Mr. Bits: A Quantitative Process Model of Simple Program Understanding & Nibbles: A Constraint-Based Model of Program Reading and Inference

Michael Hansen (mihansen@indiana.edu**)**
School of Informatics and Computing, 2719 E. 10th Street
Bloomington, IN 47408 USA

Andrew Lumsdaine (lums@indiana.edu**)**
School of Informatics and Computing, 2719 E. 10th Street
Bloomington, IN 47408 USA

Robert L. Goldstone (rgoldsto@indiana.edu**)**
Dept. of Psychological and Brain Sciences, 1101 E. 10th Street
Bloomington, IN 47405 USA

May 3, 2015

Abstract

1 Introduction

1.1 The Complexity of a Program

What makes some code hard to understand? Intuitively, we might expect the difficulty of the underlying problem the program is attempting to solve to be a major factor. A program designed to simulate the physics of an automobile is likely more complex than one that simply computes the average of a set of numbers. The latter program is also bound to be **shorter** and use **fewer operations** than the former [39]. However, short and simple-looking programs can be hard to understand, even for **experienced** programmers, when certain notational and conceptual **expectations** are violated [34]. Therefore, the **cognitive complexity** of a program – e.g., how hard it is to understand – is determined by computational, notational, and psychological factors [6]. To fully understand and predict this kind of complexity, we must create **cognitive models** of programmers that are capable of reasoning about source code and representing programs in a human-like manner. These models could be used to locate unmaintainable code in large codebases, inform design decisions for programming languages, and aid in the automated generation of programs for humans. Additionally, quantitatively modeling such a complex, real world task pushes the frontiers of Cognitive Science by combining existing models of representation, working memory, planning, and problem solving.

In this paper, we present two cognitive models, called **Mr. Bits** and **Nibbles**. Mr. Bits is designed to predict the eye movements and keystroke response times of a programmer who is tasked with reading short Python programs and guessing their printed output. The model is built on top of the **ACT-R cognitive architecture**, a computational simulation of the major components of human cognition and perception such as declarative/procedural knowledge, vision, and hearing. Additionally, it is intended to operationalize aspects of the Cognitive Complexity Metric [6] – a source code metric designed to measure how difficult a program is to *trace* through and mentally *chunk*. A major limitation of the Mr. Bits model, however, is that it *cannot make errors*. Therefore, we explore an alternative formalism to ACT-R, and present a second model called Nibbles. Unlike Mr. Bits, Nibbles generates multiple possible interpretations of a program's code, line-by-line, and selects a single (possibly incorrect) interpretation to incorporate into its running "mental model". We evaluate Mr. Bits by comparing its timing performance to 29 human programmers, each tested on 10 out of a set of 25 short Python programs. For Nibbles' evaluation, we focus on 3 of these programs, and show how the errors observed in our human programmers can be generated by Nibbles.

1.2 The Psychology of Programming

Psychologists have been studying programmers for at least forty years. Early research focused on correlations between task performance and human/language factors, such as how the presence of code comments impacts scores on a program comprehension questionnaire. More recent research has revolved around the cognitive processes underlying program comprehension. Effects of expertise, task, and available tools on program understanding have been found [9]. Studies with experienced programmers have revealed conventions, or "rules of discourse," that can have a profound impact (sometimes negative) on expert program comprehension [34].

The **qualitative** side of program comprehension modeling today is filled with "box and arrow" diagrams [9]. These models incorporate research from many aspects of cognitive science: psychology, linguistics,

computer science, and even neuroscience [24]. They are helpful for making sense of the myriad of results from controlled studies of programmers over the last four decades. Qualitative cognitive models can help emphasize which aspects of cognition are most important for program comprehension, such as language and problem solving skills [32]. Unfortunately, their predictive power is limited. The Stores Model of Code Cognition, for example, emphasizes the importance of the central executive and its relationship to strategic, semantic, and plan knowledge in code problem solving [10]. But, as with source code metrics, predicting whether or not a particular programmer will successfully comprehend a specific program is not possible with the model. A *quantitative* model is needed which formalizes aspects of existing *qualitative* models to emulate the process of reading, interpreting, and understanding a program. We develop and evaluate quantitative cognitive models of program comprehension built using two different formalisms: the **ACT-R cognitive architecture** [2], and **Cognitive Domain Ontologies** [12].

1.3 Why Model a Programmer?

The design, creation and interpretation of computer programs are some of the most cognitively challenging tasks that humans perform. Understanding the factors that impact the cognitive complexity of code is important for both applied and theoretical reasoning. Practically, an enormous amount of time is spent developing programs, and even more time is spent debugging them, and so if we can identify factors that expedite these activities, a large amount of time and money can be saved. Theoretically, programming is an excellent task for studying representation, working memory, planning, and problem solving in the real world.

- Inform language and API design
- Automated program generation
- Expand cognitive modeling into more complex tasks

Our present research focuses on programs much less complicated than those the average professional programmer typically encounters on a daily basis. The demands of our task are still high, however, because participants must predict precise program output. In this way, it is similar to debugging a short snippet of a larger program. Code studies often take the form of a code review, where programmers must locate errors or answer comprehension questions after the fact (e.g., does the program define a Professor class? [4]). Our task differs by asking programmers to mentally simulate code without necessarily understanding its purpose. In most programs, we intentionally use meaningless identifier names where appropriate (variables `a`, `b`, etc.) to avoid influencing the programmer's mental model.

Similar research has asked beginning (CS1) programming students to read and write code with simple goals, such as the Rainfall Problem [16]. To solve it, students must write a program that averages a list of numbers (rainfall amounts), where the list is terminated with a specific value – e.g., a negative number or 999999. CS1 students perform poorly on the Rainfall Problem across institutions around the world, inspiring researchers to seek better teaching methods. Our work includes many Python novices with a year or less of experience, so our results may contribute to ongoing research in early programming education.

- Develop a realizable theory of program understanding (evaluate formalisms)
- Quantitatively model output prediction task

- Predict task performance (timings, error rates, etc.) and kinds of errors

2 Background

Psychologists have been studying the behavioral aspects of programming for at least forty years [7]. In her book *Software Design - Cognitive Aspects*, Françoise Détienne proposed that psychological research on programming can be broken into two distinct periods [9]. The first period, spanning the 1960's and 1970's, is characterized by the importing of basic experimental techniques and theories from psychology into computer science. Early experiments looked for correlations between task performance and language/human factors – e.g., the presence or absence of language features, years of experience, and answers to code comprehension questionnaires. While this period marked the beginning of scientific inquiry into software usability from a programming perspective, results were limited in scope and often contradictory.

The problem is simple: if task performance depends heavily on internal cognitive processes, then it cannot be measured independent of the programmer. Early psychology of programming studies relied exclusively on statistical correlations between metrics like "presence of comments" and "number of defects detected," so researchers were unable to explain some puzzling results. For example, multiple studies in the 1970's sought to measure the effect of meaningful variable names on code understandability. Two studies found no effect [38, 31] while a third study found positive effects as programs became more complex [30]. Almost a decade later, Soloway and Ehrlich provided an explanation for these findings: experienced programmers are able to recognize code *schemas* or *programming plans* [34]. Programming plans are "program fragments that represent stereotypic action sequences in programming," and expert programmers can use them to infer intent in lieu of meaningful variable names. This and many other effects depend on internal cognitive processes, and therefore require a cognitive modeling approach to explain.

2.1 The eyeCode Experiment

Brief description of eyeCode

2.2 The Cognitive Complexity Metric

Developed in the mid-nineties, Cant et al.'s cognitive complexity metric (CCM) attempts to quantify the cognitive processes involved in program development, modification, and debugging [6]. The CCM focuses on the processes of *chunking* (understanding a block of code) and *tracing* (locating dependencies). Cant et al. provide mathematical definitions for factors that are believed to influence each process. Some of the factors in the CCM are quantified by drawing upon the existing literature, but many definitions are simply placeholders for future empirical studies.

2.2.1 Chunking and Tracing

The cognitive processes of chunking and tracing play key roles in the cognitive complexity metric (CCM). *Chunking* is defined as the process of recognizing groups of code statements (not necessarily sequential), and recording the information extracted from them as a single mental symbol or abstraction. In practice, programmers rarely read through and chunk every statement in a program. Instead, they *trace* forwards or backwards in order to find relevant chunks for the task at hand [6]. Cant et al. define a chunk as a block of

statements that must occur together (e.g., loop + conditional).[1] This definition, however, is intended only for when the programmer is reading code in a normal forward manner. When tracing backwards or forwards, a chunk is defined as the single statement involving a procedure or variable's definition.

2.2.2 Chunk Complexity (C)

To compute the complexity C_i of chunk i, Cant et al. define the following equation:

$$C_i = R_i + \sum_{j \in N} C_j + \sum_{j \in N} T_j$$

where R_i is the complexity of the immediate chunk i, C_j is the complexity of sub-chunk j, and T_j is the difficulty in tracing dependency j of chunk i. The definitions of R and T are given as follows:

$$R = R_F(R_S + R_C + R_E + R_R + R_V + R_D)$$

$$T = T_F(T_L + T_A + T_S + T_C)$$

Each right-hand side term stands for a particular factor that is thought to influence the chunking or tracing processes (Figure 1). The underlying equations for each factor can be found in [6], but are not needed for the discussion below.

2.2.3 Immediate Chunk Complexity (R)

The chunk complexity R is made up of six additive terms (R_S, R_C, R_E, R_R, R_V, R_D) and one multiplicative term R_F. These represent factors that influence how hard it is to understand a given chunk.[2]

R_F (chunk familiarity) This term captures the increased speed with which a programmer is able to understand a given chunk after repeated readings. In ACT-R, the subsymbolic layer of the declarative memory module would handle this, as repeated retrievals of the same ACT-R chunk will increase its retrieval speed. Declarative chunks are not stored independently, however, so the activation of similar chunks will potentially cause interference. This means that increased familiarity with one chunk will come at the cost of slower retrieval times for similar chunks. Production compilation could also underly familiarity. Once a compiled production's utility exceeds that of its parents, it will be fired instead and result in speed gains.

R_S (size of a chunk) This term captures two notions of a chunk's "size": (1) its structural size (e.g., lines of code) and (2) the "*psychological complexity of identifying a chunk where a long contiguous section of non-branching code must be divided up in order to be understood.*" In other words, R_S should be effected by some notion of short-term memory constraints. An ACT-R model would be influenced by a chunk's structural size simply because there would be more code for the visual module to attend to and encode (i.e., more sequential productions fired). The additional "psychological complexity" could be modeled in several ways. ACT-R

[1] When operationalizing the definition of a chunk, Cant et al. admit that "*it is difficult to determine exactly what constitutes a chunk since it is a product of the programmer's semantic knowledge, as developed through experience.*"

[2] More general forms for R and T are discussed in [6], but Cant et al. suggest starting with simple additive representations.

does not contain a distinct short-term memory component, relying on long-term memory to serve as a short-term and working memory. According to Niels Taatgen, however, the decay and interference mechanisms present in ACT-R's subsymbolic layer can produce the appearance of short-term memory constraints [36].

R_C (control structures) The type of control structure in which a chunk is embedded influences R because conditional control structures like `if` statements and loops require the programmer to comprehend additional boolean expressions. In some cases, this might involve mentally iterating through a loop. We expect that boolean expressions would be comprehended in much the same way as for the R_E factor (see below). Modeling the programmer's mental iteration through a loop could draw on existing ACT-R models for inspiration. For example, a model of children learning addition facts (e.g., $1 + 5 = 6$) might "calculate" the answer to $5 + 3$ by mentally counting up from 5.[3] After many repetitions, the pattern $5 + 3 = 8$ is retrieved directly from memory, avoiding this slow counting process. Likewise, an ACT-R model of program comprehension could start out by mentally iterating over loops, and eventually gain the ability to recognize common patterns in fewer steps.

R_E (boolean expressions) Boolean expressions are fundamental to the understanding of most programs, since they are used in conditional statements and loops. According to Cant et al., the complexity of boolean expressions depends heavily on their form and the degree to which they are nested. To incorporate boolean expressions into an ACT-R model, it would be helpful to record eye-gaze patterns from programmers answering questions based on boolean expressions. A data set with these patterns, response times, and answers to the questions could provide valuable insight into how programmers of different experience levels evaluate boolean expressions. For example, it may be the case that experienced programmers use visual cues to perform pre-processing at the perceptual level (i.e., they saccade over irrelevant parts of the expression). The data may also reveal that experienced programmers read the expression, but make efficient use of conceptual-level shortcuts (e.g., `FALSE AND ... = FALSE`). These two possibilities would result in very different ACT-R models, the former making heavy use of the visual module, and the latter depending more on declarative memory.

R_R (recognizability) Empirical studies have shown that the syntactic form of a program can have a strong effect on how a programmer mentally abstracts during comprehension [15]. An ACT-R model would show such an effect if its representation of the program was built-up over time via perceptual processes. In other words, the model would need to observe actual code rather than receiving a pre-processed version of the program as input (e.g., an abstract syntax tree). Ideally, the ACT-R model would make errors like real programmers do when the code violates Soloway's unwritten rules of discourse (e.g., the same variable is used for multiple purposes) [34]. ACT-R has the ability to partially match chunks in memory by using a model-specific similarity metric. This ability would be useful for modeling recognizability, since slight changes in code indentation and layout should not confuse the model entirely.

R_V (visual structure) This term represents the effects of visual structure on a chunk's complexity, and essentially captures how visual boundaries influence chunk identification. While Cant et al. only describe

[3]The model would likely use the subvocalize feature of ACT-R's vocal module to simulate the child's inner voice.

three kinds of chunk delineations (function, control structure, and no boundary), more general notions of textual beacons and boundaries have been shown to be important in code [41]. For example, Détienne found that advance organizers (e.g., a function's name and leading comments) had a measurable effect on programmers' expectations of the code that followed [8]. Biggerstaff et al. have also designed a system for automatic domain concept recognition in code [3]. This system considers whitespace to be meaningful, and uses it to bracket groups of related statements. As with the recognizability term (R_R), it would be crucial for an ACT-R model to observe real code instead of an abstract syntax tree. ACT-R's visual module is able to report the xy coordinates of text on the screen, so it would be simple to define a notion of whitespace in the model.

R_D (dependency disruptions) There are many disruptions in chunking caused by the need to resolve dependencies. This applies both to remote dependencies (e.g., variable definitions), and to local dependencies (e.g., nested loops and decision structures). Cant et al. cite the small capacity of short-term memory as the main reason for these disruptions. As mentioned earlier, ACT-R does not have a distinct "short-term memory" module with a fixed capacity. Instead, short-term capacity limits are an emergent property of memory decay and interference. Given the biological plausibility of ACT-R's architecture, we should expect to find empirically that R_D effects in humans are actually context-dependent. In other words, the number of disruptions that a programmer can handle without issue should depend on the situation. This is a case where the psychological theory underlying the cognitive architecture can help to suggest new experiments on humans.

2.2.4 Tracing Difficulty (T)

Most code does not stand alone. There are often dependencies that the programmer must resolve before chunking the code in memory and ultimately understanding what it does. The Cognitive Complexity Metric (CCM) operationalizes the difficulty in tracing a dependency as $T = T_F(T_L + T_A + T_S + T_C)$. For these six terms, the definition of a chunk is slightly different. Rather than being a block of statements that must co-occur, a chunk during tracing is defined as a single statement involving a procedure's name or a variable definition.

T_F (familiarity) The dependency familiarity has a similar purpose to the chunk familiarity (R_F). As with R_F, ACT-R's subsymbolic layer will facilitate a familiarization effect where repeated requests for the same dependency information from declarative memory will take less time. The effect of the available tools in the environment, however, will also be important. An often-used dependency (e.g., function definition) may be opened up in a new tab or bookmarked within the programmer's development environment. A comprehensive ACT-R model will need to interact with the same tools as a human programmer to produce "real world" familiarization effects.

T_L (localization) This term represents the degree to which a dependency may be resolved locally. Cant et al. proposed three levels of localization: embedded, local, and remote. An embedded dependency is resolvable within the same chunk. A local dependency is within modular boundaries (e.g., within the same function), while a remote dependency is outside modular boundaries. This classification would fit

an ACT-R model that tries to resolve dependencies by first shifting visual attention to statements within the current chunk (embedded), switching then to a within-module search (local), and finally resorting to a extra-modular search (remote) if the dependency cannot be resolved. It is not clear, however, what the model should consider a "module," especially when using a modern object-oriented language and development environment. It might be more useful to derive a definition of "module" empirically instead. An ACT-R model in which the effects of dependency localization were emergent from visual/tool search strategies could be used to define the term (i.e., how the language and tools make some chunks feel "closer" than others).

T_A (ambiguity) Dependency ambiguity occurs when there are multiple chunks that depend upon, or are effected by, the current chunk. The CCM considers ambiguity to be binary, so a dependency is either ambiguous or not. Whether ambiguity increases the complexity of a chunk is also dependent on the current goal, since some dependencies do not always need to be resolved (e.g., unused parameters can be ignored). We expect an ambiguity effect to emerge naturally from an ACT-R model because of partial matching and memory chunk similarity. If the model has previously chunked two definitions of the variable x, for example, then a future query (by variable name) for information about this dependency may result in a longer retrieval time or the wrong chunk entirely.

T_S (spatial distance) The distance between the current chunk and its dependent chunk will affect the difficulty in tracing. Lines of code are used in the CCM as a way of measuring distance, though this seems less relevant with modern development environments. Developers today may have multiple files open at once, and can jump to variable/function definitions with a single keystroke. The "distance" between two chunks in an ACT-R model may be appropriately modeled as how much time is spent deciding how to locate the desired chunk (e.g., keyboard shortcut, mouse commands, keyword search), making the appropriate motor movements to interact with the development environment, and visually searching until the relevant code has been identified. This more complex version of T_S would depend on many things, including the state of the development environment (e.g., which files are already open), and the programmer's familiarization with the codebase.

T_C (level of cueing) This binary term represents whether or not a reference is considered "obscure." References that are embedded within large blocks of text are considered obscure, since the surrounding text may need to be inspected and mentally broken apart. This term appears to be related to the effect of visual structure on a chunk's complexity (R_V). Clear boundaries between chunks (e.g., whitespace, headers) play a large role in R_V, and we expect them to play a similar role in T_C. Tracing is presumed to involve a more cursory scan of the code than chunking, however. An ACT-R model may need to be less sensitive to whitespace differences during tracing than during chunking.

2.3 The ACT-R Cognitive Architecture

ACT-R is "*a cognitive architecture: a theory about how human cognition works.*" [1] It is a domain-specific programming language built on top of LISP, and a simulation framework for cognitive models. There are eight *modules* in ACT-R, which represent major components of human cognition (Figure 2). Modules exist at

Term	Description
R_F	Speed of recall or review (familiarity)
R_S	Chunk size
R_C	Type of control structure in which chunk is embedded
R_E	Difficulty of understanding complex Boolean or other expressions
R_R	Recognizability of chunk
R_V	Effects of visual structure
R_D	Disruptions caused by dependencies
T_F	Dependency familiarity
T_L	Localization
T_A	Ambiguity
T_S	Spatial distance
T_C	Level of cueing

Figure 1: *Important factors in the Cognitive Complexity Metric. R_x terms affect chunk complexity. T_x terms affect tracing difficulty.*

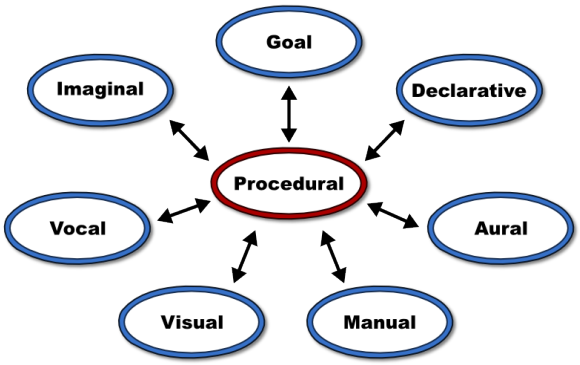
two layers: (1) the *symbolic* layer, which provides a simple programmatic interface for models, and (2) the *subsymbolic* layer, which hides real-world details like how long it takes to retrieve an item from declarative memory. ACT-R models formalize humans performance on tasks using production rules that send and receive messages between modules. Models are simulated and observed along psychologically relevant dimensions like task accuracy, response times, and simulated BOLD.[4] measures (i.e., fMRI brain activations) These simulations are reproducible, and can provide precise predictions about human behavior. We discuss the pieces of ACT-R in detail below, and provide examples of its success in other domains.

2.3.1 Buffers, Chunks, and Productions

The ACT-R architecture is divided into eight *modules*, each of which has been associated with a particular brain region (see [2] for more details). Every module has its own *buffer*, which may contain a single *chunk*. Buffers also serve as the interface to a module, and can be queried for the module's state. Chunks are the means by which ACT-R modules encode messages and store information internally. They are essentially collections of name/value pairs (called slots), and may inherit their structure from a parent chunk type. Individual chunk instances can be extended in advanced models, but simple modules tend to have chunks with a fixed set of slots (e.g., two addends and a result for a simple model of addition). Modules compute and communicate via *productions*, rules that pattern-match on the slot values of a chunk or the state of a buffer. When a production matches the current system state (i.e., all chunks in all module buffers), it "fires" a response. Responses include actions like inserting a newly constructed chunk into a buffer and modifying/removing a buffer's existing chunk.[5]

When it is possible, computations **within** a module are done in parallel. Exceptions include the serial fetching of a single memory from the *declarative* module, and the *visual* module's restriction to only focus on one item at a time. Communication **between** modules is done serially via the *procedural* module (see

[4]Blood-oxygen-level-dependent contrast. This is the change in blood-flow for a given brain region over time.
[5]Chunks that are removed from a buffer are automatically stored in the declarative memory module.



