Embedding Goal-Directed Evaluation through Transformation

A Dissertation

Presented in Partial Fulfillment of the Requirements for the

Degree of Doctorate of Philosophy

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Peter H. Mills

Major Professor: Clinton Jeffery, Ph.D.

Committee Members: James Alves-Foss, Ph.D.

Michael Anderson, Ph.D.

Gregory Donohoe, Ph.D.

Department Administrator: Frederick Sheldon, Ph.D.

May 2016

Authorization to Submit Dissertation

This dissertation of **Peter H. Mills**, submitted for the degree of Doctor of Philosophy with a Major in Computer Science and titled "**Embedding Goal-Directed Evaluation through Transformation**," has been reviewed in final form. Permission, as indicated by the signatures and dates below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: ________________________ Date:____________
Clinton Jeffery, Ph.D.

Committee Members: ________________________ Date:____________
James Alves-Foss, Ph.D.

________________________ Date:____________
Michael Anderson, Ph.D.

________________________ Date:____________
Gregory Donohoe, Ph.D.

Department
Administrator: ________________________ Date:____________
Frederick Sheldon, Ph.D.

Abstract

Goal-directed evaluation is a computational paradigm that combines the power of generators with backtracking search. In goal-directed evaluation every expression is a generator that produces a sequence of values until it fails, and operations iterate over the cross-product of their operands to find successful results. Introduced in the influential dynamic language Icon and later refined in its object-oriented descendent Unicon, goal-directed evaluation is a powerful yet unconventional paradigm that can succinctly express search over a product space as well as aggregate operations amenable to parallelization. The goal of this research is to extend the benefits of this paradigm into other more familiar object-oriented languages through mixed-language embedding. However, grafting goal-directed evaluation onto other languages in a manner that provides seamless interoperability has remained an elusive challenge.

This dissertation presents a novel approach to embedding goal-directed evaluation into existing object-oriented languages based on program transformation. We first introduce annotations for mixed-language embedding that allow mixing in Unicon functionality at the level of expressions, methods, or classes. Transformations over the annotated regions then unravel the syntax of generator expressions to a conventional form by flattening nested generators and making iteration explicit, in order to enable native evaluation. The rewriting rules then translate control constructs and operations onto a stream-like calculus for composing suspendable iterators. The transformations provide a formal semantics for Unicon that enable embedding into a broad variety of object-oriented languages. To demonstrate the utility of the approach, we have implemented the transformations for Java as well as its dynamic analogue Groovy, and housed them in an interpretive harness that can function both interactively and as a translator for compilation.

We further extend the transformations to enable mixed-language embedding of concurrent generators. We present a simple model of explicit concurrency for generators based on co-expressions and multithreaded generator proxies. We demonstrate how the concurrency abstractions can express parallel pipelining, as well as build higher-order abstractions such as map-reduce. Mixed-language embedding allows the use of concurrent generators within a familiar object-oriented setting, both for high-level coordination and the prototyping and refinement of parallel programs for multi-core architectures.

Acknowledgements

I would like to thank my advisor, Clint Jeffery, for his guidance and insight during the course of my dissertation research. I will miss our many and long interesting discussions on the complex field of programming using generators and the subtleties of Unicon semantics. I would also like to thank my committee members, Jim Alves-Foss, Gregory Donohoe, and Michael Anderson, for their review and input during this process. Lastly, I would like to thank my wife, Professor Leah Bergman, for her endless patience and support. It is to her that I dedicate this dissertation.

Table of Contents

List of Figures

Chapter 1. Introduction

1.1 Problem statement

Goal-directed evaluation is a powerful computational paradigm that was first introduced in the influential dynamic language Icon [Griswold et al. 1981, Griswold and Griswold 1996] and later refined in its object-oriented descendent Unicon [Jeffery 2001, Jeffery 2013a, Jeffery 2013b]. In goal-directed evaluation, every expression is implicitly a generator that produces a sequence of values or fails, and operators and function application iterate over the Cartesian product of their arguments to find successful results. The notion of generator functions has its origin in the language CLU [Liskov et al. 1977, Liskov et al. 1981], where a function can yield a result and then suspend until the next value is needed. In Icon and Unicon this concept is extended into a compact and dynamically typed notation that combines the power of generators with backtracking search.

Such backtracking search for successful results entails a depth-first traversal over generator expressions that rejects candidates on failure, in a manner similar to logic-programming languages such as Prolog [Kowalski 1979, Clocksin and Mellish 1981]. Unlike Prolog, however, it is the implicitly aggregate nature of operations over sequences that produces the combinations to be filtered for success. Goal-directed evaluation is thus a powerful but unconventional paradigm that can succinctly express search over a product space as well as aggregate operations that can potentially be executed in parallel.

While such a paradigm can succinctly express search, its sophistication makes it challenging to formalize and implement, as evidenced by various techniques that have been investigated for its translation. One notable effort, in particular, produced a Java bytecode generator for Icon called Jcon [Proebsting and Townsend 2000] that employed a Prolog-like Byrd-box model using fail and resume ports [Proebsting 1997]. A number of other studies looked at continuation-based approaches for cross-translation [O'Bagy and Griswold 1987, Allison 1990, Walker and Griswold 1992, O'Bagy et al. 1993] as well as for formally defining the semantics of Icon, including a denotational semantics [Gudeman 1992] and a semantics based on list and continuation monads [Danvy et al. 2002].

Despite these efforts, goal-directed evaluation is based on such a differing evaluation paradigm that interoperability with other languages is severely problematic. The Jcon implementation, in particular, faced difficulties in transparently interfacing with other Java programs due to its instrumentation of data types and expressions with suspend and resume advice as well as its reliance on direct bytecode generation. And yet, such interoperability is highly desirable to leverage Java's portability as well as its powerful facilities for concurrency and graphics. In particular, several Unicon research efforts have specifically focused on extensions for multithreaded concurrency and collaborative virtual environments [Al-Gharaibeh et al. 2012a, Al-Gharaibeh 2012b, Al-Gharaibeh et al. 2011], areas that could potentially benefit from cleanly interfacing with Java's extensive support for multithreading and 3D graphics. Similar problems of interoperability arise when calling down into goal-directed evaluation from another language. A key goal is thus to achieve seamless interoperability with the translation target, which would allow Unicon values and class fields to be passed to and from native methods, and vice-versa.

An even stronger argument can be made for the more general case of *embedding* goal-directed evaluation into object-oriented languages, to expand the reach of its power in a familiar and minimally invasive fashion. It is feasible to infuse the design and implementation of a new language with goal-directed evaluation, as was done with Converge, a Python-ish language that incorporated Icon-like expressions [Tratt 2010]. However, a capability for grafting goal-directed evaluation onto *existing* languages in a manner that provides seamless interoperability has remained an elusive challenge. The goal of this research is to extend the benefits of this paradigm into other more familiar object-oriented languages through mixed-language embedding.

1.2 Mixed-language embedding through transformation

This research presents a novel approach to embedding goal-directed evaluation into existing object-oriented languages based on program transformation. We introduce a form of annotations for mixed-language embedding, called scoped annotations, that allow mixing in Unicon functionality at the level of expressions, methods, or classes. Transformations over the annotated regions then unravel the syntax for generator expressions to a conventional form, while remaining largely oblivious to the grammar of the surrounding language. The transformations first reduce primary expressions such as function invocation and object field reference to a normal form that flattens nested generators and makes iteration explicit, in order to enable native evaluation. The rewriting rules then translate control constructs and operations into a stream-like interface for composing suspendable iterators, called an iterator calculus, that extends the bound products used in flattening with functional forms such as *reduce*, *map*, and *exists*. Lastly, to accommodate Unicon's first-class reference semantics, methods are exposed as variadic lambda expressions that produce iterators, while fields are exposed in both plain and reified form. The transformations are benign in that they leave code foreign to Unicon unchanged, while within the regions of embedded Unicon code, the mechanisms used to unravel the syntax to a conventional form provide seamless interoperability with other object-oriented languages.

To demonstrate the utility of the approach, we have implemented the transformations to target Java as well as its dynamic analogue Groovy [Dearle 2010]. The transformations are housed in a generic interpretive harness that, together with a small kernel for composing suspendable iterators, realizes both an interactive extension of Groovy and a translator of embedded goal-directed evaluation into Java. The interpreter, which we call Junicon, supports the spectrum of approaches from embedding goal-directed evaluation into Java or Groovy at the expression or method level, to providing a full-fledged Java-based implementation of Unicon at the class level [Mills and Jeffery 2016a, Mills and Jeffery 2016b, Mills and Jeffery 2014].

The transformations are formally specified at a high level as rewrite rules, in a manner applicable in general to conventional object-oriented languages. The transformations are then implemented for Java using the rule-based language XSLT (XML Language for StyleSheet Transformations) [Kay 2008, Clark 1999, Clark and DeRose 1999]. The XSLT transformations rewrite XML abstract syntax trees, derived from parsing the source language, and then deconstruct them into the target language. A compact kernel in Java then implements the calculus for composing suspendable iterators, and serves as the final target of the transformations. With only slight modification, the transforms have also been retargeted to Groovy [Dearle 2010], a dynamic language that extends Java with relaxed type declarations and dynamic dispatch of methods based on their runtime types. Using these two sets of transforms, the implementation is able to act both as an interactive interpreter, and as a translator for compilation that is free of dependencies on Groovy.

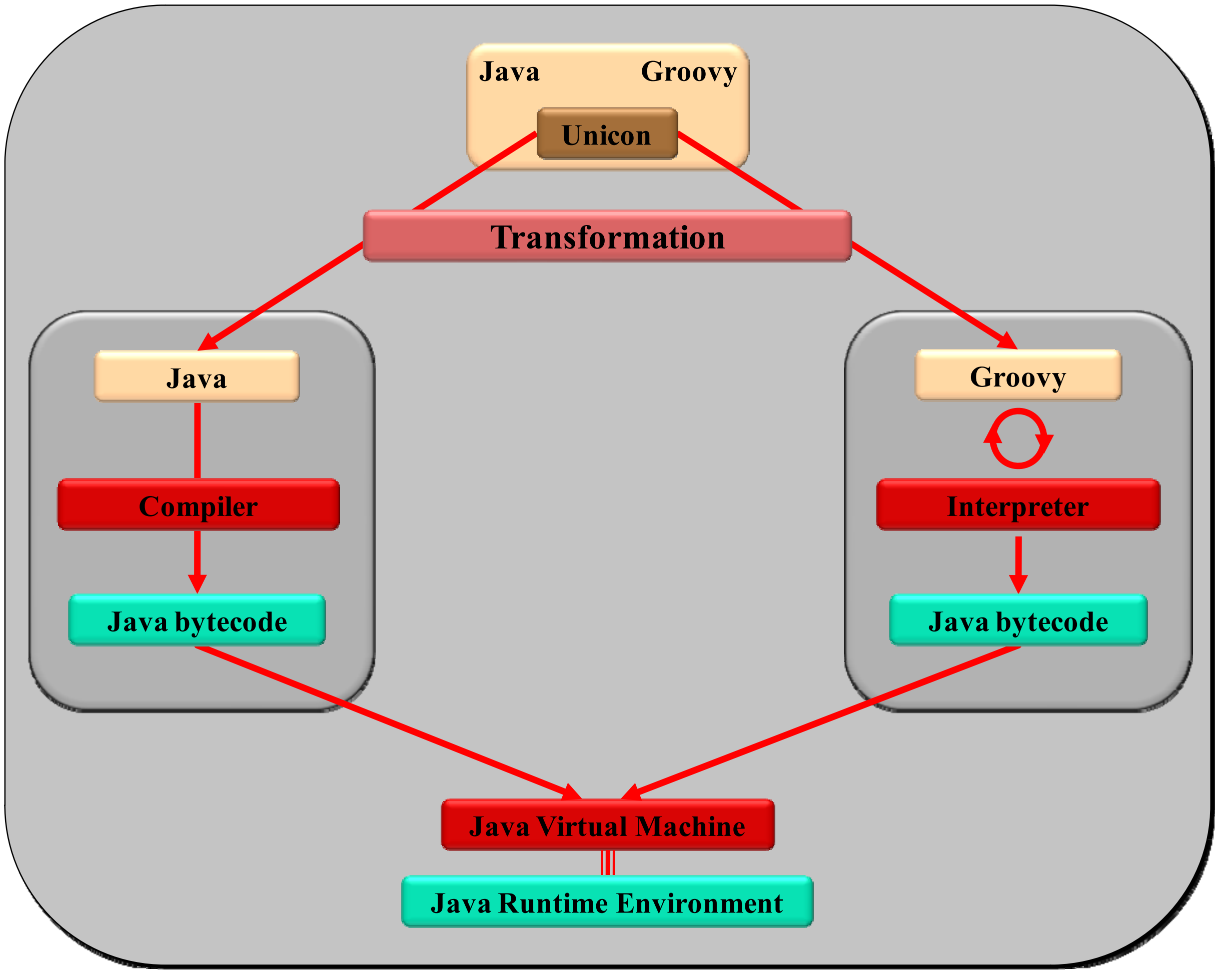


Figure 1. Embedding dynamic languages through transformation.

Figure 1 illustrates the approach of high-level transformation across dynamic languages that forms the basis of Junicon's interpretive capability, and how it contrasts with that for compilation. In Figure 1, on the left is the transformation path for compilation, in which embedded Junicon is first transformed into Java, and then translated into low-level bytecode that is run in conjunction with a Java Virtual Machine and its runtime libraries. In contrast, on the right of Figure 1 is the transformational interpreter that achieves interactive execution of embedded goal-directed evaluation by targeting Groovy, and leveraging in turn its script engine that iteratively translates statements into Java bytecode and then dynamically loads and executes it. The Junicon transformational interpreter thus can act both interactively and as a translator. The former is highly advantageous for exploration as well as rapid prototyping and refinement within an iterative development methodology, while the latter enables potentially more efficient programs that are free of dependencies on the extensive Groovy runtime support libraries. Embedded Junicon programs, when compiled to Java, are dependent only on the Junicon runtime library for suspendable iterators, which has a very small footprint of only 150K, and has no dependencies other than on core Java facilities in the standard Java distribution.

1.3 Embedding concurrent generators

We further extend the transformations to enable mixed-language embedding of concurrent generators. Goal-directed evaluation and its pervasive use of generators are potentially a natural fit for expressing concurrency. However, while such a paradigm can succinctly express search, several challenges remain in its effective application in parallel computation.

The first challenge lies in developing concurrency abstractions that mesh with pervasive generators and are cleanly amenable to implementation. To answer this challenge, in this research we present a minimalist set of concurrency mechanisms for Unicon that accommodates both Icon's coroutines, called co-expressions, and multithreaded communication between them using pipes. Co-expressions are coroutines that shadow the environment to prevent interference, while pipes are generator proxies that communicate with the original expression running in a separate thread using the put and take operations of blocking queues. Together these mechanisms are sufficient to express parallel pipelining, and to build higher-order abstractions such as map-reduce.

While it may seem a bit of an oxymoron to introduce concurrency into a dynamic language such as Unicon characterized by relaxed typing and dynamic dispatch, the benefits lie in improving performance, in enabling the prototyping and exploration of parallel algorithms as well as their iterative refinement, and in the use of concurrent generators for high-level coordination among larger-grained processes expressed in other languages. In particular, the latter benefits strongly argue for the capability to embed goal-directed evaluation into other more efficient object-oriented languages that support parallelism. Such a capability would expand the reach of generators within a familiar setting and allow their use for coordination as well as refinement. In contrast to other dynamic languages that support parallelism [Wilde et al. 2011, Smolka 1995, Bezanson et al. 2014, Lu et al. 2014], our goal is to support mixed-language programming across fundamentally differing computational paradigms, rather than just to support multiple paradigms of concurrency within a single language.

To answer the above challenge of embedding concurrent generators into existing object-oriented languages, we extend the transformational approach to yield an implementation of co-expressions as well as multithreaded generator proxies. Our work provides a model of concurrent generators that simplifies and unifies the thread-based model previously developed for Unicon in [Al-Gharaibeh et al. 2012a, Al-Gharaibeh et al. 2012b], and enables its embedding into other object-oriented languages.

First, we present a simple model of explicit concurrency for generators based on co-expressions and multithreaded generator proxies. We demonstrate the utility of the model in expressing parallel pipelining, as well as building higher-order abstractions such as map-reduce, in a mixed-language setting that uses scoped annotations to specify the embedding of concurrent generators. Second, we present techniques for transforming concurrent generators into Java that leverage Java's facilities for multi-threaded concurrency. The techniques rely on flattening as well as on synthesizing co-expressions and multithreaded generator proxies. Mixed-language embedding allows the use of the succinct notation of concurrent generators within a familiar object-oriented setting, both for high-level coordination and the prototyping of parallel programs for multi-core architectures. In particular, the provision for interactive evaluation further enhances the ability for exploration and rapid prototyping.

1.4 Contributions

The contributions of this research are as follows. ***First, we define a novel form of annotations, called scoped annotations, that can be used for mixed-language embedding.*** The annotations specify the language of a region to be embedded, and allow mixing Unicon functionality into Java, as well as of Java functionality into Unicon.

Second, we develop a technique for flattening nested generators that enables the grafting of goal-directed behavior onto other languages with seamless interoperability. By reducing pervasive generator expressions to a recognizable and explicit form, we clarify the semantics so that it can be understood in terms of conventional programming language concepts. We present a context-sensitive reduction semantics for flattening in the style of Felleisen-Hieb [Wright and Felleisen 1994]. The rewriting rules used to unravel the syntax to a conventional object-oriented form provide an operational semantics that has potentially broader applicability for embedding Unicon into a wide range of target languages.

Third, we derive a calculus for composing suspendable iterators that extends the products of bound iterators used in flattening with functional forms such as reduce, map, and exists. The iterator calculus forms a minimal basis into which Unicon's control constructs and operators are transformed, and can be efficiently implemented in a tightly knitted kernel. The equational translation of constructs into that basis further provides a clear operational semantics of Unicon.

Fourth, we extend the transformational approach to the area of concurrency in Unicon. We present a model of concurrent generators based on co-expressions and multithreaded generator proxies, called pipes. We extend the transformations to allow embedding of concurrent generators, which enables their use both for the high-level coordination of processes in other languages, and the prototyping and refinement of parallel programs for multi-core architectures.

Lastly, we realize a transformational interpreter for embedding goal-directed evaluation into Java as well as Groovy, that can function either as a translator or interactively, respectively. By carefully transforming onto Java in a manner that preserves types such as lists using their Java equivalent, we can seamlessly integrate with and leverage the full range of Java capabilities. By also targeting a language such as Groovy, we are able in turn to realize an interactive interpreter that permits line-by-line evaluation. Such interactive evaluation enhances capabilities for exploration and rapid prototyping. The implementation also demonstrates how mixed-language embedding using transformation can be used to extend languages with advanced features such as suspendable generator functions.

The remainder of this dissertation is organized as follows. Chapter 2 first provides more detailed background on Icon and Unicon. In Chapter 3 we describe scoped annotations and illustrate their application to mixed-language embedding for both Groovy and Java. Chapter 4 presents a model of concurrency for generators, and examines their application to expressing parallel pipelining and map-reduce within a mixed-language setting. In Chapter 5 we describe the transformations that flatten generator expressions, as well as the translation of methods into variadic lambda expressions. Chapter 6 then provides details on the implementation of the transformations, formalized as rewrite rules in Chapter 5. Chapter 7 presents the results of benchmarking goal-directed evaluation when translated to Java. Lastly, Chapter 8 reviews related work, and we present our conclusions in Chapter 9.

Chapter 2. Background

Icon is a unique programming language designed as the successor to the string pattern matching language SNOBOL [Griswold et al. 1971]. At the heart of Icon is the notion of a generator, which is an expression whose evaluation lazily yields a sequence of values, i.e., generates them one at a time on demand. As in CLU and later languages such as C# [Jagger et al. 2007] and Python [Rossum and Drake 2011], generators in Icon can be constructed using generator functions that use a *suspend* statement – corresponding to CLU's *yield* – to return a value and on the next iteration resume at the point of suspension.

For example, invocation of the following Icon procedure will result in a generator that produces an ascending sequence of values:

```
procedure range (from, bound)
    local count;
    count := from;
    while (count<=bound) do { suspend count; count +:= 1 }
end
```

Thus, the expression *range*(*1,2*) yields the values 1 and 2, and then fails. Such generators can be used in lieu of collections in loops and list comprehension. While Icon uses the term generator, we will use the term iterator interchangeably with it, and will distinguish it from a Java *Iterator* when necessary. In the examples that follow, we use a modernized version of the Pascal-like syntax of Icon and Unicon, in which = is used for assignment instead of :=, *def* may be used instead of *procedure* or *method*, and braces {} may be used for block structure instead of *end*.

2.1 Goal-directed evaluation

However, Icon goes a step further than CLU in its pervasive use of generators. In Icon *every* expression is a generator that produces a sequence of values or fails[1], and nested generators are *implicitly* composed by mapping functions or operations over the cross-product of their arguments, and then filtering to find successful results. For example, consider the simple expression that finds multiples of prime numbers in a given range:

```
(1 to 2)  *  isprime (4 to 7)
```

where *isprime*(*x*) is defined to produce *x* if it is prime, and otherwise fail, and the *to* construct produces a range of numbers.

The above expression will, for each value in the sequence (1,2), iterate through each value in the second sequence (4,5,6,7) and for the latter that are prime numbers, yield their product. The compound expression itself forms a generator that, at each iteration, searches to find the next successful result, and so produces the sequence 1*5, followed by 1*7, then 2*5, then 2*7. Such search has particular application in string processing, the forte of Icon and Unicon.

The implicit composition of nested generators in Icon may be more clearly understood by decomposing it in terms of Icon's product operator,

```
e & e'
```

which for each *i* in *e*, iterates over each *j* in *e'*, and yields *j* as the next successful result of iteration. Formally,

[1] Strictly speaking, in Icon parlance a generator is an expression that can produce multiple results. In this dissertation we use the term generator to also include expressions that only produce at most one result. Failure can be modeled either with an explicit fail value, as is done here, or implicitly as the empty sequence, as in [Danvy et al. 2002].

```
e & e' ≡ filter(succeed (for i in e { for j in e' { j } }))
```

Function application, *f*(*e,e'*), is then equivalent to

```
f(e,e') → (i in e) & (j in e') & (k in f(i,j))
```

where (*i* in *e*) denotes bound iteration that assigns each value in the iterator sequence for *e* to a variable *i*, and where *f* is a generator function that produces a sequence of values on invocation. The final result of the above expression will be a sequence whose values are bound to *k*. Applying the above transformation to operators, (*i*=*e*) can further be seen to be equivalent to (*i* in *e*).

The above example of prime multiplication can thus be recast as an iterator product:

```
i=(1 to 2)  &  j=(4 to 7)  &  isprime(j)  &  i*j
```

which corresponds to the Python generator expression:

```
(i*j for i in range(1,2) for j in range(4,7) if isprime(j))
```

and represents nested iteration. Conversely, a Python generator expression

```
f(x) for x in S if P(x)
```

is equivalent to Icon's

```
(x=S) & P(x) & f(x)
```

Generator expressions are also closely related to Java streams as well as monad comprehension [Mills and Jeffery 2016a]. In the terminology of Java streams,

```
f(e) → e.flatMap(x->f(x)).filter(succeed)
```

The above examples highlight how goal-directed evaluation combines generators with the concept of success and failure. An expression, at each iteration, succeeds and produces a value, or fails and terminates the iterator, which in turn fails. In other words, generators, when viewed as Java iterators, are terminated by failure of the *next*() method. Moreover, at each iteration, an operation will generally be performed only if the operands all succeed, and otherwise it fails. Thus, expression evaluation is conditioned on the success of its terms. For example, the expression:

```
x = if (k > 0) then k
```

will perform an assignment to *x* only if $k > 0$, since otherwise the *if* expression fails, which in turn causes the assignment to fail. In Icon, the concept of true or false is thus replaced with success and failure, and failure propagation is analogous to a bottom-preserving or undefined-preserving semantics. Failure propagation also applies to function invocation. For example, *f*(*x*,*y*) will fail if either of the arguments *x* or *y* fails, and therefore the call to *f* will not occur. Similarly, in the iterator product *x* & *y*, if at a given iteration point the precondition *x* fails, then *y* is not evaluated. The & operator is thus fundamental as it embodies notions of both cross-product and conditional evaluation.

It is important to note that, since every expression is a generator, their composition, and the program in toto, just yields one large iterator. Even the familiar sequence construct, *a*;*b*;*c*, denotes the concatenation of iterators that runs through *a* and *b* as singleton iterators that are limited to producing at most one result, called bounded expressions, and then delegates remaining iteration to the last term *c*. Actual iteration, i.e., executing the iterator's *next*(), only occurs at the outermost level of interaction. These points occur at interpreted statements outside a class definition, in class field initializers, and in the main method of a program.

Icon also provides a notion of reversible assignment, *x <- y*, which on more than one iteration, reverses the assignment and then fails. For example, in the unbounded expression (*x*<-*y*) & *e*, if *e* fails, *x* will revert to its original value. Reversible assignment, along with string scanning, is one of the few traces of implicit data backtracking in Icon, since elsewhere state is not saved or restored.

2.2 Reference semantics

Icon further provides an expressive reference semantics that supports lazy dereferencing. Variable references are treated as first-class citizens in generator expressions, and are only dereferenced when needed. This derefencing occurs, for example, when variables are used as arguments to operators or methods, and therefore methods have call-by-value semantics. Expressions can thus yield variable names to be assigned or invoked. For example, in the expression:

```
(if i > j then i else j) = 0
```

the value 0 is assigned to *i* or *j* depending on their comparison. Under the hood, the generator on the left-hand-side of the assignment produces a reference to *i* or *j*, which is then used by the assignment operator. Indexing operations are also first class citizens, in that an index such as *c*[*i*] is maintained as an offset into a collection until its value is needed, and so provides an updatable reference. The above implies that variables need to be exposed as reified properties with getters and setters, in order to be passed as updatable references.

Similarly function names used in invocation can themselves be generator expressions. For example,

```
(f | g)(x)
```

where **|** means concatenation of generators, is equivalent to:

```
f(x) | g(x)
```

and so iterates first through *f*(*x*) and then *g*(*x*). The above implies that method references, or some form of lambda abstraction, may be required for implementation.

Goal-directed evaluation thus embodies a conventional syntax with a very unconventional meaning. The provision of implicit search as well as the reference semantics, while powerful, pose formidable challenges in transparently interfacing with other languages.

2.3 Co-expressions and threads

Icon and Unicon also provide support for interleaving as well as true concurrency in the form of coroutines and multithreaded interaction, respectively. A coroutine is an expression that can suspend and transfer control to another expression, and when called again will resume at the point of suspension with its environment intact. Icon incorporates a notion of coroutines, called co-expressions, that preclude interference by copying local variable references upon creation, and that are explicitly stepped on each iteration using an @ operator.

Unicon then extends co-expressions with support for true concurrency in the form of multithreaded interaction. Previous work on concurrency in Unicon [Al-Gharaibeh et al. 2012a, Al-Gharaibeh et al. 2012b] extends Icon with co-expressions that can run in a separate thread, and communicate with each other through a variety of synchronization mechanisms including explicitly created blocking channels and mutexes. While sufficient to express a wide variety of parallel interaction, communication is explicit, and to date there is no mechanism to naturally chain together generators at a high level.

2.4 Java and its dynamic analogue Groovy

While our research focuses on general techniques for embedding goal-directed evaluation into any conventional object-oriented language, the choice of Java as a translation target for the implementation is carefully chosen. Java [Gosling et al. 2014] is a statically typed object-oriented language that is compiled to a platform-independent intermediate bytecode, and then run in conjunction with a platform-specific JVM (Java Virtual Machine) that interprets the bytecode. The widespread adoption and popularity of Java arises from its clean design, portability, and vast array of libraries such as for graphics and concurrency, including support for multithreaded execution on multi-core architectures.

In contrast, Groovy [Dearle 2010] is a dynamic version of Java which is implemented by compilation into Java bytecode that runs on any Java virtual machine. Like Icon, Groovy is dynamically typed, in other words variables need not be declared as having a type. Groovy also provides seamless interoperability with Java, in that Groovy class instances and data types can be transparently passed to and from Java, with class fields accessed, and similarly methods invoked, from either side. A notable feature of Groovy is its provision for parameterized closures, which expresses lambda abstraction, also called lambda expressions in Java. For example,

```
def f = {x,y -> x+y}
```

defines *f* as a function formed from the closure of the expression to the right of the arrow, and with parameters *x* and *y*. A closure in Groovy always returns the last argument. As expected, *f*(1,2) will thus yield the value 3. An equivalent lambda expression in Java is:

```
Function f = (x,y) -> { return x + y; }
```

Lastly, Groovy also provides an interpreter for interactive line-by-line evaluation: this interpreter is exposed as a Java-compliant scripting engine that can be used to create a transformational interpreter. The above features, in particular its relaxed typing and its provision of closures, make Groovy an attractive translation target for embedded Unicon in a manner that supports interactive interpretation.

While Unicon is a powerful language whose dynamic typing makes it easy to use, there was previously no way to cleanly embed it into Java or Groovy, nor arguably a clear operational semantics of its underlying goal-directed evaluation that would drive such an implementation. Yet embedding Unicon into Java, and in particular providing an interpreter for such a capability through Groovy, has many potential advantages. These advantages include portability, access to Java concurrency and graphics utilities, the use in web applications, and the potential to integrate into the increasingly widespread Java-and-Linux-based Android Operating System for mobile handhelds. In contrast to earlier efforts for a Java based Icon implementation [Proebsting and Townsend 2000], our research focuses on embedding goal-directed evaluation through higher-level cross-translation that maintains consistency with the Java type system.

Chapter 3. Multi-Language Integration

A primary goal of this research is to enable the use of goal-directed evaluation within the broader scope of other languages. In particular our approach to support multiple languages and paradigms is to specify embedding in a manner where the embedded regions are oblivious to the grammar of the surrounding context. In our implementation we do not need parsers for Java or Groovy. Rather, we only need a general metaparser that recognizes complete statements, based on grouping delimiters such as braces and parentheses, in order to recognize embedded regions. Within a transformational framework, each embedded region is then transformed and injected into the surrounding context, from the innermost outwards, to yield the final program. The exact transforms are dependent on both the embedded and surrounding language types.

3.1 Scoped annotations

To specify embedded regions we introduce a novel form of annotations, called scoped annotations, that blend Java annotations and XML. For example, scoped annotations of the form:

```
@<script lang="junicon"> x = f(g(y)); @</script>
```

are used to specify the embedded language, and delimit the sections of code where flattening of generator expressions occurs. Scoped annotations have the following admissible forms:

@<tag attr$_1$=x$_1$ ... attr$_n$=x$_n$> expression @</tag>
@<tag attr$_1$=x$_1$ ... attr$_n$=x$_n$/>
@<tag(attr$_1$=x$_1$, ... ,attr$_n$=x$_n$)> expression @</tag>
@<tag(attr$_1$=x$_1$, ... ,attr$_n$=x$_n$)/>

where the tag name may be qualified with either an XML namespace or Java package name, respectively. Like XML, such annotations can surround multiple statements, and can also be nested. Unlike conventional Java annotations that are limited to attaching metadata to declarations or type use, scoped annotations can in addition modify expressions as well as arbitrarily delimited sections of code.

The above syntax of annotations for multi-language embedding is carefully chosen so that, similar to XML, it is attributed and scoped. The syntax is also chosen to be familiar and closely allied to Java annotations as well as other notations for scripted embedding such as HTML. However, the syntax is constrained so that its embedded use does not conflict with most programming language notations, nor does it collide with other forms of annotations such as for Java. In particular it differs from other notations such as JSP (Java Server Pages) in that, because it is tag-based, it can support multiple languages, and a single syntax supports both embedding and translator directives.

