An elementary fuzzy programming language

Rafael Morales-Bueno, Ricardo Conejo and José Luis Pérez de la Cruz
Dpto. Lenguajes y Ciencias de la Computación, Facultad de Informática, Universidad de Málaga, 29071-Málaga, Spain

Buenaventura Clares
Dpto. Lenguajes y Sistemas Informáticos, Facultad de Ciencias, Universidad de Granada, 18071-Granada, Spain

Received September 1992
Revised December 1992

Abstract: In this paper, we define a fuzzy imperative programming language (called L), powerful enough to express crisp and fuzzy algorithms. Its small instruction set is designed to enhance readability, and it is proved before that the class of functions that can be programmed in L is equivalent to the class of functions computed by a W-Turing machine and to the W-recursive functions. The language is defined by means of an attribute grammar, and we present some examples of programs and their execution. A compiler has been built for the language, whose features are briefly described.

Keywords: Fuzzy programming; fuzzy computability; fuzzy language.

1. Introduction

Many papers have been devoted to the formal introduction and characterization of the intuitive notions of fuzzy algorithms and functions computable by a fuzzy algorithm [23, 4, 19, 5], fuzzy recursive function [7], computability of fuzzy predicates [6], and fuzzy programming [4, 21, 20]. Some of them define a formal model of the fuzzy algorithm and its properties. Others delimit the meaning of a fuzzy program with respect to these models. In short, they set up some formal criteria for the execution of a fuzzy program in a crisp or fuzzy model of computation.

Other papers develop some fuzzy programming languages: L.P.L. [1] and [2], FRIL [3], HALO [8], or describe the execution of some programming instructions [17]. L.P.L. is a fuzzified version of PL1 used for solving combinatorial and syntactic pattern recognition problems. It offers high level data structures and a wide set of control instructions. FRIL is a query language inspired by relational data base theory. Ostasiewicz [17] defines the execution of fuzzy assignment statements, and the fuzzy sets comparison applied to ‘IF’ and ‘WHILE’ statements. Clark [8] introduces the language HALO, oriented to the development of fuzzy intelligent systems, with LISP syntax and PASCAL semantics (with fuzzy features). Hence, it also deals with high level data structures and control instructions.

These languages are based upon high level languages and are suited for specific application development. On the other hand, our language (L), is very simple. It has a small instruction set and only one data type. It is not application oriented. One of its aims is to model the concepts of fuzzy computability.

Correspondence to: Dr. R. Morales-Bueno, Dept. Lenguajes y Ciencias de la Computación, Facultad de Informática, Plaza de El Ejido, 29013 Malaga, Spain.
Since the underlying model of calculus is a fuzzy finite state automata, it is natural to define the semantics of L from an operational, non-deterministic point of view. This approach allows parallel execution of all possible actions, so following a trend indicated by Prade [18]. In this way, the outputs of an execution of an L program is a fuzzy set. The non-fuzzy features of the language are similar to those developed in [12] and [13] for a crisp language.

The L language is defined by means of an attribute grammar with fuzzy attributes. In the following paragraph the notation used in the rest of paper is introduced. In Section 3 the L language is defined. In Section 4 the implementation is briefly commented upon. In Section 5 two examples are developed.

2. Notation and previous results

In the following we will consider that W is the ordered semiring ([0, 1], +, ⊗, <) where the degrees are evaluated [20].

We will use W-sets over \((\mathbb{Z}^+)\)\(^n\), \(n \geq 1\), with the standard representation: \(A = \mu_A(a)/a + \cdots + \mu_A(z)/z\). If \(a = (a_1, \ldots, a_n) \in (\mathbb{Z}^+)\)\(^n\) then \(a_i\) represents the \(i\)th-projection \(a_i\).

Let \(U\) and \(V\) be sets, \(U, V \subseteq (\mathbb{Z}^+)\)\(^n\). We consider that a W-function \(f\) from \(U\) into \(V\) is a function from \(U \times V\) into \(W\), and we write \(f: U \times V \rightarrow W\), being \(f(u, v)\) the weight or degree that the value of the function at \(u\) is \(v\).

In the following we are presenting, briefly, the notations used for attribute grammar. This notation is taken from [11].

\(S\) is a finite non-empty set,
\(D\) is a tree domain,
\((D, e)\) is an S-tree, with \(e: D \rightarrow S\) the labelling mapping,
\(\text{subtree}((D, e), w)\) is the subtree of \((D, e)\) at node \(w\),
\(\text{subst}((D, e), w, (D', e_1))\) is the tree obtained by replacing in \((D, e)\), \((D', e_1)\) at node \(w\),
\(\text{frontier}((D, e))\) is the sequence of labelling \(e\) of leaves nodes in lexicographic order.

**Example 1.** The following example shows the application of these concepts. \(D = \{e, [1], [2], [3], [1, 1], [1, 2], [2, 1], [3, 1], [3, 2], [3, 3]\}\) is a tree domain, whose graphical representation is as follows:

```
       e
      / \  \
     [1]  [2]  [3]
    /|\  /|\  /|\  \
[1,1] [1,2] [2,1] [3,1] [3,2] [3,3]
```

Let \(S\) be the set \(S = \{A, B, C, a, b, c\}\), and let \(e\) be the labelling mapping \(e: D \rightarrow S\) defined as follows:
\(e(e) = A,\ e([1]) = C,\ e([2]) = A,\ e([3]) = B,\ e([1, 1]) = b,\ e([1, 2]) = C,\ e([2, 1]) = a,\ e((3, 1)) = a,\ e([3, 2]) = c,\ e([3, 3]) = B\). Then \((D, e)\) is an S-tree, whose graphical representation is as follows:

```
       eA
      / \  \
   /|\  /|\  /|\  /|\  \
```
Subtree \(((D, e), [3])\) is the following S-tree:

\[ \epsilon B \]


To illustrate the `subst` operation, let us consider the following S-tree \((D_1, e_1)\):

\[ \epsilon C \]


\[ [1, 1]A \quad [1, 2]B \quad [2, 1]b \quad [2, 2]A \]

Then \(\text{subst} \(( (D, e), [3], (D_1, e_1) \))\) is an S-tree with the following graphical representation:

\[ \epsilon A \]


\[ [1, 1]b \quad [1, 2]C \quad [2, 1]a \]

\[ [3, 1]B \quad [3, 2]C \]

\[ [3, 1, 1]A \quad [3, 1, 2]B \quad [3, 2, 1]b \quad [3, 2, 2]A \]

Finally, \(bCaABbA\) is the frontier for the last S-tree.

**Note.** For the rules of a type 2 grammar \(G = (N, T, R, S)\), we write \((A, u) \in R\) or \((A \rightarrow u) \in R\).

**Definition 1** [11]. Let \(G = (N, T, R, S)\) be a type 2 grammar, \(V = N \cup T, A \in N\) and \(u \in V^*\). We say that the \((V \cup \{e\})\)-tree \((D, e)\) is a derivation tree from \(A\) to \(u\) in \(G\), if

(a) \(e(e) = A\).

(b) For each \(d \in D\) with children \(d_1, d_2, \ldots, d_k\) and \(k \geq 1\):
   - either \(e(d), e(d_1)e(d_2) \cdots e(d_k) \in R\) and \(e(d_i) \neq e, 1 \leq i \leq k\),
   - or \(k = 1, e(d_1) = e\) and \((e(d), e) \in R\).

(c) \(\text{frontier} \((D, e)\) = u\).

**Example 2.** Let \(G = (N, T, R, S)\) be a type 2 grammar, where \(N = \{A, B, C, S\}, T = \{a, b, c\}\), and \((A, CAB), (C, bC), (A, a), (B, acB) \in R\). Then the S-tree \((D, e)\) in Example 1 is a derivation tree from \(A\) to \(bCaacB\) because:

(a) \(e(e) = A\).

