Computational Linguistics Vice Versa

Sebastian Lohmeier

M.Sc. Informatik degree course
Technische Universität Berlin
sl@monochromata.de

Keywords: POP-II.B. program comprehension, POP-III.B. new language, POP-V.A. linguistics and mental models, POP-V.B. eye movements

Abstract

Computational linguists use tools developed by computer scientists to analyse language in ways not practically possible in a manual way. Can this exchange be reversed? How could computer scientists use theories developed by linguists to improve programming? To describe my work, I compiled a number of questions concerning a cognitive linguistics of object-oriented programming.

1. What do I want to know?

I want to know how the structure of source code influences how programmers comprehend it and whether ways of structuring text can inspire the structure of source code in a way that makes source code shorter but not harder to comprehend or re-use.

2. What is programming, then?

Based on my research interest, I regard programming as an activity performed by humans. A theory of programming therefore needs to be psychological. When engaged in the activity of programming, programmers manipulate structures that bear certain resemblances to natural languages which is why they are called programming languages. As a consequence, a theory of programming needs to be psycholinguistic. Linguistic aspects studied in psycholinguistics, especially when related to meaning, target many issues also investigated in philosophy, neuroscience, artificial intelligence and can be subsumed under one of the branches of cognitive linguistics. A theory of programming thus needs to be based on a form of cognitive linguistics. I limit my work to object-oriented general-purpose programming languages. I prefer object-oriented languages for their similarities with schemata used in cognitive linguistics and to limit the complexity of my work. I target a mainstream general-purpose language in contrast to e.g. end-user programming (Pane & Myers, 2006) or interactive fiction (Nelson, 2006) because the programming languages that I use myself are general-purpose programming languages. That does not mean that the empirical basis of end-user programming languages and Inform 7 is less relevant for my work.

Three fundamental aspects of a cognitive linguistics of object-oriented programming will be treated in the following: (1) the cognitive gap between compiler and programmer, (2) the information systems metaphor of the mind that influenced cognitive science and cognitive linguistics and (3) that the compiler enforces its "meaning" of a program.

1. Aspects of the cognitive gap between compiler and programmer have been touched upon a number of times already (Green, 1980; Détienne, 2001, 13f.; Knöll, Gasiunas, & Mezini, 2011, 34) and should be subject to a comprehensive analysis. Here, I will confine my analysis to identifiers used to name variables, classes and methods, however.
   To the compiler, an identifier is a sequence of characters that is declared to refer to either a variable, a type or a method. The choice of characters in the sequence does not affect the operation of the compiler as long as it is distinct from other identifiers subject to certain constraints. From the perspective of a compiler, an identifier fits a definition of proper names occurring in natural language: it refers to

a unique entity that is highlighted within a class of entities by being given a name and the meaning of the name does not (anymore) determine what the name refers to (van Langendonck, 2007, 87ff.). To a programmer, on the other hand, many identifiers are meaningful, i.e. function like common nouns and verbs, because nouns, verbs, or noun phrases like *menu*, *close*, and *EditableDataSource* are often chosen as identifiers.

Because programmers comprehend the meaning of identifiers that is not accessible to a compiler, the structures built by the compiler may not correspond to the mental representations constructed by the programmer. An identifier that functions as a common name to the programmer functions as a proper name to the compiler. Is that a problem? One can imagine situations in which a name of a class is changed such that the names of variables used to store its instances do not correctly hint at the type anymore. In functional programming this problem does not occur because the type of a variable is not declared.

Would it be possible to do the opposite, i.e. to omit the identifier and keep the type name? If there is no declaration that relates an identifier to a type and an initial value, what should be used to refer, then? In texts, unlike source code, proper names are not the only means of reference. Noun phrases are often used to refer, too. Both proper names and noun phrases are used to refer to so-called referents – entities in a mental representation constructed by the reader of the text. A second reference to a referent will make the second referring expression function as anaphor: an expression whose meaning is related to the meaning of another related expression in the prior text. Both proper and common names can function as anaphors, i.e. a proper name can be used to refer to what has been referred to by that proper name before and a common name can be used to refer to what has been referred by using that common name before. This works in object-oriented source code, too: `JMenuItem item = new JMenuItem("Hello"); item.setEnabled(false);` works just as well as `new JMenuItem("Hello"); .JMenuItem.setEnabled(false);` In the latter case no identifier holding a proper name needs to be assigned. Instead, the recurrence of the type name in the constructor invocation and the anaphor (prefixed with a dot) relates the constructor invocation and the anaphor to each other and allows readers and programmers to understand that they refer to the same referent, an instance of `JMenuItem`. I put aside the question of how often reference via type names is applicable. Work on definite descriptions in source code (Knöll et al., 2011) could overcome this issue besides keeping local variables for cases in which an actual proper name is required.

