Implementation of Logic Production System (LPS) in JavaScript

Author: Yong Shan Xian, Sam

Supervisor: Dr. Fariba Sadri

Submitted in partial fulfillment of the requirements for the MSc degree in Computer Science of Imperial College London

September 2018
Abstract

Artificial intelligence (AI) pursued the intellectual capacity of humans for machines to learn and reason about the real world. In logic programming (LP), inference rules using backward reasoning, such as in Prolog, are used to explain certain observations and queries. On the other hand, production systems uses forward reasoning as inference in production rules to react when its conditions are met. Either method of inference on its own is insufficient to fully express complex decision making and reasoning processes.

Logic Production System (LPS), developed by Professor Robert Kowalski and Dr. Fariba Sadri, unifies both forward and backward reasoning into one single framework to closely model human cognitive processes, allowing the construction of agents in LPS that can learn through observations, reason about its worldview and react according to its goals and beliefs.

This report presents the JavaScript (JS) implementation of an LPS interpreter that leverages on the features and portability provided by JS in multiple contexts and to provide an opportunity to unify multiple programming paradigms into a single system. With JS becoming the defacto language of the internet, a LPS interpreter in JS can make LP and logic-based AI more accessible to programmers.

Furthermore, using JS opened the possibility of building cross-platform creative programs based on lps.js. This is demonstrated through the implementation of LPS Studio, a desktop interactive storytelling tool implemented using Electron framework for LPS program visualisation, as an extension to this project.
Acknowledgments

Firstly, I would like to thank Dr. Fariba Sadri for the opportunity to work on the subject. This project would not have been possible without her patience, guidance and support in challenging and pushing the boundaries of what is possible.

I would also like to thank Professor Robert Kowalski, Dr. Miguel Calejo and Dr. Jacinto Dávila for their thorough support, feedback and encouragement throughout this project.

Finally, I am very grateful for the never-ceasing support and encouragement from my family, fiancée Georgina, friends, and professors & teaching staff at Imperial College London, National University of Singapore & Ngee Ann Polytechnic who have taught me and walked with me. These are the people who have made it possible for me to go far and beyond to chase my dreams.

Through this project, I have learned many new concepts and technologies, many of which I am confident will come into good use later in my career and journey in computer science. Being able to put what I have learned into practical implementation brings great joy to the creative part of me. For that, I am filled with gratitude for everyone who have been part of this project, one way or another.
“...even if Logic and LP might be only half awake today, they can at worst be only sleeping, to come back with renewed and more lasting vigour in the near future.”

— Robert Kowalski [1]
# Contents

1 Introduction .......................... 1
   1.1 Aims ................................ 2
   1.2 Requirements ........................ 3
   1.3 Report Structure ..................... 4

2 Background ............................ 5
   2.1 Production .......................... 5
   2.2 Reactive Rules ....................... 5
   2.3 Logic Programs ....................... 6
       2.3.1 Horn Clauses ...................... 6
       2.3.2 Normal Clauses .................... 6
   2.4 SLD Resolution ........................ 6
       2.4.1 SLD with Negation as Failure .... 8
   2.5 LPS Vocabulary ....................... 9
   2.6 LPS Model-Theoretic Semantics ......... 9
       2.6.1 Notation .......................... 10
       2.6.2 Event Theory ...................... 10
   2.7 LPS Operational Semantics ............. 11
       2.7.1 Rule Pre-Processing ............... 11
       2.7.2 Rule Antecedent Resolution ......... 12
       2.7.3 Rule Implicit Temporal Constraint ... 12
       2.7.4 Goal States ....................... 13
       2.7.5 Operational Semantics Cycles ....... 13
       2.7.6 Incompleteness of Operational Semantics ... 15
       2.7.7 Candidate Actions Selection Strategies .... 16
       2.7.8 Operational Semantics Example .... 16
   2.8 Rules Antecedent and Firing Issues ... 19
       2.8.1 Fluents in Rule Antecedent and Refraction ... 19
       2.8.2 Negation in Antecedent .......... 21
   2.9 Multi-Agent Systems in LPS ............ 21
       2.9.1 Introduction of MAS in LPS .......... 21
       2.9.2 Network Architecture Choices ........ 22
       2.9.3 Agent and Knowledge Representation .... 23
   2.10 JS Concurrency Model ............... 23
       2.10.1 Event Loop and Queue .............. 23
       2.10.2 JS Concurrency Example ............. 24
       2.10.3 Callbacks and Promises ......... 25
   2.11 Electron Architecture ................ 27
# Implementation

3.1 Implementation Considerations ............................................. 29
3.2 LPS Syntax Introduction .................................................. 29
3.3 Parser Implementation ................................................... 32
  3.3.1 Lexer ................................................................. 32
  3.3.2 Parser .............................................................. 33
  3.3.3 Syntax Error Reporting .............................................. 34
  3.3.4 Static Syntactical Warnings ......................................... 35
3.4 Unification ................................................................. 35
  3.4.1 Unification by Recursive Descent ................................... 35
  3.4.2 Literal Tree Maps (LTM)s .......................................... 36
3.5 LPS Programs ............................................................... 40
  3.5.1 LPS Program Software Quality Considerations .................. 40
  3.5.2 Declarations ....................................................... 41
  3.5.3 Consult and Augmentation .......................................... 42
  3.5.4 Execution Modes .................................................. 43
3.6 Real-Time Observations .................................................. 44
3.7 JS Observer Events ....................................................... 46
3.8 JS Execution from LPS .................................................... 47
  3.8.1 API Design .......................................................... 47
  3.8.2 Adding JS-based predicates from JS ............................... 50
  3.8.3 Loading JS-based predicates from LPS ............................ 50
  3.8.4 Asynchrony and JS-based Predicates ............................... 51
  3.8.5 Implementation Considerations .................................... 51
3.9 Operational Semantics Implementation ................................ 51
  3.9.1 Rule Consequent Resolution using Goal Trees .................... 52
  3.9.2 Rule Consequent Resolution using Goal Directed Acyclic Graphs 53
  3.9.3 DAGs and Refractions .............................................. 54
  3.9.4 Non-deterministic resolution and Goal DAGs ...................... 55
  3.9.5 Sorting Terms by Deadline ........................................ 55
  3.9.6 Candidate Actions Selection Strategy .............................. 57
  3.9.7 Dealing with Refractions in Rules ................................ 58
3.10 Multi-Platform Targeting ............................................... 58
  3.10.1 Setting up Transpilation .......................................... 59
3.11 Testing API .............................................................. 59
  3.11.1 Testing Introduction .............................................. 59
  3.11.2 Test Environment Simulation ...................................... 61
  3.11.3 Special Comparator Terms ........................................ 61
  3.11.4 Test Specification Generator ...................................... 62
3.12 Multi-Agent Programming Support ................................... 62
  3.12.1 Consult with Identifier .......................................... 62
  3.12.2 Peer-to-peer Module .............................................. 64
3.13 Demonstration Website .................................................. 65
3.14 LPS Interactive Storytelling: LPS Studio .............................. 66
  3.14.1 LPS Visualisation ................................................ 66
  3.14.2 LPS Studio Features ............................................. 67
  3.14.3 User Controlled Agent in MAS Context ............................ 68
3.15 Atom Text Editor Language Package .................................. 69
# Table of Contents

## 4 Results and Evaluation

4.1 Deliverables .......................... 71
   4.1.1 Publication to npm Repository .... 71
   4.1.2 Distribution of LPS Studio .......... 71

4.2 Evaluation .......................... 72
   4.2.1 Test Suite ......................... 72
   4.2.2 Analysis of Test Programs .......... 72
   4.2.3 Tic-Tac-Toe in LPS Studio .......... 75
   4.2.4 Limitations ....................... 78

## 5 Conclusion

5.1 Challenges Faced ................... 81
   5.1.1 Debugging ......................... 81
   5.1.2 Resources on Subject ............. 82
   5.1.3 Resources and open source software 82

5.2 Future Work ....................... 82
   5.2.1 LPS to JS Compiler Implementation 82
   5.2.2 Performance Improvements .......... 83
   5.2.3 Implementation of cut and fail ........ 83
   5.2.4 Hibernation and Restoration ...... 84
   5.2.5 DOM Access and Manipulation Library 84
   5.2.6 LPS Studio: Extensions ............ 84

## A Ethics Checklist

A.1 Ethics Checklist ................... A-1

## B Ethical & Professional Considerations

B.1 Ethical & Professional Considerations B-1

## C Example LPS Programs

C.1 rain.lps ............................. C-1
C.2 fire-simple.lps ...................... C-1
C.3 fire-recurrent.lps ................. C-1
C.4 binary-search.lps ................. C-2
C.5 binary-search2.lps ............... C-3
C.6 guard.lps ........................... C-4
C.7 bubbleSort.lps ..................... C-5
C.8 map-colouring.lps ................. C-6
C.9 mark.lps ............................ C-7
C.10 mark-hiccup.lps ................. C-8
C.11 trash.lps .......................... C-9
C.12 quickSort.lps ..................... C-9
C.13 dining.lps ........................ C-10
C.14 rock-paper-scissors-minimal.lps  C-11
C.15 Fire Station - LPS Studio Example . C-12
   C.15.1 fire-example.lps ............... C-12
   C.15.2 fire-example-studio.lps ........ C-14
C.16 Bubble Sort - LPS Studio Example . C-17
   C.16.1 bubblesort-studio-main.lps ..... C-17
   C.16.2 bubblesort-studio-sorter.lps ... C-18
   C.16.3 bubblesort-studio.lps .......... C-18
## Table of Contents

### C.17 Lending and Borrowing MAS - Single Program
- C.17.1 main.lps .................................................. C-20
- C.17.2 agent.lps .................................................. C-21

### C.18 Lending and Borrowing MAS - P2P-based Programs
- C.18.1 bob.lps .................................................. C-22
- C.18.2 alice.lps .................................................. C-22

### D lps.js Built-in Predicates Reference
- D.1 Declarations ................................................. D-1
- D.2 Meta .......................................................... D-3
- D.3 Comparators .................................................. D-3
- D.4 Math and Arithmetic ........................................ D-4
- D.5 Lists .......................................................... D-7
- D.6 Types .......................................................... D-7

### E lps.js User Guide
- E.1 Usage as CLI Application ................................. E-1
- E.2 Usage as Library in Node.js Runtime ................. E-1
- E.3 Usage as Library in Browser Runtime ................ E-2

### F P2P Module Reference .............................................. F-1

### G Test API
- G.1 Introduction ............................................... G-1
  - G.1.1 Running Testing Tool ................................. G-1
  - G.1.2 Running Tests from Code ............................ G-2
- G.2 Testing Predicates ......................................... G-2
  - G.2.1 expect/1 ............................................... G-2
  - G.2.2 expect/3 ............................................... G-3
  - G.2.3 expect/4 ............................................... G-4
  - G.2.4 expect_num_of/3 ....................................... G-5
  - G.2.5 expect_num_of/4 ....................................... G-6
  - G.2.6 expect_num_cycles/1 ............................... G-7
  - G.2.7 Special Comparison Terms ......................... G-8
  - G.2.8 observe/2 and observe/3 ........................... G-9

### H LPS Studio User Guide
- H.1 Introduction ............................................... H-1
- H.2 Defining Canvas Objects ................................. H-1
  - H.2.1 Loading Images ...................................... H-2
  - H.2.2 Canvas Object Properties .......................... H-2
- H.3 Updating Canvas Object Properties .................. H-5
- H.4 Animating Canvas Objects ............................... H-6
- H.5 Observed Events ......................................... H-6
Abbreviations

**AJAX**  Asynchronous JavaScript and XML.
**AOP**  Agent-Oriented Programming.
**API**  Application Programming Interface.
**BNF**  Backus-Naur form.
**CLI**  Command Line Interface.
**CPU**  Central Processing Unit.
**CSS**  Cascading Style Sheets.
**DAG**  Directed Acyclic Graph.
**DOM**  Document Object Model.
**FIFO**  First In First Out.
**FILO**  First In Last Out.
**FOL**  first-order logic.
**GPU**  Graphics Processing Unit.
**GUI**  graphical user interface.
**HTML**  Hyper Text Markup Language.
**IPC**  Inter-Process Communication.
**JS**  JavaScript.
**JSON**  JavaScript Object Notation.
**LP**  logic programming.
**LPS**  Logic Production System.
**LTM**  Literal Tree Map.
**MAS**  Multi-Agent System.
Abbreviations

**npm**  Node.js package manager.

**OS**  Operational Semantics.

**OSS**  open source software.

**P2P**  Peer-to-Peer.

**SLD resolution**  Selective Linear Definite Clause Resolution.

**SLDNF**  SLD with Negation as Failure.

**TCP**  Transmission Control Protocol.

**TR**  Teleo-Reactive.

**UI**  user interface.

**URD**  Unification by Recursive Descent.
Chapter 1

Introduction

Logic Production System (LPS) is a declarative logic-based programming paradigm that embeds the notion of forward-reasoning rules and backward-reasoning logic programming (LP) proposed by Kowalski and Sadri[2, 3, 4]. In their papers, they studied the two methods of knowledge representation – namely production rules and logic programs – and their relationship with each other to develop LPS as a single unifying framework[2].

As LPS describes a framework, it can be implemented in different programming languages such as Java\(^1\). More prominently, it has been previously implemented in XSB Prolog by Wei[5] and has been extended onto SWISH – a SWI-Prolog web application – by Wielmaker et al.[6]

JavaScript (JS) is a multi-paradigm client-side scripting language that was originally implemented for Netscape Navigator by Brendan Eich in May 1995, intended for web developers to introduce interactivity on web pages [7, 8]. Over the years, the JS specification went through many revisions and eventually a Ecma standard (ECMA-262 and ISO/IEC 16262) – called ECMAScript\(^2\) – was established to enable different implementation supporting the same specification.[9] However, it is still up to browsers and other implementers to determine how much their implementation would support these specifications.

In 2009, Node.js’s first release by Ryan Dahl marked the dawn of server-side JS programming [10]. Before Node.js, JS was written exclusively for client-side execution on the web browsers. With Node.js and other competing server-side JS implementations, developers who had frontend knowledge could also work on the backend implementation using the same programming language. This paved way to development roles that worked exclusively with JS programming. The support for JS development grew stronger with the introduction of Node.js package manager (npm)[11] as it created a developer-friendly ecosystem for reusable JS libraries to be published and distributed.

The reach of JS has expanded to the development of mobile and desktop applications. Several open source efforts, such as Electron by Github, Inc. and Ionic by Drifty Co., enabled JS developers to write a single set of source code in JS and to deploy them to multiple platforms and architectures. Some desktop application built using Electron included Skype, Atom code editor and Slack desktop client, and these applications can run on most common operating systems such as Microsoft Windows, macOS and Linux OSes[12]. Similarly, mobile app developers could use Ionic to

---

\(^1\)JLPS implemented by Alexandre Camus in 2014 [http://albex.github.io/JLps/] and extended with Bob the Simplebot by Matthieu Raymond

\(^2\)ECMAScript® is a registered trademark of Ecma International.
write a unified mobile application application that can be deployed onto several mobile OSes such as Android, Windows 10 and iOS[13].

With JS gaining popularity as a lightweight cross-platform programming language[14], an LPS implementation in JS can enable more programmers to access the advantages of a high-level logic-based production system that can run side by side with imperative programming paradigm so that LPS can be applied to solve a greater spectrum of computational problems. Figure 1.1 shows the trend in the percentage of questions asked on Stack Overflow³, with the tags for "javascript" and "node.js" in clear increasing trend in comparison with other established languages.

![Figure 1.1: Stack Overflow's question trends based on tag usage. The number of questions posted with with tags "java", "php" and "c++" showed a decline trend from 2014 onwards, while the number of questions posted with tags "javascript" and "node.js" showed increase over the same time period. Note that year 2009 was the introduction of Node.js](image)

1.1 Aims

The main objective of this project is to implement a LPS interpreter in JS – henceforth referred to as lps.js where distinction is needed. To achieve this objective, the following were required:

- Understanding the LPS vocabulary and semantics
- Designing a suitable syntax for representing LPS programs
- Implementing a syntax parser based on the syntax
- Proposing a suitable architecture for interpreter's implementation
- Implementing the LPS interpreter
- Preparing example LPS programs, usage tutorials and Application Programming Interface (API) references

³A well-known Question and Answer website for topics relating to software engineering and programming.
To demonstrate the capabilities of lps.js, the project was extended with the following additional implementations:

- A web application that can provide and execute LPS programs to demonstrate lps.js’s features
- An Electron-based graphical user interface (GUI) tool – named LPS Studio — to run LPS programs for visual simulation and interactive storytelling

The overall scope and dependency of the core JS implementation and dependent efforts in this project is visualised in Figure 1.2 below.

![Figure 1.2: The hierarchy of work done for the implementation of lps.js and its dependent subprojects. “lps.js” represents the core LPS parser and interpreter npm package and it is used by three main extension packages: lps-cli, LPS Studio and lps-demo-web-api. lps-cli is the npm package that provides Command Line Interface (CLI) tools to use lps.js in the command line context. LPS Studio is the visualisation GUI program. lps-demo-web-api is the backend program that supports the demonstration web app. lps-demo-web is the frontend app that interacts with lps-demo-web-api to provide the user interface (UI) for the demonstration web app.](image)

### 1.2 Requirements

There were several requirements put in place for the design and implementation of lps.js.

**Portable** To ensure that lps.js could be used on both server-side (i.e. on Node.js) and client-side (i.e. on common JS-enabled web browsers), it was imperative that lps.js was implemented in a standard version of JS specification common to both platforms. However, as elaborated later in the report, it is also possible to write the code in one language, then transpiling the source code for one or two of the platforms mentioned.

**Programmer-friendly** As much as possible, the implementation should be programmer-oriented. This would mean that the implementation should adopt convention over configuration paradigm: rather than providing many complicated ways for the programmer to configure lps.js for their use, lps.js should be designed with minimal choice points required for the LPS programmer to get started. Syntactic sugar, where appropriate, convenient and clear, should be added. Programmer-friendly error messages are also needed to help LPS programmers debug their code easily.

---

4For the duration of the project, the website was available at [https://lps.mauris.sg/](https://lps.mauris.sg/)
1.3 REPORT STRUCTURE

Verifiable Implementing lps.js alone was not sufficient. As programmers – users of lps.js – would heavily depend on lps.js to run LPS programs as a black box\(^5\), rigorous testing is required to ensure and verify that lps.js would work according the LPS specifications.

Extensible The implementation should allow programmers and third-parties to add or extend the features provided in the LPS runtime without having to modify the source code of lps.js. Optional features that are not required by the programmer should be switched off by default.

1.3 Report Structure

This report is structured in the following manner:

- **Chapter 2** establishes some of the background knowledge, including Kowalski and Sadri’s LPS framework, its semantics and key JS’s language features.

- **Chapter 3** introduces a dialect LPS syntax accepted by lps.js, describes the implementation of lps.js along with testing and verification strategies of lps.js. The chapter will also introduce the extension projects and features introduced by lps.js to meet requirements of implementing systems from other fields of research, such as Teleo-Reactive (TR) Programming and Multi-Agent System (MAS).

- **Chapter 4** details the experimentation results of lps.js, in particular to advances in the goal tree resolution and search strategies, and evaluation of its performance and applications.

- **Chapter 5** concludes the report, and recommends some future extension and research relevant to this project and area of research.

The report has been written with the assumption that the reader is familiar with graph theory, set theory, unification theory, propositional logic, predicate logic and first-order logic (FOL). As with Prolog’s naming convention, predicate names used in this report and in LPS would start with a lower case letter while variable names start either with an underscore or an upper case letter.

---

\(^5\)Without having to learn, inspect and understand the internal behaviour of the interpreter.
Chapter 2

Background

Key concepts and semantics required for the implementation of lps.js are described and discussed in this chapter.

2.1 Production

A production in production systems are in the form of[15, p. 1]:

\[
\text{if condition then actions}
\]  \hspace{1cm} (2.1)

Formally, McDermott and Forgy defines a production \( P_i \) to be:

\[
P_i \left[ C_1 \ldots C_n \rightarrow A_1 \ldots A_m \right]
\]

where \( C_1 \ldots C_n \) represents the conditions that are checked against the working memory and \( A_1 \ldots A_m \) represents the actions to be executed once the conditions are met. In [15], McDermott and Forgy discussed the different conflict resolution strategies needed to choose a single rule to fire when the conditions of more than one rule have been satisfied.

2.2 Reactive Rules

Like a production, reactive rules in LPS have the form: if antecedent then consequent. More formally, rules are expressed as as:

\[
\forall X [\text{antecedent} \rightarrow \exists Y \text{ consequent}]
\]  \hspace{1cm} (2.2)

where antecedent and consequent are conjunctions of state conditions expressed in FOL.[4, p. 2] A rule is said to have fired whenever its antecedent holds in the working memory (or database) and an instance of the consequent must also hold.

In addition to actions in productions, reactive rules in LPS can contain features such as composite actions that are expanded by logic programming. This allows composite events in the antecedent to recognised by the interpreter[4, p. 10] as seen in the Silly Dialogue / Turing and Mark Hiccup example programs in Appendix C. In the consequent, composite events allow the expression of alternative plans to satisfy the resolution of the consequent[4, p. 4], as used in Concurrent Towers and Goat River Crossing example programs.
2.3 Logic Programs

Logic programs are sets of Normal clauses. Clauses are FOL statements in the form: conclusion if conditions. Formally, logic program clauses are written as:

$$\forall X \left[ \text{Conclusion} \leftarrow \text{Conditions} \right]$$  \hspace{1cm} (2.3)

where conclusion is also known as the head of the clause and conditions is also known as the body of the clause. Horn and Normal clauses are defined in the following sections. While both clauses and rules have similar syntactic formulation, they have different semantics and methods for resolution which will be discussed in detail in Section 2.6. Variables of a clause are implicitly and universally quantified at the front of the clause[2, p. 4].

2.3.1 Horn Clauses

Definition 1 Horn clauses are clauses with at most one positive literal.[16] Both definite clauses and definite denials are Horn clauses.

Definition 2 Definite clauses are a subtype of Horn clauses with exactly one positive literal. For example, $$\neg b_1 \lor \cdots \lor \neg b_n \lor h$$ is a definite clause that can be written as $$h \leftarrow b_1 \land \cdots \land b_n$$.

Definition 3 Definite denials, or definite goals, have no positive literals. For example, $$\neg b_1 \lor \cdots \lor \neg b_n$$ is a denial that can be written as $$\leftarrow b_1 \land \cdots \land b_n$$.

Definition 4 Facts can be considered as Horn clauses without any negative literals. For example, $$h \leftarrow$$ is a fact that can be written simply as $$h$$.

2.3.2 Normal Clauses

Normal clauses extend Horn clauses by permitting atoms in the body or a rule or denial to be prefixed with a special operator "not" (to be read as "fail"). LPS uses normal clauses for logic program.

Definition 5 The computational meaning of the "not $$p$$" is (i) "not $$p$$" succeeds iff "$$p$$" fails finitely and (ii) "not $$p$$" fails iff "$$p$$" succeeds.

Definition 6 Normal clauses are extension of Definite clauses that allow literals that are prefixed with the not operator. For example: $$h \leftarrow b_1 \land \cdots \land b_n \land \neg b_{n+1} \land \cdots \land \neg b_m$$

Definition 7 Normal denials are extension of Definite denials that allow body literals that are prefixed with the not operator. For example: $$\leftarrow b_1 \land \cdots \land b_n \land \neg b_{n+1} \land \cdots \land \neg b_m$$

2.4 SLD Resolution

Selective Linear Definite Clause Resolution (SLD resolution) is a sound and refutation complete proof procedure.[17] Given a denial (goal) $$G_0$$ and a clausal theory $$Th$$ of definite clauses, a SLD resolution of $$G_0$$ from $$Th$$ is a (possibly infinite) sequence of denials:

$$G_0 \rightarrow C_0 \rightarrow G_1 \rightarrow \cdots \rightarrow G_{n-1} \rightarrow C_{n-1} \rightarrow G_n$$  \hspace{1cm} (2.4)

where $$G_{i+1}$$ is derived from $$G_i$$ by SLD resolution using a clause $$C_i$$ with variables appropriately renamed. Figure 2.1 shows in detail how resolution occurs at each derivation.
Chapter 2. Background

2.4. SLD RESOLUTION

Figure 2.1: In this example, the denial clause $G_i$ can be resolved with definite clause $C_i$ by unifying $\alpha_i$ and $\alpha'_i$. The unification produces a substitution $\theta$ that is applied to the remaining atoms of $G_i$ and body of $C_i$. All remaining atoms then form the next goal $G_{i+1}$.

The resolution succeeds when an empty goal clause is reached: this appropriately indicates that there is nothing else to prove. Consider the query $\exists Z proud(Z)$ and following knowledge base[18, p. 6]:

\[
\begin{align*}
proud(X) &\leftarrow parent(X, Y), newborn(Y) \\
parent(X, Y) &\leftarrow father(X, Y) \\
parent(X, Y) &\leftarrow mother(X, Y) \\
father(adam, mary) \\
newborn(mary)
\end{align*}
\]

By SLD resolution, at least one value for $Z$ was determined to exist, there being a substitution $\{Z \mapsto adam\}$ such that $proud(Z)$ holds as shown in Figure 2.2.

Figure 2.2: At $G_4$ the resolution reaches an empty denial proves that a value of $Z$ exists such that $proud(Z)$ holds in the knowledge base. The sequence of substitution answers what value $Z$ is, which in this case would be "adam".
However from $G_1$ to $G_2$, an alternative clause could have been picked to be used as $C_1$. More specifically, it could have been the clause $\text{parent}(X, Y) \leftarrow \text{mother}(X, Y)$. In such a case where a goal clause can unify with more than one clause, more than one SLD resolution can be computed. Alternative choices can be represented as a single tree, with each leaf node representing one SLD resolution path. The SLD tree for the proud parent running example is shown in Figure 2.3.

The SLD tree represents the search space for all possible derivations from a single given goal. Depth-first search is used to traverse the tree to return each refutation in order. When a leaf node with an empty goal is reached, the substitution of the completed refutation is returned as an answer.

For a finite SLD tree, the search strategy is complete and will terminate. However, it is also possible for a SLD derivation to be infinitely long and hence generating an infinite SLD tree will result in a traversal that will not terminate.[18, p. 7]

**Figure 2.3:** A branch is created at each choice point for different choices of clause or substitution used to resolve the goal. This diagram reflects the choice of clauses with parent$(X, Y)$ as the head literal in the running example.

### 2.4.1 SLD with Negation as Failure

SLD with Negation as Failure (SLDNF) is an extension of SLD resolution that uses Normal clauses to represent negation of atoms[19]. A derived subgoal $\leftarrow \neg p$ succeeds if its subproof fails and fails if its subproof succeeds. The subproof of $\leftarrow \neg p$ would be $\leftarrow p$.

SLDNF is based on closed world assumption (CWA) since its knowledge base is complete and the failure of a subgoal depends on its absence from the knowledge base[20].
If a negation’s subproof – e.g. the subgoal ”← not q(X)” – have variables in it, the proof procedure cannot instantiate those variables such that the subgoal would fail.

2.5 LPS Vocabulary

Predicate symbols in LPS are partitioned into disjointed sets as follows[4, pp. 11-12):

- **Fluent** predicates are time-varying conditions.
  - *Extensional* predicates represent facts in LPS that change over time. At time $t_i$, these predicates form the set $S_i$.
  - *Intensional* predicates are time-dependent predicates defined in the logic program whose states are determined by extensional predicates.

- **Event** predicates can be subdivided into the following:
  - *Simple* events represent all events that are either externally received or internally generated.
  - *Composite* events (or macro actions) are predicates that appear in the head of logic program definitions.

- **Auxiliary** predicates are facts that do not vary with time, such as those defined as facts clause in the logic program and provided by implementation.

- **Meta**-predicates consist of predicates that modify the behaviour of LPS runtime. Some examples include initiates/2, terminates/2 and updates/3. Fluents and events would occur as terms in the arguments to these predicates.

For brevity and clarity:

- Fluents: only extensional predicates
- Actions: internally generated simple events
- Observations: externally generated simple events, i.e. observed by LPS

2.6 LPS Model-Theoretic Semantics

The LPS framework is a state transition system that is defined by the tuple $\langle R, L, D \rangle$[4, pp. 2, 3, 15], where:

- $R$ represent the **goals** of an agent defined by the set of reactive rules
- $L$ represent the **beliefs** of an agent defined by a logic program of clauses and auxiliary\(^1\) facts.
- $D = D_{pre} \cup D_{post}$ being the domain theory consisting of
  - $D_{pre}$ the set of pre-conditions, i.e. constraints
  - $D_{post}$ the set of post-conditions, i.e. declaration of how extensional predicates are changed based on simple events

---

\(^1\)Also known as *timeless* facts because these facts stay true throughout the execution of a LPS program.
In contrast to modal temporal logics where facts are not timestamped, LPS timestamps all fluents and events. \( t_i \) is defined to be the real clock time of the discrete timestamp \( i \). If real clock time \( t_a \) is before \( t_b \) — i.e. \( t_a < t_b \) — then it will always be the case that their discrete timestamp in LPS would also be \( a < b \).

