A New Implementation of the Icon Language

Todd A. Proebsting
Microsoft Research

Gregg M. Townsend
The University of Arizona

October 4, 1999

University of Arizona Computer Science Technical Report
99–13

Microsoft Research Technical Report
MSR–TR–99–64

Department of Computer Science
University of Arizona
Tucson, AZ 85721

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

A New Implementation of the Icon Language

Todd A. Proebsting
Microsoft Research

Gregg M. Townsend
The University of Arizona

October 4, 1999

Abstract

We describe Jcon, a new, Java-based implementation of the Icon programming language. The implementation includes a compiler and runtime system. The runtime system is novel in its concise and efficient object-oriented implementation of a dynamically typed language, as well as its simple mechanism for realizing Icon generators.

1 Introduction

Since its conception in the mid-1970's, the Icon programming language has received a great deal of interest due to its rich set of built-in data types, its clean string-handling facilities, and its powerful combination of generators and goal-directed evaluation [GG97, GG96]. Icon is available for virtually every significant computer architecture and operating system [Ico]. Until now, every available implementation was based on the same source code written in the C programming language [GG86].

Using Java, we have reimplemented Icon to produce a new implementation called "Jcon". An object-oriented design yields a runtime system that is much shorter, simpler, and easier to understand. Although this particular system is specifically tailored to Icon, the techniques are equally well-suited to implementing any dynamically typed language.

2 The Icon Programming Language

Icon programs have an Algol-like appearance, but this similarity is misleading. For instance, the requisite "hello world" program looks quite ordinary.

```
procedure main()
  write("hello world")
end
```

Icon, however, has many non-Algol-like features that challenge implementors, including dynamic typing, generators, and string scanning. The following Icon program makes use of a user-defined generator to write `3` and `three` (on separate lines).

Icon Type	Description
null	unique value signifying uninitialized data
integer	unbounded precision integer value
real	floating-point value
string	sequence of zero or more text characters
cset	set of 8-bit-encoded text characters
file	handle for sequential and random file operations
record	user-defined structure containing named fields
list	ordered collection of zero or more values
set	unordered collection of unique values
table	associative array of value mappings
procedure	procedure, built-in function, or operator
co-expression	independent thread of computation

Table 1: Icon's Types

```
procedure main()
  every x := gen() do
    write(x)
end

procedure gen()
  suspend 3
  suspend "three"
end
```

Procedure gen() generates two values by virtue of *suspending* between values. Note that the variable x contains both integer and string values at different times, as allowed by Icon's dynamic typing. Similarly, the built-in procedure write() can handle arguments of many different types. Procedure main() can be written more concisely as

```
procedure main()
  every write(gen())
end
```

This concise version of main() demonstrates that generators can produce a sequence of values directly into an enclosing expression.

Rather than exhaustively enumerate Icon's syntactic structures here, we will introduce them as necessary.

2.1 Dynamic typing

Icon is a dynamically-typed language. Icon storage elements (variables, arguments, list elements, etc.) are untyped—they may hold values of any type at execution time. Table 1 summarizes Icon's built-in rich set of types.

Whenever possible, Icon performs the necessary conversions to apply operators to operands of unexpected types. For instance, when trying to add two values, both are converted to numeric values, if possible, at execution time. Thus, the Icon expression 3+"4" generates the integer value 7.

Similarly, Icon disambiguates overloaded operations at execution time based on operand types. For instance, x[y] means the yth element of x if x is a list, but it means the value associated with key y in table x if x is a table.

2.2 Generators

Generators (iterators) and goal-directed expression evaluation are extremely powerful control-flow mechanisms for succinctly expressing operations that process a sequence of values. The Prolog programming language derives much of its power from goal-directed evaluation (i.e., backtracking) in combination with unification [Byr80]. The Icon programming language is an expression-oriented language that combines generators and goal-directed evaluation into a powerful control-flow mechanism [GG97].

An Icon expression *succeeds* by producing a value. A *generator* can produce multiple values. An expression that cannot produce any more values *fails*. For example, the expression

```
1 to 5
```

generates the values 1, 2, 3, 4, 5, and then fails.

Combining expressions with operators or function calls creates a compound expression that combines all subexpression values and generates all possible result values prior to failing. The expression

```
(1 to 3) * (1 to 2)
```

generates the values 1, 2, 2, 4, 3, 6, and then fails. Subexpressions evaluate left-to-right—the previous sequence represents `1*1`, `1*2`, `2*1`, `2*2`, `3*1`, `3*2`. Note that the right-hand expression is re-evaluated for each value generated by the left-hand expression.

Generators may have generators as subexpressions. The expression

```
(1 to 2) to (2 to 3)
```

generates 1, 2, 1, 2, 3, 2, 2, 3, and then fails. Those values are produced because the outer (middle) `to` generator is actually initiated four times: `1 to 2`, `1 to 3`, `2 to 2`, and `2 to 3`.

After a generator produces a value, it *suspends* execution until another value is needed. When another value is needed, the generator *resumes* execution.

2.3 Goal-directed evaluation

Icon's expression evaluation mechanism is goal-directed. Goal-directed evaluation forces expressions to re-evaluate subexpressions as necessary to produce as many values as possible. To demonstrate this, we introduce Icon's relational operator $<$. The $<$ operator takes two numeric operands and returns the value of the right operand if it is greater than the value of the left; otherwise, it fails (and, therefore, generates no value). Goal-directed evaluation forces $<$ to re-evaluate its operand expressions as necessary to produce values on which it succeeds. The expression

```
2 < (1 to 4)
```

generates the values 3, 4, and then fails. Similarly,

```
3 < ((1 to 3) * (1 to 2))
```

generates 4, 6, and then fails.

Generators and goal-directed evaluation combine to create succinct programs with implicit control flow.

2.4 Keywords

Icon has special entities, called *keywords*, that represent values and/or variables that are specific to Icon programs. Keywords are not all created the same, and many require special consideration from both the compiler and the runtime system.

Many keywords represent pre-defined constants. For instance, `&null` represents the (only) value of Icon's "null" type, and `&pi` represents the value of π. (All Icon keywords are distinguished syntactically by their `&` prefix.)

Other keywords represent values determined by the state of an Icon program's execution. For instance, &time represents the number of milliseconds of CPU time that an Icon program has used, and &clock gives a string representing the current time of day.

Some keywords act as variables. &random is the random-number generator seed value, and it can be assigned an integer value at execution time. Assignment to some keywords can have side-effects to other keywords. For example, assigning any value to &subject also sets &pos to 1; these keywords are used in string scanning.

Other variable keywords affect the Icon virtual machine's subsequent execution. Assigning a non-zero value to &error causes subsequent computations that would normally result in runtime errors to simply "fail" (in the sense that generators fail when they cannot produce another value).

&line and &file represent the compile-time values of the line number and file name for the location of those given keywords in a program's text.

A few Icon keywords are generators that produce more than one value. &features generates strings that represent the language features supported by the given Icon implementation, such as "large integers". On the other hand, &fail produces no value at all, but fails immediately.

2.5 String scanning

Icon is a descendent of SNOBOL4 [GPP71] and shares much of SNOBOL4's emphasis on string operations. Icon provides a large number of operations for manipulating strings (concatenation, substring replacement, etc.). Unlike SNOBOL4, however, Icon's pattern matching requires no special consideration: Icon's generators and goal-directed evaluation suffice for implementing its pattern-matching operations.

3 A New Implementation of Icon

Until now, all implementations of Icon have been closely related. The Icon source code has evolved through several versions since 1978 [GG96], and though occasional branches have sprouted, there have been no independent implementations. The current mainstream edition is Version 9.3.2 of Icon for Unix, or simply "Version 9"; this is the *reference implementation* to which we will compare Jcon.

Jcon is a completely new implementation, rewritten from the ground up. It has two main parts: a compiler and a runtime system. The compiler is written in Icon and generates class files containing Java bytecodes—binary instructions for the Java "virtual machine" [LY97]. The runtime system is linked with the generated code and is written in Java.

The motivations behind Jcon are many-fold. The Jcon compiler represents an experiment with a new mechanism for translating goal-directed evaluation. The Jcon runtime system represents an experiment in a novel object-oriented architecture for a dynamically typed language implementation. And finally, Jcon represents an experiment in targeting the Java Virtual Machine from a decidedly non-Java-like language.

The Jcon compiler is responsible for effecting goal-directed control flow. Beyond this, it is little more than a translator to calls on runtime system routines. Almost all of the action—type conversion, computation, generation of values, etc.—takes place in the runtime system.

Because Icon is a dynamically-typed language, most of the functionality of the runtime system is type-dependent. For instance, evaluating a[b] requires different actions depending on a's type (string, table, list, integer, or whatever), and b's type. Icon's type system pervades the design and implementation of its runtime system. Therefore, the best way to understand Jcon is to study its data structures. We will begin with those, then move on to the runtime methods, and finally examine the Jcon compiler.

4 An Object-Oriented Runtime system

Icon variables are typeless; "type" is an attribute possessed by values at execution time. Consequently, the compiler must produce general code that adapts as necessary to the actual runtime values.

Jcon addresses dynamic typing by making heavy use of Java instance methods, which are analogous to the "virtual" methods of C++. Methods are called in the context of a particular data value, and the Java class of this value (usually corresponding to an Icon type) selects the particular method that is called.

Consider the Icon expression x[3], where the effect of subscripting depends on the type of the value x. Jcon generates code equivalent to the Java expression x.Index(3). Different versions of the Index method are defined for strings, lists, tables, and other datatypes; the correct one is selected by the value of x at each invocation.

In the reference implementation, the source code is grouped by operation: All the code that implements subscripting is in one place, and conversely the code for the many operations dealing with lists is divided among several files. Jcon inverts this relationship, grouping code by datatype. The subscripting code is scattered among several files, each specific to one datatype, but all of the list-handling code is in one place. This makes adding new datatypes considerably easier.

4.1 The vDescriptor Type Hierarchy

In the Jcon runtime system, the vDescriptor class is used for almost all data. Every Icon datatype is represented by a subclass of vDescriptor. Some other subclasses are used for internal purposes. Table 2 illustrates the hierarchy of subclasses of vDescriptor.

The three subclasses of vDescriptor are vValue, vVariable, and vClosure, and these in turn are subclassed further. The vValue class implements Icon data values, and the vVariable class implements variables. The vClosure class is used for suspending results from generators. We begin by discussing values, variables, and their operations, leaving closures for later.

4.2 Value classes

Instances of the vValue class represent Icon values. The subclass hierarchy closely reflects the Icon type hierarchy, but with finer subdivisions.

Icon transparently supports integers of arbitrary size, but both Jcon and Version 9 use a simpler representation for values small enough to be supported by the hardware. In Jcon, this results in the distinct vInteger and vBigInt classes. Icon's numeric types are real and integer, which is reflected in Jcon by their subclassing of a common vNumeric parent class.

Icon files are opened in *translated* (text) or *untranslated* (binary) mode, affecting subsequent I/O operations. The vTFile and vBFile classes reflect this distinction. The vDFile class is used when a directory is opened for reading. Graphics windows are considered a distinct Icon type, but they support most read and write operations, and so the vWindow class is also implemented as a subclass of vFile.

Icon procedures may be called with any number of arguments—excess arguments are evaluated and discarded, and missing arguments get default values. Jcon distinguishes procedures by the number of arguments expected. This minimizes the overhead for procedure calls that provide the expected number of arguments. The vProcV class is used for procedures that declare more than nine parameters or that accept an arbitrary number of arguments. Record constructors are a special case of vProcV.

Icon classifies lists, sets, tables, and records as *structures*, and this grouping is reflected in the class hierarchy. Although Icon considers different record types to be distinct, all are implemented by the vRecord class.

The vSortElem and vTableElem classes are used to hold values during a sorting operation. They correspond to no Icon data type and are not present at other times.

Not included in Table 2 are classes specific to individual procedures, operators, and keywords. Every built-in procedure subclasses one of the vProc*n* classes and defines a Call method with the appropriate number of arguments. Procedures corresponding to Icon operators are also provided. Some Icon keywords such as &clock are implemented by subclasses of vProc0, and some keywords create anonymous subclasses of vSimpleVar.

