Programming Languages in Artificial Intelligence

Günter Neumann,

German Research Center for Artificial Intelligence (LT–Lab, DFKI)

Glossary

Clauses Prolog programs consist of a collection of statements also called clauses which are used to represent both data and programs.

Higher–order function is a function definition which allows functions as arguments or returns a function as its value.

Lists Symbol structures are often represented using the list data structure, where an element of a list may be either a symbol or another list. Lists are the central structure in Lisp which are used to represent both data and programs.

Recursion An algorithmic technique where, in order to accomplish a task, a function calls itself with some part of the task.

Symbolic computation AI programming involves (mainly) manipulating symbols and not numbers. These symbols might represent objects in the world and relationships between those objects - complex structures of symbols are needed to capture our knowledge of the world.

Term The fundamental data structure in Prolog is the term which can be a constant, a variable or a structure. Structures represent atomic propositions of predicate calculus and consist of a functor name and a parameter list.

PROGRAMMING LANGUAGES IN ARTIFICIAL INTELLIGENCE (AI) are the major tool for exploring and building computer programs that can be used to simulate intelligent processes such as learning, reasoning and understanding symbolic information in context. Although in the early days of computer language design the primarily use of computers was for performing calculations with numbers, it was also found out quite soon that strings of bits could represent not only numbers but also features of arbitrary objects. Operations on such features or SYMBOLS could be used to represent rules for creating, relating or manipulating symbols. This led to the notion of *symbolic* computation as an appropriate means for defining algorithms that processed information of any type, and thus could be used for simulating human intelligence. Soon it turned out that programming with symbols required a higher level of abstraction than was possible with those programming languages which were designed especially for number processing, e.g., Fortran.

I. AI programming languages

In AI, the automation or programming of all aspects of human cognition is considered from its foundations in cognitive science through approaches to symbolic and sub-symbolic AI, natural language processing, computer vision, and evolutionary or adaptive systems. It is inherent to this very complex problem domain that in the initial phase of programming a specific AI problem, it can only be specified poorly. Only through interactive and incremental refinement does more precise specification become possible. This is also due to the fact that typical AI problems tend to be very domain specific, therefore heuristic strategies have to be developed empirically through generate–and–test approaches (also known as *rapid proto–typing*). In this way, AI programming notably differs from standard software engineering approaches where programming usually starts from a detailed formal specification. In AI programming, the implementation effort is actually part of the problem specification process.

Due to the "fuzzy" nature of many AI problems, AI programming benefits considerably if the programming language frees the AI programmer from the constraints of too many technical con-

structions (e.g., low-level construction of new data types, manual allocation of memory). Rather, a declarative programming style is more convenient using built-in high-level data structures (e.g., lists or trees) and operations (e.g., pattern matching) so that symbolic computation is supported on a much more abstract level than would be possible with standard imperative languages, such as Fortran, Pascal or C. Of course, this sort of abstraction does not come for free, since compilation of AI programs on standard von Neumann computers cannot be done as efficiently as for imperative languages. However, once a certain AI problem is understood (at least partially), it is possible to re–formulate it in form of detailed specifications as the basis for re–implementation using an imperative language.

From the requirements of symbolic computation and AI programming, two new basic programming paradigms emerged as alternatives to the imperative style: the *functional* and the *logical* programming style. Both are based on mathematical formalisms, namely *recursive function theory* and *formal logic*. The first practical and still most widely used AI programming language is the functional language Lisp developed by John McCarthy in the late 1950s. Lisp is based on mathematical function theory and the lambda abstraction. A number of important and influential AI applications have been written in Lisp so we will describe this programming language in some detail in this article. During the early 1970s, a new programming paradigm appeared, namely logic programming on the basis of predicate calculus. The first and still most important logic programming language is Prolog, developed by Alain Colmerauer, Robert Kowalski and Phillippe Roussel. Problems in Prolog are stated as facts, axioms and logical rules for deducing new facts. Prolog is mathematically founded on predicate calculus and the theoretical results obtained in the area of automatic theorem proving in the late 1960s.

II. Functional programming

A mathematical function is a mapping of one set (called the domain) to another (called the range). A function definition is the description of this mapping either explicitly by enumeration or

implicitly by an expression. The definition of a function is specified by a function name followed by a list of parameters in parenthesis, followed by the expression describing the mapping, e.g., CUBE(X) $\equiv$ X $\star$ X $\star$ X, where X is a real number. Alonso Church introduced the notation of *nameless* functions using the Lambda notation. A lambda expression specifies the parameters and the mapping of a function using the λ operator, e.g., λ(X)X $\star$ X $\star$ X. It is the function itself, so the notation of applying the example nameless function to a certain argument is, for example, (λ(X)X $\star$ X $\star$ X)(4).

Programming in a functional language consists of building function definitions and using the computer to evaluate expressions, i.e. function application with concrete arguments. The major programming task is then to construct a function for a specific problem by combining previously defined functions according to mathematical principles. The main task of the computer is to evaluate function calls and to print the resulting function values. This way the computer is used like an ordinary pocket computer, of course at a much more flexible and powerful level. A characteristic feature of functional programming is that if an expression possesses a well-defined value, then the order in which the computer performs the evaluation does not affect the result of the evaluation. Thus, the result of the evaluation of an expression is just its value. This means that in a pure functional language no side–effects exist. Side–effects are connected to variables that model memory locations. Thus, in a pure functional programming language no variables exists in the sense of imperative languages. The major control flow methods are recursion and conditional expressions. This is quite different from imperative languages, in which the basic means for control are sequencing and iteration. Functional programming also supports the specification of higher–order functions. A higher–order function is a function definition which allows functions as arguments or returns a function as its value.

All these aspects together, but especially the latter are major sources of the benefits of functional programming style in contrast to imperative programming style, viz. that functional programming provides a high-level degree of modularity. When defining a problem by deviding it into a set of sub-problems, a major issue concerns the ways in which one can glue the (sub–)

solutions together. Therefore, to increase ones ability to modularise a problem conceptually, one must provide new kinds of glue in the programming language — a major strength of functional programming.

III. Functional programming in Lisp

Lisp is the first functional programming language: It was invented to support symbolic computation using linked lists as the central data structure (Lisp stands for *List processor*). John McCarthy noticed that the control flow methods of mathematical functions – recursion and conditionals – are appropriate theoretical means for performing symbolic computations. Furthermore, the notions of functional abstraction and functional application defined in lambda calculus provide for the necessary high-level abstraction required for specifying AI problems.