In a dual manner, a scoped annotation with @<*script lang*="*java*"> specifies native Java evaluation. When used outside a Unicon region, the latter exempts the section of code from being transformed, and so it is directly compiled, or if interactive, is passed through to a Groovy script engine. When used within a Unicon region, it lifts the code into a singleton iterator over its closure, so that it can participate in goal-directed evaluation. The transformation that takes scoped annotations for native evaluation into goal-directed evaluation is as follows:

```
@<script lang="java> code @</script>
      → ! ()->{< code >}
```

```
@<script lang="groovy">
class PrimeMultiples {
  def  printAll (lower, upper) {
      for (i in findPrimes(lower, upper)) { println(i); };
  }

  @<script lang="junicon">
    def  findPrimes (lower, upper) {
      suspend (lower to upper) * isprime(lower to upper);
    }
  @</script>

  def   isprime (num) {
    def  result =
    @<script lang="junicon">
    {  local i;
      if not((i = 2 to Math.sqrt(num)) & (num % i == 0)) then num else fail
    };
    @</script>
    return result;
  }
}
@</script>
```

Figure 2. Embedding goal-directed evaluation into Groovy.

where {< … >} denotes block quotes used to capture the code text as a quoted string, which is then directly compiled to Java or evaluated by the Groovy script engine.

3.2 Mixed-language embedding

A more detailed example that illustrates embedding Unicon into Groovy is given in Figure 2, showing a chain of mixed paradigm invocations. As mentioned previously, Groovy [Dearle 2010] is a dynamic language that extends Java with relaxed type declarations and dynamic dispatch based on runtime types. The top of Figure 2 shows a Groovy method for printing all the prime multiples in a given range. The top method in turn invokes a Unicon method *findPrimes*, shown in the center of Figure 2. Invocation of the Unicon method returns a generator, exposed in the top method as a Java Iterator. Of particular note is that the Unicon method uses a *suspend* statement to create a generator function, a feature otherwise missing from Groovy as well as Java.

The Unicon method then cuts down to a Groovy method *isprime*, which in turn uses an Icon generator expression to test if the number has any divisors. The generator expression succeeds with a prime number, or fails, and that failure is passed back up to the invoking Unicon method. Lastly, within the generator expression a native Java method *Math.sqrt* is invoked. It bears noting that there is an implicit *next*() around the embedded generator expression in the annotation, so that it returns the first iteration.

Figure 2 illustrates several key features of our approach. First, goal-directed evaluation can be embedded at the method or expression level, as well as the class level if desired. Second, the embedding can be arbitrarily nested across differing languages. Third, the technique for embedding, in conjunction with the transformations described in Chapter 5, provides seamless interoperability when intermixing Unicon and Groovy, as well as Java. Specifically, native types can be transparently passed to and from Unicon, including class instances and collections such as lists, with fields accessed and methods invoked from either side. In particular, since Unicon methods return a Java iterator, they can be freely used from within Java as well as Groovy.

An equivalent example that illustrates embedding Unicon into Java is given in Figure 3. The top and middle portions of Figure 3 are largely unchanged, showing nearly transparent interoperability. However, the bottom *isprime* method has been enhanced to illustrate the differences when embedding to Java, as well as Junicon's support for lambda expressions and type coercion. At the bottom of Figure 3, the *isprime* Java method can be seen to create an

```
@<script lang="java">
public class PrimeMultiples {
  public void printAll (int lower, int upper) {
      for (Object i : findPrimes(lower, upper)) { System.out.println(i.toString()); }
  }

  @<script lang="junicon">
    def  findPrimes (lower, upper) {
      suspend (lower to upper) * this::isprime(lower to upper);
    }
  @</script>

  public Object isprime (Object num) {
    VariadicFunction<Number,Iterator> generator =
    (VariadicFunction<Number,Iterator>)
    @<script lang="junicon">
    { (num) -> {  local i;
         return if not((i = 2 to Math::sqrt((Double) num))
                & (num % i == 0)) then num else fail }
    };
    @</script>
    return generator.apply(((Number) num).doubleValue()).next();
  }
}
@</script>
```

Figure 3. Embedding goal-directed evaluation into Java.

embedded goal-directed lambda expression, and then invoke it to return the prime number or fail. Two subtleties arise here. First, the lambda expression is variadic[2] and returns an iterator, and so an explicit *next*() is required to return the first iteration, as shown in the very bottom of the figure.

Second, unlike Groovy, in Java a lambda expression is a functional interface that must be invoked with an explicit method name such as *apply*. This need for an explicit name also applies to method references, since method references in Java are just lambda expressions for

[2] The term variadic means that the function takes any number of arguments.

methods that already have a name. Since Unicon methods are exposed on the surface as method references in order to allow the use of function names in expressions, their use must be differentiated from native Java method invocation, achieved by using :: for the latter.

As illustrated in Figure 3, Junicon incorporates several syntactic features to support interoperability with Java, including casts for type coercion as well as types in variable declarations, package dot notation, instance creation using *new*, lambda expressions, and method references. The above features are provided to support accessing native Java classes and methods from within Junicon. Junicon also conservatively extends Unicon syntax to directly allow a familiar Java-like block notation for classes and methods.

The examples in Figures 2 and 3 highlight how Groovy's dynamic typing allows fully transparent bidirectional interoperability, while embedding in Java provides near-seamless interoperability that differs only slightly when calling Java methods that are foreign to Unicon. In particular, Figure 3 shows how native Java method invocation, as well as inline Java code, can be intermixed with goal-directed evaluation.

Scoped annotations are thus the first step towards grafting goal-directed evaluation onto other languages. In the next chapter we describe how scoped annotations can similarly be used to graft a model of concurrent generators onto other languages.

Chapter 4. Concurrency

4.1 Model of concurrency

Generators are a natural fit for expressing concurrency. In dynamic languages such as Icon and Unicon where every expression is a generator, there is a pervasive opportunity for exploiting concurrency. While their composition under goal-directed evaluation offers the potential for chaining generators together in parallel, no model has been proposed which leverages this potential. In this section we present a simple model of explicit concurrency for generators that naturally transforms the composition of generators into parallel pipelines tied together using blocking queues. The spartan set of concurrency operators is sufficient to express true concurrency in the form of parallel pipelining, as well as to build higher-order abstractions such as map-reduce.

Figure 4 presents a minimalist set of operators for concurrent generators. In Figure 4, the model of concurrency is based on a single unified notion of first-class iterators that must be explicitly stepped to the next iteration. The simplest first-class operator lifts a given expression into a singleton iterator that returns the original generator, in other words,

```
<>e → new Iterator() { next() { return e; } }
```

Explicitly stepping the first-class generator is through the @ operator:

```
@e → e.next()
```

Promoting the first-class entity back to a generator is through the ! operator:

```
!e → new Iterator() { next() { return @e; } }
```

which simply unravels it, or equivalently

<> e	First-class generator.
|<> e	Co-expression that shadows the local environment.
|> e	Generator proxy that runs in a separate thread.
@ c	Next, i.e., step co-expression one iteration.
! c	Promote co-expression to a generator.
^ c	Restart with a new copy of the local environment.

where ***e*** is an expression, and ***c*** is a co-expression.

Figure 4. Calculus for concurrent generators.

```
!e → repeatUntilFailure(suspend @e)
```

since the composed iterators must be suspendable. Lastly, the restart operator ^ resets the iterator to its beginning state.

4.1.1 Co-expressions

A co-expression is similar to a first-class iterator, but in addition creates a copy of its local environment, i.e., it shadows all referenced method local variables and parameters. In other words,

```
|<> e → ^(<>e)
```

where the refresh operator ^ for co-expressions is refined as follows, using Java's notation for lambda expressions and where (*x*,*y*,*z*) are all referenced locals:

```
^e → ((x,y,z)-> e) ((()->[x,y,z])())
```

Co-expressions thus minimize interference by isolating a copy of the local environment. In addition, as with a normal coroutine, the @ activation operator will transfer control between co-expressions so as to interleave the threads of execution.

4.1.2 Generator proxies

Lastly, a pipe is simply a generator proxy for a co-expression that runs in a separate thread and iterates until failure, and uses a blocking channel for the communication of results. A blocking channel, or blocking queue, has put and take operations that wait until the queue of results is not full or not empty, respectively. Each iteration of the proxy will wait until a result has been placed in the channel by the co-expression running in the separate thread. Thus the surrounding expression runs in parallel to the piped expression. In other words,

```
|>e → new Iterator() { next() { new Thread { run() {
        c=|<>e; while (!fail) { out.put(@c); }}}.start() }}
```

where *out* is the output blocking queue, and an @ operation on a pipe is *out.take*(). The output blocking queue is a plain Java *BlockingQueue*, and is exposed as a public field to permit further manipulation. Bounding the output queue buffer size can also be used to throttle a threaded co-expression.

In its simplest form, a singleton piped iterator that produces one result forms a future or mutable variable, whose put and take operations wait until the channel is empty or full respectively. Historically it has been well established that mutable variables are a fundamental building block that can be used to build higher-order concurrency abstractions. Examples of mutable variables can be found in Id Noveau's M-structures [Barth et al. 1991], the M-Vars of Parallel Haskell [Nikhil et al. 1995] and Concurrent Haskell [Peyton-Jones et al. 1996], and Linda's tuplespace operations [Carriero and Gelernter 1989], as well as in an earlier form in the single-assignment synchronization variables of CML that wait on read until being defined [Reppy 1991]. Not surprisingly, the derived blocking queues of Java are similarly powerful building blocks, and while they do not preclude the beneficial use of other synchronization mechanisms, they do provide the basis of a minimal framework for coordination.

The above calculus for concurrent generators is thus sufficient to express a wide variety of parallel computations. For example the simple expression

```
x * ! |> factorial(! |> sqrt(y))
```

will, for given generated sequences *x* and *y*, spawn off their factorial and square-root computations in parallel, effecting explicit task parallelism in the form of a pipeline. A pipeline consists of a chain of tasks where the output of each element is the input of the next, synchronized using some form of blocking queues. The above is in contrast to

```
x * y
```

which reflects implicit data parallelism over the two generator sequences, and is the type of aggregate operation naturally amenable to map-reduce. The term map-reduce [Dean and Ghemawat 2008] refers to a constrained parallel functional style whose aggregate operations consist of stages of map functors followed by an optional shuffle and then reduction. The paradigm typifies Java parallel streams, and is typically implemented by partitioning the stream and then having multiple worker threads perform data-parallel operations sequentially on the decomposed chunks of data.

For example, for a given chunk *c*, mapping a function *f* over the chunk would be expressed using generators as:

```
! |> f(!c)
```

where the ! operator lifts lists as well as co-expressions to iterators. The above formulation is subtly different from conventional map-reduce in that it enforces ordering between the results

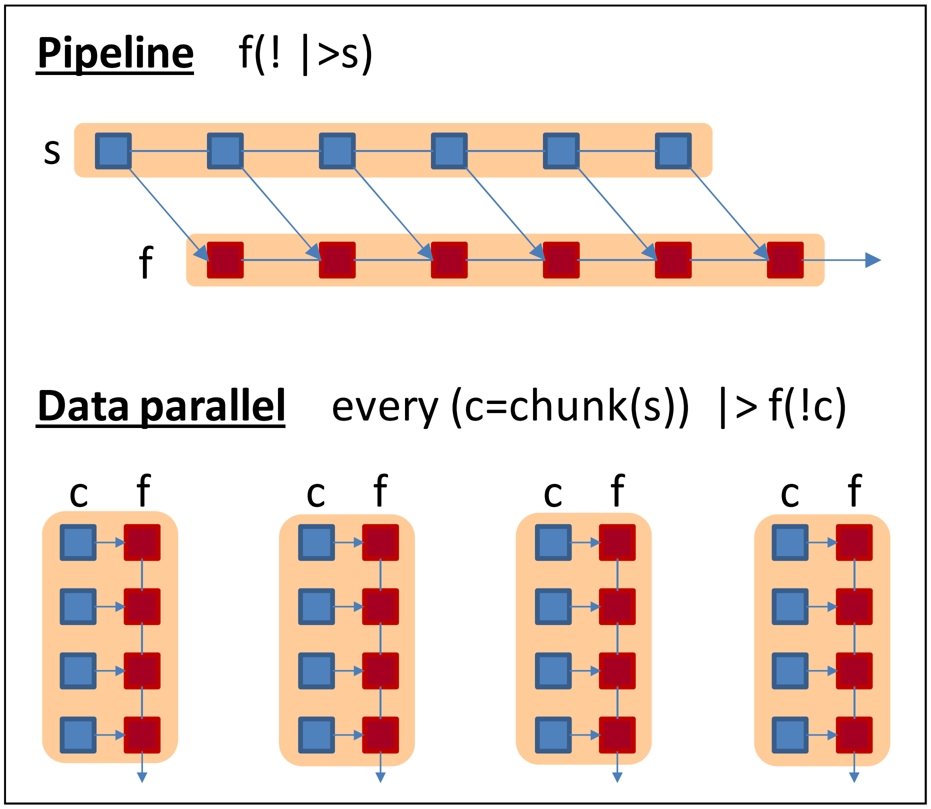


Figure 5. Pipeline and data-parallel models.

of the partitioned threads. However, such a formulation still requires the streams to be effectively splittable, i.e., have non-interference within a given stream operation.

The data-parallel decomposition of map-reduce thus differs from the calculus of concurrent generators: the former can be viewed as fixed-data that applies all pipeline stages to data distributed over threads, while the latter can be viewed as fixed-code that assigns a pipeline stage to each thread and exchanges data between them, using the terminology of [Bienia and Li 2010]. Figure 5 illustrates the relationship between pipelining and data-parallel decomposition when specified using concurrent generators. In Figure 5, the tan oblongs around the square sequence elements represent separate threads of execution, which for pipelines encapsulate an entire stream, while for data-parallelism encapsulate a chunk of the source stream over which the function is mapped.

In the next section we examine how, in conjunction with multi-language integration, concurrent generators can be used to build and explore higher-order concurrency abstractions such as map-reduce.

4.2 Mixed-language embedding of concurrent generators

A key facet of our approach is to enable the use of concurrent generators, and goal-directed evaluation in general, within the broader scope of other languages that support parallel programming. A detailed example that illustrates embedding concurrent generators into Java is given in Figure 6, showing a chain of mixed paradigm invocations. The program in Figure 6 takes lines of text, and computes a hash of the lines by splitting each line into words, converting the words into numbers, taking their square root, and then summing the result. Near the bottom of Figure 6, the Java method *runPipeline* can be seen to iterate over an embedded generator expression that spins off a pipeline to translate the words into numbers before computing their square root. The embedded expression returns a generator, exposed as a Java *Iterator* used in the *for* statement. The Unicon expression in turn cuts down to Java methods *wordToNumber* and *hashNumber*, as well as to a Unicon method *splitWords*.

At the bottom of Figure 6 is a map-reduce version of the *runPipeline* method, called *runMapReduce*. Its *mapReduce* method in turn is defined in Figure 7, and implements a simple variant of map-reduce defined in the previous section. In Figure 7, the first method *chunk* breaks up a source stream into chunks, each chunk being a list of fixed size. The second method *mapReduce* then, for each chunk derived using the generator function *s*, spins off a task to map the given function *f* over its elements, and then reduces the result with

```
class WordCount {
  static String[] lines;
  @<script lang="junicon">
    def readLines () { suspend ! lines; }
    def splitWords (line) { suspend ! ((String) line)::split("\\s+"); }
    def hashWords (line) {
       suspend this::hashNumber(this::wordToNumber( ! splitWords(line)));
    }
    def sumHash (sofar, hash) { return sofar + hash; }
  @</script>
  public Object wordToNumber (Object word) throws NumberFormatException {
      return new BigInteger((String) word, 36);
  }
  public Object hashNumber (Object word) {
    return new Double(Math.sqrt(((Number) word).doubleValue()));
  }
  public void runPipeline () {
    double total = 0;
    for (Object i :
      @<script lang="junicon">
          this::hashNumber( ! (|> this::wordToNumber( ! splitWords(readLines()))))
      @</script>
    ) { total = total + ((Double) i).doubleValue(); };
  }
  public void runMapReduce () {
    double total = 0;
    DataParallel dp = new DataParallel(1000);
    for (Object i : dp.mapReduce(hashWords, readLines, sumHash, 0) {
       total = total + ((Double) i).doubleValue();
    };
  }
}
```

Figure 6. Embedding concurrent generators into Java.

the reduction function *r* and initial value *i*. Lastly, the *mapReduce* method returns a generator over the results of each chunk.

The programs in Figures 6 and 7 demonstrate how embedding concurrent generators, and in general a dynamic language for goal-directed evaluation, can be used to explore and

```
class DataParallel {
  public DataParallel (int size) { this.chunkSize = size; }
  int chunkSize = 1000;
  @<script lang="junicon">
  def chunk(e) {                      # Partition e into chunks
    chunk = [];
    while put(chunk,@e) do {
      if (*chunk >= chunkSize) then { suspend chunk; chunk=[]; }
    };
    if (*chunk > 0) then { return chunk; };
  }
  def mapReduce(f,s,r,i) {            # Map f over s and reduce with r
    var c, t, tasks = [];
    every (c = chunk(<>s)) do {
      t = |> { var x=i; every (x=r(x, f(!c) )); x };
      ((List) tasks)::add(t);
    };
    suspend ! (! tasks);
  }
  @</script>
}
```

Figure 7. Building map-reduce using concurrent generators.

prototype the comparative performance of parallel algorithms. In particular, since the implementation of the calculus for concurrent generators leverages the Java facilities for task management and communication, their seamless integration permits iterative refinement into Java. As mentioned in the introduction and observed in evaluation, when using an embedded dynamic language, and in particular one supporting goal-directed evaluation, there is a tradeoff of performance for succinctness. The role of embedded generators in a parallel setting is thus envisioned to be one of exploration and prototyping, as well as potentially one of coordinating more computationally intensive pieces encoded in languages such as Java.

In the next chapter we describe the transformations that enable goal-directed evaluation to be embedded into other languages with seamless interoperability.

Chapter 5. Transformation

In this chapter we formalize the transformations that embed Unicon into Java, and with little modification into its dynamic analogue Groovy. A fundamental challenge in embedding goal-directed evaluation is that it is based on such a different paradigm that interoperability with other languages is severely problematic. Our approach to solve this problem is to unravel the syntax to a conventional form in a manner that enables native function invocation, and so maintains seamless interoperability with the surrounding target language.

Program transformation is a broad term that refers to changing the form of a program into another one that is semantically equivalent, or, for example in some cases of refinement, more specific [Feather 1987, Reddy 1990, Li 2010]. While program transformation encompasses translation, which includes compilation and interpretation, as well as the formal refinement of specifications and rephrasing, our focus here is on what is sometimes called migration, that is, translation into another language at the same level of abstraction [Visser 2005].

The transformations that take Unicon into Java and Groovy are formalized as term rewriting rules, and so yield a partial operational semantics. Indeed, all operational semantics can be understood as rewriting logics [Serbanuta et al. 2009]; transformations may be understood as differing in that they descend to intermediate representations rather than final values.

The motivation for targeting the transforms to both Java and Groovy arises from the desire to complement the performance and smaller footprint of Java with the interactive interpretation provided by Groovy. While it is possible to translate Junicon into Java bytecode using the Groovy compiler on the transformed Groovy program, the motivations of improved performance as well as the removal of dependencies on external Groovy libraries argue for the direct translation of Junicon into Java. Such a migration to both Java and Groovy also provides insight into the stability of the transformations under retargeting. The transformations must address key differences of Groovy from Java, including in particular the use of typed declarations as well as differences in closure and lambda expression notation and behavior. While the discussion below focuses on transformation into Java, the translation into Groovy is a relaxation of a few constraints, such as needing to use *apply* for lambda expressions in method invocation.

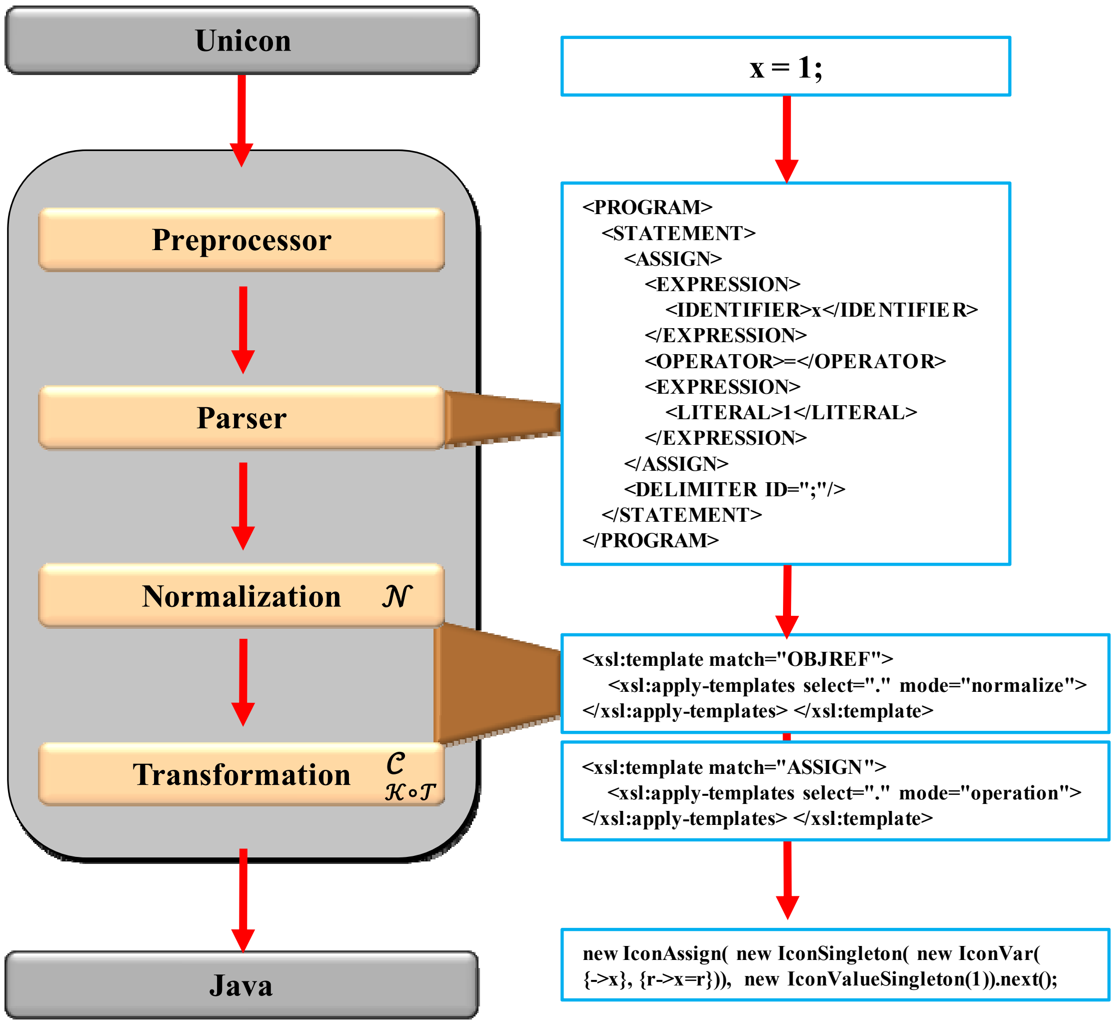
The differences can be addressed with only minor changes to the concretization transformations and class generation, and notably no changes to normalization. Translation to Java is enabled by simply aligning the syntax for closures to that of Java lambda expressions, and by slightly modifying the concretization transforms to emit types, e.g., *Object* instead of *def*. For classes, only a slight modification to method definition and invocation is required to expose methods as variadic lambda expressions. However, these modifications must carefully overcome limitations in forward references as well as subtle differences in the syntax for invoking lambda expressions. The above changes to the transforms are parameterized in the XSLT transforms of the implementation, so that the interpreter can generate either Java or Groovy code.

The transformation of Unicon into Java is broken down into four stages: a transformation $\mathcal{N}$ for flattening or normalization of primary expressions, a transformation $\mathcal{T}$ that translates larger expressions including control constructs and operations into an iterator calculus, a transformation $\mathcal{K}$ that concretizes the iterator calculus into Java, and finally a transformation $\mathcal{C}$ that handles classes and methods. In the following discussion we use the term normalization, which refers in general to reducing to a normal form that cannot be further rewritten, synonymously with flattening.

We make the normalization transform $\mathcal{N}$ independent of the later transforms for control constructs and classes, so as to enable staging them separately. Although it is feasible to equivalently define $\mathcal{N}$ to be recursively dependent on $\mathcal{T}$, the above independence of $\mathcal{N}$ is desirable, since normalization reflects aligning generator expressions with a more conventional semantics, and $\mathcal{N}$ can then be used as a standalone transformation to graft such a capability onto other languages. The latter is the approach taken in our implementation, and is reflected in the XSLT transforms as well as the staged structure of the generic transformational interpreter, described later.

Figure 8 illustrates the flow of transformation within the Junicon interpreter. While the transformations below are formalized as rewrite rules over program expressions, under the hood the transformations operate on XML abstract syntax trees that represent these expressions. Together with a front-end preprocessor and an XML-emitting parser, the transformations are thus used by the interpreter to effect embedding.

It bears emphasizing that the techniques described in this research are made possible by having either the transformation source or target be a dynamically typed language. In the presence of static typing and complex type systems, the problem of transformation is vastly more complex.


Unicon
Preprocessor
Parser
Normalization $\mathcal{N}$
Transformation $\mathcal{C}$ $\mathcal{K}\circ\mathcal{T}$
Java

```
x = 1;
```

```
<PROGRAM>
  <STATEMENT>
    <ASSIGN>
      <EXPRESSION>
        <IDENTIFIER>x</IDENTIFIER>
      </EXPRESSION>
      <OPERATOR>=</OPERATOR>
      <EXPRESSION>
        <LITERAL>1</LITERAL>
      </EXPRESSION>
    </ASSIGN>
    <DELIMITER ID=";"/>
  </STATEMENT>
</PROGRAM>
```

```
<xsl:template match="OBJREF">
   <xsl:apply-templates select="." mode="normalize">
</xsl:apply-templates> </xsl:template>
```

```
<xsl:template match="ASSIGN">
   <xsl:apply-templates select="." mode="operation">
</xsl:apply-templates> </xsl:template>
```

```
new IconAssign( new IconSingleton( new IconVar(
{->x}, {r->x=r})),  new IconValueSingleton(1)).next();
```

Figure 8.   Flow of transformation within the Junicon interpreter.

5.1 Normalization of primary expressions

The first step in the transformation that embeds Unicon into a conventional object-oriented language such as Java is the flattening or normalization of primary expressions. A key goal in the transformation of Unicon is to preserve type declarations and their use in function invocations and field references, so as to enable the use of native evaluation mechanisms and their concomitant optimization, as well as seamless interoperability. Seamless interoperability refers to the ability to transparently pass native types to and from Unicon, with fields accessed and methods invoked from either side. For example, we would want class definitions, variable declarations and simple method invocations such as *o.f*(*x*,*y*) to be left largely unchanged in migrating from Unicon to Java, and avoid elaborate reflection mechanisms or extensive instrumentation that might hinder interfacing with Java or that might preclude optimization. Following the above line of argument, more complicated expressions in Unicon that embody nested generator expressions must be reduced to the above simple form in a manner that makes iteration explicit.

To make iteration explicit, we introduce an operator for bound iteration, and decompose nested generators into products of such bound iterators. Consider the following example of a primary expression, which involves field reference and indexing in addition to function application, and where functions are allowed to be expressions that resolve to method references:

$e(e_x,e_y).c[e_i]$

This can be equivalently reformulated as:

$(f \text{ in } [\![e]\!]_{\mathcal{N}})$ & $(x \text{ in } [\![e_x]\!]_{\mathcal{N}})$ & $(y \text{ in } [\![e_y]\!]_{\mathcal{N}})$ & $(o \text{ in } !\ f(x,y))$ & $(i \text{ in } [\![e_i]\!]_{\mathcal{N}})$ & $(j \text{ in } !\ o.c[i])$

where $\mathcal{N}$ denotes the recursive application of the above transformation for flattening generators. In the above rewriting, for each step in the primary from left to right, generator expressions have been moved outside into bound iterators, and the pieces of the primary chained together using these bindings. The final result of the above expression will be a sequence whose values are bound to *j*. The ! operator denotes lifting, which reifies a term and promotes it to an iterator. Lifting a variable x turns it into a property with get and set methods, i.e., ()-> *x* and (*r*)-> *x*=*r*, and then wraps it in a singleton iterator, in order to enable it to be passed as an updatable reference. Lifting an invocation *f*(*x*) takes its closure and delegates iteration to the generator produced by its invocation. For plain Java methods, invocation just promotes the result to a singleton iterator. Lifting an index *c*[*i*] promotes it to a singleton iterator that returns an updatable reference.

The above reformulation, if applied recursively to a more complicated expression, extracts implicit generators and makes iteration explicit, reducing the expression to a normal form that is free of nested generators. The remaining residual expressions can then be evaluated using mechanisms native to the translation target. Normalization thus aligns Unicon with a more conventional semantics for list comprehension and method invocation, clarifying its meaning as well as placing it into a form more amenable to native evaluation.

5.1.1 Formalizing the transformations

The syntax for the Unicon expressions that are to be normalized is shown in Figure 9. We slightly extend Unicon syntax to incorporate several useful features such as lambda expressions and method references using *o*::*m*, as well as local declarations within blocks, and allocation using *new C*(*e*) in addition to Unicon's function-like construction using *C*(). The *new* construct as well as method references are provided to support accessing native Java classes and methods from within Junicon. Primary expressions, which consist of a chain of field references, invocation, and indexing, as well as identifiers and literals, are shown in the bottom of Figure 9. Primary expressions also include collection literals such as [*x*,*y*] and [*k*:*v*,...] for list and map construction, respectively. Since expressions may evaluate to method references, one may also chain method invocations and indexing together, for example, *f*(*x*)[*i*](*y*). It bears noting that previous semantic treatments [Gudeman 1992, Danvy et al. 2002] did not explicitly address function application using such method expressions, nor did they address propagating generators through the fields in object references. Thus our

E ::= Control | Block | Closure | Operation | Primary
Control ::= if E_1 then E_2 [else E_3]
| E_1 to E_2 [by E_3] | every E_1 [do E_2]
| while E_1 [do E_2] | until E_1 [do E_2]
| repeat E | not E
| suspend E_1 [do E_2] | return [E] | fail
Block ::= { E_1 ; ...; E_n }
| { local y_1[= E_y^1]; …; local y_m[= E_y^m]; E_1 ; ...; E_n}
Closure ::= (x_1,…,x_p) -> Block
Operation ::= E_1 & E_2
| E_1 | E_2
| E_1 op E_2
| op E where op in +, -, =, …
| new Dotname(E_1, …, E_n)
Primary ::= identifier | literal
| (E) | [E_1, ..., E_n]
| [E_k^1:E_v^1, ..., E_k^n:E_v^n]
| E(E_1, ..., E_n)
| E[E_1, ..., E_n]
| E.identifier
| Dotname::identifier
Dotname ::= identifier | Dotname.identifier

where x_i, y_i are identifiers, and E, E_i, E_i^j are expressions.

Figure 9. Syntax of expressions to be normalized.

formulation of normalization is a step forward in making clear the semantics of generator propagation.

