Combining the Functional and Oject-Oriented Paradigms in the FOBS-X Scripting Language

James Gil de Lamadrid
Bowie State University, Bowie, MD, USA, 20715

ABSTRACT

A language FOBS-X (Extensible FOBS) is described. This language is an interpreted language, intended as a universal scripting language. An interesting feature of the language is its ability to be extended, allowing it to be adapted to new scripting environments.

The interpretation process is structured as a core-language parser back-end, and a macro processor front-end. The macro processor allows the language syntax to be modified. A configurable library is used to help modify the semantics of the language, adding the required capabilities for interacting in a new scripting environment.

This paper focuses on the macro capability of the language. A macro extension to the language has been developed, called the standard extension, that gives FOBS-X a friendlier syntax. It also serves as a convenient tool for demonstrating the macro expansion process.

KEYWORDS
Hybrid, scripting, functional, object-oriented.

1. INTRODUCTION

The object-oriented programming paradigm and the functional paradigm both offer valuable tools to the programmer. Many problems lend themselves to elegant functional solutions. Others are better expressed in terms of communicating objects.

A single language with the expressive power of both paradigms allows the user to tackle both types of problems, with fluency in only one language. Many languages have both object oriented features, and functional features. For example, a language like Java, which is object-oriented, allows the user to write functions with recursion. If the programmer restricts their programming style, they can produce programs that are in line with programs produced using pure functional languages.

FOBS-X differs from other languages that support both functional and object-oriented features. These languages, are mostly centered around object-orientation. FOBS-X has a distinctly functional flavor.

FOBS-X is a modification to the FOBS language described in Gil de Lamadrid and Zimmerman [23, 24]. FOBS-X is characterized by the following features:

●A single, simple, elegant data type called a FOB, that functions both as a function and an object.
●A stateless runtime environment. In this environment, mutable objects are not allowed. Mutation is accomplished, as in functional languages, by the creation of new objects with the required changes.
●A simple form of inheritance. A *sub-FOB* is built from another *super-FOB*, inheriting all attributes from the super-FOB in the process.
●A form of scoping that supports attribute overriding in inheritance. This allows a sub-FOB to replace data or behaviors inherited from a super-FOB.
●A macro expansion capability, enabling the user to introduce new syntax.
●A library modification mechanism allowing the user to adapt the language to new scripting environments.

2. FOBS-X DESIGN OVERVIEW

In designing FOBS-X several factors were taken into consideration. Our goal was to design a language that introduced objects into a functional language without destroying the functional property of referential transparency exhibited by stateless environments. In addition, it was important that the resulting language be concise in syntax and semantics. This requirement ensures that a thorough formal definition of the language can easily be provided, and that a simple interpreter for the language can be constructed. The result is a light-weight language, that can be confidently used, knowing that it adheres unfailingly to a standard definition.

Borrowing from languages like Smalltalk[10], in which everything is an object, we sought a single homogeneous structure expressive enough to represent all necessary data objects. The FOB, described further in Section 5, is this single data type that exhibits behaviors of both an object and a function. This characteristic of FOBS-X, makes it clear that FOBS-X is not a language with a strong system of types, and type checking. In fact, the single homogeneous type limits the amount of static type checking that can be performed. The result is that FOBS-X, like Smalltalk, is dynamically typed.

In scripting languages like FOBS-X, the flexibility of dynamic typing usually outweighs the safety benefits of static typing. In these languages, where there is no separate compilation phase, we feel that performing static error checking is less important. Any information that would have been discovered at compile time can easily be discovered by an operator at run time, and an error message specific to that problem can be issued by the operator. Also, dynamic typing has its advantages. In particular it relieves the user from the type definitions, and the constraints of typed variable use.

Although a simple language is beneficial to the language implementer, it is not necessarily beneficial to the user, who may be more interested in a large set of features to choose from to easily craft their code. Considering this, we decided to allow the user the flexibility to change the syntax of the language, tailoring it to their own preferences. Extensions to the core language can be defined using a fairly sophisticated macro expansion system. This feature is covered in more detail in Section 10.

Macro expansion is one of several options for extending a language. Other common options include template meta-programming (TMP), and staged compilation. TMP and staged compilation both have as an advantage the use of semantic information in the specification of transformations, whereas macros are mostly limited to syntactic transformation. In the case of FOBS-X, there was a concern, with extension transformation available to the user, that the fundamental semantics of the language might be altered. In this light the limited nature of macro expansion seems a better fit to FOBS-X.

One of the major problems faced in the implementation of FOBS-X was scoping rules. Purely lexical scope, which is often used in modern functional languages, does not take into account the dynamic elements needed in object-oriented dynamic message binding. This problem, and the solution are discussed further in Section 7. The solution in FOBS-X is a hybrid dynamic-static scoping rule that searches dynamically through an inheritance hierarchy, and also statically through a set of nested block structures.

3. LANGUAGE FEATURES

Probably the characterizing feature of functional languages is referential transparency. In many respects, the referential transparent character of FOBS-X places it closer to the functional paradigm, rather than to the procedural aspects of most common object-oriented languages. FOBS-X partially shares several other features with functional languages.

Gil de Lamadrid & Zimmerman [23] present the formal semantic description of FOBS, the forerunner of FOBS-X. Like FOBS, FOBS-X is stateless, and as such is referentially transparent. This is a strength of the FOBS-X language, and other referentially transparent languages. It has been argued that the transparency of the code produces code that is more reliable. It also leads to code that can be more easily automatically separated into separate processes and distributed, allowing a large degree of parallelism. Referential transparency also implies a smaller dependency on a particular platform, allowing code to be more easily ported from one machine to another.

The ability to define higher-order functions is considered as a strength in functional languages. It allows a programmer to abstract out useful computational patterns. In FOBS-X the equivalent is to write a high-order FOB, meaning a FOB that takes another FOB as a parameter, or returns a FOB as its result. This is trivially possible in FOBS-X, since everything is a FOB, and no distinction is made between a primitive FOB, or a user defined FOB.

Automatic currying is another strength of many functional languages. It allows a programmer to create partially applied functions that can be completely customized at a later time. In currying, the under-application of a function results in a return value which is a function with the remaining parameters still unbound. In FOBS-X the under-application of a FOB also results in a FOB with the remaining parameters unbound. The mechanism used, however, differs substantially from currying. In currying, application of a function to an actual argument results in argument substitution. This results in a structure that is smaller than before the application. The application of a FOB to an actual argument results in a binding being added to the applied FOB, producing a structure that is larger than before the application. This process is discussed further in Section 7.

Functional languages are usually classified as either lazy or eager evaluation languages, based on their parameter passing mechanisms. FOBS-X is a lazy evaluation language, in the sense that actual parameters are passed as a closure, and are evaluated as they are used. As with most lazy evaluation languages, this gives the language a certain degree of efficiency, and allows infinite data structures. FOBS-X, however, is lacking a memo feature for its closures. This was omitted because with the hybrid dynamic-static scoping rules of FOBS-X, the memo feature would have been far more complex than with most purely statically scoped languages. Without a memo feature, the resulting parameter passing mechanism is essentially an implementation of the pass-by-name method.

Turning to the object-oriented side of FOBS-X, the choice was made to make FOBS-X a single inheritance language, as opposed to a language allowing multiple inheritance. In FOBS we characterize inheritance as a simple and elegant operation. Multiple inheritance would introduce several complexities, such as common ancestor, and name clash problems that would have to be dealt with. Further the advantages of multiple inheritance can be relatively easily implemented in single inheritance languages.

Polymorphism is a mechanism introduced into languages to improve code reuse. Since FOBS has only one data type, it is clear that all variables are polymorphic. Operators would also appear to be polymorphic, but are actually constrained by their intrinsic type requirements. For example, an addition operator would require two FOB operands that must represent numeric quantities. To help with implicit type requirements FOBS uses its library structure to provide tools for subtype polymorphism.

Although implicit conversion is completely absent from FOBS, explicit conversion functions are provided, as appropriate. The decision to leave out coercion was made observing that most useful coercions can be incorporated into a well constructed inheritance hierarchy.

4. RELATED WORK

Several researchers have built hybrid language systems, in an attempt to combine the functional and object-oriented paradigms. Ng and Luk [11] have written a survey of languages that integrate object-oriented, logical, and functional languages. This survey, although less recent, is an excellent review, and almost all of the material presented is still relevant. The paper begins by describing the significant characteristics of each paradigm, and continues by examining several languages that have integrated several of the paradigms. Many of these languages are described below.

Ng and Luk list as important features of object-oriented languages inheritance mechanisms and inter-object parallelism. Inter-object parallelism comes from the degree of object independence supported by the language.

Functional languages are characterized by comparing them to languages like ML [12], and Haskell [19]. The defining feature of these languages is referential transparency. They also support, to varying degrees, higher level functions, lazy evaluation, pattern matching, strong typing, and abstract data types. Both object-oriented and functional characteristics are considered as we discuss the works below.

Yau et al. [1] present a language called PROOF. This language, that allows assignment, is not pure functional, and lacks referential transparency. PROOF tries to fit objects into the functional paradigm with little modification to take into account the functional programming style. There are several other programming languages that suffer the same problems. The work on CLOS by Gabriel et al. [2] is an add-on to Common LISP. As with Common LISP CLOS uses eager evaluation.

The language D by Alexandrescu [7] is a rework of the language C, transforming it into a more natural scripting language similar to Ruby [20] and JavaScript. It includes object-orientation, and single inheritance. It is also possible to maintain a functional computation model, by using immutable objects. It is a language that is extensively inclusive, and the style of programming is left to the user's preference. This leads to a language with no clear computation model.

Adding features to an existing language is a common way of bringing in new paradigms. The language Oz [22] is concurrency-oriented, prominently featuring threading. It has a distinct functional feel. By default, Oz uses eager evaluation, and allows only immutable objects. However, explicit declarations allow the creation of lazy parameters, and data objects called *cells* that are mutable. The cell construct provides the foundation for building classes and writing object-oriented code. However, as with the language D, the addition of foreign features to an otherwise mostly clean computation model, diminishes the consistency of the model.

