Features of Fifth Generation Languages: A Panoramic View

Wilf R. Lalonde
John R. Pugh

SCS-TR-70
March 1985

School Of Computer Science
Carleton University
Ottawa K1S 5B6
Ontario
Canada

This research was supported by the Natural Sciences and Engineering Research Council of Canada and DREA

Features of Fifth Generation Languages: A Panoramic View

Wilf R. LaLonde and John R. Pugh
School of Computer Science
Carleton University
Ottawa, Ontario, Canada K1S 5B6

Abstract

Fifth generation languages are general purpose programming languages with capabilities beyond those of existing programming languages. Nevertheless, their basic features and capabilities are already available and distributed in existing programming languages. We survey the more important ones, provide illustrations via program segments in various languages, and discuss their inherent importance.

The survey supports the notion that two major directions are being pursued: the logic and the actor directions – each grounded on a different subset of the capabilities. However, as each approach attempts to add on additional capabilities, an ultimate convergence can be predicted. This ultimate fifth generation language is likely to contain the best of logic and actor-based languages.

1 Introduction

Fifth generation systems are based either on logic or actors.

Fifth generation systems are high-speed multiprocessor architectures supporting a general purpose programming language with capabilities beyond those of existing programming languages. Two major schools of research are involved in the design of such systems: the logic school and the actor school. The former can be profitably viewed as attempting to extend Prolog[1]; the latter, Smalltalk[2]. Each approach is based on a different set of fundamental features and capabilities, facilities that are already available to some extent in various programming languages. We survey the more important facilities, provide illustrations through sample programs in various languages, and discuss their relevance to fifth generation systems.

The survey serves as the basis for the observation that the two schools are attacking the same problems from different perspectives. As each approach attempts to add on additional capabilities, an ultimate convergence can be predicted. The resulting fifth generation language is likely to contain the best of both logic and actor-based languages.

2 Desirable Features of Fifth Generation Languages

The important features of fifth generation systems have little to do with syntactic notions.

The important issues are non-syntactic in nature. They relate to properties and features that contribute to computational power, algorithm clarity, extendibility, and software management. As a prelude to subsequent detailed discussions, we list without discussion those that we consider most important along with one or two examples of programming languages that support or exhibit the feature:

1. **Symbolic Manipulation** (Lisp[3], Prolog[1])
2. **Object-Orientation** (Smalltalk[2])
3. **Classes and Inheritance** (Smalltalk[2], Loops[4])
4. **Classes and Prototypes** (Act 1[5])
5. **Backtracking** (Icon[6], Prolog[1])
6. **Rule-Based Facilities** (Prolog[1], Loops[4])
7. **Unification** (Prolog[1])

8. **Pattern Matching** (Icon[6])
9. **Concurrency** (Csp[7])

3 Symbolic Manipulation

Symbolic manipulation is the ability to juggle any object thrown at you.

Symbolic manipulation is synonymous with fifth generation computing. For systems to be able to reason about language, programs, and sensory data, they must be able to manipulate these objects easily. Expert systems, in particular, and artificial intelligence software systems, in general, manipulate knowledge – information that is inherently symbolic. For example, the Prolog clauses below represent two kinds of knowledge – facts and rules.

```
father (wilf, emile).

ancestor (Person1, Person2) :-
   father (Person2, Person1).

ancestor (Person1, Person2) :-
   ancestor (Person1, Person),
   ancestor (Person, Person2).
```

The first clause represents the fact that wilf[1] is the father of emile. The second clause represents the rule that Person1 is an ancestor of Person2 if Person2 is the father of Person1. This may not be the only rule from which we can deduce ancestral information. The third clause is a rule which states that Person1 is also an ancestor of Person2 if Person1 is an ancestor of Person and Person is an ancestor of Person2.

When we speak of fifth generation systems, we refer to knowledge bases consisting of both data and rules rather than databases. Knowledge processing is symbolic processing.

Another obvious reason why symbolic processing is useful is the realization that many of the tasks done by programs today are in fact symbolic processing tasks. For example, if we consider the processors within a typical operating system, we find that the compilers, linkers, assemblers, editors, etc. are really symbolic processing programs. It is interesting to note that more often than not these programs are written in languages unsuited for symbolic computation. The requirement for symbolic manipulation in a language becomes critical if we examine more advanced tasks. For example, correctness provers for programs, interpreters for programs written in a specification language, transformers that convert programs written in a specification language into a more standard language, and program generators that translate queries in a natural English-like language into a special database retrieval language. All of these are symbolic processing tasks.

Finally, since symbolic manipulation includes program manipulation, advanced systems need not force users to use a language as is. Instead, they may be permitted to modify it to suit the particular task at hand. If special constructs are required for solving problems in a particular domain, they may be defined as language extensions and subsequently used as primitive constructs.

To illustrate symbolic computing we will look at Lisp and Prolog which both support it and consider four examples – two dealing with program manipulation and two, with language extension. Program manipulation is illustrated by describing a simple program loader; i.e., a program that loads other programs. Language extension is illustrated by adding a conventional if-then-else control structure.

---

[1] In standard Prolog, only variables begin with an uppercase letter.

An Example of Program Manipulation in LISP

A loader is a program which can input another program. In the example below, the LISP function **LOAD** takes two parameters; **FILE_NAME**, a file containing the program to be input and **READER**, the reader to be used to read from the file. So, for example, the Lisp function call

(LOAD "TESTFILE" READ)

loads the file **"TESTFILE"** using the standard Lisp reader **READ**. The reader used need not be the standard Lisp reader; it might instead be a reader for some specialized language. Note: in the sample programs below, a syntactically sugared version of Lisp is being used. Such a version can easily be produced with suitable macro extensions (to be demonstrated later).

```
(FUNCTION
   LOAD (FILE_NAME READER)
   BEGIN
     (LET
        (FILE BE (OPEN FILE_NAME))
     IN
        (WHILE (NOT (END_OF_FILE FILE)) DO
           (PRINT (EVALUATE (READER FILE)))
        ENDWHILE)
        (WRITE FILE_NAME " LOADED.")
     ENDLET)
   ENDFUNCTION)
```

To illustrate the use of the function, imagine that a file containing the function **LOAD** is to be input using the **LOAD** function and the standard Lisp reader! The function opens the file passed as a parameter and invokes the reader (also passed as a parameter) as long as the end of the file is not reached. The result of invoking the reader is a complete Lisp expression which is subsequently executed; the result is printed. In our example, invoking the standard Lisp reader causes the complete definition of the function **LOAD** to be input. The function definition for **LOAD** is then evaluated. In Lisp, evaluating a function definition returns the name of the function as the result. Thus, if our file contained a set of function definitions, each one would be read and evaluated and the printed output would be a list of the names of the functions defined. When end of file is reached, the **LOAD** function prints a message to indicate the file has been successfully loaded.

An Example of Program Manipulation in Prolog

Here, we tackle the same problem, but this time we use the logic programming language, Prolog. In Prolog terms, we ask whether the goal of loading the file **"TESTFILE"** with the standard Prolog reader **read** succeeds or not.