Module	Purpose
Procedural	Stores and matches production rules, facilitates inter-module communication
Goal	Holds a chunk representing the model's current goal
Declarative	Stores and retrieves declarative memory chunks
Imaginal	Holds a chunk representing the current problem state
Visual	Observes and encodes visual stimuli (color, position, etc.)
Manual	Outputs manual actions like key-presses and mouse movements
Vocal	Converts text strings to speech and performs subvocalization
Aural	Observes and encodes aural stimuli (pitch, location, etc.)

Figure 2: *ACT-R 6.0 modules. Communication between module buffers is done via the procedural module.*

Figure 2). Only one production may fire at any given time[6], making the procedural model the central bottleneck of the system.

The *visual*, *aural*, *vocal*, and *manual* modules are capable of communicating with the environment. In ACT-R, this environment is often simulated for performance and consistency. Experiments in ACT-R can be written to support both human and model input, allowing for a tight feedback loop between adjustments to the experiment and adjustments to the model. The *goal* and *imaginal* modules are used to maintain the model's current goal and problem state, respectively.

2.3.2 The Subsymbolic Layer

Productions, chunks, and buffers exist at the *symbolic* layer in ACT-R. The ability for ACT-R to simulate human performance on cognitive tasks, however, comes largely from the *subsymbolic* layer. A symbolic action, such as retrieving a chunk from declarative memory, does not return immediately. Instead, the declarative module performs calculations to determine how long the retrieval will take (in simulated time), and the probability of an error occurring. The equations for subsymbolic calculations in ACT-R's modules come from existing psychological models of learning, memory, problem solving, perception, and attention [1]. Thus, there are many constraints on models when fitting parameters to human data.

In addition to calculating timing delays and error probabilities, the subsymbolic layer of ACT-R contains

[6]This is a point of contention in the literature. Other cognitive architectures, such as EPIC [20] allow more than one production to fire at a time.

mechanisms for learning. Productions can be given *utility* values, which are used to break ties during the matching process.[7] Utility values can be learned by having ACT-R propagate rewards backwards in time to previously fired productions. Lastly, new productions can be automatically *compiled* from existing productions (representing the learning of new rules). Production compilation could occur, for example, if production P_1 retrieves stimulus-response pairs from declarative memory and production P_2 presses a key based on the response. If P_1 and P_2 co-occur often, the compiled $P_{1,2}$ production would go directly from stimulus to key press, saving a trip to memory. If $P_{1,2}$ is used enough, it will eventually replace the original productions and decrease the model's average response time.

2.3.3 Successful ACT-R Models

Over the course of its multi-decade lifespan, there have been many successful ACT-R models in a variety of domains (success here means that the models fit experimental data well, and were also considered plausible explanations). We provide a handful of examples below (more are available on the ACT-R website [1]).

David Salvucci (2001) used a multi-tasking ACT-R model to predict how different in-car cellphone dialing interfaces would affect drivers [28]. This integrated model was a combination of two more specific models: one of a human driver and another of the dialing task. By interleaving the production rules of the two specific models[8], the integrated model was able to switch between tasks. Salvucci's model successfully predicted drivers' dialing times and lateral deviation from their intended lane.

Brian Ehret (2002) developed an ACT-R model of location learning in a graphical user interface [13]. This model gives an account of the underlying mechanisms of location learning theory, and accurately captures trends in human eye-gaze and task performance data. Thanks to ACT-R's perception/motor modules, Ehret's model was able to interact with the same software as the users in his experiments.

Lastly, an ACT-R model created by Taatgen and Anderson (2004) provided an explanation for why children produce a U-shaped learning curve for irregular verbs [37]. Over the course of early language learning, children often start with the correct usage (I went), over-generalize the past-tense rule (I goed), and then return to the correct usage (I went). Taatgen and Anderson's model proposed that this U-shaped curve results from cognitive trade-offs between irregular and regular (rule-based) word forms. Retrieval of an irregular form is more efficient if its use frequency is high enough to make it available in memory. Phonetically post-processing a regular-form word according to a rule is much slower, but always produces something. Model simulations quantified how these trade-offs favor regular over irregular forms for a brief time during the learning process.

The success of these non-trivial models in their respective domains gives us confidence that ACT-R is mature enough for modeling program comprehension. In the next section, we discuss how ACT-R might serve as a base for an existing quantitative cognitive model of code complexity.

[7] This is especially useful when productions match chunks based on *similarity* instead of equality, since there are likely to be many different matches.

[8] Salvucci notes that this required hand-editing the models' production rules since ACT-R does not provide a general mechanism for combining models. See [29] for a more advanced model of multi-tasking in ACT-R.

Term	Definition
chunk	representation for declarative knowledge
production	representation for procedural knowledge
module	major components of the ACT-R system
buffer	interface between modules and procedural memory system
symbolic layer	high-level production system, pattern-matcher
subsymbolic layer	underlying equations governing symbolic processes
utility	relative cost/benefit of productions
production compilation	procedural learning, combine existing productions
similarity	relatedness of chunks

Figure 3: *ACT-R terminology*

2.4 Cognitive Domain Ontologies

A Cognitive Domain Ontology (CDO) is a formal representation of domain knowledge based on System Entity Structure (SES) theory [12]. SES theory is a formal specification framework for describing system aspects and properties [43]. For decades, researchers have used SES theory to automate the exploration of design space alternatives by enumerating the set of all possible system configurations, pruning them according to domain constraints, and then simulating/evaluating each pruned system. CDOs are a theoretical extension to SES in which "system configurations" capture *spaces of behavior or situational knowledge*. An agent uses a CDO to explore alternative courses of action, or evidence interpretations, based on a domain's structure, a-priori constraints, and situational factors. Psychologically, CDOs represent *mental models* and can produce the same underlying reasoning processes of *abduction, deduction, and induction* [18].

CDOs are formally represented as trees with *entities* as nodes, and one of three *relations* as edges (see next section for details). The pruning process for a CDO is cast as a constraint satisfaction problem (CSP), and is computationally realized using two extensions to Common LISP [35]. The Screamer [33] and Screamer+ [40] LISP extensions allow for non-deterministic execution of code by adding two special forms. The `either` form takes any number of LISP expressions, and establishes a *choice point*. The value of the first expression is returned, and control flow proceeds as normal until a `fail` form is reached. On each `fail`, Screamer backtracks to the nearest *either*, returning the value of its next expression. Once values are exhausted, evaluation jumps to the next nearest `either` or terminates. Listing 1 provides a LISP code example with `all-values`, a Screamer form that establishes a non-deterministic context and returns a list of all generated values. Screamer+ extends Screamer to allow for more complex data types in `either`, such as Common Lisp Object System (CLOS) objects [19].

```
> (all-values
    (let ((x (either 'a 'b 'c))
          (y (either 1 2 3)))
      ;; Exclude (B 2)
      (if (and (eq x 'b) (eq y 2))
          (fail))

      (list x y)))

;; ((A 1) (A 2) (A 3) (B 1) (B 3) (C 1) (C 2) (C 3))
```

Listing 1: *Example LISP code with Screamer extensions. The* `either` *and* `fail` *forms are used to generate values.*

2.4.1 Entities, Relations, and Constraints

A CDO consists of a set of *entities*, *relations* between entities, and *constraints*. Entities may also have one or more *attached variables*, which can store values like integers and strings, and may be used in constraints. Every CDO has a top-level, or root, entity and alternating levels of relations and child entities, forming a tree with entities at the leaves (see Figures 4 and 5). Each entity in the tree has a unique name and its own collection of attached variables. In a given *solution* from the domain, an entity may be active or inactive depending on the active state of its parent and the type of relation between them. The root entity is always active.

There are three types of relations between entities: *sub-parts*, *choice-point*, and *instance set*. A sub-parts relation, visually represented as an **and**, is a conjunction of its children. When the parent entity of a sub-parts relation is active, its child entities will necessarily be active. Sub-parts are used to represent required structure in a domain, such as the cost and performance characteristics of a computer component (the Details relation in Figure 5).

A choice-point relation, visually represented as an **xor**, is a disjunction of its children, with only **one child** being active at a time in a solution. Choice points are the source of generativity in CDOs, with all combinations of active/inactive choice point children forming the structure of the complete, unconstrained solution space. Because each choice under a choice point may have sub-structure, it is possible for different solutions from the same CDO to significantly differ in their tree structure.

Lastly, the instances set relation, visually represented as a **0..n**, creates n copies of its sub-structure in each solution. Like sub-parts, an instance set's child entities are active when its parent is active. Increasing n, the cardinality of the instance set, can exponentially expand the size of the solution space because all child choice points (and nested instance sets) are independent of each other. Constraints can be mapped across all child entities of an instance set, or may target specific instances by ordinal (ord).

Constraints. In addition to the structure of a domain, CDOs contain constraints that serve to prune out non-sensical or irrelevant sections of the solution space. The basic CDO constraint language (Table 1) is based on first-order logic with some additional operators for accessing and comparing entities and variables. Constraints typically set or key off of choice points, shutting down generativity in the Screamer constraint solver. In domains with instance sets, higher-order constraints can be mapped across instances (setting

instance variable values), or used to constrain the set (e.g., at most one instance with choice A). Section 2.4.3 provides details on the higher-order constraint operators.

Operator	Description	Example
`==>`	Implication (If)	`(==> p q)`
`<==>`	Biconditional (IFF)	`(<==> p q)`
`not`	Negation (Not)	`(not p)`
`or`	Disjunction (Or)	`(or p q)`
`and`	Conjunction (And)	`(and p q)`
`e@`	Entity in CDO	`(e@ e1 e2)`
`v@`	Variable in CDO entity	`(v@ (e1 e2) v1)`
`equale`	Entity equality	`(equale e1 e2)`
`equalv`	Variable equality	`(equalv v1 v2)`
`let`	Local binding	`(let ((p p')) (equale p p'))`

Table 1: *First order CDO constraint language.*

2.4.2 Ball Example

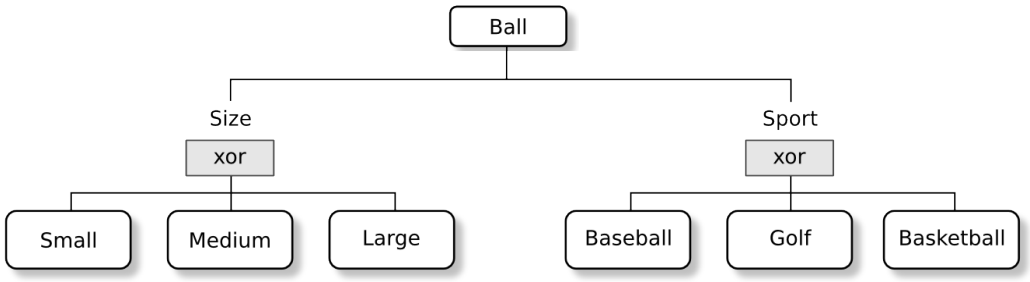


Figure 4: *The ball CDO. A ball has a size and associated sport.*

Figure 4 shows a simple example domain. In this domain, a *ball* is broken down into two components: a *size* and a *sport*. Both components are choice points with three options, making for a total of 9 possible solutions. With no constraints, the solution space consists of all combinations of *size* and *sport* (Table 2).

This solution space is not very useful, however, as it contains many nonsensical solutions. Golf, baseball, and basketballs should be small, medium, and large respectively. Three simple constraints will serve to constrain our domain:

1. $Small \iff Golf$
2. $Medium \iff Baseball$
3. $Large \iff Basketball$

Note that all three constraints are **bidirectional**. A unidirectional constraint, such as $Small \Rightarrow Golf$ would allow for more solutions, including medium and large golf balls. With these new constraints in place, our solution space size is reduced from 9 to 3 (Table 3).

Solution	Size	Sport
1	Small	Baseball
2	Small	Golf
3	Small	Basketball
4	Medium	Baseball
5	Medium	Golf
6	Medium	Basketball
7	Large	Baseball
8	Large	Golf
9	Large	Basketball

Table 2: *Unconstrained solution space of the ball CDO.*

Solution	Size	Sport
1	Small	Golf
2	Medium	Baseball
3	Large	Basketball

Table 3: *Constrained solution space of the ball CDO with all constraints.*

Translated into the Common LISP CDO framework, the top-level Ball entity becomes a `ball` function which takes a set of constraints as parameters. Each entity underneath Ball is accessible as a variable that can be referenced in constraints. The `solutions` function explores the solution space defined by the top-level entity and constraints, and enumerates `:all` solutions or `:one` solution.

```
> (solutions
    (ball
      (<==> small golf)
      (<==> medium baseball)
      (<==> large basketball))
    :all)

;; 1. ball: small, golf
;; 2. ball: medium, baseball
;; 3. ball: large, basketball
```

2.4.3 Higher-Order Constraints

When a CDO includes one or more instance set relations, it is useful to write constraints that function across sets of entities and variables. Table 4 lists the operators available for instance sets and *higher order* constraints – e.g., constraints that take other constraints as parameters. Operators like `every` and `at-least` apply some constraint to an instance set and require, respectively, that all or at least n of them hold. The `mapc` operator is similar, but is used to apply side effects to all entities in an instance set, such as calculating the value of a variable. Any first order constraint can make use of the special `ord` to determine which instance they are being applied to. This allows for higher order constraints to be sensitive to ordering in an instance set; often

used when order reflects spatial arrangement.

Operator	Description	Example
`n@`	Entity set under an instance	`(n@ inst)`
`ord`	Special variable for instance ordinal	`(equalv (v@ (e1) ord) 1)`
`exactly`	Exactly n instances match	`(exactly 2 cstr (n@ inst))`
`at-least`	At least n instances match	`(at-least 1 cstr (n@ inst))`
`at-most`	At most n instances match	`(at-most 3 cstr (n@ inst))`
`mapc`	Map constraint with side effects across instances	`(mapc cstr (n@ inst))`

Table 4: *Instance set and higher order CDO constraint operators.*

If the Ball entity in our example above was underneath an instance set (called `balls`), the 3 bidirectional constraints would now need to be applied across each `ball` instance. For example:

```
(every
  (and
    (<==> small golf)
    (<==> medium baseball)
    (<==> large basketball))
  (n@ balls))
```

The higher order `every` constraint would ensure that all `ball` instances adhered to the size/sport constraints.

2.4.4 Computer Configuration Example

To demonstrate the use of each relation (sub-parts, choice points, and instance sets) and higher order constraints, we will use a CDO that models the configuration of a desktop computer. Figure 5 shows the CDO structure. At the top level, we say that a computer configuration consists of a set of components (0..n). Each component has a type (graphics, memory, sound), a role in the configuration (active, not active), and details of the product, such as its vendor, cost, and performance. Note that cost, performance, and model are *variables* attached to the Product entity. Attached variables can be used in constraints, but unlike choice points, Screamer does **not** generate possible values for them.

With no constraints and 8 components, the space of possible solutions is quite large. The 3 component types (2 memory types), 2 roles, and 3 vendors make up $(3+2) \times 2 \times 3 = 30$ possible choices. Assuming our configuration only has **8 components**, there are a total of 30^8 possible solutions! Clearly, we need some constraints before exploring the solution space.

Table 5 lists the details of the components we will be considering in this example. The first 5 components are new, and may be purchased for the given cost. The last 3 components are currently in use, so they cost nothing to continue using. Performance numbers have been associated with each component, with higher being better. We will say that the cost and performance of an entire configuration is the sum of the costs and performances of its **active** components. To make things interesting, a performance boost of 5 will be applied to any configuration with multiple memory chips that are all the same type (A or B).

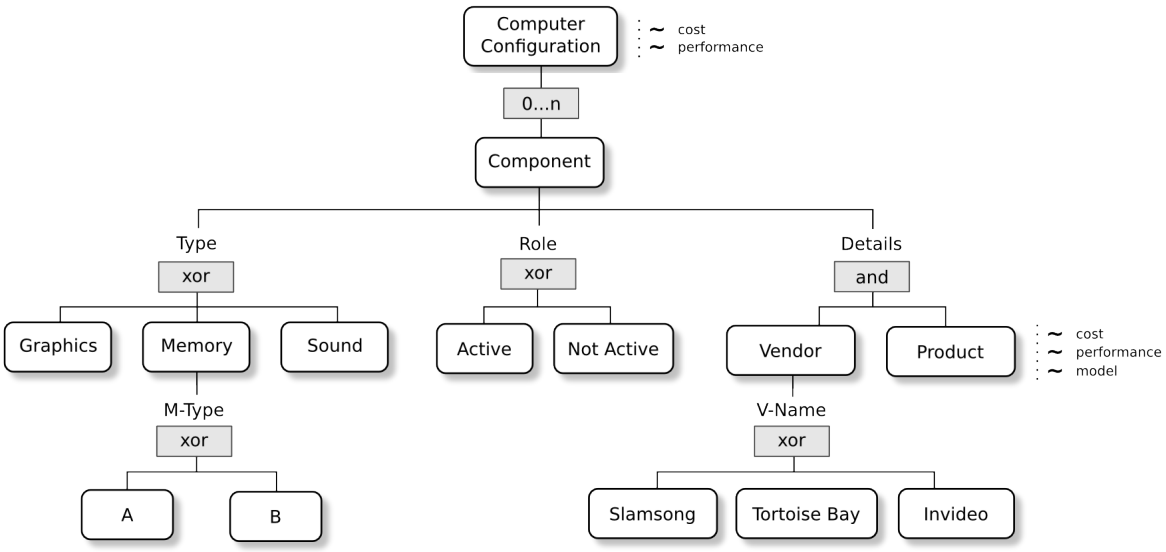


Figure 5: *A CDO representing a computer configuration with any number of components.*

	Type	Vendor	Model	Cost	Perf
1	Graphics	Invideo	D-Force	200	10
2	Memory	Slamsong	DDR9 (B)	20	10
3	Sound	Tortoise Bay	Waves	50	10
4	Memory	Slamsong	DDR7 (A)	10	5
5	Graphics	Invideo	B-Force	100	7
6	Memory	Slamsong	DDR7 (A)	0	5
7	Sound	Slamsong	Puddle	0	1
8	Graphics	Slamsong	A-Force	0	2

Table 5: *Components to consider for computer configuration example. Existing components have zero cost.*

We collect all of the constraints necessary to represent these 8 components into a LISP variable called `*all-components*` (full source code is available in Appendix B). The LISP code below prints the first solution from the constrained search space, including the summed cost and performance of all active components (those with a * next to them).

```
> (solutions
    (computer-configuration
      *available-components*)
    :one)

;; Configuration (cost:380, perf:50):
;;   1. * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;   2. * [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;   3. * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
```

```
;;   4. * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;   5. * [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;   6. * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;   7. * [SOUND] SLAMSONG Puddle (c:0, p:1)
;;   8. * [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
```

If we interpret this solution as a recommended configuration, it suggest our computer should have 3 graphics cards, two sound cards, and 3 memory chips of varying types. A typical desktop computer, however, will only have a single graphics/sound card, and one or two memory chips. We will further constrain the solution space by enforcing the constraints in Table 6. Our computer will only have 4 components slots, and must have 1 graphics card, 1 sound card, and 1 or 2 memory chips.

Constraint	Description
`max-4-active`	A maximum of 4 components may be active in any configuration.
`only-1-graphics-card`	A configuration must have one graphics card.
`only-1-sound-card`	A configuration must have one sound card.
`1-or-2-memory`	A configuration must have 1 or 2 memory chips.

Table 6: *Default constraints for computer configuration example.*

Collecting the constraints from Table 6 into a LISP variable called `*default-constraints*`, let's look at the first solution again:

```
> (solutions
    (computer-configuration
      *all-components*
      *default-constraints*)
    :one)

;; Configuration (cost:280, perf:35):
;;   1. * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;   2. * [MEMORY] SLAMSONG DRR9 B (c:20, p:10)
;;   3. * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;   4. * [MEMORY] SLAMSONG DRR7 A (c:10, p:5)
;;   5.   [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;   6.   [MEMORY] SLAMSONG DRR7 A (c:0, p:5)
;;   7.   [SOUND] SLAMSONG Puddle (c:0, p:1)
;;   8.   [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
```

Now only 4 components are active, and we have the correct number of component types for our system. Note that the performance is precisely the sum of the individual component performances because our memory chips are of different types. This configuration includes all new, high performance components, and may be the most expensive option. What would a solution look like if we didn't want to spend any money (i.e., use only existing components)?

```
;; One solution, force no cost
> (solutions
    (computer-configuration
      *all-components*
      *default-constraints*
      ;; Apply constraint across all components
      (every
        ;; If a component is active...
        (if (equale (e@ component role) active)
            ;; ...its cost must equal zero.
            (equalv (v@ (component details product) cost) 0))
        (n@ components)))
    :one)

;; Configuration (cost:0, perf:8):
;;   1.   [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;   2.   [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;   3.   [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;   4.   [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;   5.   [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;   6. * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;   7. * [SOUND] SLAMSONG Puddle (c:0, p:1)
;;   8. * [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
```

While this configuration is definitely cheaper than the previous one, its less than a quarter of the performance. Ideally, we'd like to explore the entire (constrained) solution space, and sort all solutions by their cost and performance numbers. The CDO framework supports this type of search via user-defined *utility* and *objective* functions.