Scala by Odersky et al. [8] is a language compiled to the Java Virtual Machine, which claims to implement a hybrid of functional and object oriented paradigms, but tends toward the imperative language end of the spectrum. A class based language that is proposed as a tool to write web-servers, Scala is implemented as a small core language, and many of its capabilities are implemented in the library. FOBS-X has this same structure, allowing the capabilities of the language to be easily extended.

Two languages that seek to preserve functional features, such as referential transparency, are FLC by Beaven et al. [3], and FOOPS by Goguen and Mesegner [4]. FOOPS is built around the addition of ADTs to functional features. FOOPS is stateless, and uses constraint equations to approximate state. These equations describe the state of an object in terms of the state of another object. FOOPS supports a form of inheritance, by allowing one module to import another, receiving all of the attributes that enter. This type of inheritance is very similar to the type used in FOBS-X.

We feel that the approach of FLC is much cleaner than that of FOOPS. In FLC, classes are represented as functions. This is the basis for FOBS-X also. In FOBS-X we have, however, removed the concept of the class. In a stateless environment, the job of the class as a "factory" of individual objects, each with their own state, is not applicable. In stateless systems a class of similar objects is better represented as a single prototype object that can be copied with slight modifications to produce variants.

Several hybrid languages have explored the role of static and dynamic typing. The language Needle [13] is a variation to the semantics of ML to add in object orientation. Because of this, it maintains referential transparency and first-class functions. It also is statically typed, and, as with ML, implements type inference. It has the advantages of strong typing, in terms of error checking, while at the same time relieving the user of the burden of explicit type checking.

Another language that explores typing is Thorn from Wrigstad et al. [14]. An imperative style language implemented on the JVM, Thorn implements a like-type system. In like typing, type equivalence is determined by structural equivalence. Two objects are considered type equivalent if they have not only the same attributes, but also the same methods. In this way a polymorphic operation can be ensured that a like-typed actual operand is fully substitutable for a formal operand. Both Needle and Thorn are intriguing languages, in terms of their type systems, but they require more extensive static analysis than is performed in scripting languages, slowing and complicating the execution of scripts.

Our goal is to extend the functional-object-oriented paradigm to scripting languages. Ousterhout [15] characterizes scripting languages as being dynamically typed, interpreted, and lacking complex data structures. Functional languages often share these same properties, and in fact languages like KAVA by Bothner [5] have been proposed as scripting languages. Features from functional languages have also been incorporated in more main-stream non-functional language such as JavaScript.

Object-oriented scripting languages such as Python [17] are also available. Python is a class based language, with minimalistic syntax, reminiscent of LISP type languages. Although mostly object-oriented, its support for functional programming is decent, and includes LISP characteristics such as anonymous functions and dynamic typing.

Recent revisions to Python [21] have enhanced the functional features of the language. In Python functions are viewed as objects, and so there is no problem defining higher-order functions simply using the same mechanisms available for defining functions with any other type of object as parameter or return value. But in addition, the Python library now contains a module containing function operators, allowing the programmer to more easily compose and transform higher-level functions. Python has also been endowed with support for lazy data structures, a feature often implemented in functional languages, such as Haskell, using lazy evaluation. In Python this feature is implemented by defining generator functions, which are coroutines capable of accessing potentially infinite streams of data that are only partially evaluated.

Python has influenced several other languages , such as ECMAScript[16]. ECMAScript, often implemented as JavaScript, is a prototype based object oriented language. Object prototypes, that take the place of classes, provide shared attributes to the objects constructed from them. ECMAScript was originally designed to be a web server scripting language. Syntactically it has inherited the Java, and C style syntax. The interface to the environment is through built in library objects. The emphasis in EMCAScript is object orientation and imperative programming.
Both Python and JavaScript lack referential transparency. As discussed previously, we consider this as one of the important advantages of FOBS-X. In the design of FOBS-X, we also felt that a simpler data structure could be used to implement objects and the inheritance concept, than was used in these two popular languages. FOBS-X combines object orientation and functional programming into one elegant hybrid, making both tools available to the user. Unlike languages like Python or FOOPS, this is not done by adding in features from both paradigms, but rather by searching for a single structure that embodies both paradigms, and unifies them.

5. FOBS-X DATA

FOBS-X is built around a core language, core-FOBS-X. Core-FOBS-X has only one type of data: the FOB. A *simple FOB* is a quadruplet,

[*m i* -> *e* ^ □]

The FOB has two tasks. Its first task is to bind an identifier, *i*, to an expression, *e*. The *e-expression* is unevaluated until the identifier is accessed. Its second task is to supply a return value when invoked as a function. □ (the □*-expression*) is an unevaluated expression that is evaluated and returned upon invocation.

The FOB also includes a modifier, *m*. This modifier indicates the visibility of the identifier. The possible values are: "`+", indicating public access, "`~", indicating protected access, and "`$", indicating argument access. Identifiers that are protected are visible only in the FOB, or any FOB inheriting from it. An argument identifier is one that will be used as a formal argument when the FOB is invoked as a function. All argument identifiers are also accessible as public.

As an example, the FOB

```
[`+x -> 3 ^ 6]
```

is a FOB that binds the variable *x* to the value 3. The variable *x* is considered to be public, and if the FOB is used as a function, it will return the value 6.

There are several *primitive FOBs*. Primitive data is defined in the FOBS-X library. The types *Boolean, Char, Real,* and *String* have constants with forms close to their equivalent *C* types.
The *Vector* type is the main data structure in FOBS-X, and can be thought of as a combination of a list and an array, similar to the structure of the same name in the *Java* API [18]. However, a significant semantic difference between the two implementations is that the *Java Vector* is mutable, and the FOBS-X *Vector* is not. This means that the FOBS-X *Vector* can be manipulated as a list, using the usual LISP type operations of *car*, *cdr*, and *cons*, but when used as an array, although the access of an element is allowed, the mutate operation is not. The best approximation to the mutate operation is the creation of a brand new modified vector.

Vector constants are of a form close to that of the ML list. For example, the vector

```
["abc", 3, true]
```

represents an ordered list of a string, an integer, and a Boolean value.

A special FOB, called the *empty FOB*, is often returned as an error indication, and is denoted by the identifier consisting of the underscore character.

6. OPERATIONS ON FOBS

There are three operations that can be performed on any FOB. These are called *access*, *invoke*, and *combine*.

6.1. Access operator

An access operation accesses a variable inside a FOB, provided that the variable has been given a public or argument modifier. As an example, in the expression

```
[`+x -> 3 ^ 6].x
```

the operator "." indicates an access, and is followed by the identifier being accessed. The expression would evaluate to the value of *x*, which is 3.

6.2. Invoke operator

An invoke operation invokes a FOB as a function, and is indicated by writing two adjacent FOBs. In the following example

```
[`$y -> _ ^ y.+[1]] [3]
```

a FOB is defined that binds the variable *y* to the empty FOB and returns the result of the expression *y* + 1, when used as a function. To do the addition, *y* is accessed for the FOB bound to the identifier "+", and this FOB is invoked with 1 as its actual argument. The full FOB defining *y* is then invoked, passing as actual argument the value 3. The result of the invocation is 4.

In invocation, it is assumed that the second operand is a vector. This explains why the second operand in the above example is enclosed in square braces. Invocation involves binding the actual argument to the argument variable in the FOB, and then evaluating the □□expression, giving the return value.

Actual arguments are passed as *thunks*, or closures, that contain the argument expression, and the evaluation environment. The actual arguments are evaluated only as needed in the invoked FOB. In addition, it is possible to supply fewer, or more arguments required by the FOB, resulting in over application, or under application. The result is to build a FOB with some of the argument variables left unbound, or discard the excess actual arguments, respectively.

6.3. Combine operator

A combine operation is indicated with the operator ";". It is used to implement inheritance. In the following example

```
[`+x -> 3 ^ _] ; [`$y -> _ ^ x.+[y]]
```

(1)

two FOBs are combined. The *super-FOB* defines a public variable *x*. The *sub-FOB* defines an argument variable *y*, and a □-expression. Notice that the sub-FOB has unrestricted access to the super-FOB, and is allowed access to the variable *x*, whether modified as public, argument or protected.

The FOB resulting from Expression (1) can be accessed, invoked, or further combined. For example the code

```
([`+x -> 3 ^ _] ; [`$y -> _ ^ x.+[y]]).x
```

evaluates to 3, and the code

```
([`+x -> 3 ^ _] ; [`$y -> _ ^ x.+[y]]) [5]
```

evaluates to 8.

Multiple combine operations result in *FOB stacks*, which are compound FOBs. For example, the following code creates a FOB with an attribute *x* and a two argument function that multiplies its arguments together. The code then uses the FOB to multiply 9 by 2.

```
([`+x -> 5 ^ _] ; [`$a -> _ ^ _] ; [`$b -> _ ^ a.*[b]]) [9, 2]                    (2)
```

In the invocation, the arguments are substituted in the order from top to bottom of the FOB stack, so that the formal argument *a* would be bound to the actual argument 2, and the formal argument *b* would be bound to 9.

When the operations *combine*, *invoke*, and *access* are used together, without parentheses, precedence and associativity rules are used to eliminate ambiguity. The operations *access*, and *invoke* have equal and high precedence, while the operator *combine* has a low precedence. The operations *access*, and *invoke* are also left associative, while the operator *combine* is fully associative.

7. CORE-FOBS-X SEMANTICS ISSUES

Expression evaluation in FOBS-X is fairly straight forward. Three issues, however, need some clarification. These issues are: the semantics of the redefinition of a variable, the semantics of a FOB invocation, and the interaction between dynamic and static scoping.

7.1. Variable overriding

A FOB stack may contain several definitions of the same identifier, resulting in overriding. For example, in the following FOB

```
        [`$m -> 'a' ^ m.toInt[]] ; [`+m -> 3 ^ m]
```

the variable *m* has two definitions; in the super-FOB it is defined as an argument variable, and in the sub-FOB another definition is stacked on top with *m* defined as a public variable. The consequence of stacking on a new variable definition is that it completely overrides any definition of the same variable already in the FOB stack, including the modifier. In addition, the new return value becomes the return value of the full FOB stack.

