Cognitive Models of Programming

Reading Reflection

- Did you notice that you gained richer schemas over the course of your own programming journey?
- What do you think of the critique of structure editors as being too close to the structure of the *language* instead of schemas?
- What key insight or insights (if any) stood out to you as being relevant to your future PL or programming tool design work?
    - To the problems you uncovered in your user Zoom call?

These slides draw heavily from Chapter 3 of *Software Design —Cognitive Aspects*, but I'm going to emphasize a particular subset

Reminder: no assignment this week!

Approaches

- **Knowledge-Centered**: It’s all about what syntactic knowledge, semantic knowledge, and schematic knowledge you’ve stored up

- **Strategy-Centered**: It’s all about the strategies you use for applying the knowledge types to build up programs

- **Organization-Centered**: It’s all about how the design process/design activity is organized. Do we start with a high-level plan, work down breadth-first until we have a program? Do we pursue an iterative design process, planning, drafting, and editing?

Knowledge-Centered Approaches

Programming Knowledge

Researchers in program design are generally agreed that there are three types of knowledge that serve to distinguish experts from novices:

1. **Syntactic knowledge**, which defines the syntactic and lexical elements of a programming language, for example, the fact that, in C, the if statement takes the form if (condition) statement.
2. **Semantic knowledge**, which refers to the concepts, such as the notion of a variable, that make it possible to understand what happens when a line of code is executed.
3. **Schematic knowledge**, that is, programming schemas that represent generic solutions.

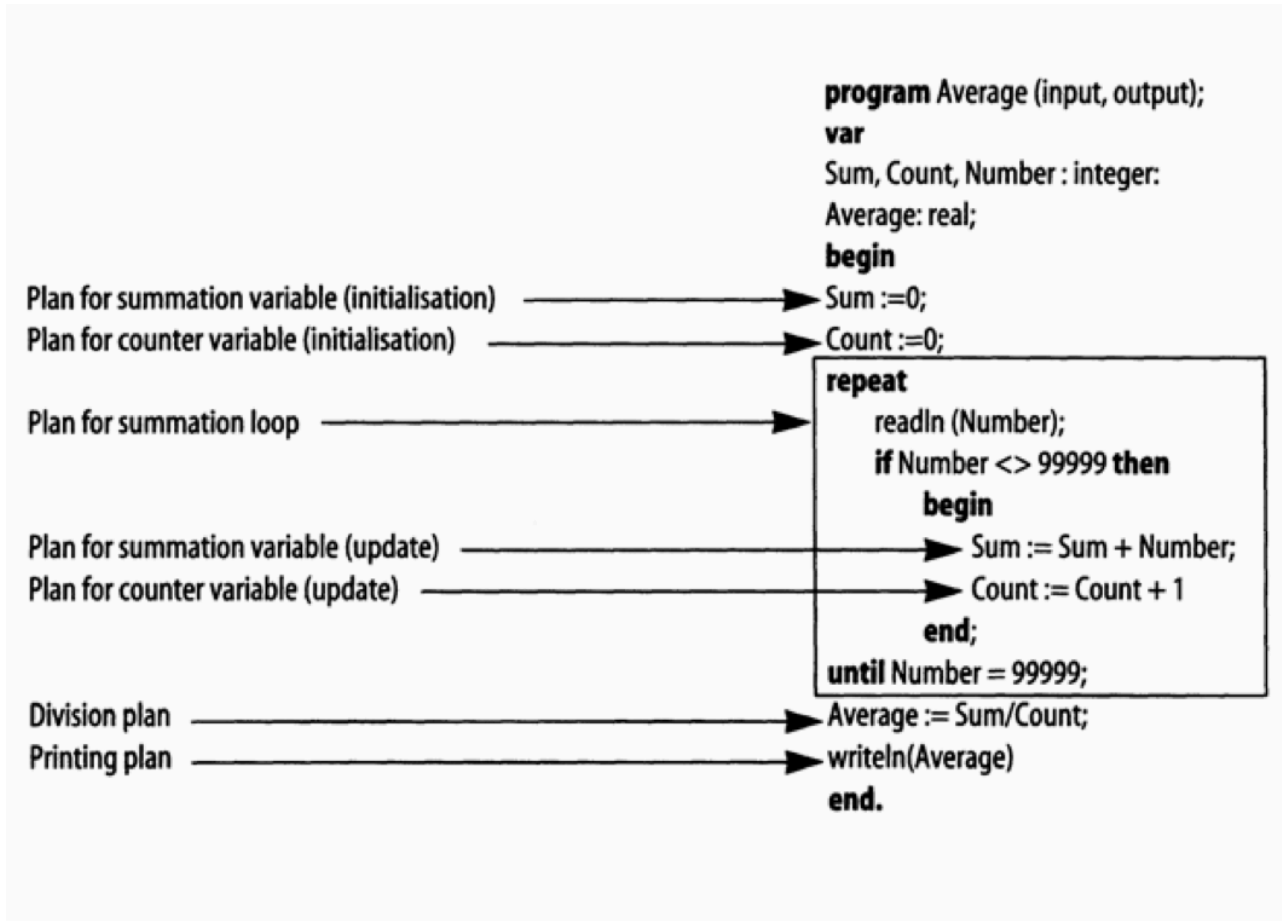
People reinvent the ideas from this slide over and over again

Elementary through Complex

Elementary programming schemas represent knowledge about control structures and variables. Think of a frame with slots. See fig. For example, a counter variable schema can be formalized as following:

