An Object Oriented Environment for Combining Different Styles of Programming - Ela

Ljubomir Jerinic
Institute of Mathematics
University of Novi Sad
Trg D. Obradovica 4
21000 Novi Sad, Yugoslavia

Ervin Varga
Institute of Control, Computer and Measurement
University of Novi Sad
Trg D. Obradovi_a 6
21000 Novi Sad, Yugoslavia

ABSTRACT

The motivation and the basic issues for designing the multi-paradigm programming system Ela is presented. In the Ela system different styles of programming (functional, logic and object-oriented) is permitted. The system is currently under development at the Institute of Mathematics in Novi Sad. At this phase of realization, the part which enables the combining the functional and object-oriented paradigm of programming is developed, and briefly described in this paper.

1. INTRODUCTION

Investigation in the field of combining different programming paradigms or programming styles in one environment or in one system, is almost old as appearing the distinct philosophy of solving problems by the aid of computers. Nowadays, there are several programming styles and appropriate programming languages in use in computing activity and practice. Any of these programming styles has its own advantages and disadvantages. In some problems or classes of problems one style is better then the others, but there is no perfect programming style or programming language appropriate for different use.

Almost all modern programming languages merge a few different programming paradigms, but in other hand all of them are at least the programming languages of only one programming style, i.e., one philosophy of solving some problem. Beside that, programming languages have distinct syntax and specific data structures on which they operated. In practice, a language that supports a paradigm well is often hard to distinguish from the paradigm itself.

2. MULTIPARADIGM IN PROGRAMMING LANGUAGES

A programming paradigm [Am92] is a collection of conceptual patterns that together develop the design process and definitively determine the way of solving some problems and program's structure. The control how we think about and formulate solutions, and even whether we arrive at solutions at all, directly judge is some program valid or not. Existing of different style of programming is the result of distinct and disparate problem spaces in which a solutions will be found. That problem space or areas require diverse solutions, and even in a single area contrasting techniques are often advantageous. According that there exist the separate styles or paradigms which served for solving different subclasses of the space of all problems. Every programming paradigm give in his definitional domain the possibility of resolving some problem on the efficient and simple way, and the describing the solutions in the form of computer program.
The need for some kind of multi-paradigm environments is almost as old as the appearance of different programming paradigms. Their advantages are obvious - they offer the possibility for operating with most appropriate features of distinct styles in a single application. On the other side, they are often too complicated and too big. The efforts in combining and merging some of different paradigms in a single one take many directions: from the design of completely new programming languages to the environments for simultaneous work with several different languages.

Logic programming shares many origins with functional programming, including an applicative nature, i.e. manipulation on values rather then with memory cells, recursion, chance for improvement for parallelism and so on. Of course there was a lot of differences between these to programming paradigms like radically different variable treatments, ability of higher order programs and fundamental support for nondeterministic executions. Hence all of that was certainly a good base for different attempts in combinations of that two paradigms or appropriate programming languages.

Robinson's and Sibert's LOGLISP was a one of the first attempt in this view that proved to be quite successful as wail as instructive. In the late '80 there was a lot of distinct attempts in combining the functional and logic style of programming in either new languages or adding the new abilities in existing one [DeG86] such as QUTE, FUNLOG, LEAF, APPLOG, EQLOG, TABLOG, Uniform and FRESH. CLOS - the Common Lisp Object System [Mo89] is a Common Lisp's entry into the object-oriented programming world, which initiated different approaches in additional investigations.

On the other hand, the programming language LEDA [Bu91] is a strongly typed, compiled language that tries to combine features of both the imperative (that portion is derived from the Algol tradition of languages like Pascal and Modula) and logical or relation-oriented styles of programming. The programming language C++ [St86] includes the imperative and object-oriented programming paradigm, and there is some attempts in combining functional and imperative style in [Oz91].

The development of truly multiparadigm languages opens up new possibilities for the computing community. They let us more easily compare solutions to problems presented in various styles and the programmers could better understand the importance and appropriate use of techniques from different paradigms while still working in only a single language. The chance of software development in a multiparadigm way allows that different parts of complex software systems might be design with different techniques which are the most suitable for that portion of the problem. For example in the field of Intelligent Tutors Systems the knowledge base of semantics nets might be best described using functional style, the graphical interfaces for a lesson design in object-oriented manner and the students module and a phase of learning could be nicely managed in relational or logic technique of programming. In a compiler design for some language, for another example, the parser could be realized using logical expressions, the symbol table as an object and the transformations on an intermediate form might be best described in a functional manner.

The approach in definition the multiparadigm programming environment ELa which combine imperative, object-oriented, logical and functional styles of programming in a single programming system is presented. We assume that all of including programming paradigms are on the same level in that environment. Also, the technique for an implementation that system is described.

Firstly, the object-oriented, functional and logical paradigms will be briefly described.

3. OBJECT-ORIENTED PARADIGM

The imperative paradigm is characterized with the von Neumann model of computer that consists of a large memory and a processor unit. Although von Neumann machines underlie the implementation of almost all paradigms, the imperative style of programming uses this computer model for conceptualizing solutions. The other paradigms, by contrast, use conceptual models removed from this implementation model. Step-by-step computational sequences characterize this style, and during computations the processor repeatedly modifies the space of the variables, i.e., the abstraction of memory cells.

