PartiQL Developer's Guide

Ion Team

1 Preface

The developer's guide aims to provide more detailed information about PartiQL's implementation and design.

This document is an early draft, contributions welcome!

1.1 Conventions

TBD

1.2 Further Reading

TBD

1.3 Bug Reports

We welcome you to use the GitHub issue tracker to report bugs or suggest features.

When filing an issue, please check existing open, or recently closed, issues to make sure somebody else hasn't already reported the issue. Please try to include as much information as you can. Details like these are incredibly useful:

- A reproducible test case or series of steps
- The version of our code being used
- Any modifications you've made relevant to the bug
- Anything unusual about your environment or deployment

2 Contributions

See our contribute guide.

3 PartiQL Design Overview

3.1 PartiQL Parser, Compiler, and Evaluator Design

This document provides the high-level design overview of the PartiQL parser and evaluator. The high-level pipeline of compilation is illustrated as follows:

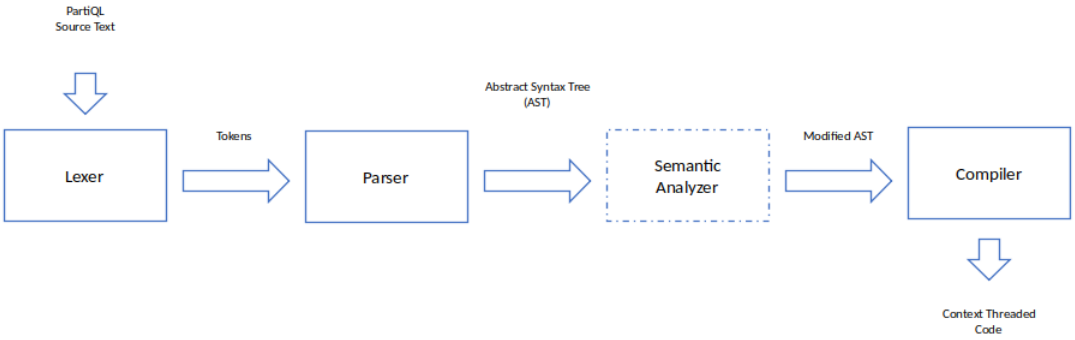

Figure 1: Parser and Compiler Diagram

- The **lexer** is a hybrid direct/table driven lexical analyzer for PartiQL lexemes that produce high-level tokens.
  - SQL is very keyword heavy, so having our own lexer implementation allows us to more easily normalize things like keywords that consist of multiple lexemes (e.g. `CHARACTER VARYING`)
- The **parser** is a basic recursive decent parser for the PartiQL language that produces an AST as an Ion S-expression.
  - For infix operator parsing, the parser is implemented as a Top-Down Operator Precedence (TDOP) Pratt parser.
- The **semantic analyzer** is a placeholder for general purpose semantic analysis. This is not yet implemented, but important optimizations such as determining which paths/columns are relevant for a given query will be done by this phase. Decoupling of the parser from the compiler, means that any application can do their own validation and processing of the AST.
- The **compiler** converts the AST nodes into context threaded code.

3.1.1 Context Threading Example

Context threaded code is used as the interpreter strategy to align the *virtual program counter* with the JVM's *program counter*. This is done by *threading* the operations of the AST nodes into a tree of indirect subroutine calls. Specifically, on the JVM, this is modeled as a series of lambdas bound to a simple functional interface.

We can illustrate this technique with a simple integer evaluator. Consider the following interface that represents an evaluation:

```
interface Operation {
  /** Evaluates the operation against the given variables. */
  int eval(Map<String, Integer> env);
}
```

We can *"hand code"* a simple compilation of `A + B` as follows:

```
public static Operation compilePlus(Operation left, Operation right) {
  return env -> left.eval(env) + right.eval(env);
}

public static Operation compileLoad(String name) {
  return env -> env.get(name);
}

public static void main(String[] args) throws Exception {
  // a "hand" compilation of A + B
  Operation aPlusB = compilePlus(compileLoad("A"), compileLoad("B"));

  // the variables to operate against (i.e. the environment)
  Map<String, Integer> globals = new HashMap<>();
  globals.put("A", 1);
  globals.put("B", 2);

  // evaluate the "compiled" expression
  System.out.println(aPlusB.eval(globals));
}
```

It can be seen that the above example leverages lexical closures (lambdas) to build an object graph of state to represent the actual interpretation, the actual dispatch leverages the native call stack differs from straight compiled code in that each "opcode" is a virtual call.

