**Running the Analyzer from a UNIX Environment**

Sean Trott (Draft 2)

Email: seantrott@icsi.berkeley.edu

**Link to “compling” directory: < >**

Should contain following directories: lib, src, and grammars

Although there is a GUI in the form of the ECG Workbench (Gilardi & Feldman, 2008), it is also possible to run the ECG Analyzer from a UNIX environment with access to the proper files. This method is used in connecting the Analyzer to an “action” end – the Problem Solver (Figure 1). This particular tutorial is based on a simple Robot simulation application, but the simulator (MORSE: <http://www.openrobots.org/morse/doc/1.2/user/installation.html>) is platform-dependent and is not included. However, utterances from the “robots” grammar will yield N-Tuples (Figure 1), which can be viewed interactively in a “debugging” mode; these N-Tuples provide specifications to the Problem Solver in the form of a Python dictionary converting the initial linguistic input to a task description.

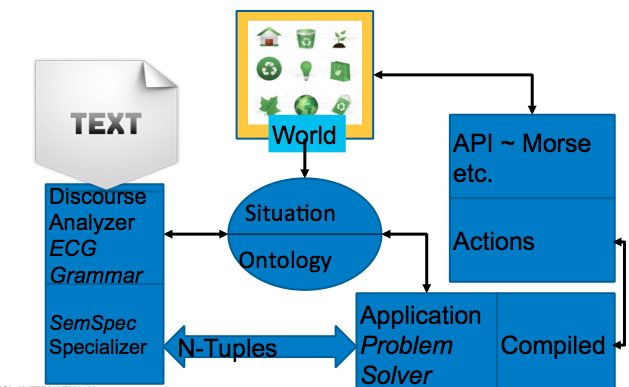


Figure 1

To run the Analyzer and Specializer properly, you will need access to the “compling” package of files <**LINK**> - this includes the Java files for the ECG Analyzer, as well as the Python files necessary to run the Specializer. This package should also include a directory called “grammars”, which comes with a version of the Robots grammar.

**Requirements:**

* You will need to be running at least Python3.3: <https://www.python.org/download/releases/3.3.0/>
* You will need to have JYTHON installed: <http://www.jython.org/downloads.html>

**Setting Up the Analyzer**

Before you can run the Specializer, you need to set up the Analyzer. Open up your favorite command line shell (examples assume you’re running a Bash shell), and navigate to your newly downloaded “compling” directory. Once there, enter the following command:

**Mac / Linux:**

The “compling” package includes two command scripts: analyzer.sh and analyzer.bat. If you’re running on a Mac or Linux system, run the shell script, called analyzer.sh:

User: compling$ sh analyzer.sh

Or:

User: compling$ bash analyzer.sh

Alternatively, if you wish to execute the commands included in the shell script manually, enter the following commands:

User:compling$ export JYTHONPATH=lib/compling.core.jar:src/main

This connects the Java ECG Analyzer, which is located in the /lib folder, to the /src/main folder, which contains all of the Python files, such as specializer.py and analyzer.py. Once this is done, enter the following JYTHON command:

User:compling$ jython –m analyzer *grammars/*robots.prefs[[1]](#footnote-1)

This connects the grammar to a local host, which can be accessed from elsewhere. For example, the Specializer module sets up a proxy Analyzer class connecting to that host, and calls the "parse" function from there.

**Windows:**

If you’re running a Windows operating system, you can run the Batch command:

$ START analyzer.bat

This executes a Batch command from the analyzer.bat file. If you wish to execute the commands separately, you can do so using the commands below:

$ set JYTHONPATH=lib\compling.core.jar;src\main

$ jython -m analyzer grammars\robots.prefs

**Specializer**

**About the Specializer**

When a line of text is entered into the command line for the ECG Analyzer, the Analyzer yields a parse (if possible), which takes the form of a Semantic Specification (SemSpec). The SemSpec is a graph-type data structure, which contains both the syntactic and semantic elements of the parse, embedded within higher-level grammatical Constructions and Schemas. You should have already tried examples with the Workbench and become familiar with the graphical presentation of some simple robot examples. The SemSpec contains all the necessary linguistic information about the sentence, but its structure is highly complex – this is where the Specializer comes in.