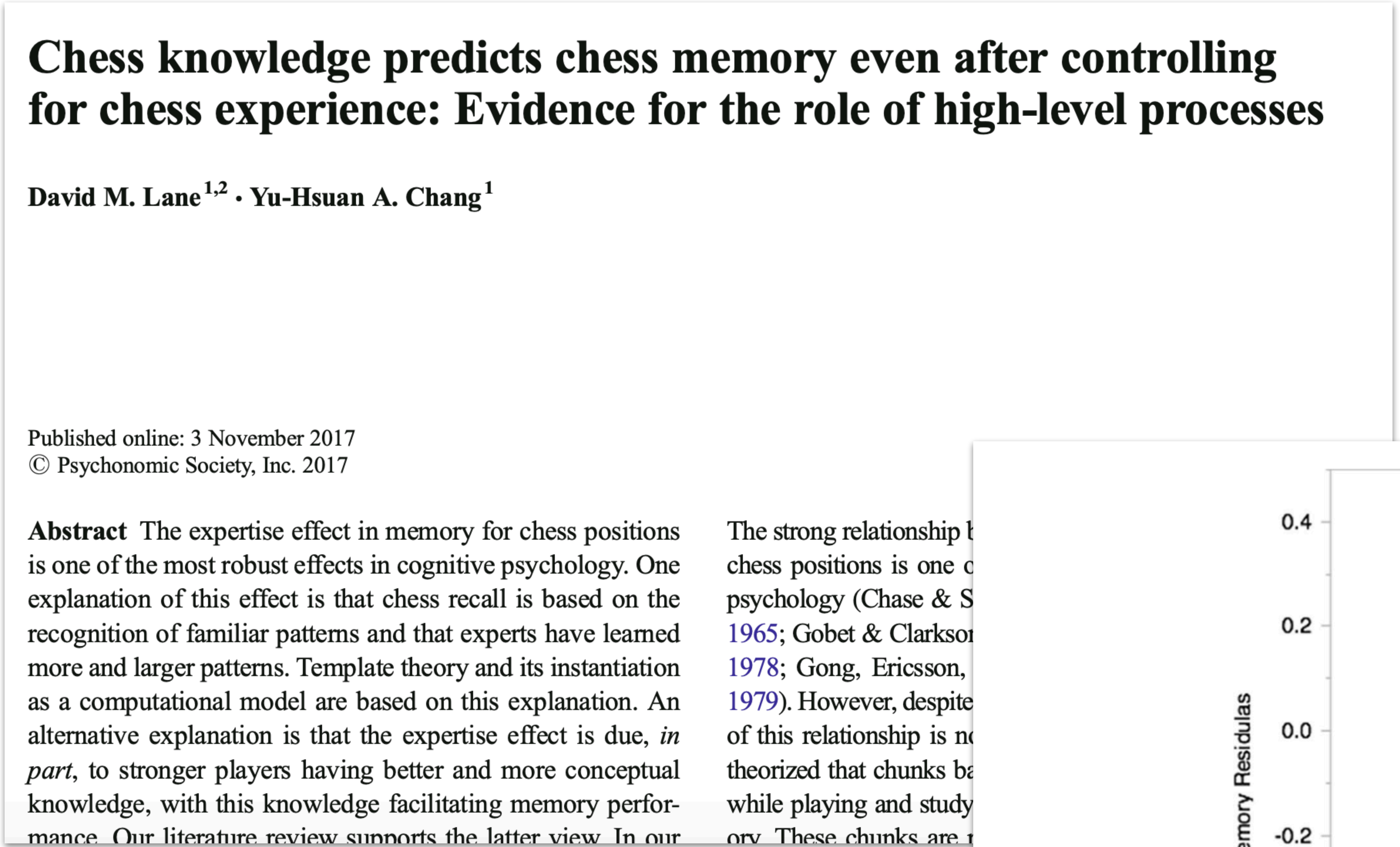
- Goal: count the occurrences of an action
- Initialization: count:= n
- Update: count:=count+increment
- Type: integer
- Context: loop

Algorithmic schemas or **complex programming schemas** represent knowledge about structure of algorithms. For example, some programmers will be familiar with a variety of algorithms for sorting and searching. These algorithms are more or less abstract and more or less independent of the programming language, and they can be described as made up of elementary schemas. For example, a sequential search schema is less abstract than a search schema and can be described as being composed, in part, of a counter variable schema.



Plan for summation variable (initialisation)
Plan for counter variable (initialisation)
Plan for summation loop
Plan for summation variable (update)
Plan for counter variable (update)
Division plan
Printing plan

```
program Average (input, output);
var
Sum, Count, Number : integer:
Average: real;
begin
Sum :=0;
Count :=0;
repeat
    readln (Number);
    if Number <> 99999 then
        begin
            Sum := Sum + Number;
            Count := Count + 1
        end;
until Number = 99999;
Average := Sum/Count;
writeln(Average)
end.
```



Chess knowledge predicts chess memory even after controlling for chess experience: Evidence for the role of high-level processes

David M. Lane[1,2] • Yu-Hsuan A. Chang[1]

Published online: 3 November 2017

Abstract The expertise effect in memory for chess positions is one of the most robust effects in cognitive psychology. One explanation of this effect is that chess recall is based on the recognition of familiar patterns and that experts have learned more and larger patterns. Template theory and its instantiation as a computational model are based on this explanation. An alternative explanation is that the expertise effect is due, *in part*, to stronger players having better and more conceptual knowledge, with this knowledge facilitating memory perfor-mance. Our literature review supports the latter view. In our

The strong relationship b
chess positions is one o
psychology (Chase & S
1965; Gobet & Clarkso
1978; Gong, Ericsson,
1979). However, despite
of this relationship is no
theorized that chunks ba
while playing and study
ory. These chunks are r

There's a whole history of work showing chess masters can memorize boards really well…unless it's a board you couldn't reach from real play.

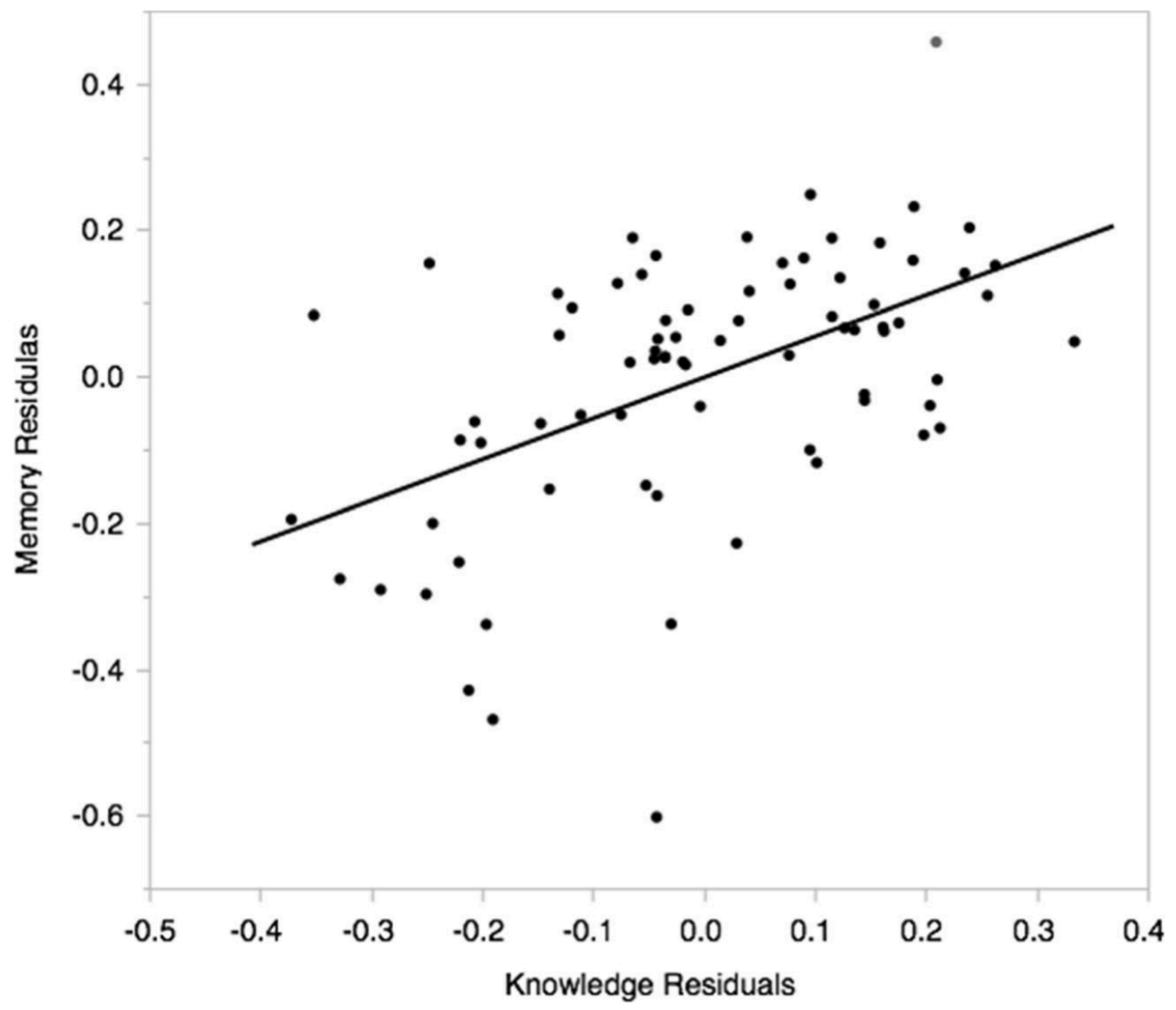
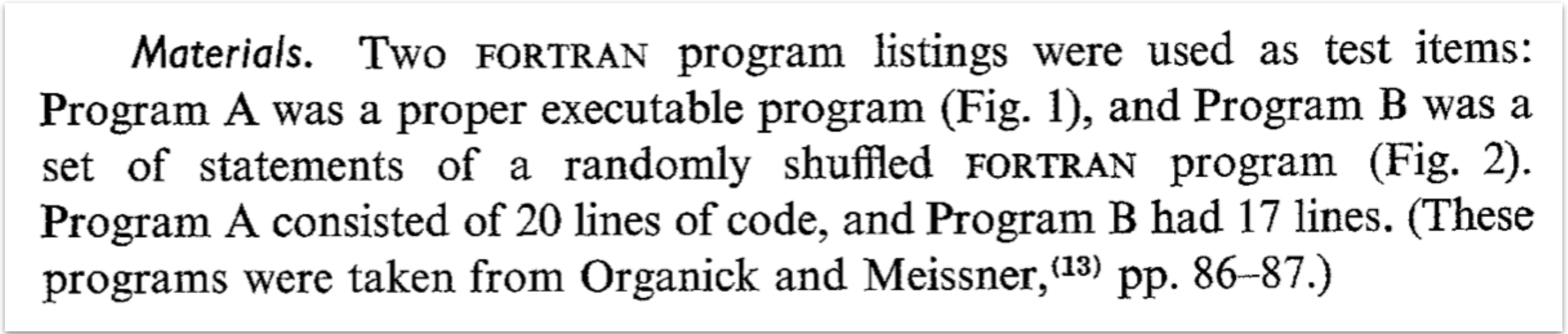