?- load ("TESTFILE", read).

As in the Lisp example, a reader for some specialized language can be used in place of the standard Prolog reader. For illustrative purposes, imagine that the file to be loaded contains the three Prolog clauses shown below. This is, of course, also the program which performs the loading.

```
load (Filename, Reader) :-
   open (Filename),
   readandevaluate (Filename, Reader).

readandevaluate (Filename, Reader) :-
```

```
    Reader (Filename, ProgramSegment),
    evaluate (ProgramSegment),
    readandevaluate (Filename, Reader).

readandevaluate (Filename, Reader) :-
    write ("loaded").
```

The goal of loading a file with a given reader succeeds if the goal of opening the file, and of subsequently reading and evaluating the file succeeds. To satisfy the goal of reading and evaluating a file with a given reader, we use the reader to read in the next object and then evaluate it. The first clause for **readandevaluate** is recursive and thus all objects are read from the file and evaluated. Ultimately, the goal of reading an object from the file will fail because we will reach the end of the file. The first clause of the predicate **readandevaluate** therefore fails and Prolog searches for an alternative way of satisfying this goal. The second clause is found and used. The message "loaded" is printed and the **readandevaluate** goal and hence the **load** goal both succeed. If the standard Prolog reader were used to input a file containing the three Prolog clauses shown above, the reader would read complete clauses from the file and the evaluator, when given a clause, would add it to the Prolog database.

In some Prolog systems, it is not legal to have a goal which begins with a variable as in **Reader (Filename, ProgramSegment)**. In that case, a special clause is provided as part of the Prolog system with the functionality required to execute the above. Additionally, we have not shown the clauses for the evaluate goal. It can be easily defined by converting its list input to a Prolog clause and then executing it as in the Reader situation above.

An Example of Language Extension in Lisp

As a further example of the power of symbolic manipulation, consider the problem of language extension. Many Lisp implementations provide only a sparse syntax for **if ... then ... else ...** conditionals. That is, they provide only a syntax of the following form.

(IF Expression Statement1 Statement2)

We might prefer an alternative which explicitly shows the **then** and **else** components as follows.

(If Expression (Then Statement1) (Else Statement2) Endif)

In Lisp, we can extend the language to enable use of the modified syntax for conditionals through the Lisp macro facility.

```
(MACRO
   If (Parameters)
   BEGIN
      (LET
         (If_Name     BE (ELEMENT 1 Parameters))
         (Expression  BE (ELEMENT 2 Parameters))
         (Then_Part   BE (ELEMENT 3 Parameters))
         (Else_Part   BE (ELEMENT 4 Parameters))
         (Endif_Name  BE (ELEMENT 5 Parameters))
      THENLET
         (Statement_1 BE (ELEMENT 2 Then_Part))
         (Statement_2 BE (ELEMENT 2 Else_Part))
      IN
         (DESTRUCTIVE_REPLACE
            Parameters
            (LIST 'IF Expression Statement_1 Statement_2))
```

```
    ENDLET)
ENDMACRO)
```

The basic strategy is to convert the verbose **If** into an equivalent standard **IF** and then evaluate it in place of the original. The Lisp macro feature provides the required capability. A macro is a special function which takes everything provided at the call point and binds it unevaluated to the single parameter of the macro. The body of the macro is then evaluated and the result of this evaluation replaces the original macro invocation. Finally, this replacement is evaluated in the original context as if it had been there all along. To illustrate the evaluation process, consider the following macro call.

(If (< N 0) (Then (← S -1)) (Else (← S 1)) Endif)

The entire verbose **If** is bound to the variable **Parameters**. To construct the sparse **IF**, function **ELEMENT** is used to extract the expression along with the then and else components from the verbose **If**. **ELEMENT** extracts the Nth component from a list where N is supplied as a parameter to the function. Thus the call **(ELEMENT 2 Parameters)** would extract **(Then (← S -1))** from the verbose **If**. **ELEMENT** can now be used again to extract the statement **(← S -1)**. When all required components have been extracted, the equivalent sparse **IF** can be constructed, namely

(IF (< N 0) (← S -1) (← S 1))

The sparse **IF** is therefore the result of evaluating the macro body which is itself then evaluated in the original context to give the final result. The effect of the **DESTRUCTIVELY_REPLACE** is to physically replace the verbose **If** with the sparse **IF**. This ensures that macro expansion is only performed once. If this code were evaluated a second time, it would already contain the sparse **IF**; i.e., the previously expanded verbose **If**.

An Example of Language Extension in Prolog

In Prolog, language extension is achieved through operator definitions; i.e., new keywords are added to the language by defining them as operators with suitable priorities and associativities. A Prolog clause using these operators is then provided describing the interpretation of the new control structure.

```
op (800, fx, if).
op (801, xfx, then).
op (802, xfx, else).
op (803, xf, endif).

if Expression then Statement1 else Statement2 endif :-
   Expression, !,
   Statement1.

if Expression then Statement1 else Statement2 endif :-
   Statement2.

test (Data) :-
   if simple (Data) then simpletest (Data) else complextest (Data) endif.
```

The operator definitions specify that "if" is a prefix operator with priority 800, "then" and "else" are infix operators with priority 801 and 802 respectively, and "endif" is a suffix operator with priority 803. In Prolog, smaller numbers have higher priorities; arithmetic expression operators, for instance, have priority values less than 100. Thus "if" has a higher

priority than "then", ..., "else" has a higher priority than "endif". Consequently, writing "if A then B else C endif" is equivalent to writing "endif (else (then (if (A), B), C)".

On the other hand, a Prolog with a Lisp-like syntax such as Micro-Prolog[8] can be extended in a more traditional way. Simply define a clause with the preferred syntax where the body of the clause contains the equivalent standard syntax.

```
((if Expression then Statement1 else Statement endif)
   Expression !
   Statement1)

((if Expression then Statement1 else Statement endif)
   Statement2)
```

The notion of language extension can be taken one stage further. For example, in a language with symbolic manipulation capability, we could design an Ada-like syntax without brackets to allow the following.

IF Expression THEN Statement1 ELSE Statement2 ENDIF

In this case, a special reader could be written to read in the syntax without brackets and convert it to a syntax with brackets. The Ada-Reader would parse the Ada input and generate equivalent Lisp or Prolog code.

4 Object-Orientation

Objects encapsulate facts and rules.

In designing a fifth generation system, it is important to consider carefully whether or not it will support the object-oriented paradigm. An object-oriented system consists solely of objects which encapsulate information (data) and methods (procedures) for performing operations on that information. Computation is achieved by passing messages to (or making requests of) the objects in a program. Objects respond to messages by producing new objects as results.

This philosophy is very different from that used in conventional languages such as Pascal and Fortran. In these languages, procedures and data are considered separate entities rather than a single entity. Procedures are active agents acting on passive data elements, whereas in object-oriented systems it is the objects which are active. Invoking a procedure with parameters and passing a message to an object are very different. Objects should be thought of as responding to requests rather than as routines performing specialized tasks.

