Declan Urbaniak-Dornstauder
3716081
COMP 456
Project

ALEX-Shell – a *Lisp expert system shell*

github.com/decales/ALEX-Shell

I. Introduction

ALEX is a universal data-driven expert system[1] shell designed to solve problems that can be represented as state sequences with simple state transition logic. The system was developed in Common Lisp, hence its name, and accepts Lisp-based knowledge-base files as input to specify the problems to be solved. Given the flexibility of its knowledge base format and the powerful utility of the Lisp language, ALEX is capable of solving many state-based problem instances extending beyond the example used and described in project.

II. Guide

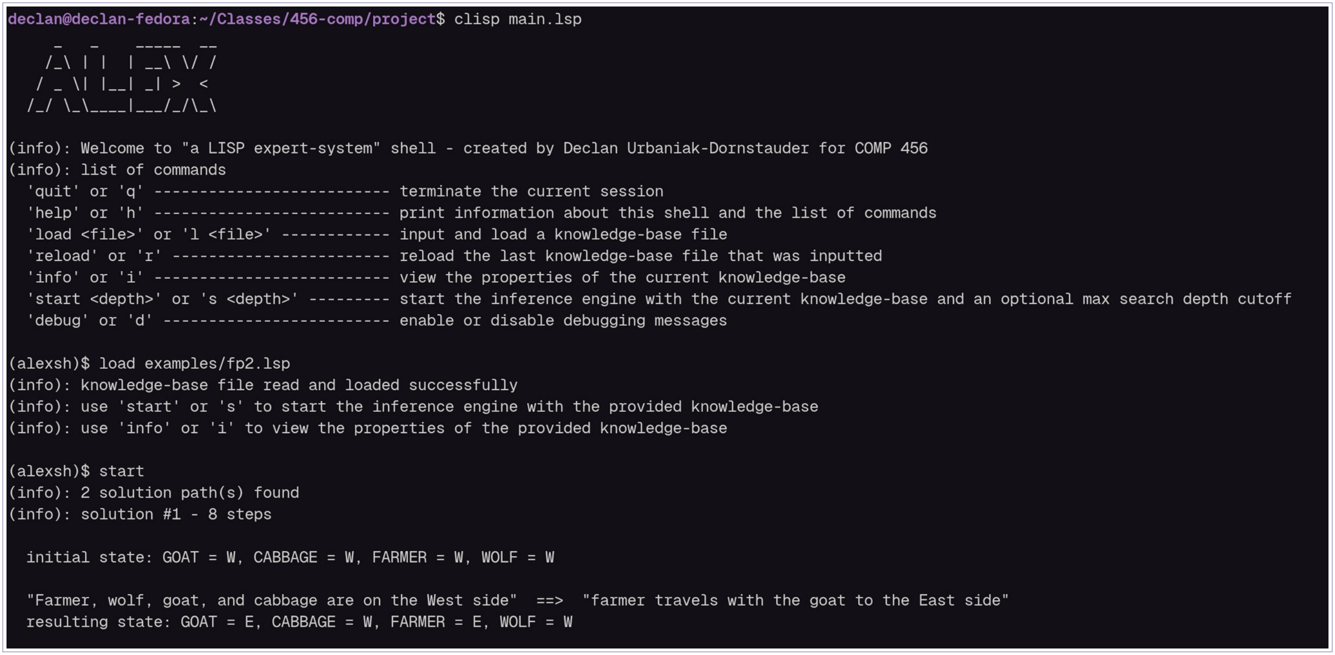
In its current state, the system can only be used with a Lisp compiler such as SBCL or CLISP. To start the shell, you may compile *main.lsp* directly or execute *(load "main.lsp")* in the REPL of your Lisp environment. Within the shell, you may execute any of the following commands, though most will only have an effect when a knowledge-base is loaded:

Command	Function
quit	-Terminate the current shell session
help	-Print the list of commands
load <file>	-Input and load a knowledge-base from a file
reload	-Reload the last knowledge base file that was inputted -Included for the convenience of quickly reloading a knowledge-base file after correcting the formatting issues detected by the parser
info	- Print the parameters of the current knowledge-base
Start <depth>	-Start the inference engine with the current knowledge-base and print the solutions, if any -A search depth may be optionally specified to prune solutions that exceed the given depth
debug	-Toggle the visibility of debug messages that are printed in the shell -While not particularly useful in the context of this project, this was left in as a feature to troubleshoot any problems that may arise with the input of custom knowledge bases

Before running the inference engine, a knowledge-base file must be loaded by inputting its relative path name after the *load* command token. The example knowledge-base files for this project can be found in the *examples* directory, and can be loaded by inputting *load examples/fp1.lsp* and *load*

1 en.wikipedia.org/wiki/Expert_system

examples/fp2.lsp. After parsing a file to verify that it contains the expected parameters and conforms to the proper format, the engine can be started by simply inputting *start*, or *start* followed by an integer to represent a search depth cutoff. Once the engine has terminated, the shell will print all solutions paths that were found for a problem, if any, and the steps between each path to reach the goal state.



```
declan@declan-fedora:~/Classes/456-comp/project$ clisp main.lsp
     _   _    -----  --
    /_\ | |  | __\ \/ /
   / _ \| |__| _| >  <
  /_/ \_\____|___/_/\_\

(info): Welcome to "a LISP expert-system" shell - created by Declan Urbaniak-Dornstauder for COMP 456
(info): list of commands
  'quit' or 'q' -------------------------- terminate the current session
  'help' or 'h' -------------------------- print information about this shell and the list of commands
  'load <file>' or 'l <file>' ------------ input and load a knowledge-base file
  'reload' or 'r' ------------------------ reload the last knowledge-base file that was inputted
  'info' or 'i' -------------------------- view the properties of the current knowledge-base
  'start <depth>' or 's <depth>' --------- start the inference engine with the current knowledge-base and an optional max search depth cutoff
  'debug' or 'd' ------------------------- enable or disable debugging messages

(alexsh)$ load examples/fp2.lsp
(info): knowledge-base file read and loaded successfully
(info): use 'start' or 's' to start the inference engine with the provided knowledge-base
(info): use 'info' or 'i' to view the properties of the provided knowledge-base

(alexsh)$ start
(info): 2 solution path(s) found
(info): solution #1 - 8 steps

  initial state: GOAT = W, CABBAGE = W, FARMER = W, WOLF = W

  "Farmer, wolf, goat, and cabbage are on the West side"  ==>  "farmer travels with the goat to the East side"
  resulting state: GOAT = E, CABBAGE = W, FARMER = E, WOLF = W
```

Example – initiating the shell, loading a knowledge-base, then starting the inference engine

III. Knowledge-base

The knowledge-base files used in the system should be customized to represent a given problem instance, but must conform to a particular format to be interpreted correctly. Firstly, the parameters of a knowledge-base file must be written using valid Lisp syntax, but not in a manner that should consider them to evaluate immediately and entirely as S-expressions. In this case, each parameter must be constructed as a raw list encompassing other raw sub-lists of atoms and expressions which may be eventually evaluated in the system. While I could attempt to elaborate further on what this means, I highly advise looking at the example knowledge base files – not only do they document the exact formatting requirements for each parameter, but the formatting requirements should be much easier to understand visually than my attempt to describe them here. As for the parameters of the knowledge base themselves, a knowledge-base file must define the four in the following order:

State definition – defines how a state should be represented in the context of the the problem instance

- may include as many name-value property pairs as necessary to represent the knowledge of a problem instance.
- formatted as ((<property-name> <property-data-type>) …) where each inner list represents a property and the outer list represents the state definition as a whole.

