LOLA 0.064

A Programming Language for
Augmenting Programming Languages

RESEARCH THESIS

Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Science

Iddo E. Zmiry

Submitted to the Senate of the Technion - Israel Institute of Technology
Nisan, 5776, Haifa, April, 2016
Acknowledgments

This Research Thesis was Done under the Supervision of Prof. Joseph Gil in the Faculty of Computer Science

I thank my parents and my family for their everlasting love, encouragement and support. I am grateful to my advisor Prof. Joseph Gil. He patiently guided me throughout the work, and taught me much about research. I would like to thank Prof. David Lorenz for his help and for inspiring discussions with him. My thanks to Tomer Levy for his great help.

The Generous Financial Help of the Technion is Gratefully Acknowledged
Contents

List of Tables ix
List of Figures xi
List of Algorithms xiii
List of Listings xv
Abstract xvii
Glossary xix

1 Introduction 1
  1.1 Thesis ...................................................... 3
  1.2 The LOLA Approach ......................................... 5
  1.3 The Max Lexies ............................................. 6
  1.4 Indentation .................................................. 10
  1.5 Ranged Loops and Loop Unrolling ......................... 11
  1.6 Directives .................................................. 12
  1.7 Contributions, Challenges and Achievements .......... 12
  Outline ....................................................... 17

2 Related Work 19
  2.1 Embedded Languages ....................................... 19
  2.2 Preprocessing ............................................... 19
  2.3 Other Related Fields ....................................... 22

3 Examples Gallery 25
  3.1 Introductory Examples ..................................... 25
    3.1.1 Comparison with the C Preprocessor ................. 25
    3.1.2 Halstead’s Code Metrics ................................ 26
  3.2 Implementing New Language Constructs .................. 27
    3.2.1 Switch Construct ...................................... 27
    3.2.2 Poor Man’s Inheritance in C ......................... 29
    3.2.3 Sets as Bit Masks ..................................... 29
  3.3 New Keywords in JAVA ...................................... 30
    3.3.1 Auxiliary Lexies for JAVA ............................. 30
    3.3.2 Invoke an Overridden Method (Super) ............... 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.3 Current Method Name (me)</td>
<td>31</td>
</tr>
<tr>
<td>3.3.4 Numbered Arguments</td>
<td>31</td>
</tr>
<tr>
<td>3.3.5 Current Class Name (This)</td>
<td>32</td>
</tr>
<tr>
<td>3.4 Miscellaneous Examples</td>
<td>33</td>
</tr>
<tr>
<td>3.4.1 Monitor an Expression (monitor)</td>
<td>33</td>
</tr>
<tr>
<td>3.4.2 Avoid Repetitive Code</td>
<td>33</td>
</tr>
<tr>
<td>3.5 Rewriting</td>
<td>35</td>
</tr>
<tr>
<td>3.6 Reflection</td>
<td>36</td>
</tr>
<tr>
<td>3.7 Island Grammars and typedef</td>
<td>37</td>
</tr>
<tr>
<td>3.8 Before Function</td>
<td>38</td>
</tr>
<tr>
<td>3.8.1 Adding an Anchor</td>
<td>38</td>
</tr>
<tr>
<td>3.8.2 Adding an Example</td>
<td>39</td>
</tr>
<tr>
<td>3.9 P4 Checksum</td>
<td>39</td>
</tr>
<tr>
<td>3.9.1 Calling Convention</td>
<td>42</td>
</tr>
<tr>
<td>4 Lola Concepts and Elements</td>
<td>43</td>
</tr>
<tr>
<td>4.1 The Host Language Configuration File</td>
<td>43</td>
</tr>
<tr>
<td>4.2 Directives</td>
<td>43</td>
</tr>
<tr>
<td>4.3 Compound Objects</td>
<td>45</td>
</tr>
<tr>
<td>4.4 Generating Expressions</td>
<td>48</td>
</tr>
<tr>
<td>4.5 Extended Regular Expressions</td>
<td>48</td>
</tr>
<tr>
<td>4.6 Lexies</td>
<td>53</td>
</tr>
<tr>
<td>4.6.1 Declarative Header Sections</td>
<td>54</td>
</tr>
<tr>
<td>4.6.2 Action Sections</td>
<td>54</td>
</tr>
<tr>
<td>4.6.3 Declarative Footer</td>
<td>54</td>
</tr>
<tr>
<td>5 The Host Language Configuration File</td>
<td>55</td>
</tr>
<tr>
<td>5.1 A High-Level Description of the Configuration File</td>
<td>56</td>
</tr>
<tr>
<td>5.2 Regular Expressions</td>
<td>56</td>
</tr>
<tr>
<td>5.3 Tokens</td>
<td>57</td>
</tr>
<tr>
<td>5.3.1 Digraphs and Trigraphs</td>
<td>57</td>
</tr>
<tr>
<td>5.4 Token Classifications</td>
<td>57</td>
</tr>
<tr>
<td>5.4.1 Reserved Token Classifications</td>
<td>57</td>
</tr>
<tr>
<td>5.5 Balanced Pairs</td>
<td>58</td>
</tr>
<tr>
<td>5.6 Configuration to an Input Language Convention</td>
<td>59</td>
</tr>
<tr>
<td>6 Lexical and Syntactical Analysis</td>
<td>61</td>
</tr>
<tr>
<td>6.1 Lexical Analysis</td>
<td>62</td>
</tr>
<tr>
<td>6.2 The Abstract Syntax Tree</td>
<td>66</td>
</tr>
<tr>
<td>6.3 Children of Constructors and Elaborators</td>
<td>66</td>
</tr>
<tr>
<td>6.4 Parsing without Explicit Grammar</td>
<td>69</td>
</tr>
<tr>
<td>6.5 The Stack of Pending Builders</td>
<td>72</td>
</tr>
<tr>
<td>6.6 Trivia Management</td>
<td>74</td>
</tr>
</tbody>
</table>
CONTENTS

7 Pruning, Matching and Triggering 75
  7.1 Input Pruning ........................................... 75
  7.2 The Match of the Matches ................................. 76
  7.3 Multiple Matches of the Same Lexi ......................... 77
  7.4 LOLA’s Iterative Fixed Point Behavior ...................... 78
  7.5 Locations and Substrings ................................. 79
    7.5.1 Locations .............................................. 79
    7.5.2 Substrings ............................................. 80
  7.6 Total Order of Overruling ................................ 81
  7.7 Understanding Overruling .................................. 83
  7.8 The Single Chance Principle and Earmarking ............... 85
  7.9 The Inner Out Principle and Backtracking ................ 86

8 Conclusions and Future Work 87
  8.1 Future Research ........................................ 88
  8.2 LOLA’s Past and Planned Evolution ........................ 89
  8.3 Reflection ............................................... 89

Bibliography .............................................. 91

Hebrew Abstract ........................................ 96
List of Tables

1.1 A sample of LOLA directives .................................................. 14
4.1 The recursive structure of LOLA generators ............................ 44
4.2 The recursive structure of LOLA’s extended regular expressions .... 50
4.3 Sections in a LOLA pattern definition .................................... 52
5.1 Examples of identifier groups .................................................. 59
6.1 Variety in restrictions placed on children by different constructors .. 67
6.2 Variety in restrictions placed on children by different elaborators .. 68
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Hare Lola Bunny. ♀ 1996–</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Hare Lexi Bunny. ♀ September 17, 2005 – May 5, 2007</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>Halstead complexity measures</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>A screen-shot of an editing session of a LOLA configuration file for C</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>An extended regular expression, specifying that “#find”, “#replace”, and</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>“#delete” may occur at any order, but at most once, and that “#replace”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>and “#delete” cannot both occur</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Full EBNF grammar of configuration file</td>
<td>55</td>
</tr>
<tr>
<td>6.1</td>
<td>The stream of lexical units (bunnies) produced from the code in Listing 6.1</td>
<td>62</td>
</tr>
<tr>
<td>6.2</td>
<td>Hare Bugs Bunny. ♂ April 30, 1938– (shown with Lola Bunny (Figure 1.1))</td>
<td>63</td>
</tr>
<tr>
<td>6.3</td>
<td>The stream of remaining lexical units (bunnies) after removing the lexi from</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>Figure 6.1</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>The stream of lexical units (bunnies) of the lexi in Listing 6.1, to be later</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>parsed as LOLA directive</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>The abstract syntax tree created for the lexi in Listing 6.1</td>
<td>66</td>
</tr>
<tr>
<td>6.6</td>
<td>An automaton describing the grammar of some generic builder</td>
<td>71</td>
</tr>
<tr>
<td>7.1</td>
<td>The four rules that impose a total order on candidate matches, whereby</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>ensuring existence of a single overruling match</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Tokens, locations and substrings for the tokens string 1,2,3,(4,5,6)</td>
<td>80</td>
</tr>
<tr>
<td>7.3</td>
<td>The total order imposed by overruling on S(7), the set of substrings of a</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>string of length 7</td>
<td></td>
</tr>
</tbody>
</table>
List of Algorithms

6.1 Procedure \texttt{percolate}(c), processing \emph{c}, a child to be fitted with the stack of pending builders. .................................................. 72

6.2 Function \texttt{directive}(c), returning the AST for the directive created by constructor \emph{c} ................................................................. 73

7.1 Function \texttt{fixedPoint}(\mathcal{L}, S) receiving a set of lexies \mathcal{L} and pruned text \emph{S}, and returning \emph{S} after all lexies in \mathcal{L} were exhaustively applies to it. ............... 78
List of Listings

1.1 JAVA code with multiple comparisons of the same string variable with a sequence of string literals ........................................ 1
1.2 Basic matching and replacing of the highlighted code ................ 3
1.3 Max macro in the C preprocessor ........................................ 6
1.4 A lexi for the comma argument (similar to C preprocessor macro's) .................................................................................. 6
1.5 Max lexi in LOLA ............................................................... 8
1.6 Max lexi in LOLA with variable number of arguments .......... 9
1.7 Using @replace for defining a loop based on range syntax ........ 11
1.8 The output of Listing 1.7 produced by LOLA ......................... 13

3.1 Making C look like PASCAL [58] (the LOLA equivalents of “#define”) .......................................................... 25
3.2 Augmenting JAVA [4] with string switch ............................... 28
3.3 Poor man’s inheritance in C ............................................... 29
3.4 Augmenting C with enumerated bit positions ....................... 30
3.5 Auxiliary lexi for lexies for methods in JAVA ....................... 30
3.6 Auxiliary stack of methods in JAVA ..................................... 31
3.7 Augmenting JAVA with a Super keyword ............................... 31
3.8 Augmenting JAVA with a me keyword ................................. 31
3.9 Augmenting JAVA with numbered arguments ...................... 32
3.10 Augmenting JAVA with a This keyword ............................... 32
3.11 Augmenting JAVA with a monitor method .......................... 33
3.12 Box it lexi for boxing JAVA’s value types into reference types ................................................................. 34
3.13 A use of the BOX IT lexi introduced in Listing 3.12 ............. 35
3.14 Using @run for defining a loop based on range syntax .......... 35
3.15 Assertion using meta-programming .................................... 36
3.16 A lexi that uses reflection ................................................. 37
3.17 A lexi that simulates typedef in JAVA ............................... 37
3.18 A lexi that simulates aspect oriented programming ................ 38
3.19 JAVA class pattern ......................................................... 39
3.20 An example of the usage of the @example (the test fails) ........ 39
3.21 An example of a lexi for creating a checksummed field in P4 (continued in Listing 3.22) ............................................. 40
3.22 The @example and @resultsIn sections of Listing 3.21 .......... 41
3.23 A C preprocessor macro definition and usage for a cross-compiler fastcall calling convention .................................. 42
3.24 LOLA’s Listing 3.23 equivalent for a lexi definition and usage for a cross-compiler fastcall calling convention ............. 42
<table>
<thead>
<tr>
<th>Section Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Using LOLA to produce C preprocessor code</td>
<td>47</td>
</tr>
<tr>
<td>6.1 A contrived LOLA lexi demonstrating how elaborators attach to their respective constructors</td>
<td>62</td>
</tr>
<tr>
<td>6.2 A meta-lexi specifying restrictions on sections of a lexi</td>
<td>69</td>
</tr>
<tr>
<td>7.1 An example of how nesting of rewrites works</td>
<td>86</td>
</tr>
</tbody>
</table>
Abstract

LOLA, the Language Of Language Augmentation, is the topic of this master of science in computer science thesis. LOLA is a modern, language-independent, preprocessor, for language extension. Inspired by the island grammars trend, LOLA makes many non-trivial language extensions possible with no more than a handful lines of code.

LOLA was used for tasks such as augmenting JAVA with keywords This (bound to the name of current class) and Super (invoke the overridden method), and positional arguments, for adding a range based for loop construct to C, for AOP like features and even for computing code metrics. In fact, LOLA is expressive enough for users to define their own variant of the C-preprocessor and other preprocessors, including the immediate and delayed semantics of the preprocessor of Unix “make”.

A user-accessible configuration file defines the tokens of the host language. LOLA analyzes the token stream (rather than the raw character stream) with extended regular expressions, and employs “generating expressions” to replace, delete, or otherwise amend pieces of its input. A language is augmented by appropriate use of “lexies”, where each lexi is a semi-declarative module describing a certain code-transformation.

Underlying LOLA is a PYTHON engine, and LOLA programmers are allowed full access to PYTHON language features. With this model the definition of LOLA can be kept short, concentrating in weaving together, as seamlessly as possible, LOLA keywords, the stream of tokens of the augmented language, and the PYTHON code the programmer chooses to use.
Glossary

**Action**  a section of a lexi, such as `@run`, `@log` and `@delete` that is executed when the lexi is triggered.

**AST**  acronym of *Abstract Syntax Tree*.

**Balanced**  as in the balancing of tokens; a stream of tokens is balanced with respect to token pairs as defined in all `⟨balanced⟩` clauses in the configuration file.

**Built-in keyword**  a keyword which is a part of LOLA and can be used in any host programming languages, e.g., `@Any`.

**Bunny**  a collective name for trivia and tokens (of either language).

**Class**  a grouping of tokens in their hierarchical taxonomy.

**Compound object**  the result of applying a constructor to its arguments.

**Configuration file**  an XML file that specifies how a character stream is broken into tokens and other lexical units of the host language.

**Constructor**  constructors are LOLA keywords that make compound objects out of atoms and other, smaller, compound objects.

**Degenerate lexi**  a lexi that does not include a `@Find` section, and therefore is triggered immediately, at the point it is encountered.

**DFA**  acronym of *Deterministic Finite Automaton*.

**Directive**  a file level instruction to a preprocessor.

**EBNF**  acronym of *Extended Backus—Naur Form*.

**Elaborator**  elaborators are bound to constructors to make compound objects.

**Generative expression**  an LOLA expression that generates code of the host language.

**Greedy semantics**  semantics of regular expression matching, by which a regular expression picks the *longest* possible matching prefix of the text (not used in LOLA).
**Host language** refers to the programming language that LOLA processes its source files.

**Host specific keyword** a keyword which is defined using the configuration file, and names a token or a class, e.g., @Identifier.

**Input** the text of the program processed by LOLA.

**Lazy semantics** semantics of regular expression matching, by which a regular expression picks the shortest possible matching prefix of the text (not used in LOLA).

**Lexi** the modular unit of computation in LOLA describing a certain code-transformation (plural is “lexies”).

**Literal** a notation for represents a fixed value.

**Location** an index into the text pointing between two tokens; a location may also be at the beginning and at the end of the input.

**Lola** acronym of the Language of Language Augmentation.

**Lola argument** a sequence of tokens and compound objects that are sent as a list argument to a constructor.

**Lola keyword** a LOLA token starts with a @ character.

**Match** the correlation of a pattern and a certain list of tokens.

**Natural numbers** $\mathbb{N}$ the set of all natural numbers.

**No lower indentation argument rule** stating that the arguments of any LOLA keyword are never found at an indentation level which is lower than that of the keyword itself.

**Parameterless constructor** a constructor such as @Any which takes no parameters, sometimes called atom.

**Pattern** an extended regular expression.

**Pruning** the process of removing directives from a source file.

**Python expression** a fragment of code that evaluates to an object.

**Python snippet** a fragment of PYTHON code, typically short.

**RE** acronym of Regular Expression, interchangeably used with the term pattern, this is a LOLA code fragment describing a pattern.

**Reifying object** the PYTHON object created upon a match.

**Right associativity rule** states that tokens and keywords found in a line associate to the left.
Section a part of a lexi definition.

Sequence of Python commands a snippet which is a suite of PYTHON statements.

Short Python string literal a string enclosed in matching double quotes.

Simple plural rule the dictation that the plural form of a name $x$ is $x\text{s}$, i.e., the name obtained by appending the letter “s” to the name $x$, irrespective of the grammatically correct plural form of the word $x$.

Substring $S_{f,t}$ a pair of two locations, the second being no lesser the first, denoting a consecutive stretch of zero or more tokens $\tau_f \cdots \tau_t$, constituting a substring $S_{f,t} \in \mathbb{S}$.

Substrings set $\mathbb{S}$ the set of all substrings: $\mathbb{S}(n) = \{S_{f,t} \mid 0 \leq f \leq t \leq n\}$.

Token a symbol usually created during lexical analysis.

Trigger the cause (usually a sequence of tokens) of a lexi invocation.

Triggering the process of a lexi invocation, caused by a trigger.

Trivia are the things that separate successive tokens, typically the space or new line characters, comments, spans of these, and even the empty string (there is also a trivia at the beginning and at the end of the stream).

User defined keyword a keyword which is defined in user code (in the name of a pattern).
Chapter 1

Introduction

Everyone knows how risky is the business of adding features to existing successful languages.

“When in doubt, always abstain.,”

one well known language designer [39, p.497–508] decreed after explaining that even innocent changes very often interfere at the least expected places.

Abstention leads to delays. A case in point is a switch over strings in JAVA, which seems so natural, especially after reading the repetitive code of Listing 1.1.

```
public static void main(String[] args) {
    for (String arg: args)
        if (arg.equals("-c") || arg.equals("--bytes"))
            ...
        else if (arg.equals("-m") || arg.equals("--chars"))
            ...
        else if (arg.equals("-w") || arg.equals("--words"))
            ...
        else if (arg.equals("-l") || arg.equals("--lines"))
            ...
        else if (arg.equals("--help"))
            ...
        else if (arg.equals("--version"))
            ...
    }
```

It took 20 years from the time requests started accumulating¹ until the language extension was finally implemented².

Designer’s prudence and delays can have a non-negligible toll on resources, i.e., the relatively late addition of generics to JAVA forced programmer to use the unsafe Vector substitution that was available at the time.

¹http://bugs.java.com/bugdatabase/view_bug.do?bug_id=1223179
²http://docs.oracle.com/javase/7/docs/technotes/guides/language/strings-switch.html
CHAPTER 1. INTRODUCTION

Yet, this price seems to be unavoidable. A programming language that haphazardly accepts changes, cannot be trusted by its clients. The design of a programming language is a delicate art of balancing between conflicting trade-offs and all changes must be weighed carefully. On the one hand, to make the language appealing, the language designer would like to equip future programmers with a variety of powerful and useful abstractions. On the other hand, the folk-lore of language design is that any feature added to a language has its price: reduced elegance of the design, a steeper language learning curve, and complication of the construction of language processing tools.

LOLA, the Language of Language Augmentation, was designed with this dilemma in mind: The approach it offers is of empowering the language users, the programmers in the trenches, by making it easier for them to introduce lightweight, and not so lightweight, extensions to the programming language they use.

Figure 1.1 Hare Lola Bunny. © 1996–
1.1. **Thesis**

The thesis that this research promotes is that with the aid of a powerful preprocessing tool, programmers would be able to add and experiment with new abstractions to support the programming language they use, without waiting years for the language to evolve. LOLA was designed to make this alternative viable. LOLA makes it easy for programmers to enhance and extend the programming language they use, without making any changes to the language core.

We describe a large number of innovative language extensions made possible with LOLA. We also show how C [29] might be enhanced with high level constructs, while preserving the language efficiency and its underlying principle of *no hidden cost* [52, p.43].

On course, we demonstrate the application of PYTHON [37] as a language for metaprogramming. LOLA was designed in accordance with the vision that PYTHON code embedded in it would provide C programmers with higher abstractions, all implemented in C, and all
realized using the familiar metaphor of a text preprocessor.

This document describes LOLA in detail. In a nutshell, we can say now that the modular unit of computation in LOLA is a lexi. Lexies employ patterns, which are rather extended regular expressions defined on tokens. Patterns define points where a lexi is triggered on the input to take some appropriate action. Actions can be code replacements, but also short snippets of Python code.

Figure 1.2 Hare Lexi Bunny.♀ September 17, 2005 – May 5, 2007

LOLA achieves the end of language independence by using an XML configuration file defining the breaking of source code of the host language into lexical units, including tokens,
The configuration file also defines a hierarchical taxonomy of tokens, e.g., the classification of identifiers into @CamelCase, @camelCase, @UPPER_CASE (and other capitalization and naming conventions), grouping of selected tokens into categories @Operator and @Punctuation, etc.

Some other unique features of LOLA include the blending of declarative and imperative approaches in the definition of lexies (going beyond assertions and JAVA_DOC [33], and reminiscent of EIFFEL [28]'s ensure, require, invariant and variant), lightweight language definition borrowing as much as possible from PYTHON, seamless weaving of code of three languages, and a novel, modular means for the definition of LOLA’s own grammar.

