A Model of J in OMeta2/Squeak

Yoshiki Ohshima, Ted Kaehler

VPRI Research Memo M-2014-001

Viewpoints Research Institute, 1025 Westwood Blvd 2nd flr, Los Angeles, CA 90024 t: (310) 208-0524

A Model of J in OMeta2/Squeak

Yoshiki Ohshima Ted Kaehler
Communications Design Group
SAP Labs and Viewpoints Research Institute
yoshiki.ohshima@vpri.org, ted@vpri.org

August 6, 2014

1. Introduction

The APL language is flabbargasting for some learners. The language is extremely terse, yet program written in it does a lot. We can summarize the contributing factors to the design:

- have very powerful mathematical operators that manipulate multi-dimensional arrays.
- have higher-order operators that modify other operators.
- have syntax that facilitates highly context-dependent interpretation of symbols and names to minimize syntactical noise.

We decided to build our own implementation of an APL family language to have a better understanding of the relationship between mathematics and programming languages. Because the language itself stays at a very high level, it allows the implementors to have vastly different implementation strategies. In fact various optimization techniques were proposed (such as "drag-along and beating" in [1], or a compiler [2]) and there exist a few industrial-strength implementations [3–5]. Our aim here, however, is to have a minimum model of the language implemented so that we can understand and play with it.

We decided to use OMeta2/Squeak [6] as the implementation language. We also decided to follow the syntax of J, a variant of APL that only uses ASCII characters (and slightly different semantics).

This memo gives an overview of the implementation. The implementation consists of the parser, the interpreter, and the set of primitives. In following sections, each of these is explained.

2. The Parser

The parser is implemented as a subclass of OMeta2 and named JParser. JParser reads a textual program in J and produces a tree made of lists.

A unit of execution in J is called a sentence, which is simply a list of "elements". In OMeta2, this could be described as a production rule as follows:

```
sentence = element+
```

To produces results from the rule, we add a semantic action:

```
sentence =
  element+:s spaces -> [{#sentence}, s reversed]
```

With this semantic action, all elements recognized are reversed, and a symbol `#sentence` is attached as the head of the list.

How does the element rule look? It is simply:

```
element =
  number number+
| number
| primitive
| name
| "(" sentence ")"
```

Namely, an element is either two or more numbers in a row, a single number, a primitive (a symbol), a variable name, or an inner sentence that is enclosed by a pair of parentheses. Again, we would like to produce some results, and not just recognized, we add semantic actions:

```
element =
  number:f number+:s      -> [{#array. f}, s]
| number
| primitive:p             -> [{#primitive. p}]
| name:n                  -> [{#name. n}]
| "(" sentence:s ")"      -> [s]
```

As you can see, for each case (except the single number case), the result is a list with a head symbol that denote the type of the result.

This is pretty much all it has. In the actual implementation, we define a few other rules like `primitive`, `name`, and `number`, but they are all trivial.

We can now parse a J sentence. For example, when we give it a string something like:

```
(2 3 $ 1 2 3) +/ 4 4 4
```

it produces:

```
{#sentence.
  {#array. {#number. 4}. {#number. 4}. {#number. 4}}.
  {#primitive. #/}.
  {#primitive. #+}.
  {#sentence.
     {#array. {#number. 1}. {#number. 2}. {#number. 3}}.
     {#primitive. #'$'}.
     {#array. {#number. 2}. {#number. 3}}}}
```

Recall that the elements in a sentence are reversed. The first element in the tree is “4 4 4”, and then / and then + and then the inner sentence.

3. The Interpreter

We write the interpreter again in OMeta2. The interpreter is called JInterpreter, and it is a subclass of `OMeta2`.

