Linj

António Menezes Leitão

October 10, 2003

Contents

Chapter 1

The Linj Language

1.1 Introduction

We will now describe the Linj language. Linj tries to imitate Common Lisp as much as possible and this manual will follow the example by trying to imitate one of the best Common Lisp manuals. We will purposely follow the same presentation sequence (and style) of [1].

For each language feature, we will discuss what is the meaning in Common Lisp, what are the restrictions on its use in Linj and what does it look like when translated into Java.

1.2 Syntax

Linj programs use the same syntax as Common Lisp programs. In fact, Linj programs are read using the Common Lisp reader. This means that all the Common Lisp *special* characters such as parenthesis, quote, semicolon, double quote, backquote, comma, etc, are available and have the same meaning.

There's one character that deserves special attention: the colon character `:`. In Common Lisp, this character is used to indicate which package a symbol belongs to. In Linj, the concept of packages is completely different and this character is not needed and should be avoided.

The use of the Common Lisp reader to read Linj code has the advantage that all Common Lisp standard macro- and dispatch-macro-characters are available and it is possible to define new macro- or dispatch-macro-characters.

One of the good things about the Common Lisp syntax is that it is possible to write really weird names. For example, `*&!-?` is an acceptable Common Lisp name. In Linj. however, one must take into account that the names must ultimately be translated into Java names. As Java imposes much more restrictions upon the constituents of names, Linj must have a way to translate from Common Lisp names to Java names without violating the Java rules. Finally, and because a language is not only syntax and semantics but also *pragmatics*, Linj also tries to translate between Common Lisp conventions and Java conventions.

The common convention in Common Lisp is to separate multi-word names using the character `-`. Thus the `multi-word` multi-word name follows Common Lisp conventions. Java, on the other hand, prefers multi-word names written with capitalization on all words except the first (except on type definitions where all words are capitalized, and constants where everything is uppercase and words are separated by the underline character). Thus the `multiWord` multi-word name follows Java conventions. As expected, Linj implements the necessary translations between these two conventions.

In the very rare situation where the Linj name employs other characters that cannot be used in Java, Linj will translate those characters to a symbolic representation. For example, the symbol `*&!-?` is converted into Java's `starAndBangP`. Obviously, this translation scheme is not bullet-proof and sometimes you might get a *clash*: a symbol that is being translated into Java might coincide with another already existent symbol. In this case, Linj warns you about this fact and you should change one of them.

Another difference in the Common Lisp and Java code conventions occurs in the treatment of constants. In Common Lisp, it is suggested that they should be written surrounded by `+`. Thus, the Common Lisp programmer expects `+max-size+` to designate a constant value. In Java, the convention says that the constant name should be written in uppercase, possibly using the character `_` to separate words. Thus, the Java programmer expects to see the same constant as `MAX_SIZE`. Linj also implements the translation between both conventions.

These translations are not enough when the programmer needs to access Java libraries that do not follow the Java conventions or when it is cumbersome to use the Common Lisp conventions. For example, if we reverse the Linj translation schemes on the Java `EOFException` type, we find out the Linj name `e-o-f-exception` which is less clear and harder to write. Given the fact that Linj is intended to mix very well with all Java libraries, there's a final translation scheme that is simply to not operate any translation scheme whenever the name is written in mixed case.

Here is an example of Linj code that shows all conventions:

```
(defun %$#@^ (!! a-normal-name reallyFunny)
  (declare (type int !! a-normal-name reallyFunny))
  (let ((BIGNAME 1)
        (ok? t))
    (if (< !! +the-biggest+)
      BIGNAME
      (%$#@^ (1- !!) 0 (+ a-normal-name reallyFunny)))))
```

and here is the translation into Java:

```
public static int percent$SharpAtUp(int bangBang, int aNormalName, int reallyFunny) {
    int BIGNAME = 1;
    boolean okP = true;
    if (bangBang < THE_BIGGEST) {
        return BIGNAME;
    } else {
        return percent$SharpAtUp(bangBang - 1, 0, aNormalName + reallyFunny);
    }
}
```

1.3 Scope and Extent

One of the fundamental differences between Java and Common Lisp lies in the concepts of *scope* and *extent*.

In Common Lisp, these concepts range from *lexical* to *indefinite* for scope and from *dynamic* to *indefinite* for extent. Each Common Lisp constructs employ a combination of these. Linj has different forms of scope and extent but tries to follow the Common Lisp approach wherever it is feasible.

Just to give one example of these concepts we will discuss one of the most useful features of Common Lisp: the indefinite extent for the bindings of a function parameters. Linj also uses it, albeit in a restricted form. Here is one example adapted from [1]:

```
(defun compose (f/function g/function)
  #'(lambda (x)
      (funcall f (funcall g x))))

(defun main (d/double)
  (princ (funcall (compose #'(sqrt double) #'(abs double)) d)))
```

The small program above defines first the function combining function `compose` that, given two functions as argument, returns its composition. The second definition is just the `main` function of the program that accepts a number and prints the square root of its absolute value. The translation to Java follows:

```
import linj.Function;

public class Compose extends Object {

    public static Function compose(final Function f, final Function g) {
        return new Function() {
                   public Object funcall(Object x) {
                       return f.funcall(g.funcall(x));
                   }};
    }

    public static void main(String[] outsideArgs) {
        double d = Double.valueOf(outsideArgs[0]).doubleValue();
        System.out.
         print(compose(new Function() {
                           public Object funcall(Object genericArg) {
                               double arg = ((Number)genericArg).doubleValue();
                               return new Double(Math.sqrt(arg));
                           }},
                       new Function() {
                           public Object funcall(Object genericArg) {
                               double arg = ((Number)genericArg).doubleValue();
                               return new Double(Math.abs(arg));
                           }}).
                funcall(new Double(d)));
    }
}
```

In general, it is extremely difficult to preserve the Common Lisp semantics and, at the same time, generate nice-looking Java code. However, in a significant number of cases, Linj could manage to achieve this goal. We will now explain how this was achieved.

1.3.1 Shadowing

We will start by looking at the concept of *shadowing*. In Common Lisp, if two constructs that establish entities with the same name are textually nested, we say that the inner one *shadows* the outer one. The following example is extracted from [1]:

```
(defun test (x z)
  (let ((z (* x 2))) (print z))
  z)
```

In the previous example, the binding of the variable `z` by the `let` construct shadows the parameter binding for the function `test`. The reference to the variable `z` in the `print` form refers to the `let` binding. The reference to `z` at the end of the function refers to the parameter named `z`.

In what regards shadowing, Java is very different from Common Lisp. In Java, the name of a local variable may not be redeclared as a local variable of the directly enclosing method, constructor or initializer block within the scope of the first declaration. This rule also applies to redeclarations as exception parameters. The exception to the rule is field names: they can be shadowed by local variable declarations (including method parameters). The main justification for the exception is that it avoids forcing the programmer to invent new names when defining constructors.

In Linj, however, we want to be closer to Common Lisp than to Java and the irritating rules for shadowing in Java are a nuisance in Common Lisp. As a result, Linj implements the Common Lisp semantics but generates Java code according to the Java semantics. This forces the compiler to invent some name mangling schemes to replace true shadowing.

To understand the compiler operation, let's look at the translation of the previous example:

```
public static Object test(Bignum x, Object z) {
    {
        Bignum z0 = x.multiply(Bignum.valueOf(2));
        System.out.print("" + '\n' + z0 + " ");
    }
    return z;
}
```

Note that the compiler has renamed the `z` variable so that it can be used as if it shadowed the `z` parameter. All occurrences of `z` inside the scope of the declaration are changed so, in fact, it is impossible to access the parameter, as intended by the shadowing process. Note also that this new variable `z0` is defined and used in a new lexical context so it does not conflict with other potential occurrences of `z0`. The rename scheme is clever enough to avoid conflicts with field names (i.e., class slots). The net result is that the Linj programmer can use shadowing just like in Common Lisp and forget about the stricter rules for Java.

1.3.2 Restricted Lexical Scope

Linj tries to follow Common Lisp rules regarding scope and extent but, sometimes, this is just impossible and we have to change semantics a bit to allow for a better Java translation.

In general, Linj constructs have lexical scope. For example, variable bindings (function parameters, local variable declarations, etc.) have lexical scope.

There are constructs, however, which have a *restricted* lexical scope. The `block` construct is such an example. Consider the following example:

```
(defun wrong-block ()
  (block exit
    (funcall #'(lambda ()
                 (return-from exit 1)))))
```

This rather simple example doesn't work in Linj because the lexical scope of a `block` label ceases when we enter another function, as is the case with the above `lambda`. This is imposed by the rule that governs the scope and extent of a `block` construct: *An exit point established by a block construct has lexical scope and dynamic extent. However, the scope cannot cross function (or method) boundaries.*

In the rest of this text, we will warn the reader whenever the scope of a Linj construct differs from the scope of the corresponding construct in Common Lisp.

1.3.3 Restricted Indefinite Extent

One of the best features of Common Lisp is the indefinite extent of variable bindings. Linj also tries to follow this excellent idea but the rule is a bit different: *Variable bindings have lexical scope. Its extent is indefinite if there are no assignments and dynamic otherwise.*

As is possible to see, it's not the full Common Lisp-like indefinite extent but rather a *restricted* form of indefinite extent. As will be explained later, different Linj constructs have different extents.

1.3.4 Modules

In Common Lisp, the main tool for modularization is the *package*. The package isolate *names* used in one module from names used in other modules, thus preventing name conflicts. Name conflicts arise, mainly, because one function defined in one module might have the same name as a different function defined on another module.

Linj also contain packages but not in the Common Lisp sense. In Linj, the packages isolate *types* used in one module from types used in other modules. The question is: how can it prevent name conflicts? The answer lies in the fact that all names are defined relative to a type. Even names for functions (defined using `defun`) belong to the type definition above them (or to an automatically generated one when there's none). Usually, functions defined on the same file without intervening type definitions can call each other without any problems but functions defined on different files or with intervening type definitions will need some extra code. Let's see an example.

Suppose in file `bar.linj` we include the following definition:

```
;;File bar.linj

(defun one-down (bottles/int)
  (format t
          "~[No more bottles of beer on the wall!~%~:;~:*~
             ~D bottle~:P of beer on the wall,~%~
```

```
             ~:*~D bottle~:P of beer,~%~
             Take one down, pass it around,~%~
             ~D bottle~:*~P of beer on the wall,~%~]"
          bottles (1- bottles)))
```

Now, in file `thirst.linj` we write:

```
;;File thirst.linj

(defun beer-please (n/int)
  (unless (< n 0)
    (in (the bar)
        (one-down n))
    (beer-please (1- n))))

(defun main ()
  (beer-please 99))
```

The file `thirst.linj` contains a complete program. The `main` function calls `beer-please` and this function calls `one-down`. However, before calling `one-down`, it *augments* the lexical context using the `in` Linj form. This form accepts a description of a module (in this case, the `(the bar)`) form) and provides a context to resolve references in its body. In this case, the body contains the function call `one-down` that will be resolved to the function definition `one-down` in module `bar`.

Before we show the Java code that is produced after translation, we should point that `in` forms can be nested, thus providing a hierarchy of contexts to resolve references. The resolution process starts at the innermost context and searches then upwards. Note that the `in` form is also a *shadowing* form. If an `in`-context form uses a context that defines a name that is also defined outside the `in`-context form, that name shadows outer definitions. Definitions that are not shadowed are still visible, obviously. Thus, in the previous case, we could also rewrite the `beer-please` function as follows:

```
(defun beer-please (n/int)
  (unless (< n 0)
    (in (the bar)
        (one-down n)
        (beer-please (1- n)))))
```

Now, to understand what the translation looks like, here is the code for the Java file `Bar.java`:

```
public class Bar extends Object {

    public static void oneDown(int bottles) {
        switch (bottles) {
        case 0:
            System.out.println("No more bottles of beer on the wall!");
            break;
        default:
            System.out.print("");
            System.out.print(bottles);
            System.out.print(" bottle");
            if (! (bottles == 1)) {
                System.out.print('s');
            }
```

```
            System.out.println(" of beer on the wall,");
            System.out.print("");
            System.out.print(bottles);
            System.out.print(" bottle");
            if (! (bottles == 1)) {
                System.out.print('s');
            }
            System.out.println(" of beer,");
            System.out.println("Take one down, pass it around,");
            System.out.print("");
            System.out.print(bottles - 1);
            System.out.print(" bottle");
            if (! ((bottles - 1) == 1)) {
                System.out.print('s');
            }
            System.out.println(" of beer on the wall,");
            break;
        }
    }
}
```

and here is the Java file `Thirst.java`:

```
public class Thirst extends Object {

    public static void beerPlease(int n) {
        if (n >= 0) {
            Bar.oneDown(n);
            beerPlease(n - 1);
        }
    }

    public static void main(String[] outsideArgs) {
        beerPlease(99);
    }
}
```

1.3.5 Types are Modules

In the previous example there were no type definitions, which is a rare situation. We will discuss type definitions later on but, for now, the important concept to remember is that a type definition *initiates* a new module that is only finished at the next type definition or at the end of the file.

Consider the following example where everything works as expected:

```
(defun f1 ()
  (f2))

(defun f2 ()
  1)
```

and imagine that a type definition is introduced between the two functions:

```
(defun f1 ()
  (f2))

(defclass foo ()
  ())
```

```
(defun f2 ()
  1)
```

With this new arrangement, `f1` cannot see the `f2` because they live in two different modules, thus resulting in a compilation error. The be able to access `f2`, `f1` will now have to use a `(in (the foo) (f2))` form.

This strategy might seem inappropriate for die-hard Common Lisp programmers but it will make much more sense when we discuss Linj object-oriented capabilities. Then, we will see that a `defun` form is nothing more than a method with some special characteristics.

1.3.6 Accessing Java

Another capability of the `in` form is that it allows access to the (accessible) static slots and methods of Java classes. For example, here is a function that accesses the Java constant `PI` (that belongs to the class `Math` in package `java.lang`):

```
(defun half-pi ()
  (in (the math)
      (/ +pi+ 2)))
```

The translation is:

```
public static double halfPi() {
    return Math.PI / 2;
}
```

1.3.7 Importing Modules

We said that Linj separates types into packages. Each type corresponds to a module. Packages are themselves organized into an hierarchy where one package can contain modules and/or other packages.

When there's the need to use a type that belongs to some package, Linj provides a form that is semantically identical to Java `import` declarations. For example, to use the Java type `TextField` in Linj programs we can write:

```
(import java.awt.text-field)

(defun test ()
  (new 'text-field))
```

This will be translated into:

```
import java.awt.TextField;

public class Imports extends Object {

    public static TextField test() {
        return new TextField();
    }
}
```

Just like in Java, it is possible to import all types/modules from a package by using a `*`. Thus, the example above could also have been written as `(import java.awt.*)`.

1.3.8 Automatic Imports

One of the nice features of Linj is its integration with the Java libraries. Linj has extensive knowledge of the most used Java packages (including `java.util`, `java.math`, `java.io`, `java.awt`, `java.awt.event`, `java.applet`, and `java.net`) and can automatically infer the appropriate `import` declarations. In fact, in the previous example no import declaration was necessary. A more elaborate example is:

```
(defun use-them-all (cal/calendar in/buffered-reader out/file-writer)
  (let ((count 1Bigdec)
        (choice (new 'choice))
        (url (new 'URL "http://www.alu.org")))
    ...))
```

The translation to Java automatically includes several import declarations:

```
import java.net.URL;
import java.awt.Choice;
import java.math.BigDecimal;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.util.Calendar;

    ...

    public static void useThemAll(Calendar cal, BufferedReader in, FileWriter out) {
        BigDecimal count = BigDecimal.valueOf(1);
        Choice choice = new Choice();
        URL url = new URL("http://www.alu.org");
        ...
    }
```

1.3.9 Creating Packages

Just like in Java, Linj programs are organized as a set of *packages*. Each package has its own set of names for types. These names do not conflict with other identical names belonging to different packages. Linj packages are hierarchical, meaning that there can be packages inside other packages.

The `package` declaration allows us to declare new packages. Again, the semantics is that of Java. For example, Linj provides a set of modules such as `cons`, `function`, etc, that are necessary to implement the Linj run-time. To avoid conflicting with other modules, they all belong to a package called `linj`. Just to illustrate, we know show the definition of the `function` module:

```
(package linj)

(defclass function ()
  ())

(defmethod funcall ((f function) arg)
  arg)
```

The translation is simply:

```
package linj;

public class Function extends Object {

    public Object funcall(Object arg) {
        return arg;
    }
}
```

1.4 Types and Classes

Due to the static typing discipline adopted in Linj the programmer must be more concerned with types than it is usual in Common Lisp. The Linj type inferencer simplifies much of the work needed to correctly declare the types used in the program but the programmer will always have to provide some typing information.

Linj has a set of types primitively defined and has access to the types defined in Java libraries. However, in most cases, this will not be enough and the Linj programmer will need to define new types. This can be achieved with the `defstruct` and `defclass` forms.

1.4.1 Type Specifiers

In Linj, type specifiers are much simpler than in Common Lisp. Linj supports type specifiers that are symbols and lists. The list case is used exclusively to indicate array (or vector) type specifiers. In this last case, Linj also provides a simplified symbol-based form that is justified by the fact that type specifiers are much more needed in Linj than in Common Lisp. There are no provision for any other type of type specifier, including predicating type specifiers, type specifiers that combine, type specifiers that specialize, and type specifiers that abbreviate. Also, there is no provision for defining new type specifiers.

The type specifier symbols recognized include all user- or library-defined types and also the following: `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, and `double`.

To specify array types, you can use the same Common Lisp simplification rules for dropping unspecified items. Thus, just like in Common Lisp, `(vector double *)` may be abbreviated to `(vector double)`, and `(vector * *)` may be abbreviated to `(vector)` and then to simply `vector`. Notice that, in Linj, `vector` and `Vector` are two distinct types.

Type specifiers for multi-dimensional arrays also follow the Common Lisp syntax. Here is one definition that exemplifies the several forms of type specifiers for declaring arrays.

```
(defun xpto (a b c d e f g h i)
  (declare (type vector a)
           (type (vector) b)
           (type (vector int) c)
           (type (vector double *) d)
           (type Vector e)
           (type (vector Vector *) f)
           (type (array * 9) g)
```

```
          (type (array int (10 20)) h)
          (type (array long (* * * *)) i))
  ...)
```

The translation is the following:

```
public static int xpto(Object[] a, Object[] b, int[] c, double[] d, Vector e, Vector[] f,
                       Object[][][][][][][][][] g, int[][] h, long[][][][] i) {
    ...
}
```

Notice that we employed the `declare` form to indicate the types of the parameters. In section 1.8 we will explain a very compact form for type declarations where the type specifier is directly associated with the declared parameter or variable. This form, however, only accepts type specifier *symbols*, which limits its usefulness when in presence of array type specifiers.

To solve these cases, Linj also allows a very compact type specifier description for arrays that is very similar to a Java type declaration. The idea is that a form `(array` ***`type`*** `(`$\mathtt{\textit{dim}}_1$ $\mathtt{\textit{dim}}_2$ `...))` can be rewritten as ***`type`***`[][]...`.

This means that the previous example could also have been written as follows:

```
(defun xpto (a/object[] b/object[] c/int[] d/double[] e/Vector f/Vector[]
             g/object[][][][][][][][][] h/int[][] i/long[][][][])
  ...)
```

1.4.2 Classes

Classes, in Linj, follow the CLOS model but with some subtle differences:

- Instead of the CLOS multiple inheritance, Linj uses single class inheritance and multiple *mixin* inheritance.
- Classes aren't (yet) first class objects.[1]
- There's no concept of *metaclass*.
- The root of the class hierarchy is `object` and not `standard-object` as in CLOS.
- The protocol for instance initialization is much simpler.
- Linj include the accessibility model of Java, thus allowing control over the access to the slots of a class.
- Being a batch-compiled language without interactive `read-eval-print-loop`, the concept of class redefinition doesn't make sense and isn't allowed.

We will explain all Linj capabilities for object-oriented programming in Section 1.11.

[1] This isn't strictly true as it is possible to use Java reflection to treat classes (and other entities) as first class objects but this isn't smoothly integrated in Linj yet. But nothing prevents its use according to the Java rules.

1.5 Program Structure

Just like in Common Lisp, Linj programs are composed by forms. These forms can be either atoms or lists. In the first case, they correspond to self-evaluating forms or variables. In the second case, they correspond to special forms, macros and function (and method) calls.

1.5.1 Self-Evaluating Forms

Self evaluating forms include characters, strings, numbers and booleans. Note that, in Linj, the boolean `nil` is not the same object as the empty list neither of the symbol whose print name is `"NIL"`.

In Common Lisp, keywords (symbols belonging to the keyword package that are written with a leading colon) are also self-evaluating. In Linj, there are no packages in the Common Lisp sense and, therefore, the concept of a keyword doesn't exist. To create a symbol that resembles a Common Lisp keyword you just have to quote a name with a leading colon, such as `':start`. Being just like any other symbol, Linj keywords are not self-evaluating and require quoting.

In Linj programs, however, there's one situation where real Common Lisp keywords are used: function calls for functions that accept keyword parameters. This will be explained later but, for now, it is only important to remember that `':start` and `:start` are two completely different entities. The former is a symbol, i.e., a first-order object that can be passed as argument, returned as value or saved in data structures. The latter is a language entity that indicates what is the intended parameter for the subsequent argument and is used only during the compilation process: no references to it remain after the translation of the program.[2]

Just to demonstrate the different roles that "keyword" symbols can take, here is a fragment of Linj code where the keyword parameter `eof` is receiving the symbol `:eof` as argument.

```
(read-till-eof :eof ':eof)
```

After compilation, the code becomes:

```
readTillEof(Symbol.intern(":eof"));
```

Notice that the keyword that specifies the parameter was *resolved* and removed.