Java Object Class	Icon Type
vValue: an Icon value	
vNull	&null, the null value
vString	string
vCset	character set
vNumeric	numeric value:
vInteger	... integer under 64 bits
vBigInt	... multiword integer
vReal	... floating-point value
vFile	file:
vTFile	... text file
vBFile	... binary file
vDFile	... directory
vWindow	... graphics "file"
vProc	procedure:
vProc0	... with no parameters
vProc1	... with 1 parameter
vProc2	... with 2 parameters
⋮	
vProc9	... with 9 parameters
vProcV	... with variable parameters
vRecordProc	...... record constructor
vCoexp	co-expression
vStructure	structured value:
vList	... list
vSet	... set
vTable	... table
vRecord	... record
vTableElem	used during sort(T)
vSortElem	used during sort(*other*)
vVariable: an assignable "lvalue"	
vSimpleVar	assignable atom
vLocalVar	local variable
vFuncVar	built-in procedure
vListVar	list element
vSubstring	substring reference
vTableRef	table element reference
vClosure: a suspended generator	
vProcClosure	special closure for &main
vTracedClosure	special closure for tracing

Table 2: Subclasses of vDescriptor

Java Return Type	Java Method	Icon Operation
vValue	Deref()	.x
vDescriptor	DerefLocal()	.x if local, else x
vVariable	Assign(vDescriptor x)	v := x
vVariable	SubjAssign(vDescriptor x)	&subject := x
vVariable	Swap(vDescriptor v)	a :=: b
vDescriptor	RevSwap(final vDescriptor v)	a <–> b
vDescriptor	RevAssign(vDescriptor v)	a <– b

Table 3: Dereferencing and Assignment Operations

4.3 Variable classes

The vVariable class encompasses program variables and also assignable intermediate values that occur within expressions. Each instance contains or references a modifiable value and supports Icon's name() procedure. A vVariable instance can also be used in any context where a value is required.

A vSimpleVar instance implements a global variable directly. The minor variation vLocalVar implements a local variable, which behaves differently when returned by a procedure. The vFuncVar class provides lazy initialization of global variables associated with built-in procedures. Icon lists are built of vListVar elements.

Subcripting a string produces a vSubstring, which can be assigned a value to affect the original parent string. Subscripting a table produces a vTableRef, which alters or adds a table element if assigned a value.

5 Runtime vDescriptor methods

Tables 3 through 9 itemize the many methods that are declared by the vDescriptor class. Any of these methods can be called with respect to any vDescriptor object.

For most methods, multiple definitions are present. In the typical case, there is one definition in the vVariable class and one in the vClosure class, and these definitions are inherited by all the subclasses of those two classes. Another definition in the vValue class serves as a default method that is inherited by most subclasses. Finally, for one or more specific datatypes, the vValue definition is overridden by a subclass method that is specific to one Icon datatype.

Jcon's use of instance calls to select different code under different conditions replaces much of the explicit type checking used in the reference version of Icon. We will examine a representative sample of vDescriptor methods to illustrate in a concrete fashion how this is used by Jcon.

It is important to realize that class-based method dispatching can be very efficient, requiring just two memory lookups to obtain the method address given the address of the object. No testing and branching of any sort is required, and the depth of the class hierarchy is not a factor. While method call speed depends on the particular Java system, it is a critical area that one can expect to be optimized in any performance-conscious Java implementation.

5.1 Dereferencing and Assignment Operations

Assignment associates a value with a variable; dereferencing extracts a value from a variable without changing the association. Table 3 summarizes the dereferencing and assignment methods of Jcon.

A variable is represented by a vVariable object and a value by a vValue object. Usually, a vVariable contains a private copy of a vValue that represents the value of the variable. The two exceptions are vSubstring and vTableRef objects, representing intermediate results from string and table subscripting that accept assignments affecting the originally subscripted value.

Dereferencing of variables is performed by the Deref method: Every subclass of vVariable defines a Deref method. In contrast to variables, values are already dereferenced; so the Deref method of the vValue class, designated vValue.Deref in Java terminology, just returns its associated object; and no subclass of vValue defines a Deref method.

The return value of an Icon procedure is not automatically dereferenced. A procedure can, for example, return a subscripted table reference as an assignable value. On the other hand, the procedure may not *want* to return an assignable value. The Icon dereferencing operation, .x, explicitly dereferences an expression and generates a call to the Deref method. The Deref method is also used heavily within the runtime system.

The DerefLocal variant is used when a procedure returns an Icon variable. Icon specifies that such a variable is dereferenced only if it is a local variable, which is not known at compilation time. The vLocalVar.DerefLocal version of the method dereferences its object; other versions just return the object unchanged. DerefLocal guarantees that local variables are not referenced after they cease to exist.

Assignment, to the compiler, is just another binary operation. The Assign methods in vVariable subclasses accomplish assignment by storing the value passed as the argument. For substrings and table references, these assignments alter the originally subscripted value. The vValue.Assign method (that is, the Assign method implemented by the vValue class) catches assignment to a constant (or expression result) and raises an error.

Variations such as swapping (x :=: y) and reversible assignment (x <− y) are implemented directly in the vDescriptor class in terms of assignment and dereferencing. The code that implements swapping is simple and instructive:

```
public vVariable Swap(vDescriptor v) {      // a :=: b
    vValue a = this.Deref();
    vValue b = v.Deref();
    vVariable rv;
    if ((rv = this.Assign(b)) == null || v.Assign(a) == null) {
        return null; /*FAIL*/
    }
    return rv;
}
```

A Java null value indicates failure of an Icon expression. This is distinct from a vNull object, which corresponds to the Icon null value, &null.

The SubjAssign method functions identically to the Assign method. It serves only to distinguish for traceback purposes the implicit assignment to &subject that initiates string scanning.

5.2 Conversion Methods

Jcon's runtime system defines several methods for converting a value to a (possibly) different type. Two sets of conversion methods are listed in Table 4.

The methods in the first set are used heavily for coercion: the implicit conversion that occurs when a value is not the type expected by an operator or built-in procedure. Most of these methods are called exclusively by code in the runtime system; only Numerate, distinguished by its initial capital, is called by generated code.

The mkString method is typical. In the vValue class, a default method raises a runtime error. A subclass of vValue that is convertible, such as vReal, overrides this method with one that creates and returns a vString value. The vString.mkString method is trivial: It just returns its own object.

Methods in the second set convert a value to an Icon string in a specialized manner for a specific purpose. The write and image methods return strings for use by the built-in procedures of those names. Contrast their handling of an Icon null value with mkString:

vNull.mkString	reports an error
vNull.write	returns ""
vNull.image	returns "&null"

Java Return Type	Java Method	Icon Operation
vString	mkString()	convert to vString
vInteger	mkInteger()	convert to vInteger
vNumeric	mkFixed()	convert to vInteger/vBigInt
vReal	mkReal()	convert to vReal
vNumeric	Numerate()	+n: convert to vNumeric
vCset	mkCset()	convert to vCset
vProc	mkProc(int i)	convert to vProc
vValue[ ]	mkArray(int errno)	convert to array of vValues
vString	write()	convert for write()
vString	image()	convert for image()
vString	report()	convert for error, traceback
vString	reportShallow()	convert without expanding

Table 4: Conversion Methods

The report and report_shallow methods are used in error reporting and traceback. They differ from image in the way they format structure values.

5.3 Simple Non-Arithmetic Functions

5.3.1 The Size Operation

Table 5 lists a large number of straightforward, non-arithmetic operations that produce one value at most. Icon's size operation, *x, is representative of these.

The meaning of a size operation depends on the type of its argument: *x returns the length of a string, the number of members of a set, and so on. For the expression *x, Jcon generates code equivalent to the Java expression x.Size(). Let us follow the action when that code is executed.

A simple method call from the generated code can lead to a cascade of additional calls at execution. This is not surprising, because of the way Jcon substitutes method overloading for tests and branches, but it presents a bit of an expository challenge. We use a table of actions reminiscent of parser rewriting rules. Here is the table tracing the execution of *x when x is a variable containing the string value "abc":

Icon	⇒	Java	Function Applied
*x	⇒	⟨x⟩.Size()	vVariable.Size() → Deref().Size()
	⇒	⟨x⟩.Deref().Size()	vVariable.Deref() → vValue
	⇒	⟨"abc"⟩.Size()	vString.Size() → vInteger.New(*length*)
	⇒	vInteger.New(3)	vInteger.New(x) → vInteger
	⇒	⟨3⟩	

The first column gives the original Icon code. Each line of the second column gives the Java code to be executed; the notation $\langle x \rangle$ means "the Java value representing the Icon value x." The third column names the specific method that is called, based on the class of the execution-time value, followed by the result produced by that method.

In this example, x is a variable, so vVariable.Size is called first. The entire body of this method is { return Deref().Size(); }. The expression effectively becomes ⟨x⟩.Deref().Size() and then Deref is called. It returns the underlying string, a vString (which is a subclass of vValue). The Size method is again called, this time reaching the vString version; it calculates the length and calls vInteger.New, a static method, to produce an Icon integer result.

Java Return Type	Java Method	Icon Operation
vInteger	Size()	*x
vCset	Complement()	~x
vCoexp	Refresh()	^C
vDescriptor	TabMatch()	=s
vNumeric	Abs()	abs(x)
vValue	Copy()	copy(x)
vString	Type()	type(x)
vString	Name()	name(v)
vInteger	Args()	args(p)
vString	LLess(vDescriptor v)	s1 << s2
vString	LLessEq(vDescriptor v)	s1 <<= s2
vString	LEqual(vDescriptor v)	s1 == s2
vString	LUnequal(vDescriptor v)	s1 ~== s2
vString	LGreaterEq(vDescriptor v)	s1 >>= s2
vString	LGreater(vDescriptor v)	s1 >> s2
vValue	VEqual(vDescriptor v)	v1 === v2
vValue	VUnequal(vDescriptor v)	v1 ~=== v2
vString	Concat(vDescriptor v)	s1 || s2
vList	ListConcat(vDescriptor v)	L1 ||| L2
vValue	Intersect(vDescriptor i)	x ** x
vValue	Union(vDescriptor i)	x ++ x
vValue	Diff(vDescriptor i)	x -- x
vList	Push(vDescriptor v)	push(L, x)
vValue	Pull()	pull(L)
vValue	Pop()	pop(L)
vValue	Get()	get(L)
vList	Put(vDescriptor v)	put(L, x)
vValue	Member(vDescriptor i)	member(X, x)
vValue	Delete(vDescriptor i)	delete(X, x)
vValue	Insert(vDescriptor i, j)	insert(X, x1, x2)
vInteger	Serial()	serial(x)
vList	Sort(int i)	sort(X, i)

Table 5: Simple Non-Arithmetic Functions

When the variable x contains an integer value, such as 5, the sequence is different. The integer is coerced to a string value before applying the size operator:

Icon	⇒	Java	Function Applied
*x	⇒	⟨x⟩.Size()	vVariable.Size() → Deref().Size()
	⇒	⟨x⟩.Deref().Size()	vVariable.Deref() → vValue
	⇒	⟨5⟩.Size()	vNumeric.Size() → mkString().Size()
	⇒	⟨5⟩.mkString().Size()	vInteger.mkString() → vString
	⇒	⟨"5"⟩.Size()	vString.Size() → vInteger.New(*length*)
	⇒	vInteger.New(1)	vInteger.New(x) → vInteger
	⇒	⟨1⟩	

Finally, suppose that x holds an illegal value for the size operation, such as a procedure value. There is no Size method defined by vProc or any of its subclasses. Instead, the default vValue.Size method issues an error. This same method would have been called in the two previous sequences had it not been overridden by Size methods in the vString and vNumeric classes.

5.3.2 String Concatenation

Binary operators are dispatched in the same manner as unary operators, but additional work is required to validate or coerce the type of the right-hand operand. This is done by calling one of the coercion methods of Table 4. String concatenation provides a good example.

Consider the Icon expression s || 3, which concatenates the string s with the integer 3 to produce a new string. The Jcon compiler produces code for this expression that is equivalent to the Java expression s.Concat(⟨3⟩). Execution proceeds in this manner:

Icon	⇒	Java	Function Applied
s || 3	⇒	⟨s⟩.Concat(⟨3⟩)	vVariable.Concat(x) → Deref().Concat(x)
	⇒	⟨s⟩.Deref().Concat(⟨3⟩)	vVariable.Deref() → vValue
	⇒	⟨"abc"⟩.Concat(⟨3⟩)	vString.Concat(x) → Concat(x.mkString())
	⇒	⟨"abc"⟩.Concat(⟨3⟩.mkString())	vInteger.mkString() → vString
	⇒	⟨"abc"⟩.Concat(⟨"3"⟩)	vString.Concat(x) → vString
	⇒	⟨"abc3"⟩	

As is typical for this sort of binary operation, vString.Concat unconditionally calls mkString to coerce its second argument to the correct type. The unconditional call saves the cost of a test. If the argument is already a string, vString.mkString immediately returns its own instance, requiring no actual work.