The difference between object-oriented programming and conventional programming can be highlighted by considering the evaluation of the following simple expression:

(1.5 tan rounded + 1) * 2

In a traditional language, computation revolves around the functional operators rather than the operands. In an object-oriented language such as Smalltalk, evaluation would result in the following message passing sequence.

1. Send the message **tan** to the object **1.5** which responds by returning the object **14.101**.
2. Send the message **rounded** to the object **14.101** which responds by returning the object **14**.
3. Send the message **+ 1** to the object **14** which responds by returning the object **15**.
4. Send the message *** 2** to the object **15** which responds by returning the object **30**.

Note the fundamental change of emphasis. It is the objects which determine how the expression is evaluated. For example, it is the object **15** that determines how to interpret the message *** 2** rather than the multiplication operator being applied to the integers 15 and 2. In

a consistently object-oriented language like Smalltalk, everything is an object. The following example illustrates how control structures follow the object metaphor.

(number / 2) = 0 ifTrue: [parity ← 0] ifFalse: [Parity ← 1]

In this case, **ifTrue:** and **ifFalse:** are keywords in a message destined for the boolean object. Evaluation of this expression can be described as follows:

1. Send the message / **2** to the object referred to by **number**, which responds by returning either the object **0** or **1**. (We will assume **1**).
2. Send the message = **0** to the object **1** which responds by returning the boolean object **false**.
3. Send the message **iftrue: [parity ← 0] ifFalse: [parity ← 1]** to the object **false**. The boolean object **false** interprets the **ifTrue:ifFalse:** message by evaluating the block argument associated with the **ifFalse:** keyword. A block object can be thought of as a sequence of expressions to be evaluated at a later time.
4. Send the message ← **1** to the variable referenced by **parity**.[2]

Object-oriented programming facilities are only now becoming known because of Smalltalk's emerging visibility. The advantages too are only now becoming evident. The most obvious ones can be summarized as follows.

1. Objects provide a natural modularization tool. An object encapsulates a data representation together with the methods or procedures which operate on that representation.
2. Objects follow the principle of **information hiding**. The specification of an object can be described by its message protocol; i.e., the messages the object responds to, together with the effect of sending each message. To use an object, it is only necessary to understand this protocol – it is not necessary to know how the object is implemented.
3. All objects are instances of a particular type which allows a strong typing facility to be provided. The strong typing is different however from that found in languages such as Pascal and Ada because it is the objects which are strongly typed, not the variables which reference the objects. It is the object which determines whether or not it is legal to send it a particular kind of message.
4. Object-oriented languages are more consistent and uniform than conventional languages. For example, the duality between data structures and control structures does not exist. Whether or not an expression behaves as a control structure depends on the behavior of particular objects (such as booleans).
5. The message passing facility provided by object-oriented languages is absolutely essential in a language which will ultimately reside on a multi-processor system.
6. Finally, object-oriented programming encourages a powerful, new programming paradigm – programming by simulation. Each object in a program can be made to model some physical or conceptual object in the world. For instance, objects representing cars, rooms, computers, desks, toasters, toys, etc. can be manipulated as bonafide objects.

Experience with the paradigm will encourage additional extensions and facilities. Other advantages and also disadvantages are likely to emerge.

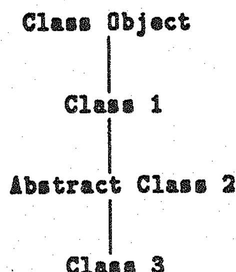
5 Classes and Inheritance

> Inheritance provides the ability to share another's wealth.

Inheritance is a feature which allows one object to share the attributes and operations of another. It is in recognition of this fact that we often create new objects that are specializations of already existing objects. By a specialization, we mean that an object can perform all the operations of a more general object but, in addition, has attributes and operations specific to itself. The specialized object can inherit the operations of the more general object. Consider the following example of a Smalltalk inheritance structure:

---

[2] Actually, the assignment in Smalltalk is magic. It is not implemented via message-passing.

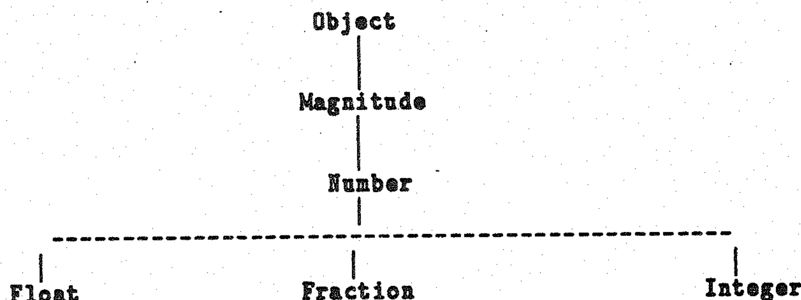


In Smalltalk, all objects are instances of some data type or class. Classes abstract out the common attributes and operations of a set of objects. As well as providing a natural classification mechanism, a class is a repository for operations that can be performed on instances belonging to the class and thus provides a code sharing capability. In addition, classes are arranged in a hierarchical structure. A class may have any number of subclasses but only a single superclass. Subclasses are thought of as specializations of classes higher up in the class hierarchy. A class inherits the operations of its superclass, its superclass's superclass and so on until the root of the tree (the class Object) is reached. In addition to these inherited capabilities, a class may be specialized by adding attributes and operations specific to itself.

In the inheritance structure shown, both classes and abstract classes are indicated. The distinction is that classes may construct instances of themselves but abstract classes may not. Abstract classes are a repository for operations shared by a number of subclasses. They are useful when classes share common operations but cannot be organized using class-subclass relationships.

Instances respond to messages by executing code found in their respective classes (or some superclass through the inheritance chain). In addition, messages may be sent to the class itself rather than to an instance. For example, we might wish to request the class to create an instance. Code for class methods (operations) as opposed to instance methods reside in the meta-class of the class. The class of a class is called its meta-class. Whenever a new class is created, a meta-class is also created with the class as its only instance. Meta-classes are arranged in a hierarchy which mirrors the class hierarchy.

The inheritance network shown below illustrates how numeric classes are organized in Smalltalk. Instances of **Float**, **Fraction**, and **Integer** are able to respond to messages specific to their classes and also to messages understood by classes **Number**, **Magnitude**, and **Object**.



Consider the following message expressions:

2.75 class	Ask the floating point number **2.75** what class of object it is. **Float** is returned.
(5/4) isMemberOf: Fraction	Ask the fraction **5/4** if it is a member of the class **Fraction**. **True** is returned.
3 class == Integer	Ask whether the class of the object **3** is identical to class **Integer**. **True** is returned.

Each of the messages **class**, **isMemberOf:**, and == reside in the class **Object** and thus the inheritance chain is used in each case to determine the method to evaluate.