1.5.2 Variables

Linj use symbols to denote variables. Although Common Lisp has *lexical* and *dynamical* variables, Linj has mainly lexical variables. The exception is a few cases of variables that have a kind of indefinite scope. The Linj's variable scope and extent is explained in section 1.3.

Linj doesn't have the Common Lisp concept of *unbound variable*. In Linj, all variables are bound, either with an explicitly assigned value or with a system-provided default value.

[2] This is similar to the difference between a Common Lisp symbol and a Common Lisp lexical variable that is denoted by a symbol. You can manipulate the former but not the latter.

All variables defined with `defconstant` become constants and cannot be assigned.

1.5.3 Special Forms

Linj implements only a fraction of the Common Lisp special forms. More forms might be implemented in the future but, for now, Linj only provides the following special operators: `block`, `return-from`, `let`, `let*`, `if`, `progn`, `setq`, `function`, `quote`, `the`, `unwind-protect`.

Note that the semantics of some of the Linj special operators might be slightly different from the corresponding Common Lisp special operators.

The following Common Lisp special operators are *not* implemented in Linj: `flet`, `labels`, `catch`, `tagbody`, `go`, `macrolet`, `symbol-macrolet`, `load-time-value`, `eval-when`, `locally`, `multiple-value-call`, `multiple-value-prog1`, `progv`.

The following Common Lisp special operators are implemented in Linj but have definitely different semantics: `throw`.

1.5.4 Macros

Linj implements a substantial part of the most used Common Lisp macros and the user can define new macros using the `defmacro` form. Macros are defined in Common Lisp and not in Linj and they have access to the Common Lisp environment.[3]

It is only possible to macroexpand a macro call in the Common Lisp environment, which requires that the macro be defined there. As a result, the macro development and testing should be done in the Common Lisp environment and only then moved to the Linj program.

Note that Linj doesn't provide `*macroexpand-hook*`, `macro-function`, `macroexpand-1`, `macroexpand` and `macrolet`.

1.5.5 Function Forms

Function forms are compound forms where the first element is a symbol naming a function. In Linj, function forms are very similar to Common Lisp's. However, there's one fundamental difference. Whenever the form has keyword arguments, the evaluation order of those arguments is unspecified and might not be strictly left-to-right as is in Common Lisp.

Functions, in Linj, are defined exclusively by `defun` and `defmethod`. Linj doesn't implement `fdefinition`, `symbol-function`, or `defgeneric`, neither does it implement `flet` or `labels`. It is not possible to test function bindings (`fboundp`) or to remove them (`fmakunbound`).

It is not possible, in Linj, to use `functionp`, `apply` or `multiple-value-call`.

1.5.6 Lambda Forms

Lambda forms are forms similar to function forms but where the function name is replaced by a *lambda expression*. Linj doesn't support lambda forms in the

[3]This might be of no particular use as the Linj environment is completely separated from the Common Lisp environment.

Common Lisp sense. Linj supports, however, a restricted form of lambda expression that can be used as argument to the `funcall` function. This is describe in section 1.7.1.

1.5.7 Lambda Lists

A *lambda list* is a list that specifies the parameters and the protocol for receiving values for those parameters. Lambda lists are used in function, method, and macro definitions.

Linj provides a restricted form of *ordinary lambda list* in function definitions, another restricted form of *specialized lambda list* in method definition, and the full Common Lisp *extended lambda lists* in macro definitions.

Ordinary Lambda Lists

A Linj ordinary lambda list can contain the lambda list keywords `&optional`, `&key`, and `&rest`. It *cannot* contain the Common Lisp lambda list keywords `&aux` and `&allow-other-keys`.

The syntax for ordinary lambda lists is the following:

```
(var*
 [&optional var | (var [init-form [supplied-p-parameter]])*]
 [[&rest var] | [&key var | (var [init-form [supplied-p-parameter]])*]])
```

The comparison with the syntax for Common Lisp's ordinary lambda lists shows three fundamental differences:

- There is no provision for `&aux` or `&allow-other-keys`.
- Keyword parameters cannot use the notation `(keyword-name variable)`. This means that the keyword name used to match arguments to parameters is necessarily the name of the *`variable`* in the keyword package and nothing else.
- It is possible to have keyword parameters *or* rest parameters, but not both.

In all other aspects, Linj's lambda lists resemble Common Lisp's lambda lists. An *`init-form`* can be any form and it may refer to any parameter variable to its left, including any *`supplied-p-parameter`* variables. Note that due to the strong typing nature of Linj, *`supplied-p-parameters`* are `booleans` and non-supplied parameters are initialized with the system-supplied default value for the parameter's type.

In Table 1.1 we present a comparison between Linj and Common Lisp lambda lists. The examples were adapted from [1]. Note the different values for the *`supplied-p-parameters`* and missing parameters for the Linj and Common Lisp cases.

One extremely important difference between Linj and Common can be seen in the call `(foo-d :a 1 :d 8 :c 6)` for function `foo-d` with lambda list `(a b &key c d)`. In Common Lisp, the parameter `a` will simply receive the symbol `:a` as argument. In Linj, keyword *arguments* are used strictly to match with the corresponding keyword *parameters* and are not arguments on itself. This means that, in Linj, the keyword `:a` in the call `(foo-d :a 1 :d 8 :c 6)` specifies that the `(defun foo-a (a b) (list a b))`

Arguments	Linj	Common Lisp
`(foo-a 4 5)`	`(4 5)`	`(4 5)`

`(defun foo-b (a &optional (b 2)) (list a b))`

Arguments	Linj	Common Lisp
`(foo-b 4 5)`	`(4 5)`	`(4 5)`
`(foo-b 4)`	`(4 2)`	`(4 2)`

`(defun foo-c (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x))`

Arguments	Linj	Common Lisp
`(foo-c)`	`(2 nil 3 nil ())`	`(2 () 3 () ())`
`(foo-c 6)`	`(6 t 3 nil ())`	`(6 t 3 () ())`
`(foo-c 6 3)`	`(6 t 3 t ())`	`(6 t 3 t ())`
`(foo-c 6 3 8)`	`(6 t 3 t (8))`	`(6 t 3 t (8))`
`(foo-c 6 3 8 9 10)`	`(6 t 3 t (8 9 10))`	`(6 t 3 t (8 9 10))`

`(defun foo-d (a b &key c d) (list a b c d))`

Arguments	Linj	Common Lisp
`(foo-d 1 2)`	`(1 2 null null)`	`(1 2 () ())`
`(foo-d 1 2 :c 6)`	`(1 2 6 null)`	`(1 2 6 ())`
`(foo-d 1 2 :d 8)`	`(1 2 null 8)`	`(1 2 () 8)`
`(foo-d 1 2 :c 6 :d 8)`	`(1 2 6 8)`	`(1 2 6 8)`
`(foo-d 1 2 :d 8 :c 6)`	`(1 2 6 8)`	`(1 2 6 8)`
`(foo-d :a 1 :d 8 :c 6)`	Doesn't compile	`(:a 1 6 8)`
`(foo-d ':a 1 :d 8 :c 6)`	`(:a 1 6 8)`	`(:a 1 6 8)`
`(foo-d :a :b :c :d)`	Doesn't compile	`(:a :b :d ())`
`(foo-d ':a ':b :c ':d)`	`(:a :b :d null)`	`(:a :b :d ())`

Table 1.1: Comparing Linj and Common Lisp lambda lists

keyword parameter `a` will receive the argument `1`. In this case, the function `foo-d` doesn't contain any such parameter and we get a compilation error. The general rule is that any keyword that occurs in an argument list is never treated as an argument. If the intent is, in fact, to use a symbol (possibly from package keyword) as an argument, then it is only necessary to quote it, like in the call `(foo-d ':a 1 :d 8 :c 6)`. The reader should compare each of the calls in Table 1.1 that generate compiler errors with the subsequent call that does not.

One big advantage of this scheme is that it removes one annoying source of bugs in Common Lisp that occurs when we mix optional and keyword parameters. To understand this problem, let's look at the Common Lisp function `read-from-string`. Its lambda list is `(string &optional eof-error-p eof-value &key start end` When we want to read from some starting position on a string without concern to the `eof` situation we tend to forget that we cannot provide arguments to the keyword parameter without first providing arguments for the optional parameters, for example, writing `(read-from-string str :start 20)`. In this case, what will happen, in fact, is that the parameter `eof-error-p` will receive the argument `:start` and the parameter `eof-value` will receive the argument `20`, definitely not what we intended.

In the Linj implementation of this function, the call `(read-from-string str :start 20)` will pass the argument `20` to the parameter `start`, leaving all the other parameters with their default values. To obtain the same behavior as in Common Lisp, one would have to write `(read-from-string str ':start 20)`. The quote operator, here, acts as an *intent* operator.

In what regards the translation from Linj to Java, each Linj method with optional or keyword arguments is translated into two Java methods: one containing the body of the function or method and with all necessary parameters, and another that calls the first defaulting all the parameters that did not receive a corresponding argument. To visualize the process we will exemplify with one hypothetical Linj implementation of the Common Lisp function `string-upcase`. The function definition is something of the form:

```
(defun string-upcase (string/string &key (start 0) (end (length string)))
  ...)
```

Notice that the function has one required parameter and two keyword parameters.[4] The translation of the function produces the following Java methods:

```
public static String stringUpcase(String string, int start, int end) {
    ...
}

public static String stringUpcaseKey(String string, int start, int end, int argsPassed) {
    return stringUpcase(((argsPassed & 1) == 0) ? null : string,
                        ((argsPassed & 2) == 0) ? 0 : start,
                        ((argsPassed & 4) == 0) ? string.length() : end);
}
```

The interesting part is the `stringUpcaseKey` Java method that is automatically generated. This method has an extra parameter named `argsPassed` that will receive an `int` value containing a bit mask, with a bit on for every parameter that has a corresponding argument in the function call. All the other

[4] Contrary to Common Lisp where the function defaults the `end` parameter to `nil`, we have to default it to a number, in this case, the length of the argument string.

parameters will receive either a passed argument or a default (and irrelevant) value. For each parameter, the method checks the bit mask to see if it had a corresponding argument and, if it did, use it; if it didn't, use the initialization expression that was present on the original function definition.[5]

Regarding the translation of method calls, it depends on the actual argument passed. Here are some examples:

```
(string-upcase "foobar" :start 1 :end 2)
(string-upcase "foobar" :end 2 :start 1)
(string-upcase "foobar" :start 1)
(string-upcase "foobar" :end 2)
```

and the translation to Java:

```
stringUpcase("foobar", 1, 2);
stringUpcase("foobar", 1, 2);
stringUpcaseKey("foobar", 1, 0, 3);
stringUpcaseKey("foobar", 0, 2, 5);
```

Notice that when all the parameters have a corresponding argument, there's no need to go through the auxiliary method `stringUpcaseKey`, thus improving code clarity.

Finally, just to show what the translation of a `&rest` parameter looks like, here is an hypothetical definition of the Scheme function `string-append` in Linj:

```
(defun append-strings (&rest strings)
  (let ((buff (new 'string-buffer)))
    (dolist (str/string strings)
      (append buff str))
    (to-string buff)))
```

The translation into Java shows that the `&rest` parameter is typed as a |cons—, a Linj run-time class that implements pair and list operations.

```
public static String appendStrings(Cons strings) {
    StringBuffer buff = new StringBuffer();
    for (Cons list = strings; ! list.endp(); list = list.rest()) {
        String str = (String)list.first();
        buff.append(str);
    }
    return buff.toString();
}
```

Now, consider the following calls:

```
(append-strings)
(append-strings "Append this!")
(append-strings x "+" y "=" z)
```

[5]This scheme has the obvious limitation that methods with keyword or optional parameters can only accept 32 parameters. Although it might be a very small value for the `call-argument-limit` but, in practice, this has not been a limitation and, in fact, 32 parameters already stretches the requirement for human-readable Java code generation. However, if the need arises, we can easily increase the limit to 64 simply using a `long` to keep the bit mask.

and the corresponding translation to Java:

```
appendStrings(Cons.EMPTY_LIST);
appendStrings(new Cons("Append this!", Cons.EMPTY_LIST));
appendStrings(new Cons(x, new Cons("+", new Cons(y, new Cons("=", new Cons(z, Cons.EMPTY_LIST))
```

Note the lists that are automatically constructed on the caller side and passed to the callee.

Specialized Lambda Lists

A specialized lambda list is syntactically identical to an ordinary lambda list except that each required parameter may optionally be associated with a class or object for which that parameter is specialized. In this case, the parameter becomes a list of its name and its specializer type.

```
(var | (var [specializer])*
 [&optional var | (var [init-form [supplied-p-parameter]])*]
 [[&rest var] | [&key var | (var [init-form [supplied-p-parameter]])*]])
```

Note that, in Linj, you can only specialize *one* of the required parameters (although it doesn't matter which).[6] Moreover, the method must be defined in the same compilation unit as the specializer type.

In what regards the translation of the specialized lambda list into Java, it is important to know that the specialized parameter is simply removed and all references to it are converted to the `this` reference in Java.

Here is one example of a method definition that specializes its second required parameter `list` for class `cons`.

```
(defmethod adjoin (elem (list cons) &key (test #'eq))
  (if (member elem list :test test)
    list
    (cons elem list)))
```

And here is its translation into Java:

```
public Cons adjoin(Object elem, Predicate2 test) {
    if (! memberKey(elem, test, null, 3).endp()) {
        return this;
    } else {
        return new Cons(elem, this);
    }
}
```

Extended Lambda Lists

Besides ordinary lambda lists, Linj uses extended lambda lists on macro definitions. These are absolutely identical to Common Lisp extended lambda lists because, in fact, Linj macros are defined using the underlying Common Lisp environment and, as a result, have access to the full power of Common Lisp.

[6] This might change in the future.

1.5.8 Top-Level Forms

Linj is drastically different from Common Lisp in what regards interaction with the user. The standard way for the user to interact with a Common Lisp implementation is via a `read-eval-print` loop. Linj programs, on the other hand, are batch-compiled to generate Java programs.[7]

However, when we are just "loading" code from a compiled file, Linj is very similar to Common Lisp. Linj files, like Common Lisp files, are composed mainly by class, function, method and "global" variable definitions, with some occasional expressions that the programmer wants to see evaluated (e.g., to initialize a given data-structure) during the load of the file.

One important difference between Common Lisp and Linj in this regard is that, in Common Lisp, it is possible (although not always appropriate) to have a form that is usually top-level (such as a `defun` or `defconstant`) being evaluated in a non top-level context. One common cause is the necessity of establishing a lexical context for the defined form. In Linj, this is strictly forbidden. All Linj top-level forms such as `defun`, `defclass`, `defmethod`, `defconstant`, etc, *must* be defined at top-level, with one exception: they can be surrounded by one or more `progn` forms. This exception is motivated by the occasional need for a macro that needs to expands into more than one top-level form, thus requiring a `progn`.

We will now discuss the different Linj top-level forms.

The `defclass` Form

The fundamental top-level form in Linj is the *type definition*. Types are defined using either the |defstruct— form or the |defclass— form or the |defmixin— form.

The `defclass` form is the fundamental type-defining form. Its syntax mimics the CLOS `defclass` syntax.

We will now exemplify a `defclass` form:

```
(defclass cons ()
  ((car :accessor car :initarg :car)
   (cdr :accessor cdr :initarg :cdr)))
```

The previous class represents a `cons` cell and its definition is just like what we would be doing if we were writing it in CLOS.

Let's now look at the translation of the above class into Java:

```
public class Cons extends Object {

    // constructors

    public Cons(Object car, Object cdr) {
        this.car = car;
        this.cdr = cdr;
    }

    // accessors
```

[7]This situation might change in the near future, with the integration of a Java interpreter in the Linj system architecture that allows interactive "evaluation" of Linj forms.

```
    public Object car() {
        return car;
    }

    public void setfCar(Object car) {
        this.car = car;
    }

    public Object cdr() {
        return cdr;
    }

    public void setfCdr(Object cdr) {
        this.cdr = cdr;
    }

    // key methods

    public Cons(Object car, Object cdr, int argsPassed) {
        this(((argsPassed & 1) == 0) ? null : car, ((argsPassed & 2) == 0) ? null : cdr);
    }

    // slots

    protected Object car;

    protected Object cdr;
}
```

Just like in CLOS, the options `:reader`, `:writer`, or `:accessor` provided on each slot definition guide the creation of methods for getting and setting the slot. Note that, just like in CLOS, the option `:accessor` defines a setter that is accessed using using the `setf` form.

The `defmixin` Form

Mixins are syntactically identical to classes. The difference is semantic: a mixin can't be instantiated and it serves as a code repository that can affect the implementation of any class the inherits from the mixin. All the options allowed for classes are valid for mixins.

When a mixin is translated into Java the compiler generates an *interface* and all the mixin's methods become abstract and *bodyless* and all slots except class-allocated ones are eliminated.

Here is an example of a mixin:

```
(defmixin property-list-mixin ()
  ((properties :accessor properties :initform (list))))

(defconstant +secret-indicator+ (new 'object))

(defmethod get ((obj property-list-mixin) indicator &optional default)
  (getf (properties obj) indicator default))
```

The previous mixin provides very simple property list capabilities to any object. For example, to provide `persons` with property lists one just has to mix the `property-list-mixin`:

```
(defclass person (property-list-mixin object)
  ())
```

The translation of the `property-list-mixin` is:[8]

```
interface PropertyListMixin {

    public abstract Cons properties();

    public abstract void setfProperties(Cons properties);

    public abstract Object get(Object indicator, Object _default);

    public abstract Object getKey(Object indicator, Object _default, int argsPassed);

    public static final Object SECRET_INDICATOR = new Object();
}
```

and the translation of the `person` class is:

```
class Person extends Object implements PropertyListMixin {

    public void setfProperties(Cons properties) {
        this.properties = properties;
    }

    public Cons properties() {
        return properties;
    }

    public Object get(Object indicator, Object _default) {
        return properties().getf(indicator, _default);
    }

    public Object getKey(Object indicator, Object _default, int argsPassed) {
        return get(((argsPassed & 1) == 0) ? null : indicator, ((argsPassed & 2) == 0) ? null : _default)
    }

    protected Cons properties = Cons.EMPTY_LIST;
}
```

Note that the mixin code is, in effect, copied to the class.

The `defstruct` Form

The `defstruct` form is the simplest type-defining form. Its syntax is similar to the corresponding Common Lisp syntax but is much more limited because it doesn't allow any `defstruct` options (such as `:include`, `:conc-name`, `:offset`, `:constructor`, etc.). This limitation is intended as the `defstruct` form should only be used for record-like type definitions that do not require inheritance. More sophisticated types should be defined using `defclass`. However, Linj does implement all slot options, namely `:read-only` and `:type`.

Contrary to Common Lisp where `defstruct`-defined types might have some performance advantages relative to `defclass`-defined types, in Linj a `defstruct` is just syntactic sugar over a `defclass`.

Here is one example of the use of the `defstruct` form:

[8] Note that the name `default` was translated into `_default` because `default` is already a reserved word in Java. The Linj programmer never has to be concerned about the use of reserved words as the compiler takes care of it automatically.

```
(defstruct ship
  position
  velocity
  (mass 0.0 :type double :read-only t))
```

The previous form is exactly equivalent to the `defclass` definition:

```
(defclass ship ()
  ((position :type object :initarg :position :accessor ship-position)
   (velocity :type object :initarg :velocity :accessor ship-velocity)
   (mass :type double :initarg :mass :reader ship-mass :initform 0.0)))
```

and they both generate the following Java code:

```
public class Ship extends Object {

    // constructors

    public Ship(Object position, Object velocity, double mass) {
        this.position = position;
        this.velocity = velocity;
        this.mass = mass;
    }

    // accessors

    public Object shipPosition() {
        return position;
    }

    public void setfShipPosition(Object position) {
        this.position = position;
    }

    public Object shipVelocity() {
        return velocity;
    }

    public void setfShipVelocity(Object velocity) {
        this.velocity = velocity;
    }

    public double shipMass() {
        return mass;
    }

    // key methods

    public Ship(Object position, Object velocity, double mass, int argsPassed) {
        this(((argsPassed & 1) == 0) ? null : position,
             ((argsPassed & 2) == 0) ? null : velocity,
             ((argsPassed & 4) == 0) ? 0.0f : mass);
    }

    // slots

    protected Object position;

    protected Object velocity;

    protected double mass = 0.0f;
}
```

Note that structure instances are created using the same mechanism for classes, namely, with `make-instance`.

Method Definitions

Methods, in Linj, are defined just like in CLOS. However, Linj methods do not need to have lambda list *congruence* and they are not associated with any `defgeneric` forms as these don't even exist. In spite of the syntactic similarity between Linj and CLOS method definitions, the model is completely different. CLOS methods do not "belong" to classes but to generic functions. Linj methods do "belong" to a class and only to one class. This class is indicated by the specialized parameter. In this aspect, Linj object-oriented model is much more similar to Flavors than to CLOS. However, Linj method definitions allow specialization on any one of the required parameters and this makes it again similar to CLOS methods.

Here is an hypothetical method definition for class `cons`:

```
(defmethod nth (n/int (l cons))
  (if (= n 0)
    (first l)
    (nth (1- n) (rest l))))
```

Note that the method lambda list includes type information for the first parameter `n` and specialization information for the second parameter `l`.

