# AI course project: Shrdlite

## Contents



*Note: This document also exists in a PDF version.*

## 1 Introduction

In this project you will implement a dialogue system for controlling a robot that lives in a virtual block world. The robot can move around objects of different forms, colors and sizes, and it can answer questions about the world and ask for clarification whenever necessary.

The inspiration is Shrdlu, by Terry Winograd 1970. A short reading list about Shrdlu is at the end of this page. For Winograd it was a PhD project, but now there are tools available that make things easier: grammar formalisms, ontologies, planners, etc.

You will not have time to create a system that can do everything that Shrdlu can, so you only have to implement a subset of Shrdlu's capabilities.

### 1.1 Try it yourself

The Shrdlite system is already running here:

http://www.grammaticalframework.org/~peter/shrdlite/shrdlite.html

But this an old version, that has a pretty lousy planner, is bad, slow and even incorrect! Your task is to make a better planner, and perhaps also extend the functionality of the system.

You can test the program right here and now. Select one of the example sentences and press 'enter', such as this one (for the "medium" world):

> *put the white ball in a box on the floor*

The system parses the sentence, and tries to infer a sequence of simple actions that accomplishes the task. Which it cannot do.

## 1.2 Install the initial project template

There is a Git repository with all files that you need to get started:

> https://github.com/heatherleaf/shrdlite-template

Hopefully this will help you to with the graphical interface, but if you can't get it to work then it's no big deal: You don't need graphics to complete the project, but it's more fun!

## 1.3 Fire

All submissions should be done via the Fire system, and the first thing you have to do is to register your group. The Fire system can be accessed from here, or just follow the shortcut in the menu bar:

> https://fire.cs.chalmers.se:8031/cgi-bin/Fire

## 1.4 The original Shrdlu

More information about Shrdlu can be found here. First an overview of Shrdlu, including a long interaction example:

> http://hci.stanford.edu/winograd/shrdlu/

Then another overview:

> http://www.semaphorecorp.com/misc/shrdlu.html

Winograd's PhD thesis is online:

> http://dspace.mit.edu/handle/1721.1/7095

And finally another nice paper with flowcharts of the Shrdlu grammar (not exactly the way we do it nowadays...):

> http://dspace.mit.edu/handle/1721.1/5791

## 2 Shrdlite: What to do

The user interface is an interactive webpage that reads input from the user, does some AI stuff and presents the results in the browser. All AI stuff is done through an Ajax CGI script, which means that you do not have to read, understand or modify the HTML or Javascript files that constitute the frontend interface.

The main goal of the project is to create a command line program that reads a user command from the standard input stream, does some AI stuff, and prints the resulting robot commands to the standard output stream. There is really no need for any CGI or Ajax or HTML or web servers.

The command line program should consist of the following pipeline:

**Input** A user utterance is read from the standard input stream, all words separated by whitespace.

*Output*: a list of strings

**Parsing** The utterance is parsed into zero, one or several semantic trees.

*Note*: the grammar is already given to you.

*Output*: a list of trees

**Interpretation** Each tree is interpreted in the current world.

*Note*: some trees might not have an interpretation, and some trees might have several interpretations.

*Output*: a list of PDDL goals

**Ambiguity resolution** If the utterance is translated to several different PDDL goals, the ambiguity should be resolved in some way.

- the easiest (and worst) solution is to simply reject the utterance
- a better solution is to asks a clarification questions, in which case the program must exit by outputting the question, and then resuming when the user has answered

*Output*: a single PDDL goal (or failure, or a clarification question)

**Planning** The PDDL goal is solved in the current world, resulting in a plan of basic commands that can take the current world into the goal state.

*Output*: a list of basic commands

**Output** The final list of basic commands is printed to the standard output.

## 3 The world

Shrdlite uses a 2-dimensional blocks world, not 3d. The main reason for this is that it is much easier to visualize, but also because spatial reasoning becomes easier.

The world contains a floor, and a number of objects. The objects can stand on or in each other (if the form and size permits). The goal of the "game" is to move around the objects in any way the user likes, a bit like Minecraft or Lego. This can only be done by talking to a robot arm which can pick up and put down objects.

The world contains at least the following forms, colors and sizes:

**Forms** Bricks, planks, balls, pyramids, boxes and tables.
**Colors** Red, black, blue, green, yellow, white.
**Sizes** Large, small.