3.1.2 Evaluation Strategy

Evaluation is done by first compiling source text of an PartiQL expression into an instance of `Expression` which provides the entry point to evaluation:

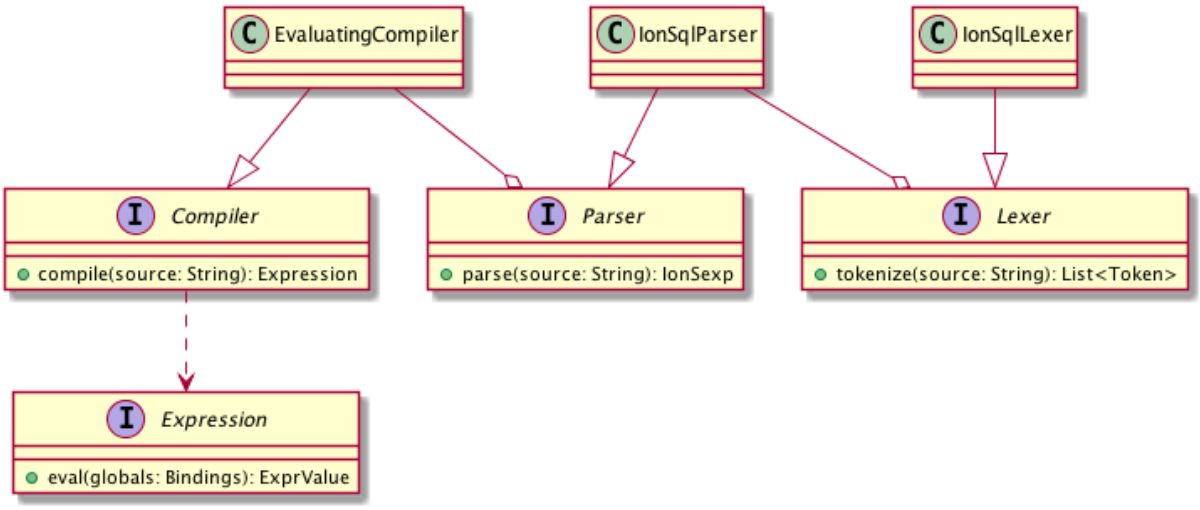

Figure 2: Parser/Compiler/Expression Class Diagram

At the core of all evaluation is an interface, `ExprValue`, that represents all values:

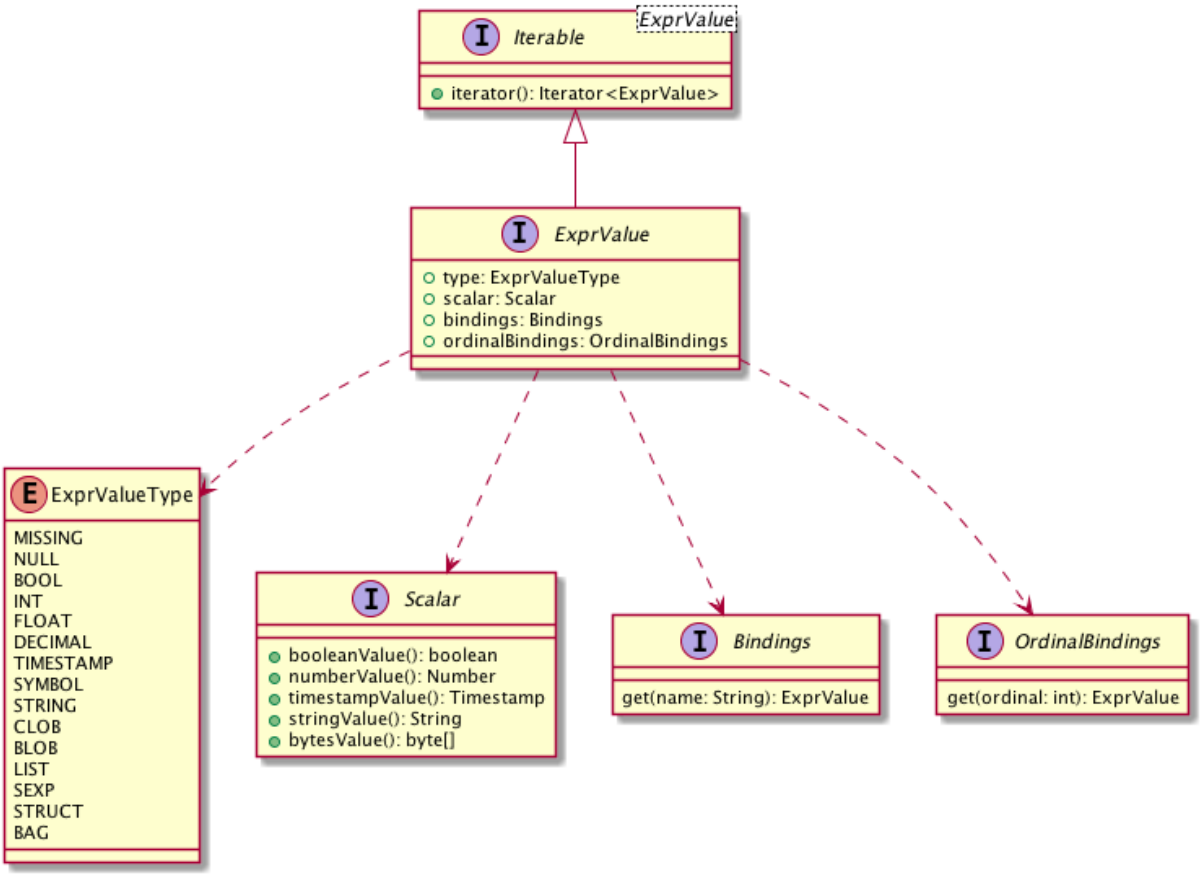

Figure 3: ExprValue Class Diagram

This interface enables any application embedding the evaluator to control and provide data used for evaluation. The other benefit of this approach is that an interface allows for *lazy* evaluation. That is, the evaluator can return an `ExprValue` that has not been fully evaluated. This approach allows for the streaming of values when possible.

All `ExprValue` implementations indicate what type of value they are and implement the parts of the interface appropriate for that type. Relational operations are modeled through the `Iterable`/`Iterator` interface. Accessing scalar data is done through the `Scalar` interface, and accessing fields by name or position are done through the `Bindings` and `OrdinalBindings` interfaces respectively.

Modeling relations (collections) as `Iterable`/`Iterator` allows the evaluator to compose the relational operators (e.g. projection, filter, joins) as lazy `Iterators` that are composed with one another. This functional pipeline is very similar to what is done in full query engines and is also similar to how Java 8 streams work.

Lazy Evaluation Example

We can use our simple integer evaluator example to demonstrate how the PartiQL evaluator lazily evaluates. Let's change this example to add a functional interface as the integer value (i.e. a **thunk** representing an integer).

```
interface IntValue {
    int intValue();
}

interface Operation {
    /** Evaluates the operation against the given variables. */
    IntValue eval(Map<String, IntValue> env);
}
```

Here, we are adding an additional layer of indirection for the result. This allows the evaluator to create values that defer computation. We can then refactor our toy evaluator to be lazy.

```
public static Operation compilePlus(Operation left, Operation right) {
  return env ->
    // we are returning a value that computes the actual addition when intValue() is
        invoked
    () -> left.eval(env).intValue() + right.eval(env).intValue();
}

public static Operation compileLoad(String name) {
  return env -> env.get(name);
}

public static void main(String[] args) throws Exception {
  // a "hand" compilation of A + B
  Operation aPlusB = compilePlus(compileLoad("A"), compileLoad("B"));

  // the variables to operate against (i.e. the environment)
  Map<String, IntValue> globals = new HashMap<>();
  // trivial values
  globals.put("A", () -> 1);
  globals.put("B", () -> 2);

  // evaluate the "compiled" expression to a lazy value
  IntValue result = aPlusB.eval(globals);

  // result doesn't get computed until here
  System.out.println(result.intValue());
}
```


Note that now, all values require an additional virtual dispatch to get the underlying value. `ExprValue` can be thought of as a thunk for all of the types of values in PartiQL.

4 Introduction to the ExprNode AST

It seems the term "AST" is often extended to mean more things than just "Abstract Syntax Tree" and our use of the term is no different. A better term might have been "Abstract Semantic Tree" because our AST was defined with the goal of modeling the intent of the PartiQL source and *not* the exact syntax. Thus, the original SQL from which an AST was constituted cannot be derived, however the *semantics* of that SQL are guaranteed to be preserved. One example that demonstrates this is the fact that we model a `CROSS JOIN` in the same way that we model an `INNER JOIN` with a condition of `TRUE`. Semantically, these have the exact same function and so they also have the same representation in the AST.

4.1 It *is* Actually a Tree

Language implementations often use the term "Abstract Syntax Tree" to refer to a data structure that is actually a graph. Our implementation for PartiQL's AST is a tree and *not* a graph. It contains no cycles and each node can only reference its children.

4.2 It's Immutable

No mechanism has been provided to mutate an instance of the AST after it has been instantiated. Modifications to an existing tree must use cloning re-writes–for instance, a visitor pattern that returns a modified version of the input tree can be utilized.

4.3 Design Patterns Used with the AST

The AST employs a number of patterns and utilizes certain features of Kotlin to aid with the development process. Without any introduction, the reasoning behind these patterns may not be completely apparent at first glance. What follows is an is an attempt to document those patterns and features.

4.3.1 Inheritance Mirrors The PartiQL Grammar

The top-most type of the AST is `org.partiql.lang.ast.ExprNode`. Most of the classes of the AST derive from this class. Most child nodes are also of type `ExprNode`. However, there are several cases where the types of child nodes that are allowed are constrained (or extended) by PartiQL's grammar. For example, not every type of `ExprNode` can exist as components of a path expression (i.e. `a.b.c`). Additionally, some

path components are allowed that do not make sense outside of the context of a path expression (i.e. `a.*.b` and `a[*].b`). If *all* nodes of the AST inherited from `ExprNode` it would be easy to accidentally construct ASTs which are structurally invalid. Thus, each grammar context has a different base class.

