Автономная некоммерческая организация
высшего образования
«Университет Иннополис»

Autonomous noncommercial organization
of higher education
“Innopolis University”

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ПО НАПРАВЛЕНИЮ ПОДГОТОВКИ
09.03.01 – «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

GRADUATE THESIS FILED OF STUDY
09.03.01 – «COMPUTER SCIENCE»

НАПРАВЛЕННОСТЬ (ПРОФИЛЬ) ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ
«ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»
AREA OF SPECIALIZATION / ACADEMIC PROGRAM TITLE:
«COMPUTER SCIENCE »

Тема	**Семантический язык запросов к темпоральным генеалогиям**
Topic	**Semantic Query Language for Temporal Genealogical Trees**

Работу выполнил / Thesis is executed by	**Грязнов Евгений Валентинович** **Gryaznov Evgeniy Valentinovich**	подпись / signature
Научный руководитель / Thesis supervisor	**Мануэль Маццара** **Manuel Mazzara**	подпись / signature

Иннополис, Innopolis, 2018

SEMANTIC QUERY LANGUAGE FOR TEMPORAL GENEALOGICAL TREES

To my parents and close relatives. This work would not be possible without their help.

Acknowledgements

We would like to express our gratitude to professor Manuel Mazzara, who kindly agreed to be our academic supervisor, and whose help and support were virtually invaluable during the course of our work.

Additional thanks go to professor Nikolay Shilov for his useful advices and examination of the original topic proposal.

Contents

List of Figures

Abstract

We, as human beings, were always interested in the history of our families. From the medieval times the subject of genealogy began to gain prominence, and now its considered one of the most important areas of history. Today computers play a crucial role in the modern ancestry management, they are used to collect, store, analyse, sort and display the genealogical data. However, current applications do not take advantage of the structure of a kinship system, and therefore they are inaccessible for a typical user.

In this thesis we propose a new domain-specific language, called KISP, based on a formalisation of the English's kinship system, for accessing and querying traditional genealogical trees. We implement its interpreter together with an ancestry visualizer and a virtual assistant that helps the user in genealogy management. KISP is a dynamically typed LISP-like programming language with such features as kinship term reduction and temporal information expression.

Our solution provides the user with a coherent genealogical framework that allows for a natural navigation over any traditional family tree.

In the future work we would like add support for other, non-English systems of kinship, extend KISP with new features and make the virtual assistant more intelligent.

Chapter 1

Introduction

From the very beginning of mankind we assembled, compiled and analyzed the information about our ancestors. The study of genealogy is regarded as a noble enterprise, often pursued with extreme care and diligence. Not only it satisfies the desire to carve out a place for one's family in the larger historical picture, but also provides the sense of responsibility to preserve the past for future generations. However, as a genealogist goes deeper into the family history, its tree grows exponentially, thus making his work more and more arduous.

With the advent of computers, we are able to manage genealogies of incredible size. Now, due to the progress in storage engineering, it became possible to maintain and expand existing ancestries of considerable size. But merely keeping the data on physical disks is not enough. To sufficiently realize the full potential of computers in genealogy management, one should also provide means of inquiry in ancestral data.

If we want to teach computers understand lineage, we need to construct some type of artificial language that will allow us to effectively navigate and query any possible family tree. But observe, that an ancestry already has its own idiosyncratic terminology and grammar, which can be successfully used as a natural basis for such a language. Our research is an attempt to do exactly that.

There is another important concept to consider when working with family trees, namely the concept of *time*. A genealogy can exist only in specific time framework that is imposed on it by the very nature of history itself. As a result, any computer representation of an ancestry that lacks this framework is exorbitantly inadequate. Therefore, its preservation is a crucial feature for any software that is aimed for effective genealogy management.

1.1 The Goal

Our main goal is to design and implement a programming language that will allow us to *efficiently* query temporal genealogical trees, while at the same time being as natural as possible. Secondary goal is to develop a software that effectively utilizes this language to provide a convenient user interface for working with ancestries.

There are three requirements that we want our language to satisfy:

1. **Expressiveness**. The language should allow for any possible consanguine as well as affinal relations to be described.
2. **Speed**. The response time must not exceed the standard for an interpreted language.
3. **Simplicity**. Language should be able to express natural kinship and temporal terms as straightforward as possible.

Here the phrase "response time" stands for the time passed between the start and finish of a programs evaluation. The last quality is what truly distinguishes our approach from the rest, allowing for the most obvious representation of genealogical and temporal information.

1.2 Existing Solutions

Of course, there are many applications available for working with family trees. According to the web resource [1] there are at least 1015 programs for genealog-

ical management. One may say that given such an abundance of software, there is nothing more to add. However, most of it accomplish goals that are drastically different from ours. Moreover, we shall see that popular solutions are over-engineered and suffer from what is known as *scope* and *feature creep*, which means that the software is overfilled with functionality and it goes far beyond its original scope.

1. **Family Tree Builder**. Started as *Software as a Service* in 2003, now became one of the most widespread [2] applications in this area. Has a lavish set of functionality, including relative and history matching, detailed privacy, family sites, maps and many more. Judging from the image B.1 of its main screen we can say that it is a Microsoft Office of genealogy software.

2. **Neo4j**. A graph database management system [3] written in Java. Initially released in 2007, it is now considered to be the most popular graph database according to DB-Engines ranking [4]. It is feasible, although unacceptable to a non-specialist, to create a temporal *ontology* of a family tree using Neo4j and its domain specific language Gremlin. However, any such attempt would fail to take advantage of the structure of kinship language, due to Gremlin's general nature.

3. **RootsMagic**. This application [5], similarly to Family Tree Builder, initially was released in 2003, it ships with full package of diverse features, such as single-file database, four navigational views, and many more.

During our review of equivalent software, we found out that existing solutions for ancestry management are either inadequate in one or more ways, or accomplish quite different goals. There is no application available that would sufficiently achieve our main aim with all its stated qualities.

1.3 Thesis Outline

In the following five chapters we will review the existing literature concerning our topic, discuss the methodology of our work, present the project's implementation, demonstrate its correctness and conclude the results.

In the second chapter "Literature Review" we examine scientific articles, books and journals from three major categories: Knowledge Representation, Ontology Building and Natural Language Processing. We focus our attention primarily on the first two, since the study of lineage is a special case of knowledge management. Moreover, we inspect various anthropological sources, such as works of Claude Levi-Strauss [6], which concerns the nature of kinship structures in different societies.

In the third chapter "Methodology" we formalize the kinship language of one particular culture, namely traditional American culture in the sense of Read [7]. Then we explore its many mathematical properties and show how it can fit into the larger framework of a programming language.

In the forth chapter "Implementation" we present the composition of the complete project, including full documentation of our new LISP-like programming language. In particular, three system's components, namely Virtual Assistant, Genealogy Manager and Interpreter are discussed.

In the fifth chapter "Evaluation and Discussion" we show how our application was tested. We use three methods to perform the audition: Assertion, Unit and Performance Testing, in which we benchmark the interpreter for our new programming language.

Finally, in the last sixth chapter "Conclusions" we compile and demonstrate the results of our work. Specifically, we confirm that our system indeed achieves the main goal of this thesis and satisfies all proposed qualities.

Chapter 2

Literature Review

The purpose of this chapter is to give a brief survey of academic literature with respect to our thesis. There are two major areas of Computer Science that are related to out research: Knowledge Representation (KR) and Natural Language Processing (NLP). We will examine them one by one and highlight some of the most important works.

2.1 Knowledge Representation

The field of Knowledge Representation is concerned with how the knowledge about our physical world can be stored, managed and utilized by computers. KR owes its existence to the more general field of Artificial Intelligence, which prompted the study of encoding information about the physical reality into an intelligent system in such a way that it can be used by that system to solve complex problems.

The main presupposition of the whole field of KR is that, in order for an intelligent agent to resolve a difficult problem, it needs an access to some form of a knowledge specific to a particular domain and that knowledge should be stored inside the agent. This presupposition is now widely known as the *Knowledge Representation Hypothesis*:

> Any mechanically embodied intelligent process will be comprised of structural ingredients that (a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits (b) independent of such external semantical attribution, play a formal but causal and essential role in engendering the behavior that manifests that knowledge.

This original formulation of the hypothesis is due to Smith [8].

Brachman and Levesque outlined the hypothesis in their article *Expressiveness and tractability in Knowledge Representation and Reasoning* [9]. The authors argue that the trade-off between *expressiveness* of a knowledge-based system and its *tractability* (i.e. the ability to reason correctly) is intrinsic to every such system, and can only be partially solved. According to them, it is impossible to implement a knowledge-based system that will be both highly expressive and completely tractable.

Moreover, the whole enterprise of encoding knowledge directly into an agent is proven to be useful only in domain-specific applications, such as the famous *block world* [10] domain. A solution with at least partially hard-coded knowledge is tremendously difficult to scale. Since the dawn of Machine Learning, the plausibility of the hypothesis is continuously challenged. Indeed, it is questionable whether we can say about a neural network that it stores knowledge in the weights of its neurons. Thus, today the field of KR does not enjoy that much popularity because the main focus of the AI community has shifted to other areas, such as Deep Learning.

The most prominent advances in the KR field were the development of Description (Terminological) and Temporal logics and the formulation of the concept of Ontology. They all are of great importance to our research, so we will survey them one by one.

2.1.1 Ontologies

The word "ontology", derived from the Greek word meaning "the study of being", was unwarrantably borrowed by computer scientists from the namesake branch of philosophy concerned with the nature of reality. Despite being a useful coinage in informatics, not all philosophers are content with such state of affairs [11].

The term "ontology" in Computer Science refers to the mechanism by which reality is compartmentalized into a strictly-defined categories only to be read by machines later. According to Josephson et al. [12] an ontology comprises a body of knowledge about a particular domain of interest. However, we must distinguish between an abstract conceptualization of a particular domain and a concrete instantiation of it. The latter is usually implied when the plural word "ontologies" is used.

The typical ontology consists of:

1. A finite set of *concepts* (a.k.a. nodes, classes). Represents entities of a domain.

2. A finite set of *properties* (a.k.a. attributes, slots, roles). Represents what can be asked of a concept.

3. A finite set of *relationships* between concepts.

4. A finite set of logical *constraints*, which put the boundaries around what can and cannot be stored in an ontology.

At the first sight, the description of an ontology highly resembles that of a database. Indeed, it is true that every database can be seen as a special case of an ontology, but not vise-versa. As noted in [9], the power of ontologies lies not in what can be said in them, but exactly *what can be left unsaid*. For example, suppose we want to store the birth date of our grandfather, but all we know about him is that he won a medal fighting in WWII. Then we are forced, using a database, to left the `birth_date` field empty, thus loosing the knowledge of his

heroic deed. But, we can eloquently express this knowledge in some ontological lisp-like language as:

```
(set birth_date (father (father me)) (during WWI))
```

Later we can use this fact to reason about our ancestor more efficiently.

Ontologies find their natural application in the context of our thesis. Since the original formulation of a concept, a lot of software has been developed to manage ontologies, including such systems as Protege, In4j and others. These systems have already been heavily used in the variety of different fields.

For instance, Tan Mee Ting [13] designed and implemented a genealogical ontology using Protege and evaluated its consistency with Pellet, HermiT and FACT++ reasoners. He showed that it is possible to construct a family ontology using *Semantic Web* [14] technologies with full capability of exchanging family history among all interested parties. Hovewer, he did not adress the issue of navigating the family tree using kinship terms.

An ontology can be used to model any kind of family tree, but the problem arises when a user wants to query his relatives using kinship terms. No standard out-of-the-box ontology query language is able to articulate statements such as in our example above. Although an ontology can be tailored to do so, it is not in any way a trivial matter. Maarten Marx [15] addressed this issue, but in the different area. He designed an extension for XPath, the first order node-selecting language for XML.

