Godiva

a Very-High Level Dialect of Java

Clinton Jeffery jeffery@cs.uidaho.edu

Draft Version 0.5b, August 29, 2011.

Language Reference Manual

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. Godiva's extensions make Java more suitable for rapid prototyping and research programming. Adding these features to the core language increases the expressive power of Java in a way that cannot be achieved by class libraries.

Despite introducing higher level structures and expression semantics, Godiva retains as much compatibility with Java as possible. Most Java code does not break when compiled with Godiva, and Godiva programs can utilize compiled Java classes without restrictions.

University of Idaho
Department of Computer Science
Moscow, ID 83844 USA

Contents

1. Introduction
2. Syntax
   - Semi-colon Insertion
   - Function Syntax
3. New Data Types and Semantics
   - Numbers
   - Strings
   - Lists
   - Sets & Bit Vectors
   - Tables
   - Multiple Inheritance
4. New Operator Semantics
   - Generalized Assignment and Comparison
   - Operations on Aggregates
   - Aggregate Numeric Operations
   - Reductions
   - Field Access Augmented Assignment
5. Goal Directed Evaluation
   - Non-boolean Logic
   - Generators
   - Control Backtracking
   - ? Generators and Aggregate Operations (not done yet)
6. Pattern Matching on Strings
   - Text-Directed Matching
   - Pattern-Directed Matching
7. Related Work
8. Summary
9. References

---

1. Introduction

Godiva is a high-level general-purpose programming language descended from Java, and hence from C. While their low-level built-in operator sets make Java and C best-suited for systems programming, Godiva has higher-level built-in facilities that orient it more towards applications programming. Godiva represents an example of what Java might become, one step further along an evolutionary path. Godiva stands for **GO**al-**DI**rected ja**VA**.

Is Java perfect? No! It is not concise or expressive enough. The Java designers note that C++ was hamstrung by backwards compatibility with C, and in Java they make numerous improvements by dropping the compatibility requirement, but they did not go far enough. Java offers solid support for networking, user interfaces, and database access, but is weaker than many other languages in its support of data structures and algorithm development required by most substantial mainstream applications. The decision to omit operator overloading from the language, although arguably the right thing to do, unfortunately precludes Java from offering even the limited means that C++ provides to facilitate concise notations for such features by means of class libraries.

The contention of this paper is that Java's language level should be raised to accomodate the needs of sophisticated applications. Class libraries are nice, but they cannot compete, notationally, with built-in operators and control structures. Minimalism is desirable, but not when substantially more expressive power is available with only minor changes in syntax. Many very high level languages offer increased productivity in a specific application area by introducing bizarre syntax. Godiva's goal is to increase the expressive power and raise the language level of Java with a minimal introduction of new syntax. In order to achieve this goal, Godiva extends the core *semantics* of the operators and the expression-evaluation mechanism.

The facilities that Godiva presents as extensions of Java are not new. They originated decades ago in special-purpose languages and have slowly propagated to other languages and been generalized to other application areas. Godiva is the first language that integrates aggregate operations, pattern-matching control structures, and goal-directed expression evaluation within an object-oriented context. Godiva aims to raise Java's level to the point where it can more seriously compete with higher-level general-purpose applications languages such as Perl or Tcl.

The major improvements Godiva makes to Java are as follows:

- goal-directed expression-evaluation semantics
- handy built-in data structures and a rich set of operators
- aggregate operations on data structures, in shallow and deep flavors when appropriate
- pattern matching on strings

This paper serves as a working reference for Godiva-0, the research prototype of Godiva.

2. Syntax

The syntax of Godiva is based on that of Java. A marked-up lexical specification in a Lex(1) specification is godlex.htm, and a YACC(1) specification is in godyacc.htm. Red text marks Java features omitted from Godiva-0, and green marks those items not part of Java that have been added to make up Godiva. light green items are part of the Godiva language not implemented in Godiva-0.