The remainder of this introductory chapter is organized as follows: Section 1.2 compares the LOLA approach with that of traditional pre-processing. This comparison is furthered in Section 1.3 which shows how LOLA compares with the C preprocessor in the implementation of the vanilla max macro. Section 1.4 disrupts the comparison to provide the reader an initial intuition of LOLA’s use of indentation. Section 1.5 continues the comparison by showing how LOLA can be used for extending the C programming language, while Section 1.6 provides a table of comparison between LOLA directives and C preprocessor directives.

### 1.2 The LOLA Approach

In a sense, the approach that LOLA takes is not new. Macros of the ancient C preprocessor made it possible to make certain additions to C. However, the many limitations of the C preprocessor and of other macro systems (such as M4 [20]), may be the reason that modern programming languages tend less to rely on a pre-processing stage.

Yes, LOLA is a preprocessor in the sense that its input is source code intermixed with directives, and its output is the source code after the application of the directives. Yes, LOLA is a macro system in the sense that directives manipulate the source without fully parsing it, figuring its semantics, inferring types etc.

However, in contrast with other macro and preprocessing systems, LOLA is specifically designed for the task of language extension. To this end, instead of macros, LOLA’s code is organized in lexies, where each lexi is a modular specification, including both imperative and declarative sections, of some source code transformation. As we shall see, a lexi augmenting JAVA with a switch construct on strings can be done in a lexi comprised of just a handful of lines.

The syntax of the C preprocessor’s macros is that of function calls. Lexies are different in that they rely on extended regular expressions, called patterns in the LOLA lingo, to determine code locations in which they should be applied. Arguments to macros are typically expressions, or a specific syntactical unit of the programming language they tend. In contrast, LOLA is language independent, and arguments to macros can be any sequence of tokens.

LOLA lexies are a bit similar to AWK [1] line patterns, except that unlike AWK (and the C preprocessor, M4, and other such systems) LOLA refuses to take the trouble of defining its own syntax for arithmetical expressions, variables, arrays, control flow constructs and other commodities that concern most programming languages. Instead, LOLA relies on PYTHON

---

3Henceforth, the term input stream, or just input refers to the text of the program processed by LOLA and the term “host language” refers to the programming language that LOLA processes its source files, C being our primary example.
for defining these, and concentrates on defining its central unique language constructs: lexies, patterns and generating expressions.

As it turns out, computation carried out by short PYTHON snippets makes it possible to analyze the host code more thoroughly than possible with traditional macro systems.

1.3 The Max Lexies

To demonstrate the difference between LOLA and traditional preprocessors, consider the problem of implementing a max macro. In the C preprocessor a max macro would look like the code in Listing 1.3.

Listing 1.3 Max macro in the C preprocessor

```c
#define max(X,Y) ((X) > (Y) ? (X) : (Y))
```

As usual with C preprocessor macros, the body of macro `max` wraps the arguments in parenthesis to protect against the predicament of a substitution being incorrect due to operator precedence.

Arguments to `max`, just like all other macros of the C preprocessor, are expressions separated by commas. Since LOLA knows nothing about the tokens of C, to define a similar macro, it is first necessary to define the syntactical construct which macros take as arguments. This is done in Listing 1.4.

Listing 1.4 A lexi for the comma argument (similar to C preprocessor macro's)

```lola
@Find(NoCommasExpression)
  @Match @Any @exceptFor @Any, @Any

@Find(Expression)
  @Either @NoCommasExpression @or (@Any)

@Find(Expressions)
  @OneOrMore @Expression @separator,
```

The first lexi in the listing, makes use of `@Any`, a LOLA builtin, which matches any balanced sequence of tokens of C, the host language. Tokens of the host language are defined in a language-specific configuration file. This configuration file also specifies which tokens of the language come in balanced pairs. There are three such pairs in C (and other curly brackets languages): “(” must be balanced against “)”, “[” against “]”, and, the “{” token which must match against its “}” counterpart. Pattern `@Any` matches therefore, e.g.,

(2,3),

Technion - Computer Science Department - M.Sc. Thesis  MSC-2016-12 - 2016
but not

\[(2,3), f(\quad)\]

The pattern in the first lexi of the listing thus means: Any sequence of tokens, which does not match \(\text{@Any, @Any}\), i.e., cannot be written as a balanced sequence of tokens, followed by a comma, and then followed by another balanced tokens’ sequence. Notice the use of conjunctions in LOLA patterns, an operator not available in most regular expression languages.

Note that pattern \(\text{@Any, @Any}\) matches token sequences such as

\[(2,3),f(4,5)\]

but not

\[f(4,5)\]

since neither “f(4, ” nor “,5)” are balanced sequences that match \(\text{@Any}\).

The interpretation of the phrase \(\text{@Match @Any @exceptFor @Any, @Any}\) as “any balanced sequence of tokens, except for those sequences which can be broken into two such balanced sequences separated by a comma token” is made by LOLA by identifying these elements in it:

- Keyword \(\text{@Match}\) is a constructor which creates a conjunctive regular expression out of its constituents.

- Keyword \(\text{@Any}\) is a \textit{parameterless constructor} as described above.

- Keyword \(\text{@exceptFor}\) is an \textit{elaborator}. Elaborators are similar to constructors in their grammatical structure, except that they do not make a pattern on their own, and must be associated with a constructor, \(\text{@Match}\) in this case.

  Despite a couple of exceptions, constructors use the “CameCase” capitalization and case convention and elaborators the “cameCase” convention.

- Host token , which is the second of the three parameters to elaborator \(\text{@exceptFor}\).

Writing

\[\text{@Find}(\text{NoCommasExpression})\]

binds the name \(\text{NoCommasExpression}\) to this pattern.

This name or, more accurately, its “plural” form, \(\text{NoCommasExpressions}\) can be used in PYTHON. Variable \(\text{NoCommasExpressions}\) is made up of a list of all cases in which \(\text{@NoCommasExpression}\) was matched.

Another possible use of the binding is in defining LOLA patterns that use the user defined pattern \(\text{@NoCommasExpression}\) as a building block.

In general, LOLA keywords fall into one of three categories:

1. \textit{Builtin}. A category includes keywords such as \(\text{@Match, @exceptFor, @Any}\), which are part of LOLA and can be used in any host programming language.
2. *Host Specific.* In this category, we find names of individual tokens, such as @Integer, and of classes of tokens, such as @Operator, all being defined in the configuration file of the host language.

3. *User Defined.* Three examples of this category, pattern @NoCommasExpression, pattern @Expression, and pattern @Expressions are shown in Listing 1.5.

We see the user defined pattern @NoCommasExpression being employed in the second lexi of Listing 1.5: User defined pattern @Expression, is either a @NoCommasExpression or, (@Any), i.e., a balanced sequence of tokens wrapped in parenthesis.

Accordingly @Expression matches

\[
\begin{align*}
  f(4, 5) \\
  (f(4), 5)
\end{align*}
\]

but not

\[
\begin{align*}
  3, f(4, 5)
\end{align*}
\]

Thus, @Expression specifies a valid argument to a C preprocessor macros.\(^4\)

The third lexi of Listing 1.4 uses pattern @Expression to define pattern @Expressions which matches a comma separated, non-empty list of @Expressions. The LOLA *builtin constructor* @OneOrMore takes elaborator @separator with its argument “,” that, together, tell that the recurring expressions in the list are comma separated.

Listing 1.5 builds on the infrastructure of Listing 1.4 to define the LOLA equivalent of the C preprocessor max macro.

**Listing 1.5 Max lexi in LOLA**

```
@Find
  max(@Expression(x), @Expression(y))
@run {
  x = '('< + str(x) + ' ')
  y = '('< + str(y) + ' ')
}
@replace
  (@(x) > @(y) ? @(x) : @(y))
```

The lexi in the listing has three sections: the starting @Find section defines the pattern which triggers the proxy, the @run section which defines a bunch of Python commands to be executed upon triggering, and the @replace section defining the replacement of the matched sequence.

Sections are just elaborators of a lexi. A section whose name is capitalized, @Find in the listing, creates an empty lexi, and attaches itself to it as one of its elaborators. Non-capitalized sections, @run and @replace here, attach as elaborators to the preceding lexi.

\(^4\)In fact, the LOLA specification is slightly different, since C preprocessor does not worry about square- nor curly- brackets
The pattern in the @Find section matches a sequence beginning with the max identifier, followed by an opening parenthesis, an expression as previously defined, a comma, another such expression and ending with a closing parenthesis.

Whenever a pattern is matched, LOLA creates a PYTHON object, called the reifying object, which can be manipulated in the @run section. The @Expression(x) sub-pattern in the @Find section invokes the @Expression pattern defined above, and storing the result of the match into an instance variable named x in this reifying object. Instance variable y is similarly defined in this object.

Now, the @run section wraps these two instance variables in parenthesis. The @(x) and @(y) references to these variables in the @replace section yield the correct parenthetical version of these.

LOLA makes it easy to define a version of max taking a variable number of arguments, as done in Listing 1.6. Notice that even though variadic macros exist in the C preprocessor, they cannot be used to create a variadic max macro.

### Listing 1.6 Max lexi in LOLA with variable number of arguments

```python
@Find
max(
    @OneOrMore @Expression(x) @separator,
)
@run {
    xs = ['(' + str(x) + ')' for x in xs]
}
@replace
@ForEach(xs[:-1])
(@xs[0]) < @(_) ? @(x) : @(xs[0]) @(')'*(len(xs)-1))
```

The pattern in this listing is exactly what it reads to be: a sequence of tokens starting with the max identifier followed by open parenthesis, and ending with closing parenthesis. In between the parenthesis pair there are one or more occurrences of the user defined @Expression pattern, separated by commas.

Occurrences of @Expression in the pattern are repeatedly stored in instance variable x, so at time of triggering x will store the last matched @Expression. However, LOLA automatically defines an instance variable which contains the list of all assignments to x during the match. This name of this variable, xs, is obtained by adding the plural suffix s to the user defined name x. The @run section uses a PYTHON programming idiom to parenthesize all elements of list xs.

The @replace section of the lexi uses a generating expression to produce C code that implements the variadic max semantics. A generating expression is a LOLA expression that generates code. The @ForEach builtin employed in this section is a useful operator in generating expressions; it uses a PYTHON iterable\(^5\) to iterate over the list. The iteration variable is named _ and it can be accessed in the body of the @ForEach by means of LOLA’s operator for escaping a PYTHON identifier, @(·).

---

\(^5\)e.g., expression `xrange` yielding multiple values by means of the PYTHON `yield` keyword
1.4 Indentation

Before proceeding to more motivational examples, note that LOLA relies on indentation for block definitions, much in the spirit of Python and Occam [16]. In Listing 1.6 the body of the @replace section is the two indented lines that follow it, and the body @replace is the subsequent indented line.

LOLA’s syntax is a combination of free style with indentation and line breaking rules. The details of implementation may be a bit complicated, but to understand LOLA code, the reader should only keep these simple rules in mind:

- the no lower indentation argument rule stating that the arguments of any LOLA keyword are never found at an indentation level which is lower than that of the keyword itself.

- the eager association rule states that keywords that elaborate other keywords (e.g., @else elaborating an @If) always attach to inner most candidate.

- the right associativity rule says that tokens and keywords found in a line associate to the left. In other words, the interpretation of the isolated such as

\[
@X \ a \ b \ c \ @Y \ d \ e \ f \ @g \ h \ i \ @Z
\]

is as if the following virtual Lisp [22] like parenthesis were written:

\[
(\@X \ a \ b \ c \ (\@Y \ d \ e \ f \ (\@g \ h \ i \ @Z)) )
\]

The above should be read with Lisp convention in mind: the first item in a list is a function whose arguments are the remainder of the list.

Right associativity in this example means that constructor @X takes four arguments: host tokens “a”, “b”, “c”, and the result of applying constructor @Y to its own arguments.

Constructor @Y also takes four arguments: host tokens “d”, “e”, “f”, and, the result of application of elaborator @g to its own arguments.

Finally, elaborator @g takes three arguments: host tokens “h” and “i”, and the parameterless constructor @Z.
1.5 Ranged Loops and Loop Unrolling

Listing 1.7 illustrates the use of LOLA for adding two new language constructs to C: ranged loops and loop unrolling.

Listing 1.7 Using @replace for defining a loop based on range syntax

```c
@Find
    for @Identifier(i) in @Literal(start) .. @Literal(end)
@replace
    for (int @i = @(start); @i < @(end); @i++)
@Find
    unroll @Identifier(i) in @Literal(start) .. @Literal(end) {
        @Any(body)
    }
    @replace
    @ForEach (range(start,end))
    {
        int @i = @(_);
        @body
    }

int main() {
    for iter in 0..2 {
        printf("now at %d", iter);
    }

    unroll iter in 0..2 {
        printf("now at %d", iter);
    }

    unroll i in 0..2 {
        unroll j in 0..3 {
            printf("now at %d", iter);
        }
    }
}
```

A usage of the ranged loop is highlighted in green in line 18. We see in the code the for keyword of C, the name of the iteration variable, and the literals that indicate the bounds of the loop (i.e., iter = 0,1), separated by two dots.

The use of the unroll loop can be seen in line 22, highlighted in red. The main difference in this added syntax is that the lexi triggers on unroll, rather than on for. The semantics is also similar. The loop body is executed the number of times as declared by the range. The loop is unrolled, where each iteration is provided with the correct value of iter, the iteration variable. Lastly, starting at line 24 are two nested unroll loops. These will be expanded as expected.

Now, having reviewed the use examples of both constructs, we return to the first lexi
definition, that is, the ranged loop lexi definition. It searches for C’s keyword for, followed by an identifier (that will be declared as a C variable in the rewritten code, as explained later). Then, we see the in word and two literals, separated by two consecutive “.” tokens. Variables i, start and end in PYTHON denote the text matched by the @Identifier and the two occurrences of @Literals, respectively.

The processing of the text in the @replace section also reminds the processing of string literals in shell, e.g., Bash. Default processing is verbatim. There are provisions for substitutions, most important of which is string expansion using the @ keyword.

Here we see that the previously defined i variable is used to name the loop iterator. The start variable is used to set the initialization value of the iterator, and the end variable is used to test in the for loop condition.

The output of Listing 1.7 is presented in Listing 1.8.

The second lexi, unroll, starts very similarly to the first, except it looks for the word unroll, and in the end a following curly-braced block. It is then replaced by a series of blocks, where each represents a single execution of the loop body. Each block is prepended with the iterator definition, initialized to the current value.

1.6 Directives

All LOLA code is contained in directives, which are those instructions that can be placed at the upper most, file, level of the source. A directive is either a lexi, which is typically executed at a later stage, or a generating expression which is executed immediately—the output of the generating expression replacing the generating expression directive.

Table 1.1 shows a sample of LOLA directives, comparing each of these with the corresponding C preprocessor approximation.

A lexi may start with any of its sections. Therefore, any lexi section may serve as a directive. The upper block of the table gives a sample of lexi sections and their use as directives. (The full list of sections can be found below in Table 4.3.)

Similarly, any generating expression might be used as a directive. The lower block of Table 1.1 shows a sample of constructors of generating expressions used as directives. (The full list of constructors of generating expressions can be found below in Table 4.1.)

Lexies can only appear at the top level of a file. Three exceptions exist to this rule: Directives @Include, @Import and @Splice may introduce new lexies to LOLA, only as long as these directives are in the top level of the file. I particular, lexies may not be created by other lexies.

1.7 Contributions, Challenges and Achievements

I believe that LOLA bears the potential of evolving into a tool that would be used by many programmers. In many ways, LOLA supersedes everyday tools such as “grep”, “sed”, AWK, Perl [57], Python and others, in the context of their application to software. In this sense, LOLA is a software engineering tool: When put in the hands of able programmers it makes it possible to “engineer” the software itself, modify it, enhance it, measure it, and reason about it. And, it makes it possible to do all that in a reproducible, concise and modular fashion.
Listing 1.8 The output of Listing 1.7 produced by LOLA.

```c
int main() {
    for (int iter = 0; iter < 2; iter++) {
        printf("now at %d", iter);
    }

    int i = 0;
    printf("now at %d", iter);
    {
        int i = 1;
        printf("now at %d", iter);
    }

    int i = 0;
    {
        int j = 0;
        printf("now at %d", iter);
    }
    {
        int j = 1;
        printf("now at %d", iter);
    }
    {
        int j = 2;
        printf("now at %d", iter);
    }

    int i = 1;
    {
        int j = 0;
        printf("now at %d", iter);
    }
    {
        int j = 1;
        printf("now at %d", iter);
    }
    {
        int j = 2;
        printf("now at %d", iter);
    }
}
```
### Lola Keywords, Python Parameters, Purpose, C Preprocessor Approximation

<table>
<thead>
<tr>
<th>Lola keyword</th>
<th>Python parameter</th>
<th>Purpose</th>
<th>C preprocessor approximation</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Find</td>
<td></td>
<td>Find an extended regular expression in the input stream, and trigger actions such as @run, @replace, @delete and @log.</td>
<td>#define VERSION 4</td>
</tr>
<tr>
<td>@Description</td>
<td>&quot;&quot;&quot;Multi-line string literal&quot;&quot;&quot;&quot;</td>
<td>A textual description (comment) which may span several paragraphs.</td>
<td></td>
</tr>
<tr>
<td>@Note</td>
<td>&quot;Short string literal&quot;</td>
<td>A short, typically, one-line description.</td>
<td></td>
</tr>
<tr>
<td>@Affirm</td>
<td>(Expression)</td>
<td>Issue a warning if Python expression evaluates to False.</td>
<td>#if VERSION &lt; 4 #warning &quot;oh&quot; #endif</td>
</tr>
<tr>
<td>@Assert</td>
<td>(Expression)</td>
<td>Issue an error if Python expression evaluates to False.</td>
<td>#if VERSION &lt; 4 #error &quot;bad&quot; #endif</td>
</tr>
<tr>
<td>@Run</td>
<td>{Command(s)}</td>
<td>A Python script to execute on match.</td>
<td></td>
</tr>
<tr>
<td>@Import</td>
<td>&quot;Short string literal&quot;</td>
<td>Insert the contents of the given filename (a file is only imported once, obviating the need for header protection).</td>
<td>ifndef STDIO_H define STDIO_H include &lt;stdio.h&gt; endif</td>
</tr>
<tr>
<td>@Include</td>
<td>&quot;Short string literal&quot;</td>
<td>Insert the contents of the given filename.</td>
<td>include &lt;stdio.h&gt;</td>
</tr>
<tr>
<td>@Splice</td>
<td>(Expression)</td>
<td>Execute a system, shell command, and insert its output.</td>
<td></td>
</tr>
<tr>
<td>@If</td>
<td>(Expression)</td>
<td>Insert replacement if expression evaluates to True.</td>
<td>#if VERSION &lt; 4 ... endif</td>
</tr>
</tbody>
</table>

**Table 1.1:** A sample of Lola directives

A combination of external script and other software tools can, and are used, to, e.g., compute the Halstead metrics, or enforce the style rule (practiced mercilessly in Google) that C++ [53] classes should never inherit from more than one class. With LOLA, the same deeds are carried out in a tongue that programs already speak: the host programming language itself, albeit augmented with LOLA’s syntax.

The central objective of my work was to make LOLA elegant, readable but also writable. Great effort was made to make LOLA terse yet not cryptic, and, above all, as language independent as possible. I learned the hard way that relying on existing language paradigms is not the way to go. I believe that LOLA invents its own little paradigm made of by assembling concepts such as patterns, generating expressions, lexies, etc.

Concepts such as regular expressions and language embedding are more familiar, other, such as generating expressions are less, and others, such as elaborators, “match all possible” (rather than lazy or eager) semantics of regular expressions, or fixed point semantics of search
and replace are quite novel. The particular way in which LOLA puts all of these together is what makes it special.
Outline

Having seen a bit of LOLA’s expressive power, we first turn to comparing it in greater detail to related work (Chapter 2). More of LOLA capabilities are demonstrated in the series of examples in Chapter 3.

Chapter 4 presents the main grammatical elements of LOLA, furthering the discussion of notions such as patterns (extended regular expressions) lexies, generative expressions, etc. This section also explains two architectural issues: how LOLA interacts with PYTHON, and how LOLA is configured to recognize the tokens of a specific host language. More details on host-language configuration can be found in Chapter 5.

Chapter 6 explains the novel modular means for the definition of LOLA’s own grammar. In Chapter 7 the order in which rewritings are applied is described.

Chapter 8 concludes, discussing the current implementation, and drawing directions for future work.
CHAPTER 1. INTRODUCTION
Chapter 2

Related Work

2.1 Embedded Languages

Preprocessors are traditionally implemented as embedded languages. In the C preprocessor, for example, the embedded text begins with the hash character and ends with an un-escaped end-of-line character. We see implementation as an embedded languages also in other preprocessors, mentioned hereafter.

The concept of embedded languages is not limited to preprocessors. In several ways, string literals and comments in programming languages are an example of embedded languages: they begin at a designated and easy to recognized point (e.g., in the character sequence /* in the case of C comments); they end at a likewise distinguishable location (character sequence */ in the case of C comments); and, their content is governed by rules which are different than those of the host language (think, e.g., of JAVADoc comments).