The rewriting rules that reduce primary expressions to normal form are shown in Figure 10, and define the normalization transform $\mathcal{N}$. The rewriting rules are presented in the style of Felleisen-Hieb, which uses evaluation contexts to specify where the term rewriting rules apply [Wright and Felleisen 1994, Klein et al. 2011, Serbanuta et al. 2009]. A context is a term that contains a hole, denoted by ⟦⟧, within which rewrite rules can be applied. Contexts are specified by a separate grammar over expressions and values. For example, given a context of the form C ::= v(⟦⟧), a pattern of the form C⟦e⟧ matches any program fragment

Expressions and values

e ::= E extended with (o in E) & E' | !p

v ::= Name | Name.Dotname | Dotname::identifier

Name ::= identifier | <identifier> | literal | [v_1,...,v_n] *// Simple term*
  | [v_k^1:v_v^1, ..., v_k^n:v_v^n]

p ::= v | v(v_1,...,v_n) | v[v_1,...,v_n] | Closure *// Flat primary*

Evaluation contexts for complex primary expressions

C ::= ⟦⟧ | C.e | C(e ,e*) | C[e ,e*] | v.C | v(v,* C ,e*) | v[C]
  | [v,* C ,e*] | [v:v,* C:v ,e:e*] | [v:v,* v:C ,e:e*]

M ::= v[⟦⟧]

P ::= E extended with (o in E), where Primary ::= ⟦⟧

Reductions to flatten primary expressions

C⟦e⟧$_{\mathcal{N}}$ → (o in e) & C⟦<o>⟧$_{\mathcal{N}}$ if e is not simple v, and E != ⟦⟧

M⟦e, e' ,e*⟧$_{\mathcal{N}}$ → M⟦e⟧$_{\mathcal{N}}$ [e' ,e*]   *// Index* c[i, j]→c[i][j]

P⟦p⟧$_{\mathcal{N}}$ → P⟦!p⟧$_{\mathcal{N}}$ *// Lift primaries*

P⟦new e⟧$_{\mathcal{N}}$ → P⟦new(e)⟧$_{\mathcal{N}}$ *// Synthetic functions*

P⟦e to e' by e''⟧$_{\mathcal{N}}$ → P⟦to(e, e', e'')⟧$_{\mathcal{N}}$

P⟦(o in !v) & e⟧$_{\mathcal{N}}$ → P⟦e[v/<o>]⟧$_{\mathcal{N}}$ *// Optimization*

P⟦(o in ((o' in e') & e))⟧$_{\mathcal{N}}$ → P⟦(o' in e') & (o in e)⟧$_{\mathcal{N}}$

where o, x_i, y_i are identifiers, v_i are simple normalized primaries,
<*x*> = *x*.deref() is the dereference of a reified variable, and
!*p* denotes lifting that reifies *p* and promotes it to an iterator.

Figure 10. Reduction semantics for flattening.

v(*e*). Rewrite rules of form C⟦*e*⟧→C⟦*e'*⟧ then use these patterns to match any occurrence of a term *v*(*e*), split it into its context *v*(⟦⟧) and hole *e*, rewrite *e* into *e'*, and finally substitute that subterm back within the context. Evaluation contexts are powerful in part because they specify an evaluation order. For example, ⟦⟧(*e*) | *v*(⟦⟧) will ensure that function names are reduced before their arguments. While the above style of context-sensitive term rewriting is typically used to capture a full reduction semantics, here it succinctly expresses one stage of transformation.

Normalization thus flattens nested generators into products of bound iterators, and indicates where lifting must occur. In the rewriting rules, lifting is denoted by *!p*, which reifies a normalized term *p* and promotes it to an iterator, while *<x>* denotes the dereference of a reified variable. The normalization transform in Figure 10 also has rules that change certain constructs such as *to* that are generator functions into synthetic function invocations, so that their arguments can be normalized. Multidimensional indexes are also reduced to chains of single index steps. Lastly, the flattening transforms are optimized to skip the creation of bound iterators whenever possible, such as for simple terms in arguments. Thus for *f*(*x*,*e'*) where *x* is an identifier, the rewriting rules for normalization yield (*y* in *e'*) & !*f*(*x*, <*y*>).

For example, the *findPrimes* method in Figure 2 is normalized as follows:

```
def findPrimes (lower, upper) {
      local x_0;
      suspend (!to(lower, upper)) *
              (x_0 in !to(lower, upper)) & !this::isprime(x_0.deref());
      !null
}
```

As shown in the above example, normalization decomposes only primary expressions such as function invocation, and otherwise leaves operators and constructs alone. This is because operators and constructs are later implicitly translated to a map operation, under the equivalence:

```
map(*,I,J)  =  (x in I) & (y in J) & ! x*y
```

It bears noting that there is a direct correspondence of products of bound iterators to monad comprehension. Monads are a notion from category theory whose map-then-join pattern can be instantiated to capture a wide variety of ways to chain together computations

[Moggi 1991, Wadler 1990, Wadler 1992]. Using Haskell's *do* notation, where `\x->e` denotes lambda abstraction and `>>=` denotes the monad bind operator which is the adjunctive form of a functor's map followed by join [Hudak et al. 1999], we have the following equivalence:

```
(x in I) & J  =  do { (x <- I) ; J }  =  I >>= \x -> do{J}
```

If we further apply the above monadic equivalence to

```
(x in I) & (y in J) & !f(x,y)
```

where the lifting operator ! corresponds to the monad unit operation, we derive the $bind2_M$ operator of [Danvy et al. 2002].

5.1.2 A context-free formulation of normalization

The above formalization of normalization is in the form of a context-sensitive term rewriting system, which while succinct does not lend itself to simple implementation. In this section we present an equivalent context-free formulation of the normalization transform $\mathcal{N}$ that is more easily implemented.

The equivalent context-free rewriting rules to reduce primary expressions to normal form are shown in Figure 11. The normalization transform begins by, for a given program consisting of larger expressions such as control constructs and operators, descending into primary expressions through a default rule that recurses down through non-primaries otherwise leaving them unchanged. For a non-primary expression or construct c composed of terms t_i, the rule is a homomorphism over its terms:

$$[\![c\langle t_1,\ldots,t_n\rangle]\!]^p_{\mathcal{N}} \rightarrow c\langle [\![t_1]\!]^p_{\mathcal{N}},\ldots, [\![t_n]\!]^p_{\mathcal{N}}\rangle$$

We define $\mathcal{T}$ to analogously leave primaries alone, since they will have already been normalized.

$[\![e]\!]_{\mathcal{N}} \rightarrow [\![e]\!]^{\emptyset}_{\mathcal{N}}$ where e is any expression **// Entry into transforms using empty prefix**

$[\![c\langle t_1,\ldots,t_n\rangle]\!]^{p}_{\mathcal{N}} \rightarrow c\langle [\![t_1]\!]^{p}_{\mathcal{N}},\ldots,[\![t_n]\!]^{p}_{\mathcal{N}}\rangle$ where c is a non-primary with terms t_i **// Descend into primaries**

$[\![e_x \text{ to } e_y \text{ by } e_z]\!]^{p}_{\mathcal{N}} \rightarrow [\![\text{range}(e_x, e_y, e_z)]\!]^{p}_{\mathcal{N}}$ **// Synthetic functions for higher-order generators**

$[\![\text{new } C(e)]\!]^{p}_{\mathcal{N}} \rightarrow [\![C.\text{new}(e)]\!]^{p}_{\mathcal{N}}$

$[\![e.e']\!]^{p}_{\mathcal{N}} \rightarrow (o \text{ in } [\![e]\!]^{p}_{\mathcal{N}})\ \&\ [\![e']\!]^{p'}_{\mathcal{N}}$ where p'= <o>, the dereference of o **// Field reference**

$[\![e(e')e'']\!]^{p}_{\mathcal{N}} \rightarrow (o \text{ in } [\![e]\!]^{p}_{\mathcal{N}})\ \&\ [\![<o>(e')e'']\!]_{\mathcal{F}}$ where $e''=(e_1'')\ldots[e_i'']\ldots$ or $[e_1'']\ldots(e_i'')\ldots$ **// Invoke**

$[\![p(e_1,\ldots,e_n)e'']\!]_{\mathcal{F}} \rightarrow (x_1 \text{ in } [\![e_1]\!]^{\emptyset}_{\mathcal{N}})\ \&\ \ldots\ \&\ (x_n \text{ in } [\![e_n]\!]^{\emptyset}_{\mathcal{N}})\ \&\ (o \text{ in } !\ [\![p(<x_1>,\ldots,<x_n>)]\!]_{\mathcal{O}})\ \&\ [\![<o>e'']\!]_{\mathcal{F}}$

$[\![p(e_1\ldots,,\ldots e_n)]\!]_{\mathcal{O}} \rightarrow p(e_1\ldots, \text{omit}, \ldots e_n)$ where skip last product if $e''=\emptyset$

$[\![e[e']e'']\!]^{p}_{\mathcal{N}} \rightarrow (o \text{ in } [\![e]\!]^{p}_{\mathcal{N}})\ \&\ [\![<o>[e']e'']\!]_{\mathcal{F}}$ where $e''=(e_1'')\ldots[e_i'']\ldots$ or $[e_1'']\ldots(e_i'')\ldots$ **// Index**

$[\![p[e_1,\ldots,e_n]e'']\!]_{\mathcal{F}} \rightarrow [\![p[e_1]\ldots[e_n]e'']\!]_{\mathcal{F}}$

$[\![p[e']e'']\!]_{\mathcal{F}} \rightarrow (x \text{ in } [\![e']\!]^{\emptyset}_{\mathcal{N}})\ \&\ (o \text{ in } !\ p[<x>])\ \&\ [\![<o>e'']\!]_{\mathcal{F}}$ where skip last product if $e''=\emptyset$

$[\![x]\!]^{p}_{\mathcal{N}} \rightarrow !p.x$ where x is an identifier in the last field of an object reference **// Simple atom**

$[\![x]\!]^{\emptyset}_{\mathcal{N}} \rightarrow !x$ where x is an identifier or literal outside of a complex primary

$[\![e::f]\!]^{p}_{\mathcal{N}} \rightarrow [\![e]\!]^{p}_{\mathcal{N}}::f$ **// Method reference**

$[\![[e_1,\ldots,e_n]]\!]^{p}_{\mathcal{N}} \rightarrow (x_1 \text{ in } [\![e_1]\!]^{\emptyset}_{\mathcal{N}})\ \&\ \ldots\ \&\ (x_n \text{ in } [e_n]\!]^{\emptyset}_{\mathcal{N}})\ \&\ ![<x_1>,\ldots,<x_n>]$ **// List**

$[\![[e_k^1:e_v^1,\ldots,e_k^n:e_v^n]]\!]^{p}_{\mathcal{N}} \rightarrow (k_1 \text{ in } [\![e_k^1]\!]^{\emptyset}_{\mathcal{N}})\ \&\ (v_1 \text{ in } [\![e_v^1]\!]^{\emptyset}_{\mathcal{N}})\ \&\ \ldots\ \&$ **// Map**
$(k_n \text{ in } [\![e_k^n]\!]^{\emptyset}_{\mathcal{N}})\ \&\ (v_n \text{ in } [\![e_v^n]\!]^{\emptyset}_{\mathcal{N}})\ \&\ ![\![<k_1>:<v_1>,\ldots,<k_n>:<v_n>]$

where !*e* denotes lifting of a primary expression, which reifies *e* and promotes it to an iterator, and <*e*> denotes the dereference of a reified variable or value. Invocation delegates to the returned generator; otherwise lifting gives a singleton iterator that yields one result before failing.

We skip bound iterator creation (o in $[\![e]\!]^{p}_{\mathcal{N}}$) if *e* is a *simple term* consisting of an identifier or literal, or a field reference, method reference, or collection literal composed of only simple terms, and just use the prefixed original term $p'=p.e$ (or $p'=e$ if $p=\emptyset$) instead of <*o*> in the above products.

We skip bound variable creation (o in $[\![e]\!]^{p}_{\mathcal{N}}$) if $[\![e]\!]^{p}_{\mathcal{N}}$ is a product that ends in an iterator with binding *b*, or is itself such an iterator, and just use $[\![e]\!]^{p}_{\mathcal{N}}$ with $p'=<b>$ instead of <*o*> in the above products. For example, (*o* in (*b* in *e*)) becomes just (*b* in *e*).

We also skip bound variable creation (*o* in !*p*(<*x*>)) or (*o* in !*p*[<*x*>]) for the last invoke or index step in a primary, and just lift the last product term, since there is no further need for chaining. For example, x.f(y)[z] → (o in !x.f(y)) & !o[z].

Lastly, for efficiency we skip over redundant parenthesis, i.e., $[\![((e))]\!]^{p}_{\mathcal{N}} \rightarrow [\![(e)]\!]^{p}_{\mathcal{N}}$, and $[\![(e)]\!]^{p}_{\mathcal{N}} \rightarrow [\![e]\!]^{p}_{\mathcal{N}}$ if not inside a primary.

Figure 11. Normalization of primary expressions.

For a given primary expression, $\mathcal{N}$ then proceeds left to right along the fields and arguments inside it, decomposing field references and invocations into separate iterator product steps, and extracting complex fields and arguments into bound iterators. Along the way, $\mathcal{N}$ carries the accumulated object reference p, or prefix, to be used as the function or collection name in the decomposed invocation and indexing steps.

The above transforms must be similarly applied to all invocation and indexing arguments when there are multiple arguments, for example in

$$e(e_1,\ldots,e_n) \quad | \quad e[e_1,\ldots,e_n]$$

where multidimensional indexes are first reduced to chains of single index steps. Moreover, to optimize the number of bound iterators, we skip the creation of bound iterators (o in $[\![e]\!]^p_{\mathcal{N}}$) if e is a simple term consisting of an identifier or literal, or a field reference, method reference, or collection literal composed of only simple terms, and just use the prefixed original term $p.e$ instead of $<o>$ in the above products. We also skip the creation of bound iterators if $[\![e]\!]^p_{\mathcal{N}}$ ends in a created iterator, and use its last binding instead of $<o>$. These optimizations avoid synthesizing unnecessary bound iterators, and shorten the chain of iterator products. Since lifting only occurs when creating bound iterators, under the above optimizations only invoke and index, and simple terms such as identifiers, literals, and residual field references and method references that appear outside a primary field or argument, are lifted.

5.2 Composing suspendable iterators

Once normalization has unraveled nested generators to a conventional form, the task of implementing the flattened expressions still remains. A significant problem presented by Unicon lies in its wide variety of control constructs and operators, each of which must deal with the complexity of implementing suspendable generators.

A solution to the above problem is to define a minimal set of operators for composing suspendable iterators, called an iterator calculus, into which Unicon expressions and control constructs can be translated. The equational translation of constructs into that basis then provides a clear operational semantics for Unicon.

The next steps in the transformation that embeds goal-directed evaluation into Java are thus the translation of normalized expressions into the iterator calculus, and from there concretely into Java. In this section we first define an abstract iterator calculus, then show how control constructs are equationally translated into it, and finally concretely map the calculus into Java.

5.2.1 Definition of the iterator calculus

The iterator calculus distills the essential concepts of goal-directed evaluation, and provides a minimal set of operators for composing suspendable iterators. Suspendable iteration refers to iteration in which, in addition to *next*, there is a *suspend* operation. In a tree of composed iterators, *suspend* will return a value that is propagated up as the result of the root iterator's *next*. The following iteration of the root will then resume at the point of suspension. In the absence of composition, *suspend* is equivalent to *next*.

As can be seen from the preceding discussion, product, bound iteration, and lifting are the first entries in the calculus needed for normalization, and are sufficient to express lazily evaluated list comprehension. The full calculus for iterator composition is shown in Figure 12. It bears noting that *map* and *forall* are included as convenience mechanisms, and can be equivalently expressed using *product* and *reduce*, respectively.

A single Java class, *IconIterator*, implements the stream-like interface for the iterator calculus in a tightly knitted logic that provides iteration that is suspendable, failure-driven,

$I ::= I_1$ & I_2	Product: for *i* in I_1 { for *j* in I_2 }
$\mid I_1 \mid I_2$	Concatenation: {for *i* in I_1}; {for *j* in I_2}
$\mid I_g \text{->} I_1 \mid I_2$	Choice: if exists(I_g) iterate over I_1, else over I_2
$\mid$ I*	Repeat until produces an empty sequence
$\mid$ I:n	Limit iterator to at most n results
$\mid$! p	Lift normalized primary expression
$\mid$ *x* in I	Bound iteration
$\mid$ reduce *op* I	Combine results from I until it fails
$\mid$ map *op* I_1 I_2	Map operator over product of its operands. = (*x* in I_1) & (*y* in I_2) & (! <*x*> *op* <*y*>)
$\mid$ forall I	Reduce by iterating over I until failure
$\mid$ exists I	Succeed if non-empty sequence
$\mid$ not I	Succeed on failure, and fail on success
$\mid$ suspend I	Suspend, yielding value from I
$\mid$ return I	Return exists(I)
$\mid$ fail	Constant iterator that always fails

// Normalized primary
p ::= v $\mid$ v(v_1,...,v_n) $\mid$ v[v_1,...,v_n] $\mid$ closure
closure ::= (x_1,…,x_p) -> { local y_1;…; local y_m; I }

where x_i, y_i are identifiers, v_i are atoms, and
<*x*> = *x*.deref() is the dereference of a reified variable.

Figure 12. Syntax of the iterator calculus.

and optionally reversible. While the *IconIterator* class implements the *java.util.Iterator* interface, it differs in that *hasNext*() tests for failure of *next*(), which terminates the iterator. After failure, the iterator is then restarted on the following *next*(). The kernel is optimized to statefully resume its point of suspension on a succeeding *next*(), incurring zero cost for suspends. Unlike other language extensions that implement *suspend* in iterators using multithreading, such as can be found for Groovy and Java, in Junicon suspend is tightly integrated into the kernel.

Subtypes of the *IconIterator* class built using the stream-like operations of the iterator calculus are then used as abbreviations for constructs such as *while*, and serve as the final target of the transformation rules. The transformation that concretely takes the iterator calculus into the Java kernel is denoted by $\mathcal{K}$. For example, *I* &*J* is translated as follows:

$[\![\text{I \& J}]\!]_{\mathcal{K}} \rightarrow$ new IconProduct($[\![\text{I}]\!]_{\mathcal{K}}$, $[\![\text{J}]\!]_{\mathcal{K}}$)

For operations such as *I* +*J*, instead of normalizing the expression into an iterator product

(i in I) & (j in J) & (k in ! <i>+<j>)

which would work, we instead build a map operation into the class performing the iterator product, *IconIterator*, so that it performs the operation at each product pair if the operands succeed. Thus, the operation *I* +*J* is translated to:

$[\![\text{I + J}]\!]_{\mathcal{T}} \rightarrow$ new IconProduct($[\![\text{I}]\!]_{\mathcal{K}}$, $[\![\text{J}]\!]_{\mathcal{K}}$).map(new IconOperator((x,y) -> x+y))

although for efficiency we only define operators once.

The iterator calculus is loosely derived from a combination of functional forms [Backus 1978], guarded commands such as in GCL [Dijkstra 1975] and CSP [Hoare 1978], and first-order formulae as found in Z schemata [Abrial et al. 1980, Spivey 1992, Davies and Woodcock 1996] and SETL [Dewar et al. 1981, Schwartz et al. 1986]. Unlike typical mechanisms for list comprehension, the iterator calculus allows specifying comprehensions, i.e., the intensional properties defining a sequence, using the more powerful first-order formulae.

The iterator calculus in many regards bears a striking similarity to the Java *Stream* interface. It differs significantly, however, in its support for suspendable iteration, as well as its provision of forms such as *repeat* and *product* that depend on the ability to be restarted.

5.2.2 Translation of expressions into the iterator calculus

The second step in the transformation of Unicon into Java is the translation of normalized expressions into the iterator calculus. The key goal here is to provide an equational translation of constructs that clarifies their semantics, is independent of the concrete translation target, and reduces the complexity of implementation.

The iterator calculus provides the basis, that is, minimal spanning set, into which control constructs and operations can be translated. Some of the operations in the iterator calculus, with the notable exception of *lift*, *reduce*, and *map*, are also primitives in Unicon. Other Unicon constructs are transformed into compositions of calculus operators.

The rules for the transformation $\mathcal{T}$ that translates program expressions into the iterator calculus are shown in Figure 13. As can be seen in the figure, the rules provide an equational definition of the semantics of Unicon control constructs. The equations are defined in terms of the calculus's stream-like interface for composing suspendable iterators using functional forms such as *product*, *concatenation*, and *reduce*. After normalization, the transformation of expressions thus proceeds by using these equations to replace constructs and operators with their equivalent iterator calculus formulae. As described later, instead of using the replacement formulae directly, such formulae are defined once as subtypes of the *IconIterator* class, and then used as abbreviations for the translation of the construct.

For example, the *every* construct corresponds to *forall* or *reduce*, and iterates until failure. The sequence construct, where terms are separated by ";", is equivalently transformed into the concatenation of bound expressions, i.e., a singleton iterator with limit 1, with the result being an iterator over the last unbounded term. Operations are simply transformed into a map over products of the operands. Lambda expressions, on the other

Sequence, block, and lambda expression

$[\![\{E_1; \ldots; E_n; E_z\}]\!]_{\mathcal{T}} \rightarrow$ forall($[\![E_1]\!]_{\mathcal{T}}$:1 | … | $[\![E_n]\!]_{\mathcal{T}}$:1) | $[\![E_z]\!]_{\mathcal{T}}$

$[\![\{$ local $y_1 = E_y^1; \ldots;$ local $y_m = E_y^m; \; E_1;\ldots;E_n \}]\!]_{\mathcal{T}} \rightarrow$

{ () -> local $y_1;\ldots;$ local y_m; $[\![\{y_1 = E_y^1; \ldots; y_m = E_y^m; \; E_1;\ldots;E_n\}]\!]_{\mathcal{T}}$ } ( )

$[\![(x_1,\ldots,x_p)$ -> { local $y_1 = E_y^1; \ldots;$ local $y_m = E_y^m; \; E_1;\ldots;E_n \}]\!]_{\mathcal{T}} \rightarrow$

! $(x_1,\ldots,x_p)$ -> { local $y_1;\ldots;$ local y_m; $[\![\{y_1 = E_y^1; \ldots; y_m = E_y^m; \; E_1;\ldots;E_n;$ fail$\}]\!]_{\mathcal{T}}$ }

Control constructs

$[\![$every E$]\!]_{\mathcal{T}} \rightarrow$ forall($[\![E]\!]_{\mathcal{T}}$)

$[\![$every E_x do $E_y]\!]_{\mathcal{T}} \rightarrow$ forall($[\![E_x]\!]_{\mathcal{T}}$ & ($[\![E_y]\!]_{\mathcal{T}}$;fail))

$[\![$while E_x do $E_y]\!]_{\mathcal{T}} \rightarrow$ forall(($[\![E_x]\!]_{\mathcal{T}}$:1 -> ($[\![E_y]\!]_{\mathcal{T}}$; not(fail):1))*)

$[\![$until E_x do $E_y]\!]_{\mathcal{T}} \rightarrow [\![$while (not E_x) do $E_y]\!]_{\mathcal{T}}$

$[\![$repeat E$]\!]_{\mathcal{T}} \rightarrow$ forall(($[\![E]\!]_{\mathcal{T}}$; not(fail):1)*)

$[\![$if E_g then E_x else $E_y]\!]_{\mathcal{T}} \rightarrow [\![E_g]\!]_{\mathcal{T}}$ -> $[\![E_x]\!]_{\mathcal{T}}$ | $[\![E_y]\!]_{\mathcal{T}}$

$[\![$if E_g then $E_x]\!]_{\mathcal{T}} \rightarrow [\![E_g]\!]_{\mathcal{T}}$ -> $[\![E_x]\!]_{\mathcal{T}}$

// *Below* $I_1;I_2$ *means* forall(I_1:1) | I_2

$[\![$not E$]\!]_{\mathcal{T}} \rightarrow$ not(exists($[\![E]\!]_{\mathcal{T}}$))

$[\![$ fail $]\!]_{\mathcal{T}} \rightarrow$ fail

$[\![$return E$]\!]_{\mathcal{T}} \rightarrow$ return(exists($[\![E]\!]_{\mathcal{T}}$))

$[\![$suspend E$]\!]_{\mathcal{T}} \rightarrow$ suspend($[\![E]\!]_{\mathcal{T}}$)

$[\![$suspend E_x do $E_y]\!]_{\mathcal{T}} \rightarrow$ (x in $[\![E_x]\!]_{\mathcal{T}}$) & (suspend(!x)); $[\![E_y]\!]_{\mathcal{T}}$;fail)

Iterator calculus operators

$[\![E_x$ & $E_y]\!]_{\mathcal{T}} \rightarrow [\![E_x]\!]_{\mathcal{T}}$ & $[\![E_y]\!]_{\mathcal{T}}$

$[\![E_x \mid E_y]\!]_{\mathcal{T}} \rightarrow [\![E_x]\!]_{\mathcal{T}} \mid [\![E_y]\!]_{\mathcal{T}}$

$[\![$ (x in E) $]\!]_{\mathcal{T}} \rightarrow$ (x in $[\![E]\!]_{\mathcal{T}}$)

Operations, e.g. +

$[\![$Ex op $E_y]\!]_{\mathcal{T}} \rightarrow$ map (op, $[\![E_x]\!]_{\mathcal{T}}$, $[\![E_y]\!]_{\mathcal{T}}$)

$[\![$op E$]\!]_{\mathcal{T}} \rightarrow$ map (op, $[\![E_x]\!]_{\mathcal{T}}$)

Default transforms

$[\![!E]\!]_{\mathcal{T}} \rightarrow ![\![E]\!]_{\mathcal{T}}$

$[\![p]\!]_{\mathcal{T}} \rightarrow p$

$[\![c\langle t_1,\ldots,t_n\rangle]\!]_{\mathcal{T}} \rightarrow c\langle [\![t_1]\!]_{\mathcal{T}}, \ldots, [\![t_n]\!]_{\mathcal{T}}\rangle$ // *Descend into* c *composed of terms* t_i

Figure 13. Translation of expressions into the iterator calculus.

hand, must move any initializers for local declarations into the function body before recursively transforming that part of the expression. Blocks with local declarations similarly shift initializers into the sequence body, and are mapped into closures which thus bind the returned generator to the local declarations.

The kernel that implements the iterator calculus also provides a number of built-in methods that optimize several of the most frequently occurring calculus expressions. These include *forall* that is equivalent to *reduce* with a don't-care operator. We also abbreviate *forall*(*x* :1) as *x.bound*(), which represents what Icon calls a bounded expression, that is, a singleton iterator that runs to failure, here optimized as an iterator that always fails but also remembers if it was non-empty. We further optimize the kernel by incorporating direct support for such frequently occurring patterns as *succeed*:*1*, where *succeed* is *not*(*fail*), as well as *exists*, which is implemented as always restarting the iterator.

5.2.3 Concretization of the iterator calculus

A final step in the translation of expressions is to take the normalized expressions, as well as the constructs which have been reduced into the iterator calculus, concretely into Java. A key problem to be solved is to correctly handle lifted terms as well as object reference and indexing over reified terms.

As mentioned previously, the normalization transform $\mathcal{N}$ for primary expressions is independent of the transform $\mathcal{T}$ for larger expressions, so as to enable staging them separately. The net transform for taking Unicon expressions into the iterator calculus is thus the composition:

$$\mathcal{T} \circ \mathcal{N}$$

and the overall transformation that takes Unicon expressions into Java is the composition:

$$\mathcal{K} \circ \mathcal{T} \circ \mathcal{N}$$

where $\mathcal{K}$ concretizes the calculus into Java. Figure 14 shows the transforms in $\mathcal{K}$ that take normalized terms in the calculus into the Java kernel that implements it.

Normalized primary expressions such as invocation and indexing are translated into small classes that extend the kernel's *IconIterator* to perform generator delegation and mutable indexing, respectively. Other primaries such as for variable and field reference must handle whether they are reified properties or just plain terms such as for arguments or subscripts. For larger Unicon constructs, such as *while*, the calculus equations are directly

Lift primary to iterator

⟦! f(e_1,…,e_n)⟧$_{\mathcal{K}}$ → new IconInvokeIterator(( ) -> ((VariadicFunction) ⟦f⟧$_{\mathcal{K}}$).apply (⟦e_1⟧$_{\mathcal{K}}$,…, ⟦e_n⟧$_{\mathcal{K}}$)})

⟦! c::f(e_1,…,e_n)⟧$_{\mathcal{K}}$ → new IconInvokeIterator(( ) -> ⟦c⟧$_{\mathcal{K}}$.f (⟦e_1⟧$_{\mathcal{K}}$,…, ⟦e_n⟧$_{\mathcal{K}}$))

⟦! c[e] ⟧$_{\mathcal{K}}$ → new IconIndexSingleton(⟦c⟧$_{\mathcal{R}}$, {-> ⟦e⟧$_{\mathcal{K}}$ })

⟦! c.new(e_1,…,e_n)⟧$_{\mathcal{K}}$ → new IconInvokeIterator({()-> new c(⟦e_1⟧$_{\mathcal{K}}$,…, ⟦e_n⟧$_{\mathcal{K}}$)})

⟦! to(e_1, e_2, e_3)⟧$_{\mathcal{K}}$ → new IconToIterator(⟦e_1⟧$_{\mathcal{K}}$, ⟦ e_2⟧$_{\mathcal{K}}$, ⟦ e_3⟧$_{\mathcal{K}}$)

⟦! l⟧$_{\mathcal{K}}$ → new IconValueSingleton(l)

⟦! p⟧$_{\mathcal{K}}$ → new IconSingleton(⟦p⟧$_{\mathcal{R}}$)

Reified primary, with getter and setter

⟦x⟧$_{\mathcal{R}}$ → x_r, if external: new IconVar(()->x, (rhs)->x=rhs)

⟦<t>⟧$_{\mathcal{R}}$ → t_r, if inside closure: t_r.get()

⟦o.x_1. … .x_n ⟧$_{\mathcal{R}}$ → new IconField(⟦o⟧$_{\mathcal{R}}$, "x_1", …, "x_n")

⟦p⟧$_{\mathcal{R}}$ → {()-> ⟦p⟧$_{\mathcal{K}}$}, if inside closure: ⟦p⟧$_{\mathcal{K}}$

Inside primary, i.e., is function argument, subscript, or field in object reference

⟦x⟧$_{\mathcal{K}}$ → x_r.deref(), if external or class field: x ⟦c::f ⟧$_{\mathcal{K}}$ → ⟦c⟧$_{\mathcal{K}}$::f

⟦<t>⟧$_{\mathcal{K}}$ → t_r.deref(), where deref()=get().get() ⟦ l ⟧$_{\mathcal{K}}$ → l

⟦o.x_1. … .x_n ⟧$_{\mathcal{K}}$ → IconField.getFieldValue(⟦o⟧$_{\mathcal{R}}$, "x_1", …, "x_n") *or* ⟦o⟧$_{\mathcal{K}}$ .x_1. … .x_n *if* o *is typed*

⟦ [e_1,...,e_n] ⟧$_{\mathcal{K}}$ → new IconList(⟦e_1⟧$_{\mathcal{K}}$,..., ⟦e_n⟧$_{\mathcal{K}}$) ⟦n⟧$_{\mathcal{K}}$ → new IconNumber(n)

⟦ [e_k^1:e_v^1,...,e_k^n:e_v^n] ⟧$_{\mathcal{K}}$ → new IconMap(⟦e_k^1⟧$_{\mathcal{K}}$,⟦e_v^1⟧$_{\mathcal{K}}$,..., ⟦e_k^n⟧$_{\mathcal{K}}$,⟦e_v^n⟧$_{\mathcal{K}}$)

Calculus operations and control constructs

⟦(t in I)⟧$_{\mathcal{K}}$ → new IconIn(⟦t⟧$_{\mathcal{R}}$, ⟦I⟧$_{\mathcal{K}}$) ⟦if I_1 then I_2⟧$_{\mathcal{K}}$ → new IconIf(⟦I_1⟧$_{\mathcal{K}}$, ⟦I_2⟧$_{\mathcal{K}}$)

⟦I_1 & I_2⟧$_{\mathcal{K}}$ → new IconProduct(⟦I_1⟧$_{\mathcal{K}}$, ⟦I_2⟧$_{\mathcal{K}}$) ⟦fail⟧$_{\mathcal{K}}$ → new IconFail()

⟦p⟧$_{\mathcal{K}}$ → p // *Default transform*

where *x* and *y* are identifiers, *t* is a temporary identifier, *p* is a normalized primary, *f*, *c*, and *e* are simple normalized primaries, *o* is a simple identifier or literal or < *t* >, *l* is a literal, *n* is a number, and
interface VariadicFunction <T,R> { R apply (T... args); }
An identifier is an external reference if it is not a class field, method local, parameter, or temporary.

