# Towards a Fuzzy Logic Environment for Android

Ramón Luján
Dept. of Computing Systems, UCLM
Albacete (02071), Spain
Email: Ramon.Lujan@alu.uclm.es

Ginés Moreno
Dept. of Computing Systems, UCLM
Albacete (02071), Spain
Email: Gines.Moreno@uclm.es

Carlos Vázquez
Dept. of Computing Systems, UCLM
Albacete (02071), Spain
Email: Carlos.Vazquez@uclm.es

***Abstract*—In the recent past, several fuzzy extensions of the popular pure logic language Prolog have been designed in order to incorporate on its core new expressive resources for dealing with uncertainty in a natural way. Following this trail, during the last decade we have developed our Fuzzy LOgic Programming Environment for Research ($\mathcal{FLOPER}$) for providing a practical support to programs coded with an interesting flexible language emerged into the Fuzzy Logic Programming arena. As an example, our system has recently served us for developing a real-world application devoted to the flexible management of eXtensible Markup Language (XML) documents by means of a fuzzy variant of the popular XPath language. Nowadays, $\mathcal{FLOPER}$ is useful on computer platforms for compiling (to standard Prolog code), executing and debugging (by drawing execution trees) fuzzy programs, and it is ready for being extended in the near future with powerful transformation and optimization techniques designed in our research group during the last five years. In order to increase the portability of the system, in this paper, we initiate a research path devoted to accommodate its core on Android platforms. Nowadays, the environment accepts lattices modeling truth-degrees beyond the simpler crisp case {true; false} together with a wide range of user-defined fuzzy connectives for manipulating such truth degree. Moreover, the tool is able to manage fuzzy program rules whose syntax is very close to the one of Prolog clauses but admitting on their bodies elements coming from the lattice of truth degrees. Executing fuzzy programs into an Android environment is now possible after using our tool for compiling the fuzzy code to standard Prolog clauses and then using any one of the currently available Prolog interpreters for Android systems.**

***Keywords–Fuzzy Logic Programming: Android; Compilers.***

## I. INTRODUCTION

Fuzzy logic programming is the area of computing intended to introduce fuzzy logic [1] into logic programming [2], i.e., to formally address the vagueness and imprecision found in human reasoning. The DEC-Tau research group of the University of Castilla-La Mancha is immersed on this topic since 2006, when the first prototype of $\mathcal{FLOPER}$ was developed (see [3], [4], and [5]). $\mathcal{FLOPER}$ is able to interpret a very general fuzzy logic language called Fuzzy Aggregators and Similarity into a Logic Language (FASILL), as described in [6] and [7]. $\mathcal{FLOPER}$ has been enterely developed using the Prolog language (see Figure 1), which it is a natural basis for supporting fuzzy logic programming. Figure 2 shows an screenshot of the graphical interface of $\mathcal{FLOPER}$. We wish to remark here that, as illustrated in Figure 3, the system has served us for developing the so called FuzzyXPath language [14] (both the interpreter and the debugger associated to this real-world application can be freely downloaded an even tested on-line via the web page presented in [15]).

The $\mathcal{FLOPER}$ tool was originally thought to be executed in the environment of a personal computer or an equivalent machine. However, as time goes on mobile devices have spread and made computation ubiquitous. Our objective now is to update $\mathcal{FLOPER}$ to be present in those devices.

In this paper, we adapt fuzzy logic programming to mobile devices (smartphones, tablets, etc.), by presenting a version of the experimental prototype of $\mathcal{FLOPER}$ for the Android operating system. The objectives of this work include: 1) to make known logic programming to an increasing number of people, and 2) to ease the development of fuzzy logic programs.

There is no standard fuzzy logic programming language. Furthermore, there are two main trends (not necessarily opposing) to fuzzify logic programming (and, in particular, the standard logic programming language, Prolog):

1) One family, that includes languages like Likeness in Logic (LIKELOG) [8], replaces the syntactic unification mechanism by some fuzzy unification algorithm. This unification algorithm can be based on different theoretical frameworks, of which one of the most successful is the use of similarity relations. In this family of languages, resolution remains basically unaltered, while unification is modified to consider that two different symbols (both predicates or both functions of the same arity) are “equal” at some degree defined by the established similarity relation. In this framework, programs are accompanied by a definition of a similarity relation, that may be in the form of a set of similarity equations, as is the case of BOUSI~PROLOG [9], developed in the DEC-Tau group.