To simplify things, the floor has only room for at most $N$ objects at the same time. We call each of these places a *stack* or a *column*, and number them $1 \ldots N$.

This means that internally, the world can be represented as a list of stacks, where each stack is a list of objects, where each object has a unique identifier. But this is not the only possible representation, another is to use a set of PDDL predicates, {`(ontop a b)`, `(ontop b floor-n)`, ...}.

### 3.1 Physical laws

The world is ruled by physical laws that constrain the placement and movement of the objects:

- The floor can support any number of objects.
- All objects must be supported by something.
- The arm can only hold one object at the time.
- The arm can only pick up free objects.
- Objects are "in" boxes, but "on" other objects.
- Balls must be in boxes or on the floor, otherwise they roll away.
- Balls cannot support anything.
- Small objects cannot support large objects.
- Boxes cannot contain pyramids or planks of the same size.
- Boxes can only be supported by tables or planks of the same size, but large boxes can also be supported by large bricks.

There is an example world in the project template called `impossible.json`, which gives examples of bricks that break the physical laws in some way.

### 3.2 Spatial relations

The following spatial relations can be used to describe how objects are to be positioned:

- $x$ is **on top of** $y$ if it is directly on top – the same relation is called **inside** if $y$ is a box.
- $x$ is **above** $y$ if it is somewhere above.
- $x$ is **under** $y$ if it is somewhere below.
- $x$ is **beside** $y$ if they are in adjacent stacks.
- $x$ is **left of** $y$ if it is somewhere to the left.
- $x$ is **right of** $y$ if it is somewhere to the right.

### 3.3 The basic commands

A plan is a sequence of basic commands, and there are only two of them:

**pick n** Picks up the topmost object from column n
**drop n** Puts the object that is carried onto column n

## 4 Parsing the user utterances

User utterance interpretation comes in two steps. First the utterance has to be parsed into a tree. To ease the burden, the grammar and parser is already given to you, in four different formats.

**GF** To use the GF grammar you either call GF from the command line, or you use the GF bindings for Python, C or Haskell.

**Prolog** There is an equivalent Prolog DCG (see section 12.6 in the course book).

**Python** There is also a feature structure grammar for use in Python together with the NLTK library.

**Haskell** The same grammar is implemented using Haskell parser combinators.

**Java** There is no obvious way of writing CFGs in Java, but the Prolog grammar above can be used from inside Java via the GNUPrologJava library.

If you want to use any other language, you have to write some kind of interface to any of these grammar formats, or rewrite the grammar yourself. It's not that hard. See below for an overview description of the grammar.

All grammars are (or at least should be) equivalent to each other, .i.e., they should all recognize the same language, and return equivalent parse results for each given input sentence.

The format of the parse results depend heavily on the format, but I have tried to make them as similar as possible. The results are "stripped" syntax trees, i.e., they do not contain all details of the specific grammar, but instead only the important semantic parts.

## 4.1 Ambiguities

Note that the grammar is ambiguous, meaning that it can return several parse trees for a single input utterance. This is on purpose, since natural language often is ambiguous. Here is an example utterance that returns two trees:

*put the white ball in a box on the floor*

The two trees correspond to the meaning of the following two unambiguous utterances:

*put the white ball* ***that is*** *in a box on the floor*
*put the white ball in a box* ***that is*** *on the floor*

The parser module should return all trees and hope that the interpretation module can disambiguate the intended meaning in the current world. For example, if there is no ball inside any box, then the first meaning can be filtered out.

## 4.2 Description of the grammar

Here is a quick overview of how the grammar works. All utterances start with the nonterminal Command.

**Command** There are three kinds of commands:

- "take/grasp/pick up" an *Entity*
- "move/put/drop" "it" at a *Location*
- "move/put/drop" an *Entity* to a *Location*

All commands can be preceded by an optional "will/can/could you", and preceded or followed by an optional "please".

**Location** A location consists of a *Relation* followed by an *Entity*.

**Entity** An *Entity* is a *Quantifier* followed by an *Object*, and both must agree in grammatical number.

Optionally it can be followed by a relative clause, i.e., "that is/are" *Location*, or just *Location*.

A special *Entity* is "the floor"

**Object** An *Object* consists of a *Size* (optional), a *Color* (optional), and a *Form* (required). The order between the *Size* and the *Color* is not fixed.