Magnitude and **Number** are abstract classes. Consequently, they are only repositories for code shared by the subclasses **Float**, **Fraction**, and **Integer**. The protocol for class **Number** includes messages for typical arithmetic operations such as plus (+), minus (-), absolute value (**abs**), and negation (**negated**). Of course, these operations are applicable to each of the subclasses **Float**, **Fraction**, and **Integer**. However, operations such as plus are primitive and must be implemented by each subclass to take account of the different representations used for floats, fractions, and integers. If a plus message were sent to the abstract class **Number**, the response would be an error message stating that the implementation of the message is the responsibility of the subclass. One advantage of the abstract class **Number** is that some operations, for example, absolute value and negation, are non-primitive. These operations are definable in terms of primitive operations. Absolute value and negation can both be defined in terms of the primitive minus operation. This means that the code for these operations can reside in the abstract class even though code for the primitive minus operation must be provided by each of the subclasses.

To illustrate the description of classes and creation of instances, we define a class **ComplexNumber** for performing operations in complex arithmetic. An implementation for the following minimal specification will be provided:

realPart
returns the real component of a complex number.
imaginaryPart
returns the imaginary component of a complex number.
setRealPart: realValue andImaginaryPart: imaginaryValue
modifies the complex number by assigning new values to the real and imaginary parts.
+ aComplexNumber
returns a complex number equal to the sum of the complex number receiving the message and the argument **aComplexNumber**.
newWithRealPart: realValue andImaginaryPart: imaginaryPart
creates an instance of class **ComplexNumber** with real and imaginary parts equal to **realValue** and **imaginaryValue** respectively.

The following message expressions illustrate *programming* with complex numbers:

```
firstComplex  ← ComplexNumber newWithRealPart: 2.5 andImaginaryPart: 3.1.
secondComplex ← ComplexNumber newWithRealPart: -1.0 andImaginaryPart: 0.5.
complexSum    ← firstComplex + secondComplex.
realPortion   ← ComplexSum realPart.
```

Note that instances are created by sending a message to the class **ComplexNumber**. The definition of the class is shown below.

```
class name                        ComplexNumber
superclass                        Number
instance variable names           realPart
                                  imaginaryPart

class methods
   creation
      newWithRealPart: realValue andImaginaryPart: imaginaryValue
         | aComplex |
         aComplex ← ComplexNumber new.
         ↑aComplex setRealPart: realValue andImaginaryPart: imaginaryValue

   instance methods

      extraction
         realPart
            ↑realPart

         imaginaryPart
            ↑imaginaryPart

      modification
         setRealPart: realValue andImaginaryPart: imaginaryValue
            realPart ← realValue.
            imaginaryPart ← imaginaryValue


      operation
         + aComplex
            | sumRealPart sumImaginaryPart |
            sumRealPart ← realPart + aComplex realPart.
            sumImaginaryPart ← imaginaryPart + aComplex imaginaryPart.
            ↑ComplexNumber
               newWithRealPart: sumRealPart andImaginaryPart: sumImaginaryPart
```

The key to understanding the instance methods is the realization that objects have access only to their own instance variables. Thus to add the corresponding real parts of two complex numbers, the object to which the + message has been sent must add its own real part with the real part of its parameter aComplex (it does not have access to aComplex's representation and must therefore explicitly request the real part of that object).

After a complete definition of complex numbers is provided, it is still the case that many operations are intentionally omitted because they are inherited through the class hierarchy. For instance, explicit operations that permit comparing complex numbers for equality and finding the maximum and minimum of two complex numbers need never be added.

An inheritance hierarchy of this kind is useful for three reasons: it is an organizational feature, it promotes sharing, and makes systems easier to modify. The hierarchical classification present in object-oriented systems is particularly important when organizing large software systems. The factorization provided by the subclassing and inheritance mechanisms helps promote the sharing of code. Since the physical amount of code is reduced and since a change made at a point high up in the hierarchy affects everything below it, modification and maintenance of the system is simplified.

The simple subclassing mechanism found in Smalltalk allows a class to have only a single superclass. The relationships between classes must therefore be described as a simple hierarchy. This view is unnatural and inadequate; new classes of objects are often not simply specializations or extensions of a single class of object. On the contrary, they embody facets or characteristics from a number of classes of object.

Alternatively, we can think of wishing to support multiple views or perspectives of the same object. For example, in a graphical simulation of shipping passing through a canal system, the classes Ship and Displayable_Object might be provided. Ships might have properties such as Weight, Length, Speed, Type etc. while instances of class Displayable_Object might have properties such as Position, Screen_Extent, Iconic_Representation etc. along with operations such as Display_At and Translate_By. A new class Displayable_Ship is required where instances may be viewed as both a ship with a set of physical characteristics and as a displayable object on a graphics screen. That is, a class which inherits from both of the classes Ship and Displayable_Object.

The ability to inherit information from more than one superclass, a feature known as **multiple inheritance**, is supported by a number of object-oriented systems, for example, Loops[4].

6 Classes and Prototypes

Prototypes are sample objects.

Classes, in the Smalltalk tradition, emphasize the view that all instances have the same representation and the same operations. Insisting that all instances have the same representation is a restriction that can be removed by associating the representation and operations, not with the class, but with a sample instance. Such a sample is called a **prototype** or **exemplar** since it serves as the model for other instances. A class can then be redefined as a repository for instances defined in terms of a small number of prototypical instances.

To illustrate the difference between Smalltalk classes and a version with prototypes, as exemplified, for instance, in Act 1[5], we will consider defining a class "List" twice - once using one exemplar and a second time using two. The benefits of being able to have several prototypes can then be discussed. The notation used below should be viewed as an experimental syntax.

```
class name                      List
class prototype                 AClass
instance prototypes             AList
class methods
   instance creation
      empty
         ↑AList empty


prototype name                  AList
prototype class                 List
prototype for prototype         AnObject
prototype instance variables    firstPart
                                restPart
                                actuallyEmpty

prototype methods
   private initialization
      setEmpty
         actuallyEmpty ← true.
         ↑self

      setNonEmpty: firstComponent and: secondComponent
         firstPart ← firstComponent.
         restPart ← secondComponent.
         actuallyEmpty ← false.
         ↑self

   creation
      empty
         ↑self clone setEmpty

   construction
```

```
      precede: newElement
         ↑self clone setNonEmpty: newElement and: self

   extraction
      first
         actuallyEmpty
            ifTrue: [self error: 'cannot take first of empty list']
            ifFalse: [↑firstPart]

      rest
         actuallyEmpty
            ifTrue: [self error: 'cannot take rest of empty list']
            ifFalse: [↑restPart]

   testing
      empty?
         ↑actuallyEmpty
```

As illustrated above, the distinction between classes and prototypes is artificial. On the one hand, List (the class) is an instance with AClass as its prototype and a list (any list) is an instance with AList as its prototype.