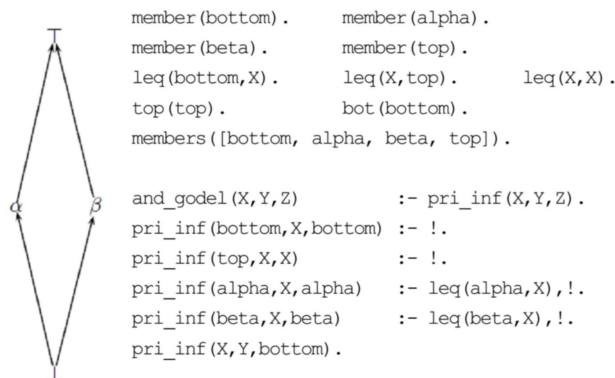


Figure 1. Lattice of truth degrees modeled in Prolog for being used by the $\mathcal{FLOPER}$ environment.

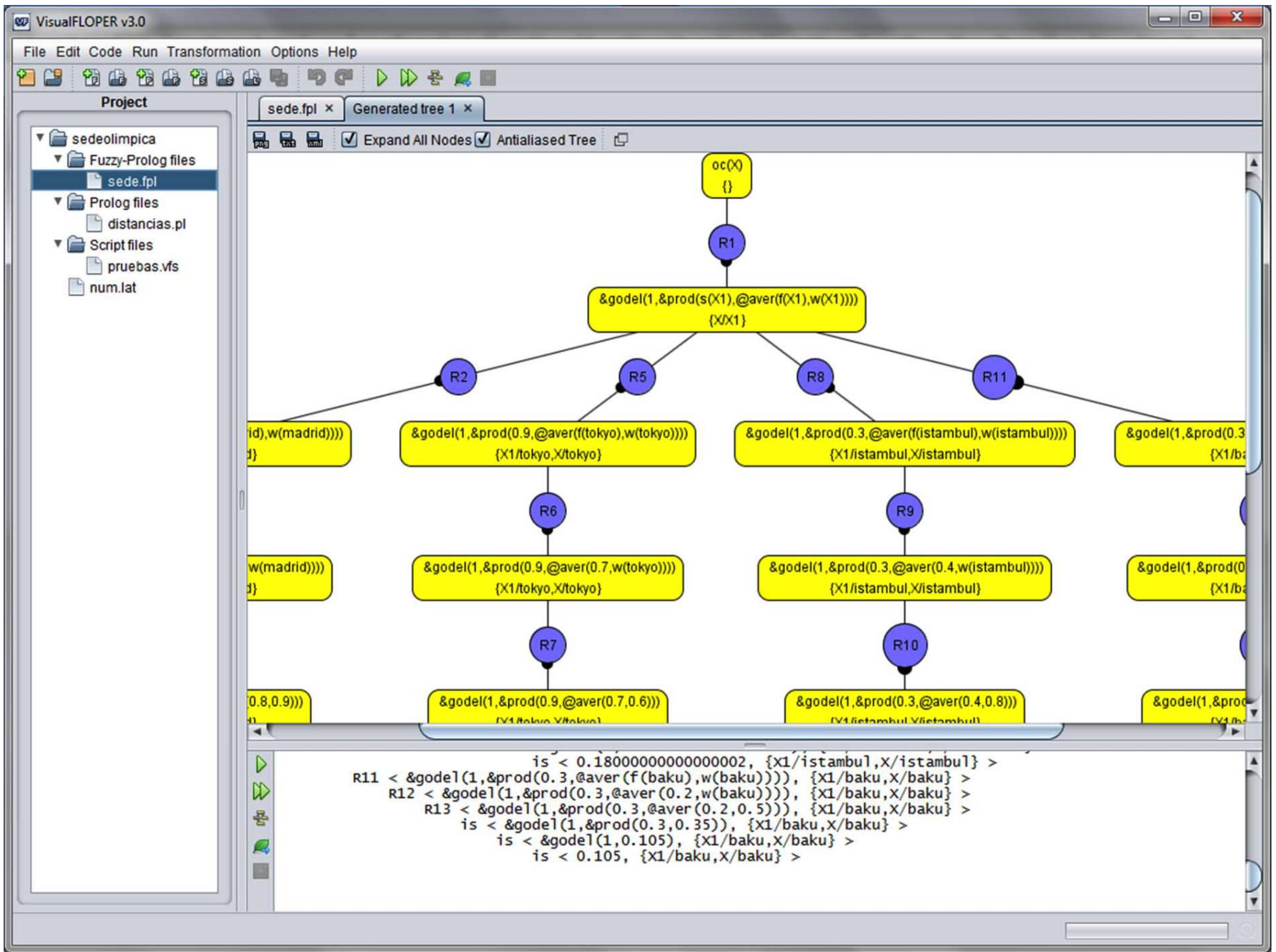


Figure 2. When installed on a computer, programmers can comfortably interact with the FLOPER system thanks to its improved graphical interface.

*SLD-Resolution ("Selective Linear Definite clause resolution") + Fuzzy (similarity) unification*

2) In the other approach, programs are fuzzy subsets of formulae, where the membership degree of a formula to the subset is called its *truth degree*, and represents the confident the program has on that rule. Truth degrees are propagated through the modified resolution principle, while the unification mechanism remains unaltered. One example of these languages is Multi-Adjoint Logic Programming (MALP) [10].