Figure 14. Concretization of iterator calculus terms.

embodied in small classes, e.g., *IconWhile*, that also extend the kernel's *IconIterator*. Concretization is optimized to avoid redundant nested closure formation and to recognize final variables that do not need to be inside closures at all.

In interactive mode, the transforms slightly alter their behavior to enable execution of outermost expressions. As with class field initializers, a simple expression such as:

```
f(x)
```

is transformed to:

$[\![f(x)]\!]_{\mathcal{K}\circ\mathcal{T}\circ\mathcal{N}}.next()$

which executes the first iteration of the generator.

There are several other ancillary but simple transformations needed to complete the concretization, for example changing collection literals and numeric literals to method invocations for list formation and arbitrary precision promotion, respectively. Method invocation must also accommodate the use of method references, further described in the following section.

5.3 Transformation of classes

When embedding at the class level, a last stage in transformation takes Unicon methods as well as class fields into the Java class model. A key problem to be solved is how to support interoperability while still accommodating Unicon's reference semantics. As mentioned previously, in Unicon, variables and subscripted collections can be passed as updatable references, and function names can be used in expressions. At the same time, Unicon class fields and methods need to be exposed in a manner that can be passed to and used by native Java methods, and conversely easily access foreign class fields and methods from within Unicon.

Our approach to solve the above problem is to expose variables in both plain and reified form, while maintaining consistency between them. Methods are similarly exposed as method references in addition to their plain form. This duality allows Java code to use the plain form, while embedded Unicon code can use the reified form. Lastly, Junicon is carefully designed to preserve native types, i.e., maps, lists, and sets are just their equivalent

Java types, so any Java methods can be used on them. Arbitrary precision arithmetic for integers, which is the default in Unicon, similarly promotes numeric literals to *BigIntegers* as needed.

Below we describe the techniques used to transform class fields and methods, as well as procedures, into Java in a manner that promotes interoperability. We denote the transformations for classes and methods, as well as procedures, by $\mathcal{C}$. Since normalization is a separately staged transformation, and since $\mathcal{T}$ and $\mathcal{K}$ are bundled into $\mathcal{C}$, the net transform that takes programs into Java is thus:

$$\mathcal{C} \circ \mathcal{N}$$

As is further described in Chapter 6, the transformations are implemented in two phases of XSLT transformation: a normalization stage $\mathcal{N}$, followed by $\mathcal{C}$ for the transformation of classes, methods, and expressions.

Unicon methods have a few differences from Java that must be accommodated in translation. Overall Unicon classes are roughly similar to those of Java, in that they are composed of fields and methods, albeit with relaxed typing and with the exception that multiple inheritance is allowed. However, each method can take any number of arguments, i.e., it is variadic. Method arguments may also be omitted in invocation even if they are interior, e.g., $f(x,,y)$, in which case they are null or resolve to parameter defaults, as well as superfluously supplied, in which case they are ignored. Each method thus has arbitrary arity, which precludes method overloading, and as a consequence each method has a unique name within its class. Unicon also has a scoping model in which variables that are not declared and are unresolved at link time are made local to a method or procedure. Unicon thus treats unresolved references as method locals. While the transforms faithfully preserve other features in Unicon, the linking model in its deferred scoping of locals is an exception, due to its incompatibility with Java.

In contrast to Unicon, Junicon follows the Java package model and its scoping rules, so that all variables must be imported or declared. Multiple inheritance is not allowed: instead, interfaces, while not directly expressible in Junicon, can be used by embedding into a Java class that implements them. Junicon also extends Unicon syntax to allow a familiar Java-like block notation for classes and methods. While either Unicon-style or Java-style notation is allowed as input to the interpreter, a preprocessing step aligns any Unicon-style notation with the Java-like syntax before transformation. For simplicity, the examples and transformation rules that follow use the Java-style block notation that exists after preprocessing.

5.3.1 Exposing fields in both plain and reified form

In order to enable interoperability under embedding, field declarations as well as method parameters are preserved as plain variables, and then also exposed in reified form. For example, a field declaration in Junicon is transformed as follows:

```
⟦ local x; ⟧𝒸 →
                Object x;       // Exposed field, with reified form below
                IconVar x_r = new IconVar(()->x, (rhs)->x=rhs);
```

The IconVar implementation maintains consistency between the reified and plain forms, so that Java methods can use the exposed class fields, while internally operations can use the reified variable. Moreover, any explicitly typed variables are carefully carried forward into derived reified variables and method parameters, so as to enable field reference without reflection when possible, as well as their use in natively typed Java methods. The actual types stored in variables are carefully chosen to be normal Java types, specifically those used by Groovy, for example *ArrayList*, *LinkedHashMap*, and *LinkedHashSet*, as well as numeric types such as *BigInteger* to support arbitrary precision arithmetic, which is the default in Unicon.

5.3.2 Exposing methods as variadic lambda expressions

The transformations must also accommodate the use of function names in expressions. As mentioned, in Unicon methods are variadic, i.e., they take any number of arguments. A naive approach to accommodate the use of function names in expressions would be to simply take Unicon methods into fields bound to variadic lambda expressions that return iterators. Under translation to Groovy, the above approach works flawlessly. However, under translation to Java, two problems arise. First, forward references are not allowed inside lambda expressions, nor to fields holding them. To handle this restriction, methods are defined as plain methods, and then also exposed as method references with the same name. Method definitions are thus transformed as follows:

$[\![$ method M(x,y) { body } $]\!]_{\mathcal{C}} \rightarrow$
Object M = (VariadicFunction) this::M;
Object M (Object... args) { $[\![$ body $]\!]_{\mathcal{M}}$ }

where $\mathcal{M}$ denotes the translation of the method body.

Second, invocation must then accommodate the use of such method references for function names. Again, under Groovy, this is not a problem, since invocation of a closure transparently uses the same notation as if it were a method, e.g., *f*(*args*). However, as mentioned previously, in Java the use of such method references, which are treated as lambda expressions, must be made explicit. This is because invocation using lambda expressions does not use the same syntax as for method invocation, as it might otherwise if function types had been introduced in Java. Rather, in Java, a lambda expression resolves to an instance that implements an interface with a single method, called a functional interface. Invocation using lambda expressions, or variables that hold lambda expressions, must thus use a field name, for example *f.apply*(*args*), instead of *f*(*args*). Since Unicon methods are exposed as variadic lambda expressions, invocation when translated to Java becomes:

$$[\![\text{M(x)}]\!]_{\mathcal{K}} \rightarrow \text{((VariadicFunction) M).apply(x)}$$

However, the artificially injected method name in turn poses a problem when invoking native Java methods. The problem that must be addressed is how to differentiate the invocation of native Java methods from other Unicon methods, given that the latter invocation is assumed to use a functional interface. In the general case the function name used in an invocation may refer either to a Unicon method, or to a Java method in external code. In the former case the name resolves to a lambda expression, while in the latter case the name resolves to a plain method. One solution to differentiate the invocation of lambda expressions from that of plain methods is to make explicit the use of non-Unicon Java methods.

We thus explicitly distinguish the invocation of plain Java methods from Unicon methods using ::, which is translated through the rewriting rule:

$$[\![\text{o::M(x)}]\!]_{\mathcal{K}} \rightarrow \text{o.M(x)}$$

When embedding in Groovy, however, since method references are treated as function types, the transforms bypass the use of functional interfaces, which provides full transparency when calling Java methods from within Unicon.

The above technique of defining methods in both plain and reference form also supports interoperability when calling Unicon methods from within Java. Because the plainly defined methods are variadic over plain arguments and return a Java Iterator, they can be used from within Java, while internally expressions can use the method references. Thus bidirectional interoperability, analogous to that for fields, is also provided for methods.

The alternative to making Java invocation explicit is to scope up through class definitions to resolve whether external references are to Java or Unicon, since they treat method invocation differently. While such a technique is feasible, our strategy has been to purposefully avoid such resolution, since it replicates many details of the Java compiler for handling types and mutual dependencies. Such added complex resolution techniques must also be maintained to track the evolving Java type system, and so the risk outweighs the minor benefits.

5.3.3 Exposing constructors as method references

Another key problem in the translation of Unicon to Java is how to map Unicon procedures into the Java class model. In particular, procedures, which are akin to methods bound to a global variable, are updatable entities. In addition we need to support Unicon's provision of function-like class instantiation using *C*(*x*) instead of *new C*(*x*), a style similar to that later adopted by Python. A solution to the above problems is to map procedures as well as constructors into static fields with the same name as the class. These fields are then made visible in scope using *import static C.C*. Support for function-like class instantiation, e.g. *C*(*x*) instead of *new C*(*x*), is realized using static fields that wrap the original constructor:

public static VariadicFunction C = $[\![$(x) -> {new C(x)}$]\!]_{\mathcal{K}}$

Procedures, such as

```
procedure P(x) {body}
```

are similarly transformed to:

public static Object $[\![\text{P(x) \{body\}}]\!]_{\mathcal{M}}$

where $\mathcal{M}$ transforms a method into Java.

5.3.4 Formalizing the transformations for classes

Figure 15 illustrates the above transformations for classes and methods as well as procedures. The class declaration proper translates fairly directly to a Java class. Field declarations within classes are preserved as plain fields, and their reification separately encoded, so as to enable integration with Java. The treatment of methods as fields bound to lambda expressions similarly enables their use as references in generator expressions. However, field initializers, having been transformed to generators, must be unraveled after transformation using *next*() so as to yield a value for assignment.

The bottom of Figure 15 illustrates the concretization $\mathcal{K}$ of methods and lambda expressions into variadic functions that return iterators. Recall that method invocations were normalized to iteration over a returned generator as follows:

```
e(e',e") -> (f in e) & (x in e') & (y in e") & (o in !f(x,y))
```

Method definitions must thus return an iterator, and so are transformed to a generator function via $\mathcal{M}$:

$[\![\text{ M(x,y) \{ local z=e; body \}}]\!]_{\mathcal{M}} \rightarrow [\![\text{ M(x,y) \{ local z; } [\![\text{\{z=e; body; fail\}}]\!]_{\mathcal{T}} \text{ \}}]\!]_{\mathcal{K}}$

Both methods and lambda expressions are then made variadic by $\mathcal{K}$ in a manner that supports argument omission and parameter defaults. Method locals are ensured to be effectively final by encapsulating them as a reified variable that holds its own value. As with block declarations, local variable initializers must be incorporated into the sequence body before

Procedures

```
⟦procedure P(x) {              →   class P {
     local z=e; body                    public static Object P = (VariadicFunction)
}⟧$_{\mathcal{C}}$                                           (Object... args) -> P(args);      // Method reference
                                        public static Object ⟦P (x) { local z=e; body }⟧$_{\mathcal{M}}$   // Plain form
                                   }  import static P.P;
```

Classes

```
⟦class C:E (field) {           →   class C extends E {
     local i=e;                         public Object M = (VariadicFunction) this::M;  // Method
     method M(x) { local z=e; body }    public Object  field;                          // reference
}⟧$_{\mathcal{C}}$                                     IconVar  field_r = new IconVar(()->field, (rhs)->{field=rhs});
                                        public C() { super();  initially() }           // Constructors
                                        public C(field) { super();  this.field=field;  initially() }
                                        public static VariadicFunction  C = ⟦(x) -> {new C(x)}⟧$_{\mathcal{K}}$
                                        public Object  i=⟦e⟧$_{\mathcal{K}\circ\mathcal{T}}$.next();           // Reified fields
                                        IconVar  i_r = new IconVar(()->i, (rhs)->{i=rhs});
                                        public Object  ⟦M (x) { local z=e; body }⟧$_{\mathcal{M}}$   // Plain methods
                                   }  import static C.C;
```

Methods and lambda expressions

```
⟦ (x,y) -> { local z=e; body}⟧$_{\mathcal{L}}$    →   ⟦ (x,y) -> { local z;  ⟦{z=e;  body;  fail}⟧$_{\mathcal{T}}$ }⟧$_{\mathcal{K}}$    // Lambdas
⟦ (x, y=d, o[]) -> {local z; body$_{\mathcal{T}}$}⟧$_{\mathcal{K}}$   →   (Object...  args) -> {                     // Concretization
                    IconVar  x_r = new IconVar();   IconVar  y_r = new IconVar();   // Parameters
                    IconVar  o_r = new IconVar();   IconVar  z_r = new IconVar();   // Locals
                    if (args == null) { args = omit.getEmptyArray(); };               // Unpack parameters
                    x_r.set((args.length > 1) ? args[1] : null);                       // Can omit argument
                    y_r.set (((args.length > 2) && (args[2] != omit)) ? args[2] : d);   // Has default
                    o_r.set((args.length > 3) ? Arrays.asList(args).subList(3,args.length) :new ArrayList());
                    return ⟦body$_{\mathcal{T}}$⟧$_{\mathcal{K}}$;     }
⟦ M(x,y) { local z=e; body }⟧$_{\mathcal{M}}$   →   ⟦ M(x,y) { local z;  ⟦{z=e;  body;  fail}⟧$_{\mathcal{T}}$ }⟧$_{\mathcal{K}}$    // Methods
⟦ M(x, y=d, o[])  {local z; body$_{\mathcal{T}}$}⟧$_{\mathcal{K}}$   → M(Object...  args) {Same as lambda concretization}
```

Figure 15. Transformation of methods using variadic lambda expressions.

further transformation. Lambda expressions as well as block declarations must also synthesize local declarations for temporaries used in bound iterators.

Figures 16 and 17 show the results of preprocessing, normalization, and transformation for an example program. The program in Figure 16 generates the values between 1 and an upper bound, and interfaces with the Java *println* method to effect printing. In Figure 16, the original Unicon program is shown on the top left, and the program after preprocessing and

```
class C(lower)          Original program
  method printRange (upto)
    local i
    every (i = C().range(lower,upto)) do
        System.out::println(i)
  end
  method range (from,bound) … end
end

class C(lower) {        After preprocessing
 method printRange (upto) {
     local i;
     every (i = C().range(lower,upto)) do
        System.out::println(i);
 }
 method range (from,bound) { … }
}
```

```
class C(lower) {        Normalized program
  method printRange (upto) {
    local x_0;
    local i;
    every (!i = (x_0 in !C()) &
          !x_0.deref().range(lower, upto)) do
       !System.out::println(i);
    !null
  }
  method range (from,bound) { ... }
}
```

Figure 16. Normalization of a sample program.

normalization are shown in the middle and bottom, respectively. Preprocessing, in addition to handling directives for conditional compilation and source inclusion, also inserts semicolons and braces to align programs with a Java-style block notation that simplifies the recognition of parseable statements. Normalization, as shown in the bottom of Figure 16, then flattens nested generators into products of bound iterators, and indicates where lifting must occur.

The Java result after the transformation $\mathcal{C}$ is then shown in Figure 17. The transformation of classes can be seen to expose methods as method references assigned to a class field with the original method name. After translation to Java, the function body itself is an iterator constructor, so that the function when invoked will return an iterator. For optimization the iterator body is cached in a stack upon method return, and then reused. Since Unicon methods are variadic, the signature of the exposed method, shown at the top of Figure 17, is a variadic lambda expression that returns an iterator.

```
public class C {
 MethodBodyCache methodCache=new MethodBodyCache();
 // Method references
 public Object printRange=(VariadicFunction) this::printRange;
 public Object range = (VariadicFunction) this::range;
 // Constructor fields
 public Object lower;
 IconVar lower_r = new IconVar(()-> lower, (r)-> lower=r);
 // Constructors
 public C() { ; }
 public C(Object lower) { this.lower = lower; }
 public static VariadicFunction C= (Object... args) -> {
      if (args ==  null) { args  = IconEnum.getEmptyArray();};
      return new C((args.length > 0) ? args[0] : null); };
 // Methods
 public Object printRange (Object... args) {
   // Reuse method body
   IconIterator body =methodCache.getFree("printRange");
   if (body != null) { return body.reset().unpackArgs(args); };
   // Parameters, temporaries, and locals
   IconVar upto_r = new IconVar().local();
   IconTmp x_0_r = new IconTmp();
   IconVar i_r = new IconVar().local();
   // Unpack parameters
   VariadicFunction unpack = (Object... params) -> {
        if (params ==  null) { params = EmptyArray(); };
        upto_r.set((params.length > 0) ? params[0] : null);
        i_r.set(null);   return null;  };
   // Method body
   body = new IconSequence(new IconEvery((new IconAssign().over(new IconSingleton(i_r),
        new IconProduct(new IconIn(x_0_r,  new IconInvokeIterator(()->
        ((VariadicFunction) C).apply())), new IconInvokeIterator(()-> ((VariadicFunction)
        IconField.getFieldValue(x_0_r, "range")).apply(lower, upto_r.deref())))))),
        new IconInvokeIterator(()-> System.out.println(i_r.deref()))),  new IconValueSingleton(null),
        new IconFail());
   // Return body after unpacking arguments
   body.setCache(methodCache, "printRange");
   body.setUnpackClosure(unpack).unpackArgs(args);
   return body;
 }
 public Object range (Object... args) { ... }
}
```

Figure 17. Transformation of the sample program to Java.

As another example, Figure 18 shows the results of applying the above transformations to the *findPrimes* method defined in Figure 3. The example demonstrates how the Junicon interpreter can accept a familiar Java-style block notation, in addition to the original Unicon

```
MethodBodyCache methodCache = new MethodBodyCache();
public Object findPrimes = (VariadicFunction) this::findPrimes;
public IIconIterator  findPrimes (Object... args) {
   // Reuse method body
   IconIterator body = methodCache.getFree("findPrimes_m");
   if (body != null) { return body.reset().unpackArgs(args); };
   // Reified parameters
   IconVar lower_r = new IconVar().local();
   IconVar upper_r = new IconVar().local();
   // Temporaries
   IconTmp x_0_r = new IconTmp();
   // Unpack parameters
   VariadicFunction unpack = (Object... params) -> {
      if (params ==  null) { params = EmptyArray; };
      lower_r.set((params.length > 0) ? params[0] : null);
      upper_r.set((params.length > 1) ? params[1] : null);
      return null;
   };
   // Method body
   body = new IconSequence(new IconSuspend(new
      IconOperation(IconOperators.times).over((new IconToIterator(lower_r, upper_r)), new
      IconProduct(new IconIn(x_0_r, new IconToIterator(lower_r, upper_r)), new
      IconInvokeIterator(()-> this.isprime(x_0_r.deref())))))), new IconNullIterator(), new IconFail());
   // Return body after unpacking arguments
   body.setCache(methodCache, "findPrimes_m");
   body.setUnpackClosure(unpack).unpackArgs(args);
   return body;
}
```

Figure 18. Transformation of the *findPrimes* method to Java.

syntax. Lambda expressions and iterators can thus be seen to be the building blocks of the transformations, as well as of the implementation of the iterator calculus for composing suspendable generators.

5.4 Synthesis of co-expressions and generator proxies

To support concurrent generators, the transformations also synthesize co-expressions as well as multi-threaded generator proxies. For co-expressions as well as their multithreaded proxies, the local environment is shadowed as described in Chapter 4, which requires textually scoping up for referenced locals and creating a lambda expression around the

```
MethodBodyCache methodCache = new MethodBodyCache();
public Object spawnMap = (VariadicFunction) this::spawnMap;
public IIconIterator spawnMap (Object... args) {
   IconIterator body = methodCache.getFree("spawnMap_m");     // Reuse method body
   if (body != null) { return body.reset().unpackArgs(args); };
   IconVar f_r = new IconVar().local();                       // Reified parameters
   IconVar chunk_r = new IconVar().local();
   IconTmp x_1_r = new IconTmp();                             // Temporaries
   IconTmp x_0_r = new IconTmp();
   VariadicFunction unpack = (Object... params) -> {          // Unpack parameters
     if (params ==  null) { params = IIconAtom.getEmptyArray(); };
     f_r.set((params.length > 0) ? params[0] : null);
     chunk_r.set((params.length > 1) ? params[1] : null);
     return null;
   };
   // Method body
   body = new IconSequence(new IconSuspend( new IconProduct(new IconIn(x_1_r, (
     new IconCoExpression( (Object... args_2) -> {
       IconVar chunk_s_r = new IconVar().local();
       IconVar f_s_r = new IconVar().local();
       VariadicFunction unpack_4 = (Object... params) -> {
         if (params ==  null) { params = IIconAtom.getEmptyArray(); };
         chunk_s_r.set((params.length > 0) ? params[0] : null);
         f_s_r.set((params.length > 1) ? params[1] : null);
         return null;
       };
       IconIterator body_3 = new IconProduct(new IconIn(x_0_r,  new IconPromote(chunk_s_r)),
         new IconInvokeIterator(()-> ((VariadicFunction) f_s_r.deref()).apply(x_0_r.deref())));
       body_3.setUnpackClosure(unpack_4).unpackArgs(args_2);
       return body_3;
     },  () -> { return IconList.createArray(chunk_r.deref(), f_r.deref()); }).createPipe())),
      new IconPromote(x_1_r))), new IconNullIterator(), new IconFail());
   // Return body after unpacking arguments
   body.setCache(methodCache, "spawnMap_m");
   body.setUnpackClosure(unpack).unpackArgs(args);
   return body;
}
```

Figure 19. Transformation of concurrent generators to Java.

generator that isolates these locals. Code for co-expression as well as proxy creation is then generated, invoking the suspendable iterator runtime. In the latter area, a single core class, *IconCoExpression*, provides a unified model for handling first-class generators as well as co-expressions and multithreaded proxies, and provides support for activating co-expressions, i.e., switching between coroutines, as well as thread creation and communication using blocking queues.

Figure 19 shows the result of applying the above transformations to a simple method for spawning a data-parallel computation using concurrent generators. The method in Junicon before translation is as follows:

```
def spawnMap (f, chunk) {
    suspend ! (|> f(!chunk));
}
```

In Figure 19, in the method body, transformation has unraveled generator expressions into the composition of iterators using forms such as product, embodied in *IconProduct* as well as similarly named classes for operations and function invocation. As can be seen in Figure 19, spawning a thread for a co-expression is transformed into an IconCoExpression constructor over a closure for invoking *f* that first copies the referenced local environment, *chunk*. The *IconCoExpression* then handles the ancillary mechanics of creating the thread, activating the closure within it, and coordinating the communication of results using blocking queues. Thread creation and allocation leverage Java's facilities for thread pool management and support for multi-core execution. As with other transformed methods, the signature of the exposed method, shown at the top of Figure 19, is a variadic lambda expression that returns an iterator.

Chapter 6. Implementation

6.1 Structure of the transformational interpreter

To demonstrate the utility of the approach, we implemented the transformations for embedding Unicon into Java as well as its dynamic analogue Groovy, and housed them within a generic interpretive harness. A key aspect of providing support for multiple languages lies in the structure of the harness itself. The harness provides a cascading set of interpreters that at each stage transforms its input and either executes it on a script engine, such as for Groovy, or chooses another interpreter to pass to for further transformation. The steps involved in each stage of transformation are broken down into preprocessing, followed by parsing either for surface statement detection or into a decorated XML syntax tree, then normalization and translation, and lastly either dispatching to another transformational interpreter or piping into a substrate for execution. In particular, the outermost instantiation of the harness is a meta-interpreter that detects the embedded language and its context using scoped annotations, and dispatches statements to the appropriate sub-interpreter for transformation.

Figure 20 precisely defines the steps involved in transformational interpretation. In detail, the steps involved in transformation begin with a chain of preprocessors that accumulate a complete statement from the history of input lines, shown in Figure 20 in *evalLine*. The next steps are to parse the input into a decorated XML syntax tree, followed by normalization and translation transforms. The *chooseDelegate* method then decides to whom to delegate the result, for example selecting an interpreter based on the input language. Lastly, the transformed expression is either dispatched to another interpreter for further

```
// Accumulate a complete statement from input history
evalLine (line, context) {
    while (null != (unit = getParseUnit(line, context))) {
        eval(unit, context);
    }
}

// Parse, transform, and dispatch or execute statements
eval (input, context) {
    parsed = decorate(parse(input));
    transformed = transform(normalize(parsed));
    dispatchTo = chooseDelegate(transformed);
    if (this == dispatchTo) {
        executeOnSubstrate(transformed, context);
    } else {
        dispatchTo.evalLine(transformed, context);
    }
}
```

Figure 20. Definition of the transformational interpreter.

transformation, or deconstructed and piped into a substrate such as a script engine. The API (Application Programming Interface) for the stages in transformational interpretation is provided in Appendix 1.

The interpreter harness allows different transformation technologies to be used, so that the exact mechanics used for transformation, as well as the transformations themselves, can be customized. The harness also supports pluggable execution substrates, as well as the cross-correlation of error messages back to the original source.

6.2 Configuring the transformational interpreter

The Junicon interpreter is an instantiation of the above harness implemented in Java, customized with a preprocessor, a Javacc LL(k) parser [Reis 2011] for Unicon that emits XML, parameterized XSLT transforms for normalization and translation to either Java or

Groovy, and a Java kernel that implements the stream-like interface for composing suspendable iterators. The complete syntax for Junicon used by the LL(k) parser is provided in Appendix 2. These four pieces of customization amount to roughly 17,400 lines of code: 2300 lines for the parser using Javacc declarations, 5300 lines in XSLT for the transforms, and 9800 lines in Java for the runtime kernel. Each component is carefully engineered to be capable of being run standalone. Together they fully define the transformation of Unicon into both Java and Groovy. The interpreter proper, which provides the interactive capability for transformation and execution on the Groovy script engine, amounts to roughly an additional 7100 lines of Java code.

Behavior of the Junicon interpreter can be rapidly customized through Spring dependency injection [Walls 2011], as well as through the XSLT transforms and a prelude that allows extensions to the kernel. Dependency injection is a type of inversion of control where the instances on which a class depends are injected from outside via setters rather than created by invoking constructors, and are typically referenced using interfaces. Dependency injection is a powerful technique for software development that isolates implementation from specification, and so improves modularity. Spring uses such a technique to wire together class instances and their dependencies, as well as to specify their properties, through declarations in an XML configuration file. In Junicon, a Spring configuration file is used to specify how to transform Unicon constructs, operations, functions, and keywords, as well as how to map them to methods in the runtime kernel. In addition, the Spring configuration file for Junicon specifies the exact grammars and XSLT transformations to use. A subset of the Spring configuration file for Junicon is provided in Appendix 3.

Another widely adopted technology for configuration and extensibility used by Junicon is the Maven build tool. Maven is a declarative tool for building Java projects that is driven by an XML configuration file. Maven is declarative in that the configuration file describes what artifacts and versions are to be built, and the dependencies among them, but not how to build them. The exact plug-ins used to perform the build process are fetched from distributed repositories, and used in conjunction with dependency inference mechanisms to compile and package all required code artifacts.

In Junicon, Maven is used to compile and package the interpreter for both Windows and Linux, as well as the runtime iterator kernel used by compiled Junicon programs. The packaging is novel in that the entire Junicon binary distribution is one executable file. For Windows the packaging consists of the concatenation of a small binary executable (.exe), followed by a shell script (.bat), and finally appended with the Junicon interpreter and runtime in a single Java jar file. On execution, the generic front-end executable finds and runs the embedded script in-memory with the command-line arguments, which in turn invokes Java with the Junicon jar file to perform either interactive interpretation or translation. For Linux the packaging is analogous to that of Windows but a bit simpler, and consists of a small shell script, followed by the jar file for the Junicon interpreter and runtime. The above technique works because Windows executables and Linux shell scripts load from the front, while zip files, of which jar files are a subset, load from the back. Thus, at each phase of execution, from running the front-end script to running Java on the Junicon jar file, execution just operates on itself, the original conjoined file.

A similar technique is used by the Junicon interpreter to package its own compiled programs to run under Java in a standalone manner. A program that embeds Junicon into

Java is first transformed into Java, and then compiled into a jar file that depends only on the Junicon runtime. A single executable file is then formed for either Windows or Linux by concatenating the Windows executable loader or Linux bash script, respectively, with the merged compiled program and Junicon runtime. The result is a standalone executable jar file that runs the embedded Junicon program in conjunction with the conjoined Junicon runtime. The latter is quite small, at about 150K, and depends only on core Java libraries in the standard distribution of the Java Runtime Environment.

As another means to facilitate ease of use, the Junicon interpreter also provides an option to choose a Java-style syntax (-J) for the embedded Junicon. In detail, such syntax uses = instead of :=, == instead of ===, and also allows *def* to be used instead of *method* or *procedure*, as well as *var* to be used instead of *local*. The above redefinitions are implemented using the capabilities of the preprocessor to perform token substitution. The above technique allows embedding generators using a familiar notation that is consistent with the surrounding Java.

6.3 Supported features

The implementation currently supports all Unicon operators, data structures, and control constructs, and four-fifths of the built-in Icon functions. In particular, full support is provided for co-expressions and threads, as well as all operators for thread communication using blocking channels. Thread creation and coordination leverages the Java facilities for concurrency including thread pool management and multi-core execution. Support is also provided for string scanning using the *subject ? expr* operator, which performs scanning operations on the subject. In the area of Icon functions, 77 out of 91 functions are implemented, including all mathematical functions, all collection functions, all system functions except *errorclear* and *getenv*, all file functions except *chdir* and *kbhit*, all string functions except *detab* and *entab*, and 3 of the 11 reflective functions, i.e., *copy*, *image*, and *type*. In the area of Icon keywords, 13 out of 41 keywords are implemented, including *&subject* and *&pos* for string scanning, *&current*, *&source*, and *&main* for co-expressions, *&input* and *&output* for file operations, and the core *&fail* and *&null* keywords.

However, Junicon does not support the multiple inheritance provided by Unicon, for reasons of consistency with Java, as well the practical difficulty in transformation posed by the need to resolve class dependencies. Although multiple inheritance could feasibly be supported when embedding into Groovy using Groovy mixins, it is more difficult when embedding into Java. The difficulty arises principally from the need to scope up through external class bytecode definitions to find the mixins, and then integrate them through delegation. As with recognizing and differentiating plain Java method invocation from Unicon method invocation, our strategy has been to purposefully avoid such resolution. As an alternative to multiple inheritance, interfaces can be used within Junicon by embedding into a Java class that implements, i.e., extends, them. Concomitantly, interfaces for Junicon methods can be defined directly in Java using the signature of a variadic function over *Objects*. All other features of Unicon classes are, however, supported at the level of fields, methods, and constructors. This support enables embedding generators into the familiar object-oriented setting of Java, as well as the full-fledged transformation of entire Unicon classes.