This pattern enlists the assistance of the Kotlin compiler to ensure that ASTs are constructed in a manner that is structurally valid. This works so well that for the most part, [ExprNodeCompiler][org.partiql.lang.eval.ExprNodeCompiler] needs to include very few structural checks on the AST. Mostly, it is possible to assume that if the compiler allowed the tree to be instantiated, then it is structurally valid. (However, that does not mean it is semantically valid.)

The base classes are:

- `org.partiql.lang.ast.ExprNode`, for any expression that is self contained and has a value.
- `org.partiql.lang.ast.SelectListItem`, for expressions that may appear between `SELECT` ... `FROM`.
- `org.partiql.lang.ast.FromSource`, for expressions that are data sources in a `FROM` clause.
- `org.partiql.lang.ast.PathComponent`, for the components of a path expression.
- `org.partiql.lang.ast.SelectProjection`, for the type of projection used by a `SELECT`,`SELECT VALUE` or `PIVOT` query. This isn't directly related to the grammar but is convenient to represent in this manner.
- `org.partiql.lang.ast.DataManipulation`, for data manipulation expressions that may optionally be wrapped with `FROM ... WHERE ...`.
- `org.partiql.lang.DmlOperation`, for the data manipulation operation itself (e.g. `INSERT INTO ...`)

All base classes of the AST are sealed classes.

To keep the inheritance hierarchy manageable, the inheritance depth does not exceed 1.

4.3.2 Data Classes

Kotlin data classes have several useful properties which aid the developer when working with the AST. Those features are the compiler generated methods shown below.

- `equals()`, can be used to compare two nodes–for instance, during unit testing.
- `toString()`, highly useful during debugging.
- `componentN()`, enables the use of destructuring (see the section on destructuring below).
- `copy()`, performs a shallow copy of the node, useful during cloning re-writes.

4.3.3 When-As-Expression over Sealed Type Derived Classes

Kotlin's when can be used as a statement or as an expression.

```
        sealed class Foo
```



```
class Bar : Foo(val i: Int)
class Bat : Foo(val n: String)
class Bonk : Foo(val o: Boolean)
val foo = //... an instance of Foo ...

// This a statement because the value is not consumed...
when(foo) -> {
    is Bar -> { println("It's a bar!" }
    is Bat -> { println("It's a bat!" }
    //A compiler warning is issued because no case for Bonk exists.
}

// This is an expression because the value is assigned to variable foo.
val foo = when(bar) {
    is Bat -> "It's a bat!"
    is Baz -> "It's a baz!"
    //A compile-time error is generated because there is no case for Bonk -- when
        branches must be exhaustive.
}
```

When `when` is used as an expression Kotlin requires that the cases are exhaustive, meaning that all possible branches are included or it has an **`else`** clause. Unfortunately, the Kotlin compiler issues a warning instead of an error when the result of the `when` expression is not consumed. We have developed a simple way to gain these compile-time checks for `when` statements as well. This method involves treating them as expressions.

Consider the following:

```
when(expr) {
    is VariableReference -> case {
        ...
    }
    is Literal -> case {
        ...
    }
    // and so on for all types derived from ExprNode
}.toUnit()
```

In order to help make sense of this, the definitions of **`case`** and `toUnit` follow:

```
inline fun case(block: () -> Unit): WhenAsExpressionHelper {
    block()
    return WhenAsExpressionHelper.Instance
```



```
        }


        class WhenAsExpressionHelper private constructor() {
            fun toUnit() {}
            companion object {
                val Instance = WhenAsExpressionHelper()
            }
        }
```

Every branch of the `when` expression calls the **`case()`** function whose first argument is a literal lambda. **`case()`** invokes the lambda and returns the sentinel instance of `WhenAsExpressionHelper`. This forces `when` to have a result. `WhenAsExpressionHelper` then has a single method, `toUnit()`, which does nothing–its purpose however is to consume of result the `when` expression.

When **`case()`** and `toUnit()` are used together in this fashion the Kotlin compiler considers the `when` an expression and will require that a branch exists for all derived types or that an **`else`** branch is present.

This helps improve maintainability of code that uses the AST because when a new type that inherits from `ExprNode` is added then those `when` expressions which do not include an **`else`** branch will generate compiler errors and the developer will know they need to be updated to include the new node type. For this reason, the developer should carefully consider the use of **`else`** branches and instead should consider explicit empty branches for each of the derived types instead.

Also note that the use of **`case()`** and `toUnit()` is *not* needed when the value of the `when` expression is consumed by other means. For example, the compiler will still require a branch for every derived type in this scenario because the result of the `when` becomes the function's return value:

```
    fun transformNode(exprNode: ExprNode): ExprNode = when(exprNode) {
        is Literal -> { // case() is not needed
            //Perform cloning transform
            ...
        }
        is VariableReference -> { // case() is not needed
            //Perform cloning transform
            ...
        }
        //and so on for all nodes
        ...
    } //.toUnit() is not needed
```


4.3.4 Destructuring

Another potential maintainability issue can arise when new properties are added to existing node types. Knowing the locations of the code which must be modified to account for the new property can be a challenge.