But, the concept of embedded languages reaches far beyond comments and string literals. Perhaps the most famous example is underlying concept of PHP [35], by which commands in a full blown programming language can be embedded in HTML [8] pages. It would not be an exaggeration to say that together with ASP [2] and JSP [5] which adopted the same idea, they revolutionized the design of Internet servers.

LOLA in comparison, is not an embedded language on itself. To the contrary, it is a language that embeds in it not one but two languages: the host language it processes and PYTHON. Snippets of the host language do not represent computation to LOLA, nor can they be viewed as ordinary text. To LOLA, the host language is a stream of tokens which can be manipulated by extended regular expressions.

Similarly, PYTHON code is embedded in LOLA; this code makes almost all computation that LOLA does: variables, iterations, function calls etc. The only executional part of the LOLA are generating expressions. By shamelessly borrowing from on PYTHON, the design of LOLA was less excruciating than what it had to be if LOLA had to define the syntax of arithmetical expressions (as done in the C preprocessor), or include a full blown language features such as variables and control flow (as in AWK).

2.2 Preprocessing

The most famous text preprocessor, at least in the domain of programming languages, is arguably the C preprocessor [18], often identified by its acronym, CPP. It was introduced in
According to D. M. Ritchie [46]:

“Many other changes occurred around 1972–3, but the most important was the introduction of the preprocessor, partly at the urging of Alan Snyder [50], but also in recognition of the utility of the file-inclusion mechanisms available in BCPL [44] and PL/1 [27]. Its original version was exceedingly simple, and provided only included files and simple string replacements: #include and #define of parameterless macros. Soon thereafter, it was extended, mostly by Mike Lesk and then by John Reiser, to incorporate macros with arguments and conditional compilation. The preprocessor was originally considered an optional adjunct to the language itself. Indeed, for some years, it was not even invoked unless the source program contained a special signal at its beginning. This attitude persisted, and explains both the incomplete integration of the syntax of the preprocessor with the rest of the language and the imprecision of its description in early reference manuals.”

(references added in above, I.E.Z).

The mentioned features, importing files (#include), #define macros and conditional compilation (#if, ...) are the key preprocessing abilities of the C preprocessor.

The C preprocessor is almost entirely independent of the host programming language [19], and can be used for other purposes. Therefore, it can be viewed as an independent programming language.

As a programming language, the C preprocessor is limited. It is possible to name parameters to a macro, however, the C preprocessor cannot perform iterations (loops) and no conditionals (on macro parameters). As well, C preprocessor does not allow recursive structure, i.e., one cannot call the macro in its own definition (without some “magic”\(^1\)), and has a minimal “library” (the C preprocessor supports only __FILE__, __LINE__ and very few more).

C preprocessor somewhat compensates for features of missing in C, liked named-values and module and file management. However C is not only very specific to C, but also very specialized for doing C enhancements but not other, Syntactic sugars such as operators # and ## and variadic macros, are the exception, demonstrating the limitations, not the rule.

The PL/1 preprocessor has several built-in types (CHARACTER and FIXED) and directives, and is somewhat similar to the C preprocessor in several respects.

M4 [20] is a more general preprocessor, not related to any specific language. It accepts only strings and manipulates them. It allows a more complete preprocessing, by allowing recursive calls for macros, and thus enabling loops. It also adds conditionals. The M4 preprocessor manual [21] describes some preprocessors related to M4. An attempt to improve it was made, named M5 [47].

A literature search revealed many preprocessors using a general purpose programming language for creating code by redirecting the output to the source code file. A simple example is using PHP as a preprocessor, as in the following example:

\(^1\)http://jhnet.co.uk/articles/cpp_magic
```c
#include <stdio.h>

int main()
{
    <?php
    for($i = 0; $i < 20; $i++)
        echo 'printf("%d\n",\$i.);';
    ?>
}
```

Some preprocessors use Python (e.g., Cog\(^2\), jinja\(^3\), django\(^4\), PYM \(^5\), pyexpander\(^5\), pypreprocessor\(^6\), Preprocessor in Python\(^7\) and Cheetah\(^8\)) to run and expand macros. As such, they provide powerful tools for macros. Others, such as PerlPP\(^9\), use Perl.

The *Generic Preprocessor*\(^10\) offers yet another approach, by which code segments marked by `#exec command` are executed by the preprocessor and are replaced by their output.

`Gema`\(^11\), also called “The general purpose macro processor”, is a veteran (first public release of Gema was in 1995) open-source project which extends the notion of preprocessing in the following way: instead of firing the preprocessor only at specific, well marked locations within the code, Gema applies a *context matching* preprocessor. To define a macro one would declare a rule to match. Then a matched input in the file will be replaced with the appropriate output. E.g., defining the macro

```
ADD * TO *.=2 \:\=\=-2 +-1;
```

over the input

```
ADD ITEM TO SUM.
```

would yield

```
SUM := SUM + ITEM;
```

The JPP: A Java Preprocessor \(^31\) suggests a preprocessor for Java that modifies and analyses the code. It, as it states, beautifies the code by reformatting it. It also does code evaluation in the sense of checking conformance to standards, analysis according to complexity metrics of the code and the documentation and conformance to object-oriented design principles. It can encourage specifying contracts (preconditions, postconditions and invariants) for classes, interfaces. The preprocessor can also generate documentation in HTML format.

\(^3\)http://jinja.pocoo.org/, jinja by Armin Ronacher.
\(^4\)https://docs.djangoproject.com/en/1.7/topics/templates/, Django Software Foundation.
\(^5\)http://pyexpander.sourceforge.net/, pyexpander by Goetz Pfeiffer.
\(^6\)https://code.google.com/p/pypreprocessor/, pypreprocessor by Evan Plaice.
\(^7\)http://orbeckst.github.io/GromacsWrapper/gromacs/core/fileformats/preprocessor.html, Preprocessor in Python by Evan Plaice and Oliver Beckstein.
\(^8\)http://www.cheetahtemplate.org/, Cheetah by Tavis Rudd.
\(^10\)https://math.berkeley.edu/~auroux/software/gpp.html, GPP—General Purpose Preprocessor by Denis Auroux.
\(^11\)http://gema.sourceforge.net/new/index.shtml
C++ offers a template system that instantiates at compile time. This mechanism has been proved to enable Turing complete computations in compile time [55]. As a consequence of this given power, a major effort has been invested in developing more and more tools to use this feature. The main advantage of using templates for compile time programming is their availability. However, it is very complex to write such programs, and maybe even more to debug [43] them. An example of its usage is found in the Boost Meta-program Library and in extensions [49] made for it.

The importance of the ability to expand a language is derived from several issues. Suppose a programming language does not provide a syntax to initialize a list (for example [1, 2, 3]). Programmers might decide to use strings for initialization (e.g. "[1, 2, 3]"), and therefore leading to correctness, performance, security, and usability issues.

Hygienic Macro Expansion. A common problem with macros is that they might capture names that are already used in context. This issue can lead to hiding variables and referring to the wrong variable. This is the kind of mistakes hygienic macros [32] are meant to solve. Sometimes it imposes a larger problem, where ambiguities in the lexical analysis stage force the JAVA SCRIPT [15] lexical analyzer and parser to be intertwined [10]. A solution for program transformations (that may cause problems when variables hide other variables with the same name) was introduced [13].

Macro Safety. Most macro systems treat arguments as strings. They do not know the host programming language, and do not type the arguments to macros—they do not consider whether the argument is an expression, an identifier, a constant or something else. The Marco preprocessor [34] wishes to add this distinction to the macro language, while not strongly coupled with the language. The users of macros are struggling with the usage of the macros (and it’s error reporting). Writing a macro system for each language from scratch is hard and unnecessary. Marco introduces a way to reduce the coupling of the language to the macro system.

Preprocessors are a useful tool. There is a degenerate version (mainly including #define and #ifdef) built-in in C# [25] and similar versions for JAVA [12, 13], one also enables #include [14].

Similar preprocessors to the C preprocessor assembly languages [3]), further stressing its importance. More preprocessors resemble the C preprocessor, allowing to define variables and to use if statements, include files, add to the start or end of a file, generate messages and warnings and use defined strings in source [17].

2.3 Other Related Fields

Domain Specific Languages The usage of PHP above relates to another subject, Domain Specific Languages, such as PHP. The above example shows some usage of PHP as a

preprocessor for C, but it demonstrated a larger idea. In PHP, code is generated using blocks of code written inside the surrounding HTML code (in a `<php>` tag in code):

```html
<html>
Hello
<br>
    c=c+1; print c;
</php>
</html>
```

**Reflection.** The ability to inspect the written program through code is a very strong feature, however costly in runtime. A solution was suggested for both JAVA [26, 40] and C# [14]. Both use a predefined syntax for code generation. A similar solution is the Lombok project, that uses JAVA’s annotations syntax to create source code transformations.

**Python Indentation**  LOLA uses indentation to define blocks of code, as in PYTHON. The idea is simple, and familiar also in other programming languages. Traditionally, a code block is started with a keyword such as `begin` and ends with a `end` (or similarly `{ }`). However, a common convention is to indent the text in the block to align. The idea is to discard the keywords and use the indentation to define a block, where a new level of indentation starts a block, until the next non-empty line of an older indentation.

**Code transformation.** Coccinelle [42] is a transformation engine which provides the language SmPL (Semantic Patch Language) for specifying desired matches and transformations in C code. It resembles to LOLA in that it recognizes patterns in the host language source code and outputs an automated result. Coccinelle can also be used to find bugs in the code.

**Aspect Oriented Programming.** Aspect-Oriented Programming [30] can also be thought of a means for automatic code transformation, code is generated to be appended before and after a call (from a certain place or from all) to a function:

```c
int main() {
    printf("world");
}

before(): execution(int main()) {
    printf("Hello ");
}

after(): execution(int main()) {
    printf(" from ACC! \n");
}
```

The idea is to separate code which is different in essence, but should be executed one after the other. A key example is a logger, which should execute immediately after the code that performs the logged action, but is not directly related to the log. This is also useful for debugging (to print something when a function is executed to notify the programmer).

---

18 [https://projectlombok.org](https://projectlombok.org)
Language Extension. Language extensions come as a language evolves. However, there are solutions that embed language extensibility in the language definition itself. SugarJ \cite{12} is a language extension tool that allows extending Java using libraries. However, a tailor made version of Java compiler was used in order to make SugarJ possible.

AST Transformation. Programming languages usually use a context free grammar in order to build an AST.

An example of an AST transformation tool is JavaScript. The HTML code constitute the DOM \cite{38} tree, while JavaScript manipulates it. JavaScript is forgiving in regard to the DOM tree.

LOLA does not use an AST of the host language. Rather, it treats the code as a token stream. The reason for this decision is that AST transformation imposes a higher overhead for each lexi definition. Transforming the AST would first require building an AST. Hence, lexies would define new context free grammar rules that define the AST.

LOLA is designed to fill the gap between a simplistic preprocessor and a full code analyzer and transformation tool. LOLA promotes a simpler approach where it uses regular expressions and balanced parenthesis, that covers many cases.

Several tools enable the transformation of the AST. However, all these tools are spectacularly cumbersome, and are very sensitive to language dialects.

Examples of these include Refine \cite{56}, Eclipse’s JDT\footnote{http://www.eclipse.org/jdt/apt/index.php}, and C to C transformation tools. In addition, Michael Ernst had some full issue with non-standard types for Java\footnote{Some claim on the AST: look for michael’s jrc java change request}.

EMF Disaster. LOLA is not designed to create wide-scale changes. Such changes are hard to track and reason about. Therefore, lexies should be simple, otherwise the transformation is probably not in the right plane. A possibly solution is to limit the number of tokens that one substitution may generate. In addition, lexies should be designed in a way so that the output is readable. In this case, one may only use the resultant code, and the written could would still be usable.

Since LOLA’s lexies are short, we are not likely to get another Eclipse EMF \cite{51} disaster, where code became unusable due to abandoning the technology. EMF generates code as if there is no tomorrow, and indeed, there was no tomorrow for EMF. It is yesterday now.

?? The Cilk \cite{6} programming language adds the \texttt{spawn} and \texttt{sync} keywords to C. Removing these keywords does not influence the semantics of the program (erasure semantics).

The revolution of binary rewriting. PIN \cite{36}.
Chapter 3

Examples Gallery

Sections 1.3 and 1.5 were our crash course of LOLA. This chapter teaches LOLA some more, and presents its capabilities through a series of examples.

3.1 Introductory Examples

3.1.1 Comparison with the C Preprocessor

Listing 3.1 shows a simple use of LOLA, much in the fashion of the C preprocessor.

<table>
<thead>
<tr>
<th>Line</th>
<th>Find</th>
<th>Replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Life</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>program</td>
<td>void</td>
</tr>
<tr>
<td>3</td>
<td>begin</td>
<td>{</td>
</tr>
<tr>
<td>4</td>
<td>end</td>
<td>}</td>
</tr>
<tr>
<td>5</td>
<td>write(@Any(parameters))</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>@replace</td>
<td>((void)printf(<strong>VA_ARGS</strong>))</td>
</tr>
<tr>
<td>7</td>
<td>program main()</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>printf(&quot;answer = %d\n&quot;, Life);</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

We see a series of definitions of lexies, each includes @Find and @replace sections. After the first lexi was defined (in line 1 of Listing 3.1), every subsequent occurrence of the word “Life” in the text (but not in other LOLA definitions) is replaced by “42”.

The C preprocessor equivalent of the five lexies in the listing is

```c
#define Life 42
#define program void
#define begin {}
#define end }
#define write(...) ((void)printf(__VA_ARGS__))
```
We thus saw some superficial, syntactical differences between LOLA and the C preprocessor: Definitions use @Find rather than #define, and an explicit @replace keyword is written (instead of the implicit break at a whitespace); the C preprocessor offers the variadic argument construct “...”, which matches a sequence of zero or more parenthesis balanced expressions separated by commas, a purpose served by @Any(...); etc.

A more fundamental difference is that LOLA’s definitions are of tokens (rather than identifiers in the C preprocessor). With definitions such as

```
@Find { @replace BEGIN
```

one can use LOLA for the converse of Listing 3.1, that is, writing C in a Pascal’ish style.

### 3.1.2 Halstead’s Code Metrics

Tokens in LOLA are organized in a hierarchical taxonomy (see Section 5.4), e.g., tokens “!” and “sizeof” belong in class @UnaryOperator, which is a subclass of @Operator.

Figure 3.1 employs this taxonomy in three simple lexies, which together compute (one particular variant of) Halstead’s metrics [24] of the code in a C file. The subject files are the C files in the current directory. These are @Spliced into the current file by the forth directive.

**Figure 3.1 Halstead complexity measures.**

(a) A pattern to detect Halstead’s operands

```
@Find(HalsteadOperand)
 Either @Identifier
 or @Literal
 or @TypeKeyword
```

(b) A pattern to detect Halstead’s operators

```
@Find(HalsteadOperator)
 Either @Operator
 or @Punctuation
 or @NonTypeKeyword
```

(c) Print a summary of all of Halstead metrics when the end-of-file is reached

```
@Find @EndOfFile @run {
 operands = [str(x) for x in HalsteadOperands]
 operators = [str(x) for x in HalsteadOperators]
 from math import log
 N1 = len(operands)
 N2 = len(operators)
 n1 = len(set(operands))
 n2 = len(set(operators))
 n = n1 + n2
 N = N1 + N2
 V = N * log(n, 2)
 D = (n1/2) * (N2 / n2)
 E = D * V
 }
 @log ( 
   __FILENAME__, N1, N2, n1, n2, n, N, V, D, E 
 )
```

(d) Includes all C files in the current directory

```
@ForEach(glob.glob("*.c"))
 @Splice('cat ' + _)
```
3.2. IMPLEMENTING NEW LANGUAGE CONSTRUCTS

The lexi named HalsteadOperand (Figure 3.1 (a)) matches all subsequent occurrences in the input of tokens in the @Identifier, @Literal and @TypeKeyword classes. Lexi HalsteadOperator (Figure 3.1 (b)) matches tokens in the @Operator, @Punctuation, and @NonTypeKeyword classes.

With the absence of a @replace clause (section in the LOLA lingo), the two patterns do not modify the code. However, each match of the lexi named HalsteadOperand creates an instance of a PYTHON object. These instances can be accessed through a list stored in a global variable, whose name is the (simple\(^1\)) plural form of the name of the lexi, HalsteadOperands. Whenever a match is made, the new object is appended to the list. A similar global list exists for the HalsteadOperator lexi.

The third lexi (Figure 3.1 (c)) matches at the end of the file (@EndOfFile), and then it calculates Halstead’s metrics (in the PYTHON code listed in the @run section). Storing the text of the operators and the operands in two appropriate lists, it uses the length of the set of those variables to access the number of distinct items in those lists, as required by the metrics. The @log outputs the string representation of the tuple composed of the filename and the different measures to stderr.

After the three lexies comes the subject code. The @ForEach directive in Figure 3.1 (d) is used to iterate over the files in the current directory with .c extension. Then, they are @Spliced into the current files by invoking the cat utility.

3.2 Implementing New Language Constructs

A significant virtue of LOLA is the freedom to create constructs which look natural in the host language. In these examples we would like to emphasis this characteristic. Each example adds a new feature to the host language, while they all blend seamlessly.

3.2.1 Switch Construct

In Chapter 1 we saw an example where it took 20 years to add the switch on Strings to the JAVA language. Listing 3.2 implements it using LOLA.

\(^1\) the plural form of x is xs (the simple addition of the letter s) regardless of the English grammar that may rule otherwise
In this example a lexi is defined to match a `stritch` construct (string `switch`), that is very similar to a regular `switch` in `Java`: A `stritch` keyword, followed by what we wish to switch over, and then `@NoneOrMore` blocks, each contains several (`@OneOrMore`) cases, and a `body` to execute in each case, terminated by the word `break`.

However, we `@replace` a matched code by a series of appropriate `if` statements, serving the same purpose of a `switch` (we do not use `if else` statements). `@ForEach` of the `blocks` (the iterated value is named `_`) an `if` statement is generated, separating the cases (each is a single call to `equals`) with a `||` operator. If the `Java` condition will evaluate to `true`, the body of the case will be executed.

Notice that this lexi can be added to any version of `Java`—either before or after the addition of the `switch` on `Strings` to the language. In addition, an advanced user may decide to implement this lexi so that the code would execute a more efficient algorithm at runtime.

To ensure a match whenever the `stritch` word appears, the lexi `@anchors` it\(^2\). In addition, an `@assert` section is added, assuring that any string case appears only once.

Keep in mind that in `Lola` we use the plural form of a variable name in order to access the list of variables with that name. Here, the `@Sequence` keyword creates a new `Python` variable, named `block`. It captures the scope of its inner pattern into the variable. However, since it is expected to appear several times in the pattern, there might be more than one instance of this variable. A list of the instances is accessible via `blocks` (a simple addition of the letter `s` to the variable’s name). Using the former means the last entry of the list.

Notice the usage of `@` followed by an expression on `Python`. It outputs the `__str__`\(^3\) result of the expression. This is convenient enough for the majority of cases.

---

\(^2\) meaning that if the word will appear, and the pattern will not match, we will be notified

\(^3\) the `Python` predefined method for converting an object to a string
3.2. IMPLEMENTING NEW LANGUAGE CONSTRUCTS

3.2.2 Poor Man’s Inheritance in C

The C programming language is not object oriented, but a poor-man’s substitute for inheritance can be manually added to the language. Listing 3.3 demonstrates how we can automatically add fields to a struct in C to emulate inheritance.

**Listing 3.3 Poor man’s inheritance in C**

```c
@Find(Struct) struct @Identifier(name) { @Any(fields) }

@Find(InheritanceStruct)
    struct @Optional @Identifier(name) extends @Identifier(inherited) {
        @Any(fields)
    }
@anchor extends
@run {
    fields = fields +
    next([x.fields] for x in Structs + InheritanceStructs
        if x.name is inherited)
}
```

The example consists of two lexies. The first searches for any new struct definition. Whenever a match is found, the new instance is added to the Structs global list.

The second lexi adds the inherited fields to the extended struct. It looks for a pattern to that of the first lexi, except that it searches for the token extends followed by an @Identifier. Only then we expect the fields to be listed. In the case an extending struct is left unmatched, an error will be issued: The @anchor on the token extends flags an error whenever the token was found in the text, but the pattern did not match.

Recall that a list named InheritanceStructs is also created.

The addition of the inherited fields occurs at the @run section. By setting the fields variable we change the appropriate part of the matched text. We use the fields of the parent struct, while seeking for it both within the regular structs and the structs recorded by the second pattern.

3.2.3 Sets as Bit Masks

Sets can be represented by a bit vector, or in the case of less than 32 possible members, a single integer (assuming an integer consists of 32 bits). The integer can store for each of the possible members of the set a bit value that indicates whether is it indeed a member of the set or not. These values are stored as a bitwise or of constant values, each with a single bit on. The concept is commonly known in C as flag constants. Listing 3.4 creates a specialized enum for the use case.
In the `bitenum` construct we catch `@NoneOrMore` entries, separated, and possibly terminated, by a comma. For each of these entries, we create an entry in an `enum` whose value is a bit-shift of `0x1` by the index of the entry. The lex `@asserts` that no more than 32 entries are declared.

### 3.3 New Keywords in Java

In order to reduce code dependency and for the comfort of the programmer, several new lexies are introduced in this section. The first introduced lexies are used in those that follow.