2.4.5 Utility and Objective Functions

In addition to `:all` and `:one`, the `solutions` function accepts requests for `:best`. The extra keyword arguments `:utility-fun` and `:objective-fun` are the functions for assessing the value of a given solution (its utility), and for sorting the utility values (the objective). The example below defined a new function `performance-utility` that simply extract the configuration's performance value. The built-in `>` function serves as our objective function, putting higher utility values (better performing solutions) on top. All solutions with the "best" utility value in the space are returned.

```
;; Extract the summed performance
(defun performance-utility (cfg)
  (v@ (cfg) performance))

;; Best performance
```

```
> (solutions
    (computer-configuration
      *available-components*
      *default-constraints*)
    :best
    :utility-fun #'performance-utility
    :objective-fun #'>)

;;  Configuration (cost:280, perf:35):
;;   1. * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;   2. * [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;   3. * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;   4. * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;   5.   [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;   6.   [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;   7.   [SOUND] SLAMSONG Puddle (c:0, p:1)
;;   8.   [GRAPHICS] SLAMSONG A-Force (c:0, p:2)

;;  Configuration (cost:270, perf:35):
;;   1. * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;   2. * [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;   3. * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;   4.   [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;   5.   [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;   6. * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;   7.   [SOUND] SLAMSONG Puddle (c:0, p:1)
;;   8.   [GRAPHICS] SLAMSONG A-Force (c:0, p:2)

;; Configuration (cost:260, perf:35):
;;   1. * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;   2.   [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;   3. * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;   4. * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;   5.   [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;   6. * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;   7.   [SOUND] SLAMSONG Puddle (c:0, p:1)
;;   8.   [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
```

Three solutions with a performance value of 35 (the maximum) exist in the solution space. The first solution should look familiar: a recommendation to purchase all new components. The second solution, however, achieves the same performance with a lower cost by reusing the existing DDR7 memory chip. The final solution reduces the cost further by purchasing a new, lower-performing, DDR7 chip instead of the new

DDR9 chip. This is due to the memory performance boost mentioned earlier – a bonus 5 points are added to any configuration with multiple memory chips of the same type (A or B). Conceptually (and verbosely), the memory boost constraint looks like this:

```
;; Get the sum of all component performance numbers.
(let ((cfg-performance
        (sum (mapc (v@ (component performance)) (n@ components)))))
  ;; Give +5 to configs with same memory types.
  (if (and
        ;; If at least 2 active memory components...
        (at-least 2 (and (equale (e@ component role) active)
                         (equale (e@ component type) memory)))
        (or
          ;; ...each active memory chip must be type A...
          (every (if (and (equale (e@ component role) active)
                          (equale (e@ component type) memory))
                     (equale (e@ component type memory m-type) A))
                 (n@ components))

          ;; ...or type B.
          (every (if (and (equale (e@ component role) active)
                          (equale (e@ component type) memory))
                     (equale (e@ component type memory m-type) B))
                 (n@ components)))

        ;; If the memory chips are the same type, give the boost...
        (equalv (v@ (configuration) performance)
                (+ 5 cfg-performance))

        ;; ...otherwise, just return the sum of performances.
        (equalv (v@ (configuration) performance)
                cfg-performance))))
```

Multi-Objective Functions. As a last demonstration, we will show how multi-objective functions can be used to order a solution space across multiple dimensions. The previous example used `performance-utility` to sort by performance alone. When performances are highest and equivalent, we would like to also minimize costs. This can be done manually for our computer configuration example because there are only 3 solutions with the maximum performance, but there may be many more in the general case.

The LISP code below defines a new utility function `performance-and-cost-utility` that creates a two-element list for each configuration with the performance value first and the cost value second. The next function, `better-2`, sorts solutions for the highest performance first, and the lowest cost second. With this as our objective function, we should get the best performing configuration with the lowest cost value. Indeed,

running `solutions` with the new function returns a single solution: the configuration with mostly new components that purchases a DDR7 memory chip! A graphical depiction of this solution is shown in Figure 6 with component 6 enlarged to show excluded choice points and the values of attached variables.

```
;; Extract performance and cost as a list
(defun performance-and-cost-utility (cfg)
  (list
   (v@ (cfg) performance)
   (v@ (cfg) cost)))

;; Highest performance (first), lowest cost (second)
(defun better-2 (a b)
  (cond
    ;; Performances equal - compare costs
    ((eq (first a) (first b)) (< (second a) (second b)))
    ;; Compare performances
    (t (> (first a) (first b)))))

;; Best performance with lowest cost
> (solutions
    (computer-configuration
      *available-components*
      *default-constraints*)
    :best
    :utility-fun #'performance-and-cost-utility
    :objective-fun #'better-2)

;; Configuration (cost:260, perf:35):
;;   1. * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;   2.   [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;   3. * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;   4. * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;   5.   [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;   6. * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;   7.   [SOUND] SLAMSONG Puddle (c:0, p:1)
;;   8.   [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
```

2.4.6 Constraint Knowledge and Cognition

The computer and ball examples above demonstrate interesting ways to use CDOs, but the *cognitive* aspect of these domain ontologies may not be immediately apparent. Depending on how they are designed

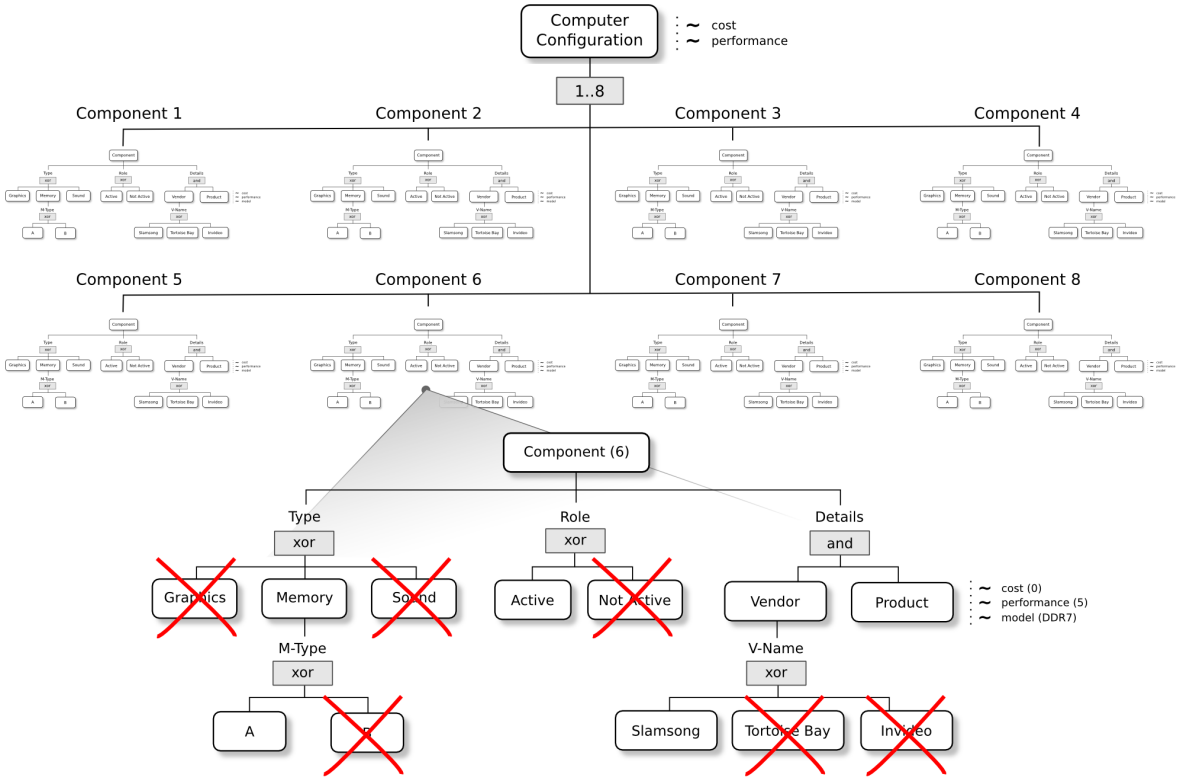


Figure 6: *Graphical representation of a single solution. One component is expanded to show excluded choice points and variable values.*

and searched, CDO solution spaces can be used to abduce likely explanations from evidence, deduce the remainder of incomplete knowledge, and induce possible consequences from observed (or desired) actions.

In *A Framework for Modeling and Simulation of the Artificial*, Douglass et. al describe a situated, autonomous agent that searches rooms in a task environment for a reward [12]. The agent contains a library of behaviors, called *behavior models*, which detail **how** the agent can achieve a specific sub-task. A CDO is used to map situational factors, such as the overall state of the task and current percepts, to specific behaviors. In other words, the agent's CDO helps the agent decide **what** to do next. This same CDO could also be used to go the other direction: from behaviors to percepts. Given a set of asserted behaviors, the CDO's solution space will contain all percepts relevant to their selection. If the agent had determined a candidate set of next behaviors based on some other knowledge, these inferred percepts would tell the agent *what to look for* in the environment in order to exclude candidate behaviors.

3 Mr. Bits

The Mr. Bits model operationalizes components of Cant et. al's Cognitive Complexity Metric [6] (CCM) within the ACT-R cognitive architecture [2]. Using the Python debugger, Mr. Bits produces an ACT-R script for a given Python program. This script simulates a programmer "reading" and evaluating the program, with the final output being fixations and keystroke timings. When rendered on top of the program's code, these data have the appearance of human trials in front of an eye-tracker (Figure 7). The **chunking** and **tracing** processes described in the CCM are realized through ACT-R's declarative memory and visual modules (specially, EMMA [27]). Keystrokes are also simulated via the ACT-R motor module, which utilizes Fitts Law [14].

While Mr. Bits *appears* to read and evaluate a program from scratch, the model does not actually parse text or determine the evaluation order of lines, and therefore **cannot make errors**. Despite this limitation, the model serves as a starting point for a truly quantitative, cognitive model of program comprehension. Mr. Bits produces human-like eye movements and keystrokes by "reading" over code tokens, recalling variable values from memory, and performing mental arithmetic. A second model, called Nibbles (Section 4), complements Mr. Bits by transforming raw text into an internal (mental) representation of the underlying program. This mental model can then be used to determine what to do next. Future work will integrate the Mr. Bits and Nibbles models, removing the need for the Python debugger. The following sections describe the details of Mr. Bits, and Section 3.5 compares the model's output to human data collected from the same 24 Python programs.

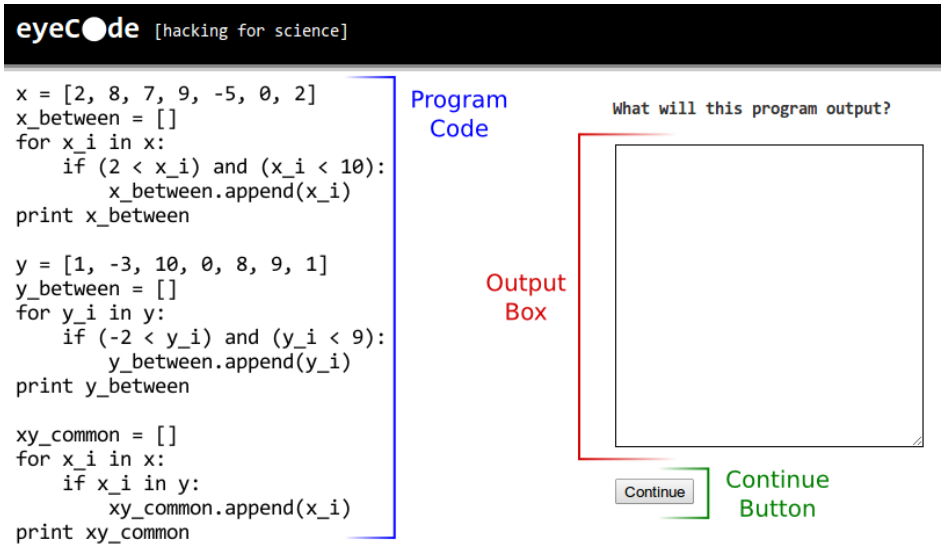


Figure 7: *Interface used by human programmers in the eyeCode experiment. Mr. Bits uses the same interface.*

3.1 Architecture

Mr. Bits takes a Python program (a text file) as input, and produces a time series of fixations and keystrokes as output. In between, there are three important stages (Figure 8): (1) the built-in Python debugger is used to

parse and evaluate the program, (2) an ACT-R LISP script is generated with a program-specific goal stack for each step of the evaluation, and (3) the ACT-R script is executed, and the resulting trace is transformed into fixation/keystroke timings. These data can then be compared to human data to, for example, see if human programmers and Mr. Bits spent the same amount of relative time on each line of the program.

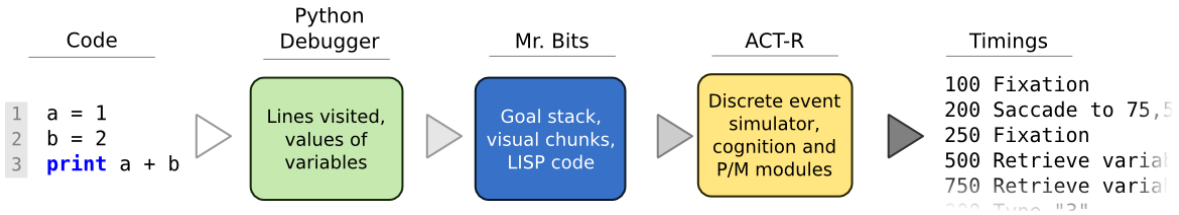


Figure 8: *Mr. Bits model workflow. Code is transformed into fixation and response timings via Python and ACT-R.*

The **first stage** of the Mr. Bits model parses and evaluates a Python program using the built-in Python debugger. The output from this stage is a series of code lines in evaluation order, and the values of each variable during the course of evaluation. Note that the ACT-R portion of Mr. Bits does **not** parse or evaluate code. This is the focus of our second model, Nibbles (see Section 4). As the Python program is parsed, variables are tagged with contextual information, such as which function they are defined in and how many times they are assigned to. When program evaluation is simulated in ACT-R, Mr. Bits uses this information to resolve variable ambiguities, and trace to specific code locations for variable values.

During the **second stage**, a self-contained LISP script is generated with a complete visicon (visual locations of code words), a goal stack with each step of program evaluation, and a collection of general-purpose productions for reading/remembering text, tracing variable values, and typing responses. The Visual, Declarative, and Manual ACT-R modules are relied upon to generate human-like timings (see Figure 2 for a description of each module). The goal stack contains four general categories of goals:

1. **Look at/remember line** - The leftmost word of every evaluated code line is first fixated by the Visual module. A declarative memory request for the "details" of the line is made and, if not present, every word in the line is fixated from left to right.
2. **Store variable value/location/reference** - Every assignment statement, `for` loop, or mutable function call (e.g., `list.append`) results in a new declarative memory chunk with the target variable's name, context, and *physical location*. Future attempts to recall the variable's value will require at least one trace to this location before the value is stored in declarative memory. Calls to functions defined in the same program (e.g., `def f(x)`) create a series of variable reference chunks, linking the function's parameters to the supplied arguments. Future requests for a function's parameter value will cause Mr. Bits to follow the reference back to the call site arguments (e.g., `f(1)`).
3. **Recall variable value** - When a variable's value is needed (for a computation, printing, etc.), a declarative memory request is made for the exact name in the context of the current function. If not present, Mr. Bits attempts to recall the variable's physical location, follow references to other variables, or consult previous variable definitions with the same name. If all else fails, the process is repeated within the previous context (either another function or the global context). The values of traced variables are stored in declarative memory, so future retrievals will succeed until the value is forgotten.

4. **Typing responses** - Every `print` statement produces a set of goals, causing Mr. Bits to fixate the experiment's output box, type each character of the response, and re-fixate the previous line. When program evaluation is complete, Mr. Bits will fixate the "continue" button, move its hand to the virtual mouse, and click the button to end the trial (see Figure 7).

State Transition Diagram. Figure 9 shows a state-transition diagram for the ACT-R simulation (final stage) of Mr. Bits. At a high level, goals are processed one by one from the goal stack produced during the previous model stage. When the goal stack is empty, Mr. Bits clicks the continue button to end the trial. Table 7 describes each goal in detail as well as the specific ACT-R modules involved. The declarative memory module is used heavily, with processes like variable look-up and arithmetic cast as memory retrievals (a common practice in ACT-R modeling [2]).

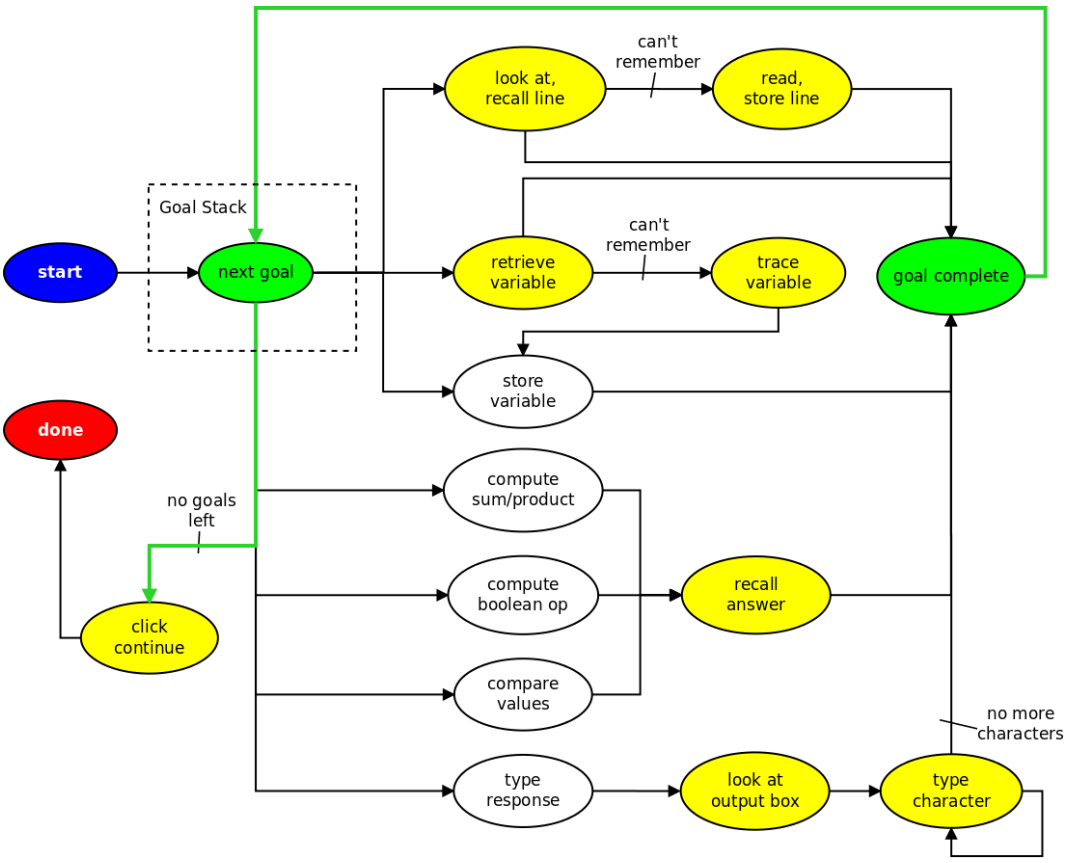


Figure 9: *General state diagram for a Mr. Bits ACT-R script. States highlighted in yellow may have variable timings when ACT-R's sub-symbolic mode is enabled.*

The Goal Stack. A sample program and goal stack produced by Mr. Bits is shown in Figure 10. The fixation timeline on the right-hand side of the figure shows the model fixating specific code lines and the output box when typing responses. While Mr. Bits' goal stack is pre-populated during the parsing stage, the fixations generated during model execution depend on dynamic factors. First, code lines are only read (i.e., each whitespace-separated token is fixated) if they have not been viewed before. Second, variable values are

Goal	ACT-R Module(s)	Description
Go to line	Visual/DM	Moves the virtual eyes to the first character of a line. If the details of the line cannot be retrieved from memory, every token is read.
Remember line	DM	Stores the details of the current line in memory.
Recall variable	Visual/DM	Attempts to retrieve the value of a variable from memory. Failure results in a retrieval of the variable's location, a visual trace to it, and storage of the result in memory.
Store variable	DM	Stores a variable's value in memory.
Compute sum/product	DM	Retrieves a sum or product from memory.
Compare numbers	DM	Retrieves a numeric relationship (less/greater than) from memory.
Evaluate Boolean expression	DM	Retrieves the result of a boolean expression (AND/OR) from memory.
Fixate output box	Visual	Locates the output box on the screen and fixates the upper-left corner.
Type response	Manual	Types a text response, character by character.

Table 7: *Possible goals in the Mr. Bits model. DM stands for Declarative Memory.*

only stored in the model's declarative memory (DM) once they have been traced. As a result, Mr. Bits will shift visual attention to variable values as they are needed. Finally, when ACT-R's sub-symbolic behavior is enabled (described in Section 3.3), declarative memory retrievals for code lines and variable values can fail after enough time has passed and their DM activations have fallen below threshold. In other words, Mr. Bits can forget. On the other side of the coin, sub-symbolic behavior will speed up the retrieval of frequently and recently used chunks. Often-used arithmetic facts, such as $5 \times 8 = 40$, may have increased activation in memory, resulting in faster in-situ retrievals. Existing DM chunks and activations at the start of a trial represent Mr. Bits' *long-term experience*, while accumulated post-trial chunks/activations represent *short-term learning*. We do not currently retain short-term learned information across Mr. Bits trials, though this would be an interesting path for future research.

3.2 Variables and Context

In the scope of our simple Python programs, a variable has three pieces of information that uniquely identify it. First, the surrounding function name is used to differentiate global and local function scope. Mr. Bits will use the local function scope first when resolving a variable by name. Second, the number of times a function has been called – its "function call index" – helps distinguish between multiple calls to the same function. In Figure 11, `f(x)` is called twice, and so the local `x` variable within `f` will potentially have different values during each call. Mr. Bits considers these two "versions" of `x` to be separate, but related, variables.