Another way in which the Kotlin compiler can be enlisted to help improve code maintainability is with the use of destructuring. There is also a shortcoming in Kotlin's destructuring feature which we solve almost by accident.

Consider the following:

```
        when(expr) {
            //...
            is VariableReference -> {
                val (id, caseSensitivity) = expr
            }
            //...
        }
```

Unlike some languages, Kotlin's destructuring feature doesn't require that all the properties are mapped to variables on the left side of `=`. Unfortunately, this means that if a new property is added to `VariableReference`, the above example of destructuring will not result in an compile error of any kind.

By chance, all of the node types in the AST contain an additional property: `metas: MetaContainer`. The fact that this is *always* the last property defined in a node type is intentional. In fact, any and all new properties should be added immediately *before* the `metas: MetaContainer`. Consider:

```
        val (id, caseSensitivity, m) = expr
```

If a new property is added to `VariableReference` *before* `metas`, the type of varialbe `m` will be the type of that new property and *this* will result in compile-time errors from the Kotlin compiler at the locations where `m` is referenced. This is great for circumstances where `m` is needed, but there are also many cases where a node's `metas` are ignored, and if `m` is unused, the Kotlin compiler will issue a warning. For that scenario use the following:

```
        val (id, caseSensitivity, _: MetaContainer) = varRef
```

The `_: MetaContainer` component here simply causes a compile-time assertion that the third property of `VariableReference` is of type `MetaContainer`, resulting in a compile-time error whenever a new property is added.

4.3.5 Arbitrary Meta Information Can Be Attached to Any Node

TODO: leaving this part unspecified for the moment because there is still some uncertainty surrounding the how metas work.

4.4 Rules for working with the AST

- Always add new properties *before* the `metas: MetaContainer` of a new property.
- When using `when` with a sealed class and branching by derived types as a statement use `case()` and `toUnit()` to trick the Kotlin compiler into being helpful.
- When using `when` with the derived types of a sealed class, use destructuring in each branch where possible.

5 PartiQL AST

By AST in this document we refer to the data structure used to represent an PartiQL query. This is also the version of the AST provided to clients that consume the PartiQL AST.

5.1 Notation

5.1.1 Abusing Grammar notation

We borrow notation used for grammars to denote data structures and the alternatives for each data structure. You can think of production rule as a sum type, e.g.,

```
    LIST ::= `null` | `(` `cell` INTEGER LIST `)`
```

defines a linked list of `INTEGER`s as being one of

- the empty list, denoted by `null`, or,
- the list of 1 or more integers, e.g., `(cell 1 (cell 2 null))` is the list holding the numbers 1 and 2 in that order.

Terminals are in lowercase and surrounded with backticks (‘). Terminals denote literals.

Non-Terminals are in all caps and denote abstract type names.

The parallel bar `|` denotes alternatives.

Square brackets `[ ]` denote optional elements

5.1.2 Ellipsis

Ellipsis `...` mean 0 or more of the element *preceding* the ellipsis. We use parenthesis to denote a composite element

- `X ...` 0 or more `X`
- `(op X Y)...` 0 or more `(op X Y)`, i.e., `(op X Y)(op X Y)(op X Y)`

5.1.3 Referencing the Ion Text Specification

`ITS(<type>)` refers to the Ion Text Specification for the Ion `type` passed as an argument. For example, `ITS(boolean)` means lookup the section on `boolean` in the Ion Text Specification.

The AST is a valid Ion S-expression, `SEXP`. For the purposes of this documentation we use the following grammar for Ion and it's `SEXP`. The grammar *should* be a refactoring of Ion's grammar that allows us to create more convenient groupings of the Ion Text grammar for our purposes.

```
ION_VALUE ::=
              ATOM
            | SEXP
            | LIST
            | STRUCT

ATOM      ::=
              BOOL
            | NUMBER
            | TIMESTAMP
            | STRING
            | SYMBOL
            | BLOB
            | CLOB
            | NULL


BOOL      ::= ITS(boolean)
NUMBER    ::=
              INTEGER
            | FLOAT
            | DECIMAL

INTEGER   ::= ITS(integer)
FLOAT     ::= ITS(float)
DECIMAL   ::= ITS(decimal)
```

```
TIMESTAMP ::= ITS(timestamp)
STRING    ::= ITS(string)
SYMBOL    ::= ITS(symbol)
BLOB      ::= ITS(blob)
CLOB      ::= ITS(clob)
NULL      ::= ITS(null)

LIST      ::= ITS(list)
STRUCT    ::= ITS(structure)

SEXP      ::= `(` ION_VALUE ... `)`
```

In the case of `SEXP` we refer to the name immediately following the `SEXP` 's open parenthesis as its **tag**. For example the `SEXP (lit 2)` is tagged with the symbol `lit`.