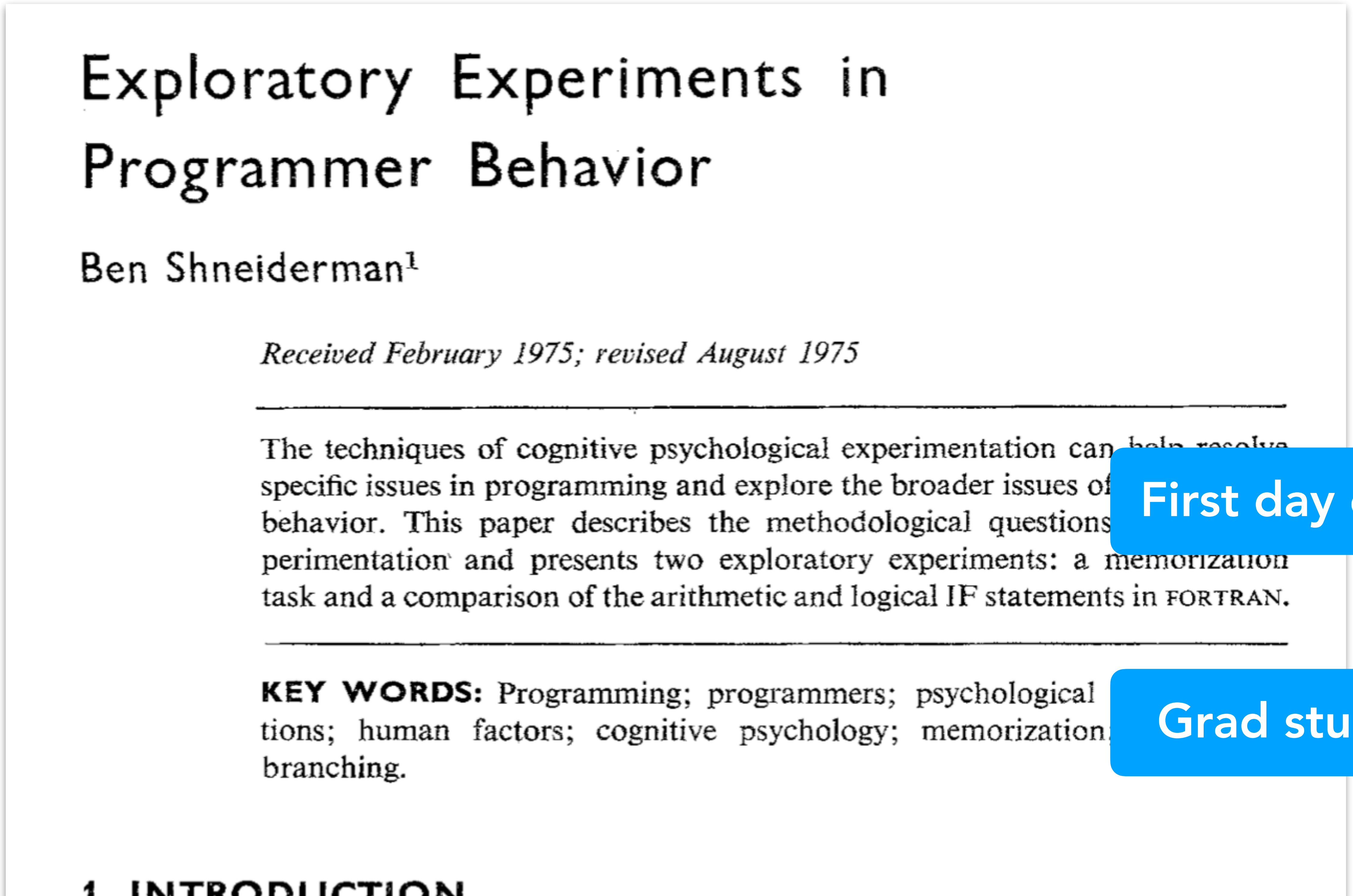
Fig. 2 Partial regression plot showing the part of memory independent of the three experience variables, as a function of the part of knowledge independent of these same variables.

This is chart is showing the data *after controlling for experience*!

Ok, back to programming…



Materials. Two FORTRAN program listings were used as test items: Program A was a proper executable program (Fig. 1), and Program B was a set of statements of a randomly shuffled FORTRAN program (Fig. 2). Program A consisted of 20 lines of code, and Program B had 17 lines. (These programs were taken from Organick and Meissner,[13] pp. 86–87.)



Exploratory Experiments in Programmer Behavior

Ben Shneiderman[1]

Received February 1975; revised August 1975

The techniques of cognitive psychological experimentation can help resolve specific issues in programming and explore the broader issues of behavior. This paper describes the methodological questions perimentation and presents two exploratory experiments: a memorization task and a comparison of the arithmetic and logical IF statements in FORTRAN.

KEY WORDS: Programming; programmers; psychological tions; human factors; cognitive psychology; memorization branching.