5.3.3 Other Simple Functions

The Size and Concat methods are typical of a large class of operations. Table 5 lists several methods that implement fundamental Icon operations. These are all simple, non-arithmetic methods that are not generators.

Not all of these methods correspond to Icon operators. Methods such as Abs, Copy, and Type are never called from the generated code, only from procedures that are part of the runtime library. It is nevertheless useful to implement them as vDescriptor methods because they share the same behavioral dependency on the underlying datatype.

5.4 Arithmetic Functions

Arithmetic operations are more complex because of their special type conversion rules. In Icon, the negation operation –x returns an integer if x is an integer, or a real number if x is real; but if x is a string, the result can be either integer or real depending on the string contents.

For binary operations, both operands are coerced to a numeric type. In most cases, the result of the operation is a real value if either operand is real; otherwise the result is an integer. (The exception is the

Java Return Type	Java Method	Icon Operation
vNumeric	Negate()	–n
vNumeric	Add(vDescriptor v)	n1 + n2
vNumeric	Sub(vDescriptor v)	n1 – n2
vNumeric	Mul(vDescriptor v)	n1 * n2
vNumeric	Div(vDescriptor v)	n1 / n2
vNumeric	Mod(vDescriptor v)	n1 % n2
vNumeric	Power(vDescriptor v)	n1 ^ n2
vNumeric	NLess(vDescriptor v)	n1 < n2
vNumeric	NLessEq(vDescriptor v)	n1 <= n2
vNumeric	NEqual(vDescriptor v)	n1 = n2
vNumeric	NUnequal(vDescriptor v)	n1 ~= n2
vNumeric	NGreaterEq(vDescriptor v)	n1 >= n2
vNumeric	NGreater(vDescriptor v)	n1 > n2
vNumeric	AddInto(vInteger a)	a + b ⇒ b.AddInto(a)
vNumeric	SubFrom(vInteger a)	a – b ⇒ b.SubFrom(a)
vNumeric	MulInto(vInteger a)	a * b ⇒ b.MulInto(a)
vNumeric	DivInto(vInteger a)	a / b ⇒ b.DivInto(a)
vNumeric	ModInto(vInteger a)	a % b ⇒ b.ModInto(a)
vNumeric	PowerOf(vInteger a)	a ^ b ⇒ b.PowerOf(a)
vNumeric	BkwLess(vInteger a)	a < b ⇒ b.BkwLess(a)
vNumeric	BkwLessEq(vInteger a)	a <= b ⇒ b.BkwLessEq(a)
vNumeric	BkwEqual(vInteger a)	a = b ⇒ b.BkwEqual(a)
vNumeric	BkwUnequal(vInteger a)	a ~= b ⇒ b.BkwUnequal(a)
vNumeric	BkwGreaterEq(vInteger a)	a < b ⇒ b.BkwGreaterEq(a)
vNumeric	BkwGreater(vInteger a)	a >= b ⇒ b.BkwGreaterEq(a)
vNumeric	AddInto(vBigInt a)	a + b ⇒ b.AddInto(a)
vNumeric	SubFrom(vBigInt a)	a – b ⇒ b.SubFrom(a)
	⋮	
vNumeric	AddInto(vReal a)	a + b ⇒ b.AddInto(a)
vNumeric	SubFrom(vReal a)	a – b ⇒ b.SubFrom(a)
	⋮	

Table 6: Arithmetic Functions

exponentiation operator, with its own special rules.) All of this is further complicated by the presence of two different integer types in the actual implementation.

Jcon's approach to binary arithmetic is to make two levels of instance calls. The first is dispatched by the left-hand operand type; then the operands are reversed and a "backwards" operation is dispatched by the right-hand operand type. Thus for a single operation such as addition there are nine methods for the 3×3 combinations of implementation types. The arithmetic operations are listed in Table 6.

For a concrete example, consider the expression r – i, where r and i are real and integer constants. The compiler generates r.Sub(i), which invokes vReal.Sub(i) based on the type of r. This in turn immediately calls i.SubFrom(self) to do the actual computation:

Java Return Type	Java Method	Icon Operation
vVariable	Field(String s)	R . f
vDescriptor	Index(vDescriptor v)	x[v]
vDescriptor	Section(vDescriptor i, j)	x[i:j]
vDescriptor	SectPlus(vDescriptor a, b)	x[i+:j]
vDescriptor	SectMinus(vDescriptor a, b)	x[i–:j]
vDescriptor	Select()	?x
vDescriptor	Bang()	!x
vDescriptor	Key()	key(T)
vValue	IndexVal(vDescriptor v)	.x[i]
vValue	SectionVal(vDescriptor i, j)	.x[i:j]
vValue	SectPlusVal(vDescriptor a, b)	.x[i+:j]
vValue	SectMinusVal(vDescriptor a, b)	.x[i–:j]
vValue	SelectVal()	.?x
vDescriptor	BangVal()	.!x

Table 7: Element Access Operations

Icon	⇒	Java	Function Applied
3.14–2	⇒	⟨3.14⟩.Sub(⟨2⟩)	vValue.Sub(x) → Numerate().Sub(x)
	⇒	⟨3.14⟩.Numerate().Sub(⟨2⟩)	vReal.Numerate() → vNumeric
	⇒	⟨3.14⟩.Sub(⟨2⟩)	vReal.Sub(x) → x.SubFrom(this)
	⇒	⟨2⟩.SubFrom(⟨3.14⟩)	vInteger.SubFrom(x) → vReal.New(*result*)
	⇒	vReal.New(3.14–2)	vReal.New(x) → vReal
	⇒	⟨1.14⟩	

There are three SubFrom methods declared by the vDescriptor class; they are distinguished by their argument types. When vReal.Sub(i) calls i.SubFrom(self), it calls an i.SubFrom(vReal) method selected by the class of i. The vInteger.SubFrom(vReal) method always deals with an integer subtracted from a real, so the actual subtraction operation is easily accomplished and a real value is returned.

This two-level call technique works well for Jcon because of the small number of interacting (numeric) types, but it should be noted that it does not scale well. The number of variants of each operation is proportional to the square of the number of interacting types.

5.5 Element Access Operations

Table 7 lists the methods that return a portion of a string or a structure. There is much overlap, with many operations applicable to either, hence the grouping in a single table; and the Select operation can also be applied to a numeric value. The second half of the table consists of specialized versions of methods from the first half.

All of these methods can operate on structures of one kind or another, often returning a variable. When a variable is returned, it is the vVariable object from within the internal representation of the structure, and assigning a value to it changes the value in the structure.

All but Field and Key can be applied to strings. The result is an assignable substring if the original string is a variable. In the Jcon implementation, this requires allocating and initializing a new vSubstring object, which incurs some extra cost. As an optimization, the Jcon compiler recognizes some situations in which the result of the access operation is used only as a value. For these cases it generates calls to the Val variants, which are streamlined methods that produce a value instead of an assignable substring when operating on a

Java Return Type	Java Method	Icon Operation
vDescriptor	Call()	call with no arguments
vDescriptor	Call(vDescriptor a)	call with one argument
vDescriptor	Call(vDescriptor a, b)	call with two arguments
	⋮	
vDescriptor	Call(vDescriptor a, b, c, d, e, f, g, h, i)	call with nine arguments
vDescriptor	Call(vDescriptor v[ ])	call with argument array
vDescriptor	ProcessArgs(vDescriptor x)	p ! L (call with arglist)
vDescriptor	Resume()	resume suspended generator

Table 8: Procedure Call Operations

string.

The Bang and Key methods are generators, which are discussed later. Its role as a generator is the reason that BangVal returns a vDescriptor and not a vValue.

5.6 Procedure Call Operations

Table 8 summarizes the procedure call operations. An Icon call that passes fewer than ten arguments generates a Java call to the corresponding Call method. An Icon call with ten or more arguments generates a Java call that passes the arguments in a vDescriptor array.

When Jcon compiles an Icon procedure, it produces a subclass of vProc that defines a Call method containing the procedure body. For an Icon procedure that declares three parameters, Jcon generates a new subclass of vProc3 with a Call(a, b, c) method. A corresponding three-argument call goes directly to this method without executing any runtime support code.

The Icon language, however, allows the actual argument count of a call to differ from the parameter count of the called procedure. The null value is substituted for missing values, and extra parameters are evaluated but discarded. The mechanism for handling argument count mismatches is provided by the vProc*n* classes and involves a brief excursion into the runtime system.

The vProc3 class defines Call methods with zero, one, and two parameters that supply Icon null values and call the three-argument Call method. The vProc3 class also defines Call methods that accept four or more arguments, discarding the extras, and a Call method that accepts a vDescriptor array. These methods are inherited by the subclasses of vProc3, which then need only to implement a single Call method—the one that takes three arguments.

The same pattern holds for the other direct subclasses of vProc. Each of the vProc0 through vProc9 classes, as well as vProcV, is an abstract class that defines all of the Call methods *except* the most appropriate one. This is left for their concrete subclasses, which need only implement a single Call method.

The Icon expression p ! L passes an Icon list as the argument list of a procedure. This operation generates a call to the ProcessArgs method, which just builds a vDescriptor array and calls the array-based Call method.

The Resume method is used with generators, which are discussed later.

5.7 Miscellaneous Operations

Table 9 summarizes the remaining runtime methods not included in the other categories.

Java Return Type	Java Method	Icon Operation
boolean	isnull()	runtime check for null
boolean	iswin()	runtime check for graphics window
vDescriptor	IsNull()	/ x
vDescriptor	IsntNull()	\ x
vDescriptor	Conjunction(vDescriptor x)	e1 & e2
vDescriptor	ToBy(vDescriptor j, k)	i to j by k
vDescriptor	Activate(vDescriptor x)	v @ C
vNumeric	Limit()	e \ n

Table 9: Miscellaneous Operations

The isnull and iswin methods perform type checks that occur many times in the runtime system. Their definitions are very simple, and they are faster than checking x instanceof vNull in Java. The vValue class defines isnull as a simple method that always returns false. The vNull class overrides this definition with a simple method that always returns true. vVariable.isnull is just { return Deref().isnull(); }, overridden by an even simpler vSubstring.isnull that always returns false. The definitions of iswin follow a similar pattern.

The IsNull and IsntNull methods correspond to the Icon operations /x and \x respectively. They differ from the simple runtime checks above, and from the operations of Table 5, by returning a variable (not just a value) if x is a variable. Except for that, though, they are similar to the isnull method.

Conjunction (e1 & e2) is, surprisingly, a very simple operation that just returns its second argument. All the real work is done by the evaluation of e1 and e2; if either of those fail, the entire expression fails. Alternation (e1 | e2), on the other hand, is a control structure that must be addressed by the compiler; there is no corresponding operation in the runtime system.

The ToBy method implements i to j and i to j by k. Despite the unusual syntax, this is essentially an operator that is a generator. The Activate method switches control to a different co-expression, which is a form of coroutine. Co-expressions are implemented using Java threads.

The Limit method works in conjunction with generated code to produce no more than n results from a generator. Actual flow control is handled by the generated code; the Limit method is responsible for coercing n to an integer and checking that it is greater than zero.

5.8 Predefined procedures

Icon defines over 130 predefined procedures, termed *functions* by the Icon book [GG97]. These procedures are written in Java and packaged along with the operators and other components of the runtime system. They are available implicitly to every Icon program. For example, if write is a global variable that is not defined by the programmer as a procedure or record constructor, then it is initialized to the value of the predefined procedure.

Each predefined procedure is an instance of a unique subclass of vProc. Because there is some cost to initializing a class, global variables that would hold predefined procedures are actually initialized as instances of the vFuncVar class. These objects function just like vSimpleVar objects except that if a Deref call occurs before any value has been stored then the value is initialized to a predefined procedure. This lazy initialization saves a noticeable amount of time for programs that reference many predefined procedures without calling them. This is especially common with linked library files.

6 Generators

A generator is a procedure that can produce a sequence of results from a single invocation, where each result is either a value or a variable. This definition includes internal procedures that implement Icon operators. A generator *suspends* to produce a result, and the calling expression can later *resume* it to request another result. The process repeats until either the generator fails, after which it cannot be resumed, or until the calling expression has no more need for the generator.