Lastly, variables can be reassigned within the same scope and function call index. In Figure 11, the

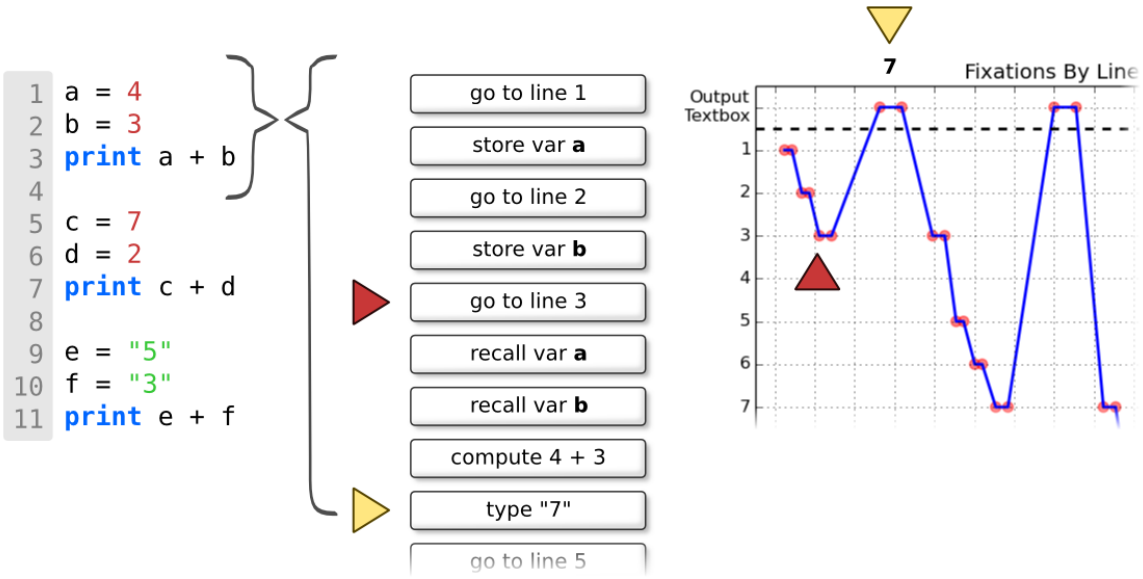


Figure 10: *Example of a Mr. Bits goal stack (center) for a Python program (left). Processing the goals produces fixations and keystrokes (right).*

variable `a` (in the global scope) is assigned a value twice. Mr. Bits treats these as two different variable definitions with the same name, and will always trace to the most recent definition first (i.e., the definition with the highest line number in the current scope).

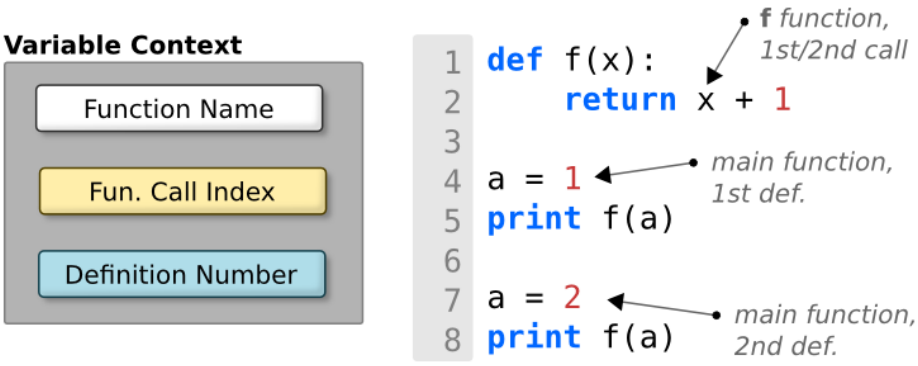


Figure 11: *Variable context*

These three pieces of information: function scope, function call index, and definition number, allow Mr. Bits to uniquely identify variables in each of our sample Python programs. More complex programs, especially those with class definitions, will require extensions to this basic notion of variable context. Presently, Mr. Bits only needs to locate values from the right-hand side of assignment statements (e.g., `x = 5`) and passed as function arguments (e.g., `f(5)`). With classes, it will be necessary to maintain the type

information for each definition of a variable. Python allows variables to change types when reassigned, so the expression `x.foo(5)` may be dispatched to different code depending on the (current) type of `x`. For now, we leave these extensions as future work.

3.3 Sub-symbolic Chunking and Tracing

ACT-R can operate in *symbolic* or *sub-symbolic* mode. In the former mode, all ACT-R modules (visual, memory, etc.) behave with consistent timings. For example, the declarative memory module in symbolic mode will retrieve chunks immediately, bypassing the activation calculus. When sub-symbolic behavior is enabled, however, retrieval times will depend on the frequency and recency of a chunk's use. Additionally, a chunk can be "forgotten" if its activation falls below a critical threshold (see `rt` in Table 8). Mr. Bits relies heavily on declarative memory for remembering code line details, variable values, traced locations, and arithmetic/boolean facts (see Section 3.4) for details). In sub-symbolic mode, Mr. Bits will get *faster* at remembering a recently or often-used variable value. These values can also be forgotten over time, though none of our sample programs are long enough to demonstrate this behavior.

The activation calculus in ACT-R subsumes the two familiarity parameters in the Cognitive Complexity Metric (Section 2.2). The R_F and T_F parameters, respectively chunk and trace familiarity, are operationalized as retrieval latency in declarative memory. Frequently-used, or familiar, code lines, variable values, and trace locations will be retrieved quickly whereas less familiar items will take more time. If we equate cognitive complexity with the time it takes Mr. Bits to read through a program, this successfully captures the intentions of the familiarity CCM parameters.

A total of 5 ACT-R parameters have been given non-default values in Mr. Bits (Table 8). The latency factor (`lf`) and retrieval threshold (`rf`) parameters were set via an informal search in order to better fit the human data. These two declarative memory parameters are among the most frequently modified ACT-R parameters, and the chosen values are within the ranges used in other human studies [42]. The `imaginal-delay` parameter was set to 0 in order to quickly move newly created chunks from the imaginal buffer into declarative memory. Mr. Bits does not use the imaginal buffer to hold a representation of the current line or variable, so there is no modeling need for a delay. Lastly, the two the motor parameters (feature preparation and burst time delays) were reduced by a factor of 15 in order to account for programmers' above average typing speeds. As with the declarative memory parameters, an informal search was used to produce visibly similar behavior relative to human trials. Neither motor parameter has an entry in the Max Planck Institute for Human Development's ACT-R parameter database [42], so it is unknown whether our values are cognitively plausible.

3.4 Sums, Comparisons, and Boolean Expressions

Mr. Bits computes sums/products/differences, does numeric comparisons ($x < y$), and evaluates simple boolean expressions ($x \wedge y$) using declarative memory. Each of these "facts" reside in Mr. Bits' declarative memory – e.g., the sum of 2 and 3 is 5 – and have been given a high *base level activation*, representing that they are very well rehearsed. We do not model an explicit *process* of addition, subtraction, etc., such as counting up or down from an anchoring number, because (1) virtually all numeric operations in our simple Python programs are done with small operands (0-10), and (2) participants in our experiment were experienced

ACT-R Parameter	Default	Value	Description
`lf`	1.0	0.01	Latency factor (F in the retrieval equation).
`rt`	0.0	-2.0	Retrieval threshold. Minimum activation for retrieval.
`imaginal-delay`	0.2	0.0	Delay in seconds for imaginal buffer request.
`motor-feature-prep-time`	0.05	0.001	Time in seconds to prepare each movement feature.
`motor-burst-time`	0.05	0.001	Minimum time in seconds for any motor movement.

Table 8: *ACT-R parameters with non-default values used in Mr. Bits.*

enough to have memorized these basic mathematical facts. ACT-R models of basic arithmetic exist [2], but we assume our participants have internalized these facts directly in long-term memory.

When Mr. Bits "computes" the sum of two numbers, x and y, a declarative memory retrieval is initiated with the constraints that it must be a `sum-fact` whose operands are x and y. Operations with more than two operands are serialized into multiple retrievals, each with only two operands. For example, the sum $1 * 2 + 3$ would be transformed into two retrievals:

1. A `prod-fact` with operands 1 and 2 (result: 2),
2. A `sum-fact` with operands 2 and 3 (result: 5)

Mr. Bits relies on Python to determine the order of operations, and so is not capable of making arithmetical errors as long as its declarative memory facts are consistent.

If ACT-R's subsymbolic behavior is disabled, there is little difference between declarative memory and a database. With subsymbolic effects, however, subsequent retrievals of the same fact will be *faster*, especially when done in quick succession. For example, Mr. Bits will take less time to compute $2 \times 2 \times 2$ than $2 \times 3 \times 2$ because, in the former case, the fact that $2 \times 2 = 4$ will be retrieved quicker the second time. Because chunks in ACT-R's declarative memory have an activation value, the same system can model both long-term and short-term/working memory [36]. Capacity limitations for short-term memory appear in ACT-R as a side effect due to (1) the ability to retrieve only one chunk at a time, and (2) the time costs associated with retrievals and productions (steps in the model). Only a fixed number of chunks can be attended to before activation decay becomes a factor in retrieval. Cognitive theories of program complexity often cite Miller's Magic Number [23] as a potential source of complexity – e.g., having a programmer attend to too many things simultaneously in a snippet of code will make it harder to understand.

Mr. Bits currently contains sum, product, difference, less-than, and greater-than facts for numbers zero to ten. In order to model more experienced programmers, additional facts could be added to declarative memory. For example, intermediate and advanced programmers will quickly note that $2 \times 2 \times 2 = 2^3 = 8$, and not need to mentally compute the intermediary products.

3.5 Results and Discussion

In order to evaluate Mr. Bits, we compare the model's performance on 24 of the 25 programs used in our eyeCode experiment (Section 2.1) with data from the 29 human programmers. All model runs used the

same ACT-R parameters, as described in Table 8. We considered 4 different versions of Mr. Bit, each with a combination of ACT-R's subsymbolic computing turned on (SSC) or off (SC), and with the ability to either perfectly remember lists (RL) or forget lists (FL).

3.5.1 Human-Model Comparison

We compared the relative spent on each line of each program by Mr. Bits to the time spent by our participants. For some programs, a large difference is visually apparent (Figure 12, human data is on the left). If Mr. Bits is allowed to remember the values of **all** variables after tracing them, it will avoid the need to continually refer back to *lists* like `x` and `y` like humans do. We created a setting for Mr. Bits that stopped it from remembering non-empty list variable values, forcing the model to trace those variables over and over. With this setting enabled, the shift in where time is spent is again visually apparent (Figure 13), and appears closer to the human data. The two versions of Mr. Bits, with the setting disabled or enabled, are referred to as RL (Remember Lists) and FL (Forget Lists). Combined with the option to place ACT-R in symbolic or sub-symbolic mode, we have four different versions of Mr. Bits to compare against human data:

Figure 12: *Relative time spent on each line of* `between functions` *by human participants (left) and Mr. Bits Remember Lists (right).*

1. **SC + RL** - Symbolic Computing + Remember Lists. All declarative memory retrievals return immediately, and lists can be fully remembered.
2. **SC + FL** - Symbolic Computing + Forget Lists. All declarative memory retrievals return immediately, and lists are traced every time.
3. **SSC + RL** - Sub-Symbolic Computing + Remember Lists. All declarative memory retrievals return immediately, and lists can be fully remembered.
4. **SSC + FL** - Sub-Symbolic Computing + Forget Lists. All declarative memory retrievals return immediately, and lists are traced every time.

Which version of Mr. Bits correlates best with human data? We compared model runs from each version to human data for each program by running a Spearman correlation between the average times spent on each line. . In addition, we created two much simpler "null" models for baseline comparison: called **line length** and **line number**. For the line length model, relative time spent on each line is simply proportional to its length, so longer lines will receive more time. In the line number model, line time is inversely proportional

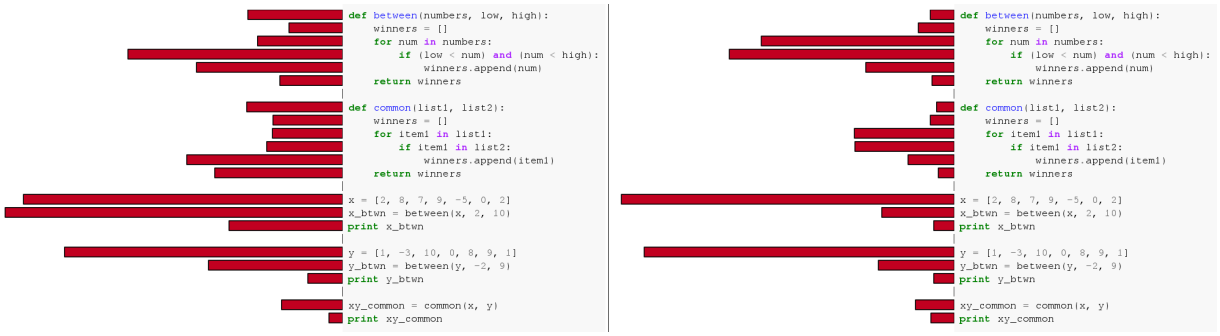


Figure 13: *Relative time spent on each line of* `between functions` *by human participants (left) and Mr. Bits Forget Lists (right).*

to line number, so lines appearing earlier in the program will receive more time. Figure 14 provides an example of each null model using the `between functions` program.

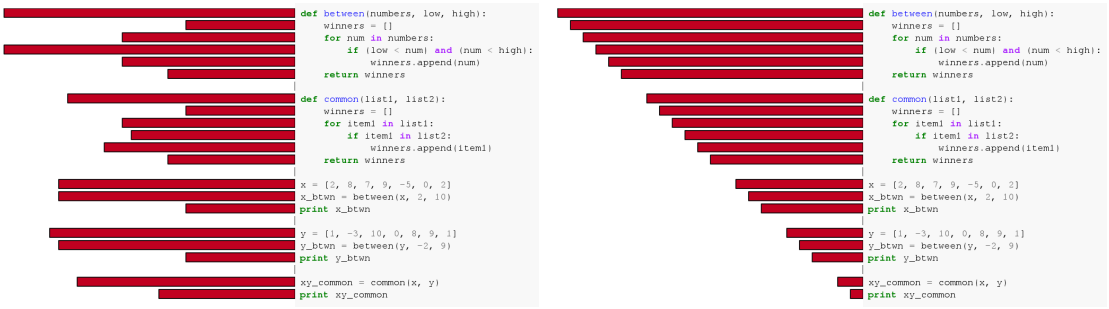


Figure 14: *Examples of relative line times generated from the two null models: line length (left) and line number (right).*

The complete correlation matrix for 4 versions of Mr. Bits and the two null models is shown in Figure 15. For each program and each model, the Spearman correlation is computed using the average times spent on each line for the model and our human programmers. A higher, green value indicates a strong positive correlation while a higher, redder value means a strong *negative* correlation with the human data. Correlation values not meeting the $\alpha = 0.05$ significance criteria or below 20 (0.2) are not displayed (though their cells are still colored).

If we take the sum of correlation values as our success criteria, then the line length and SSC+FL (sub-symbolic computing + forget lists) models come out on top, followed somewhat closely by SC+FL, SSC+RL, and SC+RL [9]. This success criteria and the use of correlations here is not intended to provide a statistical argument for one model over another. Instead, we simply take these results as evidence for two things: (1) forgetting lists is a useful model setting, and (2) line length is a strong contributor to average line time. While it is common to accept a simpler model in favor of a more complex one (e.g., line length vs. any Mr. Bits version), our goal with Mr. Bits is not just achieving a good data fit. We wish for our model to **explain** the programmer's underlying cognitive processes while achieving a reasonable fit to human data. The time spent by Mr. Bits on each line is not just a function of its character length, but also depends on when and how frequently the line is viewed. When combined with the Nibbles model (see Future Work in Section 5.1), it will be possible for the model to spend less time on more *idiomatic* lines – i.e., those fitting a commonly-seen

[9] The line number model trails *far* behind the others, so we do not consider it further.

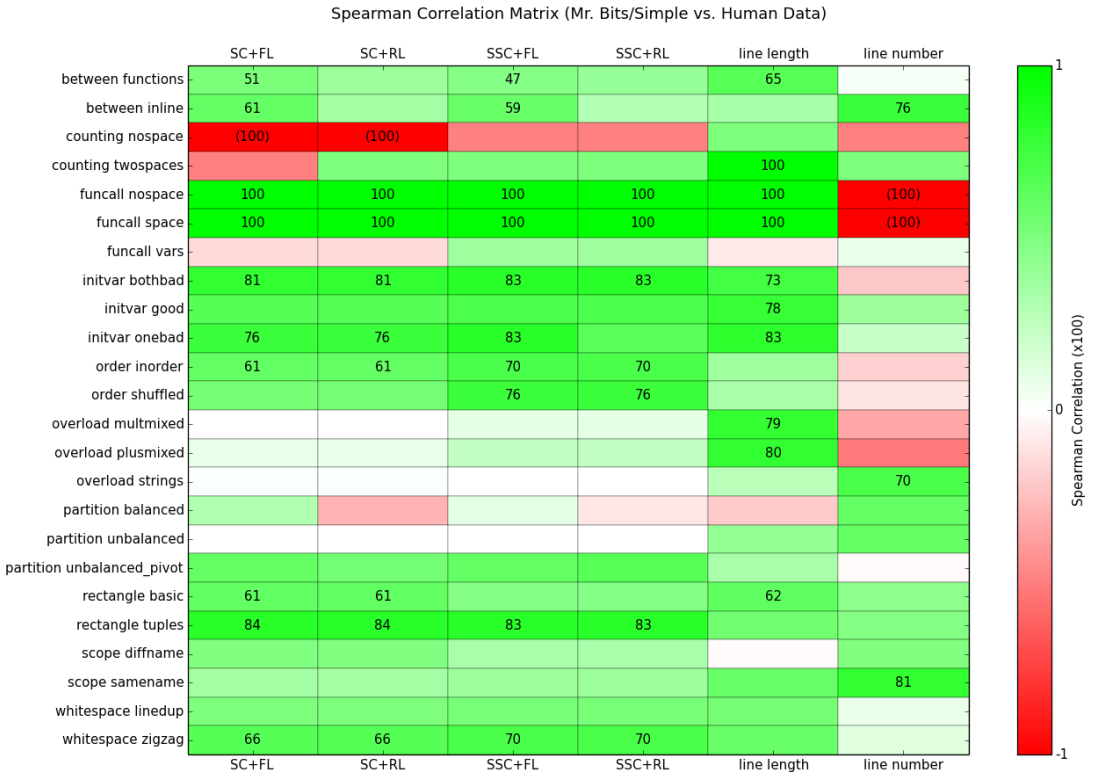


Figure 15: *Correlations between relative time spent on each line for human trials, 4 versions of Mr. Bits, and two simple models based on line length and number.*

template.

3.5.2 Human Trial Times

We now compare the **total time** taken by Mr. Bits and humans to read and evaluate each program. This "trial time" is a useful summary metric if we assume that more cognitively complex programs should take longer to understand and predict their output. Intuitively, we can imagine this is a function of program length, but with the added twist that commonly-used or repeated patterns will speed participants along. Unfortunately, *errors can reverse expectations* because failing to properly evaluate portions of the program may result in a smaller trial time (or vice-versa, depending on the program). Therefore, we must be aware of the correctness of a human trial in addition to its length when comparing it to Mr. Bits.

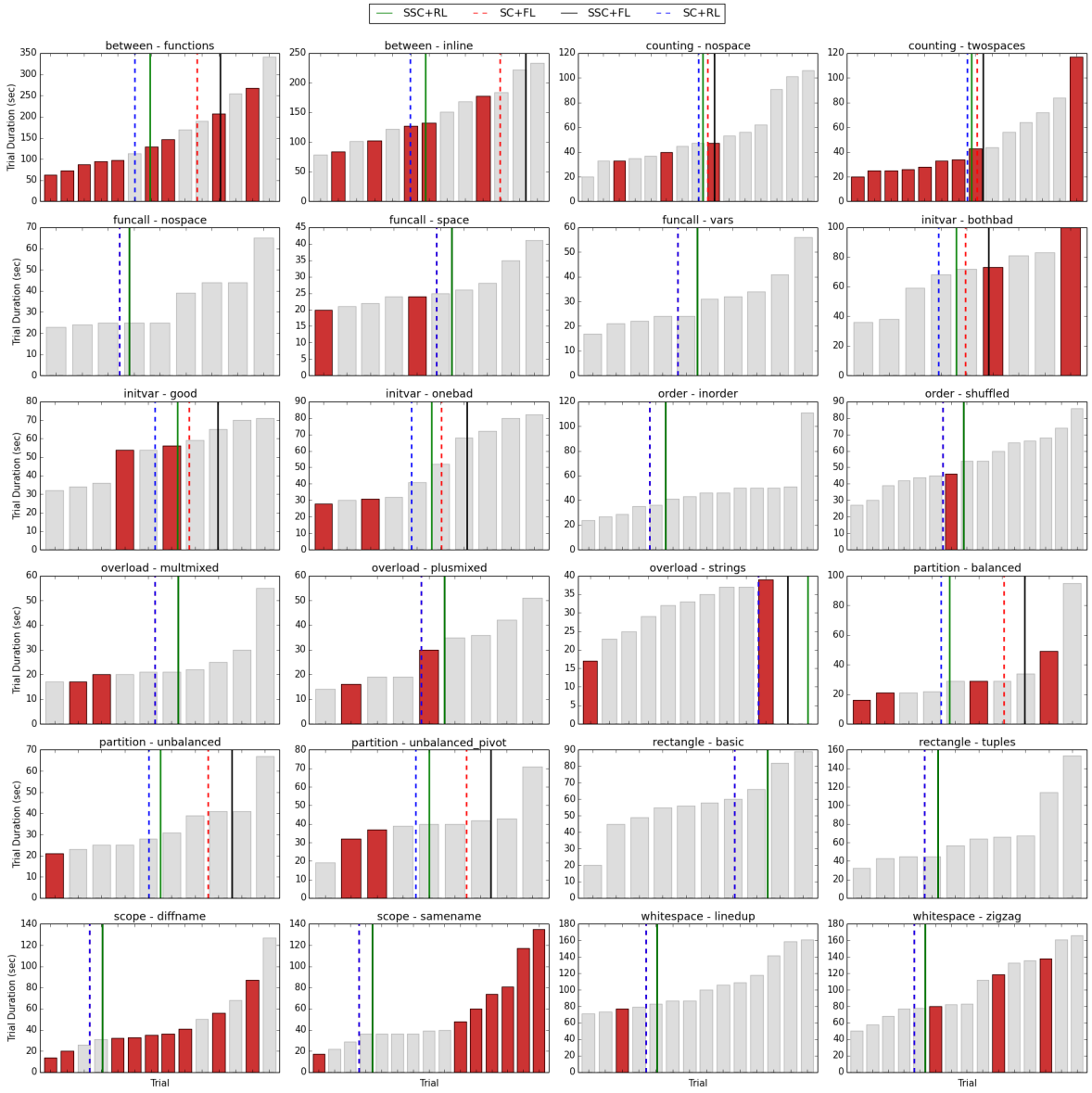


Figure 16: *Mr. Bits trial times (4 versions) compared to human trial time distributions for each program. Incorrect trials are colored red.*

The distributions of human trial times across 24 Python programs is shown in Figure 16. For each program, the individual trial times (bars) have been sorted from shortest to longest, and the trials with incorrect responses have been colored red. The (single) trial times for all 4 versions of Mr. Bits (with or without subsymbolic, remembering or forgetting lists), are shown as lines, plotted where they would fall in the sorting process. In some program sets, such `funcall`, there is no difference between the remember lists (RL) and forget lists (FL) versions of Mr. Bits because these programs do not contain lists.