### 2.6.1 Notation

**Fluent Notation** To associate a fluent \( p \) with a timestamp \( i \), it is possible to write in terms of meta-predicate \( \text{holds}(p, i) \) or be denoted as \( p(i) \). In the case that a fluent is not a nullary predicate, such a non-nullary fluent \( q(X, \ldots) \) can be associated with a timestamp \( i \) by writing \( \text{holds}(q(X, \ldots), i) \) or \( q(X, \ldots, i) \).

**Event Notation** For the case of an event \( e \) to be associated with the time period from \( i_1 \) to \( i_2 \), the meta-predicate \( \text{happens}(e, i_1, i_2) \) is used to note the association as \( \text{happens}(e, i_1, i_2) \). Alternatively the notation \( e(i_1, i_2) \) can be used to refer to the same expression. If the event is not a nullary predicate, a non-nullary \( g(X, \ldots) \) can be associated with with a time period from \( i_1 \) to \( i_2 \) by writing \( \text{happens}(g(X, \ldots), i_1, i_2) \) or \( g(X, \ldots, i_1, i_2) \).

### 2.6.2 Event Theory

The current state for \( i \) is defined as the set \( S_i \) that contains all fluents that hold at discrete timestamp \( i \). When an event \( e \) is executed from time \( t_i \) to \( t_{i+1} \), and \( D_{\text{post}} \) determines that the executed event would make a fluent \( p \) true, it is said that \( e \) initiates \( p \) from time \( t_{i+1} \) onwards. Likewise when an event \( e \) is executed from time \( t_i \) to \( t_{i+1} \), and \( D_{\text{post}} \) determines that the executed event would make a fluent \( p \) false, it is said that \( e \) terminates \( p \) from time \( t_{i+1} \) onwards. The behaviour of \( D_{\text{post}} \) can be summarised as follows:

1. If a fluent held at time \( i \) was unaffected by \( D_{\text{post}} \) (i.e. not terminated), the fluent remains held at time \( i+1 \) and is found in the set \( S_{i+1} \).
2. If a fluent held at time \( i \) was affected by \( D_{\text{post}} \) (i.e. terminated), the fluent would no longer remain into time \( i+1 \) and would not be found in the set \( S_{i+1} \).
3. If a fluent was not held at time \( i \) and was affected by \( D_{\text{post}} \) (i.e. initiated), the fluent would hold from time \( i+1 \) and be found in the set \( S_{i+1} \).

![Figure 2.4](image.png)

**Figure 2.4:** Suppose event \( e \) initiates \( b \) and \( e \) terminates \( a \), the figure shows the transition from time \( i \) to \( i+1 \) as \( e \) was executed, regardless if internally or externally executed. The fluent \( c \) remains unaffected and would remain at time \( i+1 \) due to the frame axiom properties of LPS framework.

The set \( E_i = \text{ext}_i \cup \text{act}_i \) is defined to contain all events, whether internally generated (the set of \( \text{act}_i \)) or externally received (the set of \( \text{ext}_i \)), that were executed during the state transition from \( i \) to \( i+1 \). Each \( E_i \) would therefore, by \( D_{\text{post}} \), be associated with a set of initiated fluents and a set of terminated fluents.
State transitions in LPS can be specified by an event theory which can be entirely represented by the following two clauses\cite[pp. 4-5]{4}:

\[
\begin{align*}
\text{holds}(P, T) & \leftarrow \text{initiated}(P, T) \\
\text{holds}(P, T + 1) & \leftarrow \text{holds}(P, T) \land \neg \text{terminated}(P, T + 1)
\end{align*}
\]

In LPS, states are updated destructively using the earlier mentioned Event Theory \cite[pp. 1-6; 21, p. 2]{4}. The proofs were given in \cite{4}. The set of all states, events, auxiliary facts is also the Herbrand model \( M \) of the LPS program\cite[p. 7]{21}, i.e.

\[
M \equiv L_{aux} \cup S_0 \cup S_1 \cup E_1 \cup \cdots \cup S_i \cup E_i \cup \ldots
\]  

\[ (2.5) \]

### 2.7 LPS Operational Semantics

The Operational Semantics (OS) of the LPS framework can be described in the form of a cycle that may or may not terminate\cite[p. 15]{4}. The OS defined allows different implementations to support the same model-theoretic semantics.

Rules in LPS enable simple actions to be described with a temporal ordering. Two simple actions can either happen one after another or together in the same time cycle. To achieve this temporal ordering during resolution, several techniques were introduced\cite{4}.

#### 2.7.1 Rule Pre-Processing

Rule pre-processing is needed to simplify the OS when dealing with rules with composite events in the antecedent. Rule antecedents are expanded by backward reasoning in this pre-processing step before the LPS program starts\cite[p. 16]{4}. This allows recognition of a composite event’s occurrence in terms of its simple events. Consider the following example rule:

\[
bought(dinner, T_1, T_2) \rightarrow eat(T_2, T_3)
\]

and the following example clauses:

\[
bought(dinner, T_1, T_2) \leftarrow bought(steak, T_1, T_2)
\]

\[
bought(dinner, T_1, T_2) \leftarrow bought(burger, T_1, T_2), bought(fries, T_1, T_2)
\]

The rule indicates that if dinner was bought, the program will then execute the action \( eat \). The logic program defines that a dinner is considered bought if steak was bought or a set meal of burger and fries was bought. By preprocessing the rule, the following two rules are created to replace the original rule as equivalent:

\[
bought(steak, T_1, T_2) \rightarrow eat(T_2, T_3)
\]

\[
bought(burger, T_1, T_2), bought(fries, T_1, T_2) \rightarrow eat(T_2, T_3)
\]
Any unification and instantiation of variables that occur during the rule’s antecedent pre-processing must also occur in the other parts of the antecedent and consequent of the rule.

### 2.7.2 Rule Antecedent Resolution

The OS also maintains a set of reactive rules $R_i$ for each timestamp $i$, in addition to maintaining the current state $S_i$. Whenever a conjunct in a rule’s antecedent becomes resolved with a unification $\theta$, the remaining antecedent becomes instantiated with $\theta$ and a new rule is created using the partially instantiated antecedent. Such a rule may represent an instance of the original rule that may become resolved in the future. Consider the following example rule:

\[
\text{heatDetected}(X, T_1, T_2), \text{smokeDetected}(X, T_3, T_4), |T_1 - T_3| < 10 \rightarrow \text{soundAlarm}(X, T_5, T_6)
\]

Supposed from timestamp 4 to 5, $\text{heatDetected}(\text{hallway})$ was observed, an instance of the antecedent would be created and the following rule would be added to $R_4$:

\[
\text{smokeDetected}(\text{hallway}, T_3, T_4), |4 - T_3| < 10 \rightarrow \text{soundAlarm}(\text{hallway}, T_5, T_6)
\]

The program now listens out for future possible resolution of this new rule in addition to the existing rules. Suppose a conjunct in the antecedent does not hold, the instance rule will then be discarded instead.

In their paper, Kowalski and Sadri indicated that the generation of these rules by antecedent resolution allows recording of events without storing them explicitly and supports the destructive updating of facts in LPS[4, p. 16].

### 2.7.3 Rule Implicit Temporal Constraint

Consider an example rule as follows:

\[
\text{fire}(\text{Location}, T) \rightarrow \text{extinguishFire}(\text{Location}, T_1, T_2)
\]

Suppose $\text{fire}(\text{hallway})$ holds at time 3 and the fact $\text{extinguishFire}(\text{hallway})$ holds from time 1 to time 2. The rule, along with the two facts, would have been considered fired without violating the model-theoretic semantics since it is possible to put the facts together as an instance of the rule as such:

\[
\text{fire}(\text{hallway}, 3) \rightarrow \text{extinguishFire}(\text{hallway}, 1, 2)
\]

However, the implementation of the OS does not allow such goal satisfaction. More accurately, the OS must apply implicit temporal constraints on the consequent such that no temporal variables in the consequent can be less than or equal to the last temporal instantiation in the antecedent. As such, the example rule will be interpreted by the OS explicitly as such:

\[
\text{fire}(\text{Location}, T) \rightarrow \text{extinguishFire}(\text{Location}, T_1, T_2), T < T_1, T_1 \leq T_2
\]
This applies regardless if an action or fluent was used either in the antecedent, consequent or both. It is no longer possible to query for a fluent in the consequent from a state that belong to a time before the rule has fired. For example,

\[
\text{moveTo}(\text{NewLoc}, T1, T2) \rightarrow \text{location}(\text{OldLoc}, T1), \text{moveFromTo}(\text{OldLoc}, \text{NewLoc}, T3, T4)
\]

In this case, the fired rule will never have a resolved instance of its consequent, since the processing of the consequent as a goal tree at time \(T2\) no longer have access to the state at time \(T1\) due to the destructive updating of database in LPS.

### 2.7.4 Goal States

Apart from the current state \(S_i\) and current set of reactive rules \(R_i\), the OS also maintains a goal state \(G_i\) for each timestamp \(t_i\). This goal state \(G_i\) is a forest of goal trees, where the root of each tree is an instance of the consequent of a reactive rule whose antecedent became true. The goal state \(G_i\) is a conjunction of all the goal trees. The internal nodes of each tree represent the intermediate steps taken to resolve its root node. These intermediate steps uses forward reasoning as inference for resolutions. Branches created from a node to represent the decision points of resolving a literal in the clause. Goal trees are computed independently from each other. Their computation only depends on the current state \(S_i\) and events \(E_i\).

An empty leaf node in the tree represents a successful refutation (i.e. there is nothing else to prove) of the goal in the root node of that goal tree. A goal tree is evaluated to \(\text{true}\) iff there exists at least one leaf node in the tree where the leaf node contains the empty clause. The path from the root node to that empty clause leaf node therefore shows the intermediate steps taken to show that there is nothing else to prove. Conversely, a goal tree is resolved as \(\text{false}\) iff all leaf nodes of the tree failed. Eventually, all goal trees must resolve to \(\text{true}\) before the LPS program terminates to keep the model true[4, p. 15]. Consider the following reactive rule:

\[
\text{fire}(\text{Location}, T) \rightarrow \text{available}(\text{Firefighter}, T2), \text{deploy}(\text{Firefighter}, \text{Location}, T2, T3)
\]

Suppose an instance the antecedent \(\text{fire}(\text{kitchen}, 2)\) holds, a goal tree would be created with the root node to be:

\[
\text{available}(\text{Firefighter}, T2), \text{deploy}(\text{Firefighter}, \text{kitchen}, T2, T3)
\]

The evaluation and resolution of a goal tree would be later discussed in Step 2 of the OS cycle.

### 2.7.5 Operational Semantics Cycles

At the start of the LPS program \(\langle R, L, D \rangle\), set \(R_0 = R, \text{act}_1 = \emptyset, G_0 = \emptyset\). The initially statements would prepare the initial state \(S_0\). There are three main steps for each \(i^{th}\) iteration, where \(i > 0\):

0. Updating the current state \(S_i\)
1. Process antecedents of reactive rules \(R_i\)
2. Process goal state \(G_i\)
Step 0: Updating the current state

First, a set of concurrently executed events $E_i = ext_i \cup act_i$ is selected such that $D_{pre}$ holds in the FOL-perfect model of $S_{i-1} \cup L_{int} \cup L_{aux} \cup E_i$. $\forall i > 1$, where act$_i$ is a subset of the set of candidate actions cand$_i$. The selection of actions to be executed must not violate any integrity constraint and the choice of elements from cand$_i$ to be included in act$_i$ will be discussed later in the Section 2.7.7.

The Event Theory discussed in Section 2.6 would then be applied to create the state $S_i$ from $S_{i-1}$. Then, let $G_i = G_{i-1}$, $R_i = R_{i-1}$ and reset the set of candidate actions by letting cand$_i = \emptyset$. It is important to note that once an action has been selected and executed, it is not possible to undo or rollback the execution of that action later without further complicating the OS.

Step 1: Processing antecedents of reactive rules

For every rule in $R_i$, construct every parsing of the rule into the following form:

\[
early\text{Antecedents} \land other\text{Antecedents} \rightarrow consequent
\]

where earlyAntecedents is a conjunction of state conditions and simple events such that time parameters in earlyAntecedents can be unified with the current time $i$. It is possible for earlyAntecedents to be empty, where none of the conditions and events in the antecedents unify with the current time. The construction of earlyAntecedents must not make any temporal constraints in otherAntecedents false or constrain any of the time parameters in state conditions or simple events in otherAntecedents to be less than or equal to $i$ i.e. earlier than current time.

For each of these parsing and each ground instance of earlyAntecedents, i.e. earlyAntecedents$\theta$ that is true in the FOL-perfect model of $S_i \cup L_{int} \cup L_{aux} \cup E_i$, the following instance rule would be produced:

\[
other\text{Antecedents}$\theta$ \rightarrow consequent$\theta$
\]

The antecedent of the new instance rule is then further simplified by deleting any temporal constraints that has become true in the FOL-model of $L_{aux}$. If after simplification the antecedent becomes an empty conjunction (i.e. equivalent to being true), the instance rule's consequent would then be added as the root node of a new goal tree to $G_i$. Otherwise the instance rule would be added to $R_i$. The rationale behind the addition of the instance reactive rule to $R_i$ was discussed in Section 2.7.2[4, p. 17].

Step 2: Process goal trees

If there are no further processing (i.e. new steps) that can be performed for the current iteration, then the current iteration of the cycle terminates. Otherwise, any node $C$ of any tree in $G_i$ is chosen to perform one of the following steps 2.1, 2.2 or 2.3. [4, p. 18]

Step 2.1: Reduce a composite event Choose a composite event term $A$ in $C$, unify $A$ with the head of some clause in $L$ and update $G_i$ by adding the resolvent as a child node of $C$. There are no time parameter restrictions in this step.
Step 2.2: Reduce State Conditions and Simple Events  
Consider a parsing of $C$ of the form:

$$earlyConsequents \wedge otherConsequents$$

where $earlyConsequents$ is a conjunction of state conditions and simple events, and time parameters in $earlyConsequents$ can be unified with the current time. The construction of $earlyConsequents$ must not make any temporal constraints in $otherConsequents$ false or constrain any of the time parameters in state conditions or simple events in $otherConsequents$ to be less than or equal to $i$, i.e. earlier than current time.

Each of the ground $earlyConsequents_\theta$ would generate a resolvent $otherConsequents_\theta$. The resolvent $otherConsequents_\theta$ is then simplified by deleting temporal constraints that has become true in the FOL-model of $L_{aux}$. Then after simplification, update $G_i$ by adding the generated resolvent as a child node to $C$. If the child node added is an empty conjunction / goal clause, then the goal tree would also then be true.

Step 2.3: Select conjunction of simple actions for attempted execution  
Consider a parsing of $C$ as such:

$$actions \wedge otherConsequents$$

where $actions$ is a conjunction of only simple actions

$$\text{happens}(a_1, T, T + 1) \wedge \cdots \wedge \text{happens}(a_k, T, T + 1)$$

and all time parameters $T$ and $T + 1$ can be unified with $i$ and $i + 1$ respectively. The construction of $actions$ must not make any temporal constraints in $otherConsequents$ false or constrain any of the time parameters in state conditions or simple events in $otherConsequents$ to be less than or equal to $i$, i.e. earlier than current time.

Then, all the actions $a_1, \ldots, a_k$ are added to $cand_{i+1}$ so that it can chosen for execution at Step 0 of the next iteration. When the actions are executed in the next iteration, the clause in $C$ would then be resolved by Step 2.2 of the next iteration.

Goal Tree Failure  
When the root of a goal tree becomes $false$ by having all possible subgoals becoming $false$, in theoretical sense the OS would terminate with failure since the rule has been fired and expected at least one instance of its consequent to be true at some point in the future. That would cause the rule to evaluate to $false$ as well, i.e. $(true \rightarrow false) \equiv false$ [4, p. 19].

However, [4, p. 19] also suggests that some of the subgoals that failed in the current cycle may become $true$ in future cycles and hence it would be desirable to allow the OS to continue retrying.

2.7.6 Incompleteness of Operational Semantics  
The OS described as in Section 2.7.5 has been shown to be incomplete [21]. Specifically, the OS of LPS cannot perform the following:

• preventively make a rule $true$ by making its antecedent $false$: Actions are only selected for execution from the consequent.
2.7. LPS OPERATIONAL SEMANTICS

• preventively make a rule true by making its consequent true while its antecedent is false: Actions from consequent are only selected for execution from a rule that was fired, i.e. its antecedent became true.

2.7.7 Candidate Actions Selection Strategies

The OS cycle in Section 2.7.5 also did not specify how candidate actions are selected for execution from the set \( cand_i \), specifically in Step 0 of the cycle [22]. Kowalski & Sadri [22, p. 9] and Wei [5, pp. 35-37] suggested that it was possible to implement different selection strategies that would maximise the number of goal trees becoming resolved and interleave actions from different goal trees without violating constraints in \( D_{pre} \).

There were several selection strategies used in other LPS implementations [4, p. 20-21]:

1. First In Last Out (FILO) or Stack Ordering: Priority is given to newest leaf nodes in goal trees
2. First In First Out (FIFO) or Queue Ordering: Priority is given to the oldest leaf nodes in goal trees.
3. Action Deadline Priority: Priority is given to actions of the earliest deadline.
4. Programmer Max Cycle Priority: Programmer specifies a maximum number of cycles a goal clause can wait before it needs to be processed. Priority is given to nodes that have the least number of cycles remaining to that deadline.
5. Programmer Max Retry: Programmer specifies a maximum number of cycles that a simple action or state condition in a goal clause of a goal tree can be retried for. If the number of retries has been reached, the OS needs to try another alternative way of resolving the goal tree.

It is therefore assumed that LPS programs can select any number of actions to execute in the same cycle (hence interleaving). However, there are cases where a single LPS program model a single agent which may need to reflect physical constraint of being able to only perform one action at a time. The OS does not specify such a limit on how many actions can be selected and executed in the same cycle. If the LPS programmer require such a limit, constraints can be added, as done in some of the LPS programs in the test suite.

In Wei’s thesis, several \( D_{pre} \) satisfiability considerations were discussed [5, p. 37]. More prominently, Wei suggested that in addition to the selection of actions that needs to be considered for checking for \( D_{pre} \) satisfiability, the fluents that the selected actions would initiate or terminate must also be included in the satisfiability check.

2.7.8 Operational Semantics Example

To illustrate how the OS works, this section details the intermediate steps taken by the OS in executing the following example LPS program:

```plaintext
1 maxTime(4).
2 actions ([run(FromAttacker), fight(WithAttacker)]).
3 fluent (provokingProvoker)).
4 events ([
```
attack(Attacker),
prove(Provoker).
}

initially([stronger(bob)])

observe(attack(alice), 1, 2).
observe(prove(bob), 2, 3).
inTrouble(Attacker) -> decide(Attacker, T1, T2).

% Ways the agent could be in trouble
inTrouble(Attacker) <- attack(Attacker, T1, T2).
inTrouble(Provoker) <- provoking(Provoker, T).

% Agent beliefs
decide(Attacker, T1, T2) <-
  stronger(Attacker, T1),
  run(Attacker, T2, T3).
decide(Attacker, T1, T2) <-
  not stronger(Attacker, T1),
  fight(Attacker, T2, T3).
initsiates(prove(P), provoking(P)).
terminates(fight(A), provoking(A)).
terminates(run(A), provoking(A)).

Most features, not all, of the OS would be demonstrated in the above example. The initial state $S_0$ is set to $S_0 = \{\text{stronger(bob)}\}$ from the initially declaration.

Rule Pre-processing Example

First of all, the rules get pre-processed. The only rule in the program:

\[
\text{inTrouble(Attacker) -> decide(Attacker, T1, T2)}.
\]

gets preprocessed and would be replaced by the following two rules by backward reasoning of inTrouble(Attacker):

\[
\text{attack(Attacker, T3, T4) -> decide(Attacker, T1, T2)}.
\text{provoking(Attacker, T) -> decide(Attacker, T1, T2)}.
\]

Appropriate variables renaming was applied to ensure correctness. These two rules would now replace the original rule.

Cycle 0 to 1

In this cycle, $E_1 = \emptyset$. The OS carries out Step 0 to update the current state and hence $S_1 = S_0$. Step 1 and 2 does nothing.
2.7. LPS OPERATIONAL SEMANTICS

Cycle 1 to 2

In the observation declarations, \texttt{attack(alice)} gets executed externally in this cycle. Hence \( E_2 = \{ \texttt{attack(alice)} \} \).

The OS carries out Step 0 and sets \( S_2 = S_1 \). In Step 1, the antecedent of the rule
\[
\text{attack(Attacker, T3, T4)} \rightarrow \text{decide(Attacker, T1, T2)}.
\]
gets resolved by the event \texttt{attack(alice, 1, 2)} and the resolvent would be as follows:
\[
\text{true} \rightarrow \text{decide(alice, T1, T2)}.
\]
Since the resolvent instance rule has been fired, a new goal tree – labelled as Tree A – with the root goal clause \texttt{decide(alice, T1, T2)} is added.

In Step 2, Tree A is resolved with Steps 2.1 and 2.2 as follows:

\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node (decide) at (0,0) {decide(alice, T1, T2)};
  \node (stronger) at (-3,-1) {stronger(alice, T1), \texttt{run(alice, T2, T3)}};
  \node (not_stronger) at (3,-1) {not stronger(alice, T1), \texttt{fight(alice, T2, T3)}};
  \node (attack) at (-3,-2) {\{Attacker/alice\}};
  \node (attack2) at (3,-2) {\{Attacker/alice\}};
  \node (T1/1) at (0,-3) {\{T1/1\}};
  \node (fight) at (0,-4) {fight(alice, T2, T3)};

  \draw[->] (decide) -- (stronger);
  \draw[->] (decide) -- (not_stronger);
  \draw[->] (stronger) -- (attack);
  \draw[->] (not_stronger) -- (attack2);
  \draw[->] (attack) -- (T1/1);
  \draw[->] (attack2) -- (T1/1);
  \draw[->] (T1/1) -- (fight);
\end{tikzpicture}
\caption{Tree A: Step 2.1 unwraps the composite event \texttt{decide(alice, T1, T2)} using the two clauses found in \( L \). Step 2.2 then finds that for \( \{T1/1\} \), the subproof of \texttt{stronger(alice, 1)} fails and hence results in the leaf node \texttt{fight(alice, T2, T3)}.}
\end{figure}

Step 2.3 produces the only candidate action \texttt{fight(alice, T2, T3)}.

Cycle 2 to 3

In the observation declarations, \texttt{provoke(bob)} gets executed externally in this cycle. The OS also selects \texttt{fight(alice, 2, 3)} for execution from the set of candidate actions produced by Step 2.3 of the previous cycle. Hence \( E_3 = \{ \texttt{provoke(bob)}, \texttt{fight(alice)} \} \). The OS completes Step 0 and sets \( S_3 = S_2 \cup \{ \texttt{provoking(bob)} \} \) because \texttt{provoke(bob)} initiates \texttt{provoking(bob)}.

In Step 1, the antecedent of the rule
\[
\text{provoking(Attacker, T)} \rightarrow \text{decide(Attacker, T1, T2)}.
\]
gets resolved by the fluent \texttt{provoking(bob, 3)} and the resolvent would be as follows:
true → decide(bob, T1, T2).

Since the resolvent instance rule has been fired, another new goal tree – labelled as Tree B – with the root goal clause decide(bob, T1, T2) is added. In Step 2, Tree B is resolved with Steps 2.1 and 2.2 as follows:

![Diagram of Tree B]

Figure 2.6: Tree B: Step 2.1 unwraps the composite event decide(bob, T1, T2) using the two clauses found in L. Step 2.2 then finds that for \( \{T1/3\} \), stronger(bob, 3) holds and hence results in the leaf node run(bob, T2, T3).

Steps 2.1 and 2.2 also resolves goal tree A since fight(alice, 2, 3) was executed internally. Step 2.3 produces the only candidate action run(bob, T2, T3).

Cycle 3 to 4

The OS also selects run(bob, 3, 4) for execution from the set of candidate actions produced by Step 2.3 of the previous cycle. Hence \( E_4 = \{run(bob)\} \). The OS completes Step 0 and sets \( S_4 = S_3/\{provoking(bob)\} \) because run(bob) terminates provoking(bob). Step 1 does nothing, but Step 2 resolves goal tree B since run(bob, 3, 4) was executed. The program then terminates since maxTime(4) was declared.

2.8 Rules Antecedent and Firing Issues

Mentioned earlier in Section 2.2 and 2.7.5, rules are fired as long as their antecedents become empty (i.e. true) by resolution described in Section 2.7.2. However, the manner in which rules are fired – despite being simple – may create pitfalls for LPS programmers.

2.8.1 Fluents in Rule Antecedent and Refraction

In LPS, the fluents are timestamped. Two schools of thought arise in how extensional and intensional fluents should be considered when firing rules:

In the first school of thought, fluents should be regarded by itself without its timestamp, i.e. a ‘fact(a)’ at time 1 is the same as from ‘fact(a)’ at time 2, when resolving state conditions in the
antecedent. Such an approach causes additional semantical ambiguity and complexity to the model theoretics of LPS. For example, should the resolution be considered only at the first instance when the fact holds or should it consider every time the fact becomes true (from false).

In the seconds school of thought, fluents should be considered together with its timestamp, i.e. ‘fact(a)’ at time 1 is different from ‘fact(a)’ at time 2, when resolving state conditions in the antecedent. This method is clear for the implementation of interpreter that a rule should fire for the same fluent at different time as the time variable binds to a different value and avoids any additional complexity in deciding when rules should fire. This is how the LPS model theoretics consider fluents.

Since fluents in LPS are considered together with their timestamps, consider the following example rule:

\[
\text{fire}(T) \rightarrow \text{deployFirefighter}(T_1, T_2), \text{putOutFire}(T_2, T_3), \text{returnToStation}(T_3, T_4).
\]

A simplified description of the rule is as such: Whenever there is fire, deploy a firefighter, the firefighter then puts out the fire, and finally the firefighter returns to station. The action putOutFire \((T_2, T_3)\) in the rule would terminate the fire. The unsuspecting LPS programmer might expect the rule to only fire once whenever fire(T) becomes true, but in fact it would fire twice. An example execution schedule for the rule is described in Figure 2.7.

![Figure 2.7](image)

**Figure 2.7:** As the fluent fire(T) only gets terminated in the second step of the sequence of actions described by the rule, it is possible for LPS to fire the same rule more than once in sequence as shown, producing results different from the programmer’s expectation.

LPS programmers should instead consider fluents in rule antecedents as though it is a "while-loop". Hence in the running example, the rule would be arguably better be read as "while fire holds, ...". If the programmer wishes to fire the rule only once whenever the fluent becomes true, there are two alternative solution:

- Implement the program such that an action/event gets executed when the fluent becomes true and use that action/event in the rule’s antecedent.
- Implement a mutex fluent to ensure that any time only one instance of the rule gets fired until it ends. This is implemented in the “mark.lps” example program in Appendix C.

Kowalski and Sadri [2, p. 13; 23, p. 140] considers such cases to be refractions, as described by Brownston et al.[24], since the same set of rules are fired at each timestamp by LPS when its antecedent holds for those states. [2, p. 13] suggests the use of events instead of fluents in the antecedent of rules as a solution to refraction.
2.8.2 Negation in Antecedent

The model theoretics enable the use of negation – i.e. the not operator – in the antecedent. However, it is important for the LPS programmer to recognise that variables in LPS rules abide by quantification in first order logic. Consider the following rule:

\[
\text{not } p(X), q(X) \rightarrow r(X)
\]

In the model theoretics and using negation as failure described in Section 2.4.1, the correct quantification should be interpreted formally as:

\[
\forall X \left[ \text{not } \exists X' p(X') \right], q(X) \rightarrow r(X)
\]

The variable \(X\) that occurs in \(\text{not } p(X)\) does not instantiate the variable \(X\) in the terms that appear after and hence the actual resolution of the rule would not have gone the way the programmer had expected. When the first condition (i.e. the negation) in the antecedent becomes resolved, the remaining rule would become:

\[
q(X) \rightarrow r(X)
\]

On the other hand, if the rule had been written this way:

\[
q(X), \text{not } p(X) \rightarrow r(X)
\]

the negation would be applied on each instantiation of \(X\) given by \(q(X)\) and the rule would work as the programmer expected.

2.9 Multi-Agent Systems in LPS

2.9.1 Introduction of MAS in LPS

Kowalski and Sadri in their paper mentioned how Agent-Oriented Programming (AOP) could bring about the key advantages of object-oriented programming paradigm – such as encapsulation – into LP[25]. In a different paper, they have also highlighted the importance of applying AOP in LPS to represent an agent’s beliefs and goals[22]. Wei’s thesis has also reiterated the same point[5]. There are several multi-agent platform implementations available, such as 2APL, 3APL, Jason and DALI[26].

There several key aspects of MAS and AOP[27, 25]:

- At the individual level:
  - An agent have some sort of knowledge base.
  - An agent have goals and beliefs - expression of plans and intentions.
  - An agent can carry out actions based on their individual worldview - reactivity.

