Cognitive Program Compiler

Toni Kunić

A Thesis submitted to the Faculty of Graduate Studies
in partial fulfilment of the requirements
for the degree of

Master of Science

Graduate Program in
Electrical Engineering and Computer Science
York University
Toronto, Ontario, Canada

April, 2017

Abstract

Cognitive Programs (CP) specify computational tasks for the executive controller of visual attention. They are built on top of the Selective Tuning (ST) model of attention and its extension, STAR. Currently, the common way of specifying CPs is via diagrams, which are neither standardized nor directly machine-readable. This necessitates tedious and time-consuming implementation of CPs by hand, which slows research and prevents rapid experimentation.

This thesis presents the specification and reference implementation of the Cognitive Program Compiler (CPC). The CPC reads tasks written in the Cognitive Program Description (CPD) format, based on a novel controlled natural language called Imperative English (IE). The CPC can then output executable code in a regular programming language. The reference implementation is easily extensible, and several output modules are provided.

The CPC output has been evaluated by specifying several real-world psychophysical experiments and comparing the generated code against known human performance for those experiments.

Acknowledgements

I would like to thank my supervisor, John K. Tsotsos, for his mentorship, guidance, and kindness from the first day I walked into his office as an undergraduate student, as well as for all the invaluable feedback and discussions he provided throughout the development of this work.

I would also like to thank my committee members, Franck van Breugel and Thilo Womelsdorf, for their time and expertise. I am grateful to Omar Abid, Yulia Kotseruba, Rakesh Sengupta, and Calden Wloka for their keen insights and comments.

Finally, I would like to thank my family: my uncle Željko Kunić for his support and encouragement that allowed me to attend University, my parents Ivančica and Zdravko Kunić for firmly establishing that one should always be both *dobar* (good) and *pametan* (smart), and my partner Kiera Shaw for her endless patience.

Contents

List of Tables

List of Figures

List of Code Listings

List of Abbreviations

API	Application Programming Interface
AS	Attentional Sample
CPC	Cognitive Program Compiler
CP	Cognitive Program
CPD	Cognitive Program Description
CPM	Cognitive Program Memory
FC	Fixation Controller
FHM	Fixation History Map
IDE	Integrated Development Environment
IE	Imperative English
Imp	Imperative English Interpreter
MM	Minsky Machine
NLIDB	Natural Language Interfaces to Databases
NLI	Natural Language Interface
NL	Natural Language
RSVP	Rapid Serial Visual Presentation task
STAR	Selective Tuning Attentive Reference model
ST	Selective Tuning model of visual attention
TM	Turing Machine
tWM	Task Working Memory
UML	Unified Modelling language
vAE	Visual Attention Executive
VH	Visual Hierarchy
VR	Ullman's Visual Routines
vTE	Visual Task Executive
vWM	Visual Working Memory

Chapter 1

Background

1.1 Introduction and Motivation

The human visual system may exhibit dramatically different behaviours based on the tasks it is given. A classic example of this is the work of Yarbus from 1967 [1], in which he conducted experiments involving eye movement patterns. In one of the experiments, he presented experimental subjects with an image of Ilya Repin's painting *Unexpected Visitors* and issued them various tasks as they looked at the image. While the experimental subjects were performing the tasks, Yarbus recorded their eye movements. Figure 1.1 shows some of the resulting scanpaths, with descriptions of the various instructions in the caption. The scanpaths show that the task had an immediate effect on where in the image the experimental subjects looked.

The primary goal of this thesis is to develop a novel and useful way of specifying tasks in the context of the Selective Tuning Attentive Reference model (STAR) [13], which is built on top of the Selective Tuning (ST) model

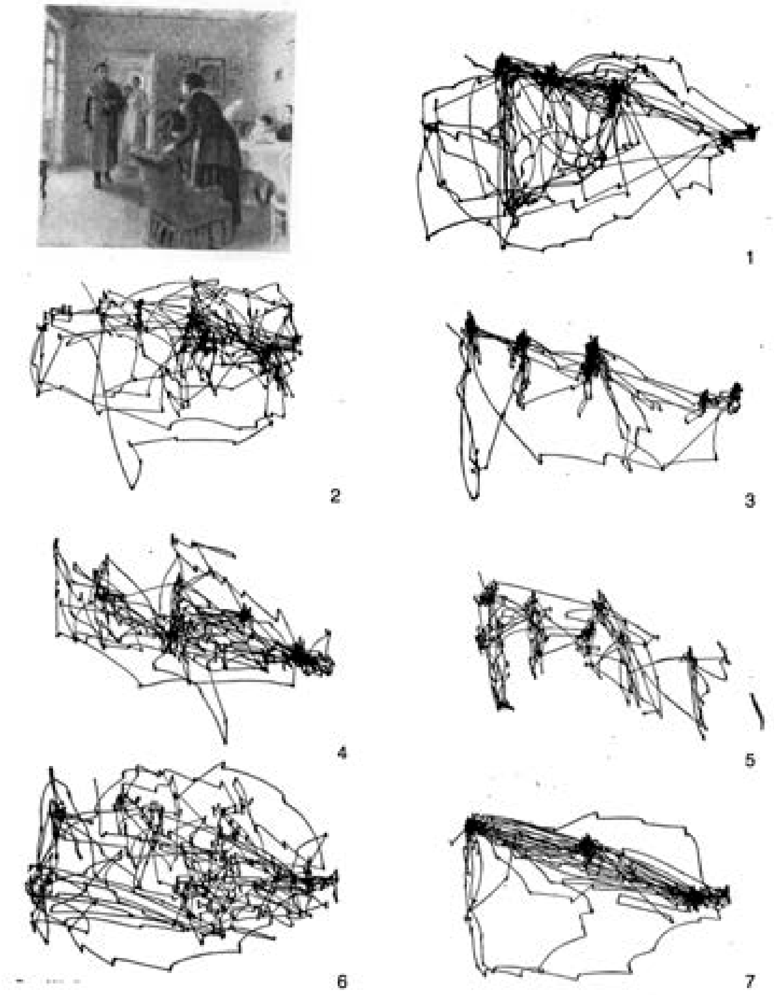

Figure 1.1: Results for Yarbus's 1967 experiment, reproduced from [1]. All images are 3 minute recordings by the same subject. The tasks are: 1. Free examination; 2. "Estimate the material circumstances of the family in the picture"; 3. "Give the ages of the people"; 4. "Surmise what the family had been doing before the arrival of the 'unexpected visitor' "; 5. "Remember the clothes worn by the people"; 6. "Remember the position of the people and objects in the room"; 7. "Estimate how long the 'unexpected visitor' had been away from the family."

of visual attention [14, 15]. STAR provides a computational model of visual cognition, and includes various elements, such as an attention executive, memory, task executive and fixation controller.

In the context of STAR, tasks take on the form of Cognitive Programs (CP) [2]. Cognitive Programs can be thought of as the software for the executive controller for visual attention, and are formulated as a modern extension of Visual Routines [16], first proposed by Ullman in 1984.

Currently, the common way of specifying CPs is via diagrams (example in Figure 1.2), which are not standardized or suitable for machine consumption. As a result, there is no easy way to specify or program CPs so that they can be used to perform experiments, or run simulations of the human visual system. To run an experiment in a neural network simulator, such as TarzaNN [17], one has to hand-code every experiment. This process is unwieldy and very time consuming due to the need to create and maintain two representations of the experiment: one in a diagram language and another in a programming language, both of which are of significant complexity.

To resolve these issues, this thesis develops a standardized specification format for Cognitive Programs called Cognitive Program Descriptions (CPD). CPDs are written in a subset of natural English and are supplemented by code in a regular programming language. They describe experimental sequences composed of two kinds of instructions: *endogenous* instructions which are issued to STAR (the experimental subject), and *exogenous* instructions which describe changes external to the experimental subject, for example which images will be shown to the experimental subject at what time, as well as management of elements the experimental

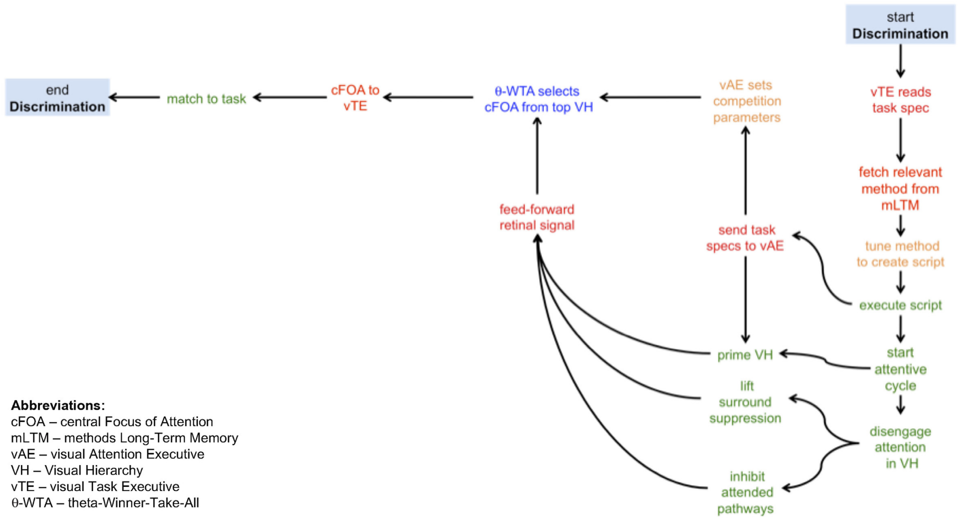

Figure 1.2: A graphical depiction of a CP discrimination task. Reproduced from [2].

subject can interact with, such as such as virtual displays and buttons.

The basis of the natural language portion of CPDs is a novel controlled subset of English called Imperative English (IE), whose simple structure is designed for issuing commands and stating facts about the world. This ensures that CPD task descriptions are easy to parse and execute for both humans and machines. However, in spite of IE's simple structure, it is a Turing complete programming language.

A reference implementation of a CPD compiler that can turn CPDs into machine-executable code is also provided. It is called the Cognitive Program Compiler (CPC), and it outputs Python code that can be used to drive the TarzaNN 3 neural network simulator, which is the successor of the original TarzaNN described in [17], extended to support STAR.

Unfortunately, TarzaNN 3 is – as of April 2017 – still under heavy development, and therefore unable to execute the output of CPC. To ensure that a suitable evaluation of CPDs is undertaken despite this fact, several real-world experiments studying visual attention are implemented as CPDs and the resulting output code discussed. The output code is also executed via a mockup program that logs the sequence of instructions that has been issued to TarzaNN.

1.2 Thesis Outline

This thesis is divided into five chapters and an appendix:

- **Chapter 1** describes the motivation behind this thesis, and provides an overview of relevant literature and background.

- **Chapter 2** provides the specification of Imperative English, proves that it is a Turing complete programming language, and provides the reference implementation of an IE interpreter called *Imp*.

- **Chapter 3** introduces Cognitive Program Descriptions as an extension of Imperative English suitable for describing task specifications in the context of STAR, and provides a reference implementation of a Cognitive Program Compiler.

- **Chapter 4** reports the results of empirical evaluation of the Cognitive Program Compiler output.

- **Chapter 5** concludes the thesis with discussion of results and notes for future work.

- **Appendix A** provides additional implementation detail relevant to work described in this thesis.

1.3 Literature Review

This brief overview of relevant literature covers STAR, Cognitive Programs, their history, predecessors and various implementation efforts, as well as the history and state of various existing paradigms for Natural Language Interfaces and Natural Language Programming.

1.3.1 STAR Model

In an informal, everyday context, people might consider visual attention to be something they intuitively understand. They might consider vision as little more than a light sensor capable of collecting images. A kind of biological camera that can be pointed at different subjects of interest. This simplistic view, however, tends to neglect the daunting amount of computation the human visual system needs to do for even the simplest tasks, such as finding an object in an image, or following a drawn line with one's gaze. This picture is further complicated by capacity limits inherent in the human visual system, which establish visual attention as a necessity in a visual system [18]. Tsotsos provides in [15] the definition of visual attention used throughout this thesis:

> "Attention is a set of mechanisms that help tune and control the search processes inherent in perception and cognition."

The model of visual attention we will focus on is Selective Tuning (ST), a biologically plausible computational model of visual attention, first proposed by Tsotsos in [18]. Since then, multiple papers have been published elaborating on its theoretical underpinnings, mechanisms, implementations, extensions, and extensive experimental support, an overview of which can be found in [15].

A recent extension of the ST model is the Selective Tuning Attentive Reference model (STAR) [2], currently under development by the Tsotsos Lab at York University. STAR was proposed as a way to extend ST to allow for executive control via Cognitive Programs (Section 1.3.2). The extensions add proposals for new model components and enhancements to existing ones to enable visual cognition. The main parts of STAR and their interactions are shown in Figure 1.3 and further discussed in the following text.

Visual Hierarchy (VH)

The Visual Hierarchy [14, 15] is the "heart" of ST. It is the part of STAR that takes as input signals from the retina (the photosensitive layer at the back of the eye), and processes those signals via a hierarchy of visual processing areas in the brain. The hierarchy takes on the form of a pyramidal lattice, a kind of graph with exactly one "least" node and exactly one "greatest" node, where every node is a sheet of neurons. It represents both the ventral and dorsal streams of visual processing areas in the brain.

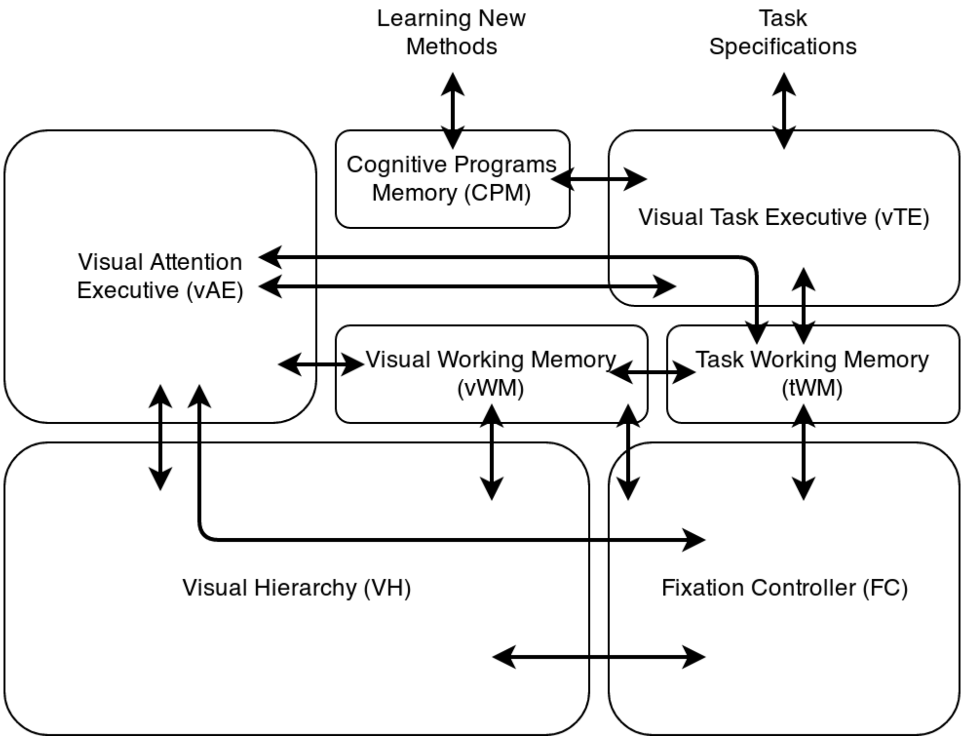

Figure 1.3: A diagram showing the main parts of STAR and the connections between them.

Fixation Controller (FC)

There are two ways one can attend to a stimulus in an image: by attending to a stimulus in one's field of view without looking at it, called *covert* attention, and by attending to a stimulus by directly pointing their gaze at it, called *overt* attention. The Fixation Controller [19] controls the eye movements that allow one to overtly direct attention. This includes controlling voluntary eye movements, such as saccades (voluntary jump-like eye movement for moving the retina from one point in the visual field to another), and smooth pursuit (voluntary eye movements for tracking objects moving in one's visual field).

Cognitive Programs Memory (CPM)

The Cognitive Programs Memory [2] (referred to as the methods Long Term Memory (mLTM) in some publications) stores Cognitive Program (CP) methods. CP methods are further described in Section 1.3.2, but the basic idea is that they are program outlines that the executive controller for visual attention can tune with some parameters and then execute. For example, "perform overt fixation" might be a method that can be tuned with a location parameter. Methods in the CPM can be composed into Cognitive Programs, and new methods could be learned and added to the CPM.

Visual Working Memory (vWM)

The Visual Working Memory [2] provides short term storage for information needed in executing visual tasks. For example, it stores the last several fixation locations in the Fixation History Map (FHM), as well as attentional samples, which are subsets of the Visual Hierarchy that represent stimuli it has processed. Attentional samples are made available to all components by the vWM.

Visual Task Executive (vTE)

The Visual Task Executive [2] is the primary focus of this thesis. It consists of three main components:

- *Script Constructor* – The Cognitive Program Compiler, in the context of this thesis. The CPC reads task specifications, consults the CPM for appropriate methods to compose the task out of, and tunes the methods with parameters extracted from the specification into executable scripts.

- *Script Executor* – Deploys the script that the script composer constructed to tune vision processes. It executes the script step by step, sending instructions to appropriate components. It also sends the task specification to the visual Attention Executive (vAE).
- *Script Monitor* – Checks every step of the execution for expected results and suggests changes for script progress when needed.

Visual Attention Executive (vAE)

The Visual Attention Executive [2] receives task parameters from the vTE and generates control signals for the Visual Hierarchy. It contains the cycle controller which initiates and controls different stages in the ST process. Control signals generated by the vAE have a wide variety of effects, from biasing the Fixation History Map to disengaging attention.

1.3.2 Cognitive Programs

A model of the visual system, such as STAR, can execute visual tasks. These can be anything from searching for object to tracing lines. In STAR, tasks are represented as Cognitive Programs (CP), which were first introduced by Tsotsos & Kruijne in 2014 [2]. Cognitive Programs are themselves a modern extension of an earlier idea called Visual Routines (VR), introduced by Ullman in 1984 [16].

Visual Routines were developed as a way of processing visual information. They are composed of sequences of elementary operations that can be performed on representations of visual information. In a basic sense, they are programs that take as input some representation of visual information and process it to encode properties and relationships between objects. The most basic kind of operations that VRs can perform are *atomic operations.* Ullman does not define the exact set of atomic operations, but he proposes some candidates, for example shift of focus, defining the next target, memorizing locations, boundary tracing, determining inside/outside relations etc. VRs can also be composed to create other VRs.

Ullman argues in [16] that there are two main stages to vision. The first one is a bottom-up creation of a *base representation* of the visible environment, which is the image from the retina that has undergone only basic processing without any information or feedback from the rest of the visual system. The second stage uses VRs to perform operations and transformations on the base representation constructed in the first stage to create *incremental representations.* VRs can be subsequently applied to those incremental representations as well.

Important, or frequently used VRs are stored as "skeletons." They are partially-completed VRs that need to be parametrized before use. For example, a shift-of-focus VR might be stored as a skeleton without a location to shift to. To use that VR, one would load it out of storage and add the location as a parameter. Ullman calls this process "assembly." Novel or uncommon VRs are composed out of known VRs at runtime and then parametrized. VRs can be applied to both *base* and *incremental* representations.

Ullman lays out the above basics of his system, however, he does not discuss implementation or exact mechanisms, nor does he provide a prototype. Since 1984, there have been multiple attempts by other researchers at implementing Visual Routines, but each attempt was based on their own, slightly different, interpretation of some ambiguous details in [16]. An overview can be seen in [20], with the majority of implementations treating VRs as a collection of program fragments that can be composed together to perform arbitrary procedures, none of which is considered the "canonical" VR implementation.

Despite VRs in general being a useful paradigm, a lot has changed regarding our understanding of vision and attention since 1984. One of the most important observations is that a complete base representation containing all information about the visual scene cannot be constructed in a single feed-forward pass as Ullman assumed, but that vision is a dynamic process [18, 15, 21, 2]. This means that VR's stimulus-driven, bottom-up base representations simply do not have the required complexity to model all the recurrent processing occurring in the brain.

Cognitive Programs try to modernize Visual Routines with modern vision science insights to allow them to model the visual system as "a dynamic, yet general-purpose processor tuned to the task and input of the moment" [2], instead of just a passively observing module that can answer simple queries about what it sees. Cognitive Programs themselves can be thought of as software for STAR's visual Task Executive.

The reader might wish to consult Figure 1.4 while reading the following, as it provides a detailed view of the parts of STAR immediately involved with Cognitive Programs. Cognitive Programs begin as *Cognitive Program Descriptions* (CPD) that are provided to the visual Task Executive (vTE) as task specifications. The Cognitive Program Compiler (script constructor) within the vTE then attempts to construct a script out of the CPD. The CPC uses the CPD to consult the Cognitive Program Memory (CPM) for what kinds of *methods* the task could be decomposed to. Methods are similar to what Ullman calls “skeletons”: they are unparametrized CP outlines. The methods could be high-level (for example, the discrimination task) or low-level (for example, disengage attention), and some methods may subsume others since CPs can be composed just like VRs can. The CPC, in addition to deciding which methods from the CPM are appropriate for the task at hand, extracts *parameters* from the task specifications to use to tune the methods into an executable *script*. The script is then handed to the Script Executor and Monitor to execute. The Executor initializes the attentive cycle and sends elements of the task specification required for attentive tuning to the visual Attention Executive (vAE), and the Monitor supervises script execution and waits for the vAE to signal that the completion conditions have been fulfilled.

As an example of this process, let us consider the task of finding Waldo, a famous children’s book character [22]. The task consists of finding a cartoon character of a unique appearance in a single large image of colorful cartoon characters, and represents a variant of the visual search task. In the context of Cognitive Programs, the process would proceed as follows:

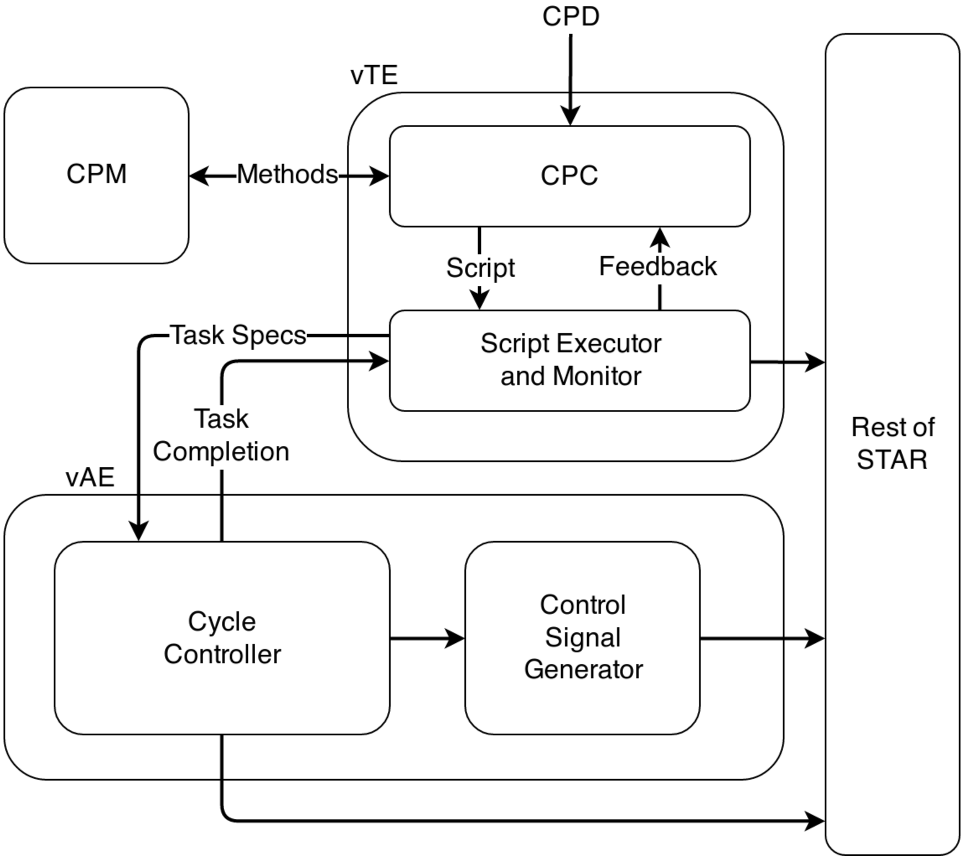

Figure 1.4: A more detailed view of the components of STAR that are important for Cognitive Programs.