With few exceptions, incorrect human trials do not appear strongly correlated with trial time. The two notable exceptions are `counting twospaces` and `scope samename`. The former is easily explainable: the most common error significantly reduces the amount of expected output, and therefore the number of response characters the participant must type. The `scope samename` error pattern is not as easy to explain, and may simply be a fluke. Regardless, it is visually apparent that Mr. Bits' trial times tend to fall in the middle of the human distributions. An interesting exception is `overload strings`, where Mr. Bits takes longer than any human participant when ACT-R's subsymbolic mode is engaged. This is likely due to ACT-R's manual (typing) module; its subsymbolic timings are subject to Fitts Law. Because this program involves typing English words like "bye" and "penny", participants are likely to be much faster than Mr. Bits, for whom "nypen" is just as slow to type as "penny".

3.5.3 Trial Time as a Complexity Metric

Because Mr. Bits' trial times track human performance (with some exceptions as caveats mentioned above), it may be useful as a complexity metric for ranking programs (or versions of a program). Figure 17 contains a correlation matrix for the trial times of all 4 Mr. Bits versions against 4 commonly-used complexity metrics: lines of code, Cyclomatic Complexity [22], Halstead Volume [17], and number of output characters. Again, non-significant ($\alpha = 0.05$) correlations and correlation values less than 20 (0.2) are not printed.

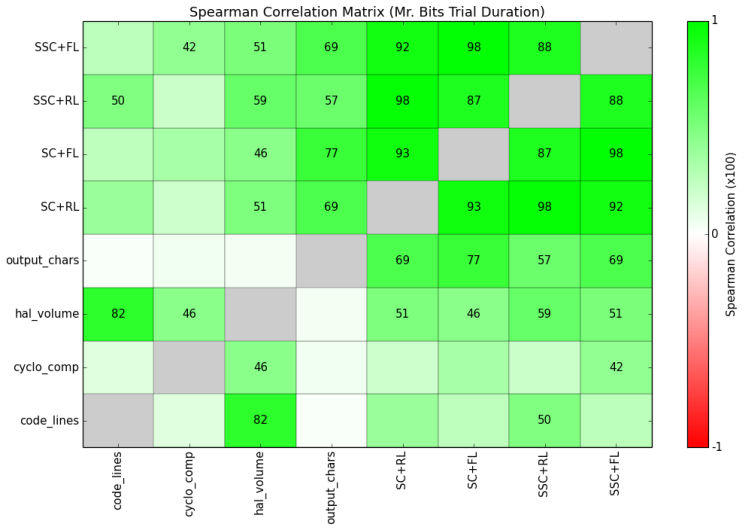


Figure 17: *Correlations between Mr. Bits trial times and code complexity metrics for all programs.*

While the correlation matrix cannot directly tell us whether or not Mr. Bits' trial time (MBTT) is a useful

complexity metric, we can at least see that it is not strongly correlated with any of the common source code metrics. In other words, MBTT is not simply restating the fact that a program has many lines or conditional statements (Cyclomatic Complexity). Interestingly, all versions of MBTT are moderately correlated with Halstead Volume, which is derived from the ratios of unique and total operators and operands. More research is needed to untangle this relationship, but we expect the way Mr. Bits computes sums and products to play a major role (Section 3.4). Lastly, the moderate to strong correlations between MBTT and output chars is expected because Mr. Bits must type a response for each program. It is good news too that the MBTT/output chars correlations are not very strong or perfect. This would suggest that Mr. Bits' trial times are almost entirely driven by its typing speed! Overall, the correlation matrix provides evidence for the utility of MBTT as a code complexity metric. An obvious path for future work (Section 5.1) would be to extend Mr. Bits to work on a larger subset of Python and run the model on a large collection of open source programs (e.g., from GitHub or SourceForge).

3.5.4 Limitations and Threats to Validity

Mr. Bits has a number of limitations and specific design choices that may threaten its validity as a model of human program comprehension. First and foremost, the model **cannot make errors**. This is due to the fact that Mr. Bits does not parse code, and therefore cannot internalize an incorrect program. Our other model, Nibbles (Section 4), does just this, but does not predict fixations and keystroke timings like Mr. Bits does. Unlike a human too, Mr. Bits fixates each and every word when reading a line (the line is skipped if it can be recalled from memory). This is not how humans read natural language text [25], and it is clear from our experimental data that this is not how programmers read code either. A more comprehensive model combining Mr. Bits and Nibbles would be capable of inferring unobserved code words or tokens, and avoid the need to fixate them. Eye-tracking studies of programmers have shown that keywords are the least fixated tokens in a program, providing evidence that easily inferred tokens (e.g., the `in` keyword in `for x in y`) are often skipped [5].

Python programs with user-defined classes, generators, and other advanced language features are not currently supported by Mr. Bits. Additionally, the program must be script-like – i.e., compute and print values to the terminal. There are many other kinds of programs, such as those with a graphical interface, but Mr. Bits has been designed to work with a single task: output prediction. Additionally, Mr. Bits' code environment is quite bare-bones compared to modern integrated development environments (IDEs). Syntax highlighting alone could drastically change the model, and is currently being investigated in the context of Nibbles.

Finally, the LISP scripts generated by Mr. Bits will likely give ACT-R modelers pause. Unlike most ACT-R models, Mr. Bits scripts contain dozens of productions, effectively removing any ability to claim that it's a *low-level* model of human program comprehension. Because each production in an ACT-R model is (in many ways) a free parameter, it is not possible to argue that their specific combination represents the best possible model when there are dozens of productions. While we cannot argue for our individual productions, we can still find value in the high-level behavior of Mr. Bits. ACT-R imposes constraints on how its modules interact, and how long particular actions take (e.g., shifting visual attention). Thus, Mr. Bits will exhibit different macro behaviors when particular programs emphasize different modules by, for example, having information physically more spread out or by including many unrelated calculations. In this way, Mr. Bits

serves as a framework for describing human program comprehension and, most importantly, as a model whose high-level predictions can be **falsified by future experiments**.

4 Nibbles

The Mr. Bits model builds on top of the ACT-R cognitive architecture to produce human-like fixation and keystroke timings while evaluating simple Python programs (Section 3). Mr. Bits does not, however, actually parse text into an internal representation and determine the evaluation order of program lines. For this purpose, we propose **Nibbles**, a model of program comprehension that makes use of Cognitive Domain Ontologies, or CDOs, to internally represent a program (Section 2.4). The process of searching for the first constraint compliant solution in the CDO's solution space allows Nibbles to handle missing text and local ambiguities. Future work will join the Nibbles and Mr. Bits models to produce an end-to-end model capable of transforming raw text into human-like fixations and keystroke timings.

4.1 Choice Point Uses

Formally, Nibbles represents a program using a Cognitive Domain Ontology, or CDO. A CDO is a tree with entities and relations, together describing a space of possible "solutions". With a set of domain and/or situation-specific constraints, a constraint solver is used to prune the tree and enumerate one or more constraint-compliant solutions. The *choice point* relation is special, in that only one of its child entities may be present in a given solution. Choice points are the source of generativity in a CDO, and Nibbles uses them in three important ways: (1) to classify entities, (2) to determine group membership, and (3) to control the size of the solution space. We expand on each of these uses of choice points below.

Choice points can be used to **classify** or categorize a particular entity. For example, the Line Type choice point in the Nibbles CDO (Figure 20) classifies a Line entity as either a function call, variable assignment, etc. When searching for solutions, each of these choices will be enumerated and checked against the constraints. Every constraint-compliant solution will contain a single choice for every active choice point.

Lines are grouped by Nibbles using spatial and semantic cues. Relative indentation and line type (e.g., a `for` loop), will determine if this and nearby lines form a cohesive group (e.g., a `for` loop plus its body). The Group Role choice point can either be Active or Inactive for every line in every line group. Each Line Group contains all lines in the program, initially with an indeterminate Group Role. When Active, Nibbles will enforce additional constraints to ensure that a line is only active in one group, and that the given line type fits within the group. This is not a classification of lines, but is instead an indication of a line's **membership in a group**.

In addition to classification and group membership, Nibbles uses choice points to **limit the size of the CDO solution space**. At every level of the CDO tree, a Detail Level choice point controls whether instances at that level will be High or Low detail. If High detail is set for every Detail Level, the complete solution space is available – down to the character level of every word in the program. Without high performance computing resources, it is infeasible to enumerate all solutions for programs with only 5 lines or less. Nibbles is able to explore much larger programs by carefully controlling the detail level such that only the currently attended line must be High detail. Using a form of *attention*, then, Nibbles is able to tame large solution spaces without being exposed to the full brunt of the combinatorics.

4.2 CSP and Text Interpretation

Nibbles parses and interprets text using three components: (1) a CDO tree to describe the structure of the domain, (2) a collection of constraints to exclude non-sensical solutions, and (3) a constraint solver enumerate constraint compliant solutions from the domain (Screamer [33]). Each line of code is provided to Nibbles with high fidelity, as if it were placed under a virtual fovea.

Figure 18 depicts the transformation of a simple three line program (bottom) into a "mental model" representation in Nibbles (right). Using its existing knowledge of Python (constraints), Nibbles abduces that the three lines form a single group, comprised of two assignment statements and one print statement. Furthermore, the types of variables *a* and *b* are inferred as well as the sense of the overloaded plus (+) operator (numeric sum versus string append).

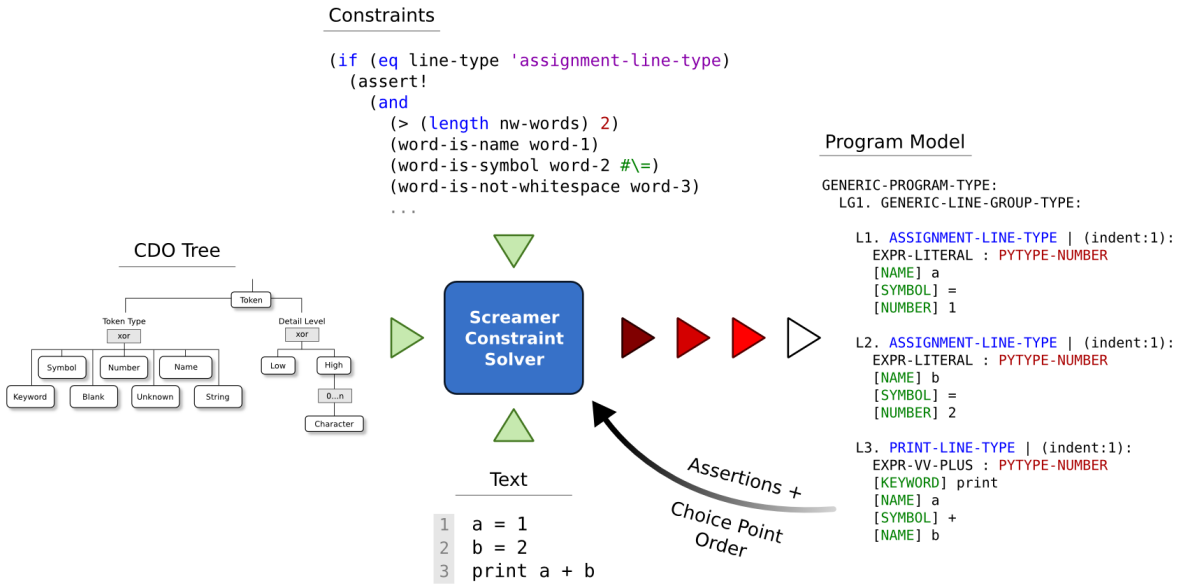


Figure 18: *Information flow for Nibbles model. The interpretation of each line involves a trip to the constraint solver and a subsequent re-ordering of choice points.*

Nibbles reads programs incrementally line-by-line by focusing a character-aligned sensor (fovea) on one line at a time. The observed characters are asserted into the CDO along with the following information:

1. The sizes of unobserved tokens on the current line and two lines below are available.
2. The indent level of the line below (relative to the current line) is available.
3. The sizes of unobserved lines is available.
4. The presence of text above and below the current line is available.

Figure 19 shows the first few steps of Nibbles reading the `overload plusmixed` program. The complete program is on the left (highlighted for the reader's benefit). On the immediate right, unobserved characters are replaced with "?". Spaces between blocks of "?" represent the availability of token sizes, while entire lines of "?" reflect knowledge of line lengths.

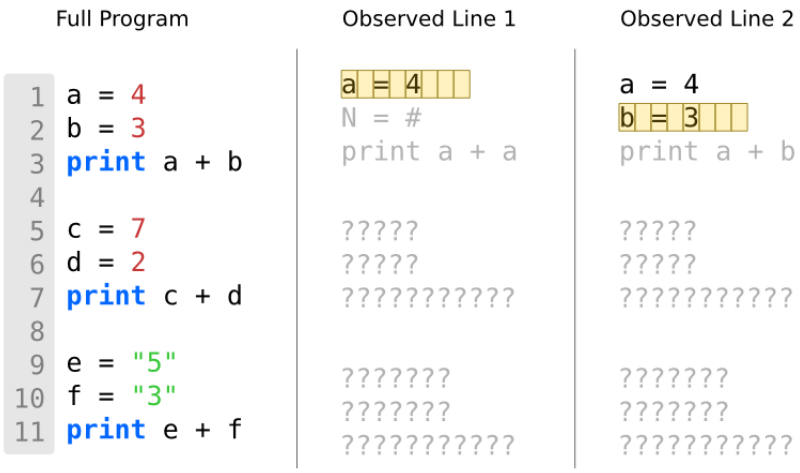


Figure 19: *Incremental reading of a program. The first two lines of the program are observed, and the third line is inferred.*

Unobserved code is represented in the Nibbles CDO by setting the line group, line, or token type to Unknown. Additionally, Low Detail choice points are used to dynamically control the size of the CDO solution space. Summary information, such as the size of a line or token, is stored in an unobserved Low Detail choice. This will stop instance set (of tokens, characters, etc.) under the High Detail choice from being enumerated by the constraint solver. This serves the dual purpose of speeding up the model and representing a form of **attention**. With every High Detail choice set, exploring the space of even a five line program can be computationally infeasible. See Section 4.1 for more details.

4.3 The Nibbles CDO

The structure of Nibbles' CDO is shown in Figure 20. A program is broken down in the following components:

- **Line Groups** - Contiguous groups of related code lines that may or may not all be at the same indent level.
- **Lines** - A single code line in a line group.
- **Tokens** - A single Python token/word in a line (usually separated by whitespace).
- **Characters** - A single character in a token.

The complete Nibbles CDO is shown in Figure 20. A program consists of one or more line groups, which are made of one or more lines, etc. As previously described, nested instance sets at each level of the CDO hierarchy fall under a High Detail choice. By toggling the detail level choice points, Nibbles can dynamically contract and expand the solution space – a form of *attention*. A particular solution for the Nibbles CDO will contain High Detail information for observed code, and Low Detail (or Unknown) information for previously/unobserved code.

Line groups, lines, and tokens are categorized using the Line Group Type, Line Type, and Token Type choice points. Table 9 contains an exhaustive list of each category, and a brief description of how the category is verified by Nibbles' constraints. Lines that contain an *expression*, such as the right-hand side of a variable

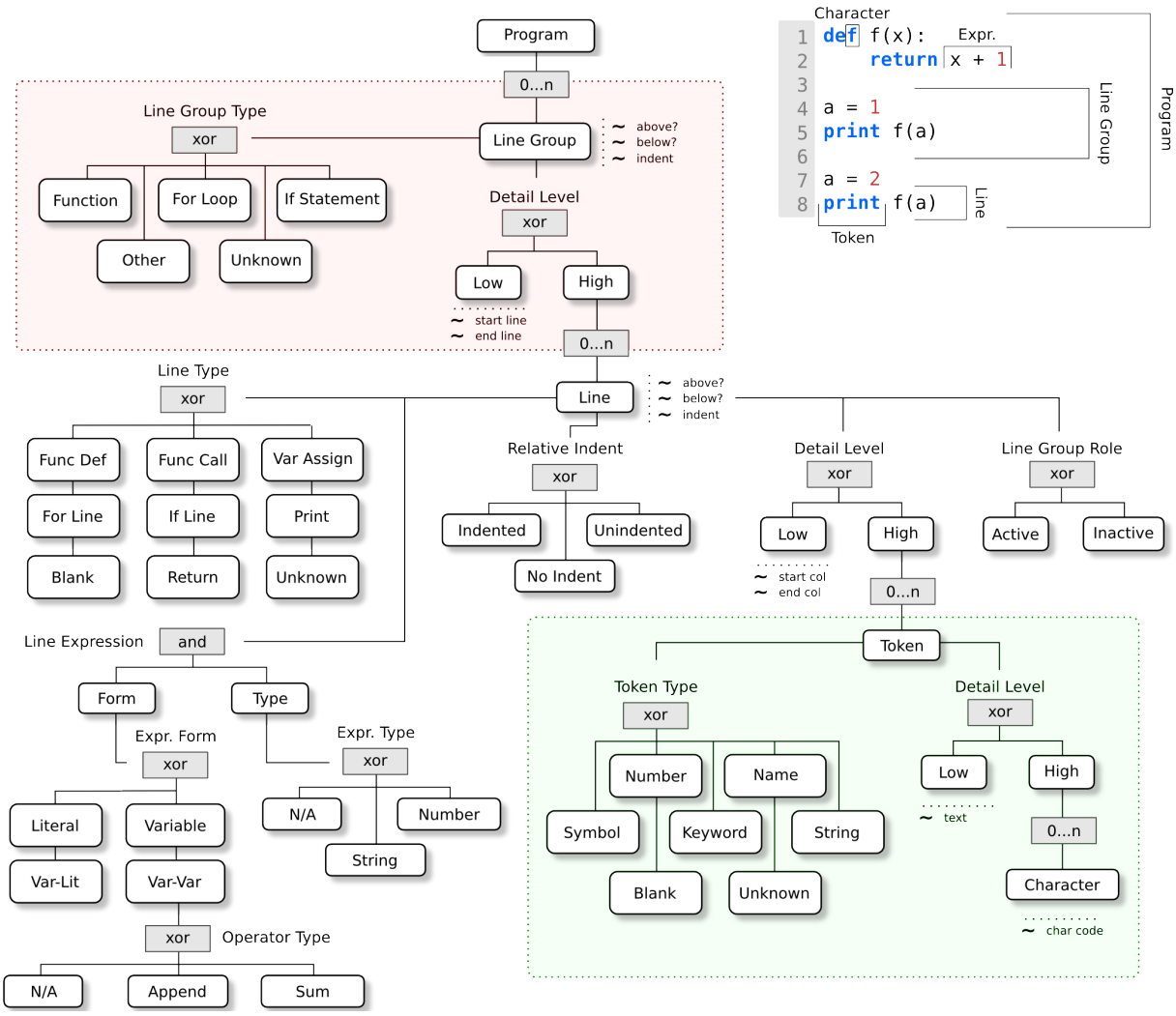


Figure 20: *Structure of the Nibbles CDO.*

assignment, make use of the additional Line Expression structure to determine the expression's form and Python type. Table 10 lists the currently supported forms/types, which focus just on single token and simple binary expressions (e.g., `x + 1`). Future work will expand this into a more comprehensive set.

4.4 Programming Knowledge as Constraints

A CDO captures the structure of a domain as well as its constraints. Nibbles encodes knowledge of Python into both. The textual structure of a program is represented hierarchically in Nibbles' CDO, as groups of lines, tokens, and characters. Constraints at every level of the hierarchy enforce Python's grammatical rules – e.g., the line after a function `def` must be indented. Table 11 provides high-level descriptions of Nibbles' constraints on Line Groups, Lines, and Tokens.

In the Screamer constraint solver, constraints are really just Common LISP functions that inspect proposed

Category	Name	Description
Line Group	Other/Generic	Default line group type (no other type matches).
	Function	A function definition. 1st line must a Function Def line, 2nd line must be indented.
	For Loop	A `for` loop block. 1st line must be a For line, 2nd line must be indented.
	If Statement	An `for` loop block. 1st line must be a For line, 2nd line must be indented.
Line	Function Def	Starts with a `def` Keyword token, followed by a Name and (optional) arguments.
	Function Call	Starts with a Name token, followed by (optional) arguments.
	Variable Assignment	Starts with a Name token, followed by an = Symbol and an expression.
	For Line	Starts with a `for` Keyword token, followed by a Name, `in` Keyword, and either a Name or list literal.
	If Line	Starts with an `if` Keyword token, followed by an expression.
	Print	Starts with a `print` Keyword token, followed by an expression.
	Return	Starts with a `return` Keyword token, followed by an expression.
	Blank	Line contains only Whitespace tokens.
Token	Symbol	A non-alphanumeric token – e.g., `=`, `(`, etc.
	Number	A single string of numeric characters – e.g., `123`.
	Name	A single string of alphabetic characters and underscores – e.g., `the_list`.
	Keyword	One of `def`, `for`, `if`, `print`, `in`, `return`.
	String/Text	Any string of characters that are not blank and not one of the other types.
	Whitespace/Blank	A token that is entirely whitespace or of zero length.