   *Fuzzy SLD-Resolution + (syntactic) unification*

3) The last two approaches have been amalgamated in the form of the FASILL language, which modifies both the resolution principle and the unification mechanism of the original languages without introducing important changes on the final syntax.

   *Fuzzy SLD-Resolution + Fuzzy (similarity) unification*

In this paper, we focus on the implementation of a programming environment based on fuzzy logic over Android, with the aims of compiling, visualizing, editing, creating and saving fuzzy logic programs in FASILL and their associated lattices of truth degrees. This environment has been implemented as an Android application using the Java language and the Eclipse Integrated Development Environment (IDE) in its kepler version.

The syntax of FASILL is similar to Prolog in the way functions and predicates are built, but it includes the advances of the multi-adjoint logic. Each program rule is accompanied by a truth degree that is an element of the associated lattice. Rules have an atomic head and a body, which is built up from atoms and truth degrees linked by connectives. Those connectives can be conjunctions, disjunctions or hybrid operators called aggregators, and can be defined by the logics of Gödel, Łukasiewicz, Product, or any other logic defined by the programmer. Furthermore, notice that all that is required for truth degrees is that belong to the multi-adjoint lattice associated to the program, so this framework goes beyond the $[0, 1]$ domain that limits the majority of fuzzy logic languages (see again Figure 1).

In this work, we provide a solution based on three main classes: a lexical analyzer for the new language containing the definition of its lexical categories; a syntactic analyzer that parses the grammar of FASILL; and a translator that produces

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-461-9 

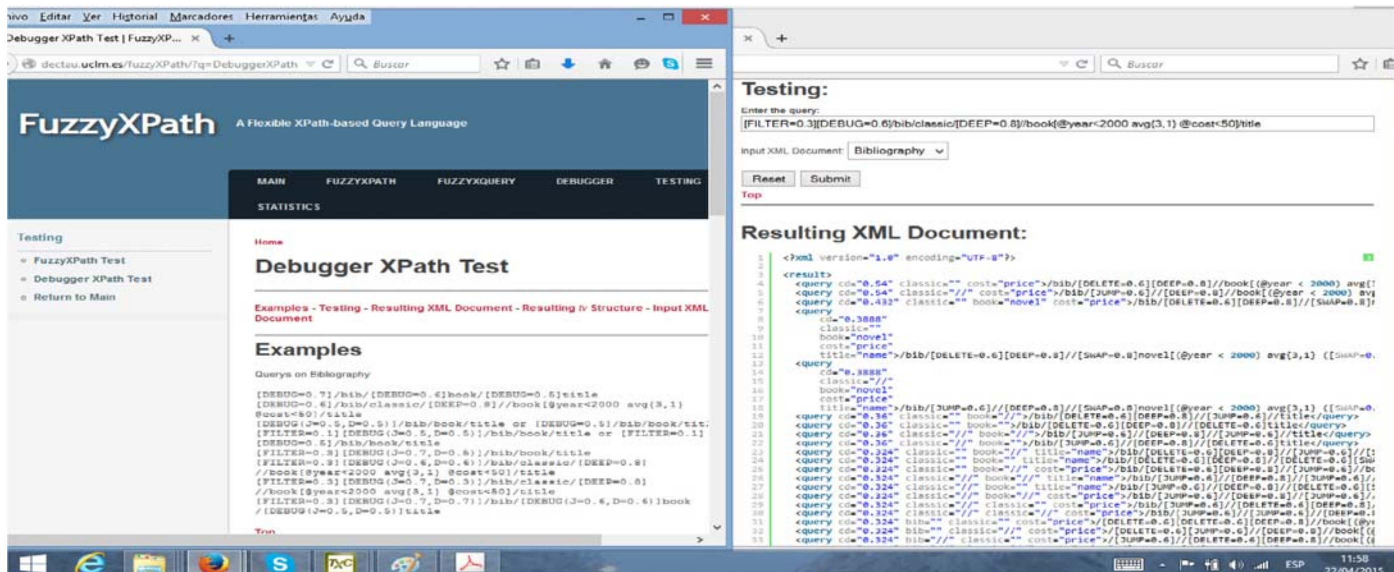


Figure 3. An on-line session with the *FuzzyXPath* debugger developed with $\mathcal{FLOPER}$.

an object program from a source program following the rules that we detail in the next sections. As part of our goal, our platform have to report possible errors in the source program detected in the lexical or syntactic analysis. Those reports must clarify where in the code is the error, its type and also a possible cause of it.

The structure of this paper is as follows. Section II resumes the main elements involved in the construction of a compiler. Next, Section III details the Android-based implementation of our tool. Finally, in Section IV, we conclude by also proposing several lines for future research.