Lisp was invented by McCarthy in 1958 and a first version of a Lisp programming environment was available in 1960 consisting of an interpreter, a compiler, and mechanisms for dynamic memory allocation and deallocation (known as *garbage collection*). A year later the first language standard was introduced, named Lisp 1.5. Since then a number of Lisp dialects and programming environments have been developed, e.g., MacLisp, FranzLisp, InterLisp, Common Lisp and Scheme. Although they differ in some specific details, their syntactic and semantic core is basically the same. It is this core which we wish to introduce in this overview. The most widely used Lisp dialects are Common Lisp and Scheme. In this article we have chosen Common Lisp to present the various aspects of Lisp with concrete examples. The examples are however easily adaptable to other Lisp dialects.

A. The syntax and semantics of Lisp

1. Symbolic expressions

The syntactic elements of Lisp are called symbolic expressions (also known as *s–expressions*). Both data and functions (i.e., Lisp programs) are represented as s–expressions which can be either *atoms*

or *lists*.

Atoms are word–like objects consisting of sequences of characters. Atoms can further be divided into different types depending on the kind of characters which are allowed to form an atom. The main subtypes are:

Numbers: 1 2 3 4 -4 3.14159265358979 -7.5 6.02E+23

Symbols: Symbol Sym23 another-one t false NIL BLUE

Strings: "This is a string" "977?" "setq" "He said: \" I'm here.\" "

Note that although a specific symbol like BLUE is used because it has a certain meaning for the programmer, for Lisp it is just a sequence of letters or just a symbol.

Lists are clause–like objects. A list consists of an open left round bracket (followed by an arbitrary number of *list elements* separated by blanks and a closing right round bracket). Each list element can be either an atom or a list. Here are some examples of lists:

```
(This is a list)  ((this) ((too))) () (((((((())))))))


(a b c d) (john mary tom)  (loves john ?X)


(* (+ 3 4) 8)  (append (a b c) (1 2 3))


(defun member (elem list)
    (if (eq elem (first list)) T
        (member elem (rest list))))
```

Note that in most examples the list elements are lists themselves. Such lists are also called *nested lists*. There is no restriction regarding the depth of the nesting. The examples also illustrate one of the strengths of Lisp: very complex representations of objects can be written with minimal effort. The only thing to watch for, is the right number of left and right round brackets. It is important to note that the meaning associated with a particular list representation or atom is not

"entered" into the list representation. This means that all s–expressions (as described above) are syntactically correct Lisp programs, but they are not necessarily semantically correct programs.

2. Semantics

The core of every Lisp programming system is the *interpreter* whose task is to compute a value for a given s–expression. This process is also called *evaluation*. The result or value of an s–expression is also an s–expression which is returned after the evaluation is completed. Note that this means that Lisp actually has operational semantics, but with a precise mathematical definition derived from recursive function theory.

Read-eval-print loop How can the Lisp interpreter be activated and used for evaluating s–expressions, and therefore for running real Lisp programs? The Lisp interpreter is actually also defined as a function usually named EVAL and part of any Lisp programming environment (such a function is called a *built-in* function). It is embedded into a Lisp system by means of the so–called *read-eval-print loop*, where an s–expression entered by the user is first *read* into the Lisp system (READ is also a built-in function). Then the Lisp interpreter is called via the call of EVAL to evaluate the s–expression and the resulting s–expression is returned by printing it to the user's device (not surprisingly calling a built-in function PRINT). When the Lisp system is started on the computer, this read-eval-print loop is automatically started and signaled to the user by means of a specific Lisp prompt sign starting a new line. In this article we will use the question mark ? as the Lisp prompt. For example:

```
? (+ 3 4)
7
```

means that the Lisp system has been started and the read-eval-print loop is activated. The s–expression (+ 3 4) entered by a Lisp hacker is interpreted by the Lisp interpreter as a call of the addition function and prints the resulting s–expression 7 in the beginning of a new line.

Evaluation The Lisp interpreter operates according to the following three rules:

1. Identity: A number, a string or the symbols T and NIL evaluate to themselves. This means that the value of the number 3 is 3 and the value of "house" is "house". The symbol T returns T which is interpreted to denote the **true** value, and NIL returns NIL meaning **false**.

2. Symbols: The evaluation of a symbol returns the s–expression *associated* to it (how this is done will be shown below). Thus, if we assume that the symbol *NAMES* is associated to the list (JOHN MARY TOM) then evaluation of *NAMES* yields that list. If the symbol COLOR is associated with the symbol GREEN then GREEN is returned as the value of COLOR. In other words, symbols are interpreted as variables *bound* to some values.

3. Lists: Every list is interpreted as a function call. The first element of the list denotes the function which has to be applied to the remaining (potentially empty) elements representing the *arguments* of that function. The fact that a function is specified before its arguments is also known as *prefix notation*. It has the advantage that functions can simply be specified and used with an arbitrary number of arguments. The empty list () has the s–expression NIL as its value. Note that this means that the symbol NIL actually has two meanings: one representing the logical false value and one representing the empty list. Although this might seem a bit odd, in Lisp there is actually no problem in identifying which sense of NIL is used.

In general, the arguments are evaluated before the function is applied to the values of the arguments. The order of evaluation of a sequence of arguments is left to right. An argument may represent an atom or a list, in which case it is also interpreted as a function call and the Lisp interpreter is called for evaluating it. For example, consider the following evaluation of a function in the Lisp system:

```
? (MAX 4 (MIN 9 8) 7 5)

8
```

Here the arguments are 4, (MIN 9 8), 7 and 5, which are evaluated in that order before the function with the name MAX is applied on the resulting argument values. The first argument 4 is a number so its value is 4. The second argument (MIN 9 8) is itself a function call. Thus,

before the third argument can be called, (MIN 9 8) has to be evaluated by the Lisp interpreter. Note that because we have to apply the Lisp interpreter for some argument *during* the evaluation of the whole function call, it is also said that the Lisp interpreter is called *recursively*. The Lisp interpreter applies the same steps, so the first argument 9 is evaluated before the second argument 8. Application of the function MIN then yields 8, assuming that the function is meant to compute the minimum of a set of integers. For the *outermost* function MAX, this means that its second argument evaluates to 8. Next the arguments 7 and 5 are evaluated which yields the values 7 and 5. Now, the maximum function named MAX can be evaluated which returns 8. This final value is then the value of whole function call.

