Stuttgart Media University
Stuttgart, Fall 2019

Do-it-yourself information software

a naturalistic programming environment for end-users

Paul Sonnentag

First Supervisor: Prof. Walter Kriha

Second Supervisor: Mariano Guerra

Stuttgart Media University

Acknowledgements

First and foremost, I want to thank my supervisors Prof. Walter Kriha and Mariano Guerra. I especially appreciate Prof. Kriha for always encouraging me during my studies to explore new ideas. He was a great enabler for many unconventional projects besides this bachelor thesis. I'm also grateful for the many conversations I had with Mariano, which had a huge impact on the ideas I've developed in my thesis. During the past year, he has become a mentor to me, and I hope we can continue this relationship in the future.

I also want to thank my family, who supported me throughout my studies and my girlfriend, Christine.

Abstract

This paper tries to bridge the dichotomy that exists in software today where there is a strict separation between the people who use software and the people who build software. This paper identifies the GUI pattern as a major cause of this problem. As an alternative, this paper proposes a concept for a programming environment based on natural language called Garlang.

The proposed solution focuses on the category of information software. This is software that helps the user to learn things, to get answers to a question, compare different alternatives, and come to a conclusion.

Instead of the software landscape we see today, where the end user has to pick a ready-made solution, Garlang proposes an ecosystem where developers provide tool-kits that can solve problems in a specific problem domain and the end user can mix and match them to assemble the tool that solves exactly the problem they have.

Keywords – naturalistic programming, end user development, Datalog, logic programming

Contents

List of Figures

1 Introduction

1.1 Introduction

Today the users of computers are separated into two classes: programmers and end users. End users are mostly restricted to the role of a consumer. They cannot build custom tools to solve their problems. Building tools is the role of experts, which makes the creation of tools expensive and limits the customization of existing tools to fit the user's needs. Because the cost is high the only economical way to build software tools is to either serve a large number of people so the cost can be divided between them or target smaller user groups who are in turn willing to pay a large amount of money for a custom solution. This situation leaves many people who have particular needs without appropriate tools because it is not cost-effective to produce software for them. The more mainstream user is not necessarily served well by this system either. Even though there are tools that solve their problem, the tools might be too complex or too simple to fit their exact needs. Because developers are trying to capture big markets, they have to unify the use cases of many users, which leads to compromises for the individual user.

This paper proposes an alternative concept that enables software tools that are moldable by the user. This new kind of software could turn the strict separation between consumers and creators into a continuum. The goal is to build tools that the users can adapt to their requirement, which would empower them to solve problems more efficiently.

1.2 Democratizing Programming

In today's culture there is a movement that pushes the idea that as the importance of software is increasing, programming will become an essential skill for everyone. This idea is propagated by authors like Marc Prensky who claims programming is going to be a new kind of literacy analogous to the history of textual literacy where reading and writing used to be a specialized skill practiced by scribes (Prensky (2006)). Politicians like the former president of the United States Barack Obama started to turn this idea into policy with his "Computer Science for all" Initiative (Obama (2016)). Organizations like the nonprofit organization Code.org[1] try to provide the platform for educators and students to learn the basics of computer science. There are benefits to teaching coding to children, but it is questionable if these efforts will give people any new abilities that are currently only accessible to professional programmers. There is a big leap from learning about basic control structures and loops in a toy environment to being able to solve real-world problems.

This paper argues that forcing non-programmers to adapt to a world built by professional programmers is a mistake. Instead, an alternative computing environment is necessary that looks very different from professional programming environments today to make the power of computing accessible to more people.

The work of Victor (2006) is helpful to rethink the role of software. In his paper, he lays out a framework to understand the purpose of software. The author proposes three categories of software derived from the basic human activities that they facilitate: Information software (learning), Manipulation Software (creation), and Communication Software (communication). He claims that most software falls in the category of information software because most of the time people are less interested in creating but instead want to use the computer to get answers to a question, compare different alternatives and come to a conclusion. His criticism is that the focus on interaction is misguided in the context of information software because the user is not interested in the manipulation of the underlying model; They care about the conclusion they can draw from it. He illustrates this with calendar applications as an example of how the learning needs of the user are not satisfied by applications today.

> For example, consider calendar or datebook software. Many current designs center around manipulating a database of "appointments," but is this really what a calendar is for? To me, it is about combining, correlating, and visualizing a vast collection of information. I want to understand what I have planned for tonight, what my friends have planned, what's going on downtown, what's showing when at the movie theater, how late the pizza place is open, and which days they are closed. I want to see my pattern of working late before milestones, and how that extrapolates to future milestones.

[1]https://code.org

He proposes that instead of designing information software as interactive machines; they should be designed as information graphics with the additional ability that they can adapt to the context of the user. This paper uses his categorization of software with emphasis on information software specifically but instead of focusing on how designers should design software the goal is to find an interface that does not force software in a fixed shape. The users should be able to mold the tools they use and even build new tools by recombining existing solutions.

1.3 The current state of information software

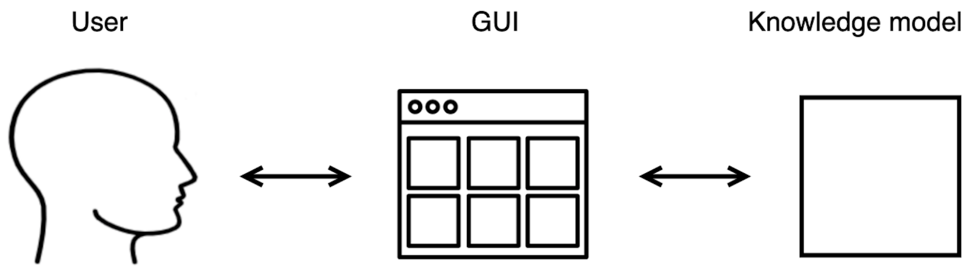


Figure 1.1: User interaction with software

Digital representations are inherently shapeless, and they can be presented in any way that's most useful to the user. This flexibility is not conveyed in most software systems today. The dominant paradigm is to bundle software into discrete apps which have a specific purpose and a fixed shape that cannot be altered by the user in a significant way. We can explain the causes and effects of this limitation by looking at how information software applications work in different scenarios.

When using information software, the goal of the user is to learn something. Therefore, the starting point is the question that the user has. As example, consider a user who is planning a trip by train. Their questions would be something like: How long does it take to get to the destination or what is the difference between different connections in terms of the duration and the cost? On the other side is the software which embodies the knowledge that can answer the user's question. The software represents the knowledge as a combination of the facts and computations that it can perform on the facts. In this use case, the facts could be the locations of the train stations and the schedule of the trains which are running. An example computation could then be the directions from location A to location B based on the schedules of the trains. To bridge the gap between the physical world of the user and the digital world of the computer, the knowledge model needs a representation with which the user can interact.

This bridge between these two worlds is the user interface. The most common form of a user interface is the graphical user interface (GUI) which the following discussion will be limited to. The GUI is the point where the flexibility of the knowledge model gets lost. Most user interfaces use physical metaphors from the analog world like forms and buttons. In the trip planning example, the GUI could consist of a form with two fields to enter the start, and the destination of the trip and a submit button to submit the search. The advantage is that these kind of interfaces are easier to learn because the user has a mental model of how buttons and forms work in the analog world. However, emulating the physical world also has drawbacks. The model itself can solve arbitrary problems within its domain by giving it a physical shape it is turned into a single purpose machine.

The developer of an app can allow the user to ask more complicated questions by building more complex interfaces. For example, The trip planner could have additional destination fields to plan trips with stops in between. However, this does not address the fundamental problem because many constraints restrict the level of complexity that an interface modeled on physical metaphors can allow. The first problem is that a modification of the interface always requires an expert. Therefore all supported question have to be known in advance, and because programmers are expensive, it is only economical to implement questions which a majority of the users is interested in. Even if all questions could be known in advance and money was of no concern, there is still the problem of how to represent all question options in an interface without overwhelming the user. The pragmatic solution that many applications choose is to only support simple questions in the interface. The consequence for the user is that they have to do much manual work to get answers to questions which have no representation in the interface. Consider the following scenario: a user makes plans to go to a restaurant after work and wants to know how much time it takes to stop by at home first if they use public transportation. The question itself is not too complicated for the knowledge model, but if the trip planner can only tell the user how to get from A to B, it takes two steps to get the answer. Whenever a question requires multiple steps, the user has to memorize intermediate answers because, with every subsequent question, the application usually resets the previous context. This mental work makes answering more complicated questions tedious and discourages the user from exploring multiple options.

Another problem arises when the user tries to solve a problem that involves knowledge models from multiple applications. Many applications expose their underlying knowledge model only through a GUI. If the user has a problem which requires multiple applications, they usually have to solve this by manually going back and forth between them. Consider the following scenario of a hypothetical user Sandra who loves reading fantasy novels. She likes to get her books at the local library. The library has an online catalog which lists all their books with the information whether or not they are currently available. In order to find out what book she wants to read, Sandra prefers a book review site that has much more detailed information, reviews of the books and curated lists with recommended books for different genres. Because the two applications live in separate worlds, she has to run a manual lookup for each book on the review site to find out if it is available at her library. Ideally, she could combine the knowledge models of both applications into a new tool that shows her interesting books from the review site combined with the availability information from her local library.

The app model also limits software to specific use cases. With the GUI paradigm an application has to provide a graphical representation for each kind of question that the user has. This pattern can be seen in information software where an application is often split up into multiple screens where each screen is dedicated to a group of questions that it can help to answer. This correlation between screens and questions can be illustrated by looking at a typical calendar application. Most calendars have different views to ask questions at different timescales. The monthly view is useful to answer questions like: When is the next free weekend this month. The daily view is helpful to answer questions like: when is the next event today. Calendars are a universal tool to deal with time. Many applications can be modeled as calendars. For example, a store manager who needs to create a shift plan for their employees could theoretically use any standard calendar application for that. For each shift, they could create an event and assign workers to a shift by adding them to the event. The problem is that a standard calendar has no views that could answer questions like: How many hours has a person worked this month or which shifts have not been assigned yet. The knowledge model of a standard calendar fits the requirements of the store manager. The problem is that the views of a standard calendar only support general questions. As a result, the manager needs a different application which provides screens specifically built for the scenario of designing a shift plan.

The intention of a GUI is to connect the user with the internal representation of the software. However, it can also act as a barrier that limits the ways the user can interact with the model: they cannot ask complex questions, and they cannot ask questions concerning multiple domains. If there are no sufficient workarounds to answer a user's question, the user has to find an application that's exclusively constructed to address their scenario. This inability makes the users very dependent on programmers to build tools for their needs.

1.4 An alternative model for information software

The previous section described how GUIs give end users simple access to digital knowledge models, but at the same time, they impede them from solving complex problems. This section gives a quick overview of an alternative model for information software which addresses these issues.

For information software, the questions of the user should be at the center of the design. Therefore we need a representation that connects the questions of the user with the underlying knowledge model of the software. In typical applications, the questions are represented indirectly in a GUI that is designed to answer predetermined kinds of questions. The user communicates their specific question to the application by interacting with the provided interface. The problem is that users cannot change the GUI to ask different kinds of questions. For complex questions, the user is forced to manually execute the interactions because the GUI provides no way to abstract complex workflows. Most applications only expose a limited subset of the underlying knowledge model through the GUI which restricts what a user could potentially express with the software. The knowledge model itself is a black box that cannot be extended easily by the user or combined with the knowledge of other applications.

This paper argues that a fundamentally different model for information software is necessary in order to address these issues. This new model requires a direct representation of questions.

This representation has to be flexible to allow the user to change how the result of a question is visualized to adapt to the user's context. The user should not depend on the developer to anticipate their exact needs. At the same time, the representation has to be expressive enough that the user can ask any question that the knowledge model can answer. Another requirement is that the representation is natural and easy for end-users to learn. In order to achieve that, the knowledge model of the software has to be presented in a form that is understandable by the user. If the user understands the underlying knowledge model of the software, this opens up many new possibilities. More advanced users could start to extend the knowledge model of a generic application like a calendar, for example, to adapt it to their specific problem of scheduling their employees. If multiple applications adopt this model, users could build entirely new applications by combining existing solutions. These are new abilities for end-users which are currently only accessible to professional developers.

