RAQUEL Token Class – Data Members

Introduction

The data members fall into the following categories :

1. Token Header fields.
2. Pointers to neighbouring Raquel Tokens in that token's parse tree.
3. Pointers to parameter parse trees that represent expressions.
4. Pointers to parameter parse trees that represent literal relational (or other container) values.
5. Name Lists that represent parameters of Raquel Tokens.

Categories 2, 3 and 4 differentiate between parse trees that represent :

- ***part*** of a RAQUEL ***statement*** that directly relates to the Raquel Token in question (category 2);
- an operator or assignment ***expression*** that is a ***parameter*** of the Raquel Token in question (category 3).
- a ***literal relational*** (or other container[1]) ***value*** that is a ***parameter*** of the Raquel Token in question (category 4).

Since categories 3 and 4 are both expressed as parameter parse trees, it might be considered simpler to merge them into one parse tree category. They are split into 2 categories for the following reasons :

- Operator/assignment expression parameters and literal value parameters are conceptually different kinds of parameter, serve different purposes, and are distinguished in RAQUEL syntax. Differentiating them within the token class reflects the differences, and so is actually simpler.
- RAQUEL permits nested relvalues. Both category 3 and 4 expressions could contain nested relvalues. There is only one method of representing relational/container values in parse trees. It is used regardless of nesting context and parameter context. To ensure that no situation could ever arise where it was impossible to distinguish between the 2 categories in a token's parameter parse trees, it is preferable to separate them rather than combine them.

C++ Implementation

Wherever character strings are required, they are implemented by C++ standard library strings.

Wherever lists are required, they are implemented by C++ standard library vectors (as in the PoC prototype code). This eliminates the need to hold the length of a list, since the ability to return the length of a vector is built into its library code.

[1] Currently only relational values (i.e. relvalues) are supported. Bag and sequence containers will additionally be supported in future, including the ability to express literal values of these kinds.

Building on the use of vectors in the PoC prototype has allowed the Raquel Token class code to be simplified compared to the original logical design of tokens. It removes the need to implement the different kinds of 'Connector' that appear in the token's logical design.

1. Token Header Fields

The logical design is directly implemented physically. All the fields of the Token Header are included. All the flags are single characters, so little data is wasted if a flag is not used in a particular token. The Name/Value field is normally used; note that if it holds a value, it will be the character representation of the value held there.

2. Pointers to Other Raquel Tokens

A Raquel Token may hold pointers to other tokens that are its neighbours in its parse tree. For any token, there are at most 3 possibilities : left operand, right operand, and result. The logical design is not directly physically implemented. Instead of a list of *n* pointers to *n* 'operands', there are always 3 named pointers, to its left and right operands and to its result.

Not all the pointers need be used, and often won't be. An unused pointer has the C++ 'nullptr' literal pointer value[2], which is a very small wastage of memory for an unused pointer.

The following describes how the pointers are used.

The token representing a dyadic operator uses the two operand pointers to link it to its operands. The token of a monadic operator uses one operand pointer, the 'Left' one, to link it to its single operand. (Were a niladic operator to be implemented, no pointer would be used).

The tokens representing assignments always use the result pointer to link them to their result token.

The token representing a pseudo dyadic assignment uses the two operand pointers to link it to them; note that the 'Left' operand pointer links to what existed before the assignment, whereas the result pointer links to an updated version of it created by the assignment. A standard dyadic assignment will only use the 'Right' operand pointer, because its second operand is the result, which is pointed to by the result pointer.

The token representing a pseudo monadic assignment uses one pointer, the 'Left' one, to link it to its single operand. Again the 'Left' operand pointer links to what existed before the assignment, whereas the result pointer links to an updated version of it created by the assignment. A standard monadic assignment uses no operand pointers, because its single operand is in fact the result, and is pointed to by the result pointer.

[2] The keyword 'nullptr' was introduced as part of the C++11 standard. For other reasons the Raquel DBMS code needs to be compiled according to this standard anyway.

