JSPY: UM MODELO OBJETIVO PARA COMPREENSÃO DE LINGUAGEM NATURAL

VINICIUS V. M. GARCIA

JSPY: UM MODELO OBJETIVO PARA COMPREENSÃO DE LINGUAGEM NATURAL

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Adriano Veloso

Belo Horizonte, Minas Gerais - Brasil

Março de 2017

VINICIUS V. M. GARCIA

JSPY: AN OBJECTIVE MODEL FOR NATURAL LANGUAGE UNDERSTANDING

Dissertation presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Adriano Veloso

Belo Horizonte, Minas Gerais - Brasil

March 2017

V. M. Garcia, Vinicius

D1234p JSpy: an Objective Model for Natural Language Understanding / Vinicius V. M. Garcia. — Belo Horizonte, Minas Gerais - Brasil, 2017
xxii, 64 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Adriano Veloso

1. Question Answering. 2. Natural Language Processing. 3. Pattern Matching

CDU 519.6*82.10

[Folha de Aprovação]

Quando a secretaria do Curso fornecer esta folha,
ela deve ser digitalizada e armazenada no disco em formato gráfico.

Se você estiver usando o `pdflatex`,
armazene o arquivo preferencialmente em formato PNG
(o formato JPEG é pior neste caso).

Se você estiver usando o `latex` (não o `pdflatex`),
terá que converter o arquivo gráfico para o formato EPS.

Em seguida, acrescente a opção `approval={`*nome do arquivo*`}`
ao comando `\ppgccufmg`.

Se a imagem da folha de aprovação precisar ser ajustada, use:
`approval=[`*ajuste*`][`*escala*`]{`*nome do arquivo*`}`
onde *ajuste* ÃĽ uma distância para deslocar a imagem para baixo
e *escala* é um fator de escala para a imagem. Por exemplo:
`approval=[-2cm][0.9]{`*nome do arquivo*`}`
desloca a imagem 2cm para cima e a escala em 90%.

Dedico esse trabalho à toda a comunidade da UFMG, meus colegas de estudos e professores, por me inspirarem e darem apoio no desenvolvimento deste projeto.

Acknowledgments

Throughout this journey I have been blessed with the company of important friends and the partnership of many colleagues. I want to give special thanks to my parents for all the structure and care, to my brother Caio, for participating on the discussions about this project right from the beginning and to my girlfriend Thayane for always giving me support and encouragement. I could not have done it without the two professors that advised me during this project: Sérgio Campos, for the support during the development and Adriano Veloso, for the discussions and enthusiasm. I also thank my friends Pablo Nunes, Jerônimo Rocha and Sávio Martins for always getting enthusiastic when we discussed this project, providing me with insights and support.

Abstract

In this work we present a model for the Human Sentence Parsing Mechanism and an implementation of this model called JSpy Programming Language. This system is capable of describing structured information in a way much similar to how humans do. Allowing complex meanings to be expressed in few lines of code. The system is evaluated by resolving a set of problems related to specific parsing skills proposed by Weston et al.. The resolution of these problems not only occupy very few lines of script but also exceeds current state of art solutions.

Keywords: Question Answering, Natural Language Processing, Pattern Matching.

Extended Abstract

1. Introduction

The human brain is considered by some the most powerful computer designed by nature, but its big complexity makes it hard to replicate. The effort to understand it has inspired many researchers in different areas of knowledge. Among them the field of Computer Science has directed its efforts on designing theoretical models for the brain that are not only able to replicate its features at some degree but that are also implementable in modern computers.

In this work we presented 2 concepts: (1) The *JSpy Model* designed to describe the human brain and (2) the *JSpy Programming Language* that is an implementation of this model in the form of a modern and well structured programming language. The JSpy Programming Language is inspired on Python and JavaScript, making it very familiar to modern programmers. The key feature of the language is the *JSpy Matcher* construct that describes a pattern matching system comparable to a Turing Machine. The JSpy Matcher design joins together the concepts of Regular Expressions and of modern programming languages with the JSpy Model for the human brain. Making it very powerful and providing new insights on how the brain represents information, meaning and data structures.

2. Methodology

To evaluate JSpy as a whole, we have solved 17 of the the 20 *bAbI* (Weston et al. [2015]) problems that are related with specific parsing skills. bAbi was made available by Facebook researcher Weston et al. and provides a set of careful planned Question Answering problems. Each problem of the 20 is responsible for testing the capacity of a program to solve a different NLU task. Our results shows very superior precision in comparison to current state of art approaches, and also reveal several distinguishing features of the JSpy model showing how powerful and interesting it can be. Furthermore the scripts are very short, hopefully making them easy to learn for future users.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The human brain is considered by some the most powerful computer designed by nature, but its big complexity makes it hard to replicate. The effort to understand it has inspired many researchers in different areas of knowledge. Among them, the field of Computer Science has directed its efforts on designing theoretical models for the brain that are not only able to replicate its features at some degree but that are also implementable in modern computers. This study has provided some interesting insights and techniques such as *Reinforcement Learning* (Richard S. Sutton [1998]), *Prolog* (Covington [1994]) and *Neural Networks* (Goldberg [2015]), that have greatly advanced the state of art in different areas. Neural Networks, in particular, is currently the most popular model and has been able to solve with a high level of accuracy very complicated problems such as image and voice recognition and Natural Language Processing tasks; often performing even better than humans. There are, however, some fields not even the Neural Networks model has been able to solve with a high level of accuracy. Among them, Natural Language Understanding (NLU) is a specially hard and interesting area.

Understanding and producing text in human language can, arguably, be the most complex activity humans do. It integrates feelings, emotions, memory and a variety of information processing systems on each time its used to write or read a message. It not just differentiates humans from animals, but it is often considered (Noormohamadi [2008]) the most important step towards human intelligence. A successful model for text comprehension could bring new paradigms for Artificial Intelligence, Machine-Learning, Human-Computer Interaction and also on other fields such as Psychology and Biology (Graham-Rowe [2007]).

Although the first attempts at NLU started as early as 1964, with Bobrow's STUDENT (Bobrow [1964]), the progress in the area has been slow. Some models have achieved some success in describing the human language structure such as Woods's ATNs (Woods [1970]), however, they struggled when dealing with larger fractions of the human language at once. In recent years there has been an increasing effort on text comprehension with Neural Networks and Machine Learning techniques that are capable of dealing with big chunks of information, solving several real-world Natural Language problems. However, the stochastic nature of these techniques prevents them from expressing well-defined structures on their internal models. As a result, NLU problems that require well defined structures such as (1) Question Answering, (2) Summarization and (3) Automatic Dialog are still mostly unsolved.

In this work we presented two concepts: (1) The *JSpy Model* designed to describe the human brain and (2) the *JSpy Programming Language* that is an implementation of this model in the form of a modern and well-structured programming language. The JSpy Model for the human brain was created to provide an adequate architecture for Natural Language Parsing systems. The model can be roughly described in three steps: (1) how the brain represents information, (2) how the brain represents patterns, and use them, and (3) how the brain groups these patterns into conceptual groups that offer a substantial number of interesting features.

The JSpy Programming Language is based on Python and JavaScript, making it very familiar to modern programmers. The key feature of the language is the *JSpy Matcher* construct that describes a pattern matching system comparable to a Turing Machine[1] and designed accordingly to the conceptual groups described in JSpy Model. The JSpy Matcher design makes use of familiar tools as Regular Expressions and a programming language to implement an interesting model for the human brain. This work aims to provide new insights on how the brain represents information, meaning and data structures, as well as providing a tool to illustrate and explore the possibilities of this model.

To evaluate JSpy as a whole, we have solved 17 from the 20 bAbI problems (Weston et al. [2015]). These problems are a set of carefully planned Question Answering problems, designed for testing different skill sets expected from Natural Language Understanding tools. Each problem of the 20 is responsible for testing the capacity of a program to solve a different NLU task. Our results show very superior accuracy in comparison to current state of art approaches, and also reveal several distinguishing features of the JSpy model showing how powerful and interesting it can be. Furthermore, the scripts are very short, hopefully making them easy to learn for future users.

[1] This claim is not proved here, but it is easy to prove that the JSpy Language is equivalent in expressive power to the ATN model described on Section 2, which is known to be Turing Complete (Bates [1978]).

1.1 Thesis Statement

In this work, we present a framework, called JSpy, on which is possible to develop NLU applications easily, rapidly and using a model that actually fits the Natural Language nuances. The JSpy "*matcher*" is simple and flexible enough to describe a diversity of structures that together are capable of organizing meaning and explicitly dealing with ambiguities with the support of a modern and flexible programming language.

1.2 Thesis Organization

On Chapter 2 we present a listing of related works and discuss their resemblances and differences, on Chapter 3 we describe the JSpy approach, its design, how it was implemented and the user-friendly syntax and features implemented for facilitating the access to the technology. On Chapter 4 the JSpy model is compared to other NLU solutions by resolving a set of problems proposed by the Facebook team to test different parsers. Finally, the conclusions are discussed in Chapter 5.

Chapter 2

Background and Related Work

In this section, we present concepts and previous works that are related to the JSpy Model and Programming Language.

2.1 Patom Theory

The Ball et al.'s Patom Theory (Ball et al. [2012]) describes a generic model for the brain, very much like the JSpy Model. Ball's company: Pat Inc. has developed a proprietary implementation of the model using Neural Networks that is capable of extracting structured meaning from texts and voice input in a very reliable manner. As much as the models look similar JSpy description of the model is much less abstract and will likely give better insights and be of more use to the scientific community. Furthermore, JSpy Language is open-source and as so can be tested and experimented with by anyone.

2.2 Augmented Transition Networks

The Augment Transition Networks (ATNs) designed by Woods (Woods [1970]) is a powerful and efficient model for describing language in terms of a Transition Network and it is as powerful as a Turing Machine as proved in Bates [1978]. The differential that makes it possible is the use of recursion tied to the fact that custom code can be inserted on each transition, helping to validate it and allowing the extraction of useful information. The ATNs are very capable of expressing much of the same concepts as the JSpy Matcher implementation and has been used in early attempts to model the Human Sentence Parsing Mechanism (Kaplan [1972]). JSpy differs from ATNs first in its conception: ATNs were designed to express grammar as an efficient automaton, while JSpy was designed as a model for brain specialized on NLU. As a consequence of this difference, JSpy (1) includes the idea of ambiguities in text not as a corner case but as an expected case, (2) JSpy is less concerned with efficiency, (3) JSpy accepts on the fly updates on its own internal structures (as humans do when learning). An extra difference between the two models is the age of them: JSpy being newer is designed with modern programming languages concepts and features, with the idea of facilitating the programmer's job as much as possible.