First day of FORTRAN class

Grad students and faculty

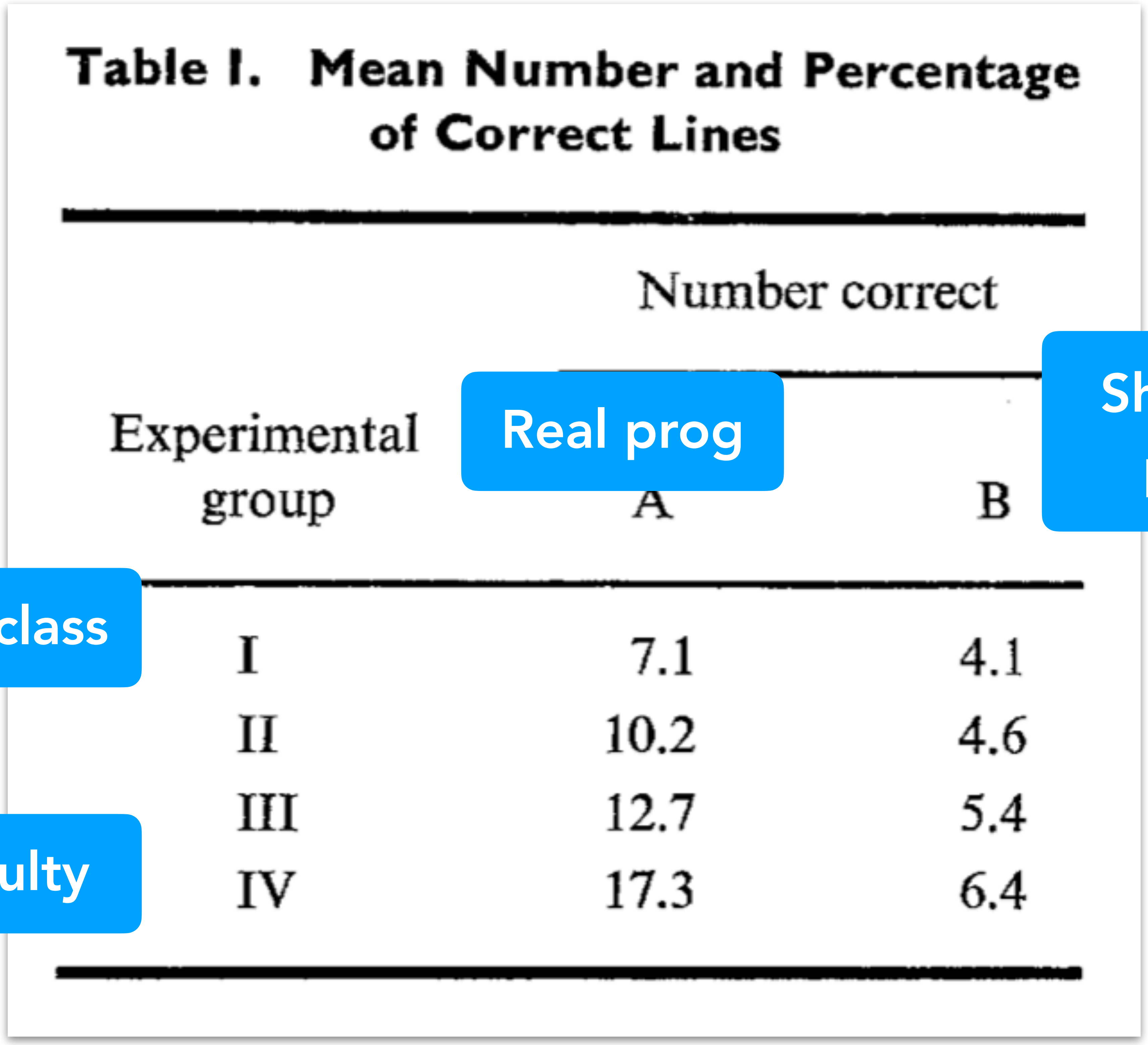
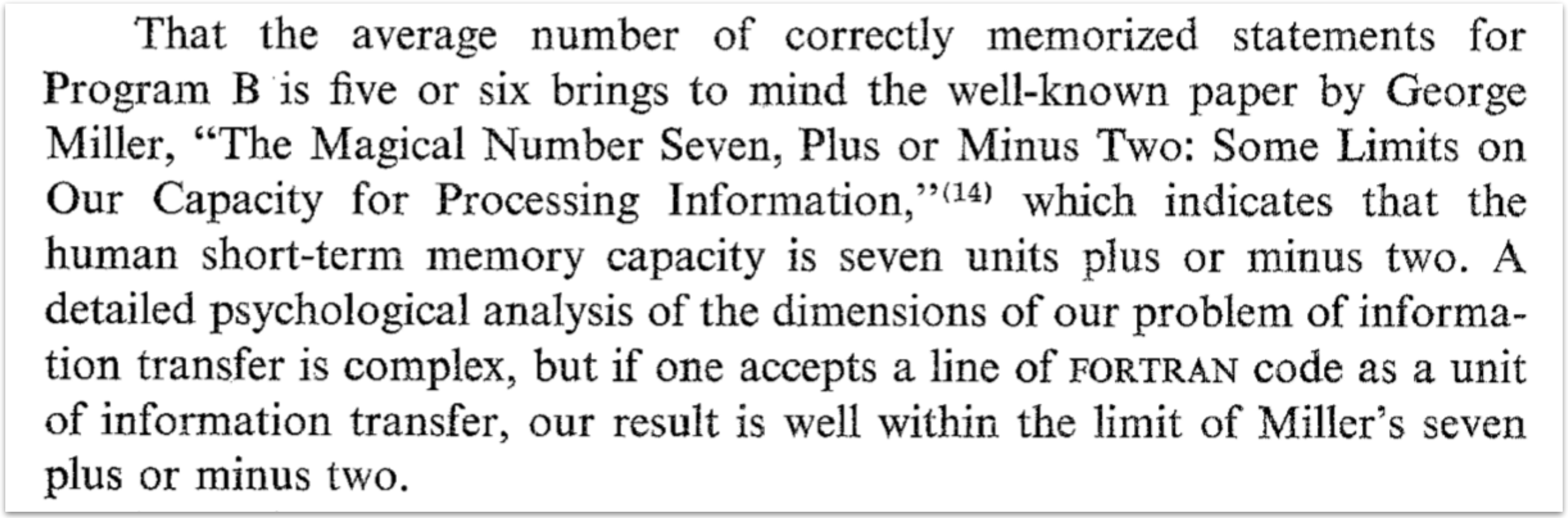


Table I. Mean Number and Percentage of Correct Lines

Experimental group	Number correct A	B
I	7.1	4.1
II	10.2	4.6
III	12.7	5.4
IV	17.3	6.4