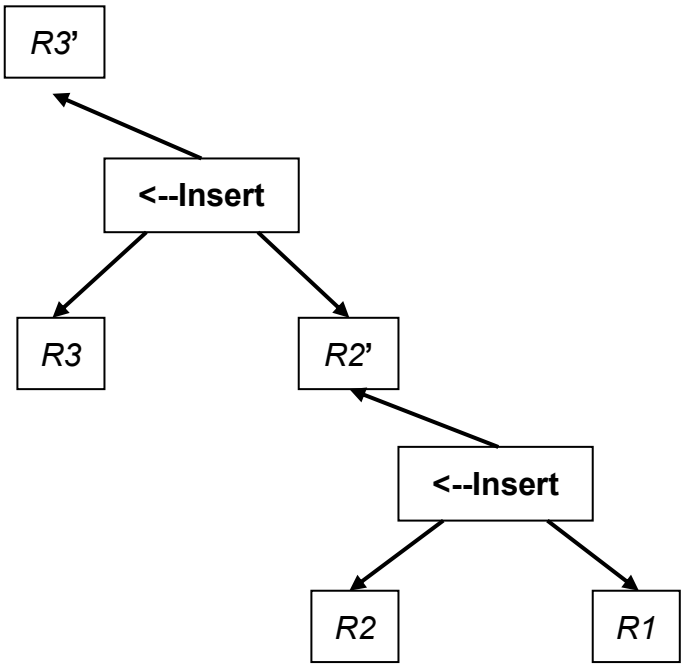
In a conventional parse tree, a variable or value is ***always*** a leaf node to an operator. So a Raquel Token representing it would use none of its pointers.

A RAQUEL parse tree is unconventional, in that a variable or a value is not a leaf node when it is the result of an assignment. In these cases, the token representing the variable/value must have a pointer to the assignment token. The reason is shown via consideration of the following simple example statement :-

R3 **<--Insert** *R2* **<--Insert** *R1*

R1, *R2*, and *R3* are relational variables (= relvars). The statement means that the value of *R1* is inserted into the value of *R2*, whose value – now updated by the insertion of *R1*'s value – is inserted into *R3*.

One would expect the parse tree representing the above statement to be :-



Because *R2* and *R3* are modified, they appear twice in the parse tree, representing their before-assignment and after-assignment states. The decoration ' after a name – as in *R2*' – indicates the named relvar after the assignment.

However a parse tree is identified by a pointer to its root node. It must be possible to follow the pointers down the tree from the root node to any leaf node.

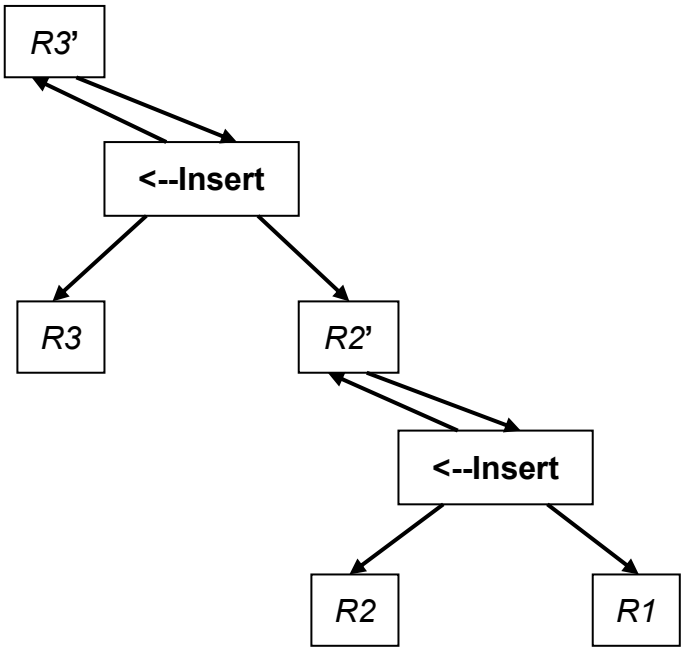
As the example illustrates, a problem arises when an assignment result is also an operand to a subsequent assignment or operator, because it is impossible to follow the pointer down the parse tree from the *R2*' node (in this case) to get to the tree's leaf nodes. The *R2*' node is 'pointed to' from both **<--Insert** assignments, but there is no pointer from *R2*' to the lower **<--Insert**.

Therefore it is necessary for the token representing *R2*' to include a pointer to the assignment that generated it. The token's result pointer is used since *R2*' is the result

of an assignment.

The same applies to *R3'*.

