A tutorial for implementing a programming language on top of q/kdb+

José Pablo Cambronero and Dennis Shasha

Courant Institute/New York University

September 17, 2016

Motivation

Why would we want our own programming language?

- Fit the language to the task: rise in popularity of Domain Specific Languages (DSL)
- Enable custom optimizations
- Make life for experts easier, allow non-experts a comfortable way to explore useful computing

Why do we want to target q/kdb+?

- High level
- Easy metaprogramming (parse/eval/value): *data* $\leftrightarrow$ *program*
- Blazingly fast (something everyone in this room knows)

Overview of Domain Specific Language Approaches

1. External: stand-alone, with own parser, (possibly analysis), translation, and execution [1]
2. Internal (aka embedded): extend an existing language
   - Shallow: domain specific language implemented directly in the host language semantics, using function calls (no abstract syntax tree built)
   - Deep: Constructs abstract syntax tree through calls, semantics are given by an "interpreter" function[2]
   - There are some very interesting, and deeper connections, between the two initially disparate-seeming approaches, see [2] for a nice overview

AQuery: a hybrid between external and internal. Has standalone parsing and translation, but relies on q for execution and allows embedding of literal q code in AQuery files

Orderly: A (very simple) DSL for market orders

- We need a language simple enough to address in 20-30 min but meaty enough for fun
- We'll follow both approaches from above:
  - "External": Standalone parsing, analysis and translation in external-orderly
  - "Internal (deep)": Parsing using o) mode, analysis and execution in internal-orderly

Orderly EBNF

$\langle \textit{program} \rangle$ ::= ($\langle \textit{order} \rangle$ `->` $\langle \textit{ident} \rangle$) +

$\langle \textit{order} \rangle$ ::= $\langle \textit{side} \rangle$ $\langle \textit{vol} \rangle$ `of` $\langle \textit{ident} \rangle$ `at` `$`$\langle \textit{num} \rangle$ $\langle \textit{modifier} \rangle$ (`for` $\langle \textit{ident} \rangle$)?

$\langle \textit{side} \rangle$ ::= `buy`
| `sell`

$\langle \textit{vol} \rangle$ ::= $\langle \textit{num} \rangle$ `shares`
| `$` $\langle \textit{num} \rangle$

$\langle \textit{mod} \rangle$ ::= `if` `"`$\langle \textit{monadic-fun-in-q} \rangle$`"`
| `on` $\langle \textit{date} \rangle$

Orderly Running Examples

```
sell 1000 shares of IBM at $50 on 05/16/2016 for BAML -> order_book
 // uses an embedded q fun
buy $1e6 of AAPL at $100 if "{[env] 10 < exec avg close_price from env where
    sector=`Tech} " -> order_book
```

```
q)o)sell 1000 shares of IBM at $50 on 05/16/2016 for BAML -> order_book
`order_book
q)o)buy $1e6 of AAPL at $100 if "{[env] 10 < exec avg close_price from env where sector=`Tech} " -> o
rder_book
`order_book
q)order_book
side volume  units  ticker px  cond                                                ..
-----------------------------------------------------------------------------------..
sell 1000    shares IBM    50  {[env;d] env[`date]=d}[;2016.05.16]                 ..
buy  1000000 usd    AAPL   100 {[env] 10 < exec avg close_price from env wher..
```

Figure 1: Internal: End-to-end execution

```
josecambronero orderly-external$ orderly examples.o | tail -n 4
// generated by Orderly
`order_book upsert flip `side`volume`units`ticker`px`cond`client!((),/:(`sell;1000.0;`shares;`IBM;50.
0;wrapDate "D"$ "05/16/2016" ;`BAML))

`order_book upsert flip `side`volume`units`ticker`px`cond`client!((),/:(`buy;1000000.0;`usd;`AAPL;100
.0;{$[isUnary x;x;'y]}[value "{[env] 10 < exec avg close_price from env where sector=`Tech} "; "Found
 error Non-unary lambda at line 3 and column 29"];`self))
```

Figure 2: Internal: External code generation (w/o optimization)

Syntax Analysis

- Accept programs that satisfy grammar, reject all others. Construct AST for accepted.

```
josecambronero orderly-external$ orderly --parse examples.o
SingleInsert(order_book,MarketOrder(Sell,IBM,NumShares(1000.0),PDouble(50.0),Right(PString(05/16/2016
)),Some(BAML)))
SingleInsert(order_book,MarketOrder(Buy,AAPL,USDAmount(1000000.0),PDouble(100.0),Left(Verbatim("{[env
] 10 < exec avg close_price from env where sector=`Tech} ")),None))
```