1. vTE begins by reading the specification of the *task* the visual system needs to perform: “Find Waldo in this image.”

2. CPC consults the CPM for appropriate *methods* to apply to this task, for example a discrimination method, a localization method, and a visual search method.

3. The *methods* are then tuned with *parameters* provided to the vTE: for example, Waldo is wearing a red and white striped shirt, a pattern

which we can use to tune the visual search method into a *script*.

4. *Scripts* are then executed and their execution monitored by the vTE. The vTE also sets control signals for the rest of the STAR system and monitors results to determine when the task is complete (when Waldo has been found).

Currently, the way one would describe an equivalent search task to the one above is by using a diagram like the one in Figure 1.5. The issue here is that there is no standard way of specifying these kinds of diagrams, and they are difficult to translate to executable code one could use to conduct simulations and experiments. This thesis contributes a way to enable a natural language specification of a visual task, as well as a way to convert it into an executable Cognitive Program. For example, the above sentence "Find Waldo in this image," would ideally be all the input a system would need to compose a program that performs a process equivalent to the one in Figure 1.5.

1.3.3 Natural Language Interfaces

At varying times in history, respected researchers have optimistically and enthusiastically claimed that we are mere years or decades away from having a conversant computer program, one that understands human speech, or even an entirely new species of *Machina Sapiens* [23]. Unfortunately, most of the research community's self-imposed deadlines for such achievements have thus far been missed [24, 25].

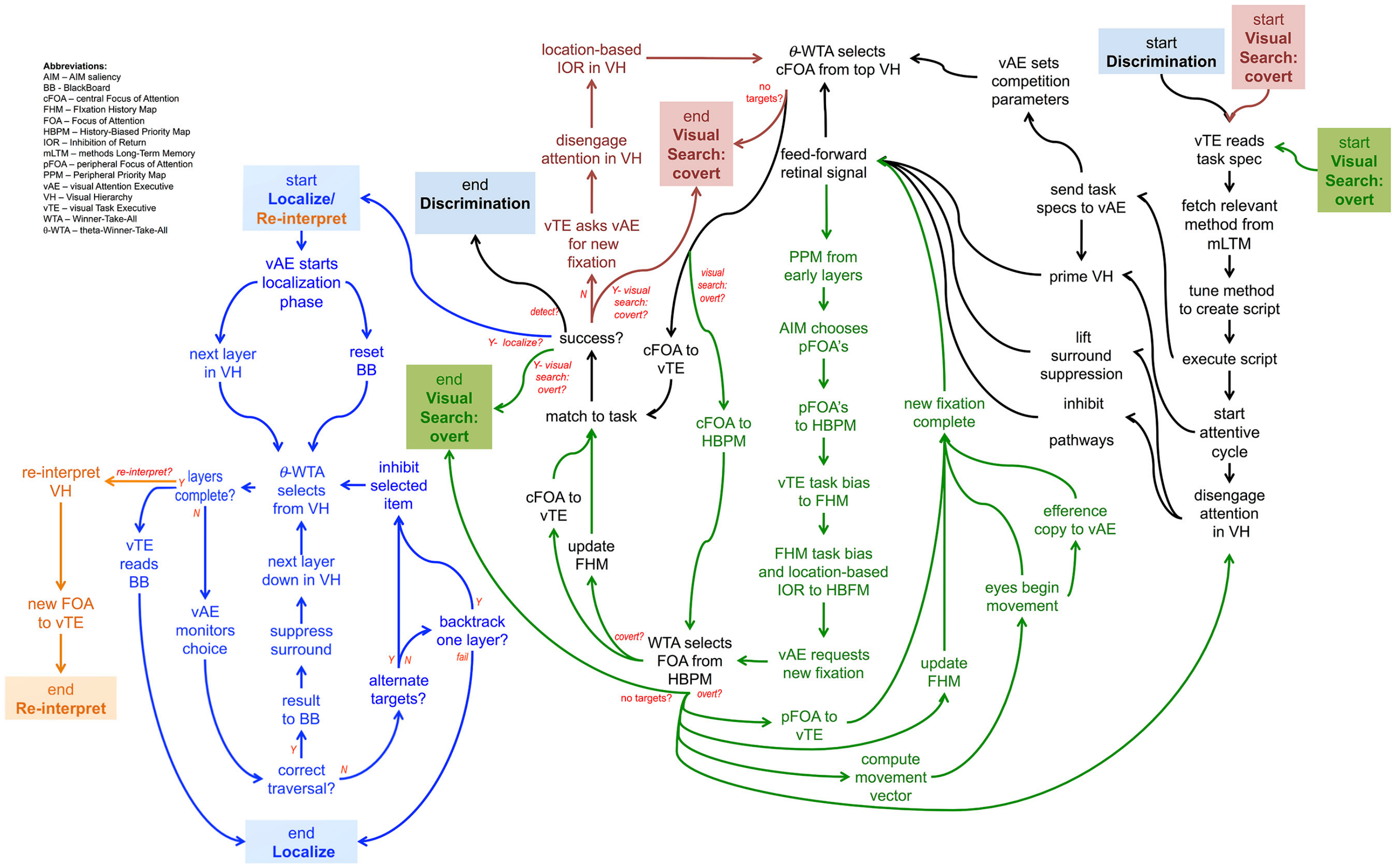

Figure 1.5: Example of a Visual Search Cognitive Program diagram composed of multiple methods. Reproduced from [2].

This optimism was especially strong in the days of early attempts at natural language understanding, and much of it was based on several successful demonstrations of basic natural language processing programs, such as **STUDENT** [26] which solved algebra problems given in natural language, and **ELIZA** [3] that simulated a conversation with a psychoanalyst. In fact, ELIZA was so convincing when it was introduced in 1964, that its inventor, Joseph Wiezenbaum, had a hard time convincing early users that ELIZA had no true emotion or understanding of the conversation. In [27], he relates his astonishment at people's belief that ELIZA demonstrated a general solution to the problem of natural language understanding, and that sometimes even his statements to the contrary were ignored. Both STUDENT and ELIZA worked by matching simple patterns in input text based on hand-crafted rules, and were only considered convincing by their contemporary audiences due to the extremely constrained domains they operated within and the novelty of such programs.

Another early success was **SHRDLU**[1] built in 1968 by Terry Winograd [4]. SHRDLU was a simulation of a simple world of 3D geometric shapes called blocks, with a simulated agent that could interact with the world and respond to commands issued via a natural language interface. The vocabulary necessary for these interactions was kept very small, by keeping the possible content of the world very restricted. Nonetheless, SHRDLU could answer queries about the block world ("how many blocks are in the box?"), and perform manipulations ("put the pyramid into the box.").

---

[1] SHRDLU's peculiar name comes from "etaoin shrdlu", a phrase resulting from running one's fingers down the two leftmost columns of keys on a linotype machine keyboard.

It can be seen from transcripts of ELIZA and SHRDLU sessions in Listings 1.1 and 1.2 why they might be considered by a naïve observer as convincing intelligences, and some might even walk away with the idea that true artificial intelligence is but a step away in research terms.

Listing 1.1: Excerpt from ELIZA session. Reproduced from [3].

```
User: I need some help, that much seems certain.
ELIZA: WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
User: Perhaps I could learn to get along with my mother.
ELIZA: TELL ME MORE ABOUT YOUR FAMILY
User: My mother takes care of me.
ELIZA: WHO ELSE IN YOUR FAMILY TAKES CARE OF YOU
User: My father.
ELIZA: YOUR FATHER
User: You are like my father in some ways.
ELIZA: WHAT RESEMBLANCE DO YOU SEE
User: You are not very aggressive but I think you don't
↪ want me to notice that.
ELIZA: WHAT MAKES YOU THINK I AM NOT VERY AGGRESSIVE
User: You don't argue with me.
ELIZA: WHY DO YOU THINK I DON'T ARGUE WITH YOU
User: You are afraid of me.
ELIZA: DOES IT PLEASE YOU TO BELIEVE I AM AFRAID OF YOU
```

Listing 1.2: Excerpt from SHRDLU session. Reproduced from [4].

```
Person: PICK UP A BIG RED BLOCK.
Computer: OK. (does it)
Person: GRASP THE PYRAMID.
Computer: I DON'T UNDERSTAND WHICH PYRAMID YOU MEAN.
Person: FIND A BLOCK WHICH IS TALLER THAN THE ONE YOU ARE
↪ HOLDING AND PUT IT INTO THE BOX.
Computer: BY "IT", I ASSUME YOU MEAN THE BLOCK WHICH IS
↪ TALLER THAN THE ONE I AM HOLDING.
Computer: OK. (does it)
Person: WHAT DOES THE BOX CONTAIN?
Computer: THE BLUE PYRAMID AND THE BLUE BLOCK.
Person: WHAT IS THE PYRAMID SUPPORTED BY?
Computer: THE BOX.
Person: HOW MANY BLOCKS ARE NOT IN THE BOX?
Computer: FOUR OF THEM.
```

```
Person: IS AT LEAST ONE OF THEM NARROWER THAN THE ONE WHICH
↪  I TOLD YOU TO PICK UP?
Computer: YES, THE RED CUBE.
Person: IS IT SUPPORTED?
Computer: YES, BY THE TABLE.
```

Unfortunately, despite early successes, enthusiasm, and optimism, natural language understanding is still an unsolved problem [25]. However, while English (and other natural languages) cannot be generally understood by a machine yet, use of constrained natural language interfaces has been successful in practice.

One particularly successful domain are Natural Language Interfaces to Databases (NLIDBs) [28]. Studies have shown that people frequently prefer interacting with databases via natural language tools to other methods. For example, Capindale and Crawford showed in 1990 [29] that out of a group of 19 subjects with varying computer expertise, 72% of users "liked very much" to enter queries to a database in natural language, and 80% felt that it was a "far superior" or "somewhat better" method of querying the database compared to a menu system interface. The study also claims that NLI is especially suitable in the following cases:

- "for tasks which involve question-answering activities in a domain whose horizons are limited,"
- "when users are knowledgeable about the task domain and intermittent use inhibits command language training," and
- "if the natural language system provides sufficient feedback to inform users about language restrictions."

The last point is particularly important. A robust error reporting system has proven to be vital to the functioning of NLI systems in practice. In particular, [28] discusses the importance of distinguishing between *linguistic* failures (occur when the natural language system is queried using words and sentences the system is unable to link to concepts) and *conceptual* failures (occur when the natural language system is queried for concepts it does not know of). Limitations of the system need to be apparent to the user to avoid frustration. One method that Intellect, the NLIDB system used in the study in [29] uses to ameliorate this issue is via a feature called *echo*: Intellect, given a line of input by the user outputs the query interpretation, allowing the user to learn how to pose clearer queries to the system, reducing training time to learn the system, and enabling frustration-free learning of the system's limits while working with it.

An additional, perhaps slightly counter-intuitive, yet encouraging observation relating to NLIs in practice is that people who use a system with a restricted vocabulary are not any slower or less detailed in expressing themselves than people who use unrestricted vocabularies (in effect, fully natural conversational language) [30, 31].

A further note comes from Pane et al. [32], who studied what kind of language constructs non-programmers use when describing computational tasks. For our purposes, non-programmers are significant because their utterances come closer to something one would call natural language, as opposed to the way programmers might assume a computer's limitations and try to work around them. Non-programmers tended to express operations in terms of sets, as opposed to loops. Consider, for example expressing a set of operations as "Peel, slice and eat the apples," instead of expressing a loop like "for every apple, peel it, slice it and eat it."

The first NLIDB system is **LUNAR** [33], developed by Woods et al. in 1972 for querying a database of lunar rock samples from then-recent Apollo 11 mission. A sample LUNAR query is "WHAT IS THE AVERAGE CONCENTRATION OF ALUMINUM IN HIGH ALKALI ROCKS?"

An example of a modern NLIDB system is **PANTO** by Wang et al. [34]. PANTO provides a natural language interface to query an ontology, a "knowledge base (KB) that includes concepts, relations, instances and instance relations that together model a domain." PANTO is able to answer queries like "which is the longest river that flows through the states neighboring Mississipi." The way PANTO works is similar to other similar systems: first, it constructs a parse tree (a hierarchical representation of the structure of a sentence, where different parts of the sentence are labelled) from the input text. PANTO uses Stanford's CoreNLP [35] as an off-the-shelf parser to generate this tree. Then, PANTO extracts phrases out of the generated parse tree that are then processed via a pipeline of operations to map via intermediate representations onto the ontology. If this mapping is successful, PANTO outputs a SPARQL [36] SELECT statement that can be used to query the database.

Damljanovic et al. [37] follow a similar approach to PANTO, in developing **FREyA**, which also uses CoreNLP and outputs SPARQL queries, but uses an interesting strategy in the case it fails to find a query mapping to the ontology: it provides the user with a list of ranked possible interpretations, and uses the user's choice for machine learning in order to be able to interpret queries better in the future.

NaLIX [38] is a system that generates XQuery [39] statements to query XML documents. There are three steps to NaLIX: parse tree classification (where NaLIX matches tokens to the XML structure, and marks tokens it cannot match), parse tree validation (where NaLIX labels a tree as "invalid" if it cannot map the tree onto XQuery), and parse tree translation (where NaLIX translates the parse tree into XQuery). The authors concede, however, that it takes users one or two tries to write a query NaLIX can understand, hence they rely on an feedback messages and an IDE-like GUI to guide the user.

Several problems arose in the practice of creating NLIDB systems, which are problems most NLI systems will have to provide a solution to – or at least a coping strategy. One of them is anaphora resolution (how to match concepts to "it" or "that one"), and the related problem of elliptical sentences (incomplete, partial sentences that can only be understood in context). A further issue is that the conjunctive "and" is sometimes used to mean set-theoretic union and other times intersection. For example, to disambiguate the sentence "List all students at the university whose GPA is above 3.9 and below 2.0," we need to have common-sense knowledge that it is impossible for a student to have a GPA that is above 3.9 and below 2.0 at the same time, and that the query therefore calls for a union operation on the set of students with a GPA greater than 3.9 and the set of students with a GPA below 2.0. There are numerous other examples of queries that may require common-sense knowledge to disambiguate.

There are many more applications of NL interfaces, too many to list here. From robot control [40, 41] to various voice recognition based personal assistants and home automation systems (Apple Siri[2], Google Assistant[3], Microsoft Cortana[4], Amazon Alexa[5]), and web query engines (Wolfram Alpha[6], Google Search[7]). Unfortunately, many of these systems are proprietary, and as a result, not much is published detailing how they work.

However, many of these systems allow for third party extensions. For example, one could write a module that allows a personal assistant to respond to a novel query. We can use the extension developer documentation that proprietary systems provide to infer the capabilities of proprietary systems. Consider, for example, Amazon Alexa. To create an Alexa third party extension, one has to provide two things: a set of input utterance patterns (for example, "get high tide for City"), and a program that generates a textual response for Alexa to speech synthesize back to the user based on the input. This requires developers to predict the exact queries that the user would make, potentially making the system quite brittle if the user does not know the exact query pattern that the extension expects. In practice, however, I assume a technique like a sentence encoder[8] may be used to map the user's queries to embeddings in a many-dimensional space, and if there exists an embedding of a template sentence similar enough to the input, assume that the user meant that template.

After observing in this section that natural language interfaces have been

---

[2] http://www.apple.com/ios/siri

[3] http://assistant.google.com

[4] http://www.microsoft.com/en-us/mobile/experiences/cortana

[5] http://developer.amazon.com/alexa

[6] http://wolframalpha.com

[7] http://google.com

[8] for example, the recently successful skip-thought vectors [42].

used more or less successfully in a variety of domains, a question is bound to arise in a computer scientist's mind: could one program computers in natural language?

1.3.4 Natural Language Programming

Natural Language Programming, in the sense of using a natural language such as English as a programming language, was first proposed by Jean Sammet in 1966 [43]. She suggested that a computer should understand natural language and have the ability to address ambiguity and other shortcomings natural language has compared to traditional programming languages. One of her primary motivators for suggesting this paradigm is that it should not be necessary to know the inner workings of a computer to program one to solve problems (Sammet gives the analogy of many people successfully driving cars while knowing virtually nothing about how they work).

In the 1960's, when Sammet proposed Natural Language Programming general systems (systems capable of disambiguating casual general natural language inputs regardless of domain, and outputting code based on that input) did not exist. Unfortunately, the quest to produce such systems has not borne fruit to this day [25]. This is not to say that research has made no progress since the 1960's, however, existing systems only work with restricted input vocabulary, in restricted domains, and often even with restricted syntax.

One of the first programming languages designed to be similar to natural language was **FLOW-MATIC**, developed from 1955 until 1959 [44] by Grace Hopper, a predecessor to the more famous **COBOL** [5]. However, one can see from Listing 1.3 that such languages, while using some semblance of English keywords and syntax, can hardly be called natural.

Listing 1.3: Example of COBOL code, reproduced from [5].

```
IF A = B THEN MOVE X to Y;
    OTHERWISE ADD D TO E;
        MULTIPLY F BY G.
MOVE H TO J.
```

Another early example is **JOSS** (JOHNNIAC Open Shop System) [6], developed by Shaw in 1964 as an experiment used by members of The RAND Corporation for small numerical computations. An example of a JOSS session in Listing 1.4, and again, we can see that the resemblance to natural English is fleeting at best.

Listing 1.4: Example of JOSS code, reproduced from [6]. U denotes the user input, J denotes the JOSS output. Note that the paper predates common keyboard layouts, and therefore the mid-dot character (·) is used for multiplication, and * for exponentiation.

```
U: Type 2+2.
J:     2+2 =     4
U: Set x=3.
U: Type x.
J:       x =     3
U: Type x+2, x-2, 2·x, x/2, x*2.
J:     x+2 =     5
       x-2 =     1
       2·x =     6
       x/2 =     1.5
       x*2 =     9
U: Type [(|x-5|·3+4)·2-15]·3+10.
J: [(|x-5|·3+4)·2-15]·3+10 = 25
```

The **Structured Query Language**, more commonly known as **SQL**, was proposed in 1974 [45], and is to this day the most widely used natural English-like programming language. SQL is used to manage databases in relational database management systems (RDBMS), a widespread relational database model used to store information in a structured way. A sample query written in SQL is provided in Listing 1.5.

Listing 1.5: Example of a SQL query issued to a database containing a table of table employee records. It lists, in lexicographic order of last name, the names of all employees that live in Toronto and are older than 50.

```
SELECT Employee.last_name, Employee.first_name FROM
↪ Employee WHERE age > 50 ORDERED BY 'last_name';
```

NLC, introduced by Ballard and Biermann in 1979 [7] is a prototype natural language (NL) programming language for interacting with matrices. It resembles natural English more closely than the previously discussed systems. The NLC language consists of sentences written in the imperative grammatical mood, the form of verb used in English to express commands and requests. Several examples of such sentences are provided in Listing 1.6. The authors made this design decision because they found that users preferred issuing commands in the imperative grammatical mood, leading to a significant simplification of the system design: the verb always came first in a sentence. NLC differentiates between three kinds of knowledge that a NL programming system should have: *linguistic knowledge* (which language constructs are allowed, and how to resolve ambiguity), *domain-specific knowledge* (the set of known domain entities), and *computational knowledge* (which computational procedures should be used to execute commands). The authors further observe that an essential feature of a NL programming language is the ability to be extensible both within its domain of application, and to other domains of application.

Listing 1.6: Example of NLC code, reproduced from [7].

```
Choose a row in the matrix.
Put the average of the first four entries in that row into
↪ its last entry.
Double its fifth entry and add that to the last entry of
↪ that row.
Divide its last entry by 3.
Repeat for the other rows.
```

Biermann and Ballard go on to perform an interesting study using NLC in [46]. The study is performed four years after the original paper, and features a version of NLC which has been extended to the domains of solving linear equations and gradebook averaging. 23 first year undergraduates typed a total of 1581 sentences, of which 81% were processed correctly, and the authors note that "none of the standard concerns about natural language programming related to vagueness, ambiguity, verbosity or correctness was a significant problem, although minor difficulties did arise occasionally." The minor difficulties they speak of were system failures due to "'bugs' or syntactic oversights which appear amenable to easy repair."

In the late 80s, the **xTalk** family of languages arose, so named for Apple's HyperTalk language [47]. HyperTalk was intended to be an easy to use, English-like language for Apple's HyperCard program [48], an early commercial excursion into hypermedia that accesses a database via a flexible user interface metaphor of "cards." The language itself was based around an object-oriented paradigm: every object in the HyperCard system inherits its properties and attributes from objects higher up in the hierarchy. Since the introduction of HyperTalk, there have been numerous xTalk dialects, including AppleScript [49], MetaTalk [9], and LiveCode [10].

Listing 1.7 contains an example of a typical xTalk program, reproduced from LiveCode developer documentation[11]. From it, we can see that its structure is very much like that of a classical procedural programming language, with keywords replaced with English words. Even though xTalk's verbosity is often touted as an asset that aids understanding, the example likely is not as intuitive to the average reader as it would be in something structurally closer to conventional English.

Listing 1.7: Example of a LiveCode program from the LiveCode documentation. It emits an audible beep upon a key press only if the key pressed corresponds to a number. The lines starting with a double hash sign are comments.

```
on keyDown pKey
  if pKey is not a number then
    ## If the parameter is not 0-9
    ## beep and prevent text entry
    beep
  else
    ## Allow text entry by passing the keyDown message
    pass keyDown
  end if
end keyDown
```

In 2000, Price et al. developed **NaturalJava** [50], which they describe as an interface in natural language for creating, modifying and examining abstract syntax trees of programs written in the Java programming language

---

[9] http://www.metacard.com

[10] https://livecode.com

[11] From: http://lessons.livecode.com/m/2592/l/125554-accepting-only-digit-characters-during-data-entry

[51]. Throughout a NaturalJava session, a window is displayed with code resulting from the currently entered set of operations. To process natural language input, the authors present a three-part pipeline: (1) a system that processes natural language into case frames (templates with slots in them for words of a certain type) by keyword matching parts of the natural language input text to one of the 27 kinds of case frames that might be relevant, (2) a frame interpreter that uses a decision tree to infer what kinds of edit operations are being requested by the user, and (3) an abstract syntax tree manager that takes those edit operations and applies them to Java's abstract syntax tree.

The decision tree used in step (2) makes an assumption that every request contains exactly one action, and that the first verb in the request determines what kind of action that is. This rule of thumb is consistent with the observations made by the developers of NLC [7] about humans prefering to use imperative, verb-first sentences to instruct computers in English. However, NaturalJava's syntactic structure is closely coupled with that of Java constructs, which makes it unsuitable for many other domains, and requires knowledge of Java constructs to structure input.

In 2005, Liu and Lieberman developed **Metafor** [52], a system that can take an input "story" written in natural language and outputs Python code fragments that represent object definitions and their relationships in the story. The work heavily leverages the authors' earlier work on the MontyLingua natural language understanding system [53], which is used as a parser that converts input text into a verb-subject-object-object (VSOO) form, and ConceptNet [54], a common-sense knowledge database based on the

Open Mind project [55]. After the creation of VSOO objects, a "small society of semantic recognizers" (presumably hand-crafted) identifies special structures and objects in the VSOOs using the common-sense database, and then the VSOO structures are mapped to changes to the output code model. Note that the structures are not mapped to the output code model, but to *changes* to it. One interesting case the authors make is that ambiguity in a story is a good thing, because it allows Metafor to postpone decisions, and to continuously re-interpret the internal representation and resulting code as more of the story is entered and disambiguated.

Unfortunately, interpreting how well Metafor works in practice as a natural language system is difficult, as the authors only provide an outline of a study to evaluate Metafor's capabilities as a natural language programming system, where most of the evaluation was qualitative, with an examiner who "would occasionally gently rephrase the volunteer's sentences if the volunteer strayed too far outside Metafor's syntactic competence," making it very difficult to infer whether the system implementation was in fact usable. The inclusion of the parse tree as part of their GUI "for debugging purposes" seems to suggest that the system was quite difficult to use in practice.