Catherine Lai and Steven Bird [16] described the domain of linguistic trees and discussed the expressive requirements for a query language. Then they presented a language that can express a wide range of queries over these trees, and showed that the language is first order complete. This language is also an extension of XPath.

Artale et al. [17] did a comprehensive survey of various temporal knowledge representation formalisms. In particular, they analysed ontology and query languages based on the linear temporal logic LTL, the multi-dimensional Halpern-Shoham interval temporal logic, as well as the metric temporal logic MTL. They

note that the W3C standard ontology languages, OWL 2 QL and OWL 2 EL, are designed to represent knowledge over a static domain, and are not well-suited for temporal data.

Modelling kinship with mathematics and programming languages, such as LISP, has been an extensive area of research. Many people committed a lot of work into the field, including Bartlema and Winkelbauer, who investigated [18] the structure of a traditional family and wrote a simple program that assigns fathers to children. Their main purpose was to understand how this structure affects fertility, mortality and nuptiality rates. Although promising, small step has been made towards designing a language to reason about kinship. Also, their program cannot express temporal information.

Another prominent attempt in modelling kinship with LISP was made [19] by Nicholas Findler, who examined various kinship structures that exist in literature and combined them together to create a LISP program that can perform arbitrary complex kinship queries. Although his solution is culture-independent, he did not take the full advantage of LISP as a programming language, and because of that it is impossible to express queries which are not about family interrelations. In contrast, our system does not suffer from that restriction.

More abstract, algebraic approach was taken [7] by D. W. Read, who analysed the terminology of American Kinship in terms of its' mathematical properties. Specifically, he invented an algebraic system, a semigroup, isomorphic to the one particular type of kinship: American Kinship, which refers to the terminology used by English speakers. His algebra clearly demonstrates that a system of kin terms obeys strict rules which can be successfully ascertained by formal methods.

Periclev et al. developed [20] a LISP program called `KINSHIP` that produces the guaranteed-simplest analyses, employing a minimum number of features and components in kin term definitions, as well as two further preference constraints that they propose in their paper, which reduce the number of multiple componential models arising from alternative simplest kin term definitions conforming to one feature set. The program is used to study the morphological and phono-

logical properties of kin terms in English and Bolgarian languages.

According to many authors [7] [20] there have been many attempts to create adequate models of kinship based on mathematics, but these models are either lacking management of temporal information or use only a limited subset of their programming language. This is not the case with our system.

2.1.2 Temporal and Description Logics

The usage of logic in the field of KR is motivated by its excellence in such areas as mathematics and computer science in general. The early researches in KR saw the unharvested power of logic – especially first-order logic – as a main component in any intelligent system. Subsequent works showed that FOL can provide semantics for specific kind of KR structures: *frames* [21]. Later, Brachman and Levesque proved [9] that we do not need the *whole* FOL for that purpose, but only certain fragments of it. Moreover, different fragments of FOL have different expressive power and tractability. Thus, a research began under the label *terminological systems*, only to be later renamed to *Description Logic* when the main focus was shifted to the properties of underlying logical systems.

Description Logic finds its application in the context of this thesis as a natural formalism for family trees. However, as expressive as any DL can be, formulating the concept of time requires adding another modal operator. Any logic which handles time is knows as *temporal logic*. Philosophers have tried to put time into a coherent framework since Aristotle, in the 20th century mathematicians and computer scientists proposed various formalisms, among which was Allens' *interval algebra* [22] and the temporal logic of Shoham [23]. Thus, what we need in this thesis is an amalgamation of a description and temporal logic.

The first successful attempt at integrating two logics is due to Schmiedel [24]. He combined the DL in the tradition of KL-ONE [25], Shohams' [23] temporal logic and Allens' [22] algebra into one unifying framework. The main features of his formalism are the complete preservation of original DL and the use of

lisp-like syntax for expressing roles, concepts and time entires.

The application of temporal logic to graphs, relational databases and ontologies is also a heavily-invested subject. Barcelo and Lubkin examined [26] several temporal logics over unranked trees and characterized commonly used fragments of first-order (FO) and monadic second-order logic (MSO) for them. They also considered MSO sibling-invariant queries, that can use the sibling ordering but do not depend on the particular one used, and captured them by a variant of the μ-calculus with modulo quantifiers.

Alexander Tuzhilin and James Clifford defined [27] a temporal algebra that is applicable to any temporal relational data model supporting discrete linear bounded time. This algebra has the five basic relational algebra operators extended to the temporal domain and an operator of linear recursion. They showed that this algebra has the expressive power of a safe temporal calculus based on the predicate temporal logic with the "until" and "since" temporal operators.

Perry in his dissertation [28] highlighted that even in state-of-the-art ontological query languages, such as OWL, expressing the concept of time is an arduous task. He augmented the *Resource Description Framework* with temporal RDF graphs and extended the W3C-recommended SPARQL query language to support these new structures.

An adequate representation of time is the holy grail among researchers in the field of ontology development. Baratis et al. [29] designed and implemented *TOQL*: a high-level SQL-like language which is capable of expressing temporal queries. They motivate the need for such a language by noting that conveying the concept of time using classical languages, such as OWL, is proven to be difficult, although feasible. They also developed an application that supports translation and execution of TOQL queries on temporal ontologies combined with a reasoning mechanism based on event calculus.

2.2 Natural Language Processing

The main goal of Natural Language Processing (NLP) field is to invent, study and implement algorithms and techniques that help a computer understand an ordinary language, such as English, Russian, French or Swahili.

A lot of research in NLP is dedicated to the problem of querying a relational database in some natural language. Since the early developments, a substantial progress has been achieved. For instance, Jeremy Ferrero et al. proposed and implemented [30] a solution to query any database, irrespective of its' schema, in virtually any natural language. They showed that it supports more operations that most of the other translators. They tested their program on English and French languages.

Another similar attempt was made [31] by Norouzifard et al. They implemented an expert system using Prolog to transform a sentence in a natural language to SQL. Chaudhari [32] presented a light weight technique of converting a natural language statement into equivalent SQL statement.

Nelken et al. took [33] a step further and presented a novel controlled NL interface to *temporal databases*, based on translating NL questions into *SQL/Temporal*, a temporal database query language. They noted that their translation method is considerably simpler than previous attempts in this direction.

2.3 Conclusions

In this literature review we surveyed several major field in Computer Science. We showed that each of these fields has been advanced considerably over the last half-century, especially the domain of ontologies. However, we did not find a research that would satisfy all of the following criteria:

1. Employ either a temporal ontology or a temporal database to store knowledge of temporal family relations.

2. Propose a solution for effective navigation in a genealogical tree via kinship terms.

3. Design and implement a text parser for querying temporal ontologies in natural language.

Although there were articles that partially fulfill some of these requirements, none of them satisfied all. Moreover, since our solution is specifically tailored to work only on family trees, we can conclude that its' performance is better than any other general one. This entails novelty of our work.

Chapter 3

Methodology

3.1 General Considerations

The study of kin structures has its roots in the field of anthropology. Among the first foundational works was Henry Morgans' *magnum opus* "Systems of Consanguity and Affinity of the Human Family" [34], in which he argues that all human societies share a basic set of principles for social organization along kinship[1] lines, based on the principles of **consanguinity** (kinship by blood) and **affinity** (kinship by marriage). At the same time, he presented a sophisticated schema of social evolution based upon the relationship terms, the categories of kinship, used by peoples around the world. Through his analysis of kinship terms, Morgan discerned that the structure of the family and social institutions develop and change according to a specific sequence. He was the first to recognize and record six kin structures that are present in numerous societies and cultures around the world.

Bearing in mind the vast variety of possible options, we are going to settle on just one specific kin structure, which we shall call *traditional kinship*. We will devote the entire chapter to describing and modelling this structure.

Following Henry Morgan, we recognize two primary types of family bonds:

[1] Recall that in this thesis, the word "kinship" includes relatives as well as in-laws

marital (affinity) and parental (consanguinity). These bonds define nine basic kin terms: *father, mother, son, daughter, husband, wife, parent, child and spouse*. Observe that combining them in different ways will yield all possible kinship terms that can and do exist. For instance, *cousin* is *a child of a child of a parent of a parent* of a particular person. Another example: *mother-in-law* is just *a mother of a spouse*.

Let us introduce several useful definitions:

Definition 1. We call a kinship term **abstract** iff it can refer to relatives of different sex. For instance, the word "parent" is an abstract kinship term, because it refers to a mother as well as to a father. Other well known examples: *cousin, spouse, sibling* and *child*.

Definition 2. In contrast, a **concrete** kin term refers only to relatives of the same sex, e.g. *brother, aunt* and *nephew*.

Definition 3. A **dyadic** kin term express the relationship between individuals as they relate to one another symmetrically, e.g. if I am your cousin (sibling), then you are also my cousin (sibling). The few, and uncommon, English dyadic terms involve in-laws: *co-mothers-in-law, co-fathers-in-law, co-brothers-in-law, co-sisters-in-law, co-grandmothers*, and *co-grandfathers*.

Definition 4. An **ego** is a focal point of a genealogy, i.e. it is a person from whose point of view we will describe all other people using kin terms.

Now Let us represent a traditional christian family tree as a special type of *ontology* with its' own concepts, attributes, relations and constraints. Concepts are people in a family, their attributes are: *name, birth date, birthplace, sex* and relations are parental and marital bonds with a wedding date.

Together with the everything stated above, we have the following cultural constraints imposed on our genealogy:

1. Each person can have any finite number of children.

2. Each person can have at most two parents of different sex.

3. Each person can have at most one spouse of different sex.

4. A spouse cannot be a *direct relative*, i.e. a sibling or a parent. In other words, direct incest is prohibited.

When considering those prerequisites one should bear in mind that we deliberately focused only on rules, taboos and customs of one particular culture, namely American culture in the sense of Read [7]. Under different assumptions and in further studies, these conditions can be relaxed and revisited.

Apart from these four, here are two additional temporal constraints that express the interrelation between birth and wedding dates:

1. No one can marry a person before he or she was born, i.e. a wedding date can only be strictly after a birth date of each spouse.

2. A parent is born strictly before all of his (her) children.

Due to the general nature of these two constraints, they are always true in every genealogy and therefore can be safely assumed in our work.

Every genealogy that meets these six requirements we shall call a **traditional family tree**. As the name "tree" suggests, we can indeed view this structure as a graph with its vertices as people and edges as bonds. Observe that every kinship term corresponds exactly to a *path* between ego and specified relative. Under such a view, kin term becomes a set of instructions, telling how to get from the starting vertex `A` to the end vertex `B`. For example, consider the term *mother-in-law*. What is it if not precisely a *directive*: "firstly, go to my spouse, then proceed to her mother". The wonderful thing is that, due to the nature of kinship terminology, we can *compose* them together to create new terms, even those which do not have their own name. This simple observation that we can see kinship terms as paths in a family tree underpins our entire thesis.

Now, if we want to efficiently query a traditional family tree, we need to further investigate the mathematical features of the language of kinship terms. In the next sections we explore a formal model of kinship language, its syntax

and semantics. Then we conclude this chapter with an examination of various approaches of tackling the concept of time. We also discuss possible augmentations to our model and the problem of *term reduction*.

3.2 Formal Language of Kinship

Here we present our attempt to model the language of traditional American, in the sense of Read [7], kinship terminology. There are three main characteristics that define every formal language: its syntax (spelling, how words are formed), semantics (what does particular word mean) and pragmatics (how a language is used). In order to describe it we must expound the first two.