(b) At node \(e\) the rule \((A, CAB)\) is applied, at node \([1]\) the rule \((C, bC)\) is applied, at node \([2]\) the rule \((A, a)\) is applied and finally at node \([3]\) the rule \((B, acB)\) is applied.

(c) \(\text{frontier} \((D, e)\) = bCaacB\).

We now consider the concepts of `attribute` and `semantic rule`. A way to describe the semantic of a language is to associate certain symbols of the grammar with their features or properties by means of attributes. The assignment of meaning to a sentence can be seen as the computation of suitable values.
R. Morales-Bueno et al. / An elementary fuzzy programming language

for the attributes. Semantic rules describe this computation and check the consistency of these values (as we will explain in Definition 4).

**Definition 2.** Let $G = (N, T, R, S)$ be a type 2 grammar, $V = N \cup T$. Let $\text{Att}$ be a finite set $\text{Att} = \{a_1, \ldots, a_n\}$, composed by the attributes of the members of $V$. For each $X \in V$, let $\text{Att}(X)$ be a subset of $\text{Att}$ (composed by the attributes of $X$), and for each $a_i \in \text{Att}$, $i = 1, \ldots, n$, let $D_i$ be the domain of $a_i$. Let $r$ be a rule giving $X_0 \rightarrow X_1 \cdots X_k$, with $X_e \in N$, $X_f \in V$, $j = 1, \ldots, k$. We define a semantic rule associated to $r$ as an ordered pair $(f, A)$ where $f$ is a function, $f: D_{i_0} \times \cdots \times D_{i_p} \rightarrow D_{j_0}$, $i_0, \ldots, i_p \in \{1, \ldots, n\}$, $p > 0$; $A \in (\text{Att} \times \{e, 1, \ldots, k\})^{p+1}$, $A = ((a_{i_0}, j_0), \ldots, (a_{i_p}, j_p))$, $j_0, \ldots, j_p \in \{e, 1, \ldots, k\}$, such as for each $h = 0, \ldots, p$, $a_{i_h} \in \text{Att}(X_{i_h})$; and, if $p = 0$, then $f \in D_{j_0}$.

**Comment.** $D_i$ is the value domain of the attribute $a_i$. As we will see in Definition 4, the value of the attribute $a_{i_0}$ in the symbol index $j_0$ (in the syntactic rule) is computed by $f$ from the values of the remaining components of $A$: $(a_{i_0}, j_1), \ldots, (a_{i_p}, j_p)$.

**Example 3.** In this example we will define a grammar also used in Examples 4 and 5. This rather simple grammar has a certain practical interest, and shows clearly the concepts of attribute and semantic rules.

Let $G = (N, T, R, A)$ be the following type 2 grammar: $N = \{A, E\}$, $T = \{b, d, 0, 1\}$, $R = \{A \rightarrow bE, A \rightarrow dE, E \rightarrow E0, E \rightarrow E1, E \rightarrow e\}$. This grammar generates every string beginning with one ‘$b$’ or ‘$d$’ followed by a finite sequence of ‘$1$’s and ‘$0$’s. Let us define two attributes $a_1$ and $a_2$ for this grammar, with the following intuitive meaning: $a_1$ represents the ‘type’ of the expression, (‘$b$’ for binary numbers and ‘$d$’ for decimal numbers), and $a_2$ represents the value of the expression: $\text{Att} = \{a_1, a_2\}$, $\text{Att}(E) = \{a_1, a_2\}$, $\text{Att}(A) = \{a_1\}$, $\text{Att}(b) = \text{Att}(d) = \{a_1\}$, $\text{Att}(0) = \text{Att}(1) = \emptyset$, $D_1 = \{2, 10\}$, $D_2 = \mathbb{Z}^+$.

Let us define the following semantic rules to obtain the value of the whole number and store it in the value of $a_2$ in $A$:

(a) For the syntactic rule $A \rightarrow bE$ ($X_e = A$, $X_1 = b$, $X_2 = E$)
   (1) $(f, A)$ with $f \in D_1$, $f = 2$, and $A = (a_1, 2)$,
   (2) $(f, A)$ with $f: D_2 \rightarrow D_2$, $f(x) = x$, and $A = ((a_2, e), (a_1, 2))$.

The semantic rule (1) ($f = 2$) means that a binary expression follows (i.e., the value is obtained by means of the powers of (2)). The semantic rule (2) means that $a_2$ in $X_e = A$, is computed from $a_2$ in $X_2 = E$, using the identity function. In Definition 4 the exact way to compute attributes will be explained.

The following rules are also needed to obtain the value of the attributes $a_1$ and $a_2$. The whole grammar will be used in the remaining examples in this section. The meaning of the rules should be clear after the above explanations.

(b) $A \rightarrow dE$
   (3) $(f, A)$ with $f \in D_1$, $f = 10$, and $A = (a_1, 2)$,
   (4) $(f, A)$ with $f: D_2 \rightarrow D_2$, $f(x) = x$, and $A = ((a_2, e), (a_1, 2))$.

(c) $E \rightarrow E0$
   (5) $(f, A)$ with $f: D_1 \rightarrow D_1$, $f(x) = x$, and $A = ((a_1, 1), (a_1, e))$,
   (6) $(f, A)$ with $f: D_1 \times D_2 \rightarrow D_1$, $f(x, y) = xy + 0$, and $A = ((a_2, e), (a_1, 1), (a_1, 1))$.

(d) $E \rightarrow E1$
   (7) $(f, A)$ with $f: D_1 \rightarrow D_1$, $f(x) = x$, and $A = ((a_1, 1), (a_1, e))$,
   (8) $(f, A)$ with $f: D_1 \times D_2 \rightarrow D_1$, $f(x, y) = xy + 1$, and $A = ((a_2, e), (a_1, 1), (a_1, 2))$.

(e) $E \rightarrow e$
   (9) $(f, A)$ with $f \in D_2$, $f = 0$, and $A = ((a_2, e))$.

**Definition 3.** An attributed grammar $AG$ is a 6-uple $AG = (G, \text{Att}, \bar{X}, \bar{x}, \bar{\gamma})$ where:

$G = (N, T, R, S)$ is a type 2 grammar $V = N \cup T$,

$\text{Att}$ is a finite set, composed by the attributes of the elements of $V$,

$\bar{X} = \{D_a | a \in \text{Att}\}$ (Attribute domains),
\( Y = \{ \text{Att}(X) \mid X \in V, \text{Att}(X) \subseteq \text{Att} \} \) and \( \text{Att} = \bigcup \{ \text{Att}(X) \mid X \in V \} \),
\( \bar{X} = \{ H(X), S(X) \mid X \in V, H(X) \cap S(X) = \emptyset \text{ and } H(X) \cup S(X) = \text{Att}(X) \} \) (\( H(X) \) is the set of inherited attributes of \( X \), and \( S(X) \) is the set of synthesized attributes of \( X \)),
\( \bar{\gamma} = \{ \bar{\gamma}_r \mid r \text{ rule in } G \} \) (the set of semantic rules), such that, if \( X_r \rightarrow X_1 \cdots X_k \) is a syntactical rule in \( G \), then the two following propositions hold:

1. For each \( a \in S(X) \), there exists precisely one semantic rule \( (f, A) \in \bar{\gamma}_r \), \( f : D_{i_1} \times \cdots \times D_{i_p} \rightarrow D_{i_0} \), \( A = ((a_{i_0}, j_{i_0}), \ldots, (a_{i_p}, j_{i_p})) \), with \( a_{i_0} = a \) and \( j_{i_0} = e \).
2. For each \( a' \in H(X_i), i = 1, \ldots, k \), there exists precisely one semantic rule \( (f, A) \in \bar{\gamma}_r \), \( f : D_{i_1} \times \cdots \times D_{i_p} \rightarrow D_{i_0} \), \( A = ((a_{i_0}, j_{i_0}), \ldots, (a_{i_p}, j_{i_p})) \), with \( a_{i_0} = a' \) and \( j_{i_0} = i \).