By enabling embedding within either Groovy or Java, the interpreter can function both interactively and as a tool that can emit its output for compilation that is free of dependencies on Groovy. The provision for interactive execution significantly enhances the ability for exploration as well as rapid prototyping and refinement within an iterative development methodology. Moreover, at a meta-level, the capability for scripted customization of the interpreter itself enables rapid prototyping of translation enhancements. Indeed, the Junicon kernel was initially prototyped in Groovy, and later refined into Java to provide improved performance.

6.4 Using XSLT for program transformation

There are a variety of transformation technologies that could be used to implement the rewriting rules derived in the previous chapter. These technologies include program transformation systems such as Stratego/XT [Bravenboer et al. 2008] and Spoofax [Kats and Visser 2010], as well as metaprogramming support in languages such as Groovy [Dearle 2010]. However, our techniques do not demand capabilities for ad-hoc syntactic extension, nor demand the power or formality of full-fledged transformation tools.

As part of this research we explored the utility of using XSLT as an alternative means of program transformation. XSLT is a rule-based language for transforming XML documents expressed in XML itself [Kay 2008, Clark 1999, Clark and DeRose 1999]. An XSLT transform consists of a set of templates, or production rules, whose preconditions are XPath patterns and that substitute the specified content for any matched XML node. The production rules can be grouped into modes as well as prioritized to effect specific rewriting strategies.

For illustration, two rewrite rules that create local declarations for all temporary variables in bound iterators within a given block scope are shown in Figure 21. The XSLT templates in Figure 21, in an extremely succinct fashion, only create local variable

```
<xsl:template match="BLOCK"  mode="findTemporaries"  priority="2">
        <xsl:copy>
                <xsl:copy-of select="@*"/>
                <xsl:variable name="blockDepth" select="count(ancestor-or-self::BLOCK)"/>
                <xsl:apply-templates select=".//EXPRESSION[@isTemporary and
                        ($blockDepth = count(ancestor::BLOCK))]"  mode="createLocal"/>
                <xsl:copy-of select="*"/>
        </xsl:copy>
</xsl:template>

<xsl:template match="*"  mode="createLocal"  priority="2">
        <xsl:param name="variableName" select="@tmpVariableName"/>
        <STATEMENT>
                <KEYWORD>local</KEYWORD>
                <DECLARATION>
                        <IDENTIFIER>
                             <xsl:value-of select="$variableName "/>
                        </IDENTIFIER>
                </DECLARATION>
                <DELIMITER ID=";"/>
        </STATEMENT>
</xsl:template>
```

Figure 21. XSLT template for synthesizing temporary variables.

definitions for temporaries that appear within a block but not in any subordinate block, i.e., if the temporary and block have the same block ancestor count.

We found XSLT to be remarkably expressive in capturing context, for example looking up or down in scope to resolve references, as well as in prioritized rewrite strategies. While XSLT was found to be effective in this small-scale scenario – the transformations for all of Junicon are less than 5300 lines – its verbosity and lack of formal basis may hinder its scalable application to other domains. On the other hand, XSLT is standardized and its Version 1.0, which we use, is bundled into the standard Java runtime environment. For our purposes, which is migration between high-level languages, it was found to be highly advantageous.

The transformations for taking Unicon into Java, formalized as rewrite rules in Chapter 5, are implemented in two stages of XSLT transformation: a normalization stage $\mathcal{N}$, followed by $\mathcal{C}$ for the transformation of classes, methods, and expressions. The XSLT templates, i.e., the transformation rules, are further partitioned into modes for each of the transforms $\mathcal{N}$, $\boldsymbol{\mathcal{F}}$ within $\mathcal{N}$, $\mathcal{K}$, and $\mathcal{C}$. The XSLT templates faithfully reproduce the rewrite rules previously described.

Chapter 7. Evaluation

Performance is not the primary goal of the Junicon implementation. Rather the concerns of seamless interoperability, compactness, and semantic clarity are paramount. However, reasonable performance comparable to that of Unicon is desirable, and an implementation with a slowdown on an order of magnitude would be of little practical use. To evaluate the practical viability of the techniques for embedding Unicon in Java, measurements of compiled Java translations were undertaken, and compared to that natively run under Unicon. In addition, the performance of suspendable iteration over time was measured when embedding in Groovy as well as when embedding in Java. Lastly, the performance of embedded concurrent generators was evaluated against equivalent Java stream-based programs.

7.1 Performance relative to Unicon

The performance of Junicon relative to that of Unicon was benchmarked using a suite of eight programs, most of which were derived from the Computer Language Benchmarks Game [BenchmarksGame 2015]. The programs exercise a wide range of Unicon features, and include Matrix Multiply, PiDigits, Quicksort, N-body, Mandelbrot, Spectral Norm, and Fannkuchen, as well as NewInstances from the pybench benchmark suite. The Java Microbenchmarking Harness (JMH) was used in measuring Junicon's performance on an AMD Dual-Core Opteron 2212 with 8GB of memory running Linux Mint 17.1, with 20 warmup iterations and 20 test iterations of 5 minutes each. Sample sizes for each program were chosen to uniformly yield the same execution time. The results were then compared to execution times under Unicon using an identical number of warmup and test iterations.

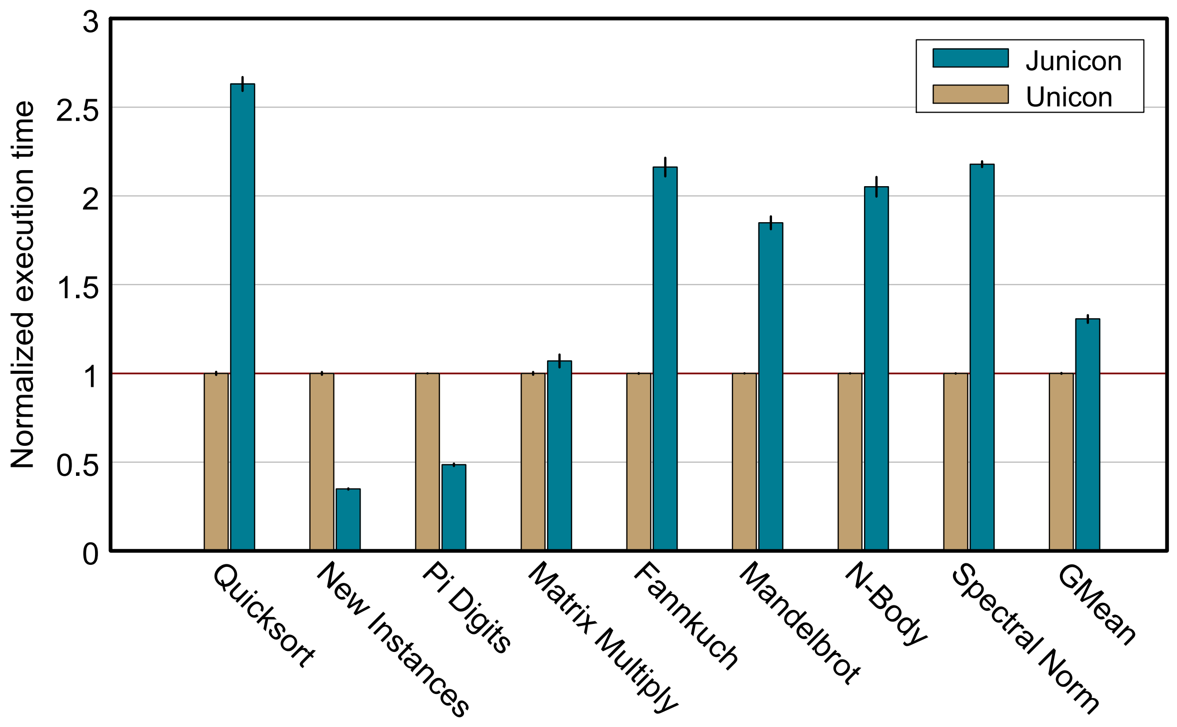


Figure 22. Performance of Junicon when translated to Java.

Figure 22 shows the relative performance of Junicon when translated to Java. Execution time is normalized with respect to that of Unicon. Confidence intervals at 99%, shown by whiskers at the top of the histogram bars, showed negligible variance. The results showed that Junicon yields only marginally worse, and sometimes better, performance than that of Unicon. The geometric mean of the normalized execution time, shown in the last column, was 1.30.

The observed performance represents a tradeoff for the advantages gained in interoperability and compactness of implementation. Although extreme measures for optimization have not been taken, obviously wasteful implementation techniques, such as redundant reified declarations and repeated iterator construction in method bodies, have been avoided. In particular the Java implementation excels at arbitrary precision arithmetic, exercised by PiDigits, and object allocation in NewInstances, but is weaker in recursive function invocation as evidenced in Quicksort. However, a potentially positive impact on performance is that our translation imposes a low overhead on calling Java methods, with no conversions needed through a foreign function interface. In the previous Unicon implementation, calling down to the implementation language C was costly, and motivated refinements for native array types [Al-Gharaibeh et al. 2012a, Al-Gharaibeh 2012b]. In contrast, the Junicon implementation provides a greater measure of transparency, and moreover, can leverage other Java foreign language interfaces such as JNI (Java Native Interface) when needed.

7.2 Performance of suspendable iteration

In addition to the above evaluation, a comparative analysis of the performance of core functions for suspendable iteration was undertaken when embedding into Java as well as into Groovy. In this case, the performance of Junicon relative to Unicon was measured over time for both compiled Groovy and compiled Java translations. The motivation for measuring Groovy's performance was, in part, to evaluate the overhead of its use of relaxed typing as well as dynamic dispatch for method selection based on runtime types. The suite of programs was also refined to gain deeper insight into the overhead of Junicon's suspendable iteration and use of lambda expressions for function invocation. Moreover, the performance was also analyzed as a function of execution time, in order to gain a better idea of the impact of Java's dynamic compilation.

The performance of Junicon relative to that of Unicon for the suspendable iteration kernel was benchmarked using a suite of six programs. The programs exercise a wide range of Unicon features: Matrix Multiply tests list creation and access and is O(n^3), Quicksort exercises recursive method invocation and is O(n log(n)), NewInstances exercises instance

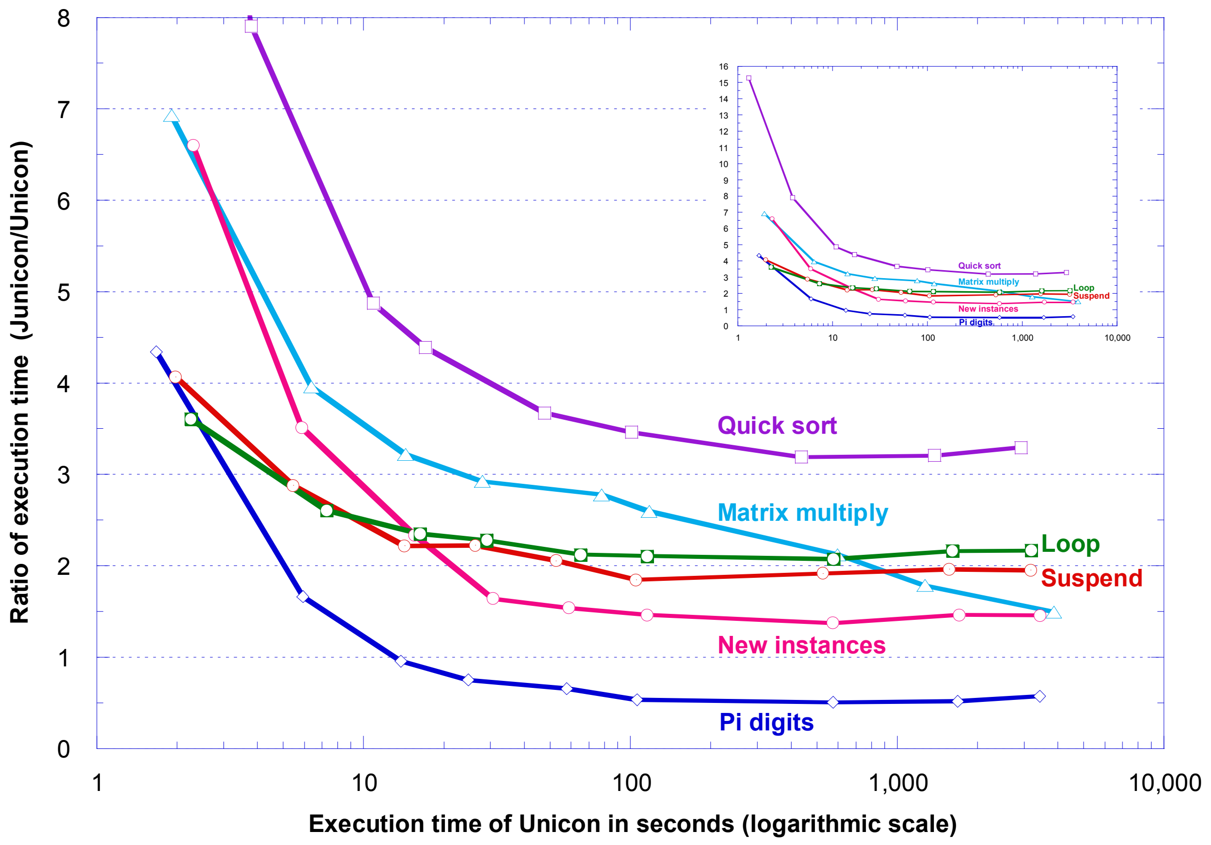


Figure 23. Performance of Junicon when translated to Groovy.

creation and field access and is O(n), PiDigits, which computes pi to a given length, exercises arbitrary precision arithmetic and is O(n^2), Loop Test measures the basic efficiency of the iterator calculus and is O(n^2), and Suspend Test, which is structurally similar to Loop Test, measures the overhead of suspend and is O(n). Sample sizes for each program were chosen to uniformly effect exponential execution time, ranging from 2 seconds to one hour, and each sample point is the average of three runs.

Figures 23 and 24 show the performance of Junicon relative to that of Unicon for compiled Groovy and compiled Java translations, respectively. The performance comparisons in Figure 23 are based on code compiled under Groovy 2.3 that exploits Java invoke dynamic support, and run under Java 1.8. In contrast, Figure 24 shows the relative

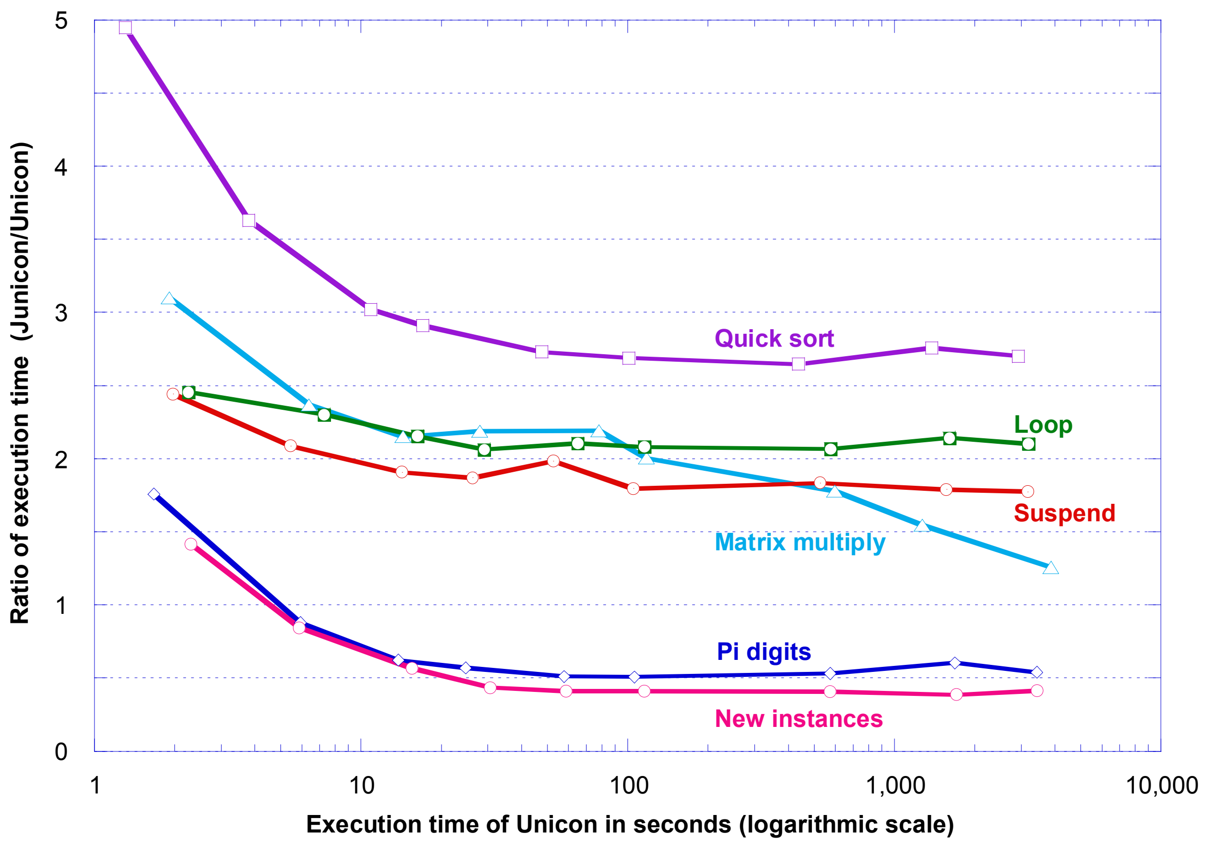


Figure 24. Performance of core iteration functions in Java.

performance of Junicon translated to Java, and then compiled and run under Java. The benchmarks were run on an AMD Dual-Core Opteron 2212 with 8GB of memory running Linux Mint 17. The execution times were measured using the *time* command by adding both the user and system time, the latter which includes the overhead of Java Just-in-Time (JIT) compilation running on the second core.

Consistent with the evaluation in the previous section, the results show that Junicon yields only marginally worse, and sometimes better, performance than that of Unicon. In Figures 23 and 24, each data point represents the ratio of Junicon's execution time to that of Unicon for a given program and sample size. Values below 1 on the y-axis demonstrate better performance than that of Unicon, while values above 1 correspond to worse performance. In both figures, in the initial stages the performance of Junicon rapidly improves over time as Java's Just-in-Time (JIT) dynamic compiler converts often used methods, both user and system, to directly executable instructions. As shown in PiDigits, the use of Java's arbitrary precision arithmetic performs better than that of Unicon. The implementation of suspend can be seen to incur little overhead, as evidenced when comparing the performance of Loop Test to Suspend Test. These two programs give a baseline overhead for using the iterator calculus to implement generator expressions, which is roughly a factor of 2 slowdown over Unicon.

Both Groovy and Java over the long term have roughly similar performance; however, Groovy's initial performance is quite a bit slower than that of Java, and as Groovy has a larger dependency set it takes longer for JIT to have full effect. For example, in Figure 23 Quicksort initially has a performance slowdown over 15 relative to that of Unicon; the inset to Figure 23 shows the full graph with the y-axis expanded to include Quicksort's first data point. Moreover, Groovy's overhead in dynamic dispatch is evidenced in the degraded performance of NewInstances, which heavily employs static method invocation to overlay instance construction.

If one views the Loop Test as giving a baseline to the overhead of using the iterator calculus to implement generator expressions, which is roughly a factor of 2 slowdown over Unicon, then the other sample programs are similarly impacted by that inherent overhead. Thus any speedup beyond the baseline of a factor of 2 slowdown, representing calculus overhead, may be a more accurate measure of improvement of a given feature's performance over that of Unicon. Using that analysis, the performance of PiDigits could be interpreted to indicate a speedup by a factor of 4 over Unicon for the feature of arbitrary precision arithmetic. Lastly, we also examined the performance of several other variants such as using inner classes instead of lambda expressions in Java, as well as exposing methods in Groovy using function references in a manner similar to that done for Java, with little difference observed.

Of particular interest is that the performance of Junicon is roughly equivalent, or even a little faster, than that reported for Jcon relative to Icon. As mentioned, Jcon used a different implementation technique based on instrumentation of fail-resume ports to produce JVM byte code, and reported an overall slowdown by a factor of two relative to Icon [Proebsting and Townsend 2000]. The roughly similar performance results between the two radically different implementation techniques of Junicon and Jcon, despite that fact that Junicon does not directly generate bytecode, might imply that the performance difference between Unicon and the two Java translations is due to the inherent overhead of translating a dynamic language for generator expressions into Java. Consequently one might be led to surmise that Junicon's technique for transformation into an iterator calculus is potentially as efficient as any other implementation technique.

7.3 Performance of concurrent generators

To evaluate the utility of the techniques for embedding concurrent generators, several variants of the program described in Figure 6 were compiled to Java, and their performance measured against equivalent Java stream-based programs. The suite of embedded Unicon programs consisted of a sequential word-count, a pipeline-parallel word-count that split the hash function into two tasks, a map-reduce word-count that spread the hash function and its summation reduction over chunks of data, and a data-parallel word-count that only differed in performing summation over the sequence returned from flattening the chunks, thus splitting out the reduction and effecting serialization. The suite of Java programs similarly consisted of a sequential word-count, a pipelined version built using *BlockingQueues* over two threads, a parallel stream-based version that implemented map-reduce, and a data-parallel version that was also stream-based but that split out the reduction. Both suites used arbitrary precision arithmetic, which is implicit in Unicon but must be made explicit in Java.

The Java Microbenchmarking Harness (JMH) was used to measure the performance of both suites on a Titan Quad AMD Opteron 6272 with 64-cores and 32GB of memory running Linux Fedora 20, with 20 warmup iterations and 20 test iterations. In addition, a second heavyweight set of variants of the programs in both suites was also benchmarked, which increased the complexity of the hash function components and so the weight of the threaded tasks, in order to explore the relative overhead of coordination using concurrent generators.

Figure 25 shows the relative performance of embedded concurrent generators when translated to Java. Execution time is normalized with respect to that of the Java parallel stream benchmark for each of the lightweight and heavyweight sets, respectively. Confidence intervals at 99%, shown by whiskers at the top of the histogram bars, showed negligible variance. In Figure 25, the eight histograms on the left use the lightweight versions of the *wordToNumber* and *hashNumber* functions described in Figure 6 that constitute the parallel computation nodes. On the right of Figure 25 are shown eight corresponding histograms that use the far more heavyweight and computationally intensive hash functions, by a factor of roughly 80, achieved using trigonometry and prime number functions of Java's *Math* and *BigInteger* libraries.

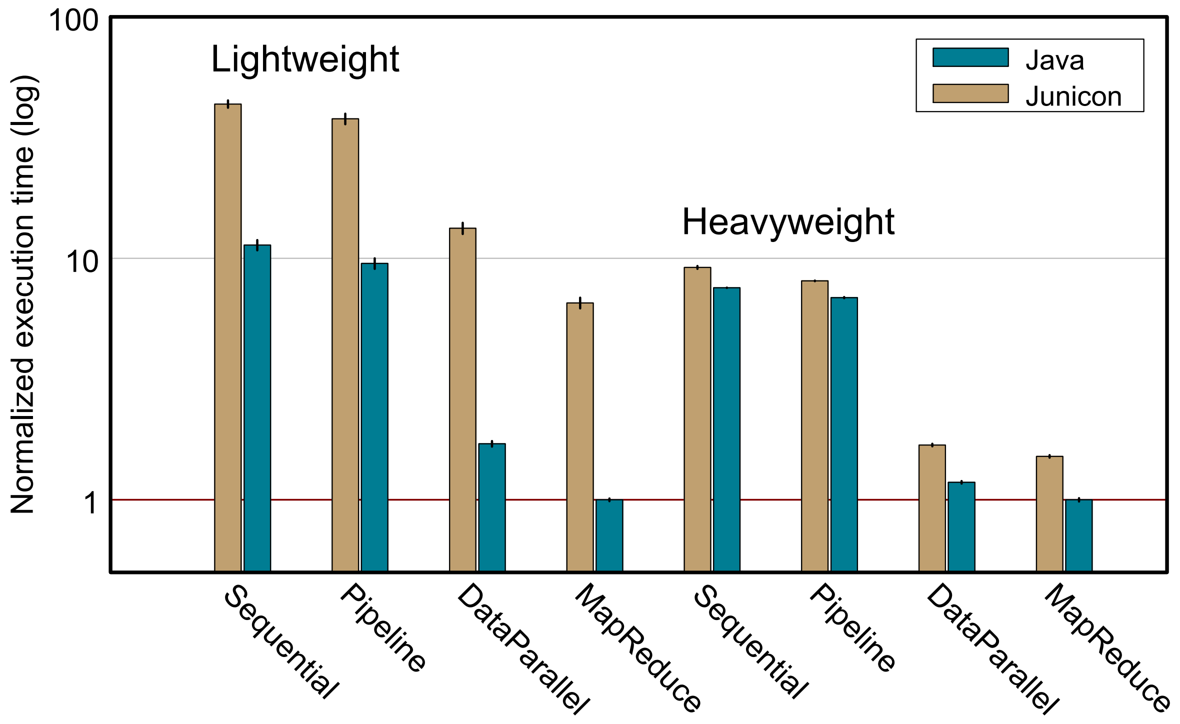


Figure 25. Performance of concurrent generators.

The results showed that, as would be expected of a dynamic language, embedded generators yield worse performance than their native Java counterparts; however, the penalty is well under an order of magnitude. Moreover, as can be observed in the right of Figure 25, as the weight of the computational nodes increases, the relative overhead of the embedded concurrent generators significantly decreases. Indeed, even with map-reduce expressed entirely using concurrent generators, the performance impact on the right of Figure 25 is negligible. When used to coordinate complex tasks, concurrent generators may thus potentially provide performance roughly comparable to that of Java streams.

Another salient point is that the relative improvement among the embedded programs is roughly consistent with that of the comparable Java programs. For the purposes of exploration in a prototyping scenario, ideally it should be the case that the relative observed performance among experimental alternatives is preserved under refinement. While the benchmark results are preliminary and a proof of concept, they demonstrate the potential feasibility of exploration using concurrent generators.

Chapter 8. Related Work

8.1 Goal-directed evaluation

The challenges of formalizing and implementing goal-directed evaluation have given rise to a variety of research efforts. Principal among these efforts was a Java implementation for Icon called Jcon [Proebsting and Townsend 2000], a bytecode generator that relied on a Prolog-like Byrd-box model [Byrd 1980] to instrument backtracking using fail and resume ports [Proebsting 1997]. Other implementation efforts for Icon primarily relied on continuation-based approaches for cross-translation, such as recursive interpretation using failure continuations [O'Bagy and Griswold 1987], cross-compilation into C using a continuation-passing-style [O'Bagy et al. 1993], an optimizing compiler into C [Walker and Griswold 1992], and a stream-based translator into Scheme [Allison 1990]. Efforts to formalize the semantics of Icon included a denotational semantics based on continuations [Gudeman 1992], as well as a monad semantics for a small subset with compilation by type-driven partial evaluation [Danvy et al. 2002]. However, the residual programs of the above approach, as with Jcon, instrument code with suspend and resume advice, which renders problematic its interfacing with other programs.

In contrast to these efforts our transformations rely on flattening to do the work of instrumentation or higher-order functions used in a continuation-based approach, and so enable interoperability. Our research also addresses a wider set of concerns including generator propagation in an object-oriented setting. Under our approach certain problematic features of Icon and Unicon such as first-class patterns [Walker 1989] and concurrency [Al-Gharaibeh et al. 2012a, Al- Gharaibeh 2012b] become simpler to implement. Lastly, and perhaps most importantly, our research focuses on the larger problem of mixed-language embedding of goal-directed evaluation into other object-oriented languages.

8.2 Program transformation

There are a broad array of transformation tools that could be brought to bear to implement the rewriting rules [Feather 1987, Visser 2005]. These technologies range from program transformation systems such as Stratego/XT [Bravenboer et al. 2008] and Spoofax [Kats and Visser 2010], to metaprogramming capabilities in languages such as Groovy [Dearle 2010] that allow extensions for domain specific languages. While the former are more formally based on concepts from term writing and theorem proving, the latter provides more ad-hoc support for manipulating the exposed syntax tree within the language itself.

However, our techniques for the transformation of dynamic languages do not demand capabilities for syntax extension, nor demand the scope or formality of full-fledged transformation tools such as those above. For the small-scale transformation of dynamic languages, XSLT is a potentially attractive alternative due to its simplicity as a rewriting system and its bundled support in the Java standard distribution.

8.3 Parallel dynamic languages

In the area of concurrency, there are a variety of dynamic languages that support parallelism. These include Swift [Wilde et al. 2011], which supports implicit parallelism in which every data-element is single-assignment and behaves like a future. Oz [Smolka 1995, Van Roy 2005] is a multi-paradigm language for distributed programming whose explicit threads also use single-assignment dataflow variables. Julia [Bezanson et al. 2014] provides explicit task spawning and synchronization based on futures as well as blocking channels, and also supports metaprogramming through hygenic macros. Parallel Ruby [Lu et al. 2014] is similarly based on explicit task parallelism using futures as well as pipelines.

In contrast to the above efforts, our approach is one of mixed-language embedding rather than multi-paradigm integration within a single language, and focuses on grafting a simple model of concurrent generators onto other languages through transformations that enable interoperability. We can thus leverage and integrate with the broader concurrency mechanisms of the underlying language.

8.4 Suspendable iteration

The aggregate operations of goal-directed evaluation, as well as the iterator calculus in which it is distilled, in many regards bear a striking similarity to the Java *Stream* interface. Indeed, many of the stream composition operators such as *map*, *reduce*, and *limit* are either implicit or present as primitives in Icon. Generators differ, however, in the support for suspendable iteration, as well as the provision of forms such as *repeat* and *product* that depend on the ability to be restarted.

The extension of Java iterators to support suspendable iteration bears some similarity to other work on interruptible iterators in JMatch [Liu et al. 2006, Liu and Myers 2003]. There the focus was on extending coroutine iterators to handle update operations on the underlying data structure. Interruptible iterators for ML were also examined by Filliatre [Filliatre 2006] using purely functional persistent cursors that also allow backtracking. In contrast, our iterators integrate suspend with failure-driven composition in a tightly knitted logic. It is also feasible to alternatively use multithreading to create a coroutine-like implementation of *suspend* in generator functions, as is provided in several Groovy and Java extension classes. However, the cost of multithreading is not minor. Our techniques for embedding goal-directed evaluation not only enable the use of *suspend* in languages otherwise missing such a capability, but implement it without multithreading.

Chapter 9. Conclusions

We developed a technique for embedding goal-directed evaluation into other object-oriented languages based on program transformation. We presented a novel form of annotations, called scoped annotations, that are used in conjunction with the transformations to support mixed-language embedding. The transformations first use flattening to unravel the syntax of pervasive generators to a conventional form, which is key to enabling interoperability. The rewriting rules then map control constructs and operations onto a stream-like calculus for composing suspendable iterators that extends the products of bound iterators used in flattening with functional forms such as *reduce* and *exists*. Lastly, to accommodate Unicon's reference semantics, methods are exposed as variadic lambda expressions. We demonstrated the utility of the approach by concretely targeting the transformations to allow embedding into Java as well as its dynamic analogue Groovy, and so realized an implementation that can function both interactively and as a translator for compilation. We evaluated the performance of embedding when translated to Java, and found it comparable to that of native Unicon.

We further extended the transformations to enable mixed-language embedding of concurrent generators. We presented a simple model of explicit concurrency for generators based on co-expressions and generator proxies, called pipes, that communicate with the original expression running in a separate thread using blocking queues. We demonstrated the utility of the approach in expressing parallel pipelining as well as in building higher-order abstractions such as map-reduce, and evaluated their performance against equivalent Java stream-based programs.