3.2.1 Syntax

We use Backus-Naur Form to designate the syntax for our formal language. Let Σ be the set of six basic kinship terms: *father, mother, son, daughter, husband, wife*. Then we can express the grammar as follows:

$$term ::= \Sigma \mid (term \cdot term) \mid (term \vee term) \mid (term)^{-1} \mid (term)^{\dagger}$$

The first operation is called *concatenation*, second – *fork*, third – *inverse* and the last – *dual*. We denote this language by $\mathcal{L}$.

Here are some examples of ordinary kinship terms expressed in our new language. Note that we deliberately omit superfluous parentheses and the composition sign for the sake of simplicity:

- Parent is $father \vee mother$.
- Child is $son \vee daughter$.
- Brother is $son(father \vee mother)$.
- Sibling is $(son \vee daughter)(father \vee mother)$.
- Uncle is $son(father \vee mother)(father \vee mother)$.

- Daughter-in-law is $daughter \cdot husband$
- Co-mother-in-law is $mother(husband \vee wife)(son \vee daughter)$

From these examples you can see the real power of this language – the power to express all possible used as well as not used kinship terms that can and do exist. Now the important step towards solving our main goal, developing a language for managing temporal genealogies, is to assign meaning to these words. From now on we distinguish between *artificial* kinship terms, i.e. well-formed terms of our formalization, and *natural* kin terms used in ordinary English. By referring to just terms, we mean the former, if nothing else is stated.

3.2.2 Semantics

Let Σ^* stand for the set of all possible kin terms generated from the basis Σ using the previously defined syntax. Let $\mathcal{G} = (V, E)$ be a traditional family tree with V as a set of its vertices (people) and E as a set of its edges (bonds). Moreover, because $\mathcal{G}$ is traditional, every person from the set V have the following attributes:

- A father. We will denote him as $father(p)$, a function that returns a *set* containing at most one element.
- A mother. We will denote her by $mother(p)$.
- A set of his or her children: $children(p)$.
- A set of his or her sons:

$$son(p) = \{c \mid c \in children(p) \wedge Male(p)\}$$

- A set of his or her daughters:

$$daughter(p) = \{c \mid c \in children(p) \wedge Female(p)\}$$

- A spouse: $spouse(p)$.

- A husband:

$$husband(p) = \{s \mid spouse(s) \wedge Male(p)\}$$

- A wife:

$$wife(p) = \{s \mid spouse(s) \wedge Female(p)\}$$

Due to the constraints stated in 3.1, result-set of *father*, *mother*, *spouse*, *husband* and *wife* can contain at most one element.

Now we are ready to introduce **Denotational Semantics** for Σ^*. This name was chosen because it highly resembles namesake semantics of programming languages. Note that we regard kinship terms as *functions* on subsets of V. Each function takes and returns a specific subset of all relatives, so its type is $f : \mathcal{P}(V) \to \mathcal{P}(V)$.

We proceed by induction on the syntactic structure of $\mathcal{L}$. Let t be an element of Σ^*, then:

1. If $t \in \Sigma$, then $[\![t]\!] = F(t)$, where $F(t)$ assigns to each basic kin term its corresponding function from the list 3.2.2.

2. Term concatenation is a composition of two functions:

$$[\![(t_1 \cdot t_2)]\!] = [\![t_1]\!] \circ [\![t_2]\!]$$

3. Fork is a set-theoretic union of results of its sub-functions:

$$[\![(t_1 \vee t_2)]\!] = p \mapsto [\![t_1]\!](p) \cup [\![t_2]\!](p)$$

4. Term inverse is exactly the inverse of its function:

$$[\![t_1^{-1}]\!] = [\![t_1]\!]^{-1}$$

The *dual* operator (†) is more difficult to define. We want it to mean exactly the same as the term, where the gender of each its basic sub-term is reversed, e.g. dual of "uncle" is "aunt", dual of "brother" is "sister" and so on. Here we can use induction once again:

1. If $t \in \Sigma$, then $[\![t]\!] = D(t)$, where $D(t)$ is a basic term of opposite sex.

2. Dual is distributive over concatenation, i.e. dual of concatenation is a concatenation of duals:

$$[\![(t_1 \cdot t_2)^\dagger]\!] = [\![(t_1^\dagger \cdot t_1^\dagger)]\!]$$

3. Dual is distributive over forking:

$$[\![(t_1 \vee t_2)^\dagger]\!] = [\![(t_1^\dagger \vee t_2^\dagger)]\!]$$

4. Inverse commutes with dual:

$$[\![(t^{-1})^\dagger]\!] = [\![(t^\dagger)^{-1}]\!]$$

Observe that we also have the distributivity of concatenation over forking. This semantics allows us to efficiently navigate any family tree.

3.3 Term Reduction

Our artificial language has a problem: its too verbose. Indeed, to encode such ubiquitous kin terms as "uncle" or "great-nephew" one must use quite lengthy phrases that are hard to write and read. It is therefore important to have some sort of reduction mechanism for our language that will shorten long terms into a small set of common kinship relations to aid their understanding by a user.

Firstly, let us analyse the problem. We have the following mapping ω be-

tween Σ^* and the set of English kinship terms $\mathcal{W}$

$$
\begin{aligned}
\mathit{son}(\mathit{father} \vee \mathit{mother}) &\mapsto \text{brother} \\
\mathit{daughter}(\mathit{father} \vee \mathit{mother}) &\mapsto \text{sister} \\
\mathit{father}(\mathit{father} \vee \mathit{mother}) &\mapsto \text{grandfather} \\
\mathit{mother}(\mathit{father} \vee \mathit{mother}) &\mapsto \text{grandmother} \\
\mathit{son}(\mathit{son} \vee \mathit{daughter}) &\mapsto \text{grandson} \\
&\vdots \\
\mathit{father}(\mathit{son} \vee \mathit{daughter})(\mathit{wife} \vee \mathit{husband})(\mathit{son} \vee \mathit{daughter}) &\mapsto \text{co-father-in-law}
\end{aligned}
$$

This dictionary allows us to effectively translate between kin terms of our artificial language $\mathcal{L}$ and their usual English equivalents. We can also view this mapping as a *regular grammar* in the sense of Chomsky hierarchy [35]. However, note that we strictly prohibit mixing these two collections and therefore we deliberately avoid using words from the RHS in the LHS, because otherwise the grammar will lose its regularity and become *at least* context-free, making the problem even more challenging. Let us define another function on top of ω that will replace the first sub-term $u \subset t$ in a term $t \in \Sigma^*$:

$$\Omega_u(t) = t[u/\omega(u)]$$

Here the only change in meaning of $t[u/\omega(u)]$ is that the substitution takes place only once.

Now the task can be stated thusly: given a term $t \in \Sigma^*$ find its shortest (in terms of the number of concatenations) translation under Ω, i.e. which sub-terms need to be replaced and in what order.

This problem can be easily reduced to that of of finding the desired point in the tree of all possible substitutions. Moreover, this point is actually a *leaf*, because otherwise it is not the shortest one. But the latter can be solved by just searching for this leaf in-depth. Unfortunately, the search space grows

exponentially with the number of entries in the dictionary ω, thus making the naive brute-force approach unfeasible.

Here we propose a heuristic greedy algorithm A.1 that, although does not work for all cases, provides an expedient solution to the reduction problem in $O(n^2)$ time. Firstly, it finds the longest sub-term u that exists in the dictionary ω, then divides the term into two parts: left and right from u, after that it applies itself recursively to them, and finally it concatenates all three sub-terms together.

Now let's analyse the time complexity of this algorithm:

Theorem 1. *The execution time of the algorithm listed in A.1 belongs to* $\Theta(n^2)$.

Proof. Let $T(n)$ be the execution time of the algorithm, where n stands for the number of concatenations in a kin term. First of all, observe that $T(n)$ obeys the following recurrence:

$$T(n) = 2\,T(n/2) + O(n^2) \tag{3.1}$$

Indeed, we make a recursive call exactly *two* times and each call receives roughly the half of the specified term. During execution the function passes through two nested cycles, so one call costs us $O(n^2)$.

Secondly, we use the **Master Method** from the famous book *Introduction to Algorithms* [36] by Cormen et al. In our case $a = 2$, $b = 2$, and $f(n) = O(n^2)$. Observe, that if we take ϵ to be any positive real number below one: $0 < \epsilon < 1$, then $f(n) = \Omega(n^{\log_b a + \epsilon}) = \Omega(n^{1+\epsilon})$.

Let us show that $f(n)$ satisfies the *regularity* criterion: $af(n/b) \leqslant cf(n)$ for

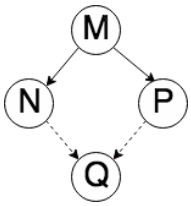

Figure 3.1: Confluence in a term rewriting system.

some constant $c < 1$. Indeed, just pick $c = 1/2$:

$$\begin{aligned} 2f(n/2) &\leqslant cf(n), \\ 2\frac{n^2}{4} &\leqslant cn^2, \\ \frac{1}{2}n^2 &\leqslant cn^2, \\ \frac{1}{2}n^2 &\leqslant \frac{1}{2}n^2 \end{aligned}$$

Thus, we can use the third case from the Master Method, which tells us that $T(n) = \Theta(n^2)$. □

3.3.1 Pursuing Confluence

Current approach listed in A.1 has one major disadvantage: like any other greedy algorithm it can fail to choose a correct reduction path between two terms with equal amount of concatenations. We can alleviate this by augmenting our rewriting system, based on ω dictionary, with a feature called *confluence*, also known as *Church-Rosser* property:

Definition 5. An abstract term rewriting system is said to possess **confluence**, if, when two terms N and P can be yielded from M, then they can be reduced to the single term Q. Figure 3.1 depicts this scenario.

Not only we can fix our reduction algorithm by introducing this property, but also we can improve the time complexity, making it linear.

One way to achieve confluence is to attach a single kinship term to any possible path in a family tree. Observe, that English kinship terms have a

specific pattern that we can exploit. All relatives who are distant enough from ego have the following structure of their kin term:

$$n^{th} \text{ cousin } m^{th} \text{ times removed}$$

In-laws also have their own pattern, where the ending "-in-law" is appended to a valid consanguine kinship term. However, this applies only to people, who are linked together by only one nuptial bond. For instance, there is no single term for a husband of ego's wife's sister. These relations can be accounted for by *prefixing* "-in-law" with an ordinal, which shows the number of marital bonds that one should pass in order to go to such person. Under this representation, last example will receive the term "brother *twice*-in-law". Generalizing that scheme we will get a pattern that looks like the following:

$$\langle \text{Consanguine kinship term} \rangle k^{th} \text{ times-in-law}$$

We can also view this as an attribution of a distinct *natural number* to every vertex with ego as an *offset*, thus imposing a natural ordering on the set of all vertices. This assignment can be made in such a way that reducing a kinship term n will correspond *exactly* to the calculation of n from some arithmetic expression like $5 \cdot (2+3)+4$, thus providing a **translation** between the language of all valid arithmetic expressions and our formal language of kinship $\mathcal{L}$.

However, it is not the topic of this thesis, so we are leaving it to the considerations of future researches.

3.4 Incorporating Time

Now the only matter that is left to address is an adequate representation of time. Historically, there are two main approaches for modelling time: point-based and interval-based. The former treats time as a single continuous line with distinguished points as specific *events*, and the latter uses *segments* of that

line to represent time entries, which is more famous. For instance, it was used in Allen's interval algebra [22]. For the sake of simplicity we chose the former approach, because it can easily imitate intervals by treating them as endpoints of a line segment.

Not only we want to talk about different events by modelling them as points on a line, but also we want to orient ourselves on that line, i.e. to know where we are, which events took place in the past and which will happen in the future. Thus, we need to select exactly one point that will stand for the present moment and call it "now". Then all point to the left will be in the past, and all point to the right will be in the future. Also, notice that any set with total ordering on it will suffice, because the continuous nature of a line is redundant in point-based model. Collecting everything together, we have the following formalisation of time:

$$\mathcal{M} = \langle T, now, \leqslant \rangle$$

Where T is a non-empty set with arbitrary elements, $now \in T$, and $\leqslant$ is a total ordering relation on T.

