Ivar Tormod Berg Ørstavik

The language of programmers: computer programming as writing practice

Thesis for the degree of Philosophiae Doctor

Trondheim, Desember 2008.

Norwegian University of Science and Technology
Faculty of arts
Department of Language and Communication studies

&

Sør-Trøndelag University College
The Faculty of Informatics and e-Learning

To Randi and John

An introduction to an article-based thesis

Acknowledgments

This PhD thesis has been performed as a collaborative effort between the Faculty of Informatics and e-Learning at Sør-Trøndelag University College (AITeL, HiST) and the Department of Language and Communication Studies at the Faculty of Arts at the Norwegian University of Science and Technology (ISK, HF, NTNU).

First, I would like to thank my supervisor in applied linguistics, Lars S. Evensen, for his highly valuable academic contributions to this PhD thesis, always considerate and thoughtful advice to me personally, and inspirational responses to all my texts.

Second, I would like to thank my supervisors in computer science, Rolv Bræk, Mildrid Ljosland, and Geir Maribu, for the many truly inspiring and instructive discussions, for their openness towards this project, and for their support and trust in me personally.

Furthermore, I would like to thank all my colleagues at AITeL for four great years! Here, I would especially like to thank Atle Nes and my other colleagues at the eLearning center for their genuine companionship and encouragement, and all the students who have participated in and contributed to the SIMAS and jTrolog projects. I would also like to thank David Shepherd at the Bakhtin Centre at Sheffield University for being such an outstanding host during my stay in 2005.

Lastly, I would like to thank Randi and John for giving me joy and pride, in all I do.

1 Background

1.1 Why study the practice of writing computer programs?

Computer programs are important parts of our societies and cultures. And as the humanities study humans and their societies and culture[1], the humanities have therefore included computer programs on their research agenda. As part of this agenda, the humanities consider computer programs not only as abstract, asocial, inhumane technical systems, but as human, cultural constructs evolving within our societies and alongside human culture.

Within the humanities, human, cultural constructs such as computer programs and programming languages are often approached as either “structured sets of forms, used to represent things in the world, [or] meaningful actions and cultural practices, interventions in the world” (Linell 2005: 4). Previous studies of computer programs in the humanities have tended to focus on computer program use. Here the object of study is the active role computer programs play in various domains of human life and how they intervene in other cultural practices. These studies focus on processes of formation in which human practices and culture are shaped, processes in which the computer programs themselves may play a pivotal role. But the computer programs themselves are often considered as technical, black-box artifacts that, more or less, already exist in the world. So even though many such studies focus on cultural practices as processes of formation, the role of computer programs within this approach appear as structured forms or “ergon” (Humboldt 1999: 49).

To approach computer programs as meaningful actions and cultural practices in themselves, one must look at the construction of computer programs. Within such an approach, computer programs appear as processes of formation or “energeia” (ibid.). Here the object of study is the internal role humans and culture play in the formation of computer programs. Such studies are rare beasts in the humanities. The processes of

[1] The ‘humanities’ seek to understand humans as “creating culture and created by culture” (Evensen 1991). As humans act and make decisions based on interpretations shaped by both causal and non-causal inner and outer influences, human creativity cannot be described by only causal, scientific methods and models. Instead, the humanities must build an understanding of human actions, creativity, and culture based on both causal *and* abductive, interpretative models (Evensen forthcoming). Furthermore, my view of the humanities is colored by my own position as an applied linguist. Applied research within the humanities use both established theories and methods to understand real-world problems, but also use real-world experience to enhance theories and methods (Evensen 1991).

computer program formation, the people constructing them, and the cultures, practices, and languages these people create within and around their computer programs are not part of the humanities purview.

For the humanities, this oversight is problematic. As new computer programs are continuously unleashed in our societies and cultures, they replace and/or displace existing programs. Studies considering only existing technology thus get a short expiration date that is quickly exceeded. To keep pace with the rapid (r)evolutions of computer programs, the humanities cannot simply consider computer programs as still forms, but must also include their processes of formation. "As long as one considers technology from the viewpoint of [the created] only, its world will remain silent – it will first begin to open up when one, even here, moves from forma formata back to forma formans, from that which has become, back to the principle of becoming" (Cassirer 2006: 91, my own translation).

To patch this oversight, this PhD thesis studies computer programs as processes of becoming. Basically, computer programs are program texts. These texts are written not only for machines, but also by humans for humans. Writing these texts, programmers interact with each other to form communities, and as these communities are constantly created and recreated over time, programming languages and other common cultural constructs evolve with them. Working with these human driven processes of becoming, this PhD attempts to address the following overall issues:

- What is the practice of writing computer programs?
- How can we understand computer programs and programming languages' social, cultural, and human principles of becoming?

1.2 What is the object of study?

Even though computer programs are texts written by humans collaborating with each other for yet other human users, the practice of writing computer programs is not generally considered a phenomenon suitable for the humanities. On the contrary, to write computer programs is commonly called computer programming, and computer programming is an object of study in the three disciplines: software engineering, information systems, and computer science (hereafter collectively referred to as ‘ICS’, cf. Article D). Hence, to study the practice of writing computer programs from the viewpoint of the humanities is to work toward an interdisciplinary object of study that draws on insight from both ICS and the humanities.

This interdisciplinary object of study can be described as follows. Initially, an object of study is extracted from ICS: computer programming. Experience with computer programming and a great many computer programs establishes an empirical platform for this study. However, the object of study is not only computer programming. In addition, this study combines this empirical phenomenon with another object of study well-known in the humanities: writing. Experience with writing and texts are also placed in focus and part of the empirical platform.

This combination of computer programming and writing is an interdisciplinary, hybrid object of study. On the one hand, the object of study is both computer programming and writing. On the other hand, the practice of writing computer programs clashes with established consensus about what writing is and what computer programming is in their respective fields. In ICS, computer programming is most often considered a process where humans adapt to a technically established frame; in the humanities, writing is most often considered a process that humans socially construct and adapt to suit their needs. So, even though the two disciplines study the same empirical phenomenon of humans writing computer programs, the two disciplines work with different objects of study that need to be merged and adjusted to each other. The resulting synthesis both resembles and diverts from its mixed origins.[2]

In Sections 2-4, a detailed account of this interdisciplinary, hybrid object of study is analyzed. The purpose of this analysis is to acquaint the reader of this thesis with the practice of writing computer programs that includes and builds on insight from both ICS and the humanities. The analysis is, therefore, also directed at an audience of readers from both ICS and the humanities, and it strives to present the object of study readily available from both sides. In the four Articles, parts of this object of study are examined in detail. These in-depth studies aim to illustrate both key problems facing

[2] To establish this topic therefore requires not only to build a synthesis of two different disciplines, but also to break down established preconceptions about how computer programs and programming can be understood. To check the premises of two objects of study in this way expands our ideas about both what computer programming is and what language can be.

computer programmers today and new ways of understanding core concepts in both writing and computer programming.

To the best of my knowledge, no other coherent body of works consistently study programming languages using dialogic theory or another set of related theories of language. There are many individual studies discussing human, social, and/or cultural aspects of computer programming, several of which are referenced in this thesis, but these studies do only to a limited extent build on each other and rarely relate to any socially oriented theories of language. On the contrary, in ICS, human, social, and cultural aspects of programming languages are often deliberately tackled as practical, empirical problems – problems that should be solved as a developer or platform tool, and not through theoretical analysis or discussion. I have therefore found that the best way to draw together, present, and build on previous research that best reflect the experience in this field is through first a thorough presentation of the empirical domain (Sections 3 and 4) and then to discuss other relevant works from within this presentation (Sections 4 and 5).

1.3 Which theories are used?

To establish an interdisciplinary, hybrid object of study has strong ramifications for the choice of theory. To study computer programming and writing as two sides of the same coin requires a multidisciplinary theoretical background that can illustrate the object from both sides. However, models describing computer programs cannot at will be swapped for models describing writing, or vice versa. The choice of theoretical model and terminology is often determined by the object of study. Hence, theories and terminology are picked from two different domains in an order prescribed by the topic at hand, and not by theoretical or disciplinary practice.

To match theories in this manner is still not a common academic practice. If the relationship between theory and the object of study at some point becomes unclear, the interdisciplinary theoretical foundation has neither established disciplinary practice nor theoretical hierarchy to fall back on. In such cases, the interdisciplinary theoretical framework will appear random and misguided. For this interdisciplinary study, it is therefore paramount not to break the link between theory and empirical object, and in order to maintain this link without relying on disciplinary practice, two interdisciplinary guidelines are established.

The first guideline is to use ‘narrow bridges’ between theory and the object of study. The narrow bridge is a metaphor to illustrate that theoretical and empirical detail should be kept at a minimum when linking theories and objects of study between two different disciplines. One example of a narrow bridge is Article C: "Forms of time and chronotope in computer programming: Run-time as adventure time?" When the empirical phenomenon of envisaging the run-time environment in ICS is described with the theoretical model of chronotope from dialogic theory, then both the run-time environment and the chronotope theory should be elaborated as *little* as possible.

The reason for this guideline is the inherent complexity of wide-spanning, interdisciplinary connections. Just by applying the chronotope model on the run-time environment of computer programs, a dual bridge is created. First, two different objects of study are connected: envisaged literary worlds and virtual run-time environments. Second, two theoretical domains are connected: system architecture and dialogic theory. Both the empirical and theoretical ambitions are therefore already quite high without adding nuance to either the empirical or theoretical framework.

The second guideline is to try to build ‘two-way bridges’. The two-way bridge is a metaphor to illustrate that readers from different disciplines should hopefully be able to use the same interdisciplinary article to journey into the domain of the other discipline. Again Article C is a good example. To describe the virtual run-time environment around computer programs using the chronotope model, the article uses some elements that are already familiar for readers from both ICS and the humanities. Readers from ICS can relate to an object of study that they already know, that is, the virtual run-time environment. This familiar object is then described in a new way using an unfamiliar theory, that is, the chronotope model. Similarly, readers from the humanities are faced with a new, unfamiliar object of study but then approach this object using a familiar theoretical perspective. The bridge thus broadens the perspective of both disciplines to include an element of the other disciplines, going in the opposite direction from familiar object of study to unfamiliar theory and familiar theory to unfamiliar object of study. Furthermore, by anchoring the bridge on both sides partly verifies its structural integrity for readers uncertain of where the journey might end.

As academic culture is disciplined at heart, it is very tricky to follow these premises. Faced with a specific text, a disciplined reader may often forget or even dislike walking on a narrow, two-way bridge. As disciplines establish both different theoretical and empirical preferences, disciplinary practice can often stipulate that either the empirical or theoretical surrounding should be elaborated. However, readers coming from the opposite discipline are likely to perceive the same text differently. For readers unfamiliar with the discipline in question, the theory or empirical object presented is new and foreign, and to surround it with more details, references, and discussions would be confusing. The further apart the two disciplines are, the less weight the theoretical-empirical bridges can carry. Readers should therefore be aware that some steps in the bridge are built or excluded to clear the path for readers coming from the other side, but reading in the same text direction.

Following these two guidelines, Sections 2-4 build a narrow, two-way bridge from computer programs and programming (familiar objects of study in ICS) to writing and dialogism (familiar theoretical ground in the humanities). In order to build such a bridge that spans two disciplines, fundamental questions about both the nature of computer programs, computer programming, language, and communication need to be revived. This favors references to classical theories that are well-known to address such questions. Furthermore, in order to avoid making too many twists and turns that weigh this bridge down, empirical and theoretical nuances within each discipline are often avoided. This disfavors contemporary studies that are often associated with current ongoing discussions within each discipline, but that might be harder to access and assess by outsiders. Hopefully, by not going too deep into the theoretical and empirical surrounding of each discipline, readers from both ICS and the humanities respectively are presented an interdisciplinary junction and a sharable theoretical and empirical platform.

In the thesis as a whole, a full theoretical framework is presented. While still trying to build narrow, two-way bridges, the thesis gradually presents and elaborates on different aspects of a currently dominant theory in writing research, dialogic theory. Dialogic theory is a theory about language and communication that centers on some early philosophical discussions by Bakhtin (1981; 1984; 1986). For a general introduction to Bakhtin’s theories of language and communication see (Holquist 1990).

In Norway today, this theoretical framework is applied in several studies of various forms of communication (Bostad 2004; Evensen 2002; Evensen 1997; Linell 1998; Rommetveit 1992). In Sections 3 and 4, this theoretical framework is extended with two diachronic perspectives on language: grammaticalization (Hopper 1993) and semantic change (Bloomfield 1933). In Article B, dialogic theory is compared to two related theoretical perspectives (Cassirer 1953; Cassirer 2006; Heidegger 1997; Poole 1998) to explore how dialogic theory might color our perception of different objects of study.

To anchor and weave the presentation of theoretical framework into the analysis of and introduction to the object of study has an important benefit. For both the humanities and ICS, this thesis' main point of interest is the object of study. By presenting both dialogic theory and previous research gradually from within this empirical platform, and not the other way around, the theoretical framework and previous research is also made relevant for both sides. To maintain interest across disciplines is a primary interdisciplinary goal of this thesis, but it is a task that greatly constrains my choices as writer and that I urge both my readers and other interdisciplinary researchers not to underestimate.

Both Sections 2-4 and Articles A-D describe the practice of writing computer programs using technical terminology from ICS.

1.4 Which research method is used?

The research method used in this PhD project is participation in primarily two open source projects writing computer programs: *SIMAS* and *jTrolog*. *SIMAS* is an acronym for "Socio-Interactionistic Multi-Agent System" (Ørstavik 2008b), and the project's goal is to develop a new dialect of Java suited for developing agents and inter-agent communication. Experiences from the *SIMAS* project and the development of a new programming language dialect establish the empirical background for Articles B and C. *jTrolog* stands for "Java Trondheim Prolog", and is a fast, simple, and consistent open source Prolog interpreter written in Java (Ørstavik 2008a). Experiences from the *jTrolog* project form the main empirical background in Sections 2-4 and Article A. The technical results from these projects are presented in Section 5.2.

To actively and subjectively participate in the social and cultural practices that form the object of study, is an ethnographic method widely used in the humanities

(Bogdewic 1992). To describe how this method can be used to study writing practices, Barton (2006: 52f) states:

> [T]o understand literacy, researchers need to observe [and participate in] literacy events as they happen in people's lives, in particular times and places. The fact that different literacies are associated with different domains of life means that this detailed observation [and participation] needs to be going on in a variety of different settings.

By participating in real-life discourse and experiencing "how language is used in and across social situations" (Farnell 1998: 411), the researcher is able to create an empirical material of "lived experiences" (Rickman 1979: 29).

Furthermore, many aspects of communication and writing practices "only becomes the subject matter of the human studies when we experience human states, give expressions to them and understand these expressions" (ibid.: 175). Having experienced discourse and how language is used first hand, these experiences need to be expressed and analyzed. In this thesis, this process of expressing and analyzing "lived experience" from writing computer programs is done primarily in two ways. Excerpts from naturally occurring discourse are presented as examples. As computer programs are written down and often recorded continuously, such excerpts are largely available as texts. However, many aspects of the practice of writing computer programs cannot be adequately described through such examples. These phenomena are often explained using introspective reflections and short stories.

In ICS, a similar method of active, subjective participation is widely used. However, while this methodological approach is well-known and much debated in the humanities, its use in ICS is largely tacit. Therefore, Article D discusses in-depth the methodological tradition in ICS at NTNU and focuses in particular on the use of active participation in writing computer programs as a research method. Article D explicates some core premises for using participation in writing computer programs as a research method in ICS, and finally summarizes the method as 'ICS engineering'. ICS engineering has been a strong methodological influence in this study.

Thus, while the interdisciplinary, hybrid characteristics of this PhD thesis reveal conflicting interests and constraints on the choice of theory and empirical background, the research method is not as controversial. To actively participate in writing is a familiar method in the humanities, and to actively construct programs is a familiar method within ICS. The task of forging an interdisciplinary method and work process from the related methodological templates in the two fields was therefore less problematic than forging the interdisciplinary object of study and theoretical framework.

1.5 What are the research questions and objectives?

Having introduced the interdisciplinary object of study and the theories and method used, the research questions addressed in this PhD thesis will be presented and discussed briefly.

As mentioned in Section 1.1, the overall issues this thesis deals with are: What is the practice of writing computer programs, and how can we understand computer programs and programming languages' social, cultural, and human principles of becoming? By addressing these issues, this thesis hopes to establish new insight into a phenomenon of how computer programs are formed, a phenomenon hitherto largely ignored by the humanities. Learning more about this phenomenon will extend the humanities' understanding of changing computer programs, their role in our societies and culture at large, and particularly the role of humans, societies, and culture in computer programs. Insight into these phenomena will also extend the understanding of computer programming in ICS. As computer systems and programming languages have evolved to become ever more complex, an increased understanding of the human and social principles of becoming that underlie them has become ever more pertinent.

Other, more specific research questions that follow from this overall objective are:

- How do computer programs reflect the social interaction of programmers?
- Can programming languages influence the interaction among programmers, and if so how?
- Can programming languages influence how programmers conceptualize their problem and solution environments, and if so how?

- What role does the practice of writing computer programs play in ICS research methods?

Like the overall research question, working with these and similar questions will extend the insight of both the humanities and ICS into the domain of each other. Hopefully, shedding light on some of these issues will reveal new and highlight existing areas where ICS and the humanities can profit and learn from one another. Furthermore, many of these questions also touch upon not only problems of knowledge, but also problems of use and application. Computer programs are important tools in our society, and if we can understand the processes behind their formation better, we can create even better tools.