Initial state – defines the initial value for each property defined in the state definition

- formatted as ((<property-name> <property-value>) …) where an inner list represents and exists for each property declared in the state definition, and the outer list represents the initial state as a whole

Goal state – defines the goal or target values for each property defined in the state definition

- formatted as ((<property-name> <property-value>) …) where an inner list represents and exists for each property declared in the state definition, and the outer list represents the goal state as a whole

Production rules – defines the production condition-action pairs used in the recognize-act cycle

- may include as many productions as necessary to cover the state-space of a problem instance and transition appropriately between states.
- formatted as ((<condition-semantics> <condition> <action-semantics> <action>) …) where each inner lists represents a production and the outer list represents the productions as a whole.
- Production sub-lists must contain the four following sub-parameters:
    - condition semantics – a semantic description of the logic of the condition
    - condition – a boolean expression that must be satisfied to fire the action
    - action semantics – a semantic description of the the action its effect on the current state
    - action - a list of property name-value pairs that specify the new value for the given properties

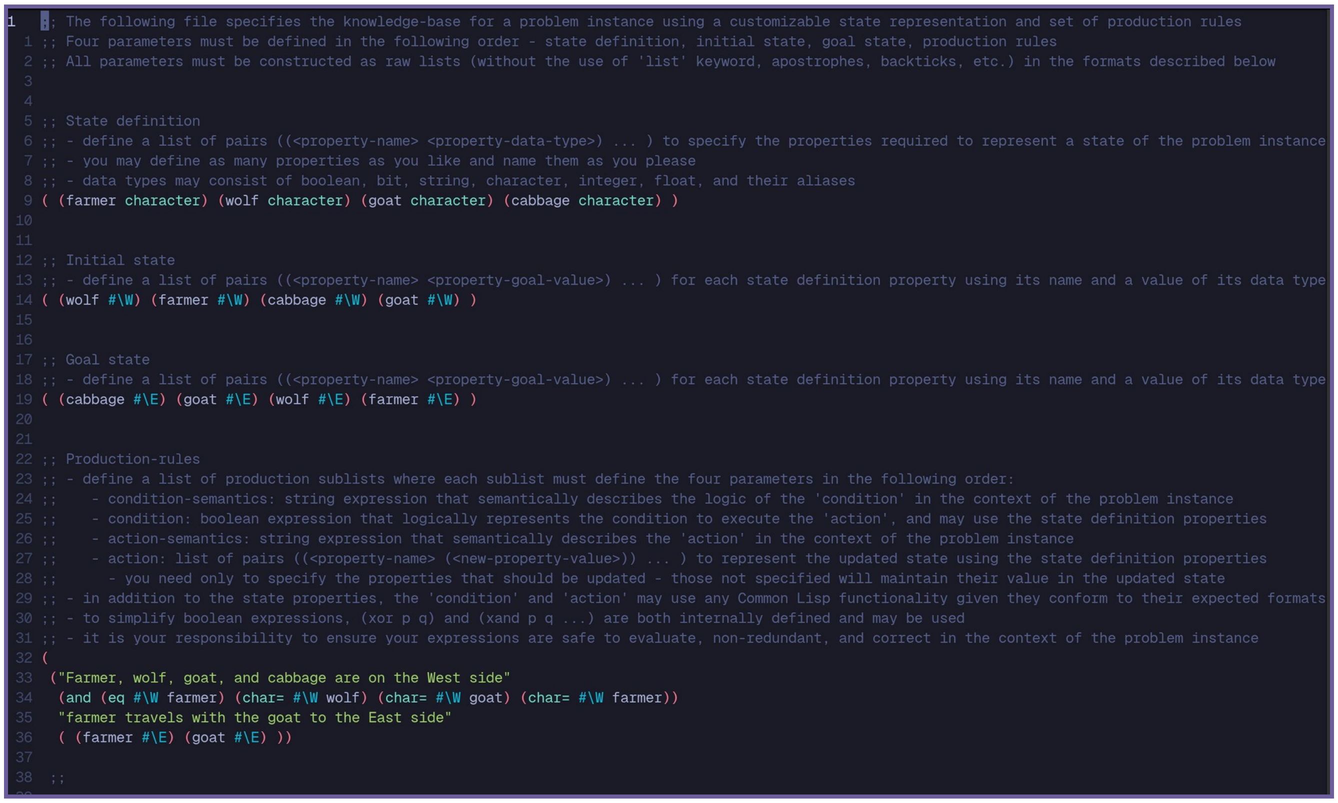
It should be noted that for all sub-parameters that use or specify values, such as the properties of the initial and goal states, the property updates of the actions, and the boolean values of the conditions, it is possible to use both direct values or expressions that evaluate to their appropriate data type. While this is naturally the way conditions should be constructed, it may be useful to represent other values as complex expressions in certain cases. So long as an expression evaluates to its expected type and its structure is correct, it can be as complex as necessary. Given this freedom, it may be possible to inject nearly any Lisp instruction into the system, so it is your responsibility to ensure the expressions in your knowledge-base are safe to evaluate.