The role of the Specializer is to extract the important information from this SemSpec and package it into an easily interpretable message, called an “N-Tuple”. This N-Tuple is passed to the Problem Solver, which unpacks the N-Tuple and determines how to use the information. Essentially, the task is to “specialize” the complex SemSpec into a more succinct set of instructions for the Problem Solver. (Cf Figure 1)

For this example, the Specializer is used in conjunction with a Robotics Problem Solver and a simple robot simulation in an environment that contains four boxes of different sizes and colors, as well as a robotic vehicle, which can obey commands and answer questions.

**Using the Specializer**

Once the Analyzer is running in Terminal, open a second Terminal window. Make sure you are in the same *compling* directory. If you’re running a Mac, enter the following command:

User:compling$ python3 src/main/specializer.py[[2]](#footnote-2)

If you’re running Windows, enter this command:

User:compling$ python src\main\specializer.py

This will result in the following prompt on the Terminal window:

Command or q/Q to quit, d/D for Debug mode>

As the prompt suggests, you can press “q” or “Q” to exit out of the interactive Specializer window. Press “d” or “D” to enter Debug mode. In Debug mode, the specializer prints out the packaged N-Tuple before sending it to the Problem Solver. Additionally, the N-Tuple is copied as a JSON object to a newly created file in the /src/main folder, which can be read into a Problem Solver separately.

\***Note**: In the current release, the Specializer does not send the N-Tuple to the Solver; the relevant lines (lines 862-863 in specializer.py) have been commented out for the sake of simplicity. If you wish to interact with a text-based Problem Solver, simply remove the “#” from the beginning of those lines, and Terminal / Command Line will now print out a representation of the Robot’s actions, as well as the N-Tuple. If you’re running the MORSE simulator, uncommenting these lines will call the Morse Problem Solver.

Additionally, the boxes in the simulated world each have assigned names, which you can use to refer to them (e.g., “Box1” instead of “the big red box”). The sizes “big” and “small” correlate to the values 2 and 1, respectively. To see the list of names and how they relate to the sizes and colors, enter the following command:

Command or q/Q to quit, d/D for Debug mode> names

Box4 is of type box, with a color of red and a size of 1.

Box3 is of type box, with a color of green and a size of 2.

Box2 is of type box, with a color of blue and a size of 2.

Box1 is of type box, with a color of red and a size of 2.

To use the Specializer, simply type the sentence you wish to parse into the prompt. The Specializer calls the “parse” function of the Analyzer, which, as mentioned above, returns a SemSpec. The Specializer then crawls the SemSpec and produces a robot specific N-Tuple:

Command or q/Q to quit, d/D for Debug mode> d

Debug mode is True

Command or q/Q to quit, d/D for Debug mode> Robot1, move to the blue box!

Struct(parameters=[

Struct(kind='execute',

action='move',

speed=0.5,

direction=None,

heading=None,

control\_state='ongoing',

protagonist='robot1\_instance',

distance=Struct(units='square', value=8),

goal={'objectDescriptor': {'color': 'blue', 'givenness': 'uniquelyIdentifiable', 'type': 'box'}})],

predicate\_type='command',

return\_type='error\_descriptor')

The “Struct” object printed below the input sentence is the final N-Tuple, which is passed to the Problem Solver. Inputting different sentences or commands will result in different N-Tuples – both in the content and the structure. For example, asking a question will result in a “query” N-Tuple, as opposed to an “executable”. Below is a comprehensive listing of the types of commands and queries currently available in the Analyzer and Specializer; it is not an exhaustive list of every possible sentence, but it does cover every sentence *type* (e.g., does not include both “move to Box1” and “move to Box2”). If you want to observe the N-Tuple produced by each command, remember to press “d” to ensure that Debug mode is set to “True”. Additionally, the Specializer is set to only encode Utterances, which utilize the “Discourse Element” schema; this is because the “Discourse Element” schema specifies a *mood* for the Utterance, such as “imperative”, “Y/N question”, “Wh-question”, “conditional imperative”, and “definition”.

**Simple Imperatives:**

Robot1, move to the big red box!

Robot1, move to Box1!

Robot1, push the blue box!

**Serial Processes:**

Robot1, move to the big red box then move to the small red box!

Robot1, move to the blue box then return!

Robot1, move to the green box then push the blue box!

**Conditionals**:

Robot1, if the green box is near the small red box, move to the blue box!

Robot1, if Box1 is red, move to Box2!

**Questions:** (Note that the parser requires the question words to begin with a lower-case letter, rather than an upper-case, because that’s how they are specified in the lexicon)

which boxes are near the green box?

which boxes are red?

which box is blue?

is the box near the blue box green?

where is the green box?