Table 9: *Types of tokens, lines, expressions, and line groups in Nibbles.*

sections of the CDO tree and produce failures when specific checks are violated. Because of this, it can be difficult to describe precisely what constraints are doing without using the code itself. While this assumes a familiarity with Common LISP, the Screamer (and Screamer+) constraint language, and the codified Nibbles CDO structure, it is still possible to meaningfully understand the constraints without them. Listing 2, for example, shows a snippet of the constraint the verifies tokens based on characters. In this particular snippet, a `print` keyword has been proposed, and is verified by a set of assertions. Even without the necessary background, this constraint snippet is fairly readable – i.e., it is clear precisely *how* specific characters are being checked.

Form	Type	Operator	Description
Literal	Number		Basic numeric literal (`123`)
	String		Empty or non-empty string literal (`""` or `"hello"`)
Variable	Number		Variable name only (`x`).
	String		Variable name only (`x`).
Variable-Literal	Number	Sum	Variable name and numeric literal (`x + 1`).
	String	Append	Variable name and string literal (`x + "hello"`).
Variable-Variable	Number	Sum	Variable names only (`x + y`).
	String	Append	Variable names only (`x + y`).

Table 10: *Expression forms in Nibbles. Used in Print statements, Variable Assignments, etc.*

Category	Name	Description
Line Group	line-group-indent	If lines A and B in same group, then B is either in-line with A or indented.
	all-lines-active	Every line must be active in exactly 1 group, in line order.
	line-group → lines	Verifies a line group type based on 1st and 2nd active lines.
Line	line → tokens	Verifies a line type based on the first few tokens. The Variable Assignment, Print, If Line, and Return line types have their expression forms checked as well.
	line-whitespace-at-end	If a line contains blank and non-blank tokens, all Whitespace/Blank tokens must be at the end of the line.
Token	token → chars	Verifies a token type based on its characters. Anything not entirely whitespace and not any other token type is considered Text.

Table 11: *Constraints in the Nibbles model.*

```
(cond
  ((eq token-type 'keyword-type)
   (assert!
    (or
     ;; print keyword
     (and
      (equalv keyword-kw 'print)
      (equalv (nth 0 char-values) #\p)
      (equalv (nth 1 char-values) #\r)
      (equalv (nth 2 char-values) #\i)
      (equalv (nth 3 char-values) #\n)
      (equalv (nth 4 char-values) #\t)
      (equalv (v@ (token) length) 5)
      (rest-chars-are-whitespace char-values 5))
     ;; ...
))))
```

Listing 2: *Snippet from the "token → chars" constraint that verifies a* `print` *keyword.*

4.5 Results

Brief introduction

4.5.1 counting - twospaces

Line grouping will favor cohesive groups without further information. Does this mean programmers will also report incorrect indentations for second print line?

Whitespace primes wrong grouping Lack of indent info fails to cull wrong solution (last print is in its own line group)

Predictions

Moving print up 2 lines will reduce errors (yes) Participants who make errors will recall last line without indent (?) Adding an ”end” token below final print will reduce errors (?)

Open Questions

Does changing ”Done counting” make a difference? Does adding a nested for look help?

```
for i in [1, 2, 3, 4]:
    print "The count is", i


    print "Done counting"
```

Listing 3: *Code from the `counting - twospaces` program in the eyeCode experiment.*

4.5.2 overload - plusmixed

Priming occurs when integer/plus choice points are moved to front. A weaker constraint for identifying variable types may also be the case (has digit versus only digits).

Priming re-orders (number/string, plus/append) choice points Weak number/string constraint fails to cull wrong solutions

Predictions

Priming with append + will reduce errors (maybe) Not including numeric characters will reduce errors (?) Adding type tokens or making strings and numbers visually distinct will reduce errors (?)

Open Questions

How does this help determine ”cognitively optimal” operator overloading?

```
a = 4
b = 3
print a + b

c = 7
d = 2
print c + d

e = "5"
f = "3"
print e + f
```

Listing 4: *Code from the* `overload - plusmixed` *program in the eyeCode experiment.*

4.5.3 scope - diffname

Expectations re-order choice points (a function must ”do something”) Incomplete parameter passing constraints allow wrong solution Right solution is buried

Predictions

Experience will reduce errors (yes) Parameter name will not influence errors (yes) Reminding participants of Python pass-by-value will reduce errors (?)

Transactions - action, location, object (target). Is location or object mistaken because of strong top-level preference to a function to have an “observable” action?

Look at transactions in [21]. Mentioned in Rist ([26]).

> From Code Fragment to Single Line - The creation of just a single fine of a code requires a great deal of reasoning and planning. Mayer (1987) described the conceptual structure underlying a line of BASIC code and found that the smallest piece of knowledge used in program understanding is a transaction. A transaction can be described using three categories: what operation takes place (action), where it takes place (location) and what object is acted upon (object). As an example, IO transactions are required to understand the line of code, LET B = A f 1, which simply adds one to an integer (A) and stores the resulting sum in memory (in B). First, the integer and the increment must be defined and stored in a temporary memory (4 steps), and added together (1 step). Then the location of the sum must be defined, the sum placed in that location and deleted from the temporary memory (3 steps). Finally, control must be transferred to the next statement and that statement executed (2 steps).

```
def add_1(num):
    num = num + 1

def twice(num):
    num = num * 2

added = 4
add_1(added)
twice(added)
add_1(added)
twice(added)
print added
```

Listing 5: *Code from the* `scope - diffname` *program in the eyeCode experiment.*

4.6 Discussion

4.6.1 Model Capabilities

Errors First solution assumed to be correct (other orderings?) Need back-tracking on null set

Attention Full CDO space is infeasible to search entirely Combinatorics are carefully controlled via detail, assertions

Abduction of Missing Code Bottom-up sensing + top-down knowledge A kind of "cogent confabulation"? (Hecht-Nielson, 2007)

4.6.2 Limitations and Threats to Validity

Incomplete Need to finish Example 3 Doesn't generate goals for end-to-end model No back-tracking when errors are noticed

Decision procedure First solution assumed to be correct Simple priming from choice point ordering

Engineered Knowledge Domain structure and constraints are hand-built Few experiments to draw from

5 Conclusion

Compare and contrast formalisms

5.1 Future Work

5.1.1 Improving Mr. Bits

- Compare to complexity metrics (e.g., [11]).
- More experiments

5.1.2 Nibbles 2.0

- Human experiments for decision procedure
- High-level schemas, ontology
- Beyond output prediction task
- Forgetting, fall-back strategies

5.1.3 Combining Mr. Bits and Nibbles

- End-to-end model

6 Acknowledgements

Grant R305A1100060 from the Institute of Education Sciences Department of Education and grant 0910218 from the National Science Foundation REESE supported this research.