- Between agents:
  - Agents can communicate and reason with each other.
  - Agents can assume roles and form hierarchy.
In practice, there are also other considerations, such as:

- **Agent Heterogeneity**: Whether agents are written using the same language / protocols
- **Communication Protocols**: The communication protocols between agents\(^{28, 29}\)
- **Shared vs Distributed Environments**: Whether the agents’ knowledge based is shared or not

Olivier et al.\(^{27}\) also had mentioned that the programming language or platform needs to be implemented with specific constructs for supporting AOP. Several example MAS programs, such as bank.lps in Appendix C, have been written for LPS interpreter written in Prolog to demonstrate the possibility of building MAS using LPS.

### 2.9.2 Network Architecture Choices

There are several choices in the architectural design of a MAS. The first would be the location of the blackboard among agents. The blackboard, based on the blackboard architectural pattern\(^{30}\), is where one or more agents will put information to or take information from\(^{31}\).

There are two mains ways of placing a blackboard in a MAS:

- A non-networked blackboard, where agents can directly access the blackboard within a single process.
- A networked blackboard, where agents communicate with over the network.

\[\text{Figure 2.8: Visualisation of the two blackboard placement. Part (a) shows a non-networked blackboard, where one or more agents can access the blackboard locally within the process. Part (b) shows a networked dedicated blackboard, where agents communicate with the blackboard over the network.}\]

The networked blackboard, despite being more scalable compared to the non-networked version, is subject to network delays and classical synchronisation issues.
2.9.3 Agent and Knowledge Representation

Another consideration is the agent architecture and knowledge representation in a MAS network. There are two possible implementations:

- Agents share a single blackboard.
- Agents each have their individual blackboard.

![Figure 2.9: Visualisation of the two placement of knowledge. In part (a), the knowledge base is stored in a central blackboard shared by agents. In part (b), each agent has their own blackboard representing their own worldview.](image)

A single blackboard offers consistency control over the environment since constraints can be written to govern the entire blackboard and the worldview of all agents at once. However, at the expense of consistency, the single blackboard does not offer agents the choice to have their own worldview without restrictions and hence enabling different agents to have conflicting worldviews.

2.10 JS Concurrency Model

To enable lps.js to receive observations while executing the cycle iterations, some form of concurrency or parallelism is required. This section discusses the concurrency model of JS.

2.10.1 Event Loop and Queue

JS – typically being single-threaded – implements a queue and event loop for its concurrency model[32]. Each message in the queue is associated with a function and that function is always guaranteed to execute completely without interruption. The JS programmer has control over the granularity of concurrency by controlling the amount of code to run in the function associated with the message in the queue.
2.10. JS CONCURRENCY MODEL

The message queue is typically implemented as a FIFO queue. While the JS programmer can determine how frequent a “context switch” happens by deciding the amount of time spent executing the function associated the message being processed, the programmer does not have control over the order of the queue. The event loop implementation for JS runtime in web browsers can be expressed as the following code:

```javascript
1 while (queue.waitForEvent()) {
2     process(queue.front());
3 }
```

JS runtime in web browsers do not have programmer control over termination as the browsers retain that control. In Node.js runtime, the JS programmer has control over program termination and hence the event loop implementation for Node.js is instead expressed as:

```javascript
1 while (!queue.isEmpty()) {
2     process(queue.front());
3 }
4 process.exit(0);
```

**Code Snippet 2.11:** In Node.js, the JS environment terminates once the queue becomes empty.

In Node.js, the JS script itself is always the first item to be added to the queue for processing. There are several methods to put additional messages onto the queue:

1. Timers, i.e. `setTimeout` and `setInterval`
2. Asynchronous functions (implemented by JS runtime)
3. UI events (clicks, key presses etc.)

2.10.2 JS Concurrency Example

Consider the execution of the following JS script in the Node.js runtime:

```javascript
1 function sayHello() {
2     console.log('Hello ');
```

---

2These are only implemented and are available on browser JS runtimes, not in Node.js runtime.
Chapter 2. Background 2.10. JS CONCURRENCY MODEL

3 }
4
5 // add sayHello to queue without delay
6 setTimeout(sayHello, 0);
7 console.log('World!');

The output would be as follows:

1 World!Hello

When the execution starts, the entire script forms the first message in the queue. The function `sayHello` gets added to the message queue at line 6, but does not get executed immediately. The string ”World!” gets printed out first as the execution of the current message in the queue needs to be completed (hence run to completion scheduling). Once the execution of the script’s entry point is done, the event loop picks the next message in the queue for execution, which is the function `sayHello`. Hence the string ”Hello” is printed after ”World!”.

2.10.3 Callbacks and Promises

JS has first-class functions: Functions can be assigned to variables and passed into other functions as an argument.[33] For example, suppose a function initiates an asynchronous task that runs concurrently or on a different thread, it is useful to be notified when the task has completed. One common convention used in JS is to use callbacks.

```
var func = function () {
    console.log('Hello there!');
};

console.log(func); // Output: [Function: func]
func(); // Output: Hello there!
```

**Code Snippet 2.12:** A function, being first class in JS, can be stored in a variable.

One example of using callbacks in the use of Asynchronous JavaScript and XML (AJAX). JS in web pages can access the browser-provided API `XMLHttpRequest` to request for more files or information from the server after the page has been loaded.[34] It is useful to know when the requested files or information are received by the browser so that the web page can perform additional tasks with these received files or information. An example of such a use case is shown below:

```
function makeRequest(url, callback) {
    var xHttpReq = new XMLHttpRequest();
    xHttpReq.addEventListener("load", callback);
    xHttpReq.open("GET", url);
    xHttpReq.send();
}

var callbackFunc = function () {
    console.log('page loaded!');
};
```
However, callbacks being used repeatedly and irresponsibly by the programmer can lead to a phenomenon called "pyramid of doom" or "callback hell"\(^3\). The callback hell code can become hard to maintain and flooded scope with irrelevant variables, making it difficult to debug and maintain. One possible mitigation for callback hell is to use newer JS programming language constructs such as Promises or await/async.

```javascript
function makeRequest(url, callback) {
  var xHttpReq = new XMLHttpRequest();
  xHttpReq.addEventListener("load", callback);
  xHttpReq.open("GET", url);
  xHttpReq.send();
}

makeRequest('https://facebook.com/', () => {
  makeRequest('https://wikipedia.org/', () => {
    makeRequest('https://google.com/', () => {
      // >>>>> pyramid of doom
      console.log('pages loaded!');
    });
  });
});
```

Promises are used in JS to chain asynchronously executed tasks. A promise is in one of the following states: pending, fulfilled and rejected. A resulting value is be associated with the fulfilled state, while an error is associated with the rejected state. \(^3\)

Consider the same example that performs an AJAX request to the server written with promises instead of callbacks.

```javascript
function makeRequest(url) {
  return new Promise((resolve) => {
    var xHttpReq = new XMLHttpRequest();
    xHttpReq.addEventListener("load", resolve);
    xHttpReq.open("GET", url);
    xHttpReq.send();
  });
}

makeRequest('http://example.com/').then(() => {
  console.log('page loaded!');
});
```

The Promise object returned from the `makeRequest()` function provides a `then()` method which can be used to define the function that gets executed when the promise has been fulfilled. It is

\(^3\)\url{http://callbackhell.com/}
possible to chain a sequential order of promises as so:

```javascript
function makeRequest(url) {
  return new Promise((resolve) => {
    var xHttpReq = new XMLHttpRequest();
    xHttpReq.addEventListener("load", resolve);
    xHttpReq.open("GET", url);
    xHttpReq.send();
  });
}
makeRequest('http://facebook.com/
  .then(() => {
    return makeRequest('http://wikipedia.org/
  })
  .then(() => {
    return makeRequest('http://google.com/
  })
  .then(() => {
    console.log('pages loaded!');
  });
```

The ES6 Promise API provides the JS programmer fine control over how promises are executed and along with error handling. The async/await keywords were introduced to further simplify promises.

### 2.11 Electron Architecture

To develop additional demonstration programs for LPS, the development of LPS Studio was done using Electron as it met the following conditions favourably:

- Desktop apps written can be written in JS under the Electron framework, allowing lps.js to be imported directly without any porting.
- Electron framework allows apps to be published to multiple desktop platforms using a single codebase.

Building upon the JS concurrency model, Electron uses Node.js (UI-less JS execution) and Chromium (Hyper Text Markup Language (HTML), Cascading Style Sheets (CSS), and frontend JS execution) to render an app written in its framework. In particular, there are two processes launched when an Electron app is launched:

- The main (backend) process that is executed in Node.js. This process also launches the Chromium renderer processes to display GUI.
- The renderer processes display the HTML and CSS and executes any JS code declared in the HTML in browser context. It is possible for the main process to launch multiple renderer processes, as each window displayed could run in its own sandbox environment in a single renderer process.
The two processes run in disjoint contexts and can only communicate with each other by means of a Inter-Process Communication (IPC) API provided by Electron through the types `ipcMain` and `ipcRenderer` for the main and renderer processes respectively. Nokes also suggests that any Central Processing Unit (CPU)-intensive operation should be executed in the main process rather than in the renderer process to avoid blocking the UI as how Google recommends in their Page-Speed tool documentation.

In particular, the renderer process has access to the Document Object Model (DOM) of the process's view and is able to receive UI events while the main process do not. Using the Electron's architecture, the overall architecture of LPS Studio's implementation would be described in Figure 2.13.

![Figure 2.13](image-url)
Chapter 3

Implementation

The implementation details of lps.js and its extensions are discussed in this chapter.

3.1 Implementation Considerations

There were several possible directions open for the implementation of LPS in JS:

- A runtime-only environment that will:
  1. Parse a LPS program
  2. Interpret the LPS program
  3. Execute the LPS program

- A compiler implementation that will:
  1. Parse a LPS program
  2. Translate the program into a suitable JS representation
  3. Package LPS-JS code into a standalone JS program with a runtime interpreter

In this project, only the first option was explored in detail. lps.js was developed as a portable LPS runtime environment that executes LPS programs in the JS environment. The project’s duration was insufficient to explore the implementation of a LPS source to JS source compiler and hence the compiler implementation would be seen as possible future work, where its details will be discussed in greater detail in Section 5.2.1. The remainder of this chapter talks about the implementation of lps.js as a runtime environment.

3.2 LPS Syntax Introduction

The syntax of LPS accepted by the parser implemented in lps.js can be described using the Backus-Naur form (BNF) description as follows:

```
QuotedString ::= "" STRING_LITERAL "\n"
| "\r" STRING_LITERAL "\r"
Constant ::= QuotedString
| (LOWER_CASE_ALPHABET NO_WS_STRING_LITERAL)
```
3.2. LPS SYNTAX INTRODUCTION

Chapter 3. Implementation

Number ::= NON_ZERO_NUMBER+ NUMERIC_LITERAL
| NON_ZERO_NUMBER+ NUMERIC_LITERAL "." NUMERIC_LITERAL
| "0." NUMERIC_LITERAL
| "0x" HEXADECIMAL+
| "0b" BINARY+

Arguments ::= Expr ("," Expr)*
Variable ::= (UPPER_CASE_ALPHABET | "_") NO_WS_STRING_LITERAL
VarOrNum ::= Variable | Number
Functor ::= LOWER_CASE_ALPHABET NO_WS_STRING_LITERAL
| (" Arguments ")"
List ::= 
| [Expr ("," Expr)*]?
| (Expr ("," Expr)*) "|" (List | Variable)
"
SimpleExpr ::= 
| (" Expr ")"
| Constant
| Functor
| VarOrNum

UnaryExpr ::= 
("not" | "!" | "-")
SimpleExpr

MultiplicationExpr ::= 
UnaryExpr
| ("**" | "*" | "/")
UnaryExpr
|
AdditionExpr ::= 
MultiplicationExpr
| ("+" | "-"
| MultiplicationExpr
|"
ComparisonExpr ::= 
AdditionExpr
| ("==" | "<>" | ">=")
| "<" | ">" | ">="
| "@<" | "@>")
AdditionExpr
|"
AssignmentExpr ::= 
ComparisonExpr
( "=" ComparisonExpr)?
Expr ::= List | AssignmentExpr
Conjunct =
  ( "not" Conjunct )
  | ( Expr
      ( "from" VarOrNum ( "to" VarOrNum )?)
      | ( "at" VarOrNum )
    )
Conjunction ::= Conjunct ( "," Conjunct ) *
Constraint ::= "<-" Conjunction "."
Fact ::= Conjunction "."
Clause ::= Conjunction "<-" Conjunction "."
Rule ::= ( Conjunction | "true") "->" Conjunction "."
Sentence ::= Clause | Rule | Fact | Constraint
Program ::= Sentence *

where:

- "STRING_LITERAL": Set of zero or more string characters, with quotes appropriately escaped.
- "LOWER_CASE_ALPHABET": Lower case alphabet "a" to "z" only.
- "NO_WS_STRING_LITERAL": Set of zero or more string characters without any whitespace characters.
- "NUMERIC_LITERAL": Set of zero or more number characters "0" to "9" only.
- "NON_ZERO_NUMBER": Number that is not zero only, i.e. "1" to "9" only.
- "HEXADECIMAL": Hexadecimal characters only, i.e. "0" to "9", "a" to "f", "A" to "F".
- "BINARY": Binary characters only i.e. "0" or "1".
- "UPPER_CASE_ALPHABET": Upper case alphabets only, i.e. "A" to "Z".

The file format needs to be saved with the UTF-8 encoding, as the interpreter would read the file using the same encoding. ASCII encoding is also supported since the UTF-8 encoding is a superset of the 7-bit ASCII encoding and hence files written in ASCII encoding is also supported[39].

Comments are supported in the syntax using the following constructs:

- The character "#" until the end of the line a line comment, as implemented in most other programming languages.
- The character "%" until the end of the line as a line comment, as implemented in Prolog.
- The sequence "/*" until the sequence "*/" as a block comment, as implemented in most programming languages.
3.3. PARSER IMPLEMENTATION

The syntax supports the Prolog convention to differentiate between variables and constants by having upper case strings considered as variables – e.g. "Person", "Location" – and lower case as constants – e.g. "alice", "kitchen".

However, in place of Prolog’s ::-/2 if-operator, lps.js uses the following operator symbols for representing the different statements:

- -> for rules, e.g. antecedent -> consequent.
- <- for constraints and clauses, e.g.
  
  <- constraint.
  
  conclusion <- condition.

Some LPS programs written in this syntax can be found in Appendix C.

Functions

In Prolog, only the is/2 predicate supported nested expressions, such as \((5 + X) \cdot Y\), and functions, such as \(\text{min}/2\). The syntax defined for lps.js instead support such nested expressions and functions within predicate arguments in the program, such as in arguments of functor: \(\text{point}(X, Y) <- \text{location}(X \cdot 20 + Y)\). In most built-in functions, the function will be evaluated as soon as sufficient number of arguments are instantiated.

OR operation

In lps.js, the OR operator (i.e. ;/2 in Prolog) is not defined. OR operations are not explicitly defined to avoid over-complication of the OS, but it still can be implemented through the use of composite events / macros, such as in the case below:

1. \(\text{fire}(T) -> \text{deal_with_fire}(T_1, T_2)\).
2. \(\text{deal_with_fire}(T_1, T_2) <- \text{eliminate}(T_1, T_2)\).
3. \(\text{deal_with_fire}(T_1, T_2) <- \text{escape}(T_1, T_2)\).

When the rule is fired, the composite event \(\text{deal_with_fire}(T_1, T_2)\) can be satisfied by either executing \(\text{eliminate}(T_1, T_2)\) or \(\text{escape}(T_1, T_2)\).

3.3 Parser Implementation

The parser implemented in lps.js is a Recursive Decent Parser (RDP) [40]. Following Ferg’s guide [41] for implementation, the lexer and parser implemented the syntax described in Section 3.2.

3.3.1 Lexer

The lexical analyser (lexer) splits the source code into sequential chunks that makes it easier to work with. Using a source code scanner that allows lookahead on the next character – i.e. having access to the current character and the next character – the lexer can determine what kind of token the current character should be part of. Some of the token types are as follows:

- Variable
- Symbol (e.g. +, -, *, =, <)
Chapter 3. Implementation

3.3. PARSER IMPLEMENTATION

- Constant
- Number
- Eof (end of file)
- Whitespace
- Comment
- Keyword

The lexer has been implemented to ignore comment and whitespace chunks of the source code. Each token is associated with the current line in the source code and it's column so that when a parsing error occurs, the associated location of the error can be reported to the programmer for debugging.

3.3.2 Parser

Using the sequence of tokens produced by the lexer, the parser uses RDP technique to produce an AST. The parser is partially aware of the grammar and would enforce some of the grammar rules when parsing the tokens. Each node in the AST is associated with one of the following node types:

- Program: Typically the root node of the entire source code.
- Sentence: Either a clause, constraint, rule or fact.
- Conjunction: A conjunction of atomic terms.
- Constant
- Variable
- Number
- Functor: A functor can be either a predicate or a function symbol
- ListHead: Head of a List
- List: A List representation
- BinaryOperator: Binary operators such as +, <=
- UnaryOperator: Unary operators such as !, -
- Symbol

At this stage, some of the syntactic expressions would not be included in the AST, such as "." at the end of sentences and "," argument or list item separators. The complete AST would then be passed to the interpreter to convert into a Program that stores facts, constraints, rules and clauses in disjoint sets.
3.3.3 Syntax Error Reporting

Syntax errors found by the parser and interpreter must be reported to the LPS programmer meaningfully. Most compilers and interpreters provide basic information such as the file pathname, line number and column where the error occurred.

In Node.js, syntax errors also show the snippet of the source code where the error had occurred, along with an arrow to indicate the exact position of the error. Such graphical representation of error can be helpful to the programmer.

![Image of syntax error in Node.js]

**Figure 3.1:** In JS, the syntax error messages indicate where in the source code did the error occurred by pointing it using a caret symbol.

lps.js followed the examples set by Node.js and other implementations in its syntax error reporting by indicating the same useful information. Unlike most imperative programming languages, syntax and execution errors in LPS cannot provide a stack trace and hence all other information associated with the error must be as helpful as possible to the programmer.
3.4. Unification

3.4.1 Unification by Recursive Descent

A unification implementation in lps.js is needed for the interpreter to perform backward and forward reasonings of queries through a given LPS program. The unification algorithm implemented was based on Unification by Recursive Descent (URD) [43, pp. 31-33; 44, pp. 447-448], where the algorithm would perform a recursive search to find disagreements between two given terms. A pseudo-code of the algorithm is given as follows:

---

3.3.4 Static Syntactical Warnings

Most programming languages provide some form of clever warnings to prevent programmer mistakes. For example, in SWI-Prolog, a singleton variable\(^1\) that does not start with an underscore is considered to be possibly a typo by the programmer[42]. Where appropriate, lps.js follow suit and implemented some warnings that could have been arisen from programmer's mistake.

---

\(^1\)variable that only appear once in the statement

---

Figure 3.2: lps.js provides useful syntax error reporting, along with code snippets of the source code where the error occurred.

The syntax errors reported by lps.js not only show the line number, column and file pathname where the error occurred, but also includes snippets of the lines before and after where the error occurred. The caret symbol points exactly where the parser thinks the error is.
Algorithm 3.1 unify($P, Q, \theta$)

Require: $P$ is a term
Require: $Q$ is a term
Require: $\theta$ is a set of substitutions

1: if $P$ is a Variable then
2: $P = \text{substitute}(P, \theta)$
3: end if
4: if $Q$ is a Variable then
5: $Q = \text{substitute}(Q, \theta)$
6: end if
7: if $P = f(p_1, \ldots, p_n)$ and $Q = g(q_1, \ldots, q_n)$ then
8: if $f = g$ then
9: for $i = 1$ to $n$ do
10: $\theta = \text{unify}(p_i, q_i, \theta)$
11: end for
12: return $\theta$
13: end if
14: return Error
15: else if $P$ is not Variable and $Q$ is not Variable then
16: if $P = Q$ then
17: return $\theta$
18: end if
19: return Error
20: else if $P$ is not Variable then
21: return unify($Q, P, \theta$)
22: end if
23: $\theta = \theta \cup \{P \rightarrow Q\}$
24: return $\theta$

The URD algorithm was implemented in the early versions of lps.js but was replaced by a dual-purpose data-structure which will be discussed in Section 3.4.2.

3.4.2 Literal Tree Maps (LTMs)

Introduction of LTMs

A special data-structure of interest was developed in lps.js to (i) contain a list of facts without duplication and (ii) allow fast membership check. A set DS was needed to efficiently store the set of active fluents that could be queried, initiated or terminated.

Prolog identifies predicates by their name and arity. For example, the predicate `append(L1, L2, L)` would have an identifier "append/3". There were no known Prolog implementation that would store constraints, clauses and facts as disjoint sets. In Tau Prolog\(^2\), facts are kept in the set of "rules", along with all the constraints and clauses, and are accessible by the predicate’s identifier[45].

The data-structure implemented in lps.js for the special purpose is a key-value store called a Literal Tree Map (LTM). Leaf nodes in the tree are uniquely identified by a given instance of a

\(^2\)A Prolog implementation in JS by José Antonio Riaza Valverde – http://tau-prolog.org/
Chapter 3. Implementation

3.4. UNIFICATION

predicate and hence LTM maps predicates – along with its arguments – in a tree structure to leaf nodes. The leaf node can be used to store anything, which by default would store a reference to the original predicate data representation. The operations supported by LTM are:

- Adding a leaf node identified by a predicate.
- Removing a leaf node identified by a predicate.
- Checking if a leaf node of a predicate exists in the tree.
- Finding the set of leaf nodes that unify with a predicate.

By default, the LTM is used to store facts and hence the key and value of each entry would be the same. In reality, LTM can be also used to store a set of clauses that share the same head term.

LTM nodes use hash tables at each internal node to store their child nodes for quick lookup and ensuring a unique set of child nodes.

For terms that have nested predicates, such as \( \text{pred}(X - Y, A + B) \), the nested predicates \( X - Y \) and \( A + B \) are stored separately in another tree, which is distinctively named as the Argument Tree. A visualisation of this can be found in Figure 3.4. An argument tree can also recursively have another argument tree, depending on the amount of nesting the programmer used.
3.4. **UNIFICATION**  

**Chapter 3. Implementation**

---

**Figure 3.4:** Illustration of how Literal Tree Maps would store nested predicates. The main tree is on the left, where it contains `pred(X - Y, A + B)`, which can also be read as: `pred(-(X, Y), +(A, B))`. Since there are function symbols used in the predicate’s argument, another tree (i.e. the Argument Tree) is needed to store those symbols.

A similar structure called classification tree was proposed by Kowalski and Kuehner in their paper [46, pp. 245-246]. Like LTM, classification trees can perform conjunction satisfaction checks down the trees as literals are be classified based on arguments’ types.

**Unification in LTM**s

Unification is also possible using LTM, since the tree structure stores predicates in a left-to-right manner. The tree structure enabled a variant of the URD algorithm to be implemented. Consider the example tree in Figure 3.5.

---

**Figure 3.5:** Example LTM containing 5 facts of `mom(X, Y)`. 

---
Suppose a unification query `mom(X, ben)` was submitted asking for the mother of Ben. The query would then be converted into a query path `[mom, 2, X, ben]`, which the unification algorithm would use to traverse down the tree. Upon reaching a variable in the path, the traversal algorithm would try all possible subbranches. The unification result would return `[X = jill]`.

Suppose another unification query `mom(jane, Y)` was submitted asking for the children of Jane. The query would then be converted into a query path `[mom, 2, jane, Y]`, which the unification algorithm would also use to traverse down the tree. The unification result would return `[Y = brad, Y = barry]`. Figure 3.6 shows the intermediate steps taken in the unification of this query.

<table>
<thead>
<tr>
<th>Query</th>
<th>External Substitution 1</th>
<th>Tree Path 1</th>
<th>Internal Substitution 1</th>
<th>External Substitution 1</th>
<th>Tree Path 2</th>
<th>Internal Substitution 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mom(jane, Y)</td>
<td>θ_{e1} = {}</td>
<td>root</td>
<td>θ_{i1} = {}</td>
<td>θ_{e2} = {}</td>
<td>root</td>
<td>θ_{i2} = {}</td>
</tr>
<tr>
<td>mom</td>
<td>θ_{e1} = {}</td>
<td>mom</td>
<td>θ_{i1} = {}</td>
<td>θ_{e2} = {}</td>
<td>mom</td>
<td>θ_{i2} = {}</td>
</tr>
<tr>
<td>2</td>
<td>θ_{e1} = {}</td>
<td>2</td>
<td>θ_{i1} = {}</td>
<td>θ_{e2} = {}</td>
<td>2</td>
<td>θ_{i2} = {}</td>
</tr>
<tr>
<td>jane</td>
<td>θ_{e1} = {}</td>
<td>jane</td>
<td>θ_{i1} = {}</td>
<td>θ_{e2} = {}</td>
<td>jane</td>
<td>θ_{i2} = {}</td>
</tr>
<tr>
<td>Y</td>
<td>θ_{o1} = {Y \mapsto \text{barry}}</td>
<td>barry</td>
<td>θ_{i1} = {}</td>
<td>θ_{o2} = {Y \mapsto \text{brad}}</td>
<td>brad</td>
<td>θ_{i2} = {}</td>
</tr>
<tr>
<td>S_θ =</td>
<td>{Y \mapsto \text{barry}, {Y \mapsto \text{brad}}}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.6: Illustration of how LTM does unification down the paths in the tree using internal and external substitutions for the example query of `mom(jane, Y)` on the tree given in Figure 3.5.

When performing unification, LTM considers a unique set of internal and external substitutions for each path from the root node to the leaf node. Whenever at an internal node the unification cannot happen for the given set of internal and external substitutions, LTM will not explore the subtree rooted at that internal node and hence all leaf nodes in that subtree are said to not unify with the given goal.

In another example, a fact stored in a LTM is given by `fact(X, X)`. The LTM unification using internal and external substitution can answer a query `fact(a, Y)` as depicted in Figure 3.7.
3.5 LPS Programs

3.5.1 LPS Program Software Quality Considerations

As the implementation of lps.js influences the implementation of the LPS programming language, it is imperative to consider how a LPS programmer would write and run LPS programs using lps.js.

Writing a LPS program is no different from writing source code of any software using other programming languages. The design and implementation of lps.js must be directed towards helping programmers apply good software engineering principles and practices when writing LPS programs.

In [47], Milicic presented several definitions and models which suggests several attributes that contribute to code quality. One key attribute of McCall's Quality Model was *product revision*, and there are three subpoints:

- Maintainability
- Flexability
- Testability
There are many software engineering principles (such as SOLID principles) and design patterns (from Gang of Four or architectural-based patterns), but only a handful may be applied to a logic programming language like LPS due to difference in programming paradigm. For example, SOLID principles only apply to programming languages that implement the Object-Oriented Programming paradigm.

For a declarative logic programming language like LPS, there are still design patterns and software engineering principles that can help to increase code quality. In particular, lps.js was designed with the following principles in mind:

- Coupling and Cohesion
- Separation of Concerns
- Reusability

**Coupling and Cohesion** was first studied extensively and mentioned by Constantine[48] targeted at reducing software complexity and increasing maintainability. Coupling refers to the connection between components (or modules) and cohesion refers to the measure of closeness between parts of each component. Good software quality strives for low coupling and high cohesiveness.

In the context of LP (and therefore LPS), it is difficult to measure – and hence reduce – coupling since programs are written declaratively as statements. However, it is possible to increase cohesiveness in a LP program by ensuring that LPS code written in a single file are highly related to each other, and non-relevant code are written in separate files.

**Separation of Concerns** The term coined by Dijkstra[49] refers to the idea of separating a program into several parts, where each part would address a single concern. Like other programming languages, LPS needed to provide separation constructs that can enable programmers to separate a LPS program into logical components[50].

A large monolithic LPS program can be hard to maintain and debug. As with Prolog, it is possible to decompose a large LPS program into several files, then combine them before execution. Each file should aim to achieve high cohesiveness and collectively all files should achieve separation of concerns. A single entry LPS file can then use the consult/1 predicate to stitch the entire LPS program together from the individual files.

**Reusability** It is also even possible to encourage code reusability[51] as a single LPS file – and hence its facts, logic program and reactive rules – can be reused by several LPS programs.

### 3.5.2 Declarations

Apart from rules, clauses, constraints and facts, LPS programmers can also add declarations that modify the behaviour of the LPS’s program execution. These declarations are written as facts that are queried by the interpreter during initialisation. For example, predicates can be distinguished as fluents, actions or events by using the following predicates:

- fluent/1, fluents/1. Examples:

[48] uses the word **Cohesiveness** instead of **cohesion**.
3.5. LPS PROGRAMS

- fluent(fire(Location))
- fluents([fire(Location), water(Amount)])

• action/1, actions/1. Examples:
  - action(run(Path))
  - actions([run(Path), take(Item)])

• event/1, events/1. Examples:
  - event(detect(Object))
  - events([detect(Object), receive(Message)])

Meta-predicates that are defined by the domain theory are also supported in lps.js:

• initiates/2. Examples:
  - initiates(ignite(Location), fire(Location))
  - initiates(lend(Amount, Borrower), loan(Borrower, Amount))

• terminates/2. Examples:
  - terminates(extinguishFire(Location), fire(Location))
  - terminates(repayLoan(Amount, Borrower), loan(Borrower, Amount))

• updates/3. Examples:
  - updates(moveTo(NewLoc), location(_), location(NewLoc))
  - updates(send(From, To, Amt), quantity(From, Bal), quantity(From, Bal - Amt))

Other declarations include:

• maxTime/1. The number of iterations the program is allowed to be executed for. If not specified in the program, lps.js uses the default value of 20 cycles.