Hence the actual parse tree representing the statement is :-



Note that the root of the parse tree is the *R3'* node, not the upper **<--Insert** node.

It is valid to take an ***operator*** token as the root node of a (sub-) parse tree, because the operator node also represents the value generated by the application of the operator to its operands. But the value generated by the application of an ***assignment*** to its operands is represented by a separate result node. Hence that must be the root node of the (sub-) parse tree representing the application of an assignment.

Since a valid RAQUEL statement is always an assignment expression, an assignment result is always the root node of a parse tree representing a valid RAQUEL statement.

A value or variable token not containing a 'nullptr' result pointer value indicates that it is the result of an assignment, and not a leaf node. It indicates that the value/variable represented by the token did not exist, or did not exist in that form, until that point in the parse tree was executed.

The token of a value or variable leaf node will not contain any pointer values.

One might consider, as an alternative, that the token representing an assignment should contain the reverse pointer value from the result, instead of the result token. However as well as being illogical, it would also be inefficient. All tokens would have to contain a fourth pointer value to accommodate the reverse pointer, yet few tokens would actually use it in practice.

3. Pointers to Parse Trees that Represent Operator/Assignment Expression Parameters

A Raquel Token has a 'Parameter Expression Vector' of pointers to Raquel Tokens. The tokens are the root nodes of parse trees representing parameters which are operator or assignment expressions. Such parameters are used by higher-order operators or assignments.

It corresponds to the 'Assignment Parameters' component in the logical design of a Raquel Token (and the 'm_ParamTokens' vector in the PoC prototype code).

If a Raquel Token has *n* expression parameters, then it has *n* parse trees to represent them. The sequence of pointers to them in the vector corresponds to the sequence of expression parameters as they appear in the invocation of the Raquel Token. The vector is empty if there are no parameter expressions. This is the case for 'Ordinary' (c.f. higher-order) operators and assignments.

The semantics and syntax of an individual operator or assignment determine the number of parameter expressions, and the nature – operator or assignment - of each individual expression.

Note that were such an expression to consist solely of a single literal relational/container value, or a single literal scalar value, then it will still be treated as 'just another parameter expression'. It would not be treated as a literal value parameter.

4. Pointers to Parse Trees that Represent Literal Relvalue/Container Value Parameters

A Raquel Token has a 'Parameter Value Vector' of length 2. Both vector elements contain a pointer to a literal relvalue (or other container value) Raquel Token. The token represents a literal value that is a parameter. A relational operator can have a maximum of 2 such parameters, a left hand and a right hand, hence the vector size. (Currently only RAQUEL operators, not assignments, have this kind of parameter).

The first pointer points to the left-hand parameter and the second to the right-hand parameter. If the invocation of an operator does not include such a parameter, then either pointer, or both, would have the C++ 'nullptr' literal pointer value.

The vector corresponds to the 'Container Values' component in the logical design of a Raquel Token (and the 'm_LiteralTokens' vector in the PoC prototype code).

The literal relvalue Raquel Token has had the physical design of its subtoken revised from that implied by the logical design. The revised design uses the token's Parameter Expression Vector of pointers to reference ***literal tuple*** value Raquel Tokens. Previously literal tuple value tokens did not exist, because tuples in isolation cannot exist in the RAQUEL logical model. Literal ***tuple values*** only exist in the RAQUEL language as components of a literal ***relational value*** (or other container value). The invention of literal tuple value tokens is solely for practical simplicity; it

enables relvalues to be implemented as parse trees of tuple Raquel Tokens in the standard way, and not as a special case.[3]

The relvalue Raquel Token's Parameter Expression Vector contains a pointer to each of the literal tuple value tokens that represent tuple values in the literal relational/container value. The pointer sequence in the vector corresponds to that in which the tuples appear in the invocation of the literal relvalue.

The literal ***tuple*** value Raquel Token also uses the Parameter Expression Vector, in a corresponding manner. The pointers in it point to the root nodes of the attribute value assignment expressions that determine the tuple value.[4] (Note that if a nested relational value is assigned to an attribute, then a literal relational value token will appear in that assignment's parse tree. Such recursion can occur to any finite depth).

