Godiva: a goal-directed dialect of Java

Ray Pereda and Clinton Jeffery
University of Texas at San Antonio
Division of Computer Science
6900 North Loop 1604 West
San Antonio, Texas 78249-0667
210-458-5557
{rpereda, jeffery}@cs.utsa.edu

1. ABSTRACT

Godiva is a dialect of Java that provides general purpose abstractions that have been shown to be valuable in several very high level languages. These facilities include additional built-in data types, higher level operators, goal-directed expression evaluation, and pattern matching on strings. Godivas extensions make Java more suitable for rapid prototyping and research programming.

1.1 Keywords

Programming language design, goal-directed evaluation, very high level languages, Java

2. INTRODUCTION

Very high level languages such as Tcl[16] and Perl[20] are becoming increasingly popular relative to traditional languages like C/C++ and Fortran. By providing notations for application-level concepts instead of abstractions of machine-level capabilities, these languages reduce the programming effort required for most applications.

These languages emphasize practical, rather than theoretical considerations. Ease of use comes from programming constructs that allow concise solutions to a wide variety of problems. In domain specific languages like MATLAB[7], there is linguistic support for objects like matrices. The language makes it intuitive to translate problem descriptions and solutions into code. A problem well stated is half solved, *and half programmed*.

Churchs conjecture states that any computation for which there exists an effective procedure can be realized by a Turing machine[1]. The conjecture only says that if a computation is do-able in one of the Turing-complete languages, it is doable in any of them. Its speaks nothing about the amount of human effort needed to solve a task using a particular programming language.

The Sapir-Whorf[21] hypothesis states that the structure of the language one speaks influences the way one thinks. It is a small step to say that the structure of a language in which one writes programs influences the way one thinks. Testing this hypothesis is beyond the scope of this paper, but it will suffice to say that many programs written in Perl and Tcl would not have been written had the programmer had a more traditional language as his only tool. Godiva gleans the best ideas in several very high level languages and delivers that expressive power with a mainstream (C-based, Java-based) syntax.

Many of the features offered in higher level languages can be implemented in a lower level language via an application programmer interface, API. If APIs were sufficient, there would be no demand for higher level languages, but this is not the case. APIs cannot introduce novel control structures or expression semantics, and even when such facilities are not required, there are advantages to providing syntactic support for commonly used functionality. The linguistic support of a set of API function calls is modest. Language embodiment of these features makes them notationally more concise and therefore easier to use, but it does more than that. Features in the core of the language are more familiar to the community of language users and likewise the language implementers are committed to having these features in their tools. This leads to greater software reuse because it discourages repeating much work that has been done many times before.

Godiva was built by extending a mainstream general-purpose language in order to show that very high level languages need not be idiosyncratic or domain specific in nature. Java was chosen because it is simple to the point of being spartan. It has been well adopted by industry and has attained mainstream status.

Structure of this report. This paper is organized as follows. Section 3 gives a brief overview of Godiva. Section 4 introduces goal-directed expression evaluation. Section 5 focuses on Godivas innovations in goal-directed expression semantics. The non-obvious features are accompanied by short Godiva code examples. Section 6 describes goal-directed pattern matching on strings. Sections 7 and 8 present related work and conclusions.

3. OVERVIEW OF GODIVA

Godiva adds the following built-in data types to Java: arbitrary precision integers, strings, lists, associative tables, and sets. Assignment, comparison, and other operators are extended to work with entire collections of elements at a time. These features and others such as automatic insertion of semi-colons are detailed in the Godiva language reference[10].

Java avoids the whole issue of multiple inheritance. Instead it relies on the Interface facility. Interfaces do not allow for unanticipated code reuse, in that a programmer has to know which methods to implement ahead of time if the class were to later be re-used via an interface. Godiva uses closure-based inheritance (CBI)[9], which provides multiple inheritance and resolves inheritance conflicts using a very simple set of rules. Godivas subversion of Javas purist object model also includes C-style global functions and variables via an implicit `Global` package.