• cycleInterval/1. The maximum amount of time allowed for each cycle in milliseconds. If not specified in the program, lps.js uses the default value of 100ms.

Declarations can also appear as head of clauses in LPS. However, if only one instance of the declaration is used, lps.js will pick one by unification.

3.5.3 Consult and Augmentation

The interpreter was implemented to enable a loaded LPS program to augment additional program files into one single program, as a Prolog program would using consult/1. This is useful for the programmer to break a large program into several files for organisation and maintenance.

The ability to augment LPS files encourages reusability of code from a single file instead of maintaining several copies of the same code across different files, which in turn increases programmer's productivity in software development[51]. For example, in the case of applying LPS to multi-agent systems modelling, agents with identical beliefs and goals can share and load their beliefs the same LPS file.
lps.js also took advantage of this augmentation to delegate the implementation of some built-in predicates to be written in LPS files rather than in JS. This is only done where the implementation in LPS would drastically reduce the number of lines of code compared to JS and where LPS would describe the predicates better in clauses than imperative statements in JS. Some of the built-in predicates are defined in terms of their JS counterparts. A part of the built-in definitions written in LPS is given as follows:

```prolog
1 fluent(F) <- fluents(L), member(F, L).
2 action(F) <- actions(L), member(F, L).
3 event(F) <- events(L), member(F, L).
4 observe(F, T, T2) <- observe(F, T), T2 = T + 1.
5 loadModule(F) <- loadModules(L), member(F, L).
```

In the interpreter's implementation, it is sufficient to traverse over one version of the predicate instead of two. For example, using the LPS declarations above, it is sufficient for lps.js to query for `fluent/1` alone as the SLD resolution will also return results that are written using `fluents/1`.

### 3.5.4 Execution Modes

LPS Programs with lps.js can run in two modes: (i) Non-Continuous Execution or (ii) Continuous Execution. In Non-Continuous Execution mode, lps.js would trigger the OS cycle at every cycle interval defined. When a cycle starts before the previous cycle ends, a failure condition would be triggered and lps.js would terminate the execution of the LPS program. In Continuous Execution mode, the OS cycles of the LPS programs happen one after another without delay. This means that the interpreter only treat the given cycle interval as a deadline for the current cycle and the following cycle always happen right after the end of a cycle. Figure 3.9 illustrates the timeline of both mode.

Note that the plural version accepts a list.
3.6 Real-Time Observations

In the initial stages of the project, a lps.js prototype was implemented to receive observations from externally triggered events. The lps.js prototype would load and execute a given LPS program while listening for observations over a Transmission Control Protocol (TCP) server. Using `netcat` or any appropriate networking utility program, observations can be sent into the lps.js prototype to be considered as an observed event.

Later, the prototype was integrated into the `lps` command line tool of the `lps-cli` package, where users have an option to turn the TCP observation server on or keep it off.

Real-time observation support is also very much needed in lps.js to support user-initiated events, such as mouse clicks and key presses, in the browser context. These events can occur at any point in time and LPS needs to handle its timing appropriately.
Chapter 3. Implementation  3.6. REAL-TIME OBSERVATIONS

The timing of observations has been defined in the OS for pre-defined observe declarations, but not declarations that can be triggered at any arbitrary timing. The definition of handling real-time observation is given using the example in Figure 3.10. All real-time observations are given to lps.js without their time variables instantiated.

Figure 3.10: A visualisation of cycle execution time vs cycle interval, along with how events are treated in the OS implemented in lps.js. The yellow diamond from time 2 to 3, shows the point in time where the lps.js variable state "currentTime" gets updated while the red circles show the point in time when lps.js receives the two observations event1 and event2.

Consider the occurrence of event1 in Figure 3.10, event1 is received by lps.js when the execution of cycle from time 2 to 3 has started. It is not possible for the event to be considered as executed in the time 2 to 3, but in 3 to 4 i.e. event1(3, 4), so that the start of next cycle can process this given event.

Similarly, the occurrence of event2 in Figure 3.10 is only received by lps.js after the execution of cycle time 2 to 3 has been ended. The event can only be treated as event2(3, 4) so that the next cycle can handle its occurrence.

The interleaving JS execution enabled observations to be inserted into lps.js at any timestamp as the observations comes in. It reflects how LPS programs cannot anticipate real-world observations ahead of time. It is also not possible to assume that real-world events that are ideally supposed to happen one after another would be observed perfectly by the LPS program in sequential timestamp order.

The Prolog interpreter also disregards any observation that violates the constraints of the program. While it is assumed that observations are model-wise correct, lps.js followed the Prolog implementation to implement a constraint check on incoming observations. Observations, like candidate actions, that violate the constraints given in the program are rejected by lps.js with a warning.

In theory, an infinite amount of observations can be received within a cycle by an LPS program. Since observations require some processing within the cycle, when an external source sends too many observations to the LPS program, it is possible that the processing of these observations would end up taking longer than the cycle interval and hence causing the cycle to time out. A possible solution could be for the interpreter or the LPS programmer to specify a maximum number of observations to be received per cycle. Beyond the maximum number of observations, later observations can either be discarded or cascaded to the following cycles.
As JS is single threaded (see Section 2.10), careful consideration in the design and implementation was required to allow interleaving the execution of receiving observations and performing cycle state transition.

### 3.7 JS Observer Events

The interpreter was implemented with Observer design pattern to enable programmers using lps.js to attach JS observer functions to be notified when a certain event related to interpreter has been fired. These events are fired from within the interpreter and are independent of the LPS program under execution by lps.js.

![State Diagram](image)

**Figure 3.11:** The state diagram of lps.js's interpreter as a finite state machine. The transition between states can be observed through the observer events implemented.

For example, the "ready" event gets fired when the main and augmented program files have been loaded from the file system and the interpreter can start the OS cycle execution. Programmers can use lps.js's interpreter API to attach an observer as such:

```javascript
engine.on('ready', () => {
    engine.run();
});
```

Some of the observer events include:

- **loaded**: When program files are loaded but before processing declarations.
- **ready**: When program files are loaded and declarations have been processed.
• **run**: When interpreter starts running the LPS program.

• **done**: When the terminating condition of the LPS program has been reached. Once done has been fired, no other observer events would be fired.

• **preCycle**: When the interpreter is going to advance from one cycle to the next.

• **postCycle**: When the interpreter has advanced to the next cycle.

• **paused**: When the interpreter has been paused. If the pause request happens in the midst of a cycle transition, the cycle transition would complete anyway.

• **unpaused**: When the interpreter has been unpaused.

• **error**: When an unrecoverable runtime error occurs. Once error has been fired, the interpreter will halt and no other observer events would be fired.

• **warning**: When a warning occurs.

In particular, the "postCycle" event is useful in retrieving the active fluents and other information after a state transition has been completed.

However, as described in Section 2.10, programmers must avoid writing observers that take up too much processing time without giving other items in the JS message queue a chance for execution. The observers cannot be called asynchronously by the interpreter as the observers may only executed at a different point in time due to the event loop execution of JS described in Section 2.10.

### 3.8 JS Execution from LPS

As mentioned in Wei’s thesis conclusion, connecting LPS to actuators and sensors can enable LPS to receive real-world events and provide physical feedback[5, p. 53]. The scheduling required for real-time observations have been resolved in Section 3.6.

Moreover, executing lps.js on Node.js runtime enables LPS to access features provided by Node.js, such as file system input/output and computer networking. By enabling the execution of JS functions from LPS programs, lps.js enable LPS to be applied to more use cases. Likewise in the web browser runtime, browser events – such as mouse clicks, window resizing, keyboard strokes – can be sent to lps.js as observations.

Programmers can choose to apply both programming paradigms offered by LPS and JS in their usage through lps.js.

#### 3.8.1 API Design

Since lps.js was developed independently from any Prolog implementation, several built-in predicates, such as `member/2` and `append/3`, had to be re-implemented in JS. This also meant that programmers using lps.js can define predicates by supplying their predicate identifiers and respective JS function handlers.
lps.js provides the API for JS-based predicates to indicate the execution result through the JS `return` statement. The expected return type is a JS array, where each item of the array has required key "theta" indicating a single suitable substitution from the result of this operation and an optional key "replacement". An empty array is interpreted by lps.js as the execution of the predicate being failed and hence any successful execution must return at least one element in the array, even when the set of substitutions is empty. Consider the example `sum/3` predicate definition below:

```javascript
engine.define('sum/3', function(op1, op2, v) {
  if (!(v instanceof Variable)) {
    // failed because third argument is an output argument
    return [];
  }
  let variableName = v.evaluate();
  let theta = {};
  theta[variableName] = op1.evaluate() + op2.evaluate();
  return [
    { theta: theta }
  ];
});
```

In the example above, it is said that the operation had failed if the third argument given was not of type `Variable`. The function needs to return an empty array to indicate that there are no possible solutions from this query. If a variable is given for the third argument, the predicate could perform the summation and return a single substitution for the variable. Suppose a query `sum(1, 2, X)` was submitted, the function would return:

```javascript
[{
  theta: { X: Value(3) }
}]
```

In some cases, the predicate needs to return more than one result for the given query. Using the same format, the predicate can return more than one element in the resulting array. One example of such use case would be the implementation of `member/2` where a query such as `member(X, [1, 2, 3])` would need to return different substitutions of `X`.

To explain the optional "replacement" key, consider the simplified implementation of the `+/2` LPS function in JS\(^5\) below:

```javascript
engine.define('+/2', function(operand1, operand2) {
  let value1 = operand1.evaluate();
  let value2 = operand2.evaluate();
  return [
    { theta: {} }
          replacement: new Value(value1 + value2)
  ];
});
```

\(^5\)That is derived from the source code of lps.js
The "replacement" key helps to indicate what should replace a \(+/2\) function as its result. This is used in the preprocessing of queries that have functions in it. For example, the query \(\text{cost}(5 + 3)\) would be read by lps.js as \(\text{cost}(+(5, 3))\), and the replacement tells lps.js to simplify the term into \(\text{cost}(8)\).

However, consider the following program that uses a replacement version of the \(\text{random}/0\) predicate.

```
1: action(test/1).
2: true -> test(random) from T1 to T2.
```

**Code Snippet 3.12:** The interpreter will try to execute \(\text{test}(\text{random})\) from time 1 to time 2, but the value given by the execution of \(\text{random}/0\) does not unify with \(\text{random}\) in the clause.

The unsuspecting programmer expects the rule to fire once from cycle 1 to 2. However, it is not the case as lps.js performs a replacement for \(\text{random}/0\). Suppose \(\text{random}/0\) returns a replacement of the value 0.5, lps.js would execute \(\text{test}(0.5)\). It becomes clear that the executed action \(\text{test}(0.5)\) does not unify with the consequent \(\text{test}(\text{random})\)\(^6\), lps.js does not resolve the goal clause and would pick \(\text{test}(\text{random})\) to be executed at every cycle.

This is one consequence of the LPS syntax allowing functions to be written outside of \(\text{is}/2\). There are two solutions to this problem. From the interpreter's part, a slight modification to the goal tree solution process could help resolve the original example. Currently when performing function evaluations, the goal tree does not add an additional node for function evaluations. By adding an additional node, the goal tree is able to have a node that resolves with the executed action. For example:

```
1: Goal Node Clause 1: test(random) from T1 to T2.
2: Goal Node Clause 2: test(0.5) from T1 to T2.
```

**Code Snippet 3.13:** When functions are executed, their result needs to be replaced in the goal node clause and an additional node needs to be added to the goal tree.

Alternatively on the programmer's part, a slight modification to the original program would produce the expected result:

```
1: action(test/1).
2: true -> R = random, test(R) from T1 to T2.
3: % Alternatively:
4: % true -> random(R), test(R) from T1 to T2
```

\(^6\)Random is treated as a constant in goal tree resolution process, despite being a function.
The full list of built-in predicates supplied by lps.js's core module can be found in Appendix D. For the LPS programmer, there are two ways of adding JS-based predicates:

- Using interpreter's API from JS
- Loading from a JS file

### 3.8.2 Adding JS-based predicates from JS

It is possible for developers to add JS functions in JS as predicates in LPS by directly using the `Engine.define()` method. For example, consider a predicate that prints a given term on the console screen can be defined using the following code:

```javascript
engine.define('logTerm/1', function (value) {
    // prints term onto console
    console.log(value);
    // result of the operation must be returned
    return [{ theta: {} }];
});
```

The predicate is associated with the given function. Whenever the interpreter encounters the predicate, it would execute the associated JS function.

### 3.8.3 Loading JS-based predicates from LPS

Since a LPS program could also be executed from the CLI tools in lps-cli, it would not be possible for the programmer to define JS-based predicates ahead of execution. lps.js provides a standard way of loading JS-based modules before the execution of the LPS programs start. The LPS program defines which JS file it would like to load as a module through the `loadModule/1` built-in predicate. This works similarly to Node.js's `require()` function and Prolog's `consult/1` predicate, except that a JS file is loaded instead.

The JS file being loaded needs to be written in a specific way in order for the execution to be compatible with lps.js. Specifically, the JS file needs to export a function with two arguments. When the function is executed, the current instance of the interpreter and the loaded program objects are passed into the function. For example, the module can define additional JS-based predicates the JS definition file named "customModule.js" would be written:

```javascript
module.exports = (engine, program) => {
    const functors = {
        'print/1': function (value) {
            console.log(value);
            return [];
        }
    };

    engine.getFunctorProvider().load(functors);
};
```
And from the LPS program, the JS file could be loaded:

```prolog
1 maxTime(5).
2 loadModule('customModule.js').
3 ...
```

The current instance of the interpreter and the program is passed into the loaded module `customModule.js`. It is possible for the loaded module to define additional events, predicates and configure the interpreter’s options before the execution starts.

### 3.8.4 Asynchrony and JS-based Predicates

JS’s programming paradigm allows asynchronous and concurrent execution of code. However, as mentioned in Section 3.8.1, JS-implemented predicates must return a result at the end of its execution. While it is possible to initiate asynchronous execution in the predicate functions, `lps.js` cannot allow JS-based predicates to delay the answer further through Promises or callbacks.

Execution time spent inside the execution of JS-based predicates count towards the cycle execution time. Tasks in JS-based predicates that can possibly block up a significant amount of execution time should be reconsidered in its design.

The programmer needs to be aware of these limitations and work around it. For example, it may be better to start an asynchronous task in the JS-based predicate and use `lps.js` interpreter’s observation API to inform the LPS program of the result later.

### 3.8.5 Implementation Considerations

While the ability to load and execute JS functions opened doors to many more application of LPS, there were cross-implementation issues that need to be considered carefully. For example, a Prolog or Java LPS interpreter would not have the exact same API for `loadModule/1` implemented and even if it has been implemented, the interpreter would not be able to load and interpret a JS file without the use of a JS runtime. Hence, at the point of writing, the design of `loadModule/1` is meant to only extend the capabilities of `lps.js` and is considered to be non-standard of LPS.

The loading of JS-based predicates from file will also be unavailable on web browser runtimes as client-side JS are loaded over HTTP and do not have access to file systems. It is considered impractical to implement `loadModule/1` such that it would perform a HTTP request for additional files over the network.

For the LPS programmer using `lps.js` on the web browser context, the programmer instead has API control over `lps.js` and can easily load additional JS files using the HTML `script` tag and add JS-based predicates to `lps.js` by JS.

### 3.9 Operational Semantics Implementation

This section discusses the details and considerations of the OS of LPS implemented in `lps.js`.  

51
3.9.1 Rule Consequent Resolution using Goal Trees

Goal trees in the LPS represent the possible resolutions of a fired rule’s consequent. In initial versions of lps.js, goal trees were implemented as explicit rooted trees where the root node represents the instance of the consequent of the fired rule, and all descendent nodes in the tree from the root node are created by Step 2.1 and 2.2 of the OS. Each path (or branch) from the root node represents one possible resolution of the root node. The path from root node to a leaf node that has the empty goal explains how consequent of the fired rule can be satisfied.

![Goal trees](image)

**Figure 3.14:** A goal forest is maintained in the OS to keep track of unresolved consequents of rules that have been fired. Illustrated here, are two goal trees representing the computation by the interpreter trying to resolve the consequent of two rules. Each node contains a single conjunction. The child nodes of a node were generated using Step 2.1 and 2.2 of the OS and are alternative choices on how the parent node can be resolved. Each alphabet in the node represent the unique subgoal / conjunction.

In Figure 3.14, an example goal state is given by the two goal trees for two hypothetical rules in a program. As mentioned in Section 2.7.4, the goal state is a conjunction of trees. Each alphabet in the tree represent a distinct conjunction. The following are appropriate representations of the goal state in Figure 3.14:

- \( a \land g \)
- \( a \land (d \lor b) \)
- \( (b \lor c) \land (d \lor b) \)
- \( ((c \lor d) \lor c) \land (d \lor b) \)
- \( ((c \lor d) \lor c) \land (e \lor b) \)
- \( ((c \lor e) \lor c) \land (e \lor (c \lor d)) \)
- \( ((c \lor e) \lor c) \land (e \lor (c \lor e)) \)
- \( \ldots \)
However, the implementation of explicit goal forest caused greater execution times of LPS programs in lps.js as compared to the existing Prolog implementation on SWISH due to repeated computations. The example goal state in Figure 3.14 supports the possibility of many unnecessary recomputation. For example, the Concurrent Towers LPS program took about 6 seconds to complete execution on lps.js as compared to less than 2 seconds of execution on the on LPS interpreter written in Prolog.

![Diagram of explicit goal trees](image)

**Figure 3.15:** The earlier versions of lps.js uses explicit goal trees like the one in Part (1) of the diagram. Node \( a \) is the consequent given at the firing of the rule, while nodes \( b, c, d, e \) and \( f \) were added to the tree through Steps 2.1 and 2.2 of the OS. In tree (1), \( b \) has a first conjunct in the node's goal as a time unbounded fluent and could be evaluated again in later cycles. \( d, e, f \) are leaf nodes that need to be further evaluated later. However in each cycle, the explicit tree in (1) needs to restart the traversal of the tree from the root. This is acceptable for small trees, but will run into longer execution times when trees are larger. The first change was to treat the tree as an implicit tree, like the one shown in Part (2). Nodes connected by dashed arrows – nodes \( b, d, e \) and \( f \) – form an evaluation queue and would be processed in that sequence. Traversal for each cycle in the implicit tree only involves nodes that could potentially still generate new nodes for the tree.

To improve performance, lps.js used an implicit tree representation. Nodes that needed to be evaluated in the next cycle are added back into a evaluation queue. The conditions for nodes being added to a evaluation queue are:

- The first conjunct in the node's goal is a time unbounded fluent.
- The node's goal clause contains at least one candidate action that could be executed in the next cycle.

### 3.9.2 Rule Consequent Resolution using Goal Directed Acyclic Graphs

However, when executing LPS programs – such as goat river crossing (goat.lps in Appendix C) – goal trees showed to generate exactly identical goal clauses in different branches and redundantly perform Step 2.1 and 2.2 on all of them. Avoiding redundant calculations and reusing previously calculated results for subsequent identical goal clauses proved to be efficient in speeding up goal tree resolution. In the later versions of lps.js, goal trees are treated as Directed Acyclic Graph (DAG) by collapsing identical goals across branches into one through the use of memoisation.
3.9. OPERATIONAL SEMANTICS IMPLEMENTATION

Figure 3.16: Some LPS programs generated the same goals across different branches in the same cycle, like the one shown in Part (3) of the figure. The node c occurred twice and its subtree would have to be generated twice. The larger the subtree, the slower the execution of the goal tree resolution. By reusing previously generated results, the number of nodes reduce significantly as shown in Part (4) of the figure.

Figure 3.17: With the example given in Figure 3.14, goal forests are combined into multiple DAGs. The amount of computation has visibly reduced for each goal graph as re-computation of the exact same subgoal has been avoided. Solid lines represent actual computation, while dashed lines represent reuse of previous computation.

At the time of writing, goal trees implemented in lps.js are both implicit and are DAGs. The implementation showed promising results as it reduced execution time of examples like goat river crossing (goat.lps in Appendix C) from about 6 seconds to less than 2 seconds.

3.9.3 DAGs and Refractions

Goal DAGs also helped to deal with rule refractions by avoiding recomputation across DAGs. Instead of keeping track of previously computed subgoals within the same tree, lps.js was modified to keep track of previously computed subgoals across all instances of unresolved consequents. Sup-
pose a rule refracts and the consequent is not time-bounded, it is possible that the first consequent instance computed for the current time is also needed in the computation of the second consequent.

![Figure 3.18](image)

**Figure 3.18:** With the example given in Figure 3.14, goal DAGs prevent recomputation across fired rule instances. Amount of computation is further reduced. Solid lines represent actual computation, while dashed lines represent reuse of previous computation.

### 3.9.4 Non-deterministic resolution and Goal DAGs

One issue with reusing previously computed result is in dealing with non-determinism intended by the programmer, for example using the `random/1` predicate. It is possible that the LPS programmer expects a different result for each subtree in the consequent resolution, and hence an occurrence of resolution for `random/1` should not be reused by another subtree. However, since it is not possible for the LPS programmer to look at the internal resolution of the goal DAGs directly, the non-deterministicness shared across the branches of the goal tree would still make sense to the LPS programmer.

### 3.9.5 Sorting Terms by Deadline

The LPS OS in Section 2.7 described the need to distinguish the sets of early conjuncts and later conjuncts given a timestamp. In lps.js, a sorting mechanism for this sole purpose was developed for use when resolving rule antecedents and in Step 2.2 of the OS cycle. The algorithm – named `sortTimables` – has a linear time complexity with respect to the number of conjuncts given in the conjunction, i.e. $O(n)$ time complexity for $n$ conjuncts given.

The function takes in an array of conjuncts as the conjunction and the current timestamp, with the assumption that the conjunction contains no conjuncts with timestamps earlier than the given timestamp. `sortTimables` will then return a pair of disjoint conjunct sets: (i) the set of early conjuncts, and (ii) the set of later conjuncts. The pseudocode of the algorithm is given in Algorithm 3.2.

Using only a left-to-right evaluation order, the interpreter cannot proceed whenever the programmer places the ordering of timestamped conjuncts wrongly. However, lps.js using `sortTimables` is
able to handle incorrect ordering of timestamped conjuncts. An example of a misordered conjunction is \([\text{action1}(T_2, T_3), \text{action2}(T_1, T_2)]\), where the \(T_2\) variable in \(\text{action1}/2\) clearly depends on \(\text{action2}/2\). Nonetheless, all non-timing related variables still need to observe left-to-right ordering.

As of writing, the current implementation does not consider timing variable orderings due to constraints like comparison operations, such as \(T > 5\) or \(T_1 < T_2\). A static analysis implementation is needed to support such time dependencies. For example:

- A time constraint \(T > 5\) can be converted into a timestamped conjunct \(true\) from 5 to \(T\). Only at time 5 the timestamped conjunct will then be part of early conjunct and freeing the variable \(T\) from the set of dependent time variables.
- A time constraint \(T_2 = T_1 + 1\) can be converted into a timestamped conjunct \((T_2 = T_1 + 1)\) from \(T_1\) to \(T_2\).

Algorithm 3.2 gives the algorithm implemented for \(\text{sortTimables}\).

**Algorithm 3.2** \(\text{sortTimables}(C, T)\)

1: Set of early conjuncts: \(C_e = \emptyset\)
2: Set of later conjuncts: \(C_l = \emptyset\)
3: Set of dependent time variables: \(D = \emptyset\)
4: for conjunct \(a \in C\) do
5: if \(a\) has a timestamp then
6: if \(\text{endTime}(a)\) is a variable \(\land \text{startTime}(a)\) is a variable then
7: \(D = D \cup \{\text{endTime}(a)\}\)
8: end if
9: end if
10: end for
11: for conjunct \(a \in C\) do
12: if \(a\) has no timestamp then
13: if \(\text{size}(C_l) > 0\) then
14: \(C_l = C_l \cup \{a\}\)
15: else
16: \(C_e = C_e \cup \{a\}\)
17: end if
18: else
19: if \(\text{startTime}(a) = T \lor \text{startTime}(a) \notin D\) then
20: \(C_e = C_e \cup \{a\}\)
21: else
22: \(C_l = C_l \cup \{a\}\)
23: end if
24: end if
25: end for
26: return \((C_e, C_l)\)
3.9.6 Candidate Actions Selection Strategy

Earlier in Section 2.7.7, several selection strategies of picking actions to be part of $act_i$ from the set $cand_i$ were discussed. Recall that the selection happens in Step 0 of each cycle, and in the first cycle – time 0 to time 1 – the set of candidate actions is empty. Candidate actions of each goal tree are produced from Step 2.1 and 2.2 of the OS.

A candidate actions set is defined to be a set of candidate actions that must be executed together in the same cycle from the same leaf node of a goal tree. For example, if a leaf node of the goal tree has the following goal:

$$act_1(T_1, T_2), act_2(T_1, T_2), act_3(T_2, T_3)$$

where $act_1$, $act_2$ and $act_3$ are actions. Step 2.3 will produce $act_1(T_1, T_2), act_2(T_1, T_2)$ as a single candidate actions set for selection.

The current selection strategy implemented in lps.js used Dynamic Programming (DP) to maximise the selection of actions across goal trees without violating constraints. A simplified description of the selection algorithm is given as follows:

**Algorithm 3.3** selectActions($P, G, N, k, A$)

1. if $k \geq N$ then
2.   return $A$
3. end if
4. let $l = 0$
5. let $M$ be the number of candidate action sets in goal tree $g_k \in G$
6. for candidate actions set $\alpha$ of goal tree $g_k \in G$ do
7.   let $\Delta_\alpha$ be the set of state changes caused by $\alpha$
8.   $P' = P \cup \alpha$
9.   if constraintsViolated($P'$) then
10.     continue for loop
11. end if
12. $P'' = P \cup \Delta_\alpha$
13. if constraintsViolated($P''$) then
14.     continue for loop
15. end if
16. $l = l + 1$
17. $A' = A \cup \alpha$
18. $R = selectActions(P', G, N, k + 1, A')$
19. if $R$ is not null then
20.     return $R$
21. end if
22. end for
23. if $l = M$ then
24.     return null
25. end if
26. return selectActions($P, G, N, k + 1, A$)
3.10. MULTI-PLATFORM TARGETING

As the selection algorithm iterates the goal trees in $G$ from a left to right fashion, the ordering of the goal trees in $G$ is important. The priority is given to the choice of actions from the left over the choice of actions from the right if only one can be chosen to be executed. In lps.js, an earliest to latest evaluation is assumed to ensure that actions that need to be executed in the current cycle is given priority over other time unbounded actions. As such, the goal trees are sorted by their earliest deadline before actions are selected.

3.9.7 Dealing with Refractions in Rules

Refractions – previously discussed in Section 2.8.1 – happen whenever rules fire multiple times for the same conditions satisfied. lps.js attempted to deal with refractions by considering a rule as fired without adding its consequent goal tree if there exists at least one other unresolved goal tree whose root clause is the same as the rule’s consequent.

For example, consider the following simplified program that illustrates the point:

```
1 initially(fire).
2 fire(T) -> deployFirefighter(T1, T2), putOutFire(T2, T3),
   returnToStation(T3, T4).
3 terminates(putOutFire, fire).
```

Since none of the variables in the consequent are bound by those in the antecedent, it is possible for the rule to only fire once for multiple instances of the antecedent. However, this may violate the implicit temporal constraint mentioned in Section 2.7.3, where time variables in the consequent must not come earlier than the antecedent. An example schedule of the example rule that violates the implicit temporal constraint:

- At time 1, fire(1) holds and the rule is fired with the consequent $[\text{deployFirefighter}(T1, T2), \text{putOutFire}(T2, T3), \text{returnToStation}(T3, T4)]$. The action deployFirefighter(1, 2) can be selected and executed, resulting in the remaining consequent given by $[\text{putOutFire}(2, T3), \text{returnToStation}(T3, T4)]$.

- At time 2, fire(2) holds and the rule is fired. However, since the goal tree for the previous firing is still yet to complete and has the same consequent as this rule, the rule is considered to be fired. However, this violates the the implicit temporal constraint since the executed action deployFirefighter(1, 2) occurs earlier than the antecedent of fire(2).

3.10 Multi-Platform Targeting

One of the main goals of a JS implementation of LPS framework was portability, where the implementation could run on both backend systems and on client-side runtimes. The current implementation of lps.js was written for the Node.js runtime as that would be the primary use case.

To enable the use of lps.js on web browser runtimes, some form of conversion and polyfilling[52] would be required to bridge the gap in implementation for the different runtimes. There are two possible solutions for the LPS programmer:

- Transpiling - Programmers would use lps.js as a dependent library they would on Node.js. When bundling their source code for client side using a transpiler or bundler such as Browserify or Webpack, all dependencies are transpiled into a compatible JS language specification for the browser as part of the bundling process[53].


- Prepackaging - Another alternative for LPS programmers to use lps.js in the browser would be to download a pre-packaged or transpiled version of lps.js. The lps.js project maintainers would need to make available the transpiled version of the codebase for LPS programmers to download. Programmers can directly use this pre-packaged JS file in their web applications.