A model of the J interpreter is described on http://www.jsoftware.com/help/dictionary/dicte.htm. The essence of evaluation is the following part:

> *Parsing proceeds by moving successive elements [...] from the tail end of a queue [...] to the top of a stack [...], and eventually executing some eligible portion of the stack and replacing it by the result of the execution.*

and the diagram below it:

```
  b =: + / 2 * a
  b =: + / 2 *          1 2 3     Move
  b =: + / 2         *  1 2 3     Move
  b =: + /        2  *  1 2 3     Move
  b =: +       /  2  *  1 2 3     Move
  b =: +             /  2 4 6   2 Dyad
  b =:            +  /  2 4 6     Move
  b           =:  +  /  2 4 6     Move
  b             =:  +/  2 4 6   3 Adverb
  b                    =:  12   0 Monad
                    b  =:  12     Move
                           12   7 Is
                           12
```

We can just transliterate this model into an OMeta2 rule for JInterpreter. We name the rule `sentence`:

```
sentence =
  {#sentence (reduce(stack) | shift)+} reduce(stack)*
     -> [stack last]
```

The main idea is that the `sentence` rule handles a list that begins with the `#sentence` symbol, and the interpreter repeatedly tries to “reduce” the contents on stack (which is a state of the interpreter). When there is nothing to reduce on the stack, then it tries to “shift” an element from the input. After processing all input, there may be elements on the stack, so it applies the reduce operation as many times as it has to and then the stack top becomes the result.

The skeleton of the `#shift` rule is as follows:

```
shift =
  ( {( #array
      ({#number _:n} [n])+:ns
       [JArray new: {ns size} data: ns offset: 0]
    | #number _:n [n]):n
   } [n]
   | _):n
  [stack addLast: n]
```

When the incoming element is a list that begins with `#array`, it creates a data structure called `JArray`. If it is a `#number`, it extracts the actual value. Otherwise any item (a primitive symbol) would be pushed on to the stack.

However, the actual implementation is slightly more complicated. See the description on the above mentioned web page: “Adverbs and conjunctions are executed before verbs;”. This means that not everything can be simply shifted in the same manner. Instead, when an adverb or a conjunction is being shifted, it has to look ahead to make a compound verb before shifting. Also, if the element being shifted is an inner sentence, it needs to be evaluated. For these modifications another version of `#shift` is:

```
shift =
  (
    parseVerb
   | {
     ( #array
     ({#number _:n} [n])+:ns
       [JArray new: {ns size} data: ns offset: 0]
     | #number _:n [n]):n
    } [n]
    | _):n
  (
   innerSentence(n):n
  | [stack addLast: n])
```

The `parseVerb` rule attempts to match more input when it sees an adverb or a conjunction. Another modification is that when the element being shifted is an inner sentence, it applies the sentence rule recursively (from within the `#innerSentence` rule). The actual implementation has some more details but we omit the explanation for them for brevity.

The rule for reducing stack, called `#reduce` can be written also by translitering the diagram shown on the J page:

```
 EDGE        VERB        NOUN   ANY         0 Monad
 EDGE+AVN    VERB        VERB   NOUN        1 Monad
 EDGE+AVN    NOUN        VERB   NOUN        2 Dyad
 EDGE+AVN    VERB+NOUN   ADV    ANY         3 Adverb
 EDGE+AVN    VERB+NOUN   CONJ   VERB+NOUN   4 Conj
             ...
Legend:
AVN denotes ADV+VERB+NOUN
...
```

into a rule called `reduce`:

```
reduce =
  {(
     arg:n conjunctions:c
       -> [self pop: 4 andPush: c andPush: n]
   | arg:r dyadOp:o ~verb arg:l
       -> [self pop: 3 andPush: (o value: l value: r)]
   | arg:r monadOp:o (end | (~arg ~conjunction _))
       -> [self pop: 2 andPush: (o value: r)]
   ):n _*}
  [n]
```

where `#conjunctions` matches three elements that constitute a compound verb with a conjunction. When any of the cases matches at the “bottom” of stack, it computes a value by sending `#value:` or `#value:value:` to the operation represented as a Squeak closure and replaces the involved elements with the result. (Note that they are not pushed onto the top of stack; rather, replacement happens at the bottom of the stack.)

