On the Meaning of Computer Programs

Josh Tenenberg

Computing and Software Systems, University of Washington, Tacoma, 1900 Commerce St, Tacoma WA 98402-3100, jtenenbg@u.washington.edu

Abstract. This paper explores how computer programmers extract meaning from the computer program texts that they read. This issue is examined from the perspective that program reading is governed by a number of *economic* choices, since resources, particularly cognitive resources, are severely constrained. These economic choices are informed by the reader's existing belief set, which includes beliefs pertaining to the overlapping and enclosing social groups to which the program reader, the original programmer, and the program's users belong. Membership within these social groups, which may be as specific as the set of programmers working within a particular organization or as general as the members of a particular nation or cultural group, implies a set of shared knowledge that characterizes membership in the social group. This shared knowledge includes both linguistic and non-linguistic components and is what ultimately provides the interpretative context in which meaning is constructed. This account is distinguished from previous theories of computer program comprehension by its emphasis on the *social* and *economic* perspective, and by its recognition of the similarities between computer program understanding and natural language understanding.

1 Program Readers

Computer programs are sequences of instructions that direct the operation of a computer. Programs are written in a programming language and are interpreted by one or more language translators into machine language and converted into electrical energy so as to query and change the energy state of the underlying computer hardware. Programs in execution can be viewed as carrying out functionality from the perspective of their role within a human and social context, such as word processing, graphical manipulation, and accounting.

If computers were programmed by the gods, perfectly and without cost, then there would be no need for people to read programs. But programs are written by people, and in order to fix errors introduced during software development or to add new functionality desired by software purchasers, programmers must read and understand the program texts written by other programmers.

The original programmer thus writes for two very different audiences – people and computers. This "original programmer" may in fact be a large group of people but for consistency will be referred to in the singular for the balance of the paper. The original programmer adds particular syntactic expressions because of her understanding that the human reader, but not the computer

reader, has beliefs, intentions, goals, desires and preferences. Further, the computer will always (short of hardware failure) read and interpret the instructions in its program while the human reader might decide that reading certain program segments is not worth the cognitive effort. In this paper *program readers* will refer to people and not computers unless otherwise indicated.

The reader undertakes their reading task with a set of existing beliefs and with a set of programming artifacts that includes the program itself and perhaps other documents such as requirements and design documents, internal memos, and technical documentation. The reader performs her actions within a sociocultural embedding in an organization, community and society. Meaning construction involves a change in the belief state of the program reader. The reader can choose from a set of actions in order to alter her belief state. Such actions include purely internal cognitive events such as recall and inference, as well as events that have external components, such as speaking with other programmers or reading program and documentation texts.

In ascribing meaning to the expressions in a program, a reader must determine both how the expressions will affect the underlying computer state, that is, its actual behavior, and how the sequence of computer state changes are related to issues of human concern. Each of these tasks will be examined in turn.

2 Syntax and Semantics: The Traditional View

The *syntax* of a language specifies the set of all legal sentences in the language. Modern programming languages fall into the language class called *deterministic context free languages* (DCFL's). The meta-language for describing DCFL's, the *context-free grammars*, describe the atomic units of the language as well as how these atomic units may be combined via a set of rewrite rules. The context-free grammars are *generative*, in that from a finite set of atomic units and a finite set of recursively specified rewrite rules, an unbounded number of legal programs can be described. The use of context-free grammars to describe programming languages, rather than more expressive meta-languages (such as *phrase-structure* grammars) is an engineered choice, since the described DCFL's balance the need for expressivity against the need for fast, automated translation to the machine language of the underlying computer hardware.

The *semantics* of a programming language, as the term is used in computer science, refers to the way in which the underlying computer state changes as a result of expression execution. This semantics is compositional in that the rewrite rules can have corresponding semantic rules; the semantics of a compound expression is determined by the semantics of its components and the semantics of the composition operations. Because the underlying semantics relates to computer state changes over time, the meta-languages generally used for describing semantics have been a combination of formal state-based dynamic logic and informal natural language. As a result, communicating the semantics of programming languages has led to a higher level of ambiguity and misunder-

standing within groups of program language users than for communicating the syntax of programming languages.