2. While the cognitive gap is relatively wide in the case of the comprehension of identifiers, it is narrow when it comes to the representation of concepts in source code and in cognitive linguistics. Structures and processes during language comprehension are described in a branch of cognitive linguistics (e.g. van Dijk & Kintsch, 1983; Kintsch, 1988; Kintsch, 1998; Schwarz, 1992; Schwarz-Friesel & Consten, 2011) and as part of some psycholinguistic research when aiming to explain experimental results (e.g. Garrod & Terras, 2000). Both forms of research are well suited to have their results transferred to object-oriented programming languages, because these languages use a knowledge representation close to conceptual knowledge and the description of psychological processes operating on conceptual knowledge structures can be compared to how compilers use knowledge encoded in source code. I.e. while the information-processing metaphor of the human mind may at least require supplementation with sub-symbolic activation processes to explain psycholinguistic evidence, the information-processing metaphor of the mind provides a suitable basis for applying cognitive linguistics to object-oriented programming languages.

   Because knowledge structures used in cognitive linguistics and in object-oriented programming share a number of features, not only those anaphors can be implemented in programming languages that are based on recurrence (i.e. identical repetition) of proper or common names. Because sub- and super-class relationships are declared in the source code and are therefore accessible to the compiler, a referent can also be referred to using its supertype. Likewise, the relations between classes that are encoded in return types of methods, field declarations, and declarations of accessor methods can be used to implement so-called associative or indirect anaphors (Garrod

& Terras, 2000; Schwarz-Friesel, 2007) in the compiler (Lohmeier, 2011). Instead of writing `void addOne(ServiceRegistrar registrar) { new RegistrarMenuItem(host, registrar.getServiceID()); }` one could use the indirect anaphor `.ServiceID` to access the ServiceID that is a part of the instance stored in the parameter `registrar` in `void addOne(ServiceRegistrar registrar) { new RegistrarMenuItem(host, .ServiceID); }`. While these forms of anaphors are only the most simple of the more complex forms that are possible, they show that complex forms of reference from natural language can be implemented in programming languages.

These indirect forms of reference also highlight a potential possible effect of reference modelled after linguistic theory: to shorten source code without making it more abstract but at the same time hiding irrelevant information. What information is irrelevant is of course dependent on programmer-related factors, as will be discussed later.

3. It also helps to note that the compiler enforces its "meaning" of a program when a programmer and a compiler create different representations, one reading and the other processing the program. When a compiler is written in a way to permit anaphors in source code that resemble anaphors known from natural language, it may process these anaphors in ways that have been found to adequately model human comprehension of anaphors. If the compiler fails to process these anaphors, it may be that humans would have a hard time or be unable to understand the anaphor written by the programmer. Thus, the compiler may be used to detect and reject undesired use of anaphors, e.g. the anaphor `.JMenuItem` that is referentially ambiguous in the presence of two variables that can hold instances of that type. The compiler therefore needs to be able to limit the input to valid code, like is typically done for programming languages. This is different from natural language, where typically there is no single communication participant who is able to enforce her interpretation of an utterance. As a result, there is no risk of introducing ambiguity to a programming language by adding anaphors, if the compiler detects and rejects referentially ambiguous anaphors.

What is programming, then? Programming is a human activity that can be modelled at a formal and at a psychological level. During programming, programmers read and write source code that is also processed by a compiler. In a psychological model of code reading, information from long-term memory is activated and integrated into a representation of the code read. The distinct parts of the representation have activation levels than increase when they are brought to attention or are read. The activation levels decrease with time. Activation levels also influence how long a word is gazed at. The gaze durations, activation levels, memory access and construction of the mental representation of the text can be modelled with a cognitive model. A formal model in the compiler will be restricted to integrating knowledge from the source code into a representation of the source code and detecting and rejecting referential ambiguity. Even though I consider programming a human activity, I do not wish to study decision making during programming.