Within this model we can reason about which event comes *before* or *after*, what events took place in the past or in the future, and so on.

When considering family trees it is necessary to define only five predicates:

1. $Before(x, y)$ is true iff $x < y$.
2. $After(x, y)$ is true iff $x > y$.
3. $During(x, s, f)$ is true iff $s \leqslant x \leqslant f$.
4. $Past(x)$ is true iff $Before(x, now)$.
5. $Future(x)$ is true iff $After(x, now)$.

Those relations are the basis from which all other operations on $\mathcal{M}$ can be defined. It is also interesting to note that, since any ordering relation generates

a *topology* over its structure, we can speak about time in terms of its topological properties.

This observation concludes our methodology chapter.

Chapter 4

Implementation

This chapter is dedicated to report the implementation of our genealogy management program. Firstly, we present a specification for our new programming language named **KISP**: **K**inship LI**SP**. Secondly, we describe the overall structure of the program, including its three major components: Database Manager, Virtual Assistant and KISP Interpreter.

4.1 KISP Language Specification

4.1.1 Grammar and Lexical Structure

Since KISP is a dialect of LISP, it inherits some syntax from the predecessor, but generally it is a new programming language. The grammar is presented with the help of Bacus-Naur notational technique. For the sake of simplicity we omit angle brackets and embolden non-terminal words. A plus, a star sign in a superscript and a question mark have the same meaning as in regular

expressions.

term ::= **literal** | **lambda** | **define** | **atom** | (**term**$^+$)

lambda ::= (lambda (**reference***) **term**)

define ::= (define **reference term**)

reference ::= **word**{-**word**}'?'?

word ::= **letter**$^+$

letter ::= a | b | ... | z | A | B | ... | Z

atom ::= $*$ | $+$ | concat | list | append | ...

literal ::= void | true | false | people | vacant | now | **numeral** | **string**

numeral ::= -?**digit***

digit ::= 0 | 1 | ... | 9

string ::= ‘**symbol***‘

symbol ::= *any non-blank ASCII symbol*

As we can see from the definition, there are three kinds of terminals in the grammar: literals, references and atom functions, which are called simply **atom**s. Literals are instances of primitive types, such as *Numeral*, *String* or *Boolean*, or special keywords. They stands for the following: "void" represents NULL type, "people" – a list of all persons in a family tree, "now" – the current time entry and "vacant" – an empty list. References are used as definientia in "define" terms and as names for parameters in lambda terms. It is possible for a reference to end in a question mark, which means that it denotes an instance of *Boolean* type. References can we written in so-called *dash case*, so "long-name" and "very-long-name" are both legal. The only exception are names which start with the dash like "-illegal", they are invalid.

Note that we allow niladic lambdas, so, for instance, this is a valid expression: `(lambda () 'Hello, World!')`. But at the same time `()` is not a well-formed term. We also prohibit "define" terms inside other terms, so this would

not work: `(+ 2 (define three 3))`. Strings are nested in single quotes, integers, in KISP we call them "numerals", can start with a zero and be prefixed by a negative sign.

Here is the complete list of all keywords in KISP: **true**, **false**, **define**, **lambda**, **people**, **now**, **void**, **if**, **vacant**. The rule is that you can use as a reference everything you want as long as it is not a keyword, so you cannot redefine their standard behaviour, thus a programmer is unable to tamper with inner workings of the interpreter.

As in all other dialects of LISP, a term `(f a b c ...)` means the *execution* of a function f with the specified arguments $f(a, b, c, ...)$. Of course, we can construct and call the higher-order functions as usual: `((twice square) 2)` will yield 16, or `((compose inc inc) 0)` which prints 2.

Recursion also works as in any other LISP-like language. For example, here is the function that computes the n-th term of the famous Fibonacci sequence:

```
(define fibonacci
    (lambda (n)
        (if (< n 2)
            1
            (+ (fibonacci (- n 1)) (fibonacci (- n 2)))
        )
    )
)
```

However, using a reference before its definition is prohibited. For instance, the following code would not execute:

```
(+ 1 two)
(define two 2)
```

The interpreted will fail to recognize "two" as a valid reference in the first term

and therefore will print an error message. The same applies to references in "define" term: cyclic definitions are forbidden:

```
(define zero (prev one))
(define one (succ zero))
```

4.1.2 Data Types

There are eight data types in KISP, four primitive and four composite. Primitive types: *Boolean*, *Numeral*, *Void* and *String*. Composite: *Function*, *List*, *Person* and *Date*. An instance of a primitive type can be obtained by a literal, whereas a composite object is created only through calling a corresponding *constructor* function. For instance, to make a list of three numbers one should use the following term: `(list 0 1 2)`. However, there are three pre-defined global constants of composite types: *now*, *vacant* and *people*.

KISP is a *dynamically* typed program language, which meas that information about a type of an entity is present only at execution time. This approach has certain benefits over *static* typing as well as some disadvantages.

For each type there are corresponding atomic functions provided by default to work with that type. Here we will present the atomic functions only for the type *Person*, since it is the main element that distinguish KISP from other dialects. For the full documentation of other functions one can refer to the Appendix CC.

In order to create an instance of the type *Person*, one should write `(person 'full name')`. This function will look up and return a person with stated full name in a linked family tree, or "void" if there is no person with such name. This function also has the following form: `(person 'first name' 'last name')`. All arguments are case-insensitive.

The function `kinship` provides a convenient way to ascertain how a person p_1 is related to the person p_2: `(kinship` p_1 p_2`)`. It returns a *list* of basic kin terms as strings. The content of such list corresponds exactly to a composite

kinship term describing the relation of p_1 and p_2 to each other. For example, let us assume that there are two persons in our family tree: Alice and Bob. Alice is a niece of Bob, so if we to evaluate the term `(kinship (person 'alice') (person 'bob'))` we would get a term which is equal to `(list 'daughter' 'parent' 'son')`. The result can also be empty, but only if two persons are not related at all, i.e. the family tree has multiple connected components.

In order to reduce a long kinship term generated by the latter method, one can use *shorten* function that accepts the list of basic kin terms and performs the reduction algorithm outlined in the listing A.1. Using the previous example, running `(shorten (kinship (person 'alice') (person 'bob')))` will yield to the list of exactly one element: "niece". This function can be of use when printing the result of *kinship*.

Another useful function which return a specified attribute of a person is *attr*. Here is how we can get a persons birthday: `(attr person 'birthday')`. This term produces the date of birth of a `person` as an object of type *Date*. The list of all attributes accepted by *attr* can be found in the full documentation C.0.36.

Of course, there are unitary functions corresponding to the denotations of basic kin terms: *mother*, *father*, *spouse* and *children*. Here is how we can get a `persons` paternal grand-mother: `(mother (father person))`. All those functions return a list of respective relatives/in-laws. This list can be empty iff there is no such relative present in a family tree.

Another interesting function is *gen-dist*, which calculates the *generational distance* between two persons. The distance d is defined as the number of the level of the second person *relative* to the first, where the level of the second person is considered to be zero:

- The distance is a symmetric function: $d(x, y) = d(y, x)$, for all x and y.
- It is zero for every two individuals from the same generation, including the person himself: $d(x, x) = 0$.

- Father located on the previous level, so $d(x, father(x)) = -1$.
- On the other hand, child inhabits the next level, thus $d(child(x), x) = 1$, and so on.

For d the calculation process is to start with zero, then increment each time we descend one level and decrement every time we ascend a level.

The last function in the repertoire of the *Person* type is *put-kinship*. It is used to append a new entry into the ω dictionary, thus enhancing the capabilities of *shorten*. For example, with `(put-kinship 'son,parent,parent' 'uncle')` we can reduce avuncular terms. To correctly specify the formal kinship term, one must follow strict guidelines:

1. Use only nine basic kinship terms.
2. Separate them with a single comma without any in-between spaces.

4.2 System Structure

Firstly, we shall determine the user requirements for our program. Since there was no individual customer, who would say what features are indispensable and what are superfluous, we had to play this role ourselves. That eliminated the potential problems which one can encounter when trying to understand the needs of the user. Also, by putting ourselves into the situation of a potential customer, who often has rather trivial knowledge of IT, we gained an understanding of his point of view, which will be particularly helpful to us later.

After careful considerations, we arrived at the following list of requirements:

1. A user must be able to see his carefully crafted family tree in a nice visualisation.
2. A user wants to move and scale this visualisation.
3. A user must have an ability to move the nodes of his family tree.
4. A user wants to add and remove members of his family.

5. A user must be able to link his relatives together with nuptial and parental bonds.

6. A user must be able to edit the profile of a person in his genealogy.

7. A user must have an ability to manage many genealogies without losing his work.

8. A user wants to ask and to get answers to meaningful questions about his family tree.

These eight items can be compartmentalized into three distinct categories: *genealogy managing*, *query answering* and *family tree visualisation*. They form the basis for the components of our program. Observe, that if we apply a software design pattern known as *Model-View-Controller*, we can merge the first and the third category into a single component, we will call it *Genealogy Manager*, which will be responsible for displaying, handling and working with multiple genealogies, thus successfully covering all but one requirement from the list.

Now, let's focus on the last requirement. Here the situation is different: in order to tackle it we need not to merge, but to split the second category, because of its challenging nature. If a user wants to ask questions, he firstly needs to *formulate* them in some language that a machine can understand, which means that this language will be virtually inaccessible to a non-specialist. Therefore, it is of utmost importance to assist a user in expressing his questions. For that we propose the following solution: create a simple bot, whose set of English questions that it can answer will be rich enough to satisfy a users need for inquiries. Let's call such bot a *Virtual Assistant*. Since we already have a formal language, we only need to develop its interpreter. Thus, we have two components that will cover the eighth requirement: a *KISP Interpreter* and a Virtual Assistant.

The figure B.2 depicts the general structure of our project. Notice how components are linked to each other and how some of them are encapsulated in others. From B.2 we can see that a user can interact only with *Assistant*

and *Genealogy View* components, everything else is hidden from his sight. This is generally acknowledged to be the best approach to design user interfaces – reducing degrees of freedom – not only in software, but in all areas of technology.

In the following sections we present the structure of the three software components that we identified.

4.2.1 Genealogy Manager

As we can see from the figure B.2, Genealogy Manager consists of three modules: Visualiser, Model and Database. The purpose of the first is to show a user nice and interactive view of his family tree. The second manages kinship records stored in DB and provides its interface for the rest of the application.

Let's have a look at the third module: the Database, which stores the ontological *axioms* of a family tree, that is, all persons and their connections to each other in the form of table records. We chose a relational database for that purpose and not our own solution mainly because the current ones are much more efficient and because it was too costly to enrich KISP with such features. We selected SQLite as the RDBMS engine for our application, since it is compact and effective at the same time. The Entity-Relation diagram has been

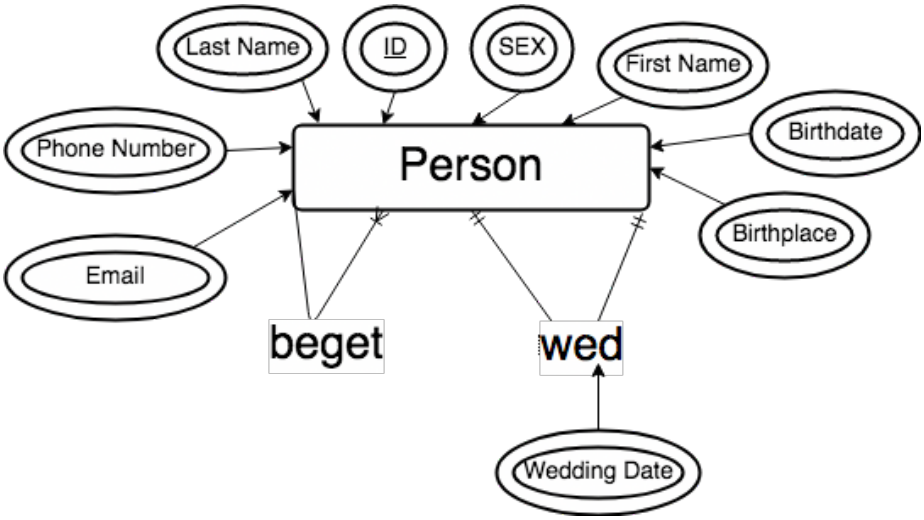

Figure 4.1: Entity-Relation Database Diagram

converted to tables in the following way:

1. The main entity *Person* is presented as a table incorporating all information about people in a family tree. It has ID field as a primary key.