5.2 PartiQL AST data definition

Starting with version 1 of the PartiQL AST, each AST must be wrapped in an `ast` node:

```
(ast (version 1)
     (root EXP))
```

If the top-most node of the AST does not have the tag `ast` then we assume that it is a legacy version 0 AST.

Within `(root ...)` we represent a valid PartiQL expression as a tree (`SEXP`) of **optionally** wrapped nodes called **term**s.

```
TERM ::= `(` `term` `(` `exp` EXP `)`
                    `(` `meta` META ...`)` `)`
```

The wrapper `term` contains 2 sub `SEXP`

- the PartiQL expression or expression fragment being wrapped tagged with `exp`
- the meta information for the PartiQL expression or expression fragment tagged with `meta`

The `meta` child node of `term` is optional but usually holds information on the location of the expression in the original query. The definition of `META` is purposely left open to allow for the addition of meta information on the AST by consumers of the PartiQL such as query rewriters.

The PartiQL implementation shall use the `$` prefix for all of its meta node tags. A naming convention such as reverse domain name notation should be used for meta node tags introduced by consumers of the PartiQL AST to avoid naming conflicts.


```
META ::= LOC | META_INFO

LOC  ::= `(` `$source_location` `(` `{` `line_num` `:` INTEGER`,` `char_offset` `:` INTEGER
     `}` `)` `)`

META_INFO ::= `(` SYMBOL `(` ION_VALUE ... `)` `)`
```

For example:

```
(term (exp (lit customerId))
      (meta ($source_location ({line_num: 10, char_offset: 12}))
            (type_env ({ customerId : Symbol }))))
```

Captures the literal `customerId` that appears on line `10` at character offset `12` in the original query. The term's meta information also captures the type environment for this expression. In this example the values bound to `customerId` are expected to be of type `Symbol`.

`EXP` defines the alternatives for an `SEXP` that that can appear inside a `term` 's `exp` tagged `SEXP`. **For readability, any nested `term`-wrapped sub components of `EXP` are not shown in this definition.**

The `term` wrapper is optional and so the previous example can be stripped down to the semantically equivalent:

```
      (lit customerId)
```

5.2.1 PartiQL AST "Grammar"

```
EXP ::=
        `(` `lit` ION_VALUE `)`
      | `(` `missing` `)`
      | `(` `id` SYMBOL CASE_SENSITIVITY `)`

      | `(` `struct` KEY_VALUE ... `)`
      | `(` `list` EXP ... `)`
      | `(` `bag`  EXP ... `)`

      | `(` NARY_OP EXP EXP ... `)`
      | `(` TYPED_OP EXP TYPE `)` `)`

      | `(` `path` EXP PATH_ELEMENT... `)`

      | `(` `call_agg` SYMBOL QSYMBOL EXP `)`
```



```
        | `(` `call_agg_wildcard` `count` `)`

        | `(` `simple_case` EXP WHEN [WHEN ...] [ELSE] `)`
        | `(` `searched_case` WHEN [WHEN ...] [ELSE] `)`

        | `(` `select` SELECT_LIST
                       FROM_SOURCE
                       [ '(' `where' EXP `)` ]
                       [ GROUP_CLAUSE ]
                       [ `(` `having` EXP `)` ]
                       [ `(` `limit` EXP `)` ]

        | `(` `pivot` MEMBER
                      FROM_SOURCE
                      [ '(' `where' EXP `)` ]
                      [ GROUP_CLAUSE ]
                      [ `(` `having` EXP `)` ]
                      [ `(` `limit` EXP `)` ]

    LENGTH ::= INTEGER


    TYPE ::= `(` `type` TYPE_NAME [LENGTH [LENGTH]] `)`
    //NOTE: the two optional length arguments above are meant to capture the length or
        precision and scale
    //arguments as defined by the SQL-92 spcification for certain data types.  While space
        exists in the AST
    //for them, they do not apply to the Ion type system and therefore are ignored at during
        compilation and
    //evaluation times.

    CASE_SENSITIVITY ::= `case_sensitive` | `case_insensitive`

    KEY_VALUE ::= '(' EXP EXP ')'

    TYPE_NAME ::= `boolean` | `smallint` | `int`  | `float` | `decimal`
                  | `numeric` | `double_precision` | `character` | `character_varying` | `
                      string`
                  | `symbol` | `varchar` | `list` | `bag` | `struct`
                  | `clob` | `blob` | `timestamp` | `symbol` | `null`
                  | `missing` | `null` | `missing`

    TYPED_OP ::= `cast` | `is`
```

```
NARY_OP ::= // NOTE:  When used with an arity of 1, the operators +, - and NOT function as
    unary operators.
          `+` | `-` | `not` | `/` | `*` | `%`
        | `<` | `<=` | `>` | `>=` | `<>`
        | `and | `or` | `||` | `in` | `call`
        | 'between` | `like` | `is`
        | `union | `union_all`
        | `intersect` | `intersect_all`
        | `except` | `except_all`

PATH_ELEMNT ::= `(` `path_element` PATH_EXP CASE_SENSITIVITY `)`
PATH_EXP ::= EXP
    | `(` `star` `)`
    | `(` `star `unpivot` `)`

MEMBER ::= `(` `member` EXP EXP `)`

QSYMBOL ::= `distinct` | `all`

SELECT_LIST ::= `(` PROJECT SELECT_PROJECTION `)`

PROJECT ::= `project` | `project_distinct`

SELECT_PROJECTION ::=
           | `(` `value` EXP `)`
           | `(` `list` SELECT_LIST_ITEM... `)`
           | `(` `list` STAR `)`

SELECT_LIST_ITEM ::= EXP
    | `(` `as` SYMBOL EXP `)`
    | `(` `path_project_all` EXP `)`

FROM_SOURCES ::= `(` `from` FROM_SOURCE `)`

JOIN_TYPE ::= `inner_join` | `outer_join` | `left_join` | `right_join`

FROM_SOURCE_TABLE ::= EXP
    | `(` `as` SYMBOL EXP `)`
    | `(` `at` SYMBOL `(` `as` SYMBOL EXP `)` `)`
    | `(` `unpivot` EXP `)`

FROM_SOURCE ::= FROM_SOURCE_TABLE
```

```
        | `(` JOIN_TYPE FROM_SOURCE EXP `)`

    GROUP_FULL ::= `group`
    GROUP_PARTIAL ::= `group_partial`
    GROUP_ALL ::= `group_all`
    GROUP_BY_EXP ::= EXP | `(` `as` SYMBOL EXP `)`
    GROUP_KIND ::= GROUP_FULL | GROUP_PARTIAL

    GROUP_CLAUSE  ::=
        `(`  GROUP_KIND
             //NOTE:  the `by` node cannot be wrapped in a term (GROUP_BY_EXPs however *can* be
                  wrapped in a term).
            `(` `by` GROUP_BY_EXP GROUP_BY_EXP... `)`
             //NOTE:  the `name` node *can* be wrapped in a term
            [ `(` `name` SYMBOL `)` ] ')'
        | `(` GROUP_ALL SYMBOL `)`


    WHEN ::= `(` `when` EXP EXP `)`
    ELSE ::= `(` `else` EXP `)`
```