Lastly, all interdisciplinary projects have objectives related to the academic politics and tugs of war that constantly go on between different disciplines. Different disciplines compete with each other over empirical, theoretical, and methodological territory. Despite these competitions often being fought in a petty, destructive manner, they are, nonetheless, an overall constructive drive that forces the disciplines to renew themselves, to question their premises, and to enhance their theories and methods. The discussion of this PhD thesis research framework thus ends with an anecdote about a similar tug of war in global politics:

> MOSCOW, Aug. 2 — A pair of Russian submersibles descended more than two miles under the ice cap on Thursday and deposited a Russian flag on the seabed at the North Pole. The dive was a symbolic move to enhance the government's disputed claim to nearly half of the floor of the Arctic Ocean and potential oil or other resources there. (Chivers 2007)[3]

1.6 Comments on format and references

The format in Sections 1-5 is as follows. Computer systems, platforms, and programs such as *jTrolog* that are distributed as a single unit are put in italics. Individual program texts such as ”bus.pl” that are either very small or not distributed as a single unit are

[3] Cf. dialogism as the Russians, computer programming as the icy North Pole, and oil is money.

quoted. This mimics the style in the humanities for naming novels and movies in italics and poems and songs in quotation marks. Individual program words from these program texts such as `plan_travel` are presented in a different font. This echoes a practice in ICS that distinguishes program words in a way that does not flood the text with either italics or quotation marks. Single quotes are used to present new terms. Sections within the umbrella introduction to this PhD thesis are called Sections 1-5, and both these Sections and Articles A-D are referenced using capital letters.

Furthermore, to make academic texts gender neutral, it is common to use 'he/she' to non-specific third person pronouns. However, a major problem in computer programming is that it is far from gender neutral. Most programmers are men, and far too few women participate in writing computer programs. To illustrate this problem, Sections 2-5 use only 'he' to refer to non-specific third persons. This is a deliberate move that attempts to address a problem so that later works can start fixing it. Society at large, women themselves, and especially the communities, languages, and cultures of programmers need many more women programmers.

Lastly, the use of internet sources in general, and Wikipedia in particular, is often considered less reliable than reviewed print sources. This, I believe, is both an ethically and academically problematic position. Open internet sources are readily accessible, while reviewed print sources can be both cumbersome and expensive to obtain. Ease of access is an important criterion for a source's reliability, making it verifiable first hand for the reader. Correspondingly, the established practice of restricting printed sources economically is ethically problematic. However, ease of access should also be considered historically. Although some internet sources such as Wikipedia can be considered accessible in the long term, others cannot. Lastly, ease of access and first hand verification is particularly important in interdisciplinary research. Readers are unlikely to be familiar with and have prior access to print sources outside their discipline. To list a forest of printed sources in more than one discipline is therefore likely to function as impressive disciplinary walls rather than a two-way bridge. In this thesis, I have therefore chosen to actively use open internet sources that are less restrictive practically, economically, and disciplinary.

1.7 This thesis

This article-based PhD thesis is structured as follows. First, Sections 1-5 function as an umbrella that introduces Articles A-D. Section 1, Background, describes some ground rules about the research project itself, its object of study, its theoretical and methodological approach, and, most importantly, the constraints of the interdisciplinary setting. Sections 2-5 contain a general introduction to the practice of writing computer programs. This introduction gives an overall view of computer programs as texts and how humans write them. The purpose of the general introduction is to present the object of study to an audience from both the humanities and ICS. Prior, rudimentary experience with writing computer programs is beneficial, but not required.

Then, Articles A-D are presented. The Articles exemplify how the practice of writing computer programs can be studied in depth. Articles A-C use dialogic theory to shed new light on the practice of writing computer programs, and vice versa. Article D discusses the methodology for studying information and communication technology (ICT) and illustrates the extent to which the practice of writing computer programs is an intrinsic part of many research methods in ICS. The Articles are not primarily directed at an interdisciplinary audience, but toward two different disciplines. While Articles A-C are primarily directed toward studies of written and verbal communication based on Dialogic theory, Article D is primarily directed toward an ongoing debate within ICS and Software Engineering. Articles A-D are introduced and related to the thesis as a whole in Sections 3.9, 4.7, 4.8, and 4.10.

Both the humanities and ICS place formal and informal restrictions on academic texts. Even though the premises often contradict each other, I have tried to compose both the Sections and Articles so as to make them accessible across disciplinary boundaries. However, as the writer of these texts, I have found it impossible to completely bridge this gap on my own. Readers from the humanities are therefore encouraged to take a philosophical outlook on language and share in this projects' genuine interest in computer programming. Readers coming from ICS are encouraged to approach their own programming practices with an open mind and to connect with both themselves and what they do on a personal level.

2 The language of programmers: an introduction

2.1 Why study the language of programmers

As suggested in Section 1.1, information and communication technology (ICT) has practically invaded almost every aspect of human life and culture. In life, ultrasound machines watch over us before we are born, mobile phones and laptops follow us everywhere, and machines such as pacemakers are with us even in death. Culturally, ICT is revolutionizing communication, collaboration, and other human practices at work and at home. Children play at their computers for days on end; the “dot.com”-bubble created and destroyed a new economy; SMS language is gradually creeping into other linguistic practices; and everyone from US presidential candidates to religious fanatics build societies on the internet. ICT has become part of who we are, and that makes ICT an important area of study in the humanities.

But, technology itself does not change our culture on its own. It is largely the men behind the technological systems who initiate new cultural forms and practices, not the systems themselves. Collectively that makes these men the most disruptive and powerful agents of cultural change, and studies of ICT and cultural change within the humanities should therefore scrutinize these technological systems’ processes of becoming, the men creating them, and these men’s artistic means and culture.

In ICT, the men behind the technological systems are commonly referred to as programmers. To study their systems’ processes of becoming and their culture is to study the language of programmers, and taking this approach can give a glimpse of what goes on behind the stage of apparent cultural change. Studying the language of programmers gives us an insight into the processes driving ICT as it develops. Understanding the social and human forces within ICT development is an important task for the humanities, and with this insight the humanities can contribute to the constant evolution of the practices and ethics of computer systems during development and use.

2.2 Computer programming as writing

At the heart of ICT development lies the practice of computer programming. To program computers essentially means to write program texts. These texts greatly resemble other forms of written communication as alphabetical symbols on an empty slate. But, unlike the ink and paper texts so ingrained in our culture, program texts are embodied in a new, twinned medium: software and hardware. Program texts live their working lives as electronic impulses stored on magnetic tape (software) or as metal fillings etched in silicon stone (hardware). These new media are foreign to our culture, and so, by the power of association, computer programming is also regarded as an alien, black-box cultural practice.

This introduction aims to break this association and to recast the alien practice of programming as the familiar practice of writing texts. To reach this goal, a two-part analysis is performed. First, a computer program and the intertextual relations saturating it are analyzed. This intertextual analysis illustrates how computer programs literally function as program texts and how the meaning of program words is reciprocally constructed. Second, and based on this intertextual analysis, the practice of writing program texts is discussed. This discussion shows how computer programmers envisage their program texts as operating in an intertextual environment and how the conventions of this environment enable and constrain the programmers' imaginations.

Sections 3 and 4 build on a simple Prolog program called "bus.pl" running on a Prolog interpreter written in Java called *jTrolog* (Ørstavik 2008a). All example texts can be found at *jTrolog*'s homepage, and interested readers are encouraged to consult *jTrolog*'s source code for more details. For details about the Prolog and Java programming languages see their specification (Gosling 2000; ISO/IEC 1995; Lindholm 1999). Since Sections 3 and 4 focus on the programs presented as text, only a rudimentary understanding of the technical functions in "bus.pl", *jTrolog*, Java, and Prolog is required from the reader.

3 An intertextual analysis of a computer program

3.1 Introduction

```
bus(risvollan, nardo, 8).
bus(nardo, lerchendal, 8).
bus(lerchendal, samf, 8).
bus(samf, dronningen, 8).
bus(dronningen, samf, 5).
bus(samf, berg, 5).
bus(berg, moholt, 5).
bus(moholt, dragvoll, 5).

plan_travel(Arrive, Arrive, Visited, [], Visited).

plan_travel(Depart, Arrive, Visited, [BusLine | RouteN], Route) :-
      bus(Depart, X, BusLine),
      not member(X, Visited),
      plan_travel(X, Arrive, [X | Visited], RouteN, Route).
```

Figure 1: A Prolog program called "bus.pl".

This analysis explores the intertextual relationship between many different program texts by deconstructing an environment of real and imagined texts. The analysis revolves around a small Prolog program called "bus.pl" (figure 1) and a Prolog interpreter written in Java called *jTrolog* (Ørstavik 2008a). The analysis is performed in the following six steps:

First, a functional description of "bus.pl" as a computer program is given. This description explains what "bus.pl" is likely to do under normal circumstances and serves primarily as a quick introduction.

Second, two direct links between "bus.pl" as a program text and another program text called "BasicLibrary.java" are presented. These two links illustrate how "bus.pl" interacts with other programs and how this interaction is mediated through intertextual links. This step presents a type of intertextual links between program texts: 'direct, symbolic links'.

Third, the process of running “bus.pl” on *jTrolog* is analyzed. This analysis reveals how *jTrolog* interprets every letter in the “bus.pl” text literally. Each individual character and some designated character and word formations in “bus.pl” are linked to one or more corresponding characters, words, and statements in *jTrolog*. This step presents a second type of intertextual links between program texts: ‘direct, literal links’.

Fourth, the inner workings of *jTrolog* are analyzed. Examining how *jTrolog* runs “bus.pl” in-depth illustrates how many different program texts are indirectly linked together to form large networks of texts. These networks connect program texts together across long chains of both direct symbolic links and direct literal links that span and crisscross many different system layers. This step presents both a third type of ‘indirect, intertextual links’ and how networks of such links give program words a semantic potential.

Fifth, the relationship between existing and future program texts is discussed. Computer programs are always written for future use. These future environments will include many already written, real programs, but they will also likely include many other programs that have not yet been written: imagined programs. To illustrate how program texts link to such future, imagined texts, a purely hypothetical web-application using “bus.pl” is presented. This step presents a fourth type of intertextual links between program texts: ‘imagined links’.

Sixth and last, even a simple program text such as “bus.pl” links directly and indirectly, symbolically and literally to many both real and imagined program texts. As the number of links connecting program texts together grows, a need to form more abstract intertextual relationships emerges. Therefore, “bus.pl” and *jTrolog* link to each other not only through a series of direct links, but also through a series of agreed-upon, shared, abstract conventions of the Prolog and Java programming languages. Programming languages establish intermediaries that concrete program texts such as “bus.pl” and *jTrolog* can directly comply with, link to, and thereby indirectly relate to other program texts that follow the same conventions.

Based on this analysis of how intertextual links permeate program texts, Article A, “The Foundation of a Dialogic Grammar”, is presented. Article A discusses how intertextual links relate to language use and communication in general and how future systematic studies of communication in general and program texts in particular can tackle this phenomenon. This presentation concludes the intertextual analysis.

In sum, Section 3 illustrates how program texts function as both texts and computer programs at the same time.

3.2 A functional description of “bus.pl”

As a computer program, the job of “bus.pl” is to calculate how to travel by bus from one destination to another. The program works by first listing a set of direct connections between two bus stops, and these direct connections are then combined into a complex travel plan. The following description illustrates where and how this functionality is achieved in the program text.

On line 1-8 in figure 1, the direct bus connections are formulated as a series of `bus` facts. Each `bus` fact lists three parameters: the point of departure, the point of arrival, and the bus line’s number. On line 10-15, two `plan_travel` rules describe how several such direct connections can be joined together to form two lists. The first two parameters of the `plan_travel` rules also function to name the points of departure and arrival; the third parameter functions as a list specifying previously visited bus stops, mainly for internal use; and the fourth and fifth parameter describes the resulting travel plans as two lists of bus line numbers and bus stops respectively.

To run “bus.pl”, a query must be passed to the program. An example query is illustrated on line 1 in figure 2. When given a proper query, “bus.pl” will first take this query and try to match it with one of its two `plan_travel` rules. Since this query contains two separate points of departure and arrival, the `plan_travel` rule on line 12 in figure 1 will be invoked. Then, on line 13, the invoked `plan_travel` rule will first try to find a `bus` fact that identifies a direct connection going from `Depart` to an unknown location `X`. Then, on line 14, the invoked rule will verify that the unknown location `X` is not already part of the travel plan. The invoked rule accomplishes this by checking that `X` is not a member of the list `Visited`. This check ensures that the program does not go in endless circles and will be described in more detail in Section 3.3. Lastly, on line 15, the invoked rule then calls itself recursively to find a route from the unknown location `X` to the specified destination `Arrive`. This recursive process will continue until one of the following two conditions is met. (a) The unknown location `X` is the same as `Arrive`. This will trigger the `plan_travel` rule on line 10 that effectively stops the program. Upon return, the last two parameters of the query will describe the resulting travel plans as a list of bus line numbers and a list of bus stops. (b) All possible combinations of bus-facts have been attempted without finding a route from `Depart` to `Arrive`. The program fails.

Figure 2 below shows the result of running bus.pl with the example query. On line 5 and 6, the two variables `X` and `Y` describe step by step which bus line and which bus stop that leads from `risvollan` to `dragvoll`. Be aware that the two lists describe the route in reverse order, from right to left.

```
1  ?- plan_travel(risvollan, dragvoll, [], X, Y).
2
3  result: plan_travel(risvollan, dragvoll, [], [8, 8, 8, 5, 5, 5],
   [dragvoll, moholt, berg, samf, lerchendal, nardo])
5  X: [8, 8, 8, 5, 5, 5]
6  Y: [dragvoll, moholt, berg, samf, lerchendal, nardo]
```

Figure 2: The result of running a query with "bus.pl".

This functional analysis illustrates what the program does and how it runs under normal circumstances. A more in-depth, functional analysis can break down these processes even further and give us deeper insight into the technical concepts within and underlying the program. However, the purpose of this introduction is not to describe technical processes. This introduction aims to illustrate the human processes of programming. So, instead of analyzing the behavior of the program, this analysis will now shift and focus on the program as a text – the program as its human creator sees it.

3.3 An analysis of direct, symbolic links between two program texts

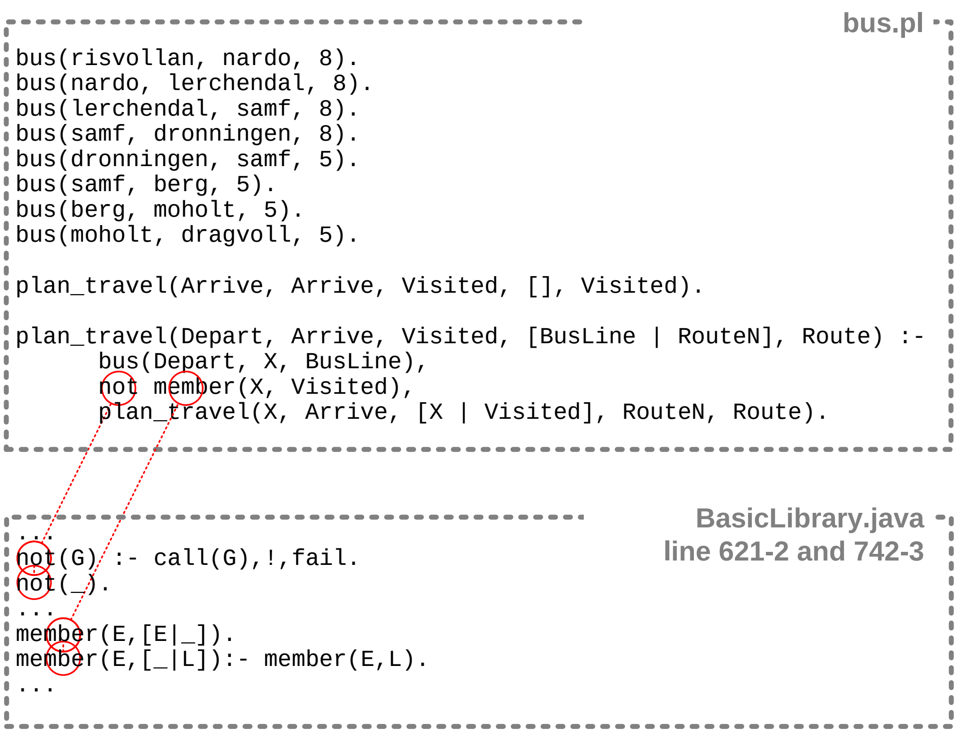
When "bus.pl" joins several direct bus connections together to form a compound travel plan, it faces one logical obstacle. As Prolog programs only do what they are told, "bus.pl" needs to be told how to avoid creating bus routes that go in circles. If allowed, such circles would cause the program to crash.

As described above, "bus.pl" avoids circular travel plans by checking that new bus stops have not already been visited. The program achieves this functionality on line

14 stating: “`not member(X, Visited)`”. However, two words, `not` and `member`, are not described elsewhere in “bus.pl”, and so another resource beside the “bus.pl” program text is needed to make sense of these words.

When bus.pl runs on *jTrolog*, this other resource is another program text called “BasicLibrary.java”. “BasicLibrary.java” describes both `not` and `member` as two Prolog rules (figure 3). The `not` rule states that if the parameter passed to it can be called successfully (`call(G)`), no alternative solutions should be made available (`!`), and the `not` rule should fail (`fail`). Otherwise, the `not` rule succeeds. The `member` rule states that if its second parameter is a list, and that the first element in this list matches the first parameter, then it succeeds. Otherwise, if the second parameter is a list that contains two or more elements, it will skip the first element and try again on the rest of the list.