2. The $1-1$ relation "wed" encodes the marital bonds between people, and $1-N$ relation "beget" encodes the parental bonds. They both are represented as separate tables, which contain the edges of a family tree. The former is also made to be symmetrical, i.e. $wed(x, y) \Leftrightarrow wed(y, x)$, which allows for more coherent reasoning about genealogy. Symmetry means that each record in the table 'wed' is doubled and reversed after being inserted.

Each row is loaded from DB and stored in the next sub-component, *Model*, as a graph data structure. The model defines all the necessary methods to work with this graph, including *Breadth-First Search* algorithm that is used to find to minimal path between two vertices.

The *Visualiser* then renders this tree inside a predefined rectangular region, which can be further scaled and moved by the special camera class. This class implements a lightweight version of a 2D graphical engine.

4.2.2 Virtual Assistant

After careful considerations, we settled on a list of English questions that our virtual assistant, named *Ami*, can answer. This list is rich enough to satisfy any user and, at the same time, short enough to allow an actual implementation. Observe, that the list actually defines a *infinite* amount of potential questions by including special `<reference>` part, which we define by again using convenient Bacus-Naur notation:

reference ::= **name** | my **relative** | my **relative**s' **relative** | **relative** of **reference** | I | me | myself

relative ::= **basic** | uncle | brother | ...

basic ::= parents | parent | son | ...

Where **relative** stands for an *image* of the dictionary ω, **name** for the full name of a person and **basic** for all primal kinship terms.

1. How is <reference> related to <reference>?
2. How am I related to <reference>?
3. Who is <reference>?
4. Who is (a or an) <relative> of <reference>?
5. Where <reference> (was or were) born?

As we can see, questions in this list have a coherent and static structure that can be parsed by machine, thus allowing us to take the most straightforward approach and implement them directly hard-coded string constants.

Besides questions, Ami also accepts KISP queries that it channels to the Interpreter. For example, in order to query all females in genealogy one need to send the following message to Ami:

```
(filter (lambda (p) (= 'FEMALE' (attr p 'sex'))) people)
```

Due to the incredibly versatile nature of our programming language, we can confidently say that there is no ineffable queries. Therefore, a user can always express himself at least in KISP, if not in English.

Secondly, Ami can perform various auxiliary commands, such as:

1. 'Show [me] <reference>.' This command will center the view of a family tree on the specific person that is mentioned in the <reference>. If there are many persons, Ami will choose the first one. It is useful when we want to quickly locate a particular relative in a big tree.
2. 'Load filename.lisp', this command reads the specified file and executes its content as KISP source code. It may be of use when we need to quickly introduce new functions and definitions.

Finally, we must note that Ami is a *context-free* assistant, which means that it takes into account only the very last message of the user. However, there is an important exception to this rule: if a user sends a message for the first time referring to himself, Ami will ask who he is, because there is no such information available by default. Context awareness of the virtual assistant is an interesting topic for future research, as well as considering different channels of user input, such as voice recognition.

4.2.3 KISP Interpreter

For the implementation of the KISP language we chose the most straightforward approach: firstly parse the source code, secondly, construct an AST, then perform an evaluation of that tree starting from its leaves, and finally output the result of the evaluation. However, in order to expedite this process, we introduced a caching mechanism, which remembers what terms yields to what results. Experienced user can manipulate this mechanism by the following commands:

1. `Flush` completely removes all entries from cache table.

2. `Set cache` enables caching mechanism.

3. `Set nocache` disables caching mechanism.

Cache table automatically updates each time new definition is introduced or an old reference is redefined.

For example, lets evaluate the following terms:

```
(define ego (person John Smith))
(= ego ego)
(define ego (person 'John Doe))
(= Doe (attr ego last name))
```

The last term will be correctly evaluated to `true`, since the result of the term `ego` was automatically updated in the cache table.

This concludes the implementation chapter.

4.3 Query Examples

In this section we will demonstrate how one can use KISP to perform various queries in a genealogical tree. Particularly, we focus our attention on statements that express kinship terms.

Let's start with a simple task of selecting people based on a certain boolean condition. Suppose we want to query only those, who have at least one child. This can be accomplished as follows:

```
(filter (lambda (p) (< 0 (count (children p)))) people)
```

Here we iterate through the list of all people in a tree and take only those, on who defined lambda predicate evaluated to `true`. The number of children for a particular person is calculated by counting elements of the list `(children p)`.

The next task is to select all husbands, that is, all men who are married. This can be done in two ways: either select only males and then discard all bachelors, or combine the two operations together in a single boolean predicate using `and` clause:

```
(filter (lambda (p) (not (= void (spouse p))))
            (filter (lambda (p) (= 'MALE' (attr p 'sex')))
            people))
(filter (lambda (p) (and (= 'MALE' (attr p 'sex'))
                         (not (= vacant (spouse p)))))
    people)
```

The advantage of the second approach is that the list `people` will be iterated only once.

Now to the more advanced queries; suppose that the term `ego` stands for the user's node in an ancestry, and he wants to know how many cousins he has:

```
(define parents (lambda (p) (join (mother p) (father p))))
(define cousins
    (lambda (p) (children (children (parents (parents p))))))
(- (count (cousins ego)) 1)
```

This is where the expressive power of KISP truly comes into play. Although `cousins` is not a standard KISP function, we can easily implement it using kinship framework of KISP, which successfully utilizes the structure of natural kinship terms. Moreover, notice how the function `parents` is expressed. Since a parent is either a mother or a father, it corresponds to the formal kinship term $(mother \vee father)$, which is implemented as a *join* of two or more lists. And because every cousin is a grand-child of one's grandparents, it corresponds to:

$$(son \vee daughter)(son \vee daughter)(mother \vee father)(mother \vee father)$$

The last decrement was made because in this scheme the `ego` itself will be included to the resulting list.

Finally, temporal queries can be expressed with the help of the type *Date*. For instance, if we need to know, who, among our relatives, was born during the WWII, we just need to evaluate:

```
(define WWII-start (date '01.09.1939'))
(define WWII-end (date '02.09.1945'))
(filter (lambda (p) (during (attr p 'birthdate') WWII-start WWII-end))
    people)
```

The type *Date* provides all the necessary functions for working with temporal information.

Because KISP is Turing-complete and inherits LISP's capabilities of meta-programming, one can easily extend it with any functionality that one wants.

Chapter 5

Evaluation and Discussion

In this chapter we are going to present our approach to the testing and evaluation of the three main components: KISP Interpreter, Visualiser and Virtual Assistant. Among many other types of software assessment, we chose the following methods due to their effectiveness and relative simplicity: Benchmarking, Conditional and Unit Testing.

5.1 Conditional Testing

In the field of Computer Science the idea of *design by contract* is well-known: programmer uses pre- or post-conditional statements to control the source code execution. There are many languages with native support of these features, including Eiffel [37], created by Bertrand Meyer. In Java we can use the keyword `assert` to simulate their behaviour. What makes Java assertions special is that we can turn them on and off manually by supplying the `-enableassertions` key to the JVM's argument list, thus removing their computational overhead.

During the coding phase, we wrote 93 assertion statements that mainly control the execution of KISP interpreter. Then we manually tested the application to see whether any of them would fail, but none ever did.

5.2 Unit Testing

This type of software testing is ubiquitous in the industry, especially in the *Test-Driven Development* process, where requirements are turn into test cases, and then the software is improved to pass these tests only. Historically, the idea of TDD was firstly used in the *Extreme Programming* methodology, invented [38] by engineer Kent Beck in 1999.

There are many solutions for the Java language that implement the concept of Unit Testing. Among many others, we decided to use JUnit [39] 5 library, which was also developed by Kent Beck et al., to write 136 test cases that cover 100% of KISP classes, $81,3\%$ of interpreter's methods and $79,7\%$ of its lines. Using this library, we have tested all standard KISP functions and types, only to ascertain that everything works as intended and all tests pass.

5.3 Interpreter Benchmarking

One of our main goals was to achieve an adequate performance of the KISP interpreter. In order to ascertain whether we attained that goal or not, we ran a benchmark test and collected its results. We chose the implementation of the `factorial` function as the benchmark target, and ran it of 40 different inputs.

```
(define factorial
    (lambda (n) (if (= n 0) 1 (* n (factorial (dec n))))))
```

From the results of the first benchmarking session 5.2, we can see how our *caching* mechanism affects the performance of the factorial procedure: if at the beginning the execution time have risen to almost 4 milliseconds, subsequent runs demonstrate significant improvements, ending in being just below a millisecond. In the second session 5.2, we supplied larger values of N, but as we can see, we got a similar picture, only the initial discrepancy is larger this time.

These two benchmarking sessions clearly show that we indeed reached our third goal. The performance of KISP interpreter is withing the expected norms,

thanks to the caching mechanism. However, it can be significantly improved further by introducing *Just-in-Time* compilation.

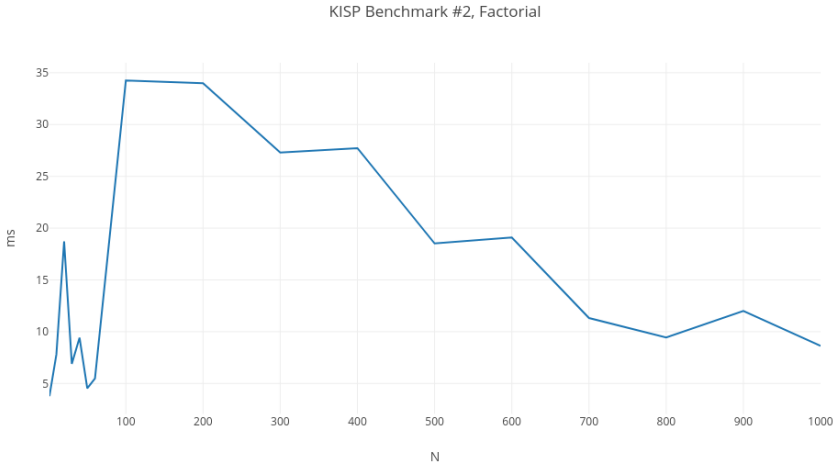

Figure 5.1: Second Benchmarking Session

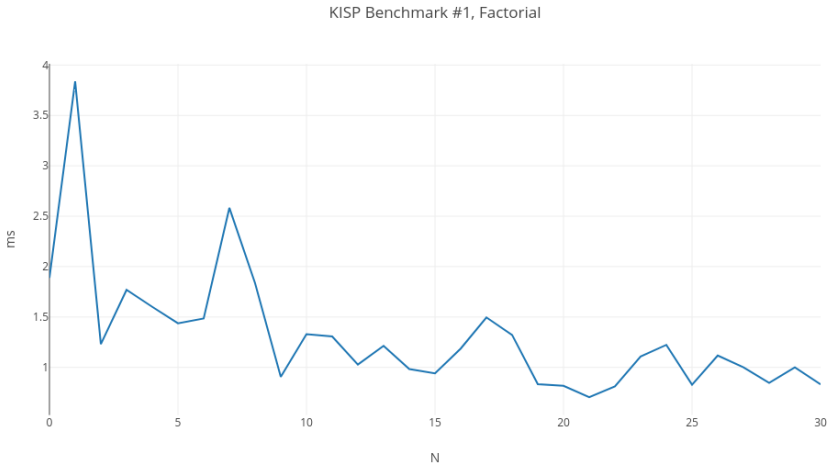

Figure 5.2: First Benchmarking Session

Chapter 6

Conclusion

In this work we solved the problem of efficient navigation in temporal genealogies by designing domain-specific programming language KISP and implementing its interpreter. To further facilitate the use of our program, tree visualiser and virtual assistant have been developed. By the term "efficient" we mean three things:

1. It should have high expressive power.
2. It must be fast enough.
3. It should be as intuitive as possible.

We are certain that our project fully covers every one of them.