As we said, in the imperative paradigm, the conceptual model is a single memory space into which abstract data values are represented. On that data space one or more procedures, which represents operations, are applied. Each procedure deals directly with
stored representation. The object-oriented paradigm [We90] hold much of this model, but, on the contrary, procedures operate on abstract values, called objects, rather then on memorized representation. As a result, this paradigm requires the capabilities of defining new objects composed of existing objects in a hierarchical order, and of manipulating them by defined procedures called methods. Object-oriented paradigm first defines suitable objects for the problem at hand, then uses these objects to describe step-by-step operational sequences. The process of solving problems in object-oriented style realize the top-down technique of modelling world.

Manipulation of abstract values rather then concrete representation requires respect for encapsulated state of objects. That means that the right of objects to define their concrete representations and to perform all manipulations upon such representation. This is accomplished by sending messages that describe the desired manipulations and leaving it to the objects to perform them. Object which are defined via other subobjects, use operational sequences to alter their internal representations. Such sequences include sending messages to their subobjects. This process proceeds until at some level in that hierarchy the objects and the methods defined on them are primitive.

**Inheritance**, single or multiple, is the second characteristics associated with object-oriented style of programming. It is based on the concept of object classes and hierarchy. A class is the definition of an object from which instances of the definition are created. Inheritance allows rapid definition of a new object class from the concrete representation and the methods of an existing class. The new class includes all of the methods defined on the inherited representation, as well as any new concrete representations and new or revised methods added to it.

The third characteristics of object-oriented paradigm is **message-passing**. The objects could be consider as active entities that send messages to one another. To support this model, procedures must be polymorphic, i.e., multiply defined with particular invocations determined by examining the types of the parameter objects.

While the message-passing mechanism of the object-oriented style is computational equivalent to the procedure call approach of the extended imperative paradigm (with a possibility to have a complex data in the procedure or the function definition), it leads to a very different way of looking at problem solutions. There is a conceptual difference between searching through a some procedures with same name for a match of the particular parameter types, and in sending messages to a particular objects that knows only one such method. In the former case of procedure calls the main program controls all the computations, while in the message-passing mechanism each object has full responsibility for correct handling requests made directly to it.

### 4. FUNCTIONAL PARADIGM

The functional programming style and relevant programming languages trace their origins to the λ-calculus, recursion equation systems and combinatory logic. We could say that in the functional paradigm [He80, Tu82] the computational model is based on mathematical model of functional compositions. In this model, the result of one computation is input to the next, and so on until some composition yields the desired result. There is no concept of a memory location that is assigned or modified. Instead of that, there are only intermediate values, which are the results of prior evaluations and the inputs to succeeding computations. There no for of command and all functions are referential transparent.

The functional style of programming [Hu89] involve the concept of functions as first-class objects, i.e., the ability of high-order functions. This means that functions can be treated as data, passed as parameters, constructed and returned as values, and composed with other forms of data. There is two ways of the functions specifications: operationally, i.e., with some control sequencing which is the part of some functional language), and mathematical or definitional without any of controlling statements.

With the operational style, the programmers are responsible for the complete sequencing of instructions, including proper termination of the evaluations. By contrast, programming in the definitional functional style provides for termination without requiring explicit control sequencing. In the operation style the conceptual model for problem solving is a construction in which a
sequence of steps that will use functional composition to compute the desired result is described. On the other hand, in the definitional model we approach the problem as a collection of disjoint transformations that, taken collectively, define a computational function.

5. LOGICAL PARADIGM

The logical programming paradigm [Ko79a, Ko79b] are based on a procedural interpretation of the first order predicate calculus. Programming in that style suppose that a set of known facts, and a set of rules that allow deduction of other facts are written. Thus, logic programming from the programmer's viewpoint is a matter of correctly declaring all required facts, rules and relevant relations between objects needed for solving some problem.

Most of the logic programming languages known today, have been based on Horn clauses, a subset of first order predicate calculus. The clausal notation of predicate logic combines variables, constants, and expressions to represent conditional propositions. Horn clauses are a restricted form of predicate logic with exactly one conclusion in any clause.

A solution of some problem in the logic style is found if a suitable set of rules and substitutions exists such that applying the substitutions to the rules produces a set of grounded rules, i.e., a set with no free variables. A process known as unification develops substitutions for free variables. However, there is no similarly deterministic algorithm for selecting rules, which leads to different implementations which could be very inefficient.

Although the Prolog language [Cl81] has become synonymous with logic programming, it actually includes a specific problem solving strategy, and a number of features not true to the logic paradigm. While its pure logic features provide the advantages of the logic style of programming, Prolog can be used in an entirely different way that look like as the imperative paradigm (for example the cut mechanism which control the backtracking technique of programming).

The logic paradigm is intended for general purpose programming. Certainly, pseudodefinitional logic is used for production programming in industry, where it is very well-suited to certain types of problems. These include backtracking search problems that may require multiple solutions, problems that are naturally represented in terms of production rules (such as natural language translation), and executable specifications for rapid prototyping.