The translation of the method into Java produces a Java method definition in the lexical scope of the Java class definition corresponding to the translation of the `cons` class:

```
public Object nth(int n) {
    if (n == 0) {
        return first();
    } else {
        return rest().nth(n - 1);
    }
}
```

Function Definitions

Functions, in Linj, are just methods that are class-allocated. However, contrary to methods, function can't have specialized parameters and, as a result, one must decide to which class they belong based on something else. Linj adopts the convention that functions "belong" to the nearest type definition that appears before the function definition in the compilation unit. If there is none, one is created automatically with the same name as the compilation unit itself.

Thus, imagine that in one compilation unit named `functions.linj` we find the following definitions:

```
(defun foo ()
  1)

(defclass bar ()
  ())
```

```
(defun baz ()
  2)
```

The Linj compiler will then generate the following Java code:

```
public class Functions extends Object {

    public static int foo() {
        return 1;
    }
}


class Bar extends Object {

    public static int baz() {
        return 2;
    }
}
```

Note the membership relation between methods and classes. Note also that the first class definition was automatically generated to include all function definitions that occur before the first `defclass`.

Function definitions are local to a module. This means that a function can be called freely from the same module but requires the use of the `in` form if from another module.

Named Constant Definitions

Named constants are defined using `defconstant`. These constants behave just like Common Lisp constants but, similarly to function definitions, they "belong" to the first type definition above them in their compilation unit.

Note that, according to Common Lisp conventions, constant names should be surrounded by "plus" signs. When this happens, Linj translates the constant name to Java using the Java conventions for constants which is to write them in uppercase.

Here are some examples of a named constant definition:

```
(defconstant +avogadro+ 6.02214199E-23)

(defconstant +newton+ 6.673E-11)

(defconstant +plank+ 6.62606876E-34)
```

and the corresponding translation into Java:

```
public static final float AVOGADRO = 6.022142e-23f;

public static final float NEWTON = 6.673e-11f;

public static final float PLANK = 6.626069e-34f;
```

In Linj, references to named constants must obey the rules that govern module access and, in most cases, require the use of the `in` form.

Variable Definitions

“Global” variables are introduced with the `defvar` and `defparameter` forms. Their semantics, in Linj, is absolutely identical and somewhat different from Common Lisp’s: these variables, in Linj, have nothing `special` about them.

Like named constants, these variable require the use of the `in` unless the references are made on the same module.

Here is one example of top-level variable definitions:

```
(defvar *print-length* 10)

(defparameter print-level 5)
```

and here is the translation:

```
public static int starPrintLengthStar = 10;

public static int printLevel = 5;
```

Note that there is no convention in Java for the naming of class-allocated slots. As a result, the Common Lisp convention of using `*` around `defvar` and `defparameter` forms should be avoided (unless the programmer isn’t concerned about the readability of the generated Java code).

Macro Definitions

Macros are defined using the `defmacro` form. This form is, if fact, the same that is used in the underlying Common Lisp where the Linj compiler runs and, therefore, has the exact same syntax and semantics. All macros defined in a compilation unit are “absorbed” by the Linj compiler and leave no trace on the generated Java code except for the expansion of the macro calls. Note also that the macros are truly global in the sense that after a compilation unit defines one it becomes active and is used in all subsequent compilation units.

Macros are further discussed in Section 1.9.

Other Top-Level Forms

Any code fragment that appears at top-level and that is not one of the previously described top-level forms is treated as an “initialization” that must be evaluated when the corresponding file is loaded. To achieve this effect, all such code fragments are translated into Java static blocks that will be automatically executed upon class loading.

For example, if we place the following statement on the top-level of a Linj file:

```
(format t "The file is being loaded at ~A~%" (new 'date))
```

it will be translated into the Java static block:

```
static {
    System.out.print("The file is being loaded at ");
    System.out.print(new Date());
    System.out.println();
}
```

that is automatically executed upon loading of the corresponding class file, writing something like:

```
The file is being loaded at Wed Oct 01 14:31:05 WEST 2003
```

Note that, for each top-level expression or statement, the compiler will generate one static block. If you prefer just one static block for all top-level expressions or statements, wrap them in a `let` form, as is shown below:

```
(let ()
  (format t "Loading file~%")
  (format t "Initializing...")
  (long-initialization)
  (format t "done~%"))
```

which is translated into the single block:

```
static {
    System.out.println("Loading file");
    System.out.print("Initializing...");
    longInitialization();
    System.out.println("done");
}
```

1.6 Predicates

In Common Lisp, a predicate is a function that tests for some condition involving its arguments and returns `nil` if the condition is false, or some non-`nil` value if the condition is true. One may *think* of a Common Lisp predicate as producing a boolean value, where `nil` stands for false and anything else stands for true.

In Linj, these terms need to be rephrased. In Linj, a predicate is a function that tests for some condition involving its arguments and returns the boolean `nil` if the condition is false or the boolean `t` if the condition is true. In Linj, predicate produce boolean values, where `nil` stands for false and `t` stands for true.

There is some provision, however, to treat other values as booleans. Values of the `cons` type (lists and pairs) can be assimilated to boolean values by treating the empty list `'()` as the boolean `nil` and any non-empty list as the boolean `t`. Values of other reference types can be assimilated to boolean values by treating the `null` value as the boolean `nil` and any other value as the boolean `t`. This treatment of other types as booleans is only done when in contexts where a boolean value is expected but a non-boolean value is found.

1.6.1 Logical Values

In Common Lisp, the boolean "false" value is exclusively represented by `nil` (or, equivalently, `()`) and any data object other than `nil` is treated as the boolean "true." The symbol `t` is used to mean "true" when no other value is more appropriate.

In Linj, truth and falsity are represented by `t` and `nil`, respectively and they are translated into Java's `true` and `false`. Note that, contrary to Common Lisp, in Linj `nil` does not represent the empty list neither does it represent the symbol `nil`. As will be explained later, the empty list is represented by `'()` (or `(list)`) and the symbol `nil` must be generated by other means.[9]

One big difference between Common Lisp and Linj is that in Linj, a function can only returns objects of a given type and, given the fact that primitive types are completely disjoint from reference types, it is impossible to build a function that returns, e.g., an `object` *or* a `boolean`, as it usually happens in Common Lisp with functions such as `find`, `find-if`, etc.

However, when dealing with reference types, there is a special value that is distinct from all others: the `null` value. This allows us to consider that values whose type is a reference type can be coerced to the boolean type simply by comparing the value with `null`.

Although very useful for most reference types, this scheme doesn't make much sense for the `cons` type. As will be explained later, pairs and lists are implemented in Linj using a reference type but where the empty list is not `null` but a special (and unique) value of the `cons` type. To allow for tests of the form `(if (member ...) ... ...)` the previous scheme is extended to also treat the empty list as a "false" value when in a context where a boolean is expected.

The following function exemplify all different uses:

```
(defun foo (a/boolean b/cons c/object)
  (not (and a b c)))
```

Note the types on the function parameters. Its translation into Java is:

```
public static boolean foo(boolean a, Cons b, Object c) {
    return ! (a && (! b.endp()) && (c != null));
}
```

1.6.2 Data Type Predicates

Given the static type discipline, Linj data type predicates are significantly simpler that in Common Lisp. For example, since all type information is known at compile-time, there's no need for a `subtypep` predicate. There are situations, however, when we need to test that a given object belongs to a given type (compatible with the real type of the object). This is accomplished by the `typep` predicate.

In Linj, the `typep` predicate can only be applied in situations where its first argument is a reference value and its second argument is a quoted symbol designating a non-primitive type definition. Here is an example:

```
(defun get-text-columns (text-obj/text-component)
  (cond ((typep text-obj 'text-area)
         (get-columns (the text-area text-obj)))
        ((typep text-obj 'text-field)
         (get-columns (the text-field text-obj)))
        (t
         (error "Unknown type of text-component ~A" text-obj))))
```

[9] In our opinion, in Common Lisp is is much less frequent to use `nil` as a symbol than it is as a boolean or as the empty list. This suggested us that we could reserve `nil` to represent a boolean and `'()` to represent the empty list.

The typep predicate is true when the first argument can be coerced to assume the type denoted in the second argument. The translation to Java is done in terms of the instanceof operator.

```
public static int getTextColumns(TextComponent textObj) {
    if (textObj instanceof TextArea) {
        return ((TextArea)textObj).getColumns();
    } else if (textObj instanceof TextField) {
        return ((TextField)textObj).getColumns();
    } else {
        throw new Error("Unknown type of text-component " + textObj);
    }
}
```

1.6.3 Specific Data Type Predicates

In Common Lisp, these predicates identify members of specific data types. In Linj, due to the static type discipline, they have much less use and they don't even exist for primitive types.

One predicate that needs discussion is null. In Linj every reference type has a null value that frequently needs to be recognized. A null predicate seems the appropriate thing to use. Unfortunately, null is a pre-defined Common Lisp predicate that is extremely used to test for an empty list and, sometimes, to invert a boolean value. In fact, in Common Lisp, the null predicate performs the same operation performed by the function not and it is explicitly recommended that the different uses should be signalled using the different functions.

To be more attractive to Common Lisp programmers, we decided to maintain the null behavior but we extended the tasks it performs so that it can also test for a null value.

Here is an example of all its uses:

```
(defun test (x/cons y/string z/boolean)
  (if (and (null x) (null y) (null z))
    (princ "Really absurd!")
    (princ "Ok!")))
```

Note that only the second null test checks whether the argument is null. However, such mixture of intents is not commendable and we recommend the programmer to be a bit more specific, e.g., writing instead:

```
(defun test (x/cons y/string z/boolean)
  (if (and (endp x) (null y) (not z))
    (princ "Really absurd!")
    (princ "Ok!")))
```

The reader will not be surprised to know that both forms are translated into the exact same Java fragment:

```
public static void test(Cons x, String y, boolean z) {
    if ((x.endp()) && (y == null) && (! z)) {
        System.out.print("Really absurd!");
    } else {
        System.out.print("Ok!");
    }
}
```

Other available type predicates are demonstrated on the following function:

```
(defun test (x/object)
  (cond ((symbolp x) (princ "It's a symbol"))
        ((atom x) (princ "It's an atom"))
        ((consp x) (princ "It's a cons"))
        ((listp x) (princ "It's a list"))
        ((numberp x) (princ "It's a number"))
        ((integerp x) (princ "It's an integer"))
        ((rationalp x) (print "It's a rational"))
        ((stringp x) (princ "It's a string"))
        ((hash-table-p x) (princ "It's an hashtable"))
        (t (princ "Uknown type"))))
```

which is translated into:

```
public static void test(Object x) {
    if (x instanceof Symbol) {
        System.out.print("It's a symbol");
    } else if ((x == Cons.EMPTY_LIST) || (! (x instanceof Consp))) {
        System.out.print("It's an atom");
    } else if ((x instanceof Cons) && (x != null)) {
        System.out.print("It's a cons");
    } else if (x instanceof Cons) {
        System.out.print("It's a list");
    } else if (x instanceof Number) {
        System.out.print("It's a number");
    } else if ((x instanceof Integer) ||
               (x instanceof Long) ||
               ((x instanceof Bignum) && ((Bignum)x).integerp()) ||
               (x instanceof BigInteger)) {
        System.out.print("It's an integer");
    } else if ((x instanceof Integer) ||
               (x instanceof Long) ||
               (x instanceof Bignum) ||
               (x instanceof BigInteger)) {
        System.out.print("" + '\n' + "It's a rational" + " ");
    } else if (x instanceof String) {
        System.out.print("It's a string");
    } else if (x instanceof Hashtable) {
        System.out.print("It's an hashtable");
    } else {
        System.out.print("Uknown type");
    }
}
```

1.6.4 Equality Predicates

Common Lisp programmers are used to the hierarchy of equality predicates `eq`, `eql`, `equal` and `equalp`. The last two predicates have been a source of much discussion regarding its usefulness and there is now general consensus that programmers should not use them and should instead define their own specialized equality predicates. Linj follows this consensus and do not provide neither `equal` nor `equalp`. However, to be more compatible with Java equality predicate for objects, Linj provides the `equals` predicate.

To understand the translation of Linj equality predicates into Java it is important to remember that, in Java, the equality operator `=` has two different semantics for primitive and reference types. This predicate means numeric equality when used with primitive values but means object identity when used with reference values.[10]

For a Lisp programmer, the difference between primitive (also called *immediate*) and reference values is meaningless and is seen just as an optimization explored in some implementations. What *is* meaningful for the Lisp programmer is that numeric equality is a well-defined operation over numbers (independently of its representation as primitive or reference values) and identity is another well-defined operation over objects. Every seasoned Lisp programmer knows about the difference between the corresponding Lisp functions `=` and `eq`, and the dangers of using `eq` on values of certain types that the implementation is free to optimize.

The good news for the Lisp programmer that wants to use Java is that Linj follows the Lisp tradition and provides both `=` and `eq` with the expected semantics. Table 1.2 presents the translation of an expression testing equality for different types of its arguments.

Type of `x` and `y`	Java translation for `(= x y)`
`long`	`x == y`
`long`	`x.longValue() == y.longValue()`
`big-integer`	`x.compareTo(y) == 0`

Table 1.2: Translations of the equality operator `(= x y)` for arguments of a given type.

The second quality predicate in the hierarchy is `eql`. According to the Common Lisp definition, the `eql` predicate is true when its arguments are `eq` or when they are both numbers of the same type with the same value or when they are characters objects that represent the same character.

Linj implements precisely the same semantics. Consider the following function that exemplifies equality tests over several types of values:

```
(defun equality (c1/char c2/char i1/int i2/int b1/bignum b2/bignum o1/object o2/objec
  (or (eq c1 c2)
      (eq i1 i2)
      (eq b1 b2)
      (eq o1 o2)
      (eql c1 c2)
      (eql i1 i2)
      (eql b1 b2)
      (eql o1 o2)
      (eql i1 b2)))
```

Its translation is as follows:

```
public static boolean equality(char c1, char c2, int i1, int i2,
                               Bignum b1, Bignum b2, Object o1, Object o2) {
```

[10]The C# language has made this issue even more annoying by providing automatic conversion between primitive types and the corresponding wrapper types without the introduction of different operations for numeric equality and object identity. It is highly probable that this "feature" will become a large source of bugs.

```
    if (c1 == c2) {
        return true;
    } else if (i1 == i2) {
        return true;
    } else if (b1 == b2) {
        return true;
    } else if (o1 == o2) {
        return true;
    } else if (c1 == c2) {
        return true;
    } else if (i1 == i2) {
        return true;
    } else if (b1.compareTo(b2) == 0) {
        return true;
    } else if ((o1 == o2) || ((o1 instanceof Number) && (o2 instanceof Number) && o1.equals(o2))) {
        return true;
    } else {
        return false;
    }
}
```

Note the different form of the tests employed in Java. The attentive reader will notice that the last test was removed because the compile could determine it could never be true.

1.6.5 Logical Operators

Linj provides the three logical operators **and**, **or**, and **not**. The **not** is translated into Java's **!**. The **and** and **or** deserve a bit more attention because, besides logical operators, they are also control structures.

In Common Lisp, **and** evaluates each argument form, one at a time, from left to right. If any form evaluates to **nil**, the value **nil** is immediately returned without evaluating the remaining forms. If every form but the last evaluates to a non-**nil** value, **and** returns whatever the last form returns.

If the Linj's **and** were implemented just like in Common Lisp, we would end up having expressions that either return **nil** (a boolean) or something else (probably not boolean). Unfortunately, this scheme can't be used because it is impossible to have an expression (or a function) that returns objects of incompatible types. This restriction reduces the usefulness of the **and** (or **or**) as a control structure in Linj. The alternative is to use other control structures such as **if**. Obviously, the **or** suffers from the exact same problems.

Another difference between Linj and Common Lisp implementations of the logical operators is in the syntactical categories. In Linj, the **and** and the **or** can be parsed either as statements or as expressions, producing different results in the Java translation. Here is one example:

```
(defun and-or-test (x/int y/int z/int)
  (and (< x y) (or (< y z) (< x z))))

(defun and-or-test (x/int y/int z/int)
  (return-from and-or-test
    (and (< x y) (or (< y z) (< x z)))))
```

In the first definition, the combination **and**/**or** is parsed as a statement. In the second definition, it will be be parsed as an expression. The compilation into Java produces:

```
public static boolean andOrTest(int x, int y, int z) {
    if (x >= y) {
        return false;
    } else if (y < z) {
        return true;
    } else {
        return x < z;
    }
}

public static boolean andOrTest(int x, int y, int z) {
    return (x < y) && ((y < z) || (x < z));
}
```

1.7 Control Structure

Linj contains most of the control structure available in Common Lisp but, in some cases, they have a slightly different flavor or they are somewhat restricted.

Just like in Common Lisp, the basic control structure is the *function application* but various other forms of control are available, such as *sequencing*, *repetition*, *selection*, *non-local exits*, *multiple values*, etc.

We will now discuss the Linj control structures.

1.7.1 Program and Data

Given the fact that Linj programs are made of Linj data objects (such as lists and symbols), Linj programs need to distinguish between an object being used as a term in the program structure or as a literal value to be used by the program. For example, in the expression `(setq a 'a)` the same symbol `a` is being used in two distinct roles: to designate a variable and to designate a symbol data structure. Another example is `(cons (list) '(list))` where the term `(list)` designates both a function call and a list structure with the symbol `'list` as its only element.

Quote

Just like in Common Lisp, the roles are disambiguated by the use of the *quote*. However, in Linj, the quote has some subtleties caused by the fact that Linj programs need to be translated into human-readable Java programs.

The first difference between Linj's `quote` and Common Lisp's `quote` is that, in the case of quoted list structures, the `quote` is "distributed" by the list elements, i.e., given a quoted dotted pair `'(`*car* `.` *cdr*`)` is translated into `(cons '`*car* `'`*cdr*`)`. Moreover, the form `'()` corresponds to the empty list (also written as `(list)`). Finally, quoting a symbol is translated into `(intern` *symbol-print-name*`)`. For example, the form `'(list)` is transformed into `(cons 'list '())` which is then transformed into `(cons (intern "list") (list))`. This form is now suitable for a translation into Java, producing something like

```
new Cons(Symbol.intern("list"), Cons.EMPTY_LIST);
```

Function

In Common Lisp, the `function` special form returns the *functional interpretation* of its argument. This argument can be either a symbol or a lambda expression and, in this last case, a lexical closure is returned. The `function` form can be abbreviated as `#'`.

Linj has some restrictions upon the implementation of the `function` form because there are no first class functions in Java that can serve as translation targets for Linj. However, Linj provides solutions for the most frequent situations of `function` use.

First, let's see some examples:

```
#'(lambda (x)
    (and (symbolp x)
         (eq x 'hello)))

#'(lambda (x)
    (format t "Nice:~A~%" x))

#'(lambda (n)
    (* n n))

#'(lambda (a b)
    (cons a (cons b a)))
```

Here is the translation:

```
new Predicate() {
    public boolean funcall(Object x) {
        if (! (x instanceof Symbol)) {
            return false;
        } else {
            return x == Symbol.intern("hello");
        }
    }};

new Procedure() {
    public void funcall(Object x) {
        System.out.print("Nice:");
        System.out.print(x);
        System.out.println();
    }};

new Function() {
    public Object funcall(Object genericN) {
        Bignum n = (Bignum)genericN;
        return n.multiply(n);
    }};

new Function2() {
    public Object funcall(Object a, Object b) {
        return new Cons(a, new Cons(b, a));
    }};
```

As is possible to see, `lambda`s are translated into anonymous inner classes that extend some already defined Linj classes such as `predicate`, `procedure`, etc. There are several methods that take advantage of lambda expressions.

For example, the `cons` class contains methods such as `position`, `position-if`, `mapcar`, `mapc`, etc., that accept one or more `functions` or `procedures`.

Here is another example:

```
(defun adder (x)
  #'(lambda (y) (+ x y)))

(defun main ()
  (let ((add3 (adder 3)))
    (print (funcall add3 5))))
```

Note that the result of `(adder 3)` is a function that will add `3` to its argument. The translation into Java is:

```
public static Function adder(final Bignum x) {
    return new Function() {
               public Object funcall(Object genericY) {
                   Bignum y = (Bignum)genericY;
                   return x.add(y);
               }};
}

public static void main(String[] outsideArgs) {
    Function add3 = adder(Bignum.valueOf(3));
    System.out.print("" + '\n' + add3.funcall(Bignum.valueOf(5)) + " ");
}
```

There are several points worth note:

- In Java only objects are first class. In order to pass a “function” as argument or return it as value, we need to create a class that implements a method that calls the “function” and we must perform the appropriate method call on a class instance.

- Anonymous inner classes have the power to capture the lexical environment. All methods of an anonymous inner classes have access to the surrounding lexical environment on the moment of creation.

- Java restricts anonymous inner classes with a rule that expresses the any local variable, formal method parameter or exception handler parameter used but not declared in an inner class must be declared `final`, and must be definitely assigned before the body of the inner class.

- Java does not provide a way to declare a type for an anonymous inner class, thus making it impossible to express a variable declaration or method returns for such type.[11]

The previous points have some implications for the Linj approach to translate `lambdas` in terms of anonymous inner classes. The first one is that when we want to use a “function” as argument of an higher-order function such as `some` or `mapcar`, the higher-order function needs to have the corresponding parameter correctly typed. To be generic, this type must be a super-type for all “functions.”

[11] Non-anonymous inner classes can be used for type declarations but they have other restrictions such as the legal places where they can occur and the understandability of the resulting code.

The second one is that the access to the surrounding lexical environment must be *read-only*.