When “bus.pl” is read together with “BasicLibrary.java”, the two words `not` and `member` in “bus.pl” can be read as references and calls to the corresponding words and rules described in “BasicLibrary.java”. These direct, symbolic links between the two texts implement the functionality on line 14 in figure 1 and inform our reading of “bus.pl”. The direct, symbolic links connecting “bus.pl” and “BasicLibrary.java” are illustrated in figure 3 below.



```
bus(risvollan, nardo, 8).
bus(nardo, lerchendal, 8).
bus(lerchendal, samf, 8).
bus(samf, dronningen, 8).
bus(dronningen, samf, 5).
bus(samf, berg, 5).
bus(berg, moholt, 5).
bus(moholt, dragvoll, 5).

plan_travel(Arrive, Arrive, Visited, [], Visited).

plan_travel(Depart, Arrive, Visited, [BusLine | RouteN], Route) :-
      bus(Depart, X, BusLine),
      not member(X, Visited),
      plan_travel(X, Arrive, [X | Visited], RouteN, Route).
```

```
...
not(G) :- call(G),!,fail.
not(_).
...
member(E,[E|_]).
member(E,[_|L]):- member(E,L).
...
```

Figure 3: Direct, symbolic links between "bus.pl" and "BasicLibrary.java". The lines and circles illustrate how words in two different texts point to each other.

The links between "bus.pl" and "BasicLibrary.java" are bidirectional, not unilateral. On the one hand, `not` and `member` in "bus.pl" function as calls to the `not` and `member` rules in "BasicLibrary.java" and implement these rules as subroutines in "bus.pl". This is described above. However, on the other hand, the `not` and `member` rules in "BasicLibrary.java" are written to be used in this way by other texts such as "bus.pl". The `not` and `member` rules in "BasicLibrary.java" anticipate "bus.pl", and the words in "BasicLibrary.java" are composed so as to be connectable by "bus.pl". Both "bus.pl" and "BasicLibrary.java" point to the other text through words that describe either a word in use or a word to be used.

This Section aims to establish links as bidirectional and reciprocal. If two texts are linked, and one text uses the other, a bidirectional, direct symbolic link between the two is established. In Section 3.6 and 3.7, references to other program texts in general and in the future will be further elaborated. These two Sections illustrate how points of reference from all sides are neither exclusively concrete nor exclusively abstract, neither exclusively oriented toward past texts nor exclusively toward future texts.

3.4 A literal analysis of a program text

A computer program cannot run on its own. To run, a program must be presented to a machine that can interpret its program text and convert it into physical or virtual operations. One such machine is *jTrolog*. *jTrolog* runs Prolog programs such as "bus.pl" and consists of about 50 program texts written in Java.

When *jTrolog* runs a program such as "bus.pl", it first categorizes each symbol and word in the "bus.pl" program text into one of four groups: syntactic markers, names, numbers, and variables (figure 4). Names, numbers, and variables are content elements whose textual value is largely preserved, while syntactic markers are essentially converted into structural organization of the other elements and then virtually erased from the text (figure 5). This process of recognizing and converting text into an alternative representation native to the machine constitute a 'literal' interpretation. Here, literal means according to the letter, non-figurative, and to view signs and words at face value only.

```
bus(risvollan, nardo, 8).
bus(nardo, lerchendal, 8).
bus(lerchendal, samf, 8).
bus(samf, dronningen, 8).
bus(dronningen, samf, 5).
bus(samf, berg, 5).
bus(berg, moholt, 5).
bus(moholt, dragvoll, 5).

plan_travel(Arrive, Arrive, Visited, [], Visited).

plan_travel(Depart, Arrive, Visited, [BusLine | RouteN], Route) :-
      bus(Depart, X, BusLine),
      not member(X, Visited),
      plan_travel(X, Arrive, [X | Visited], RouteN, Route).
```

Figure 4: The program text "bus.pl". Syntactic markers are colored grey, variables green, numbers red, and names black.

However, in order to achieve this literal interpretation, *jTrolog* must perform a series of small, concrete steps that kneads and grinds the text of "bus.pl" into an acceptable form. These small, concrete steps are made through a series of explicit, literal references between characters, symbols, and words in "bus.pl" and corresponding characters, symbols, and words in the *jTrolog* program texts. In fact, every single characters, symbol, word, and syntactical structure in "bus.pl" is explicitly and literally referenced in two *jTrolog* program texts: "Tokenizer.java" and "Parser.java".

The programmer of "bus.pl" anticipates how "Tokenizer.java" and "Parser.java" will relate to it. By choosing certain words and arrangement of symbols, the "bus.pl" programmer recognizes and manipulates the intertextual connections between "bus.pl" and the *jTrolog* program texts to achieve the desired effect. Similarly, "Tokenizer.java" and "Parser.java" also recognize, anticipate, and manipulate Prolog program texts such as "bus.pl". "Tokenizer.java" and "Parser.java" only make sense when they are viewed as oriented toward both actual, concrete texts and anticipated, generalized texts. As symbolic links, the literal links between "bus.pl" and "Tokenizer.java" and "Parser.java" are also reciprocal and bidirectional.

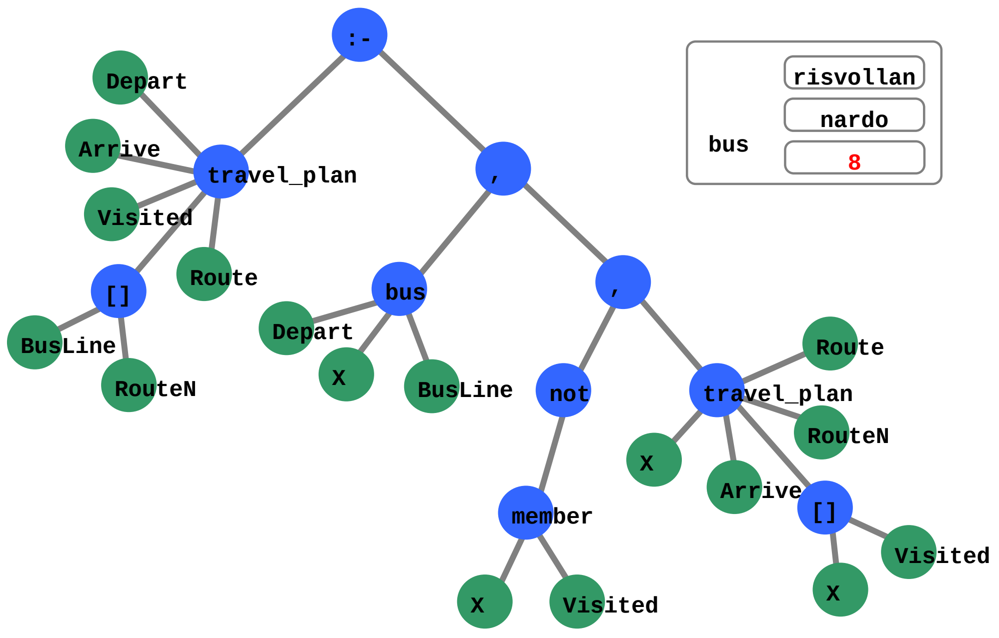
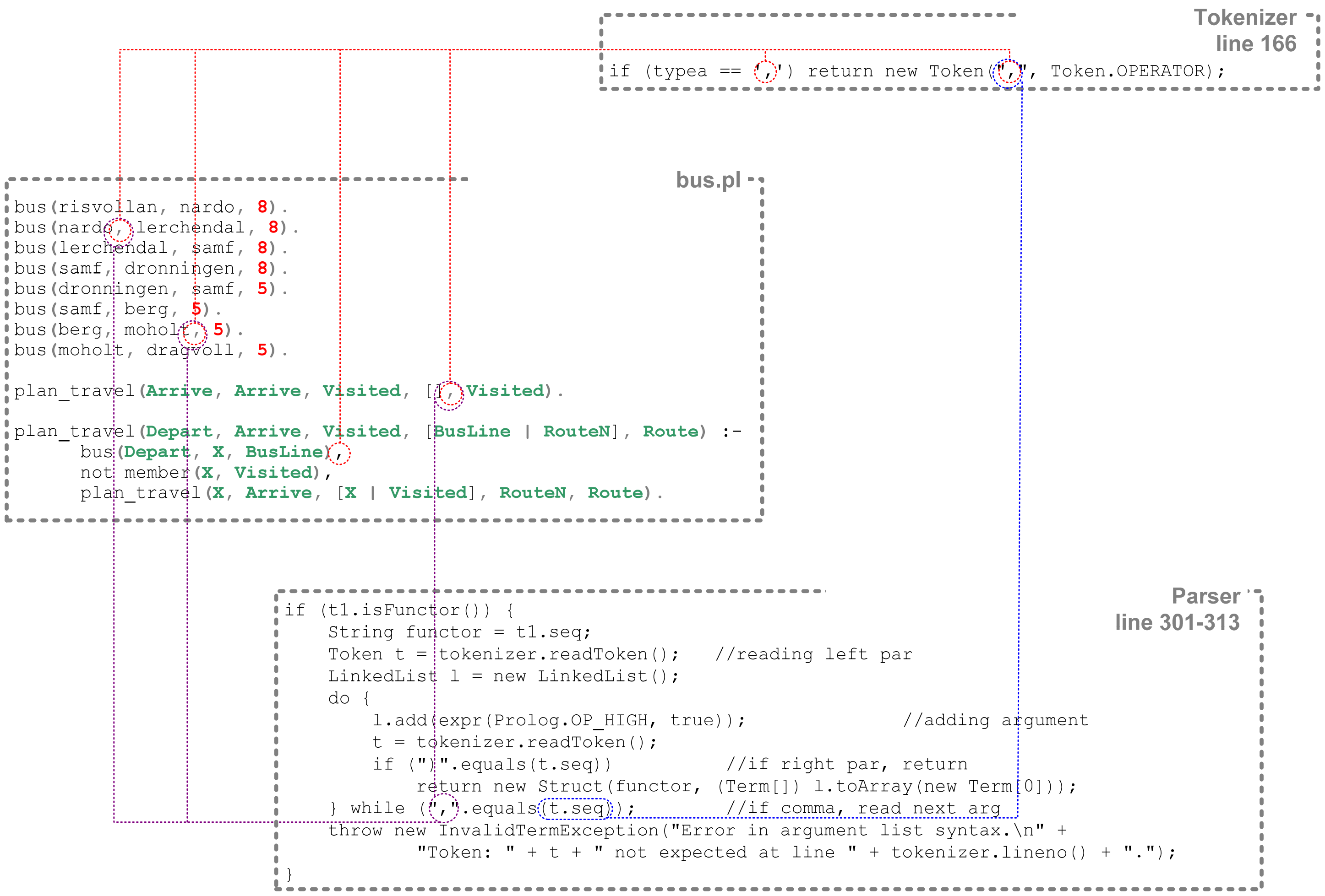


Figure 5: Two illustrations of the transformation of syntactic markers into structures of other entities. The `plan_travel` *rule on line 12-15 is depicted as a tree, and the* `bus` *fact on line 1 is presented as containers.*

Figures 6 and 7 below exemplify some of these explicit, literal references between “bus.pl”, “Tokenizer.java”, and “Parser.java”. In figure 6, some explicit references surrounding the comma symbol (“`,`”) are presented. The circles and lines illustrate how the comma characters literally point to each other and link the three texts together. These direct, literal links are instrumental in turning the text in “bus.pl” into syntactic structures in *jTrolog*. The commas in this example separate predicate parameters as a list of objects (cf. the box-in-box structure and the grey branches in figure 5). In figure 7, some explicit references surrounding Prolog variables are presented. The figure illustrates how statements such as “`Character.isUpperCase(firstChar)`” and “`variableList(indexOf(…`” in “Tokenizer.java” and “Parser.java” literally point to Prolog variables such as `X` and `Visited` in “bus.pl”. These direct, literal links are instrumental in turning some words in “bus.pl” into `Var` objects, an internal, symbolic representation in *jTrolog* (cf. the green circles in figure 5).



```
if (typea == ',') return new Token(",", Token.OPERATOR);
```

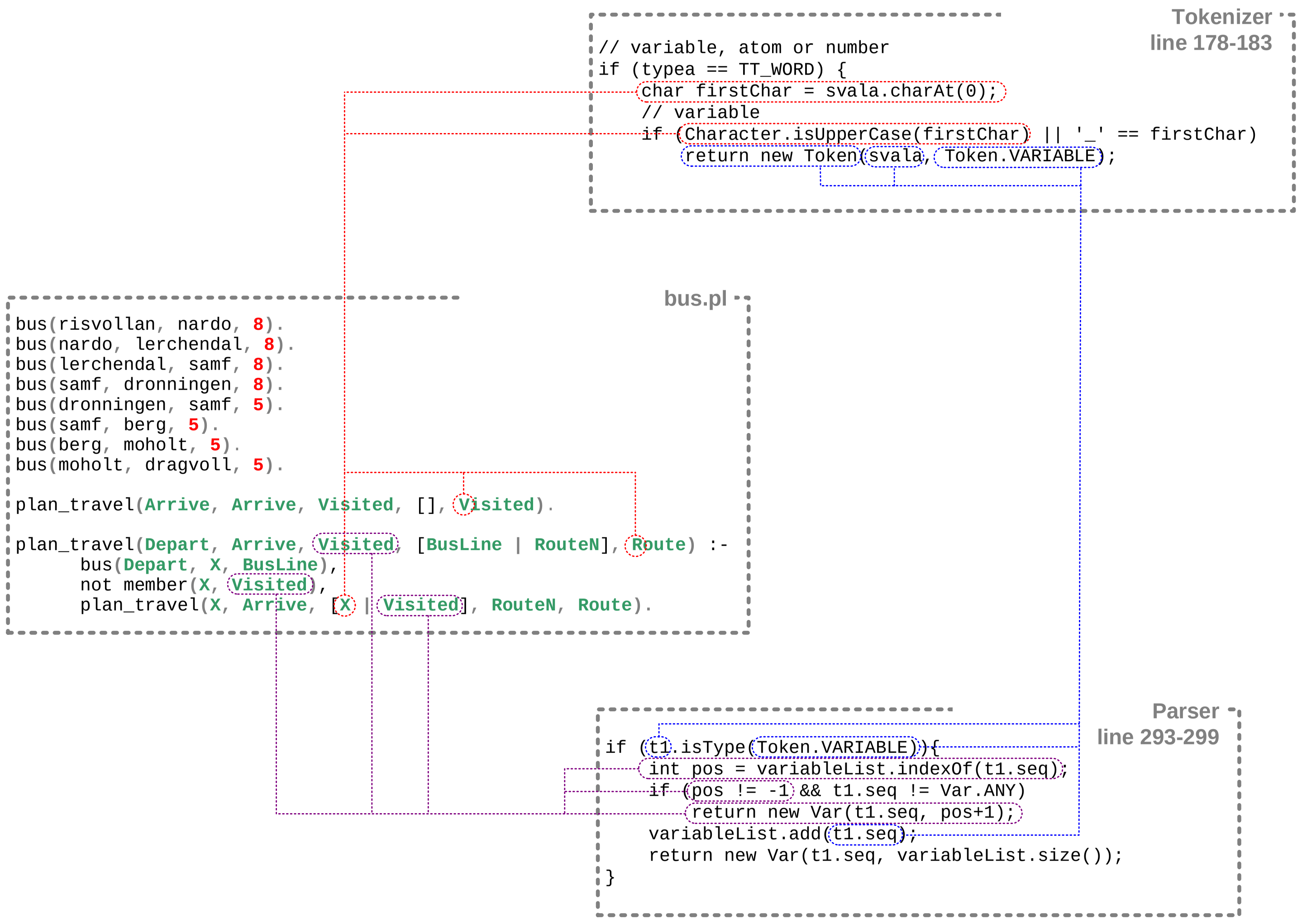
```
bus(risvollan, nardo, 8).
bus(nardo, lerchendal, 8).
bus(lerchendal, samf, 8).
bus(samf, dronningen, 8).
bus(dronningen, samf, 5).
bus(samf, berg, 5).
bus(berg, moholt, 5).
bus(moholt, dragvoll, 5).

plan_travel(Arrive, Arrive, Visited, [], Visited).

plan_travel(Depart, Arrive, Visited, [BusLine | RouteN], Route) :-
      bus(Depart, X, BusLine),
      not member(X, Visited),
      plan_travel(X, Arrive, [X | Visited], RouteN, Route).
```

```
if (t1.isFunctor()) {
    String functor = t1.seq;
    Token t = tokenizer.readToken();   //reading left par
    LinkedList l = new LinkedList();
    do {
        l.add(expr(Prolog.OP_HIGH, true));              //adding argument
        t = tokenizer.readToken();
        if (")".equals(t.seq))           //if right par, return
            return new Struct(functor, (Term[]) l.toArray(new Term[0]));
    } while (",".equals(t.seq));         //if comma, read next arg
    throw new InvalidTermException("Error in argument list syntax.\n" +
            "Token: " + t + " not expected at line " + tokenizer.lineno() + ".");
}
```

Figure 6: Some of the intertextual connections between "bus.pl", "Tokenizer.java", and "Parser.java" that literally describe commas.



```
// variable, atom or number
if (typea == TT_WORD) {
    char firstChar = svala.charAt(0);
    // variable
    if (Character.isUpperCase(firstChar) || '_' == firstChar)
        return new Token(svala, Token.VARIABLE);
```

```
bus(risvollan, nardo, 8).
bus(nardo, lerchendal, 8).
bus(lerchendal, samf, 8).
bus(samf, dronningen, 8).
bus(dronningen, samf, 5).
bus(samf, berg, 5).
bus(berg, moholt, 5).
bus(moholt, dragvoll, 5).

plan_travel(Arrive, Arrive, Visited, [], Visited).

plan_travel(Depart, Arrive, Visited, [BusLine | RouteN], Route) :-
      bus(Depart, X, BusLine),
      not member(X, Visited),
      plan_travel(X, Arrive, [X | Visited], RouteN, Route).
```

```
if (t1.isType(Token.VARIABLE)){
    int pos = variableList.indexOf(t1.seq);
    if (pos != -1 && t1.seq != Var.ANY)
     return new Var(t1.seq, pos+1);
    variableList.add(t1.seq);
    return new Var(t1.seq, variableList.size());
}
```

Figure 7: Some of the intertextual connections between "bus.pl", "Tokenizer.java", and "Parser.java" that literally describe variables.