The use of transformational interpretation for mixed-language embedding has several distinct advantages. First, the rewriting rules of the transformations clarify the semantics of

Unicon by reducing nested generators to a familiar and explicit form that is more closely allied to conventional programming concepts. Moreover, the derivation of an iterator calculus yields an equational definition of Unicon's control constructs, and enables a compact implementation of suspendable iteration. Second, the normalization techniques that make explicit the otherwise implicit generator propagation enable the grafting of goal-directed evaluation onto other object-oriented languages. Third, by carefully preserving native types and invocation mechanisms, the implementation can cleanly integrate with and leverage the full range of Java capabilities, including its portability and libraries for concurrency and graphics. Moreover, by translating onto the dynamic language Groovy and leveraging its underlying scripting engine, we are able in turn to realize an interactive interpreter.

The capability for embedding a dynamic language for goal-directed evaluation within a familiar object-oriented setting, in combination with such interactive evaluation, enables exploration and rapid prototyping. In particular, mixed-language embedding allows using the succinct notation of concurrent generators for the coordination of coarse-grained processes in other languages, as well as the rapid prototyping and refinement of parallel programs for multi-core architectures.

Lastly, the Junicon implementation demonstrates that XSLT is potentially well suited for expressing several types of program transformations. While not terribly well founded in formal methods, the efficacy of XSLT as a rewriting system and its bundled support in Java make it extremely attractive. Moreover, its use yielded an extremely compact implementation that is readily extensible and retargetable. Rapid development of translation enhancements is facilitated by the scripted nature of the XSLT transforms. We examined the ease of retargeting by migrating the transforms to Java as well as Groovy. Junicon's extensibility is further enhanced by its configuration mechanisms, based on Spring dependency injection, that specify how operations and built-in functions are mapped onto Groovy or Java classes.

Currently the implementation [Junicon 2016] supports the full set of Unicon constructs and operators, including those for concurrency and co-expressions, as well as most of Icon's built-in functions. Future efforts will focus on further refining the abstractions for concurrency in generators, as well as evaluating their performance. Lastly, program monitoring and debugging within a transformational framework is an area to be further explored.

References

Abrial, J.-R., Schuman, S. A., and Meyer, B. 1980. A specification language. In *On the Construction of Programs*, R. M. McKeag and A. M. Macnaghten, Eds., Cambridge University Press, Chapter 11, 343-410.

Al-Gharaibeh, J., Jeffery, C., and Bani-Salameh. H. 2011. Building a Collaborative Virtual Environment: a Programming Language Codesign Approach. In *2011 International Conference on Cyberworlds*. IEEE Press, 54-61.

Al-Gharaibeh, J. 2012. *Programming Language Support for Virtual Environments.* Ph.D. Dissertation. University of Idaho, Moscow, Idaho.

Al-Gharaibeh, J., Jeffery, C., and Oikonomou, K.N. 2012. An hybrid model for very high level threads. In *Proceedings of the 2012 PPOPP International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM 2012)*. ACM Press, New York, NY, 55-63.

Allison, L. 1990. Continuations implement generators and streams. *The Computer Journal 33*, 5, Oxford University Press, 460-465.

Backus, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM 21*, 8, ACM Press, New York, NY, 613-641.

Barth, P. S., Nikhil, R. S. and Arvind. 1991. M-structures: Extending a parallel, non-strict functional language with state. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. Lecture Notes in Computer Science 523, Springer-Verlag, 538-568.

BenchmarksGame. 2015. Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org.

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V.B. 2014. Julia: A Fresh Approach to Numerical Computing. Technical Report arXiv:1411.1607, Cornell University Library Archive, Computer Science. http://arxiv.org/pdf/1411.1607v4.

Bienia, C., and Li, K., 2010. Characteristics of Workloads Using the Pipeline Programming Model. In Proceedings of the International Conference on Computer Architecture, Workshop on Emerging Applications and Many-core Architecture. Springer-Verlag, 161-171.

Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming 72*, 1, Elsevier, 52-70.

Byrd, L. 1980. Understanding the control of Prolog programs. Technical Report 151. Department of Artificial Intelligence, University of Edinburgh, Scotland. 12 pages.

Carriero, N., and Gelernter, D. 1989. Linda in Context. *Communications of the ACM 32*, 4, ACM Press, New York, NY, 444-458.

Clark, J. (Ed.). 1999. XSL Transformations (XSLT), Version 1.0, W3C Recommendation 16 (November 1999). http://www.w3.org/TR/xslt.

Clark, J. and DeRose S. (Eds.). 1999. XML Path Language (XPath), Version 1.0. W3C Recommendation 16 (November 1999). http://www.w3.org/TR/xpath.

Clocksin, W. F. and Mellish, C. S. 1981. *Programming in Prolog*. Springer-Verlag, Berlin.

Danvy, O., Grobauer, B., and Rhiger, M. 2002. A unifying approach to goal-directed evaluation. *New Generation Computing 20*, 1, Springer-Verlag, 53-73.

Davies, J. and Woodcock, J. 1996. *Using Z: Specification, Refinement and Proof.* Prentice Hall International Series in Computer Science. http://www.usingz.com/text/online.

Dean, J., and Ghemawat, S. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM 51*, 1, ACM Press, New York, NY, 107-113.

Dearle, F. 2010. *Groovy for Domain-Specific Languages.* Packt Publishing. http://groovy.codehaus.org/.

Dewar, R. B. K., Schwartz, J. T., and Schonberg, E. 1981. Higher level programming: Introduction to the use of the set-theoretic programming language SETL. Technical Report, Computer Science Department, Courant Institute of Mathematical Sciences, New York University.

Dijkstra, E. W. 1975. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM 18*, 8, ACM Press, New York, NY, 453-457.

Feather, M. S. 1987. A survey and classification of some program transformation approaches and techniques. In *IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation.* North-Holland Publishing Co. Amsterdam, The Netherlands, 165-195.

Filliatre J.-C. 2006. Backtracking iterators. In *Proceedings of the 2006 Workshop on ML (ML'06).* ACM Press, New York, NY, 55-62.

Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. 2014. *The Java Language Specification, Java SE 8 Edition.* Addison-Wesley Professional.

Griswold, R.E., Poage, J. F., and Polonsky, I. P. 1971. *The SNOBOL 4 Programming Language, 2nd Edition.* Prentice-Hall, Englewood Cliffs, N.J.

Griswold, R. E., Hanson, D. R., and Korb, J. T. 1981. Generators in Icon. *ACM Transactions on Programming Language and Systems 3*, 2, ACM Press, New York, NY, 144-161.

Griswold, R. and Griswold, M. 1996. *The Icon Programming Language, Third Edition.* Peer-to-Peer Communications.

Gudeman, D. A. 1992. Denotational semantics of a goal-directed language. *ACM Transactions on Programming Language and Systems 14*, 1, ACM Press, New York, NY, 107-125.

Hoare, C.A.R. 1978. Communicating sequential processes. *Communications of the ACM 21*, 8, ACM Press, New York, NY, 666-677.

Hudak, P., Peterson, J., and Fasel, J. 1999. A gentle introduction to Haskell 98. Technical Report, Yale University, 64 pages.

Jagger, J., Perry, N., and Sestoft, P. 2007. *C# Annotated Standard.* Morgan Kaufmann Publishers Inc., San Francisco, CA.

Jeffery, C. L. 2001. Goal-directed object-oriented programming in Unicon. In *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC'01).* ACM Press, New York, NY, 306-308.

Jeffery, C., Mohamed, S., and Al-Gharaibeh, J. 2013. Unicon Language Reference. Technical Report UTR8a. University of Idaho, Moscow, Idaho. 47 pages. http://unicon.sourceforge.net/utr/utr8a.pdf.

Jeffery, C., Mohamed, S., Al-Gharaibeh, J., Pereda, R., and Parlett, R. 2013. *Programming with Unicon, 2nd Edition.* http://www.unicon.sourceforge.net/ub/ub.pdf.

Jones, S. P. and Wadler, P. 1993. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93).* ACM Press, New York, NY, 71-84.

Junicon. 2016. The Junicon project. http://sourceforge.net/projects/junicon.

Kats, L. C. L. and Visser, E. 2010. The spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10).* ACM Press, New York, NY, 444-463.

Kay, M. 2008. XSLT 2.0 and XPath 2.0 Programmer's Reference, 4th Edition. Wrox Press, New York, NY.

Klein, C., McCarthy, J., Jaconette, S. and Findler, R. B. 2011. A Semantics for Context-Sensitive Reduction Semantics. In *Proceedings of the 9th Asian Symposium on Programming Languages and Systems (APLAS 2011).* Lecture Notes in Computer Science 7078 (H. Yang, editor), Springer-Verlag, 369-383.

Kowalski, R. A. 1979. Algorithm = Logic + Control. *Communications of the ACM 22*, 7, ACM Press, New York, NY, 424-436.

Li, G. 2010. *Formal verification of Programs and Their Transformations.* Ph.D. Disseration. University of Utah, Salt Lake City, Utah.

Liang, S., Hudak, P., and Jones, M. P. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95).* ACM Press, New York, NY, 333-343.

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. 1977. Abstraction mechanisms in CLU. *Communications of the ACM 20,* 8, ACM Press, New York, NY, 564-576.

Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J. C., Scheifler, R., and Snyder, A. 1981. *CLU reference manual*. Lecture Notes in Computer Science 114*,* G. Goos and J. Hartmanis, Eds., Springer-Verlag, Berlin.

Liu, J. and Myers, A. C. 2003. JMatch: Iterable abstract pattern matching for Java. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL'03).* Lecture Notes in Computer Science 2562, Springer-Verlag, 110-127.

Liu, J., Kimball, A., and Myers, A. C. 2006. Interruptible iterators. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming (POPL'06).* ACM Press, New York, NY, 283-294.

Lu, L., Ji, W., and Scott, M.L. 2014. Dynamic Enforcement of Determinism in a Parallel Scripting Language. In *Proceeding of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 519-529.

Mills, P. and Jeffery, C. 2014. A Transformational Interpreter for Goal-Directed Evaluation. Unicon Technical Report UTR17, Department of Computer Science, University of Idaho, June 12 2014.

Mills, P. and Jeffery, C. 2016. Embedding Goal-Directed Evaluation through Transformation. To appear in *Proceedings of the 31st ACM Symposium on Applied Computing*. ACM Press, New York, NY, USA.

Mills, P. and Jeffery, C. 2016. Embedding Concurrent Generators. To appear in *Proceedings of the 21st International Workshop on High-Level Parallel Programming Models and Supportive Environments, 30th IEEE International Parallel & Distributed Processing Symposium*. IEEE Press.

Moggi, E. 1991. Notions of computation and monads. *Information and Computation 93*, 1, Elsevier, 55-92.

Nikhil, R.S., Arvind, Hicks, J., Aditya, S., Augustsson, L., Maessen, J., and Zhou, Y. January 1995. pH Language Reference Manual, Version 1.0. Technical Report Memo-369, MIT Computation Structures Group, http://csg.csail.mit.edu/pubs/memos/Memo-369/memo-369.pdf.

O'Bagy, J. and Griswold, R. E. 1987. A recursive interpreter for the Icon programming language. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques.* ACM Press, New York, NY, 138-149.

O'Bagy, J., Walker, K., and Griswold, R.E. 1993. An operational semantics for Icon: Implementation of a procedural goal-directed language. *Computer Languages 18*, 4, Elsevier, 217-239.

Peyton-Jones, S., Gordon, A., and F. Sigbjorn, 1996. Concurrent Haskell. *In Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 295-308.

Proebsting, T. A. 1997. Simple translation of goal-directed evaluation. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI'97)*. ACM Press, New York, NY, 1-6.

Proebsting, T. A. and Townsend, G. M. 2000. A new implementation of the Icon language. *Software: Practice and Experience 30,* 8, Wiley, 925-972.

Reddy, U. S. 1990. Formal methods in transformational derivation of programs. In *Proceedings on Formal Methods in Software Development.* ACM Press, New York, NY, 104-114.

Reis, A. J. D. 2011. *Compiler Construction Using Java, JavaCC, and Yacc.* Wiley-IEEE Computer Society Press.

Reppy, J.H. 1991. CML: A higher-order concurrent language. In *Proceedings of the 18th Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, 293-305.

Rossum, G. and Drake, F. L. 2011. *The Python Language Reference Manual.* Network Theory Ltd., Godalming, United Kingdom.

Schwartz, J. T., Dewar, R. B., Schonberg, E., and Dubinsky E. 1986. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York, NY.

Serbanuta, T. F., Rosu, G., and Meseguer, J. 2009. A Rewriting Logic Approach to Operational Semantics. *Information and Computation 207*, 2, Elsevier, 305-340.

Smolka, G. 1995. The Oz Programming Model. In *Computer Science Today*. Lecture Notes in Computer Science 1000, J. van Leeuwen (Ed.), Springer-Verlag, 324-343.

Spivey, J. M. 1992. *The Z Notation: A Reference Manual, 2nd Edition.* Prentice Hall International Series in Computer Science, Upper Saddle River, NJ. http://spivey.oriel.ox.ac.uk/mike/zrm.

Tratt, L. 2010. Experiences with an Icon-like expression evaluation system. In *Proceedings of the 6th Dynamic Languages Symposium*. ACM Press, New York, NY, 73-80.

Van Roy, P. (Ed.). 2005. Multiparadigm Programming in Mozart/Oz. *Second International Conference, MOZ 2004*. Lecture Notes in Computer Science 3389, Springer-Verlag.

Visser, E. 2005. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation, Special issue on Reduction Strategies in Rewriting and Programming* 40, 1, Elsevier, 831-873.

Wadler, P. 1990. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming.* ACM Press, New York, NY, 61-78.

Wadler, P. 1992. The essence of functional programming. In Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92). ACM Press, New York, NY, 1-14.

Walker, K. W. 1989. First-class patterns for Icon. *Computer Languages 14*, 3, Elsevier, 153-163.

Walker, K., and Griswold, R. 1992. An Optimizing Compiler for the Icon Programming Language. *Software: Practice and Experience 22*, 8, Wiley, 637-657.

Walls, C. 2011. *Spring in Action, 3rd Edition.* Manning Publications Co., Greenwich, CT.

Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., and Foster, I.T. 2011. Swift: A Language for Distributed Parallel Scripting. *Journal Parallel Computing 37*, 9, Elsevier Science Publishers, 633-652.

Wright, A. K. and Felleisen, M. 1994. A Syntactic Approach to Type Soundness. *Information and Computation 115*, 1, Elsevier, 38-94.

Appendix 1. Transformational Interpreter API

edu.uidaho.junicon.interpreter.interpreter

Interface IInterpreter

All Superinterfaces:

`IDispatcher`, `IEnvironment`, `IInterpreterContext`, `IInterpreterSetters`, `IPropertiesExtender`, `IThreadResolver`

All Known Implementing Classes:

`AbstractShell`, `CommandShell`, `JuniconInterpreter`, `TransformInterpreter`

```
public interface IInterpreter
extends IInterpreterContext, IInterpreterSetters, IDispatcher,
IEnvironment, IPropertiesExtender, IThreadResolver
```

Interface for transformational interpreters that parse, transform, and then execute statements on a scripting substrate. The steps involved in transformational interpretation are to handle any directives, preprocess each line of input, accumulate a complete statement and then parse it, decorate the parse tree, normalize it, test to filter it out for export, transform it, decide to whom to delegate it, and then either execute the transformed line of input using a scripting engine substrate or dispatch it to another sub-interpreter for further transformation.
The interface is common to and satisfied by both the outer interactive command shell and its sub-interpreters that map from sub-languages such as Unicon onto specific substrates such as Groovy via cross-translation. The command shell, or meta-interpreter, has the added responsibility to identify parsable units and their target language from lines of input, filter them, and then dispatch them to the appropriate sub-interpreter.
Each transformational interpreter thus has a notion of sub-interpreters as well as a substrate, onto one of which it may dispatch the transformed input for execution. Substrates differ from sub-interpreters in that they are implementation targets such as Groovy that need not obey the transformational interpreter interface.
The equational specification of the steps involved in transformational interpretation is given below in eval() and evalLine().

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method and Description
`IInterpreter`	`chooseDelegate(String transformedSource, ILineContext context)` Decide which sub-interpreter to delegate to.
`Document`	`decorate(String inputSource, Document parseTree, ILineContext context)` Decorate the parse tree with additional syntactic or semantic information.
`Object`	`dispatch(String transformedSource, ILineContext context)` Execute the transformed source code on a sub-interpreter or directly on a substrate.
`void`	`displayPrompt()` Displays the command prompt.
`Object`	`eval(String inputSource, ILineContext context)` Evaluate a parsable complete statement or set of statements.
`Object`	`evalLine(String inputLine, ILineContext context)` Evaluate all parsable complete statements found so far in the input line history.
`Object`	`exec(String inputSource, ILineContext context)` Evaluate a parsable complete statement.
`void`	`execFile(InputStream instream, ILineContext context, boolean promptflag)` Line-by-line evaluation of the input stream, extracting statements using the preprocessor and statement detector.
`void`	`execFile(InputStream instream, ILineContext context, boolean promptFlag, boolean stopOnScriptError,`

	`boolean aggregateInput)` Line-by-line evaluation of the input stream, extracting statements using the preprocessor and statement detector.
`boolean`	`execScript(String script, ILineContext context, boolean promptFlag, boolean stopOnScriptError)` Line-by-line evaluation of the input string, extracting statements using the preprocessor and statement detector.
`boolean`	`execScriptFile(String filename, boolean promptFlag, boolean stopOnScriptError)` Line-by-line evaluation of the input file, extracting statements using the preprocessor and statement detector.
`Object`	`executeOnSubstrate(String transformedSource, ILineContext context)` Execute the transformed source code on the substrate, if it exists.
`void`	`exit()` Tells the interpreter to exit.
`InterpreterException`	`filterException(Throwable cause)` Filter exceptions in the execFile interpretive loop.
`String`	`filterOut(String inputSource, Document parseTree, ILineContext context)` Test a statement based on its AST and source, and if needed take it out of the transform sequence and separately handle it.
`String`	`getParseUnit(String inputLine, ILineContext context)` Obtains a parsable unit from the history of lines in the input stream.
`ILineContext`	`getParseUnitContext()` Gets context of parse unit returned by last call to getParseUnit, or null if no parse unit returned on last call.
`String`	`handleBeginOfScript(String annotation)` Handle beginning of script.
`String`	`handleDirective(String line)` Handle a directive.

String	handleEndOfScript(String line) Handle end of script.
boolean	isClosedStatement() Returns true if the input so far to getParseUnit has matching group delimiters, but need not be terminated by a semicolon ";".
boolean	isDirective(String line) Test if an input line is a directive given by a scoped annotation of the form: @<tag attr=x attr'=y ...
boolean	isPartialStatement() Returns true if input so far to getParseUnit does not form a complete parsable statement.
Document	normalize(Document parseTree, ILineContext context) Reduce the parsed input to normal form.
String	normalize(String inputSource, Document normalized, ILineContext context) Reduce the input to normal form.
Document	parse(String sourceInput, ILineContext context) Parses the source input for the target substrate.
void	resetTransformed() Reset evaluation state.
void	setInit(String dummy) Initialize interpeter, after setters in dependency injection.
void	setRunInterpreter(String dummy) Run interpreter, if outer interactive shell.
String	transform(String inputSource, Document parseTree, ILineContext context) Perform the language-specific transformation from the input source and/or the parse tree.

Methods inherited from interface edu.uidaho.junicon.interpreter.interpreter.IInterpreterContext

getLogger, getName, getParent, getParser, getSubstrate, getType, setLogger, setName, setParent, setParser, setSubstrate, setType

Methods inherited from interface edu.uidaho.junicon.interpreter.interpreter. IInterpreterSetters

getCompileTransforms, getDefaultCompileTransforms, getDoNotDetect, getDoNotExecute, getDoNotPreprocess, getDoNotTransform, getEcho, getInheritVerbose, getIsInteractive, getIsInterpretive, getIsVerbose, getJustNormalize, getKillLineChar, getLineSeparator, getPartialPrompt, getPrompt, getResetParserOnError, getShowRawSubstrateErrors, getStopScriptOnError, getTransformSupport, setCompileTransforms, setDefaultCompileTransforms, setDoNotDetect, setDoNotExecute, setDoNotPreprocess, setDoNotTransform, setEcho, setInheritVerbose, setIsInteractive, setIsInterpretive, setIsVerbose, setJustNormalize, setKillLineChar, setPartialPrompt, setPrompt, setResetParserOnError, setShowRawSubstrateErrors, setStopScriptOnError, setTransformSupport

Methods inherited from interface edu.uidaho.junicon.interpreter.interpreter. IDispatcher

getDefaultDispatchInterpeter, getDispatchChildren, setAddDispatchChild, setDefaultDispatchInterpreter, setDefaultDispatchInterpreterByName, setDispatchChildren, setDispatchChildren

Methods inherited from interface edu.uidaho.junicon.interpreter.interpreter. IEnvironment

getDelegateEnvironment, getEnv, getEnv, getEnvironment, getEnvNames, setDelegateEnvironment, setEnv

Methods inherited from interface edu.uidaho.junicon.interpreter.interpreter.

IPropertiesExtender

`clearProperties, getDefaultDefaultProperties, getDefaultProperties, getProperties, getPropertiesDelegate, getProperty, getProperty, getPropertyNames, loadProperties, setAddProperties, setDefaultDefaultProperties, setDefaultProperties, setDefaultProperties, setDefaultPropertiesToSystem, setProperties, setPropertiesDelegate, setProperty, storeProperties`

Methods inherited from interface edu.uidaho.junicon.support.transforms.IThreadResolver

`getThreadResource`

Method Detail

setInit

```
void setInit(String dummy)
      throws InterpreterException
```

Initialize interpeter, after setters in dependency injection. Intended to be last setter in Spring dependency injection. The recommended usage scenario is as follows.

```
        main() { shell = Spring_bean_creation(XML lastly
calls setInit);
                 shell.setRunInterpreter(); }
```

Parameters:

`dummy - Ignored dummy parameter for Spring invocation.`

Throws:

`InterpreterException`

setRunInterpreter

```
void setRunInterpreter(String dummy)
                throws InterpreterException
```

Run interpreter, if outer interactive shell. Intended to be called after Spring bean creation and setInit.

Parameters:

`dummy - Ignored dummy parameter for Spring invocation.`

Throws:

`InterpreterException`

isDirective

```
boolean isDirective(String line)
```

Test if an input line is a directive given by a scoped annotation of the form: @<tag attr=x attr'=y ... > ... @</tag>, or just @<tag attr=x ... />. Directives are scoped annotations on a single line, and bypass all parsers if handled.

Scoped annotations are a hybrid of XML and Java annotations. Java style attributes such as @<tag (key=value, ...)> or @<tag (value)> can also be used, instead of XML style attributes. One can override this method so that scoped annotations might alternatively follow a different syntax such as @command(args).

Commented scoped annotations that begin with #@< are also allowed, and are treated the same way as scoped annotations.

handleDirective

```
String handleDirective(String line)
                throws InterpreterException
```

Handle a directive. By default this method returns the input line unchanged. It can be overriden for other behavior. For example, if a line in a file starts with @<script lang="...">, any file contents up to an ending @</script> tag are directly passed to the substrate interpreter, without preprocessing or transformation.

Returns:

`non-null text to continue processing, or null if directive was consumed.`

Throws:

`InterpreterException`

handleBeginOfScript

```
String handleBeginOfScript(String annotation)
```

Handle beginning of script. Invoked by handleDirective.

handleEndOfScript

```
String handleEndOfScript(String line)
```

Handle end of script. Invoked by handleDirective.

getParseUnit

```
String getParseUnit(String inputLine,
                    ILineContext context)
             throws ParseException
```

Obtains a parsable unit from the history of lines in the input stream. If a preprocessor has been set, this is used to first process the input line using its getParseUnit. The result is added to a history of preprocessed input lines. The next parsable unit, i.e. a complete statement, is then obtained from the preprocessed input history.

Parameters:

`inputLine` - a line of source input, without the line separator (e.g., carriage return), to add to the input line history.

`context` - input line context, passed to the preprocessor and carried through into the input history.

Returns:

parse unit, if the input line history has a parsable unit; null otherwise.

Throws:

`ParseException`

getParseUnitContext

```
ILineContext getParseUnitContext()
```

Gets context of parse unit returned by last call to getParseUnit, or null if no parse unit returned on last call.

isPartialStatement

```
boolean isPartialStatement()
```

Returns true if input so far to getParseUnit does not form a complete parsable statement. Complete statements have matching group delimiters "{}" "()" "[]" and are terminated by a semicolon ";".

isClosedStatement

```
boolean isClosedStatement()
```

Returns true if the input so far to getParseUnit has matching group delimiters, but need not be terminated by a semicolon ";".

parse

```
Document parse(String sourceInput,
               ILineContext context)
        throws ParseException
```

Parses the source input for the target substrate. The input must be a parsable complete statement.

Parameters:

`sourceInput - parsable unit of source input`

`context - input line context`

Returns:

`XML Abstract Syntax Tree (XML-AST), or null if sourceInput is null.`

Throws:

`ParseException`

decorate

```
Document decorate(String inputSource,
                  Document parseTree,
                  ILineContext context)
           throws InterpreterException
```

Decorate the parse tree with additional syntactic or semantic information.

Returns:

`parseTree if inputSource or parseTree is null, or if there is no transform.`

Throws:

`InterpreterException`

resetTransformed

```
void resetTransformed()
```

Reset evaluation state.

normalize

```
Document normalize(Document parseTree,
                   ILineContext context)
            throws InterpreterException
```

Reduce the parsed input to normal form.

Returns:

`parseTree if parseTree is null, or if there is no transform.`

Throws:

`InterpreterException`

normalize

```
String normalize(String inputSource,
                 Document normalized,
                 ILineContext context)
          throws InterpreterException
```

Reduce the input to normal form.

Returns:

`inputSource if inputSource is null, or if there is no transform.`

Throws:

`InterpreterException`

filterOut

```
String filterOut(String inputSource,
                 Document parseTree,
                 ILineContext context)
          throws InterpreterException
```

Test a statement based on its AST and source, and if needed take it out of the transform sequence and separately handle it. Filtering takes items out of the transform sequence, for example to export artifacts such as classes, interfaces, or webservices.

Returns:

`inputSource if should continue to process the transform stream, otherwise returns null if filtered it out.`

Throws:

`InterpreterException`

transform

```
String transform(String inputSource,
                 Document parseTree,
                 ILineContext context)
          throws InterpreterException
```

Perform the language-specific transformation from the input source and/or the parse tree.
This will typically be a transform chain with several stages: transforming the XML Abstract Syntax Tree (XML-AST) to another Document, and then deconstructing and formatting it into the specific target language.

Returns:

`inputSource if inputSource or parseTree is null, or if there is no transform.`

Throws:

`InterpreterException`

chooseDelegate

```
IInterpreter chooseDelegate(String transformedSource,
                                    ILineContext context)
```

Decide which sub-interpreter to delegate to.

Returns:

null if this interpreter should handle the command.

dispatch

```
Object dispatch(String transformedSource,
                ILineContext context)
         throws InterpreterException
```

Execute the transformed source code on a sub-interpreter or directly on a substrate. Sets currentInterpreter to this.
Equivalent to:

```
        if (null or this == chooseDelegate(source)) {
executeOnSubstrate(source);
        } else { chooseDelegate(source).evalLine(source); };
```

Parameters:

transformedSource - representing the transformed input source code.

Returns:

result of evaluation.

Throws:

InterpreterException

executeOnSubstrate

```
Object executeOnSubstrate(String transformedSource,
                          ILineContext context)
                   throws InterpreterException
```

Execute the transformed source code on the substrate, if it exists.

Parameters:

transformedSource - the transformed input source code.

Returns:

result of execution.

Throws:

`InterpreterException`

exec

```
Object exec(String inputSource,
            ILineContext context)
     throws InterpreterException
```

Evaluate a parsable complete statement. Synonymous with eval().

Returns:

`result of evaluation. Returns null on null input.`

Throws:

`InterpreterException`

eval

```
Object eval(String inputSource,
            ILineContext context)
     throws InterpreterException
```

Evaluate a parsable complete statement or set of statements. Evaluation does not use the preprocessor and statement detector. Eval() is stateful, and synonymous with exec(). Sets currentInterpreter to this.
Equivalent to:

```
        let normalized = normalize (decorate (input, parse
(input)));
        let normalizedText = normalize (input, normalized);
        if (null != filterOut (normalizedText, normalized))
{
                dispatch (transform (normalizedText,
normalized)) };
```

where dispatch is equivalent to:

```
        if (null or this == chooseDelegate(source)) {
                executeOnSubstrate(source);
        } else { chooseDelegate(source).evalLine(source); };
```

Returns:

```
result of evaluation. Returns null on null input (i.e., is
null-idempotent).
```

Throws:

`InterpreterException`

evalLine

```
Object evalLine(String inputLine,
                ILineContext context)
         throws InterpreterException
```

Evaluate all parsable complete statements found so far in the input line history. Evaluation uses the preprocessor and statement detector. Statements can thus span multiple lines and begin or end in mid-line. The statement detector is stateful in looking for parsable statements in the input passed to it.
Equivalent to:

```
        (eval (getParseUnit(isDirective(line)?
handleDirective(line):line)))*
```

where "*" means repeat until no more parseUnits returned.

Returns:

`result of evaluation.`

Throws:

`InterpreterException`

execFile

```
void execFile(InputStream instream,
              ILineContext context,
              boolean promptFlag,
              boolean stopOnScriptError,
              boolean aggregateInput)
       throws InterpreterException
```

Line-by-line evaluation of the input stream, extracting statements using the preprocessor and statement detector. Repeatedly parse, normalize, filter, transform and execute commands from the input stream.
Equivalent to:

```
        (displayPrompt; evalLine (readLine (instream)))*
                       until exit() or eof().
```

A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed. Thus, works with either Unix or Windows text file formats for both input script files and input from stdin.

Parameters:

`instream` - input stream containing source code.

`context` - file and line context

`promptFlag` - print a prompt before each getting each line of input.

`stopOnScriptError` - stop processing input lines if an error occurs.

`aggregateInput` - evaluate the whole file at once, not line by line.

Throws:

`InterpreterException`

execFile

```
void execFile(InputStream instream,
              ILineContext context,
              boolean promptflag)
       throws InterpreterException
```

Line-by-line evaluation of the input stream, extracting statements using the preprocessor and statement detector.
Prints a prompt if promptflag, and does not stop on error. Works with either Unix or Windows text file formats for both input script files and input from stdin.

Throws:

`InterpreterException`

filterException

```
InterpreterException filterException(Throwable cause)
```

Filter exceptions in the execFile interpretive loop. Default behavior is to print the error message and, if getIsDebug(), printstackTrace.

Returns:

`filtered exception.`

execScript

```
boolean execScript(String script,
                   ILineContext context,
                   boolean promptFlag,
                   boolean stopOnScriptError)
            throws InterpreterException
```

Line-by-line evaluation of the input string, extracting statements using the preprocessor and statement detector. Uses execFile to load and run the shell script from a string.

Throws:

`InterpreterException`

execScriptFile

```
boolean execScriptFile(String filename,
                       boolean promptFlag,
                       boolean stopOnScriptError)
                throws InterpreterException
```

Line-by-line evaluation of the input file, extracting statements using the preprocessor and statement detector. Uses execFile to load and run the shell script from a file.

Throws:

`InterpreterException`

exit

```
void exit()
```

Tells the interpreter to exit.

displayPrompt

```
void displayPrompt()
```

Displays the command prompt.