2.1 Semi-colon Insertion

To reduce clutter and numerous-but-trivial "semi-colon expected" errors, the compiler automatically inserts a semicolon at the end of a line if an expression ends on the line and the next line begins with another expression. So we can write

```
a = 1
b = 2
c = 0
```

as an equivalent expression for

```
a = 1; b = 2; c = 0
```

2.2 Function Syntax

Godiva supports "function syntax", where functions are declared outside any class definition. Such functions are equivalent to public static methods inside whatever class name corresponds to the current filename. For example, if the filename is foo.java, the Godiva program

```
void main(String[] args) {
   System.out.println("hello, world")
}
```

is equivalent to the Java program:

```
class foo {
public static void main(String[] args) {
   System.out.println("hello, world")
   }
}
```

3. New Data Types

Built-in data types have much more syntactic support than user created data types. Especially because Java and Godiva do not allow operator overloading, increasing the number of built-in data-types that use an expressive operator syntax significantly raises Godiva's semantic level compared with Java.

Between machine integers and object there is a niche of abstraction for built-in data types. The following criteria were used to select candidate built-ins:

- Extremely common data-types used in many different applications
- Data types for which immutable, value oriented semantics are commonly used

- Data types that require only a small set of commonly-understood operations

3.1 Numbers

To the list of integer types, there is a new primitive type called bigint. It is of arbitrary precision. If an integer literal ends in B or b, it is a bigint.

Below is a comparison of code needed to compute `(x - y) / (x + y)`

Java	Godiva

`BigInteger x = new BigInteger(100000000);`
`BigInteger y = new BigInteger(900000000);`
`BigInteger ans = new BigInteger(0);`
`BigInteger temp = new BigInteger(0);`
 	
`ans.add(x);`
`ans.sub(y);`
`temp.add(x);`
`temp.add(y);`
 	
`ans.divide(temp);`
	`bigint x = 1000000b`
	`bigint y = 9000000b`
	`bigint ans = (x - y) / (x + y)`

3.2 Strings

`string` is a new primitive type. Godiva adds syntactic support for substrings, and finding the length of a string. The operator == tests for equality of string values not their references. Strings are initialized by default to the empty string. The semantics are value oriented just like integers in Java. Negative indices may be used to denote positions from the end of the string.

Below is a list of operations on strings:

s1 s2	concatenation, short form of s1 + s2
s[i]	the i-1 character
s[i:j]	a substring with characters bounded by i and j
#s	length of the string

Below is a comparison:

Java	Godiva

`String s1 = "Dec";`
`String s2 = "2002";`
`String s3;`
`int len;`
 	
`s3 = s1 + " 25, " + s2;`
`if (s3.equal("Dec 25, 1999")) {`
`   System.out.println("It is Christmas of 2002.");`
`}`
`System.out.println("The month is " + s3.substring(0,4));`
`len = s3.length();`
`System.out.println("The year is " + s3.substring(len-5, len));`
	`string s1 = "Dec"`
	`string s2 = "2002"`
	`string s3`
	
	`s3 = s1 " 25, " s2  // the explicit '+' is optional`
	`if (s3 == "Dec 25, 2002") {`
	`   System.out.println("It is Christmas of 2002.")`
	`   }`
	`System.out.println("The month is " s3[0:3])`
	`System.out.println("The year is " s3[-4:0])`

3.3 Lists

Lists are sequences of any type of element. Lists grow and shrink dynamically, similar to the Vector class with improved syntactic support. Below is a list of operations on lists:

L1 L2	Concatenations L1 and L2
+=	appends to a list
L[i]	the i-1 element of L

L[i:j]	a slice of L with elements bounded by i and j
L.pop()	removes and returns left-most element
L.push(x,...)	appends elements x to the left end of L
L.insert(x, i)	inserts x into L at position i; i defaults to the right end of the list.
L.delete(i)	removes element at position i from L; i defaults to the right end of the list.
#L	the size of list L

A comparison of code needed to compute a list of of pairs i followed by sum of 1 to i:

Java

```
Vector v1 = new Vector();
for (int i; i < 10; i++) {
   sum += i;
   v1.add(i);
   v1.add(sum);
   }
```

Godiva

```
list l
 for (int i; i < 10; i++) {
    sum += i
    l += i sum
    }
```

3.4 Tables

Tables are associative arrays, which is to say that their indices are not in sequence and may be of any type. Tables can grow dynamically, and when you create them you can specify a default mapping for all indices that have not been assigned a value. Below is a list of operations on tables:

T[a] = b	stores a mapping from element a to element b
T[a]	Succeeds if a has a mapping using T
T[] = a	sets the default mapping to element a
T - a	pulls out a from the mapping in T
#T	the size of table T