The grammatical number of the *Object* is inherited from the number of the *Form*.

**Relation** a preposition such as "beside", "left of", "above", etc.

**Size** an adjective such as "small", "medium", "large", etc.

**Color** also an adjective, "black", "white", "blue", etc.

**Form** a noun such as "object", "ball", "box", etc. in singular – the plural variants are "objects", "balls", "boxes", etc.

# 5 Interpreting the parse trees

The parser can return several parse trees, and each of them should be interpreted in the current world. This interpretation will hopefully filter out some parse trees that simply cannot be understood, but it might also reveal that one parse tree is ambiguous in the current world.

Assume the ambiguous example utterance from above:

*put the white ball in a box on the floor*

Assuming the Prolog/Java DCG grammar, the parse trees look like follows:

```
(1) move(relative_entity(the, object(ball, -, white),
                         relative(inside, basic_entity(any, object(box, -, -)))),
         relative(ontop, floor))

(2) move(basic_entity(the, object(ball, -, white)),
         relative(inside, relative_entity(any, object(box, -, -),
                                          relative(ontop, floor))))
```

The goal of the interpreter is to convert a parse tree into a PDDL representation of the final goal of the command. This can be done in two steps:

- First you find out what objects that the different parts of the tree is referring to.

  In the example there are three different objects mentioned – the ball, a box and the floor. For the first tree to be valid, there should already exist a ball that is inside a box in the current world. For the second tree to be valid, there must already be a box that is on the floor.

- When you have identified the different objects, you formulate the goal by looking at the destination of the movement command.

- For the *take* and *put* commands, the procedure is similar but different...

Now, assume that the world consists of two balls (`c1` – white, `c2` – black), one table (`t3` – large), and three boxes (`b4` – large, `b5` – large, `b6` – small). `c1`, `t3` and `b5` is on the floor, while `b4` is on `t4`, `b6` is in `b5` and `c2` is in `b6`. This can be described in PDDL:

```
(ontop c1 floor-0)
(ontop b4 t3)  (ontop t3 floor-1)
(ontop c2 b6)  (ontop b6 b5)  (ontop b5 floor-3)
```

This assumes that the objects occupy columns 0, 1 and 3 on the floor. (This is the world represented in the file `examples/small.json`).

Note that this is not the only representation of the world! Another possibility is to explicitly state which column each object is in (as in `(column c1 0) (column c2 3)`).

Anyway, the interpretation might go like this:

- Tree (1) cannot have an interpretation in the current world, since there is no white ball in any box.
- Tree (2) does have an interpretation, where the white ball is `c1` and the only box on the floor is `b5`. The resulting goal is that `c1` should be inside of `b5`, or in PDDL terminology:

  ```
  (inside c1 b5)
  ```

## 5.1 Disjunctive goals

If, on the other hand, box `b4` had also been on the floor, then there would be two possible boxes that matched. The resulting PDDL goal would then become:

```
(or (inside c1 b4) (inside c2 b5))
```

Note that this is not standard STRIPS. An alternative is to say that a goal is a set of "basic goals", and then to call the planner with each of the basic goals in turn.

## 5.2 Quantifiers

One of the main problems with natural language interpretation is to handle quantification correctly. The Shrdlite grammar can recognize three different quantifiers, *the*, *any* and *all*, and they should all be interpreted differently.

The minimum requirement of your interpreter (and planner) is that it can handle the singular quantifiers (*the* and *any*). Furthermore, it is okay if you use the same interpretation for both *the* and *any*.

If you want your interpreter to be more interesting, it (and the planner) should also be able to handle the *all* quantifier, and distinguish between *the* and *any*. The main difference between these two is that *the* implicitly requires that there is only one object that fits the description. If there are more than one possible interpretation of a *the* quantified object, then the system can, e.g., ask a clarification question.

## 5.3 Clarification questions

If the user's utterance is ambiguous in the current world, i.e., if there are several parse trees that can be translated into goals, then it is helpful if the system can ask a clarification question.

To be able to interpret the answer to a clarification question, the system needs to know which state it was in before it asked the question. For this purpose, your Ajax script can remember the current dialogue state by sending it to the client and then receiving it again in the next dialogue turn.