Appendix 2. Junicon LL(k) Grammar

```
/************************************************************************
 * Tokens
 ************************************************************************/

//=========================================================================
// Tokens: White space
//=========================================================================
<DEFAULT> SKIP : {
<WHITESPACE: " " | "\t" | "\f">
}

<DEFAULT> SPECIAL : {
<NEWLINE: "\n" | "\r" | "\r\n">
}

//=========================================================================
// Tokens: Comments
//=========================================================================
<DEFAULT> TOKEN : {
<ANNOTATION_COMMENT_PREFIX: "#@<">
| <#REST_OF_LINE: (~["\n","\r"])* ("\n" | "\r" | "\r\n")?>
}

<DEFAULT> SPECIAL : {
<SINGLE_LINE_COMMENT: "#" (~["\n","\r"])* ("\n" | "\r" | "\r\n")?>
}

//=========================================================================
// Tokens: Reserved words
//=========================================================================
<DEFAULT> TOKEN : {
<ABSTRACT: "abstract">
| <BREAK: "break">
| <BY: "by">
| <CASE: "case">
| <CLASS: "class">
| <CREATE: "create">
| <THREAD: "thread">
| <CRITICAL: "critical">
| <DEFAULT_token: "default">
| <DO: "do">
| <ELSE: "else">
| <END: "end">
| <EVERY: "every">
| <FAIL: "fail">
| <GLOBAL: "global">
| <IF: "if">
| <IMPORT: "import">
| <IN: "in">
| <INITIAL: "initial">
| <INITIALLY: "initially">
| <INVOCABLE: "invocable">
| <LINK: "link">
| <LOCAL: "local">
| <METHOD: "method">
| <NEW: "new">
| <NEXT: "next">
| <NOT: "not">
| <NULL_LITERAL: "null">
| <OF: "of">
| <PACKAGE: "package">
| <PROCEDURE: "procedure">
| <RECORD: "record">
| <REPEAT: "repeat">
| <RETURN: "return">
| <STATIC: "static">
| <SUSPEND: "suspend">
| <THEN: "then">
| <TO: "to">
| <UNTIL: "until">
| <WHILE: "while">
}

//============================================================================
// Tokens: Separators
//============================================================================
<DEFAULT> TOKEN : {
<LPAREN: "(">
| <RPAREN: ")">
| <LBRACE: "{">
| <RBRACE: "}">
| <LBRACKET: "[">
| <RBRACKET: "]">
| <SEMICOLON: ";">
| <COMMA: ",">
| <DOTDOTDOT: "...">
| <DOT: ".">
| <ANNOTATION: "@<">
| <AT: "@">
| <COMPR: "[:">
| <COMPREND: ":]">
| <COLON: ":">
| <COLONCOLON: "::">
| <LBRACE_EBCDIC: "$("> : {
| <RBRACE_EBCDIC: "$)"> : {
| <LBRACKET_EBCDIC: "$<"> : {
| <RBRACKET_EBCDIC: "$>"> : {
}

//============================================================================
// Tokens: Operators, grouped by precedence
//============================================================================
<DEFAULT> TOKEN : {
<AND: "&">
| <QMARK: "?">
| <ASSIGN: ":=">
| <REVASSIGN: "<-">
| <REVSWAP: "<->">
| <SWAP: ":=:">
| <PMATCH: "??">
| <POR: ".|">
| <BAR: "|">
| <PAND: "&&">
| <GT: ">">
| <LT: "<">
| <EQ: "==">
| <LE: "<=">
| <GE: ">=">
| <EQUALS: "=">
| <LSHIFT: "<<">
| <RSHIFT: ">>">
| <SLE: "<<=">
| <EQUIV: "===">
| <SGE: ">>=">
| <NMNE: "~=">
| <SNE: "~==">
| <NEQUIV: "~===">
| <PIMDASSN: "$$">
| <PASSNONMATCH: "->">
| <SND: "@>">
| <SNDBK: "@>>">
| <RCV: "<@">
| <RCVBK: "<<@">
| <CONCAT: "||">
| <LCONCAT: "|||">
| <PLUS: "+">
| <MINUS: "-">
| <INCR: "++">
| <DECR: "--">
| <STAR: "*">
| <SLASH: "/">
| <PERCENT: "%">
| <INTER: "**">
| <CARET: "^">
| <BANG: "!">
| <BACKSLASH: "\\">
| <TILDE: "~">
| <PSETCUR: ".$">
| <PIPE: "|>">
| <FUTURE: "|>>">
| <COEXPR: "|<>">
| <FIRSTCLASS: "<>">
| <PARALLEL: "|<>|">
| <BACKQUOTE: "`">
| <PCOLON: "+:">
| <MCOLON: "-:">
| <DOLLAR: "$">
| <AUGMOD: "%:=">
| <AUGAND: "&:=">
| <AUGSTAR: "*:=">
| <AUGINTER: "**:=">
| <AUGPLUS: "+:=">
| <AUGUNION: "++:=">
| <AUGMINUS: "-:=">
| <AUGDIFF: "--:=">
| <AUGSLASH: "/:=">
| <AUGNMLT: "<:=">
| <AUGSLT: "<<:=">
| <AUGSLE: "<<=:=">
| <AUGNMLE: "<=:=">
| <AUGNMEQ: "=:=">
| <AUGSEQ: "==:=">
| <AUGEQUIV: "===:=">
| <AUGNMGT: ">:=">
| <AUGNMGE: ">=:=">
| <AUGSGT: ">>:=">
| <AUGSGE: ">>=:=">
| <AUGQMARK: "?:=">
| <AUGAT: "@:=">
| <AUGCARET: "^:=">
| <AUGCONCAT: "||:=">
| <AUGNEQUIV: "~===:=">
| <AUGSNE: "~==:=">
| <AUGNMNE: "~=:=">
| <AUGLCONCAT: "|||:=">
}

//==========================================================================
// Tokens: Numeric literals
//==========================================================================
<DEFAULT> TOKEN : {
<INTEGER_LITERAL: <DECIMAL_LITERAL>>
| <RADIX_LITERAL: <DECIMAL_LITERAL> ["r","R"] (["0"-"9","a"-"f","A"-"F"])+>
| <#DECIMAL_LITERAL: (["0"-"9"])+>
| <REAL_LITERAL: (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? | "." (["0"-"9"])+ (<EXPONENT>)? |
                              (["0"-"9"])+ <EXPONENT>>
| <#EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+>
}

<DEFAULT> MORE : {
"\"" : WithinQuote
}

<WithinQuote> SKIP : {
<"_" <NEWLINE> (<WHITESPACE>)*> : WithinQuote
}

<WithinQuote> TOKEN : {
<STRING_LITERAL: "\""> : DEFAULT
}

<WithinQuote> MORE : {
<~["\"","\\"] | <ESCAPE_LITERAL>>
}

//===================
// Single quote -- multiline continuations are as for double quotes.
//===================
<DEFAULT> MORE : {
"\'" : WithinSingleQuote
```

```
}

<WithinSingleQuote> SKIP : {
<"_" <NEWLINE> (<WHITESPACE>)*> : WithinSingleQuote
}

<WithinSingleQuote> TOKEN : {
<SINGLE_QUOTE_LITERAL: "\'"> : DEFAULT
}

<WithinSingleQuote> MORE : {
<~["\'","\\"] | <ESCAPE_LITERAL>>
}

//===================
// Quote regex dependencies
//===================
<DEFAULT> TOKEN : {
<#OLD_SINGLE_QUOTE_LITERAL: "\'" (~["\'","\\"] | <ESCAPE_LITERAL>)* "\'">
| <#OLD_STRING_LITERAL: "\"" (~["\"","\\"] | <ESCAPE_LITERAL>)* "\"">
| <#ESCAPE_LITERAL: "\\" (<HEX_ESCAPE> | <OCTAL_ESCAPE> | <CONTROL_ESCAPE> | ["!"-
                              "~"," ","\t","\n","\r","\f"])>
| <#ESCAPE_LITERAL_NO_NL: "\\" (<HEX_ESCAPE> | <OCTAL_ESCAPE> | <CONTROL_ESCAPE>
                              | ["!"-"~"," ","\t","\f"])>
| <#HEX_ESCAPE: ["x","X"] <HEX_DIGITS>>
| <#OCTAL_ESCAPE: ["0"-"7"] ["0"-"7"]>
| <#CONTROL_ESCAPE: "^" ["!"-"~"]>
| <#HEX_DIGITS: ["0"-"9","a"-"f","A"-"F"]>
}

//=====================================================================
// Tokens: Big literals (Block quotes, or Big quotes)
//=====================================================================
<DEFAULT> TOKEN : {
<BIG_LITERAL: "{<" (~[">","}","\\"] | ">" ~["}","\\"] | ~[">","\\"] "}" | ">" <ESCAPE_LITERAL> |
                              <ESCAPE_LITERAL> ("}")?)* ([">","}"])? ">}">
}

//=====================================================================
// Tokens: Identifiers
//=====================================================================
<DEFAULT> TOKEN : {
<IDENTIFIER: <IDENTIFIER_BASE>>
| <#IDENTIFIER_BASE: <ALPHA> (<ALPHA> | <DIGIT>)*>
| <#ALPHA: ["A"-"Z","a"-"z","_"]>
| <#LETTER: ["$","A"-"Z","_","a"-"z","\u00c0"-"\u00d6","\u00d8"-"\u00f6","\u00f8"-"\u00ff","\u0100"-
                              "\u1fff","\u3040"-"\u318f","\u3300"-"\u337f","\u3400"-"\u3d2d","\u4e00"-
                              "\u9fff","\uf900"-"\ufaff"]>
| <#DIGIT: ["0"-"9","\u0660"-"\u0669","\u06f0"-"\u06f9","\u0966"-"\u096f","\u09e6"-"\u09ef","\u0a66"-
                              "\u0a6f","\u0ae6"-"\u0aef","\u0b66"-"\u0b6f","\u0be7"-"\u0bef","\u0c66"-
                              "\u0c6f","\u0ce6"-"\u0cef","\u0d66"-"\u0d6f","\u0e50"-"\u0e59","\u0ed0"-
                              "\u0ed9","\u1040"-"\u1049"]>
}
```

NON-TERMINALS

```
/***********************************************************************
 * LANGUAGE GRAMMAR STARTS HERE
 ***********************************************************************/

//=======================================================================
// Program structuring syntax follows.
//=======================================================================
ParseAndVisit          ::=    Start <EOF>
Start                  ::=    ( PackageDeclaration | ImportDeclaration | Link | Invocable | Record |
                       Global | ProcedureNew | Procedure | Method | ClassDeclarationNew |
                       ClassDeclaration | LocalDeclaration | BlockStatement | BlockAsExpr |
                       LambdaAsExpr )*

//====
// Junicon allows optional trailing semicolons in declarations.
//====
PackageDeclaration     ::=    ( ( PACKAGE Type | PACKAGE TypeStringLiteral ) ( SEMICOL )? )
ImportDeclaration      ::=    ( IMPORT ( NameOrStringList | TypeStringLiteral | ( STATIC )?
                       ImportName ) ( SEMICOL )? )
Link                   ::=    LINK NameOrStringList ( SEMICOL )?
Invocable              ::=    INVOCABLE InvocList ( SEMICOL )?
InvocList              ::=    Invocop ( COMMA Invocop )*
Invocop                ::=    ( TypeName | TypeStringLiteral COLON TypeIntLiteral |
                       TypeStringLiteral )
Directive              ::=    DOLLAR Identifier SpacedArgList ( SEMICOL )?
Record                 ::=    ( AnnotationType )? RECORD DeclaredName LPAREN ( ParamList )?
                       RPAREN ( SEMICOL )?
Global                 ::=    ( AnnotationType )? GLOBAL ( Type GlobalVarDeclList SEMICOL |
                       GlobalVarDeclList ( SEMICOL )? )

//=======================================================================
// Procedures.
//=======================================================================
Procedure              ::=    ProcedurePrefix SEMICOL ProcBody END ( SEMICOL )?
ProcedureNew           ::=    ProcedurePrefix ProcBodyNew ( SEMICOL )?
ProcedurePrefix        ::=    ( AnnotationType )? PROCEDURE DeclaredName LPAREN (
                       ParamList )? RPAREN
Initial                ::=    INITIAL Expr SEMICOL
ProcBody               ::=    ( AllEmptyBlock | BlockLocals ( Initial )? ( ( BlockStatement )+
                       TrailingEmptyBlock | TrailingEmptyBlock ) )
ProcBodyNew            ::=    ( LBRACE AllEmptyBlock RBRACE | LBRACE BlockLocals ( Initial
                       )? ( TrailingEmptyBlock | ExprSequence ) RBRACE )
LocalDeclaration       ::=    LocalDeclarationPrefix ( Type VarInitList | VarInitList ) SEMICOL
LocalDeclarationNoSemicolon    ::=    LocalDeclarationPrefix ( Type VarInitList | VarInitList )
LocalDeclarationPrefix ::=    ( AnnotationType )? ( LOCAL | STATIC )


//=======================================================================
// Classes.
//=======================================================================
ClassDeclaration       ::=    ClassPrefix ( SEMICOL )? ClassMethods END ( SEMICOL )?
ClassDeclarationNew    ::=    ClassPrefix LBRACE ClassMethods RBRACE ( SEMICOL )?
```

```
ClassPrefix             ::=     ( AnnotationType )? CLASS DeclaredName ( Supers )? LPAREN (
                        ParamList )? RPAREN
ClassMethods            ::=     ( Method | Global | Record | LocalDeclaration | EmbeddedScript )* (
                        SEMICOL )? ( InitiallyNew | Initially )?
Supers                  ::=     COLON Super ( COLON Super )*
Super                   ::=     MethodRefType
                        |       Type
Initially               ::=     InitiallyPrefix SEMICOL ProcBody
InitiallyNew            ::=     InitiallyPrefix ProcBodyNew ( SEMICOL )?
InitiallyPrefix         ::=     INITIALLY ( LPAREN ( ParamList )? RPAREN )?
Method                  ::=     ( AbstractMethod | RealMethodNew | RealMethod )
AbstractMethod          ::=     MethodPrefix ( SEMICOL )?
RealMethod              ::=     MethodPrefix SEMICOL ProcBody END ( SEMICOL )?
RealMethodNew           ::=     MethodPrefix ProcBodyNew ( SEMICOL )?
MethodPrefix            ::=     ( AnnotationType )? ( ABSTRACT )? ( MethodModifiers )? METHOD
                        DeclaredName LPAREN ( ParamList )? RPAREN
MethodModifiers         ::=     STATIC

//===========================================================================
// Declared names (variables and parameters), Types, and Atoms in expressions.
//===========================================================================
IdList                  ::=     DeclaredName ( COMMA DeclaredName )*
DeclaredName            ::=     Identifier
VarInitList             ::=     VariableInitializer ( COMMA VariableInitializer )*
VariableInitializer     ::=     ( LocalVariable ASSIGN Expr | LocalVariable )
LocalVariable           ::=     Identifier
GlobalVarDeclList       ::=     GlobalVariable ( COMMA GlobalVariable )*
GlobalVariable          ::=     Identifier
ParamList               ::=     ( Params LBRACKET_TYPE RBRACKET_TYPE | Params )
Params                  ::=     Param ( COMMA Param )*
Param                   ::=     ( JavaTypedParameter | ( DeclaredParameterName ( COLON Type )? ( (
                        COLON | EQUALS ) TypeLiteral )? ) )

//====
// Parameter can be param[:type][: or =literal]
//====
JavaTypedParameter      ::=     ParameterType DeclaredParameterName ( ( EQUALS ) TypeLiteral )?
DeclaredParameterName   ::=     Identifier
ParameterType           ::=     ( DotName | Identifier ) ( DOTDOTDOT )?
Type                    ::=     ( GenericType | DotName | Identifier )
GenericType             ::=     DotNameMaybe LANGLE_TUPLE TypeList RANGLE_TUPLE
TypeList                ::=     Type ( COMMA Type )*
TypeName                ::=     Identifier
MethodRefType           ::=     PackageRef
PackageRef              ::=     ( DotName COLONCOLON Identifier | SingleDotName
                        COLONCOLON Identifier | EmptyDotName COLONCOLON Identifier )
DotName                 ::=     Identifier ( DOT RelaxedIdentifier )+
SingleDotName           ::=     Identifier
EmptyDotName            ::=     AllEmptyAtom
DotNameMaybe            ::=     ( DotName | Identifier )
ImportName              ::=     ( ImportDotName | Identifier )
ImportDotName           ::=     Identifier ( DOT ( Identifier | STAR ) )+
SpacedArgList           ::=     ( Identifier | Literal )*
NameOrStringList        ::=     NameOrString ( COMMA NameOrString )*
NameOrString            ::=     TypeName
                        |       TypeStringLiteral
```

```
TypeLiteral             ::=     Literal
TypeStringLiteral       ::=     StringLiteral
TypeIntLiteral          ::=     IntLiteral

//=========================================================================
// Parameterized closure.
//=========================================================================
Closure                 ::=     LPAREN ( ParamList )? RPAREN ClosureOperator LBRACE
                        ExprSequenceAsBlock RBRACE
ExprSequenceAsBlock     ::=     ( AllEmptyBlock | BlockLocals ( TrailingEmptyBlock | ExprSequence )
                        )

//=========================================================================
// Block statements and bounded expressions.
//=========================================================================
Block                   ::=     LBRACE AllEmptyBlock RBRACE
                        |       LBRACE BlockLocals TrailingEmptyBlock RBRACE
                        |       LBRACE BlockLocals ExprSequence RBRACE
BlockLocals             ::=     ( LocalDeclaration )*
ExprSequence            ::=     ( InnerEmptyBlock | Nothing ) ( BlockExpr | Nothing ) ( SEMICOL
                        BlockExpr | SEMICOL InnerEmptyBlock )*

//====
// ExprSequence is to be non-empty, and ends with an expression or innerEmpty.
//                      If it would be empty, test and use allEmpty instead.
// Invariant: {AllEmpty} or {x;TrailingEmpty} or {x;y} where x may be InnerEmpty
//====
// We pad leading ";", trailing ";", and ";;" with an empty atom.
// We indicate an empty ExprSequence with AllEmpty().
// A trailing Empty() mimics Icon sequence semantics which ends with ;null.
// Transform must later detect last expr after semicolon to be unbounded.
//====
BlockAsExpr             ::=     BlockAsGroup
BlockAsGroup            ::=     Block
LambdaAsExpr            ::=     LambdaAsGroup
LambdaAsGroup           ::=     Closure
AllEmptyExpr            ::=     AllEmptyAtom
AllEmptyBlock           ::=
AllEmptyList            ::=
InnerEmptyBlock         ::=     InnerEmptyAtom
InnerEmptyList          ::=     InnerEmptyAtom
TrailingEmptyBlock      ::=     TrailingEmptyAtom
TrailingEmptyList       ::=     TrailingEmptyAtom
AllEmptyAtom            ::=
InnerEmptyAtom          ::=
TrailingEmptyAtom       ::=
BlockStatement          ::=     InnerEmptyBlock SEMICOL
                        |       BoundedExpr SEMICOL

//====
// Blockstatement ends with semicolon.
//====
BlockExpr               ::=     BoundedExpr
//====
// BlockExpr need not have semicolon after it.
//====
BoundedExpr              ::=      ( ( Annotation )+ PlainExpression | PlainExpression )
Expr                     ::=      ( ( Annotation )+ PlainExpression | PlainExpression )
PlainExpression          ::=      ( Statement | Command | AndExpression )
StatementExpression      ::=      Statement

//===========================================================================
// Control constructs.
//===========================================================================
Statement                ::=      ( If | Case | While | Until | Every | Repeat | Create | Thread | Next | Break |
                         Fail | Suspend | Return | Critical | Puneval )
StatementLookahead       ::=      ( IF | CASE | WHILE | UNTIL | EVERY | REPEAT | CREATE |
                         THREAD | NEXT | BREAK | FAIL | SUSPEND | RETURN | CRITICAL |
                         PUNEVAL )
If                       ::=      IF BoundedExpr THEN Expr ( ELSE Expr )?
Case                     ::=      CASE BoundedExpr OF LBRACE CaseList RBRACE
CaseList                 ::=      CaseItem ( SEMICOL CaseItem )*
CaseItem                 ::=      ( ( BoundedExpr COLON Expr ) | ( DEFAULT COLON Expr ) )
While                    ::=      WHILE BoundedExpr ( DO BoundedExpr )?
Until                    ::=      UNTIL BoundedExpr ( DO BoundedExpr )?
Every                    ::=      EVERY Expr ( DO BoundedExpr )?
Repeat                   ::=      REPEAT ( BoundedExpr )?
Create                   ::=      CREATE Expr
Thread                   ::=      THREAD Expr
Next                     ::=      NEXT
Break                    ::=      BREAK ( Expr | AllEmptyExpr )
Fail                     ::=      FAIL
Suspend                  ::=      SUSPEND ( SuspendExpr | AllEmptyExpr )
SuspendExpr              ::=      Expr ( DO BoundedExpr )?
Return                   ::=      RETURN ( BoundedExpr | AllEmptyExpr )
Critical                 ::=      CRITICAL BoundedExpr
Puneval                  ::=      PUNEVAL Expr
Command                  ::=      CommandNameAsAtom ( LimitedPrimaryExpression )+
CommandNameAsAtom        ::=      ( MethodRef | DotName | StrictIdentifier )
LimitedPrimaryExpression          ::=      ( LimitedObjectRef | LimitedInvokeExpr )
LimitedObjectRef         ::=      LimitedInvokeExpr DOT ( ( FieldExpression DOT )* FieldExpression )
LimitedInvokeExpr        ::=      SingleItem ( Arguments | ArraySuffix )*

//===========================================================================
// Expressions.
//===========================================================================
//====
// Cascade down for precedence, low to high.
// Higher precedence operators bind more tightly, i.e., are executed first.
//====
AndExpression            ::=      QmarkExpression ( AND QmarkExpression )*

//====
// Iterative production is left-associative.
//====
QmarkExpression          ::=      Assignment ( QMARK Assignment )*

//====
// Recursive production is right-associative.
//====
Assignment               ::=      FirstClassExpression ( ( AssignOperator | AugmentedAssign )
                         Assignment )?
```

```
//====
// Transform must detect Lhs of assign.
//====
FirstClassExpression      ::=    ( FirstClassOperators )? PmatchExpression
PmatchExpression          ::=    ToExpression ( PmatchOperators ToExpression )*

//====
// Operations are only allowed to the left of a statement, e.g., if.
// Anything to the right of a statement will bind together.
//====
ToExpression              ::=    PorExpression ( TO PorExpression ( BY PorExpression )? )?
PorExpression             ::=    BarExpression ( PorOperators BarExpression )*
BarExpression             ::=    CompareExpression ( BarOperators CompareExpression )*
CompareExpression         ::=    ConcatExpression ( CompareOperators ConcatExpression )*
ConcatExpression          ::=    AddExpression ( ConcatOperators AddExpression )*
AddExpression             ::=    MultiplyExpression ( AddOperators MultiplyExpression )*
MultiplyExpression        ::=    CaretExpression ( MultiplyOperators CaretExpression )*

//====
// CaretExpression is right associative via right recursion.
//====
CaretExpression           ::=    BangExpression ( CARET CaretExpression )?
BangExpression            ::=    UnaryExpression ( BangOperators UnaryExpression )*

//====
// UnaryExpression is right associative via right recursion.
//====
UnaryExpression           ::=    ( ( StatementExpression | ( UnaryOperators | UnaryBooleanOperators |
                          NOT ) UnaryExpression ) | PrimaryExpression )
//====
// Transform must detect that unary Not applies to boundedExpression()
//====

//=====================================================================
// Primary expressions.
//=====================================================================
PrimaryExpression         ::=    ( CastExpression | CastEmbeddedScript | PrimaryExpressionNoCast )
CastExpression            ::=    Cast PrimaryExpressionNoCast
CastEmbeddedScript        ::=    Cast EmbeddedScript
Cast                      ::=    LPAREN Type RPAREN
PrimaryExpressionNoCast::=       ( IteratorExpression | ObjectRef | InvokeExpr )
InvokeExpr                ::=    ObjectExpr ( Arguments | ArraySuffix )*

//====
// InvokeExpr includes invoke as well as standalone atom and group.
//====
ObjectExpr                ::=    ( AllocationExpr | MethodRefAsAtom | SingleItem | GroupExpr )
ObjectRef                 ::=    InvokeExpr DOT ( ( FieldExpression DOT )* FieldExpression )
FieldExpression           ::=    Field ( Arguments | ArraySuffix )*

//====
// ObjectExpr and FieldExpression in Java 8 use: [Arguments()] (ArraySuffix())*
//====
Field                     ::=    ( RelaxedIdentifier )
AllocationExpr            ::=    NEW Type
```

```
Arguments              ::=    ( Invoke | CoExprInvoke | SuperInvoke )
Invoke                 ::=    LPAREN_EMPTY AllEmptyList RPAREN
                       |      LPAREN ExprList RPAREN
CoExprInvoke           ::=    LBRACE_EMPTYSET AllEmptyList RBRACE_SET
                       |      LBRACE_SET ExprList RBRACE_SET
SuperInvoke            ::=    DOLLAR ( INITIALLY | ( Identifier ( DOT Identifier | INITIALLY )? )
                       ) ( LPAREN_EMPTY AllEmptyList RPAREN | LPAREN ExprList RPAREN )
ArraySuffix            ::=    LBRACKET_EMPTYINDEX AllEmptyList RBRACKET_INDEX
                       |      LBRACKET_INDEX ( RangeExpr | ExprList ) RBRACKET_INDEX
RangeExpr              ::=    Expr ( COLON | PCOLON | MCOLON ) Expr
SingleItem             ::=    ( Identifier | AndKeyword | Literal )
GroupExpr              ::=    ( Closure | Tuple | Map | EmptyMapComprEnd | Set | Block | CaseMap |
                       List | EmptyMapCompr | ListComprehension )
IteratorExpression     ::=    InIterator
ArgList                ::=    ExprList
ExprList               ::=    ( InnerEmptyList | Nothing ) ( Expr | Nothing ) ( COMMA Expr |
                       COMMA InnerEmptyList )*

//====
// Exprlist is to be non-empty, and ends with an expression or innerEmpty.
//                     If it would be empty, test and use allEmpty instead.
// Invariant: (AllEmpty) or (x,TrailingEmpty) or (x,y) where x may be Empty
//====
// We pad leading ",", trailing ",", and ",," with Empty().
// We indicate an empty ExprList with AllEmpty().
//====
RightDelim             ::=    RPAREN
                       |      RBRACE
                       |      RBRACKET
Nothing                ::=
MethodRef              ::=    ( DotNameAsAtom COLONCOLON RelaxedIdentifier |
                       SingleDotNameAsAtom COLONCOLON RelaxedIdentifier |
                       CastDotNameAsAtom COLONCOLON RelaxedIdentifier | EmptyDotName
                       COLONCOLON RelaxedIdentifier )

//====
// Methodref: x.y::f | ((Cast) x).y::f(z)
//====
CastDotNameAsAtom      ::=    CastInParen ( DOT Field )*
CastInParen            ::=    LPAREN SimpleCastExpression RPAREN
SimpleCastExpression   ::=    Cast IdentifierAsAtom
DotNameAsAtom          ::=    IdentifierAsAtom ( DOT Field )+
SingleDotNameAsAtom    ::=    IdentifierAsAtom
IdentifierAsAtom       ::=    Identifier
MethodRefAsAtom        ::=    MethodRef

//====================================================================
// Tuple, map, list, and set.
//====================================================================
Tuple                  ::=    LPAREN_EMPTY AllEmptyList RPAREN
                       |      LPAREN ExprList RPAREN
List                   ::=    LBRACKET_EMPTYLIST AllEmptyList RBRACKET
                       |      LBRACKET ExprList RBRACKET
CaseMap                ::=    LBRACKET_MAP CaseList RBRACKET_MAP
Map                    ::=    LBRACKET_EMPTYMAP COLON RBRACKET
                       |      LBRACKET_MAP MapList RBRACKET_MAP
```

```
MapList                ::=     KeyColonValue ( COMMA KeyColonValue )*
KeyColonValue          ::=     Expr COLON Expr
EmptyMapCompr          ::=     COMPR_EMPTYMAP RBRACKET_MAP
EmptyMapComprEnd       ::=     LBRACKET_EMPTYMAP COMPREND
Set                    ::=     LBRACE_SET ExprList RBRACE_SET
ListComprehension      ::=     COMPR_EMPTY AllEmptyList COMPREND
                       |       COMPR Expr COMPREND

//===========================================================================
// Iterators.
//===========================================================================
InIterator             ::=     ( LPAREN COLON IN Expr RPAREN | LPAREN IdentifierAsExpr IN
                       Expr RPAREN )
IdentifierAsExpr       ::=     IdentifierAsAtom
LiteralAsAtom          ::=     Literal
BigLiteralAsAtom       ::=     BigLiteral
BigLiteralAsExpr       ::=     BigLiteralAsAtom

//===========================================================================
// Annotations.
//===========================================================================
AnnotationType         ::=     Annotation
AnnotationComment      ::=     ( ANNOTATION_COMMENT_PREFIX SLASH XmlTag GT | XmlTag
                       ( SpacedNamedArgList )? ( SLASH )? GT )
Annotation             ::=     ( ANNOTATION SLASH XmlTag GT | ANNOTATION XmlTag
                       LPAREN ( NamedArgList | AnnotationExpr )? RPAREN ( SLASH )? GT |
                       ANNOTATION XmlTag ( SpacedNamedArgList )? ( SLASH )? GT )
XmlTag                 ::=     RelaxedDotNameMaybe ( COLON RelaxedDotNameMaybe )?
RelaxedDotNameMaybe    ::=     RelaxedIdentifier ( DOT RelaxedIdentifier )*
SpacedNamedArgList     ::=     NamedArg ( NamedArg )*
NamedArgList           ::=     NamedArg ( COMMA NamedArg )*
NamedArg               ::=     Identifier EQUALS AnnotationExpr
AnnotationExpr         ::=     Expr
EmbeddedScript         ::=     AnnotationType BigLiteralAsExpr

//===========================================================================
// Grouped literals and keywords
//===========================================================================
Literal                ::=     IntLiteral
                       |       RealLiteral
                       |       StringLiteral
                       |       CharLiteral
                       |       BigLiteral
                       |       NullLiteral
AndKeyword             ::=     AND ( Identifier | Fail | NullLiteral )

/*********************************************************************
 * TERMINALS START HERE
 *********************************************************************/

//===========================================================================
// Terminals: Reserved Words
//===========================================================================
ABSTRACT               ::=     <ABSTRACT>
BREAK                  ::=     <BREAK>
BY                     ::=     <BY>
```

```
CASE                ::=     <CASE>
CLASS               ::=     <CLASS>
CREATE              ::=     <CREATE>
THREAD              ::=     <THREAD>
CRITICAL            ::=     <CRITICAL>
DEFAULT             ::=     <DEFAULT_token>
DO                  ::=     <DO>
ELSE                ::=     <ELSE>
END                 ::=     <END>
EVERY               ::=     <EVERY>
FAIL                ::=     <FAIL>
GLOBAL              ::=     <GLOBAL>
IF                  ::=     <IF>
IMPORT              ::=     <IMPORT>
IN                  ::=     <IN>
INITIAL             ::=     <INITIAL>
INITIALLY           ::=     <INITIALLY>
INVOCABLE           ::=     <INVOCABLE>
LINK                ::=     <LINK>
LOCAL               ::=     <LOCAL>
METHOD              ::=     <METHOD>
NEW                 ::=     <NEW>
NEXT                ::=     <NEXT>
NOT                 ::=     <NOT>
OF                  ::=     <OF>
PACKAGE             ::=     <PACKAGE>
PROCEDURE           ::=     <PROCEDURE>
RECORD              ::=     <RECORD>
REPEAT              ::=     <REPEAT>
RETURN              ::=     <RETURN>
STATIC              ::=     <STATIC>
SUSPEND             ::=     <SUSPEND>
THEN                ::=     <THEN>
TO                  ::=     <TO>
UNTIL               ::=     <UNTIL>
WHILE               ::=     <WHILE>

//=====================================================================
// Terminals: Delimiters
//=====================================================================
LPAREN              ::=     <LPAREN>
RPAREN              ::=     <RPAREN>
LBRACE              ::=     <LBRACE>
RBRACE              ::=     <RBRACE>
LBRACE_CLOSURE      ::=     <LBRACE>
RBRACE_CLOSURE      ::=     <RBRACE>
LBRACE_SET          ::=     <LBRACE>
RBRACE_SET          ::=     <RBRACE>
LBRACKET            ::=     <LBRACKET>
RBRACKET            ::=     <RBRACKET>
LBRACKET_INDEX      ::=     <LBRACKET>
RBRACKET_INDEX      ::=     <RBRACKET>
LBRACKET_MAP        ::=     <LBRACKET>
RBRACKET_MAP        ::=     <RBRACKET>
LBRACKET_TYPE       ::=     <LBRACKET>
RBRACKET_TYPE       ::=     <RBRACKET>
```

```
LANGLE_TUPLE          ::=     <LT>
RANGLE_TUPLE          ::=     <GT>
COMPR                 ::=     <COMPR>
COMPREND              ::=     <COMPREND>
LPAREN_EMPTY          ::=     <LPAREN>
LBRACE_EMPTYSET       ::=     <LBRACE>
LBRACKET_EMPTYINDEX       ::=     <LBRACKET>
LBRACKET_EMPTYLIST        ::=     <LBRACKET>
LBRACKET_EMPTYMAP         ::=     <LBRACKET>
COMPR_EMPTYMAP        ::=     <COMPR>
COMPR_EMPTY           ::=     <COMPR>
SEMICOL               ::=     <SEMICOLON>
COLON                 ::=     <COLON>
COLONCOLON            ::=     <COLONCOLON>
COMMA                 ::=     <COMMA>
DOT                   ::=     <DOT>
DOTDOTDOT             ::=     <DOTDOTDOT>

//==============================================================================
// Terminals: Operators
//==============================================================================
AND                   ::=     <AND>
QMARK                 ::=     <QMARK>
ASSIGN                ::=     ( <ASSIGN> )
AssignOperator        ::=     ( <ASSIGN> | <REVASSIGN> | <REVSWAP> | <SWAP> )
AugmentedAssign       ::=     ( <AUGMOD> | <AUGAND> | <AUGSTAR> | <AUGINTER> |
                      <AUGPLUS> | <AUGUNION> | <AUGMINUS> | <AUGDIFF> |
                      <AUGSLASH> | <AUGNMLT> | <AUGSLT> | <AUGSLE> | <AUGNMLE> |
                      <AUGNMEQ> | <AUGSEQ> | <AUGEQUIV> | <AUGNMGT> |
                      <AUGNMGE> | <AUGSGT> | <AUGSGE> | <AUGQMARK> | <AUGAT> |
                      <AUGCARET> | <AUGCONCAT> | <AUGNEQUIV> | <AUGSNE> |
                      <AUGNMNE> | <AUGLCONCAT> )
PmatchOperators       ::=     <PMATCH>
PorOperators          ::=     <POR>
BarOperators          ::=     ( <BAR> | <PAND> )
ClosureOperator       ::=     ( <PASSNONMATCH> )
CompareOperators      ::=     ( <GT> | <LT> | <EQ> | <LE> | <GE> | <EQUALS> | <LSHIFT> |
                      <RSHIFT> | <SLE> | <EQUIV> | <SGE> | <NMNE> | <SNE> | <NEQUIV> |
                      <PIMDASSN> | <PASSNONMATCH> )
ConcatOperators       ::=     ( <CONCAT> | <LCONCAT> )
AddOperators          ::=     ( <PLUS> | <MINUS> | <INCR> | <DECR> )
MultiplyOperators     ::=     ( <STAR> | <SLASH> | <PERCENT> | <INTER> )
CARET                 ::=     <CARET>
BangOperators         ::=     ( <BANG> | <BACKSLASH> | <AT> | <SND> | <SNDBK> )
FirstClassOperators   ::=     ( <FIRSTCLASS> | <FUTURE> | <PIPE> | <COEXPR> |
                      <PARALLEL> )
UnaryOperators        ::=     ( <TILDE> | <PSETCUR> | <AT> | <BAR> | <CONCAT> |
                      <LCONCAT> | <DOT> | <BANG> | <DECR> | <PLUS> | <STAR> | <CARET>
                      | <INTER> | <MINUS> | <EQUALS> | <NMNE> | <EQ> | <RCV> | <RCVBK> |
                      <SNE> | <EQUIV> | <INCR> | <QMARK> | <NEQUIV> )
UnaryBooleanOperators ::=     ( <SLASH> | <BACKSLASH> )
PCOLON                ::=     <PCOLON>
MCOLON                ::=     <MCOLON>
DOLLAR                ::=     <DOLLAR>
AT                    ::=     <AT>
LT                    ::=     <LT>
```

```
GT                      ::=     <GT>
SLASH                   ::=     <SLASH>
STAR                    ::=     <STAR>
EQUALS                  ::=     <EQUALS>
PUNEVAL                 ::=     <BACKQUOTE>

//=========================================================================
// Terminals: Identifiers
//=========================================================================

Identifier              ::=     ( <IDENTIFIER> | <NEW> | <IN> )
StrictIdentifier        ::=     ( <IDENTIFIER> )
RelaxedIdentifier       ::=     ( <IDENTIFIER> | <IN> | <NEXT> | <THREAD> | <INITIALLY> )

//=========================================================================
// Terminals: Literals
//=========================================================================
BigLiteral              ::=     <BIG_LITERAL>
IntLiteral              ::=     ( <INTEGER_LITERAL> | <RADIX_LITERAL> )
RealLiteral             ::=     <REAL_LITERAL>
CharLiteral             ::=     <SINGLE_QUOTE_LITERAL>
StringLiteral           ::=     <STRING_LITERAL>
NullLiteral             ::=     <NULL_LITERAL>

//=========================================================================
// Terminals: Annotations
//=========================================================================
ANNOTATION              ::=     <ANNOTATION>
ANNOTATION_COMMENT_PREFIX       ::=     <ANNOTATION_COMMENT_PREFIX>
```