2.3 Intelligent Personal Assistants

Intelligent Personal Assistants (IPAs) as described in Gong [2003] were designed as an human-interaction tool, designed to help humans perform computer tasks such as a simple query or command. Examples of such agents include *Apple's Siri*, *Google Now* and *Microsoft's Cortana*. IPAs are a product of current work of art Natural Language Processing tools, and as described by Valin (Valin [2016]), they are more closely related to Machine Learning, than with the Natural Language Understanding field. This brings some advantages when solving simple tasks: It does not require to fully understand the text to give a likely good answer. However, it struggles to deal with complex language constructs, since it does not attempt to understand the underlying structure of a sentence. This makes it useful for interpreting small contextualized texts, but incapable of fully understanding larger Natural Language sentences as JSpy proposes.

2.4 Wolfram

Wolfram technologies (Wolfram Research [2016]) have produced impressive results in NLU. In special *Wolfram Alpha* Website[1] is capable of interpreting fairly complex Natural Language queries answering with human readable graphs numbers and statistics. *Wolfram Programming Language* which is the system used to create all Wolfram products is a highly symbolic programming language. One resemblance of this system with the theory presented here is the fact that Wolfram Language has a built-in pattern matching system[2] that is responsible for evaluating what to do with the arguments of a function. However, to fully understand the theory used by Wolfram or to compare

[1]`http://www.wolframalpha.com/`

Wolfram Language with JSpy is complicated, since all Wolfram products are proprietary.

2.5 bAbI Facebook's Project

The 20 bAbI toy problems proposed by Facebook researchers (Weston et al. [2015]) were designed to test different skill sets required by Natural Language Processing and NLU applications. Each problem is composed of simulated stories where actors interact in a virtual world; each story is intermixed with questions about the virtual actors, and the answers are made available after a TAB character, so that machine learning programs can use it as train sets. These tests were designed for training machine learning tools; However, they are also ideal for studying how JSpy Features interact with each skill set.

[2] `http://reference.wolfram.com/language/guide/PatternMatchingFunctions.html`

Chapter 3

A new Approach for Natural Language Parsing

In this chapter, we will explain JSpy in details. The first part, Section 3.1, will focus on the JSpy Model, while the second part, Section 3.2, will describe the important aspects of JSpy programming language and of the JSpy Matcher construct that implements the model.

3.1 A Model for the Human Sentence Parsing Mechanism

The creation of a model for describing Natural Language is a challenge, ambiguities occur often, and even if grammars can describe *part* of its rules, there is no good way yet to describe "meaning" or to comprehend abstract sentences.

The JSpy model was created as an attempt to mimic the behavior and structural features of the human brain. It is important to point out that this model holds some resemblance with Ball et al.'s Patom Theory. Both models describe the brain as pattern matching system. They both also describe the meaning processing mechanism as a consequence of the design and interactions of the patterns as an interconnected system.

The creation of the JSpy model was based on introspective observations of how the human brain interprets language. Its guideline was to, by observation, make sure it matches *all* the features of the human brain disregarding how inefficient it might be to have it implemented on a modern computer.

As a consequence of this guideline, some important features are built into the model from the start, and others were gathered as the model developed. These features amount to a total of nine important features described in Subsection 3.1.9, to cite three of them:

1. **Explicit Ambiguity Resolution**
   Ambiguity processing is in the core of the model and should not be treated as a corner case.

2. **Dynamic Pattern Updates**
   Structures are dynamic, and new patterns can be learned, forgotten and updated while the system is running.

3. **Distributed Processing Feasibility**
   Although this is not a required feature for an implementation of the model, it is required for the model to provide full support for each processing module to be processed in parallel, not only sequentially.

The following subsections will first provide an overview of the model and then discuss the merits of its features.

3.1.1 Describing Information: The Snapshots

To recognize text and words it is necessary to have a consistent way to describe the expected text. This is valid for the human brain, as well as it is for this model.

In the human brain, these events are represented as internal signals produced by sensory receptors such as eyes, ears, tongue and skin. While in a computer they would be likely represented as byte arrays.

In this model, the most granular chunk of information considered will be labeled a **Snapshot**, according to Ball et al.'s description (Ball et al. [2012]). A Snapshot is responsible for working as a building block for describing any kind of information on a chosen context. For instance, a pixel could be chosen as the Snapshot in the context of describing figures, but since pixels are encoded as bytes, one could argue that a byte array is a better choice of representation. Since our effort is to describe language, we will often restrict the Snapshot concept to characters. But it is important to keep in mind the full depth of the Snapshot concept.

The same model can be described with different sets of Snapshots. For example, we could make the arbitrary choice of representing language with phonemes instead of letters and it would likely have the same outcome or very close. This is also valid for people, learning different languages is further complicated by the difference in the building blocks each person's brain was built upon. For example, Japanese speakers are used to a phonetic alphabet, as such, they have a hard time hearing and pronouncing consonants since they do not exist in their phonetic model.

The important thing when choosing a set of Snapshots as the tiles for a pattern system is that they should be unique and capable of expressing all the signals expected to be produced and/or received. It is also preferable, for simplicity, to avoid redundancy among them: avoiding creating complex Snapshots when they could be described as a sequence of smaller ones. In this point, our definition of Snapshot disagrees with Ball et al.'s, where he defines them as the smallest possible representation of an input. We believe this concept of the smallest representation is vague. Humans, in fact, seem to represent information in arbitrary ways according to their respective cultures and contexts.

3.1.2 Building Patterns from Snapshots

To create an actual Pattern and describe real world events, a set of Snapshots must to grouped together. There is arguably three types of operations that could describe the occurrence of a Snapshot in relation to other Snapshots to form patterns:

- **Simultaneous-occurrence**:
  When both happen at the same time.
- **Alternate-occurrence**:
  When any of them might happen and would mean the same thing.
- **Relative-occurrence**:
  When it is expected for one of them to happen before or after the other.

However, in the way characters are currently represented it is not possible for two of them to occur simultaneously. It could, however, be argued that the occurrence of a letter together with an accent would represent a simultaneous occurrence. But, for simplicity, our chosen set of Snapshots will be based on the Unicode convention: each character, with an accent and without it, are considered separated representations, not being possible for two of them to occur at the same time in a text string.

Another point deserving attention is that the Relative-occurrence operation should possibly include other types of relations; for example, when evaluating patterns in a figure one might argue that the occurrence of one pattern above or below the other has an important meaning. However, this does not concern our task, and therefore we will consider only the linear occurrence relationship between letters.

If the reader has experience with the capabilities of **Regular Expressions** he might have noticed it is quite capable of expressing the relationships of alternate and relative occurrence regarded in this section. For this reason, this artifice will be used to help to describe the model and afterward, the JSpy Matcher construct as well. For an introduction to regular expressions, there are plenty of resources available online. For a short overview of the concept see Wikipedia [2017] on the Regular Expressions topic.

3.1.3 Snapshot Polymorphism: A Pattern is a Snapshot as well

Now that we have defined what a pattern is, we should go a step back and notice that the event of a pattern being recognized can also be considered an input for the pattern matching mechanisms. Inside the brain we could argue that this event would produce an electric signal no different from the signal of a snapshot, thus making the pattern recognition itself a new type of snapshot.

The consequence of this feature is that an entire pattern could be used to build a more complex pattern using the same rules explained in the previous section. For example to denote a pattern formed by the word "foo" *or* "bar", such as denoted by the regular expression "foo|bar", would actually mean to build a pattern formed by an alternate-occurrence relationship between the patterns "foo" and "bar".

This feature also modularizes the architecture of the pattern matching. Making the meaning of pattern to depend, recursively, on other patterns spread across the system. This behavior makes it comparable to Recursive Transition Networks (Nierhaus [2009]) which are a basic structural concept behind the Augmented Transition Networks described in Chapter 2.

3.1.4 Pattern Grouping and its Consequences

At this point, we already discussed (1) how a pattern can be treated as a Snapshot, (2) that Snapshots can be joined together in the form of an alternate-occurrence relationship. With these two features described emerges a new possibility, it is now possible to join together a group of patterns as a pattern itself. For example, similar objects such as "door" and "window" could be joined on a group. To facilitate the use of this group lets give it a label such as "open-able objects".

This important construct will be referenced later as a *Pattern Group*, i.e. a pattern built by joining several patterns using the alternate-occurrence relationship.

Such a group could then be used on further patterns, for example, to interpret a phrase such as “open the door” one could describe it as a concatenation between the pattern “open the ” and the “open-able objects” group; Borrowing JSpy syntax[1], such a pattern would look like this:

```
pattern = "open the (openable_objects)"
```

This concept brings generalization to the model. A single pattern like the one described above can now work with several different objects, instead of a single one. Also since the system is bound to be *Dynamic* one group can be updated at any time, making it capable of storing information and even learn new words.

This feature has a special importance in the matter of building a comprehensible knowledge base: Using labels to group patterns, although not precisely mimicking the human brain, allows our expressions to be readable, and also, in the case of JSpy Language, to store different sets of information in different parts of the code.

3.1.5 Expressing Boolean Logic

Another natural consequence of *Snapshot Polymorphism* is that a single pattern recognition can now depend on multiple other patterns to be recognized simultaneously as well. This is possible by building a pattern using the simultaneous-occurrence relationship described in Subsection 3.1.2. This feature is useful as a guard to restrict the recognition of specific patterns to specific contexts. For example, a pattern might be constructed to be recognized only if the pattern “run” and the pattern of “being underwater” are recognized simultaneously, and could possibly be linked with the meaning of “swim away”.

This feature allows for patterns to be sensitive to contextual information available in the form of several other patterns on the system. We could even describe Boolean variables and literals in terms of patterns: A pattern that is never recognized is equivalent to False, while a pattern that is always recognized is equivalent to True. And finally, a Boolean expression could be described as a pattern built by using both the simultaneous-occurrence and the alternate-occurrence relationships to represent operations “and” and “or” respectively.

This concept of Boolean Expressions differs from the Pattern Groups construct in the matter of purpose: This construct is designed to be used as a context sensi-

[1]Although the syntax used in the example is similar to JSpy it was slightly simplified to make sure it is easy to comprehend

tive mechanism. Thus, allowing different parts of the system to be used in different environmental contexts.

This construct will be later referenced by the name of **Pattern Boolean Expressions** on sections 3.1.7.1 and 3.1.8.