5.3 Examples

Each example lists:

1. The query as a string as the title
2. The version 0 AST
3. =>
4. The version 1 AST

The examples show some `meta`/`term` nodes based on what meta information is available today. The goal is to annotate all nodes with meta information.

5.3.1 E1 : `select * from a`

```
    (select (project (*))
        (from (meta (id a case_insensitive)
                        {line:1, column:15})))
```

=>


```
(ast
    (version 1)
    (root
        (term
            (exp
                (select
                    (project
                        (list
                            (term
                                (exp (star))
                                (meta ($source_location ({line_num:1,char_offset:8}))))))
                    (from
                        (term
                            (exp (id a case_insensitive))
                            (meta ($source_location ({line_num:1,char_offset:15}))))))))))
```

5.3.2 E2 : `select a as x from a`

```
(select (project (list (meta (as x (meta (id a case_insensitive)
                                         {line:1, column:8 }))
                             { line:1, column:13 })))
        (from (meta (id a case_insensitive)
                    { line:1, column:20 })))
```

=>

```
(ast
    (version 1)
    (root
        (term
            (exp
                (select
                    (project
                        (list
                            (term
                                (exp
                                    (as
                                        x
                                        (term (exp (id a case_insensitive))
                                            (meta ($source_location ({line_num:1,
                                                char_offset:8}))))))))))))))
```

```
                                    (meta ($source_location ({line_num:1,char_offset:13})))))
                        (from
                            (term
                                (exp (id a case_insensitive))
                                (metaJ ($source_location ({line_num:1,char_offset:20})))))))))
```

5.3.3 E3 : `select x from a as x`

```
    (select (project (list (meta (id x case_insensitive)
                                 { line:1, column:8 })))
            (from (meta (as x (meta (id a case_insensitive)
                                    { line:1, column:15 }))
                        { line:1, column:20 })))
```

=>

```
    (ast
        (version 1)
        (root
            (term
                (exp
                    (select
                        (project
                            (list
                                (term
                                    (exp (id x case_insensitive))
                                    (meta ($source_location ({line_num:1,char_offset:8})))))))
                        (from
                            (term
                                (exp
                                    (as
                                        x
                                        (term
                                            (exp (id a case_insensitive))
                                            (meta ($source_location ({line_num:1,char_offset
                                                :15})))))))
                                (meta
                                    ($source_location ({line_num:1,char_offset:20})))))))))
```