#### 3.3.1 Auxiliary Lexies for Java

In Listing 3.5 sets some auxiliary lexies, each defines a pattern to match a part of the JAVA programming language. The lexies’ names describe their appropriate roles.
The lexies will not be explained. However, a worthwhile mention is that each named lexi (where a name is specified in @Find) can be referenced, and serve as a sub-pattern of the defined pattern. Moreover, this sub-pattern may be named, e.g., @JavaType(returnType) defines a sub-pattern named returnType that matches the pattern defined in JavaType.

Listing 3.6 uses the defined lexies to build the compile time global stack (named java_methods) of methods in the code. Whenever we meet a method start we record the new method, and when the method definition is complete we pop the definition from the stack.

### Listing 3.6 Auxiliary stack of methods in Java

```java
@Run {global java_methods; java_methods = []}

@Find @JavaMethodBegin @run {java_methods.append(self)}
@Find @JavaMethodWhole @run {java_methods.pop()}
```

### 3.3.2 Invoke an Overridden Method (Super)

The first lexi we see is one that makes it possible to delegate to an overridden method. The newly created Super keyword, defined in Listing 3.7, invokes the parent’s overridden method with the arguments of the current method.

### Listing 3.7 Augmenting Java with a Super keyword

```java
@Find Super @run {m = java_methods[-1]}
@replace super.@(m.name) (@(','.join(m.argNames)))
```

### 3.3.3 Current Method Name (me)

To get the name of the current method one may use the me lexi, defined in Listing 3.8.

### Listing 3.8 Augmenting Java with a me keyword

```java
@Find me
  @run {m = java_methods[-1]}
  @replace @(m.name)
```

### 3.3.4 Numbered Arguments

In many scripting languages one may access the arguments of a method using their index. We present a lexi (Listing 3.9) that makes it possible to use a similar concept in Java. Negative numbers index the arguments the same way as in Python\(^4\), and requires no special handling.

\(^4\)(access elements from the end of the list, counting backwards)
### Listing 3.9 Augmenting JAVA with numbered arguments

```java
@Find
    @OneOrMore @Sequence(hash) #
    @Optional @Integer(i)
@run {
    m = java_methods[-len(hashes)]
    i = i if hasattr(self, 'i') else 1
    v = ','.join(args) if i is 0 else args[i-1]
}
@replace @(v)
```

A sequence of one or more hash characters is matched. We optionally match a following integer. The number of hash characters indicates the method we refer to—the current method is addressed using one hash character, while the outer method is addressed using two, and so forth. If no integer was specified we have the first argument of the method. We check this condition using the `hasattr` function in Python. Otherwise we access the \(i\)'th argument, where \(i\) is the specified integer, as long as \(i\) is not 0. Using the integer 0 means we wish the lexi to be replaced by the entire list of arguments of the method.

#### 3.3.5 Current Class Name (This)

The name of the current class can be retrieved in a similar way to the way we track methods. **Listing 3.10** shows how to create a lexi that matches the identifier `This` and changes it to the current class name. The lexi uses very similar concepts to those portrayed earlier.

### Listing 3.10 Augmenting JAVA with a This keyword

```java
@Run {global class_names; class_names = []}

@Find class @Identifier (className)
    @run {class_names.append(className)}

@Find
    class @Identifier (className)
        @Optional @ExtendsPattern
        @Optional @ImplementsPattern
    { @Any }
    @run { class_names.pop() }

@Find This @replace @(class_names[-1])
```
3.4 Miscellaneous Examples

We end our tour in the gallery with two examples, where the purpose of the first is debugging, and the second handles a very repetitive source code.

3.4.1 Monitor an Expression (monitor)

Listing 3.11 demonstrates a method to create a monitor function—the preprocessor passes an expression’s describing a string and its value. A function, defined in the beginning of the file, is passed both these parameters, prints the former followed by the latter, and returns the latter. It creates the effect of logging the expression and its value.

Listing 3.11 Augmenting JAVA with a monitor method

template<typename T>
static auto &__monitor__(const char * x, T && y) {
    std::cout << x << y << std::endl;
    return y;
}

@Find monitor(@Any(arg))
@replace
    __monitor__(" +
        arg.replace('\\', '\\\\').replace('"', '"') +
        '"')
)

Notice that the lexi escapes certain characters in order to create the appropriate C++ string literal.

3.4.2 Avoid Repetitive Code

In Listing 3.12 we create lexies. These lexies are used in Listing 3.13 to immensely shorten the repetitive code of a class that boxes JAVA’s value types into reference types\(^5\).

\(^5\)https://bitbucket.org/yossi_gil/services/src/c8bb342e5ea89c91931e078a7a8a6bf7f7be7112/trunk/src/11/ac/technion/cs/ssdl/utils/Box.java?at=master fileviewer=file-view-default
Listing 3.12 Box it lexi for boxing JAVA's value types into reference types

```java
@Find
BOX_IT @Identifier(boxType) @Optional to @Identifier(boxToType);
@run {
    if not hasattr(self, 'boxToType'):
        boxToType = boxType.title()
}
@run {
    v = str(boxType)[0]
    vs = v + 's'
    comment = ""
    /**
     * Box a <code><b>{0}</b></code> into a {{@link {1}}}
     * object.
     * @param {2} some <code><b>{0}</b></code> value
     * @return a non-<code><b>null</b></code> {{@link {1}}} with the value of <code>val</code>
     */
    commentArr = ""
    /**
     * Box an array of <code><b>{0}</b></code>s into an array of {{@link {1}}}s.
     * @param {2} an array of <code><b>{0}</b></code>s
     * @return an array of {{@link {1}}} of the same length as that of the parameter, and such that it in its
     * the <tt>i</tt><em>th</em> position is the boxed value of the parameter
     */
}
@replace
@((comment))
public static @((boxToType)) it(final @((boxType)) @((v))) {
    return cantBeNull(@((boxToType)).valueOf(@((v))));
}
@((commentArr))
public static @((boxToType))[] it(final @((boxType)) @((vs))[]) {
    final @((boxToType))[] $ = new @((boxToType))[@((vs)].length];
    for (int i = 0; i < $.length; ++i)
        $[i] = it(@((vs))[i]);
    return $;
}
```
3.5. REWRITING

Listing 3.13 A use of the BOX_IT lexi introduced in Listing 3.12

```java
public enum Box {
    BOX_IT boolean;
    BOX_IT byte;
    BOX_IT char to Character;
    BOX_IT double;
    BOX_IT float;
    BOX_IT int to Integer;
    BOX_IT long;
    BOX_IT short;
}
```

The lexi matches a BOX_IT statement which is followed by the type to box\(^6\), and an optionally supplied type to box to. If the boxToType is not specified, the code just titles the boxType. The code for the comments is generated solely using PYTHON.

3.5 Rewriting

Recall the example in Listing 1.7. Another course of action to achieve the same effect of a range based loop is by using the code in Listing 3.14.

Listing 3.14 Using @run for defining a loop based on range syntax.

```c
#include <stdio.h>

@Find
for @Identifier(i) in @Literal(start) .. @Literal(end) @run {
    self = 'for (int ' + str(i) + ' = ' + str(start) + '; ' \
    + str(i) + ' < ' + str(end) + '; ' + str(i) + '++)'
}

int main() {
    for iter in 0..2 {
        printf("now at %d", iter);
    }
}
```

A local variable named self is initialized to the match’s reifying object. At the end of the execution of all of the code that is associated with a match, the contents of this variable are written to the output file, replacing the text that was matched. If self, or any of its attributes, indeed was changed, than the replacement changes the text (otherwise, if no changes were made to self, then nothing will change in the output file).

\(^6\)for the sake of this example we assume value types names are defined as identifier tokens.
In the example, the code sets the string that was matched by the pattern to a new string, and therefore a replacement of the code occurs.

3.6 Reflection

Listing 3.15 defines a lexi that matches an \texttt{enum}\textsuperscript{7}.

\begin{verbatim}
@Find(EnumPattern)
enum @Identifier(name) {
    @NoneOrMore @Identifier(item) @Optional = @Literal(value)
    @separator ,
    @closer @Optional ,
}
\end{verbatim}

Listing 3.16 is triggered upon encounter of a C preprocessor-macro-call like pattern named \texttt{CREATE_DICTIONARY}, receiving two parameters. The first parameter is the name of an \texttt{enum}, and the second is an identifier that will hold a resultant array.

The lexi generates a C array that will hold information on each of an \texttt{enum}'s entries. Each element of the array stores the name and the numerical value of an entry of the \texttt{enum}. Code can then iterate over the array, create "\texttt{to\_string}" and "\texttt{from\_string}" functions, and more.

The example builds upon Listing 3.15 in assuming there is a list that describes the declared enumerations. Indeed, each match of the previous lexi adds an instance to the list named \texttt{EnumPatterns}.

The \texttt{@ForEach} keyword accepts a \texttt{PYTHON} expression to iterate on. The loop variable is named \_\texttt{\_}, and we use the expression evaluation keyword (\texttt{@(E)})

\textsuperscript{7}note that the lexi is not fully comprehensive, since it assumes the values are only literals
3.7 Island Grammars and typedef

Island grammars [41] focus on the parts of the programming language that are of interest to the current task, rather than describing the entire programming language syntax tree.

In Java, one can use the same name for a class and for a function or variable. However, it is possible, in most cases, to deduce the role of an identifier by other criteria. This makes it possible to identify with island grammars all cases of a names used as a name of a type. We can create Lola lexies to identify these, as in Listing 3.17.

The lexi in the figure is not complete; it fails to identify occurrences of the aliased type in dot separated names, e.g., `il.ac.technion.cs.Lola.go`, and in type parameters of generics, e.g., `List<T super Lola>` or even in `Cell<Lola>`.
The latter failure can be remedied with a bit of definitional effort. However, with LOLA unable to do full type analysis, it is impossible to automatically distinguish type-name components in dotted names from other kinds of components such as package- and field- names. This predicament however may be ameliorated by assuming capitalization conventions.

3.8 Before Function

Aspect oriented programming enables inserting code before another function, where the former is written in another function. Listing 3.18 demonstrates a simplified similar effect using LOLA.

Listing 3.18 A lexi that simulates aspect oriented programming

```leven
@Find(FunctionPattern)
    @Identifier(retValue)
    @Identifier(functionName) ( @Any ) {
        @Empty(beforeFunctionBody)
        @Any
    }
@run {
    beforeFunctionBody = 'printf("got here!\n");'
}
```

3.8.1 Adding an Anchor

Listing 3.19 demonstrates LOLA’s expressiveness by matching the structure of a class declaration in JAVA. However, it also demonstrates the difficulty in writing patterns. The @anchor keyword assumes all occurrences of the provided word are captured by the pattern. This is useful to avoid unnoticed errors. For example, there is an “error” in the pattern (well, at least one). The keyword public does not have to appear before the final keyword. However, if these keywords are only used in the source code under this convention, the lexi is suitable. Moreover, if we do not follow this convention, an error is issued. This allows us to write patterns for the host programming language even if we do not know it in its entirety.
3.8.2 Adding an Example

Writing unit tests is a useful tool for the pattern writer to make some “sanity checks” to check the correctness of the pattern. The code accepted by the @Example message is tested to match the specified pattern. If the test fails, LOLA issues a warning.

In Listing 3.20 the test fails since there is no definition of the integer i in the example, but just usage of the variable. We can also add a @resultsIn section, where we can specify the output of the example (in Listing 3.20, since we did not change the text, the output should be the same as the input).

Listing 3.20 An example of the usage of the @example (the test fails)

```java
@Find for (int @Identifier(i) = @Literal(start));
  @Identifier(i) < @Literal(end);
  @Identifier(i) ++
@example for (i = 0; i < 10; i++)
@resultsIn for (i = 0; i < 10; i++)
```

3.9 P4 Checksum

P4 [7] is a domain specific language for programming network data forwarding. In being such, it allows specifying headers for packets, and their checksummed fields.

Listings 3.21 and 3.22 portrays a lexi for a checksummed field in an IPv4 header8. The required outcome can be seen in the @resultsIn section in Listing 3.22. Observe how repetitive the code is. Compare to the code in the @example section in the same listing. The suggested code is both readable and concise. It draws inspiration from the concept of attributes in other programming languages. However, it is merely a lexi that was added without any P4 language extension.

8https://github.com/p4lang/tutorials/blob/master/p4v1_1/simple_router/p4src/simple_router.p4
The main sections of the definition of the lexi are given in Listing 3.21. As necessary, a programmer may add more options to customize the lexi to support other “attributes”.

**Listing 3.21** An example of a lexi for creating a checksummed field in P4 (continued in Listing 3.22)

```p4
@Find header_type @Identifier(header_type) {
    fields {
        @OneOrMore
            @Sequence(field)
                @Optional
                    [checksum(algorithm=@Identifier(algorithm))]
                @Sequence(native_field)
                    bit<@Literal(nbits)> @Identifier(name);
    }
} instantiate into @Identifier(instance_name);
@anchor instantiate
@run {checksumed = [f for f in fields if hasattr(f, 'algorithm')]} (continued)
@assert (len(checksumed) is 1)
@run {checksumed = next(checksumed)}
@replace
    header_type @Identifier(header_type) {
        fields {
            @ForEach (native_fields)
                @(_)
        }
    }
} header @(header_type) @(instance_name);
field_list @(instance_name)_checksum_list {
    @ForEach (f.name for f in fields if not hasattr(f, 'algorithm'))
        @(header_type).@(_);
}
field_list_calculation @(instance_name)_checksum {
    input {
        @(instance_name)_checksum_list;
    }
    algorithm : @(algorithm);
    output_width : @(checksumed.nbits);
}
calculated_field @(instance_name).@(checksumed.name) {  
    verify @(instance_name)_checksum;
    update @(instance_name)_checksum;
} ...
```
Listing 3.22 The `@example` and `@resultsIn` sections of Listing 3.21

```p4
...  
@example

header_type ipv4_t {
  fields {
    bit<4> version; bit<4> ihl;
    bit<8> diffserv; bit<16> totalLen;
    bit<16> identification; bit<3> flags;
    bit<13> fragOffset; bit<8> ttl;
    bit<8> protocol;
    [checksum(algorithm=csum16)] bit<16> hdrChecksum;
    bit<32> srcAddr; bit<32> dstAddr;
  }
} instantiate into ipv4;

@resultsIn

header_type ipv4_t {
  fields {
    bit<4> version; bit<4> ihl;
    bit<8> diffserv; bit<16> totalLen;
    bit<16> identification; bit<3> flags;
    bit<13> fragOffset; bit<8> ttl;
    bit<8> protocol; bit<16> hdrChecksum;
    bit<32> srcAddr; bit<32> dstAddr;
  }
}

header ipv4_t ipv4;

field_list ipv4_checksum_list {
  ipv4.version; ipv4.ihl;
  ipv4.diffserv; ipv4.totalLen;
  ipv4.identification; ipv4.flags;
  ipv4.fragOffset; ipv4.ttl;
  ipv4.protocol; ipv4.srcAddr;
  ipv4.dstAddr;
}

field_list_calculation ipv4_checksum {
  input { ipv4_checksum_list; }  
  algorithm : csum16; output_width : 16;
}

calculated_field ipv4.hdrChecksum {
  verify ipv4_checksum; update ipv4_checksum;
}
```
3.9.1 Calling Convention

C compilers define compiler specific ways of declaring the calling convention used for invoking a function. To create a cross-compiler solution for using the fast call convention, suitable for both the *gcc* and *cl* (Visual C/C++) compilers, one can use the C preprocessor in the following manner:\(^9\):

**Listing 3.23** A C preprocessor macro definition and usage for a cross-compiler *fastcall* calling convention

```c
#include <stdio.h>

#define MSFASTCALL __fastcall
#define GCCFASTCALL __attribute__((fastcall))

int MSFASTCALL magic() GCCFASTCALL;
```

LOLA presents a more concise solution:

**Listing 3.24** LOLA’s Listing 3.23 equivalent for a lexi definition and usage for a cross-compiler *fastcall* calling convention

```c
@Find
  @Sequence(before)fastcall
  @Identifier(@Any)
  @Empty(after)
@anchorfastcall
@run{
  if _MSC_VER: before, after = '__fastcall', ''
  if __GNUC__: before, after = '', '__attribute__((fastcall))'
}

intfastcall magic();
```

\(^9\)http://stackoverflow.com/a/10253546
Chapter 4

Lola Concepts and Elements

LOLA execution model is that of a filter [45] (see also [17, Sect. 3.1]) converting an input stream into an output stream. In LOLA, the input stream is comprised of tokens of the host language (*host-tokens*), PYTHON code *snippets* and LOLA *keywords*. Just like the C preprocessor, the output stream is obtained from the input by applying “directives” found in the input on subsequent input.

A LOLA keyword is an identifier prefixed by the “@” escape character. The escape character on its own is also a keyword.

All PYTHON snippets must immediately follow a keyword, and must be appropriately *bracketed*. The semantics of the snippet (expression, literal, commands, etc.) is determined by the way it is bracketed (parenthesis, quotes, curly braces).

4.1 The Host Language Configuration File

Certain LOLA keywords, such as @Find and @replace are (as in Listing 1.5) reserved; other keywords can be defined by the user, just as @Expression was defined in Listing 1.4.

In addition, there is a language-dependent set of predefined keywords, denoting lexical tokens and classes of lexical tokens. The pre-definition are made in an XML configuration file. This file also defines the trivia of the host language, its comments (currently viewed as trivia by LOLA), special tokens, such as parenthesis, which must be balanced, and literal tokens (e.g., integers, strings, etc.) whose value is computed as part of the lexical analysis.

Figure 4.1 is a screen-shot taken the GUI for authoring files.

4.2 Directives

Host-tokens are normally passed on as is to the output stream. In contrast, keywords make directives out of subsequent keywords, tokens and snippets. Directives are parsed and executed as necessary by LOLA. There are two kinds of these:

**Generators** A generator is a LOLA generating expression (henceforth, GE) whose evaluation returns a sequence of host-tokens, to be placed in the output stream.

**Lexies** Lexies are a bunch of structured modules that describe augmentations to the host language.
### Lola Concepts and Elements

<table>
<thead>
<tr>
<th><strong>Keyword</strong></th>
<th><strong>Python parameter</strong></th>
<th><strong>Elaborators</strong></th>
<th><strong>Purpose</strong></th>
<th><strong>C preprocessor approximation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>@</strong></td>
<td>(Expression)</td>
<td></td>
<td>Insert the <code>__str__</code> result of the evaluated expression.</td>
<td></td>
</tr>
<tr>
<td><strong>@</strong></td>
<td>{Command(s)}</td>
<td></td>
<td>Execute the Python code.</td>
<td></td>
</tr>
<tr>
<td><strong>@Import</strong></td>
<td>&quot;Short string literal&quot;</td>
<td></td>
<td>Insert the contents of the given filename (a file is only imported once, obviating the need for header protection).</td>
<td>#ifndef STDIO_H #define STDIO_H #include&lt;stdio.h&gt; #endif</td>
</tr>
<tr>
<td><strong>@Include</strong></td>
<td>&quot;Short string literal&quot;</td>
<td></td>
<td>Insert the contents of the given filename.</td>
<td>#include&lt;stdio.h&gt;</td>
</tr>
<tr>
<td><strong>@Splice</strong></td>
<td>(Expression)</td>
<td></td>
<td>Execute a system, shell command, and insert its output.</td>
<td></td>
</tr>
<tr>
<td><strong>@Nothing</strong></td>
<td></td>
<td></td>
<td>Generates nothing for use as a place holder.</td>
<td></td>
</tr>
<tr>
<td><strong>@If</strong></td>
<td>(Expression)</td>
<td>@else, @elseIf</td>
<td>Insert replacement if expression evaluates to True.</td>
<td>#if VERSION &lt; 4 ... #endif</td>
</tr>
<tr>
<td><strong>@Unless</strong></td>
<td>(Expression)</td>
<td></td>
<td>Insert replacement if expression evaluates to False</td>
<td>#ifndef FOO ... #endif</td>
</tr>
<tr>
<td><strong>@Case</strong></td>
<td>(Generating expression)</td>
<td>@of, @otherwise</td>
<td>Switch over the evaluated expression.</td>
<td></td>
</tr>
<tr>
<td><strong>@ForEach</strong></td>
<td>(Generating expression)</td>
<td>@ifNone, @separator</td>
<td>Insert replacement for each entry in the iterable.</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.1:** The recursive structure of LOLA generators
4.3 COMPOUND OBJECTS

Figure 4.1 A screen-shot of an editing session of a LOLA configuration file for C

A lexi is composed of sections. The most important section is \texttt{@find}, which defines an extended regular expression (\texttt{RE} for short) to be searched in the input stream. This term and the term pattern will be used interchangeably.

Other more useful sections are \texttt{@replace}, \texttt{@log}, and \texttt{@run}. Lexies may also include declarative sections such as \texttt{@description} and \texttt{@see}.

When a generator directive (such as \texttt{@Include}) is encountered, LOLA pauses the habit of copying host tokens from the input stream to the output stream, injects into the output the sequence of tokens made by the generating expression, and then resumes its copying habit.

A lexi with no \texttt{@find} section matches exactly once, immediately after it is parsed. Otherwise, a lexi directive in the input is saved for later execution.