3.1.6 Describing Meaning

JSpy Model approach to meaning can be defined in terms of the set of internal reactions caused as a response to an input stimulus. That is to say that the model should be capable of extracting information from the text, saving it internally when required and producing a comprehensible response for each input. To produce this reaction the model will rely on three internal features for processing meaning: (1) Pattern Interactions, (2) Pattern Updating and (3) External Subroutine Calls.

The concept of **Pattern Interactions** is a broad name to describe the feasibility[2] of using several patterns as an interconnected network for processing input: Once the meaning of a single pattern in being processed it should be capable of forwarding this information to more specialized nodes in this network, effectively breaking more abstract data into simpler structures.

To illustrate this with a more tangible example, in JSpy Language it is possible to program a pattern to recursively call other patterns on recognition. These recursively activated patterns will have access to part of the information extracted from the original input and will work analogously to recursive functions.

The concept of **Pattern Updating** is a reference to an inherent capability expected to be available: The capacity for creating, removing or updating patterns on demand.

This feature, although it looks simple, is very powerful for it allows the system to change its global state as a response to an input.

In Section 3.1.4 it has been shown that a pattern can form Boolean Expressions. By tinkering with different pattern structures it is even possible to emulate data containers and the concept of types of objects. Details on these possibilities are shown on **Appendix C**.

External Subroutine Calls concept is meant to describe every possible way the system can interact with the external world.

If a person wants to interact with the world, it would be required to move muscles and other devices in order to produce an appropriated response. In the case of a

[2] This feature can be demonstrated as a natural consequence of the already described concepts, however, the explanation would be further complicated by doing this.

computer, these responses would be provided by accessing nearby devices, such as a screen, a printer or even a robotic arm.

One final consideration about these complex concepts is that they are too far away from the way we are used to describing algorithmic responses. To make such a system usable it would benefit from a set of more familiar concepts. This reasoning justifies adding a programming language as a supporting device for the model, and this is where JSpy Language comes in: It is the language responsible for describing the meaning and the recursive interactions between different patterns.

3.1.7 Dealing with ambiguities

As noted before, distributed processing is a core concept of this model. In fact, all the operations of parsing, creating and removing patterns are expected to run in parallel in several parts of the brain at the same time. As such, the normal behavior of the brain is not to obtain a single possible interpretation of every aspect of the environment, but rather to collect as much information from it as possible. For example, when a human reads an ambiguous sentence it will not perceive a single interpretation and forget about the other one, it will most likely concern with both of them and try to guess the most reasonable response to this input. A response, in this case, might even be to consider both interpretations as possibly true and take actions that respect any restriction both of them impose.

Ambiguity resolution then is not just a matter of removing ambiguous interpretations from consideration as soon as possible, but rather to make sure to reduce the interpretations to a point where it is possible to make reasonable decisions based on the remaining ones. Reducing the possible interpretations to a single one is a requirement imposed mostly by demands on the external environment such as: answering a question, making a decision or planning a course of action to deal with some problem.

3.1.7.1 Defining ambiguity

Ambiguities in this model arise in the form of multiple recognitions for the same input stimulus. But it might also appear during the recursive processing of meaning as described in the previous subsection. For both cases, we will assume there is a meaning process system monitoring each call and prepared to receive their results, that could be described as changes in the global state of the system.

This description is a little vague and opens many possibilities of how to organize the meaning modules of each pattern and the modules responsible for resolving ambiguities generated by these patterns. The approach used by JSpy *Language* is a concrete simplification of this part of the model: The same module responsible for calling a Pattern Network is also used to monitor its outputs and only stops working when all sub-processes have finished, just like any recursive function. To further facilitate this process of monitoring the pattern chain, JSpy Language makes each pattern recognized to return a value, and when multiple pattern return values for a single input their returned values is made available for the caller in the form of a list.

3.1.7.2 Solving Ambiguities

Most ambiguities are solved by simple grammatical restraints, i.e. most patterns won't match the input, leaving to deal only with the ones that have. The rest of this section will describe three mechanisms for dealing with the remaining types of ambiguities.

The first mechanism for reducing the total number of interpretations is by using Pattern Boolean Expressions to drop the most unlikely interpretations. Let us consider the response of "running away", this response would be linked to patterns that detect danger, such as an imminent attack. However, there are more than one possible response for "running away": If on foot it means actually running, if swimming it means "swim away", and on a car, it would mean "drive away" and so on. In this case, a Boolean check for the presence of water, or a car might be enough to discard some interpretations.

The second type of ambiguities are those that can be solved but would require a complex model to be solved. As an example of this type of ambiguity the phrase: "He drove down the road in his car"; In this sentence, it is not clear whether he drove in his car a road, or if he drove on the road that was inside his car. The second interpretation is quickly dismissed by any human being, for a road is unlikely to fit inside a car. However, this notion is not so clear for a machine, it requires one to be able to understand the concepts of size and space.

For these problems, the human approach is to try to reason about whether each interpretation is feasible by comparing it to an internal model of how the world is. For JSpy model to solve such a problem it would require at least the possibility to implement such models internally and to have them available for the module responsible for dealing with these different interpretations. The design of such models of the real world is not to be discussed here, but it is important to notice that the presence of these models might be the only way to solve such kind of problems.

The third and final method of resolving ambiguities is to rely on heuristics or to ask for help.

With this design, the model is believed to provide enough tools for an implementation to solve simple ambiguities by pattern matching, complex solvable ambiguities by designing internal models suitable for the task and unsolvable ambiguities by any means available. One possible workflow is to wait until the system reports an ambiguity problem and only then teach it how to solve this ambiguity the next time it arises.

3.1.8 Automatic Optimization Mechanics

Since this model is trying to describe the living human brain, there are some considerations about its natural adaptability features. This part of the model, however, is more closely related to Artificial Intelligence techniques than with Natural Language Understanding, and as such, it is not the focus of this study.

This study has, however, provided some insights on the nature of these systems: To create an optimization system over the Pattern Network would probably mean to: (1) create new patterns and connections in a randomized way, (2) prioritize the best patterns and connections, (3) erase patterns that are inefficient, or never used.

As might have become obvious on the last paragraph, this part of the JSpy Model could greatly benefit from studies on Neural Networks, Machine Learning, and Artificial Intelligence. Further studying and describing of these structures will be left as a future work.

3.1.9 Distinguishing Features

The description of the model is now complete, and several important features have been described. In this subsection, they will be organized in a set of nine different features. This features will later be used in Section 4 to provide insights of the applicability of each of them on practical experiments. Most features listed below are already present on the JSpy Language, the two that aren't are explained at the end of this subsection.

1. **Explicit Ambiguity Resolution**
   Ambiguities can never be discarded implicitly by the implementation as described in Subsection 3.1.7. Instead, the solving mechanism should have access to all information available, including internal solution models that might be useful.

2. **Dynamic Pattern Updates**
   It must be possible to add, remove or modify patterns, as well as add and remove them from existing Pattern Groups at any time.

3. **Distributed Processing Feasibility**
   This feature is an important concept: As inefficient as this model might be to im-

plement in a Von Neumann machine the possibility of designing it on a hardware fit for parallel processing might compensate for this problem.

4. **Support for Ad Hoc models**
   In Subsection 3.1.6 it is described how complex processing structures can be built from the pattern concepts alone. The important aspect of this feature is to enforce that any implementation of the model must have access to a Turing Complete system when resolving meaning. This system should be capable of describing specific models to solve specific real-world problems, making possible to ultimately describe meaning and deal with ambiguities appropriately.

5. **Snapshot Polymorphism**
   A pattern must be able to make references to other patterns as well as pattern groups, as described in subsections 3.1.3 and 3.1.4.

6. **Recursive reasoning delegation**
   It must be possible for a recognized pattern to delegate the meaning resolution task to other patterns recursively. This feature is described in Subsection 3.1.6.

7. **Boolean Logic Mechanisms**
   It must be possible for patterns to be associated with Boolean mechanisms for refuting invalid pattern recognitions when the context is enough to make the decision. This feature is described by the model on Subsection 3.1.4.

8. **Represent Data**
   It must be possible to represent different types of data. The model explains this feature as a consequence of different configurations of a pattern network on Subsection 3.1.4, 3.1.5, and Appendix C. However, these pattern based structures can be replaced by equivalent implementations; For example in JSpy Language, the basic data containers are Maps and Lists.

9. **Automatic Optimization Mechanics**
   As described in Subsection 3.1.8 these adaptive features are an important concept of the model. However unexplored in this study, it should provide some interesting results and insights in future studies.

From these features, only features 2 and 9 are not yet implemented in JSpy Language. Feature 2 “Dynamic Pattern Updates” is scheduled to be added to the implementation soon. Feature 9, on the other hand, will require further studies before an implementation can be designed.

3.2 JSpy Syntax and Design

JSpy was designed as a modern scripting language; Its syntax is based on JavaScript and most of its semantics were inspired in Python. Most of the complex aspects of the underlying model were carefully hidden or replaced by surrogates that are more familiar to programmers.

The usual way to use it is to write a script in the JSpy Language and then ask the interpreter to run it; that is, of course after downloading and compiling the interpreter code. An alternative way to use it is by running the interpreter with no arguments: This will open the REPL (Read-Evaluate-Print Loop) mode of the interpreter, allowing the user to write a command and immediately see the result on the screen; useful for testing commands and syntax.

The following sections are divided into two parts: The first one explains briefly how JSpy syntax and semantics were conceived and why, the second one explains in detail the core part of the language designed to interpret and decode patterns from the text, also called the JSpy "*Matcher*" concept. Resources for understanding JSpy in more length are available at **Appendix A** and **B** as well as on the JSpy project's page[3].

3.2.1 Why JSpy? And not an Existing Language?

The JSpy language was originally proposed as a supporting language for the, now called, JSpy Matcher concept. By the time the most important goal of the project was to make it easy to use, and so the language should be simple. As the project developed it was proposed to enhance JSpy Language as a fully featured programming language, where the Matcher concept would fit inside it as a built-in construct. This design would not only make the language more familiar to newcomers, but also make it a lot more powerful, and expressive.

In short, the reason for the JSpy language to be created was because JSpy Matcher required a meaning resolution component that would fully embrace its features. If this tool was adapted to fit inside an existing programming language it would not be possible to benefit from the syntax. Making this tool, that is already complex in nature, to become complex to use, discouraging newcomers from learning it. Nonetheless, studying if the Matcher concept can be adapted gracefully in an existing programming language is included as a future work.