Quoting Since the Lisp interpreter always tries to identify a symbol's value or interprets a list as a function call, how can we actually treat symbols and lists as data? For example, if we enter the list (PETER WALKS HOME), then the Lisp interpreter will immediately return an error saying something like ERROR: UNKNOWN FUNCTION PETER (the Lisp interpreter should be clever enough to first check whether a function definition exists for the specified function name, before it tries to evaluate each argument). Or if we simply enter HOUSE, then the Lisp interpreter will terminate with an error like ERROR: NO VALUE BOUND TO HOUSE. The solution to this problem is quite easy: since every first element of a list is interpreted as a function name, each Lisp system comes with a built-in function QUOTE which expects one s–expression as argument and returns this expression without evaluating it. For example, for the list (QUOTE (PETER WALKS HOME)) QUOTE simply returns the value (PETER WALKS HOME), and for (QUOTE HOUSE) it returns HOUSE. Since the function QUOTE is used very often, it can also be expressed by the special character '. Therefore, for the examples above we can equivalently specify '(PETER WALKS HOME) and 'HOUSE.

Programs as data Note that QUOTE also enables us to treat function calls as data by specifying for example (QUOTE (MAX 4 (MIN 9 8) 7 5)) or '(MAX 4 (MIN 9 8) 7 5). We already said that the Lisp interpreter is also a built-in unary function named EVAL. It explicitly forces its argument to be evaluated according to the rules mentioned above. In some sense, it can be seen as the

opposite function to QUOTE. Thus to explicitly require that a list specified as data to the Lisp system should be interpreted as a function call, we can specify (EVAL '(MAX 4 (MIN 9 8) 7 5)) which returns the value 8 as described above. In the same way, specifying (EVAL '(PETER WALKS HOME)) will cause an Lisp error because Lisp tries to call a function PETER.

The main advantage of being able to treat programs as data is that we can define Lisp programs (functions) which are able to construct or generate programs such that they first build the corresponding list representation and then explicitly call the Lisp interpreter using EVAL in order to evaluate the just created list as a function. It is not surprising, that due to this characteristic Lisp is still the dominant programming language in the AI area of genetic programming.

Assigning values to symbols When programming real-life practical programs, one often needs to store values computed by some program to a variable to avoid costly re-computation of that value if it is needed in another program at some later time. In a purely functional version of Lisp, the value of a function only depends on the function definition and on the value of the arguments in the call. In order to make Lisp a practical language (practical at least in the sense that it can run efficiently on von Neumann computers), we need a way to assign values to symbols.

Common Lisp comes with a built-in function called SETQ. SETQ expects two arguments: the symbol (called the variable) to which a value is bound and an s–expression which has to provide the value. The Lisp interpreter treats the evaluation of SETQ in a special way, such that it explicitly supresses evaluation of SETQ's first argument (the variable), but rather binds the value of SETQ's second argument to the variable (to understand how Lisp internally binds a value to a symbol would require too many technical details which we cannot go into in this short introduction). The value of the second argument of SETQ is returned as the value of SETQ. Here are some examples:

```
? COLOR

ERROR: UNBOUND SYMBOL COLOR

? (SETQ COLOR 'GREEN)

GREEN
```

? (SETQ MAX (MAX 3 2.5 1))

3

Note that SETQ actually changes the status of the Lisp interpreter because the next time the same variable is used, it has a value and therefore the Lisp interpreter will be able to return it. If this effect did not occur then the Lisp interpreter would signal an error because that symbol would not be bound (cf. step 2 of the Lisp interpreter). Thus, it is also said that SETQ produces a *side-effect* because it dynamically changes the status of the Lisp interpreter. When making use of SETQ one should, however, be aware of the fact that one is leaving the proper path of semantics of pure Lisp. SETQ should therefore be used with great care!

B. The List data type

Programming in Lisp actually means defining functions that operate on lists, e.g., create, traverse, copy, modify and delete lists. Since this is central to Lisp, every Lisp system comes with a basic set of primitive built-in functions that efficiently support the main list operations. We will briefly introduce the most important ones now.

Type predicate Firstly, we have to know whether a current s–expression is a list or not (i.e., an atom). This job is accomplished by the function LISTP which expects any s–expression EXPR as an argument and returns the symbol T if EXPR is a list and NIL otherwise. Examples are (we will use the right arrow $\Longrightarrow$ for pointing to the result of a function call):

(LISTP '(1 2 3)) $\Longrightarrow$ T

(LISTP '()) $\Longrightarrow$ T

(LISTP '3) $\Longrightarrow$ NIL

Selection of list elements Two basic functions exist for accessing the elements of a list: CAR and CDR. Both expect a list as their argument. The function CAR returns the first element in the

list or NIL if the empty list is the argument, and CDR returns the same list from which the first element has been removed or NIL if the empty list was the argument. Examples:

(CAR '(A B C)) ⟹ A (CDR '(A B C)) ⟹ (A B)

(CAR '()) ⟹ NIL (CDR '(A)) ⟹ NIL

(CAR '((A B) C)) ⟹ (A B) (CAR '((A B) C)) ⟹ C

By means of a sequence of CAR and CDR function calls, it is possible to traverse a list from left to right and from outer to inner list elements. For example, during evaluation of

(CAR (CDR '(SEE THE QUOTE)))

the Lisp interpreter will first evaluate the expression

(CDR '(SEE THE QUOTE))

which returns the list (THE QUOTE), which is then passed to the function CAR which returns the symbol THE. Here, are some further examples:

(CAR (CDR (CDR '(SEE THE QUOTE)))) ⟹ QUOTE

(CAR (CDR (CDR (CDR '(SEE THE QUOTE))))) ⟹ NIL

(CAR (CAR '(SEE THE QUOTE))) ⟹ ???

What will happen during evaluation of the last example? Evaluation of (CAR '(SEE THE QUOTE)) returns the symbol SEE. This is then passed as argument to the outer call of CAR. However, CAR expects a list as argument, so the Lisp interpreter will immediately stop further evaluation with an error like ERROR: ATTEMPT TO TAKE THE CAR OF SEE WHICH IS NOT LISTP.

A short historical note: the names CAR and CDR are old–fashioned because they were chosen in the first version of Lisp on the basis of the machine code operation set of the computer on which it was implemented (CAR stands for "contents of address register" and CDR stands for "contents of decrement register). In order to write more readable Lisp code, Common Lisp comes with two equivalent functions, FIRST and REST. We have used the older names here as it enables reading and understanding of older AI Lisp code.