In general, the caller of a procedure does not know whether the procedure is a generator. Similarly, a called procedure does not know whether it might possibly be resumed by the caller after returning a result. The most common situation is for a procedure to produce just a single result. Jcon's object-oriented approach allows a simple approach for the common case, with complication added only when truly needed.

A non-generator procedure simply returns a vValue or vVariable result with no additional flags. If its caller tries to resume it, this is handled in the vValue or vVariable code and requires no attention from the individual procedure. Only a true generator needs to return a more complex structure.

A procedure result can likewise be processed without regard to its generator status. A generator's result behaves just like a vValue or vVariable, and unless the context allows the possibility of resumption, the compiler treats it as such and makes no provision for the possibility that a called procedure might be a generator.

6.1 Suspension

In Jcon, a procedure fails by returning a Java null value. It succeeds by returning a vDescriptor object. For a simple Icon return, this object is either a vValue or a vVariable.

When an Icon procedure suspends, it produces a result along with enough state to enable subsequent resumption. This information is encapsulated in an object of the vClosure class, which is the third subclass (besides vValue and vVariable) of vDescriptor. The vClosure object contains:

- a retval field holding the suspended result (a vValue or vVariable object)
- a Resume method for generating subsequent values
- any data needed by the Resume method

As a subclass of vDescriptor, the vClosure class implements the full set of vDescriptor methods. Almost all of these methods just extract the suspended result from within the vClosure object and then re-invoke the same method. For example:

```
public vNumeric Add(vDescriptor v)   { return retval.Add(v); }
```

The suspended result retval is always a vValue or a vVariable.

Because a vClosure object implements all of the vDescriptor methods, it behaves just like the variable or value it contains. This means that the caller does not need to treat the result of a generator specially in order to make use of it. If the calling code cannot make use of a sequence of values, it ignores the possibility that it is calling a generator and makes no special provision for handling a vClosure result.

6.2 Resumption

Generators are resumed by the action of certain control structures. These include every, alternation, and implicit control backtracking. For such control structures, Jcon generates code that saves the result of the initial procedure call. Resumption is accomplished by calling the Resume method of this object. The Resume method returns a vDescriptor object to suspend a new value or a Java null value to fail. Until it fails, the initial object can be resumed multiple times to produce successive values.

The Resume method is the one method of the vClosure class that does not simply re-invoke the same method on the underlying result (retval). Conversely, the Resume methods of vValue and vVariable are trivial: each immediately returns a null value to signal failure. The calling code can resume a result object without checking for a vClosure. A simple vValue or vVariable fails immediately if resumed, which is the correct action in this situation.

6.3 Generator Structure

Conceptually, there are three phases of operation in a generator: initialization, production of the first result, and production of subsequent results. The third phase occurs after suspension and resumption, so the vClosure object returned as the first result must contain a pointer to this code.

Java does not provide code pointers or even function pointers. The standard substitute for a function pointer is an object containing a method with a prearranged name. In this case, the object is the vClosure object returned as the first result, and the method name is Resume().

Every generator needs a different Resume() method, so every generator must define a new subclass of vClosure. Version 1.1 of Java introduced an "inner class" notation for defining anonymous classes at point of need, so the source code is not too unwieldy. There are still some implications in terms of initialization cost and generated code bulk.

The first result of a generator must be a vClosure object, but subsequent results can be vValue or vVariable objects. However, it is often simplest to use common code to produce all results, returning a vClosure object every time.

6.4 A Generator Example

This procedure generates the factors of an integer. For simplicity, its Resume method generates all results including the first.

```
public class factors extends vProc1 {

    public vDescriptor Call(vDescriptor a) {
        final long arg = a.mkInteger().value;
        return new vClosure() {
            long n = 0;
            public vDescriptor Resume() {
                while (++n <= arg) {
                    if (arg % n == 0) {
                        retval = vInteger.New(n);
                        return this;
                    }
                }
                return null; /*FAIL*/
            }
        }.Resume();
    }

}
```

As a single-argument procedure, this example procedure extends the vProc1 class and defines a single-argument Call method. The vClosure object is created, called, and returned by the large return expression,

```
return new vClosure() { ... }.Resume();
```

which encompasses the entire definition of the anonymous subclass of vClosure.

7 Other Runtime Issues

7.1 Keywords

Each Icon keyword is implemented by a subclass of vProc0 that defines a Call method with no arguments; Jcon generates a procedure call for each keyword reference. This mechanism suffices for all the different kinds of keywords including constants like &null, read-only values like &clock, and assignable keywords like &trace.

The Jcon compiler knows that only five keywords are generators: &features, &allocations, &collections, &regions, and &storage. For the others it generates simpler code. (Of these generators, only &features is meaningful; the others are present for compatibility but generate only zero values.)

Each assignable keyword defines an inner subclass of vSimpleVar that overrides the Assign method. An object of this subclass is returned by the keyword's Call procedure. The custom Assign method validates an assigned value according to the rules of its keyword. Some assignable keywords are paired: &subject with &pos, &x with &row, and &y with &col. Except for &pos, assigning one of these also affects its partner.

String scanning, being a control structure, is mainly the compiler's responsibility. The compiler establishes and maintains scanning environments. Two runtime effects are notable: Assignment to &subject sets &pos to 1 as a side effect, and for predefined scanning procedures, the default value of a pos argument depends on whether the subject argument was defaulted.

The rarely used keyword &level provides the depth of the current procedure call. Its value is calculated by creating a Java exception and reading its stack trace. This ugly method was chosen because it eliminates the need for any separate bookkeeping.

7.2 Error Handling

The Jcon runtime system handles runtime errors in a graceful and systematic fashion. A runtime error normally terminates the program, printing a diagnostic message and a traceback of the procedure call chain. If the keyword &error is set to a nonzero value, an error instead produces a silent failure of the enclosing expression, and execution continues.

When the runtime system detects an error, it invokes the static method iRuntime.Error with the error number and, optionally, the offending value. (The Icon language definition enumerates all runtime errors [GG97].) Error simply creates and throws an iError, which subclasses java.lang.Error. Every Icon operation has an exception handler that can catch an iError. These handlers catch the exception, and depending on the value of &error either convert the error to failure (by returning null), or cooperate in creating a diagnostic stack trace.

The stack trace is constructed one call frame at a time as exceptions are caught and then re-raised. After the Icon program's stack is completely unwound, the runtime system catches the exception, prints the diagnostic, and aborts the program. The catching (and possible rethrowing) of iError for each Icon operation is handled by a *trampoline* routine. Trampoline routines are very simple. Each trampoline is a static function that takes as arguments the Icon source coordinates necessary for creating a useful diagnostic as well as the actual vDescriptors necessary to invoke the appropriate runtime method. The code below is the trampoline for assignment.

```
public static vVariable Assign(String file, int line, vDescriptor a1, vDescriptor a2) {
  try {
    return a1.Assign(a2);
  } catch (iError e) {
    e.propagate(file, line, "{$ := $}", a1, a2);
    return null;
  }
}
```

The compiler creates a call to this routine rather than invoking the Assign method directly. If Assign returns normally, its return value is bounced back to the caller. If Assign suffers a runtime error, the exception handler invokes the exception's propagate method with enough information to construct a meaningful diagnostic. If errors are being converted to failure—which is a very infrequently used Icon feature—then propagate simply returns and the trampoline fails (by producing a Java null). Otherwise, propagate simply constructs a diagnostic message for this routine, *adds* the message to the stack trace, and then re-raises itself.

Each trampoline in the Java call stack does likewise, building the stack trace, until it is ultimately caught by the runtime system's driver, which reports the message and aborts.

7.3 Co-expressions

Icon co-expressions, like co-routines, require independent runtime stacks. To get an independent stack, Jcon runs each co-expression in a separate Java thread. Co-expressions never run concurrently, so switching between co-expressions simply requires transferring control from one thread to the other.

7.4 Linking

Icon supports separate compilation and, thus, requires some facility for linking separately compiled modules together. The reference implementation of Icon has a formal linking step that brings modules together, resolves global names, and creates efficient mechanisms for accessing record fields. Jcon, on the other hand, defers almost all of this work until program execution.

Every Icon file is translated into a collection of Java class files. (Every Java class is represented by a single class file.) Each Icon procedure becomes its own class (a subclass of vProc*n*). In addition, the Jcon translator creates a special class from each Icon source file. The class subclasses the runtime system iFile class, and its job is to announce all the global declarations in the file to the runtime system at program initialization. Each iFile class announces record declarations, global variables, procedure declarations, etc., to the runtime system for linking purposes.

Because of this structure, the Jcon "linker" has few responsibilities. It must bundle all the appropriate class files together to create an executable, and it must create a Java *main* class where execution begins. This main class informs the runtime system of the iFiles that it represents, and then the runtime system queries each for its declarations. After processing all declarations, the runtime system creates a co-expression for the Icon main procedure and begins program execution.

8 The Jcon Compiler

Jcon's compiler, *Jtran*, is written in Icon. Jtran reads Icon source and produces Java class files. Jtran is a traditional syntax-directed compiler, structured as a pipeline of independent filters that translate the source code into target code.

Icon Source
↓
Preprocessor
↓ Preprocessed Icon Source
Lexical Analyzer
↓ Tokens
Parser
↓ Abstract Syntax Trees
Translator
↓ Intermediate Representation
Optimizer
↓ Intermediate Representation
Code Generator
↓
Java Class Files

Intermediate Representation	Icon
ir_Invocable(nameList, all)	invocable name1, name2 ...
ir_Link(nameList)	link file1, file2 ...
ir_Record(name, fieldList)	record name(field1, field2, ...)
ir_Global(nameList)	global variable1, variable2, ...
ir_Function(name, paramList, localList, staticList, codeList)	procedure name ... end

Table 10: Jtran Intermediate Representation for Top-level Declarations

Icon's preprocessor does simple text substitutions, file inclusions, and conditional compilation. The preprocessor consumes a source file, and produces a sequence of Icon strings that represent the preprocessed text of an Icon program.

The lexical analyzer breaks the Icon strings into a sequence of tokens, each of which is an Icon record whose type represents the lexical class of the token (integer literal, identifier, etc.). Each record contains the source file coordinates (file, line, column) where the token was recognized for use in reporting syntax errors. The lexical analyzer is responsible for inserting semicolons at the ends of some source lines [GG97]. The lexical analyzer is hand-coded.

The parser consumes tokens and produces abstract syntax trees (ASTs). The recursive-descent parser is hand-coded. The abstract syntax trees for Icon are quite simple. For the purposes of this paper, Icon's concrete syntax will be used for explanation.

The syntax-directed translator consumes ASTs and produces an Icon-specific intermediate representation (IR). The next section outlines this process in detail.

An optimizer improves the IR by eliminating trivial inefficiencies with optimizations such as constant propagation, copy propagation, and jump-to-jump removal. The optimizer not only makes resulting programs smaller and faster, it makes the *compilation* process faster, too. The time saved translating IR to Java bytecode more than makes up for the time consumed by the optimizer.

Finally, the code generator translates IR into Java class files. The process of translating IR instructions into Java Virtual Machine (JVM) instructions is accomplished through a simple macro-expansion of each IR instruction into one or more JVM instructions. The process is uncomplicated, although producing Java class files requires a fair bit of bookkeeping and attention to the JVM file format.

8.1 Translating Icon Into Intermediate Representation

Jtran's intermediate representation is a tree data structure that includes both declarative information (such as record and link declarations) and executable instructions. Table 10 lists the five IR structures for the five top-level Icon declarations: invocable, link, global, record, and procedure. Only ir_Function, which represents a compiled Icon procedure, is anything more than a simple echoing of the original Icon source. In addition to declarations of local and static variables, ir_Function contains the IR representing the executable procedure code in codeList.

The executable instructions represent a simple register-based instruction set that includes primitives for Icon's control-flow mechanisms (such as co-expression creation, procedure suspension and resumption). Table 11 lists the IR primitives, with brief descriptions of each.

AST declarations for global variables, link directives, procedure definitions, record definitions, and invocable declarations simply translate into their respective IR counterparts. The code within a procedure definition requires interesting translation into IR instructions.