[3]`https://github.com/VinGarcia/JSpy`

3.2.2 JSpy Design Requirements

When designing JSpy there were two guidelines:

- **The language should be comfortable and easy to learn.**
- **It should be as powerful and featured as possible without losing simplicity.**

To better fulfill both requirements JSpy was based on existing programming languages; The reasoning is that the easiest language to learn is the one you already know.

Python was the first choice, for it is designed to keep programmers comfortable and make readable code. But its syntax, however beautiful, is not easy to parse, making it a less desirable choice to implement. Because of that problem JavaScript has been taken into consideration. The final implementation has gathered most of JavaScript syntax and some concepts of it like *prototypical inheritance* and *closures*. In other parts where JavaScript's learning curve is a burden for newcomers, the behavior of the language was modified with concepts considered more intuitive.[4] These more intuitive concepts were mostly inspired by Python.

To learn JSpy syntax and grammar please read **Appendix A**, to learn the differences between JSpy's semantics and JavaScript's, please read **Appendix B**, finally, to test the language in first hand, check the instructions on the project's page.[5] This section will not bother the reader by fully explaining the merits of the JSpy programming language as a whole since the part that actually implements the model is the Matcher construct. Further on, examples using the JSpy Language will be kept simple as to be self-explainable.

3.2.3 JSpy Matcher: Design

The JSpy Matcher construct was designed as the starting point for the entire pattern recognition process, and it effectively implements most of the features described in Subsection 3.1.9; later on this section, these features will be referenced by the name "JSpy Features" for short. These features were first presented in Subsection 3.1.9 as an enumerated list between the numbers 1 to 9. Later this number will be used to

[4] One example of this learning curve burden is that JavaScript has some error prone syntax issues: For example, if the user fails to formally declare a variable before using it, this variable will be declared in global scope, possibly causing undesired side effects.

[5] `https://github.com/VinGarcia/JSpy`

reference specific features of the model, i.e.: JSpy Feature 1 references the first feature while JSpy Feature 9 references the last one.

To describe the JSpy Matcher in the terms of the JSpy Model, the Matcher construct is analogous to the *Pattern Groups* concept described in Section 3.1.4. As such it contains the following features:

1. **Is labeled by an unique name**
2. **It contains a number of patterns inside it**
3. **Each pattern contains an optional callback function, responsible for describing its meaning**

However, JSpy uses a different nomenclature: The Matcher construct is composed of a set of "Hooks", where each hook is a pattern followed optionally by the callback function.

To express these concepts JSpy syntax offers a relatively simple syntax, exemplified on the Matcher below:

```
matcher matcher_name {
  "simple pattern (group_name);" {
    // ... callback code ...
  }
  "more simple";
}
```

This Matcher contains two hooks, the first one makes reference to a named group "group_name". The second one, more simple, contain only literal characters. The second pattern was included in the example to show two things: (1) It is possible for multiple patterns to be declared at once inside a Matcher, and (2) it is possible with this syntax to omit the callback function by replacing it by a single semicolon.

There is also an additional syntax facilitation for cases where the Matcher is expected to contain only a single Hook. In this case, the outer brackets can be omitted as exemplified below:

```
matcher single_hook "simple pattern (group_name);" {
  // ... callback code ...
}
```

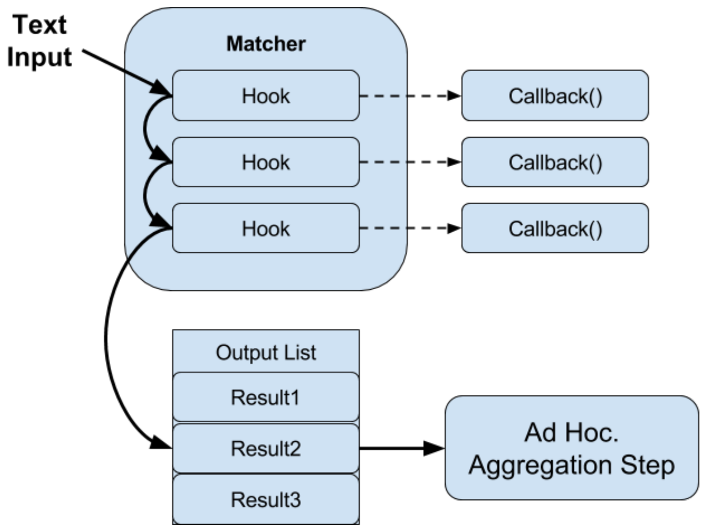

Figure 3.1. Matcher design

A Matcher on JSpy Language is a normal statement, as such it can be declared in any part of the code[6] where a "function" or a "for" loop could be declared. Please take a look at Appendix A for further details on the grammar of the language.

The name of the Matcher, in this case: "single_hook", is made available on local scope, and can later be referenced as an instance of the *Matcher Class*. The interface for this class will be detailed on Subsubsection 3.2.3.4.

Figure 3.2.3 illustrates the matcher design and structure. In the following subsections, different aspects of the matcher will be explained in details to provide a deeper understanding of the potential of this construct.

3.2.3.1 JSpy Patterns

JSpy Patterns are very similar to normal Regular Expressions (RegExps) with the visible exception of the references to the named groups between brackets, that do not exist in normal RegExps, at least not with the same meaning. These patterns are capable of representing all the same concepts of RegExps mostly with the same syntax.[7] A short explanation of the concepts involving these patterns will follow with illustrative examples.

[6] This is not entirely true, there is currently a bug that might cause unexpected behavior if a Matcher is declared inside a function or nested in another Matcher's callback.

[7] These patterns were designed and implemented more than 3 years ago and were not updated to meet the design standards used on the rest of the language: their syntax was not based on either Python nor JavaScript regular expressions, making them a little bit harder to learn and less comfortable to use. A review of this syntax is scheduled for future versions.

Some basic RegExps functionalities such as describing classes of characters between square brackets and using the Kleene Star operator for describing repetition work as expected:[8]

```
"[Nn]ame"
 matches the text: 'name' or 'Name'.


"[1-9][0-9]*"
 matches any integer number not starting with 0.
```

It is also possible to group together characters using round brackets as in normal RegExps, however it requires a special syntax to make it different from named group references:

```
"("00")*"
 matches any text with an even number of zeros
```

Some RegExp features are, however, not yet implemented in these expressions. As a consequence the disjunction operator is not expressed by the usual pipe operator "|" instead it is represented only inside the round brackets notation and using a comma to separate each item:

```
"("first", "second");"
 would match either word, 'first' or 'second'
```

The same applies to named groups, more of one of them can be represented between round brackets as a disjunction, even sharing the brackets with quoted patterns:

```
"("first", named_group, "second");"
```

This feature is of special importance for it implements JSpy Feature 5, allowing a pattern to be composed of other pattern or pattern groups.

The '+' operator and the '?' operator are also not present in this implementation yet. The main reason was because they are not indispensable. They are, however, expected to be present in future versions of the language.

One extra feature not yet mentioned of these patterns is the possibility of capturing named groups. To do so, it is only necessary to specify a variable name for a round bracket construct:

[8]There is a small issue regarding the Kleene Star operator, but it will be discussed together with other bugs and issues on the end of this section.

```
"("first", "second")captured_value;"
```

Whichever value that will be matched by the either of the patterns between brackets will be saved on the variable “captured_value”, and it will be made available inside the callback function.

3.2.3.2 Hook's Callbacks

A hook's callback is the function attached to its pattern. These callbacks have a set of arguments described by the named capturing groups of the pattern and are capable of returning a value. In the example below the callback function will receive two arguments: arg1, and arg2. Then it will return “True” if both arguments' values are the same:

```
matcher m "is this (word)arg1; equal to (word)arg2;?" {
  return arg1 == arg2
}
```

This design was chosen to facilitate the processing of the information extracted from the text. The syntax used inside the callback function is the JSpy Language syntax, accepting any kind of valid JSpy statement or expression.

The presence of normal JSpy code inside this callbacks is important for it allows the meaning of the pattern to be resolved using any solution model that JSpy is capable of describing. This effectively implements JSpy Feature 4.

The callback function's scope is hierarchically subject to the scope where the Matcher was declared and to the scope where it is being executed. As such it also has access to variables declared on these scopes, even if some of them were declared after the Matcher declaration. In short, this scope hierarchy was designed to work just like in JavaScript functions.

This scope hierarchy allied to the fact that the Matcher name is made available on the scope it is declared allows for a callback to recursively execute its own Matcher or even other Matchers it has access to. This effectively implements the JSpy Feature 6, allowing the meaning of a pattern to be resolved through recursive pattern execution calls.

3.2.3.3 Hook's Return Values and Explicit Ambiguity Resolution

It is important to note that when a hook is executed it is possible to result in more than one interpretation of the text. The toy example below illustrates this possibility:

```
matcher m "("tes","te")val1;("t","st")val2;" {
  return list(val1, val2)
}
```

This example pattern would match the string “test” twice: One of the matches would assign “tes” to *val1* and “t” to *val2*, the other one would assign “te” to *val1* and “st” to *val2*.

Ambiguous situations, as stated by the JSpy Feature **1**, are expected by the JSpy Model. Thus, the solution for this problem is explicit and not complicated: the callback will be executed twice and the return values of these callbacks will be grouped, resulting in a list of the returned values such as this:

```
[ ["tes", "t"], ["te", "st"] ]
```

It is also possible for the callback to deny the recognition of the pattern, which is useful in eliminating invalid ambiguities when enough information is already available.

This feature is the simpler way JSpy found to implement JSpy Feature 7: The possibility to revoke a recognition based on a Boolean evaluation of the current context.

To exemplify this feature the example below will revoke the match if *val1* does not equal the string “tes”:

```
matcher m "("tes","te")val1;("t","st")val2;" {
  if (val1 == "tes") {
    return list(val1, val2)
  } else {
    // Explicit return None to deny a match:
    return None
  }
}
```

As a consequence if we executed the matcher above with the text “test” it would this time return only one value:

```
[ ["tes", "t"] ]
```

3.2.3.4 Calling and Executing Matchers

Matchers are a construct of the JSpy Language, and as such, they can be called from within the language. A common workflow is to declare a set of matchers and functions, read an input file and then execute a matcher designed as the "root" for each line of that file.