7.2. Argument substitution

As mentioned earlier, the invoke operator creates bindings between formal and actual arguments, and then evaluates the □-expression of the FOB being invoked. At this point we give a more detailed description of the process.

Consider the following FOB that adds together two arguments, and is being invoked with values 10 and 6.

```
        ([`$r -> 5 ^ _] ; [`$s -> 3 ^ r.+[s]]) [10, 6]
```

The result of this invocation is the creation of the following FOB stack

```
        [`$r -> 5 ^ _] ;
        [`$s -> 3 ^ r.+[s]] ;
        [`+r -> 6 ^ r.+[s]] ;
        [`+s -> 10 ^ r.+[s]]
```

In this new FOB the formal arguments are now public variables bound to the actual arguments, and the return value of the invoked FOB has been copied up to the top of the FOB stack. The return value of the original FOB can now be computed easily with this new FOB by doing a standard evaluation of its □-expression, yielding a value of 16.

Complications arise when a FOB is evaluated with either too few or too many actual arguments. These two conditions are usually termed under-application, and over application, respectively.
As an example of under-application, consider the above example, invoked with only one argument.

```
        ([`$r -> 5 ^ _] ; [`$s -> 3 ^ r.+[s]]) [10]                    (3)
```

In under-application, argument substitution proceeds as normal starting with the top argument variable in the FOB stack, and working down the stack until all actual arguments are exhausted. The result for Expression (3) would be

```
[`$r -> 5 ^ _] ;
[`$s -> 3 ^ r.+[s]] ;
[`+s -> 10 ^ r.+[s]]
```

The result of evaluating the □-expression of this FOB would use the value of 10 for *s*, and the default value of 5 for the argument *r*.

Turning now to over-application, we consider the same example invoked with three arguments.

```
([`$r -> 5 ^ _] ; [`$s -> 3 ^ r.+[s]]) [10, 6, 0]
```

In over-application, argument substitution proceeds as normal starting with the top argument variable in the FOB stack, and working down the stack until all formal arguments are exhausted. At this point any remaining actual arguments would be discarded. The result from the example would be the same as in Expression (2).

A variant to this procedure occurs when the FOB stack is a single primitive FOB, as in the expression

```
[1, 2] [0]
```

In this case, whether there is over application, under-application, or not, the actual arguments are passed to the primitive FOB to be handled as it chooses.

7.4. Apply operator

FOBS-X includes a variant of the invoke operation, not included in the original language FOBS. The apply operator does a partial invocation of a FOB, and is intended for use in the construction of higher order functions. Simply put, *apply* does argument substitution, but does not complete the evaluation of the resulting FOB.

To illustrate the workings of the apply operation, consider Expression (3). As explained, this expression evaluates to the value 15. Now consider the same example, with the invoke operation replaced by the apply operation.

```
([`$r -> 5 ^ _] ; [`$s -> 3 ^ r.+[s]]) ;; [10]
```

The operation *apply* is indicated by the double semicolon lexicon. This expression evaluates to the FOB stack

```
[`$r -> 5 ^ _] ;
[`$s -> 3 ^ r.+[s]]) ;
[`+s -> 10 ^ r.+[s]]
```

which is the original stack after argument substitution. In effect, the apply operator has turned a function of two arguments, *r*, and *s*, into a function of one argument, *r*. This new function can later be invoked with the remaining argument.

7.3. Variable scope, and expression evaluation

Scoping rules in FOBS-X are, by nature, more complex than scoping used in most functional languages. Newer functional languages, such as *Haskell* and ML, typically use lexical scoping. Dynamic scoping associated with older dialects of LISP, has recently fallen out of favor.

Pure lexical scoping does not cope well with variable overriding, as understood in the object-oriented sense, which typically involves dynamic message binding. To address this issue, FOBS-X uses a hybrid scoping system which combines lexical and dynamic scoping.
Consider the following FOB expression.

```
[`~y -> 1^_] ;
          (4)
[`~x ->
          [`+n -> y + m ^ n] ;
          [`~m -> 2 ^_]
^_] ;
[`~z -> 3 ^x.n]
```

This expression defines a FOB stack that is three deep, containing declarations for a protected variable *y*, with value 1, a protected variable *x* with a FOB stack as its value, and a protected variable *z* with the value 3 as its value. The stack that is the value of *x* consists of two FOBs, one defining a public variable *n*, and one defining a protected variable *m*.

We are currently mostly interested in the FOB stack structure of Expression (4), and can represent it graphically with the *stack graph*, given in Figure 2. In the stack graph each node represents a simple FOB, and is labeled with the variable defined in the FOB. Three types of edges are used to connect nodes: the *s-pointer*, the *t-pointer*, and the □*-pointer*.

The s-pointer describes the lexical nested block structure of one FOB defined inside of another. The s-pointer for each node points to the FOB in which it is defined. For example *m* is defined inside of the FOB *x*.

The t-pointer for each
node points to the super-FOB of a FOB. It describes the FOB stack structure of the graph. In Figure 1 there are basically two stacks: the top level stack consists of nodes *z*, *x*, and *y*, and the nested stack consisting of nodes *m*, and *n*.

The □-pointer is a back pointer, that points up the FOB stack to the top. This provides an easy efficient mechanism for finding the top of a stack from any of the nodes in the stack.

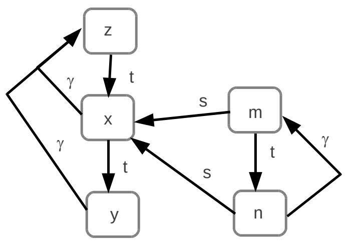


Figure 1. Stack graph of Example (4).

If the FOB *z* were invoked, it would access the FOB *n* for the value of *n*. This would cause the expression *y* + *m* to be evaluated, a process that demonstrates the use of all three pointers.

The process of resolving a reference in FOBS-X first examines the current FOB stack. The top of the current stack is reached by following the □-pointer. Then the t-pointers are used to search the stack from top to bottom. If the reference is still unresolved, the s-pointer is used to find the FOB stack enclosing the current stack. This enclosing stack now becomes the current stack, and is now searched in the same fashion, from top to bottom, using the □-pointer to find the top of the stack, and the t-pointers to descend to the bottom.

To summarize this procedure for the example, to locate the definition of the variable *y*, referenced in the FOB *n*, the □-pointer for *n* is followed up to the FOB *m*, this FOB is examined, and then its t-pointer is followed down to the FOB *n*, which is also examined. Not having found a definition for the variable *y*, the s-point for FOB *n* is followed out to the FOB *x*, and then the □-pointer is followed up to the FOB *z*. FOB *z* is examined, and its t-pointer is traversed to FOB *x*, which is also examined. Then the t-point for FOB *x* is finally followed down to the FOB *y*, which supplies the definition of *y* needed in the FOB *n*.

8. RUNNING A FOBS-X PROGRAM

The language FOBS-X is built on a core language. This core language is small, semantically concise, and has a light-weight syntax. Core-FOBS-X is interpreted and executed. An interpreter for the core language has been implemented in Perl. On top of the core, there is a family of FOBS-X extensions. The *standard extension* is considered to be the full FOBS-X language. In addition the user can create further extensions to FOBS-X, called *applied extensions*. The Standard extension has been completely developed, and is discussed in Section 14.

Extensions are processed by macro expansion and converted into core-FOBS-X before interpretation. The Macro Processor reads in user code and the standard extension, does textual substitution, and produces pure core-FOBS-X code. This code is then parsed, producing a syntax tree as an intermediate representation. Then the syntax tree is evaluated producing a FOB as the result. In the evaluation process primitive FOBs are pulled out of the FOBS-X object library, as needed. The final evaluated FOB is finally output to the user.
The FOBS-X library is a collection of primitive FOBs described in Perl. Most FOBs in the library are implementations of primitive data, both simple and composite. A further discussion of the library is given in Section 9.

9. THE LIBRARY

The FOBS-X library contains definitions for both *primitive FOBs*, and *utility FOBs*. These FOBs are all used to define the primitive data types of the FOBS-X language.

9.1. Utility FOBs

Utility FOBs are an organizational tool with similarities to *mix-in classes* described by Page-Jones [9]. Operations that are common among several FOBs are collected into utility FOBs. The utility FOBs are then stacked into other primitive FOBs in the library. Table 1 lists all utility FOBs in the library, along with the operations they supply.

Table 1. FOBS utility FOBs.

Utility FOB	**Operation**	**Description**
FOBS	FOBS.isEmpty[x]	Return the boolean value of the expression *x* = _
Numeric	n.+[x]	Add a numeric FOB *n* to a numeric FOB *x*
	n.-[x]	Subtract a numeric FOB *x* from a numeric FOB *n*
	n.*[x]	Multiply a numeric FOB *n* by a numeric FOB *x*
	n./[x]	Divide a numeric FOB *n* by a numeric FOB *x*
Comparable	c.<[x]	Return the boolean value of the expression *c* < *x*
	c.>[x]	Return the boolean value of the expression *c* > *x*
	c.<=[x]	Return the boolean value of the expression *c* □ *x*
	c.>=[x]	Return the boolean value of the expression *c* □ *x*
Eq	e.=[x]	Return the boolean value of the expression *c* □ *x*
	e.!=[x]	Return the boolean value of the expression *c* □ *x*
Printable	p.toString[]	Return the *print-string* for printable FOB *p*

The FOB *FOBS* is only partially described in Table 1. It will eventually contain operations for interacting with the outside environment, enabling the language to be used for scripting. The FOB *Printable* provides a FOB with a function that returns a string that can be used to print the value of the FOB.