From the very start of our history, we gathered, analysed and composed information about our ancestors and relatives. Although, with the ubiquitous use of computers, we can do it more effectively than ever, the present market solutions are found to be inadequate in one way or the other. Particularly, the vast majority of them are either lacking expressiveness or are too complicated and, therefore, require a special training just to be used. In contrast, our project manages to be a user-friendly application, while at the same time having a high level of expressive power.

During the course of our work we faced such challenges as reducing long kinship terms and developing a swift 2D graphical engine, all of which were successfully solved.

6.1 Future Work

However, there are some topics yet left to tackle in the area of kinship and genealogy management. On the theoretical side, there is a problem of total term reduction and formal language enrichment. It is also interesting to shift attention to other languages and cultures with different kinship structures, such as Russian or Hawaiian. The constructed formalism can be considered from the algebraical side, focusing on its many mathematical properties as a special type of an algebraic system.

On the practical side, one can consider to improve the virtual assistant component. Besides already mentioned Voice Generation & Recognition technology, it can be made context-aware, which will increase its intelligence. Additionally, the family ontology can be enhanced to incorporate information about divorces and deaths. The performance of the interpreter can be significantly improved by introducing *Just-in-Time* compilation.

Further improvements may also include new data types and standard functions for KISP language. Specifically, it is beneficial to add a `char` type that represents individual characters in a string. Another useful feature is support for *variadic lambdas* and *closures*, which will significantly increase the versatility of KISP.

Moreover, one can also consider including capabilities for a logical reasoning into KISP. They will be applicable for inferring implicit time constraints for events, whose exact date is unknown. For instance, if we are uninformed about a birthday of a person, but we do know his parents and his children birthdays, we can justifiably bound this missing date to a specific time interval.

Bibliography

[1] L. Kessler. Genealogy software reviews. 2018/May. [Online]. Available: http://www.gensoftreviews.com/

[2] I. My Heritage. Family tree builder. 2018/May. [Online]. Available: https://www.myheritage.com/family-tree-builder

[3] I. Neo4j. Neo4j graph platform. 2018/May. [Online]. Available: https://neo4j.com

[4] S. IT. Db-engines ranking of graph dbms. 2018/May. [Online]. Available: https://db-engines.com/en/ranking/graph+dbms

[5] R. Inc. Rootsmagic genealogy software. 2018/May. [Online]. Available: http://www.rootsmagic.com

[6] Levi-Strauss, *The Elementary Structures of Kinship*. London: Eyre and Spottiswoode, 1969.

[7] D. W. Read, "An algebraic account of the american kinship terminology," *Current Antropology*, vol. 25, no. 49, pp. 417–429, 1984.

[8] B. C. Smith, "Reflection and semantics in a procedural language," Ph.D. dissertation, MIT, Cambrige, 1982.

[9] H. J. Levesque and R. J. Brachman, "Expressiveness and tractability in knowledge representation and reasoning," *Comput. Intell.*, vol. 1, no. 3, pp. 78–93, 1987.

[10] T. Winograd, "Procedures as a representation for data in a computer program for understanding natural language," Ph.D. dissertation, MIT, Cambrige, 1971.

[11] H. Morowitz, "The plural of 'ontology' is 'confusion'," *Wiley Periodicals*, vol. 17, no. 6, 2012.

[12] B. Chandrasekaran and J. R. Josephson, "What are ontologies, and why do we need them?" *IEEE Intelligent Systems*, 1999.

[13] T. M. Ting, "Building a family ontology to meet consistency criteria," Master's thesis, University of Tun Hussien, 2015.

[14] J. B. F. B. T. Furche and S. Schaffert, "Web and semantic web query languages a survey."

[15] M. Marx, "Xcpath the first order complete xpath dialect."

[16] C. Lai and S. Bird, "Querying linguistic trees," 2009.

[17] A. A. R. K. A. K. V. R. F. Wolter and M. Zakharyaschev, "Ontology-mediated query answering over temporal data: A survey," *24th International Symposium on Temporal Representation and Reasoning*, vol. 1, no. 1, pp. 1–37, 2017.

[18] J. Bartlema and L. Winkelbauer, "Modelling kinship with lisp; a two-sex model of kin-counts," *IIASA Working Papers*, vol. WP-96-069, no. 1, p. 48, 1961.

[19] N. V. Findler, "Automatic rule discovery for field work in antropology," *Computers and the Humanities*, vol. 26, no. 1, pp. 285–292, 1992.

[20] V. Periclev and R. E. Valdes-Perez, "Automatic componental analysis of kinship semantics with a proposed structural solution to the problem of multiple models," *Anthropological Linguistics*, vol. 40, no. 2, pp. 272–317, 1998.

[21] M. Minsky, "A framework for representing knowledge," 1974.

[22] J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, no. 11, 1983.

[23] Y. Shohan, "Temporal logic in ai: Semantical and ontological considerations," *Artificial Intelligence*, vol. 33, no. 1, pp. 89–104, 1987.

[24] A. Schmiedel, "A temporal terminological logic," *AAAI-90 Proceedings*, vol. 1, no. 1, pp. 640–645, 1990.

[25] R. J. Brachman and J. G. Schmolze, "An overview of kl-one knowledge representation system," *Cognitive Science*, vol. 9, no. 1, pp. 171–216, 1985.

[26] P. B. L. Libkin, "Temporal logic over unranked trees," 2008.

[27] A. Tuzhilin and J. Clifford, "A temporal relational algebra as a basis for temporal relational completeness," *Proceedings of the 16th VLDB Conference*, 1990.

[28] M. S. Perry, "A framework to support spatial, temporal and thematic analysis over semantic web data," Ph.D. dissertation, University of Georgia, 2008.

[29] E. B. E. G. P. S. B. N. Maris and N. Papadakis, "Toql: Temporal ontology query language."

[30] B. Couderc and J. Ferrero, "Fr2sql : database query in french," *22eme Traitement Automatique des Langues Naturelles*, 2015.

[31] F. D. M. S. M.H.Shenassa, "Using natural language processing in order to create sql queries," *Proceedings of the International Conference on Computer and Communication Engineering*, 2008.

[32] P. P. Chaudhari, "Natural language statement to sql query translator," *International Journal of Computer Applications*, vol. 82, no. 5, 2013.

[33] R. Nelken and N. Francez, "Querying natural language databases using controlled natural language," 2001.

[34] L. H. Morgan, *Systems of consanguity and affinity of the human family*. Smithsonian Institution, 1870.

[35] N. Chomsky, "Three models for the description of language," *IRE Transactions on Information Theory*, vol. 2, no. 1, pp. 113–124, 1956.

[36] T. H. C. C. E. L. R. L. R. C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[37] B. Meyer, "Design by contract," *Technical Report TR-EI-12/CO*, vol. 12, 1986.

[38] K. Beck, *Test-Driven Development by Example*. Addison Wisley, 2003.

[39] K. B. et al. Junit 5 java library. 2018/May. [Online]. Available: https://junit.org/junit5/

Appendix A

Pseudocode Listings

Algorithm A.1: Kinship Term Reduction

```
1   input: kinship term t.
2   note: ω is a dictionary of kinship terms.
3   note: Functions "leftPart(t, u)" and "rightPart(t, u)" return the sub-term of t
4   note: from the left of sub-term u or from the right respectively.
5   note: Function "subterm(t, i, j)" returns the
6   note: sub-term of the kinship term t between indices i and j.
7   output: reduced kin term.
8   function shorten(t)
9   begin
10      maxShortenableSubterm ← empty
11      currentSubterm ← empty
12      for i ← 0 to length(t) do
13      begin
14          for j gets length(t) − i to 0 do
15          begin
16              currentSubterm ← subterm(t, i, j)
17              if length(currentSubterm) > length(maxShortenableSubterm)
18                  and ω(currentSubterm) is not empty
19              then
20                  maxShortenableSubterm = currentSubterm
21          end
22      end
23      return shorten(leftPart(t, maxShortenableSubterm))
24             · ω(maxShortenableSubterm)
25             · shorten(rightPart(t, maxShortenableSubterm))
26  end
```

Appendix B

Figures

Figure B.1: Main Screen of Family Tree Maker

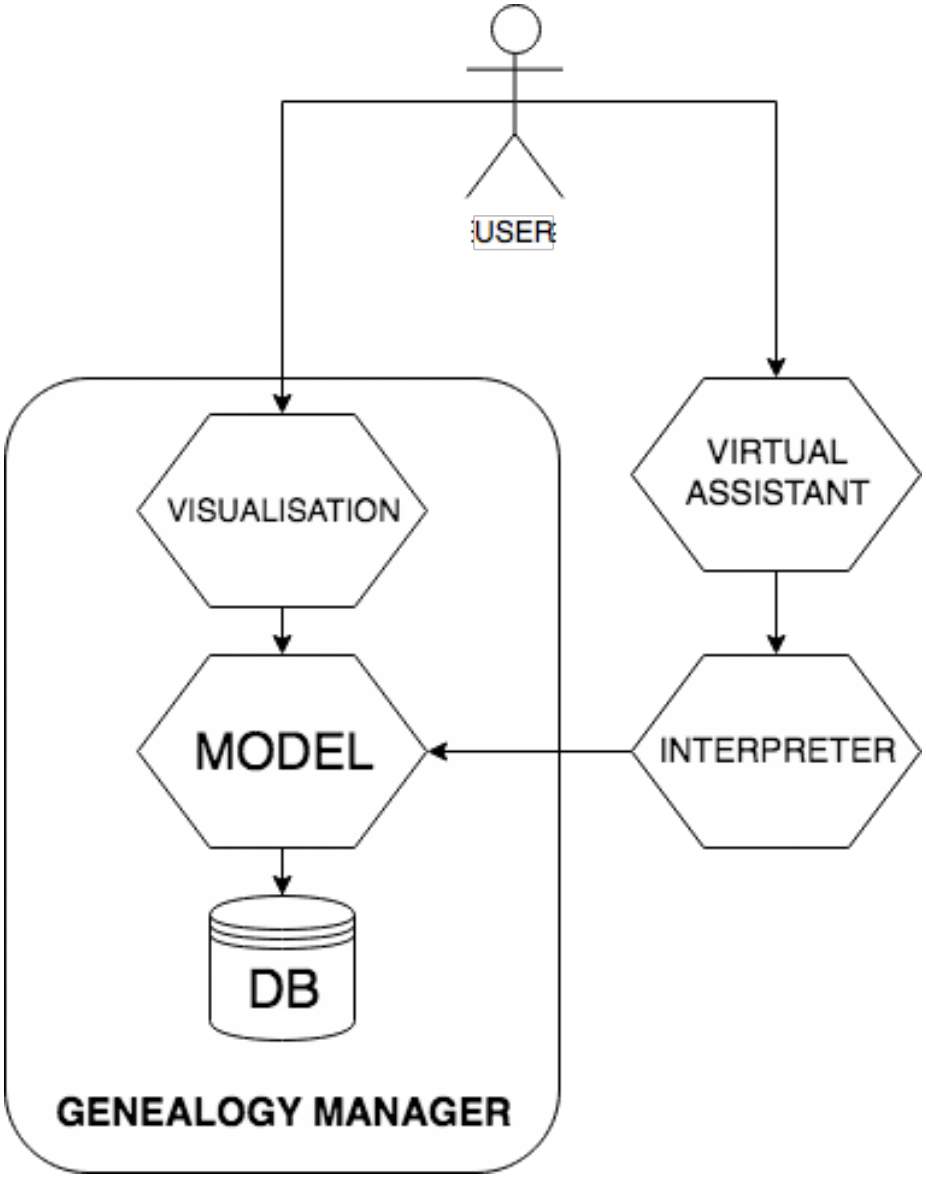

Figure B.2: Structure of System Components

Appendix C

Documentation of Standard KISP Functions

Note that two commas after a parameter name is a signature means any positive number of arguments. Also, a backtick (`) denotes a single quote (').