The syntax of tuples allows a default version in which only attribute values are given. Consequently assignment expressions are replaced by single value expressions (which would have appeared on the right-hand side of the assignments). Hence a pointer in the literal tuple token's vector points to a Raquel Token representing the single value, rather than to a root node of an assignment expression.

Part of the default is that the order in which attribute values are given must correspond to the order in which attribute names and types were given when the relvar was declared. The correspondence is used later to derive attribute value assignment expressions.

No count of the number of tuples or attributes is kept. They can be obtained from the sizes of the vectors if required.

It must be possible for literal scalar values to be assigned to attributes. Such values are represented by literal scalar value Raquel Tokens. The literal scalar value token has also had the physical design of its subtoken component revised, in a way that corresponds to that of literal relvalue tokens. However the logical design's method of coping with values of both primitive and non-primitive types is retained.

If the literal scalar value is expressed in the innate format of a primitive type, then its Parameter Expression Vector is empty. The value is held in its character format in the Name/Value field of the token header.

If the literal scalar value is expressed in the Selector format of a non-primitive type, then the Name/Value field of the token header is an empty string. The token's Parameter Expression Vector is used to specify the value. There is a pointer to the

---

[3] It also paves the way for a simple means of denoting two or more expressions that have to be executed in parallel. The expressions would be the tuple values of a relation's value. Since a relvalue is a ***set*** of tuples, not a sequence, logically the expressions would need to be executed in parallel not in sequence.

[4] Since a tuple value consists of a ***set*** of attribute values, not a sequence of them, tuples also provide a means of denoting expressions to be executed in parallel, the expressions being individual attribute values. It also clarifies the fact that the 'tuple parallelism' of the previous footnote assumes one attribute per tuple.

root node of every expression that forms a component of the Selector function parameter. (Note that the components of such a parameter have a fixed, defined sequence, which is followed by the pointer sequence in the vector).

If a Selector function invokes a Selector function to express a parameter, corresponding recursion will occur in the use of Parameter Expression Vectors and parse trees, down to any finite depth.

5. Name Lists Representing Parameters

The design for Name Lists has been revised. The logical design used 2 'Name Lists', the 'Special' list and the 'Name List Parameters' list. In order to cope with all possible circumstances, the 2 lists could end up being used in complicated ways.

There are 3 aspects to the usage of Name Lists in a Raquel Token. Having a separate name list data member for each aspect simplifies their usage. Consequently the revised design has the following 3 data member lists :

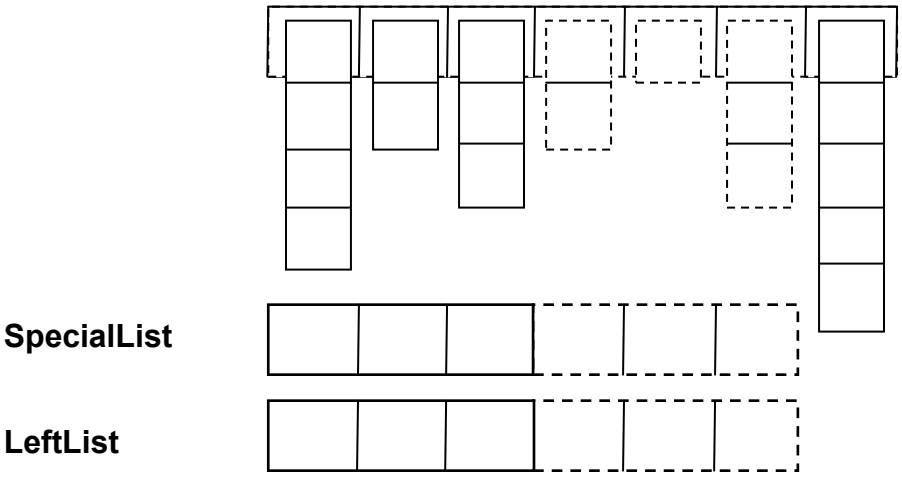
LeftList. Used to hold a list of names from the left-hand side of name list assignments.

NameLists. Used to hold one or more lists of names. A list of names is either standalone or appears on the right hand side of a name list assignment.

SpecialList. Used to hold a list of 'special' characters. They usually prefix a name list in **NameLists**.

While **LeftList** and **SpecialList** are both just one list of names/'special' characters, **NameLists** is actually a list of lists of names.