5.3.4 E4 : `select AVG(a.id) from a`

```
    (select (project (list (meta (call_agg avg all (path (meta (id a case_insensitive)
                                                               { line:1, column:12 })
                                                         (meta (case_insensitive (meta (lit "
                                                             price")
                                                                                       { line
                                                                                           :1,
                                                                                           column
                                                                                           :14
                                                                                           }))
                                                               { line:1, column:14 })))
                                    { line:1, column:8 })))
            (from (meta (id a case_insensitive)
                        { line:1, column:26 })))
```

=>

```
    (ast (version 1)
        (root
            (term
                (exp
                    (select
                        (project
                            (list
                                (term
                                    (exp
                                        (call_agg avg all
                                            (term
                                                (exp
                                                    (path
                                                        (term
                                                            (exp (id a case_insensitive))
                                                            (meta ($source_location ({line_num
                                                                :1,char_offset:12}))))
                                                        (path_element
                                                            (term
                                                                (exp (lit "id"))
                                                                (meta ($source_location ({
                                                                    line_num:1,char_offset:14})
                                                                    )))
                                                            case_insensitive)))
                                                (meta ($source_location ({line_num:1,
                                                    char_offset:12}))))))))
```

```
                                    (meta ($source_location ({line_num:1,char_offset:8})))))))
                        (from
                            (term
                                (exp (id a case_insensitive))
                                (meta ($source_location ({line_num:1,char_offset:23}))))))))))
```

5.3.5 E5 : `select * from a where a.price > 100`

```
    (select (project ( *))
            (from (meta (id a case_insensitive)
                        { line:1, column:15 }))
            (where (meta (> ( path (meta (id a case_insensitive)
                                         { line:1, column:23 })
                                   (meta (case_insensitive (meta (lit "price")
                                                                 { line:1, column:25 }))
                                         { line:1, column:25 }))
                            (meta (lit 100)
                                  { line:1, column:33 }))
                        { line:1, column:31 })))
```

=>

```
    (ast (version 1)
        (root
            (term
                (exp
                    (select
                        (project
                            (list
                                (term
                                    (exp (star))
                                    (meta
                                        ($source_location ({line_num:1,char_offset:8} ))))))
                        (from
                            (term
                                (exp (id a case_insensitive))
                                (meta ($source_location ({line_num:1,char_offset:15} )))))
                        (where
                            (term
                                (exp
                                    (>
                                        (term
```

```
                                        (exp
                                            (path
                                                (term
                                                    (exp (id a case_insensitive))
                                                    (meta ($source_location ({line_num:1,
                                                        char_offset:23} ))))
                                                (path_element
                                                    (term
                                                        (exp (lit "price"))
                                                        (meta ($source_location ({line_num
                                                            :1,char_offset:25} ))))
                                                    case_insensitive)))
                                            (meta ($source_location ({line_num:1,char_offset
                                                :23} ))))
                                    (term
                                        (exp (lit 100))
                                        (meta ($source_location ({line_num:1,char_offset
                                            :33} ))))))
                            (meta ($source_location ({line_num:1,char_offset:31} ))))))))))
```

5.3.6 E6 : `SELECT a, b FROM data GROUP BY a, b`

```
    (term
        (exp
            (select
                (project
                    (list
                        (term
                            (exp
                                (id a case_insensitive)))))
                (from
                    (term
                        (exp
                            (id data case_insensitive))))
                (group
                    (by
                        (term
                            (exp
                                (id a case_insensitive)))
                        (term
                            (exp
                                (id b case_insensitive)))))))))
```

5.3.7 E7 : `SELECT a FROM data GROUP BY a as x GROUP BY g`

```
(term
    (exp
        (select
            (project
                (list
                    (term
                        (exp
                            (id a case_insensitive)))))
            (from
                (term
                    (exp
                        (id data case_insensitive))))
            (group
                (by
                    (term
                        (exp
                            (as
                                x
                                (term
                                    (exp
                                        (id a case_insensitive)
                                    ))))))
                (term
                    (exp
                        (name g)))))))
```

5.4 Callouts

Important modifications/additions regarding changes from version 0:

1. Some nodes had 2 versions one for the node e.g., `is` and one for the nodes complement (negation) e.g. `is_not`
    - Removed the complements in favour of a nested `(not ..)` expression. The nodes affected are `like`, `is`,
2. The `*` was previously used in multiple places to denote different semantic meanings. This is no longer the case.

- `SELECT *` is now denoted with the tag `star`
- `SELECT exp.*` is now denoted with the tag `path_project_all`
- `exp[*]` and `exp.*` is still denoted with the `(*)` and `(* unpivot)` s-expressions. This is unchanged from version 0.

3. Path elements (arguments after the first in a path expression) are now contained in a `path_element` node following this pattern: `(path_element EXP CASE_SENSITIVITY)`.

5.5 ToDo

Need to address/flesh out

1. Order by (https://github.com/partiql/partiql-lang-kotlin/issues/47)

6 FAQ

TODO: add questions to the FAQ.