Construction of lists Analogously to CAR and CDR, a primitive function CONS exists which is used to construct a list. CONS expects two s–expressions and inserts the first one as a new element in front of the second one. Consider the following examples:

(CONS 'A '(B C)) ⟹ (A B C)

(CONS '(A D) '(B C)) ⟹ ((A D) B C)

(CONS (FIRST '(1 2 3)) (REST '(1 2 3))) ⟹ (1 2 3)

In principle, CONS together with the empty list suffice to build very complex lists, for example:

(CONS 'A (CONS 'B (CONS 'C '()))) ⟹ (A B C)

(CONS 'A (CONS (CONS 'B (CONS 'C '())) (CONS 'D '()))) ⟹ (A (B C) D)

However, since this is quite cumbersome work, most Lisp systems come with a number of more advanced built-in list functions. For example, the function LIST constructs a list from an arbitrary number of s–expressions, and the function APPEND constructs a new list through concatenation of its arguments which must be lists. EQUAL is a function which returns T if two lists have the same elements in the same order, otherwise NIL. Examples:

(LIST 'A 'B 'C) ⟹ (A B C) (LIST (LIST 1) 2 (LIST 1 2 3)) ⟹ ((1) 2 (1 2 3))

(APPEND '(1) (LIST 2)) ⟹ (1 2) (APPEND '(1 2) NIL '(3 4)) ⟹ (1 2 3 4)

(EQUAL '(A B C) '(A B C)) ⟹T (EQUAL '(A B C) '(A C B)) ⟹NIL

C. Defining new functions

Programming in Lisp is done by defining new functions. In principle this means: specifying lists in a certain syntactic way. Analogously to the function SETQ which is treated in a special way by the Lisp interpreter, their is a special function DEFUN which is used by the Lisp interpreter to create new function objects. DEFUN expects as its arguments a symbol denoting the *function name*, a (possibly empty) list of *parameters* for the new function and an arbitrary number of s–expressions defining the *body* of the new function. Here is the definition of a simple function named MY-SUM which expects two arguments from which it will construct the sum using the built-in function +:

```
(DEFUN MY-SUM (X Y)
  (+ X Y))
```

This expression can be entered into the Lisp system in the same way as a function call. Evaluation of a function definition returns the function name as value, but will create a function object as side-effect and adds it to the set of function definitions known by the Lisp system when it is started (which is at least the set of built-in functions). Note that in this example, the body consists only of one s–expression. However, the body might consist of an arbitrary sequence of s–expressions. The value of the last s–expression of the body determines the value of the function. This means that all other elements of the body are actually irrelevant, unless they produce intended side–effects.

The parameter list of the new function MY-SUM tells us that MY-SUM expects exactly two s–expression as arguments when it is called. Therefore, if you enter (MY-SUM 3 5) into the Lisp system, the Lisp interpreter will be able to find a definition for the specified function name, and then process the given arguments from left to right. When doing so, it binds the value of each argument to the corresponding parameter specified in the parameter list of the function definition. In our example, this means that the value of the first argument 3 (which is also 3 since 3 is a number which evaluates to itself) is bound to the parameter X. Next, the value of the second argument 5 is bound to the parameter Y. Because the value of an argument is bound to a parameter, this mechanism is also called CALL BY VALUE. After having found a value for all parameters, the Lisp interpreter is able to evaluate the body of the function. In our example, this means that (+ 3 5) will be called. The result of the call is 8 which is returned as result of the call (MY-SUM 3 5). After the function call is completed, the *temporary* binding of the parameters X and Y are deleted.

Once a new function definition has been entered into the Lisp system, it can be used as part of the definition of new functions in the same way as built-in functions are used, as shown in the following example:

```
(DEFUN DOUBLE-SUM (X Y)
```

```
(+ (MY-SUM X Y) (MY-SUM X Y)))
```

which will double the sum of its arguments by calling MY-SUM twice.

Here is another example of a function definition, demonstrating the use of multiple s–expressions in the function body:

```
(DEFUN HELLO-WORLD () (PRINT "HELLO WORLD!") 'DONE)
```

This function definition has no parameter because the parameter list is empty. Thus, when calling (HELLO-WORLD), the Lisp interpreter will immediately evaluate (PRINT "HELLO WORLD!") prints the string "Hello World!" on your display as a side–effect. Next, it will evaluate the symbol 'DONE which returns DONE as result of the function call.

D. Defining control structures

Although it is now possible to define new functions by combining built-in and user-defined functions, programming in Lisp would be very tedious if it were not possible to control the flow of information by means of conditional branches perhaps iterated many times until a stop criterion is fulfilled. Lisp branching is based on function evaluation: control functions perform tests on actual s–expressions and, depending on the results, selectively evaluate alternative s–expressions.

The fundamental function for the specification of conditional assertions in Lisp is COND. COND accepts an arbitrary number of arguments. Each argument represents one possible branch and is represented as a list where the first element is a test and the remaining elements are actions (s–expressions) which are evaluated if the test is fulfilled. The value of the last action is returned as the value of that alternative. All possible arguments of COND (i.e., branches) are evaluated from left to right until the first branch is positively tested. In that case the value of that branch is the value of the whole COND function. This sounds more complicated than it actually is. Let us consider the following function VERBALIZE–PROP which verbalizes a probability value expressed as a real number:

```
(DEFUN VERBALIZE–PROP (PROB-VALUE)
```

```
(COND ((> PROB–VALUE 0.75) 'VERY-PROBABLE)
      ((> PROB–VALUE 0.5) 'PROBABLE)
      ((> PROB–VALUE 0.25) 'IMPROBABLE)
      (T 'VERY-IMPROBABLE)))
```

When calling (VERBALIZE–PROP 0.33), the actual value of the argument is bound to the parameter PROB-VALUE. Then COND is evaluated with that binding. The first expression to be evaluated is ((> PROB-VALUE 0.75) 'VERY-PROBABLE). > is a built-in predicate which tests whether the first argument is greater then the second one. Since PROB-VALUE is 0.33, > evaluates to NIL which means that the test is not fulfilled. Therefore, evaluation of this alternative branch is terminated immediately, and the next alternative ((> PROB–VALUE 0.5) 'PROBABLE) is evaluated. Here the test function also returns NIL, so the evaluation is terminated, too. Next ((> PROB–VALUE 0.25) 'IMPROBABLE) is evaluated. Applying the test function now returns T which means that the test is fulfilled. Then all actions of this positively tested branch are evaluated and the value of the last action is returned as the value of COND. In our example, only the action 'IMPROBABLE has been specified which returns the value IMPROBABLE. Since this defines the value of COND, and because the COND expression is the only expression of the body of the function VERBALIZE–PROP, the result of the function call (VERBALIZE–PROP 0.33) is IMPROBABLE. Note that if we enter (VERBALIZE–PROP 0.1) the returned value is VERY-IMPROBABLE because the test of the third alternative will also fail and the branch (T 'VERY-IMPROBABLE) has to be evaluated. In this case, the symbol T is used as test which always returns T, so the value of this alternative is VERY-IMPROBABLE.