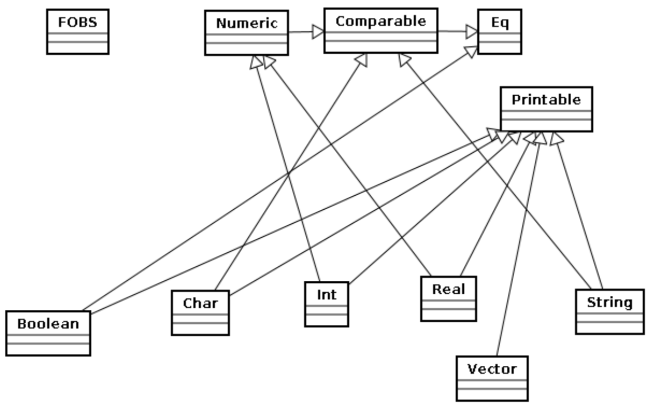


Figure 2. Mix-in structure of the FOBS library.

The primitive FOBs of the library mix in the utility FOBs to provide themselves with necessary operations. Figure 2 shows the mix-in connections in the library, using UML.

9.2. Primitive library operations

In addition to the operations supplied by the utility FOBs, each primitive FOB also has its own specific set of operations. These are given in Table 2.

Table 2. Primitive operations.

Primitive FOB	**Operation**	**Description**
Boolean	b.if[x, y]	If boolean value *b* is true, return *x*, otherwise return *y*
	b.&[x]	Return the boolean value of the expression *b* □ *x*
	b.|[x]	Return the boolean value of the expression *b* □ *x*
	b.![]	Return the boolean value of the expression □*b*
Char	c.toInt[]	Return the ASCII code for character *c*
Int	i.%[x]	Return the integer value of the expression *i* mod *x*
	i.<<[x]	Return the integer value of *i* shifted left by *x* bits
	i.>>[x]	Return the integer value of *i* shifted right by *x* bits
	i.&[x]	Return the integer value of *i* bit-wise anded with *x*
	i.|[x]	Return the integer value of *i* bit-wise ored with *x*
	i.^[x]	Return the integer value of *i* bit-wise xored with *x*
	i.toReal[]	Return the real value for integer *i*
	i.toChar[]	Return the character for ASCII code *i*
Real	r.floor[]	Return the floor of real number *r*
	r.ceil[]	Return the ceiling of real number *r*
String	s.+[x]	Return the concatenation of the strings *s* and *x*
	s.length	Return the length of the string *s*
	s.toVector[]	Return a vector of the characters composing the string *s*
	String.fromChars v	Return a string formed by the vector of characters *v*
Vector	v[i]	Return the *i*th element of vector *v*
	v.length[]	Return the length of vector *v*
	v.+[x]	Return a vector with car *x* and cdr *v*
	v./[]	Return the car of vector *v*
	v.%[]	Return the cdr of vector *v*
	v.-+[i, x]	Return a copy of the vector *v*, except that the *i*th element is *x*

These operations include the usual arithmetic, logic, and string manipulation operations. In addition, conversion functions provide conversion from one primitive type to another, when appropriate. Vector functions include the usual list manipulation functions, as well as array access functions. A vector FOB used as a function returns the *i*th element of the vector, and the *i*th element of the vector can be set to a new value, in a copy of the vector, using the "-+" operator, which we call the *replace* operator.

10. AN EXAMPLE

We present a larger example to demonstrate how FOBS code might be used to solve more complex programming problems. In this example we build the binary search tree, shown in Figure 3(a)., and then search it for the character 'f'. In the FOBS-X solution, we construct a FOB with the structure shown in Figure 3(b). The *Node* FOB is the prototype that is copied to create *Node* objects. The method called *r.v.* indicates the return value of the FOB.

The FOBS-X code for the example follows.

```
## definition of the NodeMaker FOB
(5)
([NodeMaker ->
          [`$lt -> _ ^ _] ;
          [`$rt -> _ ^ _] ;
          [`$in -> _ ^ _] ;
          [`~Node ->
                    [`~left -> lt ^ _] ;
                    [`~right  -> rt ^ _] ;
                    [`~info -> in ^ _] ;
                    [`~_ -> _ ^
                              ([`~a1 -> info.=[key] ^ _] ;
                              [`~a2 -> FOBS.isEmpty[left].|[a1].if[false, left[key]]
                                        ^ _];
                              [`~a3 -> FOBS.isEmpty[right].|[a1].if[false,
                                        right[key]]^_];
                              [`+a4 -> a1.|[a2].|[a3] ^ _]).a4] ^ _]
          ^ Node] ;
## build the tree
[`+tree ->
          NodeMaker['m', NodeMaker['g', NodeMaker['f', _, _],
                    NodeMaker['j', _, _]], NodeMaker['p', _, _]]

          ^_]
## search for 'f'
.tree['f']
#.
```

This code has two types of elements: a FOB expression, and macro directives. Macro directives begin with the “#” character, and are expanded by the macro preprocessor. The two seen here are the comment directive, “##”, and the *end of expression* directive, “#.”.

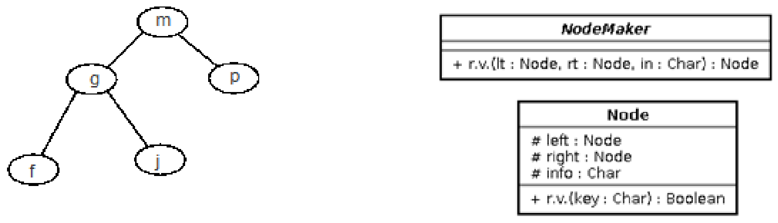

m
g
p
f
j
NodeMaker
+ r.v.(lt : Node, rt : Node, in : Char) : Node
Node
# left : Node
# right : Node
# info : Char
+ r.v.(key : Char) : Boolean

Figure 3. (a) Example binary search tree. (b) Example FOB structure.

The top-level FOB defines a FOB *NodeMaker*, and the search tree, *tree*. The top-level FOB is accessed for the tree, and it is searched for the value “f”, using the invoke operator.

NodeMaker creates a FOB with the required attributes, and a return value that does a search. The return value uses the local variables *a1*, *a2*, *a3*, and *a4* to save the results of the comparison with the node, the left child, the right child, and the final result, respectively.

Often it is necessary to compare a FOB with the empty FOB, as in the example, where it must be determined if the two subtrees are empty. This is done using code like FOBS.isEmpty[left] that uses a function from the library FOB *FOBS*. In Section 9 we saw that using the “=” operator to perform this comparison is only possible with FOBs that inherit from the utility FOB *Eq*. Since the empty fob contains nothing, and inherits nothing, it cannot be compared with the “=” operator, and the special *isEmpty* operator is supplied instead.

FOBS-X code is run by feeding FOBS-X expressions to the interpreter. Each FOBS-X expression produces as its value a single FOB, which is returned back to the user and printed. This example would cause the value true to be printed.

11. MACRO EXPANSION

FOBS-X allows extensions to its syntax using a macro processor. Macros in FOBS-X are quadruplets <$S_1 \rightarrow S_2$: *P, d*>. The semantics of the quadruplet notation is as follows.

- S_1: the search string, which includes wild-card tokens
- S_2: the replacement string, which includes wild-card tokens.
- *P*: the priority of the macro, with priority 19 being highest priority, and priority 0 being the lowest.
- *d*: the direction of the scan, with *r* indicating right-to-left, and *l* indicating left-to-right.

The quadruple gives a rewrite rule in which an occurrence of a search string is replaced by a replacement string. Incorporated into the rule is an operator precedence and associativity. As an example, a FOBS-X macro to turn multiplication into an infix operator might appear as follows.

< #?multiplicand * #?multiplier →
(6)
(#?multiplicand .* [#?multiplier]) : 18 , l >

As is often the case in macros, the above macro contains wild-card tokens. These tokens are distinguished from normal literal tokens by the fact that they begin with the "#" character. Wild-card tokens are named, in order to allow references to their bindings. In the example the wild-card tokens are #?multiplicand, and #?multiplier.

Interpreting the example macro, we start with the search string, S_1 = #?multiplicand * #?multiplier. To successfully match this string, first the wild-card token #?multiplicand is used to match what is called an *atom*. Then a token "*" is matched, and finally another atom must match the wild-card token #?multiplier.

Following the successful match, the replacement string, S_2 = (#?multiplicand . * [#?multiplier]) is used to replace the matched text, with a string composed of the text matched by the wild-cards, and a few other delimiting characters.

As an example, suppose that the search sting matched the text x * y. The wild card *#?multiplicand* would, as a result, be bound to the text x, and the wild-card *#?multiplier* would be bound to the text y. substituting these bindings into the replacement string, would yield the replacement text (x.*[y]).

From the rule in Example (6) we see that a priority of 18 is specified. The priority of the macro is used to establish the precedence of operators defined as macros. This macro defines an operator with an extremely high precedence.

To implement priority in the macros, the FOBS-X macro processor does a multi-layered expansion. In this process macro definitions are queued by their priority. Each rule is then matched in the order in which it occurs in the queue. If written correctly, high priority macros consume operands, and then their expansions become operands for lower priority macros, implementing a precedence hierarchy.

Direction in the macro is used to implement associativity for macro defined operators. In Example (6) it is indicated that the direction is left-to-right. This macro rule would then be applied by searching for the search string from the left of a FOBS-X expression to its right.

Right-to-left direction indicates that the search string is matched moving from the right of the FOBS-X expression to its left. If written correctly, direction can be used to implement associativity. For example, with left-to-right direction, the expansion of the leftmost occurrence of an operator would become an operand for the next occurrence of the operator to the right.
It Is possible to have nested macros. This happens in one of two ways: The input text may contain a macro invocation inside of another invocation, or a macro definition may contain an invocation of another macro in its replacement string. In either case, once the macro has been expanded, the replacement text still contains macro invocations. To handle this situation, each time a rule triggers, the macro processor pushes the replacement text on to the front of the input, so that any further processing will reexamine the replacement text, as well as any remaining input.