This paper focuses on the two most distinguishing features in Godiva, which are goal-directed evaluation and pattern matching. The design of these features builds on experience gained working with Icon[4], a descendent of SNOBOL4[5]. Most programmers have seen goal-directed programming in logic programming languages such as Prolog. Icon and Godiva do not rely on goal-directed programming as the sole means of control flow as Prolog[19] does, but instead support goal-directed backtracking within specific syntactic contexts, called bounded expressions.

Facilities for pattern matching on strings are present in Godiva in the form of *matching methods* whose bodies consist of a sequence of regular expressions with intermixed code blocks. Matching methods exploit goal-directed expression semantics to facilitate the splicing of ordinary code into the midst of regular expressions.

4. GOAL-DIRECTED EXPRESSION EVALUATION

Goal-directed evaluation eliminates the need for many loops, shortening code and eliminating certain kinds of errors. The basic mechanisms of goal-directed evaluation are *generators* and *control backtracking*. The examples are written in Godiva, but Godivas changes to the expression semantics introduced by Icon are modest. This section presents an overview of goal-directed expression evaluation semantics, and discusses two frequent sources of errors introduced by goal-direction in Icon. The following section discusses features of Godivas semantics for goal-direction that eliminate these errors.

4.1 Success and failure

Goal-directed evaluation starts by replacing boolean-directed evaluation with a substantially more powerful underlying semantics. Expressions in Godiva *succeed* in producing results or *fail* and produce no results. Control structures such as `if (expr)...` are directed by whether or not `expr` produces a result, not by whether that result is true or false. Expression failure propagates from inner expressions to enclosing expressions.

4.2 Non-boolean logic

Relational operators such as x < y either succeed and produce their right operand, or fail. In simple uses such as

```
    if (x < y) statement
```

the meaning is consistent with Java. Since the expression semantics are not encumbered with the need to propagate boolean (or 0 or 1) values, comparison operators instead propagate a useful value (their right hand argument), allowing expressions such as `3 < x < 7` which is rather more concise than Javas `(3 < x) && (x < 7)`.

4.3 Generators

Generators are expressions that can produce multiple results. In a goal-directed language, generators are the building blocks from which powerful algorithms are constructed. Generators respond to expression failure by producing additional results, which are delivered to surrounding expressions by means of implicit control backtracking.

The simplest generator is another post-boolean operator, alternation. Alternation is a binary operator that subsumes Javas logical OR. The expression

```
    expr1 \ expr2
```

produces any values from `expr1` followed in sequence by any value from `expr2`. For example, the expression `1\2\3` is capable of generating three values. The \ character was chosen as the alternation operator for the sake of compatibility as it has no special meaning in Java.

4.4 Backtracking

Whether a generator produces one, some, or all of its results is determined at run-time by whether the expression it is used in requires additional results in order to succeed in producing results. When an enclosing expression fails, its generator sub-expressions are resumed for additional results, and the failed outer expression is retried with those results. When more than one generator is present in an expression, they are resumed in a LIFO manner. The expression

```
        expr == (1 \ 2 \ 3)
```

succeeds if `expr` is any one of the three values. `expr` is compared with the first result, a 1, and only if that comparison (or an expression in the enclosing context) fails is the alternation resumed to produce a 2 or a 3.

The same thing can be written in Java by repeating the reference to `expr` and using three equality tests, but if `expr` is a complex object reference (say, `a[3].left.right.o[2]`), writing the above comparison in Java is longer and less efficient:

In Godiva, the code is:

```
a[3].left.right.o[2] == (1 \ 2 \ 3)
```

In Java, it is:

```
a[3].left.right.o[2] == 1 ||
a[3].left.right.o[2] == 2 ||
a[3].left.right.o[2] == 3
```

4.5 Flaws in Icons Goal-Directed Evaluation

In Icon, it is fairly easy to write expressions that backtrack unintentionally or in which it is unclear whether backtracking will occur or not. For example, given an arbitrary call to a procedure `p()`, it is impossible to know whether `p()` can backtrack or not without examining ps source code looking for suspend expressions. If the name p is bound to different procedures at different times, a given call may be a generator some of the time and a regular procedure call at other times. The language includes a limitation operator to prevent unintended backtracking, in recognition of this problem, but one must still know when to use it.