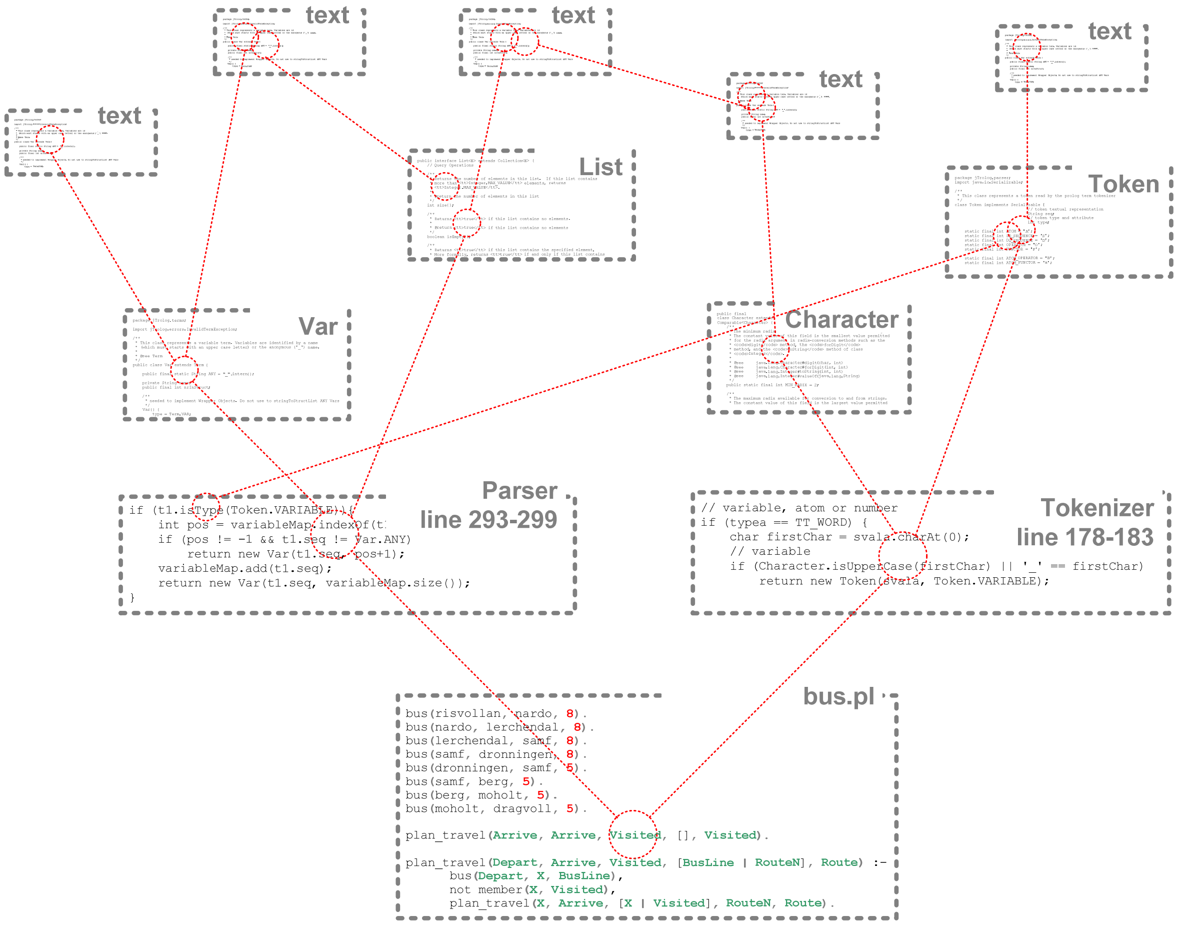
These and many other direct, literal co-references fulfill the intent implied in both the “bus.pl” and the *jTrolog* program texts to create a virtual body of lexical and syntactic entities between them. These direct, literal links inform our understanding of the words in “bus.pl” in parallel with the direct, symbolic links between texts such as “bus.pl” and “BasicLibrary.java”. Different intertextual links do not suppress or trump one another, but instead melt together and complement each other to form an ever larger and richer textual and functional picture.

3.5 Indirect links between program texts

The direct, literal links between “bus.pl”, “Tokenizer.java”, and “Parser.java” present a simple, still image of the variables in “bus.pl”. However, as described in Section 3.1, variables in Prolog programs such as “bus.pl” are not still, but changing images. Prolog variables play an active role in running programs, and reading only a literal interpretation of “bus.pl” against the *jTrolog* program texts gives us only one piece of the puzzle. To learn more about what variables do, their purpose, and function, we must, therefore, read between the lines of different program texts in a wider intertextual context.

In figure 7, the variable `Visited` in “bus.pl” is linked to both “Tokenizer.java” and “Parser.java”. First, words starting with an upper-case letter are recognized as a `Token` object. This interpretation is based on statements such as “`Character.isUpperCase(…`” and “`new Token(…`” on line 182-183 in “Tokenizer.java”. Second, each variable token is added to a list, and a new `Var` object is created using each variables’ position in this list. This interpretation is based on words such as “`variableList.indexOf(…`” and “`new Var(…`” on line 294-298 in “Parser.java”.

The literal interpretation of variables described in “Tokenizer.java” and “Parser.java” thus relies heavily on the use of other program words such as `Token`, `isUpperCase`, `indexOf`, and `Var`. These other words constitute a new set of direct, symbolic links to a new group of program texts: “Token.java”, “Character.java”, “List.java”, and “Var.java”. “Tokenizer.java” and “Parser.java” are thus part of a larger web of intertextual links, and these other links are instrumental in shaping the meaning of the links between “bus.pl”, “Tokenizer.java”, and “Parser.java”. Figure 8 below illustrates a network of indirect links which inform this interpretation of variables in “bus.pl”.



```
if (t1.isType(Token.VARIABLE)) {
    int pos = variableMap.indexOf(t
    if (pos != -1 && t1.seq != Var.ANY)
        return new Var(t1.seq, pos+1);
    variableMap.add(t1.seq);
    return new Var(t1.seq, variableMap.size());
}
```

```
// variable, atom or number
if (typea == TT_WORD) {
    char firstChar = svala.charAt(0);
    // variable
    if (Character.isUpperCase(firstChar) || '_' == firstChar)
        return new Token(svala, Token.VARIABLE);
```

```
bus(risvollan, nardo, 8).
bus(nardo, lerchendal, 8).
bus(lerchendal, samf, 8).
bus(samf, dronningen, 8).
bus(dronningen, samf, 5).
bus(samf, berg, 5).
bus(berg, moholt, 5).
bus(moholt, dragvoll, 5).

plan_travel(Arrive, Arrive, Visited, [], Visited).

plan_travel(Depart, Arrive, Visited, [BusLine | RouteN], Route) :-
      bus(Depart, X, BusLine),
      not member(X, Visited),
      plan_travel(X, Arrive, [X | Visited], RouteN, Route).
```

Figure 8: A network of intertextual links between “bus.pl”, “Tokenizer.java”, “Parser.java”, and other program texts in jTrolog *and Java supporting a functional interpretation of “bus.pl”.*

The “Var.java” program text plays a pivotal role when *jTrolog* runs ”bus.pl”. First, the Prolog variables in “bus.pl” such as `Visited` are set up as `Var` objects by “Parser.java” as parts of a predicate. “Var.java” functions as an internal representation of the “bus.pl” variables that is native to the symbolism of the *jTrolog* program texts. And to get a sense of what the words “`new Var`” means in the “Parser.java” text, we must look at the role of this program text when *jTrolog* runs “bus.pl”.

When *jTrolog* solves a query, it builds a predicate for the query. It then matches this query predicate with predicates that describe the facts and rules of its running programs (cf. figure 5 for a visual description of *jTrolog* predicates). If the query matches a fact or rule predicate that in turn contains references to yet other rules, these rules are then converted to sub-queries and attempted solved. jTrolog stacks these queries on top of each other until they are all resolved. These operations are mainly described in “Prolog.java”, “Engine.java”, “ChoicePoint.java”, and “LibraryTheoryManager.java”.

In this process of iteratively and recursively solving queries, the `Var` objects play an important role. To successfully match two predicates, the two predicates must be identical except for variables. Variables function as wild cards, and when two predicates are matched during query resolution, variables on both sides are bound to the corresponding part of the opposite predicate. This process is mainly described in “BindingsTable.java” and “LinkTable.java”. Several other program texts in *jTrolog* such as “Struct.java”, “Term.java”, and “GarbageCan.java”, as well as several program texts on the Java platform such as “HashMap.java”, “List.java”, and “Iterator.java”, are also involved in handling Prolog variables such as `Visited` in “bus.pl” within the environment described in the *jTrolog* program texts.

As all of these texts refer in turn to several other texts, the number of texts indirectly involved in any interpretation of variables in *jTrolog* grows exponentially. To further complicate this picture, two texts can refer to each other several times in different settings depending on the state of the running system. Put metaphorically, if we question the meaning behind these links by searching for the text they refer to, each answer will just produce more new questions, and some of these questions will even refer back to a previous question that we thought we had already answered. Thus, intertextual links both give texts a semantic interpretation and dilute this semantic meaning at the same time. It is as if a group of stingy churchgoers promise to give the church a donation, but instead of giving anything of value individually, they end up constantly passing the collection plate between themselves, up and down the aisle. A human-readable, complete, step by step rendering of the intertextual links informing the concept of Prolog variables in “bus.pl” and *jTrolog* is therefore practically and principally impossible.

Figure 9 below illustrates a network of indirect, intertextual links involved in a wider interpretation of variables in “bus.pl”.

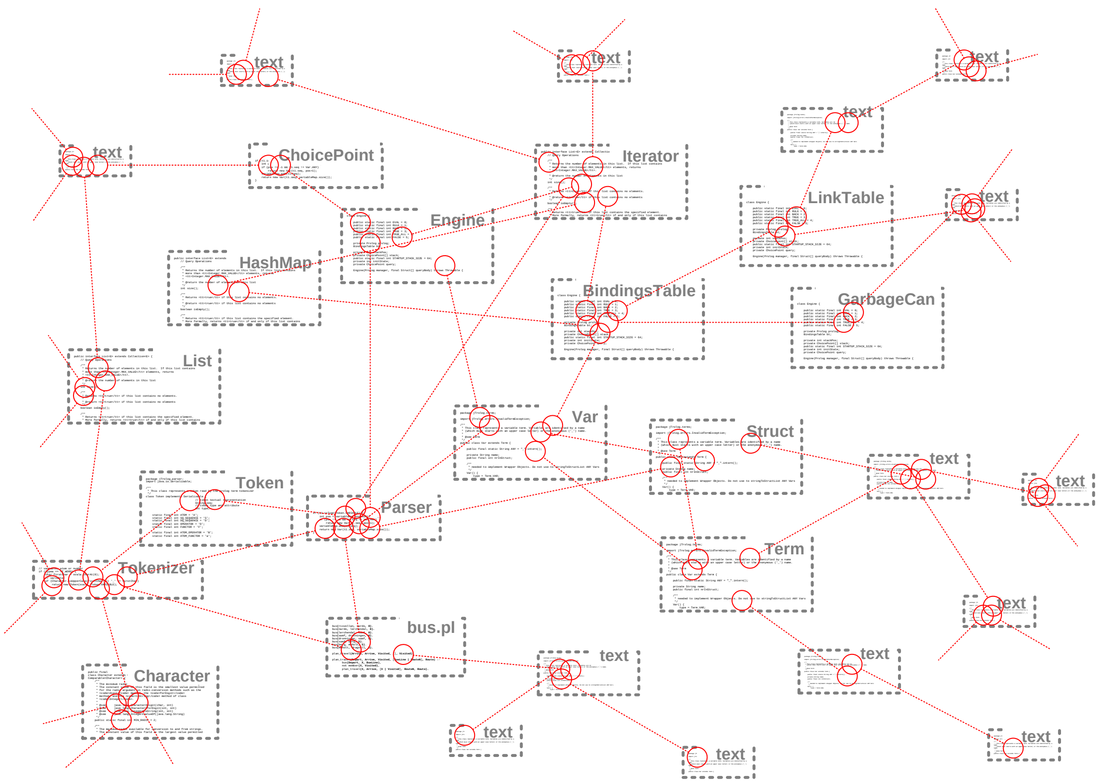


Figure 9: The links and semantic spaghetti that surround "Var.java". This figure illustrates the complexity of indirect, intertextual links connecting program texts.

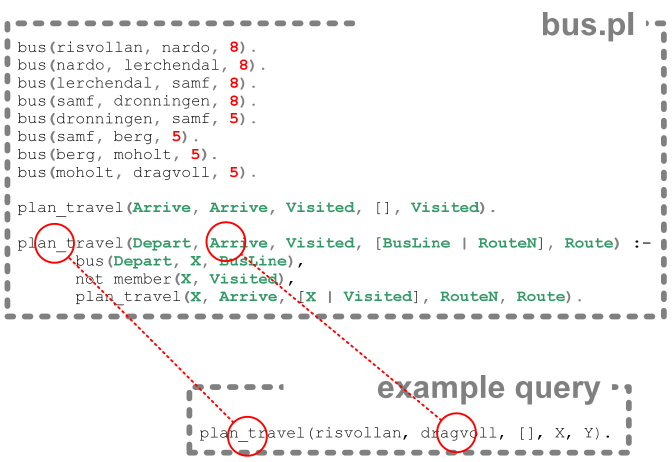
When several program texts are put together, the direct links between them become part of a network of links that indirectly connects many more texts together. These links inform our understanding of the context in which texts are interpreted. The function of a program word cannot be isolated to its direct, intertextual links, but is instead a product spread out across these networks. On the one hand, the indirect links lend each other a semantic meaning as playing a certain role in such networks of texts, while, on the other hand, the need to interpret words by continuously following yet another indirect link dilutes the semantic meaning of every word. Meaning and function of program texts cannot be meaningfully deconstructed as atomic entities, but should be understood as codependent, reciprocal references in many different, constantly renewing intertextual webs[4].

[4] The concept of intertextual web should not be confused with the concept semantic web. Semantic webs generally attempt to associate a global semantic value to a generalized word; the intertextual webs here

3.6 Links between real and imagined program texts

“bus.pl” was written with the purpose of presenting a simple example of how to use *jTrolog*. This example was later fitted to this introduction, and so the actual motive behind “bus.pl” was never to write a real bus-information application. However, this example works better when such a bus-information application is easily imaginable. Reading and analyzing the program text of “bus.pl” as if it were to be used by real people asking it for a new bus route through Trondheim, makes it easier to understand the functions and meaning of its words. The environments we imagine around programs such as “bus.pl” are important tools for our literal and symbolic interpretation of their program texts.

As illustrated by the example query in figure 2, the imagined environment of use surrounding “bus.pl” can be written down. Once formulated, an imagined query becomes a real text (cf. figure 10). The future environment thus includes imagined texts. All programmers envisage their programs in future use, and so in anticipation of these settings and future input, program words are composed so as to facilitate other future texts using them. The `plan_travel` rules in “bus.pl” were, for example, written so as to simplify the process of formulating future queries. By actively making these textual adjustments before any actual query is written, “bus.pl” actively links to an imagined query text.



```
bus(risvollan, nardo, 8).
bus(nardo, lerchendal, 8).
bus(lerchendal, samf, 8).
bus(samf, dronningen, 8).
bus(dronningen, samf, 5).
bus(samf, berg, 5).
bus(berg, moholt, 5).
bus(moholt, dragvoll, 5).

plan_travel(Arrive, Arrive, Visited, [], Visited).

plan_travel(Depart, Arrive, Visited, [BusLine | RouteN], Route) :-
      bus(Depart, X, BusLine),
      not member(X, Visited),
      plan_travel(X, Arrive, [X | Visited], RouteN, Route).
```

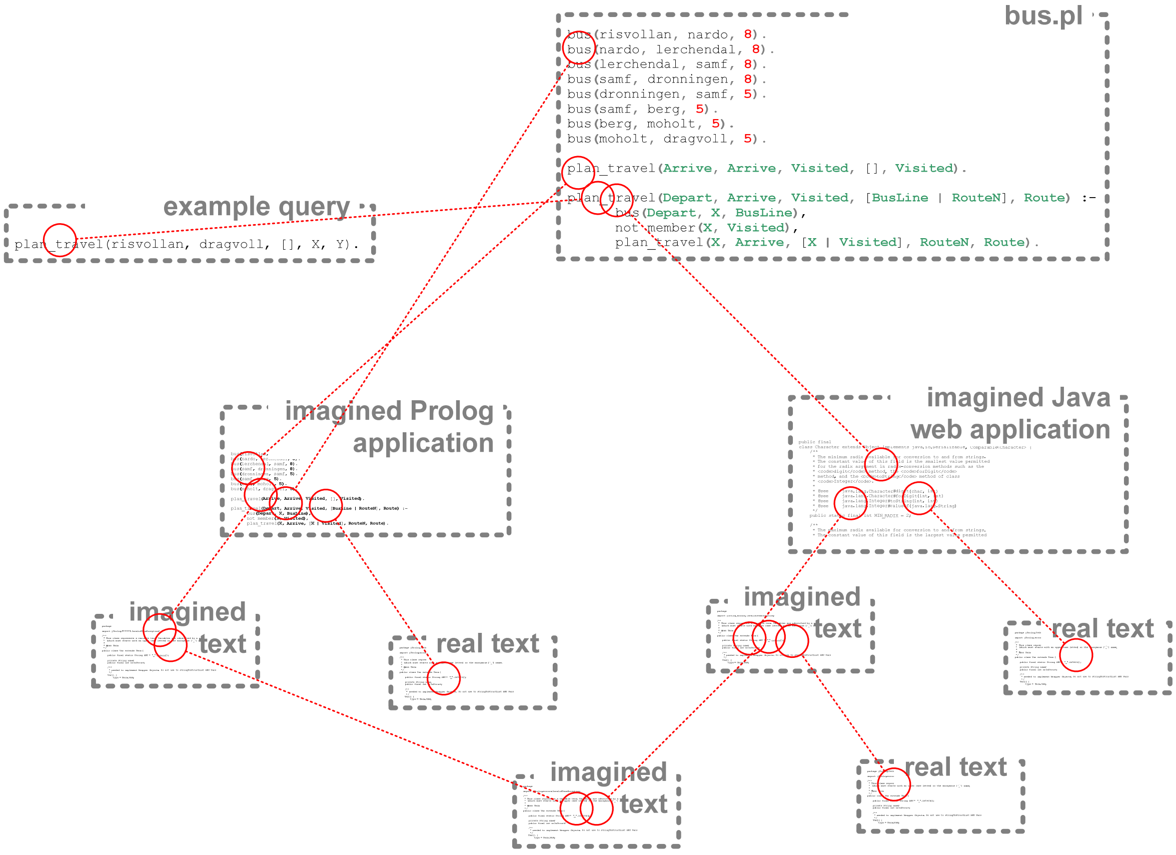
```
plan_travel(risvollan, dragvoll, [], X, Y).
```

Figure 10: “bus.pl” and an example query as two texts directly linked together.

described associate a highly local semantic value to individual instances of a word, semantic values that may differ even at different locations within the same intertextual context.