Real prog

Shuffled prog

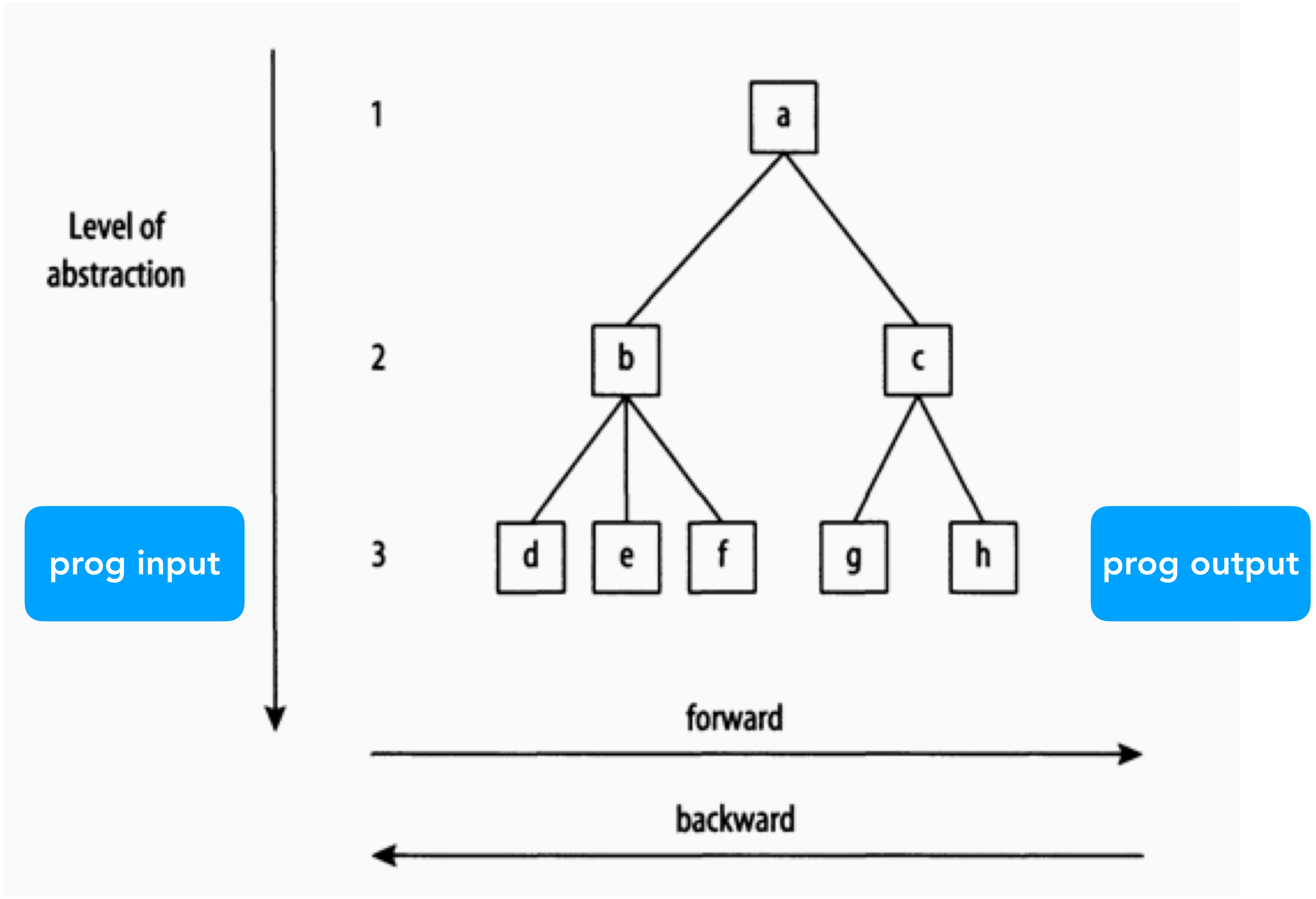


That the average number of correctly memorized statements for Program B is five or six brings to mind the well-known paper by George Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information,"(14) which indicates that the human short-term memory capacity is seven units plus or minus two. A detailed psychological analysis of the dimensions of our problem of information transfer is complex, but if one accepts a line of FORTRAN code as a unit of information transfer, our result is well within the limit of Miller's seven plus or minus two.