LOLA tries to match the \texttt{RE} provided in the \texttt{@find} section of the lexi against subsequent host tokens in the input stream. When a match is found, the lexi triggers and may invoke actions such as insertion of more tokens, deletion of others, record information for further processing, or, more generally, execute arbitrary \texttt{PYTHON} code.

4.3 Compound Objects

GEs and \textit{REs} are compound objects in the LOLA language, made by the application of \textit{constructors} to atomic elements and to results of prior constructors.

\textbf{Atomic elements.} Atomic elements of both \textit{GEs} and \textit{REs} come in three varieties: \textit{(i)} a sequence of one or more tokens of the host language, \textit{(ii)} keywords of LOLA taking an \textit{optional} \texttt{PYTHON} argument, and, \textit{(iii)} keywords of LOLA taking a \textit{mandatory} \texttt{PYTHON} argument.

Examples of token sequences are “\{” (encountered above in Listing 3.1),

\begin{verbatim}
> BinaryOperator @BinaryRightShift @BinaryLeftShift @:
> UnaryNot |
> UnaryNegate ~
> UnaryOperator @Plus @Minus @UnaryIncrement @UnaryDec;
> Semicolon ;
> Comma ,
> Colon :
> Dot .
> Arrow ->
> EqualsSign =
> QuestionMark ?
> Punctuation @TriDot @Semicolon @Comma @Colon @Dot @A
> DiOpenCurly <{
> OpenCurly {
SAME @DiOpenCurly @OpenCurly
> DiCloseCurly %>
> CloseCurly }
SAME @DiCloseCurly @CloseCurly
BALANCE @OpenCurly @CloseCurly
> OpenParen (,
> CloseParen )
\end{verbatim}

= (0x1 <<
Examples of keywords with optional parameter are @Any, @EndOfLine, and @Identifier. The atomic RE @Any, can be written as

```
@Any
```

meaning a sequence of any length of host tokens, or as

```
@Any(foo)
```

with the same meaning, except that the result of the match is stored in a field named “foo” in the reifying object (more on the subject later).

Examples of keywords taking a mandatory argument are @run (encountered above in Listing 3.6) and @assert (Listing 3.4).

An optional argument must be an identifier wrapped in parenthesis.\(^1\) However, there are several kinds of PYTHON arguments to keywords which take a mandatory argument:

- a sequence of PYTHON commands, to be executed, as in the case of @run,
- a PYTHON expression to be evaluated as in the case of @assert,
- a short PYTHON string literal, used in conjunction with the LOLA keyword introducing short comments

```
@Note "Lexi matches also nested and inner class"
```

and in file inclusions directives

```
@Import "augmented-Java.lol"
@Include "boilerplate.lol"
```

- a multi-line string literals of PYTHON (for long comments), used in conjunction with the @Description LOLA keyword

```
@Description """" Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam interdum, tellus in maximus tristique, sapien mauris accumsan odio, quis rhoncus velit eu eros. """
```

- PYTHON iterable, used in @ForEach, as in

```
@ForEach (range(1,3)) x,
```

or as in Listing 3.2 above.

\(^1\)for technical reason, it is required that no spaces occur before the opening parenthesis.
4.3. COMPOUND OBJECTS

**Constructors.** Constructors make compound objects out of atoms and other, smaller, compound objects. The compound objects are either *REs* or *GEs*.

A constructor is identified by a capitalized *keyword* such as @Optional, @OneOrMore and @If. A constructor takes up to three arguments, which must occur in this order

- a **PYTHON snippet**
- a **LOLA argument** which is a (possibly empty) list of host tokens and smaller compound objects,
- a (possibly empty) list of **elaborators**, such as @else and @elseif (for @If), @separator (for @OneOrMore), etc.

For example, the *GE* in Listing 4.1, is made by applying constructor @If to arguments:

- **PYTHON snippet**: os == "DOS"
- list of host-tokens and other atoms
  
  `#error AD @(date.today().year) DOS no longer supported`

- elaborators @elseif and @else

**Listing 4.1** Using LOLA to produce C preprocessor code

```python
@If(os == "DOS")
    #error AD @(date.today().year) DOS no longer supported
@elseif(os == "WIN31")
    #warning Windows 3.1 will be phased out soon
@else
    #
```

**Elaborators.** As can be seen in Listing 4.1, elaborators are similar to constructors in that an elaborator is made by a keyword, sometimes followed by a **PYTHON snippet**, and then by a possibly empty list of atoms and compound objects.

However, elaborators usually take no elaborator arguments, and an elaborator never makes, on its own, a complete *RE* or *GE*.

By convention, names of elaborators are "@camelCase"; all other keywords follow the "@CamelCase" convention.

Different constructors take different elaborators, and may place different constraints on their order. For example, @If requires that the elaborators it takes follow the pattern

\[ f_1 \cdots f_n [ e ] , \]

where \( n \geq 0 \), and where each \( f_i \), \( i = 1, \ldots, n \), is an @elseif elaborator (taking a mandatory parenthesized **PYTHON** expression and a possibly empty list of host-tokens and other *GEs*) and \( e \) is an @else elaborator (taking no **PYTHON** snippet argument; taking a possibly empty list of host-tokens and *GEs*).

No elaborators and no LOLA argument are taken by *REs* atoms, such as @Any and @NewLine. These can be thought of as degenerate constructors.

Similarly, keywords like @ (in the second line of Table 4.1), which is an atomic *GE*, can be thought of as constructors that take no LOLA arguments and no-elaborators.
4.4 Generating Expressions

The recursive structure of GE's is specified completely by Table 4.1.

Other than host tokens, atoms of GE's include @ constructs, which either splice the string value of a PYTHON expression (parenthesized form), or the standard output of several PYTHON commands (curly braces form).

The @Nothing constructor is included mainly for completeness; it produces no tokens.

Three other atoms of GE's are @Include (similar to #include of the C preprocessor), @Import (same as #include, but with built-in protection against multiple inclusions), and @Splice, which inserts the output of a system command into the output stream.

There are four constructors of GE's. Three of these are variations on the archetypal conditional commands: (i) constructor @If taking any number of @elseif elaborators followed by an optional @else; (ii) constructor @Unless, the inverse of @If, which takes no elaborators; and, (iii) constructor @Case, which takes no LOLA argument, but requires one or more @of elaborators followed by an optional @otherwise elaborator.

The semantics should be obvious: @If takes a PYTHON expression as argument, evaluates it, and if the result is True, it generates its "LOLA argument" (which in the context of GE's means a list of smaller GE's). Similarly, the "LOLA argument" to @Unless is generated only if the PYTHON expression argument to the @Unless evaluates to False.

Constructor @Case realizes a multi-way conditional comparing, in order, its PYTHON expression argument with the PYTHON expression argument of each of its @ofs.

The fourth constructor of GE's is @ForEach, taking a PYTHON iterable as argument, and a "LOLA argument". The "LOLA argument" is regenerated for each value that the PYTHON iterable yields.

This constructor takes also an optional @separator elaborator, specifying the GE to be used for separating items in the generated list, and, an @ifNone elaborator, specifying the output in the case that the list of the values yielded by the PYTHON expression is empty.

4.5 Extended Regular Expressions

Table 4.2 specifies the recursive structure of the RE's of LOLA. It should be added to what is said in the table that all constructors, just as all atomic RE's (but not elaborators), take an optional PYTHON name, wrapped in parenthesis.

When an RE matches, LOLA creates a PYTHON object that reifies the match, henceforth named F. This F contains all information about the location in the input in which the match occurred, and makes this information available to @run and other sections of the containing lex.

In particular, if a certain constructor uses a name, then LOLA creates a variable carrying that name in F, which will contain the matched input (or throws an AttributeError if the triggering match did not make use of the constructor).

Let us describe F's attributes:

self A reference that points to the F instance. Setting this attribute directly will replace the matched text (and will not change the self reference).

id Each variable that is defined using the various sub-pattern definition keywords imposes another attribute, by which one may access the sub-pattern.
4.5. EXTENDED REGULAR EXPRESSIONS

___iter___ The instance is iterable (i.e., it implements the ___iter___ method). Each entry of
the iterator is another token, or a sub-match, that groups more tokens. The iteration
contains entries for trivia (explained shortly), but the function has a default filter argu-
ment that filters these tokens. Therefore, in order to filter some of the tokens from the
iteration, provide the optional filter overload.

___str___ The ___str___ function of the object instance returns the text that was matched.
The Therefore, the text is accessible both in plain text mode and in parsed form.

___name___ Stores the name given to this match (in the case it is a sub-pattern of a match).

___type___ Use the ___type___ attribute to access the type according to which the match was
made, i.e., the name of the pattern as provided in @Find or the token name or classifi-
cation name as specified in the configuration file. Another two options are "Any" if it
was captured using @Any, or similarly "Empty" if it was captured using @Empty.

___pyobj___ One can access the PYTHON representation object for the match using the result
of the ___pyobj___ function, in the cases of tokens that were defined with a <convert>
element. This can also serve as a convenient convention for patterns (i.e., one can define
this method to provide a relevant PYTHON object).

___FILENAME___, ___LINE___ and ___COLUMN___ Access the name of the file, line and column of
the first character of the matched text, respectively.

An $F$ can be referenced in two ways. The first is to access the variable named self from
within @run or a similar keyword. In addition, each time an $F$ is instantiated (i.e., each time
the pattern is matched), it is appended to a list of instantiations, named after the name of
the lexi, and defined as a variable in the global symbols table.

Atomic REs. In the top block of the table we see the atomic REs. Any atom of the
host language is an atomic RE, matching only itself. The second line in the table tells us
that @SomeIdentifier invokes the RE previously defined in a configuration file, or by the
programmer, as in

@Find(SomeIdentifier) ...

A sequence of atoms is an RE that matches the same sequence of tokens in the input. It
matches a span of tokens (defining a range); atomic RE @Empty matches the empty span,
while @Any matches a span of any length.

When examined closely, the stream of input tokens, appears as an interleaved list of tokens
and “trivia” which are the thing that separates successive tokens. Trivia are typically the
space or new line characters, comments, spans of these, and even the empty string. There is
also a trivia at the beginning and at the end of the stream.

Normally, a trivia in an RE matches a trivia in the input. The LOLA programmer need
not worry about these. The four remaining atomic REs make an exception to this rule:

---

2https://docs.python.org/2/library/functions.html#globals
3a pattern to match precisely one don’t-care-token never came useful in earned experience
4see, e.g., in the implementation of the C# compiler, Roslyn https://github.com/dotnet/roslyn/wiki/Roslyn-20-overview-syntax-trivia
### Table 4.2: The recursive structure of LOLA’s extended regular expressions

<table>
<thead>
<tr>
<th>Atomic</th>
<th>LOLA keyword</th>
<th>Elaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>h o s t - t o k e n s</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@SomeIdentifier</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@Empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@Any</td>
<td>@without</td>
</tr>
<tr>
<td></td>
<td>@EndOfFile</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@NewLine</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@BeginningOfLine</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@EndOfFile</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@Sequence</td>
<td>@followedBy</td>
</tr>
<tr>
<td></td>
<td>@Optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@OneOrMore</td>
<td>@separator, @opener, @closer</td>
</tr>
<tr>
<td></td>
<td>@NoneOrMore</td>
<td>@separator, @opener, @closer, @ifNone</td>
</tr>
<tr>
<td></td>
<td>@Either</td>
<td>@or</td>
</tr>
<tr>
<td></td>
<td>@Xither</td>
<td>@xor</td>
</tr>
<tr>
<td></td>
<td>@Neither</td>
<td>@nor</td>
</tr>
<tr>
<td></td>
<td>@Match</td>
<td>@andAlso, @exceptFor</td>
</tr>
<tr>
<td></td>
<td>@SubsetOf</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@NonEmptySubsetOf</td>
<td>@and</td>
</tr>
<tr>
<td></td>
<td>@ProperSubsetOf</td>
<td>@and</td>
</tr>
<tr>
<td></td>
<td>@PermutationOf</td>
<td>@with</td>
</tr>
<tr>
<td></td>
<td>@Not</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@SameLine</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@Unbalanced</td>
<td></td>
</tr>
</tbody>
</table>

The `@EndOfFile` matches the last trivia in the input stream. The `@NewLine` matches the new line marker (typically \n or \r\n) in the trivia it occurs in. Keyword `@BeginningOfLine` matches this empty sized interval in a trivia which occurs right at the beginning of a file or after a new line marker, and `@EndOfFile` matches this empty sized interval in a trivia which occurs right before end-of-line or end-of-file.

**Traditional Constructors.** The lower block of the table lists the constructors of REs. The first group of these, the traditional constructors includes the essence of the repertoire of all RE languages. Note that the `@Sequence` constructor is present mostly for the sake of completeness. The verbose writing

```
@Sequence hello @followedBy world
```
may be preferable at times, but it is identical to

```
hello world
```

and to

```
@Sequence hello world
```

for that matter.

LOLA’s variants to the Kleene closure operator include an optional @separator elaborator, specifying the \( RE \) that separates the items in the repetition, and optional @opener and @closer variants, specifying what should occur before and after a non-empty list.

There is also an elaborator @ifNone to the @NoneOrMore constructor. With these elaborators, the complete specification of the label section of a PASCAL program\(^5\) (see also [58, p.32]) can be written as compactly as:

```
@Find(Label) @Identifier
@Find(LabelDeclarationPart)
@NoneOrMore @Label @opener @Label @separator, @closer;
@ifNone @Empty
```

\( RE \) disjunction uses @Either, the last among traditional constructors. Here is an example of how it might be used,

```
@Find(LabelDeclarationPart)
@Either @Empty @or
  @Label @OneOrMore @Label @separator,
;  
```

**Power Constructors.** LOLA uses a regular expression analysis engine that can deal with negation. Relying on negation, power constructor @Neither makes it possible to use phrases such as

```
@Neither (@nor) @nor { @nor }
```

while with the @Xither, the exclusive disjunction constructor, the template of use is @Xither... @xor... @xor... ...

Conjunction is made possible by the @Match constructor and its @andAlso and @exceptFor elaborators. The semantics of \( RE \)

```
@Match @Identifier @andAlso @UPPER_CASE @exceptFor @CamelCase
```

\(^5\)http://www.fit.vutbr.cz/study/courses/APR/public/ebnf.html label-declaration-part
<table>
<thead>
<tr>
<th><strong>Lola keyword</strong></th>
<th><strong>Python parameter</strong></th>
<th><strong>Multiple occurrences</strong></th>
<th><strong>Lola parameter</strong></th>
<th><strong>Purpose</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>@find</td>
<td></td>
<td></td>
<td>RE</td>
<td>Find an extended regular expression in the input stream, and trigger actions such as @run, @replace, @delete and @log.</td>
</tr>
<tr>
<td>@description</td>
<td>&quot;&quot;&quot;Multi-line string literal&quot;&quot;&quot;&quot;</td>
<td>✓</td>
<td></td>
<td>A textual description (comment) which may span several paragraphs.</td>
</tr>
<tr>
<td>@note</td>
<td>&quot;Short string literal&quot;</td>
<td></td>
<td></td>
<td>A short, typically, one-line description.</td>
</tr>
<tr>
<td>@run</td>
<td><code>{Command(s)}</code></td>
<td>✓</td>
<td></td>
<td>A Python script to execute on match.</td>
</tr>
<tr>
<td>@replace</td>
<td></td>
<td></td>
<td>GE</td>
<td>Alter the matched text.</td>
</tr>
<tr>
<td>@delete</td>
<td></td>
<td></td>
<td>RE</td>
<td>Delete the matched text.</td>
</tr>
<tr>
<td>@append</td>
<td>✓</td>
<td>GE</td>
<td></td>
<td>Add after the matched text.</td>
</tr>
<tr>
<td>@prepend</td>
<td>✓</td>
<td>GE</td>
<td></td>
<td>Add before the matched text always appending to prior output.</td>
</tr>
<tr>
<td>@filter</td>
<td>&quot;Short string literal&quot;</td>
<td>✓</td>
<td></td>
<td>Filter the matched text through a system shell, just before printing it.</td>
</tr>
<tr>
<td>@anchor</td>
<td>✓</td>
<td>RE</td>
<td></td>
<td>Produce an error if the pattern is matched where the defined pattern is not.</td>
</tr>
<tr>
<td>@example</td>
<td>✓</td>
<td>RE</td>
<td></td>
<td>State that the pattern matches the code.</td>
</tr>
<tr>
<td>@log</td>
<td><code>(Expression)</code></td>
<td>✓</td>
<td></td>
<td>Evaluate the expression and print the result to standard error stream.</td>
</tr>
<tr>
<td>@see</td>
<td>Identifier</td>
<td>✓</td>
<td></td>
<td>Refer to the specified pattern.</td>
</tr>
</tbody>
</table>

*a* @appends accumulate: a second @append adds its arguments after the first @append  
*b* @preends accumulate: a second @prepend places its argument before the first @prepend  
*c* multiple @filter directives accumulate in the order they are written  
*d* Has the elaborator @resultsIn.

**Table 4.3:** Sections in a LOLA pattern definition
4.6 LEXIES

is precisely what it reads to be. (Keywords @Identifier, @UPPER_CASE, and @CamelCase
are defined in the language configuration file.)

Constructors @SubsetOf, @NonEmptySubsetOf, are syntactic sugar to be used at occasions
when plain regular expressions become clumsy, such as the complex rules of specifications of
modifiers in JAVA. Figure 4.2 demonstrates.

Figure 4.2 An extended regular expression, specifying that “#find”, “#replace”, and
“#delete” may occur at any order, but at most once, and that “#replace” and “#delete”
cannot both occur

@Find(atMostOnceSections)
  @SubsetOf
    #Find
  @and
    @Either
      #replace
    @or
      #delete

arecall that if possible, LOLA does not issue errors for non-intuitive lexi definitions

The figure uses the @SubsetOf constructor in the definition of RE atMostOnceSections
which specifies the condition that a lexi may not contain more than one occurrence of @find,
@replace and @delete elaborators, and that if a lexi contains a @replace elaborator, it
cannot include @delete elaborator, and vice versa.

The purpose of constructor @Not is mostly served by the @exceptFor elaborator to @Match;
it is included for the sake of completeness.

Modifying Constructors. These are included for technical reason. Modifying constructor
@SameLine modifies the trivia in it, so that they never match an @EndOfLine. This constructor
becomes handy when augmenting a language with feature which must be contained in one
line.

The @Unbalanced modifier is discussed bellow, in conjunction with the classification of
tokens.

4.6 Lexies

A lexi definition is made of sections, such as @find, @replace, @description, etc. Sections
may occur in any order, and certain sections may occur more than once. Table 4.3 lists all
lexi sections, in their (approximate) recommended order, and their multiplicity constraints.
Table 1.1 tells us that lexi sections @Run, @Find, @Note, @Affirm, @Assert and @Description
may occur at the topmost file level, serving as LOLA directives. In truth all sections mentioned
in Table 4.3 may serve as directives; Table 1.1 mentions only the more useful ones.

A section name must be capitalized for it to serve as a directive. Creation of the pattern
object begins with this section. All lower-case named sections that follow are considered
elaborators that add sections to the pattern being created.

All sections of a lexi definition are optional. Even the @Find section is optional, in which
case the lexi is degenerate. A degenerate lexi matches precisely once, at the location it is
defined. Since degenerate lexies are allowed at the file level, we can find any section at this level.

### 4.6.1 Declarative Header Sections

There are three conceptual blocks in Table 4.3: *declarative header sections* tend to appear first. Sections `@note` and `@description` are for documentation purposes; they fire in cases of errors, at which time they print their contents, with the hope that this might come in handy to the client.

### 4.6.2 Action Sections

The *action* sections start with `@run`, which allows for arbitrary computation. Any change made to $F$ is implicitly reflected in the output.

Following is the familiar `@replace`, then `@delete` (which is just an alias for `@replace`)

```
@replace @Empty
```

and `@prepend` and `@append` (whose semantics should be obvious).

The `@filter` is used for hacks, e.g., pretty-printing the text just before producing it, converting special characters to trigraphs, etc. Sections are applied in the order they are defined. Filters are different. Regardless of their location in the lexi, filters are applied last. Specifically, any substitutions made by `@prepend`, `@append`, `@delete` and `@replace` to the captured text are made before.

The `@anchor` section typically defines a less demanding RE, than that of `@find`. It is an error for the `@anchor` to match without the `@find` matching as well. As demonstrated in Listing 3.2, the section can be used as safety measure, warning users of LOLA code that they may have used the host language in a manner not anticipated by the designer of the language augmentation.

### 4.6.3 Declarative Footer

The `@example` section is used for debugging and documentation. It is an error if the RE in the `@find` does not match the output of a `@example` section. It is also an error if the output produced by the lexi does not match the `@resultsIn` elaborator of an `@example`.

More generally, if the “LOLA argument” of an `@example` is a regular expression, then this regular expression must be contained in the RE of the `@find` section. (Containment of regular-expressions is computed in LOLA with conjunction and negation, and tests for emptiness of underlying finite automata underlying the regular expressions.\(^6\)

The use of the `@log` section was demonstrated above in Figure 3.1.

The `@see` section serves only a documentation purpose, referring the reader to other, related lexies.

---

\(^6\) LOLA refuses to negate expressions which involve pair balancing.
Chapter 5

The Host Language Configuration File

A host programming language has to be described in a configuration file in order for LOLA to execute. Figure 5.1 presents the full list of rules in the schema of the file. Later, each rule is explained separately.