E. Recursive function definitions

The second central device for defining control flow in Lisp are *recursive function definitions*. A function which partially uses its definition as part of its own definition is called *recursive*. Thus seen, a recursive definition is one in which a problem is decomposed into smaller units until

no further decomposition is possible. Then these smaller units are solved using known function definitions and the sum of the corresponding solutions form the solution of the complete program. Recursion is a natural control regime for data structures which have no definite size, such as lists, trees, and graphs. Therefore, it is particularly appropriate for problems in which a space of states has to be searched for candidate solutions.

Lisp was the first practical programming language that systematically supported the definition of recursive definitions. We will use two small examples to demonstrate recursion in Lisp. The first example is used to determine the length of an arbitrarily long list. The length of a list corresponds to the number of its elements. Its recursive function is as follows:

```
(DEFUN LENGTH (LIST)
  (COND ((NULL LIST) 0)
        (T (+ 1 (LENGTH (CDR LIST))))))
```

When defining a recursive definition, we have to identify the base cases, i.e., those units which cannot be decomposed any further. Our problem size is the list. The smallest problem size of a list is the empty list. Thus, the first thing we have to specify is a test for identifying the empty list and to define what the length of the empty list should be. The built–in function NULL tests whether a list is empty in which case it returns T. Since the empty list is a list with no elements, we define that the length of the empty list is 0. The next thing to be done is to decompose the problem size into smaller units, so that the same problem can be applied to smaller units. Decomposition of a list can be done by using the functions CAR and CDR, which means that we have to specify what is to be done with the first element of a list and its rest until the empty list is found. Since we already have identified the empty list as the base case, we can assume that decomposition will be performed on a list containing at least one element. Thus, every time we are able to apply CDR to get the rest of a list, we have found one additional element which should be used to increase the number of the already identified list elements by 1. Making use of this function definition, (LENGTH '()) will immediately return 0, and if we call (LENGTH '(A B C)), the result will be 3,

because three recursive calls have to be performed until the empty list can be determined.

As a second example, we consider the recursive definition of MEMBER, a function which tests whether a given element occurs in a given list. If the element is indeed found in the list, it returns the sublist which starts with the first occurrence of the found element. If the element cannot be found, NIL is returned. Example calls are:

(MEMBER 'B '(A F B D E B C)) $\Longrightarrow$ (B D E B C)

(MEMBER 'K '(A F B D E B C)) $\Longrightarrow$ NIL

Similarly to the recursive definition of LENGTH, we use the empty list as the base case. For MEMBER, the empty list means that the element in question is not found in the list. Thus, we have to decompose a list until the element in question is found or the empty list is determined. Decomposition is done using CAR and CDR. CAR is used to extract the first element of a list which can be used to check whether it is equal to the element in question, in which case we can directly stop further processing. If it is not equal, then we should apply the MEMBER function on the remaining elements until the empty list is determined. Thus, MEMBER can be defined as follows:

```
(DEFUN MEMBER (ELEM LIST)
  (COND ((NULL LIST) NIL)
        ((EQUAL ELEM (CAR LIST)) LIST)
        (T (MEMBER ELEM (CDR LIST)))))
```

F. Higher-order functions

In Lisp, functions can be used as arguments. A function that can take functions as its arguments is called a *higher–order function*. There are a lot of problems where one has to traverse a list (or a tree or a graph) such that a certain function has to be applied to each list element. For example, a *filter* is a function that applies a test to the list elements, removing those that fail the test. *Maps* are functions which apply the same function on each element of a list returning a list of the results.

High–order function definitions can be used for defining generic list traversal functions such that they abstract away from the specific function used to process the list elements.

In order to support high–order definitions, their is a special function FUNCALL which takes as its arguments a function and a series of arguments and applies that function to those arguments. As an example of the use of FUNCALL, we will define a generic function FILTER which may be called in this way:

(FILTER '(1 3 -9 -5 6 -3) #'PLUSP) $\Longrightarrow$ (1 3 6)

PLUSP is a built–in function which checks whether a given number is positive or not. If so, it returns that number, otherwise NIL is returned. The special symbol # is used to tell the Lisp interpreter that the argument value denotes a function object. The definition of FILTER is as follows:

```
(DEFUN FILTER (LIST TEST)
  (COND ((NULL LIST) LIST)
        ((FUNCALL TEST (CAR LIST))
           (CONS (CAR LIST) (FILTER (CDR LIST) TEST)))
        (T (FILTER (CDR LIST) TEST))))
```

If the list is empty, then it is simply returned. Otherwise, the test function is applied to the first element of the list. If the test function succeeds, CONS is used to construct a result list using this element and all elements that are determined during the recursive call of FILTER using the CDR of the list and the test function. If the test fails for the first element, this element is simply skipped by recursively applying FILTER on the remaining elements, i.e., this element will not be part of the result list. The filter function can be used for many different test functions, e.g.,

(FILTER '(1 3 A B 6 C 4) #'NUMBERP) $\Longrightarrow$ (1 3 6 4)

(FILTER '(1 2 3 4 5 6) #'EVEN) $\Longrightarrow$ (2 4 6)

As another example of a higher–order function definition, we will define a simple mapping function, which applies a function to all elements of a list returning a list of all values. If we call the function MY-MAP, then the definition looks like this:

```
(DEFUN MY-MAP (FN LIST)
  (COND ((NULL LIST) LIST)
        (T (CONS (FUNCALL FN (CAR LIST)) (MY-MAP FN (CDR LIST))))))
```

If a function DOUBLE exists which just doubles a number, then a possible call of MY-MAP could be:

```
(MY-MAP #'DOUBLE '(1 2 3 4)) ⟹ (2 4 6 8)
```

Often it is the case that a function should only be used once. Thus, it would be quite convenient if we could provide the definition of a function directly as an argument of a mapping function. To do this, Lisp supports the definition of LAMBDA–expressions. We have already informally introduced the notation of LAMBDA–expressions in section II as a means for defining nameless or *anonymous* functions. In Lisp LAMBDA–expressions are defined using the special form LAMBDA. The general form of a LAMBDA–expression is:

```
(LAMBDA (parameter ...) body ...)
```

A LAMBDA–expression allows us to separate a function definition from a function name. LAMBDA–expressions can be used in place of a function name in a FUNCALL, e.g., the LAMBDA–expression for our function DOUBLE may be:

```
(LAMBDA (X) (+ X X))
```

For example, the above function call of MY-MAP can be re–stated using the LAMBDA–expression as follows:

```
(MY-MAP #'(LAMBDA (X) (+ X X)) '(1 2 3 4) ⟹ (2 4 6 8)
```

A LAMBDA–expression returns a function object which is not bound to a function name. In the definition of MY-MAP we used the parameter FN as a function name variable. When evaluating the lambda form, the Lisp interpreter will bind the function object to that function name variable. In this way, a function parameter is used a dynamic function name. The # symbol is necessary to tell Lisp that it should not only bind a function object but should also maintain the bindings of the local and global values associated to the function object. This would not be possible by simply using the QUOTE operator alone (unfortunately, further details cannot be given here due to the space constraints) .