Strategy-Centered Approaches

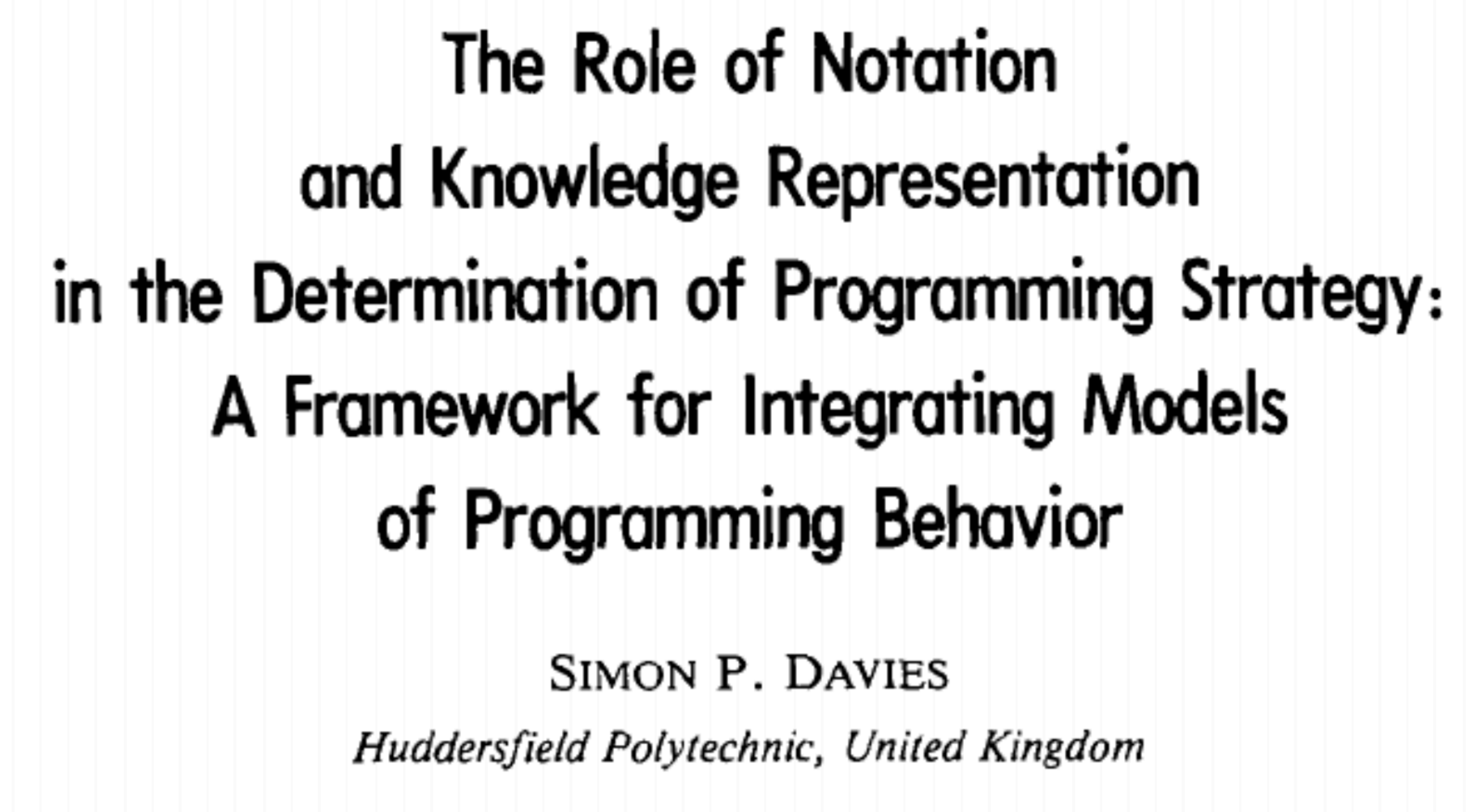
Axes

- Top-down vs. Bottom-up
- Forward vs. Backward
- Breadth-First vs. Depth-First



CDN → Strategy Changes

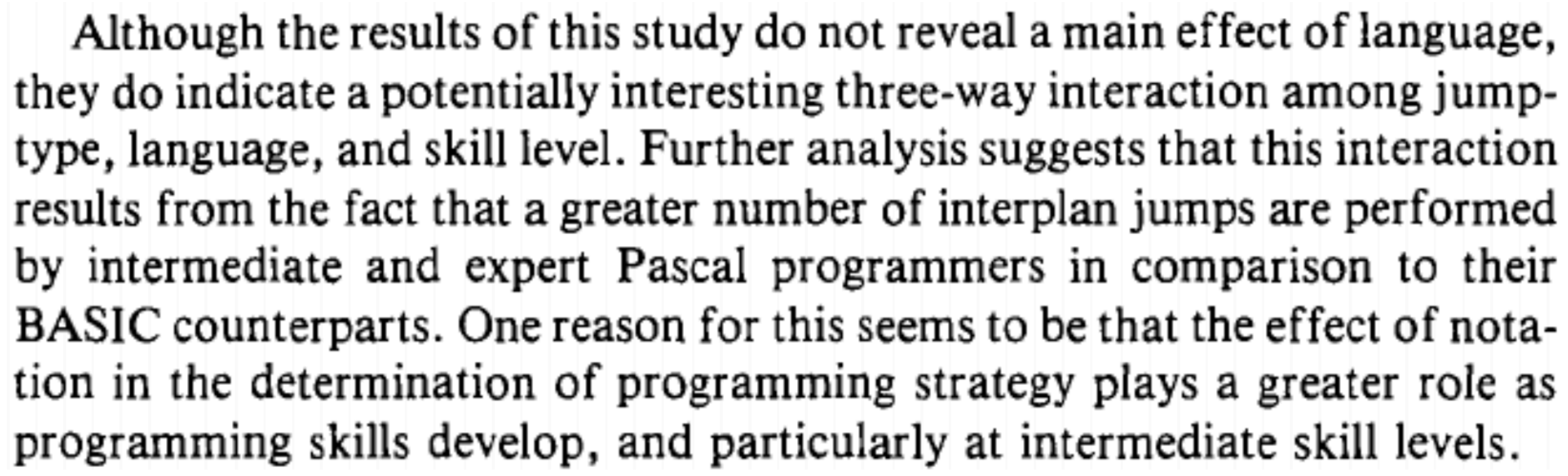
- Cognitive dimensions of notation proven to affect which strategies programmers apply



The Role of Notation and Knowledge Representation in the Determination of Programming Strategy: A Framework for Integrating Models of Programming Behavior

SIMON P. DAVIES

Huddersfield Polytechnic, United Kingdom



Although the results of this study do not reveal a main effect of language, they do indicate a potentially interesting three-way interaction among jump-type, language, and skill level. Further analysis suggests that this interaction results from the fact that a greater number of interplan jumps are performed by intermediate and expert Pascal programmers in comparison to their BASIC counterparts. One reason for this seems to be that the effect of notation in the determination of programming strategy plays a greater role as programming skills develop, and particularly at intermediate skill levels.

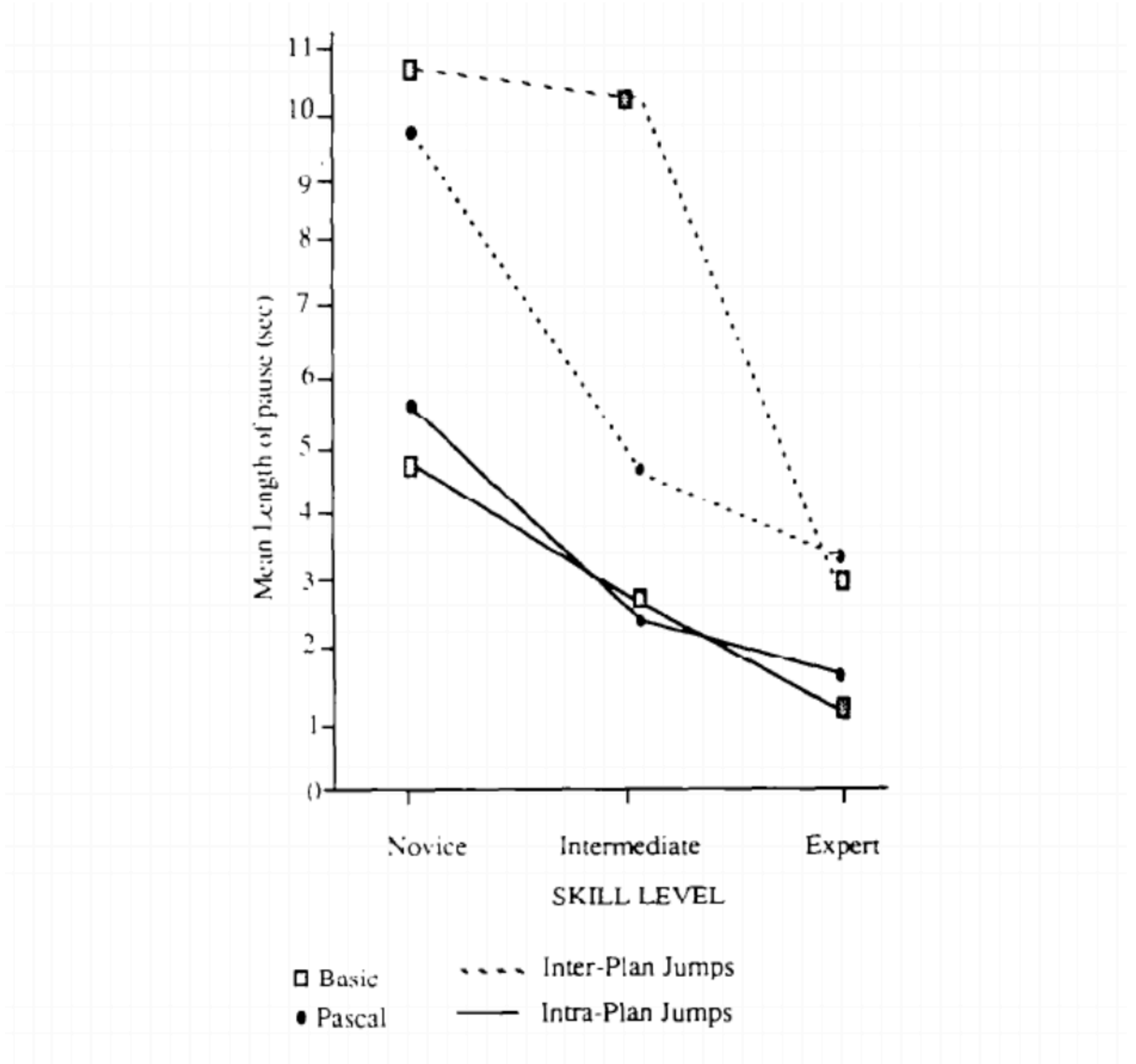


Figure 5. Mean length of pause (sec) between inter- and intra-plan jumps for different languages during program generation.