6. THE MULTIPARADIGM SYSTEM ELa

The programming system ELa Fig. 1. consists of the integrated user interface, editor, compiler and linker. The system is realized on the programming language C++ for the IBM PC compatible computers. It consists of the features imported from the object-oriented style represented in C++ programming language [St86], the functional paradigm illustrated in a subset of A_LispKit Lisp programming language [Je92] and the logic style depicted in a subset of standard Prolog [Cl81].

User interface is organized like standard menu systems in window like applications. The screen management, the editor and the whole menu system is implemented with the assistance of well known libraries for window maintenance, controlling the mouse etc.

6.1. THE FUNCTIONAL AND LOGICAL PART OF THE SYSTEM ELa
The programming in the system ELa consists of writing the program in C++, or a subset of A_LispKit Lisp, or in a subset of standard Prolog programming language, Fig. 2.

For our work, we basically take the approach to combine functional and logic programming with object-oriented style by adding the functional and logic capabilities into object-oriented framework. We started with an object-oriented framework provided by C++ because of it's efficiency and it's compile time checking. In the first part of our investigation we realized the A_LispKit Lisp language [Je92] which is improved version of the LispKit Lisp programming language P. Henderson [He80, He83]. The most important differences of A_LispKit Lisp from previous language are: completely new syntax and semantics, different method of implementing based on environment - A_SECD (modified Landin's famous abstract machine), possibility of defining and using functional libraries, implementations of some built-in functions for interactive work, and so on.

The complete syntax of A_LispKit Lisp language, in extended Backus- Naur form is:

```
<WhereRecBlok> := '{' <Expr>(Where' | WhereRec')
<Name> := ( '<Expr' | ImportFrom'<Name' | '@'<Name>)
{'And'<Name> := ( '<Expr' | ImportFrom'<Name' | '@'<Name>))
'}.'
<Expr> := <SimplExpr> [ <Rel> <SimplExpr> ]
<Rel> := '=' | '<' | '>' | '=' | '<=' | '>' | '<=' | '<>' | '='| '=>' | '><'.
<SimplExpr> := ['+' | '-'] <Term> 
<OpAdd> := '+' | '-' | '#'.
<Term> := <Factor> 
<OpMul> := '*' | 'Div' | 'Mod' | '&'.
<Factor> := '(' <Expr> ')'
| <WhereRecBlok> | <DefFun> | <SeqBlock> |
| <CondFun> | <Numb> | <OFun> |
| <DefFun> := 'Lambda' ( '(' | ')' )
| <Const> | <IOFun> |
| <EvalFun> := 'Eval(' <Name>, <Expr> ')'.
| <OFun> := 'Atom' | 'C' ( 'a' | 'd' ) | 'r' | 'Cons' | 'List' | 'Plus' | 'Times' |
| 'Null' | 'Head' | 'Tail'.
| 'DefFun' := 'Lambda' ( '(' )
| 'Case' <Expr> ':' <Expr> |
| 'Other' <Expr> ]
| 'EndCase'.
| 'IOFun' := 'Read(' <Name>, ',<Name>, ',<Expr> ')' |
| 'Write(' <Name>, ',<Name>, ')', 'Input(' <Name>, ',<Expr> ')' |
| 'Type(' <Name> ').'
| 'Numb' := [ '+' | '-' ] <Digits>.
| 'Digits' := <Digit> 
| <Letter> := 'a' | 'A' | 'b' | 'B' | ... | 'x' | 'X' | 'y' | 'Y' | 'z' | 'Z'.
| 'Digit' := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.
| 'Name' := <Letter> [ <Letter> | <Digit> ]
| 'Const' := [ '+' | '-' ] <Digits> |
| <Digits> := [ '+' | '-' ] <Digits> ]
```

```
```
Let us now define the meaning (denotational) functions for A_LispKit Lisp primitives. The S-expressions have certain subtypes:

\[ \text{subsets:} \]
\[ <\text{Pair}> \]
\[ <\text{Real}> \]
\[ <\text{Integer}> \]
\[ <\text{Constructor}> \]
\[ <\text{S} \]

\[ \text{subtypes:} \]
\[ 1 \]
\[ 1 \]
\[ 1 \]
\[ 1 \]
\[ 1 \]
\[ 1 \]
\[ 1 \]
\[ 1 \]

\[ \text{Mod} \]
\[ \text{Div} \]
\[ / \]
\[ \ast \]
\[ + \]
\[ T \]
\[ F \]

The particular definition of semantic functions for A_LispKit Lisp can be given in the following form:

\[ \text{Value} : \mathcal{R} \rightarrow \mathcal{R}, \]
\[ \text{N_Value} : \mathcal{R} \rightarrow \mathcal{R}_{\text{Numeric}} \]
\[ \text{S_Value} : \mathcal{R} \rightarrow \mathcal{R}_{\text{Symbolic}} \]
\[ \text{N_SExp} : \mathcal{R}_{\text{Numeric}} \rightarrow \mathcal{R}, \]
\[ \text{S_SExp} : \mathcal{R}_{\text{Symbolic}} \rightarrow \mathcal{R}, \]
\[ \text{L_SExp} : \mathcal{R} \rightarrow \mathcal{R}_{\text{Logic}}, \]
\[ \text{C_SExp} : \mathcal{R} \rightarrow \mathcal{R}_{\text{Atoms}}, \]
\[ \text{P_SExp} : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}_{\text{Pair}}, \]
\[ \text{List_SExp} : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}_{\text{List}}. \]