The first problem can be solved by translating lambda expressions into instances of anonymous inner classes that inherit from some supporting functional classes containing a `funcall` method which is further specialized using the lambda body. There are a few of those supporting classes: `predicate` and `predicate-2` (where the method returns `boolean` and accepts one or two arguments, respectively); `procedure`, (where the method returns `void`); and `function` (where the method returns `object`), etc. Each `lambda` instance can then be `funcall`ed. This approach limits the types of `lambda`s that can be created but Linj type inference can insert the necessary type casts to allow the `lambda`s to be used in more contexts.

The second problem can be solved by identifying all variables and parameters accessed from the inner class body and automatically declare them as `final`. This, however, has one nasty consequence: it is impossible to assign such variables or parameters.

To understand this problem, consider the following fragment (adapted from [1]):

```
(defun two-funs (x)
  (values #'(lambda () x)
          #'(lambda (y) (setq x y))))

(defun main ()
  (multiple-value-bind (fun1 fun2)
      (two-funs 6)
    (print (funcall fun1))
    (funcall fun2 43)
    (print (funcall fun1))))
```

The function `two-funs` returns two functions, one “gets” the parameter `x` while the other “sets” the same parameter.

The translation into Java is the following:

```
public static Object[] twoFuns(final Object x) {
    return new Object[] { new Function0() {
                              public Object funcall() {
                                  return x;
                              }}, new Procedure() {
                                      public void funcall(Object y) {
                                          x = y;
                                      }} };
}

public static void main(String[] outsideArgs) {
    Object[] multipleResults = twoFuns(linj.Bignum.valueOf(6));
    Function0 fun1 = (Function0)multipleResults[0];
    Procedure fun2 = (Procedure)multipleResults[1];
    System.out.print("" + '\n' + fun1.funcall() + " ");
    fun2.funcall(linj.Bignum.valueOf(43));
    System.out.print("" + '\n' + fun1.funcall() + " ");
}
```

If you try to compile this code, you will get a compilation error because the second anonymous inner class (that extends `Procedure`) is making an assignment to the `final` parameter `x`.

What's the solution to this problem? Well, due to Java's object-orientation model, the "lexical" environment is more oriented towards surrounding "objects" than toward surrounding "bindings." Surrounding objects (called *enclosing instances*) allow reading and writing of the object slots without the restrictions imposed upon variables and parameters. Thus, we can rewrite the previous example into an object-oriented form:

```
(defclass box ()
  ((x :initarg :x)))

(defmethod two-funs ((b box))
  (with-slots (x) b
    (values #'(lambda () x))
            #'(lambda (y) (setq x y))))))

(defun main ()
  (multiple-value-bind (fun1 fun2)
      (two-funs (new 'box :x 6))
    (print (funcall fun1))
    (funcall fun2 43)
    (print (funcall fun1))))
```

Note that we don't provide an immediate value for `two-funs`. Instead, we provide an object that *boxes* the value so that we can read and write the "box." The function `two-funs` is now a method of the "box" class so that it has read and write access to the class slots and, particularly, to the slot `x`. The translation of the `box` code to Java produces:

```
class Box extends Object {

    public Box(Object x) {
        this.x = x;
    }

    public Object[] twoFuns() {
        return new Object[] { new Function0() {
                                  public Object funcall() {
                                      return x;
                                  }}, new Procedure() {
                                          public void funcall(Object y) {
                                              x = y;
                                          }} };
    }
```

The `main` method becomes now:

```
public static void main(String[] outsideArgs) {
    Object[] multipleResults = new Box(linj.Bignum.valueOf(6)).twoFuns();
    Function0 fun1 = (Function0)multipleResults[0];
    Procedure fun2 = (Procedure)multipleResults[1];
    System.out.print("" + '\n' + fun1.funcall() + " ");
    fun2.funcall(linj.Bignum.valueOf(43));
    System.out.print("" + '\n' + fun1.funcall() + " ");
}
```

This Java program is correctly compilable and runs with the expected results.

Function, Again

The `lambda`s presented above either didn't have parameters or they didn't declare the type of its parameters. However, it is frequently necessary to declare those types so that Linj type inference can work correctly. For example, let's suppose we need a first-order predicate that detects that an AWT window is on the screen. Given a window, the method `is-showing` answers our question but methods are not first-order. Fortunately, nothing prevents us from writing a lambda expression. Given the need for a type declaration so that the correct method `is-showing` can be determined, here are three possible versions:

```
#'(lambda (w)
    (is-showing (the window w)))

#'(lambda (w)
    (declare (window w))
    (is-showing w))

#'(lambda (w/window)
    (is-showing w))
```

On each case, the parameter `w` will have type `window`. However, if we allow for arbitrary types in the `lambda` parameters, Linj overloading capabilities will *define* new methods on the supporting class (be it `function`, `procedure`, etc) instead of *redefining* the template method `funcall` provided in those supporting classes. To prevent this situation, Linj automatically declares the types of the `lambda` parameters to be signature-compatible with what is required in the template method so that no overloading occurs. As a result, all three versions presented above produce the exact same Java code:

```
 Predicate() {
    public boolean funcall(Object genericW) {
        Window w = (Window)genericW;
        return w.isShowing();
    }};
```

Function, Again and Again

So far, we saw the `function` operator used with lambda expressions. We will now see it used with function names. In Common Lisp, if the `function` argument is a symbol, the functional definition associated with that symbol is returned. The problem in Java (and Linj) is, as we have said before, that Java methods are not first-class. But we can easily wrap a method call with a lambda expressions and we automatically get a first class object which can be `funcall`ed. This allows us to write, e.g., `#'clone` and get a Java translation of:

```
new Function() {
    public Object funcall(Object arg) {
        return arg.clone();
    }};
```

We can even cache some of those lambda expressions to avoid the repeatedly generation of instances of anonymous inner class (which can be computationally expensive). Several Linj support classes already provide some of those lambda expressions already cached, including `#'cons`, `#'car`, `#'eq`, `#'eql`, and `#'equals`.

Due to the fact that the parameters of the `lambdas` created are necessarily of type `object`, when the method that we want to make first-order isn't defined on the `object` class or isn't already pre-defined like `#'car`, we can't just use the method name as the argument for the `function` operator. To alleviate this problem, Linj provides a simple extension of the `function` operator that allows the specification of the types of the parameters, just like in the `lambda` case. The idea is that, instead of a symbol designating the method's name, we write a list with the complete signature of the method, that is, we include, besides the method's name, the types of its parameters.

Here are two examples:

```
#'(is-empty vector)
```

```
#'(append string-buffer object)
```

and the corresponding translation:

```
new Predicate() {
    public boolean funcall(Object genericArg) {
        Vector arg = (Vector)genericArg;
        return arg.isEmpty();
    }};

new Function2() {
    public Object funcall(Object genericArg0, Object arg1) {
        StringBuffer arg0 = (StringBuffer)genericArg0;
        return arg0.append(arg1);
    }};
```

1.7.2 Assignment

Variables, in Linj, have a different flavor from variables in Common Lisp. Scope, for variables, is only lexical and there is no concept of dynamic variables.

In Linj, variables include not only local variables, formal method parameter or exception handler parameter but also class slots. Whenever a class declares a set of slots, these can be directly accessed by the methods specialized on the class. This is similar to what happens in Flavors. Due to the multiple specialization capabilities of CLOS, this can't be done in CLOS but can be simulated using the `with-slots` form. Linj allows the programmer the freedom of choice regarding this last issue: you can use either the Flavors approach or the CLOS approach. This last one has the benefit that it might allow further developments of Linj to include multiple dispatch methods without breaking old code.

The assignment form, in Linj, is `setq`. The assignment is only possible when the assigned variable is type-compatible with the value assigned. If necessary, type casts and wrapping and unwrapping operations are inserted. Here are some examples:

```
(defun foo (x/int y/bignum z/float)
  (setq x 1
        y x
        z y))
```

The translation is:

```
public static void foo(int x, Bignum y, float z) {
    x = 1;
    y = Bignum.valueOf(x);
    z = y.floatValue();
}
```

In Linj, assignments do not return a value.[12] The `setf` form is also available but has extended capabilities that will be discussed later.

1.7.3 Parallel Assignment

Another assignment form available in Linj is the *parallel* assignment form `psetq` (and `psetf`). In this case, the assignments are done in parallel. In Common Lisp, this means that all forms are evaluated first and then the variables are set to the resulting values.

Linj implementation of parallel assignment has two different strategies that can be chosen by the programmer (by setting the parameter `*parallel-assignment-can-reorder-p*`). The first one strictly follows the Common Lisp rules for parallel assignment but can generate ugly code. Here is one example:

```
(psetq x 1
       y x
       z y)
```

And its translation:

```
int x0 = 1;
int y0 = x;
int z0 = y;
x = x0;
y = y0;
z = z0;
```

The second one relaxes the Common Lisp semantics in order to produce nicer Java code. The relaxation is related to the sequential order of form evaluation that is imposed in Common Lisp. With the second strategy, this order becomes unpredictable.[13]

Here is the same example translated to Java with the second strategy:

```
z = y;
y = x;
x = 1;
```

Note that the second strategy might also need to introduce extra bindings. Here is a puzzling function that operates a sequence of parallel assignments and that shows several different situations:

[12] This is different from both Common Lisp and Java but it makes sense to avoid incorrect type inferences in the very frequent case of assignments used as statements and not as expressions. However, we might change this in the near future.

[13] In fact, it's not completely unpredictable because Linj will just reorder the assignments to avoid dependency problems.

```
(defun xyz (x/int y/int z/int)
  (psetq y x
         x 1)
  (psetq x 2
         y x)
  (psetf x y
         y x
         z (+ x y z))
  (psetf x y
         y z
         z x)
  (values x y z))
```

The reader should pay attention to the translation to Java in order to understand the reordering of the assignments and the introduction of extra variables:

```
public static int[] xyz(int x, int y, int z) {
    y = x;
    x = 1;
    y = x;
    x = 2;
    {
        int x0 = x;
        z = x0 + y + z;
        x = y;
        y = x0;
    }
    {
        int x0 = x;
        int y0 = y;
        y = z;
        x = y0;
        z = x0;
    }
    return new int[] { x, y, z };
}
```

1.7.4 Generalized References

Linj supports the concept of generalized references (also called *places*) and of its assignment using the `setf` form. However, Linj does not support any of the `setf`-defining forms available in Common Lisp except the use of function or method definitions whose name is of the form (setf *name*).

The places recognized by the Linj implementation of `setf` are restricted to:

- variable names,
- function call forms, including several pre-defined cases such as `car`, `cdr`, `aref`, `slot-value`, selector functions constructed by `defstruct`, etc,
- `the` forms,
- other compound forms excluding `values` forms, `apply` forms, `setf`-expansion forms, macro forms, and symbol macro forms.

In the case of other compound forms, Linj attempts a simple translation scheme that is illustrated by the following example:

```
(defun (setf person-name) (name/string person)
  ...
  name)

(setf (person-name p1) "Paul")
```

```
public static String setfPersonName(String name, Object person) {
    ...
    return name;
}

setfPersonName("Paul", p1);
```

1.7.5 Function Invocation

Besides the normal function invocation, Linj also includes the `funcall` form that accepts lambda expressions (obtained via lambda forms or via the name of a function). The translation of a `funcall` form to Java corresponds to the invocation of the `funcall` method on the translation of the lambda expression. This last translation, as was explained in section 1.5.6 depends on the number of parameters and returned type of the lambda expression. Linj currently provides support for zero-, one-, and two-parameter lambdas, with return types of `void`, `boolean`, and `object`. The name of the support classes is, for the one-parameter case, `procedure`, `predicate`, and `function`, for a return type of `void`, `boolean`, and `object`, respectively. For the zero- and two-parameter cases, the names are the same as before but suffixed with a `-0` or `-2`, respectively. The programmer rarely has to use this names explicitly because the Linj type inferencer can deduce them automatically. However, there are cases where they must be used so it's good to learn them. There is one exception to this scheme that occurs when statements must be locally treated as expressions. In this case, a special funcall method without parameter will be generated and there's complete freedom in its return type. We will see an example in the next section.

The `funcall` form is specially useful to implement higher-order functions. For example, the `cons` type definition could include the following method:[14]

```
(defmethod member (elem (list cons) &key (test #'eq) (key #'identity))
  (if (or (endp list)
          (funcall test elem (funcall key (first list))))
    list
    (member elem (rest list) :test test :key key)))
```

Note the two `funcall` invocations. The translation to Java is:

```
public Cons member(Object elem, Predicate2 test, Function key) {
    if (endp() || test.funcall(elem, key.funcall(first()))) {
        return this;
    } else {
        return rest().member(elem, test, key);
    }
}
```

As is possible to see, no type declarations were needed. However, if the same type definition included also a `member-if` method, then a type declaration

[14]The real implementation is implemented with greater concerns towards efficiency.

would have been needed because the functional parameter is required and can't be defaulted. In this case, we could write:

```
(defmethod member-if (test/predicate (list cons) &key (key #'identity))
  (if (or (endp list)
          (funcall test (funcall key (first list))))
    list
    (member-if test (rest list) :key key)))
```

and we would get the following Java fragment:

```
public Cons memberIf(Predicate test, Function key) {
    if (endp() || test.funcall(key.funcall(first()))) {
        return this;
    } else {
        return rest().memberIf(test, key);
    }
}
```

1.7.6 Simple Sequencing

The Linj `progn` form has the same semantics as in Common Lisp but with one *nuance*. In Linj, `progn` is a statement and it's not recommended for code fragments such as `(+ 1 (progn (format t "Computing...") 2) 3)`. If the compiler parameter `*use-statement-as-expression-p*` is false, the code will not even compile. If the parameter is true, the code will be translated into something such as:

```
1 +
new Object() {
    public int funcall() {
        System.out.print("Computing...");
        return 2;
    }}.
 funcall() +
3
```

Given the reduced clarity of the result, when there's an explicit requirement to generate human-understandable code, it's preferable to stay away of those cases.

On the normal cases, however, the `progn` is being used as a statement and its translation into Java uses the Java's compound statement. However, instead of blindly generating compound statements, the Linj compiler chooses a minimal composition that avoids generation of unnecessary bracket delimiters. Here is one example:

```
(defun test-progn ()
  (progn
    (format t "Yes!")
    (progn
      (format t "Hummm...")
      (format t "Ok."))
    (format t "More?")
    (progn
```

```
    (format t "Last one...")
    (format t "I promise!"))
  1))
```

Translating to Java, produces a nice-looking method:

```
public static int testProgn() {
    System.out.print("Yes!");
    System.out.print("Hummm...");
    System.out.print("Ok.");
    System.out.print("More?");
    System.out.print("Last one...");
    System.out.print("I promise!");
    return 1;
}
```

The effort that was spend on the generation of nice-looking Java code is motivated by the fact that the `progn` form is quite used in macros and we didn't want to force the programmer to care about the effects of the macro-generated code in the clarity of the Java code.

The `prog1` form is similar to the `progn` but, instead of returning the value of the last form, it returns the value of the first. Note that, contrary to Common Lisp, Linj's `prog1` returns multiple values when the first form returns multiple values. This makes `multiple-value-prog1` unnecessary and, in fact, Linj only provided it as an *alias* to `prog1`. However, you should use the most adequate form, not only as a source of documentation but also because Linj might evolve in the future and implement `prog1` according to the Common Lisp semantics.

The `prog2` is also implemented for historical reasons.

Here is one example involving all forms:

```
(defun test-collatz (n/long)
  (if (= n 1)
    (progn
      (format t "1")
      t)
    (if (evenp n)
      (prog1
          (test-collatz (/ n 2))
        (format t "*2"))
      (prog2
          (format t "(")
          (test-collatz (1+ (* n 3)))
        (format t "-1)/3")))))
```

In this case, some readers might say that the translation looks nicer than the original:

```
public static boolean testCollatz(long n) {
    if (n == 1) {
        System.out.print("1");
        return true;
    } else if ((n % 2) == 0) {
        boolean results = testCollatz(n / 2);
        System.out.print("*2");
        return results;
```

```
    } else {
        System.out.print("(");
        boolean results = testCollatz((n * 3) + 1);
        System.out.print("-1)/3");
        return results;
    }
}
```

We leave as an exercise for reader to discover what's the meaning of the function's output.

1.7.7 Establishing New Variable Bindings

From the plethora of Common Lisp forms that establish new variable bindings (such as `let`, `flet`, `progv`, etc), Linj only implements `let` and `let*`. Note that `let` might reorder the bindings and/or add some extra ones to avoid dependencies between them, just like what was described in section 1.7.3.

Just like in the `progn` case, Linj takes advantage of the Java rules regarding the scope of variable declarations to avoid unnecessary brackets. Here is one example:

```
(defun baz ()
  (let ((x 1))
    (print x)
    (let ((y 2))
      (print y))
    (let ((y 3))
      (print y))))
```

and its translation:

```
public static void baz() {
    int x = 1;
    System.out.print("" + '\n' + x + " ");
    {
        int y = 2;
        System.out.print("" + '\n' + y + " ");
    }
    int y = 3;
    System.out.print("" + '\n' + y + " ");
}
```

Note that the last `y` variable "reuses" the lexical context of the function body instead of creating its own.

The forms `let` and `let*` also allow for mixin type declarations with the variable names. However, this rarely is needed as the value of the variable can be used to infer the correct type.

1.7.8 Conditionals

Linj implements the conditional constructs `if` and `cond` both as statements and as expressions but their translation to Java shows obvious differences. Compare the following function:

```
(defun two-ifs (n/int)
  (let ((abs-n (if (< n 0) (- n) n)))
    (if (> abs-n 10)
      (1+ abs-n)
      (1- abs-n))))
```

with its translation:

```
public static int twoIfs(int n) {
    int absN = (n < 0) ? n : n;
    if (absN > 10) {
        return absN + 1;
    } else {
        return absN - 1;
    }
}
```

and note how the two `if` forms are translated into either a Java conditional expression or a Java `if`-statement.

Linj's `if` has one fundamental difference from Common Lisp's `if`: in Linj, the third `if` argument is not optional, meaning that the *else* form cannot be omitted. As is suggested in Common Lisp, an `if` without *else* should be rewritten either using an `and` form or the `when` form.

Linj also implements the `when` and `unless` forms but exclusively as statements.

Finally, Linj implementation of the `cond` form also distinguishes between parsing as statement and as expression. Here are two examples:

```
(defun print-code-1 (x/int y/int z/int)
  (cond ((> x y) 1)
        ((< y z) 2)
        (t 3)))

(defun print-code-2 (x/int y/int z/int)
  (return-from print-code-2
    (cond ((> x y) 1)
          ((< y z) 2)
          (t 3))))
```

The main difference between them is that the `cond` form in the first function is parsed as an expression while in the second is parsed as a statement. This causes differences in the corresponding Java code:

```
public static int printCode1(int x, int y, int z) {
    if (x > y) {
        return 1;
    } else if (y < z) {
        return 2;
    } else {
        return 3;
    }
}

public static int printCode2(int x, int y, int z) {
    return (x > y) ? 1 : (y < z) ? 2 : 3;
}
```

Note that the first use of `cond` is translated as a composition of Java *conditional expressions* while the second use is translated into Java *if statements*.

There is one subtlety that makes `cond` statements a bit different from `cond` expressions when we use of multiple consequents in `cond` clauses. These multiple consequents corresponds to an implicit `progn` that Linj will attempt to translate Java. As was already discussed in section 1.7.6, this translation is less clear (and generates less efficient code) for the expression case than for the statement case.

The use of the exact same syntax to express statement conditionals and expression conditionals has the advantage of allowing the code to move to different places without concern to the syntactical category that it assumes. For example, one common refactoring operation in Common Lisp is to cache (with a `let` form) the result of a `cond` so that it can be used in more than one place. In this case, the `cond` form changes its syntactic category from a (possibly) statement to a (definitely) expression but the only effect is on the generate Java code.

Conditionals and Type Inference

Linj's conditionals interact with Linj type inferencer in a very subtle way. We will explain this starting with an example. Let's consider the Java AWT class hierarchy, where a `TextField` and `TextArea` both inherit from `TextComponent`. Let's imagine the following alternatives for the Java "function" that returns an appropriate widget depending on the number of lines it must have:

```
public static TextComponent getTextWidget(int lines) {
    if (lines == 1) {
        return new TextField();
    } else {
        return new TextArea();
    }
}

public static TextComponent getTextWidget(int lines) {
    return (lines == 1) ? new TextField() : new TextArea();
}
```

Most Java programmers will consider both functions as equivalent but, in fact, they aren't: the second one doesn't even compile! The problem is that a Java's conditional expression has restrictions that are not present in Java's `if`-statement, namely the fact that when the branches of the conditional expression belong to different reference types, it must be possible to convert one of the types to the other type by assignment conversion. It is a compile-time error if neither type is assignment compatible with the other type, as it happens in the above example.

Fortunately, this annoying "feature" is already acknowledge by the Linj compiler. Consider the "equivalent" Linj function:

```
(defun get-text-widget (lines/int)
  (return-from get-text-widget
    (if (= lines 1)
      (new 'text-field)
      (new 'text-area))))
```

and note that, in the translation presented bellow, the appropriate type casts were inserted to comply with the Java language semantics.

```
public static TextComponent getTextWidget(int lines) {
    return (lines == 1) ? (TextComponent)new TextField() : (TextComponent)new TextArea();
}
```

Obviously, these types casts are computed according to the type hierarchy. It is not possible to return (using either an `if`-statement or an `if`-expression) values of incompatible types.