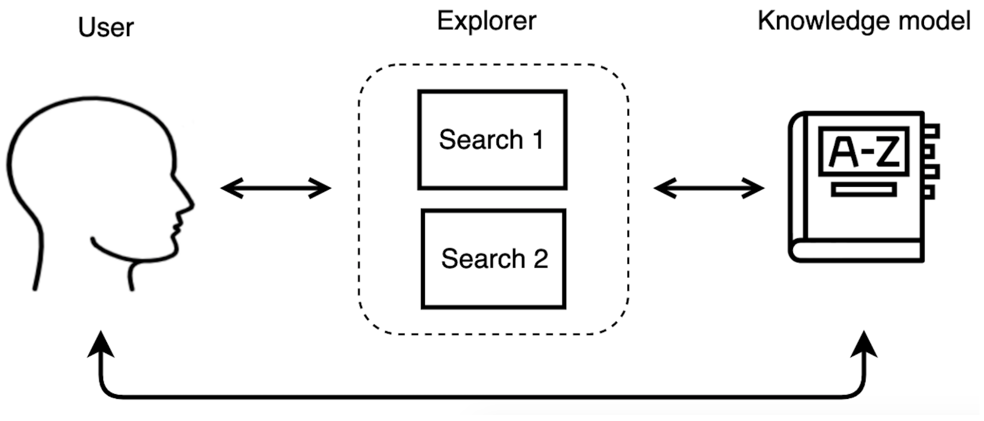


Figure 1.2: User interaction with Garlang

In order to demonstrate how such a system could look like this paper introduces a hypothetical programming language called Garlang [2]. It is a naturalistic programming language that is modeled after how natural language works. Garlang uses the metaphor of a dictionary to expose the knowledge model with definitions for the objects that exist in an application expressed as English statements. The dictionary is not just a reference. Garlang also allows the user to modify the definitions in the dictionary to add custom business logic.

The main interface of Garlang is the explorer. It helps the user find answers to their questions. The user can express questions as searches based on the concepts defined in

[2]Named after John of Garland, inventor of the word "dictionary"

the dictionary. They can display the results of a search in different views, compare them with alternative results, or run additional searches to provide more context. At the end of an exploration, the explorer might contain many different interconnected searches that highlight different aspects of the problem of the user. The state of the explorer can be stored to communicate the decision to other people or to reuse it as a reusable template for similar problems. A complete use case scenario of Garlang is described in section 4.2.

2 Naturalistic Programming

Garlang needs to present the knowledge model of an application in a form that is easy to understand for an end-user. The computer, on the other hand, requires instructions that have a precisely defined semantics. In order to solve this conflict, Garlang has to find a compromise that is natural enough that end users can understand it without losing precise semantics so a computer can interpret it. Natural language (NL) is too imprecise for computers to interpret. Human cognition can resolve ambiguity in language, but computers have not been able to emulate that process. There is a gap between what humans can express with natural language and what computers can understand. From this problem, the research field of *naturalistic languages* has emerged. Different authors have proposed various approaches to define a restricted version of a language that can be formalized to be executed by a computer and are more natural than traditional programming languages (Pulido-Prieto and Juárez-Martínez (2017))

2.1 Naturalists and Formalists

Clark et al. (2010) divide the field into two thought schools: the *naturalists* and the *formalists*. They both approach the problem from different sides. The *naturalists* start with a natural language and try to restrict it into a limited version of the full NL so the computer can interpret it. The resulting controlled language (CL) is not completely free of ambiguities; there can be multiple valid interpretations of a sentence. NL processing, in combination with heuristics, is used to pick the "best" solution. The *formalists* in comparison view a CL as a formal language that behaves more like a natural language. Therefore it is easier to use by humans than common formal languages. Formal CLs are well-defined and predictable. Because the underlying philosophy of the two approaches is very different, they lead to different kinds of CLs. In Formalistic CLs, each sentence has a single valid interpretation, and each word has only a single semantic meaning. In contrast, a naturalistic CL has small heuristics that are applied locally to make disambiguation to select the correct interpretation of a word or sentence. A formalist might consider the language complete once it is expressive enough while the naturalist tries to expand the language to a more complete CL incrementally. As a result, naturalistic CLs might be more natural to use but also harder to control because it is hard always to predict the disambiguation decisions of the system. In contrast, formalistic CLs are more predictable, but they can be less natural to read and may require the user to learn more about the domain model of the CL.

2.2 Summary

Garlang follows the philosophy of naturalistic CLs to produce a language that feels familiar to the end-user. It deals with the ambiguities that arise from this approach by continually giving the user feedback on how it is interpreting sentences. If it cannot determine the semantics of a sentence, Garlang falls back to asking the user to clarify the intended interpretation. The effectiveness of this feedback mechanism depends on the ability of the user to form a good mental model of the underlying computational model. This is one of the reasons why Garlang is based on Datalog. Datalog is a logic programming language that maps well to natural language because it is based on the assertion of statements and application of rules. Datalog will be introduced in the next chapter and how Garlang is translated to Datalog will be discussed in detail in chapter 6.

3 Datalog

3.1 Basics

The work of Ceri et al. (1989) is the basis of the following description of Datalog, which is a declarative query language based on logic programming. The basic building blocks are facts and rules. A logic program can only have a finite number of facts and rules. Facts are statements about the model of the program, such as: "Bob is the parent of Frank." As a Datalog clause this statement would be represented like this:

```
parent(bob, frank)
```

Rules can be used to deduce facts from other facts. For example, on the previous fact, we could apply the rule: "If X is the parent of Y and if Y is the parent of Z then X is the grandparent of Z." The corresponding Datalog clause looks like this.

```
grandparent(Z, X):- parent(X, Y)
```

Datalog clauses consist of two parts the left-hand side of the clause is called head, and the right-hand side is called body. The arguments of a predicate are called terms, and they can be either variables or constants. If the body of a clause is empty, it is called a fact; otherwise, it is called a rule. By convention constants and predicates begin with a lowercase letter and variables with an uppercase letter. Predicates with the same name must always have the same arity. If a clause only contains literals, it is called ground. To guarantee that a Datalog program always terminates the following conditions have to be fulfilled:

- all facts have to be ground
- all variables which occur in the head of a rule need to reoccur in the body of the rule

Often it is necessary to get answers to specific questions instead of calculating all deducible facts. For example, we might want to know just the children of Bob. To support such ad hoc queries, Datalog has the concept of goals. The goal to find bobs children could be specified like this:

```
?- parent(bob, X)
```

3.2 Extensions

Datalog models a limited subset of first-order logic. There are several extensions which extend the power of pure Datalog. The extensions which we will later need for the implementation of Garlang are built-in predicates, negation, and aggregation. We will discuss these extensions in the following sections.

3.2.1 Built-in predicates

Built-in predicates allow us to extend Datalog with custom functions. Each function is assigned to a special predicate symbol like $<$, $>$, $!=$ or $=$. Built-in predicates can only be used in the body of a Datalog clause, and they usually use infix notation. For example, the not equal predicate could be used, to define a sibling predicate, which avoids that a person is considered a sibling of themselves.

```
sibling(X,Y):-parent(X,Z),parent(Y,Z),Y != Z
```

The difference between these built-in predicates and ordinary Datalog predicates is that they are not stored in the database. Instead, they are implemented as custom functions which are evaluated when the Datalog program is executed. Most built-in predicates define infinite relations. For example, $<$ applies to all numbers. These infinite predicates endanger the restriction that a Datalog program can only consist of a finite number of facts. In order to ensure that programs with built-in predicates always terminate the arguments of all built-in predicates have to fulfill at least one of the following conditions.

- the argument also occurs in a non-built-in predicate
- the argument is bound by equality constraints to a constant or an argument that also occurs in a non-built-in predicate

The evaluation of the built-in predicates is deferred until all of its arguments are bound to constants. The equality predicate is an exception to this rule. As soon as one of its argument is bound or a constant, the other argument can be bound to the same constant.

Built-Ins can also be used to add new operations to Datalog like arithmetic operations. For example, to express additions like $X + Y = Z$ the built-in predicate *plus(X, Y, Z)* could be added with the requirement that X, Y, and Z are numbers. In Garlang, this will be later used to add basic operations for the built-in data types number and text.

3.2.2 Negation

Pure Datalog does not allow negations. The problem is that if for example the goal *?-father(bob, billy)* returns an empty result this could be either because Bob is not the father of Billy or because the facts are incomplete and do not include all the children of Bob. In order to infer negative facts from a pure Datalog program, we first have to assume that the facts of our program are complete. This assumption is also called the Closed World Assumption (CWA). The CWA can be applied to Datalog with the following reasoning.

> If a fact does not logically follow from a set of Datalog clauses, then we conclude that the negation of this fact is true.

The CWA allows us to generate negative facts from a set of Datalog clauses, but we are still missing the ability to make deductions based on negative facts. These deductions are useful for rules like "if X is a Day and X is not a weekday, then X is a weekend day." One extension that allows negations in the premise of a rule is called stratified Datalog$^{\neg}$. The idea behind this is that rules are ordered based on the predicates of which they depend. This ensures that before a rule R is evaluated, all negated predicates p^{i} that occur in its body are already evaluated. That means that all rules that generate facts for one of the predicates p^{i} have been already evaluated. For example, consider the following example definition which considers everyone who is not closely related unrelated.

```
unrelated(x, y):- ¬parent(x, y), ¬parent(y, x),
                  ¬grandparent(x, y), ¬grandparent(y, x),
                  ¬sibling(x, y)
```

Before the rule for unrelated can be evaluated, the rules that define the parent, the grandparent, and the sibling predicate have to be evaluated first to get all their facts. Then we can apply the CWA rule "locally" to get the negative facts of parent, grandparent, and sibling. Afterward, the *unrelated* predicate can be evaluated.

Finding an ordering that satisfies the constraints stratified Datalog$^{\neg}$ is not always possible. This problem can be illustrated with the following example.

```
p(x):- ¬q(x)
q(x):- ¬p(x)
```

P has the negated predicate q in its body, and q has the negated predicate p in its body. Therefore, neither of them can be evaluated first. This is a limitation of stratified Datalog$^{\neg}$. A program can be only evaluated with this approach if the dependency graph of the predicates has no cycles. The dependency graph is constructed by connecting each predicate that occurs in the body of another predicate as a dependency of that predicate. If a program fulfills this requirement, it is called *stratified*.

Other approaches add negation for all Datalog programs like *inflationary evaluation* and solutions which use three-valued logic. Stratified Datalog$^{\neg}$ was chosen because of its simplicity, and the aggregation extension described in the next section requires stratified Datalog programs as well. This area could be investigated in subsequent papers to remove this restriction.

3.2.3 Aggregation

Another feature that is required by Garlang is the ability to summarize values. For example, instead of getting all the children of a person, we might want to get the number of them instead. Green et al. (2013) describe how aggregation can be added to Datalog. With this extension we can calculate the number of children that a person has like this:

```
    numberOfChildren(X, count<Y>):- child(Y, X).
```

An aggregate function like count in this example is a function that maps from multisets of domain values to domain values. Examples for aggregate functions are count, max, min, and sum. We can define them similarly to the Built-in functions by assigning each aggregate function to a function symbol. A term that consists of an aggregate function is also called an aggregate term. Aggregates can be only used in the head of a clause and they have to fulfill these two properties:

- all variables in an aggregate term must also occur in the body of the clause
- a variable can occur in the head either only in aggregated terms or only in non aggregated terms

The variables occurring in non aggregated terms are called *grouping variables*. A rule with aggregates is evaluated by replacing all grouping variables with constants. There are some problematic cases where aggregates can lead to programs that never terminate. For example, consider the following example.

```
    p(sum<X>):- p(X).
```

When this program is evaluated on the ground facts p(1) and p(2) the second rule would deduce p(3). Now the second rule can be applied again, which would generate the fact p(6). This step would repeat itself, continuously generating new facts without ever terminating. This problem can be solved by disallowing recursion in Datalog rules if they contain aggregated terms.

3.3 Summary

There are several benefits of using Datalog as the basis of Garlang. Defining Garlang in terms of Datalog gives us a well defined semantic. This makes it easy to explain the semantic of Garlang examples without having to implement an entirely new language. Datalog gives us also much flexibility. It has been an active research topic for several decades and the result of this are many optimizations and extensions that can be beneficial for Garlang. This chapter described just a few basic extensions of Datalog that are necessary for Garlang. More research is needed to evaluate how other Datalog extensions could help to improve Garlang. A benefit on the practical side of Datalog is that it does not lock Garlang into a specific technology. There are a lot of different Implementations of Datalog that could be selected for a practical implementation of Garlang.

One deliberate limitation of Datalog is that it is not Turing complete. In return Datalog programs are always guaranteed to terminate. This trade-off is also beneficial for Garlang. Garlang is not supposed to be a general purpose language. Its goal is to expose computational models with a language that's learnable by end users. Complex algorithms in a computational model can be implemented outside of Garlang in a conventional programming language and exposed to Garlang as custom built-ins.

4 Garlang

4.1 Introduction

The following chapters will describe how Garlang works. The language is described by looking at different examples that illustrate the different aspects of it. Datalog is used to specify the semantics of Garlang and to demonstrate how it can be translated into an executable program. This paper does not provide a full formal specification of the language. There is more work required to create a real implementation of Garlang.