Let us now define the denotational semantics for some primitives of A_LispKit Lisp language:

\[
\text{N_Value}(<\text{C_SExp}(s)), s \in \mathcal{R}_{\text{Numeric}}
\]

\[
\text{Quote}(s) \Rightarrow \{ \text{S_Value}(<\text{C_SExp}(s)), s \in \mathcal{R}_{\text{Symbolic}}
\]
\[
\text{Value}(<\text{L_SExp}(s)), s \in \mathcal{R}_{\text{Logic}}
\]
\[
\text{Value}(<\text{C_SExp}(s)), \text{else}
\]

\[
\text{Cons}(s_1, s_2) \Rightarrow (\text{P_SExp}(\text{Value}(s_1), \text{Value}(s_2)), s_1, s_2 \in \mathcal{R}) \land
\]
\[
(\text{Value}(\text{P_SExp}(\text{Value}(s_1), \text{Value}(s_2))) = (s_1, s_2))
\]

\[
\text{List}(s_1, s_2, \ldots, s_k) \Rightarrow (\text{List_SExp}(\text{Value}(s_1), \text{Value}(s_2), \ldots, \text{Value}(s_k)), s_i \in \mathcal{R},
\]
\[
i=1, \ldots, k) \land (\text{Value}(\text{List_SExp}(\text{Value}(s_1), \text{Value}(s_2), \ldots, \text{Value}(s_k))) = (s_1, s_2, \ldots, s_k))
\]

\[
s_1 + s_2 \Rightarrow \text{N_Value}(s_1) + \text{N_Value}(s_2), s_1, s_2 \in \mathcal{R}_{\text{Numeric}}
\]
\[
s_1 - s_2 \Rightarrow \text{N_Value}(s_1) - \text{N_Value}(s_2), s_1, s_2 \in \mathcal{R}_{\text{Numeric}}
\]
\[
s_1 \ast s_2 \Rightarrow \text{N_Value}(s_1) \ast \text{N_Value}(s_2), s_1, s_2 \in \mathcal{R}_{\text{Numeric}}
\]
\[
s_1 / s_2 \Rightarrow \text{N_Value}(s_1) / \text{N_Value}(s_2), s_1, s_2 \in \mathcal{R}_{\text{Numeric}}
\]
\[
s_1 \text{ Mod } s_2 \Rightarrow \text{N_Value}(s_1) \text{ Mod } \text{N_Value}(s_2), s_1, s_2 \in \mathcal{R}_{\text{Numeric}},
\]

where \{+,-,\ast,,/,\text{Mod},\text{Div}\} denote the appropriate arithmetic operations in the implementation language.

\[
s_1 \& s_2 \Rightarrow \text{L_SExp}(s_1) \land \text{L_SExp}(s_2), s_1, s_2 \in \mathcal{R}_{\text{Logic}}
\]
\[ s_1 \# s_2 \Rightarrow L_{SExp}(s_1) \lor L_{SExp}(s_2), s_1, s_2 \in \mathcal{R}_{Logic} \]

\[ ! s_1 \Rightarrow L_{SExp}(s_1), s_1 \in \mathcal{R}_{Logic} \]

where \{ \&, \lor, \neg \} denote the appropriate logical operations in the implementation language.

\[ s_1 \equiv s_2 \Rightarrow L_{SExp}(\text{Value}(s_1) \equiv \text{Value}(s_2)), s_1, s_2 \in \mathcal{R} \]

\[ s_1 \not\equiv s_2 \Rightarrow L_{SExp}(\text{Value}(s_1) \not\equiv \text{Value}(s_2)), s_1, s_2 \in \mathcal{R} \]

\[ s_1 \not\approx s_2 \Rightarrow L_{SExp}(\text{Value}(s_1) \not\approx \text{Value}(s_2)), s_1, s_2 \in \mathcal{R} \]

\[ s_1 > s_2 \Rightarrow L_{SExp}(\text{Value}(s_1) > \text{Value}(s_2)), s_1, s_2 \in \mathcal{R}_{Numeric} \]

\[ s_1 < s_2 \Rightarrow L_{SExp}(\text{Value}(s_1) < \text{Value}(s_2)), s_1, s_2 \in \mathcal{R}_{Numeric} \]

\[ s_1 \geq s_2 \Rightarrow L_{SExp}(\text{Value}(s_1) \geq \text{Value}(s_2)), s_1, s_2 \in \mathcal{R}_{Numeric} \]

\[ s_1 \leq s_2 \Rightarrow L_{SExp}(\text{Value}(s_1) \leq \text{Value}(s_2)), s_1, s_2 \in \mathcal{R}_{Numeric} \]

\[ s_1 = s_2 \Rightarrow L_{SExp}(\text{Value}(s_1) = \text{Value}(s_2)), s_1, s_2 \in \mathcal{R}_{Logic} \]

where \{ \equiv, \not\equiv, >, \leq, \geq, \not\approx \} denote the appropriate relational operations in the implementation language.