The `case` is also implemented in Linj. Again, Linj tries really hard to overcome Java limitations. Let's see an example:

```
(defun test-number (n/int)
  (case n
    (1 (princ "one"))
    ((2 3) (princ "two or three"))
    (t (princ "other number"))))
```

Translating to Java, we get:

```
public static void testNumber(int n) {
    switch (n) {
    case 1:
        System.out.print("one");
        break;
    case 2:
    case 3:
        System.out.print("two or three");
        break;
    default:
        System.out.print("other number");
        break;
    }
}
```

Let's change the example just a bit: instead of accepting an `int`, the function `test-number` will accept a `long`. Here is the new translation to Java:

```
public static void testNumber(long n) {
    if (n == 1) {
        System.out.print("one");
    } else if ((n == 2) || (n == 3)) {
        System.out.print("two or three");
    } else {
        System.out.print("other number");
    }
}
```

As is possible to see, the translation result is very different. The reason for the difference is that Java's `switch` statement can only be used for an expression of type `char`, `byte`, `short`, or `int`. All other data types produce a compilation error. The Linj programmer can be confident that the Linj compiler is aware of this restrictions and will generate the best possible code.

When the `case` form is used as an expression some care must be taken because the `case` cannot be translated to a Java `switch` (independently of the type of the `case` argument) and will be translated to a `cond` in an intermediate step. Obviously, the same restrictions that we described for `cond`s still apply.

Besides `case`, Linj also implements `ecase`. The `ccase` is not implemented because, due to the huge mismatch between the interaction models of Common Lisp and Java, there's no simple translation to Java.

The forms `typecase` and `etypecase` are also implemented but they can only test type-membership for reference types. Due to the object-oriented capabilities of Linj, these forms shouldn't be used as they are a sign of bad design in object-oriented applications. However, given the fact that some of the Java libraries (particularly the AWT) already show such signs, sometimes they are useful. For example, to obtain the number of columns of an AWT `TextComponent`, a method that is defined on all subclasses but not on the class itself, we can write something of the form:

```
(defun get-text-columns (text-obj/text-component)
  (etypecase text-obj
    (text-area (get-columns (the text-area text-obj)))
    (text-field (get-columns (the text-field text-obj)))))
```

and get it translated automatically to:

```
public static int getTextColumns(TextComponent textObj) {
    if (textObj instanceof TextArea) {
        return ((TextArea)textObj).getColumns();
    } else if (textObj instanceof TextField) {
        return ((TextField)textObj).getColumns();
    } else {
        throw new Error("" + textObj + " fell through ETYPECASE expression.");
    }
}
```

1.7.9 Blocks and Exits

Linj provides the `block` and `return-from` constructs with semantics similar to Common Lisp but with some restrictions. Just like Common Lisp, function and method definitions automatically insert an hidden `block` with the same name as the function or method so that one can use `return-from` to return prematurely.

There are two fundamental restrictions upon blocks and exits. The first one is that, in Linj, the exit cannot cross the function or method boundary. This reduces somewhat the usefulness of blocks and exits when in presence of `lambdas` but it allows for a simpler translation to Java's labeled statements. The second one is that you can only return a value to the outermost block. Inner blocks cannot receive values.

Here is an example of the use of these constructs:

```
(defun test-block (x/int)
  (block xpto
    (when (> x 1)
      (block xyz
        (if (< x 3)
          (return-from xpto)
          (return-from xyz))))))
```

The translation of the previous example to Java produces an interesting result:

```
public static void testBlock(int x) {
    if (x > 1) {
        xyz: if (x < 3) {
```

```
                return;
            } else {
                break xyz;
            }
    }
}
```

Note that the return to the outermost block was translated into a Java return statement as, in fact, there's nothing to do after the outermost block. The return to the inner block was translated into a Java `break` statement. Note also that the outer block was removed because there was no need for it.

Just like in Common Lisp, several Linj iteration forms (e.g., `do`, `loop`, etc.) include implicit `nil`-labeled blocks around its body. This allows for simply `return` instead of `return-from`.

1.7.10 Iteration

Linj implements most iteration forms available in Common Lisp with the notorious exception of the extended `loop` and `tagbody`. We will now examine all the different iteration forms.

1.7.11 Indefinite Iteration

The simplest of the iterative forms is the `loop` that simply executes its body repeatedly. Just like in Common Lisp, the Linj `loop` form establishes an implicit block named `nil` around its body so that we can use the `return` form to break the loop.

Here is a version of the factorial function implemented using the `loop` form.

```
(defun fact (n/int)
  (let ((r 1))
    (loop
     (if (= n 0)
       (return r)
       (setq r (* r n)
             n (1- n))))))
```

Translating the above function to Java produces the following:

```
public static int fact(int n) {
    int r = 1;
    while (true) {
        if (n == 0) {
            return r;
        } else {
            r = r * n;
            --n;
        }
    }
}
```

1.7.12 General Iteration

In the above situation, the main occupation of the loop is to iteratively recompute the value of the variables `n` and `r`. For these cases, it is preferable to use the `do` or `do*` forms. Note that these Linj forms, contrary to their Common Lisp counterparts, do not provide an implicit tagbody.[15] on the other hand, they do provide an implicit `nil` block so that you can `return` prematurely from them.

Here is the factorial function rewritten using the `do`:

```
(defun fact (n/int)
  (do ((r 1 (* r n)))
      ((= n 0) r)
    (decf n)))
```

The translation to Java is:

```
public static int fact(int n) {
    int r = 1;
    for (; n != 0; r = r * n) {
        --n;
    }
    return r;
}
```

The factorial function is so simple that, in fact, the `do` doesn't even need a body.

```
(defun fact (n/int)
  (do ((i 1 (1+ i))
       (r 1 (* r i)))
      ((> i n) r)))
```

```
public static int fact(int n) {
    int r = 1;
    for (int i = 1; i <= n; r = r * i, ++i) {
    }
    return r;
}
```

The reader should note the "reordering" of the `do` variables to allow for its "parallelization" in Java. Sometimes, this requires the introduction of extra variables. The `do*`, obviously, doesn't have such requirements.

1.7.13 Simple Iteration Constructs

Just like in Common Lisp, the constructs `dolist` and `dotimes` execute a body of code once for each value taken by a single variable. In the `dolist` case, the variable takes as value successive elements of a list. In the `dotimes` case, the variable takes as value all integers from `0` to `n-1` for some specified positive integer `n`.

Here is the factorial function written using the `dotimes`:

```
(defun fact (n/int)
  (let ((r 1))
    (dotimes (i n r)
      (setq r (* (1+ i) r)))))
```

[15] This a very rarely used feature of the Common Lisp's general iteration forms. We suspect that the large majority of Common Lisp programmers aren't even aware of its existence.

```
public static int fact(int n) {
    int r = 1;
    for (int i = 0; i < n; ++i) {
        r = (i + 1) * r;
    }
    return r;
}
```

1.7.14 Multiple Values

Multiple returned values is a very useful feature of Common Lisp. In Common Lisp, if the caller of a function does not request multiple values, but the called function produces multiple values, then the first value is given to the caller and all others are discarded; if the called function produces zero values, then the caller gets `nil` as a value.

Linj implementation of multiple values is very similar to Common Lisp but, as usual, there are some subtle differences. In this case, they cause losses in expressive power but they also increase the error checking.

The major difference between Linj and Common Lisp in what regards multiple values occurs when the caller is expecting more than the values returned by the callee. While Common Lisp silently provides a `nil` for all expected values that were not received, Linj loudly generates an error. Related to this problem is the restriction that all exit points of a function must returned the same number of values.

Apart from these restrictions, Linj implements the same semantics as Common Lisp, particularly the rule that any function that produces multiple values can be used in a context where just one is expected. Linj methods and functions can return multiple values using the `values` form. All or some of these values can be captured using `multiple-value-bind or` \verbnth-value—. No other constructs form multiple values are available, at the moment.

Linj's implementation of multiple values is based on returning an array containing all the values. Linj will choose the most specific array that can accommodate the types of the returned values.

Here is an example of the use of multiple values and its translation into Java:

```
(defun polar (x/double y/double)
  (values (sqrt (+ (* x x) (* y y))) (atan y x)))

(defun hypotenuse (x/double y/double)
  (multiple-value-bind (r theta)
      (polar x y)
    r))
```

```
public static double[] polar(double x, double y) {
    return new double[] { Math.sqrt((x * x) + (y * y)), Math.atan2(y, x) };
}

public static double hypotenuse(double x, double y) {
    double[] multipleResults = polar(x, y);
    double r = multipleResults[0];
    double theta = multipleResults[1];
    return r;
}
```

Note in the previous example that the `hypotenuse` function ignores the second value returned by the `polar` function. In this case, a `nth-value` would seem more appropriate, i.e.:

```
(defun hypotenuse (x/double y/double)
  (nth-value 0 (polar x y)))
```

Its translation to Java is now a simpler fragment:

```
public static double hypotenuse(double x, double y) {
    return polar(x, y)[0];
}
```

Note that a function that returns multiple results can also be used where a one-valued function is expected. As in Common Lisp, we use just the first value. Here is an example:

```
(defun right-triangle-perimeter (x/double y/double)
  (+ x y (polar x y)))
```

The translation is

```
public static double rightTrianglePerimeter(double x, double y) {
    return x + y + polar(x, y)[0];
}
```

As we said, Linj translation for multiple values employs Java arrays. However, Linj is smart enough to use the most specific array it can to minimize cast and boxing operations. In the previous examples, `double[]` arrays are used because all returned values are `double`s. However consider the following functions:

```
(defun produce-funny-values ()
  (values 1 #\a 2.0 "Hi!"))

(defun receive-funny-values ()
  (multiple-value-bind (i c d s)
      (produce-funny-values)
    ...))
```

When we translate them to Java we get:

```
public static Object[] produceFunnyValues() {
    return new Object[] { new Long(1), new Character('a'), new Double(2.0f), "Hi!" };
}

public static int receiveFunnyValues() {
    Object[] multipleResults = produceFunnyValues();
    int i = ((Number)multipleResults[0]).intValue();
    char c = ((Character)multipleResults[1]).charValue();
    float d = ((Number)multipleResults[2]).floatValue();
    String s = (String)multipleResults[3];
    ...
}
```

Note that the `produce-funny-values` couldn't find a common type for all returned values (some of them are primitive values while others are reference values) and was forced to wrap them so that they can be fitted in an `object` array. However, the receiver knows the real types that were intended to be returned so it knows how to unwrap the returned values.

Other translation schemes would be possible, e.g., returning an instance of class (even an inner class) containing the intended values. However, using arrays is very efficient, in particular in the most common case where all returned values are of the same type.

1.7.15 Dynamic Non-Local Exits

Linj does not implement the traditional dynamic non-local exits constructs of Common Lisp, namely `catch` and `throw`. However, Linj implements the underlying concept of *exception*, i.e., a change in the normal flow of control that is caused either by errors or by an explicit request from the programmer.

We will discuss exceptions on a later chapter. For now, we will only present the `unwind-protect` special form, necessary for protection against dynamic exits.

Here is the prototypical example:

```
(defun process-file ()
  (open-file)
  (unwind-protect
      (process-file)
    (close-file)))
```

and its translation using Java's `try-finally` forms:

```
public static void processFile() {
    openFile();
    try {
        processFile();
    } finally {
        closeFile();
    }
}
```

1.8 Declarations

Declarations allow you to specify extra information about your program to the Linj compiler. Due to the compile-time nature of Linj type inferencer, type declarations are not optional. However, there are simplified ways to write them.

Declarations in Linj are specified using exclusively the `declare` form. The syntax is the same as in Common Lisp, including the shorthand notation for type declarations. However, the set of recognized declaration identifiers is different from Common Lisp's. In Linj, they are: `type`, `returns`, `throws`, `visibility`, `category`, `modifier`. Their syntaxes are as follows:

- (type *typespec variable**) and *typespec* is one of the Linj types
- (*typespec variable**) This is the shorthand notation for type declarations.

- (returns *typespec*) This rarely used declaration allows for declaring the returned type of a function or method.
- (throws *typespec**) This allows the specification of the *throws* raised by a function or method.
- (visibility *spec*) where *spec* is one of `:public`, `:protected`, and `:private`. This affects the accessibility properties of the generated Java code.
- (category *spec*) where *spec* is one of `:abstract`, `:reader`, and `:writer`. Its main use is for declaring a method as abstract.
- (modifiers *spec**) where *spec* is one of `:public`, `:protected`, `:private`, `:abstract`, `:static`, `:final`, `:synchronized`, `:native`, `:strictfp`. This allow full control over the method qualifiers that will be generated in the Java code.

Some of the declaration identifiers have the same effect. For example, `(declare (visibility :private))` and `(declare (modifier :private))` produce the same effects but one may be clearer than the other, depending on the intent.

Regarding type declarations, there is also a shorthand notation for the shorthand notation. Whenever a variable is written in the form *variable*/*type*, the *type* will be associated to the variable as if by a `declare` form.

Here is an example demonstrating the different forms of type declarations:

```
(defun fact (n)
  (declare (type int n))
  (if (= n 0) 1 (* n (fact (1- n)))))

(defun fact (n)
  (declare (int n))
  (if (= n 0) 1 (* n (fact (1- n)))))

(defun fact (n/int)
  (if (= n 0) 1 (* n (fact (1- n)))))
```

All of them produce the exact same result after translation to Java.

The programmer will rarely need to use declaration identifiers besides the `type` or its shorthand notation. All the others have reasonable defaults. However, when the need arises, it's good to know that Linj allows full control over the generated Java code. Here's an example:

```
(defun paranoid (a b c)
  (declare (int a b)
           (boolean c)
           (returns double)
           (visibility :private)
           (throws i-o-exception write-aborted-exception)
           (modifiers :abstract :final :synchronized)))
```

and its translation:

```
public static synchronized abstract double paranoid(int a, int b, boolean c)
  throws WriteAbortedException, IOException;
```

1.8.1 Type Declarations for Forms

Sometimes, it is necessary to declare that the value produced by the evaluation of some form will be of a particular type. Using `declare` one can declare the type of the value held by a variable, but there is no easy way to declare the type of the value of an unnamed form. Just like in Common Lisp, Linj provides the `the` type declaration for this purpose.

Usually, Linj type inferencer is sufficiently clever to infer all type information it needs. However, there are situations where type information is lost. For example, when we collect several types of objects on a list, their original types are lost because a list is a generic container prepared to deal with `objects`. Without extra information, all the "things" we can extract from a list are of type `object`.

To solve this problem, Linj provides the `the` form, that operates like a type declaration over the value returned by the evaluation of its argument.

Here is an example where type information is needed:

```
(let ((c (cons "one" 2)))
  (princ (the string (car c)))
  (princ (the int (cdr c))))
```

The translation shows some interesting type conversions:

```
Cons c = new Cons("one", linj.Bignum.valueOf(2));
System.out.print((String)c.car());
System.out.print(((java.lang.Number)c.cdr()).intValue());
```

1.9 Macros

Linj has four different types of macros but three of them are used at the compiler level and its definition requires some knowledge of the compiler internals. We will not explain them here.

The fourth type of macro is exactly like Common Lisp macros. In fact, they *are* Common Lisp macros and, as a result, they have the exact same syntax and expressive power. The only difference between Common Lisp macros and Linj macros is that, in Linj, the result of the macro expansion is parsed as Linj program and compiled to a Java program.

1.9.1 Defining Macros

Macros are defined in Linj using the `defmacro` form which has the same semantics as in Common lisp. Contrary to other Linj forms that imitate Common Lisp, a Linj macro *is* a Common Lisp form and uses Common Lisp functions and macros to produce its expansion. Obviously, it is possible to use *destructuring* on the arguments and the lambda-list may contain the lambda-list keywords `&optional`, `&rest`, `&key`, `&allow-other-keys`, `&aux`, `&body`, `&whole`. Only the Common Lisp lambda-list keyword `&environment` is unavailable.

Here is a macro example extracted directly from [1]:

```
(defmacro arithmetic-if (test neg-form
                         &optional zero-form pos-form)
```

```
  (let ((var (gensym)))
    `(let ((,var ,test))
       (cond ((< ,var 0) ,neg-form)
             ((= ,var 0) ,zero-form)
             (t ,pos-form)))))


(defun fortran-example (x/double)
  (arithmetic-if (- x 4.0)
                 (- x)
                 (error "Strange zero")
                 x))
```

The translation to Java is:

```
public static double fortranExample(double x) {
    double g1757 = x - 4.0f;
    if (g1757 < 0) {
        return - x;
    } else if (g1757 == 0) {
        throw new Error("Strange zero");
    } else {
        return x;
    }
}
```

1.9.2 Avoiding Name Capture

Due the human-readable nature of the generated Java code, the use of `gensyms` deserves a bit of attention. In fact, they have several problems in Linj:

- The least we can say is that they don't look nice. In Common Lisp, the look doesn't matter because the programmer rarely has to see the generated symbols but, in Java, things are different: (we hope that) no Java programmer will ever write or expect to read variable names such as `g1757`. The truth is that the use of `gensym` will never look nice in Java.

- The main use of `gensyms` is to avoid name conflicts that could result in unintended variable capture. This is possible because `gensyms` are *uninterned* symbols, guaranteed not to be `eq` to any other symbols. The translation to Java, however, can't benefit from that and symbols from different packages (or from no package, like `gensyms`) that have the same name will end up as the same "symbol" in Java. Thus, name conflicts can easily arise.

These problems make the use of `gensyms` much less appropriate in Linj than in Common Lisp. Linj's solution to this problem is the introduction of a macro named `with-new-names` that is very similar to the well-known `with-gensyms`. Just like `with-gensyms`, the macro `with-new-names` guarantees that there are no name conflicts between the new names and already existent names. The difference is that `with-new-names` can generate nice-looking names because it knows the already generated names and it analysis the body to search for possible conflicts.

Using the `with-new-names` macro, our previous example becomes:

```
(defmacro arithmetic-if (test neg-form
                          &optional zero-form pos-form)
  (with-new-names (var)
    `(let ((,var ,test))
       (cond ((< ,var 0) ,neg-form)
             ((= ,var 0) ,zero-form)
             (t ,pos-form)))))
```

```
(defun fortran-example (x/double)
  (arithmetic-if (- x 4.0)
                 (- x)
                 (error "Strange zero")
                 x))
```

and its translation becomes:

```
public static double fortranExample(double x) {
    double var = x - 4.0f;
    if (var < 0) {
        return - x;
    } else if (var == 0) {
        throw new Error("Strange zero");
    } else {
        return x;
    }
}
```

Just to show what it looks like to combine multiple occurrences of the `with-new-names` macro, here is a convoluted extension of the previous example:

```
(defun fortran-example (x/double)
  (arithmetic-if (- x 4.0)
                 (arithmetic-if (- x 2.0)
                                (error "Doooh!")
                                x
                                (- x))
                 (error "Strange zero")
                 (arithmetic-if (- x 3.0)
                                (error "Not good!")
                                (- x)
                                x)))
```

The translation shows the context-aware renaming of the generated names to avoid conflicts:

```
public static double fortranExample(double x) {
    double var = x - 4.0f;
    if (var < 0) {
        double var0 = x - 2.0f;
        if (var0 < 0) {
            throw new Error("Doooh!");
        } else if (var0 == 0) {
            return x;
        } else {
```

```
            return - x;
        }
    } else if (var == 0) {
        throw new Error("Strange zero");
    } else {
        double var0 = x - 3.0f;
        if (var0 < 0) {
            throw new Error("Not good!");
        } else if (var0 == 0) {
            return - x;
        } else {
            return x;
        }
    }
}
```

It is generally accepted in the Lisp world that macro-writing is a bit more difficult than function-writing. To simplify the task of the macro-writer, Common Lisp provides some helper functions such as `macroexpand` and `macroexpand-1` that allow the programmer to see the *expansion* of a macro call.

Unfortunately, due to the differences between the interactiveness models of Linj and Common Lisp, there's no `macroexpand` or `macroexpand-1` in Linj. Fortunately, there's no need for them because the Linj macros can be tested on the Common Lisp environment before using them in Linj.

1.10 Data Types

Linj provides a large number of data types. Some are translated into Java primitive types or Java library types while others are defined by Linj and need some run-time support. We will now describe all the types already provided by Linj.

1.10.1 Numbers

In Linj, numbers can be represented as immediate values or as reference values. In the first case, the numbers must belong to one of the primitive types `byte`, `short`, `int`, `long`, `float`, and `double`. In the second case, they must belong to the reference types `bignum`, `big-integer`, `big-decimal`, or to one of the *wrapper* types `integer`, `long`[16], `float`, or `double`. There is no provision for complex number, although it can be easily be added.

It is important to understand that Linj programs follow the exact same lexical rules as Common Lisp programs and, in fact, the Linj compiler reads Linj code using the Common Lisp reader. This means that numbers can be written using Common Lisp's conventions and the read-macros for reading with radices different that ten are available. For example, `#b101` means the number `5` in binary. Using either notation, this will correspond to a literal of type `int`.

The primitive types `byte`, `short`, `int`, and `long` correspond to what Common Lisp calls *fixnums* and they are much more efficient than the reference values. The reference type `bignum` correspond to the union of Common Lisp's *bignums* and *ratios*.[17]

[16] Note the capitalization.

[17] Note that, contrary to Common Lisp, the Linj's *bignums* include an equivalent (but computationally more expensive) representation for *fixnums*.

Usually, Linj knows the best representation for a literal depending on its size. Integers in the range `-2147483648` to `2147483647` inclusive are considered `ints`. Outside the previous range, from `-9223372036854775808` to `9223372036854775807` inclusive, they are treated as `longs`. Finally, outside the previous range all literal integers are treated as `bignums`.