Garlang is not trying to solve the problem of how to build software that understands natural language. Instead, it uses natural language as a metaphor similar to how the desktop metaphor leverages peoples knowledge about physical objects to build GUIs that are easier to understand. The goal is that an end user can apply their knowledge about natural language to learn the programming language more easily. The semantic model of Datalog fits well with natural language because it is based on facts and rules. This declarative model is more similar to how natural languages works compared to an imperative programming model. Garlang uses constructs from natural languages like determiners, questions, sentence structure, and compound sentences to map textual sentences to a formal Datalog representation.

4.2 Shift plan example

This section introduces Garlang by starting with a motivating example that gives an impression of how the language works. Because Garlang is based on constructs from natural languages, it should be readable as English text without knowing the semantic of Garlang. There is a learning curve for end-users if they want to modify existing code or write new Garlang programs, but understanding this example application does not require any prior knowledge about the language.

As an example, the scenario from the introduction is used, where a manager wants to know how to distribute their workers best to open shifts. This example will show how Garlang can help a manager to build a tool that can answer questions like what shift is not assigned yet and which employees are available. Before the manager can ask these questions, they have to model the problem first. For this example, Garlang has a basic notion of calendars built-in that the user can utilize. This base model consists of kinds like dates, time frames, point in times, and durations. It also describes rules, for example, to determine the duration of a date or to find out if a time frame contains a date. Garlang presents its built-in knowledge to the user as a dictionary. Each kind has its section with the facts and rules that apply to it. The following is an extract from the dictionary with the partial definitions of the event and time frame kind.

Event <plural events>

```
An event is a time frame.
An event has a title that is a text.
...
```

Time frame <plural events>

```
A time frame has a start that is a point in time.
A time frame has an end that is a point in time.

A time frame has a duration that is a duration (time frame duration):
  time frame duration:
    the duration between the end of the time frame
    and the start of the time frame
...
```

The plural form of a kind is automatically generated but the user can always override the default form for irregular nouns. The dictionary separates the definitions into statement forms and rules. The statement forms in the definition describe what statements can be made about an instance of the kind. For example, we can describe a meeting event with the following statements.

```
The example meeting is an event.
The example meeting has a title that is "meeting".
The example meeting has a start that is 05.06.2019 at 9:00.
The example meeting has an end that is 05.06.2019 at 10:00.
```

The first statement introduces the *example meeting* and establishes that it is an event. The next sentence states the title of the event. Because an event is also a time frame, the meeting has two statements that describe the start and end of it. Garlang then applies the rules of event and time frame to these facts. The duration rule of the time frame, for example, generates the fact *"The example meeting has a duration that is 1 hour."* based on the stated start and end of the example meeting.

The dictionary serves as a reference to the user, but it is also functioning as a programming environment. It is not static; the user can amend it by adding or removing definitions or defining entirely new kinds. In the next step, we will see how we can extend the dictionary to model the problem of the manager. The manager has to fulfill the following rules when making a shift plan.

> Each shift needs to be staffed with three employees; one of them has to be a manager. The employees have different working hours. Full-time employees work 40 hours per week, and part-time employees work 20 hours per week. At the end of every month, each employee should have worked the exact amount of hours defined in their contract.

We can start modeling this problem in Garlang with three new kinds: shifts, employees, and managers.

Employee <plural employees>

```
An employee has a name that is a text.
An employee works a duration per week.
An employee is assigned to many shifts.
```

Manager <plural managers>

```
A manager is an employee.
```

Shift <plural shifts>

```
a shift is a time frame.
```

An employee has a name, they work some duration per week, and they can be assigned to multiple shifts. A manager is modeled as a special kind of employee. A shift is modeled as a special kind of time frame. These specifications define only the minimum requirements to represents employees and shifts in the system. Based on the statements we have defined for each, Garlang automatically generates forms which the user can use to create instances of the kind. For example, the metadata that name is a text is used to render a text input for the name (fig. 4.1). Built-in types like a point in time can implement custom input methods, as shown in the form of the shift. Statement forms like *"An employee is assigned to many shifts"* generate input fields in both the employee and the shift. The purpose of the forms is to provide a familiar input method for end users to create and modify objects in the system.

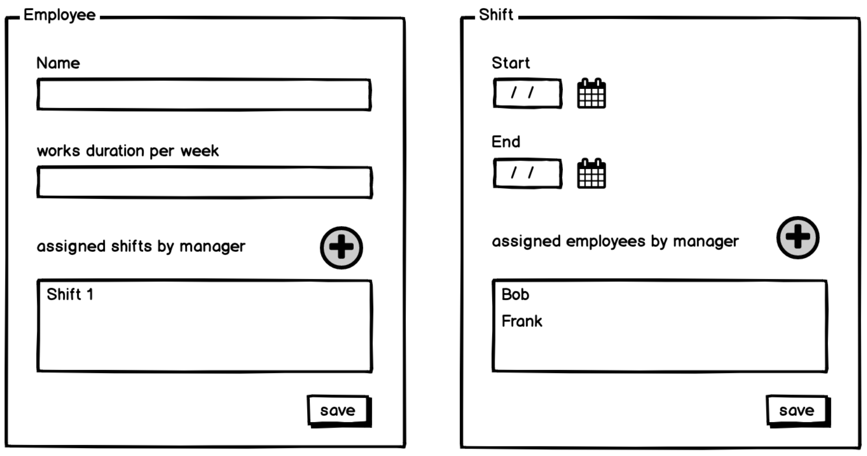


Figure 4.1: Auto generated forms for employee and shift

Defining the employees and shifts is only excise work. The manager is interested in learning how to distribute her employees best to make a decision in the form of a shift plan. Answering questions is the purpose of the explorer in Garlang. It allows the user to run searches, display the results in different views, compare them with alternative results, or ask more questions to provide additional context. The explorer is a free form canvas that allows the user to arrange result views in any way they need to aid their decision-making process.

Going back to the example of the manager, she might want to know initially what the current shifts are. She can get an answer by creating a search for *"all shifts"*. Results of a search can be visualized in different ways, depending on the kinds of the result (fig. 7.1). Built-in types can provide their custom views. In this case, the shifts are visualized as a calendar by default because shifts are time frames.

Result views can also be interactive. For example, in the calendar view, the user can go forward or backward in time or zoom out to see the shifts of a whole month. The views also give access to the underlying data. For example, if the user wants to change a shift, they can click on the shift to open the form of the shift. It is essential that all interactions are optional. There should be no required interaction before the user is presented with a visualization. Each view should provide reasonable defaults and adapt its behavior to the usage patterns of the user. For example, the calendar should present the current time by default. When the user changes the calendar to the weekly view in subsequent searches, the weekly view should be used for shift results. The work of Victor (2006) on inferring context can be used as inspiration to improve the ability of the explorer to adapt to the user.

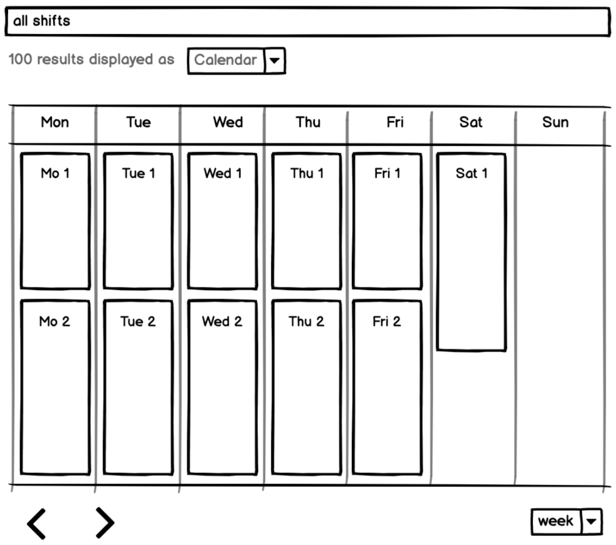


Figure 4.2: Search for shifts

Now that the manager can see all the shifts she wants to know which employees are available for the shifts. In the explorer, she can place a second search for *"all employees"* next to the shift calendar (4.3). The result of this search is a list of all the employees.

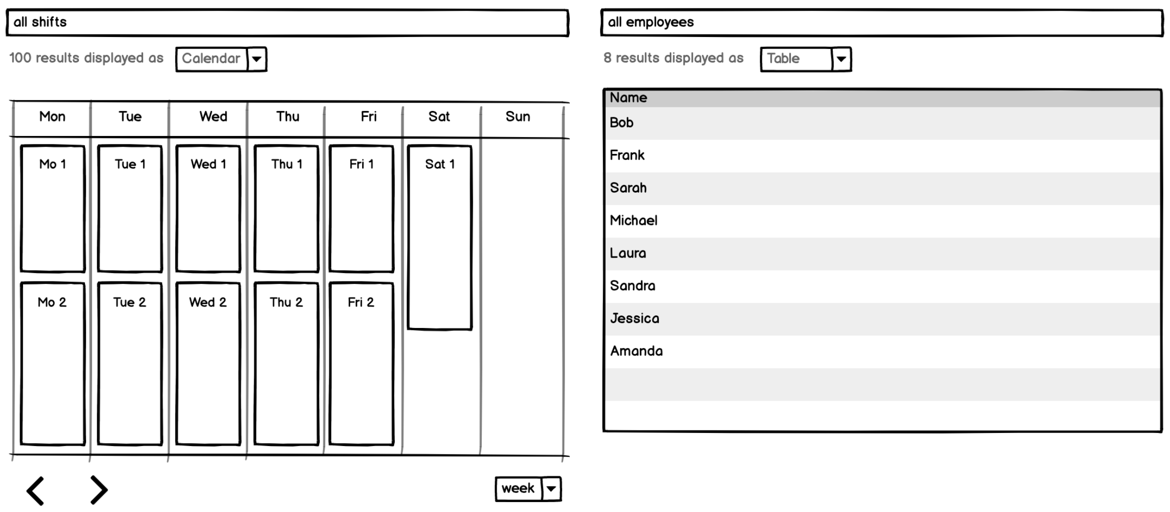


Figure 4.3: Search for shifts and for employees

This search is not helpful yet because it just shows all employees. Instead, the manager wants to contextualize the employee search and only show employees who are available during the visible time frame of the shift calendar. Before she can perform such a search, we need to define what it means that an employee is available during a time frame. For this, we have to extend the dictionary. We can do this in a top-down fashion by breaking down the concept into sub-statements until the sub-statements can be expressed in terms of statements that are already defined in the dictionary. We can begin the definition with the following rule.

```
An employee is available during a time frame when:
  The employee has available time during a month that
  overlaps with the time frame.
```

The worked hours of an employee are always calculated monthly. Therefore, we define that an employee has time during a time frame if they have time during any month in the time frame. Next, we need to define the availability for a month. This can be done with the following rule.

```
 An employee has available hours during a month when:
  The total work time is less than the worked time.
  The total work time: The total work time of the employee per month.
  The worked time: the worked time of the employee during the month.
```

The relationship *"is less"* is a built-in of Garlang. The comparison statement references *total hours* and *worked hours* which are local definitions of the rule defined below the statement. Next, we need to define the work hours and the worked hours of each employee per month. Based on the weekly work time that we have defined earlier, we can describe the monthly hours with the following rule.

```
The work hours of an employee per month is a duration (monthly work time):
  The employee works a duration (weekly work time) per week.
  The monthly work time:  the weekly work time * 4
```

This rule is a simplification because the rule assumes that each month has the same number of days, and it does not consider public holidays. With this simplification, the monthly hours can be calculated by multiplying the weekly work time by four. The next rule calculates the time an employee has worked per month.

```
The total work time of an employee during a month is a duration (work time):
  the work time:
    The sum of the duration of all shifts that are
    assigned to the person and happen during the month.
```

The number of hours is calculated by summing up the duration of all shifts assigned to the employee during the month. *"sum of"* is a built-in operation of Garlang. With this last definition, we achieved our goal of defining the availability of an employee in terms of concepts that are already known to Garlang. We can now switch back to the explorer and connect the employees search with the shifts search.

Visualizations themselves are also objects in Garlang, just like employees and shifts. They state facts about them, and the user can make new statements about them. For example, the calendar visualization makes a statement about the currently visible time frame. We can use this fact in the employee search and change it to the following.

```
all employees that are available
during the visible time frame of the shifts search
```

The employee search is now linked to the shift search and updates automatically whenever the time frame of the calendar changes. With this tool, the manager can answer both her main questions: Which shifts need to be assigned and which employee is available. However, this is not a satisfactory solution yet. The manager is missing some critical information. In order to see which shifts are fully assigned, she has to open the form of each shift. She can easily see who is available, but there is no simple way to communicate the decision whom she wants to assign to a shift back to the system. In the remainder of this section, we want to show how it is possible to customize the tool incrementally to address these issues.