Below is a comparison of the code needed to do a quick check of the randomness of the values produced by Math.random():

Java

```
Hashtable ht = new Hashtable();
for (int i = 0; i < 10000; i++) {
   Integer r = new Integer((int) (Math.random() * 20));
   if (ht.containsKey(r))
      ((Counter)ht.get(r)).i++;
   else
      ht.put(r, new Counter());
   }
```

Godiva

```
table t[] = 0

for (int i = 0; i < 10000; i++) {
   t[(int) (Math.random() * 20)] += 1
   }
```

3.5 Sets and Finite Sets

The set types are modeled after the mathematical notion of sets. Godiva sets can have members of arbitrary, heterogeneous types, and may be of arbitrary size, shrinking and growing as needed. As a special case for sets whose elements come from a finite universe, we have a *finite set* type that may typically be implemented using bit vectors. For a finite set the universe of all possible elements must be stated when the set is created. Finite sets have a set complement operator in addition to the other properties of sets. Below is a list of operations on sets:

S1 + S2	union of S1 and S2
S1 * S2	intersection of S1 and S2
S1 - S2	difference of S1 and S2
S <- a	inserts the element a into S

S - a	pulls out element a from S
a in S	succeeds if a is a member of S
S1 / a	removes element a from S1
!S1	complement of S1, ONLY on finite sets
S1 < S2	succeeds if S1 is a strict subset of S2
S1 <= S2	succeeds if S1 is a subset of S2
S1 > S2	succeeds if S1 is a strict superset of S2
S1 >= S2	succeeds if S1 is a superset of S2
#S	the size of set S

Here is a comparison of a code fragment used to manipulate a deterministic finite automaton:

Java	Godiva
`Set s = new Set();`	`set s`
`int prev_size;`	`int prev_size`
`DFA myDFA = new DFA();`	`DFA myDFA = new DFA()`
`do`	`do`
`   prev_size = s.size();`	`  prev_size = #s;`
`   s = myDFA.closure(s);`	`  s = myDFA.closure(s)`
`while (s.size > prev_size);`	`while (s.size > prev_size)`

3.6 Multiple Inheritance

Godiva's object model extends Java's model to include multiple inheritance. The multiple inheritance semantics borrow from closure-based inheritance (CBI), developed originally in the Idol programming language. CBI defines a simple rule for name conflicts in multiple inheritance. When inheriting from a list of superclasses, the compiler adds fields and methods using a left to right, depth-first search of the superclass list. The first superclass to define a field or method has strict ownership of that identifier.

The CBI rules define which superclass member variable or member function a given name refers to in the event of a name conflict. Other than providing a default resolution of name conflicts that C++ does not have, Godiva multiple inheritance is similar to C++ multiple inheritance. In particular, function overloading uses type signatures to avoid many name conflicts, and qualifying a name with an explicit superclass name allows references to superclass member variables and functions that are "hidden" by other inherited entities. The storage occupied by a multiply-inheriting instance is comparable to C++ semantics with "virtual" applied to everything.

As an example, suppose that Pen defines the methods Refill(), and Write(). Also suppose Pencil defins methods Erase(), and Write().

```
class MechanicalPencil extends Pen, Pencil {
   foo() {
      Refill()       // Uses Pen's Refill
      Erase()        // Uses Pencil's Erase()
      Write()        // Uses Pen's Write() method, Pen comes before Pencil
   }
}
```

4. New Operators and Their Semantics

4.1 Generalized Assignment and Comparison

There are three flavors of assignment and comparison: shallow, deep, and 1-level deep. Assignment and comparison are basic operations on all types that support the Cloneable interface.

Consider the following expression that represents either assignment or comparison:

```
A operator B
```

A shallow assignment uses reference semantics and copies the L-value of B to the L-value A. If B does not have an L-value, i.e. B is a literal value, then one is generated it copied to A. A deep assignment uses value semantics and makes a complete copy of the R-value of B and copies it to A. This may require recursing down each of the embedded components in B. A 1-level deep assignment only recurses one level deep in the copy; beyond that it does a shallow copy.

A shallow comparison uses reference semantics and compares the L-value of B to the L-value A. If B does not have an L-value, i.e. B, is a literal value, then a deep comparison is used. A deep comparison uses value semantics and makes a complete comparison of the R-values of A and B. This may require recursing down each of the embedded components in each of the structured times. A 1-level deep assignments only recurses one-level deep in the comparison; beyond that it does a shallow comparison.