Several program words in “bus.pl” can thus be viewed as potential anchors or pointers to words in imaginable program texts. For example, the `travel_plan` rule can be seen as a resource in a future web application listing bus routes in Trondheim; and the bus-facts can be used by another Prolog application to list bus lines for individual bus stops. Furthermore, such imagined links are not restricted to only new situations of use. Words in “bus.pl”, such as `member`, can be imagined linking to a program text part of a different Prolog machine that, for example, can run “bus.pl” on a mobile phone.

Some of these imaginable program texts may very concretely be envisaged by the programmer. Others may appear first long after the program is written, or not at all. So, regardless of how specific the foreseen, future textual environment of “bus.pl” is, it is never final and closed. New texts may always be imagined surrounding “bus.pl”, and so the future environment around “bus.pl” holds an infinite number of imaginable intertextual links. Thus, the meaning of the words in “bus.pl” is never fixed at the time of writing, but open and including evermore imaginable intertextual links. Figure 11 below illustrates both these real and imagined program texts surrounding “bus.pl”.



```
bus(risvollan, nardo, 8).
bus(nardo, lerchendal, 8).
bus(lerchendal, samf, 8).
bus(samf, dronningen, 8).
bus(dronningen, samf, 5).
bus(samf, berg, 5).
bus(berg, moholt, 5).
bus(moholt, dragvoll, 5).

plan_travel(Arrive, Arrive, Visited, [], Visited).

plan_travel(Depart, Arrive, Visited, [BusLine | RouteN], Route) :-
        bus(Depart, X, BusLine),
        not member(X, Visited),
        plan_travel(X, Arrive, [X | Visited], RouteN, Route).
```

```
plan_travel(risvollan, dragvoll, [], X, Y).
```

Figure 11: Links between "bus.pl" and other real and imagined program texts.

3.7 Links to "generalized other" [5] program texts

Computer programs are not only characterized by links between individual texts. Computer programs also describe abstract ideas, concepts, categories, and structures. A simple example of such an abstract concept is the word `bus` on line 13 figure 1. This word refers not only to one of the `bus`-facts on line 1-8 in figure 1, but to all of these `bus`-facts as a group. When "bus.pl" runs, this `bus` word directly refers to each of these individual `bus`-facts one at a time. The `bus` word thus points to a collective of `bus`-facts, a partly individual and concrete, partly general and abstract textual exemplar.

By referring to an exemplar of the `bus`-facts, the word makes sense both as a reference to an abstraction and as a reference to concrete text fragments. As new texts using the word `bus` are written around "bus.pl", `bus`-facts are added and subtracted from

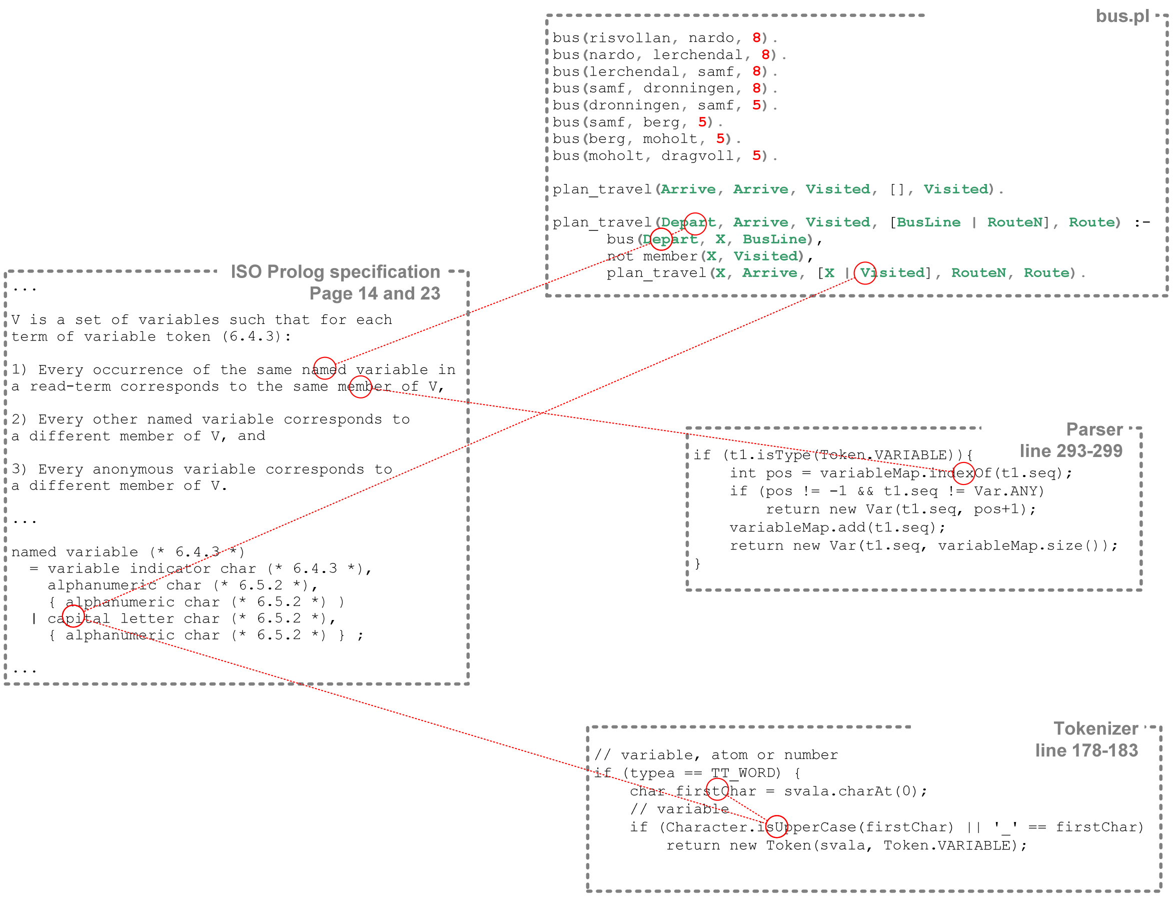
[5] See Section 4.5 for a more in-depth discussion of the use of the term "generalized other".

the intertextual environment. Thus, the specific meaning of the word `bus` has the potential for constant change, and the meaning of the word `bus` can fluctuate between being an individualized and a generalized reference. The word holds the potential for both. At the same time, the meaning of the word `bus` represent both an abstract and a concrete idea: `bus` is filled by the meaning of one individual text fragment, many individual text fragments, and a possible set of future and imagined text fragments.

As the number of concrete and imaginable texts grows, the complexity of the web of links between them becomes overwhelming. For the individual human mind it is impossible to manage all of these links at the same time. Programmers therefore create conventions for clustering similar links together as one and abstracting out orderly images on top of their webs of links. Like the texts they link up, these conventions are shared between programmers, and when several programmers use the same conventions to organize the links between their texts, they can and do link together vast bodies of individual, concrete texts.

One particularly important type of shared conventions is programming languages. The direct, literal links between “bus.pl”, “Tokenzier.java”, and “Parser.java” described in Section 3.4, also comply with the conventions for literal interpretation in the Prolog programming language. In practical programming, the references in the “bus.pl”, “Tokenizer.java”, and “Parser.java” that underpin literal interpretation are commonly perceived as directly complying and linking to the abstract concepts of the programming language. These direct links to conventions complement the direct, literal links just as the generalized and individualized links in the word `bus` described above.

The role of the Prolog programming language is to function as an intermediary. The variables in “bus.pl” such as `Visited` and statements in “Tokenizer.java” and “Parser.java” such as “`Character.isUpperCase(…`” and “`variableList.indexOf(…`”, all point to the same convention for Prolog variables. The rules of this convention is written down in the Prolog programming language specification (ISO/IEC 1995), and figure 12 below illustrates the indirect links between “bus.pl”, “Tokenizer.java”, and “Parser.java” as mediated through this convention.



```
bus(risvollan, nardo, 8).
bus(nardo, lerchendal, 8).
bus(lerchendal, samf, 8).
bus(samf, dronningen, 8).
bus(dronningen, samf, 5).
bus(samf, berg, 5).
bus(berg, moholt, 5).
bus(moholt, dragvoll, 5).

plan_travel(Arrive, Arrive, Visited, [], Visited).

plan_travel(Depart, Arrive, Visited, [BusLine | RouteN], Route) :-
      bus(Depart, X, BusLine),
      not member(X, Visited),
      plan_travel(X, Arrive, [X | Visited], RouteN, Route).
```

```
...
V is a set of variables such that for each
term of variable token (6.4.3):

1) Every occurrence of the same named variable in
a read-term corresponds to the same member of V,

2) Every other named variable corresponds to
a different member of V, and

3) Every anonymous variable corresponds to
a different member of V.

...

named variable (* 6.4.3 *)
  = variable indicator char (* 6.4.3 *),
    alphanumeric char (* 6.5.2 *),
    { alphanumeric char (* 6.5.2 *) }
  | capital letter char (* 6.5.2 *),
    { alphanumeric char (* 6.5.2 *) } ;

...
```

```
if (t1.isType(Token.VARIABLE)){
    int pos = variableMap.indexOf(t1.seq);
    if (pos != -1 && t1.seq != Var.ANY)
        return new Var(t1.seq, pos+1);
    variableMap.add(t1.seq);
    return new Var(t1.seq, variableMap.size());
}
```

```
// variable, atom or number
if (typea == TT_WORD) {
    char firstChar = svala.charAt(0);
    // variable
    if (Character.isUpperCase(firstChar) || '_' == firstChar)
        return new Token(svala, Token.VARIABLE);
```

Figure 12: "bus.pl", "Tokenizer.java", and "Parser.java" indirectly links to each other through the conventions for Prolog variables (ISO/IEC 1995: 14, 23).

Furthermore, the links to the conventions for Prolog variables are not unique to the three texts described here. Other Prolog programs use the same convention for their variables, and other Prolog machines such as *SWI* (Wielemaker 2008) and *JLog* (Holst 2008) also literally interpret Prolog variables according to the same convention. Seen from the "bus.pl" viewpoint, the Prolog programming language conventions represents several Prolog machines in general; seen from *jTrolog*, Prolog represents Prolog programs in general. The conventions thus represent "a generalized other" (Mead 1934: 154-8) of program texts, a generalized other that represents many both existing and future texts. And, by orienting themselves according to these conventions, individual, concrete texts link *directly* to such generalized other texts. Figure 13 below illustrates how programming language conventions such as Prolog functions as a nexus and generalized other.

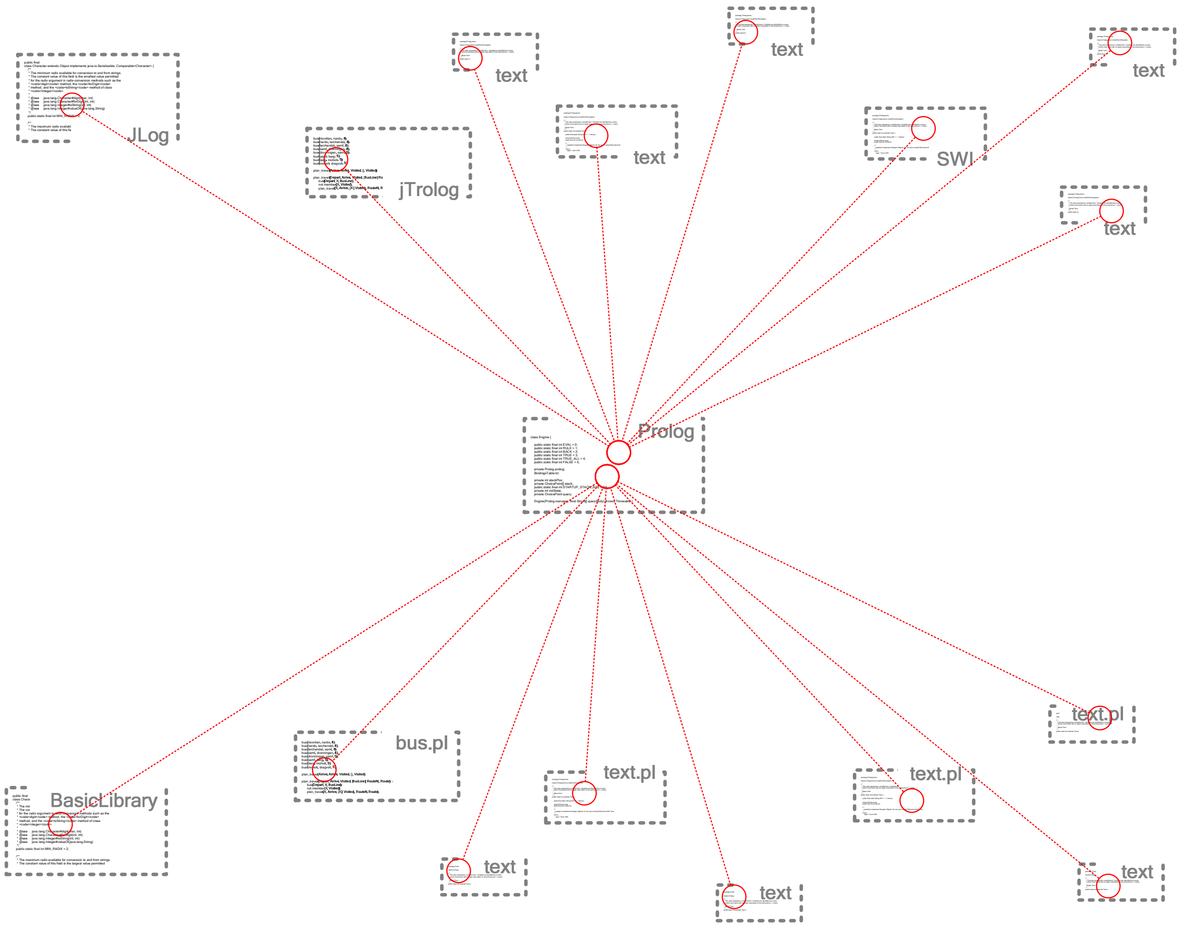


Figure 13: Prolog as a "generalized other" Prolog machine program text and a "generalized other" Prolog program.

To conclude, "bus.pl" does not exclude other programs from giving their interpretation of it. On the contrary, "bus.pl" is written adhering to the Prolog programming language, and one of the main purposes of the shared textual norms of Prolog is to make "bus.pl" interpretable and runnable on more than one individual, concrete Prolog machine. Thus, the words and symbols in "bus.pl" therefore reference both individual, concrete texts such as "Tokenizer.java" and "Parser.java" and a "generalized other" of other Prolog machines and other Prolog programs texts that might run it, build on it, and even misinterpret it.

3.8 A word that connects everything: bus

In Sections 3.3-3.7, the links connecting texts have been illustrated using many different words. To conclude this analysis, the five different links will be illustrated in one word: `bus`.

First, in Section 3.3, `not` and `member` were described as directly, symbolically linked across two different texts. Similarly, the eight `bus` facts on line 1-8 in figure 1 are directly, symbolically linked to the `bus` word in the `travel_plan` rule on line 13. The word `bus` directly links the two parts of the “bus.pl” text together, just as the word `travel_plan` directly links “bus.pl” to the example query text in figure 10. Thus, both `bus` and `travel_plan` include direct, symbolic links.

Second, when the example query text and “bus.pl” are put together, the direct links between the `travel_plan` rule, the syntactic connection between `travel_plan` and `bus`, and the direct link between the different `bus` words are also connected. These direct connections establish an indirect link that connects the `bus` facts with the example query text in figure 10.