First, we want to highlight all shifts which are not fully assigned. In order to do this, we need to add two rules to the shift. The first rule defines a shift as fully assigned when it has two employees and one manager.

```
A shift is fully assigned when:
  3 employees and 1 manager are assigned to the shift.
```

The reason why the rule requires three employees is that a manager is also an employee. The second rule adds a highlight-statement for each shift that is not fully assigned.

```
A shift is highlighted when:
    The shift is not fully assigned.
```

The highlight-statement is part of the built-in vocabulary of Garlang. It is not mapped to a specific effect. Instead, other objects can implement rules to react to it. For example, the list view and the calendar view can render results that have a highlight fact with a different background color.

As a second improvement, we want to make it easier for the manager to assign employees to a shift. The manager should be able to drag and drop employees on the shifts that she wants them assigned to. Garlang supports drag and drop by default. All objects in a view should be draggable and also allow other objects to be dropped on them. The views do not define any effects that should happen as a result of a drag and drop event; they just add the fact that an object is being dragged or dropped and the user can then define the semantic meaning of this event. For example, if the entry for Bob in the employee search was being dragged the list view would state the fact: *"Bob is being dragged"*. The drag and drop facts always reference the underlying object, not the graphical representation that is being dragged. The benefit of that is that the user can define drag and drop rules independent of how an object is visualized. We can apply drag and drop interactions to the shift plan tool with the following rules.

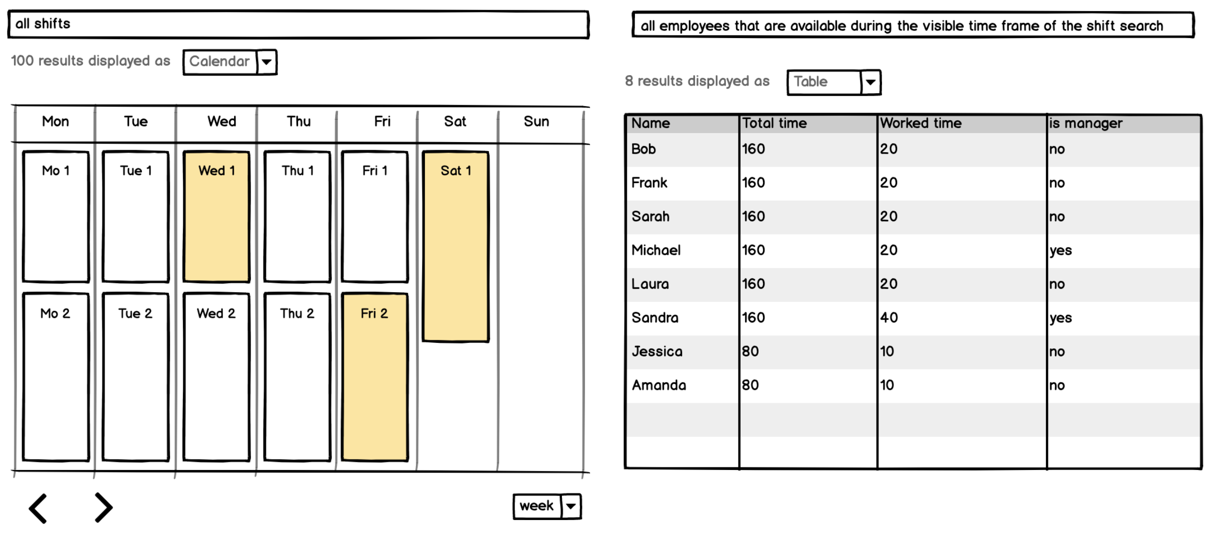
```
When an employee is being dragged:
  All shifts that the employee is available for are highlighted.

After an employee is dropped on a shift:
  The employee is assigned to the shift.
```

The first rule highlights all shifts that an employee is available for while being dragged. The second rule assigns the employee to the shift on which it is dropped. The second rule is an after rule. The difference to a when rule is that a after rule changes the state permanently while a when rule only asserts the facts in its conclusion as long as the premise is true. In this case we want the employee to be assigned to the shift after it is dropped on the shift and not just in the instance when the drop event happens.

The last improvement we will discuss is adding more information to the employee table. Right now, the manager lacks some crucial information. She can only see who is available but not how many hours each employee has already worked and which of them are managers. The table view allows the user to add additional columns which allow sub searches for each result. We can use this feature to add columns for the total work time, the worked time, and one to indicate if the employee is a manager or not. These are the corresponding search expressions. *"the employee"* refers to the employee in the result set that each sub-search is applied to.

```
 the total time:
  the sum of the total work time of the employee during
  all months that overlap with the visible time frame of the shift search

the worked time:
  the sum of the worked time of the employee during
  all months that overlap with the visible time frame of the shift search

is a manager: The employee is a manager.
```

The improved version of the shift plan is depicted in fig 4.4. This shift planner is not a complete solution. There are still many things that could be improved. For example, the current shift plan does not consider the holidays of the employees or limitations of how long an employee can work per day. The goal of this example was not to present a perfect solution but to give an impression of how programming in the Garlang environment could look. With the three improvements, we have illustrated how Garlang gives the user the ability to incrementally shape their tool to solve their specific problem. Next, we will discuss how Garlang is implemented internally.



all shifts

100 results displayed as Calendar

all employees that are available during the visible time frame of the shift search

8 results displayed as Table

Name	Total time	Worked time	is manager
Bob	160	20	no
Frank	160	20	no
Sarah	160	20	no
Michael	160	20	yes
Laura	160	20	no
Sandra	160	40	yes
Jessica	80	10	no
Amanda	80	10	no

Figure 4.4: improved shift plan tool

5 Data Model

Garlang models the world as objects that are in a relationship with each other. An object is defined by its kinds. A kind is a collection of characteristics that can be applied to an object. Kinds can also be viewed as different perspectives on the same object. For example a meeting in Garlang has the kind *event* and *time frame*. The perspective from the event kind is that the meeting has a title and people that are attending. If the meeting is viewed as a time frame, it has a start and an end. Kinds themselves can be defined in terms of other kinds. In the shift plan, for example, we have defined the kind manager as a kind of employee. If we create an object that has the kind manager, this hierarchical structure is flattened. Each object is a unique id which is used to refer to the whole object. The kinds of an object are added to the object with is-relationships between the object id and the instance of the kind. An instance of a kind is also a unique id with additional information about what kind it is. The following example shows the example Datalog representation of a manager that is assigned to a shift. Two functions are introduced here *uniqueId* returns an object id and *new* is a function that accepts a kind as an argument and returns a new instance id of the kind. All facts of a kind are attached to the kind instance.

```
sandra = uniqueId()
employeeKind = new(employee)
exampleShift = uniqueId()
shiftKind = new(shift)

is(sandra, new(manager))
is(sandra, employeeKind)

is(exampleShift, new(timeFrame))
is(exampleShift, shiftKind)

assignedTo(sandra, shiftKind)
...
```

Kinds that depend on other kinds can be expressed with Datalog rules. The following is an example that adds an employee kind to each object that has a manager kind. This rule uses the *kindOf* predicate which the *new* function adds to a kind instance when it is created.

```
is(Object, new(employee)):- is(Object, Manager), kindOf(Manager, manager)
```

Modeling objects as a collection of its kinds gives the user significant flexibility. Objects can combine multiple unrelated kinds, which is a problem in a lot of class-based hierarchical object models. Next, we have to consider a few cases to show that Garlang can resolve the right kind when dealing with objects that have multiple kinds. In most cases, the kind is explicitly stated because most rules are defined per kind in the dictionary. Here is an example rule from the definition of an employee.

```
An employee has available hours during a month when: ...
```

Both *"an employee"* and *"a month"* reference a specific kind. The rule can be applied without ambiguity to any pair of objects where one object has an employee kind, and the other has a month kind. There are exceptions where a statement references a specific kind, but the user intends to reference the object id or another kind of the object. Consider the search expression in the shift plan example that determines if an employee is a manager.

```
The employee is a manager.
```

If we interpret *"the employee"* as a reference to the kind employee, this statement will never be true. In this specific case, the right interpretation would be to interpret *"the employee"* as a reference to the object id. Garlang could run a check that detects contradictions like this which can be resolved by replacing the kind reference with a reference to the object. In this paper, we assume that Garlang implements such an algorithm, that examples like this work as expected. Future work is necessary to design the actual implementation of it. There is a similar problem that occurs if we look at the following example.

```
The name of the manager.
```

The name is part of the employee kind, which is the basis of the manager kind. In this instance, Garlang needs to replace the manager kind reference with a reference to its sibling employee kind. The reference resolution algorithm described in the previous section should be generalized to handle this case as well. The replacement of a kind reference with a reference to a sibling reference has to happen with some caution. Consider the following case.

```
The name of the month.
```

If we applied the algorithm here, Garlang would replace *"the month"* with its sibling employee kind. It is theoretically possible that an object exists that is a month and an employee, but it is more likely that the user made a mistake here. Before replacing a kind with its sibling, Garlang should check if the sibling kind has the other kind as its basis. This check would succeed for the manager because it is defined as an employee, whereas a month is not an employee. If this check fails, Garlang should warn the user to indicate that there might be a problem.

In Garlang objects can be assigned to a name. An example of this is the statement *"Sandra is a manager"*. It creates a new object and assigns it to the global name, *Sandra*. In this example, if we make statements about Sandra, Garlang has to determine to which kind the statement belongs. In most cases, there should not be much overlap in the definitions of different kinds. It is possible though that an object has multiple kinds that define the same statements. For example, if both kinds manager and employee define an hourly rate consider the semantics of the following statement:

```
Sandra has an hourly rate that is 30 dollars.
```

It is unclear if the statement references the hourly rate of the employee kind or the manager kind or both. In this case, Garlang should display a dialog to ask the user to clarify which relationship they mean. This problem can also occur when referencing objects by kind and should be solved in the same way. Generally, it is good practice to avoid this situation altogether by defining statement only in one place or express similar concepts with different wording to avoid confusion. Besides global names, Garlang also allows local names inside of rules. An example of this is the monthly hours rule of the employee from the shift plan example.

```
The work hours of an employee per month is a duration (monthly hours):
...
```

Local names are limited to the scope of the rule in which they are defined. A local name can be defined by writing a name in brackets after a kind. Instead of referring to an object by its kind, local names can be used to assign more descriptive names to them. They are also necessary for statements where multiple objects have the same kind. They do not need any special consideration when resolving references and can be treated identically to references by kind.

5.1 Primitive data types

Garlang has two primitive kinds: numbers and text. An object that has a primitive kind is also called a primitive object. Both text and numbers have built-in literal forms. Numbers can be either expressed as Integers like *5* or as decimal like *3.14*. Text literals have to be wrapped in quotation marks to explicitly indicate that Garlang should not interpret the text as a statement. Here are two example statements that use primitive kinds.

```
The work time of Bob per month is 5 hours.
Bob has a name that is "Bob".
```

Primitive objects have a few restrictions. First of all, they can either be a number or a text, not both. Secondly, a primitive object cannot have any other kinds unless the kind does not add any new behavior to the object. The problems with extending primitive objects are discussed in section 6.8.

5.2 Summary

This chapter described the data model of Garlang. The purpose of this description is to have a foundation for the next section, which explains how Garlang interprets natural language. This description is not complete. There are open issues like how individual objects can override statements that are defined for the whole kind. These are problems that need to be addressed in future iterations of Garlang.

6 Language Model

The shift planner example covered the majority of the features of Garlang, but it was just a quick overview, and it did not go into detail how they work. The explanation of the example relied on the reader's interpretation of Garlang as English text. In the previous chapter, we described how Garlang models the world. This chapter focuses on how Garlang interprets individual sentences and words.

6.1 Statements

Garlang interprets language as a combination of statements. Garlang's model of language is not based on complex natural language processing; instead, it interprets, each statement as belonging to a particular statement form which then can be mapped to a predicate in Datalog. Consider the following sentence, which consists of a single statement.

```
Bob is assigned by Sandra to the example shift.
```

Garlang starts parsing this sentence by identifying all nouns in the sentence. In this case these are *Bob*, *Sandra* and *the example shift*. These are the objects in the statement. Garlang uses the remaining text as an identifier of the statement form. Statement forms are always specific to the kind of their objects. In this example, we interpret Bob as a kind of employee, Sandra as a kind of manager and example shift as a kind of shift. Therefore, this statement belongs to the following statement form.

```
An employee is assigned by a manager to a shift.
```

The statement form is then mapped to a Datalog predicate to generate a new fact with the extracted objects as arguments. The resulting Datalog predicate looks like this.

```
isAssignedByTo(bob, sandra, exampleShift)
```

Statements can have multiple objects, although in most cases, a statement should not involve more than four objects because a statement always represents a single relationship.