## II. Language Processors

A compiler is a software tool such that one of its main input data is a language, as explained in [11] and [12]. There are many different kinds of language processors, including compilers and interpreters. Compilers are programs which can read a program written in a language (called the source language) to translate it to an equivalent program in another language (called the target language). The source program is usually a high-level language and the target language is usually a machine language.

Unlike a compiler, an interpreter is a language processor which executes a source program to return the result of that execution [11]. According to Louden [12] and Scott [13], the target machine language program that produces a compiler is usually faster than an interpreter at the moment of assigning the inputs to the outputs, because a decision made by compile time is a decision that does not need to be made at run time. However, an interpreter may provide better diagnoses because it executes the source program instruction by instruction.

Both of them, compilers and interpreters, are really complex tools. They may have around ten thousand lines of code [12]. Luckily, in spite of this complexity, as the knowledge and the tools to structure them are known, the complexity is reduced. The tasks of these kind of language processors are explained below (see Figure 4):

1) Analysis: the analysis divides the program into components and imposes a grammatical structure on them. Then, it uses this structure to create an intermediate representation of the source program. If the analysis detects that the source program is malformed in terms of syntax or semantics, then it must provide messages for the user to correct it.
    a) Scannings: the scanner, or lexical analyzer, reads the stream of characters that make up the source program and join them together to produce tokens (each token is a character string representing a unit of information in the source program). These tokens are classified into categories, which are defined using regular expressions. Typical examples of categories are reserved words, identifiers, special symbols, constants, etc. When the lexical analyzer finishes its analysis, it sends those tokens to the parser in order to help it to analyze the structure of the program.
    b) Parsing: the parser, or syntax analyzer, uses the tokens and grammatical rules of a context-free grammar in order to build an intermediate representation in a tree which describes the grammatical structure of the flow of tokens. The most common representation is the Abstract Syntax Tree (AST), wherein each internal node represents an operation and its descendants represent the arguments of the operation. An example of the abstract syntax tree is seen in Figure 5.
    c) Semantic analysis: the semantic analyzer uses the syntax tree to check the semantic consistency of the program. It checks that semantic language restrictions are met. Some examples may be unable to add a character and an integer, or not to use a variable that has not been previously declared.

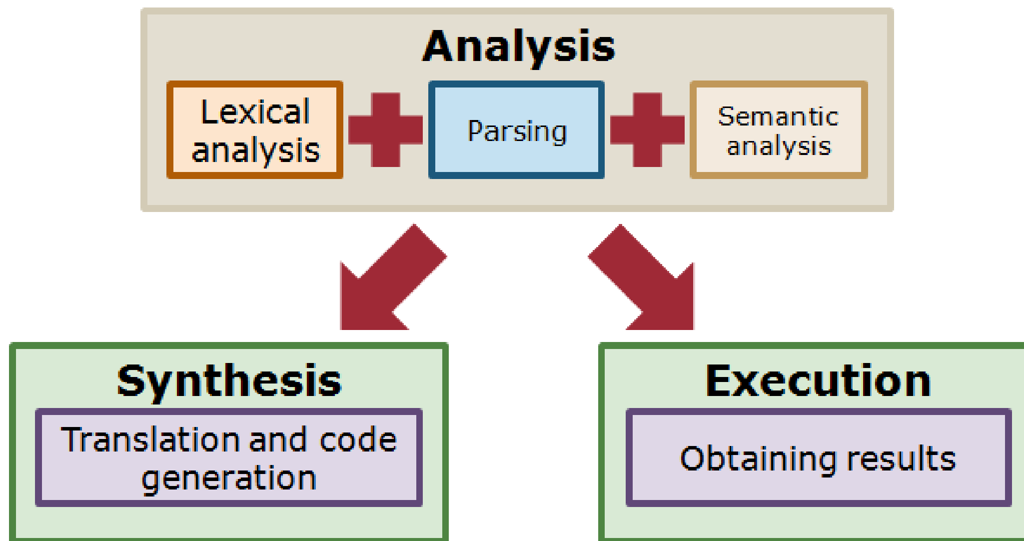


Figure 4. Translation stages.

2) Synthesis (compilers): it builds the desired target program using the intermediate representation.
3) Execution (interpreters): it executes the program using the intermediate representation and returns the results.

The aim of compilers is to obtain a translation from a source language to an object language, while the aim of interpreters consists only in the program execution and getting the outcomes (within this investigation a compiler has been developed, but the functions of an interpreter will be added in the following versions).

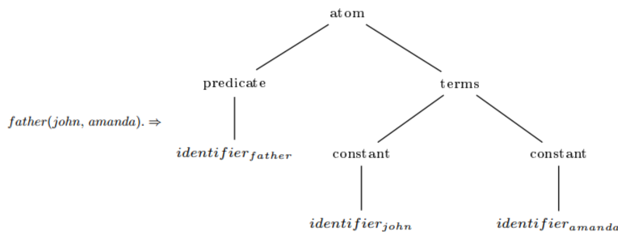


Figure 5. Abstract syntax tree.