During execution, lps.js will also need to detect the runtime and determine which features should be made available or unavailable. For example, the file system I/O features are available on Node.js runtime but not in browser runtimes. An error message needs to be raised by lps.js if the programmer attempts to use file system I/O features in the browser runtime.

To maximise compatibility and reduce issues when converting source code for browser runtimes, the original source code was written in ES6 standard\[54\]. A linter – using eslint – set up was included in the lps.js implementation to ensure all code are compliant with the ES6 standard. lps.js uses no production-mode dependencies to ensure maximum portability of lps.js\[7\].

### 3.10.1 Setting up Transpilation

A transpilation\[8\] process needs to be integrated for lps.js so that the project maintainers can pre-package lps.js for browser and upload the bundled file for release. Developers who transpile lps.js on their own should be able to do so using the same process.

There are many tools publicly available for the purpose of transpiling most dialects of JS to a browser-compatible JS. They include Browserify, Webpack and Babel. lps.js was initially configured with Browserify but the transpilation failed as Browserify could only statically analyse dependencies usage (through `require()` in lps.js\[56\]). lps.js uses some extent of dynamic `require()` in the source code. Browserify was later replaced with the experimentation of Webpack as the author of Webpack claimed that Webpack could analyse dynamic `require()` to some extent\[56\]. The transpilation was successful and a client-side demonstration of lps.js was made. The user guide in Appendix E provides the instructions needed to perform the transpilation process.

### 3.11 Testing API

#### 3.11.1 Testing Introduction

lps.js was developed along with its unit test cases written and tested using Mocha JS test framework and chai assertion library. Test code coverage were compiled by Istanbul JS test coverage. To further automate the testing process, continuous integration was set up using the Travis-CI service\[9\]. The idea was to keep the interpreter under regression testing early and to detect failure against specification when changes were made.

Later in the project, the unit testing of lps.js itself was found to be insufficient. The tests, while covering most of the components on their own, did not cover the interaction between components and their net effect altogether. To support the automated testing of LPS programs, a Tester API was implemented to check the output of a LPS program against their expected output. Using this Tester API, additional component and end-to-end tests were introduced. The test suite from the

---

7The package dependencies of lps.js has been analysed using David DM at [https://david-dm.org/mauris/lps.js](https://david-dm.org/mauris/lps.js)

8Source-to-source compilation\[55\]

9[https://travis-ci.com/mauris/lps.js](https://travis-ci.com/mauris/lps.js)
Prolog implementation\(^\text{10}\) was also translated and integrated into the testing of lps.js. Consider the "fire-simple.lps" example:

```
1  maxTime(5).
2  fluent(fire).
3  actions([eliminate, escape]).
4  event(deal_with_fire).

5  initially(fire).
6
7  fire(T1) -> deal_with_fire(T1, T2).
8  deal_with_fire(T1, T2) <- eliminate(T1, T2).
9  deal_with_fire(T1, T2) <- escape(T1, T2).
10
11  terminates(eliminate, fire).
```

A specification file "fire-simple.spec.lps" along with the expected output of the program would be written as another LPS program, with special predicates that would be interpreted by the lps.js Tester library:

```
1  expect(maxTime(5)).
2
3  expect_num_of(fluent, 1, 1).
4  expect_num_of(action, 1, 0).
5  expect(fluent, 1, fire).
6
7  expect_num_of(fluent, 2, 0).
8  expect_num_of(action, 2, 1).
9  expect(action, 1, 2, eliminate).
10
11  expect_num_of(fluent, 3, 0).
12  expect_num_of(action, 3, 0).
13
14  expect_num_of(fluent, 4, 0).
15  expect_num_of(action, 4, 0).
16
17  expect_num_of(fluent, 5, 0).
18  expect_num_of(action, 5, 0).
```

As the specification file is also an LPS program, clauses can be written to dynamically determine the expectations. The set of test programs and their specification files were integrated together with lps.js's unit test cases to ensure that regression testing covers the testing of LPS programs that utilise different parts of the OS.

The Tester API would report if the test for each program was successful or not, and if not, which expectations were not met by the program. The full description of the Tester's API can be found in Appendix G.

\(^\text{10}\)https://bitbucket.org/lpsmasters/lps_corner/src/06b0de6227f7/examples/?at=master
3.11.2 Test Environment Simulation

![Diagram of Test Environment Simulation]

Figure 3.19: The interpreter loads the program under test, while the specifications and simulated observations are loaded by the Tester object. The program under test is then tested by the Tester against the loaded specifications while feeding simulated observations to provide a test result.

In the specification file, it is possible to add simulated observations using the `observe/2` or `observe/3` predicates that would trigger as observed events to the program under test when testing.

3.11.3 Special Comparator Terms

To help LPS programmers to describe their test specifications more precisely, special operators were added to describe numeric expectations. For example, consider the following LPS program where the actual number of cycles executed is determined by a random number generated by programming:

```
maxTime(10).
fluent(timeToStop/1).
action(setTiming/1).

% set timing to stop
randomInt(2, 5, N) ->
setTiming(N) from T1 to T2.

% when cycle reached, stop
timeToStop(T) at T ->
lpsHalt.

initiates(setTiming(N), timeToStop(N)).
```

It is not possible for the test specification to expect a fix number of cycles that would be executed. However, the Tester API supports a set of special numerical operators. One of which is the `between/2` predicate, which can be used to determine a range of accepted values. The test specification would be written as follows:

```
expect(maxTime(10)).
expect_num_cycles(between(2, 5)).
```
3.11.4 Test Specification Generator

Inspired by the Prolog implementation, lps-cli carries a CLI tool named lps-generate-spec that takes a program's output and generate test specification statements for that particular output. It reduces the burden of the LPS programmer to write test specification when the programmer has already verified the output of the program through other means, e.g. by looking at the timeline produced.

To generate the test specification, lps-generate-spec executes the program as a grey box and make notes of all visible output – fluents, actions and observations – to produce the specification statements. The test specification statements generated, however, are purely based on the observed output. The generator cannot produce statements to assert the absence of some fact.

Most test specification files in the test suite of lps.js were generated using this tool and verified to ensure correctness. Some of the test specification files were modified from the generated version to reflect a more fitting expectation from the program output.

3.12 Multi-Agent Programming Support

lps.js complements the LPS OS in supporting AOP and MAS implementation using LPS. There were two key implementations in lps.js written specifically to support MAS: Consult with Identifier and Peer-to-Peer module.

3.12.1 Consult with Identifier

The Consult with Identifier predicate consult(+File, +Id) is a modified version of consult/1 where the +Id supplies an identifier to the subprogram, i.e. the file being consulted. The LPS programmer could use the consult(+File, +Id) predicate to create instances of the same subprogram.

Suppose a main program main.lps consults the subprogram agent.lps with the identifier "alice", a pre-processor instantiates all statements (clauses, constraints and rules) with the grounding of processId(alice) in the subprogram agent.lps.

1  consult('./agent.lps', alice).

Code Snippet 3.20: The main program main.lps.
Code Snippet 3.21: The original subprogram agent.lps. Statements have a special predicate processId(-Id) in their antecedent.

When the subprogram gets grounded, the statements are replaced. The updated subprogram is then augmented into the main program.

```
hasTask(alice, Task) ->
    actOn(alice, Task).

isAvailable(alice) at T <-
    not hasWork(alice, T),
    isAtBase(alice, T).

<- not isAtBase(alice),
    actOn(alice, _).
```

Code Snippet 3.22: The pre-processed program with the processId/1 instantiated.

The final program being executed is represented in the Code Snippet 3.23.

```
consult('./agent.lps', alice).

hasTask(alice, Task) ->
    actOn(alice, Task).

isAvailable(alice) at T <-
    not hasWork(alice, T),
    isAtBase(alice, T).

<- not isAtBase(alice),
    actOn(alice, _).
```

Code Snippet 3.23: The final program of main.lps augmented with agent.lps.
The downside of relying solely on building MAS in a single LPS program is inability to scale. An example MAS program of borrowing-lending money scenario using consult with identifier can be found in Appendix C.

3.12.2 Peer-to-peer Module

To support over-network Peer-to-Peer (P2P) multi-agent communication, a module called p2p was implemented. Using the loadModule/1 declaration outlined in Section 3.8.3, LPS program can access the API provided by the module to communicate with other agent instances in the network. The reference for the P2P module in lps.js can be found in Appendix F.

For simplicity, a centralised peer list tracker was implemented in the lps-cli package to keep track of peers that are online. Whenever a new peer comes online, the tracker will send the addresses of existing peers to the new peer, and notify the others about the new peer joining the network.

While enabling scalability, the distributed nature of using the P2P features to build MAS reintroduces classical synchronisation problems and scheduling issues. These problems will rely on the
LPS programmer to mitigate them in programming. An example MAS program of same borrowing-lending money scenario using the P2P module can be found in Appendix C.

The P2P module also supports heterogeneous MAS agent implementations. If communication protocols differ, it is always possible to write protocol translation or wrapper programs to enable communication between two or more different platforms implementing the MAS.

3.13 Demonstration Website

To demonstrate the capabilities of lps.js, a web interface that consisted of a Angular 5 frontend application and ExpressJS web API application was implemented. A screenshot of the web interface is shown in Figure 3.25.

![A screenshot of the web interface implemented to run LPS programs using lps.js.](https://lps.mauris.sg/)

Figure 3.25: A screenshot of the web interface implemented to run LPS programs using lps.js.

The web application provided several built-in example LPS programs that could be loaded by the user on demand, and allowed the user to modify the LPS program before running it. Upon user's request to run a LPS program, the LPS program would be sent to the backend application for execution before the result was returned to the client for display.

The result would then be shown to the user in the form of a table timeline, where the fluents, events and actions are ordered according to their timestamps.

The web app was deployed onto a publicly-accessible server\(^\text{11}\). To protect the source code of the interpreter for the duration of the project, the LPS program execution is done on the backend. As demonstrated in Section 3.10, lps.js was designed and implemented with both Node.js and browser runtimes in mind and hence it would be possible for the LPS program to be executed directly on the webpage in the browser's environment.

\(^{11}\)https://lps.mauris.sg/
3.14 LPS Interactive Storytelling: LPS Studio

3.14.1 LPS Visualisation

With the implementation of Node.js and Github’s open source release of their framework for the Atom text editor, Electron, it became clear that desktop applications can be written using languages and tools that are familiar with web developers. In particular, since lps.js was written in JS, there was little obstacle in building a desktop application using LPS.

In particular, one area of interest was visualising LPS programs. Wei, in his thesis, had mentioned that a GUI implemented as future work for exploring LPS goal trees can be very useful for the programmers. Wielemaker et al. had implemented a visualisation API for LPS on the SWISH platform[6, pp. 20-21].

![A screenshot of the LPS visualisation implemented on the SWISH platform by Wielemaker et al.]

Instead of building a tool solely for LPS programmers to debug and see the internals of the LPS program being executed, the development of LPS Studio – the visualisation tool based on lps.js – took inspiration from the LPS visualisation on SWISH. Using Electron, LPS Studio took a step further to implement real-time observations (see Section 3.6) from mouse and keyboard events. LPS Studio enables LPS programmers to use LPS for interactive storytelling as the visualisation can depend on the user’s input or external events.
3.14.2 LPS Studio Features

To enable visualisation, the LPS programmer needs to write additional LPS code on top of the original program to dictate what shows up on the canvas of LPS Studio. There were three key groups of predicates that LPS Studio needs to define for the visualisation:

- Defining objects on canvas
- Moving objects around the canvas
- Interaction with objects on the canvas

The canvas in LPS Studio used the HTML5 defined `<canvas>` object, a 2-dimensional board that can be used to draw shapes and images. Additional built-in predicates are defined by LPS Studio for LPS to access features such as defining objects on the canvas and how to handle user input given on the canvas. Objects defined on the canvas minimally have properties such as its position and if it is hidden. Other specific type of objects on the canvas have other properties, such as the image used to represent the object.

Using LPS rules, the LPS visualisation program can update the state of objects on the canvas whenever actions or events occur. For example, consider the rule extracted from the Bubble Sort (LPS Studio version) example of Appendix C.

```lps
1 swap(A, J, B, K, T1, T2) -
2  locationScreen(J, X1, Y1),
3  locationScreen(K, X2, Y2),
4  lpsAnimateObject(A, 400, [
5  position(X2, Y2)
```
3.14  LPS INTERACTIVE STORYTELLING: LPS STUDIO

Code Snippet 3.28: This rule updates the position of two items being swapped by the bubble sort algorithm on the screen. `locationScreen/3` are auxiliary facts written by the programmer to map the item’s position index to 2D coordinates on the screen. The `lpsAnimateObject/3` predicate is a built-in predicate provided by LPS Studio to animate any animatable properties of the object, such as position in this example.

The `lpsAnimateObject/3` predicate used in the Code Snippet 3.28 is provided by LPS Studio to animate any animatable properties of the objects. Some of the animatable properties include position, size, radius of circle etc. LPS Studio provides abstraction over underlying animation mechanism and a finer cycle interval of about 16ms, for 60 frames per second, written in JS to provide smooth visual animation on the canvas.

UI events handled in JS are passed to LPS via observations: mouse clicks, mouse down, mouse move, mouse up etc. LPS Studio also send events that occur on a specific object defined on the canvas. For example, the observation `lpsClick/3` tells the LPS program which canvas object was clicked and the following reactive rule extracted from the Fire Station (LPS Studio version) example performs an ingite action if the item is flammable.

Code Snippet 3.29: This rule ignites a flammable item whenever the item is clicked on in LPS Studio.

A console output and statistical charts to the right of the canvas shows the useful information about the execution of the visualisation. It is also possible for the user to enter a set of command-separated observations using the text box at the bottom. LPS Studio treats the LPS program under execution as a black box and will not query its state to produce a timeline.

LPS programs executed in LPS Studio have the same full system access as it were executed within the CLI context. In Section 4.2.3, an example LPS program written for LPS Studio demonstrates the use of the `p2p` module to implement an over-the-network 2-Player Tic Tac Toe game.

For the reference and user guide of LPS Studio, see Appendix H.

3.14.3 User Controlled Agent in MAS Context

In a MAS context, it is possible to utilise LPS Studio to run a user-input agent. Feedback from and status of the MAS can be sent to the user-input agent for visualisation in LPS Studio to reflect the state of the system. At the same time, the user can enter keyboard, mouse or arbitrary events through LPS Studio to the user-input agent. The user-input agent, acting as a front for the entire MAS, can then relay the events to the rest of the MAS.
3.15 Atom Text Editor Language Package

As much of the development work on lps.js was done through Atom text editor, a language package that would provide syntax highlighting and autocomplete features in Atom for the syntax described in Section 3.2 would help to increase developer efficiency when working on LPS programs of that syntax.

A package for this purpose was developed with reference from the Prolog language pack\textsuperscript{12} and has been published on Github\textsuperscript{13} and Atom’s packages repository. Developers who need the LPS syntax highlighting can directly install the package onto their computer.

\textsuperscript{12}Published at https://github.com/Jakehp/language-prolog
\textsuperscript{13}Published at https://github.com/mauris/language-lps
Chapter 4

Results and Evaluation

With a working implementation of lps.js completed, the features and capabilities of lps.js could be demonstrated and evaluated.

4.1 Deliverables

Throughout the project period, multiple prototypes were produced, demonstrated and tested against a test suite of LPS programs provided by the Prolog interpreter’s repository. The source code of the deliverables was also refactored along the way to ensure clarity and simplicity for future maintenance. On 17 Aug 2018, the source code for lps.js, lps-cli and the demonstration website were released open source under the BSD 3-Clause license on Github. In summary, the following were delivered:

- An early demonstration website
  - The frontend web app published at https://github.com/mauris/lps-demo-web
  - The backend web API published at https://github.com/mauris/lps-demo-web-api

4.1.1 Publication to npm Repository

Upon the open source release, both lps.js\(^1\) and lps-cli\(^2\) went under semantic versioning[57] and were published to the npm repository for public retrieval via the npm tool. Instructions to installing both packages were published on their repository README files.

4.1.2 Distribution of LPS Studio

To facilitate distribution of LPS Studio, build scripts were successfully incorporated into LPS Studio to build distributable binary forms of LPS Studio for the various platforms. The processed relied on the open source tool electron-builder, which packages LPS Studio into the following formats:

---

\(^1\)lps.js published at http://npmjs.com/package/lps

\(^2\)lps-cli published at http://npmjs.com/package/lps-cli
4.2. EVALUATION

4.2.1 Test Suite

As mentioned in Section 3.11, a set of LPS programs from the LPS Prolog’s test suite was translated for lps.js. The set of programs tested the different specific parts of the OS implemented in lps.js and brought about issues which were fixed along the way. In addition to the programs translated from LPS Prolog repository, several additional programs were added to test lps.js-specific features. A total of more than 30 programs were part of the lps.js test suite at the time of writing.

To study the execution profile of running the different LPS programs under lps.js, the execution was analysed using Node.js’s internal profiling tool and the flamegraph generator package “0x”\(^3\). The discussions of several execution profiles are found in the following sections.

The flamegraphs show the amount of time spent in each function during the execution along with the stack trace. The horizontal axis does not represent the time of execution and function blocks are sorted left to right in alphabetical order. The blocks highlighted blue belong to the code executed within lps.js.

4.2.2 Analysis of Test Programs

To study the execution profile of LPS programs in lps.js and evaluate the performance of lps.js, three programs of the longest execution times amongst programs in the test suite were selected for profiling.

Case Study 1: Concurrent Towers Building

The concurrent towers building example – named towers.lps in Appendix C – relies heavily on recursion to achieve the correct sequence in which the blocks need to be moved in order to respect physical constraints. The goal is to construct two different towers and the initial state has blocks all over the place. Before a block can be moved, all blocks on top of that block must be cleared (placed elsewhere, for example on the floor) first before it can be moved.

\(^3\)https://github.com/davidmarkclements/0x
Figure 4.1 shows the execution profile and stack sampled from the execution. It reflects correctly that higher stacks were executed in lps.js since there is more recursion involved in the evaluation of the goal trees. Likewise, most of the execution time was spent in evaluating the consequent of fired rules.

Case Study 2: Goat River Crossing

Figure 4.2: The flamegraph and execution profile of the goat river crossing example.
4.2. EVALUATION

The goat river crossing example – named goat.lps in Appendix C – is a constraint-heavy example. There are three objects which the farmer would like to bring across the river: cabbage, goat and wolf. At any given point in time, only the farmer and at most one object can cross the river. It is also not possible to leave the goat and cabbage in the absence of the farmer, as the goat would eat the cabbage. It is also not allowed to leave the wolf and goat in the absence of the farmer, lest the wolf eats the goat.

The program expresses the goals of bringing all the items across the river and constraints as described. Figure 4.2 shows the execution profile and stack sampled from the execution.

Immediately, several observations can be made from the graph. Most of the execution time was spent in expanding the consequent of the fired rules, in the evaluate() method of the GoalTree class. It was likely due to the fact that fluents were used in the rules’ antecedent and thus refraction occurred. Despite being a constraint-based program, relatively little time was spent in the constraint checking and action selection process.

Case Study 3: Silly Dialogue

The silly dialogue example – named turing.lps in Appendix C – has a significant amount of recursion in the preprocessing of rule antecedents (see Section 2.7.1). The program involves two agents, one human and the other a robot, where the human would initiate a conversation with the robot. The robot would then attempt to answer the human by constructing a sentence from the simplified recursive grammar structure defined for an English sentence.

As expected from the flamegraph in Figure 4.3, the preprocessing of the rules (blocks on the left side) took about as much time needed as the goal tree resolution (blocks on the right side). The execution were also higher onto the stack as there is greater recursion in the simplified grammar defined.
Performance Comparison with SWI Prolog LPS Interpreter

The three selected LPS programs – Concurrent Towers, Goat River Crossing and Silly Dialogue – were also executed in SWI Prolog LPS interpreter\(^4\) to compare the execution times between the Prolog implementation and JS implementation. To measure the performance:

- Execution was done on the same machine
- SWI Prolog (threaded, 64 bits, version 7.6.4), lps.js 1.0.9 and lps.js 1.0.10 were used for performance measurement
- Each LPS program was executed and measured 30 times on each implementation
- Measurement for both interpreters included the initialisation of the program, execution of cycles and printing results to screen, but not loading of file from disk

Figure 4.4: The chart comparison execution times of program-platform pairs averaged over 30 runs.

Figure 4.4 shows the averaged execution time over 30 runs for each program-platform pair. The measurements showed that the generally execution in SWI Prolog using the Prolog LPS interpreter is significantly faster compared to execution in lps.js. As discussed earlier, bulk of the execution in most examples in lps.js were spent in expanding the goals.

4.2.3 Tic-Tac-Toe in LPS Studio

To demonstrate the P2P features of lps.js and the visualisation features on LPS Studio, a simple game of 2-player tic-tac-toe (T3) was implemented. The state diagram of the game is given in Figure 4.5.

\(^4\)https://bitbucket.org/lpsmasters/lps_corner
The program was implemented from a point-of-view of a single player in a multi-player context. This meant that the player programs are symmetric and would run the same communication protocol on top of the P2P protocol. The player programs have their individual worldview and would inform each other of changes by means of sending and receiving messages. The player, upon receiving messages, can choose to ignore or accept the information from external sources. This is also known as the distributed blackboard discussed in Section 2.9. In the case of the T3 example, messages are always accepted. Figure 4.6 shows the perspective of a single player in the implementation.

The OS of LPS made it very simple to program the game. In particular, only four reactive rules were needed to implement the game:

- If a second player joins the network, send a message to the that player to initiate the game, then start the game locally and allow the local player to make the first move.
- If an initiation message was received from another player in the network, start the game locally and designate the opponent to make the first move.
- If a grid was selected\(^5\) by means of clicking, update the local blackboard on the changes to the game state and update the opponent. Also check for any end game conditions, otherwise continue with the game with the other player’s turn.

\(^5\)Selection of a move is only allowed during the player’s turn.
• If an update message was received from the opponent, update the local blackboard with the changes from the message and check for any end game conditions. If no end game conditions were met, continue with local player’s turn.

Initially, the game was meant to demonstrate some of the advanced features. Two players were able to enter their moves through their respective windows of LPS Studio. Dr. Sadri saw the human vs human demonstration then suggested the implementation of a bot player. A bot player was implemented by extending the original program with an additional rule:

• If it is the local player’s turn (i.e. the bot player), then run a strategy to choose the most optimal choice and select the grid for making a move.

The strategy was implemented as a logic program where the bot would select a choice based on the following sequence:

1. If there is two Xs or Os in a row of the grid, place my choice in the empty slot. Placing that move either prevents the opponent from making the next move for a win, or causes the bot player to win.
2. If the center grid square is available, make move there.
3. If one of the corner grid squares are available, make move there.
4. Otherwise, make a move at any available square.

Figure 4.7: The T3 game being played with the human’s view on the left and the bot player’s view on the right. Cross and Circle icons by Icons8.com, CC BY-ND 3.0 License.

Figure 4.7 shows a game play between a human on the left and the bot player on the right, where the selection strategy of the bot comes into play. Subsequently, the bot player was made to play against itself, as seen in Figure 4.8.
4.2. EVALUATION

Chapter 4. Results and Evaluation

Figure 4.8: The T3 game being played with one instance of the bot player on the left and the another bot player on the right. Cross and Circle icons by Icons8.com, CC BY-ND 3.0 License.

As the strategy for the bot player is deterministic and can be seen to be more defensive than offensive, pitting the bot player against itself always results in a draw, rather than resulting in a win/lose scenario.

The successful and hassle-free implementation of tic-tac-toe in LPS using lps.js and LPS Studio demonstrated the various features in lps.js and LPS Studio using the operational semantics offered by LPS. The tic-tac-toe was also tested across two different machines communicating over network.

4.2.4 Limitations

Direct Usage in JS

At the time of writing, lps.js does not support the construction of LPS programs by JS without explicitly writing LPS code in order to reduce the amount of code base and API implementation needed. nools⁶, a Rete-based JS rules engine, supports such a feature and its users can write the rules in JS or in the nools language. It is attractive for JS programmers who wish to use a rules engine without having to learn another programming language.

Backtracking Across Cycles

lps.js was not able to solve a specific example given by Dr. Sadri that requires backtracking across time. The example is given in Code Snippet 4.9.

lps.js cannot resolve the only goal tree of the program as it involved the commitment to one path (first path of each cycle), then only to realise that the actions down that path later cannot be executed. Without looking ahead in time, it could not backtrack to a different grounding as it would once again retry the first possible path at the current time.

| 1 | maxTime(10). |
| 2 | action(a(_)). |
| 3 | action(b(_)). |

⁶https://github.com/noolajs/nools
Chapter 4. Results and Evaluation

4.2. EVALUATION

Code Snippet 4.9: The example has a constraint that disallow the execution of any b(1). Despite the path to execute the a(2) then b(2) becoming available as the alternative from time 2, the current search strategy continues to execute a(1) repeatedly in hope that eventually it would be able to execute b(1) later.

One possible solution to this example could be to take the path with the least amount of failures in the past. However, this involves the need to count the number of failures per possible grounding in the goal trees. The modification and experimentation could not be done within the time frame of the project.
Chapter 5

Conclusion

The project ended with the successful implementation of the LPS interpreter in JS language. Through the testing of LPS programs described in Section 4.2.1, lps.js proved to meet the model theoretic semantics specified by the LPS framework described in Kowalski and Sadri’s papers[2, 4]. lps.js was designed to be used in several ways as described in the User Guide (see Appendix E):

- Through the suite of command line tools in lps-cli package
- As a Node.js runtime library in lps package
- As a web browser runtime library bundled using the open source Webpack tool

Along with the implementation, the project has also revisited some of the previous ideas and brought about new ideas discussed in Chapter 3 with their results detailed in Chapter 4.

In this chapter, some of the challenges faced during the course of this project are documented, along with possible future work that can be extended from this project.

5.1 Challenges Faced

There were several challenges faced in the duration of this project and the implementation of lps.js.

5.1.1 Debugging

As this project involved the development of an interpreter, debugging was challenging as it was difficult to determine whether a bug that occurred was in the fault of the interpreter or in the LPS program under test. Debugging tools for LPS program being executed were barely available in the initial implementation of the lps.js since the focus was getting the features of the interpreter implemented. It took significant time and effort to debug, but ultimately preventive measures (such as using regression testing and detecting mistakes in LPS programs) were put in place to make sure bugs were identified quickly.

Some LPS programs produced very large goal trees which made tracing and debugging difficult. Without any visualisation of the actual goal trees being processed by lps.js, it may be difficult to explain and debug incorrect behaviour of the interpreter. To overcome this challenge, appropriate utility JS functions such as goal tree dumping was implemented to output the goal trees as
JavaScript Object Notation (JSON)\(^1\) files, then displayed using visualisation tools such as a JSON file viewer\(^2\).

### 5.1.2 Resources on Subject

Since LPS was still an active area of research through this project, there were limited amount of resources and reference available. Unlike an established or standardised programming language such as C, Java or even Prolog, the implementation of lps.js would change with new findings surfacing through the project progressed.

Prof. Kowalski, Dr. Sadri, Dr. Calejo et al. were very kind to make themselves available and being very patient with my questions and discussions through the project. As far as possible, it is with hope that this report, along with several other additional documentations that were written, has contributed as an additional resource for reference to future work on LPS.

### 5.1.3 Resources and open source software

In the development of lps.js and LPS Studio, several issues with regard to the usage of open source software (OSS) surfaced. As OSS tend to have little or expired documentation, most of the issues encountered had to be relied on debugging and experimentation to resolve the problems. Some of the issues included incompatibility of dependencies, security vulnerabilities or even lack of documentation on using the dependencies.

### 5.2 Future Work

lps.js opened many exciting doors as it brought glslps to a prominently used language, extending LPS’s reach to multiple platforms. This enables more programmers to conveniently bring AI into their existing implementation on various platform and languages through lps.js. In spite of the relatively short project time period, the completion of a LPS runtime in JS has paved for more opportunities and work in the future.

#### 5.2.1 LPS to JS Compiler Implementation

As mentioned in Section 3.1, lps.js – at the time of writing – is a runtime environment and interpreter. The alternative of implementing a source-to-source compiler that translates LPS code into JS code can potentially increase the execution speed of LPS programs in the JS environment directly. The compiler will need to embed a compact version of the lps.js runtime into the output JS code.

Since this is the first implementation of LPS framework in JS, a significant portion of time was spent studying the LPS framework and programs, and how to implement the concepts in an imperative language. It was not possible to implement a source-to-source compiler within the timeframe given for this project, but it would be pleasant to see such an implementation in future.

With such compiler implementation, it would also be very interesting to see LPS programs being embedded with JS functions directly within the same file, allowing both declarative and imperative paradigms in one single syntax.

\(^1\)https://www.json.org/
\(^2\)Such as http://jsonviewer.stack.hu/
5.2.2 Performance Improvements

As shown in Section 4.2.2, performance of lps.js can still be further improved to meet the performance offered by other implementations. There are several possible directions lps.js can take to meet this objective.

Porting Code to Lower-Level Compiled Languages

To further push the limit on the performance of lps.js, Node.js provides support for modules being written in C/C++ programming languages[58]. It is therefore possible to rewrite parts, if not all, of lps.js in C/C++ and be built using the node-gyp tool.