G. Other functional programming languages than Lisp

We have introduced Lisp as the main representative functional programming language (especially the widely used dialect Common Lisp), because it is still a widely used programming language for a number of Artificial Intelligence problems, like Natural Language Understanding, Information Extraction, Machine Learning, AI planning, or Genetic Programming. Beside Lisp a number of alternative functional programming languages have been developed. We will briefly mention two well–known members, viz. ML and Haskell.

ML which stands for Meta-Language is a static-scoped functional programming language. The main differences to Lisp is its syntax (which is more similar to that of Pascal), and a strict polymorphic type system (i.e., using strong types and type inference, which means that variables need not be declared). The type of each declared variable and expression can be determined at compile time. ML supports the definition of abstract data types, as demonstrated by the following example:

```
DATATYPE TREE  =  L OF INT
               |  INT * TREE * TREE;
```

which can be read as "every binary tree is either a leaf containing an integer or it is a node containing an integer and two trees (the subtrees)". An example of a recursive function definition applied on a tree data structure is shown in the next example:

```
fun depth(L _) = 1
  | depth(N(i,l,r)) =
      1 + max(depth l, depth r);
```

The function DEPTH maps trees to integers. The depth of a leaf is 1 and the depth of any other tree is 1 plus the maximum of the depths of the left and right subtrees.

HASKELL is similar to ML: it uses a similar syntax, it is also static scoped, and makes use of the same type inferencing method. It differs from ML in that it is purely functional. This means that it allows no side effects and includes no imperative features of any kind, basically because it has no variables and no assignment statements. Furthermore it uses a *lazy* evaluation technique, in which no subexpression is evaluated until its value is known to be required.

Lists are a commonly used data structure in Haskell. For example, [1,2,3] is the list of three integers 1,2, and 3. The list [1,2,3] in Haskell is actually shorthand for the list 1:(2:(3:[])), where [] is the empty list and : is the infix operator that adds its first argument to the front of its second argument (a list). As an example of a user-defined function that operates on lists, consider the problem of counting the number of elements in a list by defining the function LENGTH:

```
length :: [a] -> Integer
length [] = 0
length (x:xs) = 1 + length xs
```

which can be read as "The length of the empty list is 0, and the length of a list whose first element is x and remainder is xs is 1 plus the length of xs". In HASKELL, function invocation is guided by *pattern matching*. For example, the left-hand sides of the equations contain patterns such as [] and x:xs. In a function application these patterns are matched against actual parameters ([] only matches the empty list, and x:xs will successfully match any list with at least one element, binding x to the first element and xs to the rest of the list). If the match succeeds, the right-hand side is evaluated and returned as the result of the application. If it fails, the next equation is tried, and if all equations fail, an error results.

This ends our short "tour de Lisp". We were only able to discuss the most important aspects of Lisp. Readers interested in more specific details should consult at least one of the books mentioned at the end of this chapter. The rest of this chapter will now be used to introduce another programming paradigm widely used in AI programming, namely Prolog.

IV. Logical programming in Prolog

In the 1970s an alternative paradigm for symbolic computation and AI programming arose from the success in the area of automatic theorem proving. Notably, the resolution proof procedure developed by Robinson (1965) showed that formal logic, in particular predicate calculus, could be used as a notation for defining algorithms and therefore, for performing symbolic computations. In the early 1970s, Prolog (an acronym for *Programming in Logic*), the first logical based programming language appeared. It was developed by Alain Colmerauer, Robert Kowalski and Phillippe Roussel. Basically, Prolog consists of a method for specifying predicate calculus propositions and a restricted form of resolution. Programming in Prolog consists of the specification of *facts* about objects and their relationships, and *rules* specifying their logical relationships. Prolog programs are *declarative* collections of statements about a problem because they do not specify how a result is to be computed but rather define *what* the logical structure of a result should be. This is quite different from imperative and even functional programming, in which the focus is on defining *how* a result is to be computed. Using Prolog, programming can be done at a very abstract level quite close to the formal specification of a problem. Prolog is still the most important logical programming language. There are a number of commercial programming systems on the market which include modern programming modules, i.e., compiler, debugger and visualization tools. Prolog has been used successfully in a number of AI areas such as expert systems and natural language processing, but also in such areas as relational database management systems or in education.

A very simple Prolog program Here is a very simple Prolog program consisting of two facts and one rule:

```
scientist(gödel).

scientist(einstein).

logician(X) :- scientist(X).
```

The first two statements can be paraphrased as "Gödel is a scientist" and "Einstein is a scientist". The rule statement says "X is a logician if X is a scientist". In order to test this program, we have to specify query expressions (or theorems) which Prolog tries to answer (or to prove) using the specified program. One possible query is:

```
?- scientist(gödel).
```

which can be verbalized as "Is Gödel a scientist?". Prolog, by applying its built-in proof procedure, will respond with "yes" because a fact may be found which exactly matches the query. Another possible query verbalizing the question "Who is a scientist?" and expressed in Prolog as:

```
?- scientist(X).
```

will yield the Prolog answer "X = gödel, X = einstein". In this case Prolog not only answers "yes" but will return all bindings of the variable X which it finds during the successful proof of the query. As a further example, we might also query "Who is a logician?" using the following Prolog query:

```
?- logician(X).
```

Proving this query will yield the same set of facts because of the specified rule. Finally, we might also specify the following query:

```
?- logician(mickey-mouse).
```

In this case Prolog will respond with "no". Although the rule says that someone is a logician if she is also a scientist, Prolog does not find a fact saying that Mickey Mouse is a scientist. Note, however, that Prolog can only answer relative to the given program, which actually means "no, I couldn't deduce the fact". This property is also known as the *closed world assumption* or *negation*

as failure. It means that Prolog assumes that all knowledge that is necessary to solve a problem is present in its data base.

Prolog statements Prolog programs consist of a collection of statements also called *clauses* which are used to represent both data and programs. The dot symbol is used to terminate a clause. Clauses are constructed from *terms*. A term can be a *constant* (symbolic names that have to begin with a lowercase letter, like `gödel` or `eInStein`), a *variable* (symbols that begin with a uppercase letter, like `X` or `Scientist`), or a *structure*. Structures represent atomic propositions of predicate calculus and consist of a functor name and a parameter list. Each parameter can be a term, which means that terms are recursive objects. Prolog distinguishes three types of clauses: facts, rules, and queries. A *fact* is represented by a single structure, which is logically interpreted as a simple true proposition. In the simple example program above we already introduced two simple facts. Here are some more examples:

```
male(john).

male(bill).

female(mary).

female(sue).

father(john, mary).

father(bill,john).

mother(sue,mary).
```

Note that these facts have no intrinsic semantics, i.e., the meaning of the functor name `father` is not defined. Applying common sense, we may interpret it as "John is the father of Mary.", for example. However, for Prolog, this meaning does not exist, it is just a symbol.