8.1.1 Translating Goal-Directed Evaluation

Much of the complexity in compiling Icon results from Icon's goal-directed evaluation. Goal-directed evaluation requires that generators be started, suspended and resumed in a coordinated way to generate as many

IR Instruction	Description
ir_Var(name)	variable instance
ir_Key(name)	keyword instance
ir_IntLit(val)	integer literal
ir_RealLit(val)	real literal
ir_StrLit(val)	string literal
ir_CsetLit(val)	cset literal
ir_Tmp(name)	vDescriptor temporary
ir_Label(value)	IR label (jump target)
ir_TmpLabel(name)	IR label temporary (for indirect jumps)
ir_Chunk(label, insnList)	labeled block of IR
ir_EnterInit(startLabel)	jump to startLabel if already initialized
ir_Goto(targetLabel)	unconditional branch (direct or indirect)
ir_Deref(lhs, value)	lhs ← value.Deref()
ir_Field(lhs, expr, field, failLabel)	lhs ← expr.field
ir_OpFunction(lhs, op, argList, failLabel)	infix operator, e.g. lhs ← arg1+arg2
ir_Call(lhs, fn, argList, failLabel)	lhs ← fn(arg1, . . . , argN)
ir_ResumeValue(lhs, value, failLabel)	lhs ← *resume*(value)
ir_MakeList(lhs, valueList)	lhs ← [val1, . . . , valN]
ir_Succeed(expr, resumeLabel)	suspend expr, return expr
ir_Fail()	fail
ir_Create(lhs, startLabel)	lhs ← create expr
ir_CoRet(value, resumeLabel)	produce value from co-expression
ir_CoFail()	fail co-expression
ir_Move(lhs, rhs)	lhs ← rhs, for vDescriptors
ir_MoveLabel(lhs, rhs)	lhs ← rhs, for IR labels
ir_ScanSwap(subject, pos)	enter/exit string scanning
ir_RunTimeError()	abort—program malfunction

Table 11: Jtran Executable IR

values as possible. Translating goal-directed evaluation is, therefore, a problem of decomposing operations into their various parts and then connecting those parts appropriately.

Icon's goal-directed evaluation is concisely expressed using four *chunks* of code for each operation [Pro97]. The four-chunk technique of describing backtracking control flow is the basis for translating the control flow of generators and goal-directed evaluation. This translation technique is syntax-directed. For each operator in a program's abstract syntax tree (AST), translation produces four labeled chunks of code. In addition, each AST operator has a corresponding runtime temporary variable to hold the values it computes. Thus, the translation produces four code chunks for each operator, θ:

θ**.start** The initial code executed for the entire expression rooted at θ.

θ**.resume** The code executed for resuming the expression rooted at θ.

θ**.fail** The code executed when the expression rooted at θ fails.

θ**.succeed** The code executed when the expression rooted at θ produces a value.

The specification of these code chunks is similar to the specification of attribute grammars, except that nothing is actually computed. Instead, each code chunk is specified by a simple template. The start and resume chunks are synthesized attributes. The fail and succeed chunks are inherited attributes. Having both inherited and synthesized chunks allows control to be threaded arbitrarily among an operator and its children, which is necessary for some goal-directed operations.

Translating some Icon constructs requires determining some of the branch targets between chunks at runtime. The evaluation of some Icon operators requires additional temporary variables and code chunks. Icon expressions generate values that are held in temporary variables; the temporary for the value of AST node X is called X.val.

8.1.2 Translating Icon Values to IR

Possibly the simplest expression to translate is a simple Icon value such as a variable or a numeric literal. These values represent sequences of length one. The corresponding code immediately produces a value and exits. Upon resumption, it fails. Note that the code chunks for handling success and failure are "inherited" from an enclosing expression and therefore cannot be specified here. The code for an integer literal, N, is representative.

N

literal$_N$.start	:	ir_Move(literal.val, ir_IntLit(N))
	:	ir_Goto(ir_Label(literal.succeed))
literal$_N$.resume:		ir_Goto(ir_Label(literal.fail))

8.1.3 Binary Addition

Binary operators introduce interesting threading of control among the various code chunks. Translating $E_1 + E_2$ requires that all values of E_2 be produced for *each* value of E_1 and that the sums of those values be generated in order. Thus, resuming the addition initiates a resumption of E_2, and E_1 is resumed when E_2 fails to produce another result. Starting the addition expression requires that E_1 be started, and for each value E_1 generates, E_2 must be (re-)started (*not* resumed). The addition fails when E_1 can no longer produce results. The following specification captures the semantics cleanly.

$E_1 + E_2$

plus.start	:	ir_Goto(ir_Label(E1.start))
plus.resume	:	ir_Goto(ir_Label(E2.resume))
E_1.fail	:	ir_Goto(ir_Label(plus.fail))
E_1.succeed	:	ir_Goto(ir_Label(E2.start))
E_2.fail	:	ir_Goto(ir_Label(E1.resume))
E_2.succeed	:	ir_OpFunction(plus.val, "+", [E1.val, E2.val], ir_Label(E2.resume))
	:	ir_Goto(ir_Label(plus.succeed))

Unlike addition, a relational operator such as $>$ or ~= may fail to produce a value after its subexpressions succeed. When a comparison fails, it resumes execution of its right operand (E_2) in order to have other subexpressions to compare (i.e., it is goal-directed, and seeks success). This is what the failLabel part of the ir_OpFunction instruction controls.

8.1.4 Generators

Generators such as unary "!" are also easy to translate. (This operator generates the elements of a string or a compound data structure.) The generator is initiated with the ir_OpFunction instruction. The generator is resumed with the ir_Resume instruction, which creates any subsequent values.

$!E$

bang.start	:	ir_Goto(ir_Label(E.start))
bang.resume	:	ir_Resume(bang.val, tmp, ir_Label(E.resume))
	:	ir_Goto(ir_Label(bang.succeed))
E.fail	:	ir_Goto(ir_Label(bang.fail))
E.succeed	:	ir_OpFunction(tmp, "!", [E.val], ir_Label(E.resume))
	:	ir_Move(bang.val, tmp)
	:	ir_Goto(ir_Label(bang.succeed))

8.1.5 Conditional Control Flow

The previous translations use direct gotos to connect various chunks in a fixed fashion at compile time. For some operations this is not possible. The if expression,

if E_1 then E_2 else E_3

evaluates E_1 exactly once, simply to determine if it succeeds or fails. If E_1 succeeds then the if expression generates the E_2 sequence (and fails when E_2 fails), otherwise the if generates the E_3 sequence until failure.

Translating an if statement into the four-chunk model requires deferring the if's resumption action until runtime. If E_1 succeeds, then the if's resume action must be to resume E_2. Otherwise, the if's resume action is to resume E_3. This translates into an *indirect* goto based on a temporary value, "*gate*." E_1's succeed and fail chunks set *gate* to the appropriate chunk's—either E_1's or E_2's—resume label.

if E_1 then E_2 else E_3

ifstmt.start	:	ir_Goto(ir_Label(E1.start))
ifstmt.resume	:	ir_Goto(ir_TmpLabel(if.gate))
E_1.fail	:	ir_MoveLabel(if.gate, ir_Label(E3.resume))
	:	ir_Goto(ir_Label(E3.start))
E_1.succeed	:	ir_MoveLabel(if.gate, ir_Label(E2.resume))
	:	ir_Goto(ir_Label(E2.start))
E_2.fail	:	ir_Goto(ir_Label(if.fail))
E_2.succeed	:	ir_Move(if.val, E2.val)
	:	ir_Goto(ir_Label(if.succeed))
E_3.fail	:	ir_Goto(ir_Label(if.fail))
E_3.succeed	:	ir_Move(if.val, E3.val)
	:	ir_Goto(ir_Label(if.succeed))

8.1.6 Co-expressions

Creating co-expressions is easy because most of the work is deferred to the code generator. Co-expressions require the coordinated actions of ir_Create, ir_CoRet, and ir_CoFail, which are responsible for co-expression creation, return, and failure. The creation requires the starting address of the expression. When the expression succeeds, the co-expression must return the value, and when the expression fails, the co-expression must fail.

create E

create.start	:	ir_Create(create.val, ir_Label(E.start))
	:	ir_Goto(ir_Label(create.succeed))
create.resume	:	ir_Goto(ir_Label(create.fail))
E.fail	:	ir_CoFail()
E.succeed	:	ir_CoRet(E.val, E.resume)

8.1.7 String Scanning

String scanning requires runtime bookkeeping to maintain the correct scanning environment (i.e., the values of &subject and &pos) during execution. Entering a scanning environment *hides* the previous values of &subject and &pos while establishing new values; leaving the environment re-establishes the old values. This is tricky because string scanning environments are dynamically scoped *and* they can be exited and later re-entered as nested generators produce values. To maintain state when entering and exiting scanning environments, the compiled code for all string scanning environments includes temporary values to hold the out-of-scope &subject and &pos values. Note that these temporaries hold the outer values when within a scanning environment, but they hold the inner values when the environment has been temporarily left because of generating a value. The IR instruction ir_ScanSwap swaps the values of &subject and &pos with two temporaries. The ir_OpFunction instruction calls an internal "?:" function that is responsible for initiating string scanning.

E_1 ? E_2

scan.start	:	ir_Goto(ir_Label(E1.start))
scan.resume	:	ir_ScanSwap(ir_Tmp(sub), ir_Tmp(pos))
	:	ir_Goto(ir_Label(E2.resume))
E_1.fail	:	ir_Goto(ir_Label(scan.fail))
E_1.succeed	:	ir_Deref(ir_Tmp(sub), ir_Key("subject"))
	:	ir_Deref(ir_Tmp(pos), ir_Key("pos"))
	:	ir_OpFunction(scan.val, "?:", [ir_Key("subject"), ir_Tmp(sub)], ir_Label(E1.resume))
	:	ir_Goto(ir_Label(E2.start))
E_2.fail	:	ir_Move(ir_Key("subject"), ir_Tmp(sub))
	:	ir_Move(ir_Key("pos"), ir_Tmp(pos)
	:	ir_Goto(ir_Label(E1.resume))
E_2.succeed	:	ir_ScanSwap(ir_Tmp(sub), ir_Tmp(pos))
	:	ir_Goto(ir_Label(scan.succeed))

It would have been convenient (and a cleaner design) if the same swapping could have been accomplished with a more general IR instruction for swapping values. Unfortunately, because assignment to &subject also affects &pos, their values must be swapped in a coordinated way.

8.1.8 Procedure Return/Suspend/Fail

Procedure return, suspension, and failure are handled with ir_Succeed and ir_Fail. Because suspend's expression may generate many values, ir_Succeed is given the address at which to resume the expression. Icon returns, which generate only a single value and terminate the called procedure, lack a resumption address.

If a suspend is lexically nested within one or more string scanning environments, it is necessary to restore the appropriate values of &subject and &pos when exiting the environments and when re-entering them upon resumption. The ir_ScanSwap instructions cooperate to maintain the environments.

suspend E

suspend.start	:	ir_Goto(ir_Label(E.start))
suspend.resume	:	ir_Goto(ir_Label(suspend.fail))
suspend.restore	:	ir_ScanSwap(ir_Tmp(subject), ir_Tmp(pos)) // if necessary
	:	ir_Goto(ir_Label(E.resume))
E.fail	:	ir_Goto(ir_Label(suspend.fail))
E.succeed	:	ir_ScanSwap(ir_Tmp(subject), ir_Tmp(pos)) // if necessary
	:	ir_Succeed(E.val, ir_Label(E.resume))

fail		
fail.start	:	ir_ScanSwap(ir_Tmp(subject), ir_Tmp(pos)) // if necessary
	:	ir_Fail()
fail.resume	:	ir_RunTimeError()

8.1.9 Procedure Bodies

Jtran compiles Icon procedure bodies into two code chunks: one for the *initial* block, if one exists, and one for the procedure body. The initial block must be executed only once, upon the first execution of the procedure. The IR instruction ir_EnterInit guards the execution of the initial block.

procedure *init body* end		
proc.start	:	ir_EnterInit(ir_Label(body.start))
	:	ir_Goto(ir_Label(init.start))
proc.resume	:	ir_RunTimeError()
init.fail	:	ir_Goto(ir_Label(body.start))
init.succeed	:	ir_Goto(ir_Label(body.start))
body.fail	:	ir_Fail()
body.succeed	:	ir_Fail()

8.1.10 Loops

Translating the various Icon looping constructs is straightforward. For instance, the every E do B loop, which executes B for every value generated by E, simply requires that the success and failure ports of B direct execution to E's resumption port. What complicates loop translation is the possibility of abnormal loop control via break and next. Both break and next leave the loop body and either exit the loop (break) or resume the loop at the beginning (next). Both operations may exit string scanning environments, which would require re-establishing the hidden values for &subject and &pos.