Figure 5.1 Full EBNF grammar of configuration file

```
configuration-file ::= configuration-element *
configuration-element ::= balance
                     | super-balance
                     | token
                     | classification
                     | regular-expressions-definition

regular-expressions-definition ::= name regular-expression

regular-expression ::= pattern-reference
                     | range
                     | string
                     | special-character
                     | or
                     | sequence
                     | until
                     | character-negation

token ::= name
        regular-expression
        method ?

translate ::= from to

classification ::= name + name +

balance ::= begin +, end +
```
5.1 A High-Level Description of the Configuration File

At the root of the abstract syntax, we find a configuration-file, which is nothing but a list of configuration-element s, i.e.,

\[
\text{configuration-file ::= configuration-element }^* \quad (5.1)
\]

where the notation \( x^* \) is to say that the syntactical element \( x \) may occur zero, one or more times. Thus, a configuration-file is composed by a (possibly empty) list of configuration-element s, where a configuration-element is either a regular-expressions-definition, a classification, a balance or a token:

\[
\text{configuration-element ::= balance }
\quad | \quad \text{token}
\quad | \quad \text{classification}
\quad | \quad \text{regular-expressions-definition} \quad (5.2)
\]

5.2 Regular Expressions

To define and name a regular expression, use

\[
\text{regular-expressions-definition ::= name regular-expression} \quad (5.3)
\]

The regular expression itself is described using

\[
\text{regular-expression ::= pattern-reference }
\quad | \quad \text{range}
\quad | \quad \text{string}
\quad | \quad \text{special-character}
\quad | \quad \text{or}
\quad | \quad \text{sequence}
\quad | \quad \text{until}
\quad | \quad \text{character-negation} \quad (5.4)
\]

where each of the names explains the role of the respective element. A building block worth mentioning is until, which was specially added for the case of comments. Using this element, we can specify up to two characters to read until, such as in the case of comments in C. For a single line comment we would use “//” until new-line, and for a multi-line comment—“/*” until “*/”. In the case of two characters, this is a shorthand for the regular expression

\[
([^z](z^+[[/az]]))*z^+a
\]

where we wish to continue the regular expression up to a sequence za.

For more information about each individual way of constructing the regular expression, see the XML schema.
5.3 Tokens

To declare a token, specify a token name and a regular expression. Optionally, specify a method to convert the text value of the token to a useful PYTHON object. The token name can consist only of lowercase characters and underscores.

\[
\begin{align*}
token & ::= name \\
& \quad regular-expression \\
& \quad method ?
\end{align*}
\]

The method is some PYTHON code to convert the token to a PYTHON object. To specify the method, define a PYTHON function named convert, e.g.,

```python
<method>
    def convert(text):
        return float(text)
</method>
```

Examples of good candidates for token definitions are assignment-operators (\texttt{<<=}, \ldots), unary-operators (\texttt{!}, \texttt{-}, \ldots) and binary-operators (\texttt{1}, \texttt{==}, \ldots).

5.3.1 Digraphs and Trigraphs

Programming languages define sequences of characters to use instead of other characters. For example, in C, one can use the string \texttt{<>} for \texttt{[}. To support this feature, use the translate rule:

\[
\begin{align*}
\text{translate} & ::= \text{from} \ \text{to}
\end{align*}
\]

This technique makes it easier for the programmer in LOLA to define patterns in a more concise way, without taking into consideration those rules.

5.4 Token Classifications

One may wish to refer to a token in more than one name. Moreover, one may wish to refer to several tokens in the same name. To support this functionality in the configuration file, use token classifications. Associate several names with several new names:

\[
\begin{align*}
\text{classification} & ::= name + name + \ldots
\end{align*}
\]

The source names appear first, and then the new names.

Common classifications may be a keywords class (tokens that are reserved from use by the programmer, and identify an entity in the host language, e.g., \texttt{public} and \texttt{short}) and a punctuation class (specifies punctuation marks such as \texttt{,} and \texttt{;} in C).

5.4.1 Reserved Token Classifications

Two reserved token classifications exist. They are used to indicate special semantics.
Ignore classification  The first reserved token classification is the “ignore” classification. In many programming languages, whitespace is used to separate tokens, and is disallowed, e.g., in identifiers. Configuring LOL to ignore whitespace in the host language is done using the reserved ignore token specification.

Comment classification  The second reserved token classification is the “comment” classification. This classification is used to specify comments.

5.5 Balanced Pairs

Many programming languages use balancing pairs, such as curly braces {...}, parenthesis (...), brackets [...], and in more verbose languages, do...done and if...fi pairs.

To match constructs that use these, we wish to match only text that is balanced in this sense.

To escape this behavior, one may use the @Unbalanced keyword. In addition, and as a matter of convenience, if a pattern includes an odd number of balanced tokens (in the lexicon definition itself), a @Unbalanced is implicit.

To specify pairs of tokens to be matched as a start and an end of a sequence, balance is used:

\[
\text{balance ::= } \text{begin}^+, \text{end}^+
\]  

(5.8)

The begin element specifies the token to match as the starting sequence, and the end element specifies the ending token.

The PASCAL programming language is slightly special in that the end terminator is usually balanced against begin, but, the language also admits record...end and case...end pairs. For this reason, we allow both the begin and end components of a balance to include more than one token. The semantics is that each of the tokens in the begin component is balanced with any of the tokens in the end component. Multiple tokens in a begin or end clause should not be used for non-mixable pairs such as square brackets and curly brackets. Each of these pairs must be defined in a distinct balance clause.

LISP uses parenthesis to the extreme. To help the LISP programmer, the language offers a the syntax of

\[
[[[[\ldots]]]]
\]

which means that the closing bracket ] balances all of the previous parenthesis. This syntax implies the semantics of

\[
[[[[\ldots)]]]]
\]

To this end, one may use

\[
\text{super-balance ::= } \text{begin}^+, \text{end}^+
\]  

(5.9)

so that balancing pairs that appear in this specification balance every inner balancing pairs.
5.6 Configuration to an Input Language Convention

Many programming languages have a single identifier class. This means that a single token, usually named identifier, and usually take the lexical structure of

\[ [a-zA-Z_][a-zA-Z0-9_]* \],

is responsible for representing any new object—either a class, a variable or something else. However, in Prolog [9], variable names start with an upper-case letter, whereas atom names start with a lower-case letter.

Though many programming languages do not use several identifiers, we see groups of identifiers in those languages, emerging by the use of conventions. Several of the conventions are discussed shortly in Table 5.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage</th>
<th>Regular expression$^a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPER_CASE</td>
<td>C constants, mainly enum values, are all upper-case</td>
<td>[A-Z_]*</td>
</tr>
<tr>
<td>CamelCase</td>
<td>JAVA classes are upper camel case</td>
<td>([A-Z][a-z]*)+</td>
</tr>
<tr>
<td>camelCase</td>
<td>JAVA methods are lower camel case</td>
<td>([a-z][a-z]<em>) ([A-Z][a-z]</em>)+</td>
</tr>
<tr>
<td>mCamelCase</td>
<td>JAVA member fields (mainly in android programming), start with an m</td>
<td>m([a-z][a-z]<em>) ([A-Z][a-z]</em>)+</td>
</tr>
<tr>
<td>lower_case</td>
<td>C++ standard template library uses lower case characters and underscores</td>
<td>[a-z_]*</td>
</tr>
<tr>
<td>Camel_Case</td>
<td>similar to the previous convention</td>
<td>([A-Z][a-z]<em>)(_, ([A-Z][a-z]</em>)<em>)</em></td>
</tr>
<tr>
<td>ICamelCase</td>
<td>C# interfaces start with an I</td>
<td>I([a-z][a-z]<em>) ([A-Z][a-z]</em>)+</td>
</tr>
<tr>
<td>m_FeralCamel</td>
<td>C++ member fields start with a m_</td>
<td>m([A-Z][a-z])+</td>
</tr>
<tr>
<td>m_feralCamel</td>
<td></td>
<td>m_[a-z]+ ([A-Z][a-z]<em>)</em></td>
</tr>
</tbody>
</table>

$^a$digits are disregarded

Table 5.1: Examples of identifier groups

LOLA is configured by the user to a specific programming language, but there is no reason not to develop for a specific convention. Therefore, one may write a configuration file where the lexical analyzer distinguishes between different identifier groups.

Hungarian notation [48] is another great example of a convention of writing identifiers, though it is harder to track.
Chapter 6
Lexical and Syntactical Analysis

Having seen the showcase in Chapter 3 and reading discussion in Chapter 4 should have stated that LOLA’s grammar is pretty regular. The syntactical constructs are quite repetitive. In this chapter, we shall see how this regular structure is used in parsing, and understand how the parsing of @Find, @ForEach and @Case can be made in essentially the same manner, without relying on traditional grammar based parser, despite the fact that their syntax is not identical and the fact that @Find creates a lexi, @ForEach is a REs’ constructor and @ForEach is a constructor of GEs.

In the introductory chapter (Section 1.4) it was mentioned that elaborators such as @elseIf, @of, @and, and @ifNone attach to the inner most LOLA constructor to which they fit. This chapter explains how this binding is made.

Also, recall that @Find just as many other keywords, takes (after an optional parenthesized PYTHON identifier) sequence of tokens (and compound objects). Upfront, this definition is ambiguous. Consider this example, in which both keywords @Find and OneOrMore expect a sequence of tokens:

@Find @OneOrMore token1 token2

One interpretation of the above is:

@Find {@OneOrMore {token1} token2}

However, The right associativity rule (taking aide at time from the no lower indentation argument) aggregates differently the arguments: the list argument to a keyword is the sequence of tokens (and compound objects) that starts at the first token after the keyword and ends just before the token in which the current indentation is resumed. In the example

@Find {@OneOrMore {token1 token2}}

More generally, aggregation technique means that a keyword accepts the longest possible sequence of tokens. LOLA uses nor parenthesis like characters neither begin...end and fi...fi like pairs for blocking and aggregation. (This means that these are free to use as keywords and characters of the host tokens). Blocking and aggregation are achieved in LOLA by indentation. This chapter also describes how the parser maintains the other rules of indentation mentioned in Section 1.4. The fine details are in this chapter which also explains how the parser
6.1 Lexical Analysis

The input to LOLA is a file consisting host-language code intermixed with LOLA directives. The configuration file tells LOLA how to group characters in the input into tokens, either of the host-language or of LOLA, and which characters (or sequences of characters) are trivia. For this purpose LOLA tokens are either keywords or keywords along with the Python snippets attached to them. LOLA’s lexical analyzer produces a stream of what we call “bunnies”, which is just a collective name for trivia and tokens (of either language). This stream of bunnies makes the input to LOLA’s parser.

Consider for example the code in Listing 6.1.

Listing 6.1 A contrived LOLA lexi demonstrating how elaborators attach to their respective constructors

```plaintext
// 1 2 3 4 5 6 7
// 3456789 123456789 123456789 123456789 123456789 123456789 123456789
@Find sizeof @Sequence Bugs
    @followedBy ?huge @replace @If(random() >= 0.5) 42
    @else 6 *
sizeof Bugs
? huge
```

The listing shows a short source file, containing a lexi (lines 3–6) surrounded by C++ code: comments in lines 1–2 and an illegal expression followed by an empty line (lines 7–9).

The stream of bunnies produced from this file is depicted in Figure 6.1.

Figure 6.1 The stream of lexical units (bunnies) produced from the code in Listing 6.1

In the figure, bunnies are portrayed as a nodes in a linked list. The figure’s legend shows
how frame and color determine the bunny’s type: bunny is a trivia, Bugs bunny\textsuperscript{1}, a host-token; etc.

\textbf{Figure 6.2} Hare Bugs Bunny. ♀ April 30, 1938– (shown with Lola Bunny (Figure 1.1))

All three kinds of bunnies are demonstrated in the figure:

\textbf{Trivia} A trivia is a (potentially empty) sequence of comments, non-visual characters such as spaces, tabulation, new lines, carriage returns etc., or whatever is defined as such in the configuration file.

In Listing 6.1 we see a trivia at the beginning and at the end of the stream, and a trivia between any two other bunnies. Also, observe that LOLA coalesces all consecutive occurrences of these into a single trivia.

\textsuperscript{1}pun intended, huh, huh.
CHAPTER 6. LEXICAL AND SYNTACTICAL ANALYSIS

Note that the trivia between the C++ tokens “?” and “huge” is empty; its sole purpose is to separate these two tokens. Other tokens in the listing contain one space (line 5), multiple spaces, several new line characters and even comments.

Host Tokens These are the tokens of the host language, including identifiers, keywords, literals, etc., as defined in the configuration file.

The first five host tokens in the figure are “sizeof”, “Bugs”, “?”, “huge”, and, “42”.

Lola Keywords which may be followed by properly bracketed PYTHON code snippets. We shall think of the keyword, along with the bracketed PYTHON snippet, and the brackets themselves, as one.

In the figure, we see the following keywords in order: @Find and @replace (which are sections of a lexi), @Sequence followed by the @followedBy elaborator, and @If (a constructor of GE), and its @else elaborator.

Observe that the bunny of @If carries with it the PYTHON code snippet, including the brackets (parenthesis in this case).

The lexical analyzer also attaches to each token a column number, to be used later in the parsing phase. The column number of a token is simply the number of characters that precede the first character of this token in the line in which it occurs.

Column numbers are depicted as little labels in Figure 6.1. For example the column of @If and its @else elaborator is 41, and the column number of the host token 7 is 43.

In processing the stream of bunnies LOLA does two things:

- identification of directives, and removing these from the input stream, and,
- application of directives to the remaining bunnies (which must be either trivia bunnies or host-language tokens)

Figure 6.3 shows the stream of bunnies generated from Figure 6.1 after the lexi in lines 3–6 is removed from it.

Figure 6.3 The stream of remaining lexical units (bunnies) after removing the lexi from Figure 6.1

Figure 6.4 shows the stream of bunnies of the lexi in lines 3–6 of Listing 6.1. This stream is later fed to LOLA’s parser which will turn it into a directive.
Comparing the figure to the original stream (Figure 6.1) we observe that the constructor bunny marked @Find in the original stream was converted to two bunnies: an automatically generated constructor, named Lexi, followed by a @find modifier. This conversion is carried out at the lexical phase to save the parser from special-casing sections whose name is capitalized.

Another conversion carried out by the lexical analyzer is the conversion of the bunny composed of a keyword decorated with a snippet, e.g., @If in Figure 6.1, into two bunnies: a constructor named @If, followed by a new kind of bunny marking PYTHON snippets, e.g., the bunny with the snippet

\[ \text{random()} >= 5 \]

in Figure 6.4.

Again, this conversion of snippets is carried out for non-profound reasons: Some LOLA keywords such as @Find and @Sequence take an optional snippet. This optional snippet may look as a list of tokens. For example,

@Find (a) b

might be interpreted both as a search for host tokens \((a) \ b\), and as a search for host token \(b\) to be stored in PYTHON variable \(a\). The lexical analyzer resolves the ambiguity by selecting the first alternative if one or more space characters follows the keyword. So the above input is interpreted as per the first alternative. The second alternative is selected in case no such space occurs, e.g., if the input is

@Find(a) b

then “a” is interpreted as a PYTHON identifier into which the result of the search is stored, rather than a host token to be found in the input stream.
6.2 The Abstract Syntax Tree

It is the duty of the LOLA parser to generate from all other bunnies the Abstract Syntax Tree (AST) of directives embedded in the code.

The AST for the running example (Listing 6.1) is portrayed in Figure 6.5. It is created from the stream in Figure 6.4.

Figure 6.5 The abstract syntax tree created for the lexi in Listing 6.1

The most important kind of a nodes in the AST is that of constructors. In the figure we see three of these. The node labeled @Sequence represents a constructor of a RE; the node labeled @If represents a GE. The root of the AST in Figure 6.5, denoted the word lexi is also a constructor.

Second in importance are elaborator nodes. In the figure we see four of these, labeled @find, @replace, @followedBy, and, @else.

Figure 6.5 shows two other kinds of nodes: host-tokens such as sizeof, Bugs, etc., and nodes with PYTHON code snippets (the only such node in the figure contains the snippet

\[ \text{random()} \geq 0.5 \]

6.3 Children of Constructors and Elaborators

A general constructor node has three kinds of children which must occur in a very specific order:

1. a PYTHON snippet, which may occur at most once, and must be the first child when it occurs;

2. a list of host tokens and constructors, which may occur at most once, and must immediately follow the snippet or be the first child if no snippet occurs; and,

3. any number (including none) of elaborators, which must occur last.

A general elaborator node has the same kinds of children, and these which must follow the same order as in constructor nodes. Elaborators may even have elaborators of their own.
The only difference between constructors and elaborators is that elaborators are not allowed to occur in the list child.

Each individual constructor or elaborator may impose its own specific requirements on its children, including a specification on the order, kind and cardinality of its elaborators. A lexi for example, does not admit a snippet, nor a list, while an \@If does not allow an \@else elaborator to occur before an \@elseif elaborator.

Table 6.1 below specifies these restrictions for some common constructors including all those occurring in Figure 6.5².

<table>
<thead>
<tr>
<th>Snippet</th>
<th>List</th>
<th>Elaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Lexi</code></td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><code>@Nothing</code></td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><code>@Include</code></td>
<td>String</td>
<td>×</td>
</tr>
<tr>
<td><code>@If</code></td>
<td>Boolean-Expression</td>
<td>GE</td>
</tr>
<tr>
<td><code>@NoneOrMore</code></td>
<td>Identifier_opt</td>
<td>RE</td>
</tr>
<tr>
<td><code>@Either</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>@Case</code></td>
<td>Expression</td>
<td></td>
</tr>
<tr>
<td><code>@SameLine</code></td>
<td>Identifier_opt</td>
<td>RE</td>
</tr>
<tr>
<td><code>@Sequence</code></td>
<td>Identifier_opt</td>
<td>RE</td>
</tr>
</tbody>
</table>

Table 6.1: Variety in restrictions placed on children by different constructors

The table demonstrates the great variety of restrictions placed on children by constructors: A lexi has many elaborators but does not admit neither a snippet nor a list. Another

²The full list of restrictions is implicit in tables 4.1 to 4.3
constructor that allows only one kind of children is @Include, which only allows a snippet\(^3\).

In contrast, @If has all three children. Constructor @NoneOrMore also has all three, but in this constructor, the PYTHON snippet is optional.

Constructor @Nothing is restrictive as it can be: it forbids all three kinds of children: snippet, list and elaborators.

Three constructors in Table 6.1 allow only two out of three kinds of children: @Either has no snippet, @Case has no list, and @SameLine which has no elaborators.

Table 6.2 below repeats Table 6.1, but for selected elaborators, demonstrating the variety of requirements placed by elaborators.

<table>
<thead>
<tr>
<th>Snippet</th>
<th>List</th>
<th>Elaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>@delete</td>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td>@run</td>
<td>Commands</td>
<td>(\times)</td>
</tr>
<tr>
<td>@else</td>
<td>(\times)</td>
<td>GE(s)</td>
</tr>
<tr>
<td>@example</td>
<td>(\times)</td>
<td>RE(s)</td>
</tr>
<tr>
<td>@to</td>
<td>String-Expression</td>
<td>GE(s)</td>
</tr>
<tr>
<td>@elseif</td>
<td>Boolean-Expression</td>
<td>GE(s)</td>
</tr>
<tr>
<td>@log</td>
<td>Identifier(_\text{opt})</td>
<td>GE(s)</td>
</tr>
<tr>
<td>@of</td>
<td>Expression</td>
<td>GE(s)</td>
</tr>
</tbody>
</table>

Table 6.2: Variety in restrictions placed on children by different elaborators

Both tables 6.1 and 6.2 employ LOLA syntax to specify the order of elaborators. This specification turns out to be quite readable, e.g., the elaborators of @Case are a sequence of one or more @of, followed by an optional @otherwise.

In the table (and henceforth) meta-specifications of this sort use the “@” notation for keywords such as @Sequence that defines regular expressions, and the “#” notation (e.g., #case, #of, #otherwise) notation for LOLA elaborators whose order, multiplicity and other restrictions are being specified.

Table 6.1 only hints on restrictions placed on the elaborators (sections) of a lexi. The full list of these restrictions is:

1. at least one elaborator,
2. at most one @find, one @delete and one @replace,
3. @delete and @replace are mutually exclusive, and,
4. otherwise, elaborators may occur any number of times and in any order.

Listing 6.2 is a meta-lexi filling this gap. The meta-lexi in it specifies, using LOLA syntax, the requirements that a LOLA lexi places on its elaborators.

\(^3\)None of LOLA current constructors is one that only allows a list.
### Listing 6.2 A meta-lexi specifying restrictions on sections of a lexi

@Note "A meta-lexi specifying the constraints on sections of a lexi"

@find
  @OneOrMore
    @Either
      #find @or #replace @or #delete @or #append @or #prepend
      @or
      #description @or #note @or #log @or #see @or #example
      @or
      #anchor @or #run @or #filter
    @exceptFor
      @Any #find @Any #find @Any
    @exceptFor
      @Any #delete @Any #delete @Any
    @exceptFor
      @Any #replace @Any #replace @Any
    @exceptFor
      @Any #replace @Any #delete @Any
    @exceptFor
      @Any #delete @Any #replace @Any
  @anchor
    @Either
      #find @or #replace @or #delete @or #append @or #prepend
      @or
      #description @or #note @or #log @or #see @or #example
      @or
      #anchor @or #run @or #filter

### 6.4 Parsing without Explicit Grammar

Having seen in Chapter 4 the large number of constructors and elaborators, and having seen in the above section the rich set of requirements on elaborators (e.g., Listing 6.2), and the variety on requirements on children, both in constructors (Table 6.1) and elaborators (Table 6.2) it should be clear that:

- the syntax of LOLA can be written with a context free grammar notation such as EBNF, but,

- the resulting syntax definition is likely to be large and cumbersome, and that,

- anticipated changes and additions to LOLA are likely to break this definition.