Appendix 3. Spring Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:lang="http://www.springframework.org/schema/lang" xsi:schemaLocation="
http://www.springframework.org/schema/beans
classpath:org/springframework/beans/factory/xml/spring-beans-3.2.xsd
http://www.springframework.org/schema/aop
classpath:org/springframework/aop/config/spring-aop-3.2.xsd">
  <!--

    #================================================
    # Outer command shell
    #================================================
  -->
  <bean id="CommandShell"
  class="edu.uidaho.junicon.interpreter.interpreter.CommandShell">
    <!--
    #====
    # Dependency injection
    #====  -->
    <property name="name" value="Shell"/>
    <property name="type" value="CommandShell"/>
    <property name="parent">
      <null/>
    </property>
    <property name="parser" ref="StatementDetectorParser"/>
    <property name="substrate">
      <null/>
    </property>
    <property name="logger" ref="DefaultLogger"/>
    <!--
    #====
    # Dispatcher to Groovy sub-interpreter
    #====  -->
    <property name="addDispatchChild" ref="JuniconInterpreter"/>
    <property name="defaultDispatchInterpreter" ref="JuniconInterpreter"/>
    <!--
    #====
    # Base properties
    #====  -->
    <property name="prompt" value=">>> "/>
    <property name="partialPrompt" value="... "/>
    <property name="defaultCompileTransforms" value="false"/>
    <property name="compileTransforms" value="false"/>
    <property name="showRawSubstrateErrors" value="true"/>
    <!--
    #====
    # Shell properties
    #====  -->
    <property name="isInteractive" value="false"/>
    <property name="ignoreSystemStartup" value="false"/>
    <property name="echo" value="false"/>
    <property name="echoStartup" value="false"/>
    <property name="echoSystemStartup" value="false"/>
    <property name="stopScriptOnError" value="false"/>
    <property name="exitOnScriptError" value="false"/>
    <property name="resetParserOnError" value="true"/>
    <property name="usage">
      <value>#{inputUsage.contents}</value>
    </property>
    <property name="appWindows">
      <value>#{inputAppWindows.contents}</value>
    </property>
    <property name="appLinux">
      <value>#{inputAppLinux.contents}</value>
    </property>
    <property name="javaPreface">
      <value>#{inputJavaPreface.contents}</value>
    </property>
    <property name="startupScripts">
      <list>
        <value>#{inputPrelude.contents}</value>
        <value>#{inputStartupScript.contents}</value>
      </list>
    </property>
    <property name="startupScriptNames">
      <list>
        <value>#{inputPrelude.filename}</value>
        <value>#{inputStartupScript.filename}</value>
      </list>
    </property>
  </bean>
  <!--

    #================================================
    # Logger
    #================================================
  -->
  <bean id="DefaultLogger" class="edu.uidaho.junicon.runtime.util.LoggerFactory">
    <property name="defaultPrintStream" value="#{T(java.lang.System).out}"/>
    <property name="isDebug" value="false"/>
    <property name="isTrace" value="false"/>
  </bean>
  <!--

    #================================================
    # Parsers: QuoteDetector => Preprocessor => StatementDetector => Grammar
    #================================================
  -->
  <bean id="StatementDetectorParser"
  class="edu.uidaho.junicon.interpreter.parser.ParserFromPreprocessor">
    <property name="name" value="StatementDetector"/>
    <property name="type" value="StatementDetector"/>
    <property name="parent" ref="CommandShell"/>
    <property name="preprocessor" ref="Preprocessor"/>
    <property name="parserWrapper" ref="StatementDetector"/>
  </bean>
  <bean id="StatementDetector"
  class="edu.uidaho.junicon.interpreter.parser.StatementDetector">
    <property name="quoteOnlyMode" value="false"/>
    <property name="preserveEmptyLines" value="true"/>
    <property name="allowPoundComment" value="true"/>
    <property name="allowSlashComment" value="true"/>
    <property name="allowMultilineComment" value="false"/>
    <property name="allowBlockQuote" value="true"/>
    <property name="allowPatternLiteral" value="false"/>
    <property name="allowOntologyLiteral" value="false"/>
    <property name="allowCommandLiteral" value="false"/>
    <property name="allowEscapedStartQuote" value="false"/>
    <property name="allowEscapedEndQuote" value="true"/>
    <property name="allowEscapedBlockQuote" value="false"/>
    <property name="allowEscapedComment" value="false"/>
    <property name="allowEscapedNewline" value="false"/>
    <property name="allowEscapedNewlineInQuote" value="false"/>
    <property name="allowEscapedNewlineInBlockQuote" value="true"/>
    <property name="splitMultipleStatementsOnLine" value="true"/>
  </bean>
  <bean id="Preprocessor"
  class="edu.uidaho.junicon.interpreter.parser.ParserFromPreprocessor">
    <property name="name" value="Preprocessor"/>
    <property name="type" value="Preprocessor"/>
    <property name="parent" ref="CommandShell"/>
    <property name="parentParser" ref="StatementDetectorParser"/>
    <property name="preprocessor" ref="QuoteMetaParser"/>
    <property name="parserWrapper" ref="JuniconPreprocessor"/>
  </bean>
  <bean id="JuniconPreprocessor"
  class="edu.uidaho.junicon.grammars.junicon.JuniconPreprocessor">
    <property name="relaxedUniconSyntax" value="true"/>
    <property name="doSemicolonInsertion" value="true"/>
  </bean>
  <bean id="QuoteMetaParser"
  class="edu.uidaho.junicon.interpreter.parser.ParserFromPreprocessor">
    <property name="name" value="QuoteDetector"/>
    <property name="type" value="StatementDetector"/>
    <property name="parent" ref="CommandShell"/>
    <property name="parentParser" ref="Preprocessor"/>
    <property name="parserWrapper" ref="QuoteDetector"/>
  </bean>
  <bean id="QuoteDetector"
  class="edu.uidaho.junicon.interpreter.parser.StatementDetector">
    <property name="quoteOnlyMode" value="true"/>
    <property name="preserveEmptyLines" value="true"/>
    <property name="allowPoundComment" value="true"/>
    <property name="allowSlashComment" value="false"/>
    <property name="allowMultilineComment" value="false"/>
    <property name="allowEscapedStartQuote" value="false"/>
    <property name="allowEscapedEndQuote" value="true"/>
    <property name="allowEscapedBlockQuote" value="false"/>
    <property name="allowEscapedComment" value="false"/>
    <property name="allowEscapedNewline" value="false"/>
    <property name="allowEscapedNewlineInQuote" value="false"/>
    <property name="allowEscapedNewlineInBlockQuote" value="true"/>
  </bean>
  <bean id="JuniconParser"
  class="edu.uidaho.junicon.interpreter.parser.ParserFromGrammar">
    <property name="name" value="JuniconParser"/>
    <property name="type" value="JuniconParser"/>
    <property name="parent" ref="JuniconInterpreter"/>
    <property name="parserWrapper" ref="JuniconJavaccParser"/>
  </bean>
  <bean id="JuniconJavaccParser" class="edu.uidaho.junicon.grammars.junicon.ParserBase">
  </bean>
  <!--

    #================================================
    # Numeric precision and index origin.   These can be turned on two ways.
    # 1. Transformation to Groovy.  Directives are as follows.
    #               @<index origin="1">
    #          Index origin is used for index operations c[i].
    #          If index origin is specified using a directive it is
    #          fixed at transformation, i.e., syntactially hardcoded as a setter.
    #     Otherwise it defaults at runtime to IconNumber.getIndexOrigin().
    # 2. Runtime of Java, after translation.  Setters are as follows.
    #               IconNumber.setIsIntegerPrecision(true);
    #               IconNumber.setIsRealPrecision(false);
    #     For translation to Java, numeric literals are surrounded by
    #     IconNumber calls that coerce them to BigInteger or BigDecimal if
    #     isIntegerPrecision or isRealPrecision() are on, respectively.
    #     Operators also use arbitrary precision in converting strings to
    #          numbers, and in their numeric results, if the above setters are on.
    #          Index origin is used for index and string operations.
    #               IconNumber.setIndexOrigin(1);
    #          These defaults are initially set in IconNumber from System properties
    #               junicon.isIntegerPrecision "true"
    #               junicon.isRealPrecision "false"
    #               junicon.indexOrigin "1".
    #          Java code thus can at runtime dynamically turn arbitrary
    #     precision fully off or on using the above System properties.
    # The above directives and setters can be set in this Spring file,
    #          in source code, or in the startup file.
    #================================================
  -->
  <bean id="NumberPrecision"
  class="edu.uidaho.junicon.runtime.junicon.iterators.IconNumber">
    <property name="defaultIsIntegerPrecision" value="true"/>
    <property name="defaultIsRealPrecision" value="false"/>
    <property name="defaultIndexOrigin" value="1"/>
  </bean>
  <!--

    #================================================
    # Junicon to Groovy interpreter
    #================================================
  -->
  <bean id="JuniconInterpreter"
  class="edu.uidaho.junicon.substrates.junicon.JuniconInterpreter">
    <!--
    #====
    # Dependency injection
    #====  -->
    <property name="name" value="Junicon"/>
    <property name="type" value="JuniconIntepreter"/>
    <property name="parent" ref="CommandShell"/>
    <property name="parser" ref="JuniconParser"/>
    <property name="substrate" ref="DefaultSubstrate"/>
    <property name="logger" ref="DefaultLogger"/>
    <!--
    #====
    # Delegate properties to CommandShell.
    #====  -->
    <property name="propertiesDelegate" ref="CommandShell"/>
    <!--

      #====
      # Junicon properties.
      # Interpreter properties inherit System.properties
      #       as well as any properties set here, and are used in transforms.
      #====
    -->
    <property name="addProperties">
      <props>
        <!--
           Force source code to have explicit origin hardcoded.
```

```
                 <prop key="index.origin">1</prop>

    -->
  </props>
</property>
<!--

  #====
  # Create transform support, i.e., methods used in XSLT.
  #====
-->
<property name="transformSupport" ref="TransformSupport"/>
<!--
#====
# Base properties
#====  -->
<property name="prompt" value=">>> "/>
<property name="partialPrompt" value="... "/>
<!--

          <property name="defaultCompileTransforms" value="false"/>

-->
<!--
#====
# Transforms
#====  -->
<property name="compileTransforms" value="false"/>
<property name="showRawSubstrateErrors" value="true"/>
<!--

  #====
  # Correlate source : concrete syntax nodes
  #====
-->
<property name="concreteSyntaxNodes">
  <list>
    <value>IDENTIFIER</value>
    <value>LITERAL</value>
    <value>KEYWORD</value>
    <value>OPERATOR</value>
    <value>DELIMITER</value>
  </list>
</property>
<!--
#====
# Transforms
#====  -->
<property name="normalizeTransform" value="#{inputNormalize.contents}"/>
<property name="codeTransform" value="#{inputMainTransform.contents}"/>
<property name="deconstructTransform" value="#{inputDeconstruct.contents}"/>
<property name="formatTransform" value="#{inputFormat.contents}"/>
<property name="correlateFormatTransform" value="#{inputFormatCorrelate.contents}"/>
<property name="normalizeFormatTransform" value="#{inputFormatNormalize.contents}"/>
<property name="exportTransform" value="#{inputExport.contents}"/>
<!--
#====
# Debugging
#====  -->
<property name="doNotExecute" value="false"/>
<!--  #====
     # Initialization
```

```
        #====  -->
  <property name="init">
    <null/>
  </property>
</bean>
<!--

  #================================================
  # Transform supporting methods.
  #================================================
-->
<bean id="TransformSupport"
class="edu.uidaho.junicon.support.transforms.TransformSupport">
  <!--

        #====
        # Catalog of named property maps, used to configure transforms.
        #====

  -->
  <property name="catalog">
    <map>
      <!--

        #====
        # Properties: indexOrigin.
        # Properties delegate to JuniconInterpreter
        #       and from there to CommandShell.
        # CommandShell properties are set from directives, e.g.
        #       @<index origin="0"/>
        # The directives translate into dot-separated properties:
        #       index.origin
        # When an open directive, e.g. @<index origin="0">, is encountered,
        #       it saves all properties prefixed with "index", clears them,
        #       and then sets properties from the attributes.
        # When a close directive, e.g. @</index> is encountered, it
        #       clears all properties prefixed with "index", and restores them
        #       from the saved values.
        #====
        # Index origin may be either 0 or 1.
        # For index origin 1, applies Icon rules for slicing c[i..e], which
        # extend up to but not including the end index,
        # and ignore reversal if the begin is after the end.
        # Otherwise, Groovy rules for slicing apply which includes the end index
        # and reverses results if the begin is after the end.
        # In both cases subscripting is from the end of the list if < 0.
        #====
      -->
      <entry key="Properties" value="#{CommandShell.properties}"/>
      <!--

        #====
        # Map symbols to operations over iterators using
        #   OperatorOverIterators Unary/Binary, OperatorOverAtoms Unary/Binary,
        #   OperatorAsGenerator, OperatorAsFunction, FunctionOverAtoms,
        #   SymbolAsIterator, SymbolAsProperty, SymbolAsVariable, SymbolAsValue,
        #   UndoFunction.
        #
        # Symbols include operators such as +, control constructs such as "if",
        #   and keywords such as &amp;features.
        #
        # UndoFunction maps symbols, both operators and function calls,
        #       to undo actions, and flags the symbol as undoable.
        #
        # Evaluation order is as follows.   If symbol is in:
        #   OperatorAsGenerator, treats the symbol as a generator over its
        #       atom arguments.  It is treated like a function by normalization
        #       to flatten its arguments into atoms, and then translated to
        #       an iterator constructor. Thus, the operator, first changed to a
        #       synthetic function, is changed back to an iterator over atoms.
        #       For example, !x is changed to !(x), its arguments normalized,
        #       and finally !(x) is changed back to "new IconPromote(x)".
        #   OperatorAsFunction, treats the symbol as a generator
        #       function over values or atoms that returns an iterator.
        #       The symbol is first translated into a function invocation.
        #       Normalization then flattens the arguments into atoms and makes
        #       iteration explicit, as it does for any function invocation.
        #       For example, &features would be changed to &features(x),
        #       normalized, and then mapped to:
        #       new IconInvokeIterator({-> IconKeywords.features(x)}).
        #   FunctionOverAtoms, treats the function name as over atoms.
        #       Recommended to use with built-in functions only,
        #       or with OperatorAsFunction symbols.
        #   OperatorOverIterators Unary/Binary, treats the symbol as
        #       a composition over iterators that returns an iterator. It
        #       applies the operation directly to (x) or (x,y,...) respectively,
        #       where looks in Unary if there is 1 operand and Binary otherwise.
        #       Constructs such as "if x then y" would be changed to if(x,y).
        #   OperatorOverAtoms Unary/Binary, treats the symbol as
        #       an operator over atoms that returns an atom.
        #       Promotes the operation to an iterator as:
        #       new IconOperation(UnaryOperation).over(x) or
        #               (BinaryOperation).over(x,y,...),
        #       where looks in Unary if there is 1 operand and Binary otherwise.
        #   SymbolAsIterator, treats &keyword as a field holding an iterator.
        #       This is the default used for &keywords.
        #   SymbolAsVariable, treats &keyword as an object reference to a
        #       static field in a single class.
        #   SymbolAsProperty, treats &keyword as a property with get() and set()
        #       methods, i.e., an atom that implements IIconAtom.
        #       For example: &subject => IconKeywords.subject
        #   SymbolAsValue, treats &keyword as a literal value.
        #   Default is:
        #       if operation, new IconOperation(default operator).over(x,y,...)
        #       if control construct, default is to capitalize: IconIf(x,y)
        #       if keyword, new IconNullIterator()
        #   where default operator is:
        #               ($x,$y)->$x $op $y
        #         or    ($x) -> op $x
        # Substitutes: $x, $y, $op for unique x and y and operator symbol.
        # IconOperatorIterator assumes operator is left-associative, and
        #       automatically translates variadic (x+y+z) to (x+y)+z
        #       to decompose construct into binary operations.
        #       Other OperatorOverIterators must handle variadic or fixed args.
        # Augmented assignment uses only OperatorOverAtoms for its operator:
        #       if not found there, uses default operator.
        #       Assignment is right-associative and recursively decomposed by
        #       grammar into only binary operations, so augments will be binary.
        # EXAMPLE of OverAtoms:     IconOperators.plus, where
        #       static plus = IconOperator.binary((x,y) -> x + y)
        #               or IconOperator.unary((x) -> + x)
        # EXAMPLE of OverIterators: new IconOperatorIterator(plus).over
        #       which is then invoked with "...over(x,y)"
        #====
      -->
      <entry key="OperatorOverIteratorsBinary">
        <props>
          <!--  x | y     # Concatenation  -->
          <prop key="|">new IconConcat</prop>
          <!--  x\limit   # Limit iteration  -->
          <prop key="\">new IconLimit</prop>
          <!--  s?e       # String scanning  -->
          <prop key="?">new IconScan</prop>
        </props>
      </entry>
      <entry key="OperatorOverIteratorsUnary">
        <props>
          <!--  |x                # Repeat until empty  -->
          <prop key="|">new IconRepeatUntilEmpty</prop>
          <!--  <>x       # Wrap generator as singleton iterator  -->
          <!--  <prop key="&lt;>">new IconFirstClass</prop>
                           -->
          <!--  [: x      # List comprehension  -->
          <prop key="[:">new IconListComprehension</prop>
        </props>
      </entry>
      <!--

        #====
        # OperatorAsGenerator: used for operators over atoms that
        #       return an iterator.  The operator is changed into
        #       an iterator over atoms after normalizing out its arguments.
        #       Treated like a function by normalization.
        #====
      -->
      <entry key="OperatorAsGenerator">
        <props>
          <prop key="!">new IconPromote</prop>
          <prop key="to">new IconToIterator</prop>
        </props>
      </entry>
      <!--

        #====
        # OperatorAsFunction: used for operators over atoms or values that
        #       return an iterator  The operator is changed into
        #       a function call after normalizing out its arguments.
        #       Treated like a function by normalization.
        #       OverValues is default unless used with FunctionOverAtoms.
        #====
      -->
      <entry key="OperatorAsFunction">
        <props>
          <!--
             !x       # Lift collection or Java iterator to generator
          -->
          <!--  f!x  # Handled in normalization => f(x.toArray())  -->
          <!--  x to y by z  # Prototypical generator function  -->
          <!--

                        <prop key="to">IconOperators.to</prop>
                        <prop key="!">IconOperators.promote</prop>

          -->
          <!--  |<>| f(e)  # Data-parallel => |> parallel(<>f,<>e)  -->
          <prop key="|<>|">parallel</prop>
          <!--  &features    # No-arg &keyword  -->
          <prop key="&features">IconKeywords.features</prop>
        </props>
      </entry>
      <!--

        #====
        # FunctionOverAtoms: used for built-in functions over atoms.
        #====
      -->
      <entry key="FunctionOverAtoms">
        <props>
          <!--

                        <prop key="put">IconFunctions.putOverAtoms</prop>
                        <prop key="ishift">IconFunctions.ishiftOverAtoms</prop>

          -->
        </props>
      </entry>
      <!--

        #====
        # OperatorOverAtoms: used for operations over atoms that return atoms.
        #====
      -->
      <entry key="OperatorOverAtomsBinary">
        <props>
          <prop key="+">IconOperators.plus</prop>
          <prop key="-">IconOperators.minus</prop>
          <prop key="*">IconOperators.times</prop>
          <prop key="/">IconOperators.division</prop>
          <prop key="%">IconOperators.remainder</prop>
          <prop key="^">IconOperators.powerOf</prop>
          <prop key="=">IconOperators.sameNumberAs</prop>
          <prop key="==">IconOperators.sameStringAs</prop>
          <prop key="===">IconOperators.sameValueAs</prop>
          <prop key="~=">IconOperators.notSameNumberAs</prop>
          <prop key="~==">IconOperators.notSameStringAs</prop>
          <prop key="~===">IconOperators.notSameValueAs</prop>
          <prop key="<">IconOperators.lessThan</prop>
          <prop key="<=">IconOperators.lessThanOrEquals</prop>
          <prop key=">">IconOperators.greaterThan</prop>
          <prop key=">=">IconOperators.greaterThanOrEquals</prop>
          <prop key=">>">IconOperators.stringGreaterThan</prop>
          <prop key=">>=">IconOperators.stringGreaterThanOrEquals</prop>
          <prop key="<<">IconOperators.stringLessThan</prop>
          <prop key="<<=">IconOperators.stringLessThanOrEquals</prop>
          <prop key="++">IconOperators.setUnion</prop>
          <prop key="--">IconOperators.setDifference</prop>
          <prop key="**">IconOperators.setIntersection</prop>
          <prop key="||">IconOperators.stringConcat</prop>
          <prop key="|||">IconOperators.listConcat</prop>
          <prop key="@">IconOperators.activate</prop>
          <prop key="@>">IconOperators.send</prop>
          <prop key="@>>">IconOperators.blockingSend</prop>
          <prop key="<<@">IconOperators.blockingReceive</prop>
        </props>
      </entry>
      <entry key="OperatorOverAtomsUnary">
        <props>
          <prop key="+">IconOperators.plusUnary</prop>
          <prop key="-">IconOperators.minusUnary</prop>
          <prop key="=">IconOperators.tabMatch</prop>
          <prop key="*">IconOperators.timesUnary</prop>
          <prop key="?">IconOperators.questionMarkUnary</prop>
          <prop key="\">IconOperators.failIfNull</prop>
          <prop key="/">IconOperators.failIfNonNull</prop>
          <prop key=".">IconOperators.dereference</prop>
          <prop key="~">IconOperators.csetComplement</prop>
          <prop key="^">IconOperators.refresh</prop>
          <prop key="@">IconOperators.activate</prop>
          <prop key="<@">IconOperators.receiveUnary</prop>
          <prop key="<<@">IconOperators.blockingReceiveUnary</prop>
        </props>
      </entry>
      <!--

        #====
        # UndoFunction: maps symbol to an undo factory and marks it as undoable.
        #====
      -->
      <entry key="UndoFunction">
        <props>
          <prop key="tab">IconScan.createUndo()</prop>
          <prop key="move">IconScan.createUndo()</prop>
          <prop key="<-"/>
          <prop key="<->"/>
        </props>
      </entry>
      <!--

        #====
        # SymbolAsValue: maps symbol such as &null into a literal value.
        #       Literal values won't be lifted to an interator in normalization
        #       when they are function arguments.
        #====
      -->
      <entry key="SymbolAsValue">
        <props>
          <!--  &null             # No-arg &keyword  -->
          <prop key="&null">null</prop>
        </props>
      </entry>
      <!--

        #====
        # SymbolAsVariable: maps symbol such as &digits into an object reference
        #       For example, &subject => IconKeywords.subject
        #       The simple object reference is then treated as an assignable
        #       variable by the transforms.
        #====
      -->
      <entry key="SymbolAsVariable">
        <props>
          <!--  &digits   # No-arg &keyword  -->
          <prop key="&digits">IconKeywords.digits</prop>
          <prop key="&cset">IconKeywords.cset</prop>
        </props>
      </entry>
      <!--

        #====
        # SymbolAsProperty: maps symbol such as &subject into a property
```

```
          #       with get() and set() methods, i.e., an IIconAtom.
          #====
        -->
        <entry key="SymbolAsProperty">
          <props>
            <!--  &subject  # No-arg &keyword  -->
            <prop key="&subject">IconKeywords.subject</prop>
            <prop key="&pos">IconKeywords.pos</prop>
            <prop key="&current">IconKeywords.current</prop>
            <prop key="&source">IconKeywords.source</prop>
            <prop key="&main">IconKeywords.main</prop>
            <prop key="&time">IconKeywords.time</prop>
          </props>
        </entry>
        <!--

          #====
          # SymbolAsIterator: maps symbol such as &fail into an iterator.
          #       Symbols defined this way must map to a new or method invocation
          #       that returns a mutable iterator.
          #====
        -->
        <entry key="SymbolAsIterator">
          <props>
            <!--  &fail     # No-arg &keyword  -->
            <prop key="&fail">new IconFail()</prop>
          </props>
        </entry>
      </map>
    </property>
  </bean>
  <!--

    #================================================
    # Script engine handler for scripting substrates
    #================================================
  -->
  <bean id="DefaultSubstrate"
  class="edu.uidaho.junicon.interpreter.interpreter.AbstractSubstrate">
    <property name="parent" ref="JuniconInterpreter"/>
    <property name="logger" ref="DefaultLogger"/>
    <property name="quietScriptExceptions" value="false"/>
    <!--
        <property name="defaultLineSeparator" value="\n"/>
         -->
    <property name="scriptEngineManager">
      <bean class="javax.script.ScriptEngineManager"></bean>
    </property>
    <property name="defaultScriptExtension" value="groovy"/>
    <property name="defaultScriptEngine">
      <bean
      class="edu.uidaho.junicon.substrates.groovy.groovyshell.GroovyScriptEngineImports">
      </bean>
    </property>
  </bean>
</beans>
<!--  END OF FILE  -->
```