Nonetheless, moving any code to C/C++ language breaks portability of lps.js as browsers cannot execute compiled binaries received from the internet for security reasons. To overcome this, it may be possible to write certain parts – especially performance bottlenecks – of lps.js in C/C++, then implement polyfilling for those parts when bundling for browser. That way, it is possible to attain high performance in the command line execution while retaining portability for browser context.

Parallel Processing of Goal Resolution

Being a single-threaded language, JS does not offer access to the use of additional CPUs in the system. Consequent resolution of goal trees offer processing of tree branches that are independent of one another, which is ideal for parallel processing. It is possible that, with the porting of code to lower-level compiled language, multi-threading can be implemented for such optimisation.

Static Analysis & Optimisation of AST, Clauses and Rules

There remains possible further optimisations that can be applied to lps.js at the time of writing, by implementing additional static analysis and optimisation in the AST, on the clauses and rules to reduce actual computation during the LPS program execution.

GPU Acceleration

There are projects, such as gpu.js[^3], that enable JS programs in the browser context to tap on WebGL to access Graphics Processing Unit (GPU) acceleration. There are many parts of the interpreter that can be safely executed in parallel and delegating these executions to a GPU can help to increase processing speed when executing LPS programs in the browser context. However, these features are not available on Node.js runtime at the time of writing[^4].

5.2.3 Implementation of cut and fail

Prolog’s cut implementation helps to improve code efficiency by reducing the search space during backtracking[^59]. In lps.js, cut was not implemented and hence would exhaust all possible search space through backtracking. This can be observed with the implementation of max/3 in lps.js:

```prolog
1 max(X, Y, Z) <-
2   is_variable(Z),
3   Z = max(X, Y).
4 max(X, Y, Z) <-
```

[^3]: see [https://github.com/gpujs/gpu.js](https://github.com/gpujs/gpu.js)
[^4]: see [https://github.com/gpujs/gpu.js/issues/142](https://github.com/gpujs/gpu.js/issues/142)
5.2. FUTURE WORK

Chapter 5. Conclusion

5.2.4 Hibernation and Restoration

A key feature to most database systems is the ability to shutdown and then restore from a state. lps.js at the current moment does not save its current knowledge base and working memory to a secondary storage and data is lost as soon as the interpreter is terminated.

As lps.js moves towards production usage, the ability to backup and restore is very important. Having such facilities also enable LPS programs to restore from unexpected shutdown or errors.

The current LPS interpreter written in Prolog supports hibernation and restoration[60].

5.2.5 DOM Access and Manipulation Library

Tau Prolog, an implementation of Prolog in JS, provides a library for direct DOM access and manipulation[61]. Since lps.js is also implemented in JS and can be executed in the browser context, out-of-the-box support for DOM access and manipulation will be very helpful for web developers using LPS in the web apps.

5.2.6 LPS Studio: Extensions

LPS Studio uses HTML5 Canvas for rendering 2D graphics. It is possible for 3D graphics to be implemented in LPS Studio using WebGL in Electron⁵. New built-in predicates need to be defined for the LPS programmer to choose which mode to execute in and for accessing and manipulating objects in the 3D context.

Apart from 3D graphics, it is also possible to support sound and audio through the HTML5 audio specification[62]. By extending LPS Studio with these additional features, programmers can build creative, interactive and expressive LPS programs.

It is also possible to extend LPS Studio by implementing a chart and graph module in lps.js to draw interactive data visualisation in LPS Studio.

---

⁵As demonstrated by several open source implementations such as https://github.com/zamronypj/webgl-electron.
Bibliography


Appendices
# Appendix A
## Ethics Checklist

<table>
<thead>
<tr>
<th>Section 1: HUMAN EMBRYOS/FOETUSES</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does your project involve Human Embryonic Stem Cells?</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Does your project involve the use of human embryos?</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Does your project involve the use of human foetal tissues / cells?</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 2: HUMANS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Does your project involve human participants?</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 3: HUMAN CELLS / TISSUES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Does your project involve human cells or tissues? (Other than from “Human Embryos/Foetuses” i.e. Section 1)?</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 4: PROTECTION OF PERSONAL DATA</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Does your project involve personal data collection and/or processing?</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Does it involve the collection and/or processing of sensitive personal data (e.g. health, sexual lifestyle, ethnicity, political opinion, religious or philosophical conviction)?</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Does it involve processing of genetic information?</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Does it involve tracking or observation of participants? It should be noted that this issue is not limited to surveillance or localization data. It also applies to Wan data such as IP address, MACs, cookies etc.</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Does your project involve further processing of previously collected personal data (secondary use)? For example Does your project involve merging existing data sets?</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 5: ANIMALS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Does your project involve animals?</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 6: DEVELOPING COUNTRIES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Does your project involve developing countries?</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>If your project involves low and/or lower-middle income countries, are any benefit-sharing actions planned?</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Could the situation in the country put the individuals taking part in the project at risk?</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 7: ENVIRONMENTAL PROTECTION AND SAFETY</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Does your project involve the use of elements that may cause harm to the environment, animals or plants?</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Does your project deal with endangered fauna and/or flora /protected areas?</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
### Chapter A. Ethics Checklist

<table>
<thead>
<tr>
<th><strong>Does your project involve the use of elements that may cause harm to humans, including project staff?</strong></th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Does your project involve other harmful materials or equipment, e.g. high-powered laser systems?</strong></td>
<td>✓</td>
</tr>
</tbody>
</table>

**Section 8: DUAL USE**

<table>
<thead>
<tr>
<th><strong>Does your project have the potential for military applications?</strong></th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Does your project have an exclusive civilian application focus?</strong></td>
<td>✓</td>
</tr>
<tr>
<td><strong>Will your project use or produce goods or information that will require export licenses in accordance with legislation on dual use items?</strong></td>
<td>✓</td>
</tr>
<tr>
<td><strong>Does your project affect current standards in military ethics – e.g., global ban on weapons of mass destruction, issues of proportionality, discrimination of combatants and accountability in drone and autonomous robotics developments, incendiary or laser weapons?</strong></td>
<td>✓</td>
</tr>
</tbody>
</table>

**Section 9: MISUSE**

<table>
<thead>
<tr>
<th><strong>Does your project have the potential for malevolent/criminal/terrorist abuse?</strong></th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Does your project involve information on/or the use of biological-, chemical-, nuclear/radiological-security sensitive materials and explosives, and means of their delivery?</strong></td>
<td>✓</td>
</tr>
<tr>
<td><strong>Does your project involve the development of technologies or the creation of information that could have severe negative impacts on human rights standards (e.g. privacy, stigmatization, discrimination), if misapplied?</strong></td>
<td>✓</td>
</tr>
<tr>
<td><strong>Does your project have the potential for terrorist or criminal abuse e.g. infrastructural vulnerability studies, cybersecurity related project?</strong></td>
<td>✓</td>
</tr>
</tbody>
</table>

**Section 10: LEGAL ISSUES**

<table>
<thead>
<tr>
<th><strong>Will your project use or produce software for which there are copyright licensing implications?</strong></th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Will your project use or produce goods or information for which there are data protection, or other legal implications?</strong></td>
<td>✓</td>
</tr>
</tbody>
</table>

**Section 11: OTHER ETHICS ISSUES**

| **Are there any other ethics issues that should be taken into consideration?** | ✓ |
Appendix B

Ethical & Professional Considerations

This project was computer software design and development in nature, and hence there was no involvement of humans participants, biological material or dual use. The project also had no involvement in any form of data collection, use of harmful equipment or have any intention or goal towards any possible misuse.

However, as a software implementation project, it is unavoidable that pieces of software code written by third parties would need to be included and be reused by this project. It was necessary to study the licensing implications associated with the use of these third party software.

As far as possible, only open source software with compatible licensing were used in the implementation, but never intended to be packaged and distributed together with the software produced by this project. End users of the software produced by this project would need to individually agree with their installation and usage of these third party software.

Before software produced by this project became open-sourced, permission and the choice of open source license was obtained by writing from the original authors and collaborators of the framework.
Appendix C
Example LPS Programs

The following LPS programs are some that were translated from the Prolog implementation\(^1\), contributed by LPS contributors Miguel Calejo and Jacinto Dávila, to form the test suite for lps.js. All source code in this appendix are licensed under the BSD 3-Clause License.

C.1 rain.lps

A simple example that tests rules.

```
maxTime(20).
\[\text{action(rain(\_))}.\]
\[\text{true} \rightarrow \text{rain(1, 1, T2)}.\]
\[\text{rain}(T1, T1, T2) \rightarrow \text{rain}(T2, T2, T3).\]
```

C.2 fire-simple.lps

```
maxTime(5).
\[\text{fluent(fire)}.\]
\[\text{actions([\text{eliminate, escape}]}.\]
\[\text{initially(fire)}.\]
\[\text{fire}(T1) \rightarrow \text{deal_with_fire}(T1, T2).\]
\[\text{deal_with_fire}(T1, T2) \leftarrow \text{eliminate}(T1, T2).\]
\[\text{deal_with_fire}(T1, T2) \leftarrow \text{escape}(T1, T2).\]
\[\text{terminates(eliminate, fire)}.\]
```

C.3 fire-recurrent.lps

```
maxTime(10).
\[\text{fluents([fire, water}]}.\]
\[\text{actions([\text{eliminate, ignite(\_), escape, refill}]}.\]
\[\% \text{event(deal_with_fire)}.\]
\[\text{observe(ignite(\text{sofa}), 1, 2)}.\]
\[\text{observe(ignite(\text{bed}), 4, 5)}.\]
\[\text{observe(refill, 7)}.\]
```

\(^1\)https://bitbucket.org/lpsmasters/lps_corner
initially (water).
flammable (sofa).
flammable (bed).
fire (T1) -> deal_with_fire (T1, T2).
deal_with_fire (T1, T2) <- eliminate (T1, T2).
deal_with_fire (T1, T2) <- escape (T1, T2).
initiates (ignite (Object), fire) <- flammable (Object).
terminates (eliminate, fire).
terminates (eliminate, water).
initiates (refill, water).
<- eliminate, fire, not water.

C.4 binary-search.lps

maxTime (9).

% this program introduces the shorthand notation
% when declaring fluents, actions and events.
% it’d be great if programmers can indicate shorthand notation
% which is much easier.
% in lps.js, this is recognised as a division functor, i.e. “//2”,
% which can be specially handled by the declaration processors.
fluents ([
  left/1,
  right/1,
  searching/1
]).
actions ([
  initiate/1,
  terminate/1,
  update/1,
  result/2
]).

location (a, 0).
location (b, 1).
location (c, 2).
location (d, 3).

true -> find (c, T1, T2).
find (Content, T1, T2) <-
  initiate (left(0), T1, T2),
  initiate (right(4), T1, T2),
  initiate (searching (Content), T1, T2).
C.5  binary-search2.lps

```lps
searching(Content, T1),
  left(L, T1),
  right(R, T1),
  L < R,
  middle(M, T1),
  location(Item, M),
  Item @< Content ->
  NewL = M + 1,
  update(left(NewL), T2, T3).

searching(Content, T1),
  left(L, T1),
  right(R, T1),
  L < R,
  middle(M, T1),
  location(Item, M),
  !(Item @< Content) ->
  update(right(M), T2, T3).

searching(Content, T1),
  left(L, T1),
  right(R, T1),
  R <= L ->
  terminate(searching(Content), T2, T3),
  terminate(left(L), T2, T3),
  terminate(right(R), T2, T3),
  result(Content, L, T2, T3).

middle(M, T) <-
  right(R, T),
  left(L, T),
  M = round((R + L) / 2).

initiates(update(left(A)), left(A)).
initiates(update(right(A)), right(A)).
initiates(initiate(left(A)), left(A)).
initiates(initiate(right(A)), right(A)).

terminates(terminate(right(_)), right(_)).
terminates(terminate(left(_)), left(_)).

terminates(update(right(_)), right(_)).
terminates(update(left(_)), left(_)).

initiates(initiate(searching(C)), searching(C)).
terminates(terminate(searching(C)), searching(_)).
```

C.5  binary-search2.lps

```lps
fluents([]
  left(_Position),
```
% a( Position, Content)
a(0, 10).
a(1, 12).
a(2, 20).
a(3, 25).
a(4, 30).
a(5, 31).
a(6, 35).
a(7, 60).
a(8, 65).
a(9, 500).

found(I) at T <- left(I, T), right(I, T).
do_sample(T1, T2) <-
  left(L, T1),
  right(R, T1),
  Mid = floor((R + L) / 2),
  sample(Mid, T1, T2).

% This is a special case where found(I, T) is an intensional predicate
% that is expected to retry for each T value.
% it is nested inside the negation, which LPS needs to recognise and handle.
not found(I) at T -> do_sample(T1, T2).

updates(sample(Pos), left(_), left(Mid)) <-
  searching(X, T),
  a(Pos, AX),
  AX < X,
  Mid = Pos + 1.
updates(sample(Pos), right(_), right(Pos)) <-
  searching(X),
  a(Pos, AX),
  AX >= X.

This program tests the backtracking and built-in predicate member/2.
C.7 bubblesort.lps

```lps
maxTime(10).
fluent(location(X, Y)).
action(swapp(A, B, C, D)).
initially([ location(d, 1),
            location(c, 2),
            location(b, 3),
            location(a, 4) ]).
location(X, N1, T1), location(Y, N1 + 1, T1), Y Y @< X ->
swapped(X, N1, Y, N1 + 1, T2, T3).

% swapped does not work if the order of the two clauses below is
% reversed. Perhaps for good reasons,
% namely in the hope that positions will become swapped in the
```

C-5
% without the need to swap them explicitly.

\[
\text{swapped}(X, N_1, Y, N_2, T_1, T_2) \leftarrow \\
\text{location}(X, N_1, T_1), \\
\text{location}(Y, N_2, T_1), \\
Y @< X, \\
\text{swap}(X, N_1, Y, N_2, T_1, T_2).
\]

\[
\text{swapped}(X, N_1, Y, N_2, T, T) \leftarrow \\
\text{location}(X, N_1, T), \\
\text{location}(Y, N_2, T), \\
X @< Y.
\]

\[
\text{initiates}(\text{swap}(X, N_1, Y, N_2), \text{location}(X, N_2)).
\]

\[
\text{initiates}(\text{swap}(X, N_1, Y, N_2), \text{location}(Y, N_1)).
\]

\[
\text{terminates}(\text{swap}(X, N_1, Y, N_2), \text{location}(X, N_1)).
\]

\[
\text{terminates}(\text{swap}(X, N_1, Y, N_2), \text{location}(Y, N_2)).
\]

\[
\leftarrow \text{swap}(X, N_1, Y, N_2), \text{swap}(Y, N_2, Z, N_3).
\]

\[
% \leftarrow \text{swap}(X, N_1, Y, N_2), \text{swap}(A, N_3, B, N_4), X \neq A, Y \neq B.
\]

### C.8 map-colouring.lps

Testing constraints and choices over the selection of candidate actions.

maxTime(5).
cycleInterval(200).
action(paint(_, _)).
country(iz).
country(oz).
country(az).
country(uz).
colour(red).
colour(yellow).
colour(blue).
adjacent(az, iz).
adjacent(az, oz).
adjacent(iz, oz).
adjacent(iz, uz).
adjacent(oz, uz).
country(X) -> colour(C), paint(X, C, T_1, T_2).
\[
\leftarrow \text{paint}(X, C), \text{adjacent}(X, Y), \text{paint}(Y, C).
\]
C.9 mark.lps

A program implemented after a GIF animation/video of Mark Zuckerberg’s bizarre drinking water behaviour\(^2\) – during the Cambridge Analytica testification before United States Senate – went viral on the internet. The program tests sequential execution of actions of a single plan.

```lps
# Watch this: https://i.imgur.com/DYNIfoG.mp4
maxTime(10).
fluents([thirsty(_), empty(_), drinking(_)]).
actions(
  reach(P, C, H),
  grab(P, C, H),
  raise(P, C, H),
  sip(P, C),
  putdown(P, C),
  refill(C)
).
initially(thirsty(mark)).
observe(refill(glass), 7).
hand(mark, right).
hand(mark, left).
location(glass, right).
container(glass).
container(mug).

thirsty(P, T) -> container(C), drink(P, C, T, T2).

\[\text{drink(Person, C, \text{T1}, \text{T6}) <-}
  \begin{align*}
  & \text{hand(Person, H),} \\
  & \text{location(C, H),} \\
  & \text{reach(Person, C, H, \text{T1}, \text{T2}),} \\
  & \text{grab(Person, C, H, \text{T2}, \text{T3}),} \\
  & \text{raise(Person, C, H, \text{T3}, \text{T4}),} \\
  & \text{sip(Person, C, \text{T4}, \text{T5}),} \\
  & \text{putdown(Person, C, \text{T5}, \text{T6}).}
  \end{align*}\]

# Sets up condition for drinking to prevent drinking twice
initiates(reach(Person, C, H), drinking(Person)).
terminates(sip(Person, C), drinking(Person)).

terminates(sip(Person, C), thirsty(Person)).
initiates(sip(Person, C), empty(C)).

terminates(refill(C), empty(C)).

<= reach(Person, C, H), drinking(P).
<= sip(Person, C), empty(C).
```

\(^2\)https://i.imgur.com/DYNIfoG.mp4
C.10 mark-hiccup.lps

An extension from the mark.lps program where lps.js was tested to see if it could identify the execution of composite events/macros through observations.

```prolog
# Watch this: https://i.imgur.com/DYNIfG.mp4
maxTime(8).
fluents([thirsty(_), empty(_), drinking(_)]).
actions([reach(_, _, _),
         grab(_, _, _),
         raise(_, _, _),
         sip(_, _),
         putdown(_, _),
         refill(_, 
         hiccup(_)]).
initially(thirsty(mark)).
observe(refill(glass), 7).
observe(reach(mark, glass, right), 1).
observe(grab(mark, glass, right), 2).
observe(raise(mark, glass, right), 3).
observe(sip(mark, glass), 4).
observe(putdown(mark, glass), 5).
hand(mark, right).
hand(mark, left).
location(glass, right).
container(glass).
container(mug).
drink(P, C, T1, T2) -> hiccup(P, T2, T4).
% thirsty(P, T) -> container(C), drink(P, C, T, T2).
drink(Person, C, T1, T6) <-
   hand(Person, H),
   location(C, H),
   reach(Person, C, H, T1, T2),
   grab(Person, C, H, T2, T3),
   raise(Person, C, H, T3, T4),
   sip(Person, C, T4, T5),
   putdown(Person, C, T5, T6).
# Sets up condition for drinking to prevent drinking twice
initiates(reach(Person, C, H), drinking(Person)).
terminates(sip(Person, C), drinking(Person)).
terminates(sip(Person, C), thirsty(Person)).
initiates(sip(Person, C), empty(C)).
terminates(refill(C), empty(C)).
```

C.11  trash.lps

maxTime(10).
fluenfs([locked(_), trash(_), bin(_)]).
action(dispose(X, Y)).
action(unlock(_)).
initially([locked(container1), trash(bottle1), bin(container1),
         bin(container2)]).
observe(unlock(container1), 4, 5).
trash(Object, T1) -> bin(Container, T1), dispose(Object, Container,
         T2, T3).
terminates(unlock(Container), locked(Container)).
terminates(dispose(Object, X), trash(Object)).
<- dispose(_, Container), locked(Container).

C.12  quickSort.lps

maxTime(3).
action(request(_)).
action(announce(_)).
cycleInterval(200).
observe(request(sort([2, 1, 4, 3])), 1, 2).
request(sort(X), T1, T2) ->
quicksort(X, Y),
announce(sorted(Y), T2, T3).
quicksort([X|Xs], Ys) <-
  partition([X|Xs], Y, Left, Right),
quicksort(Left, Ls),
quicksort(Right, Rs),
append(Ls, [X|Rs], Ys).
quicksort([], []).
partition([X|Xs], Y, L, Rs) <-
  X <= Y, partition(Xs, Y, Ls, Rs), append([X], Ls, L).
partition([X|Xs], Y, Ls, R) <-
C.13. **DINING.LPS**

Chapter C. Example LPS Programs

```
X > Y, partition(Xs, Y, Ls, Rs), append([X], Rs, R).
partition([], Y, [], []).  
```

C.13 **dining.lps**

The dining philosophers’ problem. Demonstrates LPS’s capability to work around constraints given.

```
maxTime(7).

fluent(available(_)).

actions([
  pickup(_, _),
  putdown(_, _)
]).

initially([
  available(fork1),
  available(fork2),
  available(fork3),
  available(fork4),
  available(fork5)
]).

philosopher(socrates).
philosopher(plato).
philosopher(aristotle).
philosopher(hume).
philosopher(kant).

adjacent(fork1, socrates, fork2).
adjacent(fork2, plato, fork3).
adjacent(fork3, aristotle, fork4).
adjacent(fork4, hume, fork5).
adjacent(fork5, kant, fork1).

philosopher(P) -> dine(P, T1, T2).

dine(P, T1, T3) <-
  adjacent(F1, P, F2),
  pickup(P, F1, T1, T2),
  pickup(P, F2, T1, T2),
  putdown(P, F1, T2, T3),
  putdown(P, F2, T2, T3).

terminates(pickup(P, F), available(F)).
initiates(putdown(P, F), available(F)).

<- pickup(P, F), not available(F).
<- pickup(P1, F), pickup(P2, F), P1 != P2.
```
C.14   rock-paper-scissors-minimal.lps

% these could be inferred:
events([  
  player_input(_Sender,_Choice,_Value)  
]).

actions([  
  send(_Winner,_Prize)  
]).

fluents([  
  reward(_),  
  played(_Sender,_Choice)  
]).

% These or others could be suggested:
observe(player_input(miguel, paper, 1000), 2, 3).
observe(player_input(fariba, rock, 1000), 2, 3).
observe(player_input(bob, rock, 1000), 3, 4).

beats(scissors, paper). % scissors beats paper, ....
beats(paper, rock).
beats(rock, scissors).

initially(reward(0)). % initially the reward is 0

% when a player ....
initiates(player_input(Player, Choice, Value), played(Player, Choice)).

% a play with a value increases the reward by that value
updates(player_input(Player, Choice, Value), reward(Old), reward(Old + Value)).

num_players(N) <- findall(P, played(P, _), L), length(L, N).

% Players must bet a positive amount
<- player_input(_, _, Value), Value <= 0.

% If a player has played, he can not play again:
<- player_input(Sender, Choice, Value), played(Sender, _).

% a play is forbidden if the number of players becomes greater than 2:
<- player_input(Sender, Choice, Value), num_players(N), N > 1.

% if a player’s choice beats the other’s, send him the whole reward
played(Player1, Choice1, T1),
played(Player2, Choice2, T1),
Player1 != Player2,
beats(Choice1, Choice2),
reward(R, T1),
R > 0 ->
  send(Player1, R, T2, T3).
% if both players choose the same choice, send half of the reward to each
played(Player1, Choice, T1),
played(Player2, Choice, T1),
Player1 != Player2,
reward(R, T1),
R > 0 ->
  % split prize
  Prize = R / 2,
  send(Player1, Prize, T2, T3),
  send(Player2, Prize, T2, T3).

% After sending prizes terminate the contract
% if send(_, _Prize) to T2
% then lps_terminate from T2.

% When sending a prize subtract it from the reward
updates(send(Player, Prize), reward(Old), reward(Old - Prize)).

% after sending a prize the reward must not become negative
<- send(_, P), reward(V), V < 0.

% It is impossible to send zero
<- send(_, 0).

C.15 Fire Station - LPS Studio Example

This example program has 2 parts to it and is designed for execution in LPS Studio mentioned in Section 3.14 of the report.

C.15.1 fire-example.lps

The first part is the main program that was modified from the fire-recurrent.lps example. The program simulates firefighters being dispatched from a fire station to fight fire at two locations: the sofa and the grill. The goal of the firemen is to put out any fire as long as there is fire outside the station and would only return to station when there is no more fire. The program was modelled with greater details compared to the original fire-recurrent.lps example as some of these details are needed by the visualisation.

This file can be executed as a standalone LPS program without the visualisation.

maxTime(100).
cycleInterval(1000).

fluents([fire(I), location(L)]).

actions([deployFirefighters,
moveFirefightersTo(_),]
Chapter C. Example LPS Programs

C.15. FIRE STATION - LPS STUDIO EXAMPLE

```lps
putOutFire(_),
ignite(_),
returnToStation )).

flammable(sofa).
flammable(grill).

initially([
  location(fireStation)
]).

fire(Item, T1),
  location(fireStation, T1) ->
    deal_with_fire(Item, T1, T2).

deal_with_fire(Item, T1, T3) <-
  location(fireStation, T1),
  deployFirefighters(T1, T2),
  handleItemWithFire(Item, T2, T3).

deal_with_fire(Item, T1, T3) <-
  not location(fireStation, T1),
  handleItemWithFire(Item, T1, T3).

% base case for deal_with_fire, that is no more fire and we return to fire station
    deal_with_fire(_, T1, T2) <-
      not fire(_, T1),
      returnToStation(T1, T2).

handleItemWithFire(Item, T1, T3) <-
  fire(Item, T1),
  location(Item, T1),
  putOutFire(Item, T1, T2),
  deal_with_fire(_, T2, T3).

handleItemWithFire(Item, T1, T3) <-
  fire(Item, T1),
  not location(Item, T1),
  moveFirefightersTo(Item, T1, T2),
  deal_with_fire(Item, T2, T3).

initiates(ignite(Object), fire(Object)) <-
  flammable(Object).

terminates(putOutFire(I), fire(I)).
updates(deployFirefighters, location(_), location(otw)).
updates(moveFirefightersTo(L), location(_), location(L)).
updates(returnToStation, location(_), location(fireStation)).

% cannot put out fire if not at the location
    <- not location(X),
    putOutFire(X).
```

C.15. FIRE STATION - LPS STUDIO EXAMPLE

C.15.2 fire-example-studio.lps

The second part of the program contains the LPS code needed to LPS Studio visualisation and interaction. When in LPS Studio, this file is loaded instead. The program ignites any flammable objects when the object was clicked on and changes to the state of the LPS program are also updated on the visualisation by this program.

```prolog
% load main program
consult('./ fire-example.lps').

lpsLoadImage(fire, 'https://png.icons8.com/cotton/64/gas-industry.png').
lpsLoadImage(fireTruck, 'https://png.icons8.com/cotton/64/fire-truck.png').
lpsLoadImage(fireStation, 'https://png.icons8.com/cotton/64/fire-station.png').
lpsLoadImage(sofa, 'https://png.icons8.com/cotton/64/living-room.png').
lpsLoadImage(grill, 'https://png.icons8.com/cotton/64/grill.png').

% define object positions
positionItem(sofa, 300, 100).
positionItem(grill, 420, 100).
positionItem(fireStation, 50, 100).
positionItem(otw, 50, 100).
fluent(fireTruckDirection(D)).
initially(fireTruckDirection(right)).
lpsClick(I, X, Y, T1, T2),
flammable(I) ->
  ignite(I, T2, T3).

% draw fire station, sofa and grill
lpsDefineObject(I, image, [
  image(I),
  position(X, Y),
  size(64, 64)
]) <-
positionItem(I, X, Y),
member(I, [fireStation, sofa, grill])

% draw fire truck
```
lpsDefineObject(fireTruck, image, [  
    image(fireTruck),  
    position(X, Y),  
    size(64, 64),  
    isHidden(1)  
]) <-  
    positionItem(fireStation, X, Y).

% draw fire for grill  
lpsDefineObject(grillFire, image, [  
    image(fire),  
    position(X, Y),  
    size(48, 48),  
    isHidden(1)  
]) <-  
    positionItem(grill, X1, Y1),  
    X = X1 + 12,  
    Y = Y1 + 12.

% draw fire for sofa  
lpsDefineObject(grillFire, image, [  
    image(fire),  
    position(X, Y),  
    size(48, 48),  
    isHidden(1)  
]) <-  
    positionItem(grill, X1, Y1),  
    X = X1 + 12,  
    Y = Y1 + 12.

lpsDefineObject(sofaFire, image, [  
    image(fire),  
    position(X, Y),  
    size(48, 48),  
    isHidden(1)  
]) <-  
    positionItem(sofa, X1, Y1),  
    X = X1 + 12,  
    Y = Y1 + 12.

deployFirefighters(T1, T2) ->  
    lpsShowObject(fireTruck) from T2.

% when moving firefighter, move fire truck  
moveFirefightersTo(Item, T1, T2) ->  
    positionItem(Item, X, Y),  
    lpsAnimateMoveObject(fireTruck, 200, X, Y).

location(Current, T1),  
    fireTruckDirection(Dir, T1),  
moveFirefightersTo(Item, T1, T2) ->  
    positionItem(Current, X1, _),  
    positionItem(Item, X, Y),  
    checkTruckFlipNeeded(X1, X, Dir, T3, T4),  
    lpsAnimateMoveObject(fireTruck, 200, X, Y).
C.15. FIRE STATION - LPS STUDIO EXAMPLE

Chapter C. Example LPS Programs

% These clauses determine if the truck needs to flip its direction when moving to somewhere else, depending on its previous position
checkTruckFlipNeeded(X1, X2, Dir, T1, T2) :-
    X1 < X2,
    Dir == left,
    changeDirection(T1, T2),
    lpsUpdateObject(fireTruck, [
        flipHorizontal(0)
    ]).

checkTruckFlipNeeded(X1, X2, Dir, T1, T2) :-
    X1 >= X2,
    Dir == left.

checkTruckFlipNeeded(X1, X2, Dir, T1, T2) :-
    X1 > X2,
    Dir == right,
    changeDirection(T1, T2),
    lpsUpdateObject(fireTruck, [
        flipHorizontal(1)
    ]).

checkTruckFlipNeeded(X1, X2, Dir, T1, T2) :-
    X1 <= X2,
    Dir == right.