Asking the class for an empty list results in the message being relayed to the list instance prototype which returns a properly initialized clone of itself (presumably, clone is an operation inherited from AnObject). The other operations should be self-explanatory.

```
class name                        List
class prototype                   AClass
instance prototypes               ANonEmptyList AnEmptyList
class methods
   instance creation
      empty
         ↑AnEmptyList clone


prototype name                    AnEmptyList
prototype class                   List
prototype for prototype           AnObject
prototype instance variables
prototype methods
   creation
      empty
         ↑self clone

   construction
      precede: newElement
         ↑ANonEmptyList clone set: newElement and: self

   extraction
      first
         self error: 'cannot take first of empty list'

      rest
         self error: 'cannot take rest of empty list'

   testing
      empty?
         ↑true
```

```
prototype name                    ANonEmptyList
prototype class                   List
prototype for prototype           AnObject
prototype instance variables      firstPart
                                  restPart

prototype methods
   private initialization
      set: firstComponent and: secondComponent
         firstPart ← firstComponent.
         restPart ← secondComponent.
         ↑self

   creation
      empty
         ↑AnEmptyList clone

   construction
      precede: newElement
         ↑self clone set: newElement and: self

   extraction
      first
         ↑firstPart

      rest
         ↑restPart

   testing
      empty?
         ↑false
```

The two prototype approach distinguishes clearly between two fundamentally different kinds of lists: empty lists and non-empty lists. Empty lists have an empty representation whereas non-empty lists keep a record of two components. Similarly, operation implementations differ for the two; for instance, "first" applied to an empty list is an error but applied to a non-empty list provides a legitimate result. With the one prototype approach, however, the two kinds of lists must be merged under a common representation and a single implementation for the operations. However, it is still necessary to be able to differentiate between the two cases.

The ability to have multiple prototypes provides space and time savings. With respect to space, the representation, in the one prototype case, must be uniform. The only way to differentiate between empty and non-empty lists is to have a special indicator in the representation. This indicator is mandatory when multiple empty lists are allowed. However, it is possible to eliminate it if only one empty list is provided. In that situation, any one of the non-empty lists can be delegated to be the unique empty list (the components in this case are unused). Testing for an empty list is a matter of determining whether or not the object in question is identical to the delegated list. With respect to time, operations such as "first" must differentiate between empty and non-empty lists with executable code. This extra overhead is eliminated when the operation is sent to objects which themselves are distinguished.

In general, prototypes are a more powerful mechanism than classes since they can be used to implement classes.

7 Backtracking

Backtracking is an amazing facility for exploring a maze.

Backtracking is a language feature that supports non-deterministic programming. It provides the programmer with the ability to program a solution to a problem by considering several solution strategies but without having to worry about which one is the proper one to use in a particular situation. If a particular strategy fails to provide a solution or if multiple solutions are desired, it is the system's responsibility to "remember" the state of the computation and consider untried alternatives; i.e., it is the system's responsibility to **backtrack**.

In traditional programming languages, the closest approximation to backtracking is recursion. Either all the possibilities are stored explicitly in some data structure or implicitly through the recursive mechanism. The program is deliberately constructed to follow all possible avenues for success and must return all answers. In a programming language that supports backtracking, control over the backtracking process is automatic. More important, there is no need to compute all answers at once because a return is not irrevocable; i.e., a return not only provides an answer but also a context from which further answers can be computed. This greatly simplifies the programming of solutions to problems for which backtracking is a suitable paradigm.

In general, backtracking is a powerful feature. This power also has a secondary but quite important effect – conciseness. To illustrate backtracking, we will examine examples from the programming languages Icon and Prolog.

Icon[6] is a programming language based on generators – mechanisms that produce successive values on demand. By "on demand", we mean that if a particular value is not conducive to a solution, the system automatically backtracks and provides a second value and then a third and a fourth until it has no more. The most obvious example of generators occurs in looping constructs. For instance, "5 to 10" is a generator that provides the value 5 and then backtracks to provide the alternative value 6, followed by 7, 8, ... until it runs out of numbers – in this case, after 10. When this occurs, the generator is said to **fail**. Additional examples include

100 to 1 by -1

2 3 5 7

if x = (5 to 11 by 2) 21 (27 to 30) then write ("hi") else write ("bye")

if mod (Value, d := 2 (3 to Value by 2)) = 0 then write (" is divisible by " d) else write (" is prime")

The first two examples are similar to the "5 to 10" generator. The third, however, makes use of nested generators in a novel manner. Intuitively, the if statement will succeed (output hi) if x has a value that is either 5, 7, 9, 11, 21, 27, 28, 29, 30; otherwise, it will fail (output bye). The last example determines whether or not Value is prime. For Value to be prime, there must exist no d such that mod (Value, d) = 0; i.e., the comparison must fail for all values of d. But d is given all values between 2 and Value (excluding even numbers greater than 2) – a simple, but not particularly efficient, way to test for primeness. A more comprehensive example is shown below.

An Example of Backtracking in Icon

```
record Node
   Data, LeftTree, RightTree
end

procedure
   Leaves (Tree)
      if null (Tree)
      then
         fail
      else
         if null (Tree.LeftTree === Tree.RightTree)
         then
```

```
            return Tree.Data
        else {
            suspend Leaves (Tree.LeftTree | Tree.RightTree)
            fail
            }
end
```

This example illustrates a user definable generator – Leaves which may now be used to return the successive values associated with the leaves of a binary tree:

```
write ("The leaves are ")
every Value := Leaves (Tree) do write (Value, " ")
```

The first test in the procedure distinguishes between a null tree (no nodes) and a non-null tree. The second test distinguishes between leaves and non-leaves. Its successful execution hinges on the === operator that returns the rightmost operand if the two operands are identical and fails otherwise; consequently, the null test succeeds only if both operands are null (the test for a leaf). The suspend expression suspends with each leaf of the left subtree and the right subtree (notice the alternation generator "|"); i.e., the fail is reached only after all subtree leaves have been generated. Although not shown, two such "Leaves" generators could be used to solve the "same-fringe" problem; i.e., the problem of determining whether or not two trees (in general, with different structures) have the same corresponding leaf values in a preorder traversal.

An Example of Backtracking in Prolog

The following examples illustrate the rather different way that backtracking is achieved in Prolog. The example below contains Prolog facts and rules concerning family relationships.

```
female (mary).
female (joan).

parents (mary, frank, heather).
parents (joan, joe  , cathy).
parents (dave, frank, heather).
parents (jack, joe  , cathy).

sisterof (Person1, Person2) :-
    female (Person1),
    parents (Person1, Father, Mother),
    parents (Person2, Father, Mother).
```

Programming in Prolog consists of stating factual relationships (facts), specifying dependency relationships (rules), and asking questions (queries). In the program (or database of facts and rules) above, the predicates **female** and **parents** exemplify the facts whereas predicate **sisterof** exemplifies a rule. If we now ask the question "*Which* X *is such that* X *is female?*",

?- female (X).

we initially get the response

X=mary

Of course, there is a second alternative solution. Since Prolog remembers the search point it reached when it found the first solution, we can ask that the search for more solutions[2] be continued from this point. In this case, we would get the response

X=joan

If we ask for the search to be continued again, we receive the response

no

since there are no more **female** clauses to yield further solutions. A less trivial query is

?- sisterof (mary, dave).

representing the question ""*Is the sister of mary, dave?*. The clause *unifies*[3] with **sisterof (Person1, Person2)** causing **Person1** to be bound to **mary** and **Person2** to **dave**. Making these substitutions, the task of proving the goal reduces to proving the subgoals

female (mary), parents (mary, Father, Mother), parents (dave, Father, Mother)

where Father and Mother are new unbound variables. The goal **female(mary)** is a fact in the database and is thus proven. To satisfy the goal **parents (mary, Father, Mother)**, Prolog searches for a **parents** clause that has **mary** as its first argument. The first clause **parents (mary, frank, heather)** *unifies* with the goal and has the side-effect of binding the variable **Father** to **frank** and **Mother** to **heather**. Thus the original goal is proven if **parents (dave, frank, heather)** can be proven. It can and Prolog returns **yes** to the original goal.