TABLE I. SPECIFICATION OF LEXICAL CATEGORIES.

| Lexical category | Pattern | Attributes |
| --- | --- | --- |
| Identifier | [a-z][_a-zA-Z0-9]* | Lexeme, line and column |
| Variable | [A-Z][_a-zA-Z0-9]* | Lexeme, line and column |
| Number | [0-9]+(.[0-9]+)? | Lexeme, line and column |
| Comment | %(.)* | Lexeme, line and column |
| Connective | [@&\|] | Lexeme, line and column |
| LPAREN | [(] | Line and column |
| RPAREN | [)] | Line and column |
| WHITE | [ \t\r] | Line and column |
| NL | [\n] | Line and column |
| COMMA | [,] | Line and column |
| PERIOD | [.] | Line and column |
| FROM | (<-) | Line and column |

## III. Implementation Based on Android

In this section, we detail how this compiler was developed, explaining the implementation of the lexical analyzer and the parser in detail. It has not been necessary to implement a semantic analyzer because the grammar of this compiler only admits Horn clauses. But first, we briefly explain the main reasons why Android has been selected as the operating system and in particular Android tablets for the implementation of this compiler.

The first reason is that many people use this operating system in their daily life; it is also really easy to get a device that works with Android. In addition, tablets have been selected as the target device because of their advantages. A tablet offers better mobility than other devices such as computer towers and laptops. Also, tablets have a better display than mobiles, which could improve the interactions between the user and the application in order to make them more comfortable. Finally, tablets are more powerful than mobiles, which is very useful for a programming environment.

Before proceeding with the implementation of the compiler in Android, it must be clarified that in this version of the compiler is the syntax analyzer which requests the tokens to the lexical analyzer when it needs them. This is known as a translation guided by the syntax.

### A. Analysis

As it was mentioned in the section of Language Processors, this is the stage in which the AST corresponding to the source program is obtained.

Below we detail the development of the lexical analyzer and the parser, showing the lexical categories and the rules which have been implemented to the language of the compiler.

*1) Lexical Analysis:* As it was mentioned before, in this stage the input is split up in tokens to be clasified in lexical categories later. Also, these tokens contain attributes which could be useful to following stages.

Table I shows the different lexical categories which have the compiler. It is easy to see patterns or regular expressions, which are used to define them, besides the attributes of the categories. Line and column attributes are used to give