% animate return to station
location(Current, T1),
  fireTruckDirection(Dir, T1),
  returnToStation(T1, T2) ->
    positionItem(Current, X1, _),
    positionItem(fireStation, X, Y),
    checkTruckFlipNeeded(X1, X, Dir, T2, T3),
    lpsAnimateMoveObject(fireTruck, 200, X, Y) from T2 to T4,
    lpsUpdateObject(fireTruck, [
        isHidden(1)
    ]) from T4.

% show fire when ignited
ignite(sofa, T2, T) ->
    lpsShowObject(sofaFire).

ignite(grill, T2, T) ->
    lpsShowObject(grillFire).

% hide fire when put out
putOutFire(sofa, T1, T2) ->
    lpsHideObject(sofaFire).

putOutFire(grill, T1, T2) ->
    lpsHideObject(grillFire).

updates(changeDirection, fireTruckDirection(_), fireTruckDirection(left)) :-
    fireTruckDirection(right, _).
C.16  Bubble Sort - LPS Studio Example

This example Bubble Sort algorithm demonstration program for LPS Studio has three parts. This program was modified from the original bubbleSort.lps program.

C.16.1  bubblesort-studio-main.lps

The first part of the program defines the environment. It can also be executed independently as a LPS program without the visualisation.

```lps
maxTime(500).
cycleInterval(500).
fluent(locked(Item)).
fluent(location(Item, Position)).
action(lock(Item1, Item2)).
action(unlock(Item1, Item2)).
action(swap(Item1, Pos1, Item2, Pos2)).
initially([
  location(d, 1),
  location(c, 2),
  location(b, 3),
  location(a, 4),
  location(e, 6),
  location(f, 5)
]).
initiates(swap(X, N1, Y, N2), location(X, N2)).
initiates(swap(X, N1, Y, N2), location(Y, N1)).
terminates(swap(X, N1, Y, N2), location(X, N1)).
terminates(swap(X, N1, Y, N2), location(Y, N2)).
initiates(lock(Item1, Item2), locked(Item1)).
initiates(lock(Item1, Item2), locked(Item2)).
terminates(unlock(Item1, Item2), locked(Item1)).
terminates(unlock(Item1, Item2), locked(Item2)).
<- swap(X, N1, Y, N2), swap(Y, N2, Z, N3).
<- lock(X, Y), lock(Y, Z).
<- lock(X, Y), locked(X).
<- lock(X, Y), locked(Y).
% Uncomment the following constraint to ensure only one
% pair of items are being swapped at any given time.
% <- swap(X, N1, Y, N2), swap(A, N3, B, N4), X != A, Y != B.
```

C-17
% Consult the sorter rules and clauses
consult('./bubblesort-studio-sorter.lps').

C.16.2 bubblesort-studio-sorter.lps

The second part of the program contains solely the beliefs and goals of a sorter agent who wishes to keep the items in sorted order.

```lps
location(X, N1, T1),
location(Y, N1 + 1, T1),
not locked(X, T1),
not locked(Y, T1),
Y @< X ->
decideSwap(X, N1, Y, N1 + 1) from T1 to T2.

% to swap
decideSwap(X, N1, Y, N2) from T1 to T5 <-
location(X, N1, T1),
location(Y, N2, T1),
Y @< X,
lock(X, Y, T1, T2),
swap(X, N1, Y, N2, T2, T3),
T4 = T1 + 5,
unlock(X, Y, T4, T5).

% or not to swap
decideSwap(X, N1, Y, N2) from T to T <-
location(X, N1, T),
location(Y, N2, T),
X @< Y.

% that is the question
% William Shakespeare’s Hamlet, Act III, Scene I
```

C.16.3 bubblesort-studio.lps

The third part of the program contains the LPS code needed to LPS Studio visualisation and interaction. When in LPS Studio, this file is loaded. When a mouse down event occurs, LPS Studio will wait for enable_drag/1 to be called before the dragging starts. In the event that the item is locked, the dragging is not enabled until the item is unlocked.

The user can aid or hinder the sorter agent in the sorting process.

```lps
consult('./main.lps').
event(decideSwapLocation(I, X, Y)).
locationScreen(1, 100, 100).
locationScreen(2, 200, 100).
locationScreen(3, 300, 100).
```
Chapter C. Example LPS Programs

C.16. BUBBLE SORT - LPS STUDIO EXAMPLE

8 locationScreen(4, 400, 100).
9 locationScreen(5, 500, 100).
10 locationScreen(6, 600, 100).
11
12 lpsDefineObject(_, text, [
    position(10, 20),
    caption("Program authored by Sam Yong. Icons by Icons8.com, CC-BY-ND 3.0"),
    font('9px Helvetica, sans-serif')
]).
13
14 lpsDefineObject(Item, image, [
    image(Item),
    size(64, 64),
    position(-32, -32)
]) <- lpsLoadImage(Item, _).
15
16 location(Item, Loc, 1) ->
  locationScreen(Loc, X, Y),
  lpsUpdateObject(Item, [
    position(X, Y)
  ]) from 1 to T2.
17
18 swap(A, J, B, K, T1, T2) ->
  locationScreen(J, X1, Y1),
  locationScreen(K, X2, Y2),
  lpsAnimateMoveObject(A, 400, X2, Y2) from T2,
  lpsAnimateMoveObject(B, 400, X1, Y1) from T2.
19
20 lpsMouseDown(I, X, Y, T1, T2),
   not locked(I, T2) ->
   lpsUpdateObject(I, [isDragEnabled(1)]).
21
22 lpsDragRelease(I, X, Y, T1, T2) ->
   decideSwapLocation(I, X, Y, T2, T3).
23
24 nearestOtherLocation(X, Y, OtherLoc) <-
  locationScreen(OtherLoc, SX, SY),
  abs(SX - X) < 60,
C.17 Lending and Borrowing MAS - Single Program

This example takes a scenario of two agents in the environment. Each of them have some money, one more than the other. The one who do not have enough money – alice – have the desire to buy something but do not have enough money. Alice would have to try to lend from Bob who has enough money. Fortunately, Bob is very kind to lend Alice some money to make up the difference for the item Alice wishes to buy.

C.17.1 main.lps

This is the main program to be launched. It contains the blackboard for both agents, as well as appropriate constraints for the environment.

```
maxTime(8).
fluent(money(Agent, Amount)).
action(borrow(Agent1, Agent2, Amount)).
action(lend(Agent1, Agent2, Amount)).
action(buy(Agent, Item)).
event(wantToBuy(Agent, Item)).
itemPrice(cereal, 4).
itemPrice(potatos, 1).
itemPrice(milk, 1).
initially([
  money(alice, 1),
  money(bob, 10)
]).
observe(wantToBuy(alice, cereal), 3, 4).
% % ---------------------
```
% This section would be better modelled in the agent file
% but because of the duplication issues, I'm leaving it here for now.

loanIfNotEnough(Agent, Price, Amount, T, T) <-
  Price <= Amount.

loanIfNotEnough(Agent, Price, Amount, T1, T2) <-
  Price > Amount,
  consult(_, Peer),
  % Peer != Agent,
  borrow(Agent, Peer, Price - Amount, T1, T2).

% You cannot buy unless you have the money for the item
<-
  buy(Agent, Item, T1, T2),
  money(Agent, Amt, T1),
  itemPrice(Item, Price),
  Price > Amt.

% You cannot lend unless you have the money
<-
  lend(Lender, Borrower, Amount, T1, T2),
  money(Lender, Balance),
  Balance < Amount.

% you cannot borrow or lend yourself
<-
  borrow(A, A, _, _, _).
<-
  lend(A, A, _, _, _).

updates(lend(Lender, Borrower, Amount),
  money(Lender, Balance),
  money(Lender, Balance - Amount)).

updates(lend(Lender, Borrower, Amount),
  money(Borrower, Balance),
  money(Borrower, Balance + Amount)).

updates(buy(Agent, Item),
  money(Agent, Balance),
  money(Agent, Balance - Price)) <-
  itemPrice(Item, Price).

consult('./agent.lps', alice).
consult('./agent.lps', bob).

C.17.2 agent.lps

This is the agent program. It says that whenever the agent has the desire to buy something, the agent will try to buy it. If the agent does not have enough money, it will try to find someone to borrow from.
C.18 Lending and Borrowing MAS - P2P-based Programs

Using the same scenario of lending and borrowing, this MAS consists of two programs that uses the lps P2P module to communicate over network. The LPS P2P tracker needs to start running before either of the programs can start running.

C.18.1 bob.lps

This is the Bob's agent program. Bob has a private blackboard and all other programs can only interact with the agent program through message passing and event observations. In this program, Bob is kind enough to lend the amount anyone requests for.

```prolog
maxTime(1000).
loadModule(p2p).
p2pJoin(lending, 4100).
fluent(money(A)).
initially(money(10)).
action(lend(A)).

% observe(p2pReceive(lending, node(1, 2), borrow(10)), 2, 3).

p2pReceive(lending, P, borrow(A), T1, T2) ->
  lend(A, T2, T3),
  p2pSend(lending, P, lend(A)).

updates(lend(Amt), money(A), money(A - Amt)).
```

C.18.2 alice.lps

This is the Alice's agent program. Alice also has a private blackboard and all other programs can only interact with the agent program through message passing and event observations. In this program, Alice wishes to buy something and will look around for peers to borrow money from.

```prolog
maxTime(8).
loadModule(p2p).
p2pJoin(lending, 4100).

event(wantToBuy(I)).

fluent(money(A)).
action(buy(I)).
action(receiveLoan(A)).

initially(money(1)).
observe(wantToBuy(cereal), 3, 4).

itemPrice(cereal, 4).
```
Chapter C. Example LPS Programs

LENDING AND BORROWING MAS - P2P-BASED PROGRAMS

18  itemPrice(potatoes, 1).
19  itemPrice(milk, 1).
20  
21  wantToBuy(I, T1, T2) ->
22      itemPrice(I, Price),
23      money(Amount, T2),
24      loanIfNotEnough(Price, Amount),
25      buy(I, T3, T4),
26      T3 >= T2.
27  
28  loanIfNotEnough(Price, Amount) <-
29      Price <= Amount.
30  
31  loanIfNotEnough(Price, Amount) <-
32      Price > Amount,
33      p2pPeer(lending, P),
34      Loan = Price - Amount,
35      p2pSend(lending, P, borrow(Loan)).
36  
37  p2pReceive(lending, P, lend(A), T1, T2) -> receiveLoan(A, T2, T3).
38  
39  updates(receiveLoan(Loan), money(A), money(A + Loan)).
40  updates(buy(I), money(A), money(A - P)) <- itemPrice(I, P).
41  
42  <- money(Amt, T1), buy(I, T1, T2), itemPrice(I, Price), Amt < Price.
Appendix D

lps.js Built-in Predicates Reference

This document lists the set of built-in declarations, predicates and functions provided by lps.js in the core module. Do note that:

- Declarations are meant to be written as facts in the program. They can also be defined as head terms of clauses. It is imperative that `loadModule/1`, `consult/1` and `consult/2` are not defined as head of clauses whose body conjuncts rely on yet-to-be loaded facts.

- Predicates are used as conjuncts in a conjunction.

- Functions are used in arguments of predicates, comparators, assignment or mathematical evaluations.

D.1 Declarations

Some predicates can be written to give lps.js some understanding on the interpretation of the LPS program. Typically they are only queried once before the execution of the first cycle. It is possible to use these predicates as the head of a clause.

`maxTime/1` The auxiliary fact `maxTime(+Cycles)` indicates how many cycles the LPS program can run before the program halts. The argument `+Cycles` must be a positive integer number. By default, the maxTime value is set to 20 in the absence of this declaration.

`cycleInterval/1` The auxiliary fact `cycleInterval(+Time)` indicates how much time a cycle is permitted to execute. The argument `+Time` must be a positive integer number in real time clock milliseconds. If the execution of a cycle is longer than this value, lps.js will halt the execution and raise an error. By default, the cycle interval is set to 100ms.

`continuousExecution/1` The auxiliary fact `continuousExecution(+Bool)` indicates how much time a cycle is permitted to execute. The argument `+Bool` must be either '1' or 0. If '1' is indicated, lps.js will eagerly execute the next cycle as soon as one ends. Otherwise, lps.js will respect the cycle interval set. By default, continuous execution mode is turned off.

`fluent/1` The auxiliary fact `fluent(+Fluent)` indicates to lps.js that the predicate in `+Fluent` is a fluent. For example, if the predicate `value(Amount)` is a fluent, then the programmer needs to indicate so by writing `fluent(value(Amount))` as a fact in the LPS program. It is also possible to use the shorthand version `fluent(value/1)`.

`fluent/1` The auxiliary fact `fluent(+List)` indicates to lps.js that all predicates in `+List` are fluents. For example, if the predicate `value(Amount)` and `quantity(I, Q)` are fluents, then the programmer can indicate so by writing `fluent([value(Amount), quantity(I, Q)])` as a fact in the LPS program as an alternative to `fluent/1`. Likewise, it's possible to use shorthand indication in the list `fluent([value/1, quantity/2])`.
action/1  The auxiliary fact \texttt{action(+Action)} indicates to lps.js that the predicate in \texttt{+Action} is an action. For example, if the predicate \texttt{run(From, To)} is an action, then the programmer needs to indicate so by writing \texttt{action(run(From, To))} as a fact in the LPS program. It is also possible to use the shorthand version \texttt{action(run/2)}.

actions/1  The auxiliary fact \texttt{actions(+List)} indicates to lps.js that all predicates in \texttt{+List} are actions. For example, if the predicate \texttt{run(From, To)} and \texttt{walk(From, To)} are actions, then the programmer can indicate so by writing \texttt{actions([run(From, To), walk(From, To)])} as a fact in the LPS program as an alternative to \texttt{action/1}. Likewise, it’s possible to use shorthand indication in the list \texttt{actions([run/2, jump/2])}.

event/1  The auxiliary fact \texttt{event(+Event)} indicates to lps.js that the predicate in \texttt{+Event} is an event. For example, if the predicate \texttt{receive(From, Item)} is an event, then the programmer needs to indicate so by writing \texttt{event(receive(From, Item))} as a fact in the LPS program. It is also possible to use the shorthand version \texttt{event(receive/2)}.

events/1  The auxiliary fact \texttt{events(+List)} indicates to lps.js that all predicates in \texttt{+List} are events. For example, if the predicate \texttt{receive(From, Item)} and \texttt{newItem(Item)} are events, then the programmer can indicate so by writing \texttt{receive(From, Item), newItem(Item)}] as a fact in the LPS program as an alternative to \texttt{event/1}. Likewise, it’s possible to use shorthand indication in the list \texttt{events([receive/2, newItem/1])}.

observe/2  The auxiliary fact \texttt{observe(+Event, +Time)} indicates to lps.js that the LPS program should observe an event \texttt{+Event} at the specified time \texttt{+Time}. This is a shorthand for \texttt{observe/3} where the end time is defined to be one after \texttt{+Time}.

observe/3  The auxiliary fact \texttt{observe(+Event, +StartTime, +EndTime)} indicates to lps.js that the LPS program should observe an event \texttt{+Event} from the specified start time \texttt{+StartTime} until the specified end time \texttt{+EndTime}. The event is not observed at the \texttt{+EndTime}.

initially/1  The auxiliary fact \texttt{initially(+Fluent)} indicates to lps.js that the fluent \texttt{+Fluent} holds from the start of the LPS program execution. \texttt{+Fluent} can also be a list of fluents.

loadModule/1  The auxiliary fact \texttt{loadModule(+Module)} indicates to lps.js which additional modules need to be loaded for the LPS program before execution starts. It can be used to either load a builtin module or a JavaScript file that describes a lps.js module. The following is the currently available builtin modules:

- \texttt{fs} - File system access module
- \texttt{p2p} - Peer to peer communication module

consult/1  The auxiliary fact \texttt{consult(+File)} indicates to lps.js which additional LPS files need to be loaded and augmented for the main LPS program before execution starts.

consult/2  The auxiliary fact \texttt{consult(+File, +ProcessId)} indicates to lps.js which additional LPS files need to be loaded and augmented for the main LPS program before execution starts. Before the additional LPS files are loaded and augmented, rules, clauses and constraints in those files are pre-processed with the given \texttt{+ProcessId} using the builtin predicate \texttt{processId/1}. 
D.2 Meta

!/1 The negation predicate uses negation as failure. The "not" keyword is ultimately executed as !/1. Given !(+Goal), the predicate succeeds if +Goal cannot be proven, and fails if +Goal can be proven.

=/>2 The assignment predicate Var = Expr assigns the value evaluated from +Expr to the variable Var.

functor/3 The predicate functor(+Term, ?Name, ?Arity) allows the retrieval and verification of the name and arity of a given term.

lpsHalt/0 When the predicate is executed, the interpreter halts execution of the LPS program. Since the halt is requested in the middle of a cycle, the interpreter stops after the completion of the current cycle.

lpsArgs/1 The predicate lpsArgs(-List), provided by lps.js, provides the list of program arguments. The sole argument provided is always a list.

D.3 Comparators

/>2 The greater than predicate ExprA > ExprB takes two operands ExprA and ExprB. Succeeds if ExprA evaluates to be a number larger than ExprB and fails otherwise.

>=/>2 The greater than or equal predicate ExprA >= ExprB takes two operands ExprA and ExprB. Succeeds if ExprA evaluates to be a number larger than or equal to ExprB and fails otherwise.

</2 The less than predicate ExprA < ExprB takes two operands ExprA and ExprB. Succeeds if ExprA evaluates to be a number smaller than ExprB and fails otherwise.

<=/>2 The less than or equal predicate ExprA <= ExprB takes two operands ExprA and ExprB. Succeeds if ExprA evaluates to be a number smaller than or equal to ExprB and fails otherwise.

==/>2 The equality predicate ExprA == ExprB takes two operands ExprA and ExprB. Succeeds if ExprA evaluates to be the same value as ExprB and fails otherwise.

!=/>2 The inequality predicate ExprA != ExprB takes two operands ExprA and ExprB. Succeeds if ExprA evaluates to be not the same value as ExprB and fails otherwise.

@/>2 The greater standard order of terms predicate ExprA @> ExprB takes two operands ExprA and ExprB. Succeeds if ExprA evaluates to be greater on the standard order of terms compared to ExprB and fails otherwise.

@=/>2 The equal standard order of terms predicate ExprA @= ExprB takes two operands ExprA and ExprB. Succeeds if ExprA evaluates to be equal on the standard order of terms compared to ExprB and fails otherwise.
D.4 Math and Arithmetic

@</2 The less standard order of terms predicate $\text{ExprA} \heartsuit \text{ExprB}$ takes two operands $\text{ExprA}$ and $\text{ExprB}$. Succeeds if $\text{ExprA}$ evaluates to be less on the standard order of terms compared to $\text{ExprB}$ and fails otherwise.

D.4 Math and Arithmetic

+/2 The addition function $\text{ExprA} + \text{ExprB}$ takes two operands $\text{ExprA}$ and $\text{ExprB}$ and returns the result of their addition.

-/2 The subtraction function $\text{ExprA} - \text{ExprB}$ takes two operands $\text{ExprA}$ and $\text{ExprB}$ and returns the result of their subtraction.

*/2 The multiplication function $\text{ExprA} * \text{ExprB}$ takes two operands $\text{ExprA}$ and $\text{ExprB}$ and returns the result of their multiplication.

//2 The division function $\text{ExprA} / \text{ExprB}$ takes two operands $\text{ExprA}$ and $\text{ExprB}$ and returns the result of their division.

**/2 The power function $\text{ExprA} ** \text{ExprB}$ takes two operands $\text{ExprA}$ and $\text{ExprB}$ and returns the result of $\text{ExprA}$ to the power of $\text{ExprB}$.

/-1 The negation function $-\text{Expr}$ takes returns the negative value of $\text{Expr}$.

abs/1 The absolute function $\text{abs}(\text{Expr})$ returns the absolute value of $\text{Expr}$.

abs/2 The absolute predicate $\text{abs}(\text{Expr}, \text{?Output})$ retrieves or verifies the absolute value of $\text{Expr}$ in $\text{?Output}$.

sin/1 The sine function $\text{sin}(\text{Expr})$ returns the sine of $\text{Expr}$. $\text{Expr}$ is considered to be in radians.

sin/2 The sine predicate $\text{sin}(\text{Expr}, \text{?Output})$ retrieves or verifies the sine value of $\text{Expr}$ in $\text{?Output}$. $\text{Expr}$ is considered to be in radians.

cos/1 The cosine function $\text{cos}(\text{Expr})$ returns the cosine of $\text{Expr}$. $\text{Expr}$ is considered to be in radians.

cos/2 The cosine predicate $\text{cos}(\text{Expr}, \text{?Output})$ retrieves or verifies the cosine value of $\text{Expr}$ in $\text{?Output}$. $\text{Expr}$ is considered to be in radians.

tan/1 The tangent function $\text{tan}(\text{Expr})$ returns the tangent of $\text{Expr}$. $\text{Expr}$ is considered to be in radians.

tan/2 The tangent predicate $\text{tan}(\text{Expr}, \text{?Output})$ retrieves or verifies the tangent value of $\text{Expr}$ in $\text{?Output}$. $\text{Expr}$ is considered to be in radians.

asin/1 The arcsine function $\text{asin}(\text{Expr})$ returns the arcsine (sine inverse) of $\text{Expr}$. The value returned is in radians.
asin/2  The arcsine predicate \( \text{asin}(+\text{Expr}, ?\text{Output}) \) retrieves or verifies the arcsine (sine inverse) value of \(+\text{Expr}\) in \(?\text{Output}\). \(?\text{Output}\) will be in radians.

acos/1  The arccosine function \( \text{acos}(+\text{Expr}) \) returns the arccosine (cosine inverse) of \(+\text{Expr}\). The value returned is in radians.

acos/2  The arccosine predicate \( \text{acos}(+\text{Expr}, ?\text{Output}) \) retrieves or verifies the arccosine (cosine inverse) value of \(+\text{Expr}\) in \(?\text{Output}\). \(?\text{Output}\) will be in radians.

atan/1  The arctangent function \( \text{atan}(+\text{Expr}) \) returns the arctangent (tangent inverse) of \(+\text{Expr}\). The value returned is in radians.

atan/2  The arctangent predicate \( \text{atan}(+\text{Expr}, ?\text{Output}) \) retrieves or verifies the arctangent (tangent inverse) value of \(+\text{Expr}\) in \(?\text{Output}\). \(?\text{Output}\) will be in radians.

min/2  The min function \( \text{min}(+\text{ExprA}, +\text{ExprB}) \) returns the smaller value between the two operands \(+\text{ExprA}\) and \(+\text{ExprB}\).

min/3  The min predicate \( \text{min}(+\text{ExprA}, +\text{ExprB}, ?\text{Min}) \) sets or verifies the smaller value between the two operands \(+\text{ExprA}\) and \(+\text{ExprB}\) as \(?\text{Min}\).

max/2  The max function \( \text{max}(+\text{ExprA}, +\text{ExprB}) \) returns the larger value between the two operands \(+\text{ExprA}\) and \(+\text{ExprB}\).

max/3  The max predicate \( \text{max}(+\text{ExprA}, +\text{ExprB}, ?\text{Max}) \) sets or verifies the larger value between the two operands \(+\text{ExprA}\) and \(+\text{ExprB}\) as \(?\text{Max}\).

sqrt/1  The square root function \( \text{sqrt}(+\text{Expr}) \) returns the square root value of \(+\text{Expr}\).

sqrt/2  The square root predicate \( \text{sqrt}(+\text{Expr}, ?\text{Output}) \) retrieves or verifies the square root value of \(+\text{Expr}\) in \(?\text{Output}\).

mod/2  The remainder function \( \text{mod}(+\text{ExprA}, +\text{ExprB}) \) takes two operands \(+\text{ExprA}\) and \(+\text{ExprB}\) and returns the remainder value of \(+\text{ExprA}\) divided by \(+\text{ExprB}\).

mod/3  The remainder predicate \( \text{mod}(+\text{ExprA}, +\text{ExprB}, ?\text{Value}) \) sets or verifies the remainder value of \(+\text{ExprA}\) divided by \(+\text{ExprB}\) as \(?\text{Value}\).

pow/2  The power function \( \text{pow}(+\text{ExprA}, +\text{ExprB}) \) takes two operands \(+\text{ExprA}\) and \(+\text{ExprB}\) and returns the result of \(+\text{ExprA}\) to the power of \(+\text{ExprB}\).

pow/3  The power predicate \( \text{pow}(+\text{ExprA}, +\text{ExprB}, ?\text{Value}) \) sets or verifies the value of \(+\text{ExprA}\) to the power of \(+\text{ExprB}\) as \(?\text{Value}\).

exp/1  The exponent function \( \text{exp}(+\text{Expr}) \) returns the exponent value of \(+\text{Expr}\).
**exp/2**  The exponent predicate `exp(+Expr, ?Output)` sets or verifies the exponent value of `+Expr` in `?Output`.

**log/1**  The natural logarithm function `log(+Expr)` returns the natural logarithm (base e) value of `+Expr`.

**log/2**  The natural logarithm predicate `log(+Expr, ?Output)` sets or verifies the natural logarithm (base e) value of `+Expr` in `?Output`.

**log2/1**  The base-2 logarithm function `log(+Expr)` returns the base-2 logarithm value of `+Expr`.

**log2/2**  The base-2 logarithm predicate `log(+Expr, ?Output)` sets or verifies the base-2 logarithm value of `+Expr` in `?Output`.

**floor/1**  The floor function `floor(+Expr)` returns the rounded down value of `+Expr`.

**floor/2**  The floor predicate `floor(+Expr, ?Output)` sets or verifies the rounded down value of `+Expr` in `?Output`.

**ceil/1**  The ceil function `ceil(+Expr)` returns the rounded up value of `+Expr`.

**ceil/2**  The ceil predicate `ceil(+Expr, ?Output)` sets or verifies the rounded up value of `+Expr` in `?Output`.

**round/1**  The round function `round(+Expr)` returns the nearest whole number of `+Expr`.

**round/2**  The round predicate `round(+Expr, ?Output)` sets or verifies the nearest whole number of `+Expr` in `?Output`.

**random/0**  The random function returns a random floating point number between 0 and 1, not inclusive of 1.

**random/1**  The random predicate `random(?Output)` sets a random floating point number between 0 and 1, not inclusive of 1, in `?Output`.

**randomInt/2**  The random integer function `randomInt(+Min, +Max)` returns a random integer between `+Min` and `+Max`, not inclusive of `+Max`.

**randomInt/3**  The random integer predicate `randomInt(+Min, +Max, ?Output)` sets a random integer between `+Min` and `+Max`, not inclusive of `+Max`, in `?Output`.

**succ/2**  The predicate `succ(?N, ?N1)` computes and verifies if the term `?N1` is the successor of `?N`.

**between/3**  The predicate `between(?Low, ?High, ?Value)` verifies if a given value is `Low <= Value <= High`. It is always true for a query like `between(A, A, A)`.

**pi/0**  The pi function returns value of π.
pi/1  The pi predicate pi(?Output) sets the value of $\pi$ in ?Output.

D.5  Lists

append/2  The function append(+List1, +List2) returns a list that is the concatenation of +List1 and +List2.

append/3  The predicate append(+List1, +List2, -Output) outputs a concatenated list of +List1 and +List2 in -Output.

length/1  The function length(+List) returns the length of the given list +List.

length/2  The predicate length(+List, ?Len) sets or verifies the length of the given list +List in ?Len.

member/2  The predicate member(?Element, ?List) verifies if the given element ?Element is a member of the given list +List.

max_list/2  The predicate max_list(+List, ?Max) retrieves or verifies if the given number ?Max is largest value among all elements in the list +List.

min_list/2  The predicate min_list(+List, ?Min) retrieves or verifies if the given number ?Min is smallest value among all elements in the list +List.

sum_list/2  The predicate sum_list(+List, ?Sum) retrieves or verifies if the given number ?Sum is total sum from all elements in the list +List.

D.6  Types

is_ground/1  The predicate is_ground(?Term) succeeds if the given term ?Term is ground, and fails otherwise.

is_variable/1  The predicate is_variable(?Term) succeeds if the given term ?Term is a variable, and fails otherwise.

is_list/1  The predicate is_list(?Term) succeeds if the given term ?Term is a list, and fails otherwise.

is_number/1  The predicate is_number(?Term) succeeds if the given term ?Term is a number, and fails otherwise.

is_integer/1  The predicate is_integer(?Term) succeeds if the given term ?Term is an integer, and fails otherwise.

is_float/1  The predicate is_float(?Term) succeeds if the given term ?Term is a float, and fails otherwise.
atom_number/1 The function `atom_number(?Atom)` returns the numeric representation of a given atom if it exists, does not evaluate otherwise.

atom_number/2 The predicate `atom_number(?Atom, -Output)` returns the numeric representation of a given atom `?Atom` in the output `-Output`, fails otherwise.

atom_string/1 The function `atom_string(?Atom)` returns the string representation of a given atom if it exists, does not evaluate otherwise.

atom_string/2 The predicate `atom_string(?Atom, -Output)` returns the string representation of a given atom `?Atom` in the output `-Output`, fails otherwise.
Appendix E

lps.js User Guide

Please note that the user guide is incomplete as long as the project source code is kept unpublished. The project source code is due to publish open source and to npm's repository at the end of the project.

lps.js is only tested and supported on Node.js versions 6, 7, 8, 9 and 10.