It is possible to override Linj's rules for representing literal numbers by suffixing the numeric representation with a text fragment that discriminates the intended type. The possibilities are:

- `l` or `L`, to request a `long` literal.
- `big` or `Big`, to request a `bignum` literal.
- `bigint` or `Bigint`, to request a `big-integer` literal.
- `bigdec` or `Bigdec`, to request a `big-decimal` literal.

Thus, 1L has the same meaning as in Java, designating the `long` value `1`. On the other hand, 1Big is the Linj (small) `bignum 1`.

Ratios are also represented using Linj's `bignum`. This type is used because it preserves the precision on all numeric operations. Thus `1/3` is a `bignum`. However, due to Common Lisp's rule of *rational canonicalization*, the reader should exercise some care when writing ratio literals. For example, `455/13` isn't a Linj `bignum` but the `int 35`.

Regarding the floating-point literals, `1.0` is a `float` but `1.0d0` is a `double`.

Here is an example of numeric literals:

```
(let ((a 1)
      (b 12345678901234567890)
      (c 123456789012345678900)
      (d 1/2)
      (e 2/6)
      (f 1L)
      (g 1Big)
      (h 1Bigint)
      (i 123456789012345678901234567890Bigint)
      (j 0.999999999999999999999999999999999999999999)
      (k 0.999999999999999999999999999999999999999999Bigdec))
  ...)
```

Its translation into Java is the following:

```
int a = 1;
long b = 1234567890123456789L;
Bignum c = Bignum.valueOf("12345678901234567890");
Bignum d = Bignum.valueOf(1, 2);
Bignum e = Bignum.valueOf(1, 3);
long f = 1L;
Bignum g = Bignum.valueOf(1);
BigInteger h = BigInteger.valueOf(1);
BigInteger i = new BigInteger("123456789012345678901234567890");
float j = 1.0f;
BigDecimal k = new BigDecimal("0.999999999999999999999999999999999999999");
...
```

Note the difference between the last two variable declarations. One is a `float` and suffers from roundoff errors while the other is a `BigDecimal` and uses the necessary precision to exactly represent the given number.

Here is one example of numeric operations over numbers of different types:

```
(let ((n 2)
      (m 2Big))
  (let ((n+n (+ n n))
        (m+m (+ m m)))
    (setq m (* n+n m))
    (setq n (* (- n) (- m)))))
```

The corresponding Java code is:

```
int n = 2;
Bignum m = Bignum.valueOf(2);
int nPlusN = n + n;
Bignum mPlusM = m.add(m);
m = Bignum.valueOf(nPlusN).multiply(m);
n = Bignum.valueOf(- n).multiply(m.negate()).intValue();
```

Predicates on Numbers

Linj implements the predicates `zerop`, `plusp`, `minusp`, `oddp`, and `evenp` over all number types.

Comparisons on Numbers

Linj implements the Common Lisp relational operators `=`, `/=`, `<`, `>`, `<=`, and `>=`. These functions each take one or more arguments.

Here is an example involving `int`s:

```
(defun quux (x/int y/int z/int w/int)
  (if (= x y)
    (< y z)
    (if (or (<= 2 z 6) (> x y z w))
      (/= z w)
      (< 3))))
```

and its translation:

```
public static boolean quux(int x, int y, int z, int w) {
    if (x == y) {
        return y < z;
    } else if (((2 <= z) && (z <= 6)) || ((x > y) && (y > z) && (z > w))) {
        return z != w;
    } else {
        return true;
    }
}
```

Note that when the relational operator have more than two arguments, some of those arguments will be evaluated more than once. To avoid problems, Linj restricts those arguments to be literals or variables.

It's interesting to compare the previous fragments with the following ones that differs only in the parameter types that are now `bignum`s:

```
(defun quux (x/bignum y/bignum z/bignum w/bignum)
  (if (= x y)
    (< y z)
    (if (or (<= 2 z 6) (> x y z w))
      (/= z w)
      (< 3))))
```

```
public static boolean quux(Bignum x, Bignum y, Bignum z, Bignum w) {
    if (x.compareTo(y) == 0) {
        return y.compareTo(z) < 0;
    } else if (((Bignum.valueOf(2).compareTo(z) <= 0) && (z.compareTo(Bignum.valueOf(6)) <= 0)) ||
               ((x.compareTo(y) > 0) && (y.compareTo(z) > 0) && (z.compareTo(w) > 0))) {
        return z.compareTo(w) != 0;
    } else {
        return true;
    }
}
```

The `max` and `min` are also implemented for any number or arguments and number types. Here are some examples:

```
(defun max-3 (x/int y/int z/int)
  (max x y z))

(defun max-3 (x/bignum y/bignum z/bignum)
  (max x y z))

(defun max-3 (x/int y/bignum z/long)
  (max x y z))
```

And here is the translation:

```
public static int max3(int x, int y, int z) {
    return Math.max(x, Math.max(y, z));
}

public static Bignum max3(Bignum x, Bignum y, Bignum z) {
    return x.max(y).max(z);
}

public static Bignum max3(int x, Bignum y, long z) {
    return Bignum.valueOf(x).max(y).max(Bignum.valueOf(z));
}
```

Arithmetic Operations

Linj implements all arithmetic operations over primitive number types and most of them over non-primitive types. The functions `+`, `-`, `*` work over all number types. The function `/` works for all number types *except* `big-integer`. Note that `/` will produce a ratio only if the arguments are `bigints` and their mathematical quotient is not an exact integer.

Here are some examples of arithmetic expressions:

```
(defun arithmetic (x/big-decimal y/big-decimal)
  (let ((a (+ 1 2 3))
        (b (/ 1 2 3.0))
```

```
        (c (- 1 2 3Big)))
    (let ((negate-a (- a))
          (inverse-b (/ b))
          (negate-c (- c)))
      (* c a))))
```

and the corresponding translation:

```
public static Bignum arithmetic(BigDecimal x, BigDecimal y) {
    int a = 1 + 2 + 3;
    float b = (1 / 2) / 3.0f;
    Bignum c = Bignum.valueOf(1).subtract(Bignum.valueOf(2)).subtract(Bignum.valueOf(3));
    int negateA = - a;
    float inverseB = 1 / b;
    Bignum negateC = c.negate();
    return c.multiply(Bignum.valueOf(a));
}
```

To divide one integer by another producing an integer result, Linj provides the functions `floor` and `round`.

Linj also implements the `1+` and `1-` functions and the `incf` and `decf`.

Logical Operations on Numbers

In Linj, the logical operations described in this section require `int`s or `long`s as arguments and they are treated as if they were represented in two's-complement notation. Linj provides the functions `lognot`, `logtest`, `logior`, `logxor`, `logand`, `logeqv`, `lognand` , `lognor`, `logandc1`, `logandc2`, `logorc1`, and `logorc2`, as well as the "generic" `boole`. In this last case, the operation argument must be known at compile-time.[18]

Random Numbers

Linj provides a `random` function which is very similar to the Common Lisp version. With a single argument, it produces a random number of the same type between 0 (inclusive) and that argument (exclusive). With two arguments, the additional argument is taken as a random state that maintains the state of the pseudo random number generator.

Linj does not provide access to the default random state (in Common Lisp, it is kept in the `*random-state*`). However, it is possible to create new random states using the function `make-random-state` with the optional argument `t`. No other argument is accepted (nor even *no* argument). An object is recognized as a random state if it satisfies the `random-state-p` predicate.

1.10.2 Characters

Characters belong to the type `char` or to its wrapper type `character`. Characters use the same `#\` notation used in Common Lisp. Here is an example:

```
(let ((a #\a)
      (b #\space)
```

[18] This somewhat limits the usefulness of the `boole` function. In fact, it becomes just a syntactic variant of the remaining logical operations.

```
      (c #\tab)
      (d #\newline)
      (e #\return)
      (f #\')
      (g #\\))
  ...)
```

The translation into Java is:

```
char a = 'a';
char b = ' ';
char c = '\t';
char d = '\n';
char e = '\r';
char f = '\'';
char g = '\\';
...
```

Regarding the available operations, Linj implements all Common Lisp relational operators for characters (such as `char=`, `char/=`, `char<`, `char-equal`, `char-lessp`, etc.), the predicates `digit-char-p`, `alpha-char-p`, and several conversion functions (such as `char-code`, `char-upcase`, `char-downcase`, `char-digit`, etc.).

1.10.3 Strings

Linj implements strings on top of Java `String` class. This has the immediate implication that Linj's strings are immutable. If the programmers needs mutable strings it can instantiate `string-buffer`s instead.

Contrary to Common Lisp, Linj strings are not a subtype of vectors. Nevertheless, the same set of fundamental operators `char` and `aref` can be used.

The `string` type implements the usual string transformation functions of Common Lisp, such as `string-upcase` and `string-downcase`. Some of the operations (such as `string-capitalize`) require an additional library that is automatically included when needed. All destructive variants `nstring-upcase`, `nstring-downcase`, and `nstring-capitalize` are not implemented because Linj strings are immutable.

Here is one example of the string operations available:

```
(defun main ()
  (let ((s "Hello, world    "))
    (dotimes (i (length s))
      (format t "~C" (aref s i)))
    (princ (concatenate 'string
                        (string-trim '(#\ ) s)
                        #\?
                        " "
                        (string-upcase s :start 6)))))
```

and its translation:

```
public static void main(String[] outsideArgs) {
    String s = "Hello, world    ";
```

```
    int limit = s.length();
    for (int i = 0; i < limit; ++i) {
        System.out.print(s.charAt(i));
    }
    System.out.print(s.trim() + "?" + " " + (s.substring(0, 6) + s.substring(6).toUpperCase()))
}
```

1.10.4 Symbols

Symbols in Linj are much simpler than in Common Lisp. Symbols are not partitioned into *packages* (which has a completely different meaning in Linj) and they do not have a function cell. However, they do have value cell and property list cell. It is important to know that Linj reads code using an `:invert` readtable case. Usually, this is what Linj programmers want[19] but the presence of escaping characters might produce unexpected results. Here is an example of several Linj symbols and its translation into Java:

```
(let ((a 'lowercase)
      (b 'UPPERCASE)
      (c 'mixedCase)
      (d 'a\ funny\ symbol)
      (e 'A\ FUNNY\ SYMBOL)
      (f 'a\ FUNNY\ symbol)
      (g '|a funny symbol|)
      (h '|A FUNNY SYMBOL|)
      (i '|a FUNNY symbol|)
      (j (intern "another funny symbol"))
      (k ':keyword)
      (l 'nil))
  ...)
```

```
Symbol a = Symbol.intern("lowercase");
Symbol b = Symbol.intern("UPPERCASE");
Symbol c = Symbol.intern("mixedCase");
Symbol d = Symbol.intern("a funny symbol");
Symbol e = Symbol.intern("A FUNNY SYMBOL");
Symbol f = Symbol.intern("a FUNNY symbol");
Symbol g = Symbol.intern("A FUNNY SYMBOL");
Symbol h = Symbol.intern("a funny symbol");
Symbol i = Symbol.intern("a FUNNY symbol");
Symbol j = Symbol.intern("another funny symbol");
Symbol k = Symbol.intern(":keyword");
Cons l = Cons.EMPTY_LIST;
...
```

Note that the last expression used—`'nil`—does not represent a symbol but the empty list instead.

The Value

The value cell of symbol can store any type of object and is accessed (getted and setted) with `symbol-value`. The `set` can also be used to assign a value to a symbol's value cell.

[19] The `:invert` case preserves the original form of symbols written in mixed case. This is necessary to allow a distinction between, e.g., the types `long` and `Long`.

Here is one example adapted from [1] that shows a few uses of the value cell:

```
(setf (symbol-value '*even-count*) 0)
(setf (symbol-value '*odd-count*) 0)

(defun tally-list (list)
  (dolist (element/int list)
    (set (if (evenp element) '*even-count* '*odd-count*)
         (+ element
            (symbol-value (if (evenp element) '*even-count* '*odd-count*))))))

(tally-list '(1 9 4 3 2 7))

(pprint (symbol-value '*even-count*))
(pprint (symbol-value '*odd-count*))
```

and here is its translation:

```
public static void tallyList(Cons list) {
    for (Cons list0 = list; ! list0.endp(); list0 = list0.rest()) {
        int element = ((Number)list0.first()).intValue();
        (((element % 2) == 0) ? Symbol.intern("*even-count*") : Symbol.intern("*odd-count*")).
         set(Bignum.valueOf(element).
              add((Bignum)(((element % 2) == 0) ? Symbol.intern("*even-count*") : Symbol.intern("*odd-cour
                           symbolValue()));
    }
}

static {
    Symbol.intern("*even-count*").setfSymbolValue(Bignum.valueOf(0));
}

static {
    Symbol.intern("*odd-count*").setfSymbolValue(Bignum.valueOf(0));
}

static {
    tallyList(new Cons(Bignum.valueOf(1),
                       Cons.
                        list(Bignum.valueOf(9),
                             Bignum.valueOf(4),
                             Bignum.valueOf(3),
                             Bignum.valueOf(2),
                             Bignum.valueOf(7))));
}

static {
    System.out.print("" + '\n' + Symbol.intern("*even-count*").symbolValue());
}

static {
    System.out.print("" + '\n' + Symbol.intern("*odd-count*").symbolValue());
}
```

The previous program outputs `6` and `20`.

The Property List

Every symbol has a property list cell that can be manipulated with `symbol-plist`, `get`, and `remprop`.

Here is one example adapted from [1]:

```
(defun make-person (first-name last-name)
   (let ((person (gensym "PERSON")))
     (setf (get person 'first-name) first-name)
     (setf (get person 'last-name) last-name)
     person))

(defvar *john* (make-person "John" "Dow"))
(defvar *sally* (make-person "Sally" "Jones"))
(pprint *john*)
(pprint (get *john* 'first-name))
(pprint (get *sally* 'last-name))

(defun marry (man/symbol woman/symbol married-name)
  (setf (get man 'wife) woman)
  (setf (get woman 'husband) man)
  (setf (get man 'last-name) married-name)
  (setf (get woman 'last-name) married-name)
  married-name)

(pprint (marry *john* *sally* "Dow-Jones"))
(pprint (get *john* 'last-name))
(pprint (get (the symbol (get *john* 'wife)) 'first-name))
(pprint (symbol-plist *john*))
```

and here is the translation into Java:

```
public static Symbol makePerson(Object firstName, Object lastName) {
    Symbol person = Symbol.gensym("PERSON");
    person.setfGetKey(firstName, Symbol.intern("first-name"), null, 3);
    person.setfGetKey(lastName, Symbol.intern("last-name"), null, 3);
    return person;
}

public static Object marry(Symbol man, Symbol woman, Object marriedName) {
    man.setfGetKey(woman, Symbol.intern("wife"), null, 3);
    woman.setfGetKey(man, Symbol.intern("husband"), null, 3);
    man.setfGetKey(marriedName, Symbol.intern("last-name"), null, 3);
    woman.setfGetKey(marriedName, Symbol.intern("last-name"), null, 3);
    return marriedName;
}

public static Symbol starJohnStar = makePerson("John", "Dow");

public static Symbol starSallyStar = makePerson("Sally", "Jones");

static {
    System.out.print("" + '\n' + starJohnStar);
}

static {
    System.out.print("" + '\n' + starJohnStar.getKey(Symbol.intern("first-name"), null, 1));
}

static {
    System.out.print("" + '\n' + starSallyStar.getKey(Symbol.intern("last-name"), null, 1));
```

```
}

static {
    System.out.print("" + '\n' + marry(starJohnStar, starSallyStar, "Dow-Jones"));
}

static {
    System.out.print("" + '\n' + starJohnStar.getKey(Symbol.intern("last-name"), null, 1));
}

static {
    System.out.
     print("" +
           '\n' +
           ((Symbol)starJohnStar.getKey(Symbol.intern("wife"), null, 1)).
            getKey(Symbol.intern("first-name"), null, 1));
}

static {
    System.out.print("" + '\n' + starJohnStar.symbolPlist());
}
```

and here is the output of the program:

```
PERSON0
John
Jones
Dow-Jones
Dow-Jones
Sally
(wife PERSON1 last-name "Dow-Jones" first-name "John")
```

The generic accessors over *disembodied* property lists `getf` and `remf` are also implemented.

The Print Name

The print name of a symbol is the name that was used to create the symbol, with one exception: when the name used started with a colon `:` then the colon doesn't appear on the print name. This scheme is used to treat "keyword" symbols just like in Common Lisp. Remember that, in Linj, symbols do not have package cell and, as result, you should write symbols without any package qualifier, except for symbols that should look like keywords, in which case you should write the empty package qualifier.

To access a symbol print name you can use the function `symbol-name`.

Creating Symbols

Usually, symbols are created automatically by the Linj compiler whenever he sees a quoted symbol. However, the programmer might need to create symbols dynamically and, for this, Linj provides the same machinery as Common Lisp, with the provision that packages are not needed. The available functions are `make-symbol`, `copy-symbol`, `gensym`, and `keywordp`. The function `gensym` is restricted in such way that its optional argument *must* be a string and never a number. The counter `*gensym-counter*` used by the `gensym` function is also available as a `defvar` defined on type `symbol`.

1.10.5 Lists and Conses

Linj provides lists and conses with a `cons` type definition written in Linj itself. Just like in Common Lisp, there are a variety of data structures that can be implemented on top of conses such as lists, dotted lists, association lists, property lists, etc. These, however, are not new data types but just particular organizations of conses. Note that, in Linj, the empty list is represented by `'()` and belongs to the `cons` type and not to the `null` type as in Common Lisp.

We recommend that the reader study carefully the following example:

```
(let ((a '())
      (b '("Hello" "world"))
      (c '("Hello" . ("world" . ())))
      (d '(dotted . list))
      (e '(#\a 3.14159 10))
      (f '(t nil))
      (g `(,t ,nil)))
  ...)
```

and compare it with the translation:

```
Cons a = Cons.EMPTY_LIST;
Cons b = new Cons("Hello", new Cons("world", Cons.EMPTY_LIST));
Cons c = new Cons("Hello", new Cons("world", Cons.EMPTY_LIST));
Cons d = new Cons(Symbol.intern("dotted"), Symbol.intern("list"));
Cons e =
    new Cons(new Character('a'), new Cons(new Double(3.14159f), new Cons(Bignum.valueOf(10), Co
Cons f = new Cons(Symbol.intern("t"), new Cons(Cons.EMPTY_LIST, Cons.EMPTY_LIST));
Cons g = new Cons(new Boolean(true), new Cons(new Boolean(false), Cons.EMPTY_LIST));
...
```

Note that Linj lists cannot contain primitive values such as `int`s or `char`s. However, Linj knows how to *wrap* a primitive value to obtain an "equivalent" reference value. In the case of numbers, it usually uses a `bignum` but if you prefer to use the Java pre-defined wrapper types you just have to change the Linj compiler's special variable `*bignum-arithmetic-p*` to `nil`.

Note also that in the last two expressions, one using *quotation* and the other using *backquotation*. To explain the translation behavior it is sufficient to know that (1) a quoted dotted pair (or list) `'(`*`car`*` . `*`cdr`*`)` is translated into `(cons '`*`car`*` '`*`cdr`*`)` and (2) the rules that we described for quoted symbols still apply.

Regarding the available operations, Linj provides the obvious `cons`, `car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`. No more `c...r` selectors are provided. As expected by Common Lisp programmers, the `car` and `cdr` of the empty list is the empty list. All of the previous functions are `setf`able and it is also possible to use `rplaca` and `rplacd`.

Lists

A list is just a particular view over `cons`es the the Lisp language promote by providing specific functions for its manipulation. For this purpose, Linj implements `endp` as a synonym to `null` and also `first` and `rest` as synonyms to `car` and `cdr`. However, the return type of `rest` is `cons` while that of `cdr` is `object`. This difference is important because it expresses the different uses of those functions: the `cdr` is used to obtain the second element of a pair (an `object`); the `rest` function is used to obtain the rest of a list (another list, i.e., a `cons`).[20] As in Common Lisp, both `first` and `rest` are `setf`able.

To measure the length of a list Linj provides both `length` and `list-length`. The `list-length` function is used to measure the length of a list that can be a circular. However, due to the Linj strict type discipline, when in presence of a circular list, the function can't return `nil` as in Common Lisp so it is modified to return `-1` to type-consistent with the other possible returned values.

To access elements in a list we can use either `nth` or the more direct `first`, `second`, `third`, `fourth`, `fifth`, `sixth`, `seventh`, `eighth`, `ninth`, `tenth`. Any of these functions is also `setf`able. To access the tails of the list Linj provides `nthcdr` and `last`.

Many more list operations are available, including `list` and `list*`, `append` and `nconc`, `copy-list`, `push` and `pop`, `butlast`, `member` and `member-if`, `adjoin`, `remove`, `remove-if-not`, `remove-duplicates`, `delete`, `reverse`, etc.