Programming languages also provide mechanisms for the introduction of new linguistic entities. For example, the following defines a new linguistic term `sum` in the Java programming language.

```
double sum( double[] A ) {
    float total = 0;
    for ( int i = 0; i < A.length; i++ )
        total += A[i];
}
```

The central idea behind these language extensions is to enable the definition of new *abstractions*. As Guy Steele writes [19, pp.xv-xvi] "The most important concept in all of computer science is abstraction. ... Abstraction consists in treating something complex as if it were simpler, throwing away detail. In the extreme case, one treats the complex quantity as atomic, unanalyzed, primitive." Programming languages are extended in order that atomic expressions can stand for larger syntactic complexes. In the Java example above, the expression `sum( S )` stands for the sum of the elements in the sequence `S`, i.e., $\sum_{i=0}^{n-1} S[i]$. "Naming is perhaps the most powerful abstraction notion we have, in any language, for it allows any complex to be reduced for linguistic purposes to a primitive atom" [19, pp.xv-xvi]. Named abstractions, such as *subroutines* (as in the Java `sum` example) or *objects*, are supported by all modern programming languages. In this way, programming languages can be extended to arbitrary levels, where complexes at one level become the atomic units at the next higher level through abstraction and naming. Any particular program is thus expressed at a number of different linguistic levels provided by the base language and each of its defined extensions.

3 Programming Languages as Social Constructs

Any particular programmer will belong to a number of overlapping and enclosing programming communities. Acculturation into a community will involve learning the set of linguistic abstractions shared by members of this community along with the associated knowledge that the abstractions stand for. Researchers examining programmer cognition have referred to such shared abstractions as *plans* [18]. Additionally, software practitioners have codified many of these abstractions in framework libraries such as C++'s *Standard Template Library* (STL) [14] and in repositories of micro-architectures called *patterns* [9]. These range from the most general abstractions that transcend programming language differences and are common to most trained programmers, (e.g. the *binary-search* routine), to abstractions common to object oriented programmers (e.g. the *iterator* pattern), to abstractions used by programmer subcultures, e.g. users of Java's Collection classes, users of a program library built by a specific company, or users of a library built for a single project. The individual programmer may even have a number

of abstractions that only they themselves use. As Harold Abelson points out [8, Forward], "Perhaps the whole distinction between program and programming language is a misleading idea, and that future programmers will see themselves not as writing programs in particular, but as creating new languages for each new application."

The names used to describe abstractions are important to human readers but of no consequence to the computer since people are able to transfer semantic knowledge associated with particular names acquired through acculturation in non-programming social settings. For example, the standard meaning of *search* in English is to look for something, and naming a computational abstraction "search" provides the reader with a strong indications of its functionality. Using names that stand for real-world concepts can thus help program readers understand the meaning of programs. But other names without real-world referents can, through social habit and convention, come to stand for particular computational abstractions, such as Lisp's *cdr* or SQL's *clob*. Similarly, terms that do have real-world referents in natural language can have such meanings over-ridden by their use within programmer communities. For example, *push* refers to a computation that places an object on top of a stack, as opposed to "push the box out of the doorway" in everyday usage. The meaning that a reader accords to such expressions will have much more to do with such things as the level of standardization of the the named abstractions, the extent to which the reader has been acculturated into the language community, and the reader's beliefs about the original programmer's acculturation into this language community, rather than to any similarity of meaning between the computational abstraction and real world operations.

This acculturation occurs explicitly through instruction as well as individual study using professional journals, textbooks, and programs written by others. But a significant amount of the acculturation happens through communication, feedback, practice, and observation within the programming setting itself. Programmers code together, look critically at one another's code, engage in online discussion groups, and attend professional meetings, workshops, and conferences. Perhaps much of the success of *pair programming* [21] (one of the central components of Extreme Programming [3]) is due to the rapid acculturation and implicit knowledge transfer that occurs when programmers work in close contact with one another.

4 Real World Models and Shared Knowledge

Syntactic constructs in computer programs refer not only to the programming objects common to programmer communities, such as numbers, lists, and functions, but also to entities in the everyday world, from the employees and payrolls of an accounting system to the paintings and painters of an art museum's inventory system.

To facilitate a reader's understanding, the writer chooses some of the linguistic expressions so as to make explicit the program's function within the real-world

context. For example, a programmer modeling biological phenomena might name some of the computational objects "locus", "genome", and "crossover" in order to establish the real-word context and mapping for these terms.