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-461-9

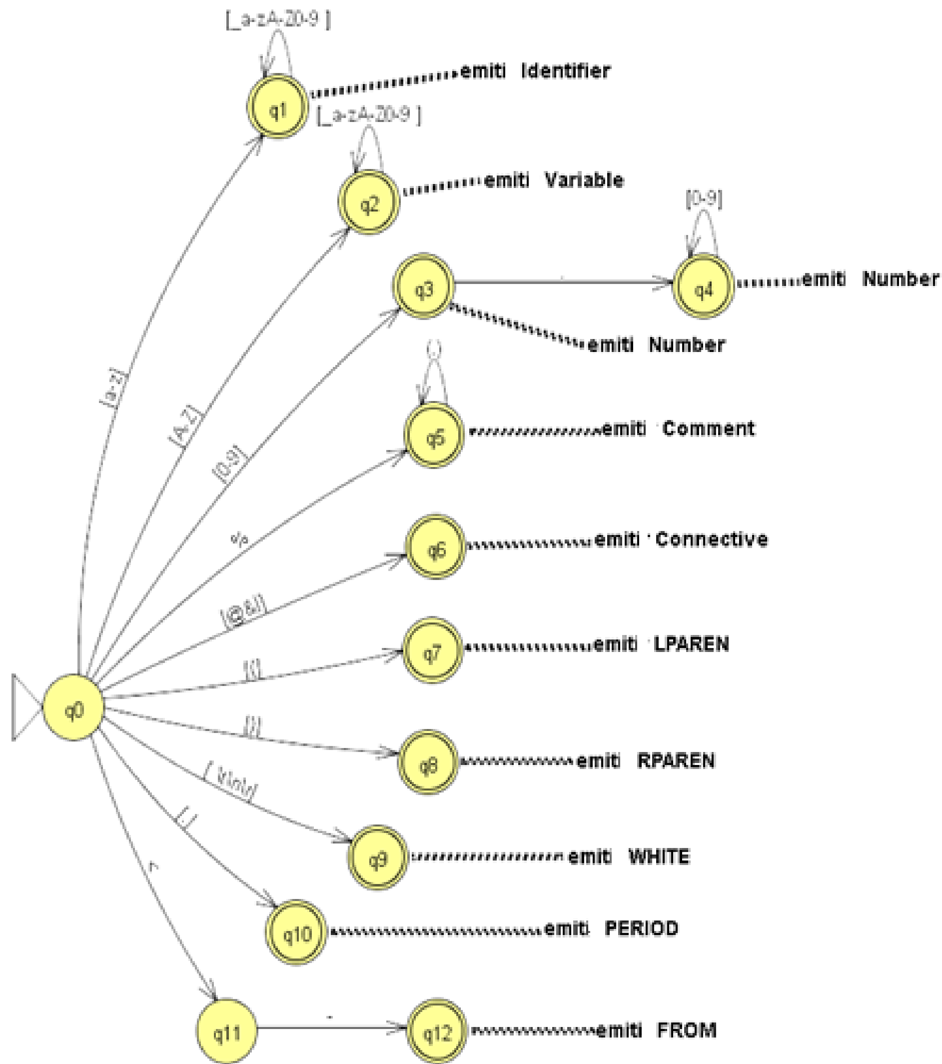


Figure 6. DDM associated to the language of the compiler.

information about the position of an error, either a lexical error or a syntax error, while the lexeme attribute is used to store the value of a token, which is used in the stage of synthesis.

To be able to perform the lexical analysis, a Deterministic Discriminating Machine (DDM) has been built. It splits the input up into individual characters and acts on them repeatedly, starting each time on a different point, but it always starts in its initial state.

The DDM follows a greedy strategy. This means that it searches the biggest token that belongs to a lexical category on the portion of input that has not been analyzed yet. Thus, splitting the input exactly matches that expected.

This DDM has been developed by Java code using the if-else sentence to compare patterns with the current character of the input. Below, the DDM associated to the language of the compiler will be shown in Figure 6.

*2) Parsing:* As it was commented before, the syntax analyzer is in charge of discovering the structures of the code using context-free grammars and the tokens which are sent from the lexical analyzer. These grammars have enough expressive power to represent most of the constructs that are in the compiler. Also, they are easy to develop and lead to a very efficient analysis.

From tokens and the context-free grammar defined for this compiler, the syntax analyzer builds a parse tree, which contains information about how to make the input using grammatical rules.

The syntax analysis is a descendent process. This means that the tree is built from the root to the leaves. This process is needed to predict what the next tokens are going to be; this is why it is also known as predictive analysis.

Below is shown the grammar used to develop this compiler:

- $\langle Program \rangle$ = $\langle Line \rangle$ (NL $\langle Line \rangle$)*: programs are composed by one or more lines separated by new lines.
- $\langle Line \rangle$ = $\langle Rule \rangle$|$\langle Goal \rangle$|COMMENT: lines can be rules, goals or comments. Unlike other compilers, in this compiler comments are considered in the analysis because it has been considered important that the comments will appear in the translated program.
- $\langle Goal \rangle$ = GOAL $\langle Body \rangle$ PERIOD: despite of the application not having the functionality of an interpreter, it include the rules necessary to analyze goals.

