Algorithm for translation from string representation to the Manchester syntax with variables

In the formalization of the algorithm we use the following pseudo-code convention, functions and types, inspired by the most popular programming languages:

- A method or a field is referred using dot, e.g. field *type* of an object n is referred as $n.type$.
- A vector v (i.e., an array with a variable size) is equipped with a method $v.add(item)$, to add *item* to the end of the vector. $|v|$ denotes number of elements in v and $v[i]$ denotes item at the position i (we assume that the first element is indexed with 1, not with 0). An empty vector is denoted by $\emptyset$.
- A stack is equipped with the following methods:

  pop() to retrieve and remove the topmost element;

  peek() to retrieve the topmost element without removing it;

  push(*item*) to add *item* on top of the stack.

- *Node* is a structure to represent a node of a tree, as described in Section 4.2 of the paper. The structure contains three fields

  type Contains one of the 10 node types listed in Section 4.2 of the paper, these types are typedsetted all caps in the pseudocode;

  l*abel* Contains textual label, interpreted according to the type of the node, e.g. URI or a lexical form a literal;

  children A vector of children of the node.

- A constant string of characters is denoted by quotes and + is used to concatenate the string.

The algorithm consists of two main parts:

1. Translation from a string of numerical labels into a tree structure in memory, presented in Algorithm 1.
2. Translation of a tree into the corresponding Manchester syntax expression with variables, obtained by calling the function presented in Algorithm 4.

```
Data: input is a string representing a tree
Result: At the end of execution, node is the root node of the tree
dict ← the label–number mappings created during the encoding
path ← empty stack
foreach item ∈ input
    if item == $ then node ← path.pop()
    else
        node ← new node
        (node.type, node.name) ← dict[item]
        path.peek().children.add(node)
        path.push(node)
    end
end
assert path is empty
```

Algorithm 1: An algorithm to transform a string of numerical labels into a tree structure.

```
function ComplementOf(node)
    assert |node.children| ≤ 1
    if node.children is empty then return "not ?classexpr"
    else return "not ("+ToManchester(node.children[1]) + ")"
end
```

Algorithm 2: A function to convert a tree node corresponding to a negation to its corresponding Manchester syntax expression

```
function AndOr(node, keyword)
    items ← ∅
    forall the child ∈ node.children do
        items.add(ToManchester(child))
    end
    while |items| < 2 do
        items.add("?classexpr")
    end
    expr ← items[1]
    for i ← 2, . . . , |items| do
        expr ← expr+ " " +keyword+ " " +items[i]
    end
    return expr
end
```

Algorithm 3: A function to convert a tree node corresponding to an intersection or union to its corresponding Manchester syntax expression

```
function ToManchester(node)
    if node.type ∈ {CLASS, INDIVIDUAL, LHS, LITERAL, DATATYPE} then
        return "<" +node.name+ ">"
    else
        assert node.type == INTERNAL
        if node.label=="someValuesFrom" then
            return ClassRestriction(node, "some")
        else if node.label=="allValuesFrom" then
            return ClassRestriction(node, "all")
        else if node.label=="value" then
            return ValueRestriction(node)
        else if node.label =="intersectionOf" then
            AndOr(node, "and")
        else if node.label =="unionOf" then
            AndOr(node, "or")
        else if node.label == "oneOf" then
            return OneOf(node)
        else if node.label == "complementOf" then
            return ComplementOf(node)
        else if node.label == "minCardinality" then
            return Cardinality(node, "min")
        else if node.label == "cardinality" then
            return Cardinality(node, "exactly")
        else if node.label == "maxCardinality" then
            return Cardinality(node, "max")
        else if node.label == "minQualifiedCardinality" then
            return QualifiedCardinality(node, "min")
        else if node.label == "qualifiedCardinality" then
            return QualifiedCardinality(node, "exactly")
        else if node.label == "maxQualifiedCardinality" then
            return QualifiedCardinality(node, "max")
        else if node.label == "DatatypeRestriction" then
            return DatatypeRestriction(node)
    end
end
```