Again, the actual interpreter has some more details (such as the implementaiton of `dyadOp`, `monadOp`, etc.), but these three rules are the core of the interpreter.

4. Operators

If the language is simple enough that it does not involve multi-dimensional arrays as first class data, making the interpreter and parser could have been the end of the story. Here, however, we need to supply enough features for simple (and complex) mathematical operators.

Let us take dyadic operators (i.e., one that takes two arguments). We would like to have a generic implementation to perform the operation given as a parameter. Since such an implementation needs to handle the rank information for each argument (with the rank operator (") in J), the signature of the generic method written in Squeak to compute the result is:

```
doDyad: op with: l with: r lRank: lr rRank: rr
```

where `op` is either a closure that provides the bottom case (for example, `[:x :y | x + y]`, for addition), or an object of class called `JOp`, which mimics such a closure but has some extra infomation to support compound operations. The arguments `l` and `r` are left and right arguments and `lr` and `rr` are the rank that the operator use to operate on left and right arguments, respectively. The implemtation of `#doDyad:...` is about 30 lines of Smalltalk code to handle the rank information correctly. (Which could be shorter but it is left as an exercise for readers.)

As we need to combine such an operator with another via adverbs and conjunctions, we naturally represent them as closures. For example, the + operator is stored in a dictionary called `Dyad` in the following manner:

```
Dyad at: #'+' put: [:a :b :ar :br |
  self doDyad: [:x :y | x + y]
    with: a with: b lRank: ar rRank: br].
```

(Note that `self` here is the interpreter and not a `JArray`. An operator does not belong to an array.)

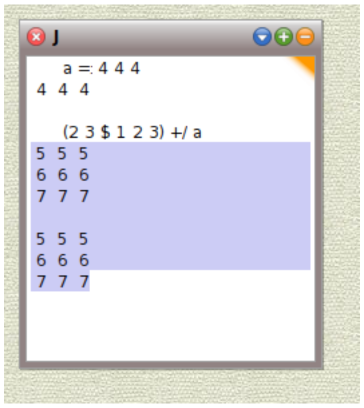
When an adverb or a conjunction is combined with another opeartors, it creates a compound operator. In other words, an adverb or a conjunction is a higher order function that takes a function and create another. Naturally, you can represent these operators as closures. For example, the adverb `&` for the monadic use is defined and stored into a dictionary named `DConj` as:

```
DConj at: #'&' put: [:left :right |
  [:l :r :lr :rr |
    left
      value: (right value: l)
      value: (right value: r)]].
```

Namely, it takes two opeartors (`left` and `right`, and returns another closure.

5. J Workspace

Because the implementation is on top of Squeak, one cannot finish without making a Workspace where you can evaluate J expressions.



```
   a =: 4 4 4
4  4  4

   (2 3 $ 1 2 3) +/ a
5  5  5
6  6  6
7  7  7

5  5  5
6  6  6
7  7  7
```

6. Conclusion

This memo outlines the implementation of the J-like language in OMeta2/Squeak. Implementing the parser in OMeta was definitely a big plus; it gives us concise description. The experience in writing this interpreter in OMeta2 was mixed. It certainly was good to be able to write a parser with pattern matching, but the interpreter requires operating on the both ends of a list, while OMeta2 has good support only to do pattern matching at the beginning of the input stream. For this reason, we flip the order of elements in the parsed result, and have the "stack" data structre that is still manipulated at both ends.

References

[1] Philip Samuel Abrams. *An Apl Machine*. PhD thesis, Stanford University, Stanford, CA, USA, 1970. AAI7022146.

[2] Timothy Budd. *An APL Compiler*. Springer, 1988.

[3] Dyalog. `http://dyalog.com/`.

[4] The j programming language. `http://jsoftware.com/`.

[5] The k programming language. `http://kparc.com/`.

[6] Alessandro Warth and Ian Piumarta. OMeta: an object-oriented language for pattern matching. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 11–19, New York, NY, USA, 2007. ACM.