants with the most information tended to over–generalize their rules.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

However, participants in the Examples group were more likely to make a particular type of error in # elimination. The correct rule is that to eliminate a # in front of the script–p, all the #s on the right–hand side become ©s, and all the ©s become #s. The single example that demonstrated this rule in the screen of examples was misleading:

Example 5: #℘↔#Γ©Δ
℘↔©Γ#Δ

One possible interpretation of that example would be that the rule is to switch the position of the operators. Indeed, the first time almost all participants tried to solve a problem which needed the # elimination rule, they would switch the operators, not invert them (across all experiments, only one participant used the correct rule on the first attempt). The Syntax(Hint) group quickly learned the correct rule. For this experiment, they attempted to switch the operators 15 times (18% of the # elimination errors). However, the Examples group perseverated in making that particular error, doing so 34 times (41% of the errors). The Syntax(No Hint) also made this error often, 41 times (31% of # elimination errors). The groups with the least information were not able to create a rule with the proper generality.

Discussion

This experiment tested the claim of the Syntactic Knowledge Contribution:

1) **In learning the rules of a task such as Symbol Fun, learners construct internal declarative representations of the examples presented to them. These declarative representations are influenced by knowledge of the task's syntax, as well as other information particular to the task (e.g., knowledge of inverse operators).**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

On all measures the group that had the most information, the Syntax(Hint) group, performed significantly better than the other two groups. The most interesting result is the 50% failure rate of the Examples group, compared to almost everyone learning the task in the two syntax conditions. The Examples group was always worst (except in total number of errors for all 12 successful participants), and the Syntax(No Hint) group would be someplace in between—sometimes they were more similar to the Syntax(Hint) group but frequently would be more similar to the Examples group. This pattern held across all the major measures of performance—whether or not they learned the task, number of references back to the examples, time to learn the task, and errors made while learning. The additional declarative information was extremely beneficial in learning the task. Such results support the Syntactic Knowledge Contribution.

Experiment 2 was conducted in order to more closely investigate the link between people learning this task and their knowledge that the task is based on algebra. One of the striking findings of this experiment is that a major determiner as to whether a person learns the task, if they are in the Examples group, is if they are reminded of algebra. Almost all the people (11 of 12) in the Examples group who learned the task were reminded of algebra, but only 1 of the 11 who did not complete the task reported the task's similarity to algebra. Experiment 2 manipulated people's knowledge as to how the task was related to algebra in an attempt to better understand this relationship.

An interesting pattern emerges from the error data, particularly on the sign elimination steps, between the people who have a lot of information with which to begin learning the task and those who have only the examples. The pattern provides some evidence for the Over Specificity Contribution:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3) Lack of adequate syntactic knowledge causes the analogy mechanism to build over–specific rules from examples.

This difference can perhaps best be characterized as one group being more theory driven (the Syntax(Hint) group) and the other being more driven by data (the Examples group). As previously stated, many of their errors with the sign elimination steps attempted to make the rule set more parsimonious. The Syntax(Hint) group knew that certain pairs of operators were related, and knew to look for those kinds of relations. Once they figured out the rules for ♥ and # elimination, they were more likely to pair the © and ® together for the sign elimination steps. The Examples group did not initially know about the pairing of operators, and so were less disposed to finding such over–arching relations. The Syntax(No Hint) group with their knowledge of syntax and goal had some idea of the underlying structure of the task, and so resembled more closely the Syntax(Hint) group on this measure.

Another instance where this occurs in the error data is in learning the # elimination rule, where one of the examples was very misleading. The Syntax(Hint) group, with their knowledge of inverse operators figured out the correct transformation after attempting to do one problem and being told the right answer. The Examples and Syntax(No Hint) groups, not knowing to perhaps look for inverse operators, perseverated in making that error. Experiment 4 was designed specifically to examine these issues more closely, but I will be mentioning them in relation to the other two experiments as well.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4
Experiment 2: Algebraic Symbols

Experiment 2 examined more closely the result from Experiment 1 that people who were reminded of algebra in the Examples group were much more likely to learn the task than those who were not reminded of it. Indeed, almost all the people in the former group (11 of 12) reported being reminded of algebra, whereas almost none of the people in the latter group did (1 of 11). People were clearly tapping into their knowledge of algebra in learning the task. Since the largest effect of this was seen in the Examples group, it is on that condition that the groups in this experiment were based. This experiment attempted to manipulate in a controlled way people's awareness of the task's similarity to algebra, thereby obtaining a better test of this dissertation's second contribution:

2) One of the strongest predictors of success for learning Symbol Fun was if the learner was able to access and use their knowledge of algebra.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In Experiment 1, nothing in the task's instructions made mention of algebra, and in fact the information presented to the participants was written in order to mask the task's basis in algebra.

The main manipulation in this experiment took the form of an explicit hint that the task was indeed related to algebra. The level of detail that the hint had was manipulated between the three groups that comprised this experiment. People either received a low detailed hint, which just said that the task was based on algebra, or an intermediate detailed hint, which not only said the task had its origins in algebra, but also mentioned the different kinds of transformations in the task. There was also a high detailed hint, which not only contained the information in the intermediate detailed hint, but also provided a mapping between the character strings in the examples and their algebraic counterparts. The expectation is that the more detailed the hint, the more efficient the learning will be.

Method

Participants. Forty–four Carnegie Mellon University undergraduates participated in this experiment for partial course credit and pay.

Materials. The task used in this experiment was exactly the same as the one used in Experiment 1. The differences between the groups, as in Experiment 1, was only in the initial information available to the participants. The screen of examples available to the participants in all groups was the same as in Experiment 1, except that for one group in this experiment it was augmented with additional information (to be described later).

Procedure. Again, the task instructions were part of the Hypercard stack used to test the participants. The informational content given to the three groups in this experiment was based on the Examples group of the last experiment. That is, none of the groups in this experiment were given knowledge of syntax or goal.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Rather, all of them were given the screen of examples, but before they were shown that, an additional screen of information was presented to them. This screen contained information as to how the task was related to algebra, and the information differed in directedness between the groups. The labels used for the groups refer to the detail level of the algebraic hint given to the participants in that group. The least directed information given to the Algebra(Low) group read as follows:

> This task is like algebra. It is not a direct mapping, so do not get caught on any one manipulation. However, as you look at the examples and start solving problems, you will find it helpful to use your knowledge of algebra in figuring out the domain.

After reading this screen, the participant went on to the screen of examples, and then proceeded like the other groups in Experiment 1.

Another group of people, which I refer to as the Algebra(Intermediate) group, saw not only the paragraph above, but also this paragraph:

> There are basically 3 types of manipulations in this task. One is adding the same thing to both sides of equation. Another is canceling, and the last is eliminating the sign in front of the ℘ (which often has consequences for the right–hand side of the character string).

These two paragraphs were presented on the same screen, and like the Algebra(Low) group, once the people in the Algebra(Intermediate) group read through these paragraphs, they were presented with the screen of examples, and the experiment proceeded as in Experiment 1.

Finally, the people in the Algebra(High) group saw the same algebra information screen as the Algebra(Intermediate) group. However, each example on the following screen of examples was annotated with the corresponding algebraic symbols, much like the example presented in Table 3.2 (Appendix B contains the full list of annotated examples).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In all three conditions, the way participants interacted with the program as they were trying to solve problems was exactly the same as in Experiment 1. They could refer back to the examples screen, which for the Algebra(High) group contained additional information, as well as the text of the algebra hint.

Again, each participant was asked to solve 32 of each of the three types of problems (one–, two–, and three–step problems) for a total of 96 problems. Each participant had 2 hr with which to solve all 96 problems. There were 19 participants in the Algebra(Low) group, 12 in the Algebra(Intermediate) group, and 13 participants in the Algebra(High) group.

Results

Background and General Results

Table 4.1 contains summary information about the performance of participants in this experiment for easy reference, with the Examples group from Experiment 1 displayed to provide reference. No difference is detected in the SAT scores of the participants in the three groups ($F < 1$) when examining the groups as a whole, but when examining just the successful participants, a difference is detected ($F(2,33) = 3.68$, $MSE = 2663$, $p < .05$). A Newman Keuls test reveals that the Algebra(Low) group is significantly higher than the Algebra(High) group ($p < .05$). This difference between the aptitude of the groups, at worst, attenuated the predicted effect, since the Algebra(Low) group was expected to perform worst.

Preparation times. Examining both time to read the information given to the participants up front, and the time spent initially studying the examples, no differences were detected between these three groups ($F < 1$). However, when compared to the amount of time spent by the Examples group from Experiment 1, all three of these groups spent significantly more time, as shown by a Newman Keuls test ($p < .05$). The Algebra(Low) group spent 2.65 min initially studying the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 4.1
Algebraic Symbols At-a-Glance

	Algebra			
	High	Intermediate	Low	Examples
Self-reported math SATs	663[a]	689[a]	692[a]	655[a]
Reading Instructions (min)	4.44[a]	4.90[a]	5.14[a]	2.94[b]
Examining Examples (min)	2.24[a]	2.43[a]	2.65[a]	0.79[b]
Successful Participants	12 of 13[a]	12 of 12[a]	12 of 19[b]	12 of 23[b]
Self-reported math SATs	674[a]	689[a]	730[b]	683[a]
Example References	16.42[a]	24.58[ab]	45.00[b]	70.83[c]
Total Time (min)	62.77[a]	65.15[a]	67.03[a]	81.54[b]
First Block (12 problems)	19.49[a]	19.36[a]	23.89[a]	33.09[b]

examples and 5.14 min with all of the initial instructions, the Algebra(Intermediate) group spent a mean of 2.43 with the examples and 4.90 min with all the instructions, and the Algebra(High) spend 2.24 min with the examples and 4.44 min with all the instructions.

Successful and unsuccessful participants. However, as in Experiment 1, a distinction needs to be made between those people finishing the task and those who did not finish in the 2 hr time limit. Twelve participants completed the task in each of the three groups. Seven people did not learn the task in the Algebra(Low) group, and one person did not finish in the Algebra(High) group. Everyone finished in the Algebra(Intermediate) group. The proportion of successful participants between the Algebra(Low) and Algebra(Intermediate) group is significant ($p < .05$). A significant difference does not exist between the proportion of successful participants in the Algebra(Low) group and the successful participants in the Examples group from Experiment 1. The one person who did not complete the task in the Algebra(High) condition did solve

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

42 problems, and the 7 people who did not finish in the Algebra(Low) group made it to problem 35.3 on average. Looking at the initial instruction time measures examined in the previous paragraph, the people who did not finish the task did not differ on those measures from the people who did finish.

Usefulness of algebra hint. In Experiment 1 participants were asked if the task reminded them of anything. It was found that, for those people who learned the task, most people were reminded of algebra. At the end of this experiment in which people were to varying degrees explicitly told the task was based on algebra, people were asked if they felt that the algebra hint was beneficial in learning the task. In the Algebra(Low) group, 8 of the 12 people who learned task reported that the hint was helpful, and perhaps surprisingly, four of the people who did not complete the task said that the hint helped. Nine of the 12 people in the Algebra(Intermediate) stated that making use of the hint aided them in learning, and everyone in the Algebra(High) group, including the one person who did not finish, said it helped. In elaborating on how it helped, most people said it allowed them to more easily notice that things were being added to both sides and then being canceled, as well as clued them in to the fact that there may be inverse operators.

Learning

Accessing information. Across the three groups participants differed significantly on the number of times the example screen was referred back to ($F(2,33) = 3.64$, $MSE = 713.9$, $p < .05$). The Algebra(Low) group referred to that page a mean of 45.00 times, which was significantly different by a Newman Keuls test ($p < .05$) from the 16.42 times on average that the Algebra(High) group looked back. The Algebra(Intermediate) group referred back to that page a mean of 24.58 times, which does not differ from either of the other two groups. All three of these groups differ from the Examples group of Experiment 1 (who

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

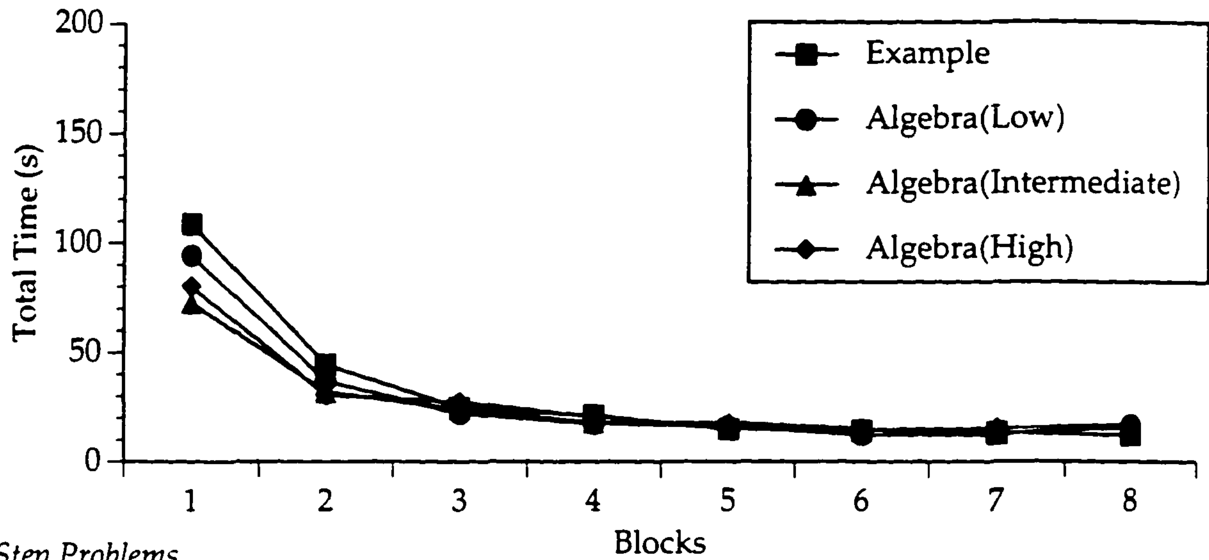
referred back to the examples screen 70.83 times). No one in these three groups referred back to the page with the hint as to how the task was related to algebra. The seven people who did not finish in the Algebra(Low) group referred back to the examples an average of 74.32 times.

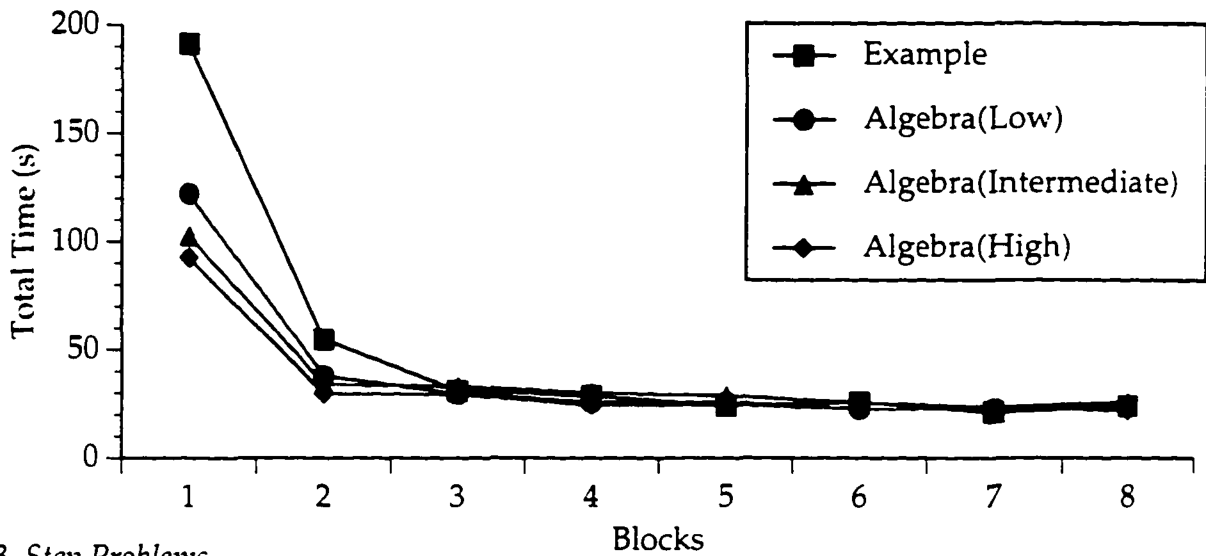
Completion time. The three groups did not differ significantly in the total time it took them to solve all 96 problems ($F < 1$). However, there was a slight suggestion that the more detailed the hint, the faster learning took place. The Algebra(Low) group spent 67.03 min on average solving all the problems, the Algebra(Intermediate) group spent 65.15 min, and the Algebra(High) group took 62.77 min. In comparing them to the Example group from the last experiment, which took a mean of 81.54 min to solve the problems, all three groups did significantly differ by a Newman Keuls test ($p < .05$). Figure 4.1 plots the performance of the three groups in this experiment, using the Examples group from Experiment 1 as a comparison, on all three types of problems. As in the graph of Figure 3.1, the groups did noticeably differ during the first block, that difference was attenuated during the second block, and by the third block all groups were performing equally on subsequent trials. Therefore, any difference in time to learn the task between the three groups occurs very early in the learning process. As in Experiment 1, performing an ANOVA on only the first block of trials, one does see a significant difference ($F(3,44) = 2.93$, $MSE = 50958$, $p < .05$), and a Newman Keuls post-hoc test revealing that the Examples group differs from the Algebra(High) group, but no other pairings are significant at the $p < .05$ level.

Errors

Table 4.2 presents the error data from this experiment in a manner similar to Table 3.5. The Examples group data from Table 3.5 is presented here for comparison purposes. The three groups did not differ significantly in the total

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.





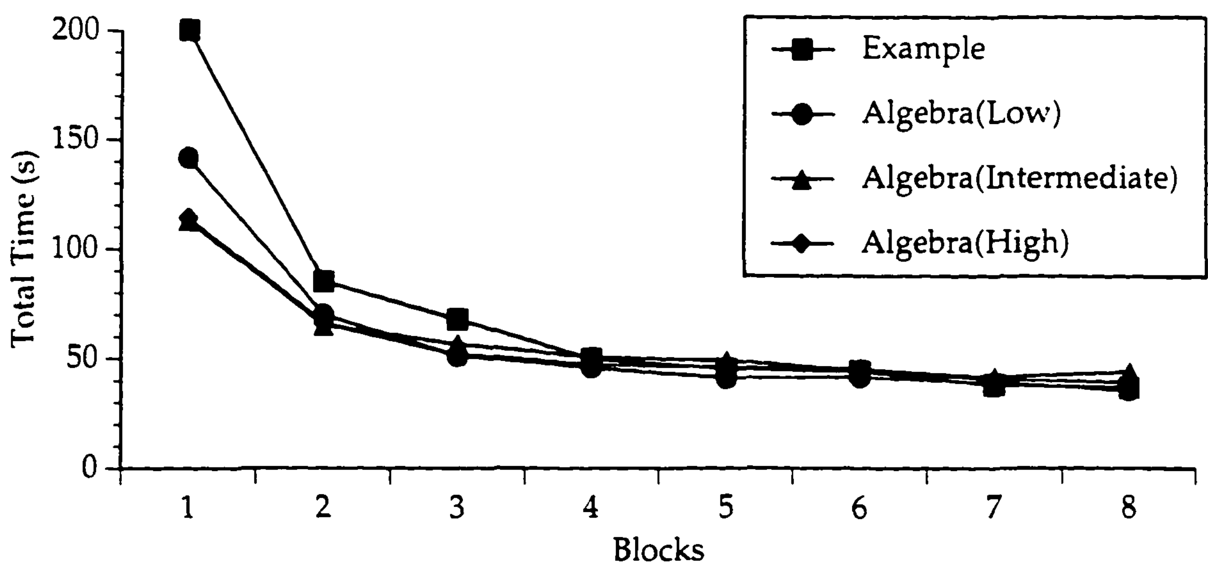


Figure 4.1: Overall time by block for each problem type (Experiment 2)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 4.2

Experiment 2 Errors

Examples Only

	Syntax	Semantics	Total
Addition	2.75 (8%)	9.42 (28%)	12.17 (36%)
Cancellation	0.50 (2%)	1.5 (4%)	2.00 (6%)
Sign Elimination	3.17 (9%)	16.50 (49%)	19.67 (58%)
Total	6.42 (19%)	27.42 (81%)	33.83

Algebra(Low)

	Syntax	Semantics	Total
Addition	2.25 (7%)	6.83 (21%)	9.08 (28%)
Cancellation	0.75 (2%)	1.42 (4%)	2.17 (7%)
Sign Elimination	1.50 (5%)	19.67 (61%)	21.17 (65%)
Total	4.50 (14%)	27.92 (86%)	32.42

Algebra(Intermediate)

	Syntax	Semantics	Total
Addition	7.17 (14%)	14.92 (29%)	22.08 (43%)
Cancellation	0.83 (2%)	1.83 (4%)	2.67 (5%)
Sign Elimination	1.92 (4%)	24.25 (49%)	26.17 (51%)
Total	9.92 (19%)	41.00 (81%)	50.92

Algebra(High)

	Syntax	Semantics	Total
Addition	4.83 (11%)	4.83 (11%)	9.67 (22%)
Cancellation	2.50 (6%)	2.42 (5%)	4.92 (11%)
Sign Elimination	2.42 (5%)	27.92 (62%)	30.33 (68%)
Total	9.75 (22%)	35.17 (78%)	44.92

number of errors they produced ($F < 1$). Again, one must keep in mind that for the Algebra(Low) the 12 people represented in the table come from a group of 19 people, whereas the 12 people in the other two groups are either all the participants in that group (the Algebra(Intermediate) group) or all but one of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

participants in the group (the Algebra(High) group). However, looking at the top 50% of participants in each group (i.e., 10 of 19 participants in the Algebra(Low) group, 6 participants in the Algebra(Intermediate) group, and 7 participants in the Algebra(High) group), in terms of least number of total errors, one does find a difference between the groups ($F(2,20) = 4.55$, $MSE = 90.83$, $p < .05$), with a Newman Keuls test showing that this subset of the Algebra(High) group made more errors than the other two groups ($p < .05$). These Algebra(High) people made an average of 29.3 errors, whereas the Algebra(Low) people made a mean of 19.8 errors, and the Algebra(Intermediate) group 12.8 errors. Both the Algebra(Low) and the Algebra(Intermediate) groups had participants who did extremely well (i.e., made less than a dozen errors), whereas the participants in Algebra(High) group all did roughly the same, making around the mean number of errors.