Comment. The elements of \( S(X) \) are called synthesized attributes of \( X \) because they are evaluated in every derivation tree from some attribute values of its children. Analogously, the elements of \( H(X) \) are called inherited attributes of \( X \) because they are evaluated from the attribute values of its parent and brothers.

Example 4. Let us consider the grammar \( G \) of Example 3. By suitably defining the sets \( \text{Att}, \bar{X}, \bar{\gamma}, \bar{\bar{X}}, \) and \( \bar{\gamma} \) we will obtain an attribute grammar:
\( \text{Att} = \{ a_1, a_2 \}, \bar{X} = \{ D_1, D_2 \}, D_1 = \{ 2, 10 \}, D_2 = \mathbb{Z}^+ \),
\( \bar{\gamma} = \{ \text{Att}(A), \text{Att}(E), \text{Att}(b), \text{Att}(d), \text{Att}(0), \text{Att}(1) \}, \)
\( \bar{\bar{X}} = \{ H(A), S(A), H(E), S(E), \ldots, H(1), S(1) \}, \)
\( S(A) = \{ a_2 \}, H(E) = \{ a_1 \}, S(E) = \{ a_2 \} \), the remaining elements are the empty set.
\( \bar{\gamma} \) is the set of semantic rules defined in Example 3: (1), (2), (3), \ldots, (9).
Notice that the existence conditions are satisfied because there is one semantic rule for each synthesized attribute \( a \in S(X_i) \) of the antecedent, and one semantic rule for each inherited attribute \( a \in H(X_i) \) of the consequents. Thus, this is an example of an attributed grammar.

We will now define the concept of satisfaction of a semantic rule and the computing procedure for the attribute values.

Definition 4. Let \( AG = (G, \text{Att}, \bar{X}, \bar{\gamma}, \bar{\bar{X}}, \bar{\gamma}) \) be an attributed grammar and \( (D, \epsilon) \) the derivation tree of a sentence with respect to the underlying grammar \( G \). For each \( a \in \text{Att} \), let \( D^{(a)} = \{ k \in D \mid a \in \text{Att}(\epsilon(k)) \} \). For each \( a \in \text{Att} \), let \( \text{val}_a : D^{(a)} \rightarrow D_a \), (labelling of tree \( D \)). Let \( X_r \rightarrow X_1 \cdots X_k \) be the rule \( r \) of \( G \) used at node \( q \). It is said that the semantic rule \( (f, A) \in \bar{\gamma}_r \), \( f : D_{i_1} \times \cdots \times D_{i_p} \rightarrow D_{i_0} \), \( A = ((a_{i_0}, j_{i_0}), \ldots, (a_{i_p}, j_{i_p})) \), is satisfied if
\( \text{val}_a(q_{i_0}) = f(\text{val}_a(q_{i_1}), \ldots, \text{val}_a(q_{i_p})). \)

Comment. The set \( D^{(a)} \) is the set of nodes of \( D \) such that their image, by the labelling mapping \( \epsilon \), is a symbol of the grammar having \( a \) in its attribute set. The \( \text{val}_a \) mapping assigns to every node in \( D^{(a)} \) a value of \( D_a \) (the domain of the attribute \( a \)).

The \( \text{val}_a \) mappings are partial labellings of the tree domain \( D \). We can say that the attributes are associated with symbols of the grammar, but attribute values are associated with nodes of the tree domain \( D \). So, the same symbol appearing in several nodes in the tree by the labelling mapping \( \epsilon \) will have different attribute values. From a certain point of view, we say that a semantic rule is satisfied when the attribute values associated with the nodes are consistent with the evaluation of the function \( f \) (semantic validation of the construction). From another point of view, we can say that the satisfaction of a semantic rule is a procedure to compute an attribute value (assignment of meaning).
Example 5. Let us consider the attributed grammar developed in Examples 3 and 4. Let \((D, e)\) be the derivation tree of the sentence \(b110\):

\[
\begin{array}{l}
\varepsilon A \\
\quad [1]b \\
\quad \quad [2]E \\
\quad \quad \quad [2, 1]E \\
\quad \quad \quad \quad [2, 1, 1]E \\
\quad \quad \quad \quad \quad [2, 1, 1, 1]E \\
\quad \quad \quad \quad \quad \quad [2, 1, 1, 1, 1]E \\
\quad \quad \quad \quad \quad \quad \quad [2, 1, 1, 1, 1, 1]E \\
\quad \quad \quad \quad \quad \quad \quad \quad [2, 1, 1, 1, 1, 1, 1]E \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad [2, 1, 1, 1, 1, 1, 1, 1]E \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \varepsilon \\
\end{array}
\]

According to the above definitions, the sets \(D(a)\) and the mappings \(\text{val}_a\) are as follows:

\[
D^{(a_1)} = \{[2], [2, 1], [2, 1, 1], [2, 1, 1, 1]\}; \quad D^{(a_2)} = \{e, [2], [2, 1], [2, 1, 1], [2, 1, 1, 1]\}.
\]

\[
\text{val}_{a_1}: D^{(a_1)} \rightarrow D_1; \quad \text{val}_{a_2}: D^{(a_2)} \rightarrow D_2.
\]

To compute attribute values, we must evaluate \(\text{val}_a\) mappings in the nodes of the tree. We first compute the values of \(a_1\), i.e., evaluate the mapping \(\text{val}_{a_1}\).

For the rule \(A \rightarrow bE\) used at the node labelled with \(e\), semantic rule (1) is applicable (Example 3):

\[
\text{val}_{a_1}(e[2]) = f, \quad \text{hence} \quad \text{val}_{a_1}([2]) = 2.
\]

For the rule \(E \rightarrow E0\) used at node \([2]\), semantic rule (5) is applicable:

\[
\text{val}_{a_1}([2]1) = \text{val}_{a_1}([2]e), \quad \text{hence} \quad \text{val}_{a_1}([2, 1]) = \text{val}_{a_1}([2]) = 2.
\]

Two subsequent applications of semantic rule (7) finally yield

\[
\text{val}_{a_1}([2, 1, 1]) = \text{val}_{a_1}([2, 1, 1, 1]) = 2.
\]

So \(a_1\) is really an inherited attribute. Now we compute the values of the attribute \(a_2\), and the computation process will show that \(a_2\) is a synthesized attribute.

For the rule \(E \rightarrow e\) we apply semantic rule (9) and obtain \(\text{val}_{a_2}([2, 1, 1]) = 0\). Now, for the rule \(E \rightarrow E1\) used at node \([2, 1, 1]\), we apply semantic rule (8), obtaining

\[
\text{val}_{a_2}([2, 1, 1]e) = \text{val}_{a_2}([2, 1, 1]1) \text{val}_{a_2}([2, 1, 1]1) + 1,
\]

hence

\[
\text{val}_{a_2}([2, 1, 1]) = \text{val}_{a_2}([2, 1, 1]) \text{val}_{a_2}([2, 1, 1]) + 1 = 2 0 + 1 = 1.
\]

Repeating this process at node \([2, 1]\) we obtain \(\text{val}_{a_2}([2, 1]) = 2 1 + 1 = 3\) and, for the rule \(E \rightarrow E0\) used at node \([2]\), we obtain \(\text{val}_{a_2}([2]) = 2 3 + 0 = 6\).