**Sequence** \( s_1, s_2, \ldots, s_k \Rightarrow \text{Value}(\text{Value}(s_1), \text{Value}(s_2), \ldots, \text{Value}(s_k)) \), \( s_i \in \mathcal{R} \), \( i = 1, \ldots, k \)

\[ \text{Eval}(e_1, e_2) \Rightarrow \text{Value}(e_i(\text{Value}(e_3)), e_i \in \mathcal{R}, i=1,2) \]

\[ \text{Value}(s_1), s \in \mathcal{R}_{Constructor}, s \equiv (s_1 \cdot p), p \in \mathcal{R} \]

**Head** \( s \Rightarrow \{
\)

\[ ?, \text{else} \]

\[ \text{Value}(s_1), s \in \mathcal{R}_{Constructor}, s \equiv (s_1 \cdot p), p \in \mathcal{R} \]

**Car** \( s \Rightarrow \{
\)

\[ ?, \text{else} \]

\[ \text{Value}(s_1), s \in \mathcal{R}_{Constructor}, s \equiv (p \cdot s_1), p \in \mathcal{R} \]

**Tail** \( s \Rightarrow \{
\)

\[ ?, \text{else} \]

\[ \text{Value}(s_1), s \in \mathcal{R}_{Constructor}, s \equiv (s_1 \cdot p), p \in \mathcal{R} \]

**Cdr** \( s \Rightarrow \{
\)

\[ ?, \text{else} \]

\[ C\{a \mid d\}r(s) \Rightarrow \text{Value}(C\{a \mid d\}r(\text{Value}(C\{a \mid d\}r(\text{Value}(C\{a \mid d\}r(\ldots \text{Value}(C\{a \mid d\}r(s))))))), s \in \mathcal{R}_{Constructor} \]

**Atom** \( s \Rightarrow L_{SExp}(\text{Value}(s)), s \in \mathcal{R}, L_{SExp}(\text{Value}(s)) = 'T', \text{if } s \in \mathcal{R}_{Atoms} \cup \{\text{NIL}\}, \]

\[ \text{else } 'F \]

**Null** \( s \Rightarrow L_{SExp}(\text{Value}(s)), s \in \mathcal{R}, L_{SExp}(\text{Value}(s)) = 'T', \text{if } s \equiv \text{NIL}, \text{else } 'F. \]

If \( I \) Then \( s_1 \) Else \( s_2 \Rightarrow (\text{Value}(s_3)) \land (L_{SExp}(l) \equiv 'T \Rightarrow s_3 \equiv s_1) \land (L_{SExp}(l) \equiv 'F \Rightarrow s_3 \equiv s_2) \land (\text{Value}(s_3) \equiv ?, \text{else}) \]

where \( l \in \mathcal{R}_{Logic}, \text{and } s_1, s_2, s_3 \in \mathcal{R} \)

**Case** \[ l_1 : s_1 \]

\[ \ldots \]

\[ \text{or } \]

\[ l_k : s_k \]

\[ \text{other } s_{k+1} \]

**EndCase** \[ \Rightarrow \text{Value}(s_{k+2}) \]

where \( l_i \in \mathcal{R}_{Logic}, i = 1, \ldots, k+1, s_j \in \mathcal{R}, j = 1, \ldots, k+2, \text{and} \)

\[ L_{SExp}(l_1) \equiv 'T \Rightarrow s_{k+2} \equiv s_1, \]

\[ \ldots \]

\[ L_{SExp}(l_k) \equiv 'T \Rightarrow s_{k+2} \equiv s_k, \text{or} \]

\[ s_{k+2} \equiv s_{k+1} \]

\[ f = \text{Lambda}(x_1, x_2, \ldots, x_n) e \Rightarrow f \_ \text{Value}(\ [e \ ] \rho) \]

where the context \( \rho \) is:

\[ \rho \equiv \{ x_1 \_ \text{Value}(x_1), x_2 \_ \text{Value}(x_2), \ldots, x_n \_ \text{Value}(x_n) \}, e \in \mathcal{R}, x_i \in \mathcal{R}, i=1,\ldots, n, \]

\[ i \in \mathcal{N} \cup \{0\} \]

the symbol _ denotes the binding operations, i.e. the binding of a variable to a value,

\[ f(x_1, x_2, \ldots, x_n) \Rightarrow \text{Value}(\ [e \ ] \rho), \text{where a function } f \text{ is defined with } f = \text{Lambda}(x_1, x_2, \ldots, x_n) e, \text{and a context } \rho \equiv \{ x_1 \_ \text{Value}(x_1), x_2 \_ \text{Value}(x_2), \ldots, x_n \_ \text{Value}(x_n) \} \]

\[ [E \ Where \{ \text{Rec} \]

\[ I_1 = (e_1 \mid \text{ImportFrom Lib}_1 \mid @Fun) \]

And \[ I_2 = (e_2 \mid \text{ImportFrom Lib}_2 \mid @Fun) \]

And \[ \ldots \]