Matchers can be invoked in a total of four different ways. The matcher below will produce a total of three results for each time it is matched with the word "test", and it will be used to exemplify the different ways of calling a matcher.

```
matcher ambiguous {
  "("tes","te")v1;("t", "st")v2;" {
    return 'hook1: ' + v1 + '-' + v2;
  }

  "test" {
    return 'hook2: ' + text;
  }
}
```

If one wants to handle all the different interpretations and then choose the correct one manually he should invoke the matcher with the "match_all" function:

```
result = ambiguous.match_all('test')
```

The content of the *result* variable would then contain a list with three strings as shown below:

```
[ "hook1: tes-t", "hook1: te-st", "hook2: test" ]
```

Other forms of calling the matcher include:

- **"match_one('test')" to output only the first match, e.g.: "hook1: tes-t"**

- **"match('test')" to output a boolean indicating true for at least one match and false otherwise.**

- **"count('test')" indicating how many matches have been found, or 0 if none was.**

3.2.4 JSpy's User-friendly Syntax and Features

JSpy is made to support the Matcher concept. Thus, its syntax and features were created to be powerful and easy to learn. While the syntax was mostly inspired in JavaScript, the features of the language are closer to Python. Hopefully, this language will feel familiar to both Python and JavaScript users. The key features of the JSpy language are:

- Strongly-typed dynamic variables (like Python)
- Built-in dictionaries, lists, strings and related functions.
- Iterators, if statements, for-range loops, and functions.
- Inheritance with prototypes implemented to look and work like classical inheritance.

3.2.5 JSpy's Known Issues

There are also some issues that are not yet fixed or implemented. Most of them are related to the JSpy Pattern syntax and implementation, since these were designed a long time ago, and are in need of a review.

Among these pattern specific bugs and issues we can list:

- **The "*" operator does not match an empty string as it should.**
- **Calling the "*" operator over a round bracket group might cause a segmentation fault.**
- **The overall syntax is unrelated to either Python or JavaScript's Regular Expressions, complicating the learning curve.**
- **Characters escaping is not being treated correctly, for example, the TAB character denoted as "\t" fails to recognize the actual TAB character when it is received as input for the pattern.**
- **Declaring Matcher in anywhere but on the most external scope of the program may cause unexpected behaviors.**

The rest of JSpy Language, on the other hand, is more robust; And contain only two issues to be pointed out:

- **It is not prepared to deal with UTF-8 character encoding yet, only ASCII.**
- **The error messages of the system are not yet very clear, making it hard to determine the actual line of the bug.**

Furthermore, there are some features that are scheduled to be implemented and might deserve some attention:

- **When a matcher is referenced inside a pattern, the return value of the referenced matcher is not accessible to the pattern's callback.**
- **It is not possible yet to add or remove hooks from an existing Matcher.**
- **There are some built-in functions that are not available yet, but should be in the final implementation.**
- **The type "boolean" does not exist yet. Boolean values are instead treated as integer values where 0 means false and anything else means true.**

Chapter 4

Experiments

The experiments described in this chapter were used to show the applications of the JSpy Features on different problems specifically designed to test different skill sets for psycholinguistic tasks.

The scripts and data discussed in this chapter are available on this address: `github.com/VinGarcia/bAbI`. The instructions to download and run the project are available directly on the project's page.

4.1 Data

The dataset we used for this evaluation is called bAbI, and it was obtained from `http://fb.ai/babi`. The data is split into a number of tasks, and each task comprises a series of facts and questions. All of the questions are noiseless and a human able to read that language can potentially achieve 100% accuracy.[1] Questions within each task demand a different skill-set to be answered.

In this experiment we have attempted to solve 17 from the 20 bAbI tasks, leaving out only tasks 18, 19 and 20. The list below describe each of the tasks we used:

- **Multiple supporting facts**
  This skill is tested by tasks 1, 2 and 3. It consists of questions where information has to be extracted from a number of supporting statements to answer the question. So, to answer the question "Where is the apple?", one has to combine information from two sentences "John is in the office" and "John picked up the apple".

[1]That is true to almost all tasks, however, tasks 2, 5, 14 and 16 present some few questions regarding insufficient information (e.g. ask where john was before informing it), but these are quite rare being restricted to less than 3% of the questions.

- **Multiple argument relations**
  This skill is tested by tasks 4 and 5. It consists of questions where the order in which the words appear in the question is crucial to their meaning. For instance, the questions "What is south of the bedroom?" and "What is the bedroom south of?" have exactly the same words, but a different order, with different answers.

- **Yes/no questions**
  This skill is tested by task 6 and consists of the capacity of the program to answer true or false to simple questions with a single supporting fact.

- **Counting and Listing**
  This skill is related to processing a set of items as answer. Task 7 tests if the program is capable of counting the number of valid answers, Task 8 tests if the program is capable of displaying the answers in the form of a list, e.g. "apple, journal, football".

- **Simple Negation and Indefinite Knowledge**
  This is tested by tasks 9 and 10. The former tests if the program is capable of effectively comprehending simple negation such as: "Fred is no longer in the office". While the later tests the capacity of expressing doubt when not sure about the answer. This doubt is tested in terms of indefinite sentences like "John is either in the office or in the kitchen".

- **Basic coreference**
  This skill is tested by task 11. It consists of facts as "John was in the office. Then he went to the studio". To answer questions about facts like this, it is crucial to detect that "he" is a reference for "John".

- **Conjunctions**
  This skill is tested by task 12. It consists of multiple subjects within a single sentence. For instance: "Mary and John were in the office".

- **Compound coreference**
  This skill is tested by task 13. It consists of sentences where the pronoun can refer to multiple actors. For instance: "John and Mary went to the office. Then *they* went to the garden".

- **Time Reasoning**
  This skill is tested by task 14. It tests the capacity of understanding time expressions such as: "John went to the cinema *yesterday*", or "*This afternoon* Mary

traveled to the office". The questions of this task, inquire about the order of the events: "Where was Mary *before* the park?".

- **Basic deduction and induction**
  These skills are tested by tasks 15 and 16. They consist of answering questions that require the use of deductive or inductive reasoning by means of inheritance of properties. For instance: "Sheep are afraid of wolves. Mary is a sheep. What is Mary afraid of?".

- **Positional reasoning**
  This skill is tested by task 17. It consists of reasoning about the position of objects. For instance, the statement "The red sphere is to the right of the blue square." might be followed by a question in the format: "Is the red sphere to the right of the blue square?".

4.2 Reasoning-based Implementation and Evaluation

We implemented a system prototype on top of JSpy using the bAbI dataset described above. In this section, we explain how JSpy handled questions within each task.

4.2.1 Ontology

To answer questions within each task, a simple ontology was built using Matcher instances to describe all concepts required by the task. The most basic definitions described how to recognize "names", "words" and "numbers" and were required by all tasks in the bAbI dataset.

Furthermore, concepts as groups of synonyms and groups of names belonging to a certain class were defined using Matcher instances with one Hook for each name and no Callbacks. In order to better illustrate this, the following code snippet was extracted from the ontology of one of the JSpy solutions:

```
matcher number "[0-9]*";
matcher word "[a-zA-Z]*";
matcher name "[A-Z][a-z]*";

matcher move {
  "moved";
```

```
    "journeyed";
    "went back";
    "went";
    "travelled";
  }
```

This ontology was inserted *manually* into the system, and there is no available way yet to produce this type of ontology automatically from online resources. The reason for that is the strong relation between JSpy and its model for the human brain: Humans do not learn thousands of concepts in an instant, and neither does the model. However, this is not the same as saying it is impossible for the model to make use of online resources as WordNet and other thesaurus, but preparing the system to receive them is a task for future works.

4.2.2 Loading and Parsing

For parsing the training data associated with each task, it was first needed to read each file and separate it into stories. Each story start is identified by a line starting with the number 1, e.g.: "1 Mary moved to the bathroom.". The stories were then parsed and executed one by one. After each parse, a "reset()" function was called to erase all data extracted from the previous story.

To solve each of the different tasks the process used was similar: (1) first the types of statements of the story were identified, (2) a Hook Expression was then created to match them and extract relevant information (e.g. names) and finally (3) a Callback was designed for each hook to update an internal model. When the question statement was asked, this model was then used as the source for obtaining the answer. To illustrate this process, consider a task with two types of statements:

- **A movement statement:** "1 Mary went to the office"
- **A question statement:** "2 Where is Mary? R: office"

For parsing it a Matcher instance was created with two Hooks, one for updating the last position of each actor, and the other to compare the model answer with the actual correct answer extracted from the text. The code snippet below displays the JSpy syntax used to describe this parser:

```
  matcher read {
    "(number); (name)n; (move); to the (place)p;" {
        where_is[n] = p;
      }

      "(number); Where is (name)n;? R: (word)A;" {
        guess = where_is[name]
        compare_results(guess, A);
      }
    }
```

4.2.3 Solutions to bAbI Tasks

In this subsection, we will explain the model used to solve some of the tasks, and then list which of the nine JSpy Features cited on Subsection 3.1.8 were required to solve the problem. From these features two were used on every task, they were:

- **Support for Ad Hoc models:** Since every task required a solution model.
- **Snapshot Polymorphism:** Since every pattern used for parsing contained references to patterns used to describe the ontology.

Supporting facts (tasks 1 to 3) The first three tasks from the bAbI dataset were about answering a question based on a number of supporting facts. The third task was the more complex one requiring a total of three supporting facts. Since all three tasks are similar we will focus only on the third to illustrate the process and the challenges. This task consisted of four different types of statements implicating in four different Hooks:

- **A move statement:** "3 John moved to the bathroom."
- **A pick up item statement:** "4 John grabbed the football."
- **A drop item statement:** "5 John left the football."
- **A question statement:** "Where was the football before the bathroom? R: toilet"

The implementation to answer questions within this task only needed to track the inventory of each actor with a Dictionary, and to track a history of the position of each object with a vector. Then, when an item was moved from some place to another by an actor, it was just necessary to save this new position on the vector. Answer the question was then possible by iterating backwards through the respective item vector.

Two Argument Relations (task 4) The fourth bAbI task required reasoning about relations between environment objects, more precisely: positional relations. It was used a total of three types of statements from which two were questions:

- **A relation statement:** "3 The office is north of the kitchen."
- **A direct question statement:** "4 What is north of the garden? R: toilet"
- **An indirect question statement:** "5 What is the garden north of? R: toilet"

In this task a special concept of opposition between north and south, east and west needed to be added to the ontology. This concept was inserted directly into the ontology, using the callback feature of the patterns of all four directions to add more information to the matched string:

```
matcher direction {
  "north" return { 'text': 'north', 'opposed': 'south' };
  "south" return { 'text': 'south', 'opposed': 'north' };
  "east" return { 'text': 'east', 'opposed': 'west' };
  "west" return { 'text': 'west', 'opposed': 'east' };
}
```

This special usage of the callbacks of the Matcher "direction" required the JSpy Feature 6: Recursive Reasoning.