In addition to providing detailed guidance on how to correctly construct a knowledge-base file, the two example files highlight the flexibility of how a problem instance can be represented. Both files describe the logic of the wolf, goat, and cabbage problem[2], though in slightly different approaches. The first example, *examples/fp1.lsp*, represents the state by defining boolean properties for all four entities of the problem, where the value of a property represents the side of the river an entity is on in a given state. Because properties are represented as booleans, the actions of their productions rules simply update the state by flipping the values of the properties to show that an entity has crossed the river. The second file, *examples/fp2.lsp*, follows the exact same logic as the first, but represents the positions of the entity properties as characters, 'E' or 'W'. Given this, the actions of the production rules assign new character values to the properties to update the state. While this variation can be dismissed as a redundant version of the previous that only doubles the number of production rules that are required, it can be argued that it simplifies the logic of the production conditions and is semantically clearer in describing how the states transition with each step of a solution. Regardless it still serves to demonstrate the capabilities of the system, and offers an additional perspective on how problems can be represented.

Regarding the example productions themselves, it should be noted that they are designed as condition-action pairs that implicitly enforce the broad constraints of the problem. Neither example knowledge-base file contains a single universal state check to prevent the system from entering invalid states, but rather, each production contains a specific condition with logic relevant to the action and its effect on the state (a pre-condition for the action). This being said, there is no single, appropriate way to design

2 en.wikipedia.org/wiki/Wolf%2C_goat_and_cabbage_problem

the productions of a problem – the productions should be designed with the context of a problem in mind, and can divert from the traditional condition-action pair style common of expert-systems when necessary.



```
;; The following file specifies the knowledge-base for a problem instance using a customizable state representation and set of production rules
;; Four parameters must be defined in the following order - state definition, initial state, goal state, production rules
;; All parameters must be constructed as raw lists (without the use of 'list' keyword, apostrophes, backticks, etc.) in the formats described below


;; State definition
;; - define a list of pairs ((<property-name> <property-data-type>) ... ) to specify the properties required to represent a state of the problem instance
;; - you may define as many properties as you like and name them as you please
;; - data types may consist of boolean, bit, string, character, integer, float, and their aliases
( (farmer character) (wolf character) (goat character) (cabbage character) )


;; Initial state
;; - define a list of pairs ((<property-name> <property-goal-value>) ... ) for each state definition property using its name and a value of its data type
( (wolf #\W) (farmer #\W) (cabbage #\W) (goat #\W) )


;; Goal state
;; - define a list of pairs ((<property-name> <property-goal-value>) ... ) for each state definition property using its name and a value of its data type
( (cabbage #\E) (goat #\E) (wolf #\E) (farmer #\E) )


;; Production-rules
;; - define a list of production sublists where each sublist must define the four parameters in the following order:
;;    - condition-semantics: string expression that semantically describes the logic of the 'condition' in the context of the problem instance
;;    - condition: boolean expression that logically represents the condition to execute the 'action', and may use the state definition properties
;;    - action-semantics: string expression that semantically describes the 'action' in the context of the problem instance
;;    - action: list of pairs ((<property-name> (<new-property-value>)) ... ) to represent the updated state using the state definition properties
;;      - you need only to specify the properties that should be updated - those not specified will maintain their value in the updated state
;; - in addition to the state properties, the 'condition' and 'action' may use any Common Lisp functionality given they conform to their expected formats
;; - to simplify boolean expressions, (xor p q) and (xand p q ...) are both internally defined and may be used
;; - it is your responsibility to ensure your expressions are safe to evaluate, non-redundant, and correct in the context of the problem instance
(
 ("Farmer, wolf, goat, and cabbage are on the West side"
  (and (eq #\W farmer) (char= #\W wolf) (char= #\W goat) (char= #\W farmer))
  "farmer travels with the goat to the East side"
  ( (farmer #\E) (goat #\E) ))

 ;;
```

