The J- Language

Version 1.8 (29/3/2007)

Joel Fenwick and Rodney Topor

We define here a simple object-oriented language called **J-** (pronounced ”J minus”)[1]. J- is a bit like Java, but is more purely object-oriented. It is intended to be simple to learn, use and implement, but complex enough to present some challenges to the implementor.

A J- program consists of one or more classes. Each class may contain field declarations, constructor declarations and method declarations. All class members are instance (nonstatic) members. Methods declarations may contain variable declarations and statements. The statement types are expressions, return-statements, method calls, compound-statements, if-statements and while-statements. There are no primitive types. The predefined classes are `Null`, `Boolean`, `Integer`, `Double`, `String`, `List`, `Object` and `System`. `Null` is a subclass of every class. `Integer` is a subclass of `Double`. Every class is a subclass of `Object`. There is a unique instance `null` of class `Null` and a unique instance `system` of class `System`. Values of any type `B` may be cast to any subclass `A` of `B`, but this cast may fail with a runtime error if the value is not of type `A` (as in Java). The language is (reasonably) strongly typed.

To run a J- program you must specify a class and a void method with no arguments of that class. The runtime system creates an instance of that class and applies the method to that instance.

Lexical conventions of J-

The reserved words of the language are the following:

```
class else elseif extends if new null return this void while
```

Special symbols are the following:

```
+ - * / % =  == != < <= > >= . , ; [ ] ( ) { }
```

Other tokens are identifiers (`ID`) , Boolean constants (`BOOLEAN`), integer constants (`INTEGER`), double precision constants (`DOUBLE`) and string constants (`STRING`).[2] These are defined by the following regular expressions:

```
ID = letter (letter | digit)*
BOOLEAN = "true" | "false"
INTEGER = digit+
DOUBLE  = digit+ (. digit+ (e (+|-)? digit+)? | e (+|-)? digit+)
STRING  = " printable-character* "
```

Boolean (resp., integer or double precision) constants evaluate to instances of the class `Boolean` (resp., `Integer` or `Double`).

White space consists of spaces, tabs and newlines. White space is ignored, except that it must separate identifiers, numbers, strings and reserved words. Quote characters (¨) in strings must be escaped with the backslash character (\).

A comment is a sequence of characters starting with the symbol `//` and continuing until the end of the line. Comments may occur anywhere white space can occur, *i.e.*, comments cannot start within tokens, including strings.

[1]”J” used to stand for Java, but now stands for Joel.

[2]The language should also support list constants but currently doesn’t.

Predefined identifiers of J-

The class `Object` has a method `equals()`. All methods here and throughout are instance methods.
The class `System` has input methods `readBoolean()`, `readInteger()`, `readDouble()` and `readString()`, and output methods `print()` and `println()`. There is a unique instance `system` of class `System`.
The class `String` has methods `compareTo()`, `length()`, `subString()`, `append()`, *etc.*
The class `List` has methods `size()`, `add()`, `remove()`, `get()`, `set()`, *etc.*
The class `Double` has methods `round()` and `truncate()`.
There is a unique instance `null` of class `Null`.
Methods of a superclass are inherited by its subclasses.

Grammar of J-

```
program --> class-dec {class-dec}
class-dec --> "class" ID [extends ID] "{" {declaration} "}"
declaration --> field-dec | constructor-dec | method-dec
field-dec --> type-specifier ID {"," ID} ";"
type-specifier -->  ID
constructor-dec --> ID "(" parameter-part ")" compound-stmt
method-dec -->
    (type-specifier | "void")  ID "(" parameter-part ")"
    compound-stmt
parameter-part --> parameter-dec {"," parameter-dec} | empty
parameter-dec --> type-specifier ID
compound-stmt --> "{" {variable-dec} {statement} "}"
variable-dec --> type-specifier ID {"," ID} ";"
statement -->
    empty-stmt |
    expression-stmt |
    return-stmt |
    compound-stmt |
    conditional-stmt |
    iteration-stmt
empty-stmt --> empty ";"
expression-stmt --> expression ";"
return-stmt --> "return" [expression] ";"
conditional-stmt -->
    "if" "(" expression ")" compound-stmt
    {"elseif" "(" expression ")" compound-stmt}
    ["else" compound-stmt]
iteration-stmt --> "while" "(" expression ")" compound-stmt
expression --> variable "=" expression | simple-expression
simple-expression -->
```

```
    additive-expression relop additive-expression |
    additive-expression
relop --> "==" | "!=" | "<" | "<=" | ">" | ">="
additive-expression --> additive-expression addop term | term
addop --> "+" | "-"
term --> term mulop factor | factor
mulop --> "*" | "/" | "%"
factor -->
    constant |
    variable |
    constructor-call |
    method-call |
    "(" expression ")" |
    "[" type-specifier "]" factor
constant --> "null" | BOOLEAN | INTEGER | DOUBLE | STRING | "this"
variable --> ID | object "." ID
object -->
    "this" |
    variable |
    constructor-call |
    method-call |
    "(" expression ")"
constructor-call --> "new" ID "(" argument-part ")"
method-call -->
    ID "(" argument-part ")" |
    object "." ID "(" argument-part ")"
argument-part --> argument {"," argument} | empty
argument --> expression
empty -->
```

Note In this grammar, "|" separates alternatives, parentheses group elements, brackets enclose optional elements, and braces enclose elements repeated zero or more times. Specific (terminal) tokens are enclosed in quotes. The nonterminal `empty` expands to the empty sequence.

Syntactic restrictions and semantics of J-

Most of these can be inferred from your knowledge of Java and from the description of C-Minus (C-) in Louden's text.

For example, each constructor name must equal the enclosing class name. Each method declaration with a non-void return type must return a value of that type. The number and type of arguments in each method (resp., constructor) call must correspond to the number and type of parameters in the corresponding method (resp., constructor) declaration. Numerical operators (`+ - * / %`) can only be applied to numerical values. Comparison operators (`< <= > >=`) can only be applied to numerical values. Equality operators (`== !=`) can be applied to any values.

Details of the predefined identifiers, other syntactic restrictions, and language semantics will be provided later, as required.

Example J- programs

See files `factorise.jm`, `queue.jm`, `dictionary.jm` available elsewhere.