And \[ I_n = (e_n \mid \text{ImportFrom Lib}_n \mid @Fun) \] \[ \Rightarrow \text{Value}(\ [E \ ] \rho) \]

where \( \rho \equiv \{ I_1 \_ \text{Value}(e_1), I_2 \_ \text{Value}(e_2), \ldots, I_n \_ \text{Value}(e_n) \} \)
A_S ECD machine can be defined as a general function \textbf{Exec} [He80] which takes a compiled version of a function \textbf{Fun}, denoted with \textbf{Fun}^*, and the S-expression representation of the arguments \textbf{Args}. Thus, it produces an S-expression representation of the result of applying \textbf{Fun} to \textbf{Args}. The formal definition of A_S ECD machine, the function \textbf{Exec}, given in terms of denotational semantics approach, is:

\[
\textbf{Exec} : \text{	extsubscript{A_S ECD}} \times \_ \rightarrow \_, \quad \text{and} \quad \textbf{Eval} [ \textbf{Exec}(\textbf{Fun}^*, \textbf{Args}) ] \rho = \textbf{Eval}_{\text{	extsubscript{A_S ECD}}} [ \textbf{Fun}^* (\textbf{Args}) ] \rho = \textbf{Res},
\]

where \textbf{Fun}^* \in \text{	extsubscript{A_S ECD}}, \textbf{Args} \in \_ and \textbf{Res} \in \_. The set \_ represents a set of all programs-functions of A_LispKit Lisp language, \_ is a set of all S-expressions, and the set \text{	extsubscript{A_S ECD}} is a set of all possible, executable, programs in the machine language of A_S ECD machine. The general denotational function \textbf{Eval} is defined by \textbf{Eval} : \_ \rightarrow \_, where \_ is a set of all expressions, and \_ is a set of all values of a language. The denotational function \textbf{Eval}_{\text{	extsubscript{A_S ECD}}} describes a semantics of the A_S ECD machine, and it is given with \textbf{Eval}_{\text{	extsubscript{A_S ECD}}} : \text{	extsubscript{A_S ECD}} \rightarrow \_, \quad \text{where} \text{	extsubscript{A_S ECD}} is a set of all valid functions written in the machine languages of A_S ECD machine.

From the point of view of operation semantics, formal definition of A_S ECD machine, given by the function \textbf{Exec} is:

\[
\textbf{Exec}(\textbf{Fun}^*, \textbf{Args}) = \text{Apply}(\textbf{Fun}, \textbf{Args}),
\]

\[
\text{Compile}(\textbf{Fun}) = \textbf{Fun}^*,
\]

where the function \textbf{Compile} translates a source code of a program function \textbf{Fun} into the machine language of A_S ECD machine. That is, in some way the A_S ECD machine, given the S-expression representations of the compiled function (a machine language program) and its arguments, executes the machine language program to compute the result of applying that function to these arguments.

The function \textbf{Exec}(\textbf{Fun}^*, \textbf{Args}) is implemented in such a way that it operates a stack for the evaluation of function calls, much as the process described. Since the program \textbf{Fun}^* is an S-expression and since the data with which it operates are S-expressions, the natural notation for expressing the state of this stack machine is the S-expression notation. Thus, if we wish to denote the stack in such a way that its top item is an S-expression of \textbf{X} we will write (\textbf{X}.s), where \textbf{s} represents the remaining items.

Strictly speaking, a pure stack is a data structure which has only two operations, the \textbf{pushing} of a new element onto the stack, and conversely, the operation of \textbf{popping} an element from the stack. It is said to be used in last-in-first-out discipline. The denotational semantics of a stack is:

\[
\text{Stack} : \_ \rightarrow \_,
\]

\[
\text{Stack}_{\text{Pop}} : \_ \rightarrow \_,
\]

\[
\text{Stack}_{\text{Push}} : \_ \times \_ \rightarrow \_,
\]

\[
\textbf{Eval} [ \text{Stack}_{\text{Pop}} ((\textbf{X}.s)) ] \rho = \textbf{Eval} [ \textbf{X} ] \rho,
\]

\[
\textbf{Eval} [ \text{Stack}_{\text{Push}} (\textbf{X}, (\textbf{Y}.s)) ] \rho = \textbf{Eval} [ (\textbf{X}.(\textbf{Y}.s)) ] \rho,
\]

for ( \forall \textbf{X} \in \_), ( \forall \textbf{Y} \in \_) and ( \forall \textbf{s} \in \_).

The A_S ECD machine consists of five registers and each of them holds an S-expression. These registers derives their names from the purpose they have in dealing with S-expressions:

\begin{itemize}
  \item \textbf{S} \quad \text{the stack, used to hold the intermediate results during computation. At the end of the program execution, the top of the stack \textbf{S} contains the final result,}
  \item \textbf{E} \quad \text{the environment, holds the values which are bound to variables during evaluation,}
  \item \textbf{C} \quad \text{the control list, used to hold the machine-language program which is currently executed. In each moment of the evaluation process, the first element of the control list is the command which will be processed next,}
  \item \textbf{D} \quad \text{the dump, which saves the values of all other registers \textbf{S}, \textbf{E}, and \textbf{C} during a new function call.}
\end{itemize}
the resident library manager, the stack which contains the resident libraries, i.e. the programs in an executable code written in the machine language of A_SECD machine, which are consulted during the evaluation of a program.