C.0.1 void

- Name: Void Constant
- Signature: `void`
- Description: A unique constant of type Void.
- Examples :
  - `(= void void) = true`
  - `(= void 3) = false`
- Type :

  Void

C.0.2 define

- Name: Define
- Signature: `(define reference term)`
- Description: Creates an abbreviation `reference` for the specified lisp `term`.
- Arguments:
  - **reference** : A shortcut for `term`. Can only be composed of lowercase English letters and a dash.
  - **term** : A well-formed lisp term as a designatum. Keep in mind that you can use `reference` in `term` to create recursion, but make sure that there are no cyclic definitions.
- Examples :
  - `(define name 'John Doe')`
  - `(define two 2)`
  - `(define twice (lambda (n) (f (f n))))`
- Type :

  Void

- Notes : This function cannot be used inside another functions, only as a top-level term. You cannot redefine standard keywords like define itself, `lambda` and so on.

C.0.3 lambda

- Name: Lambda
- Signature: `(lambda (args) lambda-term)`

- Description: Instantiates a new anonymous function which takes its arguments `args` and applies them to the `term`
- Arguments:
  - **args** : A list of argument names for this lambda-term separated by space
  - **term** : A well-formed lisp term that will be applied to arguments `args`
- Examples :
  - `(lambda () 2)`
  - `(lambda (n) (* 2 n)`
  - `(lambda (a, b) (+ a (* a b))`
- Notes : You can use only English letters and dashes for names of the arguments. Argument list is separated by single space. Empty argument list is permitted

C.0.4 and

- Name: And
- Signature: `(and boolean..)`
- Description: Performs a logical AND operation on N boolean terms
- Arguments:
  - **boolean1** : First boolean conjunct
  - **boolean2** : Second boolean conjunct
  - **...** : ...
  - **booleanN** : N-th boolean conjunct
- Examples :

  - `(and true) = true`
  - `(and false true) = false`
  - `(and true true false) = false`

- Type :

  $(Boolean)^n \, oBoolean$

- Notes : All conjuncts will be evaluated, always. Function accepts any non-zero number of arguments.

C.0.5 or

- Name: Or
- Signature: `Performs a logical OR operation on N boolean terms.`
- Description: (or boolean..)
- Arguments:
  - **boolean1** : First boolean disjunct
  - **boolean2** : Second boolean disjunct
  - **booleanN** : N-th boolean disjunct
- Examples :
  - `(or true) = true`
  - `(or false true) = true`
  - `(or false false true) = true`
- Type :

  $(Boolean)^n \rightarrow Boolean$

- Notes : All disjuncts will be evaluated, always. Function accepts any non-zero number of arguments.

C.0.6 not

- Name: Not
- Signature: `(not boolean-term)`
- Description: Performs a logical NOT operation on exactly one boolean term.
- Arguments:
  - **boolean** : A boolean term, whose value will be inverted.
- Examples :
  - `(not true) = false`
  - `(not false) = true`
- Type :

  $$Boolean \rightarrow Boolean$$

- Notes : This function accepts only one boolean argument.

C.0.7 during

- Name: During
- Signature: `(during date-point date-star date-end)`
- Description: Checks whether the date `date-point` is after `date-start` and before `date-finish`. Returns `true` iff `date-point` belongs to the time interval [`date-start`, `date-date-end`]
- Arguments:

- **date-point** : A date which is going to be checked for belonging to the time interval [`date-start`, `date-end`]
- **date-start** : A starting point of the time interval
- **date-end** : An ending point of the time interval

- Examples :
  - `(during WWII 1900 2000) = true`
- Type :

$$(Date)^3 \rightarrow Boolean$$

- Notes :

C.0.8 before

- Name: Before
- Signature: `(before date-pointA date-pointB)`
- Description: Checks whether date point A is happened before date point B. Returns true or false accordingly.
- Arguments:
  - **date-pointA** : First time point, which is needed to be before the second point for the formula to result in `true`.
  - **date-pointB** : Second time point
- Examples :
  - `(before WWII WWI) = false`
- Type :

$$(Date)^2 \rightarrow Boolean$$

- Notes :

C.0.9 after

- Name: After
- Signature: `(after date-pointA date-pointB)`
- Description: Checks whether date point A is happened after date point B. Returns true or false accordingly.
- Arguments:
  - **date-pointA** : First time point, which is needed to be after the second point for the formula to result in `true`.
  - **date-pointB** : Second time point
- Examples :
  - `(after WWII WWI) = false`
- Type :

$$(Date)^2 \rightarrow Boolean$$

- Notes :

C.0.10 date

- Name: Date
- Signature: `(date string-date)`
- Description: Parses `string-date` and returns a date object created from this string
- Arguments:

  - **string-date** : A string representation of a date object. Should be in the format: dd.mm.year.

- Examples :
  - `(date '10.11.1995')`
  - `(date '01.01.1900')`
- Type :

  $String \rightarrow Date$

- Notes : You should strictly follow the date format in order for this function to work. Void is returned iff invalid string is passed to this function.

C.0.11 list

- Name: List
- Signature: `(list args..)`
- Description: Creates a list object from passed arguments.
- Arguments:
  - **args** : A space-separated list of objects out of which list will be created
- Examples :
  - `(list 1 2 3 4)`
  - `(list 'first' 'second' 'third')`
  - `(list 1 'two' (date '1.1.1990'))`
- Type :

  $(Object)^n \rightarrow List$

- Notes : Accepts non-zero number of arguments. Parameters are allowed to be of different type. If you want an empty list, use the term `vacant`.

C.0.12 join

- Name: Join
- Signature: `(join list..)`
- Description: Concatenates N lists
- Arguments:
  - **list** : A space-separated sequence of lists to be concatenated
- Examples :
  - `(join list1 list2)`
  - `(join list)`
  - `(join (list 1 2) (list 3) (list 4)) = [1, 2, 3, 4]`
- Type :

$$(List)^n \rightarrow List$$

- Notes : Accepts non-zero number of arguments.

C.0.13 count

- Name: Count
- Signature: `(count list)`
- Description: Returns the number of elements in `list`.
- Arguments:

  - **list** : A list object.

- Examples :
  - `(count (list 1 2 3)) = 3`
  - `(count (list 1)) = 1`
- Type :

  $List \rightarrow Numeral$

- Notes : Accepts a non-zero number of arguments.

C.0.14 filter

- Name: Filter
- Signature: `(filter predicate list)`
- Description: Takes a predicate together with a list and retains only those elements, which satisfy predicates' condition.
- Arguments:
  - **predicate** : Lambda function of type $Object \rightarrow Boolean$, which we apply to all elements in `list`
  - **list** : A list which needs to be filtered
- Examples :
  - `(filter odd? (list 1 2 3 4 5)) = [1, 3, 5]`
  - `(filter uppercase? (list 'TEST' 'test' 'tEsT')) = ['TEST']`
- Type :

  $(Lambda \times List) \rightarrow List$

C.0.15 at

- Name: At
- Signature: `(at list index)`
- Description: Returns the `index`-th element of `list`.
- Arguments:
  - **list index** : A list whose element at specified place will be taken
- Examples :
  - `(at (list 1 2 3 4) 3) = 4`
  - `(at (list 0 1 2) 3) = void`
  - `(at (list 0 1 2 3 4 5) -1) = 5`
- Type :

  $(List \times Numeral) \rightarrow Object$

- Notes : Indices start with 0. If there is no item at specified index, then Void is returned. Indices can be negative, e.g. -1 points to last element, -2 – penultimate, -3 – antepenultimate and so on.

C.0.16 append

- Name: Append
- Signature: `(append list item..)`
- Description: Includes all `items` in the end of `list`.
- Arguments:
  - **list** : A list to which items will be included

  - **item..** : A space-separated sequence of objects that will be included in the `list`

- Examples :
  - `(append list 1)`
  - `(append (list 1 2 3) 4 5 6) = [1, 2, 3, 4, 5, 6]`
- Type :

$$(List \times (Object)^n) \rightarrow List$$

- Notes : You can pass any positive number of arguments

C.0.17 mod

- Name: Modulo
- Signature: `(mod a n)`
- Description: Computes a residue modulo `n`, i.e. the remainder of a natural division of `a` by `n`
- Arguments:
  - **a** : Dividend
  - **n** : Divisor
- Examples :
  - `(mod 5 2) = 1`
  - `(mod 10 4) = 2`
- Type :

$$(Numeral)^2 \rightarrow Numeral$$

- Notes : Divisor should not be zero.

C.0.18 add

- Name: Addition, or +
- Signature: `(+ number..)`
- Description: Calculates sum of all `numbers`.
- Arguments:
  - **number..** : A non-empty space-separated sequence of numeral terms.
- Examples :
  - `(+ 2) = 2`
  - `(+ 1357 10) = 1367`
  - `(+ 1 2 3) = 6`
- Type :

$$(Numeral)^n \rightarrow Numeral$$

- Notes : The function accepts any positive number of arguments

C.0.19 mul

- Name: Multiplication, or $*$
- Signature: `(* number..)`
- Description: Calculates product of all `numbers`.
- Arguments:
  - **number..** : A non-empty space-separated sequence of numeral terms.
- Examples :

  - `(* 2) = 2`
  - `(* 1357 10) = 13570`
  - `(* 1 2 3) = 6`

- Type :

  $(Numeral)^n \rightarrow Numeral$

- Notes : The function accepts any positive number of arguments

C.0.20 lessEqual

- Name: Less Or Equal, <=
- Signature: `(<= a b)`
- Description: Checks whether `a` ¡= `b`
- Arguments:
  - **a** : first numeral to be compared
  - **b** : second numeral to be compared
- Examples :
  - `(<= 2 5) = true`
  - `(<= 10 3) = false`
- Type :

  $(Numeral)^2 \rightarrow Boolean$

- Notes : Function expects exactly two numerals as arguments.

C.0.21 less

- Name: Less, or just $<$
- Signature: `(< a b)`
- Description: Checks whether $a < b$
- Arguments:
  - **a** : first numeral to be compared
  - **b** : second numeral to be compared
- Examples :
  - `(< 2 5) = true`
  - `(< 10 3) = false`
- Type :

$$(Numeral)^2 \rightarrow Boolean$$

- Notes : Function expects exactly two numerals as arguments.

C.0.22 greater

- Name: Greater, or just $>$
- Signature: `(> a b)`
- Description: Checks whether $a > b$
- Arguments:
  - **a** : first numeral to be compared
  - **b** : second numeral to be compared
- Examples :

  - `(> 5 2) = true`
  - `(> 10 13) = false`

- Type :

  $(Numeral)^2 \rightarrow Boolean$

- Notes : Function expects exactly two numerals as arguments.