	Shallow	1-Level	Deep
Assignment	X1 = X2	X1 := X2	X1 ::= X2
Comparison	X1 == X2	X1 === X2	X1 ==== X2

Because swapping is a very common operations, we introduce the following swap operator that swaps the reference of a and b.

```
a :=: b
```

4.2 Aggregate Operations

Aggregate operations operate on corresponding elements of entire compound structures at a time. By eliminating many loops over arrays and structures, aggregate operations reduce the amount of code required and eliminate many trivial bugs and oversights. Aggregate operations also simplify the task of parallelizing compilers.

In Godiva, existing numeric operators are implicitly aggregate when one or more operands is an array, list, table, or an object that supports the Vector interface. For simplicity in the subsequent description, we will refer to all forms of aggregate as a vector. In scalar-vector operations, the scalar operand is applied to each element of a vector operand; elements in vector-vector operands are generally combined pairwise. Godiva also offers several new (non-Java) aggregate operators, styled partially after Dataparallel C [Hat91].

4.3 Aggregate Numeric Operations

All binary operators are over-loaded to work on pair-wise elements of structured built-in data types. The simplest examples of aggregate operations are those that operate on vectors and combine them with scalars and other vectors. The following code examples illustrate these operations.

```
double x = 2.5
int ia[][] = {{1, 2}, {3, 4}}
double da[][] = ia * x                  // {{2.5, 5.0}, {7.5, 10.0}}
da += 1.0                               // {{3.5, 6.0}, {8.5, 11.0}}
da *= da                                // {{12.25, 36.0}, {72.25, 121.0}}
```

The most general form of aggregate operation is the element-wise application of a method to vector parameters, producing a new vector of results. This is performed implicitly as a final step of method resolution.

```
da = sqrt(da)                           // {{3.5, 6.0}, {8.5, 11.0}}
```

If multiple vector parameters are supplied, they must be of the same size and shape, and the method is invoked on corresponding elements.

```
int ia[][] = {{1, 2}, {3, 4}}
int ia2[] = pow(ia[0],ia[1])            // {1, 16}
```

Other numeric operations, such as matrix multiplication, involve more complex combinations of their operands. In APL or J there might be dozens of built-in functions for such computations. Godiva does not introduce many new operators; instead it combines aggregate operations with the goal-directed expression evaluation mechanism described below to support interesting computations.

4.4 Reductions

One new form of operators that Godiva does provide are *reductions* that produce new scalar values from an aggregate operand. The final dimension of a vector can be reduced to a scalar using unary versions of the arithmetic operators.

```
+{1, 2, 3}                          // returns the sum, 6
double red[] = +da                  // {9.5, 19.5}
double re2 = +(+da)                 // 29.0
```

Of course, this sort of thing usually makes more sense when used in combination with other operations. The following code computes the value of a polynomial whose terms are represented by the array `coefficients`.

```
double x = 2.5
double coefficients[] = {3.0, -2.0, 9.0, 4.0}
double exponents[] = {0.0, 1.0, 2.0, 3.0}
double powers[] = pow(x, exponents)          // {1.0, 2.5, 6.25, 15.625}
double terms[] = coefficients * powers      // {3.0, -5.0, 56.25, 62.5}
double sum = +terms                         // 116.75
```

or, the code can be expressed more succinctly as:

```
double sum = +(coefficients * pow(x, exponents));
```

Godiva also adds min and max operators that operate on scalars and vectors, `<?` and `>?`. These predefined operators return the minimum or maximum values of their operands as follows:

```
1 <? 2          // returns 1
3 >? 4          // returns 4
<?{2, 1, 3}     // returns 1
```

4.5 Aggregate Table Operations

Tables support the same aggregate semantics as ordinary arrays, producing new tables.

```
table debits = new table(0.0)                   // default 0.0
table credits = new table(0.0)
debits["Tucson"] = 5.9
debits["San Antonio"] = 9.5
credits["Tucson"] = 3.5
credits["San Antonio"] = 11.1
credits["Seattle"] = 4.2
debits += 1.0                                  // 6.9, 10.5
table balances = credits - debits              // -3.4, 0.6, 4.2
```

4.6 Field Access Augmented Assignment

Godiva supports an additional augmented assignment operator, `.=`, that performs a field access. It is typically used in list or tree traversals, such as

```
while (theLongVariable != null) theLongVariable .= next
```

instead of Java's

```
while (theLongVariable != null) theLongVariable = theLongVariable.next;
```

5. Goal-directed Expression Evaluation

Goal-directed evaluation does for algorithms what aggregate operators do for arrays and structures: it 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*, described below.