Third, in order to facilitate these links, the word `bus` must also be literally interpreted by “Tokenizer.java” and “Parser.java”. As the literal links embodying Prolog variables described in Section 3.4, the `bus` word is directly, literally linked to symbols, words, and statements in “Tokenizer.java” and “Parser.java”. These direct, literal links mark `bus` as the name of a predicate (called ‘predicate indicator’ in Prolog).

Fourth, the morphological rules for forming predicate indicator are also described by Prolog language conventions. By relying on and complying with these conventions, `bus` thus link to Prolog machines in general, in parallel with other Prolog programs.

Fifth and last, as described in Section 3.6, `bus` can be used as a point of reference for future Prolog applications. Other future, imagined applications might use the `bus` facts to create a visual map of bus routes, identify locations with poor bus service and plan new routes, etc.

To conclude, `bus` can and does point to both other existing texts, other “generalized” texts, and other imaginable texts, both directly and indirectly – symbolically and literally. The intertextual links are echoed, anchored, and reflected by all, and it is these links that turn the program text into a runnable computer program. Every word and entire program texts are in this way permeated by many different links at the same time. These links are nested together in large, complex, intertextual webs. These webs are principally open, accepting new texts everywhere that, in principle, can turn the interpretation of other program texts and their words up side down.

In Section 4, we will look at how programmers approach these webs of texts and how they imagine and write their programs. Section 4 addresses some of the role words, texts, and language play in the practices of writing computer programs. This gives us a deeper understanding of the social collaboration, cultural construction, and individual thinking that the intertextual webs of program texts and languages reflect.

3.9 Article A: The foundation for a dialogic grammar

Program functionality is expressed as webs of textual references that crisscross within and between program texts. To understand these webs of intertextual links, further systematic analysis of larger clusters of related, codependent program texts must be undertaken. Today, such systematic analysis is commonly associated with grammatical analysis. But, since existing grammars predominantly focus on the structure of links within isolated texts, the use of established grammatical frameworks to study intertextual links is problematic.

Thus, in order to systematically analyze such webs of directly and indirectly linked texts, new grammatical frameworks need to be worked out. In Article A, a foundation for one such framework, namely a dialogic grammar, is presented and discussed. First, Article A illustrates the need and possibilities for such a dialogic grammar in several domains of written communication. Then, the existing methodological and practical premises of previous relevant systematic analysis in linguistics are presented. This presentation illustrates how existing grammar-like studies base their investigations on individually isolated texts and do not venture out into larger intertextual contexts. Therefore, in order for a dialogic grammar to capture and systematically describe intertextual links and the structures they make up, a dialogic grammar needs a different data base that not only includes many texts, but includes many, socially interconnected texts. One model, the “diatope” (Evensen 2002), is used to figuratively illustrate this data base as a foundation for a dialogic grammar.

The dialogic grammar aims at systematically describing both the literal and symbolic links between texts, how and what webs such links form, and how many concrete links can be abstracted as links to a “generalized other”. Future research into program texts in particular and written communication in general should pursue systematic analysis of intertextual links in larger bodies of socially connected texts. Such studies will meet major obstacles in handling and analyzing textual data: (a) new technical and linguistic methods must be created; (b) new taxonomies and categories for describing intertextual structures must be developed; and (c) theoretical and disciplinary assumptions within linguistics concerning possible and desirable approaches to language must be broken down and then reassembled so as to include systematic understanding of intertextual structures.

4 Writing computer programs

4.1 Introduction

In Section 3 we saw how program texts derive their function and meaning from each other. Individual program texts are linked together in different ways in complex textual environments. But these environments of texts do not magically appear out of thin air. One by one these texts have been created by many different programmers over time. Therefore, to understand how program texts come to be and function as they do, it is not enough to view individual texts only up against a background of already existing texts. To understand how program texts are created, we must also understand the social interaction of programmers working together to write these texts.

To write computer programs is a complex writing practice that includes both individual actions and collective interaction. Barton (2006: 37-49) describes writing practices in general in terms of literacy: Literacy practices are the social and cultural practices associated with the use of the written word. These practices have an individual and a social history, and they are situated in broader social relations and different domains of life. This makes it necessary to describe the social setting in which writing occurs, including the ways in which social institutions support particular practices. Literacy practices are based upon a system of symbols used for both communication and reflection - writing is a way of representing the world both to others and to ourselves.

Based on the preceding intertextual analysis and aligned with the concept of literacy practices, Section 4 discusses some of the individual acts and social practices that make up computer programming. This analysis is divided into five sections:

First, to write program texts will be described as co-writing and addressing other programmers through their program texts and functions. Programmers collaborate with other programmers when they quote and reference each others' texts.

Second, programmers anticipate that others might use their program texts. A programmer can be very concerned, for instance, that a hacker might take advantage of a glitch in his text. To avoid being hacked, the programmer therefore works hard to close any such potential glitch. Hence, programmers do not only address and adjust to existing, concrete programmers, but also to other possible, imagined programmers that might link to them in the future.

Third, the practices of generalizing other program texts are discussed. As the individual acts of linking are done in concert by many programmers, large groups of programmers are able to directly and indirectly use each others' texts and functionality. When many programmers use the same words and texts, programmers start perceiving their reference as not only being used by one other, but many other programmers. As the complexity of this social and textual space grows incomprehensible, programmers start "generalizing" both each other's texts and social roles.

Fourth, as clusters of 'generalized other' in turn become generalized, conventions for how programmers link to each other come to form programming languages. This process is described as 'grammaticalization' (Hopper 1993), a continuous, historical movement from pragmatic and lexical to syntactic and morphological structures.

Fifth, the individual programmer's use of these generalized means in programming languages is examined. As the programmer grows accustomed to addressing the generalized other, his own role and identity in relation to this abstract other also grows stronger: the means by which the generalized other operates become the means by which the programmer himself mediates his textual actions. The generalized other thus reflects back as a "how I relate to others in general". Through this process the programming languages come to form tools for thought, means by which the programmer can frame his thoughts about his program texts and their environment.

Based on the discussion of programming languages, Articles B-D are presented. One, as programming languages help us link our program texts together, programming languages are central means for social interaction and collaboration through program texts. Article B discusses how different programming language grammars can make programmers relate to each other and interact in different ways. Two, different communities of programmers establish cultural practices that are intricately related to their programming practices. Article D discusses how one such community, researchers in Information and Computer Science, share a set of engineering methods regarding how researchers should design and implement their computer programs. Three, using programming languages, a programmer can abstract a generalized image from the chaos of concrete, intertextual program webs. Doing so, the programmer uses his languages as tools for thought. These tools help him think about program texts figuratively and metaphorically; they let him see program texts work together in new ways. Article C elaborates on this role of programming language through a discussion on how programmers perceive the basic rules regulating the time and space of the virtual, run-time environment that surrounds their programs.

4.2 To program is to address others

Section 3 illustrates how computer programs form and function as intertextual webs. By calling, referring, pointing, and linking to each other's texts, programmers express program objects, methods, facts, rules, and other functionality. To write a computer program is essentially to express instances of words which link it to other programs and submerge it into webs of texts and links.

Because programmers depend so heavily on other programmers' texts, they must orient themselves toward each other. Programmers write methods, rules, and functionality not only for their external end-user, but also for their own and other programmers' intertextual use. In this sense, to program is to write together and for one another; to program is to collaborate, interact, and communicate.

Dialogic theory describes how people refer to each other in communication in terms of 'addressivity'.

> Each individual utterance is a link in the chain of speech communication. [The utterance reflects] others' utterances, and, above all, preceding links in the chain (sometimes close and sometimes […] very distant). […] The utterance is addressed not only to its own object, but also to others' speech about it. (Bakhtin 1986: 93f). [6]

All instances of words and texts are uttered for someone by someone. Words in use are always directed at others, both near and far, directly and indirectly. To communicate is to address others in text and in chains of interaction. "[A]ddressivity,

[6] Here, 'utterance' is interpreted to mean an instance of word and text written by actual human beings in concrete situations, part of and demarcated as a link in a chain of such instances.

the quality of turning to someone, is a constitutive feature of the utterance; without it the utterance does not and cannot exist" (ibid: 99).

In programming, programmers simultaneously address several other programmers. However, these other programmers are also writing programs themselves and are therefore also addressing yet other programmers. Programmers addressing each other are therefore interacting with several other programmers who in turn are also interacting with yet more others. By using each others' words, programmers can be seen as not only linking to each others' texts, but also as entering into the interaction and communication processes of the others. Programmers are in dialogue with other programmers who also are in dialogue, recursively, in a web of interconnected chains and ongoing processes.

Programmers are no strangers to the recursive relationship between direct links in program texts. On the contrary, programmers anticipate such recursive relationships every time they invoke each others' words. For instance, a Java programmer calling "`System.out.println("hello world!");`" expects these words to recursively link through the Java machine and operating system so to display "`hello world!`" on the computer screen. The functionality of the computer screen and operating system is echoed in the Java words "`System.out.println`". To address another programmer is therefore not only to address him and his text individually, but also to address the web of links that surround this other program text, the social interaction that the other programmer participate in, and the constantly evolving textual environment of the other programmers' dialogues.

4.3 Addressing the past, present, and future

> The utterance is related not only to preceding, but also to subsequent links in the chain of speech communion. When a speaker is creating an utterance, of course, these links do not exist. But from the very beginning, the utterance is constructed while taking into account possible responsive reactions, for whose sake, in essence, it is actually created. (Bakhtin 1986: 94).

All instances of words and texts are written “in anticipation of encountering response” (ibid). Words are adjusted to this anticipated response, directed toward it, building on it, and including it. To communicate is not only to address past co-writers, but also parallel and future co-writers.

An example of the importance of anticipating future encounters in programming is the meta-syntactic convention “`@deprecated`” in Java. When a programmer writes `@deprecated` in front of a method declaration, it serves the sole purpose of telling other programmers that the method is to be removed in the future. `@deprecated` labels a method as a thing of the past that is presently in use, but that will *not* be part of the future. By alerting others of his future intentions, the programmer implies (a) that other programmers currently using the method will switch their references to other alternative methods and (b) that no new references is made to the deprecated method. When the method is later removed, other programmers will hopefully have had the time to react. To simply remove the method at once could cause other systems to crash.

The webs of program links and texts do not magically create themselves or naturally appear as if dug up from the ground. The texts and links are the medium for, a manifestation of, and almost a byproduct of programmer dialogue. Program texts do not rely only on existing program functionality, but also on other programmers and their preceding interaction. Programmers anticipate and adjust to future program texts as future programming dialogues. Programmers address their users, themselves, and other programmers in past, present, and future tense.

4.4 Addressing the words of others and changing them

When programmers create new texts, they define their own, new words. Word construction is an important part of creating new programs. But programmers also use each others’ words, not only their own. The words of other programmers are used in new contexts all the time, and the experiences and intertextual links and dependencies toward these new contexts influence and gradually change the semantic, intertextual meaning of individual program words.

An example of such a gradual change in program word meaning is `java.util.Date` (SUN 2008). In Java 1.0, `java.util.Date` was given the task of both storing and parsing time and date information. As of Java 1.1, `java.util.Date` should only store time information, and parsing functionality was conceptually moved to `java.util.Calendar` (Hill 2004). The original parsing methods in `java.util.Date` were not removed, only deprecated. And, in future releases of Java, new words such as `Clock` and `ZonedDateTime` are suggested to replace both `Date` and `Calendar` (Colebourne 2008; Farnham 2008).

In linguistics, the process of gradually changing semantic meaning of words is described as ‘semantic progression’ and ‘semantic change’:

> The surface study of semantic change indicates that refined and abstract meaning largely grow out of more concrete meanings. […] The meaning of a form in the habit of any speaker is merely the result of the utterances in which he has heard it. […] If a speaker has heard a form in an occasional meaning or in a series of occasional meanings, he will utter the form only in similar situations (Bloomfield 1933: 429, 431).

As a word is used by many in new, related contexts, its general meaning is gradually broadened. One example of such broadening is the English word ‘guy’.

> Guido (Guy) Fawkes [attempted] to blow up the English Houses of Parliament on 5 November 1605. The burning on 5 November of a grotesque effigy of Fawkes, known as a ‘guy,’ led to the use of the word "guy" as a term for any ‘person of grotesque appearance’ and then to a general reference for a man, as in ‘some guy called for you.’ In the 20th century, under the influence of American popular culture, ‘guy’ gradually replaced ‘fellow,’ ‘bloke,’ ‘chap’ and other such words throughout the English-speaking world, and is also referred to both genders (i.e., ‘Come on you guys!’ could refer to a group of men and women). (Wikipedia 2008).

The process of widening the scope of word references is bidirectional. If some contexts that a word was used in are neglected by many, the word gradually loses its association to these contexts and its meaning is narrowed. One example of such narrowing is the English word ‘meat’ (cf. the Norwegian word ‘mat’) that once was used to describe ‘food’, but that now only describe ‘eatable flesh’ (Bloomfield 1933: 430).

Lastly, a word can also be extended to refer to new, unrelated contexts. “[One example] is the extension of *bede* ‘prayer’ [cf. the Norwegian verb ‘be’] to the present meaning of *bead*: the extension is known to have occurred in connection with the use of the rosary, where one *counted one’s bedes* (originally ‘prayers,’ then ‘little spheres on a string’)” (ibid.: 440).

Program words are largely created as metaphorical extensions from English and mathematics. Program words such as `String` and `Map` borrow connotations from English and mathematics. These connotations allow programmers to imagine and convey their concepts to each other. However, as these words become widely used by many different programmers in different intertextual environments, the words are gradually populated with the intentions inherent in these contexts. As the programming languages grow with their varied contexts of use, so do their program words. These intentions gradually broaden and narrow the meaning of words away from their original semantic extensions.

In many older programming languages, `String` means a list of characters. However, as platform architecture has gradually evolved, `Strings` are being viewed more as blobs of text rather than as a series of individual characters. In Java, `Strings` are therefore primarily conceptualized as blobs of text, even though the programmer may still access their content as a list of characters. Similarly, “in mathematics, if each element x in one set corresponds to a unique element $f(x)$ in another set, there is said to be a mapping from the first set to the second because if the function $y = f(x)$ is graphed, the resulting points form a kind of map” (Schwartzman 1994: 131). In Java, `Map` means a container for pairs of objects, one pointing to the other. This container thus stores two sets, where elements in one uniquely point to elements in the other. In Java, however, objects are added to the `Map` in *externally* defined pairs. The Java `Maps`’ primary role is to store these pairs. The mapping function or algorithm that prescribes how objects become paired is thus prior and external to the Java `Map` concept, not a byproduct of the `Map` itself.

Programmers strive to strike an appropriate balance between too narrow, idiosyncratic, individual references and too broad, vague, general, and polysemous references. The syntactic structure of inheritance specifically supports semantic broadening, narrowing, and extension. To create a narrower version of a class, a new class can be created that inherits the general capabilities of its super class. This subclass literally `extends` the super-class, but tailors its functionality to suit new purposes of a particular context. Similarly, general traits of a class can be extracted as an interface. This interface can be used in a more generalized sense, facilitating its use in a wider context. How and why programmers use this particular form of syntax to facilitate semantic change is a suitable topic for further study.

To write program texts is therefore to contribute to the semantic sphere with both new words and new connotations to existing words. The intertextual web of texts is also a sphere of semantic relations that are constantly changing and evolving. As idiosyncratic needs are integrated into shared intertextual environments, existing words and concepts function as clay figures that are gradually broadened, extended, and narrowed to fit different, situated needs.

4.5 To address another is to generalize the other

By joining their texts together, programmers group together. In these groups programmers establish functional, textual, and interactional relations to one another. As more and more programmers get involved in such groups, their collective webs of texts also expand. As their webs of texts expand, they grow both positively in literary vocabulary, functional ability, and social support as well as negatively in complexity and technical dependencies.

The positive consequence of a large web of words echoing diverse functionality is that many texts can link up to the same texts. Take for example the two words `not` and `member` described in Section 3.3. These two words are referred to by “bus.pl”, “BasicLibrary.java” itself, and many other Prolog program texts. Many different Prolog program texts use these two words, and the texts in different Prolog machines all describe `not` and `member` similarly. A reference to the rule `not` and `member` is therefore not only a reference to a concrete, individual text instance such as “BasicLibrary.java” or “bus.pl”, but also and at the same time a reference to many such concrete, individual text instances in general. Similar references collocate, both from many texts such as "bus.pl" that point to and use a rule described elsewhere, and from many texts such as "BasicLibrary.java" that describe a similar rule used elsewhere (cf. the corresponding generalization of literal links described in Section 3.7).

Thus, to link is not only to point directly to another program text, but also to generalize that other text by pointing to (possibly) many texts in the same way as the other text instance in question. Programmers thus address each other both as a significant other and as a "generalized other"[7] *at the same time*. Furthermore, an individual programmer does not control the scale of concreteness/generality on his own as other programmers can link their texts with his without his control. For example, programmers often build methods in an ad-hoc, sloppy manner that are not intended for general use: a method can be built for test purposes or to temporarily fill the slot for a later, better version. However, such methods can leak into the programmer dialogue for various reasons, and here other programmers may link their texts to them or mimic their behavior in ways not originally intended. Thus, only as a collective and in continuously evolving dialogues do programmers control the extent to which a link is concretized and generalized.

By addressing each other through program texts, programmers' referents are always textual figures. These textual figures act as social roles that can be filled by concrete, individual programmers. One example of such a textual figure is *jTrolog*. When I as an individual programmer write *jTrolog*, I reenact the role of a Prolog interpreter. When such roles are played out identically as a program runs over and over again on the same system, outside observers can easily mistake the referents of programmers as finite functional figures. However, programmers do not address predetermined, finite figures. As in all communication, programmers interpret and address social figures that fluctuate between a concrete and a general other, mediated in text.