A Appendix - Mr. Bits Code

```
;; ============================================================================
;;                                  MR. BITS
;; ============================================================================

;; Maximum number of seconds the model can run
(defparameter *max-time* 500)

;; ----------------------------------------------------------------------------
;; PROGRAM (funcall - space)
;; ----------------------------------------------------------------------------
;; 1. def f(x):
;; 2.     return x + 4
;; 3.
;; 4. print f(1) * f(0) * f(-1)
;; ----------------------------------------------------------------------------


; ----------------------------------------------------------------------------
; GOAL STACK
; ----------------------------------------------------------------------------

;; Fixed goal stack for model
(defparameter *goal-stack* '(
    go-to-line-1
    remember-line
    go-to-line-4
    link-to-value-f-x-L4-C8-0-1
    go-to-line-2
    do-read-line-2-f-x-1
    compute-sum-line-2-1-1
    remember-line
    go-to-line-4
    remember-line
    link-to-value-f-x-L4-C15-1-1
    go-to-line-2
    do-read-line-2-f-x-2
    compute-sum-line-2-2-1
    remember-line
    go-to-line-4
```

```
    remember-line
    link-to-value-f-x-L4-C22-2-1
    go-to-line-2
    do-read-line-2-f-x-3
    compute-sum-line-2-3-1
    remember-line
    go-to-line-4
    compute-prod-line-4-1-1
    compute-prod-line-4-2-1
    remember-line
    fixate-output-box
    type-response
    return-from-output-box
))

;; List of responses that will be typed
(defparameter *responses* (list
    (format nil "60~%")
))

;; List of tokens to put in visicon
(defparameter *tokens* `(
    (0 0 33 23 "def")
    (44 0 55 23 "f")
    (55 0 55 23 "(x):")
    (44 23 66 23 "return")
    (121 23 11 23 "x")
    (143 23 11 23 "+")
    (165 23 11 23 "4")
    (0 69 55 23 "print")
    (66 69 44 23 "f")
    (77 69 44 23 "(1)")
    (121 69 11 23 "*")
    (143 69 44 23 "f")
    (154 69 44 23 "(0)")
    (198 69 11 23 "*")
    (220 69 55 23 "f")
    (231 69 55 23 "(-1)")
))

; -----------------------------------------------------------------------------
```

```
81
82  ;; Convert text character to a key ACT-R can type
83  (defun actr-key (c)
84    (cond
85      ((eq c #\newline) 'return)
86      ((eq c #\space) 'space)
87      ((eq c #\,) 'comma)
88      (t (string c))))
89
90  ;; Get the visicon y coordinate from a line number (1-based)
91  (defun line-to-y (line)
92    (let ((y-pos (* 23 (- line 1)))
93          (line-height (max (round 23 .8) (+ 23 1))))
94      (+ y-pos (round line-height 2))))
95
96  ;; Get the visicon x coordinate from a column number (0-based)
97  (defun col-to-x (col)
98    (* col 11))
99
100 (defmethod rpm-window-key-event-handler ((win rpm-window) key)
101   (proc-display))
102
103 ;; Evalate the goal stack and type responses
104 (defun do-trial ()
105   (reset)
106
107   ; Consider underscores, commas, parens, colons, and brackets as
108   ; word separating characters.
109   (add-word-characters #\_ #\, #\( #\) #\: #\[ #\])
110
111   ; Create an experiment window
112   (let* ((tokens *tokens*)
113          (window (open-exp-window "eyeCode Trial" :width 1600 :height 500)))
114
115     ; Add all tokens to the visicon
116     (loop for (x y width height text)
117           (integer integer integer integer string)
118           in tokens
119           do (let ((static-text
120                     (add-text-to-exp-window :text text :x x :y y
121                                             :width width :height height)))
```

```
                        ; Use actual width and height of the token text
                        (setf (text-height static-text) 23)
                        (setf (str-width-fct static-text)
                                (lambda (str) (* 11 (length str))))))

    ; Add the continue button (ends the trial)
    (add-button-to-exp-window
      :text "Continue" :x 1115 :y 370
      :width 75 :height 25)

    ; Go, Johnny, Go, Go, Go
    (progn
      (install-device window)
      (proc-display)
      (print-visicon)
      (run *max-time*))))

; ------------------------------------------------------------------------

(clear-all)

(define-model mr-bits
  ; Fixed random seed for reproducibility
  (sgp :seed (123456 0))

  ; Set ACT-R parameters
  (sgp :v t :needs-mouse nil :show-focus t :esc nil :lf 0.01 :trace-detail high
       :imaginal-delay 0 :act t :bll nil :rt -2
       :motor-feature-prep-time 0.001
       :motor-initiation-time 0.05
       :motor-burst-time 0.001
    )

  ; ------------------------------------------------------------------------
  ; DECLARATIVE MEMORY
  ; ------------------------------------------------------------------------

  ; Define chunk types
  (chunk-type model-state state token line-num response resp-idx resp-char
    last-x last-y trace-line trace-col trace-ctx trace-name trace-call trace-val
    trace-def orig-def last-ctx)
```

```
163
164 ; Details of a line of code. A missing chunk for a line forces all tokens
165 ; to be read.
166 (chunk-type line-info line-num)
167
168 ; Value or reference for a variable
169 (chunk-type variable-info
170             context      ; function name or "" for global
171             name         ; variable name
172             call-idx     ; index of current function call
173             def-num      ; index of variable definition in context
174             has-value    ; :yes if the variable's value is in memory, :no otherwise
175             ref-context  ; :empty or context of source variable
176             ref-name     ; :empty or name of source variable
177             ref-call     ; :empty or function call index of source variable
178             ref-def      ; :empty or index of source variable definition
179             val-line     ; :empty or line number of literal value
180             val-col      ; :empty or column offset of literal value
181 )
182
183 ; Knowledge of binary sums, differences, products, etc.
184 (chunk-type sum-result first second)
185 (chunk-type diff-result first second)
186 (chunk-type prod-result first second)
187 (chunk-type and-result first second)
188 (chunk-type less-than-result first second)
189 (chunk-type greater-than-result first second)
190
191 (chunk-type (trace-location (:include visual-location)) context name trace-line trace-col)
192
193 ; Add sums from -10 to 10
194 (add-dm-fct
195   (loop for i from -10 to 10
196         append (loop for j from -10 to 10
197                      collect (list (intern (string-upcase (format nil "sum-~a-~a" i j)))
198                                    'isa 'sum-result
199                                    'first i
200                                    'second j))))
201
202 ; Add subtractions from 0 to 10
203 (add-dm-fct
```

```
  (loop for i from 0 to 10
        append (loop for j from 0 to 10
                     collect (list (intern (string-upcase (format nil "diff-~a-~a" i j)))
                                   'isa 'diff-result
                                   'first i
                                   'second j))))

; Add products from -10 to 20
(add-dm-fct
  (loop for i from -10 to 20
        append (loop for j from -10 to 20
                     collect (list (intern (string-upcase (format nil "prod-~a-~a" i j)))
                                   'isa 'prod-result
                                   'first i
                                   'second j))))

; Add less-than comparisons from -10 to 10
(add-dm-fct
  (loop for i from -10 to 10
        append (loop for j from -10 to 10
                     collect (list (intern (string-upcase (format nil "less-than-~a-~a" i j)))
                                   'isa 'less-than-result
                                   'first i
                                   'second j))))

; Add greater-than comparisons from -10 to 10
(add-dm-fct
  (loop for i from -10 to 10
        append (loop for j from -10 to 10
                     collect (list (intern (string-upcase (format nil "greater-than-~a-~a" i j)))
                                   'isa 'greater-than-result
                                   'first i
                                   'second j))))

; Add binary ANDs
(add-dm-fct
  (loop for a in '(t nil)
        for x = (if a 'True 'False)
        append (loop for b in '(t nil)
                     for y = (if b 'True 'False)
                     collect (list (intern (string-upcase (format nil "and-~a-~a" x y)))
```

```
                                    'isa 'and-result
                                    'first x
                                    'second y))))

; Add visual locations of variables
(add-dm-fct
  (list
    `(trace-value-4-8
          isa trace-location screen-x ,(col-to-x 8) screen-y ,(line-to-y 4)
          trace-line 4 trace-col 8)
    `(trace-value-4-15
          isa trace-location screen-x ,(col-to-x 15) screen-y ,(line-to-y 4)
          trace-line 4 trace-col 15)
    `(trace-value-4-22
          isa trace-location screen-x ,(col-to-x 22) screen-y ,(line-to-y 4)
          trace-line 4 trace-col 22)
  ))

(add-dm
  (state isa chunk)

  ; Add chunks for static model states
  (start isa chunk)
  (finish-exp isa chunk)

  (find-next-token isa chunk)
  (search-for-token isa chunk)
  (attend-to-token isa chunk)
  (read-or-remember-line isa chunk)
  (retrieve-line-result isa chunk)

  (check-goal-stack isa chunk)

  ; Location of output box
  (output-box-location isa visual-location
    screen-x 1115 screen-y 80
    width 340 height 270
    color white)

  ; Add chunks for dynamic model states
  (link-to-value-f-x-L4-C8-0-1 isa chunk)
```

```
    (compute-prod-line-4-1-1 isa chunk)
    (compute-sum-line-2-2-1 isa chunk)
    (compute-prod-line-4-2-1 isa chunk)
    (compute-sum-line-2-1-1 isa chunk)
    (compute-sum-line-2-3-1 isa chunk)
    (do-read-line-2-f-x-3 isa chunk)
    (do-read-line-2-f-x-2 isa chunk)
    (do-read-line-2-f-x-1 isa chunk)
    (link-to-value-f-x-L4-C15-1-1 isa chunk)
    (link-to-value-f-x-L4-C22-2-1 isa chunk)

    ; Create initial goal
    (goal isa model-state state check-goal-stack line-num 1))

; ------------------------------------------------------------------------
; PRODUCTIONS
; ------------------------------------------------------------------------

; Try to remember details of line from memory
(p read-or-remember-line
   =goal>
   isa          model-state
   state        read-or-remember-line
   line-num     =line-num
   ==>
   +retrieval>
   isa          line-info
   line-num     =line-num
   =goal>
   state        retrieve-line-result

   !output!     (retrieve line =line-num)
   )

; Line details remembered, continue on without reading
(p retrieve-line-success
   =goal>
   isa          model-state
   state        retrieve-line-result
   =retrieval>
   isa          line-info
```

```
      ==>
      =goal>
      state        check-goal-stack
      )

   ; Unable to remember line, read each token before continuing
   (p retrieve-line-failed
      =goal>
      isa          model-state
      state        retrieve-line-result
      line-num     =line-num
      ?retrieval>
      state        error
      ==>
      =goal>
      state        find-next-token
      !output!     (Reading line =line-num)
      )

   ; ----------------------------------------------------------------------

   ; Shift visual attention to the next visual token
   (p shift-attention-to-line
      =goal>
      isa          model-state
      state        shift-attention-to-line
      =visual-location>
      isa          visual-location
      screen-x     =x
      screen-y     =y
      ?visual>
      state        free
      ==>
      +visual>
      isa          move-attention
      screen-pos   =visual-location
      =goal>
      state        attend-first-token
      last-x       =x
      last-y       =y
      )
```

```
368
369     ; Attend to the first token on a line, try to remember
370     ; the line's details.
371     (p attend-first-token
372        =goal>
373        isa          model-state
374        state        attend-first-token
375        =visual>
376        isa          text
377        value        =token
378        ==>
379        =goal>
380        state        read-or-remember-line
381        token        =token
382        !output!     (read token =token)
383        )
384
385     ; Shift visual attention to the next visual token
386     (p search-for-token
387        =goal>
388        isa          model-state
389        state        search-for-token
390        =visual-location>
391        isa          visual-location
392        screen-x     =x
393        screen-y     =y
394        ?visual>
395        state        free
396        ==>
397        +visual>
398        isa          move-attention
399        screen-pos   =visual-location
400        =goal>
401        state        attend-to-token
402        last-x       =x
403        last-y       =y
404        )
405
406     ; Extract the text from the attended token
407     (p attend-encoding-token
408        =goal>
```

```
      isa           model-state
      state         attend-to-token
      =visual>
      isa           text
      value         =token
      ==>
      =goal>
      state         find-next-token
      token         =token
      !output!      (read token =token)
      )

   ; Look for the next token to the right
   (p find-next-token
      =goal>
      isa           model-state
      state         find-next-token
      ==>
      +visual-location>
      isa           visual-location
      kind          text
      ; Don't attend to the continue button
      < screen-x    1115
      > screen-x    current
      screen-x      lowest
      screen-y      current
      =goal>
      state         search-for-token
      token         nil
      )

   ; Finished reading the current line, record the details
   ; in memory for later.
   (p no-more-tokens-on-line
      =goal>
      isa           model-state
      state         search-for-token
      line-num      =line-num
      ?visual-location>
      state         error
      ==>
```

```
450       =goal>
451       state         check-goal-stack
452       !output!      (Done reading line =line-num)
453       )
454
455    ; Commit the details of the current line to memory
456    (p remember-line
457       =goal>
458       isa           model-state
459       state         remember-line
460       line-num      =line-num
461       ?imaginal>
462       state         free
463       ==>
464       +imaginal>
465       isa           line-info
466       line-num      =line-num
467       =goal>
468       state         check-goal-stack
469       !output!      (remembering line =line-num)
470       )
471
472    ; ---------------------------------------------------------------------
473
474    ; Execute the next goal on the stack (done if no goals left)
475    (p check-goal-stack
476       =goal>
477       isa           model-state
478       state         check-goal-stack
479       ==>
480       =goal>
481       state         =state2
482
483       ; Crucial that we force imaginal to clear.
484       ; Otherwise the only thing that will clear it is a new chunk.
485       -imaginal>
486
487       !bind!        =state2 (car *goal-stack*)
488       !eval!        (setf *goal-stack* (cdr *goal-stack*))
489       !output!      (Next goal is =state2)
490       )
```

```
491
492     ; No goals left - experiment is over
493     ; Find the continue button and move hand to the mouse
494     (p find-continue-button
495        =goal>
496        isa          model-state
497        state        nil
498        ?manual>
499        state        free
500        ==>
501        +visual-location>
502        isa          visual-location
503        kind         oval
504        +manual>
505        isa          hand-to-mouse
506        =goal>
507        state        click-continue-1
508        )
509
510     ; Look at the continue button
511     (p click-continue-1
512        =goal>
513        isa          model-state
514        state        click-continue-1
515        =visual-location>
516        isa          visual-location
517        screen-x     =x
518        screen-y     =y
519        ?visual>
520        state        free
521        ?manual>
522        state        free
523        ==>
524        +visual>
525        isa          move-attention
526        screen-pos   =visual-location
527        +manual>
528        isa          move-cursor
529        loc          =visual-location
530        =goal>
531        state        click-continue-2
```

```
   )

; Click the continue button
(p click-continue-2
   =goal>
   isa          model-state
   state        click-continue-2
   ?visual>
   state        free
   ?manual>
   state        free
   ==>
   +manual>
   isa          click-mouse
   =goal>
   state        done
   )


; Report amount of time taken to complete the experiment
(p experiment-is-over
   =goal>
   isa          model-state
   state        done
   ?manual>
   state        free
   ==>
   -goal>

   !bind!       =time (mp-time-ms)
   !output!     (Trial completed at =time)
   )

; ------------------------------------------------------------------------

; Prepare to type a sequence of characters
(p start-typing-response
   =goal>
   isa          model-state
   state        type-response
   ==>
```

```
573       =goal>
574          state        typing-response
575          response     =response
576          resp-idx     0
577          resp-char    =resp-char
578
579          !bind!       =response (car *responses*)
580          !bind!       =resp-char (actr-key (aref =response 0))
581          !eval!       (setf *responses* (cdr *responses*))
582          !output!     (Typing =response)
583          )
584
585       ; Type the next character
586       (p typing-response
587          =goal>
588          isa          model-state
589          state        typing-response
590          response     =response
591          resp-idx     =resp-idx
592          - resp-char   nil
593          resp-char    =resp-char
594          ?manual>
595          state        free
596          ==>
597          +manual>
598          isa          press-key
599          key          =resp-char
600          =goal>
601          state        typing-response
602          resp-idx     =next-idx
603          resp-char    =next-char
604
605          !bind!       =next-idx (+ 1 =resp-idx)
606          !bind!       =next-char (if (< =next-idx (length =response)) (actr-key (aref =response =next-idx))
607          !bind!       =time (mp-time-ms)
608          !output!     (Typed =resp-char at =time)
609          )
610
611       ; No more characters to type
612       (p done-typing-response
613          =goal>
```

```
      isa           model-state
      state         typing-response
      resp-char     nil
      ==>
      =goal>
      state         check-goal-stack
      )

   ; Look at the output box
   (p fixate-output-box
      =goal>
      isa           model-state
      state         fixate-output-box
      ?visual>
      state         free
      ==>
      +visual>
      isa           move-attention
      screen-pos    output-box-location
      =goal>
      state         fixating-output-box
      )

   ; Wait for output box to be fixated
   (p output-box-fixated
      =goal>
      isa           model-state
      state         fixating-output-box
      ?visual>
      state         free
      ==>
      =goal>
      state         check-goal-stack
      )

   ; Find the visual location that was being
   ; fixated before the output box.
   (p return-from-output-box
      =goal>
      isa           model-state
      state         return-from-output-box
```

```
      last-x       =last-x
      last-y       =last-y
      ==>
      +visual-location>
      isa          visual-location
      screen-x     =last-x
      screen-y     =last-y
      =goal>
      state        return-from-output-box-move

      !output!     (Returning to =last-x =last-y)
      )

   ; Move eyes to the previous location
   (p return-from-output-box-move
      =goal>
      isa          model-state
      state        return-from-output-box-move
      =visual-location>
      isa          visual-location
      screen-x     =x
      screen-y     =y
      ?visual>
      state        free
      ==>
      +visual>
      isa          move-attention
      screen-pos   =visual-location
      =goal>
      state        return-from-output-box-finish
      )

   ; Check for the next goal
   (p return-from-output-box-finish
      =goal>
      isa          model-state
      state        return-from-output-box-finish
      ?visual>
      state        free
      ==>
      =goal>
```

```
696       state        check-goal-stack
697       )
698
699    ; -----------------------------------------------------------------------
700
701    ;; An explicit value was not found for the variable.
702    ;; Try to find a reference or location in declarative memory.
703    (p recalling-variable-no-value
704       =goal>
705       isa          model-state
706       state        recalling-variable
707       trace-ctx    =trace-ctx
708       trace-name   =trace-name
709       trace-call   =trace-call
710       trace-def    =trace-def
711       ?retrieval>
712       state        error
713       ==>
714       +retrieval>
715       isa          variable-info
716       context      =trace-ctx
717       name         =trace-name
718       call-idx     =trace-call
719       def-num      =trace-def
720       has-value    :no
721       =goal>
722       state        recalling-variable-no-value
723
724       !output!     (Recalling location or reference for =trace-name in context =trace-ctx call =trace-ca
725       )
726
727    ;; No location or reference was found for the variable.
728    ;; Try to find a previous definition in the current context.
729    (p recalling-variable-no-ref-or-loc
730       =goal>
731       isa          model-state
732       state        recalling-variable-no-value
733       trace-ctx    =trace-ctx
734       trace-name   =trace-name
735       trace-call   =trace-call
736       trace-def    =trace-def
```

```
    > trace-def  0
    ?retrieval>
    state        error
    ==>
    +retrieval>
    isa          variable-info
    context      =trace-ctx
    name         =trace-name
    call-idx     =trace-call
    def-num      =next-def-num
    has-value    :yes
    =goal>
    state        recalling-variable
    trace-def    =next-def-num

    ; Previous definition
    !bind!       =next-def-num (- =trace-def 1)
    !output!     (Trying again with =trace-name in context =trace-ctx call =trace-call def =next-def-nu
    )

(p recalling-variable-change-context
    =goal>
    isa          model-state
    state        recalling-variable-no-value
    trace-ctx    =trace-ctx
    trace-name   =trace-name
    trace-ctx    =trace-ctx
    last-ctx     =last-ctx
    <= trace-def  0
    ?retrieval>
    state        error
    ==>
    +retrieval>
    isa          variable-info
    context      =last-ctx
    name         =trace-name
    has-value    :yes
    =goal>
    state        recalling-variable

    !output!     (Changing context for =trace-name from =trace-ctx to =last-ctx)
```

```
778       )
779
780    ;; Got a reference to another variable (possibly in a different context).
781    ;; Try to retrieve a value, reference, or location for *that* variable.
782    (p recalling-variable-got-reference
783       =goal>
784          isa          model-state
785          state        recalling-variable-no-value
786       =retrieval>
787          isa          variable-info
788          has-value    :no
789          ref-context  =ref-context
790          ref-name     =ref-name
791          ref-call     =ref-call
792          ref-def      =ref-def
793          val-line     :empty
794          call-idx     =call-idx
795       ==>
796       +retrieval>
797          isa          variable-info
798          context      =ref-context
799          name         =ref-name
800          call-idx     =ref-call
801          def-num      =ref-def
802       =goal>
803          state        recalling-variable-no-value
804
805       !output!     (Following variable reference to =ref-name in context =ref-context call =ref-call def
806       )
807
808    ;; Got a location (line/column) for a variable's value.
809    ;; Recall the associated visual location, and look there.
810    (p recalling-variable-got-location
811       =goal>
812          isa          model-state
813          state        recalling-variable-no-value
814          trace-val    =trace-val
815       =retrieval>
816          isa          variable-info
817          has-value    :no
818          def-num      =def-num
```

```
      ref-context  :empty
      val-line     =val-line
      val-col      =val-col
      ==>
      +retrieval>
      isa          trace-location
      trace-line   =val-line
      trace-col    =val-col
      =goal>
      state        trace-variable-remember
      trace-line   =val-line
      trace-col    =val-col
      trace-def    =def-num

      ; Slip an extra goal into the stack to write the
      ; variable's value after tracing.
      !eval!       (setf *goal-stack*
                         (append (list 'trace-variable-write) *goal-stack*))

      !output!     (Need to trace to line =val-line col =val-col with write =trace-val)
      )

   ; Remembered the variable's trace location. Find it.
   (p trace-variable-remember
      =goal>
      isa          model-state
      state        trace-variable-remember
      =retrieval>
      isa          trace-location
      trace-line   =trace-line
      trace-col    =trace-col
      ?visual>
      state        free
      ==>
      +visual-location>
      isa          visual-location
      :nearest     =retrieval
      =goal>
      state        trace-variable-move

      !output!     (Preparing trace to line =trace-line col =trace-col)
```

```
860       )
861
862    ; Move the eyes to the trace location
863    (p trace-variable-move
864       =goal>
865       isa          model-state
866       state        trace-variable-move
867       trace-line   =trace-line
868       trace-col    =trace-col
869       =visual-location>
870       isa          visual-location
871       ?visual>
872       state        free
873       ==>
874       +visual>
875       isa          move-attention
876       screen-pos   =visual-location
877       =goal>
878       state        trace-variable-moving
879
880       !output!     (Tracing to line =trace-line col =trace-col)
881       )
882
883    ; Start reading at the current location. Reading
884    ; will proceed to the end of the line, and then
885    ; the goal stack will be checked.
886    (p trace-variable-moving
887       =goal>
888       isa          model-state
889       state        trace-variable-moving
890       trace-line   =trace-line
891       ?visual>
892       state        free
893       ==>
894       =goal>
895       state        find-next-token
896
897       !output!     (Reading line =trace-line)
898       )
899
900    ; Write the value of the traced variable to memory.
```

```
901     ; If it's remembered next time, the trace will be
902     ; avoided.
903     (p trace-variable-write
904        =goal>
905        isa          model-state
906        state        trace-variable-write
907        trace-ctx    =trace-ctx
908        trace-name   =trace-name
909        trace-call   =trace-call
910        trace-line   =trace-line
911        trace-col    =trace-col
912        trace-val    =trace-val
913        orig-def     =orig-def
914        ?imaginal>
915        state        free
916        ==>
917        +imaginal>
918        isa          variable-info
919        context      =trace-ctx
920        name         =trace-name
921        call-idx     =trace-call
922        def-num      =orig-def
923        has-value    =trace-val
924        ref-context  :empty
925        ref-name     :empty
926        ref-call     :empty
927        ref-def      :empty
928        val-line     =trace-line
929        val-col      =trace-col
930        =goal>
931        state        check-goal-stack
932
933        !output!     (Wrote traced variable =trace-ctx =trace-name call =trace-call def =orig-def with valu
934        )
935
936     ; ----------------------------------------------------------------------
937     ; DYNAMIC PRODUCTIONS
938     ; ----------------------------------------------------------------------
939
940
941     ; Shift visual attention to a specific line
```

```
(p go-to-line-1
   =goal>
      isa          model-state
      state        go-to-line-1
      ==>
      +visual-location>
      isa          visual-location
      screen-y     =screen-y
      screen-x     lowest
      =goal>
      state        shift-attention-to-line
      line-num     1

      !bind!       =screen-y (line-to-y 1)
      )


; Shift visual attention to a specific line
(p go-to-line-2
   =goal>
      isa          model-state
      state        go-to-line-2
      ==>
      +visual-location>
      isa          visual-location
      screen-y     =screen-y
      screen-x     lowest
      =goal>
      state        shift-attention-to-line
      line-num     2

      !bind!       =screen-y (line-to-y 2)
      )


; Shift visual attention to a specific line
(p go-to-line-3
   =goal>
      isa          model-state
      state        go-to-line-3
      ==>
```

```
      +visual-location>
      isa           visual-location
      screen-y      =screen-y
      screen-x      lowest
      =goal>
      state         shift-attention-to-line
      line-num      3

      !bind!        =screen-y (line-to-y 3)
      )


   ; Shift visual attention to a specific line
   (p go-to-line-4
      =goal>
      isa           model-state
      state         go-to-line-4
      ==>
      +visual-location>
      isa           visual-location
      screen-y      =screen-y
      screen-x      lowest
      =goal>
      state         shift-attention-to-line
      line-num      4

      !bind!        =screen-y (line-to-y 4)
      )


   ; Write the variable's value to memory by
   ; placing a chunk in the imaginal buffer and
   ; letting it get flushed.
   (p link-to-value-f-x-L4-C8-0-1
      =goal>
      isa           model-state
      state         link-to-value-f-x-L4-C8-0-1
      ?imaginal>
      state         free
      ==>
      +imaginal>
```

```
      isa           variable-info
      context       "f"
      name          "x"
      call-idx      0
      def-num       0
      ref-context   :empty
      ref-name      :empty
      ref-call      :empty
      ref-def       :empty
      val-line      4
      val-col       8
      has-value     :no
      =goal>
      state         check-goal-stack
      !output!      (Wrote variable f x 0)
      )


   ; Try to recall the variable's value from memory.
   ; A retrieval failure may force a trace.
   (p do-read-line-2-f-x-1-retrieve
      =goal>
      isa           model-state
      state         do-read-line-2-f-x-1
      ==>
      +retrieval>
      isa           variable-info
      context       "f"
      name          "x"
      call-idx      0
      def-num       0
      has-value     :yes
      =goal>
      state         recalling-variable
      trace-ctx     "f"
      trace-name    "x"
      trace-call    0
      trace-val     :yes
      trace-def     0
      orig-def      0
      last-ctx      ""
```

```
   !output!     (Recalling variable value for "x" in context "f" call 0 def 0)
   )

; Successfully recalled the variable's value.
; Proceed with the next goal.
(p do-read-line-2-f-x-1-success
   =goal>
   isa          model-state
   state        recalling-variable
   =retrieval>
   isa          variable-info
   has-value    :yes
   ==>
   =goal>
   state        check-goal-stack
   )


; Recall the answer for a sum, product, etc.
; This should never fail.
(p compute-sum-line-2-1-1
   =goal>
   isa          model-state
   state        compute-sum-line-2-1-1
   ==>
   +retrieval>
   isa          sum-result
   first        1
   second       4
   =goal>
   state        compute-sum-line-2-1-1-done
   !output!     (Computing sum of 1 4)
   )

; Successfully remembered the answer.
; Proceed with the next goal.
(p compute-sum-line-2-1-1-done
   =goal>
   isa          model-state
   state        compute-sum-line-2-1-1-done
```

```
      =retrieval>
      isa           sum-result
      first         1
      second        4
      ==>
      =goal>
      state         check-goal-stack
      )


   ; Write the variable's value to memory by
   ; placing a chunk in the imaginal buffer and
   ; letting it get flushed.
   (p link-to-value-f-x-L4-C15-1-1
      =goal>
      isa           model-state
      state         link-to-value-f-x-L4-C15-1-1
      ?imaginal>
      state         free
      ==>
      +imaginal>
      isa           variable-info
      context       "f"
      name          "x"
      call-idx      1
      def-num       0
      ref-context   :empty
      ref-name      :empty
      ref-call      :empty
      ref-def       :empty
      val-line      4
      val-col       15
      has-value     :no
      =goal>
      state         check-goal-stack
      !output!      (Wrote variable f x 1)
      )


   ; Try to recall the variable's value from memory.
   ; A retrieval failure may force a trace.
```

```
(p do-read-line-2-f-x-2-retrieve
   =goal>
      isa          model-state
      state        do-read-line-2-f-x-2
      ==>
      +retrieval>
      isa          variable-info
      context      "f"
      name         "x"
      call-idx     1
      def-num      0
      has-value    :yes
      =goal>
      state        recalling-variable
      trace-ctx    "f"
      trace-name   "x"
      trace-call   1
      trace-val    :yes
      trace-def    0
      orig-def     0
      last-ctx     ""

      !output!     (Recalling variable value for "x" in context "f" call 1 def 0)
      )

; Successfully recalled the variable's value.
; Proceed with the next goal.
(p do-read-line-2-f-x-2-success
   =goal>
      isa          model-state
      state        recalling-variable
      =retrieval>
      isa          variable-info
      has-value    :yes
      ==>
      =goal>
      state        check-goal-stack
      )


; Recall the answer for a sum, product, etc.
```

```
; This should never fail.
(p compute-sum-line-2-2-1
   =goal>
      isa          model-state
      state        compute-sum-line-2-2-1
      ==>
   +retrieval>
      isa          sum-result
      first        0
      second       4
      =goal>
      state        compute-sum-line-2-2-1-done
      !output!     (Computing sum of 0 4)
      )

; Successfully remembered the answer.
; Proceed with the next goal.
(p compute-sum-line-2-2-1-done
   =goal>
      isa          model-state
      state        compute-sum-line-2-2-1-done
      =retrieval>
      isa          sum-result
      first        0
      second       4
      ==>
      =goal>
      state        check-goal-stack
      )


; Write the variable's value to memory by
; placing a chunk in the imaginal buffer and
; letting it get flushed.
(p link-to-value-f-x-L4-C22-2-1
   =goal>
      isa          model-state
      state        link-to-value-f-x-L4-C22-2-1
      ?imaginal>
      state        free
      ==>
```

```
      +imaginal>
      isa           variable-info
      context       "f"
      name          "x"
      call-idx      2
      def-num       0
      ref-context   :empty
      ref-name      :empty
      ref-call      :empty
      ref-def       :empty
      val-line      4
      val-col       22
      has-value     :no
      =goal>
      state         check-goal-stack
      !output!      (Wrote variable f x 2)
      )


   ; Try to recall the variable's value from memory.
   ; A retrieval failure may force a trace.
   (p do-read-line-2-f-x-3-retrieve
      =goal>
      isa           model-state
      state         do-read-line-2-f-x-3
      ==>
      +retrieval>
      isa           variable-info
      context       "f"
      name          "x"
      call-idx      2
      def-num       0
      has-value     :yes
      =goal>
      state         recalling-variable
      trace-ctx     "f"
      trace-name    "x"
      trace-call    2
      trace-val     :yes
      trace-def     0
      orig-def      0
```

```
      last-ctx     ""

      !output!     (Recalling variable value for "x" in context "f" call 2 def 0)
      )

   ; Successfully recalled the variable's value.
   ; Proceed with the next goal.
   (p do-read-line-2-f-x-3-success
      =goal>
      isa          model-state
      state        recalling-variable
      =retrieval>
      isa          variable-info
      has-value    :yes
      ==>
      =goal>
      state        check-goal-stack
      )


   ; Recall the answer for a sum, product, etc.
   ; This should never fail.
   (p compute-sum-line-2-3-1
      =goal>
      isa          model-state
      state        compute-sum-line-2-3-1
      ==>
      +retrieval>
      isa          sum-result
      first        -1
      second       4
      =goal>
      state        compute-sum-line-2-3-1-done
      !output!     (Computing sum of -1 4)
      )

   ; Successfully remembered the answer.
   ; Proceed with the next goal.
   (p compute-sum-line-2-3-1-done
      =goal>
      isa          model-state
```

```
      state        compute-sum-line-2-3-1-done
      =retrieval>
      isa          sum-result
      first        -1
      second       4
      ==>
      =goal>
      state        check-goal-stack
      )


   ; Recall the answer for a sum, product, etc.
   ; This should never fail.
   (p compute-prod-line-4-1-1
      =goal>
      isa          model-state
      state        compute-prod-line-4-1-1
      ==>
      +retrieval>
      isa          prod-result
      first        5
      second       4
      =goal>
      state        compute-prod-line-4-1-1-done
      !output!     (Computing prod of 5 4)
      )

   ; Successfully remembered the answer.
   ; Proceed with the next goal.
   (p compute-prod-line-4-1-1-done
      =goal>
      isa          model-state
      state        compute-prod-line-4-1-1-done
      =retrieval>
      isa          prod-result
      first        5
      second       4
      ==>
      =goal>
      state        check-goal-stack
      )
```

```
1352
1353
1354    ; Recall the answer for a sum, product, etc.
1355    ; This should never fail.
1356    (p compute-prod-line-4-2-1
1357       =goal>
1358       isa          model-state
1359       state        compute-prod-line-4-2-1
1360       ==>
1361       +retrieval>
1362       isa          prod-result
1363       first        20
1364       second       3
1365       =goal>
1366       state        compute-prod-line-4-2-1-done
1367       !output!     (Computing prod of 20 3)
1368       )
1369
1370    ; Successfully remembered the answer.
1371    ; Proceed with the next goal.
1372    (p compute-prod-line-4-2-1-done
1373       =goal>
1374       isa          model-state
1375       state        compute-prod-line-4-2-1-done
1376       =retrieval>
1377       isa          prod-result
1378       first        20
1379       second       3
1380       ==>
1381       =goal>
1382       state        check-goal-stack
1383       )
1384
1385
1386    ; ----------------------------------------------------------------------------
1387
1388    ; All pre-existing DM facts (sums, etc.) are assumed to be well rehearsed
1389    (set-all-base-levels 100000 -1000)
1390    (goal-focus goal)
1391 )
```

B Appendix - Computer Example

```
;;; Author: Michael Hansen
;;; Created on 2015-03-06 14:46:21.715348

(in-package :cdo)

;;; Constraints added by default
(defparameter *computer-configuration-default-constraints* '())

;;; Instance parameters
(defparameter *components-n* 8)
(defparameter *memory-type-boost* 5)

;;; ---------------------------------------------------------------------------

(defun print-component (comp)
  (let* ((performance (v@ (comp) performance))
         (comp-type (name^ (e@ comp component-type)))
         (vendor (name^ (e@ comp component-details vendor vendor-choice)))
         (cost (v@ (comp component-details product-info) cost))
         (performance (v@ (comp component-details product-info) performance))
         (model (value-of (v@ (comp component-details product-info) model)))
         (used (name^ (e@ comp component-role)))
         (mem-type (when (equale (e@ comp component-type) memory)
                     (name^ (e@ comp component-type memory memory-type))))
         )
    (with-output-to-string
        (str)
      (format str "~a [~a] ~a ~a~a (c:~a, p:~a)"
              (if (eq used 'used) "*" " ")
              comp-type vendor
              (if (ground? model) model "Unknown Model")
              (cond
                ((eq mem-type 'type-a) ", A")
                ((eq mem-type 'type-b) ", B")
                (t ""))
              (if (ground? cost) cost "?")
              (if (ground? performance) performance "?"))
      )
    ))
```

```
(defun print-configuration (cfg)
  (let* ((cost (v@ (cfg) cost))
         (performance (v@ (cfg) performance))
         (components (entities^ (n@ cfg components)))
         )
    (with-output-to-string
        (str)
      (format str "Configuration (cost:~a, perf:~a):~%"
              (if (ground? cost) cost "?")
              (if (ground? performance) performance "?"))
      (dolist (comp components)
        (format str "    ~a~%" (print-component comp))
        )
      (format str "~%")
      )
  ))

;;; ============================================================================
;;;                                Structure
;;; ============================================================================

;;; Top-level entities
(defun computer-configuration_ ( &rest constraints)
  (multiple-value-bind (local-constraints relayed-constraints)
      (isolate-constraints :computer-configuration  (append constraints *computer-configuration-default
    `(let* (
            (components
             ,(apply #'dm 'components *components-n* #'component_ relayed-constraints))

            (computer-configuration
             (de 'computer-configuration
                 :r ( components  )
                 :v ( ,(dv 'cost (an-integerv))  ,(dv 'performance (an-integerv))   )
                 )
              ))
       ;;
       ,@local-constraints
       ;;
       computer-configuration )))