Another problem in Icon is the infamous every-while bug. In Icon, the control structure *every expr1 do expr2* is used to drive *expr1* to produce every value. There is also a *while expr1 do expr2* control structure that is used to repeatedly evaluate *expr1*. This reevaluates *expr1* to obtain its first value each time through the loop. The problem is that novices very frequently fail to understand the difference between these control structures and select the wrong form of loop. The bug appears occasionally in code written by proficient Icon users, by accident rather than through ignorance.

5. GOAL-DIRECTION IN GODIVA

Godiva implements Icon-style goal-directed expressions while avoiding the two errors identified above. Unintentional backtracking is avoided by making all generators evident from the syntax. Confusion over iterators is avoided by specifying that all loops perform iteration over generators when the control expression is a generator, and conventional loop tests otherwise. Fixing the first error enables the second error to be fixed reasonably.

5.1 Syntactic Transparency

Previous experience with goal-directed evaluation suggests that the combination of implicit control backtracking and implicit generator expressions can result in unfortunate accidents. Also, explicit generator syntax simplifies certain code optimizations. In contrast, all expressions in Godiva where goal-directed evaluation may lead to backtracking are explicit and self-evident in the syntax. This leads to more efficient translation opportunities, and avoids programming errors from accidental goal-directed evaluation.

For these reasons, in Godiva generators are explicitly identifiable from the syntax. Suppose there is a generator nextImmediate that generates each of the immediate family members. One wants to look for members with blood type 0. Below is one way to find such a member,

```
  if (fmember = @family.nextImmediate(),
      fmember.bloodType() == BloodType_O) {...
```

This if statement will succeed if there exist *any* immediate family member with blood type O. Without the @, the test is quite different. It will succeed if the *very next* immediate family member has blood type O.

There are many times that the programm er wants to call a generator to just get one value and knows that he does not want to resume the generator. Without explicit generators, one might code something like the following:

```
  fmember = family.nextImmediate(f)
  if (fmember != null &&
      fmember.bloodType() == BloodType_O) {...
```

This has the drawback of not only being more verbose, and has significant inefficiencies. If the compiler does not know that nextImmediate() cannot be resumed, it must insert code to allow for that possibility. Explicit generator invocation avoids this inefficiency.

The total number of generator operators in Godiva is small. In addition to alternation, there are only three other ways to introduce generators into an expression. The `to` operator generates values from sequences, such as `1 to 10`. Regular expressions in matching methods have generator semantics described in the next section. Lastly, the generate operator, unary `@`, is a polymorphic operator that generates values depending on the type of its operand. `@expr` is defined as follows:

if expr is of type	**@expr result sequence is**
integer	0 to expr-1
aggregate	element of expr, in sequence
method	generator invocation, described below
built-in data type	the elements of the type
object	generator invocation on standard

	method, equivalent to @expr.gen()

For example, if `a` is an array of numbers, then the expression

```
x < @a < z
```

compares elements of `a`, succeeding if any of them are between `x` and `z`. Recall that comparison operators produce their right-hand operand when they succeed; this expression produces `z` each time it succeeds. If the specific elements of `a` in the specified range are desired by the surrounding expression, a more devious expression such as

```
z > (x < @a)
```

is needed. Another possibility, is

```
x < t = @a < z, t
```

User-defined generators are the most general way to introduce generators into an expression in Godiva. User-defined generators are method calls that suspend results instead of returning them. In Godiva, a method can be invoked for at most one result using ordinary parenthesis syntax or it can be invoked as a generator by means of the @ operator.

```
 p()   // ordinary invocation
@p()   // generator invocation
```

For example, in the following statement the then-branch is executed if any result produced by a generator invocation @o.gen() is equal to 1, 2, or 3.

```
if (@o.gen() == (1 \ 2 \ 3)) ...
```

Since gen() is the standard generator method name, @o could be used in place of @o.gen(); this would not be the case if gen() required parameters.