[7] The term "generalized other" is first used by Mead (1932: 154): "The organized community or social group which gives to the individual his unity of self may be called 'the generalized other.' The attitude of the generalized other is the attitude of the whole community." While Mead's original description focuses on the process of generalizing other individuals' attitudes toward oneself, this thesis uses the term to describe the similar process of generalizing other individual texts and words.

> This addressee can be an immediate participant-interlocutor in an everyday dialogue, a differentiated collective of specialists in some particular area of cultural communication, a more or less differentiated public, ethnic group, contemporaries, like-minded people, opponents and enemies, a subordinate, a superior, someone who is lower, higher, familiar, foreign, and so fourth. And it can also be an indefinite, unconcretized *other* (Bakhtin 1986: 95).

The addressee of programmers are thus both other concrete programmers that has or will fill textual roles in their texts and a collective of such programmers, a generalized other programmer. To reference is not only to link to other individual programmers and their concrete program text instances or only to link to abstract categories, ideas, and functional roles, but to do both, at the same time.

4.6 Programmers build social communities of function and form

The growth of the intertextual webs is not only an opportunity, but also a problem. A constant growth of texts and functions could lead to a similar growth in the intertextual environment that accompanies an individual program. If this individual environment grows too large, technical resources such as memory capacity could collapse. Furthermore, textual environments around individual programs that are too large could flood the imagination of individual programmers that need to envisage these environments. Not overstraining the individual programmer's imagination is a constant battle.

To curb the growth of the intertextual web of links, programmers must collectively constrain their intertextual environments. They accomplish this by first forming social communities. In these social communities, concrete, individual programmers share overlapping needs to address certain functionality in certain settings and therefore an interest to envisage and constrain their textual environment similarly. The communities built around social groups of programmers thus also function as communities of function. Furthermore, to constrain their textual environment, these communities compromise and agree on intertextual borders. These borders encapsulate mainly a series of generalized others, words, texts, and conventions used by many in the communities. The communities of function thus also form communities of form.

‘Grammaticalization’ is a linguistic theory that describes how social communities develop their textual forms and languages in general. “[Grammaticalization refers] to the change whereby lexical items and constructions come in certain linguistic contexts to serve grammatical functions and, once grammaticalized, continue to develop new grammatical functions” (Hopper 1993: 18). Grammatical forms are neither static nor given prior to use. Instead, grammatical forms are either analogically drawn from another language or evolve over time as recurring textual habits and forms collocate into linguistic conventions (ibid.: 22).

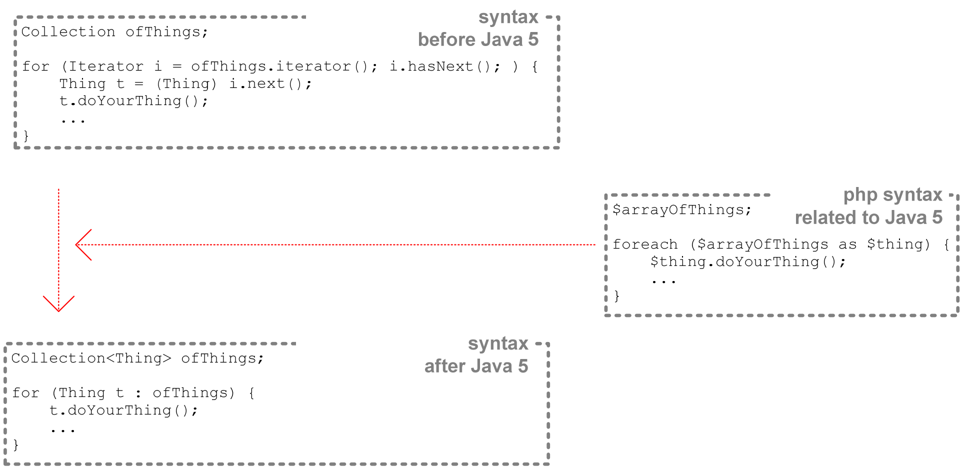
Programming languages draw many of their grammatical forms from mathematics. For example, the Prolog programming language is “an abbreviation of ‘PROgrammation en LOGique’” and adapted mathematical principles about formal logic to the processing of natural language text (Bergin 1996: 331-52). Another example is the programming language APL that was created on the premise that “the advantages of executability and universality found in programming languages can be effectively combined, in a single coherent language, with the advantages offered by mathematical notation” (Iverson 1980: 445). However, using the theory about grammaticalization, grammatical conventions in programming languages can also be understood as prompted by the recurrence of form in certain direct and indirect intertextual links within programmer communities. As communities evolve, so too do their conventions for referring to their generalized others. The many grammatical structures supporting classes, object orientation, libraries, encapsulation, etc. can be understood as socially evolved to suit the situations, needs, and intertextual constraints and opportunities in computer programming (cf. Article B; Bergin 1996).

Over time, the generalized others evolve not only as lexical symbols, but also as grammatical forms. Recurring needs and habits for addressing certain generalized others in certain ways in turn become generalized. Language conventions form on top of both intertextual links and symbolic links and are generalized as conventions for word order, word classes, morphology, and other grammatical structures. Thus, programmer communities generalize their generalized others into larger hierarchies of form that make up their programming languages.

4.7 Article B: Morphology and Power

Programming languages are constantly evolving. For instance, a program library method written in the 1970s can be embraced and used by a social community in the 1980s. When the library method is later upgraded and rewritten in the 1990s, the name and structure of the method from the 1970s must remain the same so as not to cause other programs that have become reliant on it to crash. The past of program texts is written into the present which in turn points to and is sewn into future program texts. The historical, social settings are integrated in program texts and linguistic conventions. To further study the languages that programmers presently use, we must therefore look at their history and social context of use. Article B illustrates how such historical and social settings can be integrated into a study of program languages.

Programming language grammar is also evolving. One example of a recent development in Java is the syntactical construct `foreach`. Up to Java version 5, an often recurring practice was to iterate on a `Collection` using an `Iterator` and the syntactical constructs `for` and `while`, and then cast the output from the Iterator. However, as these operations can be accomplished by the simpler `foreach` structure that abstract out the use of the Iterator and cast operation, the operation is simplified. The `foreach` syntactic structure thus both (a) reduce the weight of the original Java syntax by (b) analogically drawing in a syntactic structure known in related programming languages such as PHP. The change is illustrated in figure 14, and the grammaticalization of `foreach` is an example of syntactical structures that may well be studied to give us insight into the historical, social, and linguistic evolution in programming languages. Similarly, Article B illustrates how a new morphological form can be analogical transferred into a programming language and how a study of program languages can analyze such a process.



```
Collection ofThings;

for (Iterator i = ofThings.iterator(); i.hasNext(); ) {
    Thing t = (Thing) i.next();
    t.doYourThing();
    ...
}
```

```
$arrayOfThings;

foreach ($arrayOfThings as $thing) {
    $thing.doYourThing();
    ...
}
```

```
Collection<Thing> ofThings;

for (Thing t : ofThings) {
    t.doYourThing();
    ...
}
```

Figure 14: The new `foreach` *loop syntax in Java 5 as a grammatical change that simplifies a much used practice by borrowing an analogue syntactic structure from a surrounding language.*

The evolution of programming languages is not only influenced by practical needs. Social and cultural power structures also play a big role in shaping programming language grammar. “The importance of power relations in patterning literacy practices means literacy researchers need to develop an understanding of the processes of power in the society in which they are studying, and to take a critical approach, in the sense of making visible the power relationships which are often hidden” (Barton 2006: 52). Article B illustrates such a hidden relationship between language and power. Existing morphological forms in object oriented programming languages promote the use of words by one particular group of programmers at the expense of words being constructed by other groups. This power structure, and particularly the relationship between grammatical means and social interaction, is largely tacit and hidden from sight.

Lastly, programming languages address future needs and are oriented toward the future. Therefore, programmers are constantly looking to enhance their practices, *including* their languages, and interact with each other in new and better ways. Unlike in studies of so-called natural languages, the linguistic communities of programmers often welcome suggestions of new linguistic practices and forms. Research into programming languages should therefore not restrain itself to descriptive studies of the past and present, but open up for problems in future scenarios and actively partake in the constructive evolution of programming languages. Article B illustrates how studies into programming languages can use a subjective, experimental approach to both highlight social problems associated with past and present language structures and actively present alternative grammatical forms that change the social and intertextual dynamics so as to accommodate these problems.

4.8 Article D: Tacitly developed methods in ICS

There are many different communities of programmers. Social groups as diverse as open source communities, boy-room hackers, suit-and-tie business consultants, and academics all group together and establish their social identity through various practices. To these communities, program texts are not only functional, but also pretty, ugly, smart, and elegant. “[C]omputer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty” (Knuth 2007: 44).

To understand the particular social and technical currents that run through these cultures, individual studies directed at each culture are needed. Article D can be understood as one such study. To comply with an established agenda in the discipline Software Engineering, Article D addresses questions regarding research methodology in ICS. Research methods in ICS are largely tacit and not explicitly discussed. The article performs a three step analysis of 93 doctoral theses in ICS at NTNU, and this analysis illustrates: the extent to which research methods remain implied, the problems of using existing research methodology to describe these tacit research methods, and how an alternative approach might reveal methodological trends in the field. This analysis highlights how researchers in the ICS community actively participate in system development and computer programming. Academics working with computer programs and texts share common cultural practices that to a large part constitute various practices of writing computer program texts. As such, Article D presents the practices of one community of programmers and then shows how their practices and actions related to computer programming are woven into other social, cultural, and community practices.

4.9 Programming languages as tools for thought

When programmers write program texts, they essentially compose their programs as symphonies of quotes to and from other program texts. These other texts, real or imagined, form the building blocks by which programs can be made, and these building blocks are essentially all the programmer can "think into" his program.

These texts are written by many different people, at different times and places, and with different purposes. To transcend these barriers of difference, programmers develop programming languages with words and grammars that directly facilitate references to and the arranging of quotes in program texts. By organizing their conventions for referring to one another in programming languages, programmers are able to address a large group of texts. Programming languages dramatically simplify the structure of intertextual links in large textual environments.

It is the intertextual sphere, that is, the concrete and generalized others, we can address that enable us as programmers to imagine our program problems and solutions. Furthermore, it is the limits of the texts that we can foresee surrounding our program texts that limit our imagination of future programs. The meaning of the `bus`-facts and `plan_travel`-rules in "bus.pl" is therefore dependent on us imagining and seeing "bus.pl" in an intertextual program context of use, and the program texts and setting that surrounds them inform our understanding of words such as `plan_travel` and `bus`.

When programmers express their own purposes as functions of other programmers' texts, not only the lexical, but also the grammatical resources made available in the programming language enable and constrain the programmers. These enabling constraints of the programming languages grammar can be understood as the implicit rules of the programs' environments that come into play when programs run. And precisely because these rules dictate how programs run, they indirectly both enable and constrain what programs programmers can envisage running. As such, programming languages can be understood as not only tools for realizing computer programs, but also as tools for conceptualizing computer programs. Programming languages shape how programmers envision their programs.

4.10 Article C: Forms of time and chronotope in programming

To further illustrate how programming languages enable and constrain the imagination of programmers, the analysis of Prolog variables in Section 3 is extended.

Prolog variables are so-called ‘logical variables’ (Sterling 1986: 13). Variables in all programming languages are linguistic proforms and function as placeholders for numerical values and pointers to predicates or other variables. Logical variables are no different from variables in other programming languages in this regard. However, once assigned to a value, logical variables cannot be overwritten and reassigned. Logical variables differ from variables in most other programming languages when we view the result that a program produces as a linear series of steps.

In programming languages like Java, variables can be reassigned continuously. Once the value of a logical variable is set, however, it cannot be changed at a later point in the same linear series that leads to the solution. Take for example the following statements in Prolog: “`Var is 2, Var is Var + 1`”, and Java: “`int var = 2; var = var + 1;`”. Both examples form a similar, linear series of two steps attempting to use a variable to increment a value from “`2`” to “`3`”. However, whereas the Java statements would succeed and bind “`var`” to “`3`” in the end, the Prolog statements would fail: the “`Var`” in the Prolog example is first bound to “`2`”, and being a logical variable, it therefore cannot later be changed to “`3`” in the second step. This restriction of not being allowed to change the reference of a proform is somewhat counter-intuitive, but by including this restriction into one’s intuition, programmers start to imagine things differently.

The consequence of this restriction on Prolog variables, within the context of the other grammatical and lexical conventions of Prolog, is that every step in anticipated, imagined solutions can be viewed as beads in a bowl, as opposed to beads on a string. The syntactic conventions in Prolog specify no dependency that requires the individual steps in a solution to be arranged in a given sequential order. Syntactically, sequence of steps is of no importance.

Before proceeding further with this analysis, two qualifications about sequence of steps in Prolog need to be made. First, although the sequence of steps is not restricted syntactically, the semantics of individual Prolog predicates may rely on individual steps being performed in a specific sequence. Take for example the Prolog statement: “`A is`

`B - 1, B is 3`”. As both “`A`” and “`B`” variables are bound only once and the syntactic order of the two steps is of no importance, “`A`” should be “`2`”. However, the predicate “`is`” semantically requires that its right side parameter represents a numeric value when it is first encountered. As “`B`” is first bound to “`3`” in step two, the semantic interpretation of the step one fails. Changing the order of the two steps into “`B is 3, A is B - 1`” resolves this semantic dependency. Second, although there are no syntactic dependencies associated with the sequential order of solution steps, predicates separated by comma, there are syntactic dependencies associated with the sequential order of full Prolog rules, predicates separated by period. Prolog thus has only a partially a-linear syntax.

```
next_to(X,Y,List) :- iright(X,Y,List).
next_to(X,Y,List) :- iright(Y,X,List).

einstein(Houses,Fish_Owner) :-
   '='(Houses, [[house,norwegian,_,_,_,_],_,[house,_,_,_,milk,_],_,_]),
   member([house,brit,_,_,_,red],Houses),
   member([house,swede,dog,_,_,_],Houses),
   member([house,dane,_,_,tea,_],Houses),
   iright([house,_,_,_,_,green],[house,_,_,_,_,white],Houses),
   member([house,_,_,_,coffee,green],Houses),
   member([house,_,bird,pallmall,_,_],Houses),
   member([house,_,_,dunhill,_,yellow],Houses),
   next_to([house,_,_,dunhill,_,_],[house,_,horse,_,_,_],Houses),
   member([house,_,_,_,milk,_],Houses),
   next_to([house,_,_,marlboro,_,_],[house,_,cat,_,_,_],Houses),
   next_to([house,_,_,marlboro,_,_],[house,_,_,_,water,_],Houses),
   member([house,_,_,winfield,beer,_],Houses),
   member([house,german,_,rothmans,_,_],Houses),
   next_to([house,norwegian,_,_,_,_],[house,_,_,_,_,blue],Houses),
   member([house,Fish_Owner,fish,_,_,_],Houses).

iright(L,R,[L,R|_]).
iright(L,R,[_|Rest]) :- iright(L,R,Rest).

:- einstein(X,Y).
```

Figure 15: The Prolog program “einstein.pl” (Ørstavik 2008a).

Figure 15 above illustrates a Prolog program solution, “einstein.pl”, where the sequence of steps is arbitrary. The purpose of this program is to solve a riddle. This riddle is composed as a set of clues such as: (a) in the house on the left there lives a Norwegian; (b) in the house in the middle the owner drinks milk; (c) the Brit lives in the red house; etc. The Prolog program “einstein.pl” describes these clues as a set of `member`, `iright`, and `next_to` predicates, and by running these predicates one after the other, a complete picture of who lives next to whom, who drinks what, etc. is created.

While the content of each clue is important, the order in which the clues are given is not. Neither the `member`, `iright`, nor `next_to` predicate has any semantic restriction that constrain its variables, and so the clues can be shuffled around randomly and still produce the same set of results. If we consider each clue like a bead, the beads are gathered in a bowl, not on a string.

To imagine the steps in a Prolog solution like beads in a bowl is strange and counter-intuitive for those of us more familiar with the Java way of understanding variables. The environment in which the text fragments (the beads) is envisaged, enable us to spread them out and view them individually as points in space (the bowl). But there is essentially no time-line connecting the text fragments (no string), only the shared space connects them.

In Article C, "Forms of time and chronotope in computer programming: Run-time as adventure time?", the sets of rules regulating the time-and-space dimension that programmers imagine surrounds their programs are described as ‘chronotopes’. The conventions regulating the logical variables in Prolog thus create a “beads in a bowl” chronotope in which Prolog programs such as “enstein.pl” can be imagined operating. Within this “beads in a bowl” chronotope it does not matter which text fragments and operation is considered first and last, they must all be true in a solution anyway and no variables or other syntactic element depend on them being handled in a specific order. Thus, the “beads in a bowl” chronotope essentially comprehends the environment as if all the individual operations are performed all at once, synchronically.

The “beads in a bowl” chronotope is a deep-rooted, syntactically borne imaginary tool. On top of this chronotope we can build semantic constructs that for instance stipulate an order between some of the beads in the bowl, a semantic string that can rely on a logical variable to connect different beads. But when no such semantic dependencies are there, the synchronic characteristics of the environment reappear.

The example Prolog variables, Article C, and the previous intertextual analysis show how grammar, words, and other linguistic conventions creep into our imagination and shape our ideas. Syntax molds our ideas in its own image, regardless of whether we are speaking and thinking in ancient Greek or writing and thinking in Java or Prolog. The words and texts of others form imaginary building blocks that we can reassemble in our minds by simple text references. “Programming languages, because they were designed for the purpose of directing computers, offer important advantages as tools of thought” (Iverson 1980: 445).