As in Experiment 1 the error profiles, in terms of the percentages, are different between the groups, as evidenced in Table 4.3 which shows the correlations of those percentages with one another. Similar amounts of syntactical errors were made between the groups, with most errors being semantic in nature. The pattern of errors between the three types of steps are most similar between the Examples group and the Algebra(Intermediate) and Algebra (Low) groups, and the Algebra(Low) group and the Algebra(High) group. It is important to keep in mind that the 12 people reported in the Algebra(Low) group are, in some sense, the people who got the most out of the

Table 4.3
Correlations in Error Percentages

	Examples	Algebra(Low)	Algebra(Inter.)
Algebra(Low)	.95	—	—
Algebra(Inter.)	.96	.91	—
Algebra(High)	.81	.95	.79

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

algebra hint—enough so to make them similar to the Algebra(High) group, where many of the connections between this task and algebra were laid bare. The Algebra(Intermediate) group made the highest percentage of errors on the addition step, where knowing the inverse operators is most important. The idea of inverse operators is made apparent in the Algebra(High) group (i.e., seeing that ® is paired with + and ♥ is paired with –), and as previously stated, the participants in the Algebra(Low) group are the ones who quickly made that connection based upon the algebra hint. Like the Syntax(Hint) group in Experiment 1, the Algebra(High) and Algebra(Low) groups had most difficulty with the sign elimination steps (around 70% of the total errors).

Sign elimination errors. Examining the particular types of sign elimination errors, like in Experiment 1, one sees slight differences between these three groups. First, the people in the Algebra(Low) and Algebra(Intermediate) groups made a similar percentage of errors in confusing the ® and © elimination rules (the Algebra(Low) group made 48 errors of that type, or 34% of applicable errors, and the Algebra(Intermediate) group made 26%, or 47 total). The Algebra(High) group made 19% (34 total) of their errors on these types of problems. Another differentiating error mentioned in Experiment 1 was perseverating in switching the operators when eliminating the #, as suggested by the misleading example, not inverting the related operators. The Algebra(Low) group made that error 36 times out of 88 total errors (41%) on # elimination steps, whereas the other two groups made the error much less: the Algebra(Intermediate) group 26 times out of 103 (25%) and the Algebra(High) group 28 times out of 148 (19%).

Discussion

In all its forms, the algebra hint aided people in learning the task in comparison to the Examples group from Experiment 1, which supports the claim of the Prior Knowledge Contribution:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2) One of the strongest predictors of success for learning Symbol Fun was if the learner was able to access and use their knowledge of algebra.

In its least detailed form, only four additional lines of text, 12 of 19 people learned the task, in comparison to 12 of 23 in the Examples group. While the percentage of people who learned the task is not statistically different, the time it took the people who did learn the task (i.e., the people who truly grasped the hint) was significantly quicker. This slight hint allowed people to access previously learned knowledge which they may or may not have accessed otherwise.

With the addition of four more lines of text, the text seen by the Algebra(Intermediate) group, resulted in everyone in that group being able to learn the task in the allotted 2 hr. Those additional four lines of text contained information which would limit the search space, the possible transformations and manipulations allowed in the task, for those people. The additional lines clearly casted the problems in terms of the three basic manipulations—addition, cancellation, and sign elimination—and enabled the participants to concentrate on those types of potential rules. In sum, those lines allowed the participants to highlight the algebraic knowledge most necessary to learn the task and to not concentrate on the other aspects of algebra not necessary.

Finally, the Algebra(High) group actually saw a mapping between this task and algebra, which resulted in all but one person learning the task and, for the people who did, a suggestion that the learning was quicker than in the other two algebra hint groups, particularly across the first 24 problems. The mapping, on top of the hint seen by the Algebra(Intermediate) group, provided additional information with which the participants in the Algebra(High) group could use to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

learn the task, but apparently the four lines of text was the more crucial piece of information in learning the task.

The algebra hint was as effective as it was because it allowed the problem solvers to map existing algebraic knowledge (e.g., adding the same thing to both sides of an equation, inverse operators, etc.) onto learning the new task. Initially studied by Thorndike (1906; Thorndike & Woodworth, 1901) and then updated by Singley and Anderson (1989) to fit into Anderson's ACT theory (1993), the *identical elements theory of transfer* provides an explanation as to why and how this happens. In as much as existing knowledge, both declarative and procedural, overlaps with the knowledge needed to perform the new task, transfer will result. The more overlap that exists between the two tasks, the greater the transfer. In all conditions, the hint that the task was based on algebra allowed the participants to consider how their algebraic knowledge could be applied to this new task. The hint given to the Algebra(Intermediate) and Algebra(High) group as to what sort of manipulations were involved in this task allowed a narrowing of their consideration as to how their existing knowledge of algebra could be applied. Finally, the examples screen seen by the Algebra(High) group made the mapping between their existing algebra knowledge and knowledge of this task extremely explicit.

In analyzing the errors that people made in Experiment 1, it appeared that the participants in the group with the most information (the Syntax(Hint) group) were more theory driven than the group with the least information (the Examples group), who were more data driven. In as much as people made use of the algebra hint in this experiment, everyone should have been operating with a "theory," or set of related knowledge structures from algebra, of how the task should work as they were attempting to learn it. Therefore, the better this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

additional information, the better the learning, as stated in the Over Specificity Contribution:

3) Lack of adequate syntactic knowledge causes the analogy mechanism to build over-specific rules from examples.

This is roughly what one sees in the results of this experiment. The Algebra(Low) group tended to try to over-generalize the rule set, making the error of interchanging the ® and © sign elimination rules 34% of the time on errors involving those steps, whereas the Algebra(High) group, who could see the mapping of the symbols and could perhaps better guess at the underlying rules (e.g., for © elimination, switch the two operands) made those errors 19% of the time. Also, the Algebra(Low) group repeatedly made the error of switching the operators for # elimination (41% of their errors on that step), whereas the Algebra(High) group, who could see the mapping between inverse operators, more quickly learned that that was not correct (25% of their errors). The Algebra(Intermediate) group was someplace in between with their understanding—confusing the ® and © rules 30% of the time, but only making the # elimination error 18% of the time when they made on error on that step.

Based on the results of these two last two experiments, an adequate model of how people learn this task, and what pieces of information are necessary for people to fully understand the task, can be constructed. The following chapter discusses an ACT-R model of people learning this task.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5

The Model—ACT-SF

This chapter details an ACT–R model, ACT–SF, of people learning Symbol Fun, as examined and analyzed in the previous two chapters. An important distinction within the ACT–R architecture is between declarative knowledge, knowledge of facts (e.g., "Washington DC is the capital of the United States") and procedural knowledge, knowledge of how to perform actions (e.g., adding numbers together). One of the claims of the ACT–R theory is that all knowledge has declarative origins. That is, the only way new procedural knowledge, in the form of production rules, enters the system is by the process of analogizing from the current goal to some previous declarative knowledge. This mechanism operates by forming an analogy from examples stored in declarative memory to the current goal. Also, this mechanism accounts for how generalizations arise from prior knowledge. The analogy mechanism is built into the ACT–R architecture.

The ACT–SF model initially contains no procedural knowledge (i.e., no productions) that describe how to perform the manipulations required within the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Symbol Fun task (the model does contain two productions that perform "housekeeping" tasks). These productions are learned via the analogy mechanism, based on its initial declarative knowledge.

The goal of the ACT–SF model is to provide a full account of how people learn the Symbol Fun task, and then to compare the predictions that the model makes against participants' performance in the previous and also the later experiments. Also, this model serves as a test of ACT–R's analogy mechanism, and, to some degree, ACT–R's claim that all knowledge starts off declaratively, since that is the way the analogy mechanism works. One of the ways this was examined was by removing or modifying the model's declarative knowledge, and this will be discussed in the last part of this chapter. By such a process, human failures at learning the task were modeled.

The model which will be described now is referred to as either ACT–SF or the "Informed Model." Initially it only has declarative knowledge—that is, no procedural knowledge as to the manipulations needed to perform the task—but that knowledge is represented in such a way as to allow the best, most accurate learning of that procedural knowledge. This initial knowledge would be extremely similar to knowledge problem solvers had in the Syntax(Hint) group described in Experiment 1 ("Syntactic Symbols"): a representation of how the strings are parsed, and knowledge of inverse operators. All of this knowledge is represented within ACT–R's declarative memory.

As it stands now, ACT–SF is only a qualitative model. It does not match any quantitative data. Rather, it models the acquisition of the procedural knowledge required to perform Symbol Fun in a manner consistent with the way humans do, as discussed in the preceding two chapters. It does not model the slower, almost stage–like acquisition of this procedural knowledge as seen in some of the groups (e.g., the Example Only group of Experiment 1). Parallels

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

between the human data and ACT–SF will be highlighted in the next sections when appropriate.

Representation in ACT–SF

Given that the declarative representation of the character strings are the most important aspect for ACT–SF to learn the underlying rules of Symbol Fun, a discussion and an example of that representation will be presented here. The last half of this chapter contains a more in–depth discussion of this representation.

ACT–R's analogy mechanism works by comparing the start state of a problem to its solution state. These start and solutions states are stored as separate declarative memory structures. Often there are constraints placed on how the solution state can be reached (e.g., certain other declarative structures must be accessed, or certain values must be generalized over). The start and solution states, as well as any constraints, are recorded within declarative memory structures referred to as *dependencies*. Dependencies are predefined working memory structures within ACT–R that already the contain the positions ("slots") needed to record pointers to the start and solution states, and any constraints.

ACT–R chooses the examples it attempts to analogize with based on the activation of these dependency working memory elements (WMEs). Dependency WMEs with higher activation (e.g., those that have been most useful in the past or those that have been more strongly encoded) are chosen first. If the system's current goal matches the goal type of the start state that the chosen dependency WME points to, then an analogy is attempted. If the production that is created has no instantiation with the current goal, then the analogy mechanism will pick another dependency WME to test.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

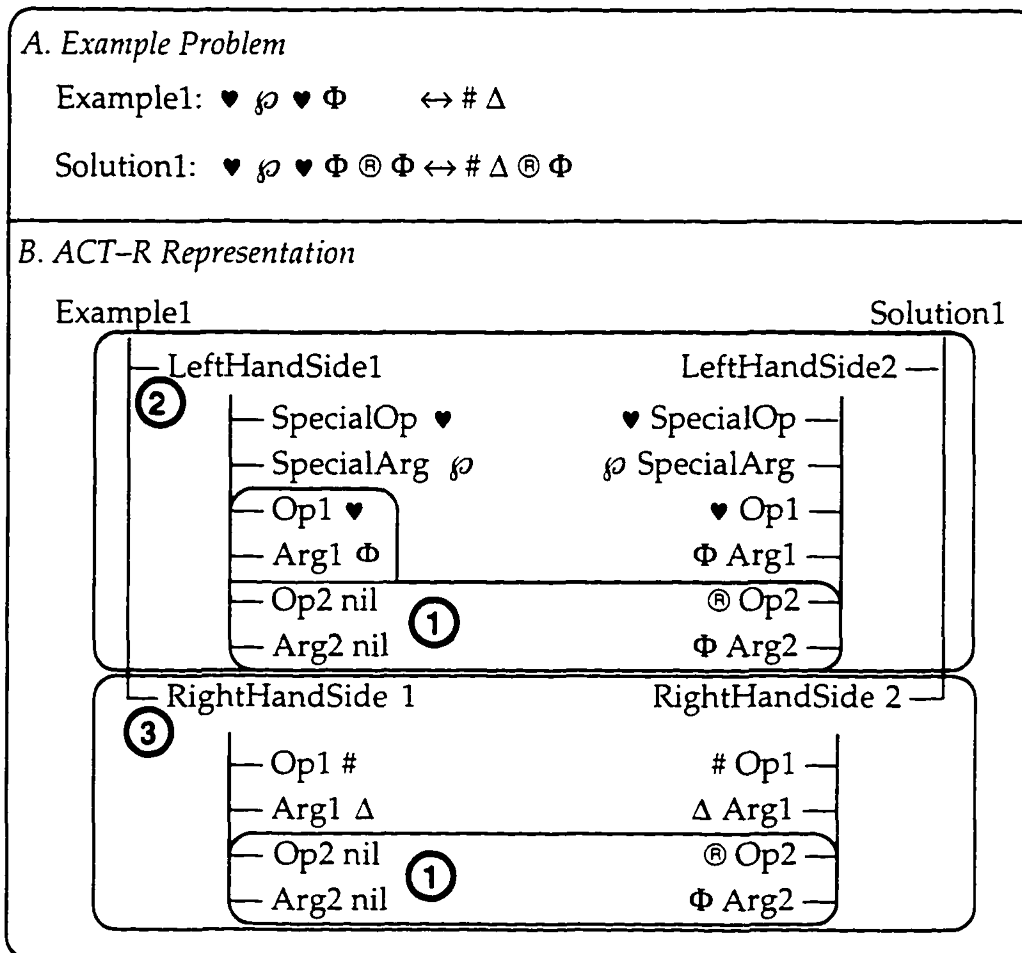


Figure 5.1: ACT–SF's Representation with Dependency Structure Highlighted

Figure 5.1 provides an example of how character strings are represented within ACT–SF. Panel A shows the two lines being represented and Panel B provides a schematic for how those two lines are represented within ACT–R's declarative memory. Each character string is composed of two parts, a right side and a left side. These two sides are then broken down into parts which contain positions for each possible character that could occur on that side. Both the right and left sides also have positions for a second operator and argument. When a position is not filled in, its value is *nil*. The circled numbers highlight the way three dependency WMEs have recorded how this example is marked up. Each dependency corresponds to one possible production (i.e., transformation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 5.1
Example for Use in Analogy

Step #	Symbol Fun	Corresponding Algebra
Given	♥℘♥Φ↔#Δ	– x – A = * C
1	♥℘♥Φ®Φ↔#Δ®Φ	– x – A + A = * C + A
2	♥℘↔#Δ®Φ	– x = * C + A
(Answer) 3	℘↔#Δ♥Φ	x = * C – A

between states), and this example will be used later to illustrate how these productions are actually created.

Operation of ACT–SF

When the first problem is presented to the system to solve, no productions are available with which to match. Therefore the analogy mechanism is invoked in order to try to induce the correct transformation. What follows is a description of that induction process as the model tries to solve the problem:

♥℘©Δ↔®Ω

using an interpreted example exactly like the problem presented in Table 3.2, and reproduced in Table 5.1.

ACT–SF has stored the eight examples all participants had available, as shown in Table 3.3, and they are marked–up (via the underlying declarative representation pictured in Figure 5.1 and the dependency WMEs) to allow the ACT–R analogy mechanism the opportunity to learn the best set of productions that it could learn in order to do the task. The actual declarative memory structures are listed in Appendix C, and the resulting productions of this process are shown in Appendix D. The following paragraphs give an illustration of that process.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

An Illustration

To begin, and as stated above, consider the situation where the current problem is ♥℘©Δ↔®Ω and the first line of the reference example is ♥℘♥Φ↔#Δ. At each step of this illustration, three things may be discussed. First, in all cases the declarative representation that gives rise to the production will be discussed and the production shown. Second, any predictions which follow from this representation and production will be considered. Finally, if any supporting protocol or other data supports the prediction, it will be presented.

Production 1: Appending the same string to both sides. This transformation is captured by a dependency WME that the line that follows ♥℘♥Φ↔#Δ is ♥℘♥Φ®Φ↔#Δ®Φ. Or, to put that in perhaps an easier to understand form, one could represent the situation as follows:

Current problem:	*Current example:*
♥℘©Δ↔®Ω	♥℘♥Φ↔#Δ
	⇓
	♥℘♥Φ®Φ↔#Δ®Φ

What the model needs to do is infer the production behind the action indicated in the example. This transformation, according to the way the example is marked–up and recorded in the dependency WME, is accomplished using two subgoals, one to add the proper thing to the left side, and the other to add the same thing to the right side. This is illustrated in Figure 5.1 by the circled ones (see Appendix C to examine how this is accomplished in the code). Therefore the example is marked in a way to make those subgoals explicit, and then those subgoals are marked so that the right side of the problem statement goes to the right side of the first line in the solution and that the left side of the problem statement goes to the left side of the first solution line. A production is created that embodies the creation of these two subgoals:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

IF	the current goal has a left side and a right side AND the left side has *op1* and *con1*	(P1)
THEN	set a subgoal to append to the left side based on *op1* and *con1* AND set another subgoal to append to the right side based on *op1* and *con1*	

Notice that the subgoals also store the important aspects of the left side necessary to the addition step, the operator and constant. This is important for the subgoal which transforms the right side, since it does not have direct access to the contents of the left side. This production is now applied to the current problem, and so the system now has two subgoals it will have to solve. What ACT–R's analogy mechanism needs to do next is to figure out how these transformations occur.

Production 2: Append an inverse–operator argument string to one side. The next transformation, appending something to one side of the character string, is indicated in Figure 5.1 by the circled two (for the left side) and the circled three (for the right side). Examining the two left sides of the current example shown above, the first four symbols are the same, and stay in their same positions. However, the fifth symbol in the solution line does not appear in the left side of the problem statement. ACT–R must use its additional declarative knowledge to determine the origin of that symbol. Since the model has knowledge of the inverse operators, and the ♥ appears earlier in the line, the dependency WME records that ® may aid in making the analogy, and the analogy mechanism encodes in the created production that the ® appears because it is the inverse of ♥. Lastly, the sixth symbol in the answer line, the Φ, also does not have a direct match in the corresponding slot of the problem string, but since that symbol is the same as one that appears elsewhere in the line, the model assumes that that will always be the case. This production is now created:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

IF the current goal is to append something to one side (P2)
AND the goal is based on *op1* and *con1*
AND *op2* is the inverse of *op1*
THEN append *op2 con1* to this side
AND pop that subgoal

Prediction. Both this production and the first one will apply to all addition steps for two– and three–step problems; it is not specific to the case where a ♥ appears as the third symbol, and nor is it specific to three–step problems. Rather, they will apply when any operator appears in the first operator position and nothing has already been added. Furthermore, these productions will apply to adding symbols to both the left– and right–hand side of the production.

Supporting data. When participants figure out the right rule for adding to both side of the equation, they do indeed generalize to all the operators and to both two– and three–step problems. Appendix E gives a sample protocol of a typical participant in the Examples only group (a successful learner). Across Problems 10 and 11 he acquires the rule for adding the inverse operator to both sides of the string, and applies it equally afterwards to any operator and to both two– and three–step problems.

Production 2 fires again. As before, P2 is applied to the current goal of appending to the left side of the character string. That goal is then popped. Figuring out what to do the right side now becomes the top goal of the system. Since P2 can apply to this current goal, it is applied to the problem's right side, that subgoal is popped, and the system has successfully transformed the current problem statement into the next correct line in the problem's solution:

♥ ℘©Δ#Δ↔®Ω#Δ

Production 3: Deleting symbols on one side. The above character string is now the system's goal, and, since no productions apply at this point either, the process of selecting an example to analogize with begins again. Though it is not constrained to, let us suppose that the system picks the second line of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

previous example (what was referred to as the solution line in the previous paragraphs) to use as the new reference example. This line, ♥℘♥Φ®Φ↔#Δ®Φ, has marked that ♥℘↔#Δ®Φ is the next correct line, and so that line becomes the new solution line:

Current problem:	*Current example:*
♥℘©Δ#Δ↔®Ω#Δ	♥℘♥Φ®Φ↔#Δ®Φ
	⇓
	♥℘↔#Δ®Φ

To get from the reference example to the solution line, something only needs to be changed on the left side, and the example is marked as such. Furthermore, the transformation is extremely easy—the first two characters are the same, and then the next four characters are dropped, and the right side remains the same. The production that gets created to account for this change does not check that the operators are inverses:

IF the current goal has a left side and right side AND both operator and operands slots on the left side are filled in (P3)
THEN drop the four rightmost symbols on the left side

Prediction. Participants do not need to have knowledge of inverse operators, and will simply think of this transformation as deleting four symbols, not canceling them. It also applies equally to two– and three–step problems.

Supporting data. It was with these cancellation steps that people had the least trouble, with only 10% of their errors coming from this transformation. Listening to people give verbal protocols at this task, across all conditions, it is evident that when people first do this step, they do not think of it as canceling (i.e., that two of the symbols being removed are inverses of one another), but rather that the symbols are merely dropping out. Problem 3 in Appendix E contains a good description of the acquisition of such a cancellation rule.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As states, this cancellation rule does not depend on knowledge of inverse operators. All 11 of the unsuccessful participants in Experiment 1's Examples only group, except for one, after 4.0 problems on average, learned this cancellation rule. None of these 11 participants learned the inverse operators (evident from their data files and exit interviews). Using the protocol participants as a representative sample, it appears the rule they were learning was just to drop the four symbols.

Production 4: Sign Elimination. The above production (P3) gets applied to the current goal, and the next step in the problem's solution is produced and becomes the top goal:

♥ ℘↔®Ω#Δ

Since the lead symbol (the ♥) is the same for the current problem and the example, and this is the only example which has a ♥ out front, the system will continue to use the same example, where ℘↔#Δ♥Φ is stored as the line that comes after ♥ ℘↔#Δ®Φ:

Current problem:	*Current example:*
♥ ℘↔®Ω#Δ	♥ ℘↔#Δ®Φ
	⇓
	℘↔#Δ♥Φ

Similar to the change between the problem statement and the first line in the problem's solution, transformations need to be done to both the left and right sides, and so the example is again marked to create two subgoals, one that manipulates the left side and one that changes the right side:

IF the current goal has a left side and a right side (P4)
AND the left side has only a front operator and the script–p
THEN set a subgoal to delete the operator on the left side
AND set another subgoal to do something to the right side
based on the front operator

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Notice that the frontmost operator is stored, so that the right side knows the proper transformation to perform. Applying this production to the goal results in the creation of two subgoals.

Production 5: Deleting the front operator. As in the transformations that occurred for the first step, the first subgoal is marked in such a way to link the left side of the reference example to the left side of the solution. Here, the difference is that the ♥ in front of the ℘ is dropped:

IF	the current goal is to do something to the left side AND there's only an operator in front of script–p and the script–p itself	(P5)
THEN	drop that operator AND pop subgoal	

Prediction. This production is another that is not operator specific—it will drop any operator that appears in front of the script–p.

Supporting data. In Appendix E, on the second problem the participant dropped the initial symbol, and on the fourth problem (the second problem which had a sign elimination step) specifically mentioned that "they lose the very leftmost thing."

Production 6: Transforming the right side. After that production is applied, the second subgoal remains, which links the right side of the reference example to the right side of its solution. In the case of ♥ and # elimination, the proper thing to do depends on what appears on the right side. The related operators need to be inverted, whereas the non–related operators remain as is. In the case of ® and © elimination, however, the transformations are more straightforward. Since ♥ elimination depends on what operators are on the right, two more subgoals need to be created, one for each right side operator/operand pair, and so a production such as this created:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

IF the front operator is a minus (P6)
THEN set a subgoal to invert the first operator/operand pair recording that the front operator is a minus
AND set another subgoal to invert the second pair recording that the front operator is a minus

Prediction. This production is particular to the operator out front. Three other productions will need to be created to handle the other three operators.

Supporting data. To correctly learn the problem set, the participants must come to this conclusion. One can see this very clearly with the protocol participant in Appendix E, problem 9 (though he is cueing off the wrong symbol).