Finally, for the rule \(A \rightarrow bE\) used at node \(e\), we obtain by semantic rule (2)

\[
\text{val}_{a_1}(e) = \text{val}_{a_1}([2]) = 6
\]

which is the value of the binary number 110.
Definition 5. Let \( AG = (G, \text{Att}, \mathcal{D}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}) \) be an attributed grammar and \( (D, e) \) the derivation tree of a sentence with respect to the underlying grammar \( G \). A labelling family \( \{ \text{val}_a : D^{(a)} \rightarrow D_a \mid a \in \text{Att} \} \) of tree \( D \) is an attribute annotation (ata) with respect to AG.

Comment. Example 5 shows clearly that the labelling family \( \{ \text{val}_a, \text{val}_b \} \) of the tree \( D \) is an ata with respect to the attributed grammar developed in Examples 3 and 4.

Definition 6. Let \( AG = (G, \text{Att}, \mathcal{D}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}) \) be an attributed grammar and \( (D, e) \) the derivation tree of a sentence with respect to the underlying grammar \( G \). An ata for the tree \( D \) is consistent with respect to AG iff every semantic rule is satisfied.

Comment. The evaluation of semantic rules, (computation of \( \text{val}_a \) labellings), can be seen as an equation system between the attribute values at the nodes of a tree \( D \). From this point of view, a consistent ata is a solution of this equation system, so validating these instances of the applied semantic rules.

Example 5 shows that this ata is consistent for this tree, since every semantic rule is satisfied.

Note. If \( a \) is a fuzzy attribute, the domain \( D_a \) is a fuzzy domain, and for a node \( v \), \( \text{val}_a(v) = C \in D_a \), that is, \( C \) is a fuzzy set. In this case we write \( \mu_C(b) = \alpha \in W \), or \( \mu_{\text{val}_a(v)}(b) = \alpha \).

3. The L language

3.1. Introduction

In this paragraph, we introduce a new programming language with fuzzy statements. The output of a program written in this language will be a fuzzy set. The language, called L, is very simple, and could be the kernel for more application-oriented fuzzy programming languages.

The theoretical inspiration for L language comes from the ideas of [7]. In that paper, Clares extends the concept of crisp recursive function to the one of \( W \)-recursive function, keeping crisp the zero and successor functions, and fuzzifying the projection functions. More formally, let \( \Psi = \{ \psi : \mathbb{Z}^n \times \mathbb{Z}^m \rightarrow W, \psi_{W \text{-computable}} \} \). Then, the fuzzy projections are

\[
\rho_{\psi}(\langle u_1, \ldots, u_n \rangle, v) = \psi(u_1, v), \quad v, u_1, \ldots, u_n \in \mathbb{Z}^+, \quad \psi \in \Psi,
\]

and the operations of composition, primitive recursion and minimalization are suitably extended to apply to \( W \)-valued functions.

Syntactical and semantical definitions of L language are based upon this idea. The only fuzzy statement is the assignation statement. Generally speaking, the output of the execution of an assignment will be a fuzzy set. Fuzzification will propagate from this point of the program to the remaining statements. Thus, the execution is nondeterministic and a degree is associated with every possible set of values.

An L program is a 3-uple \( (n, m, P) \), where \( n \geq 0 \) is the number of input variables, \( m \geq 1 \) is the number of output variables, and \( P \) is the program code.

Input variables will be enumerated \( X_1, \ldots, X_n \), and output variables will be enumerated \( X_1, \ldots, X_m \). Notice that input and output variables can be partially or totally the same set. Then, the program computes a \( W \)-function \( f: (\mathbb{Z}^+)^n \times (\mathbb{Z}^+)^m \rightarrow W \).

Of course, an L program can use other variables. A configuration (or memory configuration) is a specification of a value for every variable of the program.

When the number of input variables \( n \) and the number of output variables \( m \) are deduced clearly in the context, we will identify a program with its code.
Example 6. We will consider the following problem: Given two values, increment the former by as many units as indicated by a fuzzy value around the latter.

An L program for this problem would be the 3-uple $(2, 1, P)$. According to this 3-uple, there are two input variables, namely $X_1$ and $X_2$, and one output variable, namely $X_1$. $P$ is the following L code:

\[
\text{INCREASING;}
\]
\[
\text{begin}
\]
\[
X_3 := \text{FUZZ1}(X_2);
\]
\[
\text{REPEAT X3 TIMES}
\]
\[
X_1 := X_1 + 1
\]
\[
\text{SEMIT}
\]
\[
\text{end}
\]

where the unary function FUZZ1 is associated with \( f_1: \mathbb{Z}^+ \times \mathbb{Z}^+ \rightarrow W \) defined by

\[
f_1(X, Y) = \begin{cases}
1 & \text{if } y = x, \\
0.5 & \text{if } (y = x - 1) \text{ or } (y = x + 1), \\
0 & \text{otherwise.}
\end{cases}
\]

The role of $X$ is played in the program code by $X_2$; the role of $Y$ is played by $X_3$; and the degree does not appear in the program code. This degree is associated with the memory configuration resulting from the execution of the statement.

The meaning of the above program is determined in the following way: The initial configuration is \( C = 1 \mid (a, b, 0) \). The first statement is executed and, applying $f_1$ in the way just explained, since $X_2 = b$, three values are assigned to $X_3$ with different degrees, so obtaining the configuration

\[
0.5 \mid (a, b, b - 1) + 1 \mid (a, b, b) + 0.5 \mid (a, b, b + 1).
\]

Immediately after, the loop is executed for each element in the configuration, finally obtaining the configuration

\[
0.5 \mid (a + b - 1, b, b - 1) + 1 \mid (a + b, b, b) + 0.5 \mid (a + b + 1, b, b + 1).
\]

Thus, the final result is the projection of the last configuration onto the first component. This projection is the fuzzy subset of \( \mathbb{Z}^+ \)

\[
0.5 \mid a + b - 1 + 1 \mid a + b + 0.5 \mid a + b + 1.
\]

Hence, this program computes the following \( W \)-function: \( f: (\mathbb{Z}^+)^2 \times \mathbb{Z}^+ \rightarrow W \)

\[
f((x, y), z) = \begin{cases}
1 & \text{if } z = x + y, \\
0.5 & \text{if } (z = (x + y) - 1) \text{ or } (z = (x + y) + 1), \\
0 & \text{otherwise.}
\end{cases}
\]

Should more fuzzy assignments appear in the program, associated degrees would be obtained by means of a 'multiplicative' process, according to the \( \otimes \) operation in the semiring $W$. To sum up, an L program represents the set of all computations that the program carried out. The set of all pairs (initial value, final value) (fuzzy or crisp) determined by these computations is the input/output relation computed by the program. This is a $W$-functional relationship. That is the reason why input/output statements are not needed in the formal development of the language. However, simple \texttt{read}(X_1, \ldots, X_n) and \texttt{write}(X_1, \ldots, X_n) statements have been implemented for practical reasons.

In the following paragraph, we will develop a sublanguage $L_0$ of L without loop statements. So, assignment and fuzzy assignment statements are studied now; loop statements are studied in Section 3.3; and conditional statements are introduced in Section 3.4 by means of macro constructions. The goal of this structuring of the topics is to achieve a more understandable exposition.
3.2. The $L_0$ language

The following attributed grammar $AG_0$ defines the $L_0$ language.