The break E statement poses another complication because the value of its (optional) expression represents the value of the enclosing loop. This expression may be a generator. This means that resuming the loop to generate more values requires resuming break's expression, E. Because a loop may contain multiple breaks, this resumption address cannot be known statically and therefore must be kept in a runtime temporary, loop.continue.

every E do B		
every.start	:	ir_Goto(ir_Label(E.start))
every.resume	:	ir_Goto(ir_Label(every.continue))
every.next	:	ir_Goto(ir_Label(E.resume))
E.fail	:	ir_Goto(ir_Label(every.fail))
E.succeed	:	ir_Goto(ir_Label(B.start))
B.fail	:	ir_Goto(ir_Label(E.resume))
B.succeed	:	ir_Goto(ir_Label(E.resume))

next		
next.start	:	*Escape scanning environment(s) if necessary*
	:	ir_Goto(ir_Label(loop.next))
next.resume	:	ir_RunTimeError()

break E

break.start	:	*Escape scanning environment(s) if necessary*
	:	ir_MoveLabel(loop.continue, break.resume)
	:	ir_Goto(ir_Label(E.start))
break.resume	:	ir_Goto(ir_Label(E.resume))
E.fail	:	ir_Goto(ir_Label(loop.fail))
E.succeed	:	ir_Move(ir_Tmp(loop.val), ir_Tmp(E.val)
	:	ir_Goto(ir_Label(loop.start))

8.1.11 Default Values

Many Icon control constructs have optional clauses; for example, an if has an optional else clause. There is an equivalent default value for every optional clause. The following are equivalent in Icon:

```
if E1 then E2
if E1 then E2 else &fail
```

A simple transformation pass adds default values to ASTs so that subsequent translation to IR operates on fully populated ASTs.

8.2 Translation of IR to Java Bytecode

8.2.1 Translating Declarations to Bytecode

Translating IR into Java bytecode is straightforward. Each Icon file is translated into many classes, one for the file as a whole and one or two for each procedure. An iFile subclass announces link, invocable, global, procedure and record declarations to the runtime system. Each Icon procedure is translated into a Java class that subclasses vProc*n* (where *n* is the number of arguments expected) and defines a Call method. Procedures that suspend require an additional vClosure class.

The iFile subclass is simply a collection of methods that pass declarations to the runtime system. Consider the following Icon program, foo.icn.

```
record R(f)
global G
procedure P(a)
  write(a, 1, 3.14, "string", 'cset')
end
```

This program is translated into an iFile subclass for the file foo and a vProc1 subclass for procedure P. The iFile subclass follows.

```
class l$foo extends iFile {
  public static vVariable v$write$;
  public static vReal lr$3_14 = vReal.New("3.14");
  public static vInteger li$1 = vInteger.New("1");
  public static vString ls$1 = vString.New("string");
  public static vString lc$1 = vCset.New("cset");

  void unresolved() {
    iEnv.undeclared("write");
  }

  void declare() {
    iEnv.declareGlobal("G");
```

```
    iEnv.declareProcedure("P", new p_l$foo$P());
    String[] v = {"f" };
    iEnv.declareRecord("R", v);
  }
  void resolve() {
    v$write$ = iEnv.resolve("write");
  }
}
```

The unresolved method informs the runtime system of all the identifiers used in procedure code that are not locally bound. Method declare announces to the runtime system all of the global values in the corresponding file. The resolve method queries the runtime system for bindings of the non-local identifiers. (In Icon, identifiers that are not locally defined default to local scope if they are not declared globally.) When the runtime system is handed a list of the iFiles that represent an application, it invokes all of the iFiles's unresolved methods, followed by all declare methods, and finally every resolve method. These routines cooperate to declare and resolve global bindings.

8.2.2 Translating Procedures to Bytecode

Translating procedures—represented with the IR declaration ir_Function—into bytecode requires generating some simple prologue code as well as translating the executable IR. Each procedure is translated into either a vProc subclass, or a coordinated pair of vProc and vClosure subclasses depending on whether or not the procedure can ever suspend execution (via suspend or as a co-expression). In either case, the prologue is similar. The prologue is responsible for allocating the vLocalVar instances for each parameter and local variable. Furthermore, the prologue determines which undeclared variables are declared globally. Those bound globally use the global instance, and undeclared variables default to local instances. Static variables are allocated during class initialization. The prologue loads references to all variables referenced in a procedure into JVM local variables for quick access.

If a procedure requires indirect jumps, the prologue also includes a procedure-wide tableswitch instruction that maps integers to Java bytecode locations that correspond to IR labels. This is necessary because Java bytecode does not permit indirect jumps to program locations.

To make this more concrete, an example translation of two procedures follows. While Jtran translates Icon directly into Java bytecode, these translations show equivalent Java source, which is more readable. (To be faithful to the actual bytecode, we have added goto—a construct absent from Java, but necessary to represent the actual bytecode control flow. We also use local variables where the generated code uses the JVM's evaluation stack.) Note that the use of trampoline routines eliminates direct calls on class methods.

The first example routine, Y, simply adds a parameter to 7 and returns that value:

```
procedure Y(x)
  return x + 7
end
```

Because it takes one parameter and includes no indirect jumps or suspensions it is translated directly into a vProc1, with a single-argument Call method. This Call method dereferences its argument and then invokes the Add method with the argument li$7, which is a static variable that holds the vInteger that represents the literal value 7. Because Jtran's code generator translates each IR instruction in isolation, it cannot know that the value being returned is not a variable and, therefore, must assume that it may be a variable. This conservative assumption means that DerefLocal must be applied to the return value.

```
public final class p_l$foo$Y extends vProc1 {
  vDescriptor Call(vDescriptor arg0) {
    vDescriptor tmp1 = arg0.Deref();
    vDescriptor tmp0 = iTrampoline.Add("foo.icn", 2, tmp1, l$foo.li$7);
```

```
    if (tmp0 == null) return null;
    return tmp0.DerefLocal();
  }
}
```

Procedure X below suspends every value produced by Y(3). Suspension requires a vClosure as well as a means to restart its Resume method at appropriate points.

```
procedure X()
  suspend Y(3)
end
```

Therefore, the code generator emits two classes for X: a simple vProc0 for the original call and a vClosure that does all the actual work.

The Call method first resolves the reference to Y. If it is globally defined, a static variable (of the iFile class) holds the variable, but otherwise it is necessary to create a vLocalVar instance. Next, the Call creates a vClosure instance that does the appropriate computation in its Resume method.

```
public final class p_l$foo$X extends vProc0 {
  vDescriptor Call() {
    vDescriptor[] vars = new vDescriptor[1];
    vDescriptor Y = l$foo.v$Y$;
    vars[0] = (Y != null) ? Y : vLocalVar.NewLocal("Y");
    c_l$foo$x closure = new c_l$foo$x(vars);
    vDescriptor val = closure.Resume();
    if (val == null) return null;
    closure.retval = val;
    return closure;
  }
}
```

Of course, the computation done in the Resume method of X's vClosure is complicated by the fact that it must be able to suspend and resume execution at different program points. Suspending execution requires storing away the values of temporaries (ir_Tmp and ir_TmpLabel values) as well as local variables and parameters. These values are stored in two arrays, tmpArray and tmpVarArray, that are fields of the vClosure instance. The program location at which the Resume method should begin is kept in the PC field. After restoring state from the temporary arrays, Resume uses a switch statement to direct execution to the appropriate program location.

```
public final class c_l$foo$X extends vClosure {
  int PC;
  vDescriptor[] tmpArray[];
  vDescriptor[] tmpVarArray[];
   vDescriptor Resume() {
    vDescriptor tmp2 = tmpArray[2];
    vDescriptor tmp1 = tmpArray[1];
    vDescriptor tmp0 = tmpArray[0];
    vDescriptor Y = tmpVarArray[0];
    switch (PC) {
    case 1:
      tmp2 = Y;
      tmp2 = tmp2.Deref();
      vDescriptor stk0 = iTrampoline.Call("foo.icn", 5, tmp2, l$foo.li$3);
      if (stk0 == null) return null;
```

```
    tmp1 = stk0;
    tmp0 = tmp1;
  L87:
    PC = 2;
    tmpArray[2] = tmp2;
    tmpArray[1] = tmp1;
    tmpArray[0] = tmp0;
    return tmp0.DerefLocal();
  case 2:
    vDescriptor stk1 = iTrampoline.Resume("foo.icn", 5, tmp1);
    if (stk1 == null) return null;
    tmp0 = stk1;
    goto L87;
  default:
    return null;
  }
 }
 public c_l$foo$X(vDescriptor[] vars) {
  tmpVarArray = vars;
  PC = 1;
  tmpArray = new vDescriptor[3];
 }
}
```

Careful examination of the code for procedures X and Y reveals inefficiencies in the generated code. These inefficiencies all stem from weak analysis and optimization of either the IR or the generated code. For instance, there are unnecessary copies of temporary values. Furthermore, more temporaries than necessary exist in X. To make matters worse, no analysis is done to determine which temporaries actually need to be saved and restored between calls to Resume—the compiler conservatively stores all of them.

The following sections give a detailed account of translating each of the IR operators.

8.2.3 Values

Translating the executable IR that represents an Icon procedure's code is done via simple macro-expansion—each IR instruction is translated into one or more bytecode instructions. If a procedure never suspends, a vProc suffices for the realization of the procedure's executable code, but an additional vClosure is necessary if the procedure must be able to suspend execution to generate a value. If no closure is needed, then the procedure's executable code is translated into the Call method of the vProc. Otherwise, the vProc's Call method simply invokes the Resume method of a vClosure object for this procedure that does all the work. In either case, the translation is straightforward.

Table 12 gives the simple macro-expansion of each IR operation that represents a value. These operations get the appropriate value and push it onto the JVM's evaluation stack. The compiler translates all literal values (such as ir_RealLit) into JVM static fields of the iFile class for each source file. These static fields are set during class initialization and then accessed directly during program execution. Runtime values such as variables (except statics) and temporaries are accessed as JVM local variables. Static variables are allocated as vProc static fields. For every Icon keyword there is an associated static field in the runtime system that is accessed directly in generated code. Labels are translated to simple integers that can be used to index a JVM switch statement for control flow.

IR	Java Bytecode
ir_Var(name)	aload *name*
ir_Key(name)	getstatic *name* invokevirtual vDescriptor Call()
ir_IntLit(val)	getstatic li$*val*
ir_RealLit(val)	getstatic lr$*val*
ir_StrLit(val)	getstatic ls$*val*
ir_CsetLit(val)	getstatic lc$*val*
ir_Tmp(name)	aload *name*
ir_Label(value)	ipush *value*
ir_TmpLabel(name)	iload *name*

Table 12: Translating IR Values into Java Bytecode

8.2.4 Simple Operations

Table 13 gives the translation for the IR instructions that represent simple operations such as gotos and moves.

The JVM does not support indirect jumps. To implement the indirect jumps in the Jtran IR, it is necessary to represent labels as integers, and to use one of the switch dispatching instructions in the Java bytecode to do the actual control transfer. Thus, if the translation of IR requires an indirect jump, Jtran creates a single switch construct through which all indirect jumps are directed. Direct jumps are implemented directly as Java bytecode gotos.

The ir_EnterInit instruction accesses the per-vProc static field that guards execution of a procedure's initial expression.

The ir_Deref and ir_MakeList instructions, which dereference values and construct lists, just push their arguments and call the appropriate runtime system routines.

8.2.5 Source-level Operations

Table 14 outlines the translation of ir_Field, ir_OpFunction, ir_Call, and ir_Resume into Java bytecode. Each of these operations represents a call on a trampoline routine in the runtime system. Every possible ir_OpFunction operation is translated into the appropriate trampoline routine; for example, binary "+" uses the iTrampoline.Add method. Also, each of these operations can fail (i.e., return null) and therefore represents a conditional control flow operation. Although not shown in the table, each trampoline also takes file name and line number arguments for error reporting purposes.

It is often the case that Icon expressions return values that are never used. When the compiler determines that this is the case, it omits the lhs field of an IR operation, which is a signal to the code generator to avoid emitting code for that assignment. Similarly, many Icon expressions continue execution at the same location regardless of whether they succeed or fail. In these situations, the compiler omits the failLabel field of the IR instruction and the code generator omits the conditional jump.