5.1 Success and Failure

Goal-directed evaluation starts by replacing boolean-directed evaluation with a substantially more powerful underlying semantics. Expressions in Godiva either *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.

5.2 Non-Boolean Logic

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

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

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

5.3 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. In Godiva, unlike its goal-directed predecessor Icon, these expressions are always obvious; syntactic transparency improves readability. 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 Java's logical OR. The expression *expr[1] \ expr[2]* produces any values from *expr[1]* followed in sequence by any values from *expr[2]*. For example, the expression `1 \ 2 \ 3` is capable of generating three values.

5.4 Control Backtracking

Whether a generator produces one, some, or all of its results is driven 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:

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

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

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. For these reasons, in Godiva generators are explicitly identifiable from the syntax.

The total number of generator operators in Godiva is small. In addition to alternation, there are two other ways to introduce generators into an expression in Godiva. 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	**generator result sequence is**
integer	`0` to *expr*-1
aggregate	elements of *expr*, in sequence
method	generator invocation, described below
built-in data type	the elements of the type
object	generator invocation on a 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, the more devious expression

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

is needed.

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)) // then-statement
```

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.

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 and could produce additional results.

5.5 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 counterintuitive 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) ...
```

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`.

5.6 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 can be extremely handy. For example, tools such as perl, grep, Tcl, awk, Emacs, Python owe much of their popularity and power to extensive support for regular expressions. In light of the popularity of regular expressions, Godiva supports two broad favors of regular expression based pattern matching: text-directed and pattern-directed.

6.1 Text-Directed Pattern Matching

Pattern matching on a string is done with the scan() statement. A scan statement has the scope of a procedure declaration. The underlying matching engine is based on the construction of a deterministic finite automata, or DFA. This matching progresses as we examine the target text character by character. The syntax is as follows:

```
scan(target) {
      pattern1: {action1}
      pattern2: {action2}
      ...
      patternN: {actionN}
      else: {action N+1}
   }
```

In the same way that the UNIX utility Lex does matches, a scan statement generates a single DFA from the above N patterns. The matching happens simutaneously on all N patterns. The pattern with the longest match in target is selected and the corresponding action executed. If none of the patterns match, then the action associated with the else statement is executed if one is present. If the else pattern is omitted then the default action is to ignore the matched text and continue stepping through the target text. The action is made up of ordinary Godiva code. If a sequence of matching on the target are of interest, then an action can contain a suspend statement which will allow further matches upon resuming the scan. The matched text is available in the global string called `substring`.

The following example shows how one might use scan to do the lexical analysis for a simple expression evaluator:

```
class demo {

   public tokes(String s) {
      scan(s) {
         [0-9]+   :  { suspend substring }
         [A-Za-z]+:  { suspend substring }
         [+*-]    :  { suspend substring }
      }
   }

   public static void main(String args[]) {
      while i = @tokens("123 + 55 mod 22") {
         System.out.println(i + "\n")
      }
   }
}
```

Writes out:

```
123
+
55
mod
22
```

This table lists all the operators for the text-directed regular expressions:

.	Dot	matches any one character
[...]	character class	matches any character listed
[^ ...]	negated class	matches any character not listed
\char	escaped	removes any special meaning of char
?	question	one allowed, but is optional
*	star	any number of characters
+	plus	one required, additional are optional
{min, max}	range	min required, max allowed
^	caret	matches the start of a line
$	dollar	matches the end of a line

|	alternation	matches either expression it separates
(...)	parentheses	used to group patterns, it also captures the enclosing text that is matched into the array substring[]

6.2 Pattern-Directed Matching

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).