We now present two examples of the use of list operations, including higher-order ones. The first function returns all prime numbers up to its argument. The second returns all permutations of a list. Here is the Linj code:

```
(defun primes (l)
  (if (endp l)
    (list)
    (cons (first l)
          (primes (remove-if #'(lambda (e/long)
                                  (zerop (rem e (the long (first l)))))
                              (rest l))))))

(defun permutations (list/cons)
  (if (endp list)
    (list (list))
    (mapcan #'(lambda (first)
                (mapcar #'(lambda (rest)
                            (cons first rest))
                         (permutations (remove first list :count 1 :test #'eq))))
            list)))
```

and the obligatory translation:

```
public static Cons primes(final Cons l) {
    if (l.endp()) {
        return Cons.list();
    } else {
        return new Cons(l.first(),
                        primes(l.rest().
                               removeIf(new Predicate() {
                                            public boolean funcall(Object genericE) {
                                                long e = ((Number)genericE).longValue();
                                                return (e % ((Number)l.first()).longValue()) == 0;
                                            }})));
    }
```

[20] Paradoxically, when the `rest` function is repeatedly used over a list we say that we are *cdr*ing-down the list.

```
}

public static Cons permutations(final Cons list) {
    if (list.endp()) {
        return Cons.list(Cons.list());
    } else {
        return list.
                mapcan(new Function() {
                           public Object funcall(final Object first) {
                               return permutations(list.removeKey(first, Predicate2.EQ_FUNCTION
                                      mapcar(new Function() {
                                                 public Object funcall(Object rest) {
                                                     return new Cons(first, rest);
                                                 }});
                           }});
    }
}
```

Association Lists

Association lists have some support in Linj via the functions `acons`, `pairlis`, `assoc`, and `find`.

1.10.6 Hash Tables

Linj implementation of hash tables is based upon the `java.lang.Hashtable` implementation. Since this implementation is for `equals` test[21], no other test is accepted in Linj. Hash table creation, however, accepts the keywords `:size` and `:rehash-threshold` (only if `:size` was also provided).

Hash tables are created with the function `make-hash-table`, tested with `hash-table-p`, counted with `hash-table-count`, accessed with `gethash` and cleared with `remhash` and `clrhash`.

No hash table iterators (such as `maphash` or `with-hash-table-iterator`) are implemented. However, normal Java `Enumerations` can be used.

Here is one example (adapted from [1]) that demonstrates some of the hash table operations, including iterating the hash table keys:

```
(defun main ()
  (let ((turtles (make-hash-table :size 9)))
    (setf (gethash 'howard-kaylan turtles) '(musician lead-singer))
    (setf (gethash 'john-barbata turtles) '(musician drummer))
    (setf (gethash 'leonardo turtles) '(ninja leader blue))
    (setf (gethash 'donatello turtles) '(ninja machines purple))
    (setf (gethash 'al-nichol turtles) '(musician guitarist))
    (setf (gethash 'mark-volman turtles) '(musician great-hair))
    (setf (gethash 'raphael turtles) '(ninja cool rude red))
    (setf (gethash 'michaelangelo turtles) '(ninja party-dude orange))
    (setf (gethash 'jim-pons turtles) '(musician bassist))
    (do ((keys (keys turtles)))
        ((not (has-more-elements keys)))
      (let ((key (next-element keys)))
        (let ((value (the cons (gethash key turtles))))
```

[21] Note that `equals` is neither the same as `equal` nor `equalp` of Common Lisp heritage.

```
          (when (eq (first value) 'ninja)
            (format t "~%~:(~A~): ~{~A~^, ~}" key (rest value)))))))
```

and here is the entire Java program produced:

```
import linj.Util;
import java.util.Enumeration;
import linj.Cons;
import linj.Symbol;
import java.util.Hashtable;

public class Hash extends Object {

    // methods

    public static void main(String[] outsideArgs) {
        Hashtable turtles = new Hashtable(9);
        turtles.put(Symbol.intern("howard-kaylan"), Cons.list(Symbol.intern("musician"), Symbol.intern("le
        turtles.put(Symbol.intern("john-barbata"), Cons.list(Symbol.intern("musician"), Symbol.intern("dru
        turtles.
         put(Symbol.intern("leonardo"),
             Cons.list(Symbol.intern("ninja"), Symbol.intern("leader"), Symbol.intern("blue")));
        turtles.
         put(Symbol.intern("donatello"),
             Cons.list(Symbol.intern("ninja"), Symbol.intern("machines"), Symbol.intern("purple")));
        turtles.put(Symbol.intern("al-nichol"), Cons.list(Symbol.intern("musician"), Symbol.intern("guitar
        turtles.put(Symbol.intern("mark-volman"), Cons.list(Symbol.intern("musician"), Symbol.intern("grea
        turtles.
         put(Symbol.intern("raphael"),
             Cons.list(Symbol.intern("ninja"), Symbol.intern("cool"), Symbol.intern("rude"), Symbol.intern
        turtles.
         put(Symbol.intern("michaelangelo"),
             Cons.list(Symbol.intern("ninja"), Symbol.intern("party-dude"), Symbol.intern("orange")));
        turtles.put(Symbol.intern("jim-pons"), Cons.list(Symbol.intern("musician"), Symbol.intern("bassist
        for (Enumeration keys = turtles.keys(); keys.hasMoreElements(); ) {
            Object key = keys.nextElement();
            Cons value = (Cons)turtles.get(key);
            if (value.first() == Symbol.intern("ninja")) {
                System.out.println();
                System.out.print(Util.stringCapitalizeKey("" + key, 0, 0, 1));
                System.out.print(": ");
                Cons origArgs = value.rest();
                Cons args = origArgs;
                while (! args.endp()) {
                    if (args.endp()) {
                        throw new Error("No more arguments.");
                    } else {
                        Object arg = args.first();
                        args = args.rest();
                        System.out.print(arg);
                    }
                    if (args.endp()) {
                        break;
                    }
                    System.out.print(", ");
                }
            }
        }
    }
}
```

and here is the output of the program:

```
Donatello: machines, purple
Michaelangelo: party-dude, orange
Raphael: cool, rude, red
Leonardo: leader, blue
```

Primitive Hash Function

The function `sxhash` is implemented in Linj but only for reference types. It is translated into Java's `hashCode` method call.

1.10.7 Arrays

Linj's arrays are much simpler than Common Lisp's. In Linj, an array is neither `:adjustable` nor `:displaced-to` and it can't have a `:fill-pointer`. In other words, arrays in Linj are identical to *simple arrays* in Common Lisp.

Like in Common Lisp, it is permissible for a dimension to be zero but it is not possible for its rank to be zero.

Array Creation

The `make-array` creates arrays. It requires that the array dimensions be either a quoted list of numbers or an explicit `list` invocation.[22] The only accepted options are `:element-type` and `:initial-contents`.

The function `vector` is also implemented and is a convenient means for creating arrays with specified initial contents.

Here are some examples of the use of `make-array` and `vector`.

```
(let ((a1 (make-array 5))
      (a2 (make-array '(3 4) :element-type 'byte))
      (a3 (make-array 5 :element-type 'float))
      (v1 (vector 1 2 3))
      (v2 (vector "hi" "there"))
      (v3 (vector (vector 4 5) (vector 6 7))))
  ...)
```

```
Object[] a1 = new Object[5];
byte[][] a2 = new byte[3][4];
float[] a3 = new float[5];
int[] v1 = new int[] { 1, 2, 3 };
String[] v2 = new String[] { "hi", "there" };
int[][] v3 = new int[][] { { 4, 5 }, { 6, 7 } };
...
```

Array Access

Arrays are accessed using `aref`. `setf` may be used with `aref` to destructively replace an array element with a new value. Besides `aref`, Linj also implements the equivalent functions for uni-dimensional arrays `svref`.

One important difference between Linj and Common Lisp is that a multi-dimensional array can be accessed in Linj providing just part of the indexes. In this case, the returned value is the subarray resolved up to the indexes passed.

Here is one example that demonstrates array access:

[22]This is needed to allow computation of the *rank* of the array at compile-time.

```
(let ((array (make-array '(3 4) :element-type 'int))
  (dotimes (i 3)
    (dotimes (j 4)
      (setf (aref array i j) (* i j))))
  (pprint (aref array 2 3))
  (let ((subarray (aref array 2)))
    (pprint (aref subarray 3))))
```

with the corresponding Java translation:

```
int[][] array = new int[3][4];
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 4; ++j) {
        array[i][j] = i * j;
    }
}
System.out.print("" + '\n' + array[2][3]);
int[] subarray = array[2];
System.out.print("" + '\n' + subarray[3]);
```

The execution of the previous code fragment prints `6` twice.

Vectors

Vectors are one-dimensional arrays. Vectors can be written with the `#(...)` syntax: Linj will choose for array type the most general type that covers all the vector's elements. Note the following code fragment:

```
(let ((v1 #(1 2 3))
      (v2 #(1 2 "3"))
      (v3 #("1" "2" "3")))
  ...)
```

the types chosen by the Linj compiler for the corresponding Java code:

```
int[] v1 = new int[] { 1, 2, 3 };
Object[] v2 = new Object[] { linj.Bignum.valueOf(1), linj.Bignum.valueOf(2), "3" };
String[] v3 = new String[] { "1", "2", "3" };
...
```

When the vector elements result from the evaluation process, the proper way to build the vector is to use the `vector` function:

```
(vector 0 (+ 1 2 3) (* 4 5))
```

that corresponds to:

```
new int[] { 0, 1 + 2 + 3, 4 * 5 };
```

1.11 Classes

Linj approach to object-orientation is strongly influenced by the CLOS model. However, to achieve good human-readable translations into Java some restrictions had to be imposed. Linj classes do not support the full multiple inheritance scheme of CLOS but go beyond the simplistic single inheritance/multiple subtyping of Java. Linj methods do not support the full multiple dispatch scheme of CLOS but go beyond the simplistic receiver/arguments of Java.

We will now discuss the Linj model to object-orientation.

1.11.1 Defining Classes

The `defclass` is used to define new classes. Here is one example:

```
(defclass cons ()
  ((car :accessor car :initarg :car)
   (cdr :accessor cdr :initarg :cdr)))
```

The previous class represents a `cons` cell and its definition is just like what we would be doing if we were writing it in CLOS.

Let's now look at the translation of the above class into Java:

```
public class Cons extends Object {

    // constructors

    public Cons(Object car, Object cdr) {
        this.car = car;
        this.cdr = cdr;
    }

    // accessors

    public Object car() {
        return car;
    }

    public void setfCar(Object car) {
        this.car = car;
    }

    public Object cdr() {
        return cdr;
    }

    public void setfCdr(Object cdr) {
        this.cdr = cdr;
    }

    // key methods

    public Cons(Object car, Object cdr, int argsPassed) {
        this(((argsPassed & 1) == 0) ? null : car, ((argsPassed & 2) == 0) ? null : cdr);
    }

    // slots

    protected Object car;

    protected Object cdr;
}
```

Just like in CLOS, the options `:reader`, `:writer`, or `:accessor` provided on each slot definition guide the creation of methods for getting and setting the slot. Note that, just like in CLOS, the option `:accessor` defines a setter that is accessed using using the `setf` form.

The arguments to the `defclass` macro include:

- The name of the new class.

- The list composed of the direct superclass preceded by any number of mixins. If empty, this list defaults to just the `object` class.
- A set of slot specifiers. Each slot specifier includes the name of the slot and zero or more slot options for that slot. These options are:
  - Supplying a default initial value form for the slot (using `:initform`).
  - Requesting that methods be automatically generated for reading or writing slots (using one or more `:reader`, `:writer`, or `:accessor`).
  - Controlling whether a given slot is shared by instances of the class (`:allocation :class`) or whether each instance of the class has its own slot (the default, or `:allocation :instance`).
  - Indicating the expected type for the value stored in the slot (using `:type`). By default, this type is `object`.
  - Indicating the documentation string for the slot (using `:documentation`).
  - Controlling the *accessibility* of the slot, according to the Java's access control rules (using `:visibility` with possible values `:public`, `:private`, or the default `:protected`). This is a Linj-specific extension to `defclass` slot options.
- A set of class options, namely:
  - Supplying a set of initialization arguments and initialization argument defaults to be used in instance creation (using `:default-initargs`).
  - Controlling the *accessibility* of the class, according to the Java's access control rules (using `:visibility` with possible values `:public` (the default), `:protected`), or `:private`. This is a Linj-specific extension to `defclass` class options.
  - Requesting the automatic generation of two methods for parsing and unparsing (*la* `defstruct`) instances of the class (using `:make-parse-method`). This is a Linj-specific extension to `defclass` class options.

1.11.2 Instances

In Common Lisp, `cons` cells are used to implement *lists*. Lists are either a `cons` where the second element is a list or a special value named the *empty list*. Besides, the `car` and `cdr` of an empty list is the empty list.

To provide the concept of empty list in Linj, we just need to create an instance of a `cons` and set its `car` and `cdr` with itself, i.e.:

```
(defconstant +empty-list+ (make-instance 'cons))

(setf (car +empty-list+) +empty-list+
      (cdr +empty-list+) +empty-list+)
```

To create instances of classes we use the `make-instance` form, supplying the name of the class and its (keyword) arguments. Note that, in Linj, the class argument to `make-instance` must be known in compile-time and, more specifically, must be a quoted symbol. Note also that Linj appreciates that the programmer adopts the Common Lisp convention of surrounding the name of constants with `+` characters. This will be used in the translation to Java to force the Java convention of writing the corresponding `static` slots in uppercase, as can be seen bellow:

```
public static final Cons EMPTY_LIST = new Cons(null, null, 0);

static {
    EMPTY_LIST.setfCar(EMPTY_LIST);
    EMPTY_LIST.setfCdr(EMPTY_LIST);
}
```

Note that any code fragments included in the Linj program are translated into Java static blocks that will be automatically executed upon class loading.

It is now possible to create "bigger" lists just by *consing* elements in front of an empty list, as follows:

```
(make-instance 'cons
               :car "First"
               :cdr (make-instance 'cons
                                   :car "Second"
                                   :cdr (make-instance 'cons
                                                       :car "Third"
                                                       :cdr +empty-list+)))
```

Given the fact that pairs and lists are so used, Linj provides special syntax for its construction. As should be expected, the form `(cons` *`car cdr`*`)` can be used in place of `(make-instance 'cons :car` *`car`* `:cdr` *`cdr`*`)`. Besides, the constant `+empty-list+` can be written as `'()`. Using these conventions,[23] the previous example could have be written as:

```
(cons "First" (cons "Second" (cons "Third" +empty-list+)))
```

Finally, the `list` form is also implement as a Linj macro so that `(list` *`first`* `.` *`rest`*`)` is translated into `(cons` *`first`* `(list .` *`rest`*`))` and `(list)` is translated into `'()`. This allows us to write `(list "First" "Second" "Third")` instead.

Any of the forms used above to construct the list containing the strings `"First"`, `"Second"`, and `"Third"` will be translated into:

```
new Cons("First", new Cons("Second", new Cons("Third", Cons.EMPTY_LIST)));
```

1.11.3 Methods

To be finished!

1.12 Conditions

Exceptional situations occur in any programming language. Common Lisp uses the term *condition* to describe such situations and reserves the term *error* for

[23]These conventions can be defined by the programmer using macros.

conditions that prevent normal program execution without some form of intervention.

Unfortunately, the Java model for handling conditions does not have the capabilities found in Common Lisp and this prevents a simple translation between the two models.

Linj approach to a condition system is to follow the Common Lisp model as close as possible but restricting the model wherever necessary to allow an almost direct translation into the Java exception handling model. The most immediate consequence is that Linj does not implement the concept of *restarts* and, therefore, there is no provision for the functions `compute-restarts`, `find-restart`, `invoke-restart`, etc., or for the macros `restart-bind`, `restart-case`, `with-condition-restarts`, etc.

1.12.1 Signaling Conditions

Linj implements the `error` with similar syntax and similar semantics. If the first argument is a condition type, then that condition type is used. Thus, if is possible to write:

```
(error "This is completely wrong")
(error "This is wrong and the cause is ~A" x)
(error file-not-found-exception "Couldn't find the file ~A" file-name))
```

and it will be translated into:

```
throw new Error("This is completely wrong");
throw new Error("This is wrong and the cause is " + x);
throw new FileNotFoundException("Couldn't find the file " + fileName);
```

Due to the limitations of Java exception handling, there are no continuable errors in Linj and, therefore, the `cerror` function is not implemented. The `signal` function is also not implemented.

1.12.2 Assertions

Linj partially implements the Common Lisp facility `assert`. In Linj, the macro accepts just one argument that must be an expression used to check some property. In case the property is not verified, an error is signaled but it is not possible to resume operation: the program aborts.

Here is one example:

```
(defun protected-fact (n/int)
  (assert (>= n 0))
  (fact n))

(defun main (x/int)
  (format t "(fact ~A) -> ~A~%" x (protected-fact x)))
```

One interesting aspect of the `assert` macro is that the error message use the Linj form of the test. Here is a dribble of the program that demonstrates this:

```
$ java Factorial 5
(fact 5) -> 120
$ java Factorial -5
(fact -5) -> Exception in thread "main" java.lang.Error: The assertion (>= n 0) faile
        at Factorial.protectedFact(Factorial.java:18)
        at Factorial.main(Factorial.java:30)
```

Due to the strict type discipline adopted in Linj, the Common Lisp `check-type` macro is almost useless and, therefore, is not implemented. However, the macro `etypecase` is implemented, as is the macro `ecase` (but not `ctypecase` nor `ccase`).

1.12.3 Handling Conditions

To allow a program to gain control when a condition is signaled, Linj provides the `handler-case` macro. There's one restriction: Linj does not support the (rarely used) `:no-error` clause.

```
(defun safe-computation ()
  (handler-case (dangerous-computation)
    (file-not-found-exception (e)
      (format t "Where did I left that book?")
      (print-stack-trace e))
    (e-o-f-exception ()
      (format t "Your are reading more than you should!"))
    (i-o-exception (io-e)
      (format t "Something wrong happened while reading")
      (format t "The real error was ~A" (get-message io-e)))
    (arithmetic-exception ()
      (format t "Please, revise your math!"))))
```

Note that Linj conditions are implemented on top of Java `Throwable`s and allow access to all its method, particularly, `print-stack-trace` and `get-message`. Here is the translation of the previous example:

```
public static void safeComputation() {
    try {
        dangerousComputation();
    } catch (FileNotFoundException e) {
        System.out.print("Where did I left that book?");
        e.printStackTrace();
    } catch (EOFException e0) {
        System.out.print("Your are reading more than you should!");
    } catch (IOException ioE) {
        System.out.print("Something wrong happened while reading");
        System.out.print("The real error was ");
        System.out.print(ioE.getMessage());
    } catch (ArithmeticException e1) {
        System.out.print("Please, revise your math!");
    }
}
```

Another exception handling form present in Linj is `ignore-errors`. Here is an example of its use:

```
(defun really-safe-computation ()
  (ignore-errors
    (dangerous-computation)
    (format t "~%Let's do it again~%")
    (dangerous-computation)))
```

and the corresponding translation:

```
public static int reallySafeComputation() {
    try {
        dangerousComputation();
        System.out.println();
        System.out.println("Let's do it again");
        return dangerousComputation();
    } catch (Throwable e) {
    }
}
```

Finally, due to the limited capabilities of the Java exception handling mechanisms, Linj does not implement the `handler-bind` macro.

1.12.4 Defining Conditions

The definition of new conditions in Linj is done by subclassing `throwable` or one of its subclasses. There is no special syntax to do that so just use the normal `defclass` form.

1.12.5 Creating Conditions

Similarly to the condition definition, the creation of conditions is done using the normal syntax to create instances of classes, namely, using `new` or `make-instance`.

1.12.6 Warnings

Linj implements warnings in the form of messages sent to `*trace-output*`. This is triggered by the function `warn`. Note that warnings are not conditions (much less errors) as they are in Common Lisp.

1.12.7 Checked vs Unchecked Exceptions

Java is an innovative language in is treatment of exceptions as it separates them into two kinds—checked exceptions and unchecked exceptions—and imposes a rule that says that any checked exceptions that may be thrown in a method must either be caught or declared in the method's throws clause. The Java compiler checks all method definitions to make sure this rule is obeyed.

Whether or not an exception is checked is determined by its place in the hierarchy of throwable classes. All subclasses of `throwable` that are also subclasses of `exception` but not of `runtime-exception` are checked exceptions.

There is much discussion regarding the usefulness of checked exceptions and the growing consensus is that the benefit of the compile-time checking of uncaught errors does not payoff the annoyance of forcing the programmer of having to deal with it, either to acknowledge the possibility of the exception (by declaring the exception in the throws clause of a method) or to handle the exception in the method where it is signalled. In this last case, the common solution is to raise another exception, but this time an unchecked one. The result is that a compile-time check is being translated into a run-time check.

In most other languages, including Common Lisp, exceptions are not checked at compile time and the programmer can choose the best place to deal with them.

Linj decided to leave the decision of compile-time vs run-time checking of exceptions to the programmer. If the Linj compiler variable `*infer-method-throws*` is `t`, the compiler will compute all thrown exceptions in the body of the method and automatically declares them in the `throws` clause of the method (but it also warns you about this). If the variable is `nil`, Linj will not do any inference regarding the thrown exceptions and the Java compiler will barf on all unhandled checked exceptions.

The following example shows how the unhandled throws propagate in the call graph. Each function uses some operations of the Java API that throw checked exceptions.

```
(defun read-data ()
  (let ((f (new 'file "data.dat")))
    (let ((data (new 'data-input-stream (new 'file-input-stream f))))
      (unwind-protect
          (princ-to-string (read-int data))
        (close data)))))

(defun make-alias (name/string)
  (in (the java.rmi.naming)
      (let ((remote (lookup name)))
        (bind (concatenate 'string name "-alias")
              remote))))

(defun make-alias-from-file-data ()
  (make-alias (read-data)))
```

Note that the `make-alias-from-file-data` calls both functions so it might throw the exceptions of both. During the compilation of the above program, the Linj compiler emits several warnings regarding the uncaught exceptions and produces the following Java program:

```
import java.rmi.Remote;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.net.MalformedURLException;
import java.rmi.RemoteException;
import java.rmi.AlreadyBoundException;
import java.io.FileInputStream;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;

public class Throws extends Object {
```

```
    public static String readData() throws IOException, FileNotFoundException {
        File f = new File("data.dat");
        DataInputStream data = new DataInputStream(new FileInputStream(f));
        try {
            return "" + data.readInt();
        } finally {
            data.close();
        }
    }

    public static void makeAlias(String name)
      throws AlreadyBoundException, RemoteException, MalformedURLException, NotBoundException {
        Remote remote = Naming.lookup(name);
        Naming.bind(name + "-alias", remote);
    }

    public static void makeAliasFromFileData()
      throws NotBoundException, MalformedURLException, RemoteException, AlreadyBoundException, FileNotFou
        makeAlias(readData());
    }
}
```

The previous example represents an extreme case but, in general, large programs that do not handle the checked exceptions end up in a situation where the methods near the top of the call graph have large `throws` clauses.