As a second example of macro rules consider the definition of an *if* expression:

```
        < if { #*x } then { #*y }  else { #*z } →
                (#*x) .if[ #*y , #*z ] : 3 , l >
```

Notice that in this example the wild-card token prefixes contain a “*” as opposed to a “?”. A wild-card token beginning with a “?” is termed a *single wild-card token*, and matches a single atom. A wild-card token beginning with a “*” is termed a *multiple wild-card token*, and matches an arbitrarily long list of atoms. Both wild-card tokens match text in units of *atoms*.

The term *atom* is often used to refer to the simple tokens of a programming language, such as identifiers, constants, or delimiters. In FOBS-X the term is used in a slightly more extended manner. Certainly simple tokens are atoms. However, in addition we allow the macro processor to process compound atoms, consisting of a sequence of atoms enclosed in bracketing symbols. The recognized bracketing symbols are “(“, “)”, “{“, “}”, “[“, and “]”.

To allow compound atoms to contain nested bracketing symbols, the macro processor is balance sensitive. That is to say that any atom beginning with a bracketing symbol does not end until all bracketing symbols in the atom are balanced. For example the string {(m.n) [x, y, z]} is considered a single atom. Although bracketing symbols are contained in the atom, the atom does not terminate until the beginning “{“ is balanced with the ending “}”.

12. MACRO FILES

When defining a macro, the user can simply embed the macro definition into a standard FOBS-X source file. More commonly the macro code is separated form the FOBS-X source code into a macro file, and loaded into the source code using a mechanism similar to the *C #include* directive. The syntax for macro definition is demonstrated by the following example that defines the multiplication macro from Example (6).

```
        ## numeric multiply operator
                (7)
        #defleft
                #?op1 * #?op2
        #as
                ( #?op1 .:*: [ #?op2 ] )
        #level
                9
```

```
#end
#defleft
        :*:
#as
        *
#level
        0
#end
```

There are two macro definitions in this example. Each definition begins with the syntax #defleft, and ends with the directive #end. The markers #as, and #level separate the search string from the replacement string, and the replacement string from the priority, respectively. The direction of the macro is indicated be the beginning marker: #defleft for left-to-right, and #defright for right-to-left.

The example illustrates a common technique used when defining macros that simply move an operator from a prefix position to an infix position. Simply moving the operator leaves it vulnerable to being rematched by the same macro rule again, resulting in infinite macro expansion. The solution is to split the process of replacement between two macro rules. The top rule does indeed move the operator “*”, but also changes it to the string “:*:”. After the top macro rule has finished its work and has been removed from the priority queue, the bottom macro, which is still in the priority queue because of its lower priority, changes the name of the operator form “:*:” back to just “*”.

13. MACRO PROCESSING DETAILS

Macros are converted to an internal representation to make the matching process more efficient. Macro definitions, as in Example (7) are first split into the components of the macro, using the directive keywords. The four components, the search string, the replacement string, the priority, and the direction are then converted into different structures, that are stored in a macro table. This section discusses the different component representations.

13.1. The Search String

A search string, S_1, is converted into a deterministic finite automata (DFA). The DFA corresponding to Example 7 is shown in Figure 4. In this DFA an atom is consumed each time a transition is traversed. The middle transition, from state s_1 to state s_2, consumes the token “*”. The other two transitions accept input, and also produce output. They are marked with labels of the form *A/[x]*, where *A* is the input, and *[x]* is the output. The meaning of this label is that the input is from the set *A*, the set of all atoms, and the output is concatenated onto the rear of the value stored under the wild-card identifier *#?x*. In terms of Figure 4, the first transition indicates that an atom is matched, and the matched atom is concatenated onto the string matched by the wild-card *#?op1*. The last transition indicates a similar action for the wild-card *#?op2*.

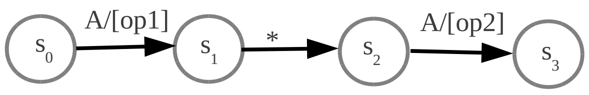

$$\begin{tikzcd} s_0 \arrow[r, "A/[op1]"] & s_1 \arrow[r, "*"] & s_2 \arrow[r, "A/[op2]"] & s_3 \end{tikzcd}$$

Figure 4. DFA version of Example 7.

Construction of the DFA from the search string is fairly straight forward. The construction begins with a single state s_0, and continues adding states to the machine, left to right. The *i*th step in the construction consumes the *i*th token off of the left of the search string, and adds the *i*th state, s_i. More specifically, let s_i be the current last state in the machine. We consider the following cases to add state s_{i+1}.

Constant: the search string currently is of the form *x* □, where *x* □□□, and □□□ □□□□ W)*, □ is the set of FOBS-X lexicons, and W is the set of wild-card identifiers. The following construct would be added to the machine.

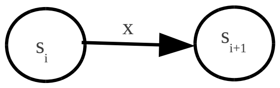


Figure 5. DFA construct for a constant.

Single wild card: the search string currently is of the form #?*x* □, where #?*x* □ W, and □□□ □□□□ W)*. The following construct is added to the machine:

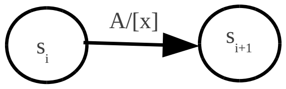


Figure 6. DFA for a single wild-card identifier.

Multiple wild card: the search string currently is of the form #**x* □, where #**x* □ W, and □□□ □□□□ W)*. This case is split into the following two sub-cases.

Succeeded by a constant: Let □ = *y* □, where *y* □□□, and □□□ □□□□ W)*. The following construct is added to the machine. Notice that no new state is added, only the new transition.

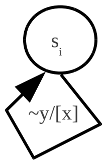


Figure 7. DFA modification for multiple wild-card succeeded by a constant.

Not succeeded by a constant: (anything that is not the first case) Nothing is added to the machine.

This multiple wild-card strategy builds a machine that matches the smallest possible string, given the next token in the search string. The last state added to the machine is made the goal state of the machine. The first state added to the machine is the start state. Any transitions missing from the machine are considered to lead to the error state.

13.2. Atoms

The DFA built by the method given in Section 13.1 consumes its input text in units of atoms. As mentioned in Section 11, an atom is a series of tokens, with balanced bracket characters, or a single data object. Data objects are all tokens other than delimiter sequences. A single wild-card consumes a single atom. A multiple wild card consumes lists of tokens with balanced parentheses. These lists are referred to as *alists*. More precisely, atoms and alists are defined by the following grammar:

> <atom> ::= <data> | { <alist> } | (<alist>) | [<alist>]
> <alist> ::= <token > <alist> | { <alist> } <alist> | (<alist>) <alist> | [<alist>] <alist> | <empty>
> <token> ::= <real> | <int> | <boolean> | <char> | <string> | <id> | <modifier> | ; | . | , | -> | ^ | ;;
> <data> ::= <real> | <int> | <boolean> | <char> | <string> | <id> | <modifier>

13.3. The Replacement String

The replacement string of a macro, S_2, is converted into a type of flowchart, called an *action diagram*. Figure 8 shows the action diagram for the replacement string from the first macro of Example (7). During the expansion process, action diagrams are traversed from the start node to the end node to generate code. As each node is visited the label of the node indicates a code piece to be written. In Figure 8, the code generation process would start by writing an open parenthesis. The label of the next node indicates that $r\{op1\}$ would be written, where $r\{x\}$ is the notation for the r-value of the identifier x. In this case the value to which *op1* is bound would be written. This would be followed by an access operator, the identifier ":*:", an open bracket, the value bound to the identifier *op2*, a close bracket, and a close parenthesis.

In order to implement wild-card tokens in a macro, the mechanism requires the cooperation of both the search string DFA, and the replacement string action diagram. The search string DFA collects and stores bindings of wild-card identifiers to strings. The replacement string action diagram access the stored bindings, retrieving the strings associated with the wild-card identifiers.

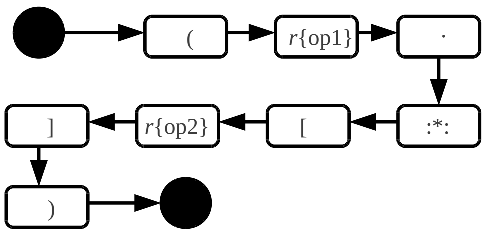


Figure 8. Action diagram for Example (7).

Construction of the action diagram is done by scanning the replacement string, and writing out nodes of the action diagram as you proceed. In particular, starting with a diagram with just a start node, n_0, nodes will be added as tokens from the replacement string are consumed. The start situation is as follows.

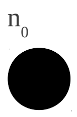


Figure 9. The start of an action diagram construction.

Constant: Suppose that the last node added was n_i, and that the current replacement string $S_2 = x$ □, where x □□□, and □□□ □□□□ W)*. The diagram is extended as follows:

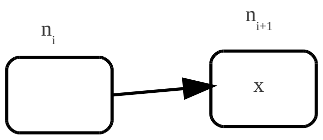


Figure 10. Adding a node for a constant to the action diagram.

The new node, n_{i+1}, causes the output of the constant *x*.
Single wild card: Suppose that the last node added was n_i, and that the current replacement string S_2 = #?*x* □, where #?*x* □ W, and □□□ □□□□ W)*. The diagram is extended as follows:

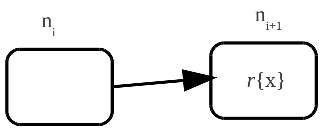


Figure 11. Adding a node to the action diagram for a wild-card identifier.

The new node, n_{i+1}, causes the output of the r-value bound to the l-value *x*.
Multiple wild card: The case for the multiple wild card #**x* is identical to the single wild card case, with #**x* substituted for *#?x*.
Termination: Suppose that the last node added was n_i, and that the current replacement string S_2 = □. The diagram is extended as follows:

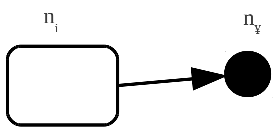


Figure 12. Terminating an action diagram.

13.4. The Macro Data Structure

Macros are stored internally as a table $M_{l,i}$, where *l* is the level or priority of the macro. Each macro is assigned an index in the table, *i*. An entry in the table is a triplet (*S*, *R*, *l*), where *S* is the search string of the macro, *R* is the replacement string, and *l* is a boolean value indicating the direction of the macro: either left or right associative.