The built-in data types all have a predefined generator that produces each of their components. For a string, @s produces each of the characters. For a list, @l produces the elements starting from the left. For a set, @s, produces the elements in no specific order. For an associative table, @t produces the keys of the table in no specific order.

Within methods, non-final results can be produced using `suspend expr` instead of return. In generator invocation, such a method can be resumed at the point in the code where it suspended, with local variables intact, to produce additional results for the enclosing expression.

`return` can not be resumed; a return statement, `return expr`, always terminates a method call and produces a result whether in ordinary or generator invocation. If a method is invoked using ordinary invocation, it will never be resumed for additional results even if it is a generator that suspended; in such invocations `suspend` is equivalent to `return`.

Without data backtracking, an assignments effects are irreversible. However, there is a reversible form of assignment, `<-`. Reversible assignment appears in an example later in the paper.

5.2 Iteration

Godiva's loop control structures distinguish between generator control expressions and single-result control expressions. For a generator, the control expression is evaluated only once and the loop body executes once per result produced by the generator expression. For a single-result control expression, the expression is re-evaluated each time through the loop, as in conventional languages.

The good news about this is that it allows very clean expression of loops based on generators, such as

```
while (i = (1 \ 1 \ 2 \ 3 \ 5 \ 8))
```

which executes a loop body with `i` set to the first few Fibonacci numbers (the example could be generalized to a user-defined generator that produced the entire Fibonacci sequence). The bad news is that a loop such as

```
while (i < 5 \ j < 10) ...
```

has the counter-intuitive effect of executing the loop 0 or 1 or 2 times. Since Java also defines the non-generator logical-OR operator ||, the above loop should be written

```
while (i < 5 || j < 10) ...
```

5.3 Iterative Array Construction

In addition to iteration in the context of loop control structures, Godiva adds one important unary operator, iterative array construction, which constructs an array of values directly from a generator's result sequence. This operator uses the curly braces, as do array initializers. An expression such as

{ @i }

produces a list of size i with elements {0..i-1}, while

{ @a }

produces a list copy of a.

Below is an extended example of using Godiva that shows an application of goal-directed evaluation. The problem solved is the classic eight queens problem. The task is to position eight queens on a chessboard so that no queen can attack another.

```
import java.io.*
public class Queens {
  int up[16]
  int down[16]
  int row[9]
  public static void main(String args[]) {
    PrintResults(@Q(1),@Q(2),@Q(3),@Q(4),
                 @Q(5),@Q(6),@Q(7),@Q(8))
  }
  public void PrintResults(
    int q1, int q2, int q3, int q4,
    int q5, int q6, int q7, int q8) {
    System.out.println(q1 + q2 + q3 + q4 +
                       q5 + q6 + q7 + q8)
  }

  public int Q(int c) {
    suspend @place(1 to 8, c)
  }
  public int Place(int r, int c) {
    if (row[r] == 0 && down[r + c - 1] == 0,
       up[8 + r - c] == 0) {
      suspend row[r] <- down[r + c - 1]
                     <- up[8 + r - c] <- r
    }
  }
}
```

When all the queens are successfully placed, the row positions are written: 15863724.

5.4 Generators and Aggregate Operations

Generators allow data to be combined in interesting ways. Consider the matrix multiplication problem. For matrices A and B, each element of the matrix AB is a sum of terms computed by multiplying a row of A and a column of B. In Java, we can write this out with a triply-nested loop:

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++) {
    c[i][j] = 0.0;
    for (int k = 0; k < N; k++)
      c[i][j] += a[i][k] * b[k][j];
  //    row i * column j
  }
```

In Godiva, the innermost loop can be replaced by aggregate operations, but only if we can construct arrays corresponding to columns of B. Column j of B can be constructed by iterative array construction: { (@B)[j] }. The matrix multiply looks like:

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    C[i][j] = + (A[i] * { (@B)[j] })
```

The example is intended to illustrate interactions between generators and aggregate operations. In a naive implementation of Godiva, it pays to reorder things and construct the columns of B up front.

```
for (int j = 0; j < N; j++)
  double temp[] = { @B[j] }
  for (int i = 0; i < N; i++)
    C[i][j] = + (A[i] * temp)
```