The authors of **Pegasus** from 2006 [56], describe "naturalistic programming" (writing computer programs with the help of natural language) as a new paradigm. It reads a natural language (English and German implemented) and outputs executable program files. It operates on a restricted grammar, with structures similar to a regular programming language, which the authors also note as one of its primary weaknesses. Its lexicon is implemented as a hierarchy of Java classes representing "ideas," sets of perceptions that describe objects in the world, which is an attempt at formalizing concepts of a common-sense knowledge base.

Inform 7, published in the same year by Nelson [57], is a domain-specific programming language whose primary purpose is developing interactive fiction games. The Inform7 whitepaper [57], which serves as the design document and user manual, argues that in a natural language system, a useful and detailed error reporting system is essential for a programming language compiler that is of practical use to real-world users.

Inform 7 extensions are written in Inform 7 as well. However, whereas the average Inform 7 program reads like natural English, its extensions read like xTalk languages, as shown in Listing 1.8 [12]. In general, with NL languages seen so far, there seems to be a trade-off between the precision of traditional (low-level) programming languages and the readability of natural language (higher-level), where structurally more complex and intricate programs are actually easier to write in lower-level languages.

Listing 1.8: Example of an Inform 7 extension for a spellcasting game mechanic, from Inform7's contributed extensions documentation.

```
Rule for printing a parser error when the latest parser
↪ error is cannot see any such thing error (this is the
↪ new cannot see any such thing rule):
  let T be indexed text;
  let T be the player's command;
  if word number 1 in T is "cast":
    let W be word number 2 in T;
    repeat with N running from 1 to the number of rows in
    ↪ the Table of Spells:
      let S be the spell in row N of the Table of Spells;
      let SS be the printed name of S;
      if W is SS:
```

---

[12] From: http://inform7.com/extensions/Jim%20Aikin/Spellcasting/source_1.html

```
        say "[cant-see-any-such]";
        stop the action;
    say "[unknown-spell]";
  otherwise:
    say "[cant-see-any-such]".
```

A dramatically simplified paradigm for programming is proposed by Little and Miller in 2007, in the form of **keyword programming** in Java [58]. It takes on the form of synthesizing most likely Java expressions based on a set of keywords and the surrounding program context. While not programming in natural language per se, the practical success of such a simplistic approach is definitely interesting.

PiE (Programming in Eliza) from 2014 [8], is an interesting adaptation of the ELIZA system [3] discussed in Section 1.3.3. PiE uses ELIZA to engage the user in a dialogue to determine what the user wants to do, and uses the result of such an interactive programming session to generate code in the LOGO programming language [59], an educational programming language where a user moves a "turtle" to draw shapes on a screen. An excerpt of a PiE session can be seen in Listing 1.9. This sort of interactive approach where the user clarifies ambiguities by answering questions posed by the system seems to be a common method of dealing with our current limitations in understanding natural language.

Listing 1.9: Excerpt from a PiE session from [8]. User input begins with prompt '>'.

```
Thanks for trying PiE. How can I help you?
> 'Lets first draw a line
Okay, what is the length of that line?
> Oh, I mean, to define a function called line
I see, what are the arguments of the function Line?
```

```
> They are count and n
Okay, I have designed a function called Line with input
↪ arguments called count and n. Then?
> let the turtle move ahead n steps when the count equals
↪ to one
Okay, if the Count is 1, turtle will forward N steps
(...)
```

SmartSynth, released in 2013 [60], is a system for synthesizing automation scripts for smartphones from a natural language description. Its rough organization follows two processing stages:

1. *Natural Language Processing Stage* – Extracts entities from text using regular expressions based on the fact that every entity known by the system has a set of common terms associated with it that users might refer to it by. A bag-of-words model (a commonly used model in natural language processing that stores words encountered in the text and their number of occurrences, to form a kind of "fingerprint" of the text) is used to categorize the input text into components and some dataflow relations to establish interactions between the generated components.

2. *Program Synthesis Stage* – Infers the remaining dataflow relations between components based on component types, and then constructs executable scripts.

The authors provide a user study evaluating SmartSynth based on 640 natural language descriptions of 50 tasks collected from smartphone help forums, where the system produced the intended scripts 90% of the time.

This study implies that even a restricted, domain-specific language can be intuitive and easy to use in practice.

Manshadi et al. [61] provide an interesting idea combining programming by example/destination (PbE/PbD) and natural language programming, and apply it to simple text editing tasks in spreadsheets. In this technique there are two inputs: a natural language task description; and a set of input/output pairs examples that the synthesized code should be able to produce.

NLCI introduced in 2016 by Landhäußer et al. [62], recognizes the issue that a lot of natural language programming languages are domain-specific and difficult to build, maintain, and port to different domains, so they provide a way to "ontologize" an Application Programming Interface (API). An API is a set of clearly defined interfaces that software components can use to communicate with one another. The authors aim to develop NLCI as a system that would allow one to issue commands to an existing API in natural language. This approach recognizes the reality that NLP is restricted to being domain-specific in practice for the time being, but nonetheless provides a way to make it practical. One downside is that the ontology generation from an API can only be automated if the API follows a particular naming convention, and otherwise needs to be created explicitly.

The authors provide two applications as demonstration of this system: a home automation API and a 3D animation API. They test NLCI on 50 input scripts and observe a recall of 67% and precision of 78%, which they concede is not acceptable for practical use, but indicate that the approach is promising. NLCI works by using the ontology to map noun phrases to classes, and roughly verbs to methods. Ambiguous matches are ranked and the highest-scoring result is selected. Knowledge of synonyms of different words is provided by WordNet [63], a lexical database of English that groups words into sets of synonyms, and provides relationships between different sets of synonyms.

An interesting class of NL programming languages are esoteric NL programming languages, which are not intended for serious software development, but as a playful test of boundaries of language design. Some of the more well known esoteric languages based on English are **Shakespeare** (2001) [64] and **Chef** (2005) [65]. Shakespeare takes as input texts that look like Shakespeare's plays, and Chef takes as input texts structured to look like recipes. Programs written in either of these languages do not look like a program at first glance, and they both interpret innocuous sounding statements as operations on a stack data structure. Nonetheless, they are both Turing complete languages, which means that they are at least as powerful as "real" programming languages like Java, Python or C.

Knuth's literate programming [66] – while not exactly programming in natural language – is also worth mentioning as a related technique. Literate programming is not tied to a particular programming or natural language. Its documents are regular text that provide an non-executable explanation of a program in natural language, which is interspersed with code and macro commands. Literate programming documents can be parsed to generate code executable by a computer, while keeping the input format self-documenting and easily readable by a human.

1.3.5 Controlled Natural Languages

As seen from Sections 1.3.3 and 1.3.4, none of the current solutions that use natural language input use general natural English. So why not embrace this fact and intentionally construct a constrained subset of English based on a set of rules?

In fact, this is an established practice, and the resulting languages are called Controlled Natural Languages (CNL) [67, 68]. Here is an overview of some examples, together with some brief comments on their suitability for specifying CPs.

- **Attempto Controlled English (ACE)** [69]: Designed to be easily parsed by machines into first order logic statements, it is well suited for representation of knowledge and relationships, and is used in theorem provers, software specifications, ontology construction, etc. Unfortunately, ACE was designed primarily with AI knowledge representation in mind, and therefore is declarative in nature (e.g. "All humans are mammals; No mammals are reptiles; No humans are reptiles."), and therefore unsuitable for CP specification, since there is no natural way to issue commands and instructions.

- **Sowa's syllogisms** [70]: Based on Aristotle's Syllogisms, restricted to simple is-a patterns (e.g. "A human is a mammal.").

- **Basic English** [71]: This system was first developed in the 1930s, prior to the development of the modern computer. It is very lightly constrained and therefore difficult to parse by a machine, but it is still over-constrained for our purposes (a sentence like "Look at the red box," is not valid in Basic English).

- **E-Prime** [72]: E-Prime is very similar to regular English, but the verb "to be" is banned. It is too broad to be easily parsed by a machine.
- **Caterpillar Fundamental English (CFE)** [73]: CFE does not have a strict definition, but is instead defined via guidelines such as "use uniform sentence structures," and its authors themselves admit its rules are difficult to enforce.
- **FAA Air Traffic Control Phraseology / AirSpeak** [74]: A set of phrase templates used by air traffic controllers. Heavily restricted, and defined by 300 template phrases selected to be easy to understand in a high-risk context where not all parties to a conversation are native English speakers. Regular English is allowed to supplement AirSpeak where no existing phrase template exists. AirSpeak consists of only Air-Control specific phrases, but its usage proves the surprising practical usefulness of templated phrases in a restricted domain.
- **ASD Simplified Technical English (ASD-STE)** [75]: Specified by 60 broad rules, and is intended for clear communication by humans more than for computer consumption.
- **Standard Language (SLANG)** [76]: Developed by the Ford Motor Company, SLANG is designed to be a machine-checkable language for instruction manuals. Its grammar is focused exclusively on imperative mood verbs.

- **SBVR Structured English** [77]: Similar to SQL, and is made for describing business rules, not for commands.
- **Drafter Language** [78]: Based on filling in blanks in existing phrasal templates.
- **E2V** [79]: Focused on specifying facts, similar to ACE.
- **Formalized-English (FE)** [80]: Focuses on knowledge representation.

There are many more CNLs in existence ([68] has approximately a hundred listed), many of them domain specific. For the purpose of specifying CPs, we require several features: the ability to issue commands for the experimental subject to perform, the ability to control the flow of execution (via conditional statements and loops), and the ability to state facts or describe events that are about to happen.

We can see from the above overview that many of the CNLs listed do not have the ability for specifying commands, and those that do have such ability lack either control flow statements or the ability to state facts. One might suggest that a possible approach to providing natural language capability to CP specifications would be to adapt one of the above CNLs and add the missing features. Unfortunately, the effort involved in such additions and adaptation to a new domain is likely to be greater than the effort required for developing our own custom CNL, which we can design to have exactly the set of features CPs require and nothing more.

1.4 Objective of Work

From the literature review, we have seen the major components of the STAR and the role of Cognitive Programs within it. The purpose of this thesis is to provide a method for translating task descriptions written in natural language into executable Cognitive Program scripts. In the context of STAR, this will be done by a component of the visual Task Executive called the Cognitive Program Compiler (CPC) which takes Cognitive Program Descriptions (CPDs) as input and composes executable scripts out of methods stored in the CPM tuned with task-specific parameters. In the context of TarzaNN 3, the Cognitive Program Compiler is an implementation of a compiler that takes CPDs written in natural language annotated with code and translates them into executable code that TarzaNN 3 can use to execute visual attention experiments.

The CPD format needs to provide both endogenous information (instructions to STAR), and exogenous information (about the environment) in natural language. The remainder of the literature review spoke about various natural language interfaces and programming paradigms that we can study for ideas on how to specify Cognitive Programs in an easy and convenient – yet rigorous – way using natural language. The optimal route seems to be to construct a Controlled Natural Language that will have a simple structure and be unambiguous like a regular programming language, yet be easy to understand by both humans and machines. This allows us to:

- enjoy greatly simplified parsing,

- have a clear mapping between parse tree and ontology,
- allow for extensibility via a single Lexicon that holds both the dictionary and ontology,
- resolve anaphorae and conjunctions via simple procedures, and avoid the ambiguity plaguing some natural language programming languages,
- lose very little or no expressive power of regular natural language due to restricted vocabulary and syntax, and
- have an easily implementable error system to help guide the user.

Taking ideas from the discussion of CNLs in Section 1.3.5, we can design the natural language component of CPDs as a novel CNL called Imperative English (IE), which is described in Chapter 2. It is simple, unambiguous, easily understandable by humans, extensible, and specifically tailored for issuing commands and stating facts about the world. IE is incorporated into Cognitive Program Descriptions, the CP description format which allows for complete specification of computational tasks, suitable for parsing and translation into scripts that can be executed by a visual Task Executive as part of a simulator of STAR, like TarzaNN [17] in Chapter 3.

Since the TarzaNN 3 implementation is not complete, evaluation will be based on implementing several real-world visual attention experiments as CPDs and a discussion of the output code. A mockup program is also provided that logs the sequence of instructions that would be issued to TarzaNN 3 when executed.

1.5 Significance and Contributions

When Shimon Ullman first described Visual Routines in 1984 [16], he did not propose a way of how a VR could be specified. Subsequent work building on Visual Routines also did not propose or implement any way of specifying them, and this thesis is poised to resolve this long-standing problem. It will allow researchers to specify Cognitive Programs in a natural and easy to use way. It has the potential to enable rapid iteration of experiments via a standardized high-level system, as opposed to the typically slow progress using various hand-coded systems. It will also serve as a valuable contribution to the final implementation of the STAR model in the future.

Furthermore, CPD and IE themselves have the potential to be significant beyond just the STAR model and Cognitive Programs. CPD can serve as a general-purpose experiment description format that is unambiguous, and yet easily understandable by both humans and machines. IE is easy to use and extensible to arbitrary commands, and therefore suitable for Natural Language Programming in various domains, as well as facilitating communication with machines via natural language.

Chapter 2

Imperative English

2.1 Overview

To construct CPDs, the first thing we need to do is precisely define their controlled natural language component. Such a language needs to be simple, unambiguous, easily understandable by humans, yet easy to parse for a machine, and extensible. It should be able to, at a minimum, serve two functions: issuing commands and stating facts.

The novel controlled natural language constructed for the purpose of this thesis is called Imperative English (IE), so named after the imperative grammatical mood it heavily leverages to issue commands and provide instructions. IE is shown in this chapter to be a Turing complete language, making it at least as powerful as any other programming language.

IE's design was influenced by an observation from the literature review that a natural language system with a carefully designed, but restricted, grammar will often be more functional in a constrained domain than a complex system that attempts to understand general English using sophisticated techniques. This is especially true in light of natural language understanding still being an unsolved problem. For example, relatively simple systems like ELIZA or Inform 7 which have been explicitly designed to work within the limitations of their domains have seen more success in the real world than dramatically more complex systems like Metafor or NCLI which attempt to be general.

Motivated by that observation, an explicit choice is made not to use general-purpose NL parsers like CoreNLP, but to impose stricter structure at the parser level akin to a CNL. That frees us from having to take into account an imperfect parser that may produce inconsistent part-of-speech tags for input sentences. It also frees us from having to compute a mapping between part-of-speech tags and IE concepts, since such a mapping can be built into the CNL itself.

Constructing IE as one would a traditional programming language also allows us to leverage an enormous amount of research and engineering experience in compiler theory to find solutions to a wide variety of problems. For example, we could create a helpful error reporting system to guide the user with suggestions in the cases where the user employs ordinary English constructs in their CPD that do not occur in Imperative English.

However, several rather obvious questions about this approach arise. How can one ensure that a constrained natural language constructed that way is general? Would a traditional compiler structure impose restrictions on the domains and usability of the language? Would it be difficult for a user that knows conversational English to use IE due to its restricted nature?

The literature review provides answers to some of those questions. For example, [30, 31] note that using constrained English does not significantly impair a person's expressiveness, as long as they know which structures are valid. IE aims to define its structure in a handful of simple guidelines that are easy to follow. As for generality, portability across domains is often achieved via an ontology that can be built for new domains. To this purpose, IE proposes the Lexicon: a combination of an ontology and dictionary. To port IE to an arbitrary new domain, all one would have to do is construct a new Lexicon.

In this chapter, the grammar of IE is discussed in Section 2.2, the Lexicon is introduced in Section 2.3, its Turing completeness proved in Section 2.5, and a reference implementation of an IE interpreter called *Imp* is presented in Section 2.6.

2.2 Grammar

Unlike natural English (a language equally suited for poetry and for legal documents), Imperative English only needs to be general enough to issue sequences of commands and facts, and does not need to recognize many other more complicated grammatical structures. This allows us to build a constrained enough grammar to make the problem of parsing English manageable.

How will a user be able to tell which sentences are valid IE and which ones are not? This is an essential prerequisite to making IE practical, as anything else would very quickly lead to user frustration. To specify IE's core in the simplest terms possible, the following three rules are provided:

1. To issue a command, start the sentence with a verb.
    - Example: "**Look at** the red circle."
2. To state a fact, start the sentence with an object phrase followed by a verb.
    - Example: "**The red circle is** in the centre of the screen."
3. To specify conditions or repetition of actions, start sentences with one of four control flow keywords: If, When, While, or For.
    - Example: "**If** the square is red, push the button."
    - Example: "**When** the square turns red, push the button."
    - Example: "**While** the square is red, hold the button."
    - Example: "**For** every red circle, push the button."

The above three rules are all one needs to write mostly-valid IE. The remainder of this chapter will discuss the specifics of the IE grammar, which has three levels of organization: *Sentences*, which are composed of *Phrases*, which are composed of *Fragments*. The remainder of this section covers those different levels of organization.

2.2.1 Fragments

We begin our overview of IE grammar with its smallest part: the *Fragment*. Every word in an IE text is either part of a Fragment or one of a handful of keywords. The majority of Fragments correspond to individual words, such as “red” or “is”, but sometimes they may correspond to multiple words that form a single indivisible Fragment, such as “Look at”. IE recognizes four kinds of Fragments: *Verbs*, *Targets*, *Modifiers*, and *Relationships.* The kinds of Fragments IE recognizes will vary depending on the domain, and are defined by the Lexicon (Section 2.3).

Verbs

Verbs correspond to the grammatical part of speech of the same name, which conveys actions, occurrences or states. They can play one of two roles in a sentence: *Imperative Verbs*, or *Fact Verbs.*

Imperative Verbs are Verbs in the imperative grammatical mood, such as “look at,” “wait,” or “go.” If they occur in a Sentence, they always occur at its beginning. Imperative Verbs always specify the kind of action that should be executed as a result of the IE instruction that contains them.

Fact Verbs are Verbs that do not occur at the beginning of a Sentence. They describe relationships or states, and are named “Fact” Verbs because they occur in Sentences that describe the state of the environment or the imperative subject.

Targets

Targets are a kind of Fragment that represents a grammatical object or subject in a sentence that Verbs can execute their actions on. Commonly, they are nouns or pronouns, such as “square”, “name”, “it”, or “they”. Plain strings or numbers can also be targets, for example `"/tmp/images/image.png"`, or 11.5.

Modifiers

Modifiers modify Targets. They can describe various properties, states, locations, or kinds of Targets. Examples are “red”, “leftmost”, or “eleven”.

Relationships

Relationships are words that have a similar role to prepositions in English. They allow prepositional phrases that modulate the execution of a Verb or the properties of a Target. Examples are “with”, “containing”, “to”, or “of”.

2.2.2 Phrases

The next level of hierarchy in the IE grammar are Phrases. Phrases are groupings of one or more Fragments, and they form the basic units of meaning of which Sentences are composed. There are three kinds of Phrases: *Verb Phrases*, *Selectors*, and *Specifiers*.

Verb Phrases

Verb Phrases are the simplest of the three kinds of Phrases, as they consist only of a Verb and an optional auxiliary verb. Auxiliary verbs are not treated as Fragments, but as keywords due to their limited number and limited semantic value. Examples of Verb Phrases are “move” and “will appear”.

Selectors

Selectors are based on Targets augmented with one or more Modifiers. They can represent objects in the environment (“leftmost red filled rectangle”) or the imperative subject itself (“you”). Usually, they evaluate to a set of one or more items, for example “red rectangle” would be interpreted as referring to the set of available red rectangles, and the IE interpreter would use context to decide whether a set or a single element of that set is required.

Specifiers

Specifiers consist of a Relationship and a Selector, and perform a role similar to prepositional phrases in English. They can apply to Verb Phrases or Selectors to modify their properties or methods of execution. For example, the Relationship “with” can be used to modulate a Target with another Target (“a box **with blue sides**”), and the Relationship “to” can be used to provide a Verb with additional information on how it should execute an action (“Set the counter **to zero**.”).

2.2.3 Sentences

The highest level of IE grammar organization are Sentences. Sentences may be composed of Phrases and other Sentences. In that respect, IE Sentences are similar to the grammatical notion of clauses, however, IE uses the word “clause” in the context of Facts, and uses “sentence” here instead to avoid ambiguity.

Commands

Commands are Sentences that begin with a Verb Phrase in the imperative mood, and their purpose is to specify actions. They have a Selector, which is the object to which the imperative Verb applies, and some optional Specifiers, which describe how the Verb applies to the Selector, or modify the Selector in some way.

This design is inspired in part by SLANG [76], and in part by interactive shell command conventions in the UNIX operating system [81]. Take, for example, the following shell command for compressing the contents of the directory `/home/user/photos` into an archive file `photos.tar`:

```
tar --create --file photos.tar --verbose /home/user/photos
```

Shell commands like the above tend to begin with the command name (Verb-like `tar`), followed by zero or more flags with or without arguments (Specifier-like `--file photos.tar` and `--create`), and an argument file or directory (Selector-like `/home/user/photos`).

As an example of a Command in Imperative English, consider the sentence "Look at the red triangle on the blue square.". In this sentence:

- "Look at" is the Verb describing the action to be undertaken,
- "the red triangle" is the Selector describing the object of that action, and
- "on the blue square" is the Specifier applied to the Selector. It consists of the relationship "on", and the Selector "the blue square".

We can express the grammatical structure of Commands and other parts of the IE grammar as grammatical rules, such as:

```
command
  : imperative_verb specifier* selector? specifier*
  ;
```

The above is an ANTLR [82] grammatical rule that specifies the structure of a Command. ANTLR is a modern parser generator: a tool that takes as input a text file containing grammar rules that describe the structure of a language, and uses that input to generate executable code that can parse text into a parse tree according to those grammatical rules. ANTLR grammars are based on the Extended Backus-Naur Form (EBNF) [83], and in that context the above rule means: a `command` is a rule matched by an `imperative_verb` that is followed by `specifier*` ("zero or more occurrences of a specifier"), which is followed by `selector?` ("zero or one occurrence of a selector"), and another `specifier*`. Each of `imperative_verb`, `selector` and `specifier` have their own rules that define their own structure. The full ANTLR grammar for IE can be seen in Appendix A.

Facts

Facts are the second kind of IE sentence. Facts, similarly to Commands, are also composed of Selectors, Specifiers and Verb Phrases. Note, however, that non-imperative mood Verbs are used in Facts. Facts have a dual purpose in IE:

1. Describe facts and changes in the world that are tautologies, unconditionally evaluated as true. These Facts are always standalone Sen-

tences.

  - Example: “An image showing five blue rectangles will appear on the screen.”

2. Logical statements that, when evaluated, result in a boolean value: true or false. One can think of these as being tested for fact-hood at evaluation time, returning true if the Fact is a fact, and returning false if it is not a fact. This kind of Fact is always part of another Sentence, such as conditional control flow Sentences using If, When and While.

  - Example: “If **you see a red circle**, press the button.”

Let’s analyze the above standalone Fact “An image showing five blue rectangles will appear on the screen”:

- “An image” is a Selector,
- “showing five blue rectangles” is a Specifier applying to the above Selector,
- “will appear” is a Verb Phrase containing the auxiliary verb “will”, and
- “on the screen” is a Specifier applying to the Verb.

Facts consist of *Fact Clauses*, which can be composed via logical operations `AND` and `OR`, for example “The shape is a rectangle **or** the shape is a square.” The logical `NOT` can be expressed via Modifiers and Verbs, for example to negate “the image is blue,” one would say “the image is not blue,” and to negate “you see a square” one would say “you see no square.”

The basic ANTLR grammar rules governing Facts are:

```
fact
  : factClause AND fact
  | factClause OR fact
  | factClause
  ;

factClause
  : factClauseLHS factVerb factClauseRHS
  ;

factClauseLHS
  : selector specifier*
  ;

factClauseRHS
  : specifier* selector? specifier*
  | modifier
  ;
```

In ANTLR, the pipe sign (`|`) separates alternatives that the rule can match, so a `fact` rule could match either `factClause AND fact`, or `factClause OR fact`, or a single `factClause`. Note the recursive definition, where a part of the alternative that `fact` matches can be another `fact`, to allow for chaining many `factClause`s together. RHS and LHS stand for “right hand side” and “left hand side”, respectively.

Control Flow

Control flow statements, in computer science, are statements that can decide the order in which instructions in a program get executed. This includes deciding the next instruction to be executed based on a boolean condition, or looping: repeating a block of instructions multiple times. IE has four kinds of Control Flow Sentences: If statements, When statements, While loops, and For loops. The Böhm-Jacopini theorem, also known as the structured program theorem [84], shows that If statements and While loops are sufficient for programming language control flow, however IE also defines When statements and For loops for convenience of programming and for the natural kinds of expressions they allow.