Complex concepts can be described with multiple statements. Sentences can contain more than one statement. An example of a sentence that contains multiple statements is the definition of the name of Bob.

```
Bob has a name that is "Bob".
```

This sentence combines the statement forms *"An employee has a name."* with the statement form *"A name is a text."* which is attached as a that-clause. Complex sentences will be discussed in detail in section 6.5.

Garlang's interpretation of language as a combination of statements has the benefit that it can be implemented with a simple model. At the same time, the end user is not limited in the way they can phrase sentences because Garlang only needs to be able to extract the nouns in each statement. A simple implementation model also leads to a simple mental model for the user. This is a problem with some naturalistic programming languages based on advanced natural language processing. If the language misinterprets a statement, it is hard for the user to know how to rephrase it because the parsing process is a black box (Clark et al. (2010)). In Garlang, a wrong interpretation means that a statement is not mapped to the correct statement form. The user can correct the interpretation by selecting the intended statement form from the dictionary. One drawback is that users need to learn the specific vocabulary of the knowledge model, how things are phrased. This limitation can also be seen as a benefit because it enforces a consistent vocabulary across a Garlang application, which makes it easier to understand. One disadvantage remains, the statement is the smallest unit and can not be split up. For example, consider the previous statement form, which described the shift assignment. The user cannot just use a single aspect of the statement form and state *"Bob is assigned by Sandra"*. The statement form can only be used precisely how it was defined as a complete statement. The next section addresses parts of this issue to relax Garlang's rigid interpretation of statements to makes statements behave more like natural language.

6.2 Flexible noun order

In English, a sentence can be phrased differently to reorder the nouns in it while keeping the original semantics. Consider how the shift assignment statement could be rephrased with different noun orders.

```
A manager assigns an employee to a shift.
A manager assigns to a shift an employee.
An employee is assigned to a shift by a manager.
An employee is assigned by a manager to a shift.
To a shift, an employee is assigned by a manager.
To a shift, a manager assigns an employee.
```

If Garlang parsed these sentences strictly following the rules from the previous chapter, the result would be six different statement forms without any semantic connection. Forcing the user to separately define all statement forms that arise when reordering the objects in a statement is not a satisfactory solution. It is also not possible to restrict the user to a single noun order because reordering words is essential to express different concepts. Consider these two searches.

```
All shifts that Bob is assigned to by a manager
All employees that are assigned to a shift by Sandra
```

Even though searches have not been introduced formally, reading these two text fragments as English text makes it clear that different noun orders are needed to express both searches. The first search matches all shifts that Bob is assigned to, and it contains the statement form *"To a shift, an employee is assigned by a manager."*. The second search returns all employees that are assigned by Sandra. The underlying statement form here is *"An employee is assigned to a shift by a manager."*.

In order to solve this problem, Garlang's interpretation of statements has to be adapted. Two statement forms should be matched to the same predicate if they describe the same semantic relationship and differ only in the order of their nouns. Garlang uses a heuristic approach for matching statements. This heuristic has to solve two subproblems. First, the heuristic needs to select the correct predicate from all defined predicates. Afterward, Garlang has to map back the order of the nouns in the statement to the order in the original statement form. It is also crucial that Garlang communicates its interpretation back to the user while giving them the option to correct it. The heuristic also needs a confidence measurement which warns the user if Garlang is not sure how to interpret a statement.

The first criterion that can be used to filter matching predicates is the arity and the kinds of their arguments. The kinds of the arguments are sometimes ambiguous. For example, the object Sandra has both kinds employee and manager. Therefore all possible kind combinations of all objects in a sentence have to be considered as potential matches. Matching arity and argument types of a predicate is only a necessary condition there can be multiple predicates with the same type signature and a predicate should be only considered a match if it has the same semantic structure.

A simple approach to check if two sentences have the same basic structure is to compare the text of the sentences without the nouns. When this method is applied to the possible alternatives of the shift assignment statement, this will result in the following strings.

```
assigns to
assigns to
is assigned to by
is assigned by to
to is assigned by
to assigns
```

These strings look similar, but they are not a strict reordering of each other for example, *assign* is turned into *assigned* when used in the passive form, and the word *by* is added. There are many algorithms which solve the problem of determining the similarity between text strings. The most well known is the Levenshtein distance, which can be used to determine the minimum number of edits that are required to turn one string into another string (Levenshtein (1966)). From a similarity measurement like the Levenshtein distance, we could also derive a confidence score. The smaller the distance, the more confident Garlang can be that it has found the correct match. The Levenshtein distance is only one example of an Algorithm that could be used. More work has to be done to compare different algorithms and to benchmark which algorithm works best in practical use case scenarios. For example, an approach that considers not only single insert, modify or delete actions but can also detect reordering of substrings would be desirable.

Another way to improve the matching is to implement more grammatical rules in Garlang. For example, if we added a rule on how to turn a sentence into passive voice, Garlang could automatically generate passive forms to add them as synonyms for the original statement forms. The grammatical rules do not have to be perfect because the synonym forms are only used for matching statements. It is acceptable if they generate slight grammatical errors for irregular forms. The generated synonyms would also solve the problem of how to match back the order of the arguments. Even though these rules make the implementation more complicated, they do not break the mental model of the end user that Garlang matches statements to predefined statement forms.

In the general case, the arguments can be mapped back based on the kind of arguments. This mapping is only possible if each kind occurs only once in the predicate. In this case and all other cases where the heuristic does not find a match with a high enough confidence, Garlang should fall back to asking the user to clarify what statement form they meant and how the nouns should be mapped back. Garlang can remember the correction of the user and store it as an alias for the selected statement form to pick the correct interpretation in the future.

6.3 Determiners

Determiners have been already used in previous examples with the implicit semantic from natural language. This section explains the role of determiners and how Garlang interprets them. Determiners are words that come before nouns. They have mainly two roles: *referring determiners* specify what a noun refers to and *quantifying determiners* specify how much of something or how many things there are (det).

6.3.1 Referring determiners

Referring determiners are most important in Garlang. There are two different kinds of them: definite and indefinite articles. The definite article *the* is used when making statements about specific objects. These are also called specific references. Consider this example which states a fact about the objects Bob, Sandra, and the example shift.

```
(The) Bob is assigned by (the) Sandra to the example shift.
```

The definite article in front of Bob and Sandra is usually omitted because the English language has a rule that prohibits the definite article in front of people's names. Because Garlang does not understand what a name refers to definite articles are always optional when used in combination with names. Besides named objects, number and text literals also fall in the category of specific references. They are always used without a determiner.

In contrast to specific references, Indefinite articles like *a* and *an* allow the user to make general statements about kinds. These are called general references. *A* and *an* can be used synonymously in Garlang, but the user should follow the grammatical rules of English. One example of the usage of indefinite articles is the definition of statement forms like this.

```
An employee is assigned by a manager to a shift.
```

In the statement form, only indefinite articles are used. It is not a statement about specific objects; it instead describes a general relationship that objects of a specific kind can have. That is why a statement that only contains general reference is called a statement form. Garlang would translate the statement to the following Datalog representation.

```
isAssignedByTo(Employee, Manager, Shift), kindOf(Employee, employee),
kindOf(Manager, manager), kindOf(Shift, shift)
```

When referring to a kind like *"an employee"* Garlang translates this to a variable in Datalog with the implicit predicate *kindOf* that indicates that the variable is an employee. This interpretation mirrors the semantic of the sentence as plain English. When we are talking about *an employee*, we mean an object that is a kind of employee.

Specific and general references can also be combined in a single statement. This combination allows the expression of specific statements about a category of objects. Consider, for example, this statement which states the hourly wage for all managers.

```
A manager earns 50 dollars per hour
```

This statement has the statement form *"A manager earns some number dollars per hour"*. In this example, *a number* is bound to the number literal 50, and *a manager* is left as a general reference to any manager. Garlang's interpretation of this statement is that the hourly rate of 50 dollars should be applied as a fact to all managers. This interpretation also matches the semantics of the statement if we would read it as a plain English statement. Internally this statement is translated to the following Datalog rule.

```
earnsDollarsPerHour(Manager, 50): - kindOf(Manager, manager)
```

There are some problems with mixing specific and general references that lead to ambiguity. For example, consider the interpretation of the following sentence.

```
Bob is assigned by a manager to a shift.
```

If we interpret this sentence as plain English text without any further context, it is not clear which manager assigned Bob to which shift. If we evaluate the sentence with Garlang's interpretation of determiners *a manager* refers to any manager and *a shift* refers to any shift. As a result, Bob would be assigned to all shifts by all managers. This interpretation clashes with the semantics of the sentence as a plain English text.

In order to avoid this problem, Garlang applies some restrictions on how specific and general references can be used together in statements. First of all, statements that combine both types of references are only allowed in the definition of a kind in the dictionary. They can only contain a single generic reference to the kind in which they are defined. These limitation cover cases like the previous example *"A manager earns 50 dollars per hour "* which is a general statement that applies to all instances of the manager kind. In this case, the interpretation of Garlang fits with the natural language interpretation of the sentence.

Definite articles have been introduced in combination with names. They can also be used together with kinds. Combining *"the"* with a kind creates a reference to an object of this kind that has been introduced before. The interpretation of a statement with a specific reference to a kind, therefore always depends on its context. An example of definite references to kinds is the highlight-rule of the shifts.

```
when a shift is not fully assigned.:
    The shift is highlighted.
```

The premise refers to shift with an indeterminant article, which means it can be applied to any shift that is not fully assigned. In the context of the rule *"the shift"* now refers to the shift that has been introduced by the premise. Garlang would translate the highlight-rule to the following Datalog clause.

```
isHighlighted(Shift):- kindOf(Shift, shift), ¬isFullyAssigned(Shift),
```

As expected both the arguments of the *isHighlighted* and the *isFullyAssigned* predicate have the same variable as an argument. The only difference is that In the Datalog clause, the premise and the conclusion are flipped.

6.3.2 Quantifying determiners

Quantifying determiners can express how many or how much there is of something. Garlang adopts its semantic of quantifiers from how they work in plain English. In general, Garlang supports four quantifiers: no, [number], many and all. The interpretation of them depends on the context. They can be used in statements, statement forms, and searches.

In order to understand how quantifiers work in the context of statements, consider this example which uses all four of them.

```
Bob is assigned to no shifts.
Laura is assigned to 5 shifts.
Michael is assigned to many shifts.
Sandra is assigned to all shifts.
```

The first sentence has a clear semantic meaning in English. After this statement has been made, all previous assignments of Bob should be removed. Implementing this in Garlang is not possible with the current model of Datalog because we have not defined a semantic for retracting facts. This is a severe limitation that needs to be removed in future versions. The problem is discussed in more detail in the conclusion in section **??**.

The second and third statements are both invalid in Garlang. The problem with the number quantifier in the second statement is that this statement cannot be true if there are less than five shifts in total furthermore if there are more than five shifts it is unclear to what subset of them Laura should be assigned. The third statement is even less clear because many can mean any number of shifts bigger than two. To prevent these ambiguities, Garlang forbids both these quantifiers in the context of statements. This problem does not arise from the implementation model of Garlang but is caused by the fact that natural language is sometimes not as precise as we need it to be to define a runnable specification. If the user uses these quantifiers in a statement, Garlang should explain to the user why this statement is not specific enough and suggest corrections.

The last statement has a clear interpretation: All shifts that exist should be assigned to Sandra. Garlang translates this sentence to the following Datalog rule.

```
assignedTo(sandra, Shift):- kindOf(Shift, shift)
```

The same quantifiers can be applied to statement forms as well. Consider the following examples.

```
An employee is assigned to no shift.
An employee is assigned to 5 shifts.
An employee is assigned to many shifts.
An employee is assigned to all shift.
```

The first statement form is not very useful because it would forbid making such a statement. When the user defines a statement form with the no-quantifier, Garlang should warn the user that it will ignore this definition.

The second statement which uses the number quantifier enforces that an employee is assigned to exactly five shifts. This validation is evaluated in the generated form of the employee, and an employee can not be saved or created if it is not assigned to precisely five shifts. This quantifier can also be used to define mandatory fields when using the quantifier 1. By default, the relationships of a kind are one to one and optional like the statement form *"An employee is assigned to a shift"*

The third statement indicates that a shift can be assigned to any number of shifts. This information is used by Garlang in the automatically generated forms to provide multi-value input fields.

The last statement is technically not a statement form because Garlang interprets this statement as a general statement that all employees should be assigned to all existing shifts. It is translated to the following Datalog rule.

```
assignedTo(Employee, Shift):-
  kindOf(Shift, shift), kindOf(Employee, employee)
```

This rule is just a more general version of the example where the all quantifier was applied to a statement about a specific employee.

Quantifiers, in combination with searches, have two roles. They can be used to start a search, and they can be used inside of a search to express quantifier constraints. These two concepts will be discussed in the next section about searches.