**Definitions:**

define visit QL1 as move to QL1 then return.

Robot1, visit the big red box!

**Referent Resolution**:

Robot1, if the small box is red, push it!

Robot1, move to Box1!

where is it?

**How the Specializer Works**

The above tutorial on using the Specializer should be sufficient to become better acquainted with how the grammar and SemSpec is converted into an easily interpretable message. However, if you wish to learn more about how the Specializer actually uses the task specification and the SemSpec to produce an N-Tuple, continue reading below:

**Templates:**

One implementation feature of the Specializer is the use of general “templates”, which are initialized as Python Dictionary types when the Specializer class is instantiated. These templates specify the task specific content and operate at several levels of abstraction:

* N-Tuple Template
* “Mood” Templates (Imperatives, Conditionals, Queries, etc.)
* Process Templates (Cause Effect, Motion Path, Stasis, Force Application, etc.)

The N-Tuple Template is the highest level, and it contains:

* Predicate type: indicates what type of “mood” the parameters are, such as “query”, “conditionals”, “assertion”, “definition”, or “command”
* Return type: indicates what type of value the Solver is expected to return – this is simply an “error-descriptor” in most cases, but is a “Boolean” for Yes/No questions and a Referent for WH-questions
* Parameters: the set of specific information about the Mood and Process – this is the core of the Specializer’s message to the Solver.

*Sample N-Tuple format for Y/N question:*

N-Tuple: {Return type: ‘Boolean’,

Predicate type: ‘Query’,

Parameters: Yes / No Question = {….}}

The “Mood” Templates are used as the Parameters entry in the N-Tuple Template. In the SemSpec, different types of Utterances have different linguistic moods, and the Discourse-Element schema maps these moods according to the type of Utterance. Depending on the mood, different templates are used as the Parameters entry in the N-Tuple.

**Example**:

(1) self.\_YN = dict(kind = 'query',

protagonist=None,

action=None,

predication=None)

(2) self.\_conditional = dict(kind='conditional',

condition=self.\_YN

command = self.\_execute)

Example (1) has the mood of “Yes / No Question”, such as “is Box2 red?”. The different keys in the dictionary begin as “None”, and are filled in by the Specializer. From this information, the Solver can return a Boolean: Yes or No.

Example (2) has the mood of “Conditional Imperative”, such as: “if Box2 is red, push it!”. This is a more complex Template, because it actually contains two simpler templates – the Y/N template and the executable template. The Solver can evaluate the condition, and depending on whether or not the condition is true, it can execute the command. In general, the structure of the Template is designed to specify the Solver’s processing.

**Filling in the Templates**:

Once the mood of the sentence has been established, the Specializer must determine the type of process the sentence refers to. There are a number of simple processes supported in both the underlying grammar and the Solver, as well as complex compound processes (Serial Processes and Causal Processes).

Different types of processes make use of different meaning Schemas, which have different roles. The sentence “Robot1, move to the big blue box!” uses the MotionPath schema, which itself contains the Source-Path-Goal schema. There are several very important role mappings from the input sentence. The sentence elements are listed below with the corresponding SemSpec mapping, as well as the parent Schema.

“Robot1” 🡪 mover / protagonist (MotionPath)

“Robot1” 🡪 trajector (Source-Path-Goal)

“Move” 🡪 actionary (MotionPath)

“the big blue box” 🡪 goal / landmark (Source-Path-Goal)

The SemSpec makes explicit a couple of necessary details to the Specializer, which help the Specializer determine how to handle processing this input. First of all, the “mood” is Imperative, so the Template should be an executable. Second, the process is MotionPath, which is a “simple” process.

As mentioned in Step 3 above, the process will be dispatched to the relevant function. In this case, the function is called “params\_for\_motionPath”. In order to function properly, the Solver needs to know information about the following roles:

* Mover: which is the mover / trajector in the Motion event?
* Actionary: is a particular action specified (run, walk, move)?
* Heading: is a heading specified (North, South, etc.)?
* Distance: is a distance specified (4 inches, etc.)?
* Direction: is a direction specified?
* Goal: Is a goal (endpoint) of the Motion event specified?