5 Future studies of the language of programmers

5.1 The language of programmers and their writing practices

In the preceding Sections and the subsequent Articles, the practice of writing computer programs is illustrated as social and cultural processes driven by human beings. Computer programs are basically texts, and these texts echo both social interaction and each other in intertextual webs. Computer programmers write program texts in dialogue with each other, and the symbolic, literal, real and imagined, direct and indirect links that fill computer programs with meaning and functionality, are essentially a reflection of programmers' dialogues.

The intertextual webs that make up computer programs and systems are written in a variety of languages. These programming languages are intricately woven into the social interaction between programmers: on the one hand, programming languages have the power to shape how programmers interact with each other, and, on the other hand, even individual programmers can shape their languages' grammar and words to suit their own particular purposes. Thus, programming languages are not only technical, but simultaneously human, social, and cultural constructs.

Programming languages play an important role in shaping how programmers as humans shape their conceptions of their problem and solutions. Programmers write words, methods, and libraries that in turn come to form building blocks that other programmers use to compose their description of problems and solutions. Syntactic rules of programming languages function not only to constrain and order the technical constructs, but also to enable the conceptualization of complex virtual worlds. Even as seemingly basic, atomic, and innocent linguistic structures as morphology can influence how the individual programmer views the words of another programmer and, thus, how he positions himself against the other. "[T]he proper, primary aim of programming is, not to produce programs, but to have the programmers build theories of the manner in which the problems at hand are solved" (Naur 1985: 253).

In sum, this PhD presents and explores a new interdisciplinary platform for understanding the language of programmers and the practice of writing computer programs. However, being new, this platform is built on questions rather than answers.

These questions both address surrounding disciplines and echo other related interdisciplinary studies of computer programming. To illustrate these questions, Section 5.2 briefly introduces some of these related interdisciplinary areas and how we might approach them. These new questions will help raise awareness about writing program texts and programming languages, the importance of understanding these phenomena, and, finally, how the findings of this PhD study can be used in future work.

5.2 Technical results

As mentioned in Section 1.4, this PhD has actively participated in two real life open source projects: *jTrolog* (Ørstavik 2008a) and *SIMAS* (Ørstavik 2008b). These projects have not only resulted in the "lived experiences" that are used in the Articles and Sections 2-4, but also in two software platforms. These platforms are briefly presented here.

> *jTrolog* is a small, fairly fast and simple Prolog Interpreter in pure Java. *jTrolog* is currently fairly consistent both in design and functional performance. The primary goal of *jTrolog* is to be easily understandable from a Java programmers' perspective without compromising too much on speed and memory performance. (Ørstavik 2008a).

jTrolog started out as a branch from the *tuProlog* (2008) project. Currently, *jTrolog* is the fastest open source Prolog interpreter written in pure Java and almost fully compliant with the Prolog specification (ISO/IEC 1995).

The *SIMAS* platform is a run-time environment for writing so-called multi-agent systems. A multi-agent system is a collection of independent programs (called agents) that collaborate in order to solve larger tasks. The *SIMAS* platform is written as an extension of Java 1.4, running agents as individual Java applications.

The main challenge for multi-agent system is to create a simple conceptual model for (a) the agents' individual and collective functionality and (b) the interpretation of messages sent between agents. To create such a simple conceptual model, the *SIMAS* project has developed a dialect of Java called 'theJ'. As discussed in Articles B and C, theJ enables the programmer to envisage the run-time environment in a way better suited to the particular challenges confronting multi-agent systems. The *SIMAS* project is a proof-of-concept platform that illustrates both the opportunities *and* problems associated with writing *and* running a ‘genre of computer programming’ still under construction.

5.3 Questions for future work

In the article “Computer Programming as an Art”, Knuth (2007) illustrates how describing computer programming as a human, social, and cultural phenomenon deviates from established disciplinary conventions. Within established conventions in both the humanities and ICS, a dichotomy between so-called natural and formal languages exists. In practice, this dichotomy hinders both computer scientists and applied linguistics from studying programming languages as human and social phenomena. The dichotomy is based on an assumption that formal languages operate under different constraints than natural languages. They are ‘un-natural’ in the sense that they are either discovered or artificially constructed, unambiguous, and finite (Chomsky 1963; Mateescu 1997), while natural languages are culturally formed through social interaction, highly ambiguous, and have an infinite meaning potential. From this premise, a second assumption follows that socially oriented linguistic theories, models, and methods can contribute little to an increased understanding of formal languages, and vice versa. The findings of this PhD study, however, question the usefulness of this set of assumptions.

First, Sections 3 and 4 and Article B and C illustrate how programming languages can be perceived as diachronically evolving, driven by social and cultural forces, rather than simply invented or derived from Newtonian laws. Section 3 and 4 and Articles A, B, and C also illustrate how the intertextual structure of program texts and language reflects social interaction and culture. Thus, this study finds that social dynamics and human culture play an important role in both the use and evolution of programming languages: programming languages resemble natural languages more than the dichotomy between natural and formal languages assumes. The strict tenability of the dichotomy between formal and natural languages is therefore questionable.

Second, to describe these human, social, and cultural phenomena of programming languages using established dialogic theories and models were found to be very useful. Applying linguistic theories to the domain of programming languages both (a) illuminated phenomena that thus far have eluded research using only technical theories, models, and methods and (b) produced new solutions to unsolved practical problems. Applied linguistic theories are useful and suit the domain of programming languages, and to deny such theories a role in this domain is counterproductive.

If the set of assumptions underpinning the dichotomy between natural and formal languages is false, a whole series of established language and communication theories can be applied in the domain of programming languages. In principle, all intellectual questions regarding natural languages might be directed at programming languages as well. Many such connections between linguistics and programming languages have already been made. Most notably, Chomsky and Schützenberger (1963) makes the connection between linguistics and mathematics, which formed "a mathematical theory of language in which I could use a computer programmer's intuition" (Knuth 2003). Below, some other examples of such established and potential connections are presented.

In diachronic linguistics, questions like: how exactly do these processes of semantic and grammatical change occur; how do so-called natural and formal languages influence each other; what social, cultural, and situational forces influence these processes of language change; and how can changes in language influence programming culture? Mateescu (1997) illustrates the relevance of diachronic theories for formal languages in general. So, to explore the evolution and "history of programming languages" (Bergin 1996) using theories describing the social and cultural influence on historical change in languages (Hopper 1993) is likely to extend the insight of both sides, as well as contribute to the growth and health of programmers' practices and cultures.

Similarly, socially oriented language and communication studies work with questions such as: what role does language play in social interaction; how are social interaction and culture reflected in language, and vice versa; and can intertextual patterns and structures be understood as manifestations of interwoven chains of social interaction, and if so, how can we study these social and textual chains/ongoing processes (cf. Article A)? The intersection between social interaction and text is also at the forefront in computer programming, and programmers struggle daily with problems such as: how can chains of dependencies between computer programs best be managed (cf. Article B); what is the best way to modularize and encapsulate computer programs; how can whole program texts and parts of program texts best be reused; and how should computer program texts be structured so that other programmers can update and maintain them as simply as possible? In ICS, different approaches have been taken to these problems: should computer programs be built using in hierarchical organized social group and/or according to well-structured procedures, or should computer programmers collaborate in a more bottom-up and/or dynamic fashion? Working with these practical problems, computer programmers have developed several empirically based models and insight into the intersection between interaction and text. Raymond (1999) discuss, for example, the social organization of open source projects in terms of a "bazaar" vs. a "cathedral" metaphor (cf. also a review of this study in Krishnamurthy 2002). Another example is the agile software movement that value "*individuals and interactions* over processes and tools" (Agilemanifesto 2008). These studies and this thesis show that to combine social studies of language and computer programming can help us answer questions such as: "what can we learn about programming languages from the social aspects of mathematics and natural languages" (Naur 1975: 676), and what can we learn about the interplay between social interaction, language, and text from programming languages and the practices of writing program texts?

Cognitively oriented, linguistic studies question: is our language structure dependent on our mental structures, and if so, how; do language structures influence our mental structures, and if so, how; how are language structures cognitively handled; and can language function as a tool for thought, and if so, how? These questions align with many similar questions in computer programming such as: what are the intrinsic structures of programming languages, and what may the next generation of programming languages look like; how do programmers learn to program; and how does different programming languages shape our perception of problems and solutions, and how can we actively control programming languages as tools for thought so as to build better computer programs smarter and faster? Compared to natural languages, computer programming is an exotic domain in which language systems and thinking interact, and so to apply linguistic theories that address the intersection between language and thinking on computer programming, can extend both our insight into the practice of writing, reading, and thinking about computer programs as well as our understanding of the relationship between thought and language in general.

Finally, computer programming is as much about learning as it is about writing. Computer programming is a constant learning process. In a study called “Promoting computer literacy through programming Python”, Miller (2004: 237-9) finds that “not only is programming *like* writing, [but] programming *is* a form of writing. […] Thus, the first important consideration for teachers of Python as a first computer language is to think of programming as writing.” To understand more about the process of writing computer programs, we must therefore also study the processes of learning computer programming. And, equally, to understand more about the social and individual learning practices involved in computer programming, we must learn more about the practice of writing computer programs and computer programming in general. “The chief goal of my work as educator and author is to help people learn how to write beautiful programs” (Knuth 2007: 39).

5.4 The language and practices of programmers

As a general ambition, the humanities aim to tell us why and how we as humans do what we do. This ambition fits well with programmer culture which is constantly searching for new models and meta-languages to describe what they do and better practices for writing computer programs. The humanities and programmers thus share both common interests and problems.

This PhD opens up many questions about the language of programmers and their practices of writing program texts. Some of these questions specifically concern the practice of writing program texts. Other questions are connected with the language of programmers in general. As the role of programmers, their technology, and their art becomes ever more important in our society, we need to pursue these questions, not only for the sake of applied science, but also in order to understand who we are becoming.

6 References

Agilemanifesto 2008. Manifesto for Agile Software Development. Retrieved 20.11.2008 from http://agilemanifesto.org/.

Bakhtin, M. 1981. *The Dialogic Imagination*. Austin: University of Texas Press.

Bakhtin, M. 1984. *Problems of Dostoevsky's poetics*. Minneapolis: University of Minnesota Press.

Bakhtin, M. 1986. The problem of speech genres. In C. Emerson and M. Holquist (ed.) *Speech Genres and Other Late Essays* (pp. 60-102). Austin: University of Texas Press.

Barton, D. 2006. *Literacy: an introduction to the ecology of written language*. Oxford: Blackwell Publishing.

Bergin, T., and Gibson, R. (eds.) 1996. *History of Programming Languages-II*. New York: ACM Press.

Bloomfield, L. 1933. *Language*. London: George Allen & Unwin LTD (reprint 1967).

Bogdewic, S. P. 1992. Participant Observation. In B. F. Crabtree and W. L. Miller (eds.) *Doing Qualitative Research* (pp. 45-69). London: Sage Publications.

Bostad, F., Brandist, C., Evensen, L., and Faber, H. (eds.) 2004. *Bakhtinian Perspectives on Language and Culture*. New York: Palgrave Macmillan.

Cassirer, E. 1953. *The philosophy of symbolic forms: Language*. New Haven: Yale University Press.

Cassirer, E. 2006. Form og teknikk. In I. Folkvord and A.S. Hoel (eds.) *Form og teknikk: utvalgte tekster* (pp. 87-142). Oslo: Cappelen Akademisk Forlag.

Chivers, C. 2007. Russia Plants Underwater Flag at North Pole. *The New York Times*. New York.

Chomsky, N., and Schützenberger, M. 1963. The algebraic theory of context-free languages. In P. Braffort and D. Hirschberg (eds.) *Computer Programming and Formal Systems* (pp. 118-161). Amsterdam: North-Holland Publishing Company.

Colebourne, S. 2008. JSR 310: Date and Time API. Retrieved 20.11.2008 from http://jcp.org/en/jsr/detail?id=310.

Evensen, L. 1991. Nytte og kritisk relevans: noen refleksjoner over anvendt humanistisk forskning. *Vitenskapsteoretisk Forum/Modernitetsprogrammet*. AVH, Trondheim.

Evensen, L. 2002. Convention from Below: Negotiating Interaction and Culture in Argumentative Writing. *Written Communication* **19**(3), 382-413.

Evensen, L. forthcoming. *Applied linguistics - toward a new integration?* London: Equinox.

Evensen, L., and Hoel, T. (eds.) 1997. *Skriveteorier og skolepraksis*. Oslo: Cappelen Akademisk Forlag.

Farnell, B., and Graham, L. R. 1998. Discourse-Centered Methods. In H. R. Bernard (ed.) *Handbook of Methods in Cultural Anthropology* (pp. Lanham: AltaMira.

Farnham, J. 2008. JSR 310: A New Java Date/Time API. Retrieved 20.11.2008 from http://today.java.net/pub/a/today/2008/09/18/jsr-310-new-java-date-time-api.html.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2000. *The Java Language Specification*. Boston: Addison-Wesley.

Heidegger, M. 1997. Appendix IV: Davos disputation between Ernst Cassirer and Martin Heidegger. In R. Taft (ed.) *Kant and the Problem of Metaphysics* (pp. 193-207). Bloomington: Indiana University Press.

Hill, P. 2004. Is java.util.Date Deprecated? Retrieved 20.11.2008 from http://www.xmission.com/~goodhill/dates/datedeprecation.htm.

Holquist, M. 1990. *Dialogism: Bakhtin and His World*. London: Routledge.

Holst, G. 2008. JLog Homepage. Retrieved 9.11.2008 from http://jlogic.sourceforge.net/.

Hopper, P., and Traugott, E. 1993. *Grammaticalization*. Cambridge: Cambridge University Press.

Humboldt, W. 1999. *On Language: On the Diversity of Human Language Construction and Its Influence on the Mental Development of the Human Species*. Cambridge: Cambridge University Press.

ISO/IEC 1995. Prolog - Part 1: General core (ISO/IEC 13211-1:1995(E)). *Information technology - Programming languages*.

Iverson, K. 1980. 1979 ACM Turing Award Lecture: Notation as a Tool for Thought. *Communications of the ACM* **23**(8), 444-465.

Knuth, D. 2003. *Selected Papers on Computer Languages*. Stanford: CSLI.

Knuth, D. 2007. Computer Programming as an Art. In *ACM Turing award lectures* (pp. 33-46 (1974)). New York: ACM.

Krishnamurthy, S. 2002. Cave or Community?: An Empirical Examination of 100 Mature Open Source Projects. *First Monday* **7**(6).

Lindholm, T., and Yellin, F. 1999. *The Java Virtual Machine Specification*. Boston: Addison-Wesley.

Linell, P. 1998. *Approaching dialogue: talk, interaction and contexts in dialogical perspectives*. Amsterdam: John Benjamins.

Linell, P. 2005. *The Written Language Bias in Linguistics: Its nature origins and transformations*. London: Routledge.

Mateescu, A., and Salomaa, A. 1997. Chapter 1. Formal Languages: an Introduction and a Synopsis. In G. Rozenberg and A. Salomaa (eds.) *Handbook of Formal Languages* (pp. 1-39). Berlin: Springer.

Mead, G. 1934. *Mind, Self & Society from the Standpoint of a Social Behaviorist: From the Standpoint of a Social Behaviorist*. Chicago: University of Chicago Press.

Miller, J. A. 2004. Promoting computer literacy through programming Python. *Department of Education*. Ann Arbor: The University of Michigan. **PhD:** 288.

Naur, P. 1975. Programming Languages, Natural Languages, and Mathematics. *Communications of the ACM* **18**(12).

Naur, P. 1985. Programming as Theory Building. *Microprocessing and Microprogramming* **15**.

Poole, B. 1998. Bakhtin and Cassirer: The Philosophical Origins of Bakhtin's Carnival Messianism. *South Atlantic Quarterly* **97**(3), 537-578.

Raymond, E. 1999. *The Cathedral and the Bazaar*. Sebastopol: O'Reilly.

Rickman, H. P. (ed.) 1979. *W. Dilthey: Selected writings*. London: Cambridge University Press.

Rommetveit, R. 1992. Outlines of a dialogically based social-cognitive approach to human communication and cognition. In A. H. Wold (ed.) *The dialogical alternative: Towards a theory of language and mind* (pp. 19-44). Oslo: Scandinavian University Press.

Schwartzman, S. 1994. *The Words of Mathematics: An Etymological Dictionary of Mathematical Terms Used in English*. Washington: The Mathematical Association of America.

Sterling, L., and Shapiro, E. 1986. *The Art of Prolog: Advanced Programming Techniques*. Cambridge: MIT Press.

SUN 2008. Date (Java Platform SE 6). Retrieved 20.11.2008 from http://java.sun.com/javase/6/docs/api/java/util/Date.html.

tuProlog 2008. Home - tuProlog. Retrieved 4.12.2008 from http://alice.unibo.it/xwiki/bin/view/Tuprolog/.

Wielemaker, J. 2008. SWI-Prolog's Home. Retrieved 9.11.2008 from http://www.swi-prolog.org/.

Wikipedia 2008. Semantic progression. Retrieved 3.11.2008 from http://en.wikipedia.org/w/index.php?title=Semantic_progression&oldid=246178395.

Ørstavik, I. 2008a. jtrolog: Project home. Retrieved 20.09.2008 from http://jtrolog.dev.java.net/.

Ørstavik, I. 2008b. simas: Project home. Retrieved 11.11.2008 from https://simas.dev.java.net.