Productions 7 and 8: Inverting symbols on the right side. In the case of the present example, the output of one of the two created subgoals will be a production that inverts the operator:

IF the current goal deals with a particular front operator and an operator/operand pair (P7)
AND the front operator is related to the pair
THEN invert the pair's operator
AND pop subgoal

and the output of the other subgoal is a production that does not invert the operator:

IF the current goal deals with a particular front operator and an operator/operand pair (P8)
AND the front operator is not related to the pair
THEN leave the pair the same
AND pop subgoal

Prediction. Neither of these productions are location specific—the operator/operand pair could either be the first or second pair that appear on the right side.

Supporting data. As will be described in Experiment 4 (Chapter 7, "General Symbols"), participants are quite good at abstracting over these positions, and so this is similar to what participants actually do (e.g., see Appendix E, problems 12, 18, and 19).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Conclusion. After applying the last production, the system now has the final line in the initial problem's solution, that the last line of ♥ ℘©Δ↔®Ω should be ℘↔♥Ω#Δ. There were eight productions created in order to make that transformation. On subsequent two– and three–step problems and problems that involve ♥ elimination, the model has available to it these productions to use. When these productions apply, however, they may or may not fire, depending on their strength. The analogy mechanism is in constant competition with the production matching process, and if the strength of the matching productions is not high enough, the analogy mechanism will attempt to execute. If the created production is identical to an already existing production, the identity will be noted, and strength will be added to that production. In such a way, the analogy mechanism will create and strengthen the productions so that eventually the problems will be solved solely by the application of productions.

Given the declarative representation used in the Informed Model, a minimum of 13 productions need to be created for the model to solve all possible problems. These are detailed in Appendix D, in which a run of the model on multiple problems is given. It is possible, if not likely, for more productions to be created due to spurious relations between the symbols in the character strings. This will be discussed in the next sections.

ACT–SF Model Discussion

The Informed Model Representation

The Informed ACT–SF Model which was just illustrated captures the important qualitative aspects of people successfully learning this task. People in the Syntax(Hint), Algebra(Intermediate), and Algebra(High) groups arguably have such a representation at the outset of starting the task, or at lease quickly acquire such a representation. Essentially all of the participants in these groups are successful at learning the task.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This chapter continues by enumerating the exact features of this representation, and how it maps on to the model. For each of the five points listed below, a description of how each point is realized in the model and a short discussion of the evidence that successful participants have such a representation is given. After those five representational points, the chapter continues with discussing how such representations can be established by those participants in conditions which did not start out with the best representation, and furthermore the consequences for when such representations are not established, as shown by both the model and participants. Finally, this chapter concludes with a discussion of a few errors commonly made by participants for which ACT–SF currently does not model.

The major representational features of the model are:

1) Definite left and right sides
2) Each line in a problem's solution is separable
3) Within a line, the characters are separable
4) Inverse operators
5) Sign elimination depends on the operator being eliminated

Definite left and right sides. As shown in Figure 5.1, the model clearly divides each character string into a left and a right side. The character strings are represented as a hierarchical structure, with each string consisting of a left and a right side, and then both of these sides formed of individual symbols. For participants, the double arrow serves as a strong initial indicator that perhaps the string should be divided at that point. Many participants in the Examples group either make that assumption from the start, or soon do so in their learning (this is evident from their data files, and also from the participants that protocols were collected from). Once that assumption is made, most of the syntactic errors disappear.

Each line in a problem's solution is separable. The model will consider the last step of a three–step problem to help in solving a one–step problem (and vice

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

versa), as well as consider different problems in formulating a multi–step solutions. Participants who are clearly on their way to mastering the rule set do this as well. This must be the case, since if given a one–step problem which involves ♥ elimination, using either of the one–step examples on the example screen (Examples 3 and 5; see Table 3.3) would result in an error.

Within a line, the characters are separable. Within a character string, the model considers each symbol individually, and it is not critical for an exact match to occur between the current problem and the example it selects to perform an analogy. Participants who have not yet started representing the strings as such restrict their considerations or clump symbols together to try to find a match. For example, if the right side of the problem contains a ®Δ, they will try to find an example with a ®Δ in it, hopefully on the right side, but may consider an example that contains it on the left as well.

Inverse operators. Perhaps the most important piece of information in representing this task, in terms of being able to learn all the correct rules and finish the task, is the inverse operators. ACT–SF is given this at the outset, as are the participants in the Syntax(Hint) group. Participants in the Algebra(High) group also are likely to infer this information from the first time they examine their annotated examples page (as in Appendix B). Participants who were not given this information and did not learn it on their own, simply did not learn the task.

Sign elimination depends on the operator being eliminated. This last representational item concerns itself with the sign–elimination steps. Participants would often approach the sign–elimination steps with the idea that there was only one, perhaps two, manipulations that were done to the right side (e.g., leave it the same or swap the constants). However, as they became more experienced with those steps, they began to realize that each of the four leading operators

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

meant a different transformation needed to be applied to the right side. Two of those operators, the ♥ and # elimination steps, require additional, indirect, knowledge beyond what is contained in the character string (i.e., knowledge of the inverse operators). Obviously people must first acquire this inverse operator knowledge before they can fully appreciate the correct rules for performing ♥ and # elimination.

Degrading the Representation

When participants lack a representation which takes into account the five points listed above, what are the consequences and how does the participant learn such a representation? Under an analysis of the protocols, it appears that the five points of representation come on–line in the order mentioned. As mentioned already, Appendix E contains a protocol of a successful participant in the Examples group from Experiment 1, and one can see in this protocol such a progression. The discussion that follows centers mostly on that particular group (the Examples only group of Experiment 1), since that is that group that started off with the least amount of information, and so provides for the clearest picture of how this information can come on–line. The next section contains a short discussion of a second model that was created which degraded points 1 and 3 from the last major section (*Definite left and right sides* and *Within a line, the characters are separable*). A discussion of degrading points 4 and 5 follows (*Inverse operators* and *Sign elimination depends on the operator being eliminated*).

ACT–RC. A second, simpler model was created that did not initially know about the difference between operators and constants, and that learned them by a variation of the rational categorization algorithm (Anderson, 1991). Except for learning that the lines are separable, this model was equivalent to removing the parsing knowledge discussed previously (definite left and right sides and separable characters). In short, this model worked by comparing across many

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

different character strings, extracting which symbols appeared in which positions most often. What it learned was that Greek symbols always appeared in a particular set of positions, four operators appears in another set, and one position always contained the double arrow. This model never fully learned the task, and so could be compared to those participants who did not learn the task as well. This unsuccessful model took considerably longer, in terms of number of problems attempted to solve and examples referred, to attain the same proficiency as the unsuccessful participants. There is still knowledge that participants have that is not being captured by the models (e.g., previous knowledge of Greek symbols), and which would be challenging to model, but beyond the scope of the current considerations.

Inverse operators. As previously alluded to, it is learning about the inverse operators that was a major determiner if a person in the Examples group successfully learned the task. All 12 people who learned the task in that group acquired the inverse operator knowledge (apparent not only from their data files, but also from the exit interview), but none of the 11 people who did not complete the task did so (again, extremely apparent from the data files and exit interviews). Rather, all except for 1 of the 11 people learned to separate the character strings into left and right sides, but failed to learn the idea of inverse operators. In observing their mistakes on the addition steps, where knowledge of inverse operators is critical, they obviously knew they had to add an operator and a constant to both sides of the string, but did not know which operator to add. This is apparent from the protocol in Appendix E over the first 8 problems. It was on Problem 10 that he stated clearly the relationship between ♥ and ®. Prior to that, the participant was adding any operator and repeating the constant to both side of the character string.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 5.2
The Correct Production and Its Over–Specific Counterpart

```
(p change-production47
   =subgoal6r2-variable>
      isa change
      operator =+-variable
      argument =b-variable
      string =right6-1-variable
      result nil
   =+-variable>
      isa operator
      inverse =--variable
   =right6-1-variable>
      isa expression
      specop =blank1-variable
      specarg =blank2-variable
      op1 =/-variable
      arg1 =c-variable
      op2 nil
      arg2 nil
 ==>
   =right6-2-variable>
      isa expression
      specop =blank1-variable
      specarg =blank2-variable
      op1 =/-variable
      arg1 =c-variable
      op2 =--variable
      arg2 =b-variable
   =subgoal6r2-variable>
      result =right6-2-variable
   !Push! =right6-2-variable
   !Pop!
   !Pop!)
```

```
(p change-production5
   =subgoal1r2-variable>
      isa change
      operator =+-variable
      argument =a-variable
      string =right1-1-variable
      result nil
   =right1-1-variable>
      isa expression
      specop =blank1-variable
      specarg =blank2-variable
      op1 =*-variable
      arg1 =b-variable
      op2 nil
      arg2 nil
  ==>
   =right1-2-variable>
      isa expression
      specop =blank1-variable
      specarg =blank2-variable
      op1 =*-variable
      arg1 =b-variable
      op2 -
      arg2 =a-variable
   =subgoal1r2-variable>
      result =right1-2-variable
   !Push! =right1-2-variable
   !Pop!
   !Pop!)
```

If the inverse knowledge is taken out of the Informed ACT–SF Model, the model becomes quite similar to these participants who did not learn the task. Consider the productions displayed in Table 5.2. The one on the left is the same as `change-production47` shown in Appendix D. The production on the right was created from a version of ACT–SF with the inverse operator knowledge excised, and is similar to the rule the participant was considering in his protocol in Appendix E for Problems 4 and 6. The production on the right differs from the left one in that it does not figure out the relation between the operators (indeed, it cannot figure out the relation), but will always add a minus sign and repeat the operand when adding the same thing to both sides of the equation—very similar

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to what the protocol participant was doing. The protocol participant does figure out this inverse relation by at least Problem 10, but the unsuccessful participants never do. They continue to think that it is specific things that you add. Indeed, some participants (7 of the 24 total protocol participants across all conditions) did believe that some variant of this rule was the correct rule at some point during their learning.

Currently the model has no way of inducing this inverse relationship between operators on its own. In the model, this would correspond to placing the relevant information into the proper dependency WME. Perhaps in some instances participants learned this knowledge by borrowing from their knowledge of algebra and arithmetic, but in the three protocols collected from the Examples group from participants who successfully learned the task, it appears that this knowledge comes about from trying to figure out where the additional operator and constant comes from, and comparing across examples to see that the ® and ♥ occurred together and that the # and © occurred together.

Sign elimination depends on the operator being eliminated. Once knowledge of inverse operators has been gained, all participants who gave verbal protocols eventually learn all the sign elimination steps. Indeed, some participants who did not learn the inverse operators had some idea of the proper manipulations for these sign elimination steps, but obviously not the correct ones for # and © elimination. Very often these individuals had not associated the proper thing to do with the leading operator of the character string. This state of affairs can be represented in the model by: 1) not marking the # and © elimination steps any differently (indeed, the most common interpretation for the model of # elimination then becomes to switch the position of the two operators, as mentioned previously an extremely common mistake by the participants in all groups) than the other sign elimination steps; and 2) not indicating the first

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

symbol as the one that dictates the proper transformation. These sign elimination steps, and how they should be marked up, are learned by the participants as they set up hypotheses as to what the transformation should be, try them out, and are then surprised when the transformation does not work and they need to find any other hypothesis.

Representational Differences

To conclude, I would like to mention a couple of places where the representation, and the process by which productions arise from those representations, of the model does not match with that of the participants. The most egregious of these occur when spurious relations occur between symbols that make up the character string being used by the analogy mechanism. This results in overly–specific productions that participants never produce. For instance, if the input to the analogy mechanism was this:

Current example:

Ⓡ℘↔#Δ#Φ

⇓

℘↔#Δ#Φ

the production that would be created to handle the right side for Ⓡ elimination steps would be:

IF	the front operator is a Ⓡ AND the right side is of the form *op1 con1 op1 con2*	(P9)
THEN	don't change anything on RHS and pop subgoal	

That is, it would only apply when both the operators on the right side were the same. This would only be the case in a small subset of the of the problems. Furthermore, if the model did not represent the character string as having two sides, but rather as one whole set of symbols, the chances that such spurious relations occur are higher, and so more overly–specific productions are generated in that case. This is one of the reasons why the hierarchical representation was

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

chosen, in addition to support from the protocols (notice how in Appendix E's protocol that he only mentions adding one thing, but in his actions he does it to both sides). Participants were very rarely caught up by these coincidental relations. For the above problem, people would notice that nothing changed on the right side of the problem, but probably would not encode the identity of the two operators. For the model, this encoding specificity results in the creation of additional, overly specific productions that participants do not create or use. These sorts of overly specific productions are the result of the ACT–R analogy mechanism and how it considers the symbols when it creates the production. It makes the usually sensible assumption that symbols which are the same should always be the same. However, that is not always the case, and participants are much better than the model in determining when that assumption does not hold.

The second of the errors not represented in the model involves the participants considering the key arrangement of the on–screen keypad as an insight into what to add and how the symbols change. The model has no representation of this keypad, but it can provide some help in learning the task. The keypad has three columns of four keys—one column contains all the operators, another all the constants, and the last all the special keys (the double arrow, the script–p, and the delete and check keys). Participants would sometimes consider this arrangement of keys, particularly the arrangement within a column, to be the deciding factor in what to type. For example, 3 of the 24 total protocol participants at some point considered the arrangement of the constants in their column to determine what to do for © elimination (where the proper rule is to just switch the position of the two constants). Since the model had no representation of the keypad, it could not produce such a production and so could not make such an error.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Conclusion

This chapter presented an ACT–R model of the participants and their data presented in the previous two chapters. The model, using the constraint within the ACT–R architecture that all knowledge starts off declaratively and gets proceduralized via the analogy mechanism, provides a full account of the qualitative changes one sees as a person learns the task. While most of the mechanisms by which a person's actual declarative representation changes (e.g., how the character strings are parsed) were not modeled, by removing certain pieces of the model's declarative knowledge, the model can mimic unsuccessful participants.

To map this on to the main contributions of this dissertation, ACT–SF provides a model of the second contribution:

> **2) In learning the rules of a task such as Symbol Fun, learners construct internal declarative representations of the examples presented to them. These declarative representations are influenced by knowledge of the task's syntax, as well as other information particular to the task.**

The model is given a particular declarative representation of the character strings. In its current state, it has no way of changing this representation over time. When given the best possible representation (the Informed Model), it correctly and quickly learns the rules of the task. This is analogous to the participants in the Syntax(Hint) group of Experiment 1. When parts of that representation is degraded, the model makes similar mistakes as to people who have not learned the task, like the people in the Example group of Experiment 1.

Where to go from here. Even in its current state, the model makes some predictions concerning participant's behavior, and these were highlighted in the illustration of the model. At this time, only preliminary evidence has been

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

analyzed to support such predictions. When appropriate, such evidence was mentioned throughout the chapter. Better analyses of the protocol data would provide better data to test these predictions. The next two chapters discuss further empirical studies and test some of the claims inherent in the model, namely the contribution of syntax in learning Symbol Fun (Chapter 6, "Prefix Symbols") and the way people generalize the rules they are learning (Chapter 7, "General Symbols").

Outside of more in–depth analyses to better test the predictions of the current model, ACT–SF should be augmented to better predict the quantitative data. This augmentation would entail two things. First, a learning mechanism which changes the model's declarative representation of the character strings should be added so that the model could progress like a successful participant in the Examples only group of Experiment 1 (i.e., like the participant in Appendix E). Second, information should be added concerning the average participant's knowledge of algebra so that it could be used in support of learning the rules of Symbol Fun. Such information would probably take both a declarative and a procedural form, but on account of that, attempting such an addition might bring about more testable predictions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 6
Experiment 3—Prefix Symbols

The following two experiments test particular claims that follow from the ACT–R model discussed in the previous chapter. Experiment 3 further investigates the effect of providing syntactical information in addition to the examples. Experiment 4 examines more closely the way in which people generalize the rules they are learning with respect to the manner in which the model generalizes its rules.

In Experiment 1 a large effect was found between providing only examples to participants versus providing them syntactical information in addition to the same examples. More people learned the task when syntactical information was available, and they did so much more quickly (the Syntactic Knowledge Contribution). They also made somewhat fewer errors, and their pattern of errors across the different types of transformations was different (the Over Specificity Contribution). However, across both groups who successfully learned the task (the Examples and the Syntax(Hint) groups), a majority of people were reminded of algebra (the Prior Knowledge Contribution). This

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

indicates they were calling upon other knowledge with which to learn the task—not only specific algebraic knowledge (e.g., adding the same thing to both side of an equation), but probably also more general arithmetic knowledge (e.g., how equations are structured). This experiment attempts to eliminate the benefit of being able to use not only algebraic knowledge, but also this more general, equation knowledge, in order to better test the Syntactic Knowledge Contribution:

1) **In learning the rules of a task such as Symbol Fun, learners construct internal declarative representations of the examples presented to them. These declarative representations are influenced by knowledge of the task's syntax, as well as other information particular to the task (e.g., knowledge of inverse operators).**

This experiment modified the task used in Experiments 1 and 2 in order to remove the similarity between it and standard arithmetic and algebra. This task is formally equivalent to the old one, but whereas the old one used an infix notation (i.e., the relevant operator is between its two operands), this one used a prefix notation (i.e., the relevant operator is in front of its two operands). The hypothesis is that people's ability to draw upon their arithmetic knowledge would be nullified. In such a way, a better test of how the syntactical information influenced learning the new task could be assessed. The prediction of the model is that, since the two systems are formally equivalent, the learning of the two groups in this experiment should be similar to the learning of the two corresponding groups in Experiment 1.

Method

Participants. Twenty-six Carnegie Mellon University undergraduates participated in this experiment for partial course credit and pay.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 6.1
Examples used in Experiment 3

Example 1
↔®℘Φ®Δ
↔♥®℘ΦΦ♥®ΔΦ
↔℘♥®ΔΦ

Example 2
↔#♥℘Γ♥Φ
↔©#♥℘ΓΓ©♥ΦΓ
↔♥℘©♥ΦΓ
↔℘©®ΦΓ

Example 3
↔©℘®#ΓΔ
↔℘®#ΔΓ

Example 4
↔♥℘Γ♥Φ
↔®♥℘ΓΓ®♥ΦΓ
↔℘®♥ΦΓ

Example 5
↔#℘©#ΓΔ
↔℘#©ΓΔ

Example 6
↔®®℘Δ©Γ
↔♥®®℘ΔΔ♥©ΓΔ
↔®℘♥©ΓΔ
↔℘♥©ΓΔ

Example 7
↔#℘Γ©Δ
↔©#℘ΓΓ©©ΔΓ
↔℘©©ΔΓ

Example 8
↔®©℘Γ®Ω
↔♥®©℘ΓΓ♥®ΩΓ
↔©℘♥®ΩΓ
↔℘♥®ΓΩ

Materials. The task used in the this experiment is a modified version of the one used in Experiments 1 and 2. Instead of an infix notation, a prefix notation was used. The two systems are formally equivalent, and a simple transformation exists to change a character string from one version of the task into the same equation in the other version. Table 6.1 contains the eight examples available for reference to the participants (this can be compared with the examples displayed in Table 3.3 in order to gain some idea of what the syntactic difference is between the two tasks). As in the previous two experiments, the task was implemented as a Hypercard 2.2.1 stack (Apple Computer, Inc., 1994) which was run on an accelerated Apple Macintosh IIci computer connected to a two–page monitor.

Procedure. With one difference, the procedure used in this experiment was identical to Experiment 1's procedure. All participants initially saw two screens that contained some introductory comments about the experiment and instructions on the task's interface (the same two screens as used in the previous two experiments). At this point, one of the two groups received information

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

relating to the task's syntax and goal structure, as well as a hint. This information is equivalent to the information given to the people in the Syntax(Hint) group of Experiment 1, and the information is displayed in Appendix F. The only difference is that one of the items in the Goal information from Experiment 1, "If a connector appears in front of the '℘', the last step is to remove that connector from it," does not have a easy, direct correspondence in this experiment, and so was dropped. The group of people who received this additional information was in the Syntax(Hint) group, and the other group was the Examples group. Again, these two groups are analogous to the liked–name groups in Experiment 1.

At this point, a slight change was made from the procedure used in Experiment 1. The Syntax(Hint) group has received the additional information, and the Examples group has only seen the initial two introductory screens. A sheet of paper on which contained the eight example problems (Table 6.1) was given to each participant. At the top of this paper was these instructions:

> There are two basic types of symbols (I may already have told you this. If that's not the case, I call them object symbols and connector symbols). For each line below, **circle** each symbol that you think is an object symbol and **underline** each symbol you believe to be a connector symbol (every symbol does not have to have something done to it). Then draw one **vertical line** to separate each line into two parts. There's no need to spend a lot of time on these.

The participants were then expected to follow these instructions using the example problems. The people in the Syntax(Hint) group was able to refer back to the screens that contained the additional information. The purpose of this form was to ensure that the Syntax(Hint) group fully understood the parsing information, and did not simply dismiss it.

After the participants completed filling out this sheet, they continued with interacting with the computer. Both groups next went to the screen that had the eight examples, and then went to the screen on which problems were presented

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for them to solve. The computer program acted the same as the one used in the first two experiments. At any point, the participants could refer back to the example screen, and the Syntax(Hint) group could refer back to the syntax information.

Each participant was asked to solve 32 of each of the three types of problems (one–, two–, and three–step problems) for a total of 96 problems. Each participant had 2 hr with which to solve all 96 problems. There were 14 participants in the Syntax(Hint) group and 12 participants in the Examples group.

Results

Background and General Results

Table 6.2 contains summary information about the performance of participants in this experiment for easy reference. No difference is detected in the SAT scores of the participants in the two groups ($t < 1$), either when examining the groups as a whole or just the successful participants. With regards to the form that both groups filled out before presented with the screen of examples on

Table 6.2
Prefix Symbols At–a–Glance

	Syntax(Hint)	Examples Only
Self–reported math SATs	693[a]	681[a]
Reading Instructions (min)	6.21[a]	3.15[b]
Examining Examples (min)	1.65[a]	0.65[b]
Successful Participants	12 of 14[a]	1 of 12[b]
Self–reported math SATs	706	—
Example References	63.83	—
Total Time (min)	92.71	—
First Block (12 problems)	34.46	—

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the computer, both groups spent the same amount of time filling it out ($t(24) = -.751$, $p > .1$), with the Syntax(Hint) group taking 6.58 min on average, and the Examples group taking 7.47 min. Of the 14 Syntax(Hint) participants, 9 of them marked the examples exactly right. The other 5 had the objects and connectors correctly circled and underlined, but had mismarked the separating line (4 always put the line right after the ℘, and the other participant put it after all of the connector symbols). For the 12 participants in the Examples group, no clear pattern emerged. Participants did have a slight tendency to group all the Greek symbols together (either underlining or circling all of them), and then, within a single participant, have a consistent set of symbols to which they would perform the other action. There was no clear pattern for where they divided a line.

Preparation times. Not surprisingly, the two groups differed in the amount of time they spent studying the instructions ($t(24) = 2.96$, $p < .01$). The Syntax(Hint) group spent a mean of 6.21 min, and the Example group spent 3.15 min on average. The groups also differed on the amount of time initially examining the examples (after already marking them up on the form), $t(24) = 2.17$, $p < .05$, with the Syntax(Hint) group spending 1.65 min on average and the Examples group 0.65 min.