If Statements can decide to execute different blocks of instructions based on a boolean condition. Their basic structure is:

```
ifStatement
  : IF fact instructionBlock (ELSE instructionBlock)?
  ;
```

To enable natural expression, both “else” and “otherwise” match the `ELSE` rule. The rule `instructionBlock` matches a list of instructions (Commands, standalone Facts, or Control Flow) chained together by the keyword “then”. For example:

```
If you see a red circle,
  press the left button, then
  press the middle button
else,
  press the right button.
```

Indentation and newline characters have been added to aid readability. If the Fact “you see a red circle” evaluates to true, then the left and middle buttons will be pressed, otherwise if the Fact evaluates to false, the right button will be pressed.

When Statements have a similar structure to If statements:

```
whenStatement
  : WHEN fact instructionBlock
  ;
```

The main difference in the execution of If and When statements is that When statements wait until the `fact` is true before executing the `instructionBlock`, whereas the If statement will evaluate the `fact` immediately and make its decision on which instruction block to execute.

While Loops are the first of two kinds of loop that IE recognizes:

```
whileStatement
  : WHILE fact instructionBlock
  ;
```

The `fact` will be evaluated, and if true, the `instructionBlock` will be executed once. That procedure will be repeated until the `fact` evaluates to false.

For Loops are the second kind of loop in IE. As opposed to While loops, which are condition-controlled, For loops are collection-controlled: they execute the instruction block once for every element of a collection of items. Their structure is:

```
forEachStatement
  : FOR EACH selector specifier* instructionBlock
  ;
```

For example: "For every plate in the stack of plates, wash the plate, then dry it, then put it into the cupboard." This statement iterates across the set of items defined by the Selector and Specifiers "plate in the stack of plates”, and the instruction block consisting of the three commands “wash the plate, then dry it, then put it into the cupboard” is executed once for every element of the set.

2.3 Lexicon

The Lexicon represents the dictionary and ontology of IE. It consists of a hierarchy of Fragment entries, where every Fragment is the child of another Fragment (for example, the Red Fragment might inherit from Color, which in turn might inherit from Modifier, which inherits from Fragment, which is the root Fragment). Each Fragment consists of:

- **Dictionary Strings** – A set of strings that this Fragment matches in the input text. Different Fragments can share the same dictionary string, as long as they are different kinds of Fragments (for example, a Lexicon can contain without conflict both a Target fragment or a Verb fragment that match the string “push”, and both a Target and a Modifier that match the string “fixation”). This set may be empty for an abstract Fragment, such as Color.
- **Fragments Accepted** – A set of the kinds of Fragments that this Fragment accepts. For example, Verbs would have a list of Targets or Relationships they recognize and can perform actions with. Due to the hierarchical organization of the Lexicon, one does not need to be exhaustive in creating entries for each individual Fragment. Instead of saying that a Square Target can accept the Modifiers Red, Blue,

Green, Violet etc., one can simply state that Square accepts Color, the parent Fragment of all colours.

- **Implementation Code** – Code that specifies what kind of output this Fragment instance should generate in response to Fragments it has accepted.

To illustrate how a Lexicon would be consulted when parsing a phrase, let us take a look at how the IE sentence “Look at the red box without blinking” would be decomposed:

- “Look at” is a Verb that accepts:
  - “box” – a Target that accepts the “red” Modifier, with which it forms the Selector “the red box”.
  - “without” – a Relationship that accepts the “blinking” Target, with which it forms the “without blinking” Specifier.

The Lexicon allows us to check IE sentences for semantic meaning (in effect, this allows an IE interpreter to act as if it had common-sense knowledge), and to generate code from those sentences. New entries can be added to the Lexicon at any time without having to change the rest of the system.

A question that may be bothering the reader is: How does one maintain a huge interrelated Lexicon of terms? Does maintaining this structure become problematic as it grows? May it introduce inconsistencies?

The answer is, unfortunately, that there is no simple solution. We are, after all, trying to represent knowledge in language, and projects tackling a similar organization of meaning or knowledge, like WordNet [63] (117,000 sets of synonyms), or Open Mind Common Sense [55] (over a million facts from 15,000 contributors), are fairly huge. Fortunately, most natural language applications today are domain-specific, and implementing Lexicons on the order of a hundred entries should be sufficient for many applications (including the purposes of this thesis), and should not be a daunting task. Most kinds of inconsistencies that a Lexicon might have can be statically checked. For example, the non-intersection of dictionary string sets among Fragments of the same kind, or ensuring the acceptance of Fragments in accord with IE grammar rules. New additions to the Lexicon should not change the way old Lexicon entries behave as long as the acceptance lists of the old entries have not changed.

2.4 Interpretation

After outlining the IE grammar and how the Lexicon works, a couple of interpretation questions remain undiscussed. The following is a list of some of the more obvious ones, as well as strategies an IE interpreter should employ to address them.

Vagueness – It is not realistic to expect a natural language system user to exhaustively specify every possible parameter, so at some stage (like in human communication), a certain amount of guesswork is necessary and warranted. Lexicon entries generally should not count on accepting non-essential Lexicon entries, and should instead assume some sensible set of defaults that the user can then override with more exact specification. For example, the entry “Bake a cake.” should not throw an error because the user did not specify what kind of cake they want, but a reasonable default (e.g. chocolate cake) should be assumed.

Anaphora – A way to resolve pronouns and references is necessary for resolving Targets like “it” or “them”, or elliptical sentences that require context for resolution. Anaphora can only validly occur within Selectors in IE, so a practical resolution strategy is to walk the parse tree backwards (relative to the usual depth-first traversal order), and find the first previously mentioned Selector that can be accepted by the current context. If such a Selector cannot be found, an informative error should be thrown.

Conjunction – The conjunctive “and” is sometimes used to mean (in set-theoretic terms) union and other times intersection, as discussed in the context of NLIDB in the literature review. IE resolves this by providing an And Relationship that matches strings “together with” and “along with,” and a separate keyword “and” that connects Fact Clauses and performs the logical AND operation on the evaluation results of those clauses. The interpretation of the And Relationship should be resolved at the level of Lexicon implementation code for applicable entries. Any Target Fragment has the default behaviour of unioning the set it represents with the set of the Target accepted by the And Relationship. This default behaviour can be overridden by Target Fragments that require intersection. Only a small part of the Lexicon should have to override this Relationship interpretation.

Specifier attachment – Is a specific Relationship Fragment accepted by a Verb or a Selector? Which one? Specifier attachment is easy to resolve via the Lexicon: the Specifier attaches to the closest Verb or Selector to its left in the parse tree that can accept it.

Unknown Syntax – How should an IE interpreter handle the user providing uninterpretable input text? As seen in the literature review, for a domain-specific or constrained natural language interface, a detailed and helpful error reporting system is essential to provide feedback and a frustration-free learning opportunity to the user. The system should provide gentle guidance based on what it can infer about the uninterpretable input. Some error reporting strategies are discussed in Section 2.6.3.

2.5 Turing completeness

Turing completeness (also called computational universality) is an extremely desirable property for a programming language to have [85]. This section demonstrates that IE is a Turing complete language.

For a programming language to be Turing complete (TC) means that it can be used to write any algorithm, and therefore used to perform any possible computation. A way of stating this is that the programming language can simulate a Turing machine [86].

A Turing machine is an abstract theoretical construct invented by Alan Turing in 1936 as a mathematical model of computation. Turing machines manipulate symbols on an infinite strip of tape, using a tape head to read or write one symbol at a time, and a finite set of instructions to an imagined "computer" who in Turing's original formulation is not a mechanical contraption, but an obedient human who executes instructions depending on the symbol under the tape head and the current state. Turing goes on to hypothesize that such a machine can compute any calculable function[1], which implies that it can execute any algorithm, making it a universal model of computation.

The primary goal of Turing in constructing this machine (which he called an "automatic machine", or "a-machine") was to provide an answer to Hilbert's Entscheidungsproblem [88], which asked whether there exist an algorithm that can prove or disprove a given logic statement based on a set of axioms. Turing shows that the answer to that question is no, based on his proof of undecideability of the *halting problem*: one can not build a Turing machine that can answer the question of whether another Turing machine will finish running (halt) or run forever.

This section shows that Imperative English is a Turing complete language, but we should immediately note one caveat: no particular *implementation* of any computer programming language can be shown to be truly Turing complete. This is because the formulation of a Turing Machine assumes infinite memory, which is not strictly true in the real world, if for no other reason than there being a limited amount of matter and space in the known universe to make memory elements out of. This fact, however, is usually only of theoretical concern.

---

[1] This hypothesis is also known as the Church-Turing thesis, as Church independently developed a similar formalism of equivalent power in [87].

Likewise, most computational facilities we take for granted today are not necessary for a Turing Machine to function as a universal computation device. For example, input-output facilities are unnecessary for computation. In a way, input is the starting state of the tape and output is the state of the tape when the halting symbol is printed on the tape.

In practice, it is very hard to write a non-Turing complete programming language (famously, some implementations of SQL are not Turing complete, while others are [89], many of the latter group without explicit intention on part of their designers). In fact, IE is in a good position to be assumed to be Turing complete, due to the results of the Structured Program Theorem [84], which implies that a programming language capable of sequential execution of instructions (a block of code), some sort of selection mechanism (if-statement), and a looping construct (while-loop) will likely be Turing complete. One could also provide an alternate argument that the underlying language of implementation for Fragments in the Lexicon – for example, Java – is Turing complete, and that since Fragments are allowed to execute arbitrary code, Java's Turing completeness is sufficient for us to implement a Turing complete IE program via Turing complete Fragments.

IE has all those things, however, one should note that we still require a proof before being able to say that IE is Turing complete. So we proceed as follows.

We need to show that for every possible computation that a Turing Machine can perform, there exists a corresponding IE program that, when executed, performs the same computation. In effect, this means that we could use IE to simulate a Turing Machine, and compute any algorithm using IE. Nothing is said of efficiency of the computation, just that it can be done.

To make this job easier, we'll try to simplify the simulation. One way to do that is to find a simple system that has been proven to be Turing complete and attempt to simulate that system. There are numerous such systems, for example the Rule 110 automaton [90], or Corrado Böhm's $\mathcal{P}''$ language [91]. We select the Minsky Machine (MM) as our target, a kind of counter machine introduced by Marvin Minsky in 1967 under the name Program Computer [85]. The Minsky Machine is known to be Turing complete, and features:

- Five registers, each of which can hold an unbounded number greater than or equal to zero.
- A set of three instructions that operate on those registers:
  - $\boxed{\texttt{0}}$ – set a register to 0, e.g. a^0,
  - $\boxed{\texttt{'}}$ – add 1 to a register, e.g. a',
  - $\boxed{\texttt{-}}$ – subtract 1 or transfer if already 0, e.g. $a^-(n)$, where n is the instruction to jump to.

The Minsky Machine also assumes that instructions are executed in order (taking into account the jump instruction), and that, consistent with Turing Machines, there is a halt state.

Minsky proves in [85] the existence of many other counter machines, the smallest of which has as little as one counter and two instructions. However, these machines impose some complex restrictions (notably, the use of

Table 2.1: Mapping from Minsky Machine commands to Imperative English commands.

MM	IE	Description
a^0	Set register labelled a to 0, then increment the register labelled "pc".	Sets register a to 0.
a'	Increment register labelled a, then increment the register labelled "pc".	Increments register a.
$a^-(n)$	If value of register labelled a is 0, set the register labelled "pc" to n, else decrement register labelled a, then increment the register labelled "pc".	Jump to instruction n iff register a is zero, otherwise decrement it.

Gödel encodings for counter values), which would make the proof unnecessarily more complicated, when compared to the more "ordinary" and easily understandable version described above.

To prove that IE is Turing complete, we show that for all possible Minsky Machine programs, there exists an equivalent IE program. In other words, we show that Minsky Machines are IE-complete. We begin by showing for every Minsky Machine command the equivalent IE command in Table 2.1.

One important thing to note is that while Table 2.1 translates MM's jump instruction into valid IE, this is not the entirety of the generated program. The correct translation of the jump instruction necessitates some additional machinery. The generated instructions are embedded in a template program containing a while loop and a program counter variable to account for the fact that IE uses if-statements and while-loops, whereas MM only uses jump instructions for control flow. Luckily, the structured program theorem [84] proves that one can trivially flatten if-conditionals and loops into jump statements, and expand jump statements into if-conditionals and loops. The result is ugly, but guaranteed to work, and can be seen in Listing 2.1, showing the translation of a short MM program of three commands 3^0, $3'$, $3^-(1)$ to IE. The variable "pc" is the program counter.

Listing 2.1: Translation of a short MM program of three commands 3^0, $3'$, $3^-(1)$ to an IE program.

```
Set the register labelled "pc" to 0.

While 1 is 1,
  if the value of register labelled "pc" is 0,
    # $3^0$
    set register labelled 3 to 0, then
    increment the register labelled "pc";
  else if the value of register labelled "pc" is 1,
    # $3'$
    increment the register labelled 3, then
    increment the register labelled "pc";
  else if the value of the register labelled "pc" is 2,
    # $3^-(1)$
    if value of register labelled 3 is 0,
      set the register labelled "pc" to 1,
    else,
      decrement the register labelled 3, then
      increment the register labelled "pc";
  else if the value of the register labelled "pc" is 3,
    return 0.
```

For IE programs constructed according to the above to work, the Fragments used have to be implemented in the Lexicon in a particular way:

- Register – Target that represents a counter with an integer value, accepts Relationship "labelled" to select a specific counter out of the ones in memory.
- Value – Target that accepts Relationship "of."

- Integer – Target that holds numbers.
- Increment/Decrement – Verb that accepts a Register and generates code to increment or decrement its integer value by one.
- Set – Verb that accepts a Register and a Relationship "to," and sets the Register to the Integer accepted by the Relationship.
- Is – Verb that returns a boolean value representing the equivalence of the Targets on its left hand side and right hand side.
- Labelled – Relationship that accepts an Integer.
- To – Relationship that accepts an Integer.
- Of – Relationship that accepts a Register.
- Return – Verb that accepts an Integer return value and terminates the execution of the IE program when called.

This procedure allows us to convert any MM program to an IE program by translating individual instructions according to Table 2.1, and embedding them in a template like in Listing 2.1, with the loop expanded to accommodate the number of instructions in the MM program. Thereby, we have shown that IE is Turing complete.

2.6 Implementation

> "In the computer field, the moment of truth is a running program; all else is prophecy." – Herbert Simon

This thesis, in addition to defining IE, provides a reference implementation of an IE interpreter called *Imp* that can execute Imperative English programs. A sample Lexicon is also provided. An overview of *Imp*'s implementation follows.

Imp is implemented in the Java programming language [51], and its Lexicon outputs code in Python [92]. Java was chosen because it is a portable, statically typed language with extensive library support. Python was chosen as the output language because it is dynamically typed, and thus forgiving to write and easy to read, as well as widely used in the scientific community.

2.6.1 Lexicon

Imperative English is designed with extensibility as a first-order concern. This is because even though the *Imp* reference implementation provides a core set of Fragment implementations, these cannot hope to cover everything an *Imp* user might potentially want to do. Therefore, for IE to be of any practical use, a user must be able to provide their own entries to the Lexicon.

The question arises of how one should structure the Lexicon implementation. Fragment definitions should be available at every stage of the compiler, so one wise design decision would be to build the Lexicon as completely separate from the compiler implementation: no Fragments should be built into the compiler itself. Another good idea would be to keep all the information about a single Fragment in the Lexicon (the set of strings it matches, the set of Fragments it accepts, how it generates code) inside a single file.

The Lexicon is implemented as a hierarchy of Java classes. Java is an object oriented language, and in that context classes are code templates for creating *objects*, each of which can have *fields* containing data and *methods* that describe procedures one can execute on the objects. Java classes may inherit from one another. For example, a class Red might inherit from the class Color, which in turn might inherit from the class Appearance. This greatly reduces the amount of repetition in code, and we can heavily leverage this hierarchical nature of Java classes in implementing Fragments. In our case, every Fragment will be its own Java class. That allows, for example, a Target to accept a Colour, instead of implementing the acceptance of each one of its children separately.

Fragments are implemented in Java instead of IE because specifying low-level detail in a high-level language tends to be cumbersome (consider the example of Inform 7 extensions written in Inform 7 [57] discussed in the literature review). An added benefit to using Java is the extensive library support it provides. This means that if one would like to write a novel IE Verb, for example "Download", as in "Download all pdfs from "http://jtl.lassonde.yorku.ca"," one would have at their disposal existing Java libraries that know how to search websites and download files from them, a task which would be a lot harder to accomplish in IE on its own.

Imp puts some restrictions on the structure of Lexicon entries. For example, Targets are assumed by the rest of the Lexicon implementation to always return code that evaluates to a collection. There is no simple way to ensure this at Lexicon-compile time, but it is anticipated by the system: failure to do so would result in non-functioning code.

The standard Lexicon included with *Imp* includes 53 Fragments, most of them suitable for common automation tasks (e.g. Download and Extract

Verbs, File and Directory Targets), and the remainder relevant to the Turing completeness program in Section 2.5. A list of implemented Fragments and their relationships is in Appendix A. The basic structure of a Fragment class is:

- `void accept(Fragment s) throws FragmentAcceptanceException`
  - Validates and accepts a given Fragment or throws an Exception.
- `String[] dict`
  - An array of strings that this Fragment should match.
- `String generate()`
  - Generates code based on the Fragment implementation and accepted Fragments.
- `String generateHeader()`
  - Generates imports and other top-of-file code.

The basic Fragment types (Target, Modifier, Verb and Relationship) inherit directly from Fragment. Every Lexicon implementation should contain these Fragment types, and all other Fragments in the Lexicon should inherit from these basic Fragment types or their children. So to implement a new Target Fragment, one would have to create a new Java class that inherits from Target and implement the above methods and fields: the acceptor method, the dictionary of strings this Fragment should match, and the two code generation functions. Everything else is a matter of the developer's imagination, and one is free to leverage as little or as much of Java functionality to implement their Fragments as one wants.

2.6.2 Compiler Stages

Compilers are usually structured in stages. This enables one to replace or make improvements to parts of the compiler without having to modify all of it. It also imposes structure on the compiler pipeline, making development and debugging easier. Figure 2.1 contains an overview of *Imp*'s compiler stages. Its pipeline begins with a *lexing* that generates a stream of tokens from the input text. These will correspond to Lexicon strings, keywords and punctuation. Next is the parsing step, which assembles a parse tree out of those tokens based on the rules in the IE grammar. A semantic analysis step enforces semantic structure and detects errors, followed by the generation of an intermediate representation that one can use to generate executable output code. An optional final step can execute the compiled code as it is being generated. The Lexicon is independent from this pipeline, but consulted at virtually every stage.

Lexing and Parsing

The first step of the compiler is achieved by a lexer-parser generated via the ANTLR v4 parser generator [82]. ANTLR was selected among other parser generators because of its modern design that uses a single grammar file to generate both a lexer and a parser. It also provides convenient facilities for traversing parse trees using different programming languages. These

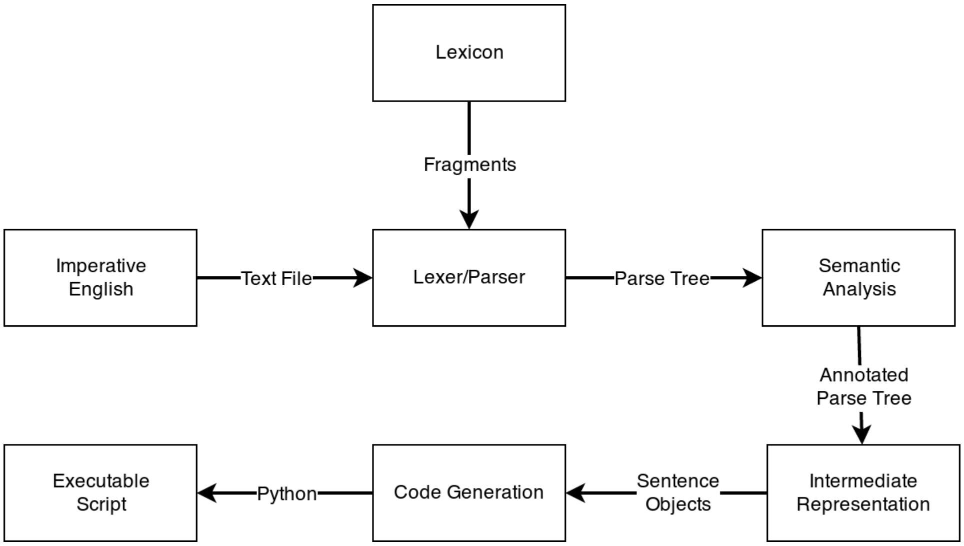

Figure 2.1: Overview of *Imp*'s compilation stages.

facilities come in the form of Listeners and Visitor classes, which are common patterns for "walking" along a tree data structure while executing commands based on one's location in the tree and context.

The lexer-parser generated by ANTLR lexes the input to produce tokens, and then parses the tokens to produce a parse tree. An example of a parse tree resulting from this procedure can be seen in Figure 2.2.

For the lexing step that recognizes individual words as different kinds of tokens, the ANTLR grammar is augmented by the Lexicon. This allows the lexer to dynamically load the Fragment Java classes at runtime using reflection [93] and build a dictionary datastructure that it can consult to infer which Fragment tokens match which words. If an unknown word is encountered, an error message is thrown (Section 2.6.3).

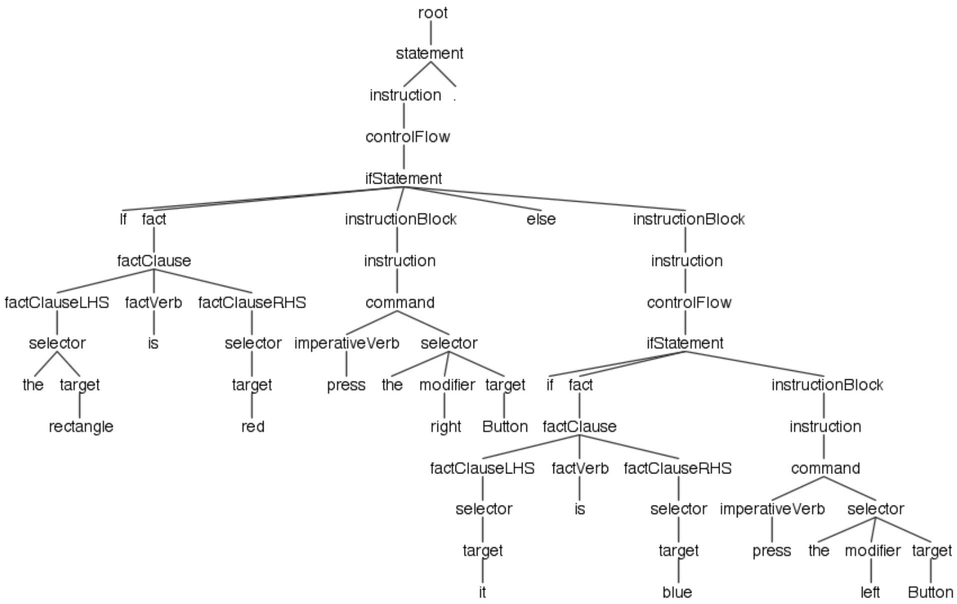

Figure 2.2: Parse tree for an IE If-then-else statement.

Semantic Analysis

The main purpose of the Semantic Analysis stage is to ensure that the parse tree is semantically correct. One can imagine numerous syntactically correct IE statements that would semantically be inexecutable nonsense (e.g. "Push the three four background with any rectangle."). This is done by implementing several Listener classes that will traverse the parse tree searching for errors, for example:

- Fragment attachment: For every Fragment in the sentence, ensure that it is accepted by one other Fragment in the sentence, or the Sentence itself.
- Anaphora resolution: Ensure all anaphora instances are resolvable to Targets.