Let $AG_0 = (G_0, Att, \bar{\Sigma}, \bar{\tau}, \bar{\lambda}_0)$ be a 6-uple where:

(A) $G_0 = (N_0, T_0, R_0, S_0)$ is the following context free grammar:

$N_0 = \{ \text{letter}, \text{no-zero}, \text{ident}, \text{digit}, \text{number}, \text{variable}, \text{assignment}, \text{statement}, \text{sequence}, \text{F}, \text{code} \}$

$T_0 = \{X, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, :=, \text{fuzz}, (;), \text{suc}, ;, \text{begin}, \text{end} \}$

$R_0$ is the set of the following syntactic rules (in EBNF notation):

- $\langle \text{letter} \rangle ::= A | B | \ldots | Z$
- $\langle \text{ident} \rangle ::= \langle \text{letter} \rangle (\langle \text{ident} \rangle \langle \text{letter} \rangle)$
- $\langle \text{no-zero} \rangle ::= 1 | \ldots | 9$
- $\langle \text{digit} \rangle ::= \langle \text{no-zero} \rangle \langle \text{digit} \rangle$
- $\langle \text{number} \rangle ::= \langle \text{no-zero} \rangle \langle \text{number} \rangle \langle \text{digit} \rangle$
- $\langle \text{variable} \rangle ::= \langle \text{number} \rangle \langle \text{variable} \rangle \langle \text{assignment} \rangle$
- $\langle \text{assignment} \rangle ::= \langle \text{variable} \rangle = 0 | \langle \text{variable} \rangle = \langle \text{F} \rangle (\langle \text{variable} \rangle)$
- $\langle \text{F} \rangle ::= \text{fuzz} \langle \text{number} \rangle$
- $\langle \text{statement} \rangle ::= \langle \text{assignment} \rangle$
- $\langle \text{sequence} \rangle ::= \langle \text{statement} \rangle | (\langle \text{sequence} \rangle ; \langle \text{statement} \rangle)$
- $\langle \text{code} \rangle ::= \langle \text{ident} \rangle ; \text{begin} \text{end} | \langle \text{ident} \rangle ; \text{begin} \langle \text{sequence} \rangle \text{end}$
- $S ::= \langle \text{number} \rangle \langle \text{number} \rangle \langle \text{code} \rangle$

(B) $Att = \{ \text{max}, \nu, \text{ind}, \text{coin}, \text{cofi} \}$ is the set of attributes, which have domains $D_{\text{max}} = D_\nu = D_{\text{ind}} = Z^+$, and $D_{\text{coin}} = D_{\text{cofi}} = D$, where $D = \bigcup_{k>0} D_k$, $D_k$ being the set of $W$-fuzzy sets on $(Z^+)^*$, $k \neq 0$.

The intuitive meaning of every attribute in the grammar is the following:

- $\text{max}$ is the maximum number of variables used in a program,
- $\nu$ is a number value,
- $\text{ind}$ is an index for a variable or a fuzzy function,
- $\text{coin}$ is a memory configuration before the execution of a statement,
- $\text{cofi}$ is a memory configuration after the execution of a statement.

max, $\nu$ and ind have crisp domains; coin and cofi have fuzzy domains.

(C) $\bar{\Sigma} = \{ Z^+, D \}$.

(D) The elements of $X$ are the sets of attributes for each symbol of the grammar:
- $Att(\langle \text{no-zero} \rangle) = Att(\langle \text{digit} \rangle) = Att(\langle \text{number} \rangle) = \{ \nu \}$,
- $Att(\langle \text{variable} \rangle) = Att(\langle \text{F} \rangle) = \{ \text{ind} \}$,
- $Att(\langle \text{assignment} \rangle) = Att(\langle \text{statement} \rangle) = Att(\langle \text{sequence} \rangle) = Att(\langle \text{code} \rangle) = \{ \text{max}, \text{coin}, \text{cofi} \}$.

For any other symbol $s$, $Att(s) = \emptyset$.

(E) The elements of $\bar{\tau}$ are the sets of synthesized and inherited attributes for each symbol of the grammar:
- $S(\langle \text{no-zero} \rangle) = S(\langle \text{digit} \rangle) = S(\langle \text{number} \rangle) = \{ \nu \}$,
- $S(\langle \text{variable} \rangle) = S(\langle \text{F} \rangle) = \{ \text{ind} \}$,
- $H(\langle \text{assignment} \rangle) = H(\langle \text{statement} \rangle) = H(\langle \text{sequence} \rangle) = \{ \text{coin} \}$,
- $S(\langle \text{code} \rangle) = \{ \text{max}, \text{coin}, \text{cofi} \}$,
- $S(\langle \text{assignment} \rangle) = S(\langle \text{statement} \rangle) = S(\langle \text{sequence} \rangle) = \{ \text{max}, \text{cofi} \}$,

For any other symbol $s$, $S(s) = H(s) = \emptyset$.

(F) The elements of $\bar{\lambda}_0$ are the sets of semantic rules assigned to each syntactic rule of the grammar.

We write only the relevant semantic rules below the matching syntactic rule:

- $(\text{assignment}) \rightarrow \langle \text{variable} \rangle; = 0$.

Knowing the attributes coin and ind, the semantic rule that evaluates the attribute cofi is the following: $A = ((\text{cofi}, \epsilon), (\text{coin}, \epsilon), (\text{ind}, 1)), f : D \times Z^+ \rightarrow D, f(A, i) = B$, if the following holds: For each $a$, with $\mu_\alpha(a) = \alpha, \exists b$, with $\mu_\beta(b) = \beta$ such that $(\pi_j(b) = 0) \wedge (\forall j \neq i \pi_j(b) = \pi_j(a)) \wedge (\beta = \alpha)$.

- $(\text{assignment}) \rightarrow \langle \text{variable} \rangle; = \text{suc}(\langle \text{variable} \rangle)$.
A = ((cofi, e), (coin, e), (ind, 1), (ind, 3), (ind, 5)), f:D × Z+ × Z* → D, f(A, i, j, k) = B, if the following holds: For each a, μ_A(a) = α, ∃b, with μ_B(b) = β such that (π_i(b) = π_i(a) + 1) ∧ (∀k ≠ i π_k(b) = π_k(a)) ∧ (β = α).

\[ \langle \text{assignment} \rangle \rightarrow \langle \text{variable} \rangle : = \langle F \rangle(\langle \text{variable} \rangle) \]

A = ((cofi, e), (coin, e), (ind, 1), (ind, 3), (ind, 5)) f:D × Z+ × Z+ × Z* → D, f(A, i, j, k) = B, if the following holds: For each a, with μ_A(a) = α ∃B_a = Σ_A β_a/b_a, such that (π_i(b_a) = d_a) ∧ (∀h ≠ i π_h(b_a) = π_h(a)) ∧ (β_a = α ⊗ f_i(π_i(a), d_a)) and B = \bigcup_a B_a, where f_i:Z+ × Z* → W is an external W-computable function in [7] sense.

Example 7. Let us apply the latter semantic rule to the assignment statement in Example 6

\[ X3 := \text{FUZZI}(X2); \]

the initial configuration (coin) of root node is A = 1/(a, b, 0), and the remaining attributes are

(\text{ind, 1}) = 3, \quad (\text{ind, 3}) = 1, \quad (\text{ind, 5}) = 2.

For (a, b, 0), μ_A(a, b, 0) = 1, and then:

\[ \begin{align*}
\pi_3(b_1) &= b - 1, \quad \beta_1 = 1 \otimes f_i(b, b - 1) = 1 \otimes 0.5 = 0.5, \\
\pi_3(b_1) &= b, \quad \beta_2 = 1 \otimes f_i(b, b) = 1 \otimes 1 = 1, \\
\pi_3(b_1) &= b + 1, \quad \beta_4 = 1 \otimes f_i(b, b + 1) = 1 \otimes 0.5 = 0.5.
\end{align*} \]

Therefore, the value of the final configuration (cofi) attribute is at this node:

\[ B = 0.5/(a, b, b - 1) + 1/(a, b, b) + 0.5/(a, b, b + 1). \]

The proof of the following result can be found in [16]

Theorem 1. The 6-uple AG_0 = (G_0, Att, D, F, X, X_0) of the former section is an attribute grammar.