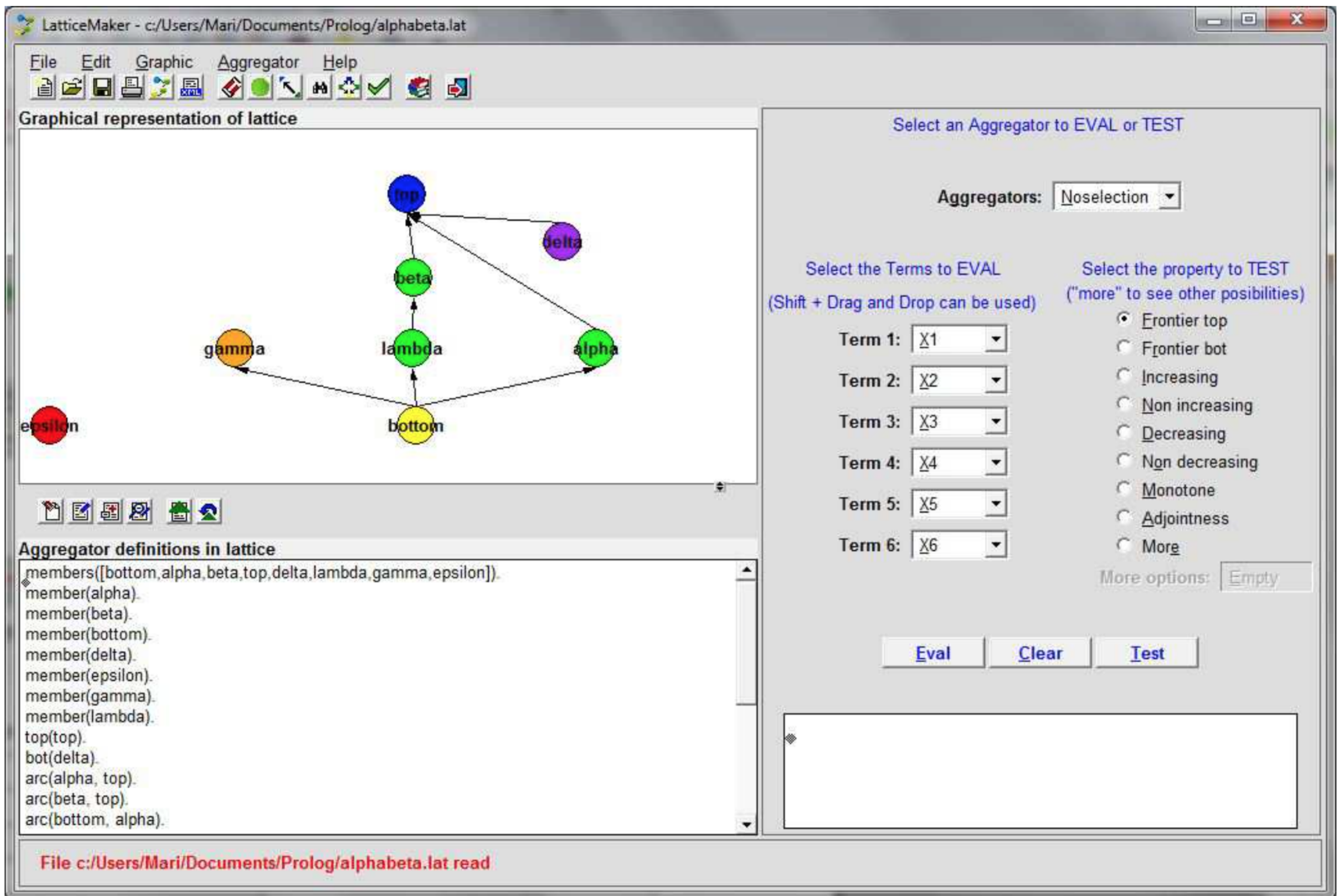


Figure 7. Screenshot of the *"LatticeMaker"* tool assisting the graphical design of a lattice of truth degrees.

- $\langle Rule \rangle$ = $\langle Head \rangle$ (FROM $\langle BODY \rangle$)? PERIOD: rules can be FASILL facts (rules without a body) or rules with a head and a body.
- $\langle Head \rangle$ = $\langle Atom \rangle$: a head is an atom.
- $\langle Atom \rangle$ = $\langle Predicate \rangle$ (LPAREN $\langle Terms \rangle$ RPAREN)?: an atom is a predicate which could contain a term list.
- $\langle Terms \rangle$ = $\langle Term \rangle$ (COMMA $\langle Term \rangle$)*: a term list consist on terms separated by commas.
- $\langle Term \rangle$ = IDENTIFIER (LPAREN $\langle Terms \rangle$ RPAREN)?|VARIABLE: a term can be a function (an identifier which in this case would be a function symbol followed of a term list), a variable or a constant (a single identifier).
- $\langle Body \rangle$ = $\langle Atom \rangle$ | $\langle TruthDegree \rangle$ | $\langle Connective \rangle$ LPAREN $\langle ArgumentList \rangle$ RPAREN: a body can be an atom, a truth degree or a connective with an argument list.
- $\langle TruthDegree \rangle$ = IDENTIFIER | NUMBER: a truth degree can be an identifier or a number, depending on how the lattice has been defined.
- $\langle Connective \rangle$ = CONNECTIVE IDENTIFIER?: connectives consist on a connective symbol (@, & o |) which can be followed of an identifier tag.
- $\langle ArgumentList \rangle$ = $\langle Body \rangle$ (COMMA $\langle Body \rangle$)*: an argument list consist on bodies separated by commas.

### B. Synthesis

In this stage, the program with the FASILL syntax is translated to Prolog. A translation is generated from the tree made in the stage of analysis. These are the rules that have been used to generate the translation:

- Atoms: every atom written in FASILL is translated to Prolog by adding a truth degree. In some cases it will be a variable TVX (where X corresponds to an index depending on the position of the atom) which will contain the value and in other cases it will be the value.

  q(X, Y). ≡ q(X, Y, TV0).
- Facts: these atoms will be translated to Prolog by adding the maximum truth degree of the lattice, as a fact is always true in fuzzy logic.

  q(X, Y). ≡ q(X, Y, 1).
- Rules with body: there are two different cases.
  - The body is a truth degree: this case is very simple to treat, because it is translated as a Prolog fact with the truth degree added.

    q(X, Y) <- 0.6. ≡ q(X, Y, 0.6).
  - The body consists on a connective with arguments: in this case atoms will be translated in order, following the steps described above. The translation of the connectives consists on an

atom whose parameters are the variables of the truth degrees of the atoms which it contained, added to its own truth variable.

q(X, Y) <- &prod(0.6,p(X)). ≡ q(X, Y, TV0) :- p(X,TV1), and_prod(0.6,TV1,TV0).