Thus, instead of the traditional approach of feeding a grammar specification to a parser generator, LOLA uses a different approach: A universal parser reads the bunnies. The grammar specification for keywords is made in a modular fashion: the module that realizes a keyword specifies its syntax.

LOLA thus encapsulates the idiosyncrasies of each of the constructors and elaborators in their respective builder objects. When the LOLA parser encounters a keyword, it creates a corresponding builder, whose life purpose is maturing into the correct AST node.
Builders support these methods:

**Query**  \( b.\text{accepts}(c) \) which determines whether builder \( b \) can adopt a child \( c \), which can be a bunny, a compound, or a compound elaborator.

**Operation**  \( b.\text{adopt}(c) \) which, on the presumption \( b.\text{accepts}(c) \), adds child \( c \) to builder \( b \).

**Query**  \( b.\text{mature}() \) which returns \textbf{True} if, and only if, builder \( b \) matured into an AST node.

**Operation**  \( b.\text{done}() \) telling \( b \) that it may expect no more \( b.\text{adopt}() \) nor \( b.\text{accept}(\cdot) \) invocations.

If at the time of a \( \text{done}(\cdot) \), \( b \) notices that it still needs more children, it emits an appropriate error message. After the call, \( b.\text{mature}() \)s shall affirm that the \( b \) is complete, regardless of error messages.

Grammatical derivation rules are thus imitated by the behavior of specific builders, e.g., @If builder refusing to receive any more elaborators after seeing an @else elaborator, and a @Case builder issuing an error message if told it is done() prior to seeing an @of elaborator.

Since the order of children is fixed, builders follow a similar protocol. Initially, a builder is in a “Snippet” state expecting a snippet. When the snippet is accepted, the builder moves to a “List” state, expecting and accepting any number of host tokens and compound objects. When the first elaborator child comes, this list is locked and the builder moves to the “Elaborators” state, expecting and accepting elaborators.

When an unexpected child arrives, the builder goes into a final state “Done” or to final state “Error”. In both cases, the builder matures into an AST node, and will take no more children. State “Done” is reached if two conditions are met:

- The current builder received all the children it needs to become a valid AST node of the appropriate kind.

- The pending child is of type elaborator, which might be taken by a pending builder in which the current builder is nested.

Hence, when the builder sees the first elaborator, the general list is locked.

**Figure 6.6**, showing the automaton behind the implementation of some generic builder, demonstrates how function \( b.\text{accepts}(x) \) is implemented.
The generic builder in the figure taking a mandatory snippet argument, followed by a list of trivia, host tokens, and compound objects, followed by zero or more elaborators.

For the purpose of presentation it is tacitly assumed that the automaton knows which elaborators are expected; no automaton states are allocated to represent the underlying logic (which may be as complicated as in Listing 6.2).

The automaton follows an eager policy of trying to collect children as long as it is possible to do so, and only declare an error if the builder cannot possibly be made into a legal AST node.

Since the AST node must start with a snippet, any other input at the starting state “Snippet” leads to an error. Also, an occurrence of a second snippet, i.e., in states “List” and “Elaborator”, leads to an error.

However, when the accepted \( x \) is an elaborator which the builder does not recognize, and the builder has already seen and stored its (i.e., it is in the either the “List” or the “Elaborator” state), the builder moves into a “Done” state, refusing to collect any more children.

Still if the builder sees an unrecognized elaborator prior to seeing all elaborators it must see, it moves into the “Error” state. This stops the builder from taking any more children, and effectively lets other builders try to accommodate the unfamiliar elaborator.

Another case of error in the building process may occur in the “List” state. If a builder of a \( GE \) receives into the list a compound of \( REs \), it takes it, reports an error, and moves into the “Error” state.

Of course, the implementations of the variety of builders (tables 6.1 and 6.2) are slightly different, to deal with the cases of optional snippet, no snippet at all, no list, etc., but the eager child collection policy is shared by all.
6.5 The Stack of Pending Builders

LOLA’s parser maintains a stack of pending builders, and tries to add children to the top most builder. If this builder refuses to adopt an additional child, the parser pops the stack, and tries to add the child to the next pending builder.

Function \texttt{percolate}(c), processing \(c\), a child to be fitted with the stack of pending builders, listed in Algorithm 6.1 describes the details.

**Algorithm 6.1** Procedure \texttt{percolate}(c), processing \(c\), a child to be fitted with the stack of pending builders.

1: While \(\neg \text{empty}()\) do  
2: \hspace{1em} Let \(b \leftarrow \text{top}()\) \hspace{1em} // current candidate builder  
3: \hspace{1em} If \(\text{column}(c) < \text{column}(b) \text{ or else } \neg b.\text{accepts}(c)\) then \hspace{1em} // not the right builder  
4: \hspace{1em} \hspace{1em} \hspace{1em} \(b.\text{done}()\) \hspace{1em} // tell it can expect no more arguments  
5: \hspace{1em} \hspace{1em} \hspace{1em} \(\text{pop}()\) \hspace{1em} // b is complete now; remove it from the stack  
6: \hspace{1em} \hspace{1em} \hspace{1em} \(\text{percolate}(b)\) \hspace{1em} // send it to its destined builder  
7: \hspace{1em} \hspace{1em} \hspace{1em} \(\text{continue}\) \hspace{1em} // percolating \(c\) to builders on stack  
8: \hspace{1em} \hspace{1em} \hspace{1em} \(b.\text{adopt}(c)\) \hspace{1em} // save complete object \(c\) in builder \(b\)  
9: \hspace{1em} \hspace{1em} \hspace{1em} \(\text{If } \neg b.\text{mature}() \text{ then}\) \hspace{1em} // b may take further arguments  
10: \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \(\text{Return}\) \hspace{1em} // proceed to next token in the input  
11: \hspace{1em} \hspace{1em} \hspace{1em} \(\text{pop}()\) \hspace{1em} // \(b\) matured, \(\text{percolate}\) completed \(b\)  
12: \hspace{1em} \hspace{1em} \hspace{1em} \(c \leftarrow b\) \hspace{1em} // old \(c\) consumed; proceed with completed \(b\)  
13: \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} // empty stack; no more builders to take argument \(c\)

Procedure \texttt{percolate}() uses operation \texttt{pop()} and queries \texttt{empty()} and \texttt{top()} on an implicit, global, stack data-structure of builders whose actions were suspended.

The loop in \texttt{percolate}(b) tries to fit a prospective child \(c\) with \(b\), the builder on top of the stack. If the indentation of the child is lower than that of the builder, or the builder cannot accept the child (line 3 in Algorithm 6.1), the procedure loops again, trying to send the child to the next builder pending on top of the stack.

Take note that before a child is sent for adoption by a builder deeper into the stack, all builders on top of this builder must be declared as done. In particular, procedure \texttt{percolate}() declares the current builder \(b\) as done (line 4), since once a prospective child is sent to an ancestor, the current parent cannot expect any more children. For this reason, it is \texttt{pop()}ed from the stack at line 5.

The recursive call (line 6 in the algorithm) is crucial, and it must be made before the prospective child \(c\) is tried against the builders deep in the stack. Since the current builder \(b\) can take no more children, \(b\) must be adopted itself before \(c\), which follows it, can find its adopting parent.

Procedure \texttt{percolate}() is used by LOLA to generate an AST node for directives. Concretely, \texttt{percolate}() serves as a subroutine of \texttt{directive}(), portrayed in Algorithm 6.2 below, which is in charge of creating the AST of directives.

Function \texttt{directive}() is called whenever LOLA’s main notices on the input stream a bunny which might be a builder, i.e., a “@CamelCase” keyword, or the special “Lexi” bunny (generated, as explained above, from a capitalized section such as @Find).
6.5. **THE STACK OF PENDING BUILDERS**

If the input is legal, $\text{directive}(c)$ returns the AST for the directive created by constructor $c$. If the input is illegal, an AST is still returned, but it might be invalid.

<table>
<thead>
<tr>
<th>Algorithm 6.2 Function $\text{directive}(c)$, returning the AST for the directive created by constructor $c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: $\text{push}(c)$  // collect $c$’s children</td>
</tr>
<tr>
<td>2: While $\neg \text{eof()}$ do  // More bunnies pending</td>
</tr>
<tr>
<td>3: Let $b \leftarrow \text{getBunny}()$</td>
</tr>
<tr>
<td>4: If $\text{leaf}(b)$ then  // host, trivia, or snippet token</td>
</tr>
<tr>
<td>5: $\text{percolate}(b)$  // try to find a parent for $b$</td>
</tr>
<tr>
<td>6: else  // must be a constructor or an elaborator</td>
</tr>
<tr>
<td>7: $\text{push}(c)$  // will be done when children are collected</td>
</tr>
<tr>
<td>8: If $\text{empty}()$ then  // all $c$’s children were collected</td>
</tr>
<tr>
<td>9: $\text{ungetBunny}(b)$  // a directive can only end with a bunny it cannot adopt</td>
</tr>
<tr>
<td>10: Return $c$  // end of file reached, must notify all pending builders they are done</td>
</tr>
<tr>
<td>11: While $\neg \text{empty}()$ do  // builders awaiting arguments still</td>
</tr>
<tr>
<td>12: Let $b \leftarrow \text{top}()$  // current unfinished builder</td>
</tr>
<tr>
<td>13: $\text{pop}()$  // $b$ is complete now; remove it from the stack</td>
</tr>
<tr>
<td>14: $b$.done()  // tell $b$ it can expect no more arguments</td>
</tr>
<tr>
<td>15: $\text{pop}()$  // $b$ is complete now; remove it from the stack</td>
</tr>
<tr>
<td>16: $\text{percolate}(b)$  // send $b$ to its destined builder</td>
</tr>
<tr>
<td>17: Return $c$  // all children of argument $c$ were collected</td>
</tr>
</tbody>
</table>

Function $\text{directive}(c)$ employs

- query $\text{leaf}(b)$ determining whether bunny $b$ is trivia, host or snippet,
- operation $\text{push}(t)$ on the implicit global stack of builders whose actions action was suspended, and,
- function $\text{percolate}(t)$ for sending complete object to these, potentially creating more complete objects in the process, and, trying to determine their matching, incomplete builders in the stack.

Elaborators, which are also builders, typically following the “@camelCase” convention, can never start a directive. Whenever an elaborator occurs with no preceding constructor, the input is erroneous. Still, for the purpose of better error recovery and reporting, the main loop invokes function $\text{directive}(\cdot)$ on these as well.

The computation in Figure 6.4 consumes bunnies until argument $c$ can collect no more children. If the bunny is a trivia, a host token, or snippet, it cannot serve as an internal node, and it is sent for finding a parent by invoking function $\text{percolate}(\cdot)$.

If however the next bunny is an elaborator or a constructor (line 6 in the algorithm), i.e., it is a prospective parent, it is pushed into the builders stack (line 7). This push means that on later steps of the parsing, $\text{percolate}(\cdot)$ will find the children for this parent (assuming that the input is syntactically correct). Later on, this parent bunny will become a child or a farther descendant of argument $c$.  

Technion - Computer Science Department - M.Sc. Thesis  MSC-2016-12 - 2016
6.6 Trivia Management

The AST depicted in Figure 6.5 does not portray and trivia nodes. These are generally omitted when creating ASTs of patterns. The truth of the matter is that trivia nodes receive special cases in several occasions:

- When LOLA creates an AST of a regular expression, it does not save in it bunnies found in the input. But, special keywords @NewLine, @EndOfFile, etc., can be used to match designated trivia in the input.

- When scanning for a match, LOLA ignores for the purpose of the matching the trivia between matched tokens, but this trivia is recorded for a later use. If any two or more consecutive tokens are printed, the printout includes also the trivia(s) that separate them.

- A trivia in the input can be matched specifically by the special keywords (parameterless constructors), that match them. A trivia in the input can be unmatched, i.e., cause the current match to fail, if it contains a new line, and the matching object is qualifies with a @SameLine keyword.

- In contrast with regular expression, trivia bunnies found in generating expressions are kept and used.

  However, if a trivia that includes a beginning of a line (optionally followed by spaces) is stored with a generating expression, this trivia is truncated, bringing its column down to that of the outer most operator of the generating expression.
Chapter 7

Pruning, Matching and Triggering

7.1 Input Pruning

The previous chapter explained how directives in the input stream are identified, converted into ASTs, and pruned out of the input stream. After this pruning, LOLA applies the directives on the pruned input.

*Immediates* are directives that execute immediately. A directive that does not execute immediately can only be a non-degenerate, i.e., a lexi which has a @find section. Henceforth, unless otherwise specified, the term “lexi” refers to non-degenerate lexies.

Immediates are either *degenerate lexies*, i.e., lexies missing a @find section, or, *generative directives* such as @Include and @If. Immediates executed right after their parsing is complete. The output produced by this execution is interpolated, as is, into the input stream.

The interpolated output may contain more LOLA directives: This is because interpolation into input stream means that the characters in this output are tokenized and parsed exactly like characters that originate from the main stream. Therefore, if directives are found in the interpolation, they are processed in the same way that directives on the main input stream are processed.

Specifically, a lexi found in the output of an immediate is stored for later execution, while immediates found in the output of immediates are recursively executed.

The output of recursive execution of immediates is recursively interpolated into the input stream. Recursive interpolation makes it possible to @Include files that @Import other files that @Splice the output of a system command which prints some more LOLA directives, and it makes it possible to write if LOLA, the equivalent of this C preprocessor programming idiom:

```plaintext
#if 0
#define A 1
#else
#define A 2
#endif
```

The *pruned text* is obtained by removing all directives, and recursively executing immediate. Ignoring trivia for now, the pruned text is tokenized, and converted into a string of \( n \) tokens \( \tau_0 \cdots \tau_{n-1} \) stored in an appropriate buffer, on which LOLA employs a fixed point (work-list) computation, applying substitutions defined by lexies, until no more apply.
More concretely, LOLA steps through this buffer, and after reading each token in it, checks to see when, where and which of the stored lexies concludes in a match. Think of match \( M \) as having two properties: the lexi (and the reifying object), and the substring of \( \tau_0 \cdots \tau_{n-1} \) with which the match occurred. Notice that a lexi may be involved in several matches that conclude at the same token, but have started at different locations.

Under certain conditions, a matching lexi triggers, in which case it conducts a replacement. In the general case, the substituted text could have been generated by PYTHON (e.g., by using the PYTHON escape keyword @) and needs to re-tokenized. However, unlike immediates, the output of lexies may not contain other LOLA directives: LOLA is willing to process directives only at the initial phase, when immediates are executed and expanded recursively to produce the pruned text.

### 7.2 The Match of the Matches

LOLA must run a match among the all possible matches, to determine the winning match. The winning match then proceeds to trigger its lexi. A triggered lexi typically changes the text it matched; the matched tokens are removed, and is replaced by the substitution text (which may occasionally be identical to the matched tokens).

A total ordering of matches is required in this match of matches. Without it, LOLA programs cannot be guaranteed to exhibit deterministic behavior.

Clearly, least, @run sections must be executed in a deterministic, predictable order. But, even without @run sections, the order of application of @replace sections of different lexies matters.

For examples, here are two lexies that do not commute:

- First is
  \[
  \texttt{@Find \textasciitilde\textasciitilde @replace \textasciitilde\textasciitilde} \quad (7.1)
  \]

- Second is
  \[
  \texttt{@Find \textasciitilde\textasciitilde @replace \textasciitilde\textasciitilde} \
  \]

If the lexies are applied to the input

\[
\texttt{\textasciitilde\textasciitilde\textasciitilde},
\]

the first and only then the second, the final output is

\[
\texttt{\textasciitilde\textasciitilde}.
\]

(The first lexi yields “\textasciitilde\textasciitilde” which is then converted by the second lexi to “\textasciitilde\textasciitilde”.)

However, the application of the second lexi and then the first, to the same input, yields the final output

\[
\texttt{++}
\]

(The second lexi, which is applied first, converts input “\textasciitilde\textasciitilde\textasciitilde\textasciitilde” to “\textasciitilde\textasciitilde\textasciitilde\textasciitilde”, which does not trigger the first lexi that is applied only second.)
7.3 Multiple Matches of the Same Lexi

The action of a lexi can interfere with itself, since the match may span overlapping and contained substrings of the buffer.

Take for example pattern @Expressions that matches a sequence of one or more comma separated matches of @Expression (both defined in Listing 1.4 above), where @Expression is a valid C (say) expression that does not match pattern @Expressions. (Patterns @Expression and @Expressions are mutually recursive: Each may directly contain the other, but not itself.)

Consider now a lexi intended to parenthesizes a comma separated list of expressions.

```
@Find @Expressions @prepend( @append)
```

A definition of an execution order must be able to deal with the application of this lexi to (pruned) input such as

```
1,2,(3,4,5)
```

As it turns out, @Expressions matches the above input a dozen times:

- There are 6 distinct matches in the outer comma separated list:
  - 3 matches in which the list has only one @Expression: “1”, “2”, and “(3,4,5)”;
  - 2 matches in which the list is made by two matches of @Expression separated by a match with a comma: “1,2” and “2,(3,4,5)”;
  - 1 match of list of three elements: “1,2,(3,4,5)”.

- Similarly, there are 6 additional matches in the inner list:
  - 3 with list of size one: 3”, “4”, and “5”;
  - 2 with list of size two: “3,4”, “4,5”; and,
  - 1 with list of size three: and, “3,4,5”.

The canonical Perl “regexen” and many other regular expression subsystems exercise greedy matching policy in which only the longest possible match is used. There is often also a lazy version of the Kleene star regular expression constructor and of the “optional” element constructor.

In contrast, LOLA uses the full range of semantics between lazy to greedy: the matching engine is generating, at least conceptually, all possible matches.

Not all matches trigger the lexi. In fact, at each iteration of LOLA’s main loop, only one match triggers: this is single match that “overrides” all other candidate matches. The effect of triggering this overruling match is a replacement of the matched substring of the input by a substitution string that might be longer, short, of the same length, and even identical to the matched string.

To guard against accidental infinite recursion, the overruling match is earmarked as have been applied to the substitution string.

At the next iteration LOLA regenerates (conceptually) the set of all matches making sure to ignore earmarked matches. This set is typically different than the set of matches in the previous iteration, since the input string has changed. But, again, the overruling match is determined, earmarked, triggers its lexi.
7.4 Lola’s Iterative Fixed Point Behavior

The triggering of lexies in LOLA follows an iterative and stops only when a fixed-point is reached. At each step, LOLA examines all matches of all of lexies, selecting only one of them, triggering it (possibly changing the text), and reiterates. Iteration stops when no more matches can be found.

More formally, the conceptual iterative procedure that LOLA uses to exhaustively apply lexies to the pruned input is presented in Algorithm 7.1.

Algorithm 7.1 Function fixedPoint(ℒ, S) receiving a set of lexies ℒ and pruned text S, and returning S after all lexies in ℒ were exhaustively applies to it.

Lines 2–10 creates the list of all matches, by examining all substrings and all lexies. Two exceptions are made though: a lexi is not allowed to match substring that started before the lexi was defined (lines 5–6), and a match which was earmarked for having been triggered before is not considered (lines 8–9).

Lines 11–12 stop execution when the fixed point was reached, i.e., no more valid non-earmarked matches were found. Lines 13–16 selects the winning match, i.e., match M that overrules all M′ ∈ ℳ, M′ ≠ M. Match M is then triggered at line 17.

Earmarking, done at line 18, means that the lexi of M will never be triggered again, neither for the concrete substring it matched nor for any substring of it, however these substrings are transformed by later applications.

The four rules that determine the total order of overruling among matches are listed in Figure 7.1.

---

1The actual implementation is of course much more efficient. LOLA never produces a match if it is guaranteed to be overruled.
7.5. LOCATIONS AND SUBSTRINGS

The first rule is that lexies work their way bottom-up, preferring matches to longer ones. The second is that in the absence of containment relationship, matches are executed left to right, i.e., in the order of their beginning point.

The third rule is to guard against accidental recursive definition, and to make it possible for a lexi to @append or @prepend code, without going into an infinite series of applications. The rule does not provide absolute production. A stubborn program can write two lexies whose fixed computation will inflate the code for ever.

The fourth rule, giving higher priority to lexies defined later, is to make it easier for clients to override lexies imported from an external library.

In what follows we formulate these rules more precisely, and try to make them more intuitive.

7.5 Locations and Substrings

7.5.1 Locations

A location in the string of \( n \) tokens \( \tau_0 \cdots \tau_{n-1} \) might be either the point just before the first token, the point just after the last token, or, any point between two consecutive tokens. For \( i \) in the range \( 0 \ldots n \), integer, \( \ell_i \) is the \( (i + 1)^{th} \) location.