The power of the built-in backtracking mechanism can be illustrated by choosing a query with variables. The following query requests all pairs **X** and **Y** such that **X** is a **sisterof Y** and **X** differs from **Y**. As actually defined, the **sisterof** predicate is not sufficiently precise to prevent a female from being her own sister.

?- sisterof (X, Y), X \= Y.

As in the previous example, the goal initially reduces to the following with **X** bound to **Person1** and **Person1** bound to **mary**. Similarly, **Y** is bound to **Person2** which is itself unbound.

parents (mary, Father, Mother), parents (Person2, Father, Mother)

The clause **parents (mary, Father, Mother)** *unifies* with the first **parents** clause binding **Father** to **frank** and **Mother** to **heather**. Now, Prolog must search for a **Person2** such that **parents (Person2, frank, heather)** holds. This *unifies* with the first **parents** clause binding **Person2** to **mary**. The goal **sisterof (X, Y)** succeeds with both **X** and **Y** bound to **mary**. Unfortunately, the goal **X \= Y** fails and Prolog must backtrack to look for another solution.

Backtracking in Prolog implies attempting to re-satisfy some goal. In this case, since there is no alternative way of satisfying the goal **X \= Y**, Prolog returns to the previous goal, **parents (Person2, frank, heather)** and attempts to re-satisfy it. For each previous goal, Prolog remembers the point reached during its search for a solution. Thus, the search for an alternative solution can start from the correct place in the database. The clause **parents (joan, joe, cathy)** is tried next. This fails but there are still more alternative clauses to use. The third clause **parents (dave, frank, heather)** succeeds and we now have a solution to **sisterof (X, Y)** with **X** and **Y** bound to **mary** and **dave** respectively. The goal **X \= Y** is also satisfied and a solution to the query has been found.

---

[2] By simply typing "; return" instead of "return".

[3] Unification is discussed in detail in a subsequent section. For now, the term can be viewed as a synonym for "match".

X=mary Y=dave

If we ask for another solution, Prolog will again backtrack to the previous goal **parents (Person2, frank, heather)**. The only remaining alternative is **parents (jack, joe, cathy)**. Since this does not *unify*, Prolog must backtrack further to the next previous goal and search for another alternative. At the risk of becoming tedious, the following second and final solution is found.

X=joan Y=jack

8 Unification

Unification is a mechanism for making twins
out of similar objects.

In any language, even a fifth generation language, some mechanism is needed to associate actual parameters with formal parameters. Traditional languages, however, distinguish between variables and values but fail to treat them uniformly. In particular, actual parameters are allowed to be variables or values, but formal parameters are almost certainly restricted to variables. To remove this restriction, it is necessary to treat variables as manipulatable objects and to have a syntactic notation for distinguishing variables from non-variables. For instance, in Prolog, variables begin in uppercase and non-variables lowercase. A value such as the list [A, b, C, d] contains both variables (A and C) and constants (b and d). When bound, the variables in the list become transparent; only their value is seen. A more general mechanism is obtained by allowing both actual parameters and formal parameters to be arbitrary values. The association process involves creating new formal parameters (in particular, new unbound variables must be provided) and matching the actual parameters with the corresponding formal parameters. This matching process is called **unification**.

For example, in the case of actual and formal parameters that are lists, corresponding constant elements must be equal, variables in one must be bound to the corresponding constant element in the other, and corresponding variables must be bound together (in the future, if either one is ultimately bound, so is the other). Additionally, multiple occurrences of the same variable must be bound to the same value. If the matching process is unsuccessful, alternative routines with the same name but different formal parameters can be considered. Obviously, generalizing the parameter matching mechanism can also result in generalizing the routine selection mechanism.

Intuitively, unification is simply a mechanism for making actual and formal parameters identical. Using a pictorial analogy, if the actual and formal parameters correspond to two faces with one missing a nose (a variable) and the other missing a mouth (another variable), then unification causes each face to acquire the missing features of the other; i.e., to become twins.

Simple Examples of Unification in Prolog

Caller	Receiver	Bindings
test (1, 2)	test (1, 2)	-
test (1, 2)	test (1, Y)	Y⇔2
test (1, 2)	test (X, 2)	X⇔1
test (1, 2)	test (X, Y)	X⇔1, Y⇔2
test (X, 2)	test (1, 2)	X⇔1
test (1, Y)	test (1, 2)	Y⇔2
test (X, Y)	test (1, 2)	X⇔1, Y⇔2
test (X_1,2)	test (X, 2)	X_1⇔X (No value)

Unification succeeds on all of the above examples. The first example illustrates constant matching; the next three, traditional parameter matching; the three after that illustrate bindings in the opposite direction; finally, the last illustrates bindings between variables.

The unification mechanism is quite powerful – especially when it is coupled with backtracking. For instance,

1. It allows simple syntactic restrictions to be specified on actual parameters.
2. It eliminates the need for evaluation in many cases.
3. It does not (although it can) differentiate between input and output variables.

8.1 Supports Syntactic Restrictions

Unification can be viewed as a mechanism for restricting parameters to certain syntactic classes. For instance, specifying a parameter in the form [A] insists that it be a one-element list; the form [A, B] insists that it be a two-element list, ... In a backtracking environment, more complex semantic restrictions can be provided by adding appropriate run-time checks. For example,

```
test ([A, B]) :-
   restriction1 (A),
   restriction2 (A),
   restriction3 (A),
       ...
test ([A, B]) :-
   restriction10 (A),
   restriction20 (A),
   restriction30 (A),
       ...
```

Without backtracking, the restrictions might simply be viewed as error checks. This can be done voluntarily in traditional languages. With backtracking, they play the role of selection mechanisms for appropriate execution code.

8.2 Evaluates As a Side-Effect

Only simple examples of unification have been presented so far. More complex examples are needed to provide a good view of its power. These examples will also illustrate that unification is an evaluation mechanism; i.e., not only does it perform syntactic matching but it also constructs complex values which could only be obtained through evaluation in more traditional languages.