Rules belong to the next type of clauses. A rule clause consists of two parts, the *head* which is a single term and the *body* which is either a single term or a conjunction. A conjunction is a set of terms separated by the comma symbol. Logically, a rule clause is interpreted as an implication

such that if the elements of the body are all true, then the head element is also true. Therefore, the body of a clause is also denoted as the *if* part and the head as the *then* part of a rule. Here is an example for a set of rule clauses:

```
parent(X,Y) :- mother(X, Y).

parent(X,Y) :- father(X, Y).

grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
```

where the last rule can be read as "X is a grandparent of Z, if X is a parent of Y *and* Y is a parent of Z.". The first two rules say "someone is a parent if it is the father or mother of someone else". The reason we treat the first two rules as a disjunction will become clear when we introduce Prolog's proof procedure. Before doing this, we shall introduce the last type of clause, the *query* clause (also called the *goal* clause). A query is used to activate Prolog's proof procedure. Logically, a query corresponds to an unknown theorem. It has the same form as a fact. In order to tell Prolog that a query has to be proven, the special query operator `?-` is usually written in front of the query. In the simple Prolog program introduced above, we have already seen an informal description of how a query is used by Prolog.

Prolog's inference process consists of two basic components: a search strategy and a unifier. The search strategy is used to search through the fact and rule data base while unification is used for pattern matching and returns the bindings that make an expression true.

The unifier is applied on two terms and tries to combine them both to form a new term. If unification is not possible, then unification is said to have *failed*. If the two terms contain no variables, then unification actually reduces to checking whether the terms are equal. For example, unification of the two terms

`father(john,mary)` and `father(john,mary)`

succeeds, whereas unification of the following term pairs will fail:

`father(X,mary)` and `father(john,sue)`

`sequence(a,b,c)` and `sequence(a,b)`

If a term contains a variable (or more), then the unifier checks whether the variable can be bound with some information from the second term, however, only if the remaining parts of the terms unify. For example, for the following two terms

`father(X,mary)` and `father(john,mary)`

the unifier will bind `X` to `john` because the remaining terms are equal. However, for the following pair:

`father(X,mary)` and `father(john,sue)`

the binding would not make sense, since `mary` and `sue` do not match.

The search strategy is used to traverse the search space spanned by the facts and rules of a Prolog program. Prolog uses a *top-down*, *depth-first* search strategy. What does this mean? The whole process is quite similar to the function evaluation strategy used in Lisp. If a query Q is specified, then it may either match a fact or a rule. In case of a rule R, Prolog first tries to match the head of R, and if it succeeds, it then tries to match all elements from the body of R which are also called *sub–queries*. If the head of R contains variables, then the bindings will be used during the proof of the sub–queries. Since the bindings are only valid for the sub–queries, it is also said that they are *local* to a rule. A sub–query can either be a fact or a rule. If it is a rule, then Prolog's inference process is applied recursively to the body of such sub–query. This makes up the top–down part of the search strategy. The elements of a rule body are applied from left to right, and only if the current element can be proven successfully is the next element tried. This makes up the depth–first strategy. It is possible that for the proof of a sub–query two or more alternative facts or rules are defined. In that case Prolog selects one alternative A and tries to prove it, if necessary by processing sub–queries of A. If A fails, Prolog goes back to the point where it started the proof of A (by removing all bindings that have been assigned during A's test) and tries to prove the next alternative. This process is also called *back–tracking*. In order to clarify the whole strategy, we can consider the following example query (using the example clauses introduced in

the previous paragraph as Prolog's data base):

```
?- grandparent(bill,mary).
```

The only clause that can match this query is the following rule

```
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
```

and unification of the query with the rule's head will return the following bindings: `X = bill`, `Z = mary`. In order to prove the rule, the two elements of the rule body have to be proven from left to right. Note that both rules share variables with the rule head, and therefore the bindings computed during the match of the head with the query are also available for the respective sub-queries. Thus, the first sub–query is actually instantiated as `parent(bill,Y)` and the second sub–query as `parent(Y,mary)`. Now, to prove the first clause, Prolog finds two alternative `parent`–rules. Let us assume that Prolog chooses the first alternative (in order to remember that more than one alternative is possible, Prolog sets a *choice point*)

```
parent(X,Y) :- mother(X, Y).
```

Unification of the sub–query with the rule head is easily possible and will bind the `X` variable to the term `bill`. This *partially* instantiates the single body element as `mother(bill,Y)`. Unfortunately, there are no facts in the data base which validate this sub–query. Because the unification of `mother(bill,Y)` fails, so does the whole rule. Then, Prolog back–tracks to the choice point where it selected the first possible `parent`–rule and chooses the second alternative

```
parent(X,Y) :- father(X, Y).
```

Unification of the (still active) sub–query `parent(bill,Y)` will instantiate `father(bill,Y)`. This time unification is possible, returning the binding `Y = john`. Now the first `parent` sub–query of the `grandparent`–rule has been proven and the actual variables are: `X = bill`, `Y = john`, `Z = mary`. This instantiates the second element of the `grandparent`–rule body to `parent(john,mary)` (note that the `Z` value had already been bound after the `grandparent`–rule was selected). The same strategy is then applied for this sub–query and Prolog will find enough facts to prove it

successfully. Since both body elements of the `grandparent`–rule have been proven to be valid, Prolog concludes that the initial query is also true.

Prolog extensions In order to use Prolog for practical programming, it comes with a number of extensions, e.g., list data structures, operators for explicitly controlling the traversal of the search space by a Prolog program (namely the `cut` operator) and routines for IO interfaces, tracing and debugging. We cannot describe all these extensions in the context of this short review. We will only briefly show how lists can be used in Prolog.