Successful and unsuccessful participants. As in the first two experiments, a distinction needs to be made between those people finishing the task and those who did not finish in the 2 hr time limit. Twelve participants completed the task in the Syntax(Hint) group and one person in the Examples group. Two people did not learn the task in the Syntax(Hint) group, and eleven people did not finish in the Examples group. Significantly fewer people ($p < .01$) finished in the Examples group. The two people in the Syntax(Hint) group who did not complete the task made it to Problem 40 in one case and Problem 38 in the other.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

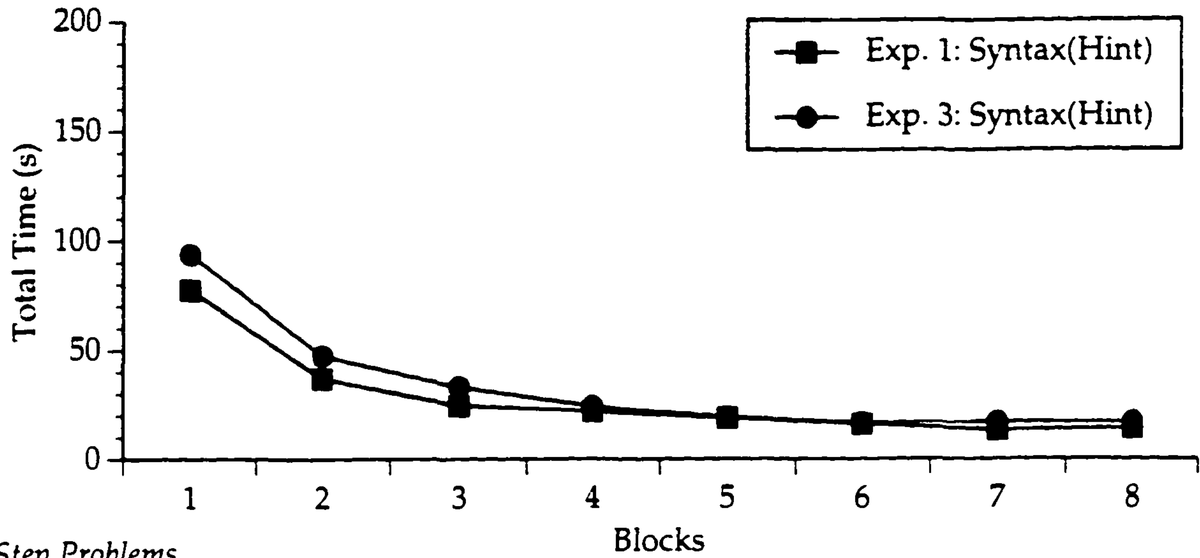
The eleven people in the Examples group who did not complete the experiment made it to problem 49.02 on average.

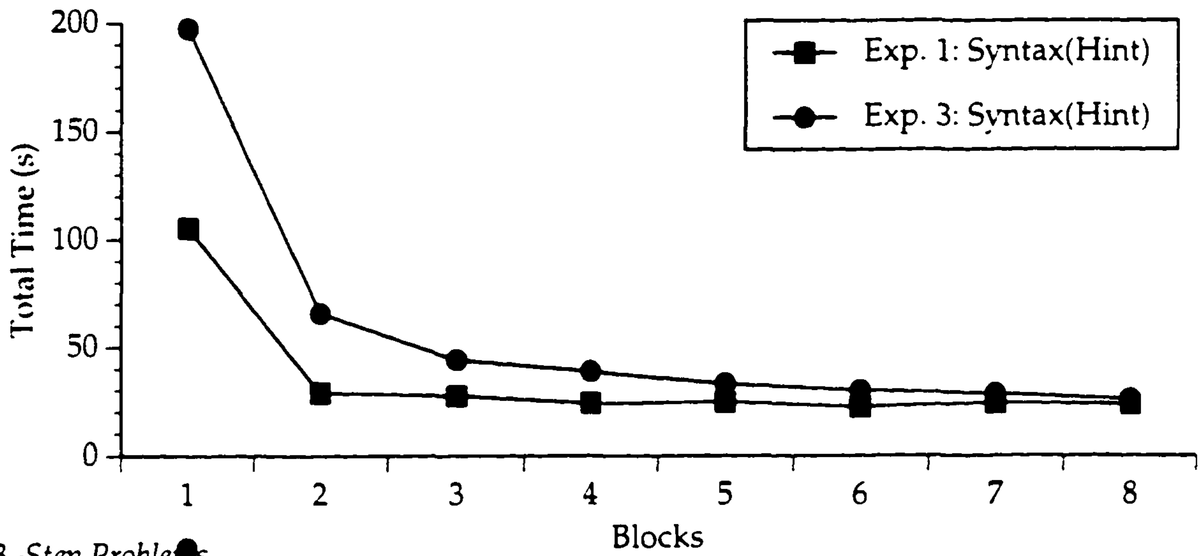
Remindings. Participants at the end of this experiment were asked if the task reminded them of anything. In the Syntax(Hint) group 6 of 14 people were reminded of algebra, and 3 of 12 people in the Examples group were. Of the two people who did not finish in the Syntax(Hint) group, one of them thought it was similar to algebra. The one person who finished in the Examples group was reminded of algebra. The most common answer to this question, across both groups, (besides "nothing") was "pattern finding." Participants were also asked at the end of the experiment if they had ever used a prefix or postfix notation for arithmetic before. None had.

Learning

Accessing information. Comparing the total number of references back to the example page by both groups and including both successful and unsuccessful participants, no difference was detected ($t(24) = 1.32$, $p > .1$), with the Syntax(Hint) group referring back to the examples screen 63.83 times on average and the Examples group a mean of 86.00 times (but remember, participants in the Examples group only made it through an average of half the problem set). However, comparing the successful participants in the Syntax(Hint) group of this experiment to the successful Syntax(Hint) participants of Experiment 1, a difference is detected ($t(22) = -3.67$, $p < .01$), where the Experiment 1 Syntax(Hint) participants referred back to the example screen an average of 23.67 times. The Syntax(Hint) participants of this experiment also referred back to the Goal information page slightly more often ($t(22) = -2.37$, $p < .05$; 0.5 times versus 1.25 times on average), but the references back to the Syntax and Hint information pages did not differ.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.





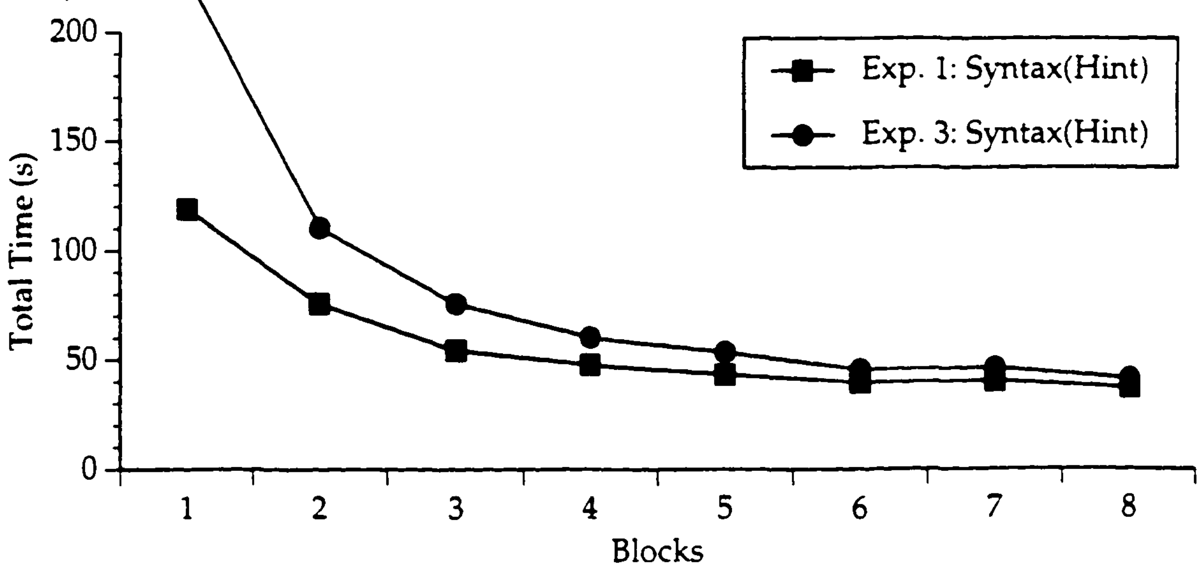


Figure 6.1: Overall time by block for each problem type (Experiment 3)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Completion time. Since most participants in the Examples group did not complete the task, it makes it difficult to compare their total time at solving the task to the total time of the Syntax(Hint) group. However, the total time of the Syntax(Hint) group can be compared to the total time of Experiment 1's Syntax(Hint) group. The Syntax(Hint) group from Experiment 1 took a mean of 64.09 min to solve all the problem and the Syntax(Hint) group from this experiment spent 92.71 min on average, a significant difference ($t(22) = -5.31$, $p < .001$). Figure 6.1 plots the performance of these two groups. The Syntax(Hint) group performed somewhat worse, in terms of time to solve the problems, to the Examples group of Experiment 1, where the successful participants of that group spent 81.54 min on average.

Errors

All together, the participants in the Examples group of this experiment made a lot of errors—a total of 1454 or a mean of 121.17 per participant. Considering the number of lines that each participant attempted to click out, about 96 on average, 1.19 errors were made per line. Essentially all of these errors were syntactic in nature, with the participants never learning any of the correct transformations. Six of the participants did apparently learn the cancellation step, and a subset of these learned some of the sign elimination transformations (e.g., leave it the same or swap the operands) but not when to correctly apply them.

The Syntax(Hint) group fared much better. Their results are displayed in Table 6.3, which can be compared with Tables 3.5 and 4.2. The Syntax(Hint) group from Experiment 1 performed much better than this Syntax(Hint) group ($t(22) = -2.72$, $p < .05$). The profile of the percentages are not different, however, from previous groups. Most errors are semantic in nature and there are very few errors on the cancellation step. Comparing the percentage error profile of this Syntax(Hint) group to the Examples and Syntax(Hint) group of Experiment 1, the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 6.3

Experiment 3 Errors Per Participant

Syntax(Hint)

	Syntax	Semantics	Total
Addition	7.50(11%)	13.75 (20%)	21.25 (30%)
Cancellation	1.67 (2%)	6.17 (9%)	7.83 (11%)
Sign Elimination	2.17 (3%)	39.00 (56%)	41.17 (59%)
Total	11.33 (16%)	58.92 (84%)	70.25

Syntax(Hint) groups are slightly more similar to one another (.96) than this Syntax(Hint) group is to the Experiment 1's Examples group (.91). Looking at the top 50% of participants in both groups (six participants in each), one sees a similar result ($t(10) = -2.57$, $p < .05$), with the Syntax(Hint) group from this experiment making a mean of 32.00 errors compared with 16.17 errors for the Experiment 1 Syntax(Hint) group.

Sign elimination errors. The participants in the Syntax(Hint) condition of this experiment made similar errors as to the participants in previous groups. That is, they swapped operands or operators, left everything the same, or inverted one of the operators when those transformations were not appropriate. Incorrect transformations that might have been peculiar to the prefix notation were not observed, at least not in significant numbers. That being the case, the expectation is that the Syntax(Hint) group of this experiment would be similar to Experiment 1's Syntax(Hint) group in terms of the sign elimination errors, and one does find this. This experiment's Syntax(Hint) group confused the ® and © elimination rules 82 times (34% of those errors), compared to 42% of Experiment 1's Syntax(Hint) group (the Examples group of Experiment 1 only made this error 14% of the time). Furthermore, this experiment's Syntax(Hint) group only made the swap operators error for # elimination 41 times (24% of # elimination

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

errors), which is more similar to the 18% of Experiment 1's Syntax(Hint) group than it is to the 41% of that experiment's Examples group.

Discussion

Although this task is formally equivalent to the task used in Experiment 1, it is more difficult for participants to learn, a result contradicted to what would be predicted by the ACT–SF model, but predicted by the Syntactic Knowledge Contribution:

1) In learning the rules of a task such as Symbol Fun, learners construct internal declarative representations of the examples presented to them. These declarative representations are influenced by knowledge of the task's syntax, as well as other information particular to the task (e.g., knowledge of inverse operators).

If given only examples to learn from, almost no one learning the prefix version learned the task, compared to almost a 50% success rate with the infix version used in Experiment 1. If those examples are augmented with syntactic and other information, people learning either version of the task eventually learn it. However, the people learning the prefix version needed more references back to the examples, took longer, and made more errors. What makes the prefix version more difficult to learn?

As mentioned in the introduction, the prefix version of the task eliminates, or at least reduces greatly, the benefit of being able to parse the character strings in a standard, arithmetic way (i.e., operators to the immediate left of their operands and an obvious divider between the left– and right–hand sides of the equation). This is part, if not most, of the knowledge contained within the syntactic information given to the participants in the Syntax(Hint) group. Instead of perhaps relying on past knowledge of how equations are structured,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

participants were forced to use the provided information to help them parse the strings, in the case of the Syntax(Hint) group, or to induce that parsing information in the case of the Examples group (and with disastrous results).

The ACT–SF Model has some of this syntactic knowledge of arithmetic not only explicit in its representation, but also implicit as well. The model, with its hierarchic organization, uses the double arrow as a divider between the character string's left and right sides. However, a flat representation could also have been used, which would correspond to having each symbol that made up the character strings contained within a separate slot in a single working memory element. (As noted in Chapter 5, such a representation was not used in order to avoid spurious relations between the symbols during the analogy process and to also allow for the creation of a more compact production system.) Once created, this basic, flat representation would have equal difficulty learning either the infix or the prefix version of the task. In its analogy process, ACT–R merely matches up the symbols on the left–hand side of the created production with the symbols on the right–hand side of the production, regardless of order. However, as seen in the data, participants have a much more difficult time matching up the symbols when they are in prefix order. Some aspect of the infix notation is easier for the participants to grasp. This differential between the infix and prefix conditions is not part of the model and is what is implicit within the ACT–SF Model as it stands. This aspect corresponds to a familiarity the participants have in dealing with equations in an infix order.

The final experiment investigates the rule generalization process (in accordance to the Over Specificity Contribution) in a more detailed manner than the previous experiments have attempted.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 7
Experiment 4—General Symbols

Experiment 4 examined more closely the rule generalization process, as mentioned in the third main contribution of the introduction:

3) Lack of adequate syntactic knowledge causes the analogy mechanism to build over-specific rules from examples.

Specifically, this experiment investigated the process by which structures in a rule are variablized and the relations that people believe hold between those structures. This is a finer level than what the generalization process has been studied at before. The ACT–R model, as described in Chapter 5, makes some plain predictions for process in this task. The analogy process in ACT–R is quite simple. If it can directly map symbols on the left–hand side of a production with symbols on the right, those symbols are linked and variablized to be the same. If multiple instances of that symbol appear, on either the left or the right, then ACT–R assumes that that must always be the case. If a symbol cannot be mapped between sides, but ACT–R has supporting information to make the mapping (e.g., knowledge of inverse operators), the link is made and that relation is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

embedded within the production. If, however, no supporting information can be found, the symbol is assumed to be a constant.

In each of the past experiments, analyses have been done that shed some light on the generalization process. The error data from the experiments provide most of the evidence. The hypothesis put forward has been that the more information initially given to people with which to learn a task, the more liberal they will be in their generalizations. For example, participants in the Syntax(Hint) conditions attempted to meld the ® and © elimination rules, like the way the ♥ and # elimination rules are similar to one another. People in the Examples group did not attempt this blending of rules. A similar phenomenon occurs within the # elimination rule. People in the Syntax(Hint) group quickly see through the misleading example that seems to indicate the proper rule is to swap the position of the two operators, and not necessarily invert them. The conservative Examples group persisted in making this error. In large part this distinction can be seen as the groups with more information being more theory–driven, since they could, see the bigger picture, whereas the groups with less information were more data–driven.

This experiment investigated such issues. By using a slightly modified version of the task used so far, one that just contained the one–step problems (those that deal with the sign–elimination steps), significantly more data was gathered to test how people generalized the rules they were learning. Furthermore, these sign elimination steps have been the most informative in the past experiments in studying this process. The task has also been modified so that people first solved simpler problems than what have been used thus far, and then in the latter part of the problem set solve the standard sign elimination problems that participants in the previous experiments have solved. These simpler problems involved only one operator/operator pair on the right–hand

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

side, as opposed to the two pairs seen in the one–step problems used in the previous experiments. This transition from simple to complex problems shed further light on how people generalize the rules they are learning. In keeping with the analyses done so far, the prediction was that the participants with less information will be most conservative in their generalizations, whereas the people with more information will be more liberal. In the model, this liberalness arises from being able to augment the learned rules with the additional declarative information, such as the inverse operators. When such additional information is not available, the rules formed must perforce be conservative and specific to only that situation.

Method

Participants. Thirty Carnegie Mellon University undergraduates participated in this experiment for partial course credit.

Materials. The task used in the this experiment was a modified version of the one used in the previous experiments. For this experiment, only one–step problems were used. Furthermore, the first part of the experiment was comprised of simpler problems, ones that had only two symbols on the right–hand side (an operator/operand pair). Table 7.1 provides examples of these simpler problems. The rules of this simpler task were largely the same as for the more complex version , except for the rule for © elimination. Since the right side only had one operator/operand pair, there were not two operands to be switched. The rule for © elimination in this simpler version was to just leave the

Table 7.1
Example Simple Problems Used in Experiment 4

Example 1	Example 2	Example 3	Example 4
® ℘↔♥Δ	♥ ℘↔®Φ	© ℘↔#Γ	# ℘↔♥Γ
℘↔♥Δ	℘↔♥Φ	℘↔#Γ	℘↔♥Γ

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 7.2

The eight simple problems participants saw

®℘↔♥Ω	♥℘↔®Δ	©℘↔®Ω	#℘↔©Φ
®℘↔©Δ	♥℘↔#Φ	©℘↔#Γ	#℘↔♥Γ

right-hand side the same (thus mirroring the ® elimination rule). The elimination rules for ♥ and # elimination were the same as in the complex version, invert related operators and leave the same any unrelated operators.

As in the previous experiments, every participant received the same problem set. The first 64 problems were all of the simpler type, and then the last 128 were all of the complex type. The first 64 were grouped into 8 sets of 8 problems. Table 7.2 contains all 8 of these problems (the operand was randomly picked). Within each set, each operator appeared as the first symbol (the symbol to be eliminated, hereafter referred to as the "elimination symbol") twice. Each elimination symbol was paired with two operators (e.g., ® was paired with ♥ and ©, whereas ♥ was paired with ® and #). When an operator was an elimination symbol, one of the operators it was paired with appeared in the right-hand side of the character string. The next time that operator appeared as the elimination symbol, its other paired operator would be on the right. The pairings were chosen such that half of the symbols that appeared when ♥ and # was the elimination symbol would be related, and thus need to be inverted. However, for both of those two symbols, participants would only see half of the possible inversions (i.e., they would see © paired with # as the elimination symbol, but not # when # was the elimination symbol).

A similar pattern was used for the last 128 problems, which were grouped into 4 sets of 32 problems. For these problems, two operators appear in the right-hand side. For each elimination symbol, the first operator on the right was chosen from one of the two operators that it did not appear with during the first

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

64 problems (e.g., when ® was the elimination symbol, the first operator would either be a ® or a #, whereas if ♥ was the elimination symbol, the first operator would be either a ♥ or a ©). The second operator could be any of the four. This results in eight combinations for each of the four elimination symbols, or 32 total different problems. This mildly complicated scheme of generating problems was used in order to test how participants would generalize to seeing other symbols in the same position, as well as to the second operator position at the start of the complex problems.

Procedure. Outside of the different problem set, the procedure for this experiment was similar as to the previous ones. Like Experiment 3, this experiment was comprised of two groups, an Examples group and a Syntax(Hint) group. Both groups initially went through two screens of introductory material (the same as all previous groups saw). The Syntax(Hint) group next received the syntax, goal, and hint information that Experiment 1's Syntax(Hint) group received (see Appendix A). Both groups next went through a screen of examples (the four examples displayed in Table 7.1), and then started solving the 192 that made up the problem set. They were told that at some point the problems would get more complicated, but not exactly when. The participants interacted with the program the same way as participants in the previous experiment—clicking out their solutions, having the computer check their line, and then receiving feedback. Participants had two chances per problem to enter the right character string. If both guesses were incorrect, the computer would display the right answer before giving them the next problem. As before, the participants were able to refer back to the examples screen at any time. When they made it to the complex problems, the examples did not change.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Each participant had 1 hr with which to solve all 192 problems. There were 16 participants in the Syntax(Hint) group and 14 participants in the Examples group.

Results

Background and General Results

Table 7.3 contains summary information about the performance of participants in this experment for easy reference. No difference is detected in the SAT scores of the participants in the two groups ($t < 1$), either when examining the groups as a whole or just the successful participants. The two groups differed in the amount of time they spent studying the instructions ($t(28) = -4.23$, $p < .001$). The Syntax(Hint) group spent a mean of 4.43 min, and the Example group spent 2.99 min on average. The groups, however, did not differ on the amount of time initially examining the examples, $t(28) = -1.27$, $p > .1$, with the Syntax(Hint) group spending 0.83 min on average and the Examples group 0.68 min.

Successful and unsuccessful participants. As in the prior experiments, a distinction can be made between those people finishing the task and those who did not finish in the 1 hr time limit. Twelve participants completed the task in the

Table 7.3
General Symbols At–a–Glance

	Syntax(Hint)	Examples Only
Self–reported math SATs	695^a	674^a
Reading Instructions (min)	4.43^a	2.99^b
Examining Examples (min)	0.83^a	0.68^a
Successful Participants	12 of 16^a	12 of 14^a
Self–reported math SATs	690^a	672^a
Example References	3.92^a	9.75^a
Total Time (min)	48.01^a	49.15^a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

both the Syntax(Hint) and Examples groups. This means that four people did not learn the task in the Syntax(Hint) group, and two people did not finish in the Examples group. This difference in proportions is not statistically significant. The four people in the Syntax(Hint) group who did not complete the task completed an average of 101.75 problems. One person in the Examples group who did not finish made it to Problem 100, and the other person actually did complete all the problems. However, this participant made 156 errors over the course of the 128 complex problems.

Remindings. After the experiment, the participants were asked what the task reminded them of (as in Experiments 1 and 3). In the Examples group, 3 of 14 people answered algebra. In the Syntax(Hint) group, 5 of 16 people replied algebra (this is not a significant difference, $p > .1$). Almost everyone else was not reminded of anything. None of the people who did not complete the task were reminded of algebra.