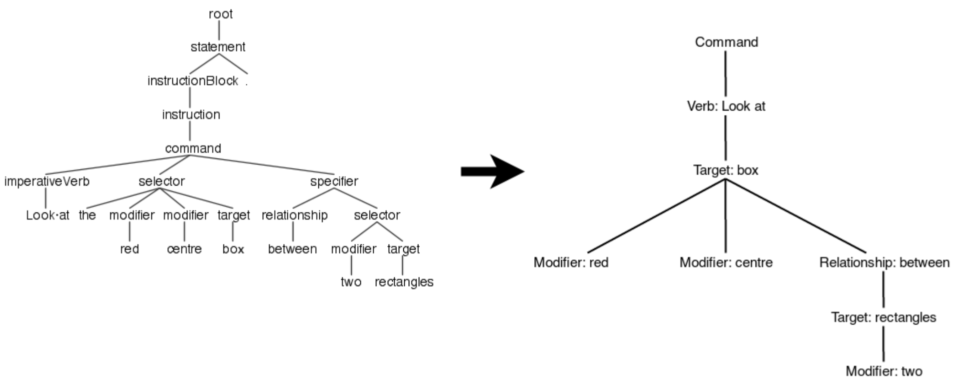

Figure 2.3: Example of how a validated parse tree of an IE Command is transformed into an intermediate representation tree.

The necessary details for ensuring correct Fragment attachments are encapsulated within the implementation of the `accept` method of Fragments in the Lexicon.

Intermediate Representation

After verifying that the parse tree is semantically correct, an intermediate representation is constructed. The intermediate representation is composed of Fragment and Sentence objects, and it is a tree data structure created based on acceptances established during the semantic analysis stage. An example can be seen in Figure 2.3.

Code Generation

The Code Generation stage traverses the top-level Sentence objects, and calls their `generate` methods, which will call the `generate` methods of the Sentences and Fragments they contain, which will in turn call the ones of their Sentences and Fragments and so on, recursively. Every Sentence or Fragment then decides how to compose the code returned by the Fragments it accepted, and passes on the result to the Fragment or Sentence that accepted it. The output of this process is a Python script in the case of *Imp*, but all one needs to do to generate output in different programming languages is replace the `generate` methods of the Lexicon objects and Sentences with ones in a different programming language.

2.6.3 Error Reporting

An important fact that needs to be recognized is that new IE users might get frustrated when writing English sentences that *Imp* cannot interpret because they are not valid IE. Therefore, a robust error reporting system is required. It should provide guidance to the user via helpful error messages that contain suggestions on how to correct the error in a straightforward way. There are several classes of errors this kind of system needs to be able to handle:

- **Dictionary Errors** – A class of errors that arises when the user attempts to use a word unknown to *Imp*, a word whose string is not matched by an entry in the Lexicon. This can be due to one of two reasons: the word represents an unimplemented concept absent from the Lexicon (an instance of a conceptual failure [28]), or it is an unexpected synonym to a known word (a linguistic failure). The former case cannot be helped except by extending the Lexicon, but the latter case is handled by querying

WordNet [63] for synonyms to the unknown word, and if a synonym is in the *Imp* Dictionary, suggesting the known synonym to the user. An example error message for this case looks something like: `Unknown verb `press' in phrase `press the button'. Do you mean one of the following: `push', `click'?`

- **Syntactic Errors** – A class of errors where the user writes a grammatical structure that is not part of IE. This generally leads to a parsing failure, and the error messages given to the user try to refer back to one of the three rules discussed at the beginning of this chapter when suggesting the next step for correction, for example: `I do not understand the phrase `Wait until a square appears on the screen.' Please start Facts with a Selector, and Commands with a Verb.`
- **Semantic Errors** – For example, unclear pronoun references and other errors where the semantic meaning is difficult to discern. This class of errors are handled via messages like: `I do not understand what `it' references in the line `A square and a circle will appear. Click on it.' Do you mean the `square' or the `circle'?`

2.7 Summary

This chapter has defined Imperative English, a novel Controlled Natural language for issuing commands and stating facts that can be used for writing programs in natural language. Imperative English is simple, unambiguous and easy for humans to understand, yet easy for machines to parse. It is readily extensible via the a Lexicon that contains known words and concepts.

This chapter also described *Imp*, the reference implementation of Imperative English, and its implementation of a common Lexicon of 53 Fragments.

Imperative English was shown to be Turing complete, and thus, capable of universal computation and equal in power to traditional programming languages.

Chapter 3

Cognitive Program Descriptions

3.1 Overview

This chapter extends Imperative English developed in Chapter 2 to create the Cognitive Program Description language (CPD). CPD is a Cognitive Program-specific extension to IE, designed to be a human readable, high-level task description language that can be compiled into executable Cognitive Programs, while remaining flexible enough to allow for arbitrary extension to new paradigms that might arise in future research of STAR.

Sammet advocated in [43] for natural language programming, but noted that she has no desire to type mathematical formulae in English, preferring to write them in a standard mathematical notation. Similarly, CPD gives its users the ability to express some ideas as code instead of writing pure IE.

In the context of STAR, CPDs represent task specifications that the visual Task Executive can compile and run (see Figure 3.1). In the context of neural network simulators, like TarzaNN [17], CPDs describe sequences of events that are to be simulated. To specify an entire experiment, CPDs distinguish between *endogenous* instructions that are issued to the experimental subject, and *exogenous* instructions issued to the "experimental rig" (everything in an experiment that is not the experimental subject, for example displays and buttons).

Strictly speaking, only the endogenous instructions form part of the Cognitive Program task specification, but in order to run experiments, additional information is required about the environment, and that information is provided in a CPD by the exogenous instructions.

A reference implementation of a CPD compiler is also presented, and is called the Cognitive Program Compiler (CPC). It reads a CPD and outputs Python code that can be used to run experiments using TarzaNN 3, the successor of the original TarzaNN neural network simulator.

3.2 Structure

CPDs have a dual purpose of issuing both endogenous instructions for the experimental subject (which can be thought of as the CP task specification), and exogenous instructions that control the experimental rig. In TarzaNN 3, the "experimental subject" is a simulation of the STAR model. The "experimental rig" is defined as a simulation of all parts of an experiment that are not the experimental subject: the computer and display that generate vi-

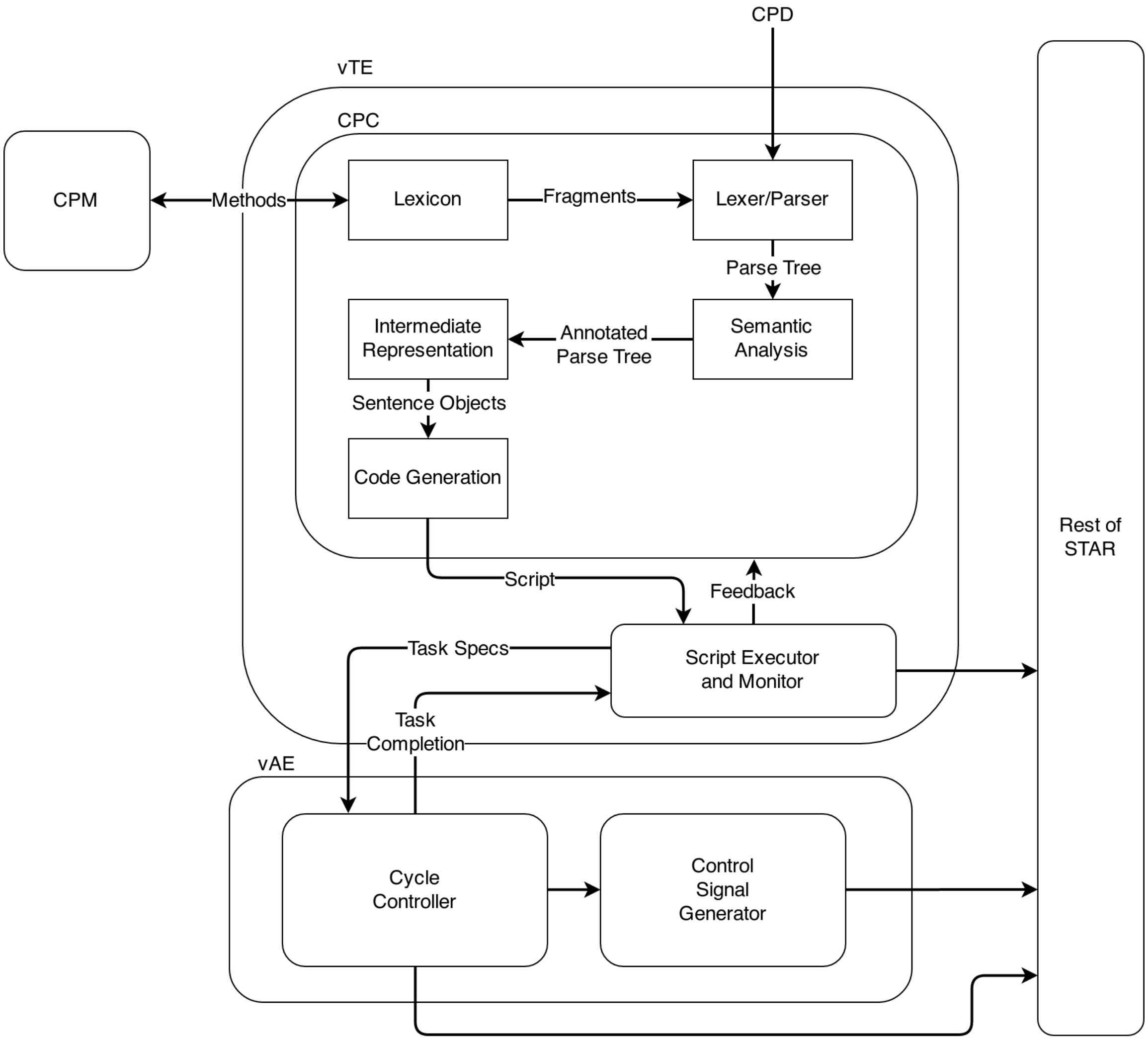

Figure 3.1: Overview of how CPDs fit into STAR.

sual stimuli for the experimental subject to look at, the keyboard or joystick that they provide responses with, as well as various other equipment, such as speakers that can sound auditory feedback. A sample CPD showcasing some of these features can be seen in Listing 3.1.

Listing 3.1: Example Cognitive Program Description of the Egly and Driver 1994 experiment [9].

```
{%
  self.button = self.rig.buttons['LMB']
%}

[%
  self.rig.display_image("img/intertrial.png")
  self.rig.wait(500)
%]

A fixation cross ("img/fixation.png") along with two
↪ rectangles will appear on the screen. Look at the
↪ fixation cross. A cue ("img/cue.png") will appear (1000)
↪ , then it will disappear (100). Please do not look away
↪ when it does.

A filled square ("img/target.png") will appear (200) on the
↪  screen.

[%
  self.button.listen(2000)
%]

When you see the square, press the left mouse button.
↪ Please respond as quickly as possible, as the response
↪ latency will be recorded, but it is important to
↪ minimize the number of errors.

[%
  rt = self.button.wait().result().time
  self.save_result(rt)
%]
```

The first thing one notices are *Code Blocks* surrounded by `{% %}` and `[%`

`%]` that are placed in between IE paragraphs. These exogenous instructions that are intended for execution by the the experimental rig, not the vTE. Their purpose is to set up the experimental environment to provide stimuli and record responses from the experimental participant. There are two kinds of code blocks:

- *Setup Blocks* – A block of code that gets executed only once, usually contains setup code for the experimental rig. Delimited by `{% %}`.
- *Code Blocks* – A block of code that gets executed at every run of the trial the CPD describes to control the experimental rig. Delimited by `[% %]`.

Code blocks also allow CPDs to provide a way to implement new experimental rig features of arbitrary complexity. Chapter 4 contains an example of a juice dispenser being integrated into a CPD. Support for such hardware is essential for CPD's primary audience of vision researchers, who will presumably be continuously devising unpredictably novel experiments. Note that, even though the code in the blocks of the example Listing is Python, CPDs do not put limits on which programming languages their code blocks may contain. This decision only depends on the compiler or interpreter that uses CPDs. The Cognitive Program Compiler, discussed in Section 3.4, uses Python.

The second new feature are *Code Annotations.* They are the snippets of code or parameters in brackets that can be used to annotate IE Fragments and provide them with additional information. For example, when using an "image" Target, one could annotate it with the path of that image so that the Target Fragment knows which image to load. Other examples could be exact delay durations or calls to external classes to dynamically generate visual stimuli for the experiment. The Fragment implementation in the Lexicon specifies if the Fragment accepts annotations or not.

The last, and least interesting, new feature of CPDs are *Please Statements*. Please statements are nothing but comments in disguise, and they do not have to consist of valid Fragments in the Lexicon. They are ignored by the compiler and are defined as an arbitrary string beginning with “Please” and ending with a period. They are only intended to offer additional instruction or clarification to a human reader in a way that does not seem conspicuous in the context of the rest of the CPD. Please Statements do not result in any output code when processed by the compiler – if one wishes to express an idea that should result in code, one should express it as part of a regular IE statement.

An overview of which parts of a CPD are endogenous and which are exogenous is provided in Table 3.1. Commands and Control Flow form the endogenous part of a CPD, since they are used to specify the procedure that the experimental subject or STAR should follow. All code blocks and annotations are purely exogenous, and control the experimental rig. One can use this code to provide the experimental rig with instructions, while keeping them secret from the experimental subject. Facts are an interesting case, as they specify changes that the rig should perform, but that the experimental subject should be aware of and anticipate. This is why Facts can generate both endogenous and exogenous instructions.

One might observe that the self-documenting interweaving of English

Table 3.1: Different parts of CPD, by whether they describe exogenous or endogenous influences.

Endogenous	Exogenous	Both
• Commands • Control Flow	• Setup Blocks • Code Blocks • Code Annotations	• Facts

with code in CPDs has a similar effect on a reader as a program written in the tradition of Knuth's Literate Programming [66], discussed in the literature review. An additional observation is that CPDs are in fact Turing complete, since for a language to be Turing complete, it is sufficient for any subset of it to be Turing complete. The rationale for this is that one would only use that subset of the language in a proof of Turing completeness, so additional features of the language are irrelevant. In the case of CPD, there are two such subsets: Python code ([94] provides a Python evaluator for lambda calculus, which is known to be Turing complete) and IE (shown to be Turing complete in Section 2.5).

3.3 Translation

How can the visual Task Executive (vTE) interpret a CPD like the one in Listing 3.1 and use it to construct an executable Cognitive Program script? As discussed in Section 1.3.2, in STAR the task specification is analyzed by the vTE to select relevant *methods* stored in the Cognitive Programs Memory (CPM), and to extract *parameters* it can use to tune the methods into executable *scripts*. As an example, consider the analysis of the IE sentence "When you see a square, press the button," in Figure 3.2. It shows

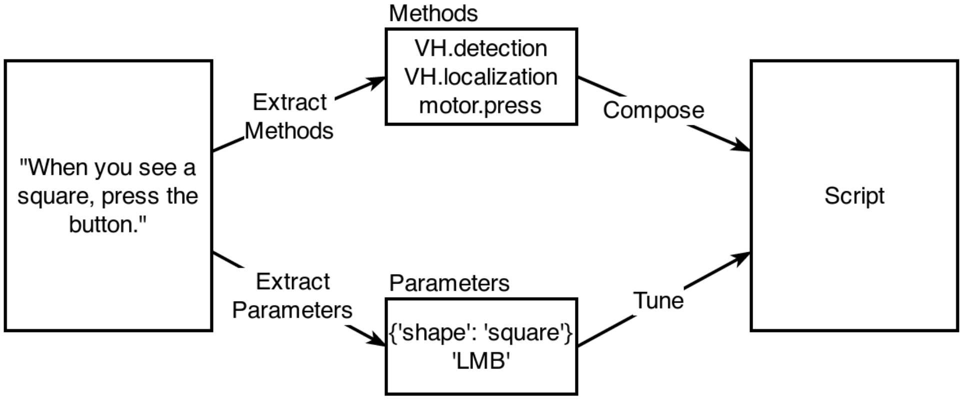

Figure 3.2: Overview of how CPDs are converted into CP scripts. First, methods and parameters are extracted, and then the methods are composed and tuned with parameters to create executable scripts.

the extraction of appropriate methods and parameters out of the CPD, and their assembly into a script. The resulting Python code is shown in Listing 3.2.

As a rule of thumb, Verb Fragment implementations decide what kinds of methods are required for the task being described and Targets and Relationships decide what kind of parameters should be generated to tune those methods into scripts. The CPD's Imperative English component uses a STAR-specific Lexicon, which is aware of the methods contained in the CPM.

Abid's Master's thesis [95] – developed concurrently with this work – deals with the definition and implementation of neural primitives essential for defining the CP methods available in the CPM. A list of the methods stored in the CPM and used in this thesis is available in Section 4.2. Most of them are defined by Abid's work, with a minority of additional methods for interacting with the environment specified by this thesis, for example, motor commands to press buttons.

After tuning CP methods with parameters, CPC generates Python code, like the code shown in Listing 3.2. Note that in the Listing, a single Verb “see” actually evaluates to the composition of two methods (detection and localization), as well as some control flow code to compose the two methods. Section 3.4.4 discusses how this generated code interacts with TarzaNN 3.

Listing 3.2: Script created by parsing the CPD sentence “When you see a square, press the button.” `VH` stands for visual hierarchy, and `vae` for visual Attention Executive.

```
# When you see a square, press the button.
detection1 = self.vae.VH.detection({'shape': 'square'})
if detection1:
  self.vae.motor.press_button('LMB')
else:
  localization1 = self.vae.VH.localization({'shape': '
  ↪ square'})
  if localization1:
    self.vae.motor.press_button('LMB')
```

This covers the translation of the endogenous parts of a CPD. Translation of exogenous parts is less interesting: Non-IE parts of CPDs (for example setup blocks, code blocks, and code annotations), are simply injected into the output code based on their position in the CPD.

3.4 Implementation

The Cognitive Program Compiler (CPC) is the reference implementation of a CPD compiler. It is based on *Imp* (described in Section 2.6), with which it shares the basic compiler architecture and extension mechanism.

However, due to the many features CPD has that IE does not, additional compiler features are implemented, as well as new Sentence objects and a new Lexicon. An overview of the changes follows.

3.4.1 Compiler Stages

CPC's architecture can be seen in Figure 3.3, and we can see that it is similar overall to the *Imp* architecture in Figure 2.1, the most significant difference between the two being CPC's addition of multiple output streams.

Lexing and Parsing

Lexing and parsing is done via code generated by an ANTLR lexer-parser, whose grammar is listed in Appendix A. The grammar is very similar to the IE grammar discussed in Chapter 2 that drives *Imp*'s lexing and parsing, but with extensions to accommodate CPD features like code blocks, annotations and Please statements. The Lexicon is consulted for dictionary entries in exactly the same way as well.

Semantic Analysis

Semantic Analysis traverses and verifies the parse tree generated by the lexer-parser, enforcing a variety of rules to ensure that it indeed represents a valid CPD. This includes consulting the Lexicon for correct attachment of Fragments, enforcing the existence of exactly zero or one setup blocks, enforcing that all code annotations given to Fragments are relevant, and a variety of other rules.

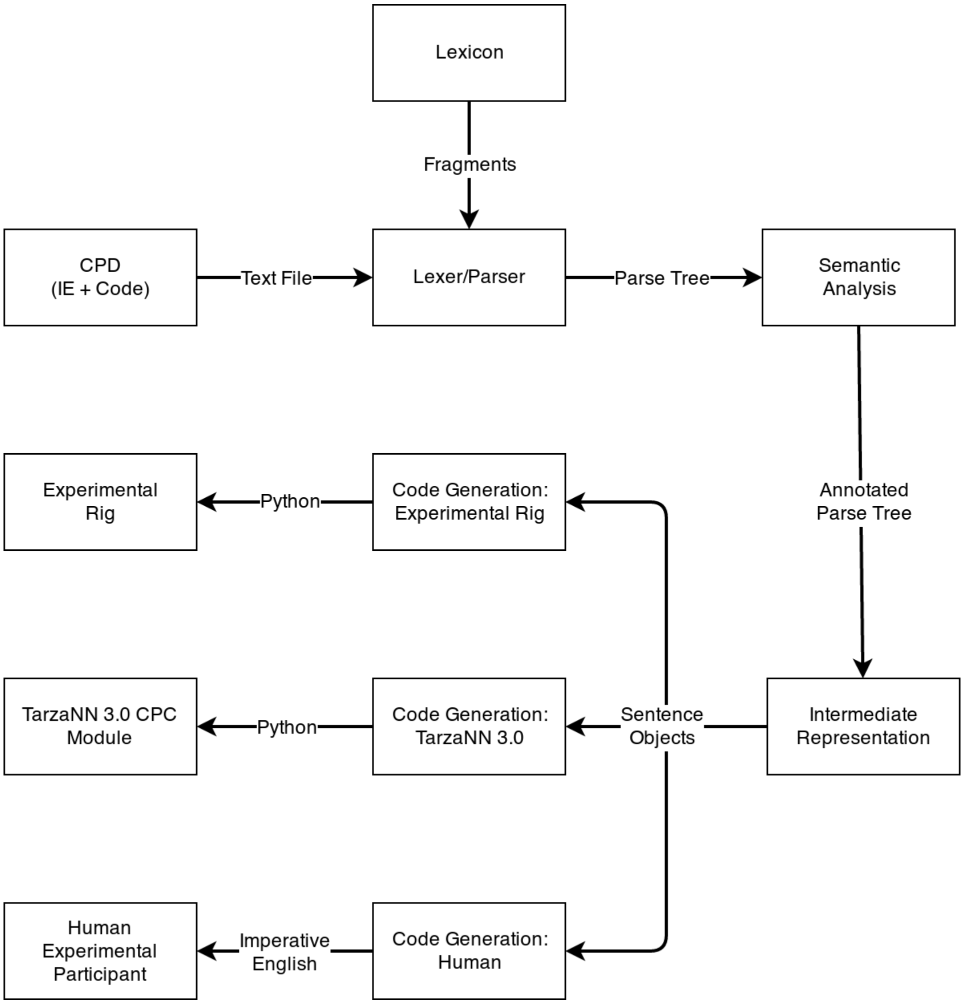

Figure 3.3: Overview of CPC compiler stages.

For each class of errors, an error checker Listener is implemented, and each Listener traverses the parse tree to verify its rules are being enforced by the CPD, or to emit error messages if they are not.

Intermediate Representation

Given a semantically correct parse tree, we can construct an intermediate representation of the CPD. *Imp*'s Sentence objects are reused without modification in CPC, with two new ones added to represent Setup Blocks and Code Blocks. Code Annotations are treated as a special Fragment object. The implementation of Code Block Sentences is fairly straightforward, as they are simple containers for the text string of the code they hold. The intermediate representation tree can then be traversed by the code generation stage to produce output code.

Code Generation

The code generation stage traverses the topmost Sentences in the intermediate representation, queries them to generate code, and uses the results to populate a Template. The Template is scaffolding for a Python class that defines the layout of the output file that CPC will generate. Templates are convenient because they reduce the amount of boilerplate code that needs to be generated, as well as allowing for some flexibility, for example generating Python class files instead of simple Python scripts.

Figure 3.3 shows that CPC has three different Code Generation stages that allow for a wide variety of outputs by combining different parts of the intermediate representation. Section 3.4.3 discusses each of the three output modules.

3.4.2 Lexicon

The CPC provides a STAR-specific Lexicon that contains Fragments useful for describing CPs in the context of STAR and TarzaNN. Targets and Modifiers that describe various kinds of visual stimuli (e.g. Square, Colour, Cross), and objects in the experimental rig (e.g. Display, Button) are provided. Imperative Verbs define actions the experimental subject can undertake, including visual tasks (e.g. LookAt, Find) and non-visual ones (e.g. Push, Say).

The Verb implementations provide a link between the CPD's text and the methods available in the CPM. The Fragments that those Verbs accept are interpreted as parameters to the methods the Verbs will generate.

The set of available CPM methods in TarzaNN 3 are defined in `tn3cpc`, a Python interface to TarzaNN 3 developed as part of this thesis. This interface is discussed in more detail in Section 3.4.4. The interface calls the TarzaNN implementation of CPM methods developed in [95].

Details of the CPC Lexicon hierarchy can be seen in Appendix A.

3.4.3 Output

The modular design of CPC allows us to generate a variety of outputs simply by swapping out the code generation stage. The following three kinds of output have been implemented so far:

- TarzaNN 3 Program

- Human Readable Text
- Experimental Rig Program

Each of these outputs works by generating code based on the intermediate representation, and injecting that code into a different Template. A brief overview of each follows.