Organization-Centered Approaches

Organization-Centered

- This is the strand that's most concerned with observing how people actually organize their work
- Also the strand that recognizes the iterative nature of so much programming
  - Plan
  - Code
  - Revise

Organization-Centered: Programming + Memory

- See this body of literature and especially work by Simon P. Davies for work on the effects of working memory on programming

Organization-Centered: Programming + Text

- See this body of literature and especially work by Rachel K. E. Bellamy and Simon P. Davies for more on how programmers co-design code and supporting natural language
  - Also see Bellamy's related work on pseudocode
    - "Four categories of pseudo-code emerged from the data: diagrams, semiformal annotations, coding on paper, and text… Results suggest that programmers use pseudo-code and pen and paper to reduce the cognitive complexity of the programming task."
      - *What does pseudo-code do? A psychological analysis of the use of pseudo-code by experienced programmers.* By Rachel K. E. Bellamy

Novices vs. Experts

Novices vs. Experts

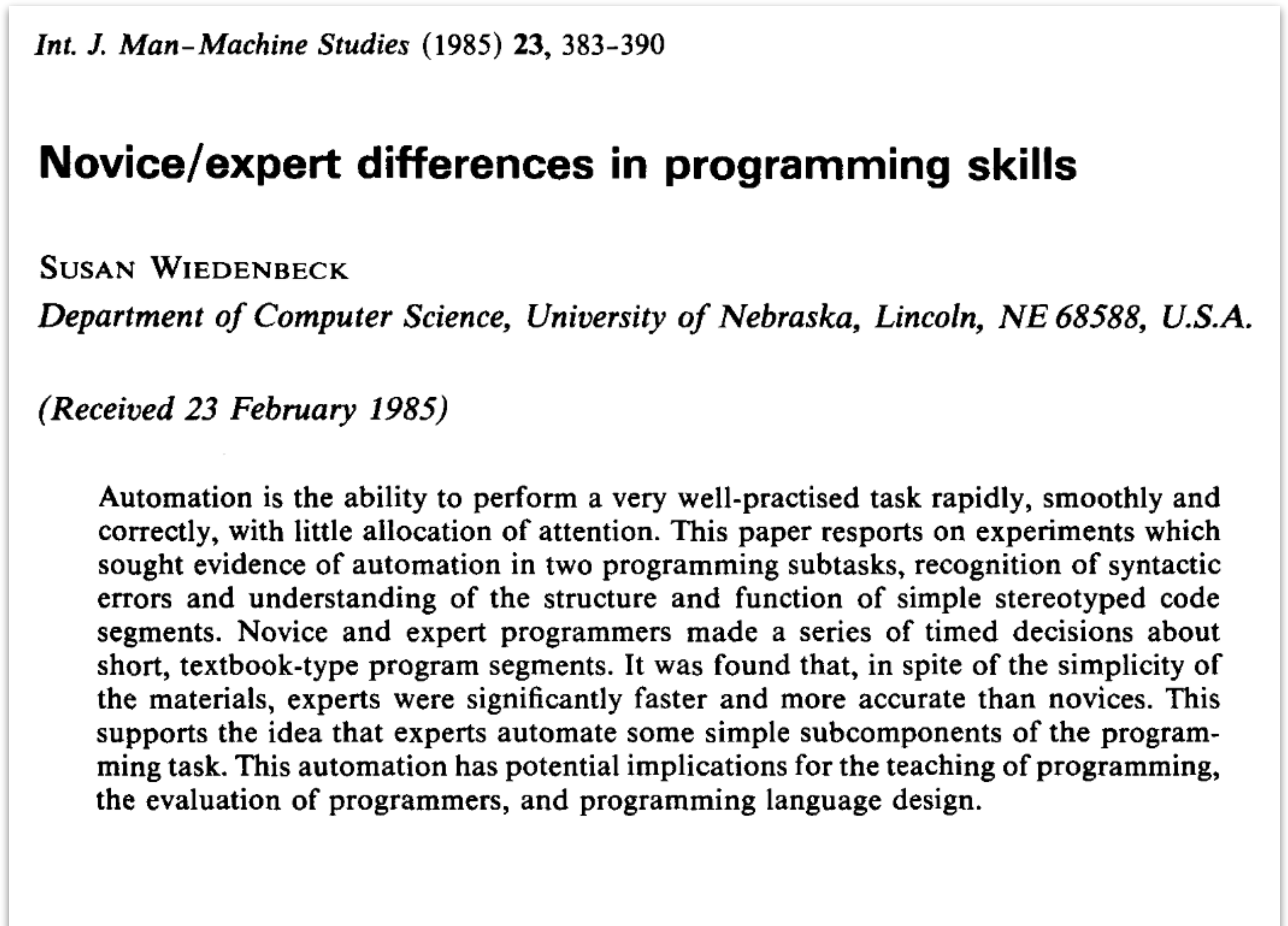
Compared with novices, experts:

- construct a more complete problem representation before embarking on the process of solving it
- use more rules of discourse
- use more meta-cognitive knowledge about programming tasks and about suitable and optimal strategies for completing them; know a number of possible strategies for completing a task and are able to compare them to select a good approach
- are capable of generating several alternative solutions before making a choice
- use more external devices, particularly as external memory; their design strategy is top-down and forward for familiar and not too complex problems, while novices go bottom-up and backwards
- do some aspects of programming tasks “automatically”

International Journal of Man-Machine Studies

Volume 23, Issue 4, October 1985, Pages 383-390

Susan Wiedenbeck's work is especially useful here



Int. J. Man–Machine Studies (1985) **23**, 383–390

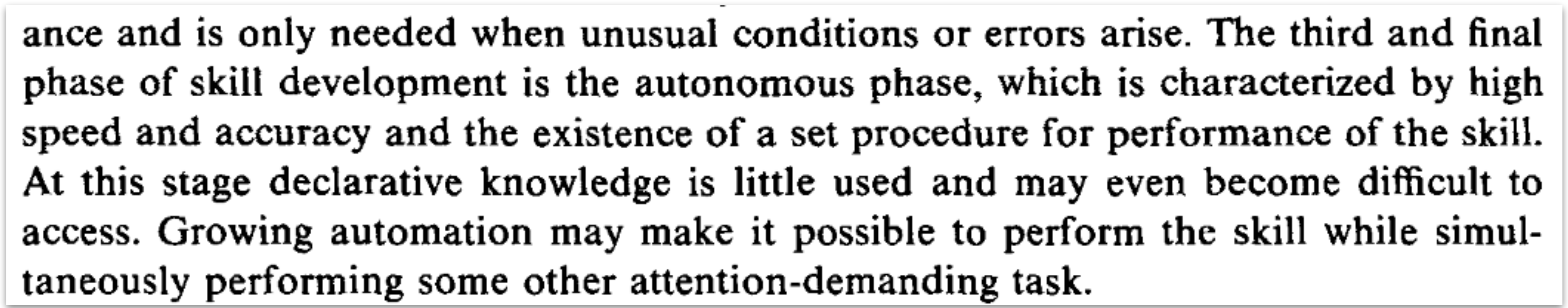
Novice/expert differences in programming skills

SUSAN WIEDENBECK

Department of Computer Science, University of Nebraska, Lincoln, NE 68588, U.S.A.