Rule learning. Another question asked of the participants after the experiment was for them to relate the rules of the task. Of the people who successfully completed the task, only 6 participants in each group could successfully enunciate the rules. Success was indicated by knowing the two pairs of inverse operators and when they were needed (for # and ♥ elimination, and by knowing the rules for ® and © elimination). Four people in the Examples group and 5 people in the Syntax(Hint) group had a "fractured" set of rules (either incomplete or they went mostly by specific instances). The remaining three participants could not formulate an answer to the question.

Learning

Accessing information. The examples available for the participant's reference in the experiment were not as useful as the examples available in the prior experiments. There were only four examples, and they were all within the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

simple version of the task. No difference was detected between the number of times the Syntax(Hint) group referred back to the examples versus the Examples group's references ($t(22) = -1.29$, $p > .1$). The Syntax(Hint) group referred back to the examples screen a mean of 3.92 times, and the Examples group referred back to that screen 9.75 times on average. No one in the Syntax(Hint) group referred back to the hint screen, but they did refer back to the syntax screen 2.33 times on average and to the goal screen 1.33 times.

Completion time. In terms of total time to complete the problem set, the two groups did not differ ($t < 1$). Figure 7.1 plots the performance of the Examples and Syntax(Hint) group across the problem set, with the data blocked into groups of 16 problems. The performance of the Syntax(Hint) group from Experiment 1 is plotted for comparison purposes (this group only received a total of 32 one–step problems). Even when the complex problems are compared separately, no difference exists ($t < 1$). The Syntax(Hint) group took a mean of 48.01 min to solve all the problems, and the Examples group took an average of 49.15 min.

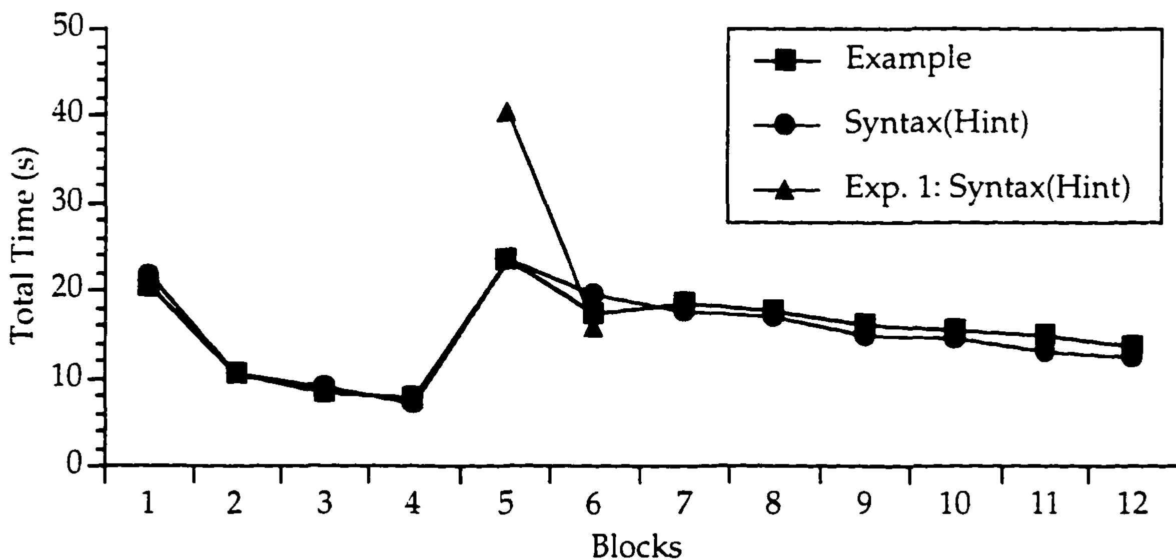


Figure 7.1: Average time spent per problem

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 7.4
Experiment 4 Errors Per Participant

Examples

	Syntax	Semantics	Total
Simple	3.92 (23%)	13.33 (77%)	17.25
Complex	5.50 (6%)	81.92 (94%)	87.42

Syntax(Hint)

	Syntax	Semantics	Total
Simple	1.25 (8%)	14.33 (92%)	15.58
Complex	4.67 (5%)	82.00 (95%)	86.67

Errors

Table 7.4 displays the mean number of errors per participant, split into the two different groups. The numbers for the earlier, simple problems are listed separately from the later, complex problems. Since there was only one–step problems, there were no addition or cancellation steps—every line was a sign elimination step. In terms of total number of errors, there is no difference between the two groups ($t(22) = -1.57$, $p > .1$). Looking at the various subgroupings (e.g., syntax errors on simple problems), no significant differences were found.

Sign elimination errors. The lack of difference in the total number of errors was surprising, but a difference in the type of errors could still exist. Table 7.5 separates the errors made on the complex problems by the four operators that could appear as the elimination symbol. For each elimination, the mean number of errors made for each error type is listed, along with the percent of errors for that elimination symbol's total errors. Leave Same errors occurred when participants did not do anything to the right–hand side of the character when eliminating the elimination symbol (that is the proper thing to do for eliminating a ®). A Switch Operators (or Switch Operands) error was when the participant

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 7.5

Experiment 4 Errors in Complex Problems by Sign Elimination Type

Examples

	Leave Same	Switch Operators	Switch Operands	Invert 1st Operator	Invert 2nd Operator	Invert Both	Other	Total
♥	5.17 (24%)	1.42 (7%)	6.92 (32%)	0.58 (3%)	2.50 (12%)	0.08 (1%)	5.00 (23%)	21.67 (25%)
#	6.25 (27%)	2.42 (10%)	6.08 (26%)	1.00 (4%)	2.92 (12%)	0.08 (1%)	4.83 (20%)	23.58 (27%)
©	11.67 (45%)	4.08 (16%)		2.67 (10%)	1.92 (7%)	1.42 (5%)	4.17 (16%)	25.92 (30%)
®		3.08 (19%)	6.17 (38%)	1.67 (10%)	0.83 (5%)	1.00 (6%)	3.50 (22%)	16.25 (19%)

Syntax(Hint)

	Leave Same	Switch Operators	Switch Operands	Invert 1st Operator	Invert 2nd Operator	Invert Both	Other	Total
♥	6.50 (30%)	2.33 (11%)	6.83 (32%)	0.83 (4%)	2.08 (10%)	0.17 (1%)	2.58 (12%)	21.33 (25%)
#	6.33 (30%)	0.83 (4%)	5.25 (25%)	1.08 (5%)	3.08 (14%)	0.00 (0%)	4.75 (22%)	21.33 (25%)
©	14.42 (50%)	2.17 (7%)		3.42 (12%)	3.17 (11%)	1.17 (4%)	4.67 (16%)	29.00 (34%)
®		1.33 (9%)	5.75 (38%)	2.17 (14%)	1.58 (11%)	0.58 (4%)	3.58 (24%)	15.00 (17%)

switched the operators (or Operands) when eliminating the leading operator. Switching operators was a common mistake in the last experiments, because of one of a misleading example (Example 5). Switching operands is the right transformation for © elimination. The three inversion errors (Invert 1st, Invert 2nd, and Invert Both Operators) refer to when a participant inverted an operator (either the first, the second, or perhaps both) incorrectly. Depending on the operators on the right-hand side, inverting is sometimes the right thing to do for ♥ and # elimination. Finally, there is an Other category for errors that did not fall into one of the other six. These included the syntax errors and also errors in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

clicking (e.g., clicking delta instead of omega). The percentage under the columns labeled Total are the number of errors for that particular elimination symbol over the total number of errors.

Overall, the correlation between the percentages of the Examples group with those of the Syntax(Hint) group is 0.87, indicating that there are more similarities between the two groups than differences. There are main effects of elimination symbol ($F(3,66) = 14.78$, $MSE = 6.08$, $p < .001$) and error type ($F(6,132) = 23,12$, $MSE = 12.46$, $p < .001$). The interaction of group by elimination symbol is not significant ($F(3,66) = 1.48$, $MSE = 6.08$, $p > .1$), indicating that within the two groups, the participants made a similar pattern of errors across the four elimination symbols. The interaction of elimination symbol by error type and the three–way interaction of operator by error type by group are significant ($F(18,396) = 30.31$, $MSE = 4.32$, $p < .001$ for the two–way, and $F(18,396) = 1.84$, $MSE = 4.32$, $p < .05$ for the three–way), meaning that the different elimination symbols elicited different types of errors, and that those errors differed at least slightly between the Syntax(Hint) and Examples participants. However, the interaction of group by error type is not significant ($F(6,132) = 1.64$, $MSE = 12.46$, $p > .1$), indicating that the two groups, on the whole, made similar error patterns overall.

Variablization. One of the main interests in this experiment was to see how people variablized the rules they are learning and how they generalized symbol position and type. The best measurement of this is to look at transfer from the simple to the complex problems. Table 7.6 displays percentages relating to the first time participants had the opportunity to transfer knowledge to the complex problems. It displays data collapsed across ♥ and # sign elimination problems, which both involve inverting operators. The first column, Same Operator in 2nd Position, refers to when participants correctly inverted the same operator they

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 7.6

Transfer from simple to complex problems

	Same Operator in 2nd Position	Related Operator in 1st Position	Related Operator in 2ndPosition
Examples	71%	25%	17%
Syntax(Hint)	88%	13%	13%

had seen inverted in the simple problems, but in the second position, not the first (e.g., a ® in second position when it was ♥ elimination). The other two columns refer to correctly inverting the related operator when it appeared either in first or second position (e.g., a ♥ in first or second position when it was ♥ elimination). Only the Syntax(Hint) group knew that these two pairs of operators were related. A test of the proportions show that the two groups are not significantly different from one another, but both groups were much better at generalizing the same operator than the related operator ($p < .01$).

Discussion

The main manipulation of this experiment, between the Examples group and the Syntax(Hint) group, did not appear to make a difference. Only slight differences existed in the error data, and people in the Syntax(Hint) group were no different at transferring, in either position, to the related operator when it needed to be inverted. Based on participants' answers to what they thought the rules of the task were, the two groups were surprisingly equal. Several participants in the Syntax(Hint) group could not articulate why the hint of the inverse operators was important to the task. Due to this lack of difference between the groups, only weak evidence was found for the Over Specificity Contribution in this experiment:

3) Lack of adequate syntactic knowledge causes the analogy mechanism to build over-specific rules from examples.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The manipulations used in this experiment, transferring from the simple to the complex problems and having just the sign elimination steps, might not have been sufficient to elicit the effects seen in the previous experiments. An informal examination of three protocol participants in the Syntax(Hint) condition reveals that when transitioning from the simple to the complex problems, all of participants felt that the two types of problems were disjoint, and one even felt that the rules had radically changed. When solving the complex problems, two of them did not fully reflect on how the hint might be able to help. All three of the participants, and this was true of many of the other participants as expressed in e exit interview, felt that when they were first trying to solve the complex problems, that many rules existed.

The sign elimination steps by themselves might not be enough to engage many participants in the right mindset to correctly learn the task. These steps may be far enough removed from algebra that participants do not see it as such, and so do not make use of that knowledge. Furthermore, the transformations appear strange enough that even up–front knowledge of the inverse operators helps. Perhaps it is only in combination with the addition and cancellation steps that the differences in sign elimination between the Examples groups and the Syntax groups seen in previous experiments emerge. The addition and cancellation steps depend heavily upon the knowledge of inverse operators. The sign elimination steps, while the most succinct set of rules use inverse operators, can be adequately learned either by remembering a set of specific incidences or by learning what many participants referred to as "heuristics" (e.g., "if a ♥ appeared out front, and a ♥ appeared later, it tended to change to a ®").

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 8
Conclusions

I began this dissertation by asking three questions: 1) How do people learn a new task, given the instructions and information available to them? 2) How do they bring their existing knowledge, when appropriate, to bear on learning the new task? and 3) Is there is a simple, underlying mechanism which can account for this learning? The preceding chapters have provided four experiments and an ACT–R model which attempted to shed light on these questions. In this chapter I will summarize and discuss the results. In the first chapter I presented three main points I wanted to make in this dissertation. In service to answering the three questions mentioned above. I will summarize the results of this dissertation in the context of these three points, as well as how the model bears on these issues.

1) In learning the rules of a task such as Symbol Fun, learners construct internal declarative representations of the examples presented to them. These declarative representations are influenced by knowledge of the task's syntax, as well as other

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

information particular to the task (e.g., knowledge of inverse operators).

Experiments 1 and 3 clearly demonstrated this point. Both of these experiments had groups that were only given examples and groups that were given syntactical information with the examples. The groups given the additional information performed better across most measures, even though in most cases they only referred to the additional information once, at the time of initial instruction. People would only refer back to the examples screen while actually learning the task, but these examples are being interpreted through the additional declarative information that the problem solver has. This declarative knowledge could either be given to them, in the case of the syntax groups, or it could be induced, in the case of the examples only groups. This interpretive process results in a rich elaboration of the examples by which the rules of the task can be more easily and accurately learned by the problem solver.

The full ACT–SF model presented in Chapter 5 has the best representation possible with which to learn the task. That is, the elaborations it has of the examples enables it to learn the correct rules of the task with little difficulty. It represents each character string as having a left– and right–hand side and that each symbol within the character is separate from the others. It knows about the inverse operators, and the examples are marked to allow the most efficient learning of the sign elimination steps. This roughly corresponds to the elaborate, declarative information that the Syntax(Hint) group had at the beginning of the task, or the representation that successful Examples group participants eventually build. The model takes into account the additional information it has when it forms the rules, and the learning is better when such information is available. If that information is taken out of the model (e.g., the knowledge of inverse operators, or the underlying equation representation that the model uses

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is simplified), the model mimics performance of unsuccessful participants, or participants who are just beginning to learn.

2) One of the strongest predictors of success for learning Symbol Fun was if the learner was able to access and use their knowledge of algebra.

In Experiment 1, the people in the Examples group had a significantly better chance of learning the task if, while in the process of learning the task, they were reminded of algebra. Of the 23 people in that condition, 12 learned the task. Of those 12, 11 were reminded of algebra. Of the 11 people who did not learn the task, only 1 person was reminded of algebra. In both the syntax groups, 9 of the 12 people who finished were reminded of algebra. People's knowledge of algebra was affecting how (and if) they learned this task, and Experiment 2 manipulated people's awareness as to how the task was related to algebra.

Experiment 2 directly tested this claim. Three levels of hint were provided, each level subsuming the one below it. Twelve of 19 participants successfully completed the task in the Algebra(Low) group (the group with the least verbose hint), and 12 of 12 participants in the Algebra(Intermediate) and 12 of 13 in the Algebra(High) groups did likewise. The latter two proportions are significantly different from the Examples group ($p < .05$). All three of the algebra hint groups completed the task in significantly shorter time ($p < .05$) than the Examples group of Experiment 1. The algebra hint helped the participants considerably, with the suggestion that the more explicit the hint, the better the learning.

The model does not explicitly represent people's knowledge of algebra. A safe assumption would be that all participants in these experiments had the knowledge and representations of basic algebra that Symbol Fun utilizes, but some participants may have been more practiced with it than others. Inasmuch as Symbol Fun makes use of the same (or, at least, similar) underlying

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

representations, providing people the information that the task is based on algebra upfront should increase the levels of activation of those structures and make them primed to be used. The Algebra(Intermediate) and Algebra(High) groups were more successful than the Algebra(Low) group because their hint specified better which parts of their algebraic knowledge would be needed.

3) Lack of adequate syntactic knowledge causes the analogy mechanism to build over–specific rules from examples.

Experiment 4 was designed to directly test this claim, but the first three experiments each provided some additional evidence. In these three experiments, the more information people were given, the more liberal their generalizations. The syntax groups were more likely to attempt to meld the ® and © sign elimination rules together, and they did not perseverate in making the error of switching the operators around for # elimination. These participants appeared to be more theory–driven, whereas the participants in the Examples groups were more data–driven. That is, since the participants in the syntax groups had more declarative information with which to elaborate their rule formation, they did so. The Examples groups were more conservative.

Unfortunately, this particular finding did not appear in Experiment 4. The Syntax(Hint) group made similar errors as the Examples group. The main reason for this lack of effect was that participants perceived the scaled–down version of the task used in this experiment (which only used one–step problems) as less algebra–like than the full version of the task used in the previous experiments. This resulted in a number of participants not fully learning the rules of the task, and instead either relying on specific instances or partial rules to do the task.

However, one can still use the results of this experiment to examine how people variablize the rules of a task they are learning. Participants are extremely likely to transfer to different positions. That is, for this task they would the same

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

thing to the same operator when it appeared in a different position. They would not, though, transfer to related operators. The participants in the Syntax(Hint) group, who knew about the related operators, were no more likely than the Examples group participants to invert the related operator, either when it appeared in the same or a different location.

The model can account for these effects. It has supporting declarative information, such as the syntax and hint information, which the examples are filtered through. The model will use these marked–up examples in forming the rules it is learning. These embellished rules can be more general in their application, since they can take into account that a symbol is being inverted, and that is why that change occurs. In the case of mis–marked–up examples, misgeneralizations occur. In Experiment 4, participants were not using, in the case of the Syntax(Hint) group, the information provided to them to the best advantage. Both the Examples and Syntax(Hint) groups had a sparse, non–algebraic representation of the task, and so neither group transferred to the related operator quickly. The model accounts for this by not using its knowledge of inverse operators when given those kinds of transfer problems.

Implications

Psychological. Perhaps the main feature of this dissertation is in bringing together several threads of past psychological research—learning from examples, transfer of cognitive skill, and forming generalizations—and providing a model of those processes within an existing unified theory of cognition, Anderson's ACT–R theory. As discussed in Chapter 2, few models of learning have attempted to model the acquisition of a large part of a domain. Those that have, Alex and ZBIE for example, have largely been separate models of learning, not tied to any existing theory. Inasmuch as that indicates the generality of the approach, that is good. However, humans have a specific implementation of such

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

learning mechanisms, and ACT-R has been used to successfully model humans in many other domains. Furthermore, both Alex and ZBIE, as well as many of the other models discussed in Chapter 2 were not compared to empirical results obtained from humans..

The model developed in Chapter 5 was created on the basis of the empirical results of the first two experiments (Chapters 3 and 4), and had testable predictions (Chapters 6 and 7). It can therefore stand as a strong test of Anderson's claim that all knowledge begins in a declarative form, and that procedures arise out of that declarative knowledge. The model captures the important aspects of people learning the task in all the conditions, and contains explanations for why people in certain conditions are facilitated in their learning. The only notable exception is the complete failure to learn the task in the Examples group in the prefix version of the task (Chapter 6). Specifically, the model, and the ACT-R analogy mechanism in general, is very good at matching symbols between lines of a problem's solution. Given the formal equivalence of the prefix and infix version of the task, the model would predict the Examples groups in both versions to perform the same. One could provide an explanation within ACT-R, that the declarative representations that underlie infix notation are stronger than those for a prefix notation (due to more previous exposure to infix notation), and so the learning, and also the probability of being reminded of algebra, is increased. This fact is not captured by the current model.

Pedagogical. I would like to conclude with a short discussion of the implications of this research on educational issues. This dissertation lends itself to such a discussion, even though it focused on modeling the initial learning of a task, and not necessarily on retention of that knowledge. A future study would bring back participants six months or a year later and measure how well they

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

remembered the task. However, due to the moderately simplistic nature of it, this task may not be the best one to use.[1]

The empirical results, and the model which was based on them, argue that the best learning occurs when what is created within the student's mind is an appropriate representation of the examples used to illustrate the domain. Or put another way, students can learn by example, but to be most effective, these examples need to be embellished with additional declarative knowledge. For this domain, this additional declarative information could be either telling the student that the task is based on algebra (and how it is related), or by telling the task's syntax, including the fact that two pairs of symbols are related to one another. This points to the importance of doing a careful task analysis of the domain to be taught, and to use that task analysis in designing instructional material. This has been argued before by other researchers (e.g., Resnick, 1973). However, in the case of this dissertation, the model provides an explanation of the importance of each piece of additional declarative information, and can provide clues in diagnosing a student's deficiency in learning the task.

[1]Anecdotally, once learned, people remember this task. Out of the many Carnegie Mellon University Subject Pool participants who have learned this task, three have mistakenly signed up for different versions of this task conducted across different semesters. All three remembered the task sufficiently well as soon as they started that they were able to perform the task with few errors (though unmeasured).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

References

Anderson, J. R. (1983). *The architecture of cognition*. Cambridge, MA: Harvard University Press.

Anderson, J. R. (1991). The adaptive nature of human categorization. *Psychological Review*, 98, 409–429.

Anderson, J. R. (1993). *Rules of the mind*. Hillsdale, NJ: Erlbaum.

Bartlett, F. C. (1932). *Remembering: A study in experimental and social psychology*. Cambridge: Cambridge University Press

Bassok, M. (1990). Transfer of domain-specific problem-solving procedures. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *16*(3), 522 - 533.

Bassok, M., & Holyoak, K. J. (1989). Interdomain transfer between isomorphic topics in algebra and physics. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *15*(1), 153–166.

Bassok, M., Wu, L., & Olseth, K. L. (1995). Judging a book by its cover: Interpretative effects of content on problem–solving transfer. *Memory & Cognition*, *23*(3), 354–367.

Bernardo, A. B. I. (1994). Problem–specific information and the development of problem–type schemata. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *20*, 379–395.

Blessing, S. B. & Anderson, J. R. (1996). How people learn to skip steps. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *22*, 792–810.

Blessing, S. B. & Ross, B. H. (1996). Content effects in problem categorization and problem solving. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *22*, 792–810.

Bransford, J. D. & Johnson, M. K. (1972). Contextual prerequisites for understanding: Some investigations of comprehension and recall. *Journal of Verbal Learning and Verbal Behaviour*, *11*, 717–726.

Cheng, P. W., Holyoak, K. J., Nisbett, R. E., & Oliver, L. M. (1986). Pragmatic versus syntactic approaches to training deductive reasoning. *Cognitive Psychology*, *18*, 293–328.

Chi, M. T. H., Feltovich, P. J., & Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. *Cognitive Science*, *5*, 121–152.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Cummins, D. D. (1992). The role of analogical reasoning in the induction of problem categories. *Journal of Experimental Psychology: Learning, Memory, & Cognition, 18,* 1103–1124.

Fong, G.T., Krantz, D. H., & Nisbett, R.E. (1986). The effects of statistical training on thinking about everyday problems. *Cognitive Psychology, 18,* 253–292.

Hardiman, P. T., Dufresne, R., & Mestre, J. P. (1989). The relation between problem categorization and problem solving among experts and novices. *Memory & Cognition, 17,* 627–638.

Hayes, J. R., & Simon, H. A. (1974). Understanding written problem instructions. In Lee W. Gregg (Ed.), *Knowledge and Cognition*, Hillsdale, NJ: Erlbaum.

Hinsley, D. A., Hayes, J.R., & Simon, H.A. (1977). From words to equations: Meaning and representation in algebra word problems. In M. A. Just & P. A. Carpenter (Eds.), *Cognitive Processes in Comprehension*, Hillsdale, NJ: Erlbaum.