6. PATTERN MATCHING

Pattern matching facilities on strings are extremely useful. For example, tools such as perl, grep, Tcl, awk, Emacs, and Python owe much of their popularity and power to extensive support for regular expressions (regex for short). In light of regex popularity, Godiva supports two broad flavors of pattern matching: text-directed and pattern-directed. One matches a pattern against some target text. Text-directed pattern matching is based on the construction of automata much like `lex(1)`. Once the automaton is built, the characters in the target text are used to drive the matching. Pattern-directed matching is more goal-directed in nature and deserves a closer look. Many of the ideas for goal-directed evaluation arose during the implementation effort of SNOBOLs pattern matching facilities.

In pattern-directed matching, the order and structure of the pattern is used to guide the matching process. The patterns are defined as special methods with the keyword `pattern` as a method modifier. Any class that contains pattern methods implicitly subclasses `Match`. Rather than simultaneously matching many patterns, pattern-directed matching uses a single pattern that may contain several parts. The matching proceeds by trying to match parts of the regex on a component by component basis. The syntax of matching methods is as follows:

```
pattern MethodHeader {
   <ExtRegEx>
   else   {ElseAction}
}
<ExtRegEx> :== {<RegEx> <action>}
<RegEx>    :== <SimpleRegEx> |
                ( <ExtReg> )  [ + | * ]
```

A regex is composed of either a simple regex or an extended regex surrounded by parentheses followed by either a plus or a star. The general ideal is that code fragments may be nested anywhere within an extended regular expression as long as it is delimited by curly braces.

The regexs and actions are executed left to right, outer to inner. They are resumed in a LIFO manner, i.e. stack discipline, should either an regex or action fail.

In a pattern, the operator | is used to specify another alternative. So, given `a|b`, first we try to match an `a`. If that fails, then upon backtracking we try to match the `b`. Similarly, given `a*` we try to match the zero `a s`, if that produces a failed match later down the process, the matching engine tries to match one `a`, then two, and so on. The pattern `a+`, assumes that zero `a s` fails, and behaves just like `a*` otherwise.

Patterns are a special case of generators that suspend on a match. If a regex fails, it propagates the failure to the last suspended expression, which is resumed if possible. When a pattern is resumed, the last position of the last successful partial match is restored. The engine then continues matching until it either succeeds with a complete match or fails and executes the corresponding code in the else-action.

Below is an extended example:

```
import godiva.pattern
class MyMatch extends Match {
  // matches a comma separated list of dates
  pattern void dlist(String s)
  {
    ( date() {
        this.start = this.pos
        // prevents backtracking
      }
      WS()
      \,
      WS()
    )+
  }

  // matches a date
  pattern void date(String s) {
    (january|february|march|april|may|
     june|july|august|september|october|
     november|december)
    WS()
    ([0-9]+) {
    system.out.writeln(substring[0]+ \t +
                       substring[1])
  }
  // matches white space
  pattern void WS(String s) {
    [ \t\n]*
  }
  public static void main(String args[]) {
    MyMatch m = MyMatch()
    String s = " january 12, feburary 19,   +
        march 22, december 25"
    m.dlist(s)
  }
}
```

The above program writes out the following:

```
january     12
feburary    19
march       22
december    25
```

By default, the patterns have the remainder of the unmatched target string as a default argument when called inside of a regex. If a pattern is called from an ordinary expresion then the argument must be specified as in the example with `m.dlist(s)`.

For comparison here is the same program to parse comma separated list of dates in perl,

```
#!/usr/local/bin/perl

  $_ =  january 12, feburary 19,   +
         march 22, december 25";
  while ( m{
            (january|febuary|march|april|may|
             june|july|august|september|october|
             november|december)
            \s+
            ([0-9]+)
            \,?
            \s*
           }ixg
        )
  {
    print "$1\t$2\n";
  }
}
```

In perl, the actions cannot be arbitrarily intermixed with the patterns. It is an *ad hoc* effect, that it is possible to iterate over the matches with the while loop. The \s is a built-in regex for matching white space. The Godiva program has a similar structure but we do have to create our own low level patterns where we do not have built-in patterns.