More Complex Examples of Unification in Prolog

Caller	Receiver	Bindings
`test ([1, 2, 3])`	`test (X)`	`X⇔[1, 2, 3]`
`test ([1, 2])`	`test ([X, Y])`	`X⇔1, Y⇔2`
`test (Z)`	`test ([X, Y])`	`Z⇔[X, Y]`
`test ([sum, 1, 2, 3])`	`test ([sum | X])`	`X⇔[1, 2, 3]`
`test ([1, 2], [sum, 1, 2])`	`test (X, [sum | X])`	`X⇔[1, 2]`
`test ([1, 2], [sum, Z])`	`test (X, [sum, X])`	`X⇔[1, 2], Z⇔X or Z⇔[1, 2]`
`test (Z, [sum, [1, 2]])`	`test (X, [sum, X])`	`X⇔[1, 2], Z⇔X or Z⇔[1, 2]`

The notation [a, b | X] denotes the list with first element a, second element b, and remaining elements X (X must be a list with zero or more elements). In Lisp, for example, the effect of the second example would be achieved by explicitly extracting the first and second elements from the list [1, 2]. The last three examples are particularly interesting because they use the same variable twice in the receiver. In the last example, for instance, Z and X must be bound together (the first parameter) while X must be bound to [1, 2]. It should be clear that binding order is unimportant for unification.

8.3 Makes Inputs and Outputs Interchangeable

A consequence of the symmetry provided by unification is that the caller can decide on the direction of information flow. For instance, a factorial routine could be designed in Micro-Prolog so that (1) (Factorial 4 24) succeeds, (2) (Factorial 4 !X)[4] succeeds with !X bound to 24, (3) (Factorial !X 24) succeeds with !X bound to 4, and finally (4) (Factorial !X !Y) succeeds with successive values !X=0 and !Y=1, !X=1 and !Y=1, !X=2 and !Y=2, etc. The routine can be written quite compactly by a Prolog expert by computing the answers bottom up as for case (4) above. On the other hand, traditional programmers tend to provide the following implementation.

```
((Factorial 0 1)

((Factorial !N !Result)
    (Known !N)
    (Known !Result)
    (GREATER !N 0)
    (Factorial !N !ActualResult)
    (EQ !Result !ActualResult)
    /)

((Factorial !N !Result)
    (Known !N)
    (Unknown !Result)
    (GREATER !N 0)
    (- !N 1 !NMinus1)
    (Factorial !NMinus1 !PartialResult)
    (PROD !N !PartialResult !Result)
    /)

((Factorial !N !Result)
    (Unknown !N)
    (Known !Result)
    (Successor 0 !N)
    (Factorial !N !PotentialResult)
    (GREATEROREQUAL !PotentialResult !Result)
    /
    (EQ !PotentialResult !Result))

((Factorial !N !Result)
    (Unknown !N)
    (Unknown !Result)
    (Successor 0 !N)
    (Factorial !N !Result))
------------------------------------------------------
```

[4] Micro-Prolog can be customized to recognize names prefixed by "!" as variables

```
((Successor !Value !NextValue)
    (SUM !Value 1 !NextValue))

((Successor !Value !ArbitrarilyMoreThanValue)
    (SUM !Value 1 !NextValue)
    (Successor !NextValue !ArbitrarilyMoreThanValue))

-------------------------------------------------------

((GREATER !Number1 !Number2)
    (LESS !Number2 !Number1))

((GREATEROREQUAL !Number1 !Number2)
    (LESS !Number2 !Number1))

((GREATEROREQUAL !Number1 !Number2)
    (EQ !Number1 !Number2))

-------------------------------------------------------

((Unknown !Variable)
    (VAR !Variable))

((Known !Value)
    (NOT VAR !Value))
```

Known succeeds only if the actual parameter has a value; Unknown succeeds only if it is an unbound variable. The cut symbol "/" signifies that no more values exist; more generally – it forbids backtracking through the cut but this viewpoint may be more confusing than the former. The first definition of Factorial is the basis case and as such, it plays a quadruple role: it will succeed if (1) the input and output[5] is 0 and 1 respectively, (2) the input is 0 and the output unknown (1 is computed), (3) the input is unknown (0 is computed) and the output 1, or (4) the input is unknown (0 is computed) and also the output (1 is computed). The second definition is a checking version: as long as the first three conditions hold, the value of Factorial (!N) is computed and checked against the answer provided. The third is the traditional recursive version which computes Factorial (!N-1) and returns the result multiplied by !N. The last definition returns an infinite stream of pairs: (1, 1) (2, 2) (3, 6), etc. Backtracking and the use of Successor is crucial to its operation. Backtracking, for instance, can be forced by writing

? ((Factorial !X !Y) (PP "Factorial " !X " is " !Y) FAIL)

The crucial point about Successor is that it returns many answers; 1 the first time, then 2, then 3, ... The first time that Factorial is invoked with both parameters unknown, Successor binds !N to 1. Factorial (1) is then computed and the invocation succeeds. Upon backtracking (when FAIL is encountered above), the computation of a new value for Factorial (1) is attempted. There are no additional successful results; hence further backtracking to the Successor operation occurs. It succeeds and binds the value 2 to !N and the forward computation repeats as above. When FAIL is reached again, backtracking repeats identically and 3 is bound to !N, ...

In more detail, consider Successor (0). Upon first invoking it, !Value is 0, !NextValue is set to 1, and the computation succeeds. Upon backtracking, SUM in the first definition fails to return a second value. Hence the second definition is tried next. Consequently, !NextValue is set to 1 and the Successor (1) is computed using the first definition (2 is returned). Upon backtracking, the second definition of Successor is used to compute a new value for Successor (2). It returns Successor (3) ...

[5] We refer to the first parameter as the input and the second, as the output.

8.4 Relationship Between Traditional Syntax and Bracketted Forms

So far, three different syntactic notations have been used: (1) parenthesis free form (the syntactic sugared variety provided by traditional languages), (2) fully bracketted functional form (as exemplified by Lisp), and (3) fully bracketted unification form (as exemplified by Micro-Prolog). Functional form provides the result **in place** whereas Unification form provides the result in an additional explicit parameter.

Parenthesis-Free Form	1 + 2 * 3
Functional Form	(+ 1 (* 2 3))
Unification Form	(* 2 3 X_1) (+ 1 X_1 X_2) X_2 is the answer

Each of these syntactic notations is associated with a corresponding class of programming languages. An alternative viewpoint considers them different translation stages in a multi-notation language. Using traditional parser technology, it is easy to provide a conventional reader that parses bracketless notation into functional form. Similarly, it is easy to provide a functional form reader that converts functional form to unification form. A sample converter performing basic binary expression unrolling; i.e. removal of nesting, is provided below in Micro-Prolog.

```
((TRANSLATE (!Operation !Operand1 !Operand2) !Output !AnswerObject)
   (TRANSLATE !Operand1 !Output1 !AnswerObject1)
   (TRANSLATE !Operand2 !Output2 !AnswerObject2)
   (APPEND
      !Output1
      !Output2
      ((!Operation !AnswerObject1 !answerVariable2 !answerVariable))
      !Output)
   /)

((TRANSLATE !Operand () !Operand)
   /)
```

The translation routine converts a functional form expression into a list of unification expressions and also returns the variable containing the final answer. Thus (TRANSLATE (* 1 (+ 2 3)) !X !Y) binds !X to ((+ 2 3 !X_1) (* 1 !X_2 !X_3)) and !Y to !X_3.