Hofstadter, D. R., Mitchell, M. & French, R. M. (1987). *Fluid concepts and creative analogies: A theory and its computer implementation* (Tech. Rep. No. 10). Ann Arbor, MI: University of Michigan, Cognitive Science and Machine Intelligence Laboratory.

Holyoak, K. J., & Koh, K. (1987). Surface and structural similarity in analogical transfer. *Memory & Cognition, 15,* 332–340.

Hypercard 2.1.1 [Computer software]. (1994). Cupertino, CA: Apple Computer.

Kieras, D. E. & Bovair, S. (1984). The role of a mental model in learning to operate a device. *Cognitive Science, 8,* 255–273.

Lewis, C. (1988). Why and how to learn why: Analysis–based generalization of procedures. *Cognitive Science,* 12, 211–256.

Neves, D. M. (1978). A computer program that learns algebraic procedures by examining examples and by working test problems in a textbook. *Proceedings of the Second National Conference of the Canadian Society for Computational Studies of Intelligence.*

Neves, D. M. (1981). Learning procedures from examples. Unpublished doctoral dissertation, Carnegie Mellon University, Pittsburgh, PA.

Newell, A., & Simon, H. A. (1972). *Human problem solving.* Englewood Cliffs, NJ: Prentice–Hall.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Novick, L. R. (1988). Analogical transfer, problem similarity, and expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 14*(3), 510–520.

Reed, S. K., & Bolstad, C. A. (1991). Use of examples and procedures in problem solving. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 17*(4), 753–766.

Resnick, L. B., Wang, M. C., & Kaplan, J. (1973). Task analysis in curriculum design: A hierarchically sequenced introductory mathematics curriculum. *Journal of Applied Behavior Analysis, 6*, 679–710.

Ross, B. H. (1984). Remindings and their effects in learning a cognitive skill. *Cognitive Psychology, 16*, 371–416.

Ross, B. H. (1989). Distinguishing types of superficial similarities: Different effects on the access and use of earlier problems. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 15*, 456–468.

Ross, B. H., & Kennedy, P. T. (1991). Generalizing from the use of earlier examples in problem solving. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 16*(1), 42–55.

Rumelhart, D. E., & McClelland, J. L. (1986). On learning the past tenses of English verbs. In J. L. McClelland & D. E. Rumelhart (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* (Vol. 2, pp. 216–271). Cambridge, MA: MIT Press.

Rumelhart, D. E., & Ortony, A. (1977). The representation of knowledge in memory. In R. C. Anderson, R. J. Spiro, & W. E. Montague (Eds.), *Schooling and the acquisition of knowledge*. Hillsdale, NJ: Erlbaum.

Schank, R. C., & Abelson, R. (1977). *Scripts, plans, goals, and understanding*. Hillsdale, NJ: Erlbaum.

Shrager, J., & Klahr, D. (1986). Instructionless learning about a complex device: the paradigm and observations. *International Journal of Man–Machine Studies, 25*, 153–189.

Siklóssy, L. (1972). Natural language learning by computer. In H. A. Simon and L. Siklóssy (Eds.), *Representation and Meaning*. Englewood Cliffs, NJ: Prentice–Hall.

Simon, H. A. (1972). The Heuristic Compiler. In H. A. Simon & L. Siklóssy (Eds.), *Representation and Meaning*. Englewood Cliffs, NJ: Prentice–Hall.

Singley, M. K., & Anderson, J. R. (1989). *The transfer of cognitive skill*. Cambridge, MA: Harvard University Press.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Sweller, J., & Cooper, G. A. (1985). The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction, 2*(1), 59–89.

Thorndike, E. L. (1906). *Principles of teaching*. New York, NY: A. G. Seiler.

Thorndike, E. L., & Woodworth, R. S. (1901). The influence of improvement in one mental function upon the efficiency of other functions. *Psychological Review*, 8, 247–261.

Williams, D. S. (1972). Computer program organization induced from problem examples. In H. A. Simon and L. Siklóssy (Eds.), *Representation and Meaning*. Englewood Cliffs, NJ: Prentice–Hall.

Zhu, X. & Simon, H. A. (1987). Learning mathematics from examples and by doing. *Cognition and Instruction, 4*, 137–166.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix A: Additional Information

Information available to both the Syntax(Hint) and Syntax(No Hint) groups:

Syntax

The problem takes the form of a string of characters. The characters are selected from the following:

©, ®, #, ♥	Are the connector symbols
Δ, Γ, Ω, Φ	Are the object symbols
↔, ℘	Are special symbols

The ↔ character serves to divide the character string into a left–hand side and a right–hand side.

Object symbols always have a connector to their left, and may appear on either the left or right side of the character string.

The '℘', which only appears on the left side, may or may not have a connector to its left.

Goal

Your goal is to isolate the '℘' character on the left–hand side.

A set of rules exist that dictates how you can change the current character string into a new character string.

Only one rule is applicable for any particular character string.

If a connector appears in front of the '℘', the last step is to remove that connector from it.

Information available only to the Syntax(Hint) group:

Hint

The ® and the ♥ symbols, as well as the © and the # symbols, are associated with one another.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix B: The Annotated Examples

Example 1

℘®Φ↔®Δ	X+A=+C
℘®Φ♥Φ↔®Δ♥Φ	X+A–A=+C–A
℘↔®Δ♥Φ	X=+C–A

Example 2

♥℘#Γ↔♥Φ	–X*B=–A
♥℘#Γ©Γ↔♥Φ©Γ	–X*B÷B=–A÷B
♥℘↔♥Φ©Γ	–X=–A÷B
℘↔®Φ©Γ	X=+A÷B

Example 3

©℘↔#Γ®Δ	÷X=*B+C
℘↔#Δ®Γ	X=*C+B

Example 4

℘♥Γ↔♥Φ	X–B=–A
℘♥Γ®Γ↔♥Φ®Γ	X–B+B=–A+B
℘↔♥Φ®Γ	X=–A+B

Example 5

#℘↔#Γ©Δ	*X=*B÷C
℘↔©Γ#Δ	X=÷B*C

Example 6

®℘®Δ↔©Γ	+X+C=÷B
®℘®Δ♥Δ↔©Γ♥Δ	+X+C–C=÷B–C
®℘↔©Γ♥Δ	+X=÷B–C
℘↔©Γ♥Δ	X=÷B–C

Example 7

℘#Γ↔©Δ	X*B=÷C
℘#Γ©Γ↔©Δ©Γ	X*B÷B=÷C÷B
℘↔©Δ©Γ	X=÷C÷B

Example 8

©℘®Γ↔®Ω	÷X+B=+D
©℘®Γ♥Γ↔®Ω♥Γ	÷X+B–B=+D –B
©℘↔®Ω♥Γ	÷X=+D –B
℘↔®Γ♥Ω	X=+B –D

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix C: The ACT-SF Model

```
(clearall)

(sgp :ea 'restricted :at nil)

(wmetype transform-string left
         right)
(wmetype expression specop specarg
         op1 arg1 op2 arg2)
(wmetype change operator argument
         string result)
(wmetype invert operator op1 arg1)
(wmetype setup operator argument
         string result left right)
(wmetype operator inverse type)

(addwm
; Problem 1
;    X - A = * C
 (Problem1
  isa transform-string
  left Problem1Left
  right Problem1Right)
 (Problem1Left
  isa expression
  specop blank1
  specarg X
  op1 -
  arg1 A)
 (Problem1Right
  isa expression
  specop blank1
  specarg blank2
  op1 *
  arg1 C)

; Problem 2
;    + X = - B / D
 (Problem2
  isa transform-string
  left Problem2Left
  right Problem2Right)
 (Problem2Left
  isa expression
  specop plus
  specarg X)
 (Problem2Right
  isa expression
  specop blank1
  specarg blank2
  op1 -
  arg1 B
  op2 /
  arg2 D)

; Problem 3
;    / X * D = + B
 (Problem3
  isa transform-string
  left Problem3Left
  right Problem3Right)
 (Problem3Left
  isa expression
  specop divide
  specarg X
  op1 *
  arg1 D)
 (Problem3Right
  isa expression
  specop blank1
  specarg blank2
  op1 +
  arg1 B)

; Problem 4
;    - X = - D * A
 (Problem4
  isa transform-string
  left Problem4Left
  right Problem4Right)
 (Problem4Left
  isa expression
  specop minus
  specarg X)
 (Problem4Right
  isa expression
  specop blank1
  specarg blank2
  op1 -
  arg1 D
  op2 *
  arg2 A)

; Problem 5
;    * X = * A + C
 (Problem5
  isa transform-string
  left Problem5Left
  right Problem5Right)
 (Problem5Left
  isa expression
  specop multiply
  specarg X
  op1 +
  arg1 C)
 (Problem5Right
  isa expression
  specop blank1
  specarg blank2
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  op1 *
  arg1 A)

 (NewLeft
  isa expression)
 (NewRight
  isa expression)
 (*
  isa operator
  inverse /
  type multiplication)
 (/
  isa operator
  inverse *
  type multiplication)
 (-
  isa operator
  inverse +
  type addition)
 (+
  isa operator
  inverse -
  type addition))

;;
;;  Example 1
;;  X + A = * B
;;
(addwm
;Used by analogy mechanism to set
;the initial 2 subgoals of adding
;to both sides
 (Example1-Dependency
  isa dependency
  goal Example1Line1
  subgoals (Subgoal1L1 Subgoal1R1)
  modified (Newgoal1-1)
  constraints (Left1-1)
  dont-cares (blank1 blank2 X))
 (Example1Line1
  isa transform-string
  left Left1-1
  right Right1-1)
 (Left1-1
  isa expression
  specop blank1
  specarg X
  op1 +
  arg1 A)
 (Right1-1
  isa expression
  specop blank1
  specarg blank2
  op1 *
  arg1 B)
 (Subgoal1L1
  isa change
  operator +
  argument A
  string Left1-1
  result NewLeft)
 (Subgoal1R1
  isa change
  operator +
  argument A
  string Right1-1
  result NewRight)
 (NewGoal1-1
  isa transform-string
  left NewLeft
  right NewRight)

;Adds a - A to the left hand side
;of the equation
 (Subgoal1L2-Dependency
  isa dependency
  goal Subgoal1L2
  subgoals (Left1-2)
  modified (Subgoal1L3)
  constraints (+ Left1-1)
  dont-cares (addition)
  generals (blank1 X)
  success 1
  actions ((!pop!)))
 (Subgoal1L2
  isa change
  operator +
  argument A
  string Left1-1
  result nil)
 (Left1-2
  isa expression
  specop blank1
  specarg X
  op1 +
  arg1 A
  op2 -
  arg2 A)
 (Subgoal1L3
  isa change
  operator +
  argument A
  string Left1-1
  result Left1-2)

;Adds a - A to the right hand side
;of the equation
 (Subgoal1R2-Dependency
  isa dependency
  goal Subgoal1R2
  subgoals (Right1-2)
  modified (Subgoal1R3)
  constraints (+ Right1-1)
  dont-cares (addition)
  generals (blank1 blank2)
  success 1
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  actions ((!pop!)))
 (Subgoal1R2
  isa change
  operator +
  argument A
  string Right1-1
  result nil)
 (Right1-2
  isa expression
  specop blank1
  specarg blank2
  op1 *
  arg1 B
  op2 -
  arg2 A)
 (Subgoal1R3
  isa change
  operator +
  argument A
  string Right1-1
  result Right1-2)

;Cancels the + A - A on the left
;side of the equation
 (Example1Line2-Dependency
  isa dependency
  goal Example1Line2
  subgoals (Left1-3)
  modified (Subgoal1L4)
  constraints (Left1-2)
  success 1
  generals (+ - A))
 (Example1Line2
  isa transform-string
  left Left1-2
  right Right1-2)
 (Left1-3
  isa expression
  specop blank1
  specarg X)
 (Subgoal1L4
  isa transform-string
  left Left1-3
  right Right1-2))

;;
;;  Example 2
;;  - X * C = - A
;;
(addwm
;Used by analogy mechanism to set
;the initial 2 subgoals of adding
;to both sides
 (Example2Line1-Dependency
  isa dependency
  goal Example2Line1
  subgoals (Subgoal2L1 Subgoal2R1)
  modified (Newgoal2-1)
  constraints (Left2-1)
  dont-cares (minus blank1 blank2
      X))
 (Example2Line1
  isa transform-string
  left Left2-1
  right Right2-1)
 (Left2-1
  isa expression
  specop minus
  specarg X
  op1 *
  arg1 C)
 (Right2-1
  isa expression
  specop blank1
  specarg blank2
  op1 -
  arg1 A)
 (Subgoal2L1
  isa change
  operator *
  argument C
  string Left2-1
  result NewLeft)
 (Subgoal2R1
  isa change
  operator *
  argument C
  string Right2-1
  result NewRight)
 (NewGoal2-1
  isa transform-string
  left NewLeft
  right NewRight)

;Adds a / C to the left hand side
;of the equation
 (Subgoal2L2-Dependency
  isa dependency
  goal Subgoal2L2
  subgoals (Left2-2)
  modified (Subgoal2L3)
  constraints (* Left2-1)
  dont-cares (multiplication)
  generals (minus X)
  success 1
  actions ((!pop!)))
 (Subgoal2L2
  isa change
  operator *
  argument C
  string Left2-1
  result nil)
 (Left2-2
  isa expression
  specop minus
  specarg X
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  op1 *
  arg1 C
  op2 /
  arg2 C)
 (Subgoal2L3
  isa change
  operator *
  argument C
  string Left2-1
  result Left2-2)

;Adds a / C to the right hand side
;of the equation
 (Subgoal2R2-Dependency
  isa dependency
  goal Subgoal2R2
  subgoals (Right2-2)
  modified (Subgoal2R3)
  constraints (* Right2-1)
  dont-cares (multiplication)
  generals (blank1 blank2)
  success 1
  actions ((!pop!)))
 (Subgoal2R2
  isa change
  operator *
  argument C
  string Right2-1
  result nil)
 (Right2-2
  isa expression
  specop blank1
  specarg blank2
  op1 -
  arg1 A
  op2 /
  arg2 C)
 (Subgoal2R3
  isa change
  operator *
  argument C
  string Right2-1
  result Right2-2)

;Cancels the * C / C on the left
;side of the equation
 (Example2Line2-Dependency
  isa dependency
  goal Example2Line2
  subgoals (Left2-3)
  modified (Subgoal2L4)
  constraints (Left2-2)
  success 1
  generals (* / C))
 (Example2Line2
  isa transform-string
  left Left2-2
  right Right2-2)
 (Left2-3
  isa expression
  specop minus
  specarg X)
 (Subgoal2L4
  isa transform-string
  left Left2-3
  right Right2-2)

;Used by analogy mechanism to set
;the subgoals of eliminating the
;sign in front of X, then doing
;correct thing to the RHS
 (Example2Line3-Dependency
  isa dependency
  goal Example2Line3
  subgoals (Subgoal2L5 Subgoal2R4)
  modified (NewGoal2-2)
  constraints (Left2-4))
 (Example2Line3
  isa transform-string
  left Left2-4
  right Right2-2)
 (Subgoal2L5
  isa change
  operator minus
  string Left2-4
  result NewLeft)
 (Left2-4
  isa expression
  specop minus
  specarg X)
 (Subgoal2R4
  isa setup
  operator minus
  string Right2-2
  result NewRight)
 (NewGoal2-2
  isa transform-string
  left NewLeft
  right NewRight)

; Remove sign in front of X
 (Subgoal2L6-Dependency
  isa dependency
  goal Subgoal2L6
  subgoals (Left2-5)
  modified (Subgoal2L7)
  constraints (Left2-4)
  generals (minus)
  success 1
  actions ((!pop!)))
 (Subgoal2L6
  isa change
  operator minus
  string Left2-4
  result nil)
 (Left2-5
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  isa expression
  specop nil
  specarg X)
 (Subgoal2L7
  isa change
  operator minus
  string Left2-4
  result Left2-5)

; Set up RHS for possible
; inversion, subgoaling on the two
; pairs
 (Subgoal2R5-Dependency
  isa dependency
  goal Subgoal2R5
  subgoals (Right2-3 Right2-4)
  modified (Subgoal2R6)
  constraints (Right2-2)
  specifics (minus))
 (Subgoal2R5
  isa setup
  operator minus
  string Right2-2
  result nil)
 (Right2-3
  isa invert
  operator minus
  op1 -
  arg1 A)
 (Right2-4
  isa invert
  operator minus
  op1 /
  arg1 C)
 (Subgoal2R6
  isa setup
  string Right2-2
  result Right2-2
  left Right2-3
  right Right2-4)

; Invert the first op
 (Right2-3-Dependency
  isa dependency
  goal Right2-3
  modified (Right2-5)
  constraints (+)
  success 1)
 (Right2-5
  isa invert
  op1 +
  arg1 A)

; Leave the second one
 (Right2-4-Dependency
  isa dependency
  goal Right2-4
  modified (Right2-6)
  constraints (/)
  dont-cares (*)
  success 1)
 (Right2-6
  isa invert
  op1 /
  arg1 C))

;;
;;  Example 3
;;  / X = * C + B
;;
(addwm
;Used by analogy mechanism to set
;the initial 2 subgoals of adding
;to both sides
 (Example3Line1-Dependency
  isa dependency
  goal Example3Line1
  subgoals (Subgoal3L1 Subgoal3R1)
  modified (NewGoal3-1)
  constraints (Left3-1))
 (Example3Line1
  isa transform-string
  left Left3-1
  right Right3-1)
 (Subgoal3L1
  isa change
  operator divide
  string Left3-1
  result NewLeft)
 (Left3-1
  isa expression
  specop divide
  specarg X)
 (Right3-1
  isa expression
  specop blank1
  specarg blank2
  op1 *
  arg1 C
  op2 +
  arg2 B)
 (Subgoal3R1
  isa setup
  operator divide
  string Right3-1
  result NewRight)
 (NewGoal3-1
  isa transform-string
  left NewLeft
  right NewRight)

; Remove sign in front of X
 (Subgoal3L2-Dependency
  isa dependency
  goal Subgoal3L2
  subgoals (Left3-2)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
   modified (Subgoal3L3)
   constraints (Left3-1)
   success 1
   actions ((!pop!)))
  (Subgoal3L2
   isa change
   operator divide
   string Left3-1
   result nil)
  (Left3-2
   isa expression
   specop nil
   specarg X)
  (Subgoal3L3
   isa change
   operator divide
   string Left3-1
   result Left3-2)

; Switch the two operands around
(Subgoal3R2-Dependency
   isa dependency
   goal Subgoal3R2
   subgoals (Right3-2)
   modified (Subgoal3R3)
   constraints (Right3-1)
   success 1
   actions ((!pop!)))
  (Subgoal3R2
   isa setup
   operator divide
   string Right3-1
   result nil)
  (Right3-2
   isa expression
   specop blank1
   specarg blank2
   op1 *
   arg1 B
   op2 +
   arg2 C)
  (Subgoal3R3
   isa setup
   string Right3-1
   result Right3-2))

;;
;;  Example 4
;;  X - C = - A
;;
(addwm
;Used by analogy mechanism to set
;the initial 2 subgoals of adding
;to both sides
 (Example4Line1-Dependency
  isa dependency
  goal Example4Line1
  subgoals (Subgoal4L1 Subgoal4R1)
   modified (Newgoal4-1)
   constraints (Left4-1)
   dont-cares (blank1 blank2 X))
  (Example4Line1
   isa transform-string
   left Left4-1
   right Right4-1)
  (Left4-1
   isa expression
   specop blank1
   specarg X
   op1 -
   arg1 C)
  (Right4-1
   isa expression
   specop blank1
   specarg blank2
   op1 -
   arg1 A)
  (Subgoal4L1
   isa change
   operator -
   argument C
   string Left4-1
   result NewLeft)
  (Subgoal4R1
   isa change
   operator -
   argument C
   string Right4-1
   result NewRight)
  (NewGoal4-1
   isa transform-string
   left NewLeft
   right NewRight)

;Adds a + C to the left hand side
;of the equation
 (Subgoal4L2-Dependency
  isa dependency
  goal Subgoal4L2
  subgoals (Left4-2)
  modified (Subgoal4L3)
  constraints (- Left4-1)
  dont-cares (addition)
  generals (blank1 X)
  success 1
  actions ((!pop!)))
 (Subgoal4L2
  isa change
  operator -
  argument C
  string Left4-1
  result nil)
 (Left4-2
  isa expression
  specop blank1
  specarg X
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  op1 -
  arg1 C
  op2 +
  arg2 C)
 (Subgoal4L3
  isa change
  operator -
  argument C
  string Left4-1
  result Left4-2)

;Adds a + C to the right hand side
;of the equation
 (Subgoal4R2-Dependency
  isa dependency
  goal Subgoal4R2
  subgoals (Right4-2)
  modified (Subgoal4R3)
  constraints (- Right4-1)
  dont-cares (addition)
  generals (blank1 blank2)
  success 1
  actions ((!pop!)))
 (Subgoal4R2
  isa change
  operator -
  argument C
  string Right4-1
  result nil)
 (Right4-2
  isa expression
  specop blank1
  specarg blank2
  op1 -
  arg1 A
  op2 +
  arg2 C)
 (Subgoal4R3
  isa change
  operator -
  argument C
  string Right4-1
  result Right4-2)

;Cancels the - C + C on the left
;side of the equation
 (Example4Line2-Dependency
  isa dependency
  goal Example4Line2
  subgoals (Left4-3)
  modified (Subgoal4L4)
  constraints (Left4-2)
  success 1
  generals (- + C))
 (Example4Line2
  isa transform-string
  left Left4-2
  right Right4-2)
 (Left4-3
  isa expression
  specop blank1
  specarg X)
 (Subgoal4L4
  isa transform-string
  left Left4-3
  right Right4-2))

;;
;;  Example 5
;;   * X = * C / D
;;
(addwm
;Used by analogy mechanism to set
;the initial 2 subgoals of adding
;to both sides
 (Example5Line1-Dependency
  isa dependency
  goal Example5Line1
  subgoals (Subgoal5L1 Subgoal5R1)
  modified (NewGoal5-1)
  constraints (Left5-1))
 (Example5Line1
  isa transform-string
  left Left5-1
  right Right5-1)
 (Subgoal5L1
  isa change
  operator multiply
  string Left5-1
  result NewLeft)
 (Left5-1
  isa expression
  specop multiply
  specarg X)
 (Right5-1
  isa expression
  specop blank1
  specarg blank2
  op1 *
  arg1 C
  op2 /
  arg2 D)
 (Subgoal5R1
  isa setup
  operator multiply
  string Right5-1
  result NewRight)
 (NewGoal5-1
  isa transform-string
  left NewLeft
  right NewRight)

; Remove sign in front of X
 (Subgoal5L2-Dependency
  isa dependency
  goal Subgoal5L2
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  subgoals (Left5-2)
  modified (Subgoal5L3)
  constraints (Left5-1)
  success 1
  actions ((!pop!)))
 (Subgoal5L2
  isa change
  operator multiply
  string Left5-1
  result nil)
 (Left5-2
  isa expression
  specop nil
  specarg X)
 (Subgoal5L3
  isa change
  operator multiply
  string Left5-1
  result Left5-2)

;Set up RHS for possible
;inversion, subgoaling on the two
;pairs
 (Subgoal5R2-Dependency
  isa dependency
  goal Subgoal5R2
  subgoals (Right5-2 Right5-3)
  modified (Subgoal5R3)
  constraints (Right5-1)
  specifics (multiply))
 (Subgoal5R2
  isa setup
  operator multiply
  string Right5-1
  result nil)
 (Right5-2
  isa invert
  operator multiply
  op1 *
  arg1 C)
 (Right5-3
  isa invert
  operator multiply
  op1 /
  arg1 D)
 (Subgoal5R3
  isa setup
  string Right5-1
  result Right5-1
  left Right5-2
  right Right5-3)

; Invert the first op
 (Right5-2-Dependency
  isa dependency
  goal Right5-2
  modified (Right5-4)
  constraints (*)
  success 1)
 (Right5-4
  isa invert
  op1 /
  arg1 C)

; Invert the second op
 (Right5-3-Dependency
  isa dependency
  goal Right5-3
  modified (Right5-5)
  constraints (*)
  success 1)
 (Right5-5
  isa invert
  op1 *
  arg1 D))

;;
;;  Example 6
;;  + X + B = / C
;;
(addwm
;Used by analogy mechanism to set
;the initial 2 subgoals of adding
;to both sides
 (Example6Line1-Dependency
  isa dependency
  goal Example6Line1
  subgoals (Subgoal6L1 Subgoal6R1)
  modified (Newgoal6-1)
  constraints (Left6-1)
  dont-cares (plus blank1 blank2
      X))
 (Example6Line1
  isa transform-string
  left Left6-1
  right Right6-1)
 (Left6-1
  isa expression
  specop plus
  specarg X
  op1 +
  arg1 B)
 (Right6-1
  isa expression
  specop blank1
  specarg blank2
  op1 /
  arg1 C)
 (Subgoal6L1
  isa change
  operator +
  argument B
  string Left6-1
  result NewLeft)
 (Subgoal6R1
  isa change
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  operator +
  argument B
  string Right6-1
  result NewRight)
 (NewGoal6-1
  isa transform-string
  left NewLeft
  right NewRight)

;Adds a - B to the left hand side
;of the equation
 (Subgoal6L2-Dependency
  isa dependency
  goal Subgoal6L2
  subgoals (Left6-2)
  modified (Subgoal6L3)
  constraints (+ Left6-1)
  dont-cares (addition)
  generals (plus X)
  success 1
  actions ((!pop!)))
 (Subgoal6L2
  isa change
  operator +
  argument B
  string Left6-1
  result nil)
 (Left6-2
  isa expression
  specop plus
  specarg X
  op1 +
  arg1 B
  op2 -
  arg2 B)
 (Subgoal6L3
  isa change
  operator +
  argument B
  string Left6-1
  result Left6-2)

;Adds a - B to the right hand side
;of the equation
 (Subgoal6R2-Dependency
  isa dependency
  goal Subgoal6R2
  subgoals (Right6-2)
  modified (Subgoal6R3)
  constraints (+ Right6-1)
  dont-cares (addition)
  generals (blank1 blank2)
  success 1
  actions ((!pop!)))
 (Subgoal6R2
  isa change
  operator +
  argument B
  string Right6-1
  result nil)
 (Right6-2
  isa expression
  specop blank1
  specarg blank2
  op1 /
  arg1 C
  op2 -
  arg2 B)
 (Subgoal6R3
  isa change
  operator +
  argument B
  string Right6-1
  result Right6-2)

;Cancels the * C / C on the left
;side of the equation
 (Example6Line2-Dependency
  isa dependency
  goal Example6Line2
  subgoals (Left6-3)
  modified (Subgoal6L4)
  constraints (Left6-2)
  success 1
  generals (+ - B))
 (Example6Line2
  isa transform-string
  left Left6-2
  right Right6-2)
 (Left6-3
  isa expression
  specop plus
  specarg X)
 (Subgoal6L4
  isa transform-string
  left Left6-3
  right Right6-2)

;Used by analogy mechanism to set
;the subgoals of eliminating the
;sign in front of X, then doing
;correct thing to the RHS
 (Example6Line3-Dependency
  isa dependency
  goal Example6Line3
  subgoals (Subgoal6L5 Subgoal6R4)
  modified (NewGoal6-2)
  constraints (Left6-4))
 (Example6Line3
  isa transform-string
  left Left6-4
  right Right6-2)
 (Subgoal6L5
  isa change
  operator plus
  string Left6-4
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  result NewLeft)
 (Left6-4
  isa expression
  specop plus
  specarg X)
 (Subgoal6R4
  isa setup
  operator plus
  string Right6-2
  result NewRight)
 (NewGoal6-2
  isa transform-string
  left NewLeft
  right NewRight)

; Remove sign in front of X
 (Subgoal6L6-Dependency
  isa dependency
  goal Subgoal6L6
  subgoals (Left6-5)
  modified (Subgoal6L7)
  constraints (Left6-4)
  generals (plus)
  success 1
  actions ((!pop!)))
 (Subgoal6L6
  isa change
  operator plus
  string Left6-4
  result nil)
 (Left6-5
  isa expression
  specop nil
  specarg X)
 (Subgoal6L7
  isa change
  operator plus
  string Left6-4
  result Left6-5)

;Nothing happens to the RHS for
;plus elim
 (Subgoal6R5-Dependency
  isa dependency
  goal Subgoal6R5
  subgoals (Right6-3)
  modified (Subgoal6R6)
  constraints (Right6-2)
  success 1
  actions ((!pop!)))
 (Subgoal6R5
  isa setup
  operator plus
  string Right6-2
  result nil)
 (Right6-3
  isa expression
  specop blank1
  specarg blank2
  op1 /
  arg1 C
  op2 -
  arg2 B)
 (Subgoal6R6
  isa setup
  string Right6-2
  result Right6-3))

;;
;;  Example 7
;;  X * C = / B
;;
(addwm
;Used by analogy mechanism to set
;the initial 2 subgoals of adding
;to both sides
 (Example7Line1-Dependency
  isa dependency
  goal Example7Line1
  subgoals (Subgoal7L1 Subgoal7R1)
  modified (Newgoal7-1)
  constraints (left7-1)
  dont-cares (blank1 blank2 X))
 (Example7Line1
  isa transform-string
  left left7-1
  right right7-1)
 (left7-1
  isa expression
  specop blank1
  specarg X
  op1 *
  arg1 C)
 (right7-1
  isa expression
  specop blank1
  specarg blank2
  op1 /
  arg1 B)
 (Subgoal7L1
  isa change
  operator *
  argument C
  string left7-1
  result NewLeft)
 (Subgoal7R1
  isa change
  operator *
  argument C
  string right7-1
  result NewRight)
 (Newgoal7-1
  isa transform-string
  left NewLeft
  right NewRight)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
;Adds a / C to the left hand side
;of the equation
 (Subgoal7L2-Dependency
  isa dependency
  goal Subgoal7L2
  subgoals (left7-2)
  modified (Subgoal7L3)
  constraints (* left7-1)
  dont-cares (addition)
  generals (blank1 X)
  success 1
  actions ((!pop!)))
 (Subgoal7L2
  isa change
  operator *
  argument C
  string left7-1
  result nil)
 (left7-2
  isa expression
  specop blank1
  specarg X
  op1 *
  arg1 C
  op2 /
  arg2 C)
 (Subgoal7L3
  isa change
  operator -
  argument C
  string left7-1
  result left7-2)

;Adds a / C to the right hand side
;of the equation
 (Subgoal7R2-Dependency
  isa dependency
  goal Subgoal7R2
  subgoals (right7-2)
  modified (Subgoal7R3)
  constraints (* right7-1)
  dont-cares (addition)
  generals (blank1 blank2)
  success 1
  actions ((!pop!)))
 (Subgoal7R2
  isa change
  operator *
  argument C
  string right7-1
  result nil)
 (right7-2
  isa expression
  specop blank1
  specarg blank2
  op1 /
  arg1 B
  op2 /
  arg2 C)
 (Subgoal7R3
  isa change
  operator -
  argument C
  string right7-1
  result right7-2)

;Cancels the * C / C on the left
;side of the equation
 (Example7Line2-Dependency
  isa dependency
  goal Example7Line2
  subgoals (left7-3)
  modified (Subgoal7L4)
  constraints (left7-2)
  actions ((!pop!))
  success 1
  generals (* / C))
 (Example7Line2
  isa transform-string
  left left7-2
  right right7-2)
 (left7-3
  isa expression
  specop blank1
  specarg X)
 (Subgoal7L4
  isa transform-string
  left left7-3
  right right7-2))

;;
;;  Example 8
;;  / X + C = + D
;;
(addwm
;Used by analogy mechanism to set
;the initial 2 subgoals of adding
;to both sides
 (Example8Line1-Dependency
  isa dependency
  goal Example8Line1
  subgoals (Subgoal8L1 Subgoal8R1)
  modified (Newgoal8-1)
  constraints (left8-1)
  dont-cares (divide blank1 blank2
      X))
 (Example8Line1
  isa transform-string
  left left8-1
  right right8-1)
 (left8-1
  isa expression
  specop divide
  specarg X
  op1 +
  arg1 C)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
(right8-1
 isa expression
 specop blank1
 specarg blank2
 op1 +
 arg1 D)
(Subgoal8L1
 isa change
 operator +
 argument C
 string left8-1
 result NewLeft)
(Subgoal8R1
 isa change
 operator +
 argument C
 string right8-1
 result NewRight)
(Newgoal8-1
 isa transform-string
 left NewLeft
 right NewRight)

;Adds a - C to the left hand side
;of the equation
(Subgoal8L2-Dependency
 isa dependency
 goal Subgoal8L2
 subgoals (left8-2)
 modified (Subgoal8L3)
 constraints (+ left8-1)
 dont-cares (addition)
 generals (divide X)
 success 1
 actions ((!pop!)))
(Subgoal8L2
 isa change
 operator +
 argument C
 string left8-1
 result nil)
(left8-2
 isa expression
 specop divide
 specarg X
 op1 +
 arg1 C
 op2 -
 arg2 C)
(Subgoal8L3
 isa change
 operator +
 argument C
 string left8-1
 result left8-2)

;Adds a - C to the left hand side
;of the equation
(Subgoal8R2-Dependency
 isa dependency
 goal Subgoal8R2
 subgoals (right8-2)
 modified (Subgoal8R3)
 constraints (+ right8-1)
 dont-cares (addition)
 generals (blank1 blank2)
 success 1
 actions ((!pop!)))
(Subgoal8R2
 isa change
 operator +
 argument C
 string right8-1
 result nil)
(right8-2
 isa expression
 specop blank1
 specarg blank2
 op1 +
 arg1 D
 op2 -
 arg2 C)
(Subgoal8R3
 isa change
 operator +
 argument C
 string right8-1
 result right8-2)

;Cancels the + C - C on the left
;side of the equation
(Example8Line2-Dependency
 isa dependency
 goal Example8Line2
 subgoals (left8-3)
 modified (Subgoal8L4)
 constraints (left8-2)
 success 1
 generals (+ - C))
(Example8Line2
 isa transform-string
 left left8-2
 right right8-2)
(left8-3
 isa expression
 specop divide
 specarg X)
(Subgoal8L4
 isa transform-string
 left left8-3
 right right8-2)

;Used by analogy mechanism to set
;the subgoals of eliminating the
;sign in front of X, then doing
;correct thing to the RHS
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
(Example8Line3-Dependency
 isa dependency
 goal Example8Line3
 subgoals (Subgoal8L5 Subgoal8R4)
 modified (Newgoal8-2)
 constraints (left8-4))
(Example8Line3
 isa transform-string
 left left8-4
 right right8-2)
(Subgoal8L5
 isa change
 operator divide
 string left8-4
 result NewLeft)
(left8-4
 isa expression
 specop divide
 specarg X)
(Subgoal8R4
 isa setup
 operator divide
 string right8-2
 result NewRight)
(Newgoal8-2
 isa transform-string
 left NewLeft
 right NewRight)

; Remove sign in front of X
(Subgoal8L6-Dependency
 isa dependency
 goal Subgoal8L6
 subgoals (left8-5)
 modified (Subgoal8L7)
 constraints (left8-4)
 generals (divide)
 success 1
 actions ((!pop!)))
(Subgoal8L6
 isa change
 operator divide
 string left8-4
 result nil)
(left8-5
 isa expression
 specop nil
 specarg X)
(Subgoal8L7
 isa change
 operator divide
 string left8-4
 result left8-5)

; Switch the two operands around
(Subgoal8R5-Dependency
 isa dependency
 goal Subgoal8R5
 subgoals (Right8-3)
 modified (Subgoal8R6)
 constraints (right8-2)
 success 1
 actions ((!pop!)))
(Subgoal8R5
 isa setup
 operator divide
 string right8-2
 result nil)
(right8-3
 isa expression
 specop blank1
 specarg blank2
 op1 +
 arg1 C
 op2 -
 arg2 D)
(Subgoal8R6
 isa setup
 string right8-2
 result right8-3))

(wmfocus problem1)

(p glue
   =subgoal>
      isa setup
      result =original
      left =part1
      right =part2
   =part1>
      isa invert
      op1 =op1
      arg1 =arg1
   =part2>
      isa invert
      op1 =op2
      arg1 =arg2
==>
   =original>
      isa expression
      op1 =op1
      arg1 =arg1
      op2 =op2
      arg2 =arg2
   !pop!)

(p detectgoalstate
   =goal>
      isa transform-string
      left =left
   =left>
      isa expression
      specop nil
      specarg X
==>
   !pop!)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix D: Model Run

Below is a listing of the model solving five problems. These five problems are:

1) X - A = * C
2) + X = - B / D
3) / X * D = + B
4) - X = - D * A
5) * X = * A + C

In the actual runs (the cycle statements), the productions that are being created by the analogy mechanism are bolded. After the model has solved the problem, those productions which were newly created are titled and displayed. The notation in the parentheses (like P1) refers to the production numbers in Chapter 5, which illustrate how those particular productions arose.

The model is solving the problem:
X - A = * C