As depicted in Figure 7.2 the input string of 13 tokens

\[ 1, 2, 3, (4, 5, 6) \]
has 13 + 1 locations, denoted

$$\ell_0, \ell_1, \ldots, \ell_{12}, \ell_{13}.$$  

These 14 locations are placed in the string “1, 2, 3, (4, 5, 6)”:
just before the first token ($\ell_0$), between two consecutive tokens ($\ell_1, \ldots, \ell_{12}$), or, just after the last token ($\ell_{13}$).

### 7.5.2 Substrings

Let $\ell_f$ be some location, and let $\ell_t$, $f \leq t$ be some other location. Together, $\ell_f$ and $\ell_t$ define a substring

$$S_{f,t} = \tau_f \cdots \tau_t,$$

i.e., the zero-or-more tokens (precisely $f - t$) between $\ell_f$ and $\ell_t$.

In the figure we see that $S_{4,4}$ is empty, that $S_{0,13}$ contains all tokens, that $S_{9,10}$ is made by the single token $\tau_9 = 5$, and that $S_{2,5}$ contains three tokens numbered 2 up until (but not including) 5: $\tau_2 = 2$, $\tau_3 = v$, and, $\tau_4 = 3$.

Any pair of two substrings is either

1. **Disjoint.** Figure 7.2 visualizes e.g., $S_{2,5}$ being disjoint of $S_{6,12}$ and $S_{0,3}$ being disjoint of $S_{4,4}$.

2. **Overlapping.** The only two overlapping substrings in the figure are $S_{0,3}$ and $S_{2,5}$.

3. **Strictly Contained.** In the figure we see e.g., $S_{9,10}$, contained in turn in $S_{7,12}$, contained in turn in $S_{6,13}$, contained in turn in $S_{0,13}$.

4. **Equal.** If two substrings are not disjoint, neither contained in each other, nor overlapping, then they must be equal.

Let

$$S = S(n) = \{S_{f,t} \mid 0 \leq f \leq t \leq n\}.$$  \hspace{1cm} (7.3)

be the set of all such substrings. Now the evaluation order question is firstly a matter of imposing a total order on set $S$. 

---

![Figure 7.2 Tokens, locations and substrings for the tokens string 1,2,3,(4,5,6)](image-url)
7.6 Total Order of Overruling

We say that a substring \( s = S_{f_1, t_1} \) overrules another substring \( s' = S_{f_2, t_2} \) if either one of the following two holds:

- it is strictly contained in it, \( S \subset S' \), i.e.,
  \[
  (f_2 < f_1 \leq t_1 \leq t_2) \lor (f_2 \leq f_1 \leq t_1 < t_2) \Rightarrow S'.\text{overrules}(S) \tag{7.4}
  \]

  or,

- if \( S' \) does not (strictly) contain \( S \), i.e., \( S' \not\subset S \), and \( S \) begins before \( S' \), formally,
  \[
  \neg((f_1 < f_2 \leq t_2 \leq t_1) \lor (f_1 \leq f_2 \leq t_2 < t_1)) \land (f_1 < f_2) \Rightarrow S'.\text{overrules}(S). \tag{7.5}
  \]

More generally, the overruling relationship is the order relationship generated as the transitive closure of the above two conditions.

We now show that overruling is well defined, i.e., that no cycles are do not lead to a cycle, and that it is a total order, i.e., that it is defined over all substrings. To do so, define a ranking function \( r : S \to \mathbb{N} \),

\[
  r(S_{f, t}) = \frac{t(t + 1)}{2} + t - f. \tag{7.6}
  \]

Figure 7.3 demonstrates that \( r \) defines a total order on \( S(7) \); the formal proof follows.

**Figure 7.3** The total order imposed by overruling on \( S(7) \), the set of substrings of a string of length 7

The following technical lemma is pertinent:
Lemma 7.1. Let $S_{f_1,t_1}, S_{f_2,t_2} \in \mathbb{S}$, $t_1 < t_2$. Then $r(S_{f_1,t_1}) < r(S_{f_2,t_2})$.

Proof.

$$
\begin{align*}
    r(S_{f_2,t_2}) &= \frac{t_2(t_2 + 1)}{2} + t_2 - f_2 \\
    &\geq \frac{(t_1 + 1)(t_1 + 2)}{2} + t_2 - f_2 \\
    &= \frac{t_1(t_1 + 1)}{2} + t_1 + 1 + t_2 - f_2 \\
    &> \frac{t_1(t_1 + 1)}{2} + t_1 - f_1 \\
    &\equiv r(S_{f_1,t_1}).
\end{align*}
$$

(1) By definition.

(2) Since $t_2 > t_1$, it must be that $t_2 \geq t_1 + 1$. The substituted expression is an increasing function.

(3) Simplifying the first expression.

(4)

$$
\begin{align*}
    \frac{t_1(t_1 + 1)}{2} + t_1 + 1 + t_2 - f_2 &> \frac{t_1(t_1 + 1)}{2} + t_1 - f_1 \\
    \text{iff} & \\
    t_1 + 1 + t_2 - f_2 &> t_1 - f_1 \\
    \text{iff} & \\
    1 + t_2 - f_2 &> -f_1.
\end{align*}
$$

The last equation holds because $t_2 - f_2 \geq 0$ (since, by the definition of a substring we have $0 \leq f \leq t$).

(5) By definition.

\[\square\]

Lemma 7.2. Function $r$ is a mapping (a one-to-one function).

Proof. Let $S_{f_1,t_1}, S_{f_2,t_2} \in \mathbb{S}$. If $t_1 < t_2$ then Lemma 7.1 implies that $r(S_{f_1,t_1}) \neq r(S_{f_2,t_2})$. Similarly for the case that $t_1 > t_2$. If $f_1 \neq f_2$, $t_1 = t_2$ then

$$
\begin{align*}
    \frac{t_1(t_1 + 1)}{2} + t_1 - f_1 &\neq \frac{t_2(t_2 + 1)}{2} + t_2 - f_2 \\
    \text{if and only if,} & \\
    \frac{t_1(t_1 + 1)}{2} + t_1 - f_1 &\neq \frac{t_1(t_1 + 1)}{2} + t_1 - f_2
\end{align*}
$$
if and only if,
\[ -f_1 \neq -f_2 \]
if and only if,
\[ f_1 \neq f_2. \]

**Lemma 7.3.** Function \( r(\cdot) \) follows the precedes desiderata eq. (7.5).

**Proof.** Equation (7.5) implies \( t_1 < t_2 \), and therefore, by Lemma 7.1 we have
\[ r(S_{f_1,t_1}) < r(S_{f_2,t_2}). \]

**Lemma 7.4.** Function \( r(\cdot) \) follows the containment desiderata eq. (7.4).

**Proof.** The equation \( r(S_{f_1,t_1}) < r(S_{f_2,t_2}) \), which translates to
\[ \frac{t_1(t_1 + 1)}{2} + t_1 - f_1 < \frac{t_2(t_2 + 1)}{2} + t_2 - f_2 \] (7.8)
holds since
\[ t_1 - f_1 < t_2 - f_2 \] (7.9)
and
\[ \frac{t_1(t_1 + 1)}{2} < \frac{t_2(t_2 + 1)}{2} \] (7.10)
(because \( t_1 < t_2 \)).

**Lemma 7.5.** The desiderata describes a total ordering.

**Proof.** Since \( \leq \) is a total ordering over \( \mathbb{N} \), and \( r \) preserves the order of substrings, a total ordering that conforms to the described desiderata exists.

### 7.7 Understanding Overruling

To understand the rationale of overruling, eqs. (7.4) and (7.5), and the figure examine the four cases in which \( s \) overrules \( s' \):

- the substrings \( s \) and \( s' \) are disjoint, and \( S_{f_1,t_1} \) begins and ends before \( S_{f_2,t_2} \) begins, or,
- the substrings \( s \) and \( s' \) overlap and \( S_{f_1,t_1} \) begins before \( S_{f_2,t_2} \) begins, or,
- substring \( S_{f_1,t_1} \) is strictly contained in \( S_{f_2,t_2} \), or,
- substrings \( S_{f_1,t_1} \) and \( S_{f_2,t_2} \) are equal, and the definition of the lexi of \( M_1 \) took place after the definition of the lexi of \( M_2 \).
Let us consider these four cases in order:

1. \( M_1 \) precedes \( M_2 \), that is to say,
   \[
   f_1 < f_2 \text{ and } t_1 < t_2.
   \]  
   The rationale is that if a certain substring both begins before another substring and ends before that substring, then a match against the first substring must be processed before a match with the second. Consider an enumeration that starts with the \texttt{Enum} word, and ends with a \texttt{Mune}:
   \begin{verbatim}
Enum
  ONE = iota
  TWO = iota
  NUM = iota
Mune
\end{verbatim}

As in the Go \cite{Go} programming language, the user may use the word \texttt{iota} to indicate a literal whose value is incremented by one any time it is used (in Go, the value is incremented in a slightly different way).

The order of matches, as probably expected, is \texttt{Enum}, \texttt{Iota} and then \texttt{Mune}.

2. \( M_1 \) is a prefix of \( M_1 \), or, in other words,
   \[
   f_1 = f_2 \text{ and } t_1 < t_2.
   \]  
   Continuing the previous example, and assuming the lexi
   \begin{verbatim}
@Find(Enum) Enum @run {global i; i = 1}
@Find(Iota) iota @replace @(i) @i+=1
@Find(Mune) Mune @log ('enum is over')
\end{verbatim}

we see that the lexi \texttt{Enum} is matched before lexi \texttt{Whole}.

The presented lexi is used to convert the special \texttt{Enum} to a C \texttt{enum}. Note that this lexi is not aware of the \texttt{Iota} lexi. A longer pattern should “understand” what it sees, i.e., what it contains. Its contents, or the contents of the \texttt{body} variable, should be based on the host language, rather than some macros that will be expanded later.

This rule, and the two following ones, are the reason that this pattern matches over the code after the rewriting.

3. \( M_1 \) is contained in \( M_2 \), that is to say,
   \[
   f_1 > f_2 \text{ and } t_1 < t_2.
   \]  
   According to this rule, the \texttt{Iota} lexi is matched before the \texttt{Whole} lexi.

Another interesting case that belongs to this category is the case of nested classes. In this case, it means that the inner class will match first.
4. \( M_1 \) is suffix of \( M_2 \), or, more formally,

\[
f_1 > f_2 \text{ and } t_1 = t_2. \tag{7.14}
\]

The reasoning is similar to previous rules, and is demonstrated in that the \texttt{Mune} lexi is matched before \texttt{Whole}.

5. \( M_1 \) and \( M_2 \) are equal, i.e.,

\[
f_1 = f_2 \text{ and } t_1 = t_2, \tag{7.15}
\]

and \( M_1 \)'s lexi was defined after \( M_2 \)'s lexi. We would like to enable the overriding of a previously defined lexi with a new lexi, since it is assumed that the latter is aware of the former.

For example, a definition of the lexi

\begin{verbatim}
    @Find iota @replace @(1 << i) @{i+=1}
\end{verbatim}

after the \texttt{Iota} lexi, and before the usage, will override the previous definition (the \texttt{Iota} lexi), resulting in an \texttt{iota} that yields a power of 2 each time it is accessed.

7.8 The Single Chance Principle and Earmarking

The representation of substrings and locations is such that the replacement of content of substring by another content, does not invalidate the substring data structure. A substring is invalidated only if one of its end points was substituted, being contained in another substring, disappeared when this containing substring was substituted itself.

As per the \textit{Single Chance} principle, a triggered lexi tracks (as long as possible) the range of tokens in the buffer string which it processed, regardless of how this range shrinks or enlarges, and will never match this range again. We say that the lexi is earmarked with this substring it manipulated, and will not reprocess it again.

In particular, this means that if lexi \( a \) was triggered on substring \( s \), and lexi \( b \) was triggered on a substring \( s' \subseteq s \), then \( a \) will not be triggered for the \( s \) even though it was partly or even fully modified.

Here is an example: When the two lexies

\begin{verbatim}
    @Find Hello @replace Hi
    @Find Hi @replace Hello Hello
\end{verbatim}

are applied to

\begin{verbatim}
    Hello
\end{verbatim}

After the first triggering the string of tokens will be

\begin{verbatim}
    Hi
\end{verbatim}

and after the second triggering
There will be no third triggering.

This earmarking technique works nicely with these two lexies, which seem to be mutually recursive:

```
@Find Hello @replace Hi
@Find Hi @replace Hello
```

When applied to `Hello` the buffer changes to `Hi` after the first triggering, and back to `Hello` in the second, and final, triggering.

### 7.9 The Inner Out Principle and Backtracking

At each input token LOLA scans in the buffer, it starts one listener for every lexi it stores. This listener tries to match the pattern starting at this token. Also, as tokens are read, all active listeners are notified. This notification can be replied by the listener, telling LOLA that a match was found. If the listener also detects that other, longer matches can be found as well, it forks, and the forked listened proceeds with the search for matches. The matched listener however creates a match data structure, which is then send to the grand match of matches.

LOLA maintains a concept of a current token in processing its buffer. Often this token is read and copied as is to the output stream. However, if one or more listeners mature at the current token, a lexi is triggered, changing the current buffer, the matched substring is removed and replaced with the output of this lexi.

After the substitution LOLA backtracks the current-token to the beginning of the replaced substrings, restoring the state of all pattern matchers as they were at that point. Lexies therefore get a chance to further transform the output of other lexies.

Nested matches of the same pattern are shown in Listing 7.1. The Inner to Outer principle says that triggering is on the matches the inner `squared`. The second triggering, only after the restart the second one will also match.

Listing 7.1 An example of how nesting of rewrites works

```
@Find squared @Literal(f)
@replace @(f * f)
@example double sixteen = squared squared 2.0;
@resultsIn double sixteen = 16.0;
```
Chapter 8

Conclusions and Future Work

In a world where new programming languages are introduced frequently but change slowly, in a world where the M4 preprocessor is a string substitution tool collecting dust, C preprocessor dying, and \TeX{} is baffling, \textsc{LOLA}, comes in to offer a new perspective on pre-processing.

\textsc{LOLA} maintains the usual pre-processing principles: non-intrusive extension to the programming language, and not any runtime impact, while supporting high-level programming language structure. \textsc{LOLA}'s should be useful for identifying structures in source files, and it has some capacity for reflection. \textsc{LOLA} suggests a flexible syntax for macro calls, for adding constructs that are more intuitive to the individual programmer, and that fit better in the specific programming domain.

My work was intended to make \textsc{LOLA} terse, elegant, readable, writable and language independent. Towards the end of readability, \textsc{LOLA}'s statements try to follow English like sentence structure, including capitalization conventions to make reading smoother: \texttt{@CamelCase} convention for constructors, \texttt{@camelCasé} convention for for elaborators, and absolutely no abbreviations or acronyms, not even for familiar terms such as EOF. (The rationale is this: Obviate the need to remember these times when acronyms and common abbreviations are used. This never happens). Elegance and brevity are achieved through a powerful model of pattern matching and code generation.

I hope that the fruits of my work can be appreciated by readers of this essay and that the resulting \textsc{LOLA} bears the potential of evolving into a tool that would be used by many programmers.

In many ways, \textsc{LOLA} supersedes everyday tools such as “grep”, “sed”, \textsc{AWK}, \textsc{PERL}, \textsc{PYTHON} and others, in the context of applying of their application to software. In this sense, \textsc{LOLA} is a software engineering tool: When put in the hands of able programmers it makes it possible to “engineer” the software itself, modify it, enhance it, measure it, and reason about it.

A combination of external scripts and other software tools can, and are, used, to, e.g., compute the Halstead metrics, or enforce the style rule (practiced mercilessly in Google) that C++ classes should never inherit from more than one class. With \textsc{LOLA}, the same deeds are carried out in a tongue that programs already speak: the host programming language itself, albeit augmented with \textsc{LOLA}'s syntax. And, it makes it possible to do all that in a reproducible, concise and modular fashion.

The work on \textsc{LOLA} turned out to be much larger than planned. The major challenges encountered on course were:

- Maintaining expressive power, elegance and brevity. Achievements toward meeting this
challenge can be seen in the examples gallery (Chapter 3).

- A simple theoretical model unifying the major language concepts including constructors, elaborators, lexies, generating expressions etc. (Chapter 4).

- The design of the configuration file so that it can adapt LOLA to many programming languages (Chapter 5).

- The unique use of indentation in the language, and the algorithm for parsing in presence of the three kinds of children: PYTHON snippets, the list, and the elaborators (Chapter 6).

- The exact semantics and ordering of evaluation of immediates and lexies (Chapter 7).

8.1 Future Research

Here are some directions for future research of major enhancements to LOLA.

- Currently, LOLA support for patterns that explore the contents of a trivia is limited to atomic patterns such as @NewLine and @EndOfFile and the @Unbalanced modifier. A future extension is to support finer grained pattern matching that makes it possible to further examine the contents of trivia. The need for such an extension may arise if LOLA is employed to process comments’ text, as may be necessary for the processing of JAVADOC.

- Similarly, the need for matching patterns against the contents of the string literals may arise. LOLA must be extended to be able to do this.

- More generally, it should be also possible change the host programming language on the fly, so that LOLA can handle XML structure embedded in JAVA, SQL embedded in AWK, etc. Such a generalization should supersede LOLA enhancements for processing strings and comments.

- Since LOLA is a language for extending languages, it is only natural to ask whether LOLA can be used for extending LOLA itself. Currently such a recursive application is not possible, mostly because LOLA directives are removed from the input immediately. But, just like many recursive questions of this sort, this research direction is likely to be fun more than useful.

- Keywords in LOLA are implemented using an API is intended to be accessible by mortal programmers. The implementation of a new keyword should be a matter of defining a simple and short PYTHON class. We wish to make it possible to define new LOLA keywords from within LOLA itself.
8.2 Lola’s Past and Planned Evolution

Candidly speaking, premature optimization concerns and the sheer challenge of implementing a Deterministic Finite Automaton (DFA) for conjunction and negation constructors of regular expressions, were the reasons why “Lola 0.032” was implemented with a DFA engine.

LOLA steps its versions in multiples of two, with this current short past and future grand plan:

**Lola 0.032** *First experimental implementation.* This was my initial, DFA based implementation. Examples in this thesis are based on experience gained while experimenting with this version. Available for demonstration purpose, but otherwise defunct.

**Lola 0.064** LOLA as specified in this document (\textsc{\LaTeX}ed on September 25, 2016), revising LOLA 0.032 with cleaner and tighter design, but still based on a DFA engine. *In preparation, might be canceled.*

**Lola 0.128** *Planned first public release.*

**Lola 0.256** *Cisco deployment.*

Revising the language design and implementation to use the less efficient recursive-descent pattern recognizer, with the advantage of clearer semantics. Based on ideas drawn from Snobol [23], the hypothesis is that the AST of the lexi compound objects is interpreted to be just an elegant way of writing a recursive descent parser for the pattern.

This addition should make it possible to evaluate a general Python expression, a *predicate*. A pattern will match only if the predicate evaluates to \texttt{True}, and backtrack otherwise.

**Lola 0.512** *Interim version.*

Introducing lexies that analyze spacing, strings and comments. Modifier for lexies to allow greedy and lazy semantics.

**Lola 1.024** *First industrial strength release.*

8.3 Reflection

I learned the hard way that relying on cryptic, above all, existing language paradigms is not the way to go. I believe that LOLA invents its own little paradigm made of by assembling concepts such as patterns, generating expressions, lexies, etc.

Concepts such as regular expressions and language embedding are more familiar, other, such as generating expressions are less, and others, such as elaborators, “match all possible” (rather than lazy or eager) semantics of regular expressions, or fixed point semantics of search and replace, are quite novel. The particular way in which LOLA puts all of these together is what makes it special.
Bibliography


This computer science paper dealt with the topic of computer science, focusing on the evolution of programming languages, particularly Python, and the introduction of optional features (for marking questions) using a symbol.

Modern computer work, especially C and its structure, is presented, allowing for a modular approach with loops and conditions. The combination of these elements is the basis for the evolution of programming languages.

In Unix, the concept of a file is fundamental, and the structure of the file system is presented, showing the relationship between files and directories. The programming language Python is introduced, showing its unique features and advantages.

This paper discusses the evolution of programming languages, focusing on the evolution of Python, its unique features, and its advantages over other languages. It introduces the concept of optional elements, allowing for a more flexible approach to programming.

The paper concludes by discussing the future of programming languages, with an emphasis on the evolution of Python and its potential impact on the field of computer science.
הערכה וה onloadנה

המחק נעשש בהנחיית פרופ' יוסף גל הפוטלשת למודי התמחשה

אני מודה למורר וThêm המתאחת על תקשורת, ה Printf וحصر התמחשה. אני אסף מורה לתחנה שלidf, Проf' יוסף גל. הום מורה
אותו באפסת במשר דבר,々יימע אאני ר_planes על מחקל. אני רוצה להודות לתוף' וודי לוחמים על תחושה על כדי
 mücadeה וה işוארה יועדו. תודה להורר, על טורעת הרבים.

אני מורה להישארה יועדו. תודה להורר, על טורעת הרבים.

אני מורה להישארה יועדו. תודה להורר, על טורעת הרבים.
לולה
שפת חכונה
ל_telephат שפת חכונה

דרובר על מחקר
לשם מיילוński חלקי של הדירישות לקבוצת החותם
מגיסטרים למדעי המחשב

עייד: א. זמיר

הוגש ישנה מט başına - מכון טכנולוגי לישראל
תכנית תشم"כ: חיפה, אפריל 2016