TarzaNN 3 Program

The first kind of output CPC can generate is a Python module that controls a TarzaNN 3 simulation of STAR. TarzaNN 3 is a modern redevelopment of the original TarzaNN neural network simulator [17] that will implement a simulation of all the parts of STAR, allowing researchers to conduct simulations of psychophysical experiments in TarzaNN 3. An example of CPC output that has been generated from the Egly experiment CPD in Listing 3.1 can be seen in Listing 3.3.

Listing 3.3: TarzaNN 3 code generated from Egly & Driver 1994 experiment in Listing 3.1 to drive TarzaNN 3.

```
from cpctn3.runners import TN3Runner

class egly1994simple(TN3Runner):
  def __init__(self):
    super().__init__()

    self.button = self.rig.buttons['LMB']

  def run_trial(self):
    self.rig.display_image("img/intertrial.png")
    self.rig.wait(500)

    # A fixation cross ("img/fixation.png") along with two
    ↪ rectangles will appear on the screen.
```

```
    self.vae.VH.prime({'shape': 'cross', 'kind': 'fixation'
    ↪ , 'union': {'shape': 'rectangle', 'count': 'two'}})
    handle1 = self.rig.display_image("img/fixation.png")

    # Look at the fixation cross.
    localization1 = self.vae.VH.localization({'shape': '
    ↪ cross', 'kind': 'fixation'})
    self.vae.FC.setFHMBias(localization1)

    # A cue ("img/cue.png") will appear (1000), then it
    ↪ will disappear (100).
    self.rig.wait(1000)
    handle2 = self.rig.display_image("img/cue.png")
    self.rig.wait(100)
    self.rig.remove_image(handle2)

    handle3 = self.vae.VH.getAS()
    self.vae.VH.prime(handle3)

    # A filled square ("img/target.png") will appear (200)
    ↪ on the screen.
    self.vae.VH.prime({'shape': 'square', 'appearance': '
    ↪ filled'})
    self.rig.wait(200)
    handle4 = self.rig.display_image("img/target.png")

    self.button.listen(2000)

    # When you see the square, press the left mouse button.
    def handle5():
      detection1 = self.vae.VH.detection({'shape': 'square'
      ↪ })
      if detection1.result():
        self.vae.motor.press_button('LMB')
      else:
        localization2 = self.vae.VH.localization({'shape':
        ↪ 'square'})
        if localization2.result():
          self.vae.motor.press_button('LMB')
    self.vae.executor.submit(handle5)

    rt = self.button.wait().result().time
    self.save_result(rt)

if __name__ == "__main__":
```

```
    er = egly1994simple()
    er.run()
```

This code is created by injecting code generated from the intermediate representation into a Template object that represents a Python class inheriting from `TN3Runner`. `TN3Runner` provides an interface to different parts of TarzaNN, in particular: STAR (via `self.vae`), and the experimental rig (via `self.rig`). The `TN3Runner` parent class handles most of the initialization and default setup, such as how many blocks of trials and trials per block the experimental run should consist of.

Note that `self.rig` and `self.vae` have to be able to operate in parallel and execute commands at the same time. An example for why this is neccessary is the call to `self.rig.wait`, which should only pause the execution for the experimental rig portion of TarzaNN, but not for STAR. Operating in a multi-threaded way allows the rig to wait while STAR continues executing. The mechanism used to achieve this is an asynchronous interface built into Python 3 called Futures[1]. The return of every method in `self.rig` and `self.vae` is a Future object, and `self.rig` and `self.vae` each contain its own set of execution queues that resolve method calls one by one.

The Future object is a placeholder for the result of a computation. This allows a method to return a Future object before it has computed the result that the Future object represents, and then proceed to compute the result in the background. When the result has been computed, it will be provided to the Future object, from where it can be read via its `result()` method. If the `result()` method is called before the result is ready, execution is paused

---

[1] https://docs.python.org/3.5/library/concurrent.futures.html

until the result is computed.

The great advantage of this system is that the CPD executes virtually instantly, and builds a network of interconnected Futures that the rig and vAE have to resolve one at a time in parallel by working through their queues of method calls. When correctly implemented, there is no way for one of them to stall the other (also known as deadlock) due to the CPD waiting for a result, because the only thing the CPD does is populate their task queues. This is not to say that the two task queues cannot depend on one another: for example, the rig might be waiting for the vAE to initiate a button press, or the vAE might be waiting for the rig to display an image. In those cases, the waiting is intentional and desired.

However, this means that the generated CPD code should avoid reading the results of Futures, lest it stall the execution. In cases where this is necessary (for example a Fragment generating code that makes a decision based on the result of a Future), a Fragment needs to read Futures inside a function that is then provided as a whole to a task queue for execution.

Human Readable Text

The second, and simplest of the three output modules, is the Human Readable Text module. It simply strips out all the code annotations from the input text to leave behind Imperative English sentences and Please statements, as shown in Listing 3.4.

Listing 3.4: Human Readable Text generated from Egly & Driver 1994 experiment in Listing 3.1 to drive TarzaNN 3.

```
A fixation cross along with two rectangles will appear on
↪ the screen. Look at the fixation cross. A cue will
↪ appear, then it will disappear. Please do not look away
↪ when it does.

A filled square will appear on the screen.

When you see the square, press the left mouse button.
↪ Please respond as quickly as possible, as the response
↪ latency will be recorded, but it is important to
↪ minimize the number of errors.
```

This output can be provided to human experimental participants as experiment instructions. Note the usefulness of Please statements in providing additional information to the human that might be useless to a system like TarzaNN, for example “Please respond as quickly as possible,” which is something TarzaNN will do by design, but a human might not without being explicitly told to prioritize short response times. Of course, if the contents of the Please statement are something that TarzaNN should take into account (for example, if TarzaNN indeed had an option to prioritize short response times over quality of response), one should express it as part of an ordinary IE statement, potentially as a Specifier such as “prioritizing speed” that would inform a Verb’s execution.

Experimental Rig Program

The last CPC output module is based on the following observation: Since CPC can generate TarzaNN 3 code which drives both STAR and the experimental rig, could we take just the experimental rig code, and generate a program to drive a real-world experimental rig, one for running experiments on humans?

The answer to that question is yes, enabling CPC to offer functionality similar to experimental builders such as Psychtoolbox [96] and OpenSesame [97]. Listing 3.5 shows a program generated from the CPD in Listing 3.1. Note, however, that the CPD used here has been modified to use an external image generator instead of relying on filpaths to images (can see modifications in comments in Listing 3.5), as well as to record the participant's id and save results to a named file.

Listing 3.5: Experimental Rig program generated from a modified CPD of the Egly & Driver 1994 experiment in Listing 3.1 to drive a physical experimental rig.

```
from cpctn3.runners import ExperimentRunner

class egly1994(ExperimentRunner):
  def __init__(self):
    super().__init__()

    from eglyig import EglyImageGenerator
    self.ig = EglyImageGenerator()
    self.button = self.rig.buttons['LMB']
    initials = raw_input("Participant Initials?")
    self.results_filename = "results_{}.csv".format(
    ↪ initials)

  def run_trial(self):
    # Reset the button and display inter-trial image.
    self.button.clear()
    self.rig.display_image("img/intertrial.png")
    self.rig.wait(500)

    trial = self.ig.next_trial()
    condition_number = trial["condition_number"]
    fixation_image = trial["fixation"]
    cue_image = trial["cue"]
    target_image = trial["target"]

    # A fixation cross, and two rectangles (fixation_image)
    ↪  will appear on the screen.
```

```
    handle1 = self.rig.display_image(fixation_image)

    # A cue (cue_image) will appear (1000), then it will
    ↪ disappear (100).
    self.rig.wait(1000)
    handle2 = self.rig.display_image(cue_image)
    self.rig.wait(100)
    self.rig.remove_image(handle2)

    # A filled square (target_image) will appear (200) on
    ↪ the screen.
    self.rig.wait(200)
    handle3 = self.rig.display_image(target_image)

    self.button.listen(2000)

    rt = self.button.wait().time
    if rt < 150:
      self.rig.beep(500)
    self.save_result("{}, {}\n".format(condition_number, rt
    ↪ ))

if __name__ == "__main__":
  er = egly1994()
  er.run()
```

Note that the Experimental Rig Program inherits from a different base class, `ExperimentRunner`, where `self.rig` refers to an implementation of the experimental rig in the PyGame library [98], which controls physical displays, buttons and speakers. When run, this Python code will display stimuli full-screen, and record user responses into a file.

3.4.4 TarzaNN 3 CPC Interface

TarzaNN 3 needs to know how to interact with the Python classes generated by CPC. For this, an interface needs to be implemented, such as a Python package that `TN3Runner` can call to control TarzaNN. For this purpose, the

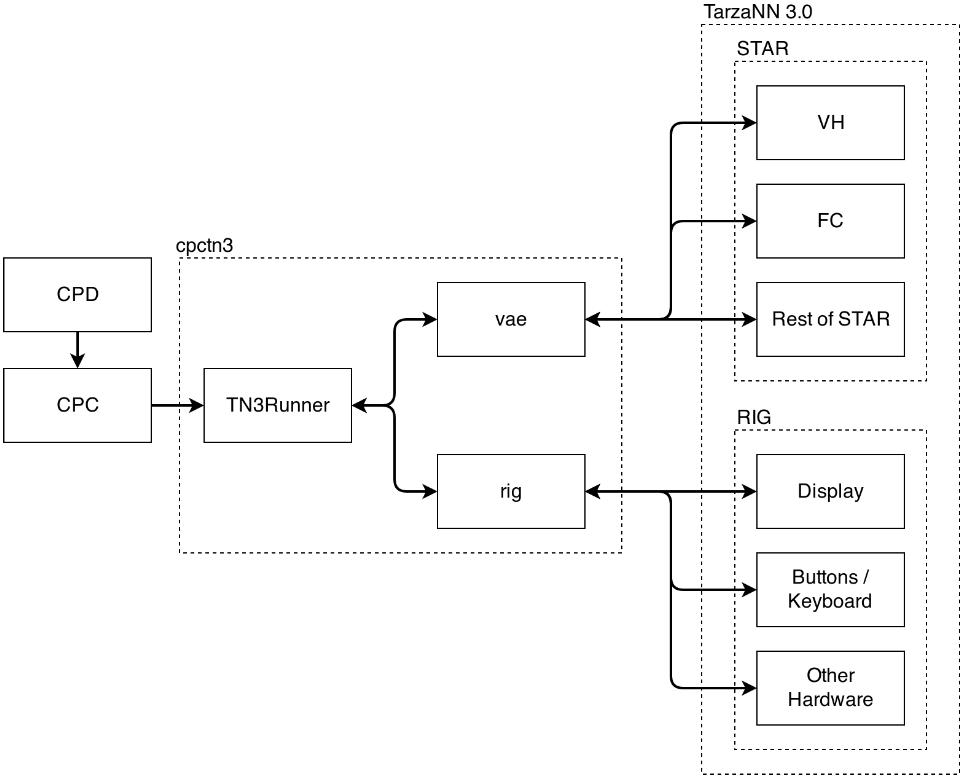

Figure 3.4: Overview of how the `cpctn3` package interacts with TarzaNN 3.

Python package `cpctn3` is implemented. It contains three modules with simple interfaces, `vae`, `runner`, and `rig`. Their relationships are shown in Figure 3.4.

The `vae` module represents the vAE in the context of mapping TarzaNN 3 components to STAR. It controls the attentional cycle, translates task parameters into control signals, determines attentional samples etc. It allows access to various TarzaNN components, like the Visual Hierarchy or Fixation Controller, via interfaces `vae.VH` and `vae.FC`. However, since the development of TarzaNN 3 has not progressed yet to the point where a simulation of STAR can be run, most CPC output that invokes `vae` will not execute as anticipated. To get around that temporary hurdle, and still be able to provide an evaluation of CPC in Chapter 4, calls to `vae` go to a TarzaNN 3 mockup that outputs log files of calls to show that the correct sequence of commands is executed by CPC output.

The `runner` module contains the `TN3Runner` class, which is the parent class from which all CPC output generated by the TarzaNN 3 output module inherits. It takes care of initializing a TarzaNN instance and setting defaults for a TarzaNN experiment.

The `rig` is an interface to the parts of TarzaNN that represent the experimental rig. It controls which visual stimuli are displayed to the visual system simulation, as well as buttons and other means STAR has of interacting with the system.

3.5 Summary

This chapter presented the specification of Cognitive Program Descriptions (CPD), a novel format for Cognitive Program task specifications in the context of the STAR model. CPDs and their reference implementation compiler, CPC, provide a way to write CP task specifications in natural language. This allows CPDs to be human readable, yet flexible enough to allow for arbitrary extension facilitating future research in the STAR model.

An interface for performing experiments with TarzaNN called `tn3cpc` is also provided, which allows CPC output to control the execution of TarzaNN 3 simulations.

Chapter 4

Empirical Evaluation

4.1 Overview

This chapter provides some evidence for the soundness and correctness of CPC output. Since CPC implementation has progressed more quickly than the implementation of TarzaNN 3, as of April 2017 TarzaNN 3 does not have all the components necessary to execute CPC output programs. So to persuade the reader that the correct sequence of commands is generated by CPC, output code is provided, as well as a mockup TarzaNN 3 interface which can approximate the duration of TarzaNN events generated by the execution of the CPC output and produce a log file with the sequence of operations it performed.

Section 4.2 shows examples of CPDs exercising STAR methods contained in the CPM, as well as the code generated by CPC from those examples, to demonstrate that the methods selected by the compiler are reasonable choices.

Section 4.3 shows CPDs for several published visual attention experiments, and provides some commentary and evaluation of the output code.

Section 4.4 compares CPDs to real-world experiment instructions. In particular, an instruction text for a psychology experiment written by a researcher unfamiliar with CPDs is converted into a real CPD.

4.2 CPM Methods

The CPM holds all methods that are available to the vTE to compose Cognitive Programs out of. In the case of CPC and TarzaNN 3, the set of methods available in the CPC is defined by Abid's Master's thesis [95], and supplemented with some additional methods for interacting with the world. Abid's thesis uses definitions of recognition tasks in experimental paradigms provided by Macmillan and Creelman in [99]. This section demonstrates that all those methods are easy to invoke via standard CPD constructs. Table 4.1 lists high-level methods in the CPM used by the CPC. Note that, while these methods are composed of lower-level methods and CPC uses the higher-level ones listed, nothing is barring the Lexicon from using lower-level methods, for example `VH.liftSuppressiveSurround` or `vTE.getcFOA`. These methods could be used by the Verb implementations in the Lexicon to any level of granularity.

An additional thing to note is that this list of CPM contents is in no way complete, and that many obvious methods are missing. Most notably, methods for interacting with memory are not implemented.

The following sections discuss the methods in Table 4.1, in addition to

Table 4.1: Listing of methods contained in the CPM that are used by CPC's Lexicon.

Recognition task methods in [95]	Fixation Controller methods	Motor methods
• VH.Prime • VH.Discrimination • VH.Detection • VH.Recognition • VH.Identification • VH.Localization	• FC.SetFHMBias	• motor.Click • motor.Press • motor.Say

providing CPD and output code listings.

4.2.1 Recognition Methods

VH.Prime

Priming prepares the Visual Hierarchy (VH) for input: it biases the VH to be more selective for task-relevant features, objects, and locations by suppressing task irrelevant ones. Therefore, priming can be *spatial*, where the VH is primed for a location, or *featural*, where the VH is primed for features or an object.

An example of featural priming:

```
A red square will appear.
```

and the generated code:

```
self.vae.VH.prime({'shape': 'square', 'color': 'red'})
```

An example of spatial priming:

```
A stimulus will appear on the left side of the screen.
```

the generated code:

```
self.vae.VH.prime({'location': 'left'})
```

Note that there exists an alternative to endogenous spatial priming shown in the above example. In a real experiment, one is more likely to use cueing as a mechanism for specifying spatial priming. This is done simply by informing the experimental subject that a cue will appear on the screen, and then displaying the cue stimulus. The experimental subject's Visual Hierarchy will respond to this by processing the stimulus, attending to it, deriving an attentional sample, and priming itself with that attentional sample, while disengaging attention and applying object-based inhibition of return to previous stimuli. An example of this is:

```
A cue ("img/cue.png") will appear (1000), then it will
↪ disappear (100).
```

Since the above statement is an IE Fact, not a Command, it features annotations for the benefit of the experimental rig, and both endogenous (`self.vae`) and exogenous (`self.rig`) instructions are generated:

```
self.rig.wait(1000)
handle1 = self.rig.display_image("img/cue.png")
self.rig.wait(100)
self.rig.remove_image(handle1)

handle2 = self.vae.VH.getAS()
self.vae.VH.prime(handle2)
```

VH.Discrimination

In the Discrimination task, the experimental subject is expected to differentiate between two kinds of stimuli. They may be asked to identify a stimulus as belonging to one of two classes, one of which may or may not be noise. In the case where one class is noise, the task is called **Detection**, and in the case where neither class is noise, the task is called **Recognition**.

VH.Detection

Detection is a special case of Discrimination where one of the classes is noise (null). This is equivalent to a situation where one class is noise, and the other is signal and noise. An experimental subject may be asked to say whether or not an instance of an item is present in the display. For example:

```
If you see a red square, press the button.
```

```
detection1 = self.vae.VH.detection({'shape': 'square', '
↪ color': 'red'})
if detection1:
  self.vae.motor.press_button('LMB')
```

VH.Recognition

Recognition is a kind of Discrimination where neither of the two classes is noise. For example:

```
Wait until you see a square.
If the square is red, press "z", else if the square is blue
↪ , press "m".
```

Keyboard keys “z” and “m” are a common choice for providing responses in psychology experiments because they are located at extreme ends of the bottom row of a QWERTY keyboard, allowing the experimental subject to place an index finger on each key.

```
# Wait until you see a square.
detection1 = None
while not detection1:
  detection1 = self.vae.VH.detection({'shape': 'square'})

# If the square is red, press "z", else if the square is
↪ blue, press "m".
if detection1.matchToFeature({'color': 'red'}):
  self.rig.keyboard.press("z")
else:
  if detection1.matchToFeature({'color': 'blue'}):
    self.rig.keyboard.press("m")
```

The definite article allows us to identify that the square we should compare in the condition “the square is red” had already been referenced, and that we should use that reference in the code that follows. The object resulting from the detection task called `detection1` is an Attentional Sample, which we can use to ask questions about the stimulus that had been detected.

VH.Identification

Identification is similar to the Discrimination task, but it allows for multiple classes: one stimulus, from a set of more than two, is presented on each trial and needs to be assigned a class label.

```
Wait until you see a circle.
If the circle is blue, press "b", else if the circle is red
↪ , press "r", else if the circle is green, press "g".
```

```
# Wait until you see a circle.
detection1 = None
while not detection1:
  detection1 = self.vae.VH.detection({'shape': 'circle'})

# If the circle is blue, press "b", else if the circle is
↪ red, press "r", else if the circle is green, press "g".
```

```
if detection1.matchToFeature({'color': 'blue'}):
  self.rig.keyboard.press("b")
else:
  if detection1.matchToFeature({'color': 'red'}):
    self.rig.keyboard.press("r")
  else:
    if detection1.matchToFeature({'color': 'green'})
      self.rig.keyboard.press("g")
```

The awkward "else: if" is used in the output code instead of the more pythonic "elif" because IE's grammar structures a chain of if-then-else statements as a chain of nested if-then-else statements, each of which generates its own code. This could be avoided by collapsing the nested if-then-else chain into a single Sentence during the intermediate representation construction stage of the compiler.

VH.Localization

In Localization, the task is to return the location of a stimulus. In STAR terms, an Attentional Sample (AS) [19] is returned by the Localization method. The AS can then be used as a parameter to the FC, motor system, spatial priming etc.

```
Click on the green square.
```

```
# Click on the green square.
localization1 = self.vae.VH.localization({'shape': 'square'
↪ , 'color': 'green'})
self.rig.mouse.click(localization1)
```

The localization task may also be invoked if a detection takes a long time, as the experimental subject will start to search for stimuli they cannot detect. In that case, a sentence like:

```
When you see a green square, press the button.
```

will get translated as:

```
detection1 = self.vae.VH.detection({'shape': 'square', '
↪ color': 'green'})
if detection1.result():
  self.vae.motor.press_button('LMB')
else:
  localization1 = self.vae.VH.localization({'shape': '
  ↪ square', 'color': 'green'})
  if localization1.result():
    self.vae.motor.press_button('LMB')
```

4.2.2 Fixation Controller Methods

FC.SetFHMBias

The Fixation History Map (FHM) is a part of the Fixation Controller that represents the 2D visual space. It is larger than the visual field and combines locations of recent fixations with task specific biases. This method allows us to set a bias in the FHM to explicitly influence the Fixation Controller's next eye movement. This method takes a location as a parameter.

```
Look at the fixation cross.
```

```
location1 = self.vae.VH.localization({'shape': 'cross', '
↪ kind': 'fixation'})
self.vae.FC.setFHMBias(location1)
```

4.2.3 Motor Methods

The experimental subject needs to have some means of communicating responses based on the results of the computations they perform. Therefore, at least some rudimentary methods to control the motor system need to be implemented. Examples for these methods have been given in Section 4.2.1, so they will not be repeated here.

motor.Click

This method simulates a mouse click, and returns the location of the click as a response. Note that this method is not implemented to any biologically faithful level of detail. It takes in a location as a parameter, and when run as a script, an event is emitted with information about a click on that location. Anything else, like matching visual stimuli to motor actions, would be far out of scope of this thesis.

motor.Press

This method simulates the press of a previously defined button or keyboard key, allowing the experimental rig to log the timing of the button press.

motor.Say

Simulates the experimental subject providing verbal feedback. Verbalization is even more complex than clicking and pushing buttons, so the implementation of this method simply outputs a string.

```
Find the square.
Say the colour of the square.
```

```
# Find the square.
localization1 = self.vae.VH.localization({'shape': 'square'
↪ })

# Say the colour of the square.
```

```
self.vae.motor.say(localization1.getColour())
```

4.3 Visual Attention Experiments

After exploring what kinds of methods are available to us, let us try to recreate some visual attention experiments in CPD form and examine the generated code.

4.3.1 Egly & Driver 1994

The Egly & Driver 1994 experiment [9], used throughout this thesis as a running example of a CPD, was originally performed to examine how spatial location and the perception of an object impact visual attention, in particular how response times change relative to valid and invalid attentional cueing. The CPD for the experiment is in Listing 3.1, the generated code for TarzaNN is in Listing 3.3, and the log file from the mockup simulator executing the TarzaNN code is in Listing 4.1. Figure 4.1 contains an illustration of the experimental sequence for the experiment.