## IV. Conclusions and Future Work

In this paper, we have described the first prototype of $\mathcal{FLOPER}$ for Android, which tries to facilitate the access to fuzzy logic programming to the people.

Our application is still in an early stage of development and can be improved in several ways, that will constitute our focus of attention. We plan to implement the similarity management present in the personal-computer version of $\mathcal{FLOPER}$, as well as an interpreter able to evaluate program goals. Furthermore, we intend to introduce a mechanism to allow the graphical design of lattices, which would greatly facilitate the creation and visualization of truth degree lattices (in Figure 7 we show a preliminary tool that we have recently developed in this sense for computers [16]).

Other possibilities concern system usability. We intend to improve the error detection system in both the lexical and syntactic levels, so as to provide more information related to errors. Also, since smartphone and tablet keyboards are neither easy nor comfortable to use, we intend to provide direct access to the most commonly used symbols in FASILL programs, like "<-", ".", "(", ")", etc. In this sense, we also plan to introduce word completion for identifiers and variables in order to gain ease of use.

## Acknowlegments

This work was supported by the EU (FEDER), and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grant TIN2013-45732-C4-2-P.

## References

[1] L. A. Zadeh, "Fuzzy logic and approximate reasoning," Synthese, vol. 30, 1965, pp. 407–428.

[2] J. Lloyd, Foundations of Logic Programming. Springer-Verlag, Berlin, 1987, second edition.

[3] J. Abietar, P. Morcillo, and G. Moreno, "Designing a Software Tool for Fuzzy Logic Programming," in Proc. of the ICCMSE'07 International Conference of Computational Methods in Sciences and Engineering, Corfu, Greece, September 25-30, T. Simos and G. Maroulis, Eds., vol. 2. American Institute of Physics, 2007, pp. 1117–1120, distributed by Springer Verlag. ISBN 978-0-7354-0478-6.

[4] G. Moreno and C. Vázquez, "Fuzzy logic programming in action with FLOPER," Journal of Software Engineering and Applications, vol. 7, 2014, pp. 273–298.

[5] DEC-TAU research group (University of Castilla-La Mancha). The Fuzzy Logic Programming Environmment for Research FLOPER, web page at http://dectau.uclm.es/floper/?q=sim, 2015.

[6] G. Moreno, J. Penabad, and C. Vázquez, "Beyond multi-adjoint logic programming," International Journal of Computer Mathematics, vol. 92, 2014, pp. 1956–1975.

[7] P. J. Iranzo, G. Moreno, J. Penabad, and C. Vázquez, "A fuzzy logic programming environment for managing similarity and truth degrees," in Proceedings XIV Jornadas sobre Programación y Lenguajes, PROLE 2015, Cadiz, Spain, September 16-19, 2014., ser. EPTCS, S. Escobar, Ed., vol. 173, 2015, pp. 71–86. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.173.6

[8] F. Arcelli and F. Formato, "Likelog: A logic programming language for flexible data retrieval," in Proc. of the ACM Symposium on Applied Computing, SAC'99, San Antonio. ACM, Artificial Intelligence and Computational Logic, 1999, pp. 260–267.

[9] P. Julián and C. Rubio, "An efficient fuzzy unification method and its implementation into the Bousi∼Prolog system," in 2010 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE'10, July 18-23, Barcelona, IEEE, Ed., 2010, pp. 658–665.

[10] J. Medina, M. Ojeda-Aciego, and P. Vojtáš, "Multi-adjoint logic programming with continuous semantics," in Proc. of Logic Programming and Non-Monotonic Reasoning, LPNMR'01, Vienna, Lecture Notes in Artificial Intelligence 2173, 2001, pp. 351–364.

[11] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley, August 2006. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0321486811

[12] K. C. Louden, Compiler Construction: Principles and Practices. Boston, MA, USA: PWS Publishing Co., 1997.

[13] M. L. Scott, Programming Language Pragmatics. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009.

[14] J. M. Almendros-Jiménez, A. L. Tedesqui, and G. Moreno, "Fuzzy xpath through fuzzy logic programming," New Generation Computing, vol. 33, no. 2, 2015, pp. 173–209. [Online]. Available: http://dx.doi.org/10.1007/s00354-015-0201-y

[15] DEC-TAU research group (University of Castilla-La Mancha). The fuzzyXPath language, interpreter and debugger, web page at http://dectau.uclm.es/fuzzyXPath/, 2010.

[16] J. Guerrero, M. Martínez, G. Moreno, and C. Vázquez, "Designing lattices of truth degrees for fuzzy logic programming environments," in 2015 IEEE Symposium Series on Computational Intelligence: Symposium on Foundations of Computational Intelligence (2015 IEEE FOCI), Cape Town, South Africa, 2015, pp. 10 (In press, IEEE).