The machine language of the A_SECD machine consists of a certain number of commands. The execution of a command forces the machine to change its state, i.e. the contents of its registers. We call this a **machine transition**, and it can be denoted, from the point of operational semantics, in the following way:

$$S \quad E \quad C \quad D \quad L \quad \Rightarrow \quad S' \quad E' \quad C' \quad D' \quad L', $$

where S, E, C, D and L are the contexts of the registers before the next command execution, and S’, E’, C’, D’, and L’ denote the new contexts of the all registers after that. For example, a machine transition of arithmetic operations defined under the S_SECD machine is:

$$(a \cdot b. S') \quad (OpA.C) \quad D \quad L \quad \Rightarrow \quad (b \cdot a. S) \quad E \quad C \quad D \quad L,$$

where $OpA \in \{ADD, SUB, MUL, DIV \ldots \}$ and $SiA \in \{\oplus, \ominus, \otimes, \ldots \}$. For another example, a machine transition of relation between the data of A_SECD machine is:

$$(a \cdot b. S) \quad (ReA.C) \quad D \quad L \quad \Rightarrow \quad (b \cdot a. S) \quad E \quad C \quad D \quad L,$$

where $ReA \in \{GT, GE, LE, NE \ldots \}$ and $SiA \in \{>, \geq, \leq, \neq, \ldots \}$.

As all registers of A_SECD machine, according to the rules of machine transition, perform some operations on S-expressions, the simulator of A_SECD machine is naturally implemented by mapping these rules in some procedures of the implementation language.

On the other hand, the meaning of the instructions of A_SECD machine, in a mathematical sense is not clear. A much better way to describing the meaning of A_SECD machine, i.e. of its semantics, is to used denotational semantics approach.

A_SECD machine, from operational semantics standpoint, has been defined by an **Exec** function. Now, the modified denotational semantics of A_SECD machine will be presented. The meaning function of A_SECD machine in the terms of denotation is $Eval_{A\_SECD} : \_ \rightarrow \_$. where _ is a set of all S-expressions.

Firstly, we must define the semantics of the abstract data type, which is used in the implementation of S-expression, and A_SECD machine operations, denoted with **LispCell**. The denotational semantics of some operations which are defined under that data type, is given by a semantic function: $Sem_{L\_C} : \_ \rightarrow \_$. where _ is a set of operations and functions of A_SECD machine and/or operations defined under data type **LispCell**. The set _ is a set of standard types of implementation language like **Real**, **Boolean**, **Integer** and so on, together with data type **LispCell**, _ = { **Real**, **Boolean**, **Integer**, **String**, **LispCell** }.

With _L_C C _ , we denoted a set of all possible values of the type **LispCell**. Let us define all of the subsets of the set _L_C:

- _L_Creal (the set of real values), _L_Cinteger (the set of integer values), _L_Cpair (the set of all pairs), _L_Clist (the set of all lists), _L_Csymbo (the set of all symbolic atoms), _L_Cboolean (_L_Cboolean = {T, F}), and _L_Cnull (_L_Cnull = {NIL}).

Also, with _L_L C _ , we denoted the set of all values of the types of the implementation language. This set has the following subsets: _L_Lreal (the set of all real values), _L_Linteger (the set of all integer values), _L_Lboolean (the set of all logical values), and _L_Lstring (the set of all string values).

For $\forall f \in _$ we will define: $f : _ \times \ldots \times _ \rightarrow _$, where the _ is defined by _ = {•, **LispCell**, **Integer**, **Real**, **String**}, where the • represents that some operations or functions of A_SECD machine have no arguments.

At the end of the paper, we will give some examples of the denotational semantics definitions using the object-oriented modifications for some operations defined under the S-expressions.

**Converting Functions**. These functions convert data between some subtypes of LispCell data type, which are used in the implementation of S-expressions. The converting functions are: ValueRealLC, ValueIntegerLC, CostIntegerLC, ConstRealLC, ConstStringLC, ValueStringLC, etc. Their semantic definitions are:

**Class ConvFun**
Mapping
ConstRealLC : __ILreal → __LCreal
ValueRealLC : __LCreal → __ILreal
ConstIntegerLC : __ILinteger → __LCinteger
ValueIntegerLC : __LCinteger → __ILinteger

Rules
ConstRealLC(r : __ILreal) : __LCreal ≡ EvalA_SECD [ ConstRealLC(r) ] ρ = r;
ValueRealLC(r : __LCreal) : __ILreal ≡ EvalA_SECD [ ValueRealLC(r) ] ρ = r;
ConstIntegerLC(i : __ILinteger) : __LCinteger ≡ EvalA_SECD [ ConstIntegerLC(i) ] ρ = i;
ValueIntegerLC(i : __LCinteger) : __ILinteger ≡ EvalA_SECD [ ValueIntegerLC(i) ] ρ = i;

End; (* ConvFun *), where ρ is an arbitrary context in which all bindings of variable to their values are performed.

Now, we could easily proved the following statements:

Theorem 1. (∀ x ∈ __ILreal)(ValueRealLC(ConstRealLC(x)) = x),
Theorem 2. (∀ y ∈ __LCreal)(ConstRealLC(ValueRealLC(y)) = y),
Theorem 3. (∀ i ∈ __ILinteger)(ValueIntegerLC(ConstIntegerLC(i)) = i),
Theorem 4. (∀ j ∈ __LCinteger)(ConstIntegerLC(ValueIntegerLC(j)) = j), etc.