3. Will I need my linguist friends?

Although every student of computer science hears of Chomsky and generative grammar, nothing at all is taught about contemporary linguistics in courses on programming languages and compiler construction. The only words on natural language that one may expect in such a course may be on syntactic ambiguity of natural language. There is research on programming languages that incorporates or at least touches upon further ideas from linguistics, though. How is this research related to linguistic theory? Lopes, Dourish, Lorenz, and Lieberherr (2003) list a number of works from linguistics. I am not able to judge how useful these works are for implementing anaphors in programming, but they provide a starting point into some linguistic research. Knöll and Mezini (2006) were obviously inspired by cognitive theories of the mind, but do not cite any such theories and use seemingly custom wording like "ideas" instead of concepts, as well as "implicit" and "explicit" reference that could also be replaced by standard terminology from psychology and linguistics. Knöll et al. (2011) describe the counterparts of complex noun phrases in programming but do not refer to any literature from (psycho-) linguistics at all, that

could have provided them with the explanation that 1-character identifiers are harder to understand than identifiers with known nouns because they ease association-based memory access. Instead, they suppose that name assignment is an indirection that occurs in the mind of the programmer, no only in the code. Détienne (2001, 19) states that anaphors are a case of "indirect reference" and that only pronouns like *he* and *it* and pronominal adjectives like *this* function as anaphors, which is both wrong, as can not only be seen from psycholinguistic studies like Garrod, Freudenthal, and Boyle (1994). Based on her description of anaphors, Détienne concludes that anaphors are rare in source code and procedural text, while it is actually pronouns and pronominal adjectives that are rare in both kinds of text. In summary, in the few works where it is due, reference to linguistic theory is at times missing or incorrect. These works can hardly help reduce negative preconceptions of natural language and linguistics while it would be necessary to invite curiosity for and provide access to linguistic theory. I am therefore thankful to my linguistic friends who keep me in touch with contemporary linguistics.

4. What about naturalistic programming?

Lopes et al. (2003, 199) proposed the concept of naturalistic programming and stated that "the primitive abstractions in programming languages should be drawn from the study of Natural Languages, rather than from Computer Engineering or Mathematics or ad-hoc metaphors such as Objects." Lopes et al. also claim that naturalistic programming is not for "'natural language programming,' an idea that has been around for some decades and that has been instantiated occasionally [...]. We don't advocate implementing English! The languages we are proposing are naturalistic, but not natural. (Lopes et al., 2003, 204)" Still, their paper uses code to illustrate a naturalistic programming language that "reads like English (Lopes et al., 2003, 202)". In another deviation from real-world programming practice, Knöll et al. (2011) use the description of a house and a garden to exemplify their "naturalistic types". I do instead start from a commercially used programming language, so that even the addition of features from natural language will not make the new language look like English. I am also interested in evaluating the new language features using the abstract technical concepts found in real-world source code and in paying attention to problems peculiar to complex evolving software systems, e.g. the fragile base class problem (Mikhajlov & Sekerinski, 1997). Because computer programming is a very artificial activity compared to gardening, e.g., I would also like to avoid associating nature and programming.

There is another label, "programming linguistics" under which an early work comparable to what I intend to do was written (Kanada, 1981) – in Japanese, unfortunately. There has also been a book under that label, but its use of the word "linguistics" is metaphorical only (Gelernter & Jagannathan, 1990). I used another work from Japan that studies programming from a semiotic perspective (Tanaka-Ishii, 2010) as the blueprint for the label "linguistics of programming".

5. What to do?

I use linguistic theory to construct *and* implement a narrowly-defined new language feature (indirect anaphors) for an existing mainstream programming language. That way I avoid the problem described by Pane and Myers (2006, 36): that results from the psychology of programming are not reflected in new programming languages.

Indirect anaphors in source code could make programming more efficient by shortening the source code, but will not be understood equally well by all programmers, for it depends on domain knowledge. Based on a previous implementation of a compiler for indirect anaphors in Java (Lohmeier, 2011) and prior work on cognitive models of text comprehension (Lohmeier & Russwinkel, 2013), I prepared an eye-tracking experiment in which programmers read indirect anaphors in source code (see the paper in the proceedings of this workshop). The gaze durations on and after the indirect anaphors are treated as indicators of processing difficulty that will be manipulated by how often and recently a programmer read the information required to comprehend the indirect anaphor. I expect that indirect anaphors whose comprehension incorporates less familiar domain knowledge are harder to understand and will be gazed

at for a longer duration. The results of the experiment will be compared to predictions generated using a cognitive model.