Search String

The search string *S* is a transition table representing the DFA used to represent the search string. An entry in the table is a sextuple [*i-op*, *i*, *s*, *ns*, *p-op*, *p*].

- *i-op*: If the input is the token *x*, the choices for this field are
    - EXC – The input on the transition is ~*x*.
    - INC – The input on the transition is *x*.
    - ALL – The input on the transition is □.
- *i*: This is the input token for the transition. If the i-op is ALL, then it is unused.
- *s*: The state number of the current state.

- *ns*: The state number of the next state.
- *p-op*: If the output is the token *x*, the choices for this field are
    - PUSH – The output on the transition is [*x*]
    - NOOP – There is no output on the transition.
- *p*: This is the output token for the transition. If the p-op is NOOP, it is unused.

The sextuple entry is basically a standard entry for a transition table, containing an input, the current state, the next state, and the output. But, because there are several options for input and output in the macro processing, an input-option field (i-op) and an output-option (p-op) field have been added.

For the i-op, the reader will notice that in the construction of the DFA presented in Section 13.1, transitions can be labeled with an exclusive input, such as ~*x*, an inclusive input, such as just *x*, or wild-card transitions are labeled with any element of the set A. These correspond to the three input options.

For the p-op, output consists of pushing on to the rear of a value stored under a wild-card identifier. When the transition belongs to a wild-card the push is performed, and when the transition belongs to a constant no output is performed. These two options correspond to the two output options in the transition table.
To help clarify the table structure, Table 3 shows the transition table for the example shown in Figure 4. It demonstrates the input methods INC, and ALL. EXC would only come into play in the construction for a multiple wild-card.

Table 3. Transition table for Figure 4.

i-op	***i***	***s***	***ns***	***p-op***	***p***
ALL		0	1	PUSH	op1
INC	*	1	2	NOOP	
ALL		2	3	PUSH	op2

Replacement String

The replacement string *R* is thought of as a type of flow chart, but because the action diagram contains no decision points, it can also be thought of as a fairly standard linked list. We represent R as a list of elements representing nodes in the flow diagram. Each element is a pair (*q-op*, *q*)

- *q-op*: If the output for a node is *x*, then the choices for this field are
    - STOP – This is a terminating node.
    - START – This is a start node.
    - RVAL – Output *r*{*x*}.
    - LVAL – Output *x*.
- *q*: The output for the node. This field is ignored for STOP and START.

Reexamining the construct for the action diagram from Section 13.3, we see that there are several output options (q-op) for each node. Start and stop nodes have no output. Other nodes either output a constant, or the r-value bound to an identifier. These four options correspond to the q-ops. The representation of the example presented in Figure 8 is given as Table 4.

Table 4. List form of the action diagram of Figure 8.

q-op	*q*
START	
LVAL	(
RVAL	op1
LVAL	.
LVAL	:*:
LVAL	[
RVAL	op2
LVAL	]
LVAL	)
STOP	

13.5. Macro Expansion

We now present the details of the macro expansion process. In the previous section we discussed the data structure, *M*, used to store macros. This section describes the algorithm used, once the table has been built, and how it processes an input string *Z*.

The top level algorithm is shown in Example (8).

```
expand
        (8)
        for(l = 19; l □ 0; l--)
                expandLevel(l)
```

The procedure *expand* is used to successively process the macros in the table, one level at a time, starting with the highest priority level.

```
expandLevel(l)
        (9)
        i = 0
        while(i □ $M_{l*}$.size)
                m = $M_{l,i}$
                r = match(m)
                if(r)
                        i = 0
                else
                        i++
```

In Example (9) the code for the procedure *expandLevel* is shown. This procedure processes all of the macros for a given level, *l*. It uses the procedure *match* to perform the actual expansion of a macro. *Match* returns a result, *r*, that indicates a successful match, or a failure to match. If the match is a failure, *expandLevel* goes on to try the next macro at level *l*. If the match is a success, *expandLevel* resets *i* to zero, causing it to retry all previously processed macros at level *l* again.

```
match(m)
        (10)
        s = m.S
        r = m.R
        l = m.l
        if(l)
                return(matchLeft(s, r))
        else
                return(matchRight(s, r))
```

The code for the procedure *match* is shown in Example (10). It receives a macro, *m*, from the table and unpacks the three components of the macro. Depending on the direction of the macro, *l*, it uses either *matchLeft* to perform a left-to-right scan, or *matchRight* to perform a right-to-left scan.

```
matchLeft(s, r)
(11)
start = 0
while(start < Z.size)
        end = start
        state = 0
        B = □
        while(state □ GOAL & state □ ERR)
                entry = find(s, Z[end], state)
                if(entry □ NULL)
                        state = entry.ns
                        if(entry.p-op = PUSH)
                                push(B, Z[end], entry.p)
                        end++
                else
                        state = ERR
        if(state = GOAL)
                replace(r, B, start, end)
                return(TRUE)
        start++
return(FALSE)
```

MatchLeft, shown in Example (11), is given the transition table for the search string, *s*, and the action list for the replacement string, *r*. Two pointers are used to run through the input string: *start*, which points to the beginning of the current match attempt, and *end*, which points at the current position in the match attempt. The inner *while* loop of Example (11) is a fairly standard driver for a transition table DFA, first finding the table entry corresponding to the current state, and the current input token, *Z*[*end*]. If the procedure *find* successfully finds the entry, *matchLeft* updates the state, does a push into the wild-card store *B*, if necessary, and updates the pointer *end*. After exiting the loop if the DFA terminated at the goal state, a replace is performed.

```
matchRight(s, r)
(12)
start =Z.size – 1
while(start □ 0)
        end = start
        state = 0
        B = □
        while(state □ GOAL & state □ ERR)
                entry = find(s, Z[end], state)
                if(entry □ NULL)
                        state = entry.ns
                        if(entry.p-op = PUSH)
                                push(B, Z[end], entry.p)
                        end++
                else
                        state = ERR
        if(state = GOAL)
                replace(r, B, start, end)
                return(TRUE)
        start--
        return(FALSE)
```

MatchRight, shown in Example (12), is identical to *matchLeft* except that the *start* now runs through the input string *Z* starting from its rear, and proceeding to its front.

```
find(s, atom, state)
(13)
        for(i = 0; i □ s.size; i++)
                ent = $s_i$
                iop = ent.i-op
                in = ent.i
                st = ent.s
                if(state = st)
                        if(iop = ALL)
                                return(ent)
                        else if(iop = INC & atom = in)
                                return(ent)
                        else if(iop = EXC & atom □ in)
                                return(ent)
        return(NULL)
```

Example (13) gives the code for *find*, a procedure with arguments *s*, the transition table, *atom*, the current input token, and the current state. It runs through the table unpacking each entry, checking if it matches the current state, checking the i-op validity, and if all goes well returning the entry.

```
push(B, atom, id)
        (14)
        B[jhash(id)] = concat(B[hash(id)], atom)
```

Push, shown in Example (14), pushes a token *atom* on to the rear of the entry in the hash table *B* that corresponds to the wild-card identifier *id*.

```
replace(r, B, start, end)
(15)
v = □
i = 0
while(i < r.size)
        ent = $r_i$
        q = ent.q
        qop = ent.q-op
        if(qop = LVAL)
                v = concat(v, q)
        else if(qop = RVAL)
                f = B[hash(q)]
                v = concat(v, f)
        i++
        Z[start..end – 1] = v
```

Finally, Example (15) presents the code for the procedure *replace*, that replaces the search string with the replacement string. It is given an action list, *r*, the wild-card store, *B*, and the pointers *start*, and *end*, that delimit the location of the matched string in the input text *Z*. *Replace* builds its result in the variable *v*, running through the actions in the list. Each action is unpacked, and its token is either written to *v*, if the action specifies an l-value, or the token is looked up in the wild-card store, if the action specifies an r-value.

13.6. Macro Expansion Problems

The procedure given in Section 13.5 is not an algorithm, in the sense that it may not always terminate. It Is possible to have nested macros. As explained in Section 10, this happens in one of two ways: The input text may contain a macro invocation inside of another invocation, or a macro definition may contain an invocation of another macro in its replacement string. In either case, once the macro has been expanded, the replacement text still contains macro invocations.

We have restricted the nesting of macros to a degree. It is impossible for a macro of lower priority to call a macro of higher priority. This would make little sense, since when a macro is called, presumably, all macros of higher priority have already been processed. However even with this restriction it is still easy to produce an example macro definition that results in an infinite sequence of macro expansions, such as a macro that rewrites an invocation of itself. This would cause the code in Example (9) to match the macro, rewind the input, successfully rematch the same macro, and continue resetting the input and matching the same macro forever.

Although infinite nesting of macros is a theoretical problem, it is fortune that correctly written extensions are mostly well behaved. For instance, the standard extension has several constructs that produce nested macro invocations. The most notable one is the *fob* construct. With this construct, the depth of nesting is the same as the maximum FOB stack size, and since most FOB stacks are relatively small, this keeps the nesting controlled. However, even with controlled nesting, although the macro expansion procedure will terminate, the macro nesting affects the time needed to do the expansion. Two factors affect the time for expansion significantly: the size of the input string, and the number of successfully matched macros. Both of these factors, which are affected by macro nesting, are now briefly explained.

As macros are processed, most likely the size of the input string will increase. As macros are matched, sections of the input text are removed and replaced by the replacement string. Many macros allow more compact notations for language features, and so when invoked it is common to find the macro writing out more text than was matched. This is what you find in SE-FOBS-X when infix operators are converted to the usual core-FOBS-X notation, as described in Section 14. As the input text size increases, the number of iterations of the *while* loop in Example (11) increases, and an increase in search time is observed.

The number of successful matches also affects expansion time. When a macro successfully matches, and performs a replacement, the variable *i*, in Example (9), is reset to 0, causing all macros at the current level *l* to be rematched. This increases the number of iterations performed by the *while* loop in Example (9), and results in longer search times.