Example – contents of fp2.lsp

IV. Structure and Design

The system is implemented solely in Common Lisp without the use of a particular design pattern, and is instead organized into various components based on their function. The entire project structure consists of the following:

- *examples/*
  - *fp1.lsp* – boolean-based farmer problem example knowledge-base file
  - *fp2.lsp* – character-based farmer problem example knowledge-base file
- *src/*
  - *engine.lsp* – inference engine to drive the system's problem solving capabilities
  - *input.lsp* – component to receive and read knowledge-base files
  - *parse.lsp* – component to parse the contents of knowledge-base files
  - *shell.lsp* – interface component containing REPL to interact with users
- *main.lsp* – file that loads the components of the system, and the entry-point of its execution

It is likely unnecessary to go into great detail describing the shell, the input, and the parsing components of the system, as these are relatively self-explanatory and simply support the system's primary function. It should be mentioned that the parser is fairly thorough, performing a strict check of an inputted knowledge-base file's validity and providing clear feedback to describe which formatting issues have occurred and where. While maybe ever so slightly overkill for the project specifically, this was necessary to ensure that custom knowledge bases could be interpreted correctly (but who am I kidding, I know nobody else is going to use the system, but I still felt oddly compelled to do this anyway).

At the heart of the system's problem-solving capabilities is the inference engine. Simply put, this component performs a state-space search of the problem and returns all solution paths commencing from the initial state and ending at the goal state, both specified in the knowledge-base. This is accomplished through a depth-first search, where the current state is patterned matched to the conditions specified in the production rules to transition to a new state using the updated properties of the production actions (a process formally known as the recognize-act cycle).

Though the system has both forward-chaining and backward-chaining characteristics, it is ultimately considered to be forward-chaining, or data driven. From one perspective, and despite the fact that the system is stated to begin at an initial state and determine the solutions path towards a goal state, the 'initial' and 'goal' states are ambiguous and interchangeable in the system. This is particular evident in the example problem, where the state transition logic is equally applicable to the case where the initial and goal states are inverted. In this sense, each valid but non-solution state can be considered as a sub-goal of the primary goal, suggesting that the system is at least goal-driven in intent. Mechanically, however the way the system expands the state-space aligns with how a forward-chaining chaining system infers new knowledge by matching working knowledge with the conditions of the production-rules. Furthermore, the inference system performs an exhaustive search of the state-space, something characteristic of data-driven systems. For this reason, it is possible to specify a search depth-cutoff, as a problem could many solution paths and it may be worthwhile to prune those that are sub-optimal. In function alone, this is enough to categorize the system as forward-chaining, though realistically, it is reasonable to consider the system as a hybrid of the two strategies.

V. Conclusion

ALEX is a simple demonstration of how expert systems can effectively model and solve state-based problems, something something, the end.