Figure 3: External: Succesful parse's resulting AST

Syntax Analysis (aka tokenization + parsing)

- Ideally parsing code is clear and extendable

```
def order: Parser[MarketOrder] = positioned(
    side ~ volume ~ (OF ~> cleanIdent) ~
    (AT ~> "$" ~> floatingPointNumber) ~ modifier ~
    (FOR ~> cleanIdent).? ^^ {
        case s ~ v ~ sym ~ px ~ m ~ c => ...
    }
    | failure ("Marked orders should specify side, volume of purchase, ticker,
     price, modifier and optionally a client")
    )
```

Figure 4: External: main parsing function in Scala

```
// orderly grammar as a list (we have a simple grammar :) )
grammar:(side;vol;accept[OF;"of"];ident;accept[AT;"at"];price;modifier;forClause
     ;accept[{x~"->"};"->"];ident);

// wrap to avoid having errors return deeper functions (no need to worry
// user with implementation)
parser:{@[raze consume[grammar; ] tokenize@;x;{'x}]}
```

Figure 5: Internal: main parsing function in q

Syntax Analysis Goals

- Generate useful parser errors (with context and/or source location if possible)
- Let's modify our otherwise correct sentences

```
q)parser "sell 1000 shares of IBM at $50 on 05/2016 for BAML -> order_book"
{@[raze consume[grammar; ] tokenize@;x;{'x}]}
'error: Expected date found 05/2016
```

Figure 6: Internal: bad date error

```
josecambronero orderly-external$ echo 'buy $1e6 of AAPL at $100 "{[env] 10 < exec avg close_price fro
m env where sector=`Tech}" -> order_book ' | orderly --parse
[1.26] failure: `ON' expected but `"' found

buy $1e6 of AAPL at $100 "{[env] 10 < exec avg close_price from env where sector=`Tech}" -> order_boo
k
                         ^
```

Figure 7: External: missing modifier keyword (if)

Semantic Analysis

- We've decided the input satisfies our grammar, now we provide meaning
- Provide checks: many things can be checked before runtime, even for simple languages
- Make sure these checks are composable and well documented

Semantic Analysis: Internal

```
// added 'should' to .q, but removed after defining ;)
check0:{
  getVolume[x] should be ({x > 0};{"Volume should be positive"});
  getPrice[x] should be ({x > 0};{"Price should be positive"});
  x:setModifier[x;] wrapDate getModifier x;
  getModifier[x] should be (isUnary;{"Expected unary function"});
  x
 };
// users dont need to know underlying checks
check:{@[check0;x;{'x}]}
```

Semantic Analysis: Internal (pre-processor level)

- Validating user input is critical. For example, in a full fledged language you might perform type checking.
- Orderly's market orders have "if" modifiers that are meant to be evaluated within a context to determine if the market order should be executed. Given this, we want the modifier to be a monadic function (1-argument). Evaluation of an "if-modifier" clause is then simply a call to the function with the environment (i.e. context) as an argument.

Semantic Analysis: Internal (pre-processor level) Example

Assume we have the following Orderly code

```
buy $1e6 of AAPL at $100 if "{[env;sec] 10<exec avg close_price from env where
    sector=sec}" -> order_book
```

- We must verify that the code within double-quotes corresponds to a q monadic function
- We can use runtime to resolve type of modifier clause, resolving identifier to function if necessary. Calling `type` yields 100h here.
- q's `value` allows us to further explore functions (another useful ability for metaprogramming). In this case, we can check that `value` doesn't show a partially evaluated function and (`@[;1] value`) shows two formal parameters.

Semantic Analysis: Internal (pre-processor level) Example

- Given that this doesn't satisfy our monadic requirement, trying to pass this through the checking function will result in an appropriate error

```
q)check parser "buy $1e6 of AAPL at $100 if \"{[env;sec] 10<exec avg close_price from env where secto
r=sec}\" -> order_book"
{@[check0;x;{'x}]}
'Expected unary function
```

Figure 8: Internal: Extra parameter in modifier clause lambda causes issues

- The user can fix this by providing a partially evaluated function (aka. projected function), which in effect makes the call monadic.

```
q)(check parser "buy $1e6 of AAPL at $100 if \"{[env;sec] 10<exec avg close_price from env where sect
or=sec}[;`Tech]\" -> order_book") 5
{[env;sec] 10<exec avg close_price from env where sector=sec}[;`Tech]
```

Figure 9: Internal: Valid lambda in modifier clause

Semantic Analysis: Internal (pre-processor level)