By assigning to `this.start`, one can set the left most possible position from which backtracking can begin. Once we are satisfied with the current component match, we can use this to prevent any other possible matches on a portion of the target string left of start. The keyword `fail` unmatches the currently matched string, and starts backtracking. `this.pos` returns an integer that is the current position within the matched string. Because there is hidden backtracking, minimal matches are also unique to pattern-directed matching. These operators match the minimal amount of text in order to succeed. Parentheses capture the portion of the target string that the inclosed regex matches. The results are store in an instance field called `substring`, which is an array of strings. The parenthesis captures are number from left to right by the opening parenthesis. The following regex demonstrates minimal matching, `substring[i]` captures, and the fail statement:

```
^.*?[0-9][0-9] {
    // matches the last two digits of a
    //line
    }
([0-9]+) {
      if ( ToInteger(substring[1]) > 31)
         fail
    }
```

To summarize, patterns are composed of the following atomic patterns and operators:

?	matches 0 or 1 occurrences of preceding pattern
|	Alternation
*	non-greedy star
+	non-greedy +
{min, max}	non-greedy range
[ ]	matches any of the listed characters
a	directly matches non-special character, can be escaped with \a if *a* is a special character
( )	used to group patterns, it also captures the enclosing text that is match into the array substring[]

The first five operators are generators. Non-greedy means that the pattern matches the minimum number of characters in order to satisfy a match. In an alternation such as `a|b`, the matching proceeds by first trying to match `a`. If that fails, the matching engine tries to match `b`. The matching engine will only try to match more characters if resumed after matching too few characters causes by a failure later during the matching.

The following is a summary of the special data members and expressions that are available within the action part of a pattern:

this.substring[i]	i-th matched substring, an r-value
this.start	an l-value to set the left most position for backtracking
this.pos	returns the current position in target text
fail	ignores the current subpattern match and starts backtracking

7. RELATED WORK

Godiva is closely related to Java and Icon, and also borrows ideas from Perl, REXX[2], Python[12], and APL[8]. Many of the practical ideas in Godiva come from useful facilities found in the above languages. They are a rich set of built-in data types with and powerful set of operators

Goal-directed expression evaluation originated in Icon and generalizes both the pattern-matching facilities in SNOBOL4 as well as iterator mechanisms such as those found in CLU[11]. An efficient model for the implementation of goal-directed evaluation has recently been developed that will facilitate the adoption of goal-directed evaluation in new languages[17]. Numerous other language implementation projects are now targeting code generation for the Java VM, including other very high-level languages that provide advantages over Java, such as NetRexx. Many dialects of the Java language that add features from C++ have been developed. For example, a research group has added parameterized types to Java[14].

Pizza is a Java dialect that incorporates features from functional languages[15]. Pizza adds parametric polymorphism, higher-order functions, and algebraic data types to Java. These extensions are largely orthogonal to those proposed in Godiva, and take Java in a very different direction, increasing the language's expressive power by extending the type system with a strong theoretical foundation. GJ is an extension of the Java programming language that supports generic types[3].

The UTSA group initially implemented a prototype of Godiva that targets the Icon virtual machine. The Icon VM has support for goal-directed evaluation. Our current efforts are targeting the Java VM. Jcon is a new Java-based implementation of the Icon programming language[18]. Jcons implementation demonstrates that goal-directed evaluation can be efficiently implemented on the Java VM. Godiva has the potential for greater efficiency due to strong typing as well as the syntactic transparency of generators.

8. CONCLUSIONS

Java's design is elegant in comparison with most languages, but it is a spartan elegance. Java's lack of higher-level features, aside from classes, makes a fine philosophical stand, and if one ignores the mountain of class libraries Java is fairly easy to learn, as was Pascal. We contend that this spartan elegance is unnecessary, and have attempted to prove so in a very high level language dialect that retains Java's main assets.