To solve this task, each place was represented by a class instance that kept track of its neighbors in all four directions. Answering the questions consisted of checking the neighbor at the cited direction or the neighbor in the opposed direction in the case of the indirect question statement.

Three Argument Relations (task 5) This task was built around the relation of giving items; Each time an actor give an item for someone else there were three names involved: The giver, the taker and the object that was given/taken. There were a total of six relevant statements in this task, one of which was the "giving" statement: "(name)n1; (give); the (object)obj; to (name)n2;", and the other five were *questions*. These questions regarded the three terms of the giving relation in different ways:

1. "What did (name)n1; give to (name)n2;?"

2. “Who received the (object)obj;?”

3. “Who gave the (object)obj;?”

4. “Who did (name)n; give the (object)o; to?”

5. “Who gave the (object)obj; to (name)n;?”

To solve this task it was only required to record every transaction between characters in a list, and when asked a question to search the most recent matching transaction backwards on this list.

Yes/No Questions (task 6) This task dealt with the same problem of task 1: Actors moved on the environment, and then it was asked where they were at the end. The questions, however, were a little different, instead of asking *where* the actor was, it asked if he was in a specific location, so the correct answer would be “Yes” or “No”.

To solve this task the model used was very simple: A dictionary keeping track of the last known place each actor has been seen, where the key of the dictionary is the actor’s name and the value the location. To answer the Yes/No questions it was only required to compare the location specified on the question with the current location on the dictionary, and answer “Yes” if they were the same and “No” otherwise.

Basic Coreference (task 11) The next bAbI task dealt with coreference in a simple context, where after one actor has moved from someplace to another place a new statement referred to him as “he” or “she”. To solve this task it was necessary to add the concept of male and female names so that the four actors of the story were divided as two males and two females. These stories were told using a total of three types of statements:

- **A move statement:** “3 Mary went to the office.”

- **A coreference statement:** “4 Then *she* moved to the garden.”

- **A question statement:** “5 Where is Mary? R: garden”

To solve this task the JSpy Matcher used the JSpy Feature 6: Recursion Reasoning. The Callback of the coreference statement was responsible only for identifying the reference and then replacing “he” or “she” for the proper name of the actor. After that, it would feed back the system with a new string such as: “*Mary* moved to the bathroom”. This solution effectively decouples the task of identifying the actor and the task of resolving the movement of the actor, shortening the number of lines required and simplifying the code.

Compound Coreference (task 13) This bAbI task involves the skill sets for identifying conjunction *and* coreference. And as so it was composed of three types of statements:

- **A move statement:** "3 Mary *and* John went to the office."
- **A coreference statement:** "4 Then *they* moved to the garden."
- **A question statement:** "5 Where is Mary? R: garden"

But for simplicity, we reutilized the Hooks from the basic coreference task. Now instead of "he" or "she" we kept track of the pronoun "they" and saved the last seen actors in a vector container.

To solve the coreference we still used the recursive approach: Each coreference sentence like "They moved to the office" was recursively reevaluated with proper names of the characters: "Mary and John moved to the office".

Interpreting the conjunction concept was then facilitated again by the use of recursion: The names of the actors involved in the conjunction were extracted from the text, and then a new simpler sentence was produced, such as: "Mary moved to the office" and "John moved to the office". Please note that this approach would work even if there was a list of several actors involved in the conjunction, greatly facilitating the process.

Basic deduction (task 15) This bAbI task was about deduction and there was a total of three different types statement:

- "Mice are afraid of Cats."
- "Gertrude is a mouse."
- "What is Gertrude afraid of?"

To answer questions in this task we needed to add the concept of plural and singular to the ontology, so that "Mice" was related to "mouse" and "Wolf" to "Wolves". This was possible by adding that information on the Matchers responsible for describing them on the ontology:

```
matcher animal {
  "[Mm]ice" return 'mouse';
  "[Ww]olves" return 'wolf';
  "[Cc]ats" return 'cat';
  "[Ss]heep" return 'sheep';
}
```

This model for “animal” was possible by using JSpy Feature 6: The Recursive Reasoning.

Further, to solve the logical deduction part we have used the prototypical inheritance feature: A species was defined as an object and an animal as an object whose prototype was its species. So the attribute “afraid_of” of each animal was stored in its super class, making it trivial to answer the question statements.

Basic induction (task 16)

The induction task was similar to the deduction one, but this time it was necessary to guess the color of an animal given the knowledge of other animals of his same species.

To store this knowledge it was necessary to save information about the species on a prototype and to instantiate each animal as a child object of this prototype. When an animal was said to have a certain color, this information was stored on its prototype. Consequentially when the model was asked the color of an animal all it had to do was to search the color of its species.

Positional reasoning (task 17)

This task tests the problem of relative positions of geometric objects, and is composed of only two types of statements:

- **A relative positional statement:**
  “The blue square is to the left of the triangle.’
- **A question about relative positions of objects:**
  “Is the pink rectangle to the right of the blue square?”

This task, as well as task 4, dealt with the problem of positional reasoning. But in task 4 the questions regarded only the immediate neighborhood of the objects. As such if object A was to the left of object B and B was to the left of object C, it was *not* required for the program to comprehend by transitivity that the object A must also be to the left of object C. In this task, however, it is necessary, and since there are no absolute coordinates for any given object the only way to identify this type of transitive relationship is by performing a Breath First Graph Search.

To solve this problem, given its extra complexity it was possible to apply JSpy Features in new interesting ways. For example, each geometrical form described by the problem might optionally be associated with a color, e.g. “The *blue* square”. For be possible for the Matcher on the ontology section of the code to represent this information it was necessary for it to make a reference to other Matchers on the ontology. The result works much like a grammar:

```
matcher form {
  "(color); (form);";
  "rectangle";
  "square";
  "triangle";
  "sphere";
}
```

Furthermore to denote the concept of opposed directions and the concept of relative positions as Cartesian coordinates it was required for the respective Matcher to return all this information. The resulting structure of the Matcher is shown below:

```
matcher position {
  "below" return {
    'pos': 'below',
    'oppose': 'above',
    'offset': {'x': 0, 'y': -1}
  };

  "above" return {
    'pos': 'above',
    'oppose': 'below',
    'offset': {'x': 0, 'y': 1}
  };

  "to the right of" return {
    'pos': 'right',
    'oppose': 'left',
    'offset': {'x': 1, 'y': 0}
    };

    "to the left of" return {
      'pos': 'left',
      'oppose': 'right',
      'offset': {'x': -1, 'y': 0}
    };
  }
```

To solve the problem a model was used to search part of the graph and update the relative positions of each object every time a new relation was parsed. This solution although not optimal was still capable of obtaining a result superior to those of the baselines as shown on 4.1 on the end of the next subsection.

It is important to note that it is perfectly possible to implement the Breath First Search using JSpy, and thus obtaining an optimal algorithm to solve this task. It was not done for time restraints.

4.2.4 Evaluation

In this section, we report the results of the evaluation of the proposed JSpy framework. First, we present the baselines used for comparison. Then, we discuss the results.

Baselines The n-gram classifier baseline is inspired by the baselines in Richardson et al. [2013], but applied to the case of producing a 1-word answer rather than a multiple choice question. We also used a structured SVM (Support Vector Machines) Joachims et al. [2009], which incorporates coreference resolution. Another baseline is LSTM Sutskever et al. [2014] (long short term memory Recurrent Neural Networks) which works by reading the story until the point they reach a question and then have to output an answer. Finally, we also provide performance comparison against state-of-the-art machine learning memory networks Sukhbaatar et al. [2015].

Results Table 4.1 shows the results in terms of accuracy, that is, the fraction of correct answers provided by the system[2]. JSpy solutions performed extremely well on all tasks, offering superior results than the baselines. It is important to notice, however, that JSpy solutions are specialized, requiring applied knowledge and understanding of the problem. Still, Table 4.2 shows the amount of lines required to produce QA solutions

[2]Baseline results were obtained directly from Weston et al. [2015].

using JSpy as to offer a metric for the complexity of the Natural Language skill required. This complexity is divided into three steps, namely: ontology, model (reasoning), and parsing. It is clear that JSpy solutions are simple due to the small number of lines required to implement them.

Task	n-gram	LSTM	Struct. SVM	Mem. Nets	JSpy
1 - Single supporting fact	0.36	0.50	0.99	1.00	1.00
2 - Two supporting facts	0.02	0.20	0.74	1.00	0.98
3 - Three supporting facts	0.07	0.20	0.17	0.20	1.00
4 - Two argument relations	0.50	0.61	0.98	0.71	1.00
5 - Three argument relations	0.20	0.70	0.83	0.83	0.99
6 - Yes/no questions	0.49	0.48	0.99	0.47	1.00
7 - Counting	0.52	0.49	0.69	0.68	1.00
8 - Lists/sets	0.40	0.45	0.70	0.77	1.00
9 - Simple negation	0.62	0.64	1.00	0.65	1.00
10 - Indefinite knowledge	0.45	0.44	0.99	0.59	1.00
11 - Basic coreference	0.45	0.72	1.00	1.00	1.00
12 - Conjunction	0.09	0.74	0.96	1.00	1.00
13 - Compound coreference	0.26	0.94	0.99	1.00	1.00
14 - Time reasoning	0.19	0.27	0.99	0.99	0.97
15 - Basic deduction	0.20	0.24	0.96	0.74	1.00
16 - Basic induction	0.43	0.23	0.24	0.27	0.99
17 - Positional reasoning	0.46	0.51	0.61	0.54	0.78

Table 4.1. Accuracy on bAbI tasks for different methods.

Task	Ontology	Model	Parsing	Total
1 - Single supporting fact	20	1	10	31
2 - Two supporting facts	41	28	53	122
3 - Three supporting facts	41	31	48	120
4 - Two argument relation	19	24	18	61
5 - Three argument relations	47	26	61	134
6 - Yes/no questions	20	2	11	33
7 - Counting	46	13	49	108
8 - Lists/sets	40	29	36	105
9 - Simple negation	20	2	22	44
10 - Two argument relation	23	23	18	64
11 - Basic coreference	33	8	25	66
12 - Conjunction	20	1	16	37
13 - Compound coreference	27	2	24	53
14 - Time reasoning	30	10	25	65
15 - Basic deduction	10	7	19	36
16 - Basic induction	17	15	18	50
17 - Positional reasoning	45	54	38	137

Table 4.2. Number of lines for programming solutions to each bAbI task.

