Discussions

Volume 3 | Issue 1 Article 1

Discourse in Programming- Chafe Applied to Computer Code

Steven Dee
Case Western Reserve University

Recommended Citation
Dee, Steven () "Discourse in Programming- Chafe Applied to Computer Code," *Discussions*: Vol. 3: Iss. 1, Article 1.
DOI: https://doi.org/10.28953/2997-2582.1081
Available at: https://commons.case.edu/discussions/vol3/iss1/1

Discourse in Programming: Chafe Applied to Computer Code

-Steven Dee-

Steven is currently in the second year of a double major in computer science and cognitive science here at Case. Prior to coming here, he did some post-secondary coursework at the University of Akron, He's pledged Theta Chi and has also been involved in CWRU Film Society, the ACM, and the math club. Steven's current plans are to start up a recreational computer science and programming club on campus.

-Acknowledgements-

I'd like to thank Todd Oakley for his assistance in writing the paper, as well as for encouraging me to submit it for publication; Discussions, for electing to publish it; the cognitive science department here for providing a platform for this sort of research; Dr. Kathy Liszka at Akron for first encouraging me to branch out into alternative computer languages; Dr. Jeantet for introducing me to the field of linguistics; and Dr. Ulm for reminding me that as a computer scientist, I must decide whether to be a good wizard or a bad wizard.

Discourse, at its absolute broadest definition, can be seen quite simply as the exchange of information. In its most common occurrence, this information is human thought exchanged between participants in conversation; however, the participants need not be constrained exclusively to humans. What a programmer, for instance, does in writing code can be viewed very easily as holding discourse with a computer. Indeed, it can be very instructive to analyze programming from a discursive perspective; deficiencies in programming languages can be revealed by their deviation from conversational rules, and the nature of all discourse and cognition can be explored with an eye to phenomena that occur in programming. An analysis with regard to the linguistic framework Wallace Chafe presents in Discourse, Consciousness, and Time can be particularly enlightening in this respect.

Chafe, a professor of linguistics at the University of California Santa Barbara, has analyzed human speech with special attention to its prosodic features—its flow, speed, tone, et cetera. From this analysis, he has produced a means of reasoning about human speech production. Using this framework, it is possible to draw conjectures about language, and perhaps also about the nature of human thought.

In order to analyze programming languages with respect to Chafe's framework, we first need some heuristics. Chafe analyzes language primarily with respect to intonation units, but also with respect to subject/verb/object constructions and identifiers. An application of Chafe's framework to programming languages requires at least that we find analogous constructions in these languages.

In natural language, sentences can be broken up into a subject, a verb (or verb phrase), and an object. In programming languages, it is

useful to chunk statements in a similar way; in Java, for instance, the statement "`subject.action (object)`" can be read as "subject performs action on object." In some languages, though, actions are not tied to subjects in that manner. In C or Lua, for instance, one frequently sees constructions such as "`action (subject)`" or "`action(subject, object)`", and in Smalltalk, statements are of the form "`Subject messageVerb preposition: object`". In general, this paper and the accompanying examples will use the above constructions except when others are useful to illustrate a point.

Intonation units are of course a phenomenon in spoken language, which is rather far-removed from program source code. Source is not, however, devoid of analogous constructs. Lines of code in programming languages exhibit many traits of intonation units—they typically, for instance, obey (with some notable exceptions) Chafe's one-new-idea constraint. This is a limited heuristic—it does not account for the role of control structures or for the existence of whitespace-insensitive languages—but it is useful enough to be used throughout this preliminary survey.

Chafe observes the presence of a one-new-idea constraint (109) in intonation units. In programming, there exists an adage: "do one thing" (Atwood). That is, a function or variable should serve only one purpose. Generally referred to simply as common wisdom, this adage can be explained as a means of enforcing the one-new-idea constraint. It might be that a programmer has an easier time chunking and comprehending code that fits with his expectations for the contents of intonation units.

In his discussion of activation cost, Chafe separates information in conversation into the categories *given*, *accessible*, and *new* (74). In Chafe's framework, these categories correspond to speakers' and listeners' mental states during conversation. In this analysis, there are two components to this concept of accessibility: the programmer's mental state, and the information accessible to the computer during compilation or execution of the program. The programmer's perspective is rather loosely defined; when looking at or working on certain parts of a program, information relevant to those parts is likely to be given or accessible to the programmer.

From the computer's perspective, on the other hand, things are more openly defined. A non-trivial program frequently makes use of hundreds or thousands of variables throughout its source, and modern computers have the ability to hold effectively all of these in main memory. However, many of these are relevant only to one particular part of the program; they may be accessible to the computer, but not to the programmer, and their reference can cause confusion when reading source code. A problem in programming language design, then, is to provide as much correlation as possible between knowledge that is given or accessible to the program and knowledge that is given or accessible to the programmer.

At the low end of the spectrum is C, which provides only a bare minimum of scoping functionality. It is possible to refer to global variables from anywhere in the program code, and no means is provided of limiting access. This makes it possible for the program to reference information that is unknown to the programmer.