A reader's interpretation of the linguistic expressions in the program text crucially depends upon the shared knowledge between program writer and reader about the real-world. With respect to natural language understanding, James Allen writes [1, 548]

> shared knowledge ... is the knowledge that both agents know and know that the other knows. Shared knowledge arises from the common background and situation that the agents find themselves in and includes general knowledge about the world (such as how common actions are done, the standard type of hierarchy classifications of objects, general facts about the society we live in, and so on). Agents that know each other or share a common profession will also have considerable shared knowledge as a result of their previous interactions and their education in that field. ... While individual beliefs may play a central role in the content of a conversation, most of the knowledge brought to bear to interpret the other's actions will be shared knowledge.

Programmers are members of various overlapping and enclosing social groups within the larger society – for example, professional organizations, civic clubs, local communities, and national and ethnic cultural groups. These social groups have shared knowledge that is learned as part of the acculturation process within the group. Language that is specific to members of the group stands for the shared background knowledge of its members and provides an efficient means for discussing such knowledge. That is, acculturated users know not only the jargon, colloquialisms and idioms of the social group, but understand the concepts and knowledge underlying the terms, using them appropriately and understanding their appropriate use by others. As De Mauro comments on Wittgenstein's ideas concerning socially shared language [7, 53-4]:

> But in the measure in which you belong to my own community, you have been subjected to a linguistic and cultural training similar to my own and I have valid grounds for supposing that your propositions have a similar meaning for both of us. And the 'hypothesis' which I make when I hear you speak, and which you make speaking to me, is confirmed for both of us by both your and my total behavior.

Each individual may belong to many such "communities", each with its own linguistic and cultural training.

One of the perennial difficulties of software development is that programmers may not be members of the same social groups as the software *users*, the people who will interact directly with the program after it is developed. It is no surprise, then, that determining the software *requirements*, i.e. what the software is intended to do, accounts for a significant proportion of the software development budget and that a large percentage of software errors can be traced to errors

in the requirements (up to almost 50% by some estimates [2]). The process of determining requirements involves a transfer of knowledge from users and clients (those who pay for the software) to programmers. This process is time consuming, error prone, and costly because not only is the sheer quantity of knowledge that programmers must acquire significant, much of this knowledge is implicit and taken-for-granted by the users, acquired by them via the informal, context-embedded processes described above for programmers. There might thus be a vast language and culture gap to bridge between the users – ranging from doctors and accountants to dancers and photographers – and the software developers.

Practices of placing expert users into software development organizations for the duration of a development project – one of the operating principles of Extreme Programming [3] – should enhance knowledge transfer by lowering communication costs and increasing communication bandwidth. What we may see in the future is an increasing movement of personnel in the opposite direction, where software developers join the embedding user organization for the duration of the software lifecycle, exploiting the fact that much of the knowledge about a program's meaning is encoded only in the neurons of the users and programmers.

5 The Cognitive Economics of Meaning Construction

The knowledge content of a message can far exceed the information-theoretic limit imposed by the number of bits used to encode the message, due to the immense amount of extant shared knowledge that a message can activate in the reader's mind. In his writings on cognitive sociology [6], Cicourel uses the term *indexicality* to refer to the aspects of language "that require the attribution of meaning beyond the surface form" since linguistic expressions serve as indexes for mental encodings of previous experiences.

The central economic choice of a writer of natural language text, then, concerns what to explicitly include in the text and what to leave out of the text, i.e., its degree of indexicality. In other words, what knowledge and abstractions can the writer assume that the reader already possesses? The writer trades text size against the risks and costs associated with ambiguity and misunderstanding; short texts are preferred to large texts, all other things being equal, since people are under various selection pressures to manage their own resources efficiently.

As with natural languages, programming languages are also indexical, since meaning construction requires that readers possess both a model of the executing hardware and a model of the embedding social context in which the program executes. Writers of computer programs make similar, though not identical kinds of choices as writers of natural language text. The difference concerns the fact that programs are read by computer as well as human readers. Programmers are constrained to use only those linguistic abstractions for which there exist explicit translations (via the above described extension mechanisms, and/or through the existing interpreters and language translators) to the underlying machine language of the executing hardware. Because programs are written at a number of different levels of description the program writer has considerable latitude in

choosing the set of linguistic abstractions and the associated names within their programs.

There are three important reasons why a program reader makes economic choices when reading and interpreting a program. First, the understanding task itself is known to be computationally intractable [22]. That is, no efficient algorithm exists guaranteeing that meaning can always be correctly discerned. Second, the human brain has particular limitations, for example with respect to memory and processing speed that constrain the manner and rate with which inferences can be made. And third, human tasks are performed within a social and economic environment that limits the expenditure of resources on any particular task.