C.0.23 greaterOrEqual

- Name: Greater Or Equal, >=
- Signature: `(>= a b)`
- Description: Checks whether $a \geqslant b$
- Arguments:
  - **a** : first numeral to be compared
  - **b** : second numeral to be compared
- Examples :
  - `(>= 5 5) = true`
  - `(>= 3 5) = false`
- Type :

  $(Numeral)^2 \rightarrow Boolean$

- Notes : Function expects exactly two numerals as arguments.

C.0.24 concat

- Name: String Concatenation
- Signature: `(concat string..)`
- Description: Concatenates all `strings`
- Arguments:
  - **string** : A non-empty space-separated sequence of string objects.
- Examples :
  - `(concat 'test' 'ing' ' concat') = 'testing concat'`
  - `(concat 'hello ' 'world!') = 'hello world!'`
- Type :

$$(String)^n \rightarrow String$$

- Notes : This function accepts any positive number of arguments.

C.0.25 of-type?

- Name: Of Type?
- Signature: `(of-type? o type)`
- Description: Returns true iff object `o` has a `type`.
- Arguments:
  - **o type** : An object whose type needs to be checked
  - **type** : A name of a type
- Examples :

  - `(of-type? 'test' 'string) = true`
  - `(of-type? 3 'date') = false`

- Type :

$$(Object \times String) \rightarrow Boolean$$

- Notes : The name of a type should be entered precisely

C.0.26 equals

- Name: Equals, or =
- Signature: `(= objA objB)`
- Description: Returns true iff `objA` and `objB` are exactly the same thing.
- Arguments:
  - **objA** : First object to be compared
  - **objB** : Second object to be compared
- Examples :
  - `(= '3' 3) = false`
  - `(= 10 10) = true`
  - `(= (date '1.1.1999') (date '1.1.1999')) = true`
- Type :

$$(Object)^2 \rightarrow Boolean$$

- Notes : Function accepts exactly two arguments

C.0.27 sub

- Name: Subtraction, or −
- Signature: `(sub m s)`
- Description: Calculates the difference between m (minuend) and s (subtrahend).
- Arguments:
  - **m** : Minuend
  - **s** : Subtrahend
- Examples :
  - `(sub 5 2) = 3`
  - `(sub 0 1) = -1`
- Type :

$$(Numeral)^2 \rightarrow Numeral$$

- Notes : Accepts exactly two arguments.

C.0.28 div

- Name: Integer division
- Signature: `(div n m)`
- Description: Calculates the quotient from integer division of n (dividend) by m (divisor)
- Arguments:
  - **n** : Dividend
  - **m** : Divisor

- Examples :
  - `(div 10 2) = 5`
  - `(div 5 2) = 2`
  - `(div 3 2) = 1`
- Type :

  $(Numeral)^2 \rightarrow Numeral$

- Notes : Accepts exactly two arguments. Divisor shouldn't be zero, or exception will be thrown

C.0.29 father

- Name: Father
- Signature: `(father person)`
- Description: Returns father of the specified person
- Arguments:
  - **person** : A link to a person object
- Examples :
  - `(father (person 'John' 'Golt'))`
  - `(father (person 'Emma Clark'))`
- Type :

  $Person \rightarrow Person$

- Notes : Accepts and returns exactly one object of type Person. If provided person doesn't have a father, returns empty list.

C.0.30 mother

- Name: Mother
- Signature: `(mother person)`
- Description: Returns mother of the specified person
- Arguments:
  - **person** : A link to a person object
- Examples :
  - `(mother (person 'John' 'Golt'))`
  - `(mother (person 'Emma Clark'))`
- Type :

  $Person \rightarrow Person$

- Notes : Accepts and returns exactly one object of type Person. If provided person doesn't have a mother, returns empty list.

C.0.31 spouse

- Name: Spouse
- Signature: `(spouse person)`
- Description: Returns spouse of the specified person
- Arguments:
  - **person** : A link to a person object
- Examples :
  - `(spouse (person 'John' 'Golt'))`

  - `(spouse (person 'Emma Clark'))`

- Type :

  $Person \rightarrow Person$

- Notes : Accepts and returns exactly one object of type Person. If provided person doesn't have a spouse, returns empty list.

C.0.32 children

- Name: Children
- Signature: `(children person)`
- Description: Returns list of persons' children
- Arguments:
  - **person** : A link to a person object
- Examples :
  - `(children (person 'John' 'Golt'))`
  - `(children (person 'Emma Clark'))`
- Type :

  $Person \rightarrow List$

- Notes : Returns empty list if the specified person is childless.

C.0.33 gen-dist

- Name: Generation Distance
- Signature: `(gen-dist person1 person2)`

- Description: Calculates the `generation distance` between two people
- Arguments:
  - **person1** : First person
  - **person2** : Second person
- Examples :
  - `(gen-dist me (father me)) = -1`
  - `(gen-dist me (grandfather me)) = -2`
  - `(gen-dist me (son me)) = 1`
- Type :

$$(Person)^2 \rightarrow Numeral$$

- Notes : Generation distance is the difference between generations. The generation of the first person is assumed to be zero, and the generation of the second person is calculated accordingly.

C.0.34 person

- Name: Person
- Signature: `(person first-name)`
- Description: Returns the person with specified first and last name
- Arguments:
  - **first-name** : A string with the first name of a person
- Examples :
  - `(person 'John' 'Doe')`

- Type :

  $String^2 \rightarrow Person$

- Notes : The function also accepts full name in one string: (person 'Full Name')

C.0.35 kinship

- Name: Kinship
- Signature: `(kinship person1 person2)`
- Description: Returns a list of basic kinship terms (strings) that represents how is `person1` related to `person2`.
- Arguments:
  - **person1 person2** : First person
- Examples :
  - `(kinship (father me) me) = ['father']`
  - `(kinship (uncle me) me) = ['son','parent','parent']`
- Type :

  $Person^2 \rightarrow List$

- Notes : Basic kinship terms are: `father`, `mother`, `parent`, `son`, `daughter`, `child`, `husband`, `wife`, `spouse`

C.0.36 attr

- Name: Get Attribute
- Signature: `(attr person prop)`

- Description: Returns requested property `prop` of the specified `person`.
- Arguments:
  - **person** : Person of interest
  - **prop** : Property of interest
- Examples :
  - `(attr (person 'John Golt') 'first name') = 'John'`
  - `(attr (person 'John Golt') 'sex') = 'MALE'))`
- Type :

$$(Person \times String) \rightarrow Object$$

- Notes : All possible properties are: `first name`, `last name`, `full name`, `second name`, `birth`, `birth date`, `date of birth`, `gender`, `sex`, `birthplace`, `phone`, `phone number`, `tel`, `email`, `e-mail`, `wedding`.

C.0.37 shorten

- Name: Shorten Kinship Term
- Signature: `(shorten list)`
- Description: Reduces the list of basic kinship terms.
- Arguments:
  - **list** : List of basic kinship terms.
- Examples :
  - `(shorten (kinship person1 person2))`

- Type :

  $List \rightarrow List$

- Notes :

C.0.38 put-kinship-term

- Name: Put Kinship Term
- Signature: `(put-kinship-term list-basic shortcut)`
- Description: Registers a new custom kinship term `shortcut` from the list of basic terms `list-basic`. The new term can be used later in `shorten` function.
- Arguments:
  - **list-basic** : Non-empty list of basic kinship terms
  - **shortcut** : A string with a definition of a new kinship term
- Examples :
  - `(put-kinship-term (list 'son' 'parent' 'parent') 'uncle')`
  - `(put-kinship-term (list 'son' 'parent') 'brother')`
- Type :

  Void

- Notes :

C.0.39 vacant

- Name: Vacant List
- Description: An empty list constant
- Examples :
  - `(count vacant) = 0`
- Type :

  List

C.0.40 map

- Name: List Mapping Function
- Signature: `(map f list) or (map f l1 l2)`
- Description: Applies the unary function f to each element of a specified list or constructs a third list by applying binary f to each consecutive pair of elements from lists l1 and l2.
- Arguments:
  - **f** : An unary or binary mapping function
  - **list** : A list whose elements will be mapped one by one
  - **l1** : A list, each item of which will be supplied to f as a first argument
  - **l2** : A list, each item of which will be supplied to f as a second argument
- Examples :
  - `(map square (first-n 3)) = [1, 4, 9]`
  - `(map + (list 1 2 3) (list 4 5 6)) = [5, 7, 9]`

- Type :

$$EitherLambda \times List \rightarrow ListorLambda \times (List)^2 \rightarrow List$$

- Notes :

C.0.41 tail

- Name: Tail
- Signature: `(tail list)`
- Description: Return the tail, i.e. all but the first element of a list
- Arguments:
  - **list** : A specified list
- Examples :
  - `(tail (list 0 1 2)) = (list 1 2)`
  - `(tail (list 1)) = vacant`
- Type :

$$List \rightarrow List$$

- Notes : This function is analogous to `cdr` in Common Lisp.

C.0.42 head

- Name: Head
- Signature: `(head list)`
- Description: Returns the head, i.e. the first element of a list
- Arguments:

  - **list** : A specified list

- Examples :

  - `(head (list 1 2 3))= 1`

  - `(head vacant) = void`

- Type :

  List → *Object*

- Notes : This function is analogous to `car` in Common Lisp.

C.0.43 now

- Name: Now

- Signature: `now`

- Description: A date constant that hold the current time.

- Examples :

  - `(before now now) = false`

- Type :

  Date

C.0.44 people

- Name: People

- Signature: `people`

- Description: A constant of type List that holds all persons in a family tree.

- Examples :
  - `(head people)`
- Type :

  List

C.0.45 string

- Name: String
- Signature: `(string numeral)`
- Description: Converts given integer to string.
- Arguments:
  - **numeral** : A specified integer.
- Examples :
  - `(string 12) = '12'`
  - `(string 10000) = '10000'`
- Type :

  Numeral → *String*

C.0.46 substr

- Name: Sub-string
- Signature: `(substr source start [end])`
- Description: Return a substring of the specified source, starting from 'start' and ending just before 'end'.
- Arguments:

  - **source** : A super-string, from which we want to get a sub-string
  - **start** : An inclusive starting index of the substring. Should be positive.
  - **end** : An optional exclusive ending index of the substring. Should be positive and not less than 'start'.

- Examples :
  - `(substr '01234' 0 0) = ''`
  - `(substr '01234' 0 2) = '01'`
  - `(substr '01234' 1 3) = '13'`
  - `(substr '01234' 1) = '1234'`
- Type :

$$String \times Numeral \times Numeral \rightarrow String$$

- Note : The last argument can be omitted, resulting in it being equal to the length of a source string.

C.0.47 day

- Name: Day of month
- Signature: `(day date)`
- Description: Return the day of a month of the specified date as a numeral.
- Arguments:
  - **date** : A date object, whose day component is needed.
- Examples :
  - `(day (date '1.2.2000')) = 1`

  - `(day (date '25.3.2001')) = 25`

- Type :

  Date → *Numeral*

C.0.48 month

- Name: Month
- Signature: `(month date)`
- Description: Return the month of the specified date as a numeral.
- Arguments:
  - **date** : A date object, whose month component is needed.
- Examples :
  - `(month (date '1.2.2000')) = 2`
  - `(month (date '25.3.2001')) = 3`
- Type :

  Date → *Numeral*

C.0.49 year

- Name: Year
- Signature: `(year date)`
- Description: Return the year of the specified date as a numeral.
- Arguments:
  - **year** : A date object, whose year component is needed.

- Examples :
  - `(year (date '1.2.2000')) = 2000`
  - `(year (date '25.3.2001')) = 2001`
- Type :

  $Date \rightarrow Numeral$