**Ontology:** One of the key elements of the Robot system is its foundation in a complex ontology. Elements from this ontology are shared between the Analyzer and the Solver, and represent the entities and values that exist within the scope of the simulated world. The meaning Schemas and grammatical Constructions are supported by a lattice of ontology items (Oliva, Feldman, Gilardi, & Dodge, 2013). Ultimately, all meaning roles point back towards ontology items, such as “box” or “blue”, and these are the values encoded within the N-Tuple.

(**type entity sub item)**

**(type shaped sub entity)**

(**type artifact sub shaped)**

(**type container sub artifact)**

(**type box sub container)**

Sample ontology lattice excerpt (Oliva, Feldman, Gilardi, & Dodge, 2013)

**Object-Descriptors:** In the sentence given previously (“Robot1, move to the big blue box”), three of these roles are referenced. The mover is “Robot1”, the actionary is “move”, and the goal is “the big blue box”. In this case, the goal is a description of an object, not a reference to a particular instance (such as Box1). In order to include all of the relevant information about the goal, the Specializer represents the goal as a general Object Descriptor, which is a Python dictionary embedded within the larger template. The Specializer performs a depth-first-search[[3]](#footnote-3) on the SemSpec and collects relevant information about the object, adding it to the object descriptor. Ultimately, the Object Descriptor looks like this:

Object-Descriptor: {type: box,

color: blue,

size: big,

givenness: uniquely-Identifiable}

Then, the Solver can use this information to locate the relevant object. In certain cases, an object is *also* described by its location, such as: “the box near the red box”. In these cases, a *Location-Descriptor* is embedded within the Object-descriptor Python dictionary.

**Location-Descriptors:** Location descriptors are handled in a similar way as modifiers or adjectives, but the representation is considerably more complex, due to the added complexity of spatial relations. If an item is specified as “**the box near the red box**”, the Solver needs to know what type of spatial relation is being referenced (near, behind, in front of, in, etc.), as well as information about the landmark itself. Thus, the landmark is represented as another Object-Descriptor.

Object-Descriptor: {type: box,

givenness: uniquely-Identifiable,

Location-Descriptor:

{relation: near,

Object-Descriptor: {type: box,

color: red,

givenness: uniquely- Identifiable}}}

Using this information, the Solver can identify the box that is “near” the red box. The format of Object-Descriptors and embedded Location-Descriptors allows them to be recursive, which is intended to model the recursive nature of language.

**Referent Resolution:**

The Specializer also performs basic referent resolution. Co-reference is when two expressions in a text refer to the same thing, such as:

“**Sally** and **her** [Sally’s] friend went for a walk.”

Co-reference resolution involves the resolving of a pronoun or otherwise anaphoric semantic element with its predecessor. It is a difficult problem in computational linguistics because of the high degree of complexity in language. However, in certain cases, there are systems of rules that seem to work. Additionally, we can use ontological classification and the semantic parse to guide the resolution, in addition to syntactic placement (Oliva, Feldman, Gilardi, & Dodge, 2013).

Consider the sentence: “Robot1, move to the green box then push it.” Here, *it* should be co-indexed with *the green box* – what the sentence really means is “move to the green box then push the green box.” However, the Analyzer does not attempt to co-index pronouns with their antecedents. The Solver, of course, only receives information in the N-Tuple, and would not be able to do this. Thus, it is up to the Specializer to make an estimation of what a particular pronoun refers to.

So, in this sentence, when the Specializer crawls the SemSpec and reaches the second of the serial process (“push it”), it determines that the “acted upon” object is a pronoun in need of resolution. The grammar simplifies this process by having pronoun Referents map to the “antecedent” type in the ontology:

self.m.referent 🡨@antecedent

Antecedent is constrained in the ontology to be an entity:

(type antecedent sub entity)

The current system for referent resolution is relatively simple.

**Stack:** First of all, the Specializer keeps a backlog of object descriptors and location descriptors in the form of a Stack: “first in, last out”. The most recent entries are at the top of the Stack, which means they will be the first observed during the resolution process. This gets at one of the core rules that seems to govern co-reference: in general, more recent expressions are more likely to be the antecedent for a pronoun.

**Syntactic Heads:** Second, not every object descriptor or location descriptor is stored. Another general grammatical rule is that the syntactic “heads” of an NP are more likely to be the antecedent of a pronoun than any NP contained within the branch. The “head” determines the syntactic type of a branch; in “the box near the table”, *the box* would be the head, while *the table* is an NP whose relational position modifies *the box*. English tends to be a head-initial language, resulting in a right-branching parse. In other words, the Specializer would not store *the table* in its Stack, but it would store *the box near the table*.