14. THE STANDARD EXTENSION

The syntax in core-FOBS-X is a little cumbersome. It has been designed for ease of syntactic analysis, with minimalistic notation. It is not necessarily attractive to the programmer. Standard extension (SE) FOBS-X attempts to rectify this situation. In particular, SE-FOBS-X includes constructs to implement the following

- Allow infix notation for most operators.
- Eliminate the cumbersome syntax associated with declaring a FOB.
- Introduce English keywords to replace some of the more cryptic notation.
- Allow some parts of the syntax to be optionally omitted.

14.1. Infix operations

SE-FOBS introduces infix operations, for most of the appropriate library operators. Transformations are included for both binary and unary operators. Binary operators are transformed using rules of the form

<#?x1 ω #?x2 → #?x1 . ω [#?x2]: *p*, *l*>,

where ω is the operator, and *p* is the priority. All operators in SE-FOBS are naturally left associative, and so the direction is always *l*. As an example the expression 3 + 5 would transform to 3.+[5].

Unary operators are transformed using the rule

<ω #?x1 → #?x1 . ω []: *p*, *l*>.

As an example the expression !true would transform to true.![]. Table 5 lists all infix operators available in SE-FOBS-X. It is a fairly standard precedence table, with the only unary operator at the highest precedence, and the binary operators arranged with precedence levels mimicking the language *C*. Each operator has a parenthesized indication of the library FOB in which it is defined.

Table 5. Standard extension infix operators.

Precedence	**Direction**	**Operator**
10	Left	! (Boolean),
9	Left	* (Numeric), / (Numeric), % (Int)
8	Left	+ (Numeric), - (Numeric), + (String), + (Vector)
7	Left	<< (Int), >> (Int)
6	Left	< (Comparable), > (Comparable), <= (Comparable), >= (Comparable), = (Eq), != (Eq)
5	Left	& (Int), & (Boolean)
4	Left	| (Int), | (Boolean), ^ (Int)

14.2. FOB definition macros

SE-FOBS-X contains several macro definitions to facilitate the definition of a FOB. These macros introduce keywords to be used instead of cryptic symbols, and allow parts of the definition to be implicit. Table 6 gives the macros for FOB definitions.

Table 6. Standard extension macros quadruples for FOB constant syntax.

Search String	**Replacement String**	**Precedence**	**Direction**
public	`+	3	Left
private	`~	3	Left
argument	`$	3	Left

fob { #?m #?id
val { #*v }
ret { #*r } \ #*x }
	[#?m #?id -> #*v ^ #*r] ;
	fob { #*x }
		3	Left

fob { #?m #?id val { #*v } \ #*x }	[#?m #?id -> #*v ^ _] ; fob { #*x }	3	Left
fob { #?m #?id ret { #*r } \ #*x }	[#?m #?id -> _ ^ #?r] ; fob { #*x }	3	Left
fob { #?m #?id \ #*x }	[#?m #?id -> _ ^ _] ; fob { #*x }	3	Left
fob {ret { #*r } \ #*x }	[`~ _ -> _ ^ #*r] ; fob { #*x }	3	Left
fob {}	(_)	3	Left

fob { #?id
val { #*v }
ret { #?r } \ #*x }
	[`~ #?id -> #*v ^ #*r] ;
	fob { #*x }
		3	Left

Search String	**Replacement String**	**Precedence**	**Direction**
fob { #?id val { #*v } \ #*x }	[`~ #?id -> #*v ^ _] ; fob { #*x }	3	Left
fob { #?id ret { #*r } \ #*x }	[`~ #?id -> _ ^ #*r] ; fob { #*x }	3	Left
fob { #?id \ #*x }	[`~ #?id -> _ ^ _] ; fob { #*x }	3	Left

The first of these rules allows the modifier to be specified using keywords *public*, *protected*, and *argument*. The other rules introduce a more concise notation for defining a FOB stack. The construct begins with the keyword *fob,* and the definition of the FOBs in the stack are then enclosed in braces, with the backslash used as a termination character for each individual FOB definition. The keywords *val* and *ret* are used to delimit the value and return parts of the FOB definition. Several rules are repeated to specify alternate notations with optional parts. In particular, a FOB without a return value, or a FOB without a variable binding is allowed, and the modifier is optional. The default values used for omitted parts are always the empty FOB, and the default modifier is the protected modifier.

As an example of the SE-FOBS notation, Consider the following code.

```
fob{
        u val{1} \
        public f val{
                fob{v val {2} ret {u + v} \}
        } \
}
```

In this example we use the SE-FOBS-X notation for defining FOB stacks, with several FOB definitions with omitted parts, such as the definitions of the variables *u* and *f*, which are missing their return values. Modification is specified using the keyword *public*, or omitting the modifier, resulting in the default modification of protected.

The processing on this example starts with the application of the macro for the addition operator, with precedence 8. The next highest priority macro invocation in the code is at level 3. The first macro invocation encountered at level 3 is the outermost *fob* invocation. This is matched, at which point the transformed code would read as follows.

```
[`~u -> 1 ^ _] ;
fob{
        public f val{
                fob{v val {2} ret {u.+[v]} \}
        } \
}
```

Proceeding still at precedence level 3, the next leftmost *fob* macro invocation is expanded, yielding the following text.

```
[`~u -> 1 ^ _] ;
[`+f ->
        fob{v val {2} ret {u.+[v]} \}
^ _] ;
fob{}
```

Processing the next invocation of the *fob* macro yields the following.

```
[`~u -> 1 ^ _] ;
[`+f ->
        [`~v -> 2 ^ u.+[v]] ; fob{}
^ _] ;
fob{}
```

Two more invocations of the macro are then processed for the empty *fob* constructs, resulting in the following core-FOBS-X expression.

```
[`~u -> 1 ^ _] ;
[`+f ->
        [`~v -> 2 ^ u.+[v]] ; (_)
^ _] ; (_)
```

14.3. Keyword macros

SE-FOBS-X introduces several macros that introduce keyword versions of several commonly used library operations. These notations are meant to make a FOBS-X program more readable than the notation available in core-FOBS-X. Table 7 gives the macro definitions for the new notations.

Table 7. Standard extension keyword macro quadruples.

Search String	Replacement String	Precedence	Direction
if { #*x } then { #*y } else { #*z }	(#*x) .if[(#*y) , (#*z)]	3	Left
hd #?x	(#?x)./[]	10	Left
tl #?x	(#?x).%[]	10	Left
#?x [#*i] <- #?y	(#?x) .-+ [#?i , #?y]	3	Left
+/- #?x	0.-[#?x]	10	Left
nofob #?x	(FOBS.isEmpty[#?x])	10	Left

The first notation renders the *if* operation more readable. For example, the core FOBS expression

```
x.<[3].if[“yes”,”no”]
```

could be written in SE-FOBS as

```
if {x < 3} then {“yes”} else {“no”}
```

using in addition the macro for the infix notation of the less-than operator. The second two notations can be used to convert the head and tail operations for Vectors to a fairly standard prefix form, using the keywords *hd* and *tl*. The last two notations are used to convert the replace operation from Section 9.2, the negation operation, and the *isEmpty* predicate to more readable forms. To illustrate the effect of the last five notations, the following core FOBS expressions

```
## head of v
        v./[]
## tail of v
        v.%[]
## replace the ith element in v with 3
        v.-+[i, 3]
## negate n
```

```
            0.-[n]
        ## check if FOB f is the empty FOB
            FOBS.isEmpty[f]
```

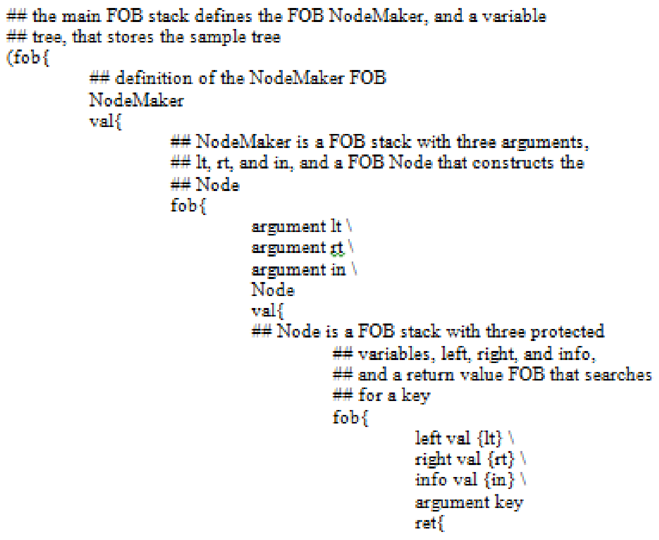
could be expressed in SE-FOBS as

```
        ## head of v
            hd v
        ## tail of v
            tl v
        ## replace the ith element in v with 3
            v[i] <- 3
        ## negate n
            +/- n
        ## check if FOB f is the empty FOB
            nofob f
```