Algorithm 4: A function to convert a tree node to its corresponding Manchester syntax expression

```
function OneOf(node)
    items ← ∅
    forall the child ∈ node.children do
        items.add(ToManchester(child)
    end
    while |items| < 2 do
        items.add("?individual")
    end
    expr ← items[1]
    for i ← 2, ..., |items| do
        expr ← expr + ", " + items[i]
    end
    return "{"+expr+"}"
end
```

Algorithm 5: A function to convert a tree node corresponding to an enumeration to its corresponding Manchester syntax expression

```
function ClassRestriction(node, keyword)
    assert |node.children| ≤ 2
    type ← OBJECT
    p ← NULL
    c ← NULL
    forall the child ∈ node.children do
        if child.type ∈ {DATATYPE_PROPERTY, DATATYPE } then
            type ← DATATYPE
        if child.type ∈ {DATATYPE_PROPERTY, OBJECT_PROPERTY} then
            p ← child.label
        else c ← ToManchester(child)
    end
    if type == DATATYPE then
        if p == NULL then p ← "?dp"
        if c == NULL then p ← "?datatype"
    else
        if p == NULL then p ← "?op"
        if c == NULL then p ← "?classexpr"
    end
    return p + " " + keyword + " (" + c + ")"
end
```

Algorithm 6: A function to convert a tree node corresponding to a class restriction to its corresponding Manchester syntax expression

```
function ValueRestriction(node)
    assert |node.children| ≤ 2
    type ← OBJECT
    p ← NULL
    v ← NULL
    forall the child ∈ node.children do
        if child.type ∈ {DATATYPE_PROPERTY, LITERAL } then type ← DATATYPE
        if child.type ∈ {DATATYPE_PROPERTY, OBJECT_PROPERTY} then
            p ← child.label
        else v ← ToManchester(child)
    end
    if type == DATATYPE then
        if p == NULL then p ← "?dp"
        if v == NULL then v ← "?literal"
    else
        if p == NULL then p ← "?op"
        if v == NULL then v ← "?individual"
    end
    return p + " value (" + v + ")"
end
```

Algorithm 7: A function to convert a tree node corresponding to a value restriction to its corresponding Manchester syntax expression

```
function Cardinality(node, keyword)
    assert |node.children| ≤ 2
    p ← "?op"
    n ← "?cardinality"
    forall the child ∈ node.children do
        if child.type == CARDINALITY then n ← child.label
        else p ← child.label
    end
    return p + " " + keyword + " " + n
end
```

Algorithm 8: A function to convert a tree node corresponding to a cardinality restriction to its corresponding Manchester syntax expression

```
function QualifiedCardinality(node, keyword)
    assert |node.children| ≤ 3
    type ← OBJECT
    p ← NULL
    c ← NULL
    n ← "?cardinality"
    forall the child ∈ node.children do
        if child.type ∈ {DATATYPE_PROPERTY, DATATYPE } then
            type ← DATATYPE
        if child.type ∈ {DATATYPE_PROPERTY, OBJECT_PROPERTY} then
            p ← child.label
        else if child.type == CARDINALITY then n ← child.label
        else c ← ToManchester(child)
    end
    if type == DATATYPE then
        if p == NULL then p ← "?dp"
        if c == NULL then c ← "?datatype"
    else
        if p == NULL then p ← "?op"
        if c == NULL then c ← "?classexpr"
    end
    return p + " " + keyword + n + " (" + c + ")"
end
```

Algorithm 9: A function to convert a tree node corresponding to a qualified cardinality restriction to its corresponding Manchester syntax expression

```
function DatatypeRestriction(node)
    dt ← "?datatype"
    facet ← "?facet"
    forall the child ∈ node.children do
        if child.type == DATATYPE then
            dt ← ToManchester(child)
        else
            literal ← "?literal"
            if node.children is not empty then
                literal ← ToManchester(node.children[1])
            facet ← node.label + " " + literal
    end
    return dt + "[" + facet + "]"
end
```