- In general: type checks/validation, local transformations
- Validate shares and prices to be ≥ 0 (more interesting validation: within certain standard deviation of market price, for risk purposes)

```
q)check parser "sell -1000 shares of IBM at $50 on 05/16/2016 for BAML -> order_book"
{@[check0;x;{'x}]}
'Volume should be positive
```

Figure 10: Internal: Invalid share number

- Rewrite date-based modifier to lambda: uniform AST means easier code generation in our case

```
q)(check parser "sell 1000 shares of IBM at $50 on 05/16/2016 for BAML -> order_book") 5
{[env;d] env[`date]=d}[;2016.05.16]
```

Figure 11: Internal: Dates become wrapped in lambda

Semantic Analysis: External

- soft type checking (AQuery), global/local transformations (pre-processor)
- Some of our checks cannot be performed until runtime (e.g. checking if function is monadic)
- Need context in error messages generated at runtime
- Our approach: insert check and error message into code generated

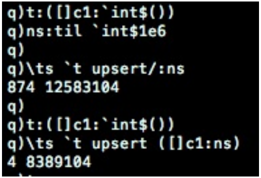
```
josecambronero orderly-external$ orderly --analyze examples.o
SingleInsert(order_book,MarketOrder(Sell,IBM,NumShares(1000.0),PDouble(50.0),Left(Verbatim(wrapDate "
D"$ "05/16/2016" )),Some(BAML)))
SingleInsert(order_book,MarketOrder(Buy,AAPL,USDAmount(1000000.0),PDouble(100.0),Left(Verbatim({$[isU
nary x;x;'y]}[value "{[env] 10 < exec avg close_price from env where sector=Tech} "; "Found error Non
-unary lambda at line 3 and column 29"])),None))
```

External: Static Rewrites

- Take advantage of global knowledge to perform global optimizations
  - Multiple passes can be done before generation. This allows increasingly complete knowledge
- Consider if rewrites should be idempotent and composable (most likely yes!)
  - In AQuery: Scala partial functions combined with pattern matching allow us to capture specific rewrites, while ignoring all other cases
  - In Orderly: transformations return new datatype, guaranteeing that transformation is called at appropriate time

External: Static Rewrites

- In Orderly: single inserts become bulk inserts (significant speedup)
- We group *SingleInsert* nodes in AST by the table they will be inserted into. We wrap these orders in a new datatype: *BulkInsert*
- Code generation treats *SingleInsert* as a simple single upsert, while *BulkInsert* creates a table and upserts multiple records at once



```
q)t:([]c1:`int$())
q)ns:til `int$1e6
q)
q)\ts `t upsert/:ns
874 12583104
q)
q)t:([]c1:`int$())
q)\ts `t upsert ([]c1:ns)
4 8389104
```

Figure 12: Upserting multiple records as a table can generate significant speed ups

External: Static Rewrites

- Pick an implementation language that allows nice rewrites (e.g. Scala's pattern matching makes life easier)
- Be explicit about any assumptions in the resulting code (e.g. in Orderly, we assume writes to different tables can be moved around, as long as the intra-table ordering remains constant)

```scala
  // reorders inter-table insertions, keeps order intra-table
def collectInserts(s: Seq[SingleInsert]): Seq[BulkInsert] =
    s.groupBy(_.t).map {
      case (t, os) => BulkInsert(t, os.map(_.order))
    }.toList
```

External: Static Rewrites

- Don't rely on later stages to perform rewrite-aware transformations/code-generation without any kind of checks. Leverage new AST node, along with common missing-case functionality in implementation languages to guarantee completeness of translation (e.g. BulkInsert datatype)

```
josecambronero orderly-external$ orderly --analyze --optimize examples.o
BulkInsert(order_book,List(MarketOrder(Sell,IBM,NumShares(1000.0),PDouble(50.0),Left(Verbatim(wrapDat
e "D"$ "05/16/2016" )),Some(BAML)), MarketOrder(Buy,AAPL,USDAmount(1000000.0),PDouble(100.0),Left(Ver
batim({$[isUnary x;x;'y]}[value "{[env] 10 < exec avg close_price from env where sector=Tech} "; "Fou
nd error Non-unary lambda at line 3 and column 29"])),None)))
```

Internal: Execution

- We have a simple `interpret` function that provides meaning to our AST. Note that in general this function doesn't necessarily need to act as an interpreter, but can also generated code etc. Avoid performing drastic transformations of the AST at this point (i.e. move deeper analysis to earlier stages)
- Orderly simply inserts the order details into the specified order book
- We don't generate code here, but rather use q directly to interpret the AST

```
toTable:{flip `side`volume`units`ticker`px`cond`client!(),/:x}
add:{[ast] (last ast) upsert toTable 7#(-1 _ ast),`self}
interpret:add
// orderly mode
.o.e:interpret check parser@
```