# 6 Planning the actions

The planner should take a representation of the goal, and a representation of the current world. Preferrably both should be in PDDL format, to keep things more standardized.

Let's continue with our example utterance and world as before, giving the same goal:

```
(inside c1 b5)
```

Now, the planner should take the world representation and the goal and return a sequence of basic *pick* and *drop* actions. Here is a possible solution:

```
(pick 3)  (drop 4)
(pick 3)  (drop 2)
(pick 0)  (drop 3)
```

I.e., first it moves ball `c2` to the floor, then box `b6` to the floor, and finally it moves ball `c1` inside box `b5`.

Note that to be able to complete this, the planner also needs a representation of the background knowledge (e.g., which kinds of objects can be on top of, or inside, which objects).

## 6.1 PDDL examples

There are some PDDL examples for you to check out and play around with, here:

http://www.cse.chalmers.se/edu/course/TIN172/aips2000-testdata/

That directory contains 5 different domains, and in total 2217 PDDL files. You can download everything as a 3.3 MB zip file:

http://www.cse.chalmers.se/edu/course/TIN172/aips2000-testdata.zip

The PDDL examples are all taken from IPC00, the 2nd International Planning Competition which took place in the year 2000:

http://ipc00.icaps-conference.org/

## 6.2 The plan and the GUI

The Javscript GUI wants the final plan as a list of strings, where each string is either a basic action (e.g., "pick 4" or "drop 5"), or a system utterance that will be printed as part of the dialogue. Like this:

```
["First I move the black ball",
 "pick 3", "drop 4",
 "Then I move the small box",
 "pick 3", "drop 2",
 "Finally I put the white ball in the yellow box",
 "pick 0", "drop 3"]
```

# 7 Requirements and assessment

To be acceptable, your program must fulfill at least the following:

- It must obey all requirements of the world, in particular the physical laws and the spatial relations.
- It must try to disambiguate ambiguous utterances but interpreting each parse tree in the current world. If there are several possible meaningful parse trees, it must at least fail by saying that the utterance is ambiguous.
- It must translate a meaningful parse tree into a goal, into PDDL or a similar representation.
- It does not have to handle the plural *all* quantifier, and it may use the same meaning for *the* as the singular *any*.
- It must be able to plan a command sequence that solves a goal in a small world (such as the one described above), still obeying the physical laws.

## 7.1 Extensions

To get a higher grade, you must extend the program in some ways, such as:

- To be able to handle all quantifiers in a sensible manner.
- To ask the user clarification questions when the utterance is ambiguous, e.g. "do you mean the large red pyramid or the small green?"
- To make the planner handle large worlds and complicated goals.
- To make the planner describe what it is doing, in a way that is understandable to humans. One important part of this will be to know how to describe the different objects in concise way. (E.g., if there is only one ball, then you don't have to say that it is white.)
- To add new linguistic structures to the grammar, such as:
  - user questions (e.g., "where is the white ball?")
  - complex commands (e.g., "stack up all red bricks")
  - anaphoric references (e.g., "put the red brick under the green one")
- Or something else, but talk to your supervisor first!

# 8 Writing the report

Your report should describe how you have solved the different tasks, especially interpretation and planning. You should give pseudo-code snippets when that is necessary, or mathematical formulae.

The report should also contain some examples, both good and bad. I.e., at least some example where your program gives a good solution, and at least some where the solution is not the best one (e.g., where the suggested plan is too complicated or where your interpretation goes wrong in some way).

Also, don't forget that each group member should include a 1-page description of her/his own contributions to the final project. These personal descriptions should be included as an appendix to the report.

## 8.1 Formatting guidelines

Your report should adhere to the official style guidelines for this year's EACL conference. There is a zip archive that contains style files for both LaTeX and Microsoft Word, and a PDF document that describes the guidelines in detail:

http://www.eacl2014.org/files/eacl-2014-styles.zip

Everything that is written in the guidelines are to be followed, except the following details:

- You do not have to use the author-year citation style (Gusfield, 1997), but you can choose to use another style if you prefer, such as numeric [3] or alphanumeric [Gus97].
- You don't have to care about anonymization.
- The length of you report must not be longer than 8 A4 pages, plus up to two pages of references. Furthermore, you must include an appendix of one page per group member, where each member writes a summary of her/his contributions to the group work.
- Do not exceed the page limits!