Not only must the individual program reader efficiently manage their internal cognitive resources, but they must also take account of the external environment in order to estimate costs associated with different knowledge acquisition actions and values associated with possible outcomes of these actions. Examples of external action-cost constraints include the software and hardware systems available to the reader for executing and maintaining the program, the presence of other personnel with expertise related to the task, communication technologies and policies that enable the sharing of data and knowledge, project development practices that provide a documented historical trajectory of the program's evolution, and opportunities for further education and training related to the task at hand. Subjective outcome values, though particular to the reader, will certainly be influenced by such things as her individual value system, beliefs about the institutional and social tolerance for errors, and beliefs about the economic climate in which the organization operates.

As a consequence of these economic constraints, we can conjecture that readers employ understanding processes that allow them to tradeoff the amount of resource that they devote to the understanding task – most importantly *time* – against the level or quality of meaning that they construct. We can take this process to be approximately monotonic, i.e., more resource will in general produce more and better understanding. Without such a process, the reader would have no basis for expending further resources in pursuit of greater understanding.

How much and what kinds of resource a reader devotes to any particular reading episode will depend upon their tradeoff of the perceived costs and benefits as mentioned above associated with their different action choices, and the level of understanding that they believe they possess at different times during the problem solving episode. Empirical studies supporting this conjecture indicate that programmers read only a portion of the program text related to their task rather than the entire program – the so-called *as-needed* reading strategies [13]. Further, we can expect program readers to exploit their shared knowledge. This prediction is consistent with the study described in [12], where programmers used the abstraction structure and names to determine which particular parts of the program to read and which to ignore: "Subjects spent a major part of their time searching for code segments relevant to the modification task and no time understanding parts of the program that were perceived to be of little

or no relevance. ... Subjects hypothesized relevance based on their knowledge about the task domain and programming in general. Subjects used procedure and variable names to infer functionality. ... While looking for code subjects guessed correctly the names of procedures they had not seen." The program reader's economic choices therefore concern determining the level of description at which the program text should be read and the level of understanding that must be achieved in order to carry out the task at hand.

Several studies have attempted to determine if readers traverse program text and construct mental representations of the text by starting at lower levels of program description and moving to more abstract levels, or by movement in the opposite direction, from abstractions to concrete descriptions [15, 16, 4]. The above discussion, however, implies that there is no such fixed strategy; rather, a reader might move in either direction depending upon estimates of the costs and benefits of their action choices at any given time. This is not to say that readers employ a strict decision-theoretic policy, such as that described in [11]. Such a strategy violates the computational constraints mentioned above, since this meta-cognitive activity, i.e., enumerating preferences and evaluating expectations, is itself too costly an activity to perform optimally. Nonetheless, we can expect some type of *minimal*, or *bounded* rationality, as proposed by Cherniak [5] and Simon [17], that enables agents to pursue preferred world states to the extent that they are known, but in an approximate and heuristic fashion. This is consistent with reports by von Mayrhauser and Vans [20] in observing program understanding behavior among experts in large-scale comprehension tasks, which they describe as moving in either the upward or downward direction opportunistically.

6 Conversational Maxims and Cooperative Social Norms

An additional factor that helps a reader gain computational efficiencies is the extent to which the reader believes that the writer intends her text to be understood. Grice [10] argues that hearers in natural language conversation assume that speakers follow cooperative social norms. These norms can be viewed as a set of implicit rules, which Grice called *conversational maxims*. These maxims, as summarized in [1, p566] are:

Maxim of Quality – Do not say things for which you lack evidence.
Maxim of Quantity – Make your contribution as informative as required, but not overly informative.
Maxim of Manner – Avoid obscurity of expression and ambiguity.
Maxim of Relation – What you say should be relevant to the current topic.

With respect to programs, these maxims are generally satisfied in the organizational and social settings in which programs are produced. The Maxim of Quality is met since programs must be translatable and executable in order to provide functionality. The Maxim of Quantity is met when programs are written

at different levels of abstraction, so that readers can balance information content with resource expenditure and the amount of knowledge that the reader brings to the reading task. The Maxim of Manner is met when the original programmer uses public, shared language, such as that codified by standards committees and user groups, instead of private, ad-hoc language that will take the reader longer to decode. And the Maxim of Relation is met when the program writer structures their code into cohesive units, e.g. subroutine libraries, objects, frameworks, plans, and patterns. That is, the goal structure provided by programming plans provides cohesiveness and "topicality" to program text.