9 Pattern Matching

Pattern matching is compromised unification.

Pattern matching is a special case of unification in which the actual parameters do not contain variables as values. Thus only the formal parameters contain variables and the binding is one-way.

Traditionally, pattern matching has been associated with programming languages such as Snobol[9]. The patterns have evolved to an elaborate degree. By contrast, Prolog and its unification patterns are extremely simple. Research is needed to develop more complex patterns and the corresponding unification algorithms.

10 Concurrency

> Concurrency provides users with the opportunity to interact with independent bugs at the same time.

Fifth generation systems are expected to be multi-processor machines with highly parallel architectures. Aside from the obvious speed improvements and the potential for increased levels of redundancy, parallelism will enable new programming styles that were previously impractical. For example, some problems are sufficiently complicated that only heuristic algorithms are possible; i.e., the algorithms may provide good solutions for certain special cases but may be computationally too expensive for others. Unfortunately, deciding which algorithm to use is just as difficult a problem. One approach to solving these types of problems is to have several algorithms work on the same problem in parallel. As long as one or two algorithms can successfully solve the problem, it doesn't matter that others might never solve it or never terminate. All the algorithms can be initiated as if it were a race; the first to finish successfully wins and provides the solution; all others are aborted.

In such an environment, special facilities for controlling and coordinating parallel computation are therefore essential. Fortunately, this aspect of fifth generation systems has a ring of familiarity to traditional computer science and engineering disciplines. Concurrency, synchronization, and mutual exclusion have a long history of development in operating systems, database systems, and computer networks. Corresponding language features range from low level semaphores in Smalltalk[2], higher level send and wait primitives in C[10], corresponding implicit mechanisms in Csp[7], rendezvous in Ada[11], to mention a few.

From the point of view of fifth generation systems, the problem is not whether to have concurrency controlling primitives but rather what class of primitives to provide. Indeed, a reasonable choice seems to be merely a detail once the other language features are settled upon. More important, the kind of fifth generation system that results is more profoundly influenced by the designers' underlying world model; i.e., whether they view the world as basically serial, basically parallel, or an irrelevant mixture of the two.

10.1 The View That the World is Basically Sequential

This is the traditional and most obvious viewpoint. It is predicated on the premise that the majority of programming problems are serial in nature and that parallelism can be usefully employed for executing many such serial tasks in parallel. Under this view, the obvious strategy for developing a fifth generation system is to begin with a reasonable sequential language and parallelize it. The underlying serial architecture that supports the sequential language is then modified into a parallel architecture. The task is considerably simplified if the designers start with a base language with a majority of the desirable features already built-in. Obvious examples include attempts to add concurrency to Lisp[12,13] and Prolog[14] (in the latter case, the language called Concurrent Prolog is totally different from the initial base language, Prolog).

10.2 The View That the World is Basically Parallel

This viewpoint is predicated on the opposite premise; that the majority of programming problems are parallel in nature and that serialism is a consequence of control imposed on the parallelism. Under this view, the obvious strategy for developing a fifth generation system is to begin with a highly parallel language and serialize it; i.e., add control to slow it down. The resulting architecture is designed for parallelism as the rule and serialism as the exception. The only example of a general purpose language designed under this viewpoint is the Act 1[5] language.

10.3 The View That Serialism and Parallelism in the World is Basically Irrelevant

Under this viewpoint, the language user is not expected to worry about parallelism. It is the system's responsibility to carry out the computation with as much parallelism as possible and to worry about synchronization problems. The approach works best when the programming language appears on the surface to be a purely sequential language but which may be viewed or re-interpreted as if it were a parallel language. The programming language Prolog is an example sequential language which can be re-interpreted with a parallel language semantics. The aim in

this case is to make the parallel version of the language behave in the same way as the sequential version; the parallel version simply solves the same problem with considerably higher speed. Whether or not this can be successfully achieved has yet to be demonstrated.

Conclusions

We have provided a panoramic view of some of the more important features of fifth-generation programming languages. Two major schools of research into the design of fifth generation programming systems are currently being pursued: the logic school and the actor school. We conclude with the observation that the two schools are attacking the same problem from different perspectives and that, in the long run, languages that unite the best features of each will ultimately prevail.

We note that the logic school, working from a base of Prolog with its backtracking and unification capabilities, are investigating extensions to provide object-oriented programming facilities.[15] The actor school, working from a base of object-oriented programming and inheritance capabilities as exemplified by Smalltalk, can be expected to consider support for backtracking and unification. Act 1, for example, is already based on unification.

In general, fifth generation languages will need to provide support for a large subset of the following facilities: symbolic manipulation, object management, classes, prototypes, inheritance, backtracking, rule-based programming, unification and concurrency.

References

1. Clocksin, W.F. and Mellish, C.S., *Programming in Prolog*. Springer-Verlag, 1982.
2. Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, Reading, Mass., 1983.
3. Winston, P.H. and Horn, B.K.P., *Lisp (Second Edition)*. Addison-Wesley, Reading, Mass., 1984.
4. Bobrow, D.G. and Stefik, M.J., "The LOOPS Manual (Preliminary Version)". *Knowledge-Based VLSI Design Group Technical Report KB-VLSI-81-13*, Stanford University, August 1984.
5. Lieberman, H., "A Preview of ACT 1". *MIT AI Laboratory*, Memo No. 625, June 1981.
6. Griswold, R.E. and Griswold, M.T., *The Icon Programming Language*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1984.
7. Hoare, C.A.R., "Communicating Sequential Processes". *Comm. ACM*, Vol 21, No 8, August 1978, pp. 666-677.
8. Clark, K.L. and McCabe, F.G., *Micro-PROLOG: Programming in Logic*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1984.
9. Griswold, R.E. and Griswold, M.T., *A Snobol4 Primer*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1973.
10. Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1978.
11. *US DoD, Ada Programming Language, ANSI/MIL-STD-1815A-1983*. Feb 17, 1983.
12. Halstead, R.H., "Implementation of Multilisp: Lisp on a Multiprocessor". *ACM Symposium on Lisp and Functional Programming*, Austin, Texas. August 1984.
13. Gabriel, R.P. and McCarthy, J., "Queue-based Multi-processing Lisp". *ACM Symposium on Lisp and Functional Programming*, Austin, Texas. August 1984.
14. Shapiro, E.Y., "A Subset of Concurrent Prolog and its Interpreter". *ICOT Technical Report*, TR-003, 1983.
15. Zaniolo, C., "Object-oriented Programming in Prolog". *1984 International Symposium on Logic Programming*, New Jersey, Feb. 1984, p. 265-271.