```
? (run)
 cycle 0  time  0.000: transform-
            string-production42
 action latency:  0.050

       cycle 1  time  0.050:
            change-production47
       action latency:  0.050

    cycle 2  time  0.100: change-
            production47
    action latency:  0.050

 cycle 3  time  0.150: transform-
            string-production48
 action latency:  0.050
```

1) Production that sets up either a two– or three–step problem (P1):

```
(p transform-string-production42
   =example8line1-variable>
      isa transform-string
      left =left8-1-variable
      right =right8-1-variable
   =left8-1-variable>
      isa expression
      op1 =+-variable
      arg1 =c-variable
      op2 nil
      arg2 nil
==>
   =subgoal8l1-variable>
      isa change
      operator =+-variable
      argument =c-variable
      string =left8-1-variable
      result =newleft-variable
   =subgoal8r1-variable>
      isa change
      operator =+-variable
      argument =c-variable
      string =right8-1-variable
      result =newright-variable
   =example8line1-variable>
      left =newleft-variable
      right =newright-variable
   !Push! =subgoal8r1-variable
   !Push! =subgoal8l1-variable)
```

2) Production adds the proper operator and operand to one side of the equation (P2):

```
(p change-production47
   =subgoal6r2-variable>
      isa change
      operator =+-variable
      argument =b-variable
      string =right6-1-variable
      result nil
   =+-variable>
      isa operator
      inverse =--variable
   =right6-1-variable>
      isa expression
      specop =blank1-variable
      specarg =blank2-variable
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        op1 =/-variable
        arg1 =c-variable
        op2 nil
        arg2 nil
 ==>
    =right6-2-variable>
        isa expression
        specop =blank1-variable
        specarg =blank2-variable
        op1 =/-variable
        arg1 =c-variable
        op2 =--variable
        arg2 =b-variable
    =subgoal6r2-variable>
        result =right6-2-variable
    !Push! =right6-2-variable
    !Pop!
    !Pop!)
```

3) Production that eliminates four symbols from the LHS (P3):

```
(p transform-string-production48
    =example8line2-variable>
        isa transform-string
        left =left8-2-variable
        right =right8-2-variable
    =left8-2-variable>
        isa expression
        specop =divide-variable
        specarg =x-variable
        op1 =+-variable
        arg1 =c-variable
        op2 =--variable
        arg2 =c-variable
 ==>
    =left8-3-variable>
        isa expression
        specop =divide-variable
        specarg =x-variable
        op1 nil
        arg1 nil
        op2 nil
        arg2 nil
    =example8line2-variable>
        left =left8-3-variable
    !Push! =left8-3-variable
    !Pop!)
```

The model is solving the problem: + X = - B / D

```
Cycle 4  time  0.200: transform-
         string-production49
Action latency:  0.050

    cycle 5  time  0.250:
             change-production49
    action latency:  0.050

  cycle 6  time  0.300: setup-
           production50
  action latency:  0.050

Cycle 7  time  0.350:
         detectgoalstate
Action latency:  0.050

Top goal popped.
Run latency:  0.200
```

4) Production that sets up the sign elimination step (P4):

```
(p transform-string-production49
   =example8line3-variable>
      isa transform-string
      left =left8-4-variable
      right =right8-2-variable
   =left8-4-variable>
      isa expression
      specop =divide-variable
      specarg x
      op1 nil
      arg1 nil
      op2 nil
      arg2 nil
 ==>
   =subgoal815-variable>
      isa change
      operator =divide-variable
      argument nil
      string =left8-4-variable
      result =newleft-variable
   =subgoal8r4-variable>
      isa setup
      operator =divide-variable
      argument nil
      string =right8-2-variable
      result =newright-variable
      left nil
      right nil
   =example8line3-variable>
      left =newleft-variable
      right =newright-variable
   !Push! =subgoal8r4-variable
   !Push! =subgoal815-variable)
```

5) Production that deletes the sign in front of X (P5):

```
(p change-production49
   =subgoal816-variable>
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        isa change
        operator =divide-variable
        argument nil
        string =left8-4-variable
        result nil
    =left8-4-variable>
        isa expression
        specop =divide-variable
        specarg =x-variable
        op1 nil
        arg1 nil
        op2 nil
        arg2 nil
==>
    =left8-5-variable>
        isa expression
        specop nil
        specarg =x-variable
        op1 nil
        arg1 nil
        op2 nil
        arg2 nil
    =subgoal816-variable>
        result =left8-5-variable
    !Push! =left8-5-variable
    !Pop!
    !Pop!)
```

6) Production that does plus (®) elimination:

```
(p setup-production50
    =subgoal6r5-variable>
        isa setup
        operator plus
        argument nil
        string =right6-2-variable
        result nil
        left nil
        right nil
    =right6-2-variable>
        isa expression
        specop =blank1-variable
        specarg =blank2-variable
        op1 =/-variable
        arg1 =c-variable
        op2 =--variable
        arg2 =b-variable
==>
    =right6-3-variable>
        isa expression
        specop =blank1-variable
        specarg =blank2-variable
        op1 =/-variable
        arg1 =c-variable
        op2 =--variable
        arg2 =b-variable
    =subgoal6r5-variable>
        operator nil
        result =right6-3-variable
    !Push! =right6-3-variable
    !Pop!
    !Pop!)
```

*The model is solving the problem: / X * D = + B*

```
Cycle 8  time  0.400: transform-
            string-production42
Action latency:  0.050

        cycle 9  time  0.450:
            change-production47
        action latency:  0.050

    cycle 10  time  0.500: change-
            production47
    action latency:  0.050

Cycle 11  time  0.550: transform-
            string-production48
Action latency:  0.050

Cycle 12  time  0.600: transform-
            string-production49
Action latency:  0.050

        cycle 13  time  0.650:
            change-production49
        action latency:  0.050

    cycle 14  time  0.700: setup-
            production51
    action latency:  0.050

Cycle 15  time  0.750:
            detectgoalstate
Action latency:  0.050

Top goal popped.
Run latency:  0.400
```

7) Production that does divide (©) elimination:

```
(p setup-production51
    =subgoal8r5-variable>
        isa setup
        operator divide
        argument nil
        string =right8-2-variable
        result nil
        left nil
        right nil
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    =right8-2-variable>
       isa expression
       specop =blank1-variable
       specarg =blank2-variable
       op1 =+-variable
       arg1 =d-variable
       op2 =--variable
       arg2 =c-variable
 ==>
    =right8-3-variable>
       isa expression
       specop =blank1-variable
       specarg =blank2-variable
       op1 =+-variable
       arg1 =c-variable
       op2 =--variable
       arg2 =d-variable
    =subgoal8r5-variable>
       operator nil
       result =right8-3-variable
    !Push! =right8-3-variable
    !Pop!
    !Pop!)
```

The model is solving the problem:
- X = - D * A

```
Cycle 16  time  0.800: transform-
          string-production49
Action latency:  0.050

     cycle 17  time  0.850:
          change-production49
     action latency:  0.050

  cycle 18  time  0.900: setup-
          production56
  action latency:  0.050

       cycle 19  time  0.950:
          invert-production59
       action latency:  0.050

     cycle 20  time  1.000:
          invert-production62
     action latency:  0.050

  cycle 21  time  1.050: glue
  action latency:  0.050

Cycle 22  time  1.100:
          detectgoalstate
Action latency:  0.050

Top goal popped.
Run latency:  0.350
```

8) Production that does minus (♥) elimination (P6):

```
(p setup-production56
    =subgoal2r5-variable>
       isa setup
       operator minus
       argument nil
       string =right2-2-variable
       result nil
       left nil
       right nil
    =right2-2-variable>
       isa expression
       specop blank1
       specarg blank2
       op1 =--variable
       arg1 =a-variable
       op2 =/-variable
       arg2 =c-variable
 ==>
    =right2-3-variable>
       isa invert
       operator minus
       op1 =--variable
       arg1 =a-variable
    =right2-4-variable>
       isa invert
       operator minus
       op1 =/-variable
       arg1 =c-variable
    =subgoal2r5-variable>
       operator nil
       result =right2-2-variable
       left =right2-3-variable
       right =right2-4-variable
    !Push! =right2-4-variable
    !Push! =right2-3-variable)
```

9) Production that inverts for minus (♥) elimination (P7):

```
(p invert-production59
    =right2-3-variable>
       isa invert
       operator minus
       op1 =--variable
       arg1 =a-variable
    =+-variable>
       isa operator
       inverse =--variable
       type addition
 ==>
    =right2-3-variable>
       operator nil
       op1 =+-variable
    !Pop!)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

10) Production that does not invert for minus (♥) elimination (P8):

```
(p invert-production62
   =right2-4-variable>
      isa invert
      operator minus
      op1 =/-variable
      arg1 =c-variable
   =/-variable>
      isa operator
      type multiplication
 ==>
   =right2-4-variable>
      operator nil
   !Pop!)
```