Listing 4.1: Log file generated by mockup simulator using code generated from the CPD in Listing 3.1 of the Egly & Driver 1994 experiment [9].

```
[00000 ms] RIG:    Displaying image 'img/intertrial.png'.
[00000 ms] VH:     Priming for {'shape': 'cross', 'kind': '
↪ fixation', 'union': {'count': 'two', 'shape': 'rectangle
↪ '}}.
[00000 ms] RIG:    Waiting for 500 ms.
[00134 ms] VH:     Localization for {'shape': 'cross', '
↪ kind': 'fixation'}.
[00388 ms] VH:     Localization found {'shape': 'cross', '
↪ kind': 'fixation'}.
```

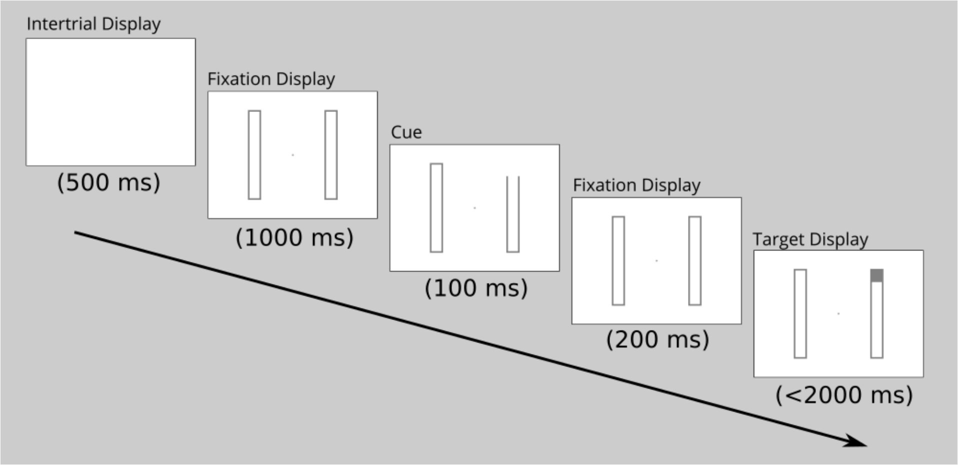

Figure 4.1: Experimental sequence for Egly & Driver 1994 experiment.

```
[00389 ms] FC:     Setting FHM bias for location (127, 127)
↪ .
[00501 ms] RIG:    Displaying image 'img/fixation.png'.
[00502 ms] RIG:    Waiting for 1000 ms.
[01503 ms] RIG:    Displaying image 'img/cue.png'.
[01503 ms] RIG:    Waiting for 100 ms.
[01540 ms] VH:     Priming for {'location': (100, 150)}.
[01603 ms] RIG:    Removing image 'img/cue.png'.
[01603 ms] RIG:    Waiting for 200 ms.
[01673 ms] VH:     Priming for {'shape': 'square', '
↪ appearance': 'filled'}.
[01803 ms] RIG:    Displaying image 'img/target.png'.
[01803 ms] BUTTON: Button 'LMB' listening for 2000 ms.
[01807 ms] VH:     Detection for {'shape': 'square'}.
[01974 ms] VH:     Detection found {'shape': 'square'}.
[01974 ms] MOTOR:  Pushing button 'LMB'.
[02191 ms] BUTTON: Button press detected on 'LMB'.
[02193 ms] RUNNER: Saving result (388,) in 'results.csv'.
```

From the Listing, we can see that the sequence of RIG and STAR events issued by the TarzaNN mockup executing the CPD roughly follows the timings and order one would expect based on the results reported in [9]. Unfortunately, exact results are impossible without a simulator like TarzaNN:

the mockup has no way of knowing what kinds of stimuli are being displayed or how to process them. Most actions are implemented with a delay to simulate the duration of different instructions, but some simply cannot be recreated in a mockup, for example any action that requires STAR to answer questions about the stimuli currently displayed by the experimental rig.

Note, however, that while the mockup simulates the issuance of various commands, their duration, and their returns, the durations of the various operations were tuned to correspond to experimental results. This is due to the aforementioned implementation timeline difference between TarzaNN and CPC, preventing any actual simulation of STAR from being done at the time this thesis work was undertaken.

4.3.2 Cutzu & Tsotsos 2003

The Cutzu & Tsotsos 2003 experiment [100] is a kind of same-different N-interval design experimental task. It involves a circular display of letters, two of which are red, the remainder being black, and the experimental subject is asked to answer if the two red letters are the same or different. The aim of the experiment is to prove the existence of a suppressive annulus around an attended item, which is one of ST's predictions [14]. The CPD for this experiment is in Listing 4.2, and the code generated is in Listing 4.3.

Listing 4.2: CPD describing the Cutzu & Tsotsos 2003 experiment.

```
A fixation cross ("img/fixation_cross.png") will appear.
↪ Look at the fixation cross. Please do not look away from
↪  the cross throughout the experiment.
```

```
A cue ("img/experiment_cue.png") will flash (1000, 180),
↪ then a test image ("img/experiment_test.png") with red
↪ characters will flash (0, 100).

[%
  self.rig.keyboard.listen()
%]

When you see red characters, if the red characters are both
↪  Ls or they are both Ts, press "z", else press "m".

A mask ("img/mask.png") will appear.

[%
  key_event = self.rig.keyboard.wait().result()
  self.save_result(key_event.time, key_event.character)
%]
```

Note how the Verb “flash” is used in this CPD instead of the “appear and disappear” idiom used in the Egly experiment. The two statements produce equivalent code in effect. The first part of Flash’s annotation is the delay in miliseconds before stimulus appearance, and the second part is the number of miliseconds before disappearance.

Listing 4.3: TarzaNN 3 code generated from the CPD for the Cutzu & Tsotsos 2003 experiment from Listing 4.2.

```
from cpctn3.runners import TN3Runner

class ct2003simple(TN3Runner):
  def __init__(self):
    super().__init__()

  def run_trial(self):
    # A fixation cross ("img/fixation_cross.png") will
    ↪ appear.
    self.vae.VH.prime({'shape': 'cross', 'kind': 'fixation'
    ↪ })
    handle1 = self.rig.display_image("img/fixation_cross.
    ↪ png")
```

```
# Look at the fixation cross.
localization1 = self.vae.VH.localization({'shape': '
↪ cross', 'kind': 'fixation'})
self.vae.FC.setFHMBias(localization1)

# A cue ("img/experiment_cue.png") will flash (1000,
↪ 180), then a test image ("img/experiment_test.png")
↪ with red characters will flash (0, 100).
self.rig.wait(1000)
handle2 = self.rig.display_image("img/experiment_cue.
↪ png")
self.rig.wait(180)
self.rig.remove_image(handle2)

handle3 = self.vae.VH.getAS()
self.vae.VH.prime(handle3)

self.vae.VH.prime({'shape': 'character', 'color': 'red'
↪ })
self.rig.wait(0)
handle4 = self.rig.display_image("img/experiment_test.
↪ png")
self.rig.wait(100)
self.rig.remove_image(handle4)

self.rig.keyboard.listen()

# When you see red characters, if the red characters
↪ are both Ls or they are both Ts, press "z", else
↪ press "m".
def handle5():
  detection1 = self.vae.VH.detection({'shape': '
  ↪ character', 'color': 'red'})
  if detection1.result():
    if len(detection1.matchToFeature({'shape': 'L'}))
    ↪ == 2 or len(detection1.matchToFeature({'shape':
    ↪ 'T'})) == 2:
      self.rig.keyboard.press("z")
    else:
      self.rig.keyboard.press("m")
  else:
    localization2 = self.vae.VH.localization({'shape':
    ↪ 'character', 'color': 'red'})
    if localization2.result():
```

```
          if len(localization2.matchToFeature({'shape': 'L'
          ↪ })) == 2 or len(localization2.matchToFeature({
          ↪ 'shape': 'T'})) == 2:
            self.rig.keyboard.press("z")
          else:
            self.rig.keyboard.press("m")
    self.vae.executor.submit(handle5)

    # A mask ("img/mask.png") will appear.
    handle6 = self.rig.display_image("img/mask.png")

    key_event = self.rig.keyboard.wait().result()
    self.save_result(key_event.time, key_event.character)

if __name__ == "__main__":
  er = ct2003simple()
  er.run()
```

4.3.3 Raymond et al. 1992

Listing 4.4 contains the CPD of Experiment 2 from the Raymond et al. 1992 paper [10]. The experiment is a rapid serial visual presentation (RSVP) task where a sequence of 15 letters is presented on the screen for 15 ms per letter, with an inter-stimulus period of 75 ms. After the sequence, the experimental subject is asked to state if an X appeared or not, and to name the white letter in the sequence. Listing 4.5 contains the generated TarzaNN code.

Listing 4.4: CPD of Experiment 2 from Raymond et al. 1992 [10].

```
{%
  from experiments import RaymondImageGenerator
  self.ig = RaymondImageGenerator()
%}

Hold the mouse button.
A centre fixation dot ("fixation_dot.png") will appear on
↪ the screen. Look at the fixation dot, then when you are
↪ ready, release the mouse button.
```

```
A sequence (ig.get_raymond_image_set, 15) of characters
↪ will be presented (180), then a mask ("mask.png") will
↪ flash (0, 60).

[%
  self.rig.outstream.clear()
%]

Say the white letter.
If you saw an X after the white letter, say "yes", else say
↪  "no".
[%
  output_line1 = self.rig.outstream.readln().result()
  output_line2 = self.rig.outstream.readln().result()
  self.save_result(output_line1, output_line2)
%]
```

In this example, the images for the RSVP task are defined and presented as a sequence, instead of appearing one by one like individual images. To achieve that, an image generator class provides a list of images and their timings to the Specifier “sequence”, and the Verb “be presented” takes that sequence and displays its images on the display one by one, with delays according to timings generated by the image sequence parameter.

Listing 4.5: TarzaNN 3 code generated from the CPD in Listing 4.4 of the Raymond et al. 1992 Experiment 2 from [10].

```
from cpctn3.runners import TN3Runner

class raymond1992(TN3Runner):
  def __init__(self):
    super().__init__()

    from experiments import RaymondImageGenerator
    self.ig = RaymondImageGenerator()

  def run_trial(self):
    # Hold the mouse button.
```

```
self.vae.motor.hold_button('LMB')

# A centre fixation dot ("fixation_dot.png") will
↪ appear on the screen.
self.vae.VH.prime({'shape': 'dot', 'kind': 'fixation'})
handle1 = self.rig.display_image("img/fixation_dot.png"
↪ )

# Look at the fixation dot, then when you are ready,
↪ release the mouse button.
localization1 = self.vae.VH.localization({'shape': 'dot
↪ ', 'kind': 'fixation'})
self.vae.FC.setFHMBias(localization1)

self.vae.motor.release_button('LMB')

# A sequence (ig.get_raymond_image_set(15)) of
↪ characters will be presented (180), then a mask ("
↪ mask.png") will flash (0, 60).
self.vae.VH.prime({'shape': 'character'})
def handle2():
  self.rig.wait(180)
  for ite1 in ig.get_raymond_image_set(15):
    self.rig.wait(ite1.appear_delay)
    handle3 = self.rig.display_image(ite1.image)
    self.rig.wait(ite1.disappear_delay)
    self.rig.remove_image(handle3)
self.rig.executor.submit(handle2)

self.rig.wait(0)
handle4 = self.rig.display_image("mask.png")
self.rig.wait(60)
self.rig.remove_image(handle4)

self.rig.outstream.clear()

# Say the white letter.
handle5 = self.vae.VH.getAS()
handle6 = handle5.matchToFeature({'color': 'white'})[0]
self.vae.motor.say(handle6.getShape())

# If you saw an X after the white letter, say "yes",
↪ else say "no".
def handle7():
  handle8 = self.vae.vWM.get({'shape': 'X', 'after': {'
```

```
        ↪ shape': 'letter', 'color': 'white'}})
        if handle8:
          self.vae.motor.say("yes")
        else:
          self.vae.motor.say("no")
      self.vae.executor.submit(handle7)

      output_line1 = self.rig.outstream.readln().result()
      output_line2 = self.rig.outstream.readln().result()
      self.save_result(output_line1, output_line2)

if __name__ == "__main__":
  er = raymond1992()
  er.run()
```

The “saw” Verb currently generates some non-functional code, as `vWM.get` is not implemented or defined yet in published work, and developing necessary working memory machinery is beyond the scope of this thesis. The relationships in the query object define the relationship between the “X” whose attentional sample we want returned, and its relationship to the “white letter” attentional sample.

4.3.4 Folk et al. 1992

Listing 4.6 contains a CPD of experiment 1 from the 1992 experimental paper by Folk et al. [11]. The experiment aims to prove that abrupt onset cues only capture attention when an abrupt onset is used to locate the target, and not when some other property is used for target location. Listing 4.7 shows the output generated from the CPD.

Listing 4.6: CPD of experiment 1 from Folk et al. 1992 [11].

```
{%
  from random import randint
  from experiments import FolkImageGenerator
```

```
  self.ig = FolkImageGenerator()
%}

[%
  trial = self.ig.next_trial()
  cue_img = trial['cue']
  target_img = trial['target']
%]

A fixation image ("img/fixation_display.png") will appear.
↪ Look at the centre fixation box. Please do not look away
↪  from the box throughout the experiment, because the
↪ trial events will be occurring very rapidly, so
↪ attempting to make eye movements will impair performance
↪ .

[%
  self.rig.wait(500)
  blink_handle = self.rig.display_image("img/
  ↪ fixation_display_blink.png")
  self.rig.wait(100)
  self.rig.remove_image(blink_handle)

  self.rig.wait(1000+randint(0,4)*100)
%]

Please respond as quickly as you can while making as few
↪ errors as possible.

A cue (cue_img) will flash (0, 50), then a red target (
↪ target_img) will flash (100, 50).

[%
  self.rig.keyboard.listen(1500)
%]

If the red target was a "=", press "z", else if the target
↪ was an X, press "m".

[%
  key_event = self.rig.keyboard.wait().result()
  self.save_result(key_event.time, key_event.character)

  self.rig.wait(500)
  blink_handle = self.rig.display_image("img/
```

```
  ↪ fixation_display_blink.png")
  self.rig.wait(100)
  self.rig.remove_image(blink_handle)
%]
```

Listing 4.7: TarzaNN 3 code generated from the CPD in Listing 4.6 of the Folk et al. 1992 Experiment 1 from [11].

```
from cpctn3.runners import TN3Runner

class folk1992(TN3Runner):
  def __init__(self):
    super().__init__()

    from random import randint
    from experiments import FolkImageGenerator
    self.ig = FolkImageGenerator()

  def run_trial(self):
    trial = self.ig.next_trial()
    cue_img = trial['cue']
    target_img = trial['target']

    # A fixation image ("img/fixation_display.png") will
    ↪ appear.
    self.vae.VH.prime({'kind': 'fixation'})
    handle1 = self.rig.display_image("img/fixation_display.
    ↪ png")

    # Look at the centre fixation box.
    localization1 = self.vae.VH.localization({'shape': '
    ↪ square', 'kind': 'fixation', 'location': 'center'})
    self.vae.FC.setFHMBias(localization1)

    self.rig.wait(500)
    blink_handle = self.rig.display_image("img/
    ↪ fixation_display_blink.png")
    self.rig.wait(100)
    self.rig.remove_image(blink_handle)

    self.rig.wait(1000+randint(0,4)*100)

    # A cue (cue_img) will flash (0, 50), then a red target
```

```
    ↪  (target_img) will flash (100, 50).
    self.rig.wait(0)
    handle2 = self.rig.display_image(cue_img)
    self.rig.wait(50)
    self.rig.remove_image(handle2)

    handle3 = self.vae.VH.getAS()
    self.vae.VH.prime(handle3)

    self.vae.VH.prime({'color': 'red'})
    self.rig.wait(100)
    handle4 = self.rig.display_image(target_img)
    self.rig.wait(50)
    self.rig.remove_image(handle4)

    self.rig.keyboard.listen(1500)

    # If the red target was a "=", press "z", else if the
    ↪ target was an X, press "m".
    def handle5():
      handle6 = self.vae.VH.getAS()
      handle7 = handle6.matchToFeature({'color': 'red'})
      if handle7.matchToFeature({'shape': '='}):
        self.rig.keyboard.press("z")
      else:
        if handle7.matchToFeature({'shape': 'X'}):
          self.rig.keyboard.press("m")
    self.vae.executor.submit(handle5)

    key_event = self.rig.keyboard.wait().result()
    self.save_result(key_event.time, key_event.character)

    self.rig.wait(500)
    blink_handle = self.rig.display_image("img/
    ↪ fixation_display_blink.png")
    self.rig.wait(100)
    self.rig.remove_image(blink_handle)

if __name__ == "__main__":
  er = folk1992()
  er.run()
```

4.3.5 Bichot & Schall 2002

Bichot & Schall conducted an experiment in 2002 [12] on priming in macaque monkeys during popout visual search. The monkeys were presented with four stimuli, and their task was to make a saccade to the colour singleton stimulus among them. The CPD describing the experiment is in Listing 4.8, and the generated code is in Listing 4.9

Listing 4.8: CPD of experiment from Bichot & Schall 2002 [12].

```
{%
  from experiments import BS2002ImageGenerator
  from hardware import JuiceDispenser
  self.ig = BS2002ImageGenerator()
  self.jd = JuiceDispenser()
%}

[%
  trial = self.ig.next_trial()
  target_img = trial['target_image']
  target_location = trial['target_location']
%]

A fixation image ("img/fixation_square.png") will appear.
↪ Look at the centre fixation square. Please do not look
↪ away.

[%
  # Wait until the subject has been looking at the centre
  ↪ of the screen for 500 ms.
  centre_success = self.rig.eye_tracker.wait_or_fail((127,
  ↪ 127), 500)
  if not centre_success.result():
    self.rig.wait(40)
    return # no juice
%]

An image (target_img) containing red rectangles together
↪ with green rectangles will appear.

[%
```

```
  start_measurement = self.rig.now().result()
  # Timeout if the subject does not saccade within 2000 ms
  still_looking = self.rig.eye_tracker.wait_or_fail((127,
  ↪ 127), 2000)
%]

Look at the color singleton rectangle.

[%
  if still_looking.result():
    self.rig.wait(40)
    return # no juice

  # Wait until the subject has been looking at the target
  ↪ for 500ms
  target_success = self.rig.eye_tracker.wait_or_fail(
  ↪ target_location, 500)
  if not target_success.result():
    self.rig.wait(40)
    return # no juice

  self.save_result(self.rig.now().result() -
  ↪ start_measurement)
  # Dispense 0.1 milliliter of juice
  self.jd.dispense(0.1)
%]
```

The original experiment calls for monitoring fixation locations using scleral coils as an eye tracker. This facility is implemented as `self.rig.eye_tracker`. This object can be queried to return the current screen-coordinates of the gaze, or to return if the subject looks away from a given set of coordinates. We can also see the demonstration of the trial abort functionality via `return`.

An external facility is the juice-reward dispenser for the monkey. This feature was implemented simply to demonstrate the ease of extensibility of the CPD format to novel experimental features, even though TarzaNN 3 does not care much for juice rewards.

Listing 4.9: TarzaNN 3 code generated from the CPD in Listing 4.8 of the Bichot & Schall 2002 experiment [12].

```
from cpctn3.runners import TN3Runner

class bs2002(TN3Runner):
  def __init__(self):
    super().__init__()

    from experiments import BS2002ImageGenerator
    from hardware import JuiceDispenser
    self.ig = BS2002ImageGenerator()
    self.jd = JuiceDispenser()

  def run_trial(self):
    trial = self.ig.next_trial()
    target_img = trial['target_image']
    target_location = trial['target_location']

    # A fixation image ("img/fixation_square.png") will
    ↪ appear.
    handle1 = self.rig.display_image("img/fixation_display.
    ↪ png")

    # Look at the centre fixation square.
    localization1 = self.vae.VH.localization({'shape': '
    ↪ square', 'kind': 'fixation', 'location': 'center'})
    self.vae.FC.setFHMBias(localization1)

    # Wait until the subject has been looking at the centre
    ↪  of the screen for 500 ms.
    centre_success = self.rig.eye_tracker.wait_or_fail
    ↪ ((127, 127), 500)
    if not centre_success.result():
      self.rig.wait(40)
      return # no juice

    # An image (target_img) containing red rectangles
    ↪ together with green rectangles will appear.
    self.vae.VH.prime({'shape': 'rectangle', 'color': 'red|
    ↪ green'})
    handle2 = self.rig.display_image(target_img)

    start_measurement = self.rig.now().result()
```

```
    # Timeout if the subject does not saccade within 2000
    ↪ ms
    still_looking = self.rig.eye_tracker.wait_or_fail((127,
    ↪  127), 2000)

    # Look at the color singleton rectangle.
    localization2 = self.vae.VH.localization({'shape': '
    ↪ rectangle', 'singleton': 'color'})
    self.vae.FC.setFHMBias(localization2)

    if still_looking.result():
      self.rig.wait(40)
      return # no juice

    # Wait until the subject has been looking at the target
    ↪  for 500ms
    target_success = self.rig.eye_tracker.wait_or_fail(
    ↪ target_location, 500)
    if not target_success.result():
      self.rig.wait(40)
      return # no juice

    self.save_result(self.rig.now().result() -
    ↪ start_measurement)
    # Dispense 0.1 milliliter of juice
    self.jd.dispense(0.1)

if __name__ == "__main__":
  er = bs2002()
  er.run()
```

4.4 Ease of Use

There is no standard experimental instruction format in psychology, however, most experimental instruction texts look alike. They often consist of short imperative mood sentences similar to IE, showing that it would be no great leap to imagine adopting CPD for the purpose of providing a standard experimental instruction format. Studies confirming readability and write-ability of IE and CPD are out of scope of this thesis, however, to attempt to persuade the reader that this might be true, this section offers a brief demonstration of translation between ordinary English to IE.

Listing 4.10 shows instructions for a color visual search experiment, written by Sang-Ah Yoo, a psychology researcher unfamiliar with Imperative English [101]. Listing 4.11 shows those same instructions in CPD form.

Listing 4.10: Experimental instruction text for color visual search experiment, written by a researcher unfamiliar with IE/CPD.

```
1. Look at the center of the display and press the spacebar
↪  when you are ready to start the experiment.
2. Coloured patches will be presented on the screen. There
↪ are target-present and target-absent trials. In target-
↪ present trials, one distinctive colour patch will be
↪ presented among the distractor patches (distractor
↪ patches have the same colour). In target-absent trials,
↪ all patches will have the same colour so no colour
↪ singleton.
3. Press 'z' once you find a target (colour singleton) and
↪ press 'm' if there is no target. Please press button as
↪ rapid as possible.
4. Visual search display will be shown until you respond.
↪ Once you respond, the next trial will be presented
↪ automatically.
```

Listing 4.11: Translation of experimental instruction text in Listing 4.10 into a CPD.

```
Look at the center of the display, then press the spacebar
↪ when ready.
Coloured patches will appear on the screen. Please note
↪ that there are target-present and target-absent trials:
↪ in target-present trials one distinctive colour patch is
↪  present among distractor patches, and in target-absent
↪ trials all patches have the same colour, so the colour
↪ singleton is absent.
If you see a colour singleton, press "z", otherwise, press
↪ "m". Please press the button as rapidly as possible,
```

```
↪ upon which the next trial will be presented.
```

From the above two listings, we can see that many of the instructions are commands that a neural network simulator would be unlikely to understand or need. Those instructions can easily be be translated into Please statements, keeping them available for the human reader, but ignorable for the machine. Other sentences have been modified minimally to fit CPD's syntax and currently implemented Lexicon.

4.5 Summary

This chapter showed an evaluation of code generated by TarzaNN. Section 4.2 showed examples of snippets of CPDs that activate particular methods in the CPM, and Section 4.3 showed examples of fully-featured real-world psychophysical experiments implemented in CPM. Even though we cannot test the code in TarzaNN at the present moment, a mockup TarzaNN interface was constructed that produces log files of execution sequences. Section 4.4 showed an example of translation between a real-world piece of experimental instructions to CPD, to demonstrate the usability of this format.

Chapter 5

Discussion and Conclusions

5.1 Summary

At the beginning of this thesis, its objective was specified as:

> "How can we specify Cognitive Programs in an easy and convenient, yet rigorous way using natural language?"

Cognitive Program Descriptions provide an answer to that question. They use a special Constrained Natural Language called Imperative English to describe tasks for the executive controller of visual attention in the context of the Selective Tuning Attentive Reference model of the human visual system.

CPDs succeed not only in defining a specification format for Cognitive Programs, but also a format for an experimental builder and structured experiment instruction texts. It is equally easy to read for humans and machines, and removes the need to communicate Cognitive Programs via diagrams or other unwieldy formats in favour of a simple and expressive text file.

An implementation of a CPD compiler, called the Cognitive Program Compiler, is also designed and developed by this thesis. CPC eliminates the need for writing CP experiments in TarzaNN by hand and thereby opens the door for rapid experimental iteration.

5.2 Future Work

There is much future work planned for Cognitive Program Descriptions, some of which is overviewed in this section.

Testing with TarzaNN 3 Even though there has not been an opportunity to test CPC output with TarzaNN, this is definitely a plan for the near future when TarzaNN development progresses to the point where such tests are possible. This would allow comparison of experiments generated in Section 4.3 with data from human experiments.

Extending CPC Lexicon The CPC Lexicon as it stands covers a lot of ground, but more functionality could be implemented, most notably support for memory methods.

Declarative Programming Support for a declarative style of programming would be a significant boon to CPDs. It would allow defining new concepts and relationships on the fly, in effect allowing one to extend the Lexicon via IE text. For example, “A stack is a kind of collection. It has a method called pop. Pop means ‘remove top element of the stack’.”