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
    }
```

In addition to the operators found with text-directed pattern matching, we add the following operators which can be used with pattern-directed pattern matching:

*?	non-greedy star
+?	non-greedy ?
{min,max}?	non-greedy range
this.substring[i]	the i-th matched substring, an r-value
this.start	set the left most position for backtracking, an l-value
this.pos	returns the current position in target text

fail	ignore the current component match and starts backtracking

7. Related Work

Godiva's extensions to Java are a synthesis of ideas found in other languages. Aggregate operations are a hallmark feature of APL [APL], and widely adopted in languages for numeric and scientific computing, such as J, C*, and High Performance FORTRAN. Of these languages, C* and Dataparallel C are of particular interest since they describe features similar to Godiva's aggregate operators in languages that use C syntax as a base, as to Java and Godiva [CPG] [Hat91].

Sets are arguably the most simple of the dynamic data types in that they don't even impose an order constraint on the elements unlike arrays. They are part of our foundations of mathematics and fundamentals of algorithms. The designers of Modula-3 [Modula-3] saw a need for supporting sets at the most basic part of the language just like arrays. They are not relegated to a remote place in a standard library.

Associative arrays appeared in SNOBOL4 [Snobol] and have since been adopted in most string processing languages, such as Perl, Rexx, and Icon. Pattern matching also first appeared in SNOBOL4, and pattern-matching based on regular expressions is found in numerous UNIX tools, such as Perl [Perl]. A comprehensive study how regular expressions are used in a large variety of successful and popular software tools can be found in [Regular Expressions].

Goal-directed expression evaluation originated in the Icon programming language [Icon]. Goal-directed evaluation generalizes both the pattern-matching facilities in SNOBOL4 as well as iterator mechanisms such as those found in CLU [CLU]. 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 [Pro97].

Numerous other language implementation projects are now targetting code generation for the Java VM, including other very high-level languages that provide advantages over Java, such as NetRexx [NetRexx]. Many dialects of the Java language that add features from C++ have been developed. Pizza is a Java dialect that incorporates features from functional languages [Pizza]. 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.

Summary

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 language design without losing sight of 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. 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.

Acknowledgements

Ray Pereda contributed to early drafts of this paper. Hasan Al-Sheboul supplied valuable comments and corrections.

References

1. [APL] Iverson, Kenneth E. 1962. *A Programming Language*. New York:John Wiley and Sons.

2. [CLU] "Abstraction Mechanisms in CLU". August 1977. *Communications of the ACM*, (8):564-576.
3. [CPG] *C* Programming Guide Version 6.0*. 1990. Cambridge, MA.:Thinking Machines Corporation.
4. [Hat91] *Data-Parallel Programming on MIMD Computers*. 1991. Cambridge, MA:MIT Press.
5. [Icon] Griswold, Ralph E. and Madge T. Griswold. 1997. *The Icon Programming Language*, 3rd Edition, San Jose, Peer-to-Peer Communications.
6. [Modula-3] Harbison, Samuel P. 1992. *Modula-3*, Englewood Cliffs, New Jersey: Prentice Hall.
7. [NetRexx] Cowlishaw, M. F. 1977. *The NetREXX Language*, New York:Prentice-Hall.
8. [Perl] Wall, Larry and Randal L. Schwartz. 1991. *Programming Perl*, Sevastopol, CA:O'Reilly and Associates.
9. [Pizza] Odersky, Martin and Philip Walder. January 1997. "Translating theory into practice," *24th ACM Symposium on the Principles of Programming Languages*, Paris, France.
10. [Pro97] Proebsting, Todd. June 1997. "Simple Translation of Goal-Directed Evaluation", *Proceedings of the SIGPLAN 97 Conference on Programming Language Design and Implementation*. Las Vegas, NV.
11. [Regular Expressions] Friedl, Jeffery E. F. 1997. *Mastering Regular Expressions*. Sevastopol, CA:O'Reilly and Associates.
12. [Snobol] Griswold, R. E., J. F. Poage, and I. P. Polonsky. 1971. *The SNOBOL4 Programming Language*, 2nd Edition, New York:Prentice-Hall.
13. [Sun] *Java JIT Compiler Overview*. `www.sun.com/workshop/java/jit/explanation.html`.
14. [Toba] John H. Hartman, Tim Newsham, and Scott A. Watterson. June 16-20, 1997. *Toba: Java for Applications -- A Way Ahead of Time (WAT) Compiler*. USENIX 3rd Conference on Object-Oriented Technologies and Systems (COOTS). Portland, OR.