Chapter 5

Conclusions

It was described in this work the JSpy Model and the JSpy Programming Language. JSpy Model is an objective implementation of how the brain organizes itself in terms of Pattern Parsing Systems. One strong aspect of this model is that it refrain itself from making unnecessary new assumptions about how the brain works, and show how the few assumptions used are capable of describing a complex system. JSpy Programming Language in the other hand was designed as an implementation of this model. The most strong aspect of this implementation is the attempt to convert the complex features of the model into well defined and comprehensible architectural features of the language. Hopefully, this design will make the Language more accessible for new users.

The experiments performed demonstrated that the language is capable of solving the proposed problems in few lines of code and with good results. Also, it was useful for illustrating a concrete example of how the features of the JSpy Model would fit for solving different aspects of Natural Language Problems, such as Conjunction, Coreference, and Basic Deduction.

5.1 Future Works

Several future works are planned, some were already mentioned others are first presented below:

- **Further studying the automatic optimization features:**
  Since JSpy Model is a model for the human brain, it is natural to expect it to contain an optimization mechanism present on its architecture. The nature of this mechanism was not discussed here, but it is likely that it would have some resemblances with the way Neural Networks learn. Also if the JSpy Model is

correct, and it, in fact, represents an underlying architecture of the brain, it would be expected for the patterns networks described to emerge as a natural outcome of these optimization mechanics.

- **Study if the model can be fully implemented in other programming languages**
  The reason for creating JSpy Language instead of implementing the model inside a popular programming language like Python was to be sure that the syntax of the language would embrace the model instead of imposing an obstacle for it. However, now that the first implementation offers a concrete model of how an implementation of this model should look like, it is the right moment to study how many of the important features of JSpy Language could be implemented in other programming languages.

- **Review the JSpy Pattern's syntax:**
  The current syntax, and also the implementation of the patterns used by the JSpy Language are not a strong asset of the implementation: The syntax is not very simple and it is not based on JavaScript or Python's regular expressions. Also, most of the bugs of JSpy are caused by implementation problems on this part of the code.

- **Study possible methods for collecting information for the ontology automatically instead of manually:**
  Currently, the only way to add new information to the ontology is manually. This is not acceptable if the goal is to fully comprehend Natural Language. To fill this ontology automatically there are two possibilities: either the language will have to be implemented with some automatic learning mechanism or it will need to be able to harness data from available online resources. The latter option is likely the most feasible, and should probably be pursued first.

- **Study in depth the latent properties of the JSpy Model:**
  If the JSpy Model, in fact, describes an underlying mechanism in the human brain, it is likely to provide us with insights about other structures and processes of the brain. An example of such task is to understand how the Pattern structures described could be used to explain the learning of Generalization and Specialization by human children, or how the model could describe different types of information such as images and sounds.

- **Build an experimental comparison between the model and the human brain**
  If the JSpy Model truly describes an internal structure of the human brain, it should also suffer from the same performance issues and be capable of taking advantage of the same optimizations as the human brain. The model should also be able to explain some of the peculiarities observed in the performance of humans when parsing grammar as described by Johnson [2004] as the "Evidential Basis for Syntactic Theory".

- **Study different architectures to implement the model**
  Since the model was inspired on the human brain, implementing it on a Von Neumann machine might not be the most efficient approach. It might be interesting to consider the design of an architecture specialized on implementing this model.

- **Prove the Turing Completeness aspect of the model**
  It is believed that the JSpy Model using only the patterns to build structures and algorithms has the same expressive power as a Turing Machine. This proof was not formally done yet but it is an important step.

Bibliography

Ball, J., Sc, B., Sc, M. C., and Ball, J. (2012). Patom Theory.

Bates, M. (1978). The theory and practice of augmented transition network grammars.

Bobrow, D. G. (1964). *Natural Language Input for a Computer Problem Solving System*. PhD thesis, Massachusetts Institute of Technology.

Covington, M. A. (1994). *Natural Language Processing for Prolog Programmers*. Prentice Hall.

Goldberg, Y. (2015). A primer on neural network models for natural language processing. *CoRR*, abs/1510.00726.

Gong, L. (2003). Intelligent personal assistants. US Patent App. 10/158,213.

Graham-Rowe, D. (2007). A working brain model. `https://www.technologyreview.com/s/409107/a-working-brain-model/`.

Joachims, T., Hofmann, T., Yue, Y., and Yu, C. J. (2009). Predicting structured objects with support vector machines. *Commun. ACM*, 52(11):97--104.

Johnson, K. (2004). Introduction to Transformational Grammar. *The Modern Language Journal*, 63(1/2):68. ISSN 00267902.

Kaplan, R. M. (1972). Augmented transition networks as psychological models of sentence comprehension. *Artificial Intelligence*, 3(1972):77--100. ISSN 00043702.

Nierhaus, G. (2009). *Transition Networks*, pages 121--130. Springer Vienna, Vienna.

Noormohamadi, R. (2008). Mother Tongue, a Necessary Step to Intellectual Development. *Pan-Pacific Association of Applied Linguistics*, 12(2):25--36. ISSN 13458353.

Richard S. Sutton, A. G. B. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Richardson, M., Burges, C., and Renshaw, E. (2013). Mctest: A challenge dataset for the open-domain machine comprehension of text. In *EMNLP*, pages 193--203, Seattle, Washington, USA.

Sukhbaatar, S., Szlam, A., Weston, J., and Fergus, R. (2015). Weakly supervised memory networks. *CoRR*, abs/1503.08895.

Sutskever, I., Vinyals, O., and Le, Q. (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104--3112, Montreal, Canada.

Valin, R. D. V. (2016). From NLP to NLU.

Weston, J., Bordes, A., Chopra, S., and Mikolov, T. (2015). Towards ai-complete question answering: A set of prerequisite toy tasks. *CoRR*, abs/1502.05698.

Wikipedia (2017). Regular expression — wikipedia, the free encyclopedia. [Online; accessed 26-January-2017].

Wolfram Research, I. (2016). Mathematica.

Woods, W. A. (1970). Transition network grammars for natural language analysis. *Communications of the ACM*, 13:591--606. ISSN 00010782.

Appendix A

JSpy Grammar

The JSpy grammar was based on JavaScript's grammar and as such they are very similar with the noted minor differences:

- The name "matcher" is a reserved key-word for JSpy, but it has no meaning for JavaScript.
- There is a new construct called "Matcher" that only exists in JSpy.
- C-like "for" statements (e.g. *for(i=0; i<n; ++i)*) do not exist in JSpy; The only "for" loop available is the "for-in" loop: *for(name in expression)*
- It is not possible (or necessary) to write "var" inside a for loop like it is in JavaScript. This means that the following construct would represent a syntax error: "*for(var name in expression)*"
- Some constructs are not yet implemented, but will in the future. For instance "*try−catch−finally*" and the "*switch*" constructs are among these.

Having said that, the grammar follows below. Please note that between any two symbols of the grammar a sequence of zero or more white-space characters is acceptable, but this will not be made explicit on the grammar for simplicity.

A.1 JSpy Statements Grammar:

This first topic will focus on the grammar regarding the JSpy Statements, it will provide a good overview of the language. The next two sections will explain the symbols: "*expressions*" and "*JSpyRegEx*" That are very complex and as such should be explained in separate.

```
stmt → for−stmt | if−stmt | while−stmt |
       var−stmt | function−stmt | matcher−stmt |
       '{' block−stmt '}' | expression

delimiter → ';' | '\n'

var−name → [a−zA−Z_][a−zA−Z0−9_]*

block−stmt → stmt delimiter block−stmt | λ

for−stmt → 'for' '(' var−name 'in' expression ')' stmt

if−stmt → 'if' '(' expression ')' stmt |
          'if' '(' expression ')' stmt 'else' stmt

while−stmt → 'while' '(' expression ')' stmt

var−stmt → 'var ' var−name |
           'var ' var−name '=' expression |
           'var ' var−name ',' var−stmt |
           'var ' var−name '=' expression ',' var−stmt

function−stmt → 'function' var−name '(' arguments ')'
                '{' block−stmt '}'

arguments → var−name | var−name ',' arguments

matcher−stmt → 'matcher ' var−name '{' hook−list '}' |
               'matcher ' var−name hook−decl

hook−list → hook−decl hook−list | λ

hook−decl → '"' JSpyRegEx '"' ';' |
            '"' JSpyRegEx '"' '{' block−stmt '}'
```

A.2 JSpy Expression's Grammar:

Within the expression grammar, there are some operators that do not exists in JavaScript. They were inspired by Python and implemented accordingly such as the power operator (**), the tuple constructor operator (,) and the formatting operator (%)

```
expression → token | token op token

op → '+' | '-' | '*' | '/' | '**' | ',' | '%'

token → string | int | real | reference | literal |
        '(' expression ')' | function-call

string → '"' ([^"\n] | '\"')* '"'
         ''' ([^'\n] | '\'')* '''

int → [+-]? [0-9]+

real → [+-]? [0-9]*\.[0-9]+ |
       [+-]? [0-9]+\.[0-9]*

reference → var-name |
            reference '.' var-name |
            reference '[' expression ']' |
            reference '[' string ']' |
            reference '[' int ']'

var-name → [a-zA-Z_][a-zA-Z0-9_]*

literal → 'None' | 'True' | 'False' | inline-function

function-call → inline-function '(' expression? ')' |
                reference '(' expression? ')'

inline-function → 'function' '(' arguments ')'
                  '{' block-stmt '}'
```

```
arguments → var−name ',' arguments | var−name | λ
```

A.3 JSpy Advanced Regular Expression's Grammar:

The JSpy Regular Expressions were designed as a superset of normal regular expressions. In order to simplify the grammar the symbol "RegEx" will denote JavaScript Regular Expressions with one difference: The "(" symbol can only appear after a backslash e.g.: "\(", as to avoid ambiguity in the grammar.

Please note that, as this document was written, the parser for these expressions still presented some bugs and unwanted complexity, and an architectural review is being planned. This review might change drastically the overall syntax of these expressions, but for now, they are as described bellow. Also differently from the rest of the grammar spaces are not ignored, and will be considered part of the expression from now on:

```
JSpyRegEx → matcher−exp | matcher−exp JSpyRegEx

matcher−exp → RegEx |
              '(' disjunction−list ')' var−name ';'
              '(' disjunction−list ')' var−name '*'

disjunction−list → matcher−reference ',' disjunction−list |
                   matcher−exp ',' disjunction−list |
                   matcher−exp

matcher−reference → var−name
```

Appendix B

JSpy vs JavaScript Semantics

JSpy, as the name implies, was inspired by two popular scripting languages: JavaScript and Python. While most of the syntax was gathered from JavaScript, the semantics were inspired by Python to be easier to learn, understand and harder to introduce bugs.