6.4 Searches

Searches are used to find objects that match specific criterions. The all- and the number-quantifier can be used to define a search. In the most simple form, a search can consist of a quantifier and a kind. More complex searches will be discussed in the next section about complex sentences. Here are example searches using the two quantifiers.

```
all employees
5 employees
```

Garlang translates these searches into goals in Datalog. In this case, both searches are translated to the same goal because, in Datalog, we cannot express a limitation on the number of results. The limit of a search is applied after the Datalog goal has been evaluated. In the future Datalog could be extended to support partial evaluation of goals to optimize the speed of the search execution. The following Datalog goal represents the two searches.

```
?- kindOf(X, employee).
```

Quantifiers can also be used inside of searches. For the following examples, that-clauses are needed. They will be explained in the next section. Without these details, the four example searches can be understood as English sentence fragments. The examples search for all employees without a shift, with five shifts, with more than one shift and with all shifts that exist assigned to them.

```
all employees that are assigned to no shift
all employees that are assigned to 5 shifts
all employees that are assigned to many shifts
all employees that are assigned to all shift
```

Garlang translates the first search to Datalog using the negation extension. The result is the following goal.

```
? - kindOf(Employee, employee), ¬isAssignedTo(Employee, Shift), kindOf(Shift, shift)
```

In order to translate the other searches to Datalog, we need to define two new predicates first. The predicate *countAssignedShifts* is a helper that defines the number of assigned shifts for each employee. This predicate is specific to this example whenever a number-, many- or all- quantifier is used in combination with a statement Garlang has to generate such a predicate. The second predicate *instancesOfKind* defines how many instances each kind has. It is necessary to implement the all-quantifier. The two predicates have the following definition.

```
countAssignedShifts(Employee, count<Shift>):- isAssignedTo(Shift, Employee)
countInstancesOfKind(Kind, count<Instance>):- kindOf(Instance, Kind)
```

With these two predicates, the three quantifier statements can be translated into the following Datalog goals. The comparators $=$ and $>$ are built-in predicates.

```
?- kindOf(employee, Employee), countAssignedShifts(Employee, X), X = 0
?- kindOf(employee, Employee), countAssignedShifts(Employee, X), X = 5
?- kindOf(employee, Employee), countAssignedShifts(Employee, X), X > 1

?- kindOf(employee, Employee), countAssignedShifts(Employee, X),
countInstancesOfKind(employee, Y), X = Y
```

6.5 Complex sentences

What makes language expressive is the fact that statements are not stated in isolation. Instead, complex concepts can be described by combining multiple statements in a single sentence. English is highly expressive and allows a large number of different constructions. In comparison, Garlang is focused on supporting a minimal set of constructions that is sufficiently expressive. The two concepts discussed here are that-clauses and conjunctions. Complex sentences have been used extensively in the shift plan example. The following sections explain how they are implemented in Garlang.

6.5.1 That-clauses

A That-clause allows the user to attach an additional statement to an object. Consider this example that searches for all employees who have been assigned to at least one shift.

```
all employees that are assigned to a shift
```

The that-clause is attached to the employee and adds the requirement that the employee must be assigned to a shift. The attached statement has the form *"an employee is assigned to a shift"*. When attaching such a statement with a that-clause, the first noun in the statement form is omitted and implicitly set to the object to which the statement is attached. In this case, the statement has to be adjusted to match the plural form of the object. To match this statement even though it deviates from the original form, Garlang uses the heuristic discussed in section 6.2 about flexible noun order. This example is translated into the following Datalog goal.

```
?-kindOf(Employee, employee), isAssignedTo(Employee, Shift), kindOf(Shift, shift)
```

The Datalog goal is a combination of the *kindOf* predicate from the base search which asserts that the result has to be an employee and the *isAssignedTo* predicate which has been attached by the that.

Garlang allows stacking multiple that-clauses together. For example, we could attach a second that-clause to the shift to search for all employees who work together on a shift with Bob. This example, with its corresponding Datalog representation, looks like this.

```
all employees that are assigned to a shift that is assigned to Bob
```

```
?- kindOf(X, employees), isAssignedTo(X, Shift), kindOf(Shift, shift),
   isAssignedTo(Shift, bob)
```

That-clauses can also be used in regular statements. One typical example is the declaration of properties of an object. At first, consider the statement form of this example, which defines the name of an employee.

```
An employee has a name that is a text.
```

This sentence defines the statement form *"An employee has a name"*. A new kind *name* is mentioned here which has not been defined before. Garlang allows introducing new kinds in statement forms as long as the new kind is based on a previously defined kind. The that-clause is used here to define the base kind of name. It attaches the statement *"A name is a text"* to a name. These inline definitions are especially useful in combination with primitive values. A benefit of defining properties as new kinds is that it makes the matching of statements to statement forms more robust. Consider, if all properties of the employee are defined without new kinds like this.

```
An employee's name is a text.
An employee's address is a text.
...
```

All these statements have the same parameter kinds one employee and one text. When the same statements are defined with custom kinds, the statement can be differentiated purely on the kinds the objects in the relationship.

That-clauses are not limited to attaching kind assertions. They can be used to add any assertion. The attached statement is interpreted as a validation that is evaluated before any statement of this statement form is accepted. The restrictions can express arbitrary business logic. For example, the length of the name could be limited to a specific length. Because the validation rules are expressed in natural language Garlang can automatically generate validation messages that are understandable to the user.

That-clauses can also be applied to statements which state facts and are not statement forms. Based on the previous example, we can state Bob's name like this.

```
Bob has a name that is "Bob"
```

This statement seems to conflict with the rule stated in section **??** about referring determiners because general references are only allowed in the context of the definition of a kind. The interpretation of this statement is clear when reading it as an English sentence. The intention is to assign a new name to Bob. Garlang can deduce in this case, that *"a name"* is not a reference to any instance of the name kind but instead references specifically the text *"Bob"*. If the statements in this sentence are translated to Datalog, they will look like this.

```
has(bob, Name), kindOf(name, Name), is(Name, "Bob")
```

These are not valid Datalog clauses because ground facts cannot contain variables. In this example, we can replace the variable with a constant by applying the following reasoning. The relationship *"is"* can be interpreted as an equality between the variable Name and the text *"Bob"* which means that the variable could be substituted with the text "Bob". In order to allow this substitution, the predicate *kindOf(name, Name)* has to be fulfilled. Garlang can convert the text "Bob" implicitly to a name because name is a kind of text. The result of this substitution are the following Datalog clauses.

```
name = uniqueId()
is(name, "Bob")
is(name, new(name))
has(bob, name)
```

This example is the exception; otherwise, the same restrictions apply as with sentences without that-clauses.

In summary, that-clauses can be described with the following pattern.

[MainStatement | Search] that [RestrictionStatement]

The *that* always attaches to the last general reference in the search or main statement. In the restriction statement, the first noun is omitted and implicitly set to the noun which is referenced by the that.

6.5.2 Conjunctions

Conjunctions allow chaining multiple statements together. They have not been used in the shift plan example, but they are nonetheless important. For example, without conjunctions, that-clauses can only attach a single statement to an object. This section is just a rough outline of how conjunctions can be implemented in Garlang illustrated by looking at how the conjunction *and* should work. Further work is necessary to add additional conjunctions like *or*.

First of all, *and* can be used as a logical operation to combine multiple statements into one statement which evaluates to true if all of the conjoined statements are true. An example of this is a search that finds all of Bob's Monday shifts.

```
all shifts that are assigned to Bob and happen on a Monday
```

In this example, the statement forms *"A shift is assigned to Bob."* and *"A shift happens on a Monday"* are fused into a single statement which is attached with the that to the all shifts search. The resulting Datalog goal looks like this.

```
?- kindOf(X, shift), assignedTo(X, bob), happensOnMonday(X)
```

In English *and* can also be used in combination with nouns to create an enumeration of things. It is crucial that Garlang does not break the expectation of the user. Garlang is inherently more restricted than natural language, but the linguistic constructs that

Garlang adopts should work in the same way how the user knows them from their usage in natural language. Therefore *and* can also be used in Garlang to describe enumerations like in this example.

```
Bob and Sandra are assigned to the example shift.
```

This statement can be viewed as a shorthand form for writing the same statement twice with each person individually. Garlang would translate this sentence to the following Datalog clauses.

```
isAssignedTo(bob, exampleShift)
isAssignedTo(sandra, exampleShift)
```

6.6 Rules

Rules allow the user to make more general statements. Instead of stating facts for individual instances with rules facts can be automatically deduced for a whole group of instances that match specific criterions. For example here is a rule that adds a "is assigned"-fact to all shifts that have been assigned to an employee.

```
a shift is assigned when:
    a manager assigned an employee to the shift.
```

This rule is translated to the following Datalog representation.

```
isAssigned(Shift):- assignedTo(X, Y, Shift)
```

The form of the Garlang rule is very similar to the Datalog representation. The head of the Datalog clause is described by the sentence *"a shift is assigned"* which is mapped to the predicate *isAssigned*. The body clause consists of the sentence *"an manager assigned an employee to the shift"* which is mapped to the predicate *assignedTo*. A definite article is used for "the shift" to indicate that this shift refers to the shift from the head clause.

Whereas the manager and employee are referred to with indefinite articles because the shift should be considered assigned as long as any manager has assigned any employee. The Datalog representation reflects the semantic one would expect from interpreting the rule as plain English. The shift argument is the same variable as the shift in the clause of the head while the employee and manager argument of the *assignedTo* predicate are unrestricted variables. Garlang allows the order of the premise and the conclusion to be switched. The previous rule could be rephrased like this:

```
when a manager assigns an employee to the shift:
    the shift is assigned
```

Both forms are semantically equivalent. They can be described formally with these two patterns:

[Statement] **when**: [Statement]

when [Statement]: [Statement]

There is another rule form which is introduced with the keyword *after* instead of *when*. An example of this is the rule which allows the assignment of employees by dropping them on a shift.

```
After an employee is dropped on a shift:
  The employee is assigned to the shift.
```

After-rules can be used to create new facts that are permanent. In this example, the employee keeps being assigned to the shift after the drop event is no longer happening. In contrast, the facts that are generated by when-rules exist only as long as the premise of the rule is true — both *when* and *after* generate the same Datalog representation. The difference is how the generated facts of them are treated. How the retraction of statements works in Garlang is described in the next section.

6.6.1 Retraction of facts

The retraction of facts is necessary when rules are applied to changing facts. Every time the ground facts change the rules that apply to them have to be reevaluated and previous facts generated by the rules have to be retracted. Ground facts either change when the user modifies an object or when the Garlang environment publishes new facts, for example, when the user starts dragging an object.

The standard model of Datalog does not allow the retraction of facts. This is a gap in the language that needs to be addressed. There are extensions to Datalog like the approach of Lam and Cervesato (2012) which adds fact retraction. Incorporating such a solution to Garlang is beyond the scope of this paper. Instead, a simpler approach is described here that works without altering the semantics of Datalog with the trade-off that it is less efficient.

What makes retracting difficult is the problem that we need to track which rules applied to the retracted fact and what facts have they generated that are no longer valid. An alternative is that Garlang recomputes all rules whenever the ground facts change. For that, Datalog has to track which facts have been added by a rule. Whenever a ground fact changes, all derived facts have to be removed, and all rules have to be reevaluated on the new ground facts.

After-rules are an exception because the facts they generate should persist after the premise of the rule has been true once. Facts generated by such rules should be treated as ground facts. This also means that after-rules can trigger a reevaluation of all Datalog Rules. Oftentimes after-rules are used in combination with "event"-facts like when an object is being dropped. These facts should be only active during a single reevalution to avoid infinite loops caused by after-rules applied to "event"-facts.

Overall this solution requires many redundant recomputations of rules, but it is simpler to implement. This is an acceptable trade-off for a prototype which is only used to evaluate the usability of Garlang. For a more practical implementation additional work is required to improve the performance. The paper of **??** goes into Detail how Datalog can be incrementally reevaluated in Chapter 4 about incremental maintenance.

6.7 Value Statements

The previous section explained how rules work that defined yes/no statements. Besides these simple rules, the shift planner example also introduced rules that calculate values like the monthly hours of an employee. To implement rules which express calculations Garlang has a special statement form called value form. A value form can be constructed from a regular statement form by associating a value with it. A value form has the following pattern.

[StatementForm] **is** [Value]

Following this pattern, the age of an employee could be defined as a value form like this.

```
An employee's age is a number.
```

In this case, the statement form *"An employee's age"* is associated with a number by using an is-construction to create a new value form. This value form can be used as a regular statement form. For example, it can be used to state the age of Bob.

```
Bob's age is 35
```

A statement like this is called a value statement. Internally each value statements is represented with a single Datalog predicate, the same way as a regular statement. The previous statement has the following representation in Datalog.

```
ageIs(bob, 35)
```

What differentiates value statements is the fact that each of them has a corresponding value expression that can be used to refer to its value. When a statement matches only the statement form part of a value form it is interpreted as a value expression. For example, this is the value expression that refers to the age of Bob.

```
Bob's age
```

This is translated to the following Garlang goal which returns the age of Bob.

```
?- ageIs(bob, X)
```

Value forms and value statement are a generalization of the is-relationship. Previously the is-relationship has been introduced as an equality relationship between objects. With value forms, is-constructs can also be used to express an equality relationship between a statement and an object.