Programmers have worked around the deficiencies in C scoping in a number of ways. One common

practice is to prefix variables relevant to only one portion of the program with a particular string (usually an underscore), so, for instance, `_aCounter` refers to a "private" variable that should not be used elsewhere.

Another practice, demonstrated here within the Lua programming language, is to provide "closures" (Ierusalimschy §6.1). A closure is a block of code that can contain information both local to itself and inaccessible to the rest of the program. In this manner, a programmer will only see references to information that is given (previously defined within a closure) or accessible (relevant to the larger program) at any given point. For instance, on appendix A line 2, a variable ("`count`") is defined that is local to the closure. For the duration of code through which it is relevant (i.e., the closure), it can be referred to. In another closure, though, it cannot be accessed (and attempting to do so would raise an error).

Chafe indicates the existence of a light subject constraint. That is, subjects in conversational language are always either low-cost (given or accessible) or trivial (92). This constraint is enforced to some degree in most programming languages, but particularly well in object-oriented languages like Smalltalk or Java.

In Smalltalk, for instance, if one wanted to work with an instance of the `Person` object, one would first have to declare an instance variable: "`Jim := Person new.`" This can be read as "I declare `Jim` to be an arbitrary new `Person`." Here, `Jim` best fills the grammatical role of object rather than subject. Further, now that `Jim` has been declared (i.e., is *given* in the source code "conversation"), it can take on the role of subject: one can ask questions of it (e.g., "`Jim isHungry?`"), tell it to do things (e.g., "`Jim eat: aBurger`"), or (to use another example), manipulate it: "`3.14 truncated negated`", for instance, evaluates to -3 (Sharp 6). Smalltalk thus enforces the light subject constraint.

To a certain extent, the light subject constraint is satisfied by most programming languages in that variables must be declared (or at minimum, used in function arguments) before they can be used in a "subject" role. The complicating factor is, as discussed earlier, the discrepancy between accessibility of information to a program and to its programmers.

A common practice among experienced programmers is that of "playing computer." That is, an experienced programmer will frequently take on the computer's role in the "discourse" of programming, trying to determine the computer's internal state and actions at each step of execution of a program. Stepping back, it is informative to note the roles of the "participants" in this discourse: it is the programmer's goal to translate thought into code, and the computer's goal to translate code into computation and output. Yet, in performing his role, the programmer must continuously consider the role of the computer. In order for the discourse to succeed, the computer must interpret the programmer's code as intended; for this to occur, the programmer must know how the computer will interpret his code.

One particularly interesting potential area of study is the relation that this "playing computer" has to what we do in discourse with other people. It is currently an open question, for instance, whether activation cost is a reflection of delay incurred by the speaker in accessing information or an anticipation of delay incurred by the listener in accepting information. I submit that we may (albeit on a more natural, unconscious level) undergo a similar process in conversation to that

of the programmer in computing—that is, that in our discourse with other people, we unconsciously devote some energy to "playing human" in anticipating our listeners' role in conversation.

Chafe's framework appears particularly well-suited to studying the art of programming. Programming languages seem to adhere in many respects to the constraints he sets forth, and where they don't, problems and workarounds can be seen to arise. Moreover, the techniques used by programmers in this particularly specialized form of discourse can be instructive in the analysis of more general conversation. This preliminary analysis only scratches the surface; there are many further avenues of study available.

NOTES

1 Java is a popular object-oriented programming language, used frequently in computer science curricula as well as in corpo rate environments.

2 The C programming language, dating back to the 1970s, is one of the oldest computer languages still in wide use today. It is frequently used for low-level systems programming.

3 The Lua scripting language is commonly embedded within other programs as a means for users to extend them; it was de signed to be easy to use, even by people unfamiliar with it.

4 Smalltalk is a "pure" object-oriented language, whose goals are to be consistent and (as the name suggests) small—in fact, it first arose out of a challenge to fit an entire language specification on a single sheet of A4 paper.

5 In fact, there is a little more to C scoping than discussed here—in particular, static variables and per-file visibility can be used to reasonable effect in situations like the one shown. The example given here, though, while contrived, is indicative of real-world programming practice.

6 Alternatively, it could be stated that `Jim` is here new but trivial, or that this construction is in fact a violation of the light-subject constraint. Indeed, object declaration is one possible avenue of further exploration; however, the given explanation suffices for this survey.

RESOURCES

Atwood, Jeff. "Curly's Law: Do One Thing." Coding Horror. 01 March 2007. 9 April 2007 <http://www.codinghorror.com/blog/archives/000805.html>.

Chafe, Wallace. Discourse, Consciousness, and Time. Chicago: The University of Chicago Press, 1994.

Ierusalimschy, Roberto. Programming in Lua. Lua.org, 2003. 9 April 2007 <http://www.lua.org/pil/>.

Sharp, Alec. Smalltalk by Example: The Developer's Guide. Berne: The University of Berne, 1997. 9 April 2007 <http://www.iam.unibe.ch/~ducasse/FreeBooks/ ByExample/SmalltalkByExampleNewRelease.pdf>.