Prolog supports lists as a basic data structure using conventional syntax. The list elements are separated by commas. The whole list is delimited by square brackets. A list element can be an arbitrary term or a list itself. Thus, it is quite similar to the list structures in Lisp. Here is an example of a Prolog list:

```
[john, mary, bill]
```

The empty list is represented as [ ]. In order to be able to create or traverse lists, Prolog provides a special construction for explicitly denoting the head and tail of a list. `[X | Y]` is a list consisting of a head `X` and a tail `Y`. For example, the above list could also be specified as

```
[john | mary, bill]
```

We will use the member predicate as an example for how lists are treated in Prolog. This predicate will determine whether a given element occurs in a given list. Using the above notation, an element is in a list if it is the head of that list or if it occurs somewhere in the tail of the list. Using this informal definition of the member predicate, we can formulate the following Prolog program (the symbol _ denotes an *anonymous* variable, used to tell Prolog that it does not matter which value the unifier binds to it)

```
member(Element,[Element | _]).

member(Element,[_ | List]) :- member(Element,List).
```

Assuming the following query

```
?- member(a, [b,c,a,d]).
```

Prolog will first check whether the head of `[b | c,a,d]` is equal to `a`. This causes the first clause to fail, so the second is tried. This will instantiate the sub–query `member(a, [c,a,d])`, which means that the first list element is simply skipped. Recursively applying `member`, Prolog tries to prove whether the head of `[c | a,d]` is equal to `a` which also fails, leading to a new sub–query `member(a, [a,d])` through instantiation of the second clause. The next recursive step will check the list `[a | d]`. This time, `a` is indeed equal to the head element of this list, so that Prolog will terminate with "yes".

Constraint logic programming (CLP) CLP is a generalization of the (simple) Prolog programming style. In CLP term unification is generalized to constraint solving. In constraint logic programs basic components of a problem are stated as constraints (i.e., the structure of the objects in question) and the problem as a whole is represented by putting the various constraints together by means of rules (basically by means of definite clauses). For example the following definite clause — representing a tiny fraction of a Natural Language grammar like English:

$$
\begin{aligned}
&sign(X_0) \leftarrow \\
&\quad sign(X_1), \\
&\quad sign(X_2), \\
&\quad X_0\ syn\ cat\ \doteq\ s, \\
&\quad X_1\ syn\ cat\ \doteq\ np, \\
&\quad X_2\ syn\ cat\ \doteq\ vp, \\
&\quad X_1\ syn\ agr\ \doteq\ X_2\ syn\ agr
\end{aligned}
$$

expresses that for a linguistic object to be classified as an S(entence) phrase it must be composed of an object classified as an NP (nominal phrase) and by an object classified as a VP (verbal phrase) and the agreement information (e.g., person, case) between NP and VP must be the same. All objects that fulfill at least these constraints are members of S objects. Note that there is no ordering presupposed for NP and VP as is the case for NL grammar–based formalisms

that rely on a context-free backbone. If such a restriction is required additional constraints have to be added to the rule, for instance that substrings have to be combined by concatenation. Since the constraints in the example above only specify necessary conditions for an object of class S, they express partial information. This is very important for knowledge–based reasoning, because in general we have only partial information about the world we want to reason with. Processing of such specifications is then based upon constraint solving and the logic programming paradigm. Because unification is but a special case of constraint solving, constraint logic programs have superior expressive power.

A number of constraint-based logic programming languages (together with high-level user interface and development tools) have been realized, e.g., CHIP or the Oz language, which supports declarative programming, object-oriented programming, constraint programming, and concurrency as part of a coherent whole. Oz is a powerful constraint language with logic variables, finite domains, finite sets, rational trees and record constraints. It goes beyond Horn-clauses to provide a unique and flexible approach to logic programming. Oz distinguishes between directed and undirected styles of declarative logic programming.

V. Other programming approaches

In this chapter, we have already compared AI languages with imperative programming approaches. Object–oriented languages belong to another well–known programming paradigm. In such languages the primary means for specifying problems is to specify abstract data structures also called objects or classes. A class consists of a data structure together with its main operations often called *methods*. An important characteristic is that it is possible to arrange classes in a hierarchy consisting of classes and sub–classes. A sub–class can inherit properties of its super–classes which supports modularity. Popular object–oriented languages are Eiffel, C++ and Java. Common Lisp Object–Oriented System is an extension of Common Lisp. It supports full integration of functional and object–oriented programming. Recently, Java has become quite popular in some areas of AI,

especially for intelligent agent technology, internet search engines or data mining. Java is based on C++ and is the main language for the programming of Internet applications. Language features that makes Java interesting from an AI perspective are its built-in automatic garbage collection and multi-threading mechanism.

With the increase of research in the area of web intelligence a new programming paradigm is emerging, viz. *agent oriented programming*. Agent-oriented programming is a fairly new programming paradigm that supports a societal view of computation. In AOP, objects known as agents interact to achieve individual goals. Agents can exist in a structure as complex as a global internet or one as simple as a module of a common program. Agents can be autonomous entities, deciding their next step without the interference of a user, or they can be controllable, serving as a mediary between the user and another agent. Since agents are viewed as living, evolving software entities, there seems also to emerge a shift from the more language programming point of view towards a more software platform development point of view. Here the emphasis is on system design, development platforms and connectivity. Critical questions are then how the rich number of existing AI resources developed in different languages and platforms can be integrated with other resources making use of modern system development tools like CORBA (Common Object Request Broker Architecture), generic abstract data type and annotation languages like XML, and standardized agent–oriented communication language like KQML (Knowledge Query and Manipulation Language). So the future of AI programming might less be concerned with questions like "what is the best suited programming paradigm?" but will have to find answers for questions like "how can I integrate different programming paradigms under one umbrella?" and "what are the best communication languages for intelligent autonomous software modules?".

VI. Further reading

1. Charniak, E., Riesbeck, C.K., McDermott, D.V. and Meehan, J.R., 1980, Artificial Intelligence Programming, Lawrence Erlbaum Associates, Hillsdale, New Jersey.

2. Clocksin, W.F. and Mellish, C.S, 1987, Programming in Prolog, Springer, Berlin, Germany.

3. Keene, S.E., 1988, Object–Oriented Programming in Common Lisp, Addison–Wesley, Reading, Massachusetts.

4. Luger, G.F. and Stubblefield, W.A., 1993, Artificial Intelligence: Structures and Strategies for Complex Problem Solving, second edition, Benjamin/Cummings, Redwood City, California.

5. Norvig, P., 1992, Artificial Intelligence Programming, Morgan Kaufman Publishers, San Mateo, California.

6. Pereira, F.C.N. and Shieber, S.M., 1987, Prolog and Natural Language Analysis, CSLI Lecture Notes, Number 10, Stanford University Press, Stanford, California, 1987.

7. Sebesta, R.W., 1999, Concepts of Programming Languages, fourth edition, Addison–Wesley, Reading, Massachusetts.

8. Ullman, J.D., 1997, Elements of ML Programming, second edition, Prentice-Hall.

9. Watson, M., 1997, Intelligent Java Applications for the Internet and Intranets, Morgan Kaufman Publishers, San Mateo, California.