Studies by Soloway [18] confirm that readers have strong expectations that program writers follow such relevance constraints, which he termed *discourse rules*, and comprehension was negatively impacted when programs violated these discourse rules. Although program writers are not obliged to follow these maxims, it would nonetheless be surprising for writers to violate the very principles that provide such efficiencies in their natural language communications.

7 Summary

Programs are written so as to be both executable by computers in order to carry out useful work, and to be read by other people who must maintain the programs in order to fix errors and to extend the program's functionality. In order to construct meaning from a program the program reader makes economic choices about her actions where the costs and benefits are influenced not only by cognitive constraints but also by the organizational and social context in which the program-related activities occur. This context affects the costs that the reader assigns to the different actions available to her, as well as to the values associated with the different expected outcomes of performing these actions. Of fundamental importance is the extent to which the reader believes that she shares common knowledge with the program writer, both in programming and application domains. This common knowledge is associated with the different social and language-using groups to which the reader and writer belong. Group-specific language is used to economically index the large quantity of group-specific knowledge that provides the interpretative context for meaning construction. Following cooperative conversational maxims, program writers exploit shared knowledge and language by using the abstraction and naming mechanisms of programming languages to express programs at a variety of different levels. Program readers likewise exploit this shared knowledge and language as well as the cooperative communicative intent of the writer to balance the level of meaning that they construct against the resource constraints under which they operate.

References

[1] James Allen. *Natural language understanding.* Benjamin Cummings, 2nd edition, 1995.

[2] Victor Basili and Barry Perricone. Software errors and complexity: An empirical investigation. *Communications of the Association of Computing Machinery*, 27(1):42–52, 1984.

[3] Kent Beck. *Extreme programming explained.* Addison Wesley, 2000.

[4] Ruven Brooks. Towards a theory of comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[5] Christopher Cherniak. *Minimal rationality.* MIT Press, 1986.

[6] Aaron Cicourel. *Cognitive sociology: Language and meaning in social interaction.* Penguin Education, 1973.

[7] T. De Mauro. *Ludwig Wittgenstein: His place in the development of semantics.* Reidel, D., 1967.

[8] D. Friedman, M. Wand, and C. Haynes. *Essentials of programming languages.* McGraw Hill, 1992.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley, 1994.

[10] H. P. Grice. Logic and conversation. In P. Cole and J. Morgan, editors, *Syntax and Semantics*, volume 3. Speech Acts, pages 41–58. Academic Press, 1975.

[11] R. Jeffrey. *The logic of decision.* McGraw-Hill, 1965.

[12] J. Koenemann and S. Robertson. Expert problem solving strategies for program comprehension. In *ACM Human Factors in Computing Systems CHI'91*, pages 125–130, 1991.

[13] D. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In Soloway and Iyengar, editors, *Empirical studies of programmers.* Ablex publishing corporation, 1986.

[14] D. Musser and A. Saini. *STL tutorial and reference guide.* Addison-Wesley, 1996.

[15] N. Pennington. Comprehension strategies in programming. In Olson, Sheppard, and Soloway, editors, *Empirical studies of programmers, second workshop.* Ablex publishing corporation, 1987.

[16] Teresa Shaft and Iris Vessey. The relevance of application domain knowledge: The case of computer program comprehension. *Information systems research*, 6(3):286–299, September 1995.

[17] H. Simon. *Models of bounded rationality.* MIT Press, 1958.

[18] E. Soloway, B. Adelson, and K. Ehrlich. Knowledge and processes in the comprehension of computer programs. In Chi, Glaser, and Farr, editors, *The nature of expertise.* Erlbaum, 1988.

[19] G. Springer and D. Friedman. *Scheme and the art of programming.* McGraw Hill, 1989.

[20] A. von Mayrhauser and A. Vans. Program comprehension during software maintenance and evolution. *Computer*, pages 44–55, August 1995.

[21] Laurie Williams and Robert Kessler. The effects of "pair-pressure" and "pair-learning" on software engineering education. In *Proceedings of the 13th Conference on Software Engineering Education and Training*, pages 59–65, 2000.

[22] Steve Woods and Qiang Yang. The program understanding problem: Analysis and a heuristic approach. In *Proceedings of the 18th International Conference on Software Engineering (ICSE-96)*, pages 6–15, Berlin, Germany, 1996.