Syntactic Robustness One could potentially increase syntactic robustness of CPC by utilizing an off-the-shelf linguistic parser like CoreNLP to handle “fuzzier” output and unfamiliar syntactic structures. This could be implemented by parsing text via CoreNLP first, and then traversing that parse tree to extract recognizable elements using the Lexicon, which could then be used to build the intermediate representation. This might mean sacrificing some of the power of the error reporting system which is currently operating on a “rigid” grammar, but perhaps a balance could be found.

IE Usability Study Evaluation by different humans to see how natural or unnatural the structure of IE is, and how easy or hard CPDs are to write in practice. One could structure these experiments similarly to how [46] evaluated NLC, with human test subjects being asked to verbalize a set of tasks and then evaluating the frequency of them verbalizing valid IE, and how frequently the interpretations of the verbalizations matched the intended meaning.

CPC IDE An Integrated Development Environment for CPC could be implemented by coupling a part of the compiler pipeline into a plugin for an existing text editor, like Atom[1]. This could allow for dynamic feedback between CPC and the user, similar to the workflow in the Inform 7 IDE [57]. The general idea is to continuously parse the user input and suggest

---

[1] https://atom.io

valid autocompletion based on the Lexicon, as well as provide localized error messages on the fly to help the user write valid IE.

Bibliography

[1] A. L. Yarbus, *Eye movements during perception of complex objects.* Springer, 1967.

[2] J. K. Tsotsos and W. Kruijne, “Cognitive programs: Software for attention’s executive,” *Frontiers in Psychology - Cognition*, vol. 5, no. 1260, pp. doi–10, 2014.

[3] J. Weizenbaum, “Eliza—a computer program for the study of natural language communication between man and machine,” *Communications of the ACM*, vol. 9, no. 1, pp. 36–45, 1966.

[4] T. Winograd, “Procedures as a representation for data in a computer program for understanding natural language,” tech. rep., DTIC Document, 1971.

[5] J. E. Sammet, “The early history of cobol,” in *History of programming languages I*, pp. 199–243, ACM, 1978.

[6] J. Shaw, “Joss: A designer’s view of an experimental on-line computing system,” in *Proceedings of the October 27-29, 1964, fall joint computer conference, part I*, pp. 455–464, ACM, 1964.

[7] B. W. Ballard and A. W. Biermann, “Programming in natural language: “nlc” as a prototype,” in *Proceedings of the 1979 annual conference*, pp. 228–237, ACM, 1979.

[8] X. Liu and D. Wu, “Pie: programming in eliza,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 695–700, ACM, 2014.

[9] R. Egly, J. Driver, and R. D. Rafal, “Shifting visual attention between objects and locations: evidence from normal and parietal lesion subjects.,” *Journal of Experimental Psychology: General*, vol. 123, no. 2, p. 161, 1994.

[10] J. E. Raymond, K. L. Shapiro, and K. M. Arnell, “Temporary suppression of visual processing in an rsvp task: An attentional blink?,” *Journal of experimental psychology: Human perception and performance*, vol. 18, no. 3, p. 849, 1992.

[11] C. L. Folk, R. W. Remington, and J. C. Johnston, “Involuntary covert orienting is contingent on attentional control settings.,” *Journal of Experimental Psychology: Human perception and performance*, vol. 18, no. 4, p. 1030, 1992.

[12] N. P. Bichot and J. D. Schall, “Priming in macaque frontal cortex during popout visual search: feature-based facilitation and location-based inhibition of return,” *Journal of Neuroscience*, vol. 22, no. 11, pp. 4675–4685, 2002.

[13] J. K. Tsotsos, “Putting saliency in its place,” 2015.

[14] J. K. Tsotsos, S. M. Culhane, W. Y. K. Wai, Y. Lai, N. Davis, and F. Nuflo, “Modeling visual attention via selective tuning,” *Artificial intelligence*, vol. 78, no. 1, pp. 507–545, 1995.

[15] J. K. Tsotsos, *A computational perspective on visual attention.* MIT Press, 2011.

[16] S. Ullman, “Visual routines,” *Cognition*, vol. 18, no. 1-3, pp. 97–159, 1984.

[17] A. L. Rothenstein, A. Zaharescu, and J. K. Tsotsos, “Tarzann: A general purpose neural network simulator for visual attention modeling,” in *International Workshop on Attention and Performance in Computational Vision*, pp. 159–167, Springer, 2004.

[18] J. K. Tsotsos, “Analyzing vision at the complexity level,” *Behavioral and brain sciences*, vol. 13, no. 03, pp. 423–445, 1990.

[19] J. Tsotsos, I. Kotseruba, and C. Wloka, “A focus on selection for fixation,” *Journal of Eye Movement Research*, vol. 9, no. 5, 2016.

[20] I. Kotseruba, *Visual Attention in Dynamic Environments and Its Application To Playing Online Games.* PhD thesis, York University, 2016.

[21] J. K. Tsotsos, A. J. Rodríguez-Sánchez, A. L. Rothenstein, and E. Simine, “The different stages of visual recognition need different attentional binding strategies,” *Brain research*, vol. 1225, pp. 119–132, 2008.

[22] M. Handford, *Where's Waldo?* Candlewick Press, 1997.

[23] T. Winograd, *Thinking machines: Can there be? Are we?*, vol. 200. University of California Press, Berkeley, 1991.

[24] B. C. Smith, "The owl and the electric encyclopedia," *Artificial Intelligence*, vol. 47, no. 1, pp. 251–288, 1991.

[25] E. Cambria and B. White, "Jumping nlp curves: a review of natural language processing research," *IEEE Computational Intelligence Magazine*, vol. 9, no. 2, pp. 48–57, 2014.

[26] D. G. Bobrow, "Natural language input for a computer problem solving system," 1964.

[27] J. Weinzembaum, "Computer power and human reason: From judgement to calculation," 1976.

[28] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, "Natural language interfaces to databases–an introduction," *Natural language engineering*, vol. 1, no. 01, pp. 29–81, 1995.

[29] R. A. Capindale and R. G. Crawford, "Using a natural language interface with casual users," *International Journal of Man-Machine Studies*, vol. 32, no. 3, pp. 341–361, 1990.

[30] W. R. Ford, A. Chapanis, and G. D. Weeks, "Self-limited and unlimited word usage during problem solving in two telecommunication modes," *Journal of psycholinguistic Research*, vol. 8, no. 5, pp. 451–475, 1979.

[31] P. R. Michaelis, A. Chapanis, G. D. Weeks, and M. J. Kelly, "Word usage in interactive dialog with restricted and unrestricted vocabularies," *IEEE Transactions on Professional Communication*, no. 4, pp. 214–221, 1977.

[32] J. F. Pane, B. A. Myers, *et al.*, "Studying the language and structure in non-programmers' solutions to programming problems," *International Journal of Human-Computer Studies*, vol. 54, no. 2, pp. 237–264, 2001.

[33] W. A. Woods, R. M. Kaplan, and B. Nash-Webber, *The Lunar Sciences: Natural Language Information System: Final Report.* Bolt Beranek and Newman, 1972.

[34] C. Wang, M. Xiong, Q. Zhou, and Y. Yu, "Panto: A portable natural language interface to ontologies," in *European Semantic Web Conference*, pp. 473–487, Springer, 2007.

[35] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," *ACL 2014*, p. 55, 2014.

[36] E. Prud, A. Seaborne, *et al.*, "Sparql query language for rdf," 2006.

[37] D. Damljanovic, M. Agatonovic, and H. Cunningham, "Natural language interfaces to ontologies: Combining syntactic analysis and ontology-based lookup through the user interaction," in *Extended Semantic Web Conference*, pp. 106–120, Springer, 2010.

[38] Y. Li, H. Yang, and H. Jagadish, "Nalix: an interactive natural language interface for querying xml," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp. 900–902, ACM, 2005.

[39] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu, "Xquery 1.0: An xml query language," 2002.

[40] C. Finucane, G. Jing, and H. Kress-Gazit, "Ltlmop: Experimenting with language, temporal logic and robot control," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pp. 1988–1993, IEEE, 2010.

[41] I. Song, F. Guedea, F. Karray, Y. Dai, and I. El Khalil, "Natural language interface for mobile robot navigation control," in *Intelligent Control, 2004. Proceedings of the 2004 IEEE International Symposium on*, pp. 210–215, IEEE, 2004.

[42] R. Kiros, Y. Zhu, R. R. Salakhutdinov, R. Zemel, R. Urtasun, A. Torralba, and S. Fidler, "Skip-thought vectors," in *Advances in neural information processing systems*, pp. 3294–3302, 2015.

[43] J. E. Sammet, "The use of english as a programming language," *Communications of the ACM*, vol. 9, no. 3, pp. 228–230, 1966.

[44] A. Taylor, "The flow-matic and math-matic automatic programming systems," *Annual Review in Automatic Programming*, vol. 1, pp. 196–206, 1960.

[45] D. D. Chamberlin and R. F. Boyce, “Sequel: A structured english query language,” in *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pp. 249–264, ACM, 1974.

[46] A. W. Biermann, B. W. Ballard, and A. H. Sigmon, “An experimental study of natural language programming,” *International journal of man-machine studies*, vol. 18, no. 1, pp. 71–87, 1983.

[47] D. Shafer, *HyperTalk Programming.* Indianapolis, IN, USA: Sams, 1988.

[48] D. Goodman, *Complete HyperCard 2.0 Handbook.* Random House Inc., 1990.

[49] W. R. Cook, “Applescript,” in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 1–1, ACM, 2007.

[50] D. Price, E. Rilofff, J. Zachary, and B. Harvey, “Naturaljava: a natural language interface for programming in java,” in *Proceedings of the 5th international conference on Intelligent user interfaces*, pp. 207–211, ACM, 2000.

[51] J. Gosling, *The Java language specification.* Addison-Wesley Professional, 2000.

[52] H. Liu and H. Lieberman, “Metafor: Visualizing stories as code,” in *Proceedings of the 10th international conference on Intelligent user interfaces*, pp. 305–307, ACM, 2005.

[53] H. Liu, “Montylingua: An end-to-end natural language processor with common sense,” 2004.

[54] H. Liu and P. Singh, “Conceptnet—a practical commonsense reasoning tool-kit,” *BT technology journal*, vol. 22, no. 4, pp. 211–226, 2004.

[55] P. Singh, T. Lin, E. T. Mueller, G. Lim, T. Perkins, and W. L. Zhu, “Open mind common sense: Knowledge acquisition from the general public,” in *OTM Confederated International Conferences “On the Move to Meaningful Internet Systems”*, pp. 1223–1237, Springer, 2002.

[56] R. Knöll and M. Mezini, “Pegasus: first steps toward a naturalistic programming language,” in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 542–559, ACM, 2006.

[57] G. Nelson, “Natural language, semantic analysis, and interactive fiction,” *IF Theory Reader*, vol. 141, 2006.

[58] G. Little and R. C. Miller, “Keyword programming in java,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 84–93, ACM, 2007.

[59] W. Feurzeig and S. Papert, “The logo programming language,” 1967.

[60] V. Le, S. Gulwani, and Z. Su, “Smartsynth: Synthesizing smartphone automation scripts from natural language,” in *Proceeding of the 11th*

annual international conference on Mobile systems, applications, and services, pp. 193–206, ACM, 2013.

[61] M. H. Manshadi, D. Gildea, and J. F. Allen, "Integrating programming by example and natural language programming.," in *AAAI*, 2013.

[62] M. Landhäußer, S. Weigelt, and W. F. Tichy, "Nlci: a natural language command interpreter," *Automated Software Engineering*, pp. 1–23, 2016.

[63] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[64] K. Hasselström and J. Åslund, "The shakespeare programming language," Apr 2001.

[65] S. Cozens, *Advanced perl programming.* O'Reilly, 2005.

[66] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[67] W.-O. Huijsen, "Controlled language–an introduction," in *Proceedings of CLAW*, vol. 98, pp. 1–15, 1998.

[68] T. Kuhn, "A survey and classification of controlled natural languages," *Computational Linguistics*, vol. 40, no. 1, pp. 121–170, 2014.

[69] N. E. Fuchs, K. Kaljurand, and T. Kuhn, "Attempto controlled english for knowledge representation," in *Reasoning Web*, pp. 104–124, Springer, 2008.

[70] J. F. Sowa, "Ontology, metadata, and semiotics," in *Conceptual structures: Logical, linguistic, and computational issues*, pp. 55–81, Springer, 2000.

[71] C. K. Ogden, *Basic English: A general introduction with rules and grammar.* No. 29, K. Paul, Trench, Trubner, 1944.

[72] D. D. Bourland Jr, "A linguistic note: Writing in e-prime," *General Semantics Bulletin*, vol. 32, no. 3, pp. 111–114, 1965.

[73] C. A. Verbeke, "Caterpillar fundamental english.," *Training and Development Journal*, vol. 27, no. 2, pp. 36–40, 1973.

[74] F. Robertson, "Airspeak: Radiotelegraphy for pilots," 1987.

[75] AeroSpace and D. I. A. of Europe, *ASD Simplified Technical English: Specification ASD-STE100.* ASD AeroSpace and Defence, 2005.

[76] N. Rychtyckyj, "Standard language at ford motor company: A case study in controlled language development and deployment," *Cambridge, Massachussets*, 2006.

[77] P. Bollen, *SBVR: A Fact-Oriented OMG Standard*, pp. 718–727. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[78] R. Power and D. Scott, "Multilingual authoring using feedback texts," in *Proceedings of the 17th international conference on Computational linguistics-Volume 2*, pp. 1053–1059, Association for Computational Linguistics, 1998.

[79] I. Pratt-Hartmann, “A two-variable fragment of english,” *Journal of Logic, Language and Information*, vol. 12, no. 1, pp. 13–45, 2003.

[80] P. Martin, “Knowledge representation in cglf, cgif, kif, frame-cg and formalized-english,” in *Conceptual Structures: Integration and Interfaces*, pp. 77–91, Springer, 2002.

[81] O. Ritchie and K. Thompson, “The unix time-sharing system,” *The Bell System Technical Journal*, vol. 57, no. 6, pp. 1905–1929, 1978.

[82] T. Parr, *The definitive ANTLR 4 reference.* Pragmatic Bookshelf, 2013.

[83] N. Wirth, “Extended backus-naur form (ebnf),” *ISO/IEC*, vol. 14977, p. 2996, 1996.

[84] C. Böhm and G. Jacopini, “Flow diagrams, turing machines and languages with only two formation rules,” *Communications of the ACM*, vol. 9, no. 5, pp. 366–371, 1966.

[85] M. L. Minsky, *Computation: finite and infinite machines.* Prentice-Hall, Inc., 1967.

[86] A. M. Turing, “On computable numbers, with an application to the entscheidungsproblem,” *Proceedings of the London mathematical society*, vol. 2, no. 1, pp. 230–265, 1937.

[87] A. Church, “An unsolvable problem of elementary number theory,” *American journal of mathematics*, vol. 58, no. 2, pp. 345–363, 1936.

[88] D. Hilbert, W. Ackermann, and R. E. Luce, *Principles of mathematical logic*, vol. 69. American Mathematical Soc., 1950.

[89] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases: the logical level.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[90] M. Cook, "Universality in elementary cellular automata," *Complex systems*, vol. 15, no. 1, pp. 1–40, 2004.

[91] C. Böhm, "On a family of turing machines and the related programming language," *ICC Bull*, vol. 3, no. 3, pp. 187–194, 1964.

[92] G. Van Rossum and F. L. Drake, *Python language reference manual.* Network Theory, 2003.

[93] B. C. Smith, *Procedural reflection in programming languages.* PhD thesis, Massachusetts Institute of Technology, 1982.

[94] S. Houben, "Turing compliant?." Python mailing list (python-list@python.org), Sept 1999.

[95] O. Abid, "Decomposing cognitive programs into neural primitives: An incremental approach," Master's thesis, York University, 2017. Unpublished work.

[96] M. Kleiner, D. Brainard, D. Pelli, A. Ingling, R. Murray, C. Broussard, *et al.*, "What's new in psychtoolbox-3," *Perception*, vol. 36, no. 14, p. 1, 2007.

[97] S. Mathôt, D. Schreij, and J. Theeuwes, “Opensesame: An open-source, graphical experiment builder for the social sciences,” *Behavior research methods*, vol. 44, no. 2, pp. 314–324, 2012.

[98] W. McGugan, *Beginning game development with Python and Pygame: from novice to professional.* Apress, 2007.

[99] N. A. Macmillan and C. D. Creelman, *Detection theory: A user’s guide.* Psychology press, 2004.

[100] F. Cutzu and J. K. Tsotsos, “The selective tuning model of attention: psychophysical evidence for a suppressive annulus around an attended item,” *Vision research*, vol. 43, no. 2, pp. 205–219, 2003.

[101] S. Yoo. Private communication, 2017.

Appendix A

Additional Implementation Details

A.1 CPD ANTLR Grammar

Listing A.1 contains the complete ANTLR grammar that describes CPDs. Since the IE grammar is a subset of the CPD grammar, IE's grammar is not shown in a separate listing, as everything needed to parse IE is included in Listing A.1.

Listing A.1: ANTLR grammar describing CPDs.

```
grammar CPC;

@lexer::header {
import lexicon.*;
import lexicon.base.*;
}

@lexer::members {
private LexiconManager lexiconManager = new LexiconManager
↪ ();
```

```
public CPCLexer(CharStream input, LexiconManager
↪ lexiconManager) {
        this(input);
        this.lexiconManager = lexiconManager;
}
}

/*******************************************************
******************** SENTENCES ***********************
*******************************************************/

root
    : SETUP_BLOCK? statement*
    ;

SETUP_BLOCK
    : OPENSB .*? CLOSESB
    ;

statement
    : instructionBlock PERIOD
    | pleaseStatement PERIOD
    | CODE_BLOCK
    ;

CODE_BLOCK
    : OPENCB .*? CLOSECB
    ;

instructionBlock
    : instruction (THEN instruction)* SEMICOLON?
    ;

instruction
    : command
    | fact
    | controlFlow
    ;

controlFlow
    : ifStatement
    | forEachStatement
    | whileStatement
    | whenStatement
```

```
    ;

fact
    : factClause AND fact
    | factClause OR fact
    | factClause
    ;

command
    : imperativeVerb specifier* selector specifier*
    ;

factClause
    : factClauseLHS factVerb factClauseRHS
    ;

ifStatement
    : IF fact instructionBlock (ELSE instructionBlock)?
    ;

whenStatement
    : WHEN fact instructionBlock
    ;

whileStatement
    : WHILE fact instructionBlock
    ;

forEachStatement
    : FOR EACH selector specifier* instructionBlock
    ;

pleaseStatement
    : PLEASE_STATEMENT
    ;

factClauseLHS
    : selector specifier*
    ;

factClauseRHS
    : specifier* selector? specifier*
    | modifier
    ;
```

```
PLEASE_STATEMENT
    : PLEASE ~'.'*
    ;


/********************************************************
*********************** PHRASES *************************
*********************************************************/

selector
    : ARTICLE? modifier* target
    ;

specifier
    : relationship selector
    ;

/********************************************************
************************ FRAGMENTS **********************
*********************************************************/

factVerb
    : AUX_VERB? FACT_VERB_DICT parameters?
    ;

imperativeVerb
    : IMPERATIVE_VERB_DICT parameters?
    ;

modifier
    : MODIFIER_DICT
    ;

target
    : TARGET_DICT parameters?
    | (STRING|INT)
    ;

relationship
    : RELATIONSHIP_DICT parameters?
    ;

parameters
    : LPAREN parameter+ RPAREN
    ;
```

```
parameter
    : (STRING|INT|IDENTIFIER)
    ;

/********************************************************
************************ LEXICON ************************
*********************************************************/

TARGET_DICT
    : [a-zA-Z ]+ { this.lexiconManager.hasDictionaryEntry(
    ↪ Target.class, getText()) }?
    ;

MODIFIER_DICT
    : [a-zA-Z ]+ { this.lexiconManager.hasDictionaryEntry(
    ↪ Modifier.class, getText()) }?
    ;

RELATIONSHIP_DICT
    : [a-zA-Z ]+ { this.lexiconManager.hasDictionaryEntry(
    ↪ Relationship.class, getText()) }?
    ;

IMPERATIVE_VERB_DICT
    : [a-zA-Z ]+ { this.lexiconManager.hasDictionaryEntry(
    ↪ ImperativeVerb.class, getText()) }?
    ;

FACT_VERB_DICT
    : [a-zA-Z ]+ { this.lexiconManager.hasDictionaryEntry(
    ↪ FactVerb.class, getText()) }?
    ;

AUX_VERB: 'will';

IF: ('If'|'if');
FOR: ('For'|'for');
WHILE: ('While'|'while');
WHEN: ('When'|'when');

THEN: 'then';
ELSE: ('else'|'otherwise');
EACH: ('each'|'every');
```

```
PLEASE: 'Please';

ARTICLE
    : ('An'|'an'
    | 'A'|'a'
    | 'The'|'the')
    ;

AND: 'and';
OR: 'or';


/*******************************************************
******************** TERMINALS ***********************
*******************************************************/

INT: [0-9]+ ;
STRING
    : '"' (ESC | ~ ["\\])* '"'
    ;

fragment ESC
    : '\\' (["\\/bfnrt] | UNICODE)
    ;

fragment UNICODE
    : 'u' HEX HEX HEX HEX
    ;

fragment HEX
    : [0-9a-fA-F]
    ;

PERIOD: '.';
LPAREN: '(';
RPAREN: ')';
OPENCB: '[%';
CLOSECB: '%]';
OPENSB: '{%';
CLOSESB: '%}';
SEMICOLON: ';';

IDENTIFIER: [a-zA-Z_][a-zA-Z0-9_]*('.'[a-zA-Z0-9_]+)*;

/*******************************************************
```

```
***************** IGNORE OR SKIP **********************
*******************************************************/

COMMENT
    : '#'.*? [\r\n]+ -> channel(2)
    ;

WS
    : [ \t\n\r]+ -> channel(HIDDEN)
    ;

PUNCTUATION
    : [,] -> channel(HIDDEN)
    ;
```

A.2 Lexicon Entries

Imp and CPC implementations have separate Lexicons. *Imp*'s Lexicon contains Fragments for common operations on files and directories. They are enumerated in a hierarchical fashion in Listing A.2. CPC's Lexicon Fragments focus on describing CPDs for TarzaNN 3.0, and are enumerated in Listing A.3.

Listing A.2: Hierarchical listing of Fragments in *Imp*'s Lexicon.

```
- Fragment
  - Target
    - Background
    - Register
    - Value
    - Line
     - Image
    - Time
    - Name
    - Reference
      - It
    - File
      - Directory
    - Parent
    - Child
  - Verb
    - ImperativeVerb
      - Change
      - Extract
      - Find
      - Increment
      - Return
      - FileOp
        - Move
        - Rename
        - Copy
      - Download
      - Run
      - Set
      - Decrement
      - Say
    - FactVerb
      - Is
  - Relationship
    - Named
    - On
    - To
    - In
    - SetOperation
      - SetIntersection
      - SetUnion
    - Of
```

```
        - Containing
        - At
    - Modifier
        - Position
        - Count
        - Appearance
            - Color
                - Green
                - Blue
                - Black
                - White
                - Red
```

Listing A.3: Hierarchical listing of Fragments in CPC's Lexicon.

```
- Fragment
    - Target
        - Cue
        - TargetFrag
        - Mask
        - RigElement
            - Display
            - Button
            - Image
                - Sequence
                - Video
            - Keyboard
        - Reference
            - You
            - It
        - Shape
            - Dot
            - Rectangle
                - Square
            - Character
                - T
                - X
                - PlusSign
                - L
                - EqualsSign
        - Ready
    - Verb
        - ImperativeVerb
```

```
            - Look
            - Release
            - Press
            - Say
            - Hold
        - FactVerb
            - Flash
            - Saw
            - Appear
            - See
            - BePresented
            - Is
            - Disappear
    - Relationship
        - After
        - Between
        - On
        - To
        - In
        - SetOperation
            - SetIntersection
            - SetUnion
        - Of
        - At
    - Modifier
        - Fixation
        - Position
            - Center
            - Right
            - Left
        - Count
            - Singleton
        - Appearance
            - Color
                - Green
                - Blue
                - Black
                - White
                - Red
            - Filled
        - Mouse
        - Test
```