Godiva adds substantially to the expressive power of Java by integrating some of the best ideas associated with very-high level languages. Java's strengths are augmented in the areas of numeric and string processing, as well as data structure and algorithm development. Remarkably, this expressive power is achieved with very few additions to Java's syntax that might reduce readability or decrease the language's broad appeal.

Any proposal to extend a language must address the issue of generality. Goal-directed programming is perhaps the most general capability added to Java. Not every feature proposed in Godiva is necessary for every application, but one or several of the features will be valuable in most programs of medium or large size. The end result is a notation that substantially more attractive for the development of technical applications.

Godivas designers have tried to heed the advice of C. A. R. Hoare: *The language designer should be* *familiar with many alternative features designed by others, and should have excellent judgement in choosing the best and rejecting any that are mutually inconsistent One thing he should not do is to include untried ideas of his own. His task is consolidate, not innovate* [13]. Each feature provided by Godiva has been tested in one or more languages with a significant user base and has proven its worth in real programs.

9. REFERENCES

[1] Church, Alonzo, An Unsolvable Problem of Elementary Number Theory, *American Journal of Mathematics*, 58:345-363, 1936.

[2] Cowlishaw, M. F., *The NetREXX Language.* Prentice-Hall, 1997.

[3] GJ - A Generic Java Language Extension, http://www.cs.bell-labs.com/who/wadler/pizza/gj/

[4] Griswold, Ralph E. and Madge T. Griswold, *The Icon Programming Language*, 3rd Edition, Peer-to-Peer Communications, 1997.

[5] Griswold, R. E., J. F. Poage, and I. P. Polonsky, *The SNOBOL4 Programming Language, 2nd Edition*, Prentice-Hall, 1971.

[6] Gudeman, David A., Denotational Semantics of a Goal-Directed Language, *ACM Transactions on Programming Languages and Systems*, Vol 14, pp 107-125. Jan. 1, 1992.

[7] Hansel, Duane and Bruce Littlefield, *Master MATLAB: a Comprehensive Tutorial and Reference*, Simon & Schuster, 1996.

[8] Iverson, Kenneth E., *A Programming Language*. John Wiley and Sons, 1962.

[9] Jeffery, Clinton L., *Closure-Based Inheritance and Inheritance Cycles in Idol*, Technical Report: 98-3, UTSA, Division of Computer Science, July 23, 1998.

[10] Jeffery, Clinton L. and Ray Pereda, *Godiva: a Very-High Level Dialect of Java*, Technical Report: 98-4, UTSA, Division of Computer Science, July 23, 1998.

[11] Liskov, B. H., A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM*, Vol. 20, No. 8, August, pp. 564 - 576, 1977.

[12] Lutz, Mark. *Programming Python*, OReilly and Associates, Inc., 1996.

[13] Modula-3: Language definition (multi-page), http://www.research.digital.com/SRC/m3defn/html/index.html

[14] Myers, Andrew C., Joseph A. Bank, Barbara Liskov: *Parameterized Types for Java*, pp. 132-145, POPL 1997.

[15] Odersky, Martin and Philip Walder, Translating theory into practice, *24th ACM Symposium on the Principles of Programming Languages*, Paris, France. January 1997.

[16] Ousterhout, John, *Tcl and the Tk Tookit*, Addison-Wesley, 1994.

[17] Proebsting, Todd. Simple Translation of Goal-Directed Evaluation, *Proceedings of the SIGPLAN 97 Conference on Programming Language Design and Implementation*. Las Vegas, NV. pp. 1-6, June 1997.

[18] Proebsting, Todd and Gregg Townsend, Jcon: A Java-Based Icon Implementation, http://www.cs.arizona.edu/icon/jcon/

[19] Sterling, L, and Shapiro, E. *The Art of Prolog.* MIT Press, Cambridge, Mass., 1986.

[20] Wall, Larry and Randal L. Schwartz, *Programming Perl*, OReilly and Associates, 1991.

[21] Whorf, Benjamin Lee, *Language Thought & Reality*, MIP Press, Cambrige, Mass., 1956.