Expression Notation (CXN)

Expressions in CDS definitions and queries can be one of:

Table of Contents

```js
expr = // one of...
  val    |   // [literal values]: #literal-values
  ref    |   // references or functions
  xpr    |   // operator expressions
  func   |   // function calls
  list   |   // lists/tupels
  param  |   // binding parameters
  sym    |   // enum symbol
  SELECT     // subqueries
```

Literal Values

Literal values are represented as `{val:...}` with property `val` holding the actual literal value as specified in JSON.

```js
val = {val:literal}
literal = string | number | true | false | null
```

Examples:

```js
cds.parse.expr(`'a string'`)  == {val:'a string'}
cds.parse.expr(`11`)  == {val:11}
cds.parse.expr(`true`)  == {val:true}
cds.parse.expr(`null`)  == {val:null}
cds.parse.expr(`date'2023-04-15'`)  == {val: '2023-04-15', literal: 'date'}
cds.parse.expr(`time'13:05:23Z'`)  == {val: '13:05:23Z', literal: 'time'}
cds.parse.expr(`timestamp'2023-04-15T13:05:23Z'`)  == {val: '2023-04-15T13:05
```

References

A reference is represented as `{ ref: … }` with property `ref`. This property holds an array of reference segments as plain identifier strings. Only in case of infix filters and/or arguments, the property holds an object `{ id: 'identifier', … }` and all properties except `id` are optional, as shown in the following snippet:

```js
ref = {ref:[..._segment]}
_segment = string | { id: string, args: _named, where: _xpr,
                      groupBy: [ ...expr ], having: _xpr,
                      orderBy: [ ...ordering_term ], limit: { rows: expr, off
_named = { ... <name>:expr }
```

Examples:

```js
let cqn4 = cds.parse.expr
cqn4(`![keyword]`) == {ref:['keyword']}
cqn4(`foo.bar`) == {ref:['foo','bar']}
cqn4(`foo[9].bar`) == {ref:[{ id:'foo', where:[{val:9}] }, 'bar' ]}
```

```js
cqn4(`foo(p:x).bar`) == {ref:[{ id:'foo', args:{p:{ref:['x']}} }, 'bar' ]}
cqn4(`foo[where a=1 group by b having b>2 order by c limit 7].bar`)
  == {ref:[{ id:'foo', where:[{ref:['a']}, '=', {val:9}],
                     groupBy: [{ref: ['b']}], having: [{ref: ['b']}, '>',
                     orderBy: [{ref: ['c']}], limit: {rows: {val: 7}} },
           'bar' ]}
```

Function Calls

Function calls are represented as follows:

```js
func = { func:string, args: _positional | _named, xpr:_xpr }
_positional = [ ...expr ]
_named = { ... <name>:expr }
```

The optional attribute `xpr` is used for the `over` clause of SQL window functions.

Examples:

```js
let cqn4 = cds.parse.expr
cqn4(`foo(p=>x)`) == {func:'foo', args:{p:{ref:['x']}}}
cqn4(`sum(x)`)    == {func:'sum', args:[{ref:['x']}]}
cqn4(`count(*)`) == {func:'count', args:['*']}
cqn4(`rank() over (...)`) == {func:'rank', args:[], xpr:['over', {xpr:[...]}]
```

Method style function calls and instantiation syntax for spatial functions are represented as `{xpr:...}` with `.` and `new` as operators:

```js
cqn4(`shape.ST_Area()`)
  == {xpr: [{ref: ['shape']}, '.', {func: 'ST_Area', 'args': []}]}
cqn4(`new ST_Point(2, 3)`)
  == {xpr: ['new', {func: 'ST_Point', args: [{val: 2}, {val: 3}]}]}
```

Lists

Lists or tupels are represented as `{list:...}`, with property `list` holding an array of the list entries.

Examples:

```js
cds.parse.expr(`(1, 2, 3)`) == {list: [{val: 1}, {val: 2}, {val: 3}]}
cds.parse.expr(`(foo, bar)`) == {list: [{ref: ['foo']}, {ref: ['bar']}]}
```

Operator Expressions

Operators join one or more expressions into complex ones, represented as `{xpr:...}`. The property `xpr` holds a sequence of operators and operands.

```js
xpr = {xpr:_xpr}
_xpr = [...( _operand | _operator )]
_operand = expr
_operator = string
```

- *Operands* can be any kind of expression
- *Operators* are represented as plain strings, like `'='` or `'and'`
- Parentheses `( ... )` around sub-expressions are represented as nested `xpr`

Examples:

```js
[dev] cds repl
> cds.parse.expr(`x<9`)  ==
{xpr:[ {ref:['x']}, '<', {val:9} ]}

> cds.parse.expr(`x<9 and (y=1 or z=2)`)  ==
{xpr:[
  {ref:['x']}, '<', {val:9}, 'and', {xpr:[
    {ref:['y']}, '=', {val:1}, 'or', {ref:['z']}, '=', {val:2}
  ]}
]}

> cds.parse.expr(`exists books[year = 2000]`)  ==
{xpr:[
  'exists',
  {ref: [ {id:'books', where:[ {'ref':['year']}, '=', {'val': 2000} ]}]}
]}
```

CQN intentionally doesn't aim to *understand* the individual operators and related expressions. It captures them as arbitrary sequences, in the same lexical structure and order they're written in the source. This 'ignorance' allows us to stay open to any kind of operators and keywords. For example, we can easily express native extensions of underlying database dialects.

As an exception to that rule, CDS supports the ternary conditional operator on source level, but immediately converts it to the corresponding CASE expression in CXN:

```js
[dev] cds repl
> cds.parse.expr(`x<10 ? y : z`)  ==
{xpr:['case', 'when', {ref:['x']}, '<', {val:10},
      'then', {ref:['y']}, 'else', {ref:['z']}, 'end']}
```

Binding Parameters

Binding parameters for prepared statements are represented as *`{ref:...,`* *`param:true}`* with values for *`ref`* as follows.

```js
param = { ref:[ '?' | number | name ], param:true }
```

Examples:

```js
[dev] cds repl
> cds.parse.expr(`x=:1`) //> [{ref:['x']}, '=', {ref:[1], param:true}]
> cds.parse.expr(`x=:y`) //> [{ref:['x']}, '=', {ref:['y'], param:true}]
> cds.parse.expr(`x=?`)  //> [{ref:['x']}, '=', {ref:['?'], param:true}]
```

Sub Queries

↳ *See CQN*

Edit this page

Last updated: 30/01/2025, 11:59

Previous page
Query Notation (CQN)

Next page
Core / Built-in Types

Was this page helpful?

👍 👎