The main changes are listed below.

B.1 For-in Loops

The "*for-in*" loops implemented by JSpy use JavaScript syntax but its behavior was based on Python. The most important difference between these 2 implementations is the behavior when the loop is iterating over a list:

```
// JavaScript:
for(var i in ['a', 'b', 'c']) {
  console.log(i) // Would print: 0 1 2
}

// JSpy:
for(i in ['a', 'b', 'c']) {
  print(i) // Would print: a b c
}
```

B.2 Iterators and Generators

In JavaScript until the ECMAs Script 5, generators and iterators were undefined. In ECMAs Script 6 it was finally defined, but the syntax differs from the one described here.

Iterators are a concept designed to be used on *for-in* loops. These loops are capable of iterating over four data types: Lists, Maps, Strings and *Iterators*. Some built-in functions of the language return Iterators, such as the “list.reverse()” function:

```
for(item in vector.reversed()) {
  // Do something
}
```

On each iteration, the object returned by the “reversed()” function will produce a new item, run the loop’s code and then proceed on producing the next item. This differs from an iteration over a list, since the iterator is not required to keep all items in memory at once, instead it has the liberty of producing only when asked to.

Generators is a concept that will be added to the language in future versions, and consists of a concise way for the programmer to build an Ad Hoc iterator. Describing a generator is much like describing a normal function; however, instead of returning a single value, this function will return the next value each time it is asked to do so.

For that to be possible a generator has a special return statement: The “yield” statement. Every time a generator yields it returns a value and pauses its execution. After the loop is executed and the function will be asked again for the next item, and then it will resume executing from where it stopped until it finds the next “yield” statement.
An example of the syntax and usage of this concept is illustrated below, however, the syntax used is not yet implemented on JSpy Language:

```
generator my_gen(arg) {
  while (arg > 0) {
    yield arg;
    arg = arg - 1
  }
}
```

```
for (item in my_gen(3)) {
  print(val) // Would print: 3 2 1
}
```

B.3 Reversed Index Feature

To facilitate iterating over a list in JSpy, as in Python, offers the possibility of indexing with negative numbers: This feature allows to access the last position of a vector using the index of "-1" and the N'th item from the back of the list by indexing with "-N".

B.4 Implicitly Declared Variable's Scope

An implicit declaration of a variable in both JavaScript and JSpy is when a programmer makes an assignment on a variable before declaring it using the special "var" statement:

```
function F1() {
  my_var = 'inside function';
}
```

In JavaScript, this variable would be implicitly assigned to the global scope, while in JSpy this variable would be implicitly assigned to the local scope. This means that in JavaScript it is easier to declare a global variable by accident, the example below illustrates this difference using the function "F1" declared earlier:

```
// JavaScript:
F1()
console.log(my_var) // Would print: 'inside function'

// JSpy
F1()
print(my_var) // Would throw an undefined variable exception
```

This change was designed for two reasons: (1) it protects the programmer from accidentally declaring global variables and (2) this saves the programmer from having to declare explicitly all the variables he uses with the "var" statement, possibly saving some time.

One important aspect of this feature is that the variable will *not* be implicit declared on the local scope if it already exists in a higher hierarchy scope. Otherwise, it would not be possible to make assignments on global variables from inside functions. The example below illustrates this feature in JSpy Language:

```
var external1;
function F2() {
  external1 = 1
  external2 = 2
  internal = 3
}
F2()

var external2;
print(external1, external2) // Would print: 1 2
print(internal) // Would throw an undefined variable exception
```

Moreover if the programmer wants to force a variable to be declared in local scope it may use the “var” statement to enforce that:

```
var V1 = ’external’, V2 = ’external’
function F3() {
  // Declared on local scope:
  var V1 = ’F3’;

  // Using the external variable ‘V2’:
  V2 = ’F3’;
}
F3()

print(V1) // Would print: ‘external’
print(V2) // Would print: ‘F3’
```

Finally if the user wants to force a variable to be declared in global scope intentionally, the key-word “global” contains a reference to the global scope and may be used like to achieve this:

```
function F4() {
  var V1 = ’internal’;
  global.V1 = ’global’
}
F4()

print(V1) // Would print: ‘global‘
```

B.5 Global Scope Protection

When working with libraries overwriting a built-in function might cause unpredictable behavior on the program. However, imposing to programmers the task of memorizing all built-in global variables as to avoid overwriting them is error-prone and likely to cause discomfort for the programmer.

As an alternative JSpy has two layers of protection for the global scope variables: The first is that the default scope of any program is not the global scope itself, instead, it is a child scope from the global one. This protects locally declared variables from being visible by external libraries. The second layer is that assignments to global variables work as if no global variable existed: Causing the program to declare a local variable with that name. This prevents the user from accidentally overwriting any built-in functions:

```
// Both statements will declare a new local variable:
var list = None;
map = None;

// Global variables remain intact:
print(global.list) // [Function: list]
print(global.map) // [Function: map]
```

If the user wants to overwrite or to declare a global variable it is still possible to be done with this syntax:

```
global.list = ’new value’
global.map = ’new value’
```

B.6 Named Arguments for Functions

In Python it is possible to make explicit which argument is being passed on a function call like this:

```
def func(a, b, c):
  print(a, b, c)

func(1, 2, 3) # This would print: 1 2 3
func(a=1, b=2, c=3) # This would also print: 1 2 3
```

It is even possible to change the order of the arguments:

```
func(b=2, b=1, c=3) # This would also print: 1 2 3
```

This Python feature is very useful in two situations:

- When a function has several default arguments, and the user only wants to use one of them. In JavaScript, the solution is either to join the optional arguments into a single object instance or to set all the default arguments, the programmer does not want to change, to null and the one he intends to change to the desired value. In both cases it is complicated and verbose.

- When calling a function, adding the name of argument together with its value can often be used as documentation, making it easier for other programmers to understand what that argument is there for.

For that reason, this feature was included on JSpy. However, for avoiding ambiguities on the language JSpy replaced the '=' by the colon character ':'. It is then possible to do the same thing in JSpy, with a slightly different syntax:

```
function func(a, b, c) {
  print(a, b, c)
}

func(1, 2, 3) // This would print: 1 2 3
func('a': 1, 'b': 2, 'c': 3) // This also would print: 1 2 3
func('b': 2, 'a': 1, 'c': 3) // This also would print: 1 2 3
```

In future versions of the language, it is expected for the quotes to be optional, making this syntax less verbose.

B.7 Prototypical Inheritance

In JavaScript the syntax used to implement inheritance is very different from the way it is in most languages, specially Java and C++, as exemplified below:

```
function F(value) { this.value = value }
F.prototype.attribute = 'Attribute'
var instance = new F('Value')

console.log(instance.attribute) // Attribute
console.log(instance.value)     // Value
```

We believe this syntax to be unnecessarily verbose and a little complicated to be learned. Specially the concept of calling the "new" operator over a function instead of doing it on a class definition. In JSpy the syntax for doing the same thing was modified:

```
var F = {
  '__init__': function(value) { this.value = value },
  'attribute': 'Attribute'
}
var instance = new F('value')

console.log(instance.attribute) // Attribute
console.log(instance.value)     // Value
```

This syntax is believed to be easier to understand and to be more friendly for programmers that are used to programming in Java and C++. One of the advantages is that the "new" operator is called upon the "class definition", making it more familiar.

With this syntax the prototype is declared as a "map" and the constructor function is optionally declared inside it using a reserved name: "__init__".

There is, however, one feature that is not yet implemented: There is no way yet for any function of this object to refer to it's "super" function as it would be expected on a classic inheritance system. This feature is expected to be included in future versions.

Appendix C

Describing Data with JSpy Patterns

This section will demonstrate that it is possible to emulate some reasonable complex structures using only the features attributed to the patterns of the JSpy Model.

To fully comprehend the examples here it is advisable to read at least until the end of Section 3 before reading this appendix.

To express these structures we will use a syntax very similar to JSpy Language's syntax, however, there will be only three functions available to keep it as close as possible to the model:

- **create("pattern" {...})**: Creates a new pattern or overwrite its meaning.
- **erase("pattern")**: Removes an existing pattern.
- **call("input")**: Produces a signal on the brain formed by the sequence of snapshots described by the quoted string. This signal is capable of activating any pattern that matches it. And any value returned by this pattern will be returned by this function.

  To describe simultaneous signals they will be separated by commas, e.g.: call("input1", "input2").

There will three operations that will be added. The operators "|" and "&" will work inside a pattern to represent the operations of alternate-occurrence and simultaneous-occurrence respectively. The third operator will allow the concatenation of existing patterns: "+", e.g.:

Also, it will be possible to concatenate a pattern using the operator "+", e.g.:

```
"pattern1" + "&" + "pattern2"
```

Would result in the pattern:

```
"pattern1&pattern2"
```

C.1 Describing a Dictionary Container

A dictionary is given by a pair of key and value. The concepts described by the JSpy Model are ideal to provide this functionality, so, for example, this three patterns:

```
"my_dict key1" { return "value1" }
"my_dict key2" { return "value2" }
"my_dict key3" { return "value3" }
```

Would be enough to declare a dictionary named "my_dict" and containing three values with three different keys. To erase or create a new value it would be just a matter of creating the respective pattern. And to have access to one of these values it would be required to use the "call()" function like this:

```
call("my_dict key3");
```

And it would return "value3".

C.2 Describing a List Container

A list container is a little bit more complex to implement, but also possible. This list will expose four operations:

- **head**: Returns the item on the top of the list.
- **tail**: Returns the list with top removed.
- **empty**: Is recognized if the list is empty.
- **push**: Adds a new item on the top of the list.

To implement this interface it will be necessary a total of four patterns:

```
"list head ('[^,]*')head;,?('.*')tail;" { return head }
"list tail ('[^,]*')head;,?('.*')tail;" { return tail }
"list empty " { return True }
"list push ('.*')list;&value ('.*')value;" { return value+','+list }
```

With this setup it is possible to create a new list just by creating a pattern that will return the entire list whenever it is recognized:

```
create("my list" { return "item1,item2,item3" })
```

Now it is a matter of using the list:

```
call("list head " + call("my list"))
```

Would return “item1”.

```
call("list tail " + call("my list"))
```

Would return “item2,item3”.

```
call("list empty " + call("my list"))
```

Would not be recognized, implicating in False.

```
call("list push " + call("my list"), "value item0")
```

Would return “item0,item1,item2,item3”