The model is solving the problem:
$*X = *A + C$

```
Cycle 27  time  1.350: transform-
            string-production49
Action latency:  0.050

      cycle 28  time  1.400:
            change-production49
      action latency:  0.050

   cycle 29  time  1.450: setup-
            production64
   action latency:  0.050

         cycle 30  time  1.500:
            invert-production64
         action latency:  0.050

      no instantiation found.
      run latency:  0.400
```

11) Production that does multiply (#) elimination (P6):

```
(p setup-production64
   =subgoal5r2-variable>
      isa setup
      operator multiply
      argument nil
      string =right5-1-variable
      result nil
      left nil
      right nil
   =right5-1-variable>
      isa expression
      specop blank1
      specarg blank2
      op1 =*-variable
      arg1 =c-variable
      op2 =/-variable
      arg2 =d-variable
==>
   =right5-2-variable>
      isa invert
      operator multiply
      op1 =*-variable
      arg1 =c-variable
   =right5-3-variable>
      isa invert
      operator multiply
      op1 =/-variable
      arg1 =d-variable
   =subgoal5r2-variable>
      operator nil
      result =right5-1-variable
      left =right5-2-variable
      right =right5-3-variable
   !Push! =right5-3-variable
   !Push! =right5-2-variable)
```

12) Production that inverts for multiply (#) elimination (P7):

```
(p invert-production64
   =right5-3-variable>
      isa invert
      operator multiply
      op1 =/-variable
      arg1 =d-variable
   =*-variable>
      isa operator
      inverse =/-variable
      type multiplication
 ==>
   =right5-3-variable>
      operator nil
      op1 =*-variable)
```

Note that the main 8 Examples (Table 3.3) do not have an example of a sign not inverting during # elimination. Going off these examples, ACT–R cannot generate the last production necessary to do all problems. One way around this is to have the model remember the past problems it has solved, and have those as reference as well (such a model is trivial and has

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

been implemented). This last production looks like this:

13) Production that does not invert for multiply (#) elimination (P7):

```
(p invert-production64
    =right5-3-variable>
       isa invert
       operator multiply
       op1 =/-variable
       arg1 =d-variable
    =+-variable>
       isa operator
       type addition
 ==>
    =right5-3-variable>
       operator nil)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix E: Example Protocol

This protocol is from Participant #22 and was taken on March 15, 1995. He was in the Examples group of Experiment 1. In the following transcription, P is the Participant, E is the Experimenter, and C is the Computer. Lines that are asterisked and italicized indicate either what the participant typed, what information the computer gave, or specific examples referred to by the participant. The second column (appearing through Problem 19) contains comments concerning the participant's acquisition of the rules of the task, including references to rules in Appendix D.

Problem 1: ℘®Γ↔©Φ

Protocol	Notes
P: Problem #1. Workspace. Right now I'm just putting in the exact same thing they have.	
* *Participant typed ℘®Γ↔©Φ*	
C: Try again.	
P: Okay. Umm. Examples 1. Going to the examples. Umm mumble Click on a box to reveal the whole problem. Umm.	
E: This up here is showing you the last correct thing that has been typed in.	
P: Oh, okay. Okay. So I'm going to look for a match. With the first three characters. I don't see one. Okay, for the next two? See. No. We have something like, something similar. So I'll try... Umm, Example 7, the last correct line, everything is the same except for this R comes this number symbol, and this phi, I guess, becomes a delta. So, if R is a number symbol...	
* *Participant typed ®↔#*	
C: Here's the correct line.	
* Computer responded ℘®Γ♥Γ↔©Φ♥Γ	
P: No. Okay. Heart, heart that, heart that. Okay. Well. Umm. Okay, so they just added on to what they had. So maybe I'll try adding on to it. You got the heart, and it's not clear to me... Example 1, R and heart. That's what I had before. So next line, well, we'll try that.	

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

** Participant referred to Example 1*
P: We're just going to go by Example 1, mumble
** Participant typed ℘↔©Φ♥Γ*
C: Excellent.
P: Oh, okay.

Participant has no understanding of the inverse relation of the ® and ♥ but rather simply deletes the four symbols.

Problem 2: #℘↔#Δ©Γ

P: All right, I'll try what they did, on the first problem. Oops.
** Participant typed #℘♥℘↔#Δ©Γ♥℘*
C: Try again.
P: Okay. Go back to the examples. Select this one. Swap those. We'll try example 3.
** Participant referred to Example 1*
** Participant typed ℘↔#Γ©Δ*
C: Here's the correct line.
** Computer responded ℘®Γ♥Γ↔©Φ♥Γ*
P: This is the correct line? Okay. Oh, they swapped... Hmm, okay. Swapped those two. Go to next problem.
** Participant referred to Example 3*

Participant recognizes that in some cases the same thing needs to be added to both sides of the string, but has no idea of when that is appropriate or what exactly to add.

Participant assumed the operators have swapped, not inverted.

Problem 3: ℘♥Γ↔©Φ

P: So heart, C. This one is exactly like, three characters, two. And that one is three. So example 1 and example 4 and example 7 have the same form, 3 characters, with arrow, 2 characters. So, I'm going to assume, you can solve it by one of the these examples. Using one of these examples.
** Participant referred to Example 4*
P: Umm, C. Okay, we'll try, we'll try it by example 4. R, we just add R.
** Participant typed ℘♥Γ®Γ↔©Φ®Γ*
C: Good.
P: It worked! Okay. Now click the arrow. Umm, so example 4 works, so we'll keep going with it. It keeps the first character, and everything else is the same order. So we'll go this, arrow C phi R
** Participant referred to Example 4*
** Participant typed ℘↔©Φ®Γ*
C: Excellent.

Participant picked an example based on number of symbols on either side of the character string.

No real understanding of why ® was added (a lucky guess in this case).

Again, no understanding of why the symbols can be eliminated.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Problem 4: ♥℘©Γ↔©Φ

P: Okay, problem 4. Has 4 characters and 2 character, so I'm looking for something 4, 2. Four, 2, 4, 2. This is 3 of the same, on the left hand side, so we'll try this one. Umm, see here. Keeps the heart, should add a C and last character.

** Participant referred to Examples 2 and 6*

** Participant typed ♥℘©Γ©Γ↔©Φ©Γ*

C: Try again.

P: Okay. Well, we'll try example 6 then, since it has one similar character, and what does it do? It adds a heart to the end, and last character. Try this, if this doesn't work...

** Participant referred to Examples 2, 6 and 8*

** Participant typed ♥℘©Γ♥Γ↔©Φ♥Γ*

C: Here's the correct line.

** Computer responded ♥℘©Γ#Γ↔©Φ#Γ*

** Participant referred to Examples 2, 6 and 8*

P: A number. Okay. Number symbol. Why did it add a number symbol? So it added a C?. Number symbol. Don't know why it added a number symbol. It's 2, that's 3.

** Participant referred to Examples 2, 6 and 8*

P: Okay, I guess we'll go on. Number symbol. This is on at the beginning, keeps the back the same. Okay. According to the three examples, they keep the first 2 characters, and lose everything else on the left–hand side of the arrow, and keep the right–hand side the same.

** Participant typed ♥℘↔©Φ#Γ*

C: Good.

** Participant referred to Examples 2, 6 and 8*

P: Good, and next, what do they do. They lose the very leftmost thing, and flop around... Or do they? This one flops, this with that, so... Hmm. Okay. Ahh, we'll stick this thing to the left–hand side, arrow, after that, and R in there, a C here. C just stays the same, the R flops things around, so we'll see what the C one does. C phi number this.

** Participant referred to Examples 2, 6 and 8*

** Participant typed ℘↔©Φ#Γ*

C: Excellent.

P: Okay.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Problem 5: ®℘↔©Γ♥Δ

P: Two, 4. We need something with a 2, 4. There's a 2, 4. It goes to 1, 4. It goes to 1, 4. Loses the leftmost thing, umm. Has a C, doesn't have a C. So this has a C, I'll just follow this example. Umm, instead of a C there, wait, it flopped it. If the C is at the front, I'm going to keep it the same. Okay. Arrow, let's see.

The participant is paying attention to the wrong symbol—to the first symbol of the right-hand side, not the frontmost symbol.

* *Participant referred to Examples 3, 5, and 6*

* *Participant typed ℘↔©Γ♥Δ*

C: Excellent.

Problem 6: ©℘#Φ↔®Δ

P: A 4, 2. So, a 4, 2; 4, 2. Here we go. There's a R, that has a C. So this has a C at the front, go by that. And example 8. Stick a heart with the last, yeah, and last symbol on left side. C this, number, phi, heart, phi, arrow, R triangle, heart phi.

* *Participant referred to Examples 1, 2, 6, and 8*

* *Participant typed ©℘#Φ♥Φ↔®Δ♥Φ*

C: Try again.

P: Hmm. That R has anything to do with it. No, it shouldn't. Triangle. Ends with a triangle. That shouldn't do anything. Number symbol, does it do anything. Maybe try a C, since that number symbol is there, maybe that means you're supposed to add a C. Let's try that. C number phi

The participant might have made a connection between the # and the ©.

* *Participant referred to all examples*

* *Participant typed ©℘#Φ©Φ↔®Δ©Φ*

C: Good.

P: Okay, umm. Follow example 7. Sort of. Okay, after that, all the ones that start out with 4 on the left and two on the right, umm, after they add something, they lose everything, and just keep the two characters on he left side, the two leftmost ones. Umm, so we lose all that, and what do we put on the right? Since it starts with a R, we'll follow this example, example 8. We'll just keep that and that, and R triangle C phi.

The participant used a 2-step problem to help with a 3-step problem.

* *Participant referred to Examples 2, 6, 7, and 8*

* *Participant typed ©℘↔®Δ©Φ*

C: Good.

Again, though, paying attention to the wrong symbol.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

P: And the R at the front, you flip things around. Flip the second, so we have arrow R phi C delta.
* *Participant referred to Example 8*
* *Participant typed ℘↔®Φ©Δ*

C: Excellent.
P: Okay, got that right.

Problem 7: ©℘↔#Γ#Φ

P: Got two and four. Two and four. Well, just lose it. Hmm. It's a C, since there's a C in front, we'll go by this one. Keep the number, flop those two around. Okay, I'll try that. Arrow, keep the number symbol, phi.
* *Participant referred to Examples 1, 2, 3, 4, 5 and 7*
* *Participant typed ℘↔#Φ#Γ*

C: Excellent.

However, here he did use the correct symbol to figure out the proper rule.

Problem 8: ℘®Δ↔♥Φ

P: Got it right. Three and 2. Umm, a heart. You add a R. R delta. So let's try this. R delta.
* *Participant referred to Examples 1 and 4*
* *Participant typed ℘®Δ®Δ↔♥Φ®Δ*

C: Try again.
P: What do I do now? R delta. Okay. So since it's a R, maybe we'll try this. We need three. Go by example 1. With a heart. Triangle heart delta.
* *Participant referred to Examples 1, 3, 5, 6, 7, and 8*
* *Participant typed ℘®Δ♥Δ↔♥Φ♥Δ*

C: Good.
P: Okay, so it's like example 1. And for example 1, lose everything except for the very first character on the left, keep the right the same. Okay, we'll try that. Arrow heart phi, heart delta.
* *Participant referred to Example 1*
* *Participant typed ℘↔♥Φ♥Δ*

C: Excellent.

Even here, though, believed first symbol of the right-hand side dictates what should be done.

Problem 9 #℘↔♥Φ#Γ

P: Okay, we have a 2 and 4. That flops those two. We'll try by example 5. Number symbol, phi heart, this.
* *Participant referred to Example 5*
* Participant typed ℘↔#Φ♥Γ

Made the common mistake of swapping operators for # elimination.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

C: Try again.P: It's not by example 5. Two... Keep it the same? Or we could... Six. Two, 2, 2. There's a heart there. What does it do? Changes the heart to a R. We'll try by the last two steps of example 2. Change the heart to a R.

* *Participant referred to all examples*

* *Participant typed ℘↔®Φ#Γ*

C: Here's the correct line.

P: No, don't change the heart, change the number to a C. Okay, why do we change the number sign to a C? Change a number sign to a C—that confuses me. R over heart, number symbol's over C I think that's what it says. Okay. We always end up with that. Okay.

* Participant referred to all examples

Problem10 ℘♥Ω↔©Φ

P: Three to 2. Since it's a C we'll add a, maybe we'll add a C omega. C omega. C phi C omega.

* *Participant referred to Examples 1, 4, and 7*

* *Participant typed ℘♥Ω©Ω↔©Φ©Ω*

C: Try again.

P: Okay, so it's not like example 7. So most likely add a heart or a R. If you have a heart there you add a R, and if you have a R you add a heart. So if you have a heart you'd add a R. R omega.

* *Participant referred to Examples 1 and 4*

* *Participant typed ℘♥Ω®Ω↔©Φ®Ω*

C: Good.

This is clearly where the participant figured out the inverse relation between ♥ and ®.

P: Okay. Next what do we do? Just lose everything now? And keep it all the same. Okay. Simple enough.

* *Participant referred to Example 4*

* *Participant typed ℘↔©Φ®Ω*

C: Excellent.

Problem 11 ®℘#Φ↔#Δ

P: Starts out with a R. Starts out with this. Okay, umm. Start out with a R at the beginning. But it has a number symbol there. So what does the number symbol mean? Number symbol means you write a C. Number symbol means you add a C. Number phi, add a C phi, delta, oops, delete, delete, arrow, C, delete, triangle, C phi.

And here the relation between # and ©.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

* *Participant referred to Examples 2, 6, and 8*
* *Participant typed ®℘#Φ©Φ↔#Δ©Φ*
C: Good.
P: Okay, knowing that, then you lose everything except for the R and that. And we keep them all the same. Keep the R, that the same, number triangle.
* *Participant referred to Example 6*
* *Participant typed ®℘↔#Δ©Φ*
C: Good.
P: And we got left, the R, keep everything the same. Sign elimination for ®.
* *Participant referred to Example 6*
* *Participant typed ℘↔#Δ©Φ*
C: Excellent.

Problem 12 ♥℘©Ω↔®Δ

P: Starts out with heart and has a C. Having a C, probably add a number symbol. Let's try it. Number symbol, R triangle.
* *Participant typed ♥℘©Ω#Ω↔®Δ#Ω*
C: Good.
P: Yup, I was right. You lose everything, except for the heart and that funny symbol. Arrow, and what does heart imply? Keep everything the same when you change it to a R. That was a R already.
* *Participant referred to Example 2*
* *Participant typed ♥℘↔®Δ#Ω*
C: Good.
P: Should be this. Triangle.
* *Participant typed ℘↔®Δ#Ω*
C: Try again.
P: Maybe exchange all Rs and hearts. Try that. Heart triangle. Sign elimination for ♥.
* *Participant referred to Example 2*
* *Participant typed ℘↔♥Δ#Ω*
C: Excellent.
P: Okay.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Problem 13 ℘©Γ↔©Ω

P: Umm, I got 3 and 2. It starts out with the symbol you solve for, and has a C. What does the C mean? C means you add a number symbol.

* *Participant typed ℘©Γ#Γ↔©Ω#Γ*

C: Good.

P: Okay, then you're supposed to lose everything and leave it like it is. Is that right? That's right. Don't change anything.

* *Participant referred to Examples 1, 4, and 7*

* *Participant typed ℘↔©Ω#Γ*

C: Excellent.

Problem 14 ®℘♥Ω↔®Γ

P: This one starts out with a R and a heart. A heart should be, okay. R heart, omega, R omega, arrow, R.

* *Participant typed ®℘♥Ω®Ω↔®Γ®Ω*

C: Good.

P: Hmm, and the R at the beginning. Shouldn't do anything yet.

* *Participant typed ®℘↔®Γ®Ω*

C: Good.

P: And this means replace all Rs with hearts or something like that. R, so replace all hearts with Rs. The R means delete. We'll try to just put it in the way it is.

Initially confused with ♥ elimination, but figured out the correct rule again.

* *Participant referred to Examples 2, 3, and 6*

* *Participant typed ℘↔®Γ®Ω*

C: Excellent.

P: Okay.

Problem 15 ♥℘↔#Δ#Ω

* *Participant typed ℘↔#Δ#Ω*

C: Excellent.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Problem 16 ℘®Φ↔♥Δ

P: Okay, umm. The R means to add a heart phi. And then, on both sides.

* *Participant typed ℘®Φ♥Φ↔♥Δ♥Φ*

C: Good.

P: Then we lose everything, and just keep that the way it is.

* *Participant typed ℘↔♥Δ♥Φ*

C: Excellent.

Problem 17 ©℘↔©Δ®Ω

P: The C means, what does the C mean? Lose everything and exchange. Delta with those.

* *Participant referred to Example 3*

* *Participant typed ℘↔©Γ®Ω*

He might have thought that inverses existed for the Greek letters as well.

C: Try again.

P: Didn't think so. C, see if just swapping them makes any sense. It probably won't, but.

* *Participant typed ℘↔©Ω®Δ*

C: Excellent.

P: Okay, it did.

Problem 18 #℘#Δ↔©Φ

P: Has a number symbol, and a number symbol, so I believe you add a C. C phi C Delta.

* *Participant typed #℘#Δ©Δ↔©Φ©Δ*

C: Good.

P: Then you lose everything on that side. And you should keep this side the same.

* *Participant typed #℘↔©Φ©Δ*

C: Good.

P: And the number symbol means. Swap, but they're the same. Maybe you switch it with a number symbol. I think you do.

Guesses at what the right rule for # elimination is.

* *Participant referred to Example 5*

* *Participant typed ℘↔#Φ#Δ*

C: Excellent.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Problem 19 ♥ ℘↔®Δ©Γ

P: Okay, umm. A heart means you, something to do with the R. Put all, for all hearts put in a R, but there are no hearts, so.
* *Participant referred to Example 2*
* *Participant typed ℘↔®Δ©Γ*
C: Try again.
P: That means put the heart in here.
* *Participant typed ℘↔♥Δ©Γ*
C: Excellent.

The participant apparently didn't generalize inverting ®s to ♥s from inverting ♥s to ®s.

Problem 20 ℘©Δ↔#Φ

P: Delta, the C means you add a number symbol, the last character, add a number symbol delta.
* *Participant typed ℘©Δ#Δ↔#Φ#Δ*
C: Good.
P: And you just keep it the same.
* *Participant typed ℘↔#Φ#Δ*
C: Excellent.

Problem 21 #℘#Δ↔©Γ

P: Number symbol, delta, so the number symbol means you add a C, delta, arrow, C, add a C delta.
* *Participant typed #℘#Δ©Δ↔©Γ©Δ*
C: Good.
P: You lose everything else. That side stays.
* *Participant typed #℘↔©Γ©Δ*
C: Good.
P: Number symbol, put number symbols in for Cs.
* *Participant typed ℘↔#Γ#Δ*
C: Excellent.
P: Yeah.

Problem 22 ©℘↔#Φ©Ω

P: Okay, umm. This should be C is swap, right? C, swap the, yeah. Same for the R, right. No. R you keep the same. Okay the C swaps. Umm, on the right side, that, then that. Oops.
* *Participant referred to Example 6*
* *Participant typed ℘↔#Ω©Φ*
C: Excellent.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Problem 23 ®℘®Γ↔®Ω

P: Okay, R symbol, R, this, second R means you add a heart.

* *Participant typed ®℘®Γ♥Γ↔®Ω♥Γ*

C: Good.

P: Then you lose everything to the right of that symbol. Umm, yeah, always keep this side the same.

* *Participant typed ®℘↔®Ω♥Γ*

C: Good.

P: And then, the R means you keep everything the same. It means you don't switch the R and the heart. No, it means you keep everything the same.

* *Participant typed ℘↔®Ω♥Γ*

C: Excellent.

P: Yeah.

Problem 24 ℘♥Φ↔®Δ

P: Okay, phi, the heart means you add a R phi.

* *Participant typed ℘♥Φ®Φ↔®Δ®Φ*

C: Good.

P: Then you lose everything, keep everything the same on the right.

* *Participant typed ℘↔®Δ®Φ*

C: Excellent.

Problem 25 ®℘↔©Γ©Δ

P: R means you keep everything the same.

* *Participant typed ℘↔©Γ©Δ*

C: Excellent.

Problem 26 ℘®Φ↔♥Γ

P: Okay, R phi. R means you add a heart phi.

* *Participant typed ℘®Φ♥Φ↔♥Γ♥Φ*

C: Good.

* *Participant typed ℘↔♥Γ♥Φ*

C: Excellent.

Problem 27 ©℘#Ω↔#Γ

P: Number symbol means you put a C there. Phi arrow, number symbol, C omega.

* *Participant typed ©℘#Ω©Ω↔#Γ©Ω*

C: Good.

P: Drop off that stuff.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

* *Participant typed ©℘↔#Γ©Ω*
C: Good.
P: And the C means you, the C means you swap. The C means you swap, yeah. The C means you swap the omega and that symbol.
* *Participant referred to Example 8*
* *Participant typed ℘↔#Ω©Γ*
C: Excellent.

Problem 28 #℘↔©Ω©Δ

P: A number symbol means you replace Cs with number symbols and vice versa.
* *Participant typed ℘↔#Ω#Δ*
C: Excellent.

Problem 29 ℘♥Φ↔©Γ

P: Phi, heart means you add a R. R phi.
* *Participant typed ℘♥Φ®Φ↔©Γ®Φ*
C: Good.
P: And then just put everything.
* *Participant typed ℘↔©Γ®Φ*
C: Excellent.

Problem 30 ♥℘©Γ↔♥Φ

P: Heart, that thing, the C means number symbol.
* *Participant typed ♥℘©Γ#Γ↔♥Φ#Γ*
C: Good.
P: Now, drop everything, keep everything the same.
* *Participant typed ♥℘↔♥Φ#Γ*
C: Good.
P: And the heart means you replace all hearts with R and vice versa.
* *Participant typed ℘↔®Φ#Γ*
C: Excellent.

Problem 31 ®℘↔♥Φ♥Δ

P: This means replace all hearts with a R.
* *Participant typed ℘↔®Φ®Δ*
C: Try again.
P: Oh, the R you keep the same. No, yeah. R keeps, R you keep the same. I forgot.
* *Participant typed ℘↔♥Φ♥Δ*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix F: Additional Information II

Information available to both the Syntax(Hint):

Syntax

The problem takes the form of a string of characters. The characters are selected from the following:

©, ®, #, ♥	Are the connector symbols
Δ, Γ, Ω, Φ	Are the object symbols
↔, ℘	Are special symbols

The ↔ character always comes first.

After the ↔, either 0, 1, 2, or 3 connector symbols will appear. Next comes the ℘, followed by either 0, 1, or 2 object symbols. Consider this part 1 of the string (if there are any object symbols after the ℘, they belong to this part).

Part two of the string consists of either one connector and then one object symbol, or two connectors and then two object symbols.

Goal

Your goal is to make the ℘ character the second symbol of the string.

A set of rules exist that dictates how you can change the current character string into a new character string.

Only one rule is applicable for any particular character string.

Hint

The ® and the ♥ symbols, as well as the © and the # symbols, are associated with one another.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.