6.7.1 Operators

Garlang supports various operators for example to perform mathematical calculations or to aggregate results. These built-in operators are implemented as custom functions which are exposed to the end user as value forms. For example, the plus-operator has the following form.

```
a number + a number is a number
```

This operator could be used to calculate the result of Bob's age plus one like this.

```
Bob's age + 1
```

This expression would be translated to the following goal which returns the sum as a result.

```
? - ageOf(bob, Age), plus(Age, 1, Y)
```

In the Datalog goal, the predicate plus refers to a built-in operator. Because Garlang implements the plus operator as a custom function it gets called after all other variables in the Datalog clause are resolved. In this example, age would be resolved to 35 and passed together with 1 to the plus-function which runs the calculation and resolves as a result Y with 36. Unlike regular Datalog predicates, built-in predicates expect all arguments which occur in the statement form part of the definition to be a constant and the assigned value to be a variable. For example, this statement would be invalid because the plus function expects constants as the first two arguments and a variable as the third argument.

```
a number plus Bob's age is 5
```

Aggregations are exposed to the user the same way as other built-in operators. For example, the "number of"-operator which returns the count of a result has the following form.

```
the number of some objects is a number
```

The some-quantifier indicates that this operator can be only applied to a list of objects. Searches return lists of objects. They can be used together with value expressions. Consider as an example, how the "number of"-operator can be used to count the number of results of the search "all employees".

```
the number of all employees
```

Aggregators are internally implemented with the aggregation extension of Datalog. Because aggregations can be only used in the head of a clause Garlang has to generate a temporary rule to evaluate this example. In this example, the temporary rule is called *aggregatorValue*. In the real implementation, Garlang would generate a globally unique name for it to avoid name collisions.

```
aggregatorValue(count<Employee>):- kindOf(Employee, employee).
```

The resulting goal then queries for the value of the rule.

```
?- aggregatorValue(x)
```

6.7.2 Special value form

In previous examples properties of an object like age have been defined with a has-statement and a that-clause. Unfortunately, the is-construction from the previous section can't be applied to this form. Garlang solves this problem by automatically generating value forms for each property declaration. This generated value form has the following pattern.

the [Kind] **of** [Object] **is** [Object]

This form can be used as an expression to access both singular values like the name of an employee or values that an object has multiple of like the shifts of an employee.

```
the name of Bob
the shifts of Bob
```

The value form described by this of-relationship is equivalent to the original has-relationship. The conversion between these forms works in both directions. If a rule is defined with a of-relationship the equivalent has-relationship is automatically generated and the other way around. This flexibility allows the user to pick the most natural form. An example of a rule using the value form is the definition of the monthly working hours of an employee.

```
The work time of an employee per month is a duration: ...
```

As a has-relationship, an equivalent rule can be defined like this.

```
An employee has a work time per month that is a duration: ...
```

Defining this special value form for has-relationships is not the most elegant solution. There is a more general pattern behind this. For example, consider this statement form that defines how much an employee works per week.

```
An employee works a duration per week.
```

This is not a has-statement but it can be similarly restructured to turn it into a statement that references the working time of an employee per month.

```
The duration an employee works per week.
```

From this example, we might infer the logic that a statement can be turned into a value expression by reordering the statement: The object we want the expression to refer to has to start the sentence and we need to refer to it with a determinate article. The problem with this logic is that there is no clear way how Garlang can differentiate between this special case and regular definite determiners that reference a previously defined object. This problem can be illustrated with the following example which describes the leader of a department.

```
A department is lead by a manager.
```

With this statement form, we could formulate these two statements.

```
The manager leads the department
The manager leading the department
```

When read as English text there is a clear interpretation. The first line makes a statement that the manager is the leader of the department and the second line is an expression that refers to the manager of the department. Garlang is not able to differentiate this nuance because of its limited language model. Unlike previous ambiguous cases, there is no simple heuristic that can differentiate between these two cases. Instead of guessing randomly Garlang implements this special form for has-statements which covers most practical cases as we have seen in the shift planner example. This issue should be addressed in future iterations of Garlang.

6.7.3 Rule examples

The original motivation to introduce value statements was to allow the definition of rules that can calculate values. In this section, we will dissect two rules from the shift planner example to explain how value expressions are used in the context of rules. The first rule calculates the monthly work time of an employee.

```
The work time of an employee per month is a duration (monthly work time) when:
  The monthly work time: the weekly work time * 4
  The employee works a duration (weekly work time) per week.
```

This rule uses the value form to define the monthly work time. The value is a duration, and it is given the local name "monthly work time". The body of the rule creates a local definition for the monthly work. This definition references the weekly work time, which is defined in the line below. Statements in a rule body do not follow a specific order. Because the statement in the second line has no value form a local name is used to refer to the duration.

When translating this rule multiple Datalog rules are generated. It is necessary to split this rule up because a Datalog clause can only have a single predicate in the head. The monthly work time calculation generates two predicates because the assertion that an employee has a worktime consists of the two statements: "an employee has a worktime" and *"a worktime is a duration"*.

First of all, Garlang generates a helper function that calculates that contains the logic to calculate the duration. In this example the function is called *hasMonthlyWorkTime.* In the real implementation, the name would be a globally unique id. The helper function has the following definition.

```
hasWorkTimePerMonth(Employee, Duration):
    kindOf(Employee, employee),
    kindOf(Duration, duration),

    multiply(WeeklyDuration, 4, Duration),

    worksPerWeek(Employee, WeeklyDuration),
    kindOf(WeeklyDuration, duration).
```

The first two clauses in the body are generated by the definite references to employee and duration in the condition of the Garlang rule. The third clause maps to the multiply operator of the duration kind which accepts a number and a duration to create a result duration. The last two clauses are generated by the statement that references the weekly work time of the employee.

The next two rules use the value of the helper function to create the facts that are actually needed. The first rule adds the work time kind to the duration.

```
is(Object, new(workTime)):
    is(Object, Duration),
    hasWorkTimePerMonth(Employee, Duration).
```

Finally, the second rule adds the work time that the previous rule created to the employee.

```
has(Employee, Worktime):
    hasWorkTimePerMonth(Employee, Duration),
    is(Object, Duration),
    is(Object, WorkTime).
```

Value statements can also be used with yes/no rules. An example of this is the rule that determines the availability of an employee during a month.

```
An employee has available time during a month when:
     The worked time is less than the total work time.

    The total work time: the total work time of the employee per month.
    The worked time: the worked time of the employee during the month
```

This Garalang rule can be translated to the following Datalog representation.

```
hasAvailableTimeDuring(Employee, Month):
    kindOf(Employee, employee),
    kindOf(Month, month),

    lessThan(WorkedTime, TotalWorkTime),

    has(TotalWorkTime, Employee),
    kindOf(totalWorkTime, totalWorkTime),
    has(WorkedTime, Employee),
    kindOf(WorkTime, workedTime),
```

The rule follows a similar pattern to the previous rule. The variables of the *hasAvailableTimeDuring* predicate are restricted to be an employee and a month. The less than comparison is mapped to the *lessThan* predicated defined by the duration kind which is defined in terms of the built-in $<$ comparison for numbers. The comparison is implemented as a custom function which only generates a matching clause if the less than relationship is fulfilled. The two values *"total work time"* and *"worked time"* are referenced by their kind.

6.8 Custom object literals

Garlang has special literal forms for numbers and text. With value forms, the users can define custom literals for their own kinds based on number and text literals. Many of the built-in kinds of Garlang also provide literal forms using value forms. As an example, consider the duration kind which has the following definition.

Duration <plural durations>

```
A duration has a value that is a number.
...
```

The value of the duration is the time in seconds. Without a literal form for the duration, the definition of the work time of an employee requires 3 statements.

```
The work time is a duration.
The value of the work time is 40 * 60 * 60.
Bob works the work time per week.
```

Instead it would be more convenient if the user could declare durations with expressions like "40 hours" or "1:00:00". As an example, the hour literal can be implemented with the following value form.

```
a number (hours) hours is a duration when:
    the value of the duration is the hours * 60 * 60
```

Because of Garlang's heuristic-based matching, the value form is flexible and can be also used in the singular like *"1 hour"*

6.9 Negation

Negation is another concept that Garlang can borrow from the English language. With negation, a statement can be turned into a negative statement that asserts that the original statement is not true. In English, the negative form of a statement can be constructed with the adverb not. Consider these two example statements with their negated version.

```
Bob is assigned by Sandra to the example shift.
Bob is not assigned by Sandra to the example shift.

Sandra assigns Bob to the example shift.
Sandra did not assign Bob to the example shift.
```

The main differences in both of these examples between the positive and the negative statement is the word *not*. There are cases where the rest of the sentence has to be adapted as well, like the second sentence, but this can be dealt with by the heuristic matching. Generally, Garlang can use the existence of the word *not* as an indicator that a statement should be matched to the negated version of the original predicate.

In Garlang, negations could be potentially used in five different contexts: statements, statement forms, rules, and searches. It is necessary that we look at each context to ensure that the semantics of not in Garlang match the semantics of how not is used in English.

The primary use case of negation is to define negative constraints. Such constraints can be applied in searches and rules. This explanation uses the highlight-rule of the shifts as an example, but negative constraints can be applied to searches in the same way. The highlight-rule has the following definition.

```
A shift is highlighted when:
    The shift is not fully assigned.
```

The translation of this rule to Datalog is straight forward. Because the "is fully assigned"-statement is negated the resulting predicate in the Datalog representation is negated as well.

```
isHighlighted(Shift): kindOf(Shift, shift), ¬isFullyAssigned(Shift).
```

Negations can also be used to define rules as a negative form. For example, instead of defining when a shift is assigned with a negative form, we can define when a shift is not assigned.

```
a shift is not assigned when:
    No employee is assigned by a manager to the shift.
```

This rule is translated to the following Datalog representation.

```
isNotAssigned(Shift):
kindOf(Shift, shift),

 ¬isAssignedByTo(Employee, Manager, Shift),
kindOf(Employee, employee),
kindOf(Manager, manager)
```

When Garlang matches statements to a rule defined in the negative form the negation works in reverse. If the statement contains no not the statement should be matched to the negated version of the predicate if it does contain a *not* it should be matched to the original version of the predicate.

Using not in the context of statements is redundant. For example, stating *"Bob is not assigned by Sandra to the example shift."* has the same effect as not making this statement. We have used the closed world assumption to add negation to Datalog. Therefore, for any positive fact, that is not stated, we assume that the negated fact is true. Negative statements could become useful if we added the ability to retract statements to Datalog. A negative form could then be interpreted as a retraction of a previous statement. Determining how the semantics of this work in detail is outside the scope of this paper and has to be addressed in future works.

Negations in statement forms behave similarly to negations in statements that state facts. A potential use case is to override previous statement forms. For example, in the manager kind, we could define the following statement form to disallow the assignment of managers to shifts.

```
A manager is not assigned by a manager to a shift.
```

While this might be useful in some situations, this overriding semantic conflicts with the idea that kids are different perspectives on the same object. Instead, this problem can be solved by modeling the problem differently and moving the assignment statement form to a separate kind that manager is not based on.

7 Summary

7.1 Related work

Garlang takes a lot of inspiration from the programming language Realtalk which is an experimental programming language developed by the Research Lab Dynamicland [3]. Realtalk is part of a bigger research project that explores a new computational medium that is embedded in the physical world.

Realtalk is an extension of Lua that adds a natural-language-like version of Datalog to it. Rules are defined with *"when"*, and facts are stated as claims and wishes. Programs in Dynamicland are attached to physical objects which can communicate through Realtalk. It has a centralized Datastore of all facts and wishes that all programs can access. This example from Rizwan (2018) describes a physical object which adds the label "it's a bird" when it is pointed at another object that claims to be an airplane.