I assume that eye tracking is a suitable method for studying the comprehension of indirect anaphors in source code, because it has been used to study indirect anaphors in linguistics (Garrod & Terras, 2000) and because it allows on-line data collection without interrupting the main task. It may be possible to extend the experimental setting after repetitions of the experiment with further independent variables: E.g. a Java source code editor may be created to support efficient source code reading and writing using indirect anaphors. The editor will display indirect anaphors instead of corresponding normal Java expressions only if the programmer is able to comprehend the indirect anaphors. Comprehension will be predicted using a cognitive model that reads the eye tracking record of the code that the programmer read and wrote previously.

Using the example of indirect anaphors, I am going to try out whether linguistics can advance the development of programming languages, comparable to the interdisciplinary relationship in computational linguistics, but vice versa.

6. References

Détienne, F. (2001). *Software design – cognitive aspects*. London: Springer.

Garrod, S., Freudenthal, D., & Boyle, E. (1994). The role of different types of anaphor in the on-line resolution of sentences in a discourse. *Journal of Memory and Language*, *33*(1), 39–68.

Garrod, S., & Terras, M. (2000). The contribution of lexical and situational knowledge to resolving discourse roles: Bonding and resolution. *Journal of Memory and Language*, *42*, 526–544.

Gelernter, D., & Jagannathan, S. (1990). *Programming linguistics*. Cambridge, MA: MIT Press.

Green, T. R. G. (1980). Programming as a cognitive activity. In H. T. Smith & T. R. G. Green (Eds.), *Human interaction with computers* (pp. 271–320). London: Academic Press.

Kanada, Y. (1981). *Toward programming linguistics*. Master's thesis (in Japanese with english abstract and table of contents), University of Tokyo Graduate School. Retrieved 2014/05/13, from http://www.kanadas.com/papers-e/1981/03/toward_programming_linguistics.html

Kintsch, W. (1988). The role of knowledge in discourse comprehension: A construction integration model. *Psychological Review*, *92*(5).

Kintsch, W. (1998). *Comprehension: a paradigm for cognition*. Cambridge: Cambridge University Press.

Knöll, R., Gasiunas, V., & Mezini, M. (2011). Naturalistic types. In *Onward '11: Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software* (pp. 33–47).

Knöll, R., & Mezini, M. (2006). Pegasus: first steps toward a naturalistic programming language. In *Companion to the 21st ACM SIGPLAN symposium on object-oriented programming, systems, languages & applications (OOPSLA '06)* (pp. 542–559). New York: ACM.

Lohmeier, S. (2011). *Continuing to shape statically-resolved indirect anaphora for naturalistic programming.* Retrieved 2014/06/01, from http://monochromata.de/shapingIA/

Lohmeier, S., & Russwinkel, N. (2013). Issues in implementing three-level semantics with ACT-R. In *Proceedings of the 12th international conference on cognitive modeling (ICCM).*

Lopes, C. V., Dourish, P., Lorenz, D. H., & Lieberherr, K. (2003). Beyond AOP: toward naturalistic programming. *SIGPLAN Notices*, *38*(12), 34–43.

Mikhajlov, L., & Sekerinski, E. (1997). *The fragile base class problem and its solution* (Tech. Rep.).

Nelson, G. (2006). *Natural language, semantic analysis, and interactive fiction.* Retrieved 2014/06/10, from http://www.inform7.com/learn/documents/WhitePaper.pdf

Pane, J. F., & Myers, B. A. (2006). More natural programming languages and environments. In H. Lieberman, F. Paterno, & V. Wulf (Eds.), *End-user development* (pp. 31–50). Dordrecht: Springer.

Schwarz, M. (1992). *Kognitive Semantiktheorie und neuropsychologische Realität: repräsentationale und prozedurale Aspekte der semantischen Kompetenz.* Tübingen: Niemeyer.

Schwarz-Friesel, M. (2007). Indirect anaphora in text: A cognitive account. In M. Schwarz-Friesel, M. Consten, & M. Knees (Eds.), *Anaphors in text : cognitive, formal and applied approaches to anaphoric reference* (pp. 3–20). Amsterdam: Benjamins.

Schwarz-Friesel, M., & Consten, M. (2011). Reference and anaphora. In *Foundations of pragmatics* (Vol. 1, pp. 347–372). Berlin: De Gruyter.

Tanaka-Ishii, K. (2010). *Semiotics of programming*. Cambridge: Cambridge University Press.

van Dijk, T. A., & Kintsch, W. (1983). *Strategies of discourse comprehension*. New York: Academic Press.

van Langendonck, W. (2007). *Theory and typology of proper names*. Berlin: Mouton de Gruyter.