Obviously, for each derivation tree there is a unique annotation of attribute values (ata), and therefore this ata is consistent with the semantic rules.

Definition 7. Let AG_0 be the former attribute grammar and G_0 the underlying context free grammar. We define L_0 = L(G_0). A program P in L_0 is an element of L_0. If P is a program in L_0, then the derivation tree (D, e) such that frontier ((D, e)) = P is the derivation tree of P.

Definition 8. We will say that P is a k-program in L_0 iff:

(a) P is a program in L_0 with the derivation tree (D, e),
(b) The value of max for the root node of D is k.

Informally, P is a k-program if it uses k variables.

Definition 9. Let P be a k-program in L_0 and (D, e) its derivation tree. We define the function Ω_p:(Z*)^k × (Z*)^k → W as Ω_p(a, b) = β iff there is a consistent ata for (D, e) with

\[ \text{val}_{\text{coin}}(e) = 1/a \quad \text{and} \quad \mu_{\text{val}_{\text{coin}}}(b) = β. \]

Otherwise, Ω_p(a, b) is not defined.

The former expression states that the value of coin – initial configuration – at the root node is a crisp set
with only one element \( a \), and that the value of \( \Omega \) – final configuration – at the root node is a \( W \)-set which contains the element \( b \) with degree \( \beta \). Therefore, the function \( \Omega \) gives the degree associated to the output \( b \) when the input is the crisp value \( a \).

Example 8. Let us consider a program that computes a fuzzy successor. The program is \( P = (2, 1, Q) \), where \( Q \) is:

\[
\begin{align*}
\text{FOLLOW:} & \\
\text{begin} & \\
X2 & := \text{suc}(X1); \\
X1 & := \text{FUZZ1}(X2); \\
\text{end}
\end{align*}
\]

and the \( W \)-function associated with \( \text{FUZZ1} \) is

\[
f_1(x, y) =
\begin{cases}
1 & \text{if } y = x, \\
0.75 & \text{if } (y = x - 1) \text{ or } (y = x + 1), \\
0.5 & \text{if } (y = x - 2) \text{ or } (y = x + 2), \\
0.25 & \text{if } (y = x - 3) \text{ or } (y = x + 3), \\
0 & \text{otherwise.}
\end{cases}
\]

Then the associated \( \Omega_P \) function is \( \Omega_P : (Z^+)^2 \times (Z^+)^2 \to W \)

\[
\Omega_P((a, b), (c, d)) =
\begin{cases}
1 & \text{if } (d = a + 1) \text{ and } (c = a + 1), \\
0.75 & \text{if } (d = a + 1) \text{ and } (c = a \text{ or } c = a + 2), \\
0.5 & \text{if } (d = a + 1) \text{ and } (c = a - 1 \text{ or } c = a + 3), \\
0.25 & \text{if } (d = a + 1) \text{ and } (c = a - 2 \text{ or } c = a + 4), \\
0 & \text{otherwise.}
\end{cases}
\]

3.3. The \( L \) language

In this paragraph the \( L_0 \) language is extended with a define loop statement.

Definition 10. Let \( G_0 = (N_0, T_0, R_0, S_0) \) be the former context free grammar. We define \( G = (N, T, R, S) \) from \( G_0 \) in the following way:

\[
\begin{align*}
N & = N_0, \\
T & = T_0 \cup \{\text{repeat, times, semit}\}, \\
R & = R_0 \cup \{\text{statement} \to \text{repeat(\text{variable})times(\text{sequence})semit}\}, \\
S & = S_0.
\end{align*}
\]

Obviously \( G \) is a context free grammar. We call the new syntactic rule ‘definite loop rule’.

Definition 11. \( L \) is the language generated by \( G \): \( L = L(G) \).

Analogously to \( L_0 \), a program in \( L \) is a string \( P, P \in L \), and its derivation tree is a pair \( (D, e) \) such that the frontier is \( P \).
From another point of view, the code of a program \( P \) in \( L \) with derivation tree \((D, e)\) is a string 
`identifier; begin \( I_1I_2 \cdots I_n \) end`, where \( I_j \) are assigment or definite loop statements; \( I_j \) is an assigment statement if \( I_j = \) frontier(subtree((\(D, e\), \(v\))), \(v \in D\) being a node where the rule (statement) \( \rightarrow \) (assignment) is applied; and \( I_j \) is a definite loop statement if \( I_j = \) frontier(subtree((\(D, e\), \(v\))), \(v \in D\) being a node where the definite loop rule is applied.

Now we refine our statement classification and introduce a classification of \( L \) programs.

**Definition 12.** Let \( P \) be an \( L \)-program and \((D, e)\) its derivation tree. We say that \( P \) includes a statement of type \( i \geq 1 \) if there are nodes \( v, v_1, \ldots, v_{i-1} \in D \) where the loop rule is applied and \( \forall u > vv_1 \cdots v_{i-1} \in D \) in \( u \) the loop rule is not applied. A type 0 statement is an assignment.

**Comment.** Informally speaking, a type 0 statement has no nesting; and a type \( i \) statement is built by nesting \( i \) definite loops.

**Definition 13.** We say that an \( L \)-program \( P \) is of type \( n \) iff \( P \) includes at least one statement of type \( n \) and the remaining statements are of a type less than or equal to \( n \).

Starting from the attributed grammar \( A_G_0 \) and adding the semantic rules for the definite loop statement, we obtain an attributed grammar \( A_G \) for \( L \).

Let \( A_G = (G, \text{Att}, \mathcal{D}, \mathcal{X}, \Sigma) \) be a 6-uple where \( G \) is as defined in Definition 10, \( \text{Att}, \mathcal{D}, \mathcal{X} \) are defined in Section 3.2, and \( \Sigma = \Sigma_0 \cup \Sigma_b \) where \( \Sigma_b \) is the set of semantics rules for the syntactic definite loop rule:

\[(\text{statement}) \rightarrow \text{repeat}(\text{variable})\text{times}(\text{sequence})\text{semit}.\]

The semantic rule that evaluates the attribute \( \text{cof} \) is \( A = ((\text{cof}, e), (\text{coin}, e), (\text{ind}, 2)), f: D \times Z^+ \rightarrow D \), define as follows: For \( C \in D \) and \( r \in Z^+ \) we compute \( f(C, r) \) in the following way:

(A) Type 1. Let \((D, e)\) be the derivation tree of a type 1 \( L \)-program. Let \( v \in D \) be a node where the loop rule is applied. Let \((D', e')\) be the following tree: \( D' = \{e, [1], [2], [3], [4], [5]\}, e'(e) = \) (code), \( e'(1) = \) (ident), \( e'(2) = ; \), \( e'(3) = \) begin, \( e'(4) = \) (sequence), \( e'(5) = \) end \((D_1, e_1) = \text{subst}((D', e'), [4], \text{subtree}((D, e), v4)). (D_1, e_1) \) is a derivation tree (of a type 0 \( L \)-program). Let \( v_1 = b \) be the attribute valuation functions for the derivation tree \((D_1, e_1)\). For each \( b, \mu_c(b) = \alpha \), and \( \pi_c(b) = m \).

\[
\text{val}_1^{\text{conf}}(e) = \alpha/b \quad \text{yields, in a loop execution,} \quad \text{val}_1^{\text{conf}}(e) = B_1^b;
\]
\[
\text{val}_1^{\text{conf}}(e) = B_1^b \quad \text{yields, in a loop execution,} \quad \text{val}_1^{\text{conf}}(e) = B_2^b;
\]
\[
\vdots \quad (m \text{ steps})
\]
\[
\text{val}_1^{\text{conf}}(e) = B_{m-1}^b \quad \text{yields, in a loop execution,} \quad \text{val}_1^{\text{conf}}(e) = B^b.
\]

Then \( f(C, r) = \bigcup_{\mu_c(b) \neq 0} B^b = \text{val}_1^{\text{conf}}(v) \) in the derivation tree \((D, e)\).