15. AN SE-FOBS-X EXAMPLE

We present a larger example to demonstrate how SE-FOBS-X code is used to solve more complex programming problems. This example is the SE-FOBS-X version of Example (5).

```
        #use #SE
            (16)
```



```
## the main FOB stack defines the FOB NodeMaker, and a variable
## tree, that stores the sample tree
(fob{
        ## definition of the NodeMaker FOB
        NodeMaker
        val{
                ## NodeMaker is a FOB stack with three arguments,
                ## lt, rt, and in, and a FOB Node that constructs the
                ## Node
                fob{
                        argument lt \
                        argument rt \
                        argument in \
                        Node
                        val{
                        ## Node is a FOB stack with three protected
                                ## variables, left, right, and info,
                                ## and a return value FOB that searches
                                ## for a key
                                fob{
                                        left val {lt} \
                                        right val {rt} \
                                        info val {in} \
                                        argument key
                                        ret{
```

```
                                                ## define a FOB stack with four
                                                ## protected variables, a1, a2,
                                                ## a3, and a4 that store the
                                                ## results of the comparisons with
                                                ## the Node, the left child, and
                                                ## the right child, and the final
                                                ## result, respectively
                                                (fob{
                                                        a1 val {info = key} \
                                                        ## for both left and right
                                                        ## child, search the child
                                                        ## only if it is not empty
                                                        ## and the key was not
                                                        ## previously found
                                                        a2
                                                        val{
                                                                if {nofob left | a1}
                                                                then {false}
                                                                else {left[key]}
                                                        } \
                                                                                    a3
                                                                                    val{
                                                                                            if {nofob right | a1 | a2}
                                                                                            then {false}
                                                                                            else {right[key]}
                                                                                    } \
                                                                                    public a4 val{a1 | a2 | a3} \
                                                                            ## the return value of Node is a4
                                                                            } ).a4
                                                                    } \
                                                            }
                                                    }
                                                    ## the return value of NodeMaker is Node
                                                    ret {Node} \
                                            }
                                    } \
                                    ## build the sample tree
                                    public tree
                                    val{
                                            NodeMaker['m', NodeMaker['g', NodeMaker['f', _, _],
                                                    NodeMaker['j', _, _]], NodeMaker['p', _, _]]
                                    } \
                            } )
                            ## use the main FOB tree variable to search for 'f'
                            .tree['f']
                            #.
                            #!
```

Just as in the core-FOBS-X Example (5), this code has two types of elements: a FOB expression, and macro directives. The four directives seen here are the comment directive, “##”, the *end of expression* directive, “#.”, the *end of script* directive, “#!”, and the “#use” directive which is described in the next section. In this case the “#use” directive is notifying the FOBS-X interpreter that the standard extension (#SE) is being used.

16. SCRIPTING AND EXTENSIONS

FOBS-X is intended as a universal scripting language. To function in this way, it must be possible to adapt FOBS-X to new scripting environments. This requires changing the language in terms of its capabilities, and often altering its syntax to streamline the the scripting process in the new environment.

In FOBS-X the mechanism for adding capability is the *extension*. Example (16) from the previous section uses the *standard extension*, but other extensions would be created by the user to handle the demands of various scripting environments. An extension consists of two parts: a macro definition to introduce syntactic changes, and a library module to supply environment dependent capabilities.

The standard extension is not a typical extension. It introduces no new capabilities to FOBS-X, and because of this does not include a library module. The typical extension would, however, define a new primitive FOB which would be merged into the FOBS library FOB.

The mechanism used to activate an extension is the *#use* directive. This directive installs an extension with the given name, by locating the macro file and processing it, and also locating the corresponding library modules and loading them into the FOBS primitive FOB.

Although the macro processing work has been completed, there still remains work to do on the library module inclusion for extensions. Currently it is possible for the user to rewrite the FOBS library module, directly, using the host language, Perl, but it is our intention to devise and implement a library module description language. This language would allow the user to describe the capabilities of a new primitive FOB, at a higher level, and the language processor would automatically generate the required library files, relieving the user of the need to be familiar with the structure of the FOBS-X library, and the intricacies of Perl.

17. FUTURE WORK

This research continues the development of a hybrid scripting language, described by Gil de Lamadrid & Zimmerman [23, 24]. Previously we developed an interpreter for a small core-FOBS language. This initial language was slightly changed, semantically, to the language core-FOBS-X.

The syntax of core-FOBS-X, although nicely consistent, is rigid and not particularly user friendly. The standard extension to FOBS-X, SE-FOBS-X, is a language with a significantly different syntax, and addresses the problems with the core syntax. The Macro Processor translates SE-FOBS-X code into core-FOBS code, and allows our same back-end to process it.

Work on the Macro Processor is now completed. The mechanism for defining macros has been implemented. This facility has been used to implement the standard extension, and the user can use it to define applied extensions.

We are currently working on the interface between FOBS-X and its operating environment. To be used as a scripting language, FOBS-X must be provided with a “hook” into the system being scripted. Our intent is to use the FOB *FOBS* to supply environmental controls.

The FOBS-X language is supposed to be a general-purpose scripting language. Because of this, it is difficult to build into the *FOBS* FOB, all possible scripting control. Rather it would be more practical to make the *FOBS* FOB reconfigurable. As explained in Section 16, our vision would have the user describe the controls they wish to incorporate in the *FOBS* FOB, in some structured *control description language*, and have those controls automatically incorporated into the *FOBS* FOB. Given a control description language with enough generality, this would make FOBS-X almost universal in its scripting capability.

There are problems with allowing the *FOBS* FOB to be reconfigurable using a control description language. A very general description language allows extensive changes to the semantics and computing model of FOBS-X. This problem is helped somewhat by isolating the changes to the FOB *FOBS*. To completely eliminate the problem, however is almost impossible. Given the desire to allow the user to use FOBS-X in new unforeseen scripting applications, it is necessary to provide a mechanism for the user to define environmental controls, implying the use of some sort of control description language. It might be possible to restrict the language so that semantic changes to FOBS-X are limited, but there will be a trade-off between generality, and the ability to adapt FOBS-X to novel scripting applications, versus specialization, and the preservation of purity in the semantics of FOBS-X. This trade-off is a subject for future research.

18. CONCLUSION

We have described a core-FOBS-X language. This language is designed as the basis of a universal scripting language. It has a simple syntax and semantics.

FOBS-X is a hybrid language, which combines the tools and features of object oriented languages with the tools and features of functional languages. In fact, the defining data structure of FOBS-X is a combination of an object and a function. The language provides the advantages of referential transparency, as well as the ability to easily build structures that encapsulate data and behavior. This provides the user the choice of paradigms.

Core-FOBS-X is the core of an extended language, SE-FOBS-X, in which programs are translated into the core by a macro processor. This allows for a language with syntactic "sugar", that still has the simple semantics of our core-FOBS-X language.

Because of the ability to be extended, which is utilized by SE-FOBS-X, the FOBS-X language gains the flexibility that enables it to be a universal scripting language. The language can be adapted syntactically, using the macro capability, to new scripting applications. In the future, the library will also be adaptable, allowing the user to add the operations necessary to adapt it to interact with a new environment.

REFERENCES

[1] Yau, S.S., Jia, X., Bae, D. H. (1991) "Proof: A Parallel Object-Oriented Functional Computation Model", *Journal of Parallel Distributed Computing*, Vol.12.

[2] Gabriel, R. P., White, J. L., Bobrow, D. G. (Sept.1991) "CLOS: Integrating Object-Oriented and Functional Programming", *Communications of the ACM*.

[3] Beaven, M., Stansifer, R., Wetlow, D. (1991) "A Functional Language with Classes", *Lecture Notices in Computer Science*, Vol. 507.

[4] Goguen, J. A., Mesegner, J. (1987) "Unifying Functional, Object-Oriented, and Relational Programming with Logical Semantics", *Research Directions in Object-Oriented Programming*, MIT Press, pp. 417-478.

[5] Bothner, P. (Sept, 1999) "Functional Scripting Languages for the JVM", *3rd Annual European Conference on Java*^{TM} *and Object Orientation,* Århus, Denmark.

[6] Krishnaswami, N (Nov. 2002) "The Needle Programming Language", *Lightweight Languages Workshop*, Cambridge, MA.

[7] Alexandrescu, A. (2010) *The D Programming Language*, Adison Wesley.

[8] Odersky, M, Spoon, L, Venners, B. (2008) *Programming in Scala*, Artima, Inc.

[9] Page-Jones, M. (2000) *Fundamentals of Object-Oriented Design in UML*, Addison Wesley, pp. 327-336.

[10] Goldberg, A., Robson, D., Harrison, M. A. (1983) *Smalltalk-80: The Language and its Implementation*, Addison Wesley.

[11] Ng, K. W., Luk, C. K. (April, 1995) "A survey of languages integrating functional, object-oriented and logic programming", *Journal of Microprocessing and Microprogramming*, Vol. 41, No. 1.

[12] Milner, R., Tofte, M., Harper, R. (1990) *The Definition of Standard ML*, MIT Press.

[13] Krishnaswami, N. (2002) "The Needle Programming Language", *Lightweight Languages Workshop*, Cambridge, MA.

[14] Wringstad, T., Nardelli, F, Z., Lebresne, S., Ostlund, J., Vitek, J. (2010) "Integerating Typed and Untyped Code in a Scripting Language", *Symposium on Principles of Programming Languages*, Madrid, Spain, pp.377-388.

[15] Ousterhout, J. K. (March, 1998) “Scripting: Higher-Level Programming for the 21st Century”, *IEEE Computer*, Vol. 31, No. 3, pp. 23-30.

[16] (2009) *Standard ECMA-262: ECMAScript Language Specification, 5th ed.*, ECMA International, Geneva, Switzerland.

[17] Beazley, D., Van Rossum, G (1999), *Python; Essential Reference*, New Riders Publishing, Thousand Oaks, Ca.

[18] Flanagan, D., *Java in a Nutshell: 3rd ed.*, O’Reilly, Sebastopol, CA, 1999, pp.540-541.

[19] Peyton-Jones, S. (2003) *Haskell 98 language and libraries: the Revised Report* , Cambridge University Press, Cambridge, UK.

[20] Ruby, S., Thomas, D., Hansson, D. (2008) *Agile Web Development with Rails (3rd edition)*, Pragmatic Bookshelf.

[21] Beazley, D. M. (2009) *Python Esential Reference (4th ed.)*, Pearson Education, Upper Saddle River, NJ, , pp. 93-116, 257-276.

[22] Van Roy, P., Haridi, S. (2004) *Concepts, Techniques, and Models of Computer Programming*, MIT Press, Cambridge, MA.

[23] Gil de Lamadrid, J., Zimmerman, J. (2012) “Core FOBS: A Hybrid Functional and Object-Oriented Language”, *Computer Languages, Systems & Structures*, Vol. 38.

[24] Gil de Lamadrid, J., Zimmerman, J. (2011) “FunctionalObjectOriented Hybrid Programming with FOBS”, *International Conference on Software Engeneering Research & Practice*, Las Vegas, NV.