The following diagram depicts the 3 lists, containing arbitrary numbers of 'names'[5] :-



[5] A 'name' is often that of an attribute, but may be of some other entity. It depends on the context in which the Name Lists are used.

The lengths of the lists vary, depending on the particular purpose to which they are being put, and the number of 'names' that happen to be involved in the particular instance of that purpose. Where two or more lists are used, their lengths typically correlate.

The 3 data member lists are used to hold representations of the 7 different kinds of logical list that can be appear in parameters to RAQUEL operators and assignments.

It is easiest to explain the usage of the data member lists by explaining how they are used to represent each of the logical lists[6].

The most complex logical list is the **Multiple&Tilde List**. The remaining 6 are simpler cases of it. The **Multiple&Tilde List** is described first and then the remaining 6, in order of greater simplicity. It is useful to understand the different simplifications. Their use predominates in practice.

Multiple&Tilde List.
Its purpose is to hold a list of assignments, each of multiple names to a single name, ***plus*** a tilde[7] list of names, i.e. a list of individual names that are each prefixed by '~'.

The assignments are implemented as m lists (where $m > 0$) of n_m names (where $n_m \geq 0$) in **NameLists**, with **SpecialList** being a list of m names, each of one character which is either a space or the special symbol '~', and with **LeftList** being a list of m names.

NameLists also has an additional list – the $(m+1)^{th}$ list – which is the tilde list.[8] Only the names themselves are held in the list, since the tilde can be presumed.

Should there be only a tilde list and no assignments, then there is only that one list in **NameLists**, and **SpecialList** and **LeftList** are both empty.

Should there be no tilde list, then **NameLists** is the same length as **SpecialList** and **LeftList**.

The remaining 6 purposes are now considered, as special cases of the **Multiple&Tilde List** :

Multiple-Assignment List of multiple names to another name.
Implemented as the **Multiple&Tilde List**, but omitting the **Tilde List**.

---

[6] Originally there was a 8th logical list, viz. the name of an operator/assignment that was the single parameter of a higher-order operator/assignment; for example, to hold the name **Union** in the Raquel Token representing **Dist[Union]**. The revised token design is more rational. In the illustrative example, a token representing **Dist** would use the Parameter Expression Vector to hold a single pointer to a **Union** token.

[7] Following mathematical logic, the tilde ('~') is used to represent the negation or opposite of the name(s) that follow it.

[8] The additional list is held in **NameLists** rather than in a separate ***new*** list variable, for 'future-proofing'. Should yet another list be found to be desirable in future, it too can be added to **NameLists**, rather than having to amend the Raquel Token class by adding yet another list variable to it as a data member.

Tilde List of names.

Implemented as the **Multiple&Tilde List**, but omitting the **Multiple-Assignment List**.

Multiple-Name List of names.

Implemented as a **Multiple-Assignment List**, but omitting the list of left-hand names; also the multiple names now constitute one set. So it is implemented as one list of names (possibly empty) in **NameLists**, with **SpecialList** being a list of one name, which is either the space character or the special symbol '~'. **LeftList** is empty.

Single-Assignment List of one attribute name to another attribute name.

It is a different simplification of the **Multiple-Assignment List**. It is implemented as *m* lists (where *m* > 0) in **NameLists**, each list being of a single name. **LeftList** is also a list of *m* corresponding names. **SpecialList** is empty.

Sort-Name List of names.

It is yet another simplification of the **Multiple-Assignment List**. It is implemented as *m* lists (where *m* > 0) in **NameLists**, each list being of a single name. **SpecialList** is also a list of *m* corresponding names, each of one character that is either a space or the special symbol '~'. **LeftList** is empty.

Pattern-List[9] for use in pattern matching.

A special simplification. It is implemented as one list of one 'name' in **SpecialList**, the 'name' being the entire pattern-matching character sequence**.** **NameLists** is an empty list of lists, and **LeftList** is empty.

Where 2 or more of **NameList**, **SpecialList**, and **LeftList** have a list of the same number of items, the i^{th} item in **ListX** relates to the i^{th} item in **ListY**, for all values of *i*.

A name list which is unused in a token is left empty, so wastes very little memory space.

(N.B. A name or single character is implemented by a C++ standard string).

[9] Referred to as a **Special-Sequence List** in the logical token design.