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 need-finding homework?
  • To the problems you’ve uncovered in final project part 1 so far?
Reminder: assignment this week is to prepare for short presentations that you’ll give next week!

See assignment doc on the website!
Reward for having to make a short talk—no other readings next week! Just your chosen paper!
These slides draw heavily from Chapter 3 of *Software Design—Cognitive Aspects*, but I’m going to emphasize a particular subset.
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
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.
Chess knowledge predicts chess memory even after controlling for chess experience: Evidence for the role of high-level processes

David M. Lane1, 2 · Yu-Hsuan A. Chang1

Published online: 3 November 2017
© Psychonomic Society, Inc. 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 performance. Our literature review supports the latter view. In our

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.
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.)

### Table 1. Mean Number and Percentage of Correct Lines

<table>
<thead>
<tr>
<th>Experimental group</th>
<th>Number correct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>I</td>
<td>7.1</td>
</tr>
<tr>
<td>II</td>
<td>10.2</td>
</tr>
<tr>
<td>III</td>
<td>12.7</td>
</tr>
<tr>
<td>IV</td>
<td>17.3</td>
</tr>
</tbody>
</table>

**First day of FORTRAN class**

**Grad students and faculty**

**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 4. Mean number of inter- and intra-plan jumps performed by programmers of different skill levels in Pascal and BASIC during program generation.
• 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.
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.”

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”
Susan Wiedenbeck’s work is especially useful here.

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 reports 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.

Table 4

<table>
<thead>
<tr>
<th></th>
<th>Syntactic prime</th>
<th></th>
<th>Functional prime</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Novice</td>
<td>7-0% 4603 ms</td>
<td>7-4% 3811 ms</td>
<td>12-2% 8414 ms</td>
<td>6-3% 7588 ms</td>
</tr>
<tr>
<td>Expert</td>
<td>3-3% 2089 ms</td>
<td>3-0% 1886 ms</td>
<td>6-7% 4873 ms</td>
<td>1-1% 3880 ms</td>
</tr>
<tr>
<td>Column mean</td>
<td>5-2% 3346 ms</td>
<td>5-2% 2848 ms</td>
<td>9-4% 6643 ms</td>
<td>3-7% 5734 ms</td>
</tr>
</tbody>
</table>

There was a significant difference in accuracy between novices and experts ($F(1, 18) = 18-80, p = 0-0004$). 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 to the presentation in the slack channel!
• Choose someone to present your slides when we come back together as a group