8.2.6 Procedural Operations

Table 15 gives the translations of ir_Succeed and ir_Fail into Java bytecode. These operations represent the mechanisms by which a call to an Icon procedure may return execution to the caller. Failure simply requires returning a null value. Success, however, requires suspending the current computation by saving temporary values (JVM local variables) and the resumption PC in the vClosure for a subsequent call to Resume.

IR	Java Bytecode
ir_EnterInit(startLabel)	getstatic initialized ifne *startlabel* iconst_1 putstatic initialized
ir_Goto(ir_Tmp(name))	goto *name*
ir_Goto(ir_TmpLabel(name))	*push ir_TmpLabel(name)* goto switchstatement
ir_MoveLabel(ir_TmpLabel(name), rhs)	*push rhs* istore *name*
ir_Move(ir_Tmp(name), rhs)	*push rhs* astore *name*
ir_Move(ir_Var(name), rhs) *or* ir_Move(ir_Key(name), rhs)	*push rhs* *push lhs* swap invokevirtual vDescriptor Assign() pop
ir_Deref(lhs, value)	*push value* invokevirtual vValue Deref() *Move to lhs*
ir_MakeList(lhs, valueList)	*push* $value_1$ ⋮ *push* $value_N$ invokestatic vDescriptor MakeList(vDescriptor, ...) *Move to lhs*

Table 13: Translating IR Utility Instructions to Java Bytecode

8.2.7 Co-expression Operations

Table 16 outlines the translation of IR instructions related to co-expressions. Co-expression returns and failures are little more than calls on runtime system routines. Co-expression creation, however, requires the creation of a newly cloned instance of the currently executing vClosure.

8.2.8 Miscellaneous

Table 17 includes the translations of the remaining IR instruction into Java bytecode. ir_Chunk requires no translation other than the translation of its instruction list. ir_ScanSwap swaps the values of &subject and &pos with temporary values via a sequence of references and updates. ir_RunTimeError is a simple call on the runtime system followed by a little bit of code that is never reached—its sole purpose is to quiet the JVM code verifier that does not know the code is unreachable. A executing program should *never* reach an ir_RunTimeError; reaching one indicates a Jcon-system malfunction (as opposed to a errant user program).

9 Implementation Status

Jcon is a mostly complete implementation of Icon, omitting only a few features that cannot be written in Java. Co-expressions, large integers, and pipes are provided, and a preprocessor is included. String invocation is supported. Tracing, error recovery, and debugging functions are all included, although for performance reasons they are disabled by default.

IR	Java Bytecode
ir_Field(lhs, expr, field, failLabel)	*push expr* ldc *"field"* invokestatic vValue Field(String) dup ifnonnull *L* pop goto *failLabel* *L:* *Move to lhs*
ir_OpFunction(lhs, op, argList, failLabel)	*push* arg_1 ⋮ *push* arg_N invokestatic vDescriptor *Operation*(vDescriptor, ...) dup ifnonnull *L* pop goto *failLabel* *L:* *Move to lhs*
ir_Call(lhs, fn, argList, failLabel)	*push fn* *push* arg_1 ⋮ *push* arg_N invokestatic vDescriptor Call(vDescriptor, ...) dup ifnonnull *L* pop goto *failLabel* *L:* *Move to lhs*
ir_ResumeValue(lhs, value, failLabel)	*push value* invokestatic vDescriptor Resume() dup ifnonnull *L* pop goto *failLabel* *L:* *Move to lhs*

Table 14: Translating IR Computation Instructions to Java Bytecode

IR	Java Bytecode
ir_Succeed(expr, resumeLabel)	aconst_0 *push resumeLabel* putfield PC *save temporaries in this closure* *push expr* invokevirtual vDescriptor DerefLocal() areturn
ir_Fail()	aconst_null areturn

Table 15: Translating IR Procedure Instructions to Java Bytecode

IR	Java Bytecode
ir_Create(lhs, startLabel)	new *vClosure for this procedure* dup aload_0 getfield vDescriptor[] tmpVarArray *push startLabel* invokespecial <init> invokestatic void vCoexp.New() dup invokevirtual create() *Move to lhs*
ir_CoRet(value, resumeLabel)	getstatic iEnv.cur_coexp *push value* invokevirtual coret() *goto resumeLabel*
ir_CoFail()	getstatic iEnv.cur_coexp invokevirtual cofail() aconst_null // for verifier areturn

Table 16: Translating IR Co-expression Instructions to Java Bytecode

The core Icon language is defined by *The Icon Programming Language* [GG97]. Jcon follows this specification closely, implementing all the operators and all the predefined procedures except chdir(), getch(), getche(), and kbhit(). There are a few I/O differences: &input does not support random access, &errout is always unbuffered, and input from a pipe is not available until the process finishes. Keywords related to memory allocation always produce zeroes, and &time reports wall-clock time instead of CPU time. All of these differences are related to limitations of Java and affect only a small fraction of Icon programs.

Dynamic loading differs from the reference implementation. In Version 9 for Unix, C functions are loaded from a shared library. In Jcon, Java classes are loaded from a Zip archive or Jar file. Both systems require that the loaded procedures conform to a specified interface. Jcon offers the additional feature of compiling Icon code into loadable procedures; a running program can generate Icon source code, spawn a jcont process to compile it, then load and execute the result.

Jcon includes almost all of Icon's standard graphics facilities as defined by *Graphics Programming in Icon* [GJT97]. Line width control, textured drawing, and mutable colors are the most notable exceptions.

IR	Java Bytecode
ir_ScanSwap(subject, pos)	*push &pos* invokevirtual vValue Deref() *push &subject* invokevirtual vValue Deref() *push &subject* aload *subjectTemporary* invokevirtual vDescriptor Assign() pop *push &pos* aload *posTemporary* invokevirtual vDescriptor Assign() pop astore *subjectTemporary* astore *posTemporary*
ir_RunTimeError()	ipush 902 invokestatic Error aconst_null // for verifier areturn
ir_Chunk(label, insnList)	Labeled block of IR

Table 17: Translating Miscellaneous IR Instructions to Java Bytecode

There are other minor omissions as well as several differences in system-dependent areas that already vary between the Unix and Windows implementations of Version 9. Jcon's minor additions include the ability to read and write JPEG images and to display more than 256 simultaneous colors under Unix.

Because Java supports the Unicode character set [Uni], it would seem natural for Jcon to be a Unicode-based version of Icon. We have not done this because we did not wish to address the many language issues that would arise as a consequence. Icon is specified in terms of an eight-bit character set and the many character-set keywords are defined accordingly. Changing these things could invalidate many existing programs, probably in unforseen ways.

However, Jcon would provide a good foundation for a Unicode version of Icon. Changing Jcon to use 16-bit characters would be relatively straightforward. The most interesting implementation challenge would be deciding how to implement csets: Simply extending the current 256-bit vectors to 65536-bit vectors might not be the best approach.

10 Performance

Programs built by Jcon are slower than those built by Version 9 of Icon. Startup delays are significant, and execution is slower. These differences are attributed to the Java language and the quality of currently available Java implementations.

When measuring Jcon's performance, we select the Jtran compiler option that makes direct runtime calls instead of using trampolines. This gives faster execution by disabling a few diagnostic features (including tracing and detailed error messages) but does not otherwise affect program semantics.

10.1 Startup Costs

Java systems typically impose a noticeable delay when starting up any Java program. This is due partly to the design of the Java language, which specifies on-demand loading and initialization as classes are refer-

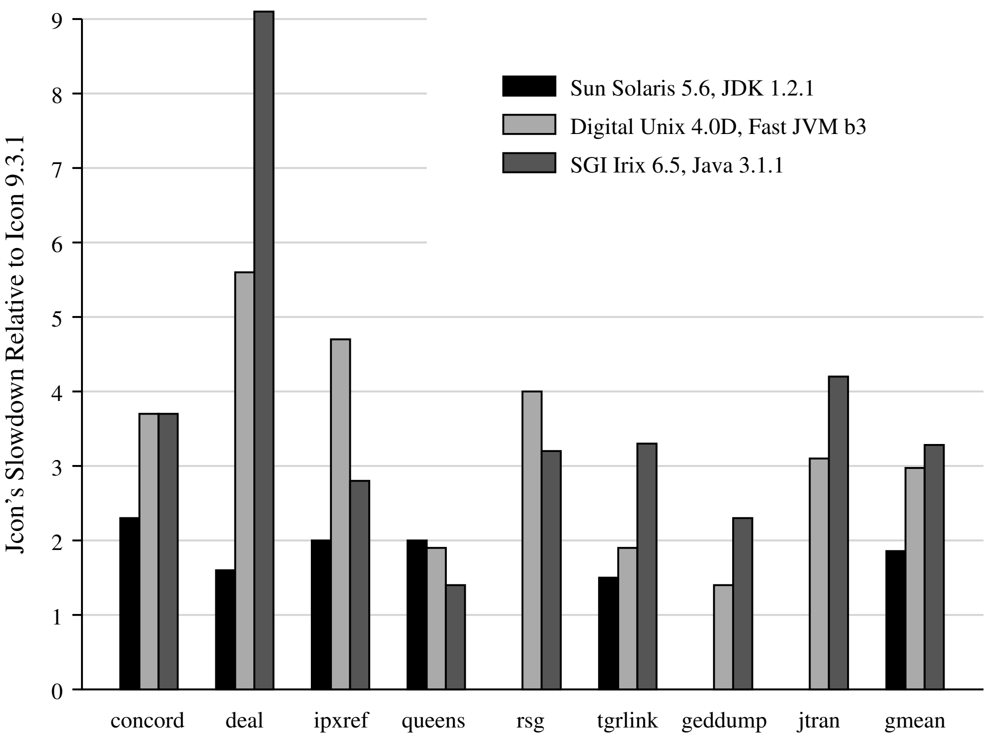

Table 18: Execution Time relative to Icon 9.3.1

enced. Just-in-time compilers can add additional front-end costs that are not always regained through faster execution.

Some figures from a Sun system illustrate this startup cost. A minimal "hello world" program executes in a barely measurable 0.01 seconds when written in C. Version 9 executes the equivalent Icon program in 0.03 seconds. But a "hello world" program written in Java takes 1.15 seconds to execute, and the Icon version takes 1.22 seconds when built by Jcon.

10.2 Execution Speed

Current Java implementations run a typical Icon program at one half to one fourth the speed of Version 9 of Icon. The slowdown factor varies with both the Icon program and the Java platform; the same program may run quickly on one Java platform and slowly on another.

Table 18 shows the execution times for the standard Icon benchmarks and for three additional long-running applications. Each time has been normalized relative to a Version 9 execution time of 1.0. The benchmark programs are as follows:

concord	produces a text concordance (a word index)
deal	deals bridge hands
ipxref	cross-references Icon programs
queens	places non-attacking queens on a chessboard
rsg	generates random sentences
tgrlink	optimizes vectors for drawing street maps
geddump	formats and prints a genealogical data base
jtran	translates Icon into Java class files

The final cluster on the chart displays the geometric means of the other benchmarks.

The standard benchmark programs were taken from Icon version 9.3.1 and run unmodified, but to prevent startup costs from dominating, some data files and command options were changed to make them run longer. Each program was run both by Jcon and by the reference implementation, and user time was recorded.

All the Java systems tested incorporate just-in-time (JIT) compilers; three data values are missing because of bugs in one JIT compiler. All programs were run with default memory allocation settings except for geddump, which required an increased Java memory pool to complete.

10.3 Analysis

Version 9 of Icon presents a challenging baseline for evaluating Jcon's speed. It is written in C and has been carefully honed by several programmers over a span of many years. The better it performs, the harder it is for Jcon to look good.

We spent a significant amount of time working to improve the performance of the Jcon system. However, the system is still not as fast as it might be. We consciously rejected some ideas that seemed to add too much complexity. We also set aside some reasonable ideas that promised modest gains but did not seem urgent. The resulting system is somewhat of a compromise, but one with which we are comfortable. It could certainly be further improved, but probably not by a large amount.

Performance tuning was sometimes frustrating. We used profiling to identify costly operations and altered the code or the data structures when this could be done without adding undue complication. This usually worked well, but sometimes we would make a local improvement only to find an overall slowdown due to increased execution time in unrelated areas. One can postulate plausible reasons such as cache effects or asynchronous garbage collection, but we did not find a truly satisfying explanation.