(Received 23 February 1985)

Automation is the ability to perform a very well-practised task rapidly, smoothly and correctly, with little allocation of attention. This paper resports on experiments which sought evidence of automation in two programming subtasks, recognition of syntactic errors and understanding of the structure and function of simple stereotyped code segments. Novice and expert programmers made a series of timed decisions about short, textbook-type program segments. It was found that, in spite of the simplicity of the materials, experts were significantly faster and more accurate than novices. This supports the idea that experts automate some simple subcomponents of the programming task. This automation has potential implications for the teaching of programming, the evaluation of programmers, and programming language design.



ance and is only needed when unusual conditions or errors arise. The third and final phase of skill development is the autonomous phase, which is characterized by high speed and accuracy and the existence of a set procedure for performance of the skill. At this stage declarative knowledge is little used and may even become difficult to access. Growing automation may make it possible to perform the skill while simultaneously performing some other attention-demanding task.

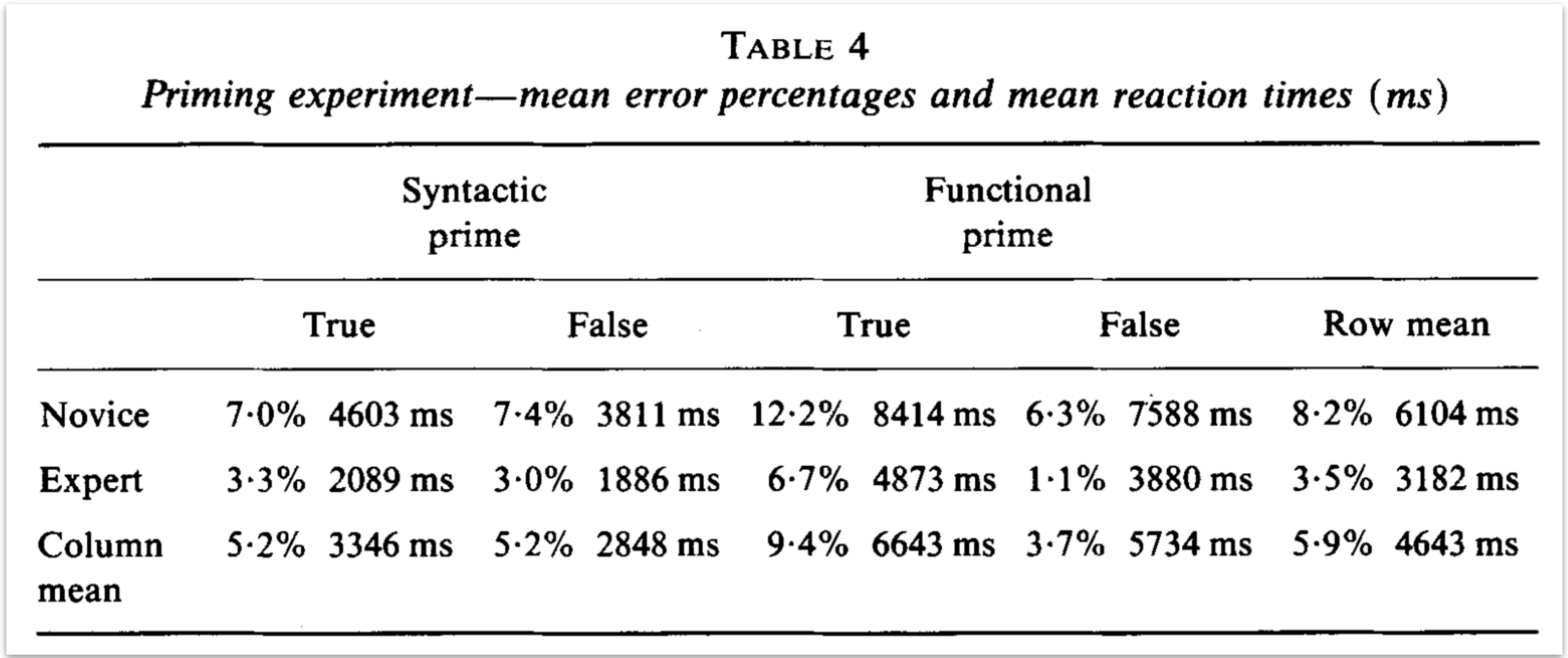
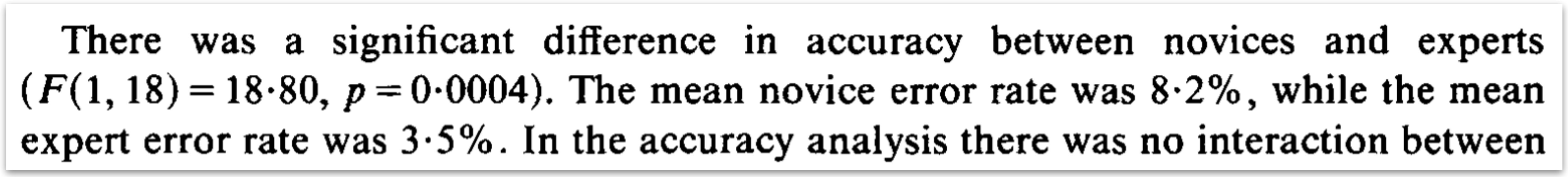


TABLE 4
Priming experiment—mean error percentages and mean reaction times (ms)

	Syntactic prime				Functional prime					
	True		False		True		False		Row mean	
Novice	7·0%	4603 ms	7·4%	3811 ms	12·2%	8414 ms	6·3%	7588 ms	8·2%	6104 ms
Expert	3·3%	2089 ms	3·0%	1886 ms	6·7%	4873 ms	1·1%	3880 ms	3·5%	3182 ms
Column mean	5·2%	3346 ms	5·2%	2848 ms	9·4%	6643 ms	3·7%	5734 ms	5·9%	4643 ms



There was a significant difference in accuracy between novices and experts ($F(1, 18) = 18{\cdot}80$, $p = 0{\cdot}0004$). The mean novice error rate was 8·2%, while the mean expert error rate was 3·5%. In the accuracy analysis there was no interaction between

Why??

- Other than the fact that the findings of individual papers in this space are super fascinating, why are we taking the time to cover this?
  - So you know the key terms when you need to find these papers to answer design questions of your own (without running a study!)
  - So when you find these same patterns or related patterns in your own studies, you know what line of work you're continuing or extending
    - So when you write related works sections, you don't miss key background…
    - …and don't reinvent the wheel! :)
  - But above all, because this line of research offers a glimpse of how much we can learn about programmers' internal state from well-designed experiments!
    - Read Thursday's paper with this framing in mind
    - With the right stimuli, we can start inferring really low-level details of programmers' mental models

Activity!

- Section 3.6 of the reading for today suggests ways we could make programming tools more suited to programmers' real needs.
- With your group, review Section 3.6 and brainstorm an intervention that draws on these recommendations. Feel free to draw from other parts of the reading if the 3.6 ideas don't inspire you. Your intervention could be:
  - A new PL, programming environment, or programming tool
  - Modifications to an existing PL, programming environment, or programming tool
- Write up three slides on your intervention (see template in the linked slideshow)
  - At least one slide should be devoted to the concepts or passages from the reading that support your design
  - Add your slides here: https://docs.google.com/presentation/d/1bvwotTndW3kVU8yNGfrxnqs39oy5AXU7uLJArRZFU8A/edit?usp=sharing
  - Choose someone to present your slides when we come back together as a group