Internal: Execution

- Consider providing additional functions that might be relevant to your domain but don't necessarily merit embedding in representation.
- Users can leverage functions to manipulate the results produced through DSL (since embedded in same language)

```
satisfies:{[t;env] select from t where first each @[;env;0b] each cond }
```

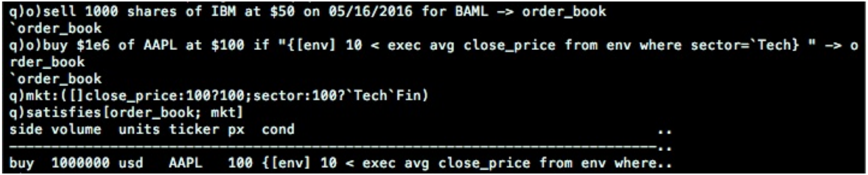
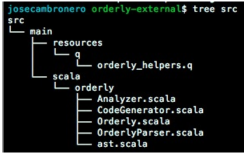


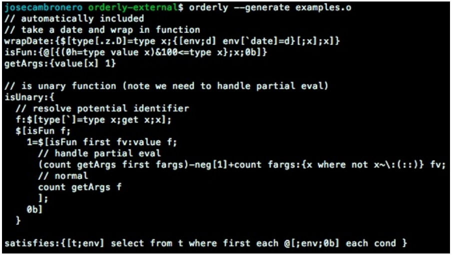
Figure 13: Internal: Users can use satisfies to extract orders that fit certain criteria

External: Code Generation

- Generate intelligible code (i.e. comment what is generated, indent etc to make human readable)
- Include any helper functions as a prelude in your generated code. This creates easily movable files



```
josecambronero orderly-external$ tree src
src
└── main
    ├── resources
    │   └── q
    │       └── orderly_helpers.q
    └── scala
        └── orderly
            ├── Analyzer.scala
            ├── CodeGenerator.scala
            ├── Orderly.scala
            ├── OrderlyParser.scala
            └── ast.scala
```



```
josecambronero orderly-external$ orderly --generate examples.o
// automatically included
// take a date and wrap in function
wrapDate:{$[type[.z.D]=type x;{[env;d] env[`date]=d}[;x];x]}
isFun:{@[{(0h=type value x)&100<=type x};x;0b]}
getArgs:{value[x] 1}

// is unary function (note we need to handle partial eval)
isUnary:{
  // resolve potential identifier
  f:$[type[`]=type x;get x;x];
  $[isFun f;
    1=$[isFun first fv:value f;
      // handle partial eval
      (count getArgs first fargs)-neg[1]+count fargs:{x where not x~\:(::)} fv;
      // normal
      count getArgs f
      ];
    0b]
  }

satisfies:{[t;env] select from t where first each @[;env;0b] each cond }
```

Figure 14: External: Use helper functions and include in generated code

External: Code Generation

- Write code that takes advantage of constructs like string interpolation to create a clean and maintainable generator

```
def genValues(m: MarketOrder): String = {
    val side = m.side match {
      case Buy => kdbSym("buy")
      case Sell => kdbSym("sell")
    }
    val (volume, units) = m.volume match {
      case NumShares(n) => (n, kdbSym("shares"))
      case USDAmount(d) => (d, kdbSym("usd"))
    }
    val ticker = kdbSym(m.sym)
    val px = m.px.v
    val cond = m.when match {
      case Left(Verbatim(q)) => q
      case _ => throw new Exception("when conditions should be translated to q
     code for generation")
    }
    val client = kdbSym(m.client.getOrElse("self"))
    s"($side;$volume;$units;$ticker;$px;$cond;$client)"
  }
```

External: Code Generation

- Provide explicit means of turning on/off rewrites. Can help user become familiar with transformations and increases transparency of DSL.

```
josecambronero orderly-external$ orderly --generate examples.o | tail -n 4
// generated by Orderly
`order_book upsert flip `side`volume`units`ticker`px`cond`client!((),/:(`sell;1000.0;`shares;`IBM;50.
0;wrapDate "D"$ "05/16/2016" ;`BAML))

`order_book upsert flip `side`volume`units`ticker`px`cond`client!((),/:(`buy;1000000.0;`usd;`AAPL;100
.0;{$[isUnary x;x;'y]}[value