**Storage of entire Object-Descriptor:** Third, a pronoun often refers to an entire Noun-Phrase, which can be considerably more complex than “Box1” or “the green box”. A Noun-Phrase could be: “the box behind the green box near the small red box”. This is why the Stack contains the entire Object Descriptor or Location Descriptor. The NP could be used in the following command:

>>> “Robot1, if the box behind the green box near the small red box is blue, push **it**!”

The Specializer realizes that *it* must be resolved, so it calls the Reference Resolution function. This function “pops” off the item at the top of the Stack, which happens to be the following Object Descriptor:

{'objectDescriptor':

{'locationDescriptor': {'relation': 'behind', 'objectDescriptor':

{'locationDescriptor': {'relation': 'near', 'objectDescriptor': {'size': 'small', 'givenness': 'uniquelyIdentifiable',

'color': 'red',

'type': 'box'}},

'givenness': 'uniquelyIdentifiable',

'color': 'green',

'type': 'box'}},

'givenness': 'uniquelyIdentifiable',

'type': 'box'}}

After this Object Descriptor is inspected for compatibility, it is simply passed into the N-Tuple in the place of the pronoun, so the Solver gets a message identical to the message it would get from the following sentence:

>>> “Robot1, if the box behind the green box near the small red box is blue, push the box behind the green box near the small red box!”

**Ontological Compatibility:** Fourth, the Specializer checks the ontological category of the Object Descriptor or Location Descriptor, and evaluates whether it is compatible with the usage of the pronoun. This relies on the *Ontological Lattice*, a complex hierarchy of category types. In the grammar, these ontological categories constrain the output of the SemSpec. Objects in the world are mapped to a particular ontological category, such as “box”:

(type **box** sub **container** **moveable**)

If the command is something like “push it”, *it* must refer to something that can be moved. Thus, the Specializer checks the ontological category of the “popped” referent, and ensures that it is compatible with the *actionary* (such as “force application” or “move”).

self.analyzer.issubtype('ONTOLOGY', popped['objectDescriptor']['type'], 'moveable')

The above code checks if the “type” of the object referenced is a *child* of “moveable” in the ontology – in other words, whether the object can be moved.

Another example of checking compatibility is with predication or questions, such as: “is the box red?” In the grammar, adjectives rely on a Modification schema, and the “color” property’s domain (the range of ontological types it can apply to) is restricted to Physical Entities. In other words, abstract entities cannot have the Color property – at least not in the literal sense. “Box” is a Physical Entity, so that sentence would parse properly.

However, as before, the grammar does not perform co-reference resolution, so it cannot check if *it* in “is it red?” is a Physical Entity; *it* could refer to any type of antecedent. Thus, the Reference Resolution function must also take in predication as an argument. Ultimately, it tests using the same ontological lattice function:

self.analyzer.issubtype('ONTOLOGY', popped['objectDescriptor']['type'], 'physicalEntity')

In summation, the co-reference resolution currently used in the Specializer operates under four assumptions:

1. More recent referents are more likely to be an antecedent.
2. Referents that are syntactic “heads” of an NP are more likely to be an antecedent than an NP contained within the branch.
3. An antecedent NP could be considerably more complex than a determiner and a noun; thus, the whole branch is stored as an Object Descriptor.
4. The antecedent of a pronoun must be semantically compatible; this involves ontology-driven classification.

**References:**

Gilardi, Luca; Feldman, Jerome. 2008. *A Brief Introduction to ECG Workbench and a First English Grammar.* <http://www1.icsi.berkeley.edu/~lucag/>

Oliva, Jesus; Feldman, Jerome; Gilardi, Luca; Dodge, Ellen. 2013. *Ontology Driven Contextual Best Fit in Embodied Construction Grammar.*

1. Note that the italicized file-path above (compling/grammars) depends on where you’ve installed the “robots” grammar. [↑](#footnote-ref-1)
2. If you are running with the MORSE simulator, the command will instead be: python3 src/main/specializer.py “-s morse”. For the purposes of this tutorial, we will focus only on the text-based Specializer and Problem Solver. [↑](#footnote-ref-2)
3. A type of algorithm used to traverse tree / graph structures, in which the pointer begins at an “entry” node and explores as far as possible along each branch before backtracking. This method was chosen rather than a breadth-first-search because the relevant information is generally contained very far down along a branch. [↑](#footnote-ref-3)