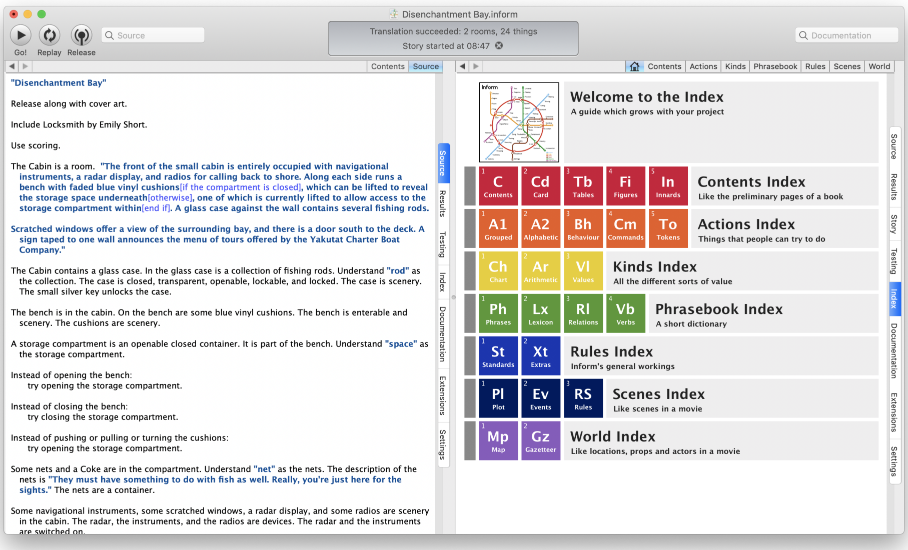
```
When /a/ is a "airplane",
     /a/ points "down" at /p/:

    Wish (p) is labelled "it's a bird!"
End
```

From Realtalk Garlang borrows the idea of formalizing natural language by mapping sentences to Datalog predicates. While Garlang's ultimate goal is to be as natural as possible, Realtalk is more pragmatic. It allows users to mix Realtalk rules and claims with ordinary Lua code. This makes Realtalk more powerful, which allows it to be bootstrapped, but in return, it requires users to learn at least some Lua. Instead, Garlang pushes conventional program code to the boundaries of the system by encapsulating it in custom built-ins.

Another programming environment that was very inspirational to Garlang is Inform 7[4], which is a design system for interactive fiction. It is one of the very few natural

[3]https://dynamicland.org/
[4]http://inform7.com

programming languages that is actually based on natural language constructs instead of only attaching a more natural looking syntax to a conventional programming language. From Inform 7 Garlang borrowed the idea of using metaphors from existing textual mediums. Examples of this are the concept of a dictionary as the representation of the knowledge model or reusing syntactic conventions like parenthesis as the basis for the language construct of local names.



```
"Disenchantment Bay"

Release along with cover art.

Include Locksmith by Emily Short.

Use scoring.

The Cabin is a room.  "The front of the small cabin is entirely occupied with navigational
   instruments, a radar display, and radios for calling back to shore. Along each side runs a
   bench with faded blue vinyl cushions[if the compartment is closed], which can be lifted to reveal
   the storage space underneath[otherwise], one of which is currently lifted to allow access to the
   storage compartment within[end if]. A glass case against the wall contains several fishing rods.

Scratched windows offer a view of the surrounding bay, and there is a door south to the deck. A
   sign taped to one wall announces the menu of tours offered by the Yakutat Charter Boat
   Company."

The Cabin contains a glass case. In the glass case is a collection of fishing rods. Understand "rod" as
   the collection. The case is closed, transparent, openable, lockable, and locked. The case is scenery.
   The small silver key unlocks the case.

The bench is in the cabin. On the bench are some blue vinyl cushions. The bench is enterable and
   scenery. The cushions are scenery.

A storage compartment is an openable closed container. It is part of the bench. Understand "space" as
   the storage compartment.

Instead of opening the bench:
     try opening the storage compartment.

Instead of closing the bench:
     try closing the storage compartment.

Instead of pushing or pulling or turning the cushions:
     try opening the storage compartment.

Some nets and a Coke are in the compartment. Understand "net" as the nets. The description of the
   nets is "They must have something to do with fish as well. Really, you're just here for the
   sights." The nets are a container.

Some navigational instruments, some scratched windows, a radar display, and some radios are scenery
   in the cabin. The radar, the instruments, and the radios are devices. The radar and the instruments
   are switched on.
```

Figure 7.1: Inform 7 IDE with its dictionary

Besides these direct influences, there is also a broader historical context that has to be considered. The main focus of this paper is to sketch out a new paradigm for information software as an alternative to the prevalent GUI model. Relating all contained ideas to their origin is beyond the scope of this paper. This section provides a few pointers to guide future research to embed the concept of this paper in a broader historical context.

The introduction just touched on the topic of programming as a new kind of literacy to reflect the current state in our culture. The original idea goes back all the way to the beginnings of computing in 1960. Alan Perlis advocated in 1961 that learning to program should be mandatory for all undergraduates, the same way all of them have to take a writing course Vee (2013). Since then numerous environments have been developed to make programming more accessible for non-programmers. From this, the field of end-user development emerged. Besides natural language programming, other popular approaches are programming by example, visual programming, and spreadsheets (Paternò). A thorough survey is necessary to embed Garlang within that field.

The end-goal of the concept proposed in this paper is not just to make programming accessible to end-users so they can generate apps in the existing GUI application world. Instead, our solution should provide a uniform environment that utilizes these new abilities of the user. The user should be able to freely combine different pieces of software and alter them as they need without barriers between applications. Besides the "boxed application"-model there exist other models for software that have similar goals to Garlang.

The Unix philosophy is a very popular example. It relies on the common abstraction that everything is a file. Instead of building big applications that solve complex problems it advocates for single-purpose tools that can be combined into specific solutions to solve the exact problem of the user (Doug McIlroy (1978)).

Another more recent examples are Mashups which have become popular with the Web 2.0 movement. A web app is seen as a provider of some service that can be combined with other applications. This is possible through the APIs they provide. An example for a current iteration of a mashup is the app IFTTT (If this then that) [5] which allows the user to create new workflows from the apps they use.

These are just two examples more research is necessary to compile a more exhaustive list of alternative models which Garlang can be compared to.

7.2 Conclusion

This paper tries to bridge the dichotomy that exists in software today where there is a strict separation between the people who use software and the people who build software. It investigates the question why there is such a large discrepancy between the flexible nature of software and what the end-user experiences: fixed applications that are built for a specific purpose that cannot be adapted and are hard to combine with one another.

This paper identifies the GUI pattern as a major cause of this problem. The GUI gives the inherently shapeless software a quasi-physical shape and thereby it loses a lot of its flexible qualities. With a GUI the only way to get access to the underlying representation is manual interaction.

[5] https://ifttt.com/

In order to tackle this problem, the scope of this paper is limited to information software. This is software that helps the user to learn things, to get answers to a question, compare different alternatives and come to a conclusion. In this category of software, the focus of the user is to answer a question. This is a mismatch with the GUI pattern which focuses on the manipulation of virtual representations. GUIs also lacks a representation for complex questions. The user can only express them by navigation through the software to manually piece together the information they need.

As an alternative, this paper proposes a concept for a programming environment based on natural language called Garlang. Instead of utilizing the user's knowledge about the physical world, as GUIs do, it utilizes their knowledge about language. Language has the benefit that it is able to express complicated concepts. Another advantage of language is that is accessible for the user to edit and create textual representation. Compared to interactive interfaces which are a read-only medium with the exception of the programmers who built it.

Garlang uses language representations for both the knowledge model of the application and the questions of the user which are represented as searches on the knowledge model. This gets the user closer to the actual capabilities of the software.

The environment consists of two parts the dictionary and the explorer. The dictionary represents the knowledge model of the application. It describes the kind of objects the application can represent together with their characteristics, the relationships between the objects and the computations that can be performed on them. Garlang uses the metaphor of a Dictionary to provide a representation that is familiar to the user.

The explorer is the main interface of the Garlang environment. It helps the user find answers to their questions. The user can express their questions as searches based on the concepts defined in the dictionary. They can display the results of a search in different views, compare them with alternative results, or run additional searches to provide more context.

Garlang has intentionally no separation between development environment and runtime. It provides a unified environment that allows going back and forth between using an application and modifying it without switching contexts. This helps to remove the barrier of entry for moving from a passive to an active user. This duality between using and modifying an application is reflected both in the dictionary and the explorer. The dictionary can be used as a reference for the user to understand how they can formulate their searches. At the same time, the dictionary serves as an editor that allows the user to extend the existing definitions. The explorer follows the same pattern. The user can start by loading a template for their specific use case with preconfigured searches. If the configuration of the template is not sufficient for their problem they can just adapt the tool in the context of their problem without having to switch into a different mode.

The concepts of Garlang rely heavily on a good language representation for both the knowledge model and the searches that the user can run on the knowledge model. Natural language is easy to understand for humans but computers have a hard time interpreting it. Especially the ambiguous nature of language and the context-dependent interpretation are a problem that hasn't been solved yet. To solve this problem, Garlang defines a naturalistic controlled language that tries to be naturalistic enough to be learnable by end users and at the same time formal enough that a computer can generate a runnable program from it.

Garlang interprets language as a combination of statements. Garlang's model of language is not based on complex natural language processing; Instead, it parses statements by identifying all nouns in the sentence. The nouns become the objects in the statement. The remaining text is used as an identifier to match it to one of the predefined statements in the dictionary. Internally each statement in Garlang is translated to a Datalog predicate. Garlang supports a limited set of grammatical constructions like determiners, negation, conjunctions, and that-clauses. In combination, these are sufficient enough to express rules, searches, and basic computations.

In order to illustrate the feasibility of the language, this paper walked through a hypothetical scenario of a shift manager who wants to know how to distribute his employees to make a shift plan. This scenario demonstrated that Garlang can model non-trivial problems. It also showed how Garang can make specific applications like a shift planner potentially redundant because the user is provided with a base model, in this case, a calendar, and they can build the specific tool they need themselves.

Garlang is not the ultimate solution that will replace professional software development, but it points to a potential future where the roles of developers and users are changed. Users are empowered to build their own tools which makes them more independent of developers. Professional developers are still necessary. While Garlang allows users to write custom definitions in a naturalistic programming language, modeling a knowledge domain with it still requires skill. Most end-users probably do not want to write their own applications from scratch. They have a problem they need to solve. For that, they want a predefined model that they can adjust to fit their needs. Garlang is also not completely bootstrapped there are some things that have to be programmed outside of it like adding custom built-in predicates, defining new visualization and custom input for new data types. Instead of the software landscape, we see today where the end user has to pick a ready-made solution a system like Garlang could become an ecosystem where developers provide tool-kits that can solve problems in a specific problem domain and the end user can mix and match tool-kits to assemble the tool that solves exactly the problem they have.

7.3 Future work

What this paper proposes is only a concept so far. The next step is to create a series of prototypes to evaluate different aspects of Garlang. The biggest challenge of this project is to find a solution that appeals to end-users. Developers will be much easier to convince to build software for a different platform once they see that there are potential users.

Appealing to end-users mainly boils down to two challenges. First of all the language has to be easily learnable. This paper just laid the groundwork and showed that such a language is theoretically possible. More extensive user studies are necessary to find a solution that works. This is not only a problem of designing the language but also designing an environment that supports the user, guiding them to solve their problem.

The second problem is how to communicate to end-users what they can do with an environment like Garlang. Users are trying to solve a specific problem. They do not want a programming environment they want a solution to their problem. Instead of selling Garlang as a programming environment it has to be communicated to the user as a solution to a specific problem where the solution then benefits from the flexibility that it is part of a full programming environment.

References

Determiners. *URL https://dictionary.cambridge.org/us/grammar/british-grammar/determiners-the-my-some-this.*

Ceri, S., Gottlob, G., and Tanca, L. (1989). What you always wanted to know about datalog (and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166.

Clark, P., Murray, W. R., Harrison, P., and Thompson, J. (2010). Naturalness vs. predictability: A key debate in controlled languages. In Fuchs, N. E., editor, *Controlled Natural Language*, pages 65–81. Springer Berlin Heidelberg.

Doug McIlroy, E. N. Pinson, B. A. T. (1978). Unix time-sharing system: Foreword.

Green, T., Huang, S., Loo, B., and Zhou, W. (2013). *Datalog and Recursive Query Processing.* Foundations and trends in databases. Now Publishers.

Lam, E. S. L. and Cervesato, I. (2012). Modeling datalog fact assertion and retraction in linear logic. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, pages 67–78. ACM.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710.

Obama, B. (2016). Weekly address: Giving every student an opportunity to learn through computer science for all.

Paternò, F. End user development: Survey of an emerging field for empowering people. 2013:1–11.

Prensky, M. (2006). Listen to the natives. *Educational Leadership*, 63(4):8–13.

Pulido-Prieto, O. and Juárez-Martínez, U. (2017). A survey of naturalistic programming technologies. *ACM Comput. Surv.*, 50(5):70:1–70:35.

Rizwan, O. (2018). Notes from dynamicland: Geokit. *URL https://rsnous.com/posts/notes-from-dynamicland-geokit/background-on-realtalk.*

Vee, A. (2013). Understanding computer programming as a literacy. 1(2):42–64.

Victor, B. (2006). Magic ink: Information software and the graphical interface, 2006. *URL http://worrydream.com/MagicInk.*