```

```
(defun component_ (n &rest constraints)
  (multiple-value-bind (local-constraints relayed-constraints)
      (isolate-constraints :component  constraints  n)
    `(let* (
            (component-details
             (da 'component-details
                 (de 'product-info
                     :v ( ,(dv 'cost (an-integerv))  ,(dv 'performance (an-integerv)) ,(dv 'model (a-st
                     )
                 ,(apply #'vendor_ relayed-constraints)
                 ))
            (component-role
             (ds 'component-role
                 (de 'used
                     )
                 (de 'not-used
                     )
                 ))
            (component-type
             (ds 'component-type
                 ,(apply #'memory_ relayed-constraints)
                 (de 'graphics
                     )
                 (de 'sound
                     )
                 ))

            (component
             (de 'component
                 :r ( component-details  component-role  component-type  )
                 )
              ))
       ;;
       ,@local-constraints
       ;;
       component )))

(defun vendor_ ( &rest constraints)
  (multiple-value-bind (local-constraints relayed-constraints)
      (isolate-constraints :vendor  constraints  )
    `(let* (
```

```
122             (vendor-choice
123              (ds 'vendor-choice
124                  (de 'invideo
125                      )
126                  (de 'slamsong
127                      )
128                  (de 'tortoise-bay
129                      )
130                  ))
131
132             (vendor
133              (de 'vendor
134                  :r ( vendor-choice  )
135                  )
136               ))
137        ;;
138        ,@local-constraints
139        ;;
140        vendor )))
141
142 (defun memory_ ( &rest constraints)
143   (multiple-value-bind (local-constraints relayed-constraints)
144       (isolate-constraints :memory  constraints  )
145     `(let* (
146             (memory-type
147              (ds 'memory-type
148                  (de 'type-a
149                      )
150                  (de 'type-b
151                      )))
152
153             (memory
154              (de 'memory
155                  :r ( memory-type  )
156                  )
157               ))
158        ;;
159        ,@local-constraints
160        ;;
161        memory )))
162
```

```
163
164 ;;; ---------------------------------------------------------------------------
165
166 ;;; Function to count solutions
167 (defun computer-configuration-counter_ (&rest constraints)
168   (multiple-value-bind (local-constraints relayed-constraints)
169       (isolate-constraints :computer-configuration constraints)
170     `(let ((count 0))
171        (for-effects
172         (let ((return-value
173                (progn
174                  (let* (
175                         (components
176                          ,(apply #'dm 'components *components-n* #'component_ relayed-constraints))
177
178                         (computer-configuration
179                          (de 'computer-configuration
180                              :r ( components  )
181                              :v ( ,(dv 'cost (an-integerv))  ,(dv 'performance (an-integerv))  )
182                              )
183                           ))
184                    ;;
185                    ,@local-constraints
186                    ;;
187                    computer-configuration ))))
188           return-value
189           (global
190            (setf count (1+ count)))))
191         count)))
192
193 ;;; ===========================================================================
194 ;;;                               Constraints
195 ;;; ===========================================================================
196
197 (define-ma-constraint-fn component-active (comp)
198     (equale (e@ comp component-role) used))
199
200 (define-ma-constraint-fn has-graphics (comp)
201   (andv
202    (equale (e@ comp component-role) used)
203    (equale (e@ comp component-type) graphics)))
```

```
204
205 (define-ma-constraint-fn has-sound (comp)
206   (andv
207    (equale (e@ comp component-role) used)
208    (equale (e@ comp component-type) sound)))
209
210 (define-ma-constraint-fn has-memory (comp)
211   (andv
212    (equale (e@ comp component-role) used)
213    (equale (e@ comp component-type) memory)))
214
215 (define-constraint max-4-active
216     :computer-configuration
217     (at-most-ma 4 component-active (n@ components)))
218
219 (define-constraint only-1-graphics-card
220     :computer-configuration
221     (exactly-ma 1 has-graphics (n@ components)))
222
223 (define-constraint only-1-sound-card
224     :computer-configuration
225     (exactly-ma 1 has-sound (n@ components)))
226
227 (define-constraint 1-or-2-memory
228     :computer-configuration
229     (at-least-ma 1 has-memory (n@ components)))
230
231
232 ;; Constrain product types by vendor
233 (define-constraint vendors-types
234     :component
235   (let* ((vendor-choice (e@ component component-details vendor vendor-choice))
236          (component-type (e@ component component-type))
237          )
238   (cond
239     ((equale vendor-choice invideo) (equale component-type graphics))
240     ((equale vendor-choice tortoise-bay) (equale component-type sound))
241     ((equale vendor-choice slamsong) (orv
242                                       (equale component-type graphics)
243                                       (equale component-type sound)
244                                       (equale component-type memory)))
```

```
245       (t t)
246       )
247     ))
248
249   ;; Sum component costs for a configuration
250   (define-constraint config-cost
251       :computer-configuration
252     (let ((comp-costs (mapcar (lambda (comp)
253                                 (ifv (equale (e@ comp component-role) used)
254                                      (v@ (comp component-details product-info) cost)
255                                      0))
256                               (entities^ (n@ components))))
257           )
258       (equalv (v@ (computer-configuration) cost) (applyv #'+v comp-costs))
259       )
260     )
261
262   ;; Sum component performance for a configuration (add boost for same memory types)
263   (define-constraint config-perf
264       :computer-configuration
265     (let* ((mem-count (applyv #'+v (mapcar (lambda (comp)
266                                             (ifv (andv
267                                                   (equale (e@ comp component-role) used)
268                                                   (equale (e@ comp component-type) memory))
269                                                  1
270                                                  0))
271                                           (entities^ (n@ components)))))
272            (mem-types (remove-duplicates
273                        (remove 'nil
274                                (mapcar (lambda (comp)
275                                          (ifv (andv
276                                                (equale (e@ comp component-role) used)
277                                                (equale (e@ comp component-type) memory))
278                                               (name^ (e@ comp component-type memory memory-type))
279                                               nil))
280                                        (entities^ (n@ components))))))
281            (mem-boost (ifv (andv (>v mem-count 1)
282                                  (eq (length mem-types) 1))
283                            *memory-type-boost*
284                            0))
285            (comp-perfs (mapcar (lambda (comp)
```

```
                                (ifv (equale (e@ comp component-role) used)
                                     (v@ (comp component-details product-info) performance)
                                     0))
                               (entities^ (n@ components))))
        )
    (equalv (v@ (computer-configuration) performance)
            (+v mem-boost (applyv #'+v comp-perfs)))
    )
  )


;; Force zero cost
(define-constraint no-cost
    :computer-configuration
  (equalv (v@ (computer-configuration) cost) 0))


;; Force new graphics card
(define-constraint new-graphics-card
    :component
  (ifv (andv
        (equale (e@ component component-role) active)
        (equale (e@ component component-type) graphics))
       (>v (v@ (component component-details product-info) cost) 0)
       t
       ))

;;; ============================================================================
;;;                                Examples
;;; ============================================================================

;; One solution, no constraints

(print
 (soaCDO-solutions
  (computer-configuration_)
  :one
  :print-fun #'print-configuration))

;; Configuration (cost:?, perf:?):
;;     * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
```

```
;;     * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
;;     * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
;;     * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
;;     * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
;;     * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
;;     * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
;;     * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)

;; ------------------------------------------------------------------------

(defparameter *available-components*
  (list
   ;; New components
   (set-ma-instance-properties
    '(:component) '(1)
    :choices '(
               ((component component-details vendor vendor-choice) invideo)
               ((component component-type) graphics)
               )

    :variables '(
                 (((component component-details product-info) model) "D-Force")
                 (((component component-details product-info) cost) 200)
                 (((component component-details product-info) performance) 10)
                 )
    )

   (set-ma-instance-properties
    '(:component) '(2)
    :choices '(
               ((component component-details vendor vendor-choice) slamsong)
               ((component component-type) memory)
               ((component component-type memory memory-type) type-b)
               )

    :variables '(
                 (((component component-details product-info) model) "DRR9")
                 (((component component-details product-info) cost) 20)
                 (((component component-details product-info) performance) 10)
                 )
    )
```

```
368
369     (set-ma-instance-properties
370      '(:component) '(3)
371      :choices '(
372                 ((component component-details vendor vendor-choice) tortoise-bay)
373                 ((component component-type) sound)
374                 )
375
376      :variables '(
377                   (((component component-details product-info) model) "Waves")
378                   (((component component-details product-info) cost) 50)
379                   (((component component-details product-info) performance) 10)
380                   )
381      )
382
383     (set-ma-instance-properties
384      '(:component) '(4)
385      :choices '(
386                 ((component component-details vendor vendor-choice) slamsong)
387                 ((component component-type) memory)
388                 ((component component-type memory memory-type) type-a)
389                 ;; ((component component-role) used)
390                 )
391
392      :variables '(
393                   (((component component-details product-info) model) "DRR7")
394                   (((component component-details product-info) cost) 10)
395                   (((component component-details product-info) performance) 5)
396                   )
397      )
398
399     (set-ma-instance-properties
400      '(:component) '(5)
401      :choices '(
402                 ((component component-details vendor vendor-choice) invideo)
403                 ((component component-type) graphics)
404                 )
405
406      :variables '(
407                   (((component component-details product-info) model) "B-Force")
408                   (((component component-details product-info) cost) 100)
```

```
                (((component component-details product-info) performance) 7)
                )
 )

;; Existing components
(set-ma-instance-properties
 '(:component) '(6)
 :choices '(
            ((component component-details vendor vendor-choice) slamsong)
            ((component component-type) memory)
            ((component component-type memory memory-type) type-a)
            ;; ((component component-role) used)
            )

 :variables '(
              (((component component-details product-info) model) "DRR7")
              (((component component-details product-info) cost) 0)
              (((component component-details product-info) performance) 5)
              )
 )

(set-ma-instance-properties
 '(:component) '(7)
 :choices '(
            ((component component-details vendor vendor-choice) slamsong)
            ((component component-type) sound)
            )

 :variables '(
              (((component component-details product-info) model) "Puddle")
              (((component component-details product-info) cost) 0)
              (((component component-details product-info) performance) 1)
              )
 )

(set-ma-instance-properties
 '(:component) '(8)
 :choices '(
            ((component component-details vendor vendor-choice) slamsong)
            ((component component-type) graphics)
            )
```

```
450
451     :variables '(
452                  (((component component-details product-info) model) "A-Force")
453                  (((component component-details product-info) cost) 0)
454                  (((component component-details product-info) performance) 2)
455                  )
456     )))
457
458 ;; ---------------------------------------------------------------------------
459
460 ;; One solution, just components
461 (print
462   (soaCDO-solutions
463    (apply #'computer-configuration_
464           (cons config-cost
465                 (cons config-perf
466                       *available-components*)))
467    :one
468    :print-fun #'print-configuration))
469
470 ;; Configuration (cost:380, perf:50):
471 ;;      * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
472 ;;      * [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
473 ;;      * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
474 ;;      * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
475 ;;      * [GRAPHICS] INVIDEO B-Force (c:100, p:7)
476 ;;      * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
477 ;;      * [SOUND] SLAMSONG Puddle (c:0, p:1)
478 ;;      * [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
479
480 ;; ---------------------------------------------------------------------------
481
482 (defparameter *default-constraints*
483   `(
484     ,config-cost
485     ,config-perf
486
487     ,max-4-active
488     ,only-1-graphics-card
489     ,only-1-sound-card
490     ,1-or-2-memory
```

```
491
492       ,vendors-types
493
494       ,@available-components
495       ))
496
497   ;; ---------------------------------------------------------------------------
498
499   ;; One solution, no additonal constraints
500   (print
501     (soaCDO-solutions
502      (apply #'computer-configuration_
503             *default-constraints*)
504      :one
505      :print-fun #'print-configuration))
506
507   ;; Configuration (cost:280, perf:35):
508   ;;      * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
509   ;;      * [MEMORY] SLAMSONG DRR9 B (c:20, p:10)
510   ;;      * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
511   ;;      * [MEMORY] SLAMSONG DRR7 A (c:10, p:5)
512   ;;        [GRAPHICS] INVIDEO B-Force (c:100, p:7)
513   ;;        [MEMORY] SLAMSONG DRR7 A (c:0, p:5)
514   ;;        [SOUND] SLAMSONG Puddle (c:0, p:1)
515   ;;        [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
516
517   ;; ---------------------------------------------------------------------------
518
519   ;; One solution, force no cost
520   (print
521     (soaCDO-solutions
522      (apply #'computer-configuration_
523             (cons no-cost *default-constraints*))
524      :one
525      :print-fun #'print-configuration))
526
527   ;; Configuration (cost:0, perf:8):
528   ;;        [GRAPHICS] INVIDEO D-Force (c:200, p:10)
529   ;;        [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
530   ;;        [SOUND] TORTOISE-BAY Waves (c:50, p:10)
531   ;;        [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
```

```
;;       [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;     * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;     * [SOUND] SLAMSONG Puddle (c:0, p:1)
;;     * [GRAPHICS] SLAMSONG A-Force (c:0, p:2)

;; ------------------------------------------------------------------------

(defun performance-utility (sol)
  (value-of (v@ (sol) performance)))

;; Best performance
(multiple-value-bind (best-solutions best-value util-values)
    (soaCDO-solutions
     (apply #'computer-configuration_
            *default-constraints*)
     :best
     :utility-fun #'performance-utility
     :objective-fun #'>
     :print-fun #'print-configuration)

  (print best-solutions))

;; Configuration (cost:260, perf:35):
;;     * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;       [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;     * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;     * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;       [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;     * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;       [SOUND] SLAMSONG Puddle (c:0, p:1)
;;       [GRAPHICS] SLAMSONG A-Force (c:0, p:2)

;;  Configuration (cost:270, perf:35):
;;     * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;     * [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;     * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;       [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;       [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;     * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;       [SOUND] SLAMSONG Puddle (c:0, p:1)
;;       [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
```

```
573
574 ;;  Configuration (cost:280, perf:35):
575 ;;      * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
576 ;;      * [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
577 ;;      * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
578 ;;      * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
579 ;;        [GRAPHICS] INVIDEO B-Force (c:100, p:7)
580 ;;        [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
581 ;;        [SOUND] SLAMSONG Puddle (c:0, p:1)
582 ;;        [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
583
584 ;; ------------------------------------------------------------------------
585
586 ;; Highest performance (first), lowest cost (second)
587 (defun better-2 (a b)
588   (cond
589     ((eq (first a) (first b)) (< (second a) (second b)))
590     (t (> (first a) (first b)))))
591
592 (defun performance-and-cost-utility (sol)
593   (list
594    (value-of (v@ (sol) performance))
595    (value-of (v@ (sol) cost))))
596
597 ;; Best performance with lowest cost
598 (multiple-value-bind (best-solutions best-value util-values)
599     (soaCDO-solutions
600      (apply #'computer-configuration_
601             *default-constraints*)
602      :best
603      :utility-fun #'performance-and-cost-utility
604      :objective-fun #'better-2
605      :print-fun #'print-configuration)
606
607   (print best-solutions))
608
609 ;; Configuration (cost:260, perf:35):
610 ;;      * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
611 ;;        [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
612 ;;      * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
613 ;;      * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
```

```
614 ;;        [GRAPHICS] INVIDEO B-Force (c:100, p:7)
615 ;;      * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
616 ;;        [SOUND] SLAMSONG Puddle (c:0, p:1)
617 ;;        [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
618
619 ;; ------------------------------------------------------------------------
```

References

[1] ACT-R Research Group. About ACT-R. `http://act-r.psy.cmu.edu/about/`, mar 2012.

[2] John R. Anderson. *How can the human mind occur in the physical universe?*, volume 3. Oxford University Press, USA, 2007.

[3] T.J. Biggerstaff, B.G. Mitbander, and D.E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[4] Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, 7(2):115–156, 2002.

[5] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, pages 1–9. ACM, 2011.

[6] Simon Cant, David Jeffery, and Brian Henderson-Sellers. A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, 37(7):351–362, 1995.

[7] Bob Curtis. Fifteen years of psychology in software engineering: Individual differences and cognitive science. In *Proceedings of the 7th international conference on Software engineering*, pages 97–106. IEEE Press, 1984.

[8] Françoise Détienne. *La compréhension de programmes informatiques par l'expert: un modéle en termes de schémas*. PhD thesis, Université Paris V. Sciences humaines, 1986.

[9] Françoise Détienne and Frank Bott. *Software design–cognitive aspects*. Springer Verlag, 2002.

[10] Christopher Douce. The stores model of code cognition. 2008.

[11] Christopher Douce, Paul J. Layzell, and Jim Buckley. Spatial measures of software complexity. 1999.

[12] Scott A Douglass and Saurabh Mittal. A framework for modeling and simulation of the artificial. In *Ontology, Epistemology, and Teleology for Modeling and Simulation*, pages 271–317. Springer, 2013.

[13] B.D. Ehret. Learning where to look: Location learning in graphical user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, pages 211–218. ACM, 2002.

[14] Paul M Fitts and James R Peterson. Information capacity of discrete motor responses. *Journal of experimental psychology*, 67(2):103, 1964.

[15] DJ Gilmore and TRG Green. The comprehensibility of programming notations. In *Human-Computer Interaction-Interact*, volume 84, pages 461–464, 1985.

[16] Mark Guzdial. From science to engineering. *Commun. ACM*, 54(2):37–39, February 2011.

[17] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[18] Philip N Johnson-Laird. *Mental models*. Number 6. Harvard University Press, 1986.

[19] Sonya E Keene, Dan Gerson, and David A Moon. *Object-oriented programming in Common Lisp: A programmer's guide to CLOS*, volume 8. Addison-Wesley Reading, Massachusetts, 1989.

[20] David E. Kieras and David E. Meyer. An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Hum.-Comput. Interact.*, 12(4):391–438, December 1997.

[21] Richard E Mayer. Cognitive aspects of learning and using a programming language. 1987.

[22] Thomas J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[23] George A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[24] Chris Parnin. A cognitive neuroscience perspective on memory for programming tasks. In *In the Proceedings of the 22nd Annual Meeting of the Psychology of Programming Interest Group (PPIG)*. Citeseer, 2010.

[25] Keith Rayner. Eye movements in reading and information processing: 20 years of research. *Psychological bulletin*, 124(3):372, 1998.

[26] Robert S. Rist. Schema creation in programming. *Cognitive Science*, 13(3):389–414, 1989.

[27] Dario D Salvucci. A model of eye movements and visual attention. In *Proceedings of the International Conference on Cognitive Modeling*, pages 252–259, 2000.

[28] Dario D. Salvucci. Predicting the effects of in-car interface use on driver performance: An integrated model approach. *International Journal of Human-Computer Studies*, 55(1):85–107, 2001.

[29] Dario D. Salvucci and N.A. Taatgen. Threaded cognition: An integrated theory of concurrent multitasking. *Psychological Review*, 115(1):101, 2008.

[30] B. Schneiderman. Interactive interface issues. *Software Psychology: Human Factors in Computer and Information Systems*, pages 216–251, 1980.

[31] S.B. Sheppard, Bob Curtis, P. Milliman, MA Borst, and T. Love. First-year results from a research program on human factors in software engineering. In *Proceedings of the National Computer Conference*, page 1021. IEEE Computer Society, 1979.

[32] Janet Siegmund, André Brechmann, Sven Apel, Christian Kästner, Jörg Liebig, Thomas Leich, and Gunter Saake. Toward measuring program comprehension with functional magnetic resonance imaging. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 24. ACM, 2012.

[33] Jeffrey Mark Siskind and David Allen McAllester. Screamer: A portable efficient implementation of nondeterministic common lisp. *IRCS Technical Reports Series*, page 14, 1993.

[34] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, (5):595–609, 1984.

[35] Guy L Steele. *Common LISP: the language*. Digital press, 1990.

[36] N.A. Taatgen. Dispelling the magic: Towards memory without capacity. *Behavioral and Brain Sciences*, 24(01):147–148, 2001.

[37] N.A. Taatgen, John R. Anderson, et al. Why do children learn to say "broke"? A model of learning the past tense without feedback. *Cognition*, 86(2):123–155, 2004.

[38] L. Weissman. Psychological complexity of computer programs: an experimental methodology. *ACM Sigplan Notices*, 9(6):25–36, 1974.

[39] E.J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14:1357–1365, 1988.

[40] Simon White and D Sleeman. Constraint handling in common lisp. *Department of Computing Science Technical Report AUCS/TR9805, University of Aberdeen, Aberdeen, UK*, 1998.

[41] Susan Wiedenbeck. Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, 25(6):697 – 709, 1986.

[42] Tsunhin John Wong, Edward T Cokely, and Lael J Schooler. An online database of act-r parameters: Towards a transparent community-based approach to model development. In *Proceedings of the Tenth International Conference on Cognitive Modeling, Philadelphia, PA, USA*, pages 282–286. Citeseer, 2010.

[43] Bernard P. Zeigler and Phillip E. Hammonds. *Modeling & Simulation-Based Data Engineering: Introducing Pragmatics into Ontologies for Net-Centric Information Exchange*. Academic Press, 2007.