E.1 Usage as CLI Application

To use lps.js as a command-line interface (CLI) application, you need to follow these steps:

1. Install Node.js on your system if you haven’t.
2. Install lps-cli as a globally-accessible program by running "npm -g install lps-cli".

Once installed, the following CLI binaries becomes available on your system.

- `lps` - Used to run LPS programs
- `lps-test` - Used to test LPS program’s output according to test specification
- `lps-generate-spec` Used to generate LPS program’s output as test specification
- `lps-p2p-tracker` Used to provide discovery service when using P2P module in lps.js

To run a LPS program file – for example "program.lps" – run the following command in a CLI environment:

```
lps program.lps
```

For each CLI tool, it is possible to retrieve its usage information by using the "--help" flag.

E.2 Usage as Library in Node.js Runtime

It is possible to use lps.js as a library to your Node.js application by accessing its publicly available API. Assuming you already have a `package.json` file¹, you need to first install lps.js as a npm dependency to your project by running the command²:

```
npm install --save lps
```

¹For more information on `package.json`, see documentation https://docs.npmjs.com/getting-started/using-a-package.json
²Only available at the end of project
npm would add a new entry about lps.js, along with its version number, in your application’s package.json. You can then use lps.js directly by requiring the library in your Node.js JS files, as in the following example:

```javascript
const LPS = require('lps');

LPS.load('example.lps')
  .then((engine) => {
    engine.on('postCycle', () => {
      console.log('[Time ' + engine.getCurrentTime() + ']');

      console.log('Actions: ' + engine.getLastStepActions());
      console.log('Fluents: ' + engine.getActiveFluents());
      console.log('Obs: ' + engine.getLastStepObservations());
    });

    engine.on('warning', (err) => {
      // catch LPS runtime warnings
      console.error('LPS runtime warning: ');
      console.error(err);
    });

    engine.on('error', (err) => {
      // catch LPS runtime errors
      console.error('LPS execution halted due to error');
      console.error(err);
    });

    engine.on('done', () => {
      console.log('Execution complete');
    });

    engine.run();
  })
  .catch((err) => {
    // catch any instantiation errors
    console.error(err);
  });
```

### E.3 Usage as Library in Browser Runtime

If you are already using Webpack for transpiling your existing client-side JavaScript for the browser runtime, you can install and use lps.js as you would on the Node.js runtime without any additional changes.

If you are using vanilla JS or need a transpiled version, you may download the transpiled version from lps.js’s Github repository\(^3\). Assuming you have saved the transpiled JS file as "lps.js", you can then use lps.js by adding the appropriate `<script>` tag to load the transpiled JS file and the LPS object would provide the same API.

\(^3\)https://github.com/mauris/lps.js/releases
Appendix F
P2P Module Reference

The P2P module in lps.js provides peer to peer communication when LPS programs are executed under the Node.js environment. It supports AOP and MAS implementations.

To load the module, use the declaration `loadModule(p2p)`. There are three predicates available for use to tell the P2P module which networks to join:

- `p2pJoin(+Network)` - an auxiliary fact to determine which P2P network to join on port 4100 (of the tracker) of the local machine.
- `p2pJoin(+Network, +Port)` - an auxiliary fact to determine which P2P network to join using a specific tracker port number of the local machine.
- `p2pJoin(+Network, +Address, +Port)` - an auxiliary fact to determine which P2P network to join using a specific tracker port number from a specific IP address.

It is possible to use the `lpsArgs/1` LPS program arguments to indicate the IP address to connect to by configuration.

Peer identifiers are in the form of a tuple `node(Address, Port)`, where `Address` is the IP address of the peer and `Port` is the port number.

To receive notifications on peers joining the network, there are a few events available:

- `p2pConnected(+Network)` - when the current program gets connected to the P2P network specified.
- `p2pPeerConnected(+Network, +Peer)` - when a new peer joins the P2P network specified.

To send or receive messages from other peers, there are several predicates and events that can be used:

- `p2pPeer(+Network, ?Peer)` - Retrieve or verify if a peer is on the network.
- `p2pSend(+Network, +Peer, +Message)` - Sends a message to another peer. The message can be a complex term.
- `p2pReceive(+Network, +Peer, +Message)` - an event received whenever a message is received. The message can be a complex term.
Appendix G

Test API

G.1 Introduction

lps.js provides the environment and means of testing LPS programs automatically through the Tester class. LPS programmers can specify the expected output observed when executing a LPS program given certain input, and the tool will report any discrepancies. Such testing automation reduces programmer’s testing fatigue by replacing manual testing with automated and repeatable tests. When repeatable testing becomes available, LPS programmers can rely on such tests to perform regression testing as they make changes to their LPS programs.

Test specifications are written in LPS syntax. An example specification file will look like the following:

```
expect(maxTime(5)).
expect_num_of(fluent, 1, 1).
expect_num_of(action, 1, 0).
expect(fluent, 1, fire).
expect_num_of(fluent, 2, 0).
expect_num_of(action, 2, 1).
expect(action, 1, 2, eliminate).
expect_num_of(fluent, 3, 0).
expect_num_of(action, 3, 0).
expect_num_of(fluent, 4, 0).
expect_num_of(action, 4, 0).
expect_num_of(fluent, 5, 0).
expect_num_of(action, 5, 0).
```

The following sections in this documentation will guide you through the API provided by lps’s tester tool.

G.1.1 Running Testing Tool

When lps-cli npm package is installed globally, the command lps-test will become available for use. To install lps-cli, run the command line:

```
npm -g install lps-cli
```

To test a LPS program, you need to supply both the pathname to the LPS program file under test and the pathname to the LPS specification file as arguments to the lps-test program. For example:

```
lps-test program.lps program.spec.lps
```
Once started, the LPS program will then be executed. External resources that the LPS program require during test must be made available for the duration of the test: for example P2P peers when using the P2P module. If the expectations specified by the specification file are not met, the tester will report them on the standard output and the tester will exit with a non-zero exit code. Otherwise if all expectations are met, a success message will show and the tester will exit with a zero exit code.

G.1.2 Running Tests from Code

Alternatively, the testing process can be triggered from JS code. An example code given below runs the test from a program given in a string and a test specification given in a string.

```javascript
const LPS = require('lps');
const Tester = LPS.Tester;

// suppose program under test is stored in the variable "source"
LPS.loadString(source)
.then((engine) => {
  let tester = new Tester(engine);

  // suppose test specification is in testSpec
  return tester.test(testSpec);
}).then((result) => {
  if (result.success) {
    // successful
    console.log('Successful');
    return;
  }

  // log errors
  console.error(result.errors.length + ' errors:');
  result.errors.forEach((err) => {
    console.error(' ' + err);
  });
});
```

Code Snippet G.1: An example JS code that executes test using the Tester class directly.

G.2 Testing Predicates

To specify program output expectations, there are several predicates that can be used in the test specification file.

G.2.1 expect/1

\[ \text{expect}(+\text{Goal}) \]

expect/1 can be used to verify the presence of timeless or auxiliary facts in the program. This check is done before the processing of the first state transition (i.e. first cycle).
Consider the example LPS program:

```prolog
maxTime(3).
location(a, 4, 6).
location(b, 2, 2).

% Provides the location coordinate (J, K) between two locations X and Y.
middle(X, Y, J, K) <-
    location(X, A1, B1),
    location(Y, A2, B2),
    X != Y,
    J = (A1+A2)/2,
    K = (B1+B2)/2.
```

and the following test specification. For clarity, the result of each expectation is written in the comment in the same line as the expectation.

```prolog
expect(maxTime(3)). % success
expect(maxTime(2)). % fail
expect(maxTime(A)). % success, A = 3
expect(location(a, 4, 6)). % success
expect(location(b, 2, 2)). % success
expect(location(_, 3, _)). % fail
expect(middle(a, b, J, K)) % success, J = 3, K = 4
expect(middle(a, b, 3, 4)) % success
expect(middle(a, a, J, K)) % fail
```

### G.2.2 `expect/3`

`expect(+Type, +Time, +Goal)`

`expect/3` can be used to query a goal at a specified time of LPS execution. The `+Type` parameter must be one of the values: (i) action, (ii) event, (iii) fluent, or (iv) query. In the case of actions and observed events, `+Time` will be matched to the end time (i.e. second timing argument) of the action or event. The `query` type is used to query intensional predicates or any of the other type and will succeed as long as the goal succeeds at the specified time.

Consider the example LPS program:

```prolog
maxTime(3).
action(move_to/2).
fluent(location/2).
initially(location(0, 0)).
goal_location(1, 1).
```
G.2. TESTING PREDICATES

Chapter G. Test API

9 goal_location(A, B),
10   not location(A, B, T) ->
11   move_to(A, B, T1, T2).
12
13 updates(
14   move_to(A, B),
15   location(_, _),
16   location(A, B)
17 ).

and the following test specification. For clarity, the result of each expectation is written in the comment in the same line as the expectation.

1 expect(fluent, 1, location(0, 0)). % success
2 expect(fluent, 1, location(1, 1)). % fail
3
4 expect(fluent, 2, location(0, 0)). % fail
5 expect(fluent, 2, location(1, 1)). % success
6
7 expect(action, 1, move_to(_, _)). % fail
8 expect(action, 2, move_to(_, _)). % success

G.2.3 expect/4

expect(+Type, +StartTime, ?EndTime, +Goal)

To specify an expectation across a time period, expect/4 can query a goal repeatedly at each state transition. If a variable is given in the ?EndTime argument, the specification becomes perpetual and will check until the end of the last executed cycle.

Consider the example LPS program:

1 maxTime(10).
2
3 action(move_to/2).
4 fluent(location/2).
5
6 initially(location(0, 0)).
7 goal_location(1, 1).
8
9 goal_location(A, B),
10    not location(A, B, T),
11    T > 5 ->
12    move_to(A, B, T1, T2).
13
14 updates(
15   move_to(A, B),
16   location(_, _),
17   location(A, B)
18 ).
and the following test specification. For clarity, the result of each expectation is written in the comment in the same line as the expectation.

```
1 expect (fluent, 1, 6, location (0, 0)). % success  
2 expect (fluent, 1, 6, location (1, 1)). % fail  
3 expect (fluent, 7, _, location (0, 0)). % fail  
4 expect (fluent, 7, _, location (1, 1)). % success
```

### G.2.4 expect_num_of/3

```
expect_num_of (+Type, +Time, +Num)
```

expect_num_of/3 can be used to query the number of actions, events or fluents observed at a specified time of LPS execution. The +Type parameter must be one of the following values:

1. action
2. observation
3. fluent
4. firedRule
5. failedGoal
6. resolvedGoal
7. unresolvedGoal

In the case of actions and observed events, +Time will be matched to the end time (i.e. second timing argument) of the action or event.

The +Num parameter supports the use of special comparison terms to further describe the expectation. By default if a number is passed as +Num, an equality expectation is assumed. See Section G.2.7 for full list of available operators.

Consider the example LPS program:

```
1 maxTime (3).
2 action (move_to /2).
3 fluent (location /2).
4 initially (location (0, 0)).
5 goal_location (1, 1).
6 goal_location (A, B),  
7 not location (A, B, T) ->  
8 move_to (A, B, T1, T2).
9 updates (}
```
and the following test specification. For clarity, the result of each expectation is written in the comment in the same line as the expectation.

```
expect_num_of (fluent, 1, 1). % success
expect_num_of (fluent, 1, 0). % fail
expect_num_of (action, 1, 0). % success
expect_num_of (action, 1, 1). % fail
expect_num_of (action, 2, 1). % success
expect_num_of (action, 2, 0). % fail
```

G.2.5 \texttt{expect\_num\_of/4}

\texttt{expect\_num\_of(+Type, +StartTime, ?EndTime, +Num)}

\texttt{expect\_num\_of/4} can be used to query the number of actions, events or fluents observed across a period of time through LPS execution. The \texttt{+Type} parameter must be one of the values:

1. action
2. observation
3. fluent
4. firedRule
5. failedGoal
6. resolvedGoal
7. unresolvedGoal

In the case of actions and observed events, \texttt{+Time} will be matched to the end time (i.e. second timing argument) of the action or event.

Like \texttt{expect\_num\_of/3}, it is possible to use comparison terms in \texttt{+Num} to further express the actual expectation.

Consider the example LPS program:

```
maxTime(3).
action(move_to/2).
fluent(location/2).
initially(location(0, 0)).
```
Chapter G. Test API

G.2. TESTING PREDICATES

```prolog
goal_location(1, 1).
goal_location(A, B),
   not location(A, B, T) ->
   move_to(A, B, T1, T2).
updates(
   move_to(A, B),
   location(_, _),
   location(A, B)
).
```

and the following test specification. For clarity, the result of each expectation is written in
the comment in the same line as the expectation.

```prolog
expect_num_of(fluent, 1, _, 1).  % success
expect_num_of(fluent, 1, _, 0).  % fail
expect_num_of(action, 1, 1, 0).  % success
expect_num_of(action, 1, 1, 1).  % fail
expect_num_of(action, 2, 2, 1).  % success
expect_num_of(action, 2, 2, 0).  % fail
```

G.2.6 expect_num_cycles/1

```
expect_num_cycles(+Num)
```

`expect_num_cycles/1` can be used to express the number of cycles the LPS program is expected to execute. Despite having the `maxTime/1` declaration, since it is possible for the LPS program to exit early using `lpsHalt/0`, `expect_num_cycles/1` will be useful in determining the number of executed cycles.

Like `expect_num_of/3` and `expect_num_of/4`, it is possible to use comparison terms in `+Num` to further express the actual expectation.

Consider the example LPS program:

```prolog
maxTime(5).
action(move_to/2).
fluent(location/2).
initially(location(0, 0)).
goal_location(1, 1).
goal_location(A, B),
   not location(A, B, T) ->
   move_to(A, B, T1, T2).
goal_location(A, B),
```
and the following test specification. Notice that despite `maxTime(5)` has been declared, the use of `lpsHalt` enable the program to exit early once the goal has been reached. For clarity, the result of each expectation is written in the comment in the same line as the expectation.

```prolog
1 expect_num_cycles(between(1, 5)). % success
2 expect_num_cycles(at_most(5)). % success
3 expect_num_cycles(at_most(1)). % fail
4 expect_num_cycles(2). % success
```

### G.2.7 Special Comparison Terms

Testing declarations that allow special comparison terms include:

- `expect_num_of/3`
- `expect_num_of/4`
- `expect_num_cycles/1`

The following comparison terms are supported:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq/1</td>
<td>The outcome must be equal to the given expected value.</td>
</tr>
<tr>
<td>equal/1</td>
<td></td>
</tr>
<tr>
<td>not_eq/1</td>
<td>The outcome must be not equal to the given expected value.</td>
</tr>
<tr>
<td>notequal/1</td>
<td></td>
</tr>
<tr>
<td>not_equal/1</td>
<td></td>
</tr>
<tr>
<td>atleast/1</td>
<td>The outcome must be at least (no less than) the given expected value.</td>
</tr>
<tr>
<td>at_least/1</td>
<td></td>
</tr>
<tr>
<td>min/1</td>
<td></td>
</tr>
<tr>
<td>atmost/1</td>
<td>The outcome must be at most (no more than) the given expected value.</td>
</tr>
<tr>
<td>at_most/1</td>
<td></td>
</tr>
<tr>
<td>max/1</td>
<td></td>
</tr>
<tr>
<td>between/2</td>
<td>The outcome must between the two given expected values (min and max inclusive).</td>
</tr>
</tbody>
</table>
G.2.8 observe/2 and observe/3

\[
\begin{align*}
\text{observe}(\text{Event}, \text{AtTime}) \\
\text{observe}(\text{Event}, \text{FromTime}, \text{ToTime})
\end{align*}
\]

To leave the original program intact and simulate possible input received by the program under test as observations, it is possible to construct the set of observations that the program under test will receive in the specification file using observe/2 and observe/3. The API for the observe predicates are the same as what LPS provides.

1. \text{observe}(\text{purchase(apple)}, 2, 5).
2. \text{observe}(\text{requestExchange(orange, pear), 3}).
Appendix H

LPS Studio User Guide

H.1 Introduction

LPS Studio is an interactive storytelling tool for visualising LPS programs using lps.js - the LPS interpreter implementation in JavaScript. LPS Studio provides a set of API in LPS to draw objects on the canvas and for LPS programs to receive user input events, such as mouse clicks.

A LPS program can be written and edited in a text editor, then loaded into LPS Studio for execution. Some features supported by LPS Studio include:

- Drawing of shapes and images on the canvas
- Animating shapes and images around the canvas
- Passing mouse events from the canvas to LPS programs

This guide and reference serves to help you get started with building a LPS program for LPS Studio.

H.2 Defining Canvas Objects

LPS Studio keeps track of items that appear on the screen as objects. Each object is associated with an identifier, a type and their properties. The identifier of each object should be unique and it holds a reference back to the original object for updating. There are several object types supported:

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rectangle</td>
<td>A rectangle shape</td>
</tr>
<tr>
<td>rect</td>
<td></td>
</tr>
<tr>
<td>square</td>
<td>A square shape</td>
</tr>
<tr>
<td>circle</td>
<td>A circle shape</td>
</tr>
<tr>
<td>line</td>
<td>A line from a start position to an end position</td>
</tr>
<tr>
<td>image</td>
<td>An image, either loaded from the internet (HTTP/HTTPS) or from file system</td>
</tr>
<tr>
<td>img</td>
<td></td>
</tr>
<tr>
<td>text</td>
<td>An object to display a message</td>
</tr>
<tr>
<td>label</td>
<td></td>
</tr>
</tbody>
</table>

Table H.1: The table of object types and their respective descriptions.

To define an object, write an auxiliary fact for lpsDefineObject(?Id, +Type, +Properties), where the arguments are defined as:

- ?Id: The identifier of the object. If the object will not be referenced later, it’s possible to put
an anonymous variable "_" to indicate that the object does not need an identifier and will not be referenced later.

- +Type: The object’s type, one of the values in Table H.1.
- +Properties: A list of properties. The list of available properties differ object to object and will be discussed later in Section H.2.2.

If the object uses all default properties, the +Properties argument can be an empty list, or left out, i.e. use $lpsDefineObject(?Id, +Type)$.

### H.2.1 Loading Images

To preload images, write auxiliary facts $lpsLoadImage(+ImageId, +UrlOrPath)$. The +ImageID namespace is not shared with the objects, and multiple objects can use the same image. URLs with HTTP and HTTPS protocols are supported, or otherwise a pathname to a local file relative to the loaded program can also be used. Examples:

- $lpsLoadImage(fireSymbol, './fire.png')$.
- $lpsLoadImage(homeIcon, 'https://png.icons8.com/flat_round/2x/cottage.png')$.

### H.2.2 Canvas Object Properties

Different object types have different properties. Each object type and their properties are documented in this section. Properties are written as a list of terms of certain arity.

Note that the canvas and all coordinates uses the inverted Y-axis: The top left corner of the canvas is defined to be $(0, 0)$.

#### Rectangle

- `isHidden(+Bool)`: Set if the object is hidden or not. Defaults to not hidden. +Bool can take one of the three values
  - 1: Indicates that the object is hidden.
  - 0: Indicates that the object is visible.
  - flip: Tells LPS Studio to flip the property of the object from 0 to 1 or from 1 to 0.
- `position(+X, +Y)`: The X and Y coordinate of the center of the shape on the canvas. Both values must be an integer. This property is animatable.
- `size(+Width, +Height)`: The width and height of the shape. Both values must be an integer. This property is animatable.
- `strokeWeight(+Weight)`: The thickness of the stroke around the shape, a number. This property is animatable. Defaults to '1'.
- `strokeDash(+DashPattern)`: The dash/dot pattern of the stroke around the shape, +DashPattern is a list of numbers in pixels. Defaults to the empty list '[]'.
- `strokeStyle(+Style)`: The styling of the stroke around the shape. +Style needs to be a CSS colour string. Defaults to '#000'.
• **fillStyle(+Style):** The styling of the shape fill. +Style needs to be a CSS colour string. Defaults to '#FFF'.

**Square**

• **isHidden(+Bool):** Set if the object is hidden or not. Defaults to not hidden. +Bool can take one of the three values
  - 1: Indicates that the object is hidden.
  - 0: Indicates that the object is visible.
  - flip: Tells LPS Studio to flip the property of the object from 0 to 1 or from 1 to 0.

• **position(+X, +Y):** The X and Y coordinate of the center of the shape on the canvas. Both values must be an integer. This property is animatable.

• **size(+Side):** The length of one side of the shape. Value must be an integer. This property is animatable.

• **strokeWeight(+Weight):** The thickness of the stroke around the shape, a number. This property is animatable. Defaults to '1'.

• **strokeDash(+DashPattern):** The dash/dot pattern of the stroke around the shape, +DashPattern is a list of numbers in pixels.

• **strokeStyle(+Style):** The styling of the stroke around the shape. +Style needs to be a CSS colour string. Defaults to '#000'.

• **fillStyle(+Style):** The styling of the shape fill. +Style needs to be a CSS colour string. Defaults to '#FFF'.

**Circle**

• **isHidden(+Bool):** Set if the object is hidden or not. Defaults to not hidden. +Bool can take one of the three values
  - 1: Indicates that the object is hidden.
  - 0: Indicates that the object is visible.
  - flip: Tells LPS Studio to flip the property of the object from 0 to 1 or from 1 to 0.

• **position(+X, +Y):** The X and Y coordinate of the center of the shape on the canvas. Both values must be an integer. This property is animatable.

• **radius(+Length):** The radius of the circle. Value must be an integer. This property is animatable.

• **strokeWeight(+Weight):** The thickness of the stroke around the shape, a number. This property is animatable. Defaults to '1'.

• **strokeDash(+DashPattern):** The dash/dot pattern of the stroke around the shape, +DashPattern is a list of numbers in pixels.

• **strokeStyle(+Style):** The styling of the stroke around the shape. +Style needs to be a CSS colour string. Defaults to '#000'.

• **fillStyle(+Style):** The styling of the shape fill. +Style needs to be a CSS colour string. Defaults to '#FFF'.
Line

- `isHidden(+Bool)`: Set if the object is hidden or not. Defaults to not hidden. `+Bool` can take one of the three values
  - `1`: Indicates that the object is hidden.
  - `0`: Indicates that the object is visible.
  - `flip`: Tells LPS Studio to flip the property of the object from `0` to `1` or from `1` to `0`.

- `start(+X1, +Y1)`: The X and Y coordinate of the start of the line on the canvas. This property is animatable.

- `end(+X2, Y2)`: The X and Y coordinates of the end of the line on the canvas. This property is animatable.

- `strokeWeight(+Weight)`: The thickness of the stroke, a number. This property is animatable. Defaults to `

- `strokeDash(+DashPattern)`: The dash/dot pattern of the stroke, `+DashPattern` is a list of numbers in pixels.

- `strokeStyle(+Style)`: The styling of the stroke around the shape. `+Style` needs to be a CSS colour string. Defaults to `'#000'`.

Image

- `isHidden(+Bool)`: Set if the object is hidden or not. Defaults to not hidden. `+Bool` can take one of the three values
  - `1`: Indicates that the object is hidden.
  - `0`: Indicates that the object is visible.
  - `flip`: Tells LPS Studio to flip the property of the object from `0` to `1` or from `1` to `0`.

- `flipHorizontal(+Bool)`: Set if the image is flipped horizontally or not. Defaults to not flipped horizontally. `+Bool` can take one of the three values
  - `1`: Indicates that the object is flipped horizontally.
  - `0`: Indicates that the object is not flipped horizontally.
  - `flip`: Tells LPS Studio to flip the property of the object from `0` to `1` or from `1` to `0`.

- `flipVertical(+Bool)`: Set if the image is flipped vertically or not. Defaults to not flipped vertically. `+Bool` can take one of the three values
  - `1`: Indicates that the object is flipped vertically.
  - `0`: Indicates that the object is not flipped vertically.
  - `flip`: Tells LPS Studio to flip the property of the object from `0` to `1` or from `1` to `0`.

- `position(+X, +Y)`: The X and Y coordinate of the center of the shape on the canvas. Both values must be an integer. This property is animatable.

- `size(+Width, +Height)`: The width and height of the shape. Both values must be an integer. This property is animatable.

- `image(+ImageId)`: The image to show, image must be loaded using `lpsLoadImage/2`. See also Section H.2.1.
Text

• isHidden(+Bool): Set if the object is hidden or not. Defaults to not hidden. +Bool can take one of the three values
  - 1: Indicates that the object is hidden.
  - 0: Indicates that the object is visible.
  - flip: Tells LPS Studio to flip the property of the object from 0 to 1 or from 1 to 0.

• position(+X, +Y): The X and Y coordinate of bottom left of the text area. Both values must be an integer. This property is animatable.

• font(+Font): The font size and font family to use, a CSS font value.

• caption(+Text): The string to show on the screen. Defaults to '12px sans-serif'.

• maxWidth(+Size): The maximum width the text object should stop at. If the size of the text object computes to be longer than the maxWidth value, either a narrower compatible font would be chosen or the text would be scaled down. A number in pixels. Defaults to undefined.

• strokeWeight(+Weight): The thickness of the stroke, a number, around the font. This property is animatable. Defaults to '0'.

• strokeStyle(+Style): The styling of the stroke around the font. +Style needs to be a CSS colour string. Defaults to '#FFF'.

• fillStyle(+Style): The styling of the font. +Style needs to be a CSS colour string. Defaults to '#000'.

H.3 Updating Canvas Object Properties

The lpsUpdateObject(+Id, +UpdatedProperties) predicate can be used in rules and clauses of your LPS program to update properties of an object identified by its +Id identifier given during definition. Like the +Properties argument in lpsDefineObject/3, the +UpdatedProperties argument contains the list of properties to update. For example, to update the string in a Text object when an event occurs:

```
walking(_, Destination, T1, T2) ->
  lpsUpdateObject(statusText, [
    caption('Walking to ' + atom_string(Destination))
  ]).
```

Additional shorthands are supported by LPS Studio to reduce the amount of code for frequently used updating:

• lpsShowObject(+Id): Sets the isHidden property of the object identified by +Id to false.

• lpsHideObject(+Id): Sets the isHidden property of the object identified by +Id to true.
H.4 Animating Canvas Objects

Some properties marked "animatable" of an object identified by +Id can be animated using the lpsAnimateObject(+Id, +Duration, +Properties) predicate. The +Duration property is the duration of the animation in milliseconds, and the animation can complete out-of-sync with the cycles of LPS program.

For example, to animate the increase in size of a circle:

```
1  hit(CircleId, T1, T2),
2  lpsDefineObject(CircleId, circle, _) ->
3    lpsAnimateObject(CircleId, 500, [
4      radius(20)
5    ]).
```

**Code Snippet H.1:** Animating a circle’s radius. Notice that the lpsDefineObject is used to constrain and ensure that the CircleId variable a valid identifier that corresponds to a circle.

H.5 Observed Events

LPS programs executing in LPS Studio can observe events triggered by the user on the canvas or observations entered by through LPS Studio console. These observations are automatically declared as events when used with LPS Studio.

lpsClick(-X, -Y) Occurs when the user clicks somewhere on the canvas, and the arguments -X and -Y provide the coordinate of the click relative to the canvas.

lpsClick(-ObjectId, -X, -Y) Occurs when the user clicks on a known object on the canvas, and the argument -ObjectId provides the identifier of the object, and arguments -X and -Y provide the coordinate of the click relative to the canvas.

lpsMouseDown(-X, -Y) Occurs when the user press on the mouse left button somewhere on the canvas, and the arguments -X and -Y provide the coordinate of the click relative to the canvas.

lpsMouseDown(-ObjectId, -X, -Y) Occurs when the user press on the mouse left button on a known object on the canvas, and the argument -ObjectId provides the identifier of the object, and arguments -X and -Y provide the coordinate of the click relative to the canvas.

lpsMouseUp(-X, -Y) Occurs when the user releases on the mouse left button somewhere on the canvas, and the arguments -X and -Y provide the coordinate of the click relative to the canvas.

lpsMouseUp(-ObjectId, -X, -Y) Occurs when the user releases on the mouse left button on a known object on the canvas, and the argument -ObjectId provides the identifier of the object, and arguments -X and -Y provide the coordinate of the click relative to the canvas.

lpsMouseMove(-X, -Y) Occurs when the user moves the mouse somewhere over the canvas, and the arguments -X and -Y provide the coordinate of the click relative to the canvas.
lpsMouseMove(-objectId, -X, -Y) Occurs when the user moves the mouse over a known object on the canvas, and the argument -objectId provides the identifier of the object, and arguments -X and -Y provide the coordinate of the click relative to the canvas.