1.13 Input/Output

Linj's input/output is, naturally, restricted by the Java model for input/output, particularly, the `java.lang.System` class and is much, much poorer than the Common Lisp model. In Linj, there is no such thing as printer escaping, printer dispatching, printer radix, printer case, pretty printing, etc, etc. There are, however, standard ways to print most objects. With the exception of the `format` function (when its stream argument is `nil`), none of the output function return anything useful.

Linj implements the global variables *standard-output*, *standard-input* and *error-output* (with the same meaning as in Common Lisp) and also implements *trace-output* as a synonym for *error-output*. Note that, in Linj, these are *not* special variables like in Common Lisp: you can assign them but you cannot bind them.

One important feature of Linj is the ability to input and output instances of classes using a notation that resembles the external representation of Common Lisp structures defined with `defstruct`. To use this, each class definition should include the `:make-parse-method` with a true value.This that causes the automatic generation of two methods: one that produces a string representation of an instance and another (class-allocated), that, given an association list containing slot names and slot values, is able to construct an instance from it. These two methods provide the necessary support for *serialization*, that is, the capability to save a graph of objects and to rebuild it latter. The following text is an example of the serialization:

```
#S(image
   :name "Drawing 1"
   :objects (#S(line :start #S(point :x 10 :y 20)
                     :end #S(point :x 15 :y 100))
             #S(circle :center #S(point :x 50 :y 50)
                       :radius 25)))
```

Given the fact that Java already has native support for serialization, it might seem unnecessary to include another form of serialization. However, Java serialization is illegible. Using Linj, not only we gain easier debugging of instances but we can also use human-writable descriptions of instances.

1.13.1 Input

Regarding input operations, Linj is much more restricted than Common Lisp. There is one class name |linj-reader— that operates as a filter over other streams and that implements a `read` function that is capable of reading several Linj data types, including instances of classes that were made externalizable with the `:make-parse-method` option. The function `read-from-string` is also implemented but is restricted to accept only one argument (no options are available) and only reads the first object present in the string.

The next program demonstrates a `read-print` loop:[24]

```
(defun main ()
  (let ((stream (new 'linj-reader (new 'input-stream-reader *standard-input*)))
        (eof (new 'object)))
    (loop (print '>)
      (let ((expr (read stream eof)))
        (when (eq expr eof)
          (return))
        (princ expr)))))
```

Here is a dribble of the program execution:

```
> 1/3
1/3
> 2/6
1/3
> (1 3/9 4.5 ("Hi" there))
(1 1/3 4.5 ("Hi" there))
```

Note that the `read` function can only return members of reference types. In the case of numbers, either a `bignum` or a `big-decimal` is used.

1.13.2 Output

The output operations provided by Linj are the following:

- `princ` Has the same semantics as in Common Lisp.
- `print` Has the same semantics as in Common Lisp with the exception that objects are printed as if by princ (but are preceded by a newline and followed by a space, just like in Common Lisp).

[24] Note the absence of the `eval` step.

- `prin1` As Linj does not support printer escaping, `prin1` is just a synonym for `princ`.
- `pprint` As Linj does not support pretty printing, `pprint` is just like `print` but without the trailing space.
- `terpri` Has the same semantics as in Common Lisp.
- `fresh-line` is just a synonym for `terpri`.
- `write-byte`, `write-char`, `write-string` and `write-line` have the same semantics as in Common Lisp and they also provide valuable information to the type inferencer.
- `format` Has the same semantics as in Common Lisp but with a number of restrictions:
  - The control string argument must be a literal string (so that format can be partially evaluated).
  - Some of the directives are not implemented or are only partially implemented or only work when `format` is being used with a non-`nil` stream argument.

Formatted Output

In spite of the restrictions on the `format` "function", it is still very useful. For example, consider the following form (where `n` is of type `int`):

```
(format t "Here ~[are~;is~:;are~] ~:*~D pupp~:@P." n))
```

Its translation into Java is:

```
System.out.print("Here ");
switch (n) {
case 0:
    System.out.print("are");
    break;
case 1:
    System.out.print("is");
    break;
default:
    System.out.print("are");
    break;
}
System.out.print(" ");
System.out.print(n);
System.out.print(" pupp");
System.out.print((n == 1) ? "y" : "ies");
System.out.print(".");
```

Neat, isn't it? If, on the other hand, the `format` were being used for its value, that is, the form was:

```
(format nil "Here ~[are~;is~:;are~] ~:*~D pupp~:@P." n))
```

the translation into Java would be:

```
"Here " +
((n == 0) ? "are" : ((n == 1) ? "is" : "are")) +
" " +
n +
" pupp" +
((n == 1) ? "y" : "ies") +
".";
```

In this last case where the `format` function is being used for its value, there's a particular situation where Linj can produce better looking code: when the `format` is the last call in a method body. For example, the translation of the following function

```
(defun foo (n/int)
  (format nil "Here ~[are~;is~:;are~] ~:*~D pupp~:@P." n))
```

is

```
public static String foo(int n) {
    StringBuffer buf = new StringBuffer();
    buf.append("Here ");
    switch (n) {
    case 0:
        buf.append("are");
    case 1:
        buf.append("is");
    default:
        buf.append("are");
    }
    buf.append(" ");
    buf.append(n);
    buf.append(" pupp");
    buf.append((n == 1) ? "y" : "ies");
    buf.append(".");
    return buf.toString();
}
```

In fact, the use of `string-buffer` can be explicitly requested by using a `string-buffer` value as the *destination* argument to the `format` call, similarly to the Common Lisp situation where a `string` with a fill pointer is used.

The `format` function can also be used to process *lists* of elements. Here is a sophisticated example (adapted from [1] that implements the English conventions for printing lists:

```
(defun print-list (l/cons)
  (format t "Items: ~:[none~;~:*~{~#[~;~S~;~S and ~S~:;~@{~#[~;and ~]~S~^, ~}~]~}~].~

(defun main ()
 (print-list '())
 (print-list '(foo))
 (print-list '(foo bar))
 (print-list '(foo bar baz))
 (print-list '(foo bar baz quux)))
```

The above program is translated into the following Java program.

```
import linj.Symbol;
import linj.Cons;

public class Format extends Object {

    public static void printList(Cons l) {
        System.out.print("Items: ");
        if (! l.endp()) {
            Cons origArgs = l;
            Cons args = origArgs;
            while (! args.endp()) {
                switch (args.length()) {
                case 0:

                    break;
                case 1:
                    if (args.endp()) {
                        throw new Error("No more arguments.");
                    } else {
                        Object arg = args.first();
                        args = args.rest();
                        System.out.print(arg);
                    }
                    break;
                case 2:
                    if (args.endp()) {
                        throw new Error("No more arguments.");
                    } else {
                        Object arg = args.first();
                        args = args.rest();
                        System.out.print(arg);
                    }
                    System.out.print(" and ");
                    if (args.endp()) {
                        throw new Error("No more arguments.");
                    } else {
                        Object arg = args.first();
                        args = args.rest();
                        System.out.print(arg);
                    }
                    break;
                default:
                    while (! args.endp()) {
                        switch (args.length()) {
                        case 0:

                            break;
                        case 1:
                            System.out.print("and ");
                            break;
                        }
                        if (args.endp()) {
                            throw new Error("No more arguments.");
                        } else {
                            Object arg = args.first();
                            args = args.rest();
                            System.out.print(arg);
                        }
                        if (args.endp()) {
                            break;
                        }
                        System.out.print(", ");
                    }
```

```
                }
            }
            System.out.println(".");
        } else {
            System.out.println("none.");
        }
    }

    public static void main(String[] outsideArgs) {
        printList(Cons.EMPTY_LIST);
        printList(Cons.list(Symbol.intern("foo")));
        printList(Cons.list(Symbol.intern("foo"), Symbol.intern("bar")));
        printList(Cons.list(Symbol.intern("foo"), Symbol.intern("bar"), Symbol.intern("baz")));
        printList(Cons.list(Symbol.intern("foo"), Symbol.intern("bar"), Symbol.intern("baz"), S
    }
}
```

whose execution produces:

```
Items: none.
Items: foo.
Items: foo and bar.
Items: foo, bar, and baz.
Items: foo, bar, baz, and quux.
```

Note that some of the `format` directives can only be used in some contexts. For example, the tilde left brace directive (representing iteration) that was used above cannot be used when the `format` function is being used for its value and the call is not in tail position.

Besides the very powerful capabilities of the `format` function, Linj can also take advantage of the large Java libraries that also include formatting tools. We will see more about this later.

Chapter 2

Using Java Libraries

One of the strongest points of Linj is in the ability to use the Java Libraries. Linj sees Java classes and methods as if they were defined in Linj, thus allowing its use just like any other Linj classes and methods.

We will now present a few examples of Linj programs that use and/or extend Java classes.

2.1 Input/Output

As we said in Section 1.13.1, Linj provides few Linj-specific operations for doing input. However, nothing prevents us of using the huge Java libraries for input (and output). The following Linj program exemplifies such use by providing a "word count" program that counts lines, words, numbers and separators of any text file.

```
(defun main (file-name/string)
  (let ((source
         (new 'stream-tokenizer
              (new 'buffered-reader
                   (new 'file-reader
                        (new 'file file-name)))))
        (lines 1)
        (words 0)
        (numbers 0)
        (separators 0))
    (eol-is-significant source t)
    (loop
     (let ((token-type (next-token source)))
       (in (the stream-tokenizer)
           (case token-type
             (+tt-eof+
              (return))
             (+tt-number+
              (incf numbers))
             (+tt-word+
              (incf words))
```

```
              (+tt-eol+
               (incf lines))
              (t
               (incf separators))))))
    (format t
            "Lines:~A~%Words:~A~%Numbers:~A~%Separators:~A~%"
            lines words numbers separators)))
```

Obviously, the major part of the program functionality is contained is the Java input/output classes used. The translation of the program to Java produces:

```
import java.io.File;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.StreamTokenizer;
import java.io.FileNotFoundException;
import java.io.IOException;

public class Wc extends Object {
    // methods

    public static void main(String[] outsideArgs) throws IOException, FileNotFoundException {
        String fileName = outsideArgs[0];
        StreamTokenizer source = new StreamTokenizer(new BufferedReader(new FileReader(new File
        int lines = 1;
        int words = 0;
        int numbers = 0;
        int separators = 0;
        source.eolIsSignificant(true);
        nil: {
            while (true) {
                int tokenType = source.nextToken();
                switch (tokenType) {
                case StreamTokenizer.TT_EOF:
                    break nil;
                case StreamTokenizer.TT_NUMBER:
                    ++numbers;
                    break;
                case StreamTokenizer.TT_WORD:
                    ++words;
                    break;
                case StreamTokenizer.TT_EOL:
                    ++lines;
                    break;
                default:
                    ++separators;
                    break;
                }
            }
        }
        System.out.print("Lines:");
        System.out.println(lines);
        System.out.print("Words:");
        System.out.println(words);
        System.out.print("Numbers:");
        System.out.println(numbers);
        System.out.print("Separators:");
        System.out.println(separators);
    }
}
```

It is interesting to use the program with its own definition, both in Linj and in Java:

```
$ javac Wc.java
$ java Wc wc.linj
Lines:29
Words:46
Numbers:4
Separators:73
$ java Wc Wc.java
Lines:50
Words:93
Numbers:5
Separators:109
```

2.2 AWT

The first versions of Java come with a graphical library called AWT. Although very incipient, the library was powerful enough to help in the creation of sophisticated user interfaces.

To demonstrate the Linj ability to use the AWT, we will develop a very simple application to convert between the Centigrade and Fahrenheit scale of temperatures. The idea is to provide a window with two text fields, one for centigrades and the other for Fahrenheit. Whenever we write a temperature in one of the text fields and press the enter key, the other text field will be updated with the converted temperature.

First, we need formulas to convert from centigrade to Fahrenheit and vice-versa:

```
(defun cent-to-fahr (c/long)
  (+ (/ (* c 9) 5) 32))

(defun fahr-to-cent (f/long)
  (/ (* (- f 32) 5) 9))
```

Next, we need to read and write numbers from text fields. In Java, a `TextField` is a subclass of a `TextComponent` so we can be a bit less specific here and define more generic procedures:

```
(defun get-value (inst/text-component)
  (parse-integer (get-text inst)))

(defun set-value (inst/text-component value/long)
  (set-text inst (princ-to-string value)))
```

Now, we need to create the two text fields and connect them to each other so that one action in one of them causes the other to update its value. The AWT solution for doing this is to use `ActionListeners`, writing something of the form:

```
(defun main ()
  (let ((cent (new 'text-field 10))
        (fahr (new 'text-field 10)))
    (add-action-listener
     cent
     (new (class action-listener
                 (defmethod action-performed (event/action-event)
                   (ignore-errors
                     (set-value fahr (cent-to-fahr (get-value cent))))))))
    ...))
```

The previous code fragment connects the centigrade text field to the Fahrenheit text field so that an action on the first causes the second to compute its value using the `cent-to-fahr` function. However, as is possible to see, the code is too verbose to be acceptable by any seasoned Common Lisp programmer. Besides, we will have to write a very similar fragment to connect the text fields in the other direction. Common Lisp macros, as usual, provide a nice solution to this problem. We will hide the details of the connecting code in a macro call. Let's define the macro `on-action` to do this:

```
(defmacro on-action ((expr) &body body)
  `(add-action-listener
    ,expr
    (new (class action-listener
                (defmethod action-performed (event/action-event)
                  (ignore-errors
                    ,@body))))))
```

We can now write instead:

```
(defun main ()
  (let ((cent (new 'text-field 10))
        (fahr (new 'text-field 10)))
    (on-action (cent)
      (set-value fahr (cent-to-fahr (get-value cent))))
    (on-action (fahr)
      (set-value cent (fahr-to-cent (get-value fahr))))
    ...))
```

Now, we need to create a `Frame` to contain the text fields and also a few labels to make the application more user-friendly. Unfortunately, this simple descriptions doesn't quite match what is needed in terms or Java code: it is necessary first to create and install an appropriate layout manager and then we need to add the objects one by one, creating instances of `Labels` wherever necessary and, finally, we need to ask the components to be laid out at their preferred size, according to the layout manager chosen. To hide all this complexity we will, once again, define a macro:

```
(defmacro add-in-flow (inst &rest comps)
  `(progn
     (set-layout ,inst (new 'flow-layout))
```

```
     ,@(mapcar #'(lambda (comp)
                   `(add ,inst
                         ,(if (stringp comp)
                              `(new 'label ,comp)
                              comp)))
               comps)))
```

This macro allows us to write the rest of the application as follows:

```
(defun main ()
  (let ((cent (new 'text-field 10))
        (fahr (new 'text-field 10)))
    (on-action (cent)
      (set-value fahr (cent-to-fahr (get-value cent))))
    (on-action (fahr)
      (set-value cent (fahr-to-cent (get-value fahr))))
    (let ((frame (new 'frame "Centigrade Fahrenheit Converter")))
      (add-in-flow frame "Centigrade" cent " - " fahr "Fahrenheit")
      (pack frame)
      (show frame))))
```

If we write the above fragments in a Linj file named `converter.linj` and translate it into Java, we get:

```
import java.awt.Label;
import java.awt.FlowLayout;
import java.awt.Frame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.TextField;
import java.awt.TextComponent;

public class Converter extends Object {

    // methods

    public static long centToFahr(long c) {
        return ((c * 9) / 5) + 32;
    }

    public static long fahrToCent(long f) {
        return ((f - 32) * 5) / 9;
    }

    public static long getValue(TextComponent inst) {
        return Long.parseLong(inst.getText());
    }

    public static void setValue(TextComponent inst, long value) {
        inst.setText("" + value);
    }

    public static void main(String[] outsideArgs) {
        final TextField cent = new TextField(10);
        final TextField fahr = new TextField(10);
        cent.
         addActionListener(new ActionListener() {
                               public void actionPerformed(ActionEvent event) {
```

```
                                   try {
                                       setValue(fahr, centToFahr(getValue(cent)));
                                   } catch (Throwable e) {
                                   }
                               }});
        fahr.
         addActionListener(new ActionListener() {
                               public void actionPerformed(ActionEvent event) {
                                   try {
                                       setValue(cent, fahrToCent(getValue(fahr)));
                                   } catch (Throwable e) {
                                   }
                               }});
        Frame frame = new Frame("Centigrade Fahrenheit Converter");
        frame.setLayout(new FlowLayout());
        frame.add(new Label("Centigrade"));
        frame.add(cent);
        frame.add(new Label(" - "));
        frame.add(fahr);
        frame.add(new Label("Fahrenheit"));
        frame.pack();
        frame.show();
    }
}
```

Compiling and running the above code produces the following application shown in Figure 2.1

Figure 2.1: A Java application that converts between the Centigrade and Fahrenheit scales of temperature.

2.3 Swing

Swing is a much more powerful graphical toolkit that extends AWT. We will not explain the mechanics of Swing. Instead, we will show how an example that demonstrates how Linj can easily access to the huge Swing libraries. We will take advantage of the macros developed in the previous section.

Our Linj/Swing example is a simple web browser. It contains a text field where the user can write an URL, a large pane where the corresponding HTML page is shown and a “back” button that allows us to return to the previous page. Whenever the user clicks on an HTML link the browser shows the corresponding page. Without further explanation, here is the solution:

```
(import javax.swing.*)
(import javax.swing.event.*)

(defun main ()
  (let ((url-text (new 'j-text-field 30))
        (back-button (make-instance 'j-button "Back"))
        (html-pane (new 'j-editor-pane))
        (url-stack (new 'stack))
        (top-panel (new 'j-panel)))
    (on-action (url-text)
```

```
      (push url-stack (get-page html-pane))
      (set-page html-pane (get-text url-text)))
    (on-action (back-button)
      (unless (empty url-stack)
        (set-page html-pane (pop url-stack))))
    (add-in-flow top-panel back-button "URL:" url-text)
    (set-editable html-pane nil)
    (add-hyperlink-listener
     html-pane
     (new (class hyperlink-listener
                 (defmethod hyperlink-update (e/hyperlink-event)
                   (when (= (get-event-type e)
                            (slot-value (the hyperlink-event/event-type) '+activated+))
                     (push url-stack (get-page html-pane))
                     (ignore-errors (set-page html-pane (getURL e))))))))
    (let ((frame (new 'j-frame "Browser")))
      (let ((content-pane (get-content-pane frame)))
        (add content-pane top-panel (slot-value (the border-layout) '+north+))
        (let ((scroll-pane (new 'j-scroll-pane html-pane)))
          (set-preferred-size scroll-pane (new 'dimension 800 600))
          (add content-pane scroll-pane))
        (pack frame)
        (show frame)))))
```

Note that almost no type declarations were needed. Here is the generated Java code:

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.Dimension;
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.Label;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Stack;
import java.net.URL;

public class Browser extends Object {

    public static void main(String[] outsideArgs) {
        final JTextField urlText = new JTextField(30);
        JButton backButton = new JButton("Back");
        final JEditorPane htmlPane = new JEditorPane();
        final Stack urlStack = new Stack();
        JPanel topPanel = new JPanel();
        urlText.
         addActionListener(new ActionListener() {
                               public void actionPerformed(ActionEvent event) {
                                   try {
                                       urlStack.push(htmlPane.getPage());
                                       htmlPane.setPage(urlText.getText());
                                   } catch (Throwable e) {
                                   }
                               }});
        backButton.
```

```
         addActionListener(new ActionListener() {
                               public void actionPerformed(ActionEvent event) {
                                   try {
                                       if (! urlStack.empty()) {
                                           htmlPane.setPage((URL)urlStack.pop());
                                       }
                                   } catch (Throwable e) {
                                   }
                               }});
        topPanel.setLayout(new FlowLayout());
        topPanel.add(backButton);
        topPanel.add(new Label("URL:"));
        topPanel.add(urlText);
        htmlPane.setEditable(false);
        htmlPane.
         addHyperlinkListener(new HyperlinkListener() {
                                  public void hyperlinkUpdate(HyperlinkEvent e) {
                                      if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATE
                                          urlStack.push(htmlPane.getPage());
                                          try {
                                              htmlPane.setPage(e.getURL());
                                          } catch (Throwable e0) {
                                          }
                                      }
                                  }});
        JFrame frame = new JFrame("Browser");
        Container contentPane = frame.getContentPane();
        contentPane.add(topPanel, BorderLayout.NORTH);
        JScrollPane scrollPane = new JScrollPane(htmlPane);
        scrollPane.setPreferredSize(new Dimension(800, 600));
        contentPane.add(scrollPane);
        frame.pack();
        frame.show();
    }
}
```

The Figure 2.2 shows the browser in action.

Figure 2.2: The browser in action.

Bibliography

[1] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, second edition edition, 89.

Index