Logical Functions. The semantic class for some logical operations are:

Class LogFun
Mapping
AndLC : __LCBoolean × __LCBoolean → __LCBoolean

Rules
AndLC(l1 : __LCBoolean, l2 : __LCBoolean) : __LCBoolean ≡
EvalA_SECD [ AndLC(l1,l2) ] ρ =
{ EvalA_SECD [ l1 ] ρ ∧ EvalA_SECD [ l2 ] ρ } ∈ __ILBoolean,
EvalA_SECD [ l1 ] ∈ __ILBoolean,
EvalA_SECD [ l2 ] ∈ __ILBoolean,
SemLC [ l1 ] ∈ __LCBoolean,
SemLC [ l2 ] ∈ __LCBoolean,
StateTrans((l1 l2.S), E, (AND.C), D, L);

End; (* LogFun *).

Arithmetic Functions. Before we define the class of arithmetic functions, let us introduced some new notations:

__ILnumpod ≡ __ILinteger ∪ __ILreal,
__LCnumpod ≡ __LCinteger ∪ __LCreal, i
__LCstruct ≡ __LCpar ∪ __LClist,

for the subsets of numeric data for the implementation language and the data type LispCell. Then, the semantic definitions of the arithmetic functions are:

Class ArithFun
Mapping
AddLC : __LCnumpod × __LCnumpod → __LCnumpod

Rules
AddLC(l1 : __LCnumpod, l2 : __LCnumpod) : __LCnumpod ≡
EvalA_SECD [ AddLC(l1,l2) ] ρ =
{ EvalA_SECD [ l1 ] ρ ⊕ EvalA_SECD [ l2 ] ρ } ∈ __ILnumpod,
EvalA_SECD [ l1 ] ∈ __ILnumpod,
EvalA_SECD [ l2 ] ∈ __ILnumpod,
SemLC [ l1 ] ∈ __LCnumpod,
SemLC [ l2 ] ∈ __LCnumpod,
Constructing Operations. Constructing operations form some structure object (lists or pairs), using the top stack items. We define, for example, a semantic class for the A_SECD machine operation CONS, which forms a pair of two top stack items, with:

**Class ConstructOper**

**Mapping**

\[
\text{ConsLC} : _\times_ \rightarrow _\times_\text{LCpair}
\]

**Rules**

\[
\text{ConsLC}(l_1 : \_ : l_2 : \_ : \_) : _\times_\text{LCpair} \equiv
\]

\[
\text{Eval}^{A\_\text{SECD}} \left[ \text{ConsLC}(l_1 l_2) \right] \rho = \left\{
\begin{array}{l}
'(' \odot \text{Eval}^{A\_\text{SECD}} [l_1 \_] \rho \odot ' : \odot \text{Eval}^{A\_\text{SECD}} [l_2 \_] \rho \odot ')') \in _\times_\text{LCpair}, \\
\text{Eval}^{A\_\text{SECD}} [l_1] \in _, \\
\text{Eval}^{A\_\text{SECD}} [l_2] \in _, \\
\text{Sem}^{L\_C} [l_1] \in _, \\
\text{Sem}^{L\_C} [l_2] \in _, \\
\text{StateTrans}(l_1 l_2 S), E, (\text{CONS}.C), D, L);
\end{array}
\right.
\]

End; (* ConstructOper *).

Selecting Operations. The class of selecting operations of A_SECD machine, in terms of denotational semantics, in the object-oriented approach is defined with:

**Class SelectOper**

**Mapping**

\[
\text{HeadLC} : _\rightarrow _
\]

**Rules**

\[
\text{Eval}^{A\_\text{SECD}} \left[ \text{HeadLC}(l_i) \right] \rho = \left\{
\begin{array}{l}
\text{HeadLC}(l_i : \_ : \_) : _\equiv \text{Eval}^{A\_\text{SECD}} [x] \rho \in _., \text{if } l_i=(x,y) \land x \in _.
\end{array}
\right.
\]

\[
\text{Eval}^{A\_\text{SECD}} [l_1] \in _, \\
\text{Sem}^{L\_C} [l_1] \in _, \\
\text{StateTrans}(l_1 l_2 S), E, (\text{CAR}.C), D, L);
\]

End; (* SelectOper *).

In all above definitions, the variables S, E, C, D and L are global, and \( \text{Eval}^{A\_\text{SECD}} [l_1] \in _{\_} \) where \( l \in \{S, E, C, D, L\} \).
The actual process of translation is done in two phases, Fig. 3. During the first phase of the translation the part of program written in A_LispKit Lisp language is decoded in the A_SECD machine instruction, and during the second phase, the C++ codes are actually generated according to the information stored in the symbol and operation table.

**CONCLUSION**

In this paper we presented the system **ELa** in which the possibilities of combining functional and logical styles of programming in object-oriented environment is allowed. The combination of these different paradigm shows a great potential to support the representation of domain knowledge into declarative data types and make them reusable in different context.

However, much work needs to be done. The system is currently under the development, and in the further work the subset of the Prolog Language and the Warren Prolog machine will be incorporating in the system **ELa**, on the same manner as we do with A_LispKit Lisp and A_SECD machine. Then we must realize the vertical communication between data and different programming paradigms.

**REFERENCES**