Hot spots revealed by program profiling are not always due to weak areas of the language implementation; more often they are characteristic of the program being examined. The hot spots of the benchmark programs show a wide variety:

Program	Peak	Expression
concord	27%	`tab(many(&digits))`
deal	51%	`every !s :=: ?s`
ipxref	32%	`s1 == s2`
queens	39%	`L[i]` used as value
rsg	35%	`L ||| L`
tgrlink	19%	`right(i, j)`
geddump	8%	`R . f`

The "Peak" column gives the percentage of execution time attributable to the most active Icon expression when run by Jcon on an SGI system. Some of these expressions are more expensive than others, but analysis of the programs shows that they dominate the profiles mainly because of the number of times they are performed. The presence of program hot spots is another reason to mistrust benchmarks as representative of overall system performance; but at least these seven cases are reasonably dissimilar.

The performance of a Jcon-built program, or any Java program, depends heavily on the quality of the underlying Java system. We have seen significant speedups over a span of just one year as vendors released new and better Java implementations. These improvements have not come risk-free: as the systems grow faster and more complex, the incidence of bugs seems to be increasing. Jcon is perhaps especially prone to exposing bugs because it generates some legal code patterns that are never seen as the output of a Java compiler. Still, the performance trend is encouraging, and we look forward to further improvements.

11 Java as a Language Platform

For some programming tasks, object-oriented programming is mildly useful or perhaps just irrelevant. Without the right challenge, new Java programmers may wonder what all the fuss is about. That is not the case here. We found that object-oriented programming made a truly spectacular difference to the Jcon runtime system. The leverage provided by method inheritance and overloading made the programming much easier and faster. It also allowed a truly elegant solution to the handling of suspended generators that adds no cost to the normal case.

Java's automatic memory management was also a great benefit. It allowed Jcon to dispense with all the memory management and garbage collection code that complicates the runtime system of the reference implementation. As we are long-time Icon programmers, this was no surprise, for Icon also provides automatic garbage collection.

A quantitative measure of the difference is available. The Jcon runtime system comprises 18,000 lines of Java code. The corresponding portion of the reference implementation, counting only the code used under Unix, is over 50,000 lines of C code. While there are other, minor factors, the increased power of Java accounts for the overwhelming majority of this difference.

Application packaging is an issue that has not been well addressed by Java implementations. There is no standard way to compile a Java program into a self-contained executable that can be treated just like a Unix a.out file. Java applications are often distributed as packages of several files directed by a shell script; the javac compiler is a typical example. However, Icon programmers expect an Icon compiler to produce a single, self-contained executable file.

Our solution was to make Jcon bundle a script with the generated class files in a single output file. This file looks superficially like a Korn shell script and is just as easily executed. When run, though, the script extracts an archive of Java class files from the back of its own file and then calls Java to execute that. This is not an ideal solution, but it does provide a self-contained executable that can be copied or renamed without special consideration. Jcon also provides an option to generate a standard Java archive instead.

In working on several Java-based projects we have been pleasantly surprised at the overall reliability and robustness of the early Java implementations. We encountered no show-stopping bugs despite the relative immaturity of these Java systems. Recently, though, we have seen a disturbing trend: two of the new just-in-time compilers exhibit segmentation faults and other catastrophic failures when running Jcon-generated code. It may be that simple and straightforward interpreters are giving way to complex and aggressive compilers that make assumptions that only hold for code generated by the vendor's own Java compiler. This is clearly wrong, according to the Java virtual machine specification[LY97], and we hope that it will not persist.

Overall, we found Java to be a good platform for re-implementing Icon. Java has already established itself as a mainstream language that is widely available. Its object-oriented features and automatic memory management led to a much simpler implementation. As the technology matures we expect to see further performance improvements.

11.1 Java Bytecode as a Target Language

Java bytecode is designed as a target for the Java source language. As such, it omits features that cannot be expressed in Java source. Two omitted features would have been useful for generating more efficient code from Icon sources: static method references and reference arithmetic.

Static method references—the ability to capture a reference to a method in a variable or field—is a standard technique of higher-order programming. Java (and Java bytecode) do not allow a method reference to be captured; therefore, to parameterize a value by the functioning of a particular method, it is necessary to create a new class that overrides the method behavior. While this works, it comes at the cost of creating an entirely new class for what may be a single instance. In the Jcon implementation, this happens with subclasses of vProc*n* that differ only in the code of their Call methods (and similarly for subclasses of vClosure). For each new subclass, standard Java compilation techniques build a complete virtual method dispatch table that consists of pointers to every virtual method—for vDescriptor subclasses, this is over 100 pointers. Often the new method is much smaller than the additional virtual table. With static method references, it would be possible to parameterize instances of a single subclass so that virtual tables could be shared.

Java enforces the integrity of references—it is illegal to do arithmetic on an object reference. Unfortunately, this means that it is impossible to create *tagged* integers that share storage space with object references. (Tagged integers are distinguished from references by virtue of having some *tag* bit(s) set that references cannot have set.) Tagged integers avoid the necessity of keeping small integers in the heap, and therefore can greatly speed up arithmetic-intensive applications. While there are obvious safety concerns with respect to using tagged integers, they are beneficial in some domains.

12 Related Work

12.1 Compilation Techniques

Independently, Byrd, and Finkel and Solomon developed a four-port model for describing control flow [Byr80, FS80, Fin96]. Byrd used the four-port box to describe Prolog control flow, but it is not clear whether it was for translation purposes or for debugging purposes [Byr80]. It appears that Byrd used the boxes to model control flow between calls within a single clause, but not to model the flow of control between clauses within a procedure, nor to model the control flow in and out of a procedure. Finkel and Solomon used their four-port scheme to describe power loops. Power loops backtrack and thus the start/succeed/resume/fail model describes their behavior well. Unlike Prolog, however, power loops cannot be described by a simple sequential connection of four-port boxes. In neither case was the idea of four-ports generalized into a mechanism for describing how four pieces of code might be generated and stitched together for various operators in a goal-directed language.

The reference Icon translation system, which translates Icon into a bytecode for interpretation, controls goal-directed evaluation by maintaining a stack of *generator* frames that indicate, among other things, what action should be taken upon failure [GG86, ALS78]. Special bytecodes act to manipulate this stack—by pushing, popping or modifying generator frames—to achieve the desired goal-directed behavior. Icon's reference implementation is an interpreter that consumes bytecode for the Icon Virtual Machine. The Icon VM is stack based and relies on generator frames to control goal-directed evaluation. Jcon's IR is based on explicit temporaries, which allows for the generation of more efficient code for accessing temporary values. Also, Jcon's avoidance of generator frames provides a more direct realization of goal-directed evaluation. The four-port scheme requires nothing more powerful than conditional, direct, and indirect jumps.

O'Bagy and Griswold developed a technique for translating Icon that utilizes *recursive interpreters* [OG87]. The basic idea behind recursive interpreters for goal-directed evaluation is that each generator that produces a value does so by recursively invoking the interpreter. Doing so preserves (*suspends*) the generator's state for possible resumption when the just-invoked interpreter returns. A recursively invoked interpreter's return value indicates whether the suspended generator should resume or fail. O'Bagy's interpreter executes the same bytecode as the original Icon interpreter. Jcon uses its vClosure mechanism for suspending generators.

Gudeman developed a goal-directed evaluation mechanism that uses *continuation-passing* to direct control flow [Gud92]. Different continuations for failure and success are maintained for each generator. While continuations can be compiled into efficient code they are notoriously difficult to understand, and few target languages directly support them.

Walker created an Icon-to-C compiler that used the reference compiler's runtime system [Wal91]. By doing extensive type inference on Icon source programs, the compiler could generate programs that avoided unnecessary type checks and type conversions. These optimizations can significantly increase the speed of an Icon program.

12.2 Runtime System

The runtime system of Icon's reference implementation is structured differently than Jcon's [GG86]. Where Jcon's runtime is object-oriented, relying heavily on dynamic method dispatch to bind operations to objects of a particular type, the reference implementation takes an operation-centric approach in C. Each operation (indexing, size-of, etc.) is conceptually implemented via a single function. This function explicitly checks the dynamic types of its arguments and does necessary coercions and then proceeds with the action appropriate for the given types. This checking and coercion code contributes a redundancy absent in Jcon's object-oriented implementation.

The SNOBOL4 implementation employs "trapped variables" for variables which, when read or written, execute code for side effects. Hanson elevated trapped variables to source-level constructs for programmer manipulation [Han76]. The reference Icon implementation uses a similar construct for implementing many keywords [GG86]. Jcon's use of vVariable subclasses for similar purposes generalizes this technique.

12.3 Other Java Virtual Machine Targets

Creating JVM-based implementations of non-Java languages is becoming quite popular. Many implementations of well-known languages have targeted the JVM, including Scheme, ML, Ada 95, COBOL, and Pascal. [JVM].

Meehan and Joy targeted the JVM from the lazy functional language, Ginger [MJ99]. Through a clever use of Java's *reflection* mechanisms, they were able to simulate static method references, which allowed them to avoid creating a new class for every new function they wished to add to their system. Unlike Jcon, this system did not create its own class hierarchy for all of its data types, but rather used Java's types such as java.lang.Integer wherever possible. This has the advantage of re-use, but suffers from the need to do type discrimination via explicit type tests rather than via the efficient method invocation mechanism that Jcon uses.

13 Conclusion

Jcon represents a novel new implementation of the Icon programming language. The compiler utilizes a simple and efficient four-chunk mechanism for controlling goal-directed evaluation. The runtime system employs an object-oriented architecture for handling Icon's dynamic typing. This new runtime system architecture has resulted in a cleaner and significantly smaller implementation than previous techniques. Furthermore, the object-oriented architecture is applicable to any dynamically typed language implementation (such as LISP, Perl, etc.).

14 Availability

Jcon is freely available from http://www.cs.arizona.edu/icon/jcon/. Its documentation is viewable on-line, including the complete list of differences with respect to Version 9 of Icon.

15 Acknowledgments

Denise Todd Smith implemented the compiler's lexical analyzer and parser. Bob Alexander implemented the compiler preprocessor. Ralph Griswold offered valuable advice. This research is partially supported by grants from IBM Corp., ATT Foundation, NSF (CCR-9502397, CCR-9415932), and ARPA (N66001-96-C-8518, DABJ-63-85-C-0075).

References

[ALS78] Russell R. Atkinson, Barbara H. Liskov, and Robert W. Scheifler. Aspects of implementing CLU. In *Proceedings of the ACM National Conference*, pages 123–129, December 1978.

[Byr80] Lawrence Byrd. Understanding the control of Prolog programs. Technical Report 151, University of Edinburgh, 1980.

[Fin96] Raphael A. Finkel. *Advanced Programming Language Design*. Addison-Wesley Publishing Company, 1996. ISBN 0-8053-1192-0.

[FS80] Raphael Finkel and Marvin Solomon. Nested iterators and recursive backtracking. Technical Report 388, University of Wisconsin-Madison, June 1980.

[GG86] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, 1986.

[GG96] Ralph E. Griswold and Madge T. Griswold. History of the Icon programming language. In Thomas J. Bergin and Richard G. Gibson, editors, *History of Programming Languages*, pages 599–621. Addison-Wesley, 1996.

[GG97] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, third edition, 1997.

[GJT97] Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend. *Graphics Programming in Icon*. Peer-to-Peer Communications, 1997.

[GPP71] R. Griswold, J. Poage, and I. Polonsky. *The Snobol 4 Programming Language.* Prentice-Hall, 1971.

[Gud92] David A. Gudeman. Denotational semantics of a goal-directed language. *ACM Transactions on Programming Languages and Systems*, 14(1):107–125, January 1992.

[Han76] David R. Hanson. Variable associations in SNOBOL4. *Software Practice and Experience*, 6(2):245–254, April 1976.

[Ico] The Icon programming language. http://www.cs.arizona.edu/icon/.

[JVM] Languages for the Java VM. http://grunge.cs.tu-berlin.de/ tolk/vmlanguages.html.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997. ISBN 0-201-63452-X.

[MJ99] Gary Meehan and Mike Joy. Compiling lazy functional programs to Java bytecode. *Software Practice and Experience*, 29(7):617–645, July 1999.

[OG87] Janalee O'Bagy and Ralph E. Griswold. A recursive interpreter for the Icon programming language. In *Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, pages 138–149, St. Paul, Minnesota, June 1987.

[Pro97] Todd A. Proebsting. Simple translation of goal-directed evaluation. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 1–6, June 1997.

[Uni] The Unicode standard. http://www.unicode.org/unicode/standard/standard.html.

[Wal91] Kenneth Walker. *The Implementation of an Optimizing Compiler for Icon*. PhD thesis, University of Arizona, August 1991.