(B) Type \( n > 1 \).

The above process is recursively extended.

The proof of the following result can also be found in [16].

**Theorem 2.** The former structure \( A_G = (G, \text{Att}, \mathcal{D}, \mathcal{X}, \Sigma) \) is an attribute grammar.

As in Section 3.2, for each derivation tree there is a unique annotation of attribute values (ata), and therefore this ata is consistent with the semantic rules; and the \( \Omega_p \) function is defined as in Definition 9.

**Example 9.** Let us again consider the \( L \) program in Example 6. In Example 7 the value of \( \text{cof} \) for the fuzzy assignment statement \( X3 := \text{FUZZI}(X2) \) has been computed. This value is
A = 0.5/(a, b, b - 1) + 1/(a, b, b) + 0.5/(a, b, b + 1). Due to semantic rules, this configuration is the value of the coin attribute for the loop statement

REPEAT X3 TIMES X1 := X1 + 1 SEMIT.

Then the value of the cofi attribute for the loop statement is computed in the following way:

The element 0.5/(a, b, b - 1) yields in one step the element 0.5/(a + 1, b, b - 1), the element 0.5/(a + 1, b, b - 1) yields in one step the element 0.5/(a + 2, b, b - 1),
\[ (b - 1 \text{ times}) \]
finally, 0.5/(a + b - 2, b, b - 1) yields in one step 0.5/(a + b - 1, b, b - 1).

In the same way

The element 1/(a, b, b) yields in b steps 1/(a + b, b, b),
the element 0.5/(a, b, b + 1) yields in b + 1 steps 0.5/(a + b + 1, b, b + 1),
and hence the value of cofi is
\[ 0.5/(a + b - 1, b, b - 1) + 1/(a + b, b, b) + 0.5/(a + b + 1, b, b + 1). \]

3.4. L-Programmable functions

**Definition 14.** Let \( P = (j, m, Q) \) be a \( k \)-program in \( L \), \( j, m \leq k \). The semantic fuzzy function \( f_p \) for \( P \), \( f_p : (Z^+)^j \times (Z^+)^m \to W \) is defined as follows: Let \( 1/(a_1, \ldots, a_j, 0, \ldots, 0) = 1/a = \text{val}_{\text{coin}}(\varepsilon) \). Then \( f_p(a_1, \ldots, a_j, b_1, \ldots, b_m) = \sum \Omega_p(a, a') \), where \( \sum \) denotes the \( \oplus \) operation extended to all \( a' \), such that its \( m \) first components match \( b_1, \ldots, b_m \).

Informally, if a program has less input variables than used variables, then
(A) Input variables will be the first ones, and
(B) remaining variables are implicitly initialized to 0.

On the other hand, if a program has less output variables than used variables, then
(C) the final configuration is projected on the first variables, accumulating the degrees additively by means of the \( \oplus \) operation of the semiring.

**Definition 15.** A \( W \)-function \( \phi : (Z^+)^j \times (Z^+)^r \to W \) is ‘L-programmable’ if \( \phi = f_p \) for a certain \( k \)-program \( P = (r, s, Q) \).

The following examples are needed in subsequent paragraphs.

**Example 10.** Program previous \( P = (1, 1, Q) \).

```
PREV;
begin
X2:= X1;
repeat X2 times
  X1:= FUZZI(X3);
  X3:= suc(X3)
semit
end.
```

This program computes, for a natural number \( n \), the \( W \)-set that can be considered previous to \( n \). The
concept of previous is context dependent, and this fact is reflected in the choice of the $W$-function for FUZZ1.

**Example 11.** Program $add\ P = (2, 1, Q)$.

```
ADD;
begin
    repeat $X1$ times
        $X3$ := suc($X3$)
        semit;
    repeat $X2$ times
        $X3$ := suc($X3$)
        semit;
    $X1$ := $X3$
end
```

This program computes the sum of the numbers stored in $X1$ and $X2$.

**Example 12.** Program $sign\ P = (1, 1, Q)$.

```
SG;
begin
    $X2$ := $X1$;
    repeat $X2$ times
        $X1$ := 0;
        $X1$ := suc($X1$)
    semit
end
```

This program computes the crisp function sign.

3.5. **Macros in L**

The L language has a very reduced set of statements. L programs quickly become too long and illegible. To avoid this problem, the *macro* concept is introduced. Thus, in this paragraph, the language L is extended to allow the inclusion of whole programs in another program by means of a single statement. The introduction of this concept does not enlarge the power of the language, but only the legibility. This is the reason why macro statements are not formally introduced as an extension of the syntax and semantics of L language.

**Definition 16.** A macro-statement has the following form:

$$Xi := \langle ident \rangle (Xi_1, \ldots, Xi_n)$$

where $\langle ident \rangle$ is the identifier associated with a program in L: $R = (s, 1, T)$. We write for programs in the Examples 10, 11, and 12:

- $Xi := \text{prev}(Xi_1)$,
- $Xi := \text{add}(Xi_1, Xi_2)$,
- $Xi := \text{sg}(Xi_1)$.

**Definition 17.** A macro-program is a program where each statement is either a statement of the L language or a macro-statement.
Definition 18. Let $P = (j, m, Q)$ be a macro-program. Let us consider that there is in $Q$ the macro-statement $X_i := \text{word}(X_{i_1}, \ldots, X_{i_s})$, where \text{word} is the identifier associated with $R = (s, 1, T)$. In this situation, the macro-program $P$ is transformed into an L program $P' = (j, m, Q')$ as follows:

- let $q$ be the number of variables used in $P$;
- let $h$ be the number of variables used in $R$.

Then replace the macro-statement with the following sequence of statements:

- $X_q + 1 := X_{i_1}; \ldots; X_q + s := X_{i_s};$
- ‘The sequence of statements of $T$ by substituting $X_q + i$ for $X_i$, $1 \leq i \leq h$’;
- $X_i := X_q + 1$.

If there are more macro statements, they are expanded one after another.

Example 13. The following macro-program $P = (2, 1, Q)$, called \textit{decr}, decreases $X_1$ in a fuzzy way as many units as indicated by $X_2$. The macro ‘prev’ is used in it.

```
DECR;
begin
repeat $X_2$ times
  $X_1 := \text{prev}(X_1)$
semit
end
```

In this case we write: $X_i := \text{decr}(X_{i_1}, X_{i_2})$.

If the statement $X_1 := \text{fuzzl}(X_3)$ is substituted by $X_1 := X_3$ in Example 10, a program is obtained that computes the crisp predecessor, whose identifier will be \textit{pred}. If the macro \textit{prev} is replaced with \textit{pred} in Example 13, the new program computes the crisp substraction, whose identifier will be \textit{substract}.

To show the expressivity of language L enlarged by macro features, we will build the conditional statement. The conditions will be expressions with inequalities and logical connectives. In this order, we assign a fuzzy set over \{0, 1\} to each expression.

For the condition $C = X_a < X_b$, the corresponding L-value is

$$V_C = \text{sg}(\text{decr}(\text{decr}(X_b, X_a), \text{prev}(\text{decr}(X_b, X_a)))).$$

### Negation

Let $V_C$ be the L-value for the condition $C$, then the L-value for $\neg C$ is

$$V_{\neg C} = \text{decr}(1, V_C).$$

### Conjunction

Let $A$ and $B$ be conditions with L-values $V_A$ and $V_B$. Then the L-value for $A \land B$ is $\text{sg}(\text{prev}(\text{sum}(V_A, V_B)))$.

### Disjunction

This is defined by $A \lor B = \neg(\neg A \land \neg B)$.

Then we can write the atomic conditions $X > Y$, $X < Y$, $X \neq Y$, $X = Y$, $X \geq Y$, $X \leq Y$, and any condition built by means of the above connectives.

Now we can define the conditional statement: Let $C$ be a condition, and $V_C$ its L-value; let $P$ be a $k$-program. Then we shall write the following program $P'$:

```
begin
  $X_k + 1 := V_C$;
  repeat $X_k + 1$ times $P$ semit
end
```

in the form IF $C$ THEN $P$. 

Analogously, let \( C \) be a condition with L-value \( V_C \); let \( P \) and \( Q \) be a \( k \)-program and an \( h \)-program respectively; then, we shall write the following program \( R \):

\[
\begin{align*}
\text{begin} & \\
& X_k + h + 1 := V_C; \\
& \text{repeat } X_k + h + 1 \text{ times } P \text{ semit; } \\
& X_k + h + 1 := \text{decr}(1, X_k + h + 1); \\
& \text{repeat } X_k + h + 1 \text{ times } Q' \text{ semit}
\end{align*}
\]

end

in the form IF \( C \) THEN \( P \) ELSE \( Q \). Here \( Q' \) is obtained from \( Q \) by substituting \( X_k + i \) for \( X_i \), \( 1 \leq i \leq h \).

Notice the use of the program 'sign', that ensures that the only returned L-values are of the form \( \alpha/0 + \beta/1 \), and hence the definite loops will be executed 0 or 1 times. This formulation is consistent with the idea expressed in [18]. The developments of [17] are along the same lines, by contemplating both alternatives with their associated degrees.

Notice also that the whole set of logical operators are obtained using only \( \text{add} \) and \( \text{previous} \). So, by using a fuzzy version of \( \text{add} \), and associating several meanings with \( \text{fuzzl} \), we can adapt conditional constructs to several applications. In the same way, logical operators could be obtained from another set of operations (for example, \( \text{previous} \) could be omitted by defining the conjunction in terms of the product, and the product in terms of \( \text{add} \)).

In any case, the conditional statement is easily defined in L language, as the above developments show. This point reveals that other constructs (for example, case statements) could be implemented, finally obtaining a powerful tool to describe and execute fuzzy algorithms.

4. Implementation

L language has been implemented on the basis of a source-to-source translator. For a given L program, the translator generates the corresponding C code. Then, by using a C compiler and the system linker, an executable module is obtained. To build this translator, we have used a lexical analyzer generator, namely LEX, a parser generator, namely YACC, and some modules directly written in C.

From an implementation point of view, the main feature of L is the memory model. To implement the theoretical construct of configuration, we have used a linked list of records, each record representing an element of the configuration (with the associated degree).

However, this device only would not be enough to implement the complete execution model of an L program, due to the block structures allowed in L language. At the beginning of the execution of a block, there are several elements in the configuration with a nonzero degree. Every element should be processed in the block independently.

This problem is solved by means of multilevel lists. Every time a REPEAT statement appears in the program, and for every element of the list corresponding to the present level \( n \), the following actions are executed:

- Generate a new level with index \( n + 1 \). This level becomes the present level.
- The list at this level is initialized with a copy of the element.
- The block is executed at the present level.
- Replace the element at level \( n \) with the list at level \( n + 1 \).
- Delete the present level.

Macro statements have been implemented as Pascal-like functions, with parameters only by value, as defined in Section 3.5. A preprocessor allows the user to redefine semiring operations \( \oplus \) and \( \otimes \). An execution parameter \( +nn \), \( nn \in [0, 1] \), specifies the threshold for the \( \alpha \)-cuts during program execution.
5. Examples

The expressive power of L language will be demonstrated with two examples: a fuzzy instruction proposed in [23] and an example of [17]. In the following code, some of the macros in Section 3.5 will appear, but only for crisp computations, like add or subtract.

Example 14. Let us consider the following fuzzy instruction: “If \( x \) is large, increase \( y \) by several units; if \( x \) is small, decrease \( y \) by several units; otherwise keep \( y \) unchanged”. In this example, we will consider the fuzzification only with respect to the concept ‘several’. Thus, we consider \( x \) large if \( x > A \) and \( x \) small if \( x < B \). The value \( x \) is read in the variable \( X_1 \), \( y \) in \( X_2 \), \( A \) in \( X_3 \), and \( B \) in \( X_4 \). For the concept ‘several’, we first read a value in \( X_5 \) and by means of a fuzzy instruction, we then assign a \( W \)-set around this input value to \( X_5 \). The code is as follows:

```
EXNINE;
begin
read(X1, X2, X3, X4, X5);
X5 := fuzz1(X5);
X6 := subtract(X1, X3);
X6 := sg(X6);
repeat X6 times
  repeat X5 times X2 := suc(X2) semit
    semit;
  X6 := subtract(X4, X1);
  X6 := sg(X6);
repeat X6 times
  repeat X5 times X2 := pred(X2) semit
    semit;
X1 := X2;
write(X1)
end.
```

Let the \( W \)-function for FUZZ1 be:

\[
f_1(x, y) = \begin{cases} 
1 - \frac{|y - x|}{x} & \text{if } |y - x| \leq 1, \ x \neq 0, \\
0 & \text{otherwise.}
\end{cases}
\]

Then, with input \((8, 9, 100, 10, 4)\) we obtain the output \(0.75/12 + 1/13 + 0.75/14\). In the same way, with input \((107, 20, 100, 10, 5)\) we obtain \(0.8/24 + 1/25 + 0.8/26\), and with input \((43, 17, 100, 10, 9)\) we obtain \(1/17\).

Example 15. Let us now consider the following situation: “Let the following sequence of numbers be given: 2, 4, 5, 3, 6, 7, 1, 87. The problem is to obtain a sum of ‘a few’ of these numbers” [17]. To solve this problem, let us assume that the data are stored in a sequential file. At the beginning of every execution of the loop, the file is rewound. To model the fuzzy number ‘a few’, we first read a value in \( X_1 \) and by means of a fuzzy instruction, we then assign a \( W \)-set around this input value to \( X_1 \).
The code is as follows:

```
EXTEN;
begin
  read(X1);
  X3:= fuzz1(X1);
  repeat X3 times
    read(X1);
    X2:= add(X2, X1)
    semit;
  X1:= X2;
  write(X1)
end.
```

Let the $W$-function for FUZZ1 be:

$$f_1(x, y) = \begin{cases} 
1 - \frac{|y - x|}{x} & \text{if } |y - x| \leq 3, x \neq 0, \\
0 & \text{otherwise}.
\end{cases}$$

Then with input 4, we obtain $0.25/2 + 0.5/6 + 0.75/11 + 1/14 + 0.75/20 + 0.5/27 + 0.25/28$.

6. Conclusions and future work

L language can be seen from two points of view: that of the theory of fuzzy computability, and that of fuzzy programming. In relation to fuzzy computability, the following result has been proved [16]: the class of L-programmable functions is equivalent to the class of $W$-recursive functions (in the sense of [7]). In relation to fuzzy programming, more powerful instructions and data structures could be added, in the way suggested by macro statements. We are presently working on some of these extensions.

Due to its Pascal-like syntax, L language is easy to learn. L is written entirely in ANSI C and thus is highly portable. To speed up the execution of L programs, a parallel implementation of L would be desirable. We are also working along this line, following the idea of assigning one configuration element to each processor.

We have also defined an indefinite loop statement (while loop). However, indefinite fuzzy loops have the same difficulties arising in nondeterministic programming with regard to non-termination [10].

Acknowledgements

The authors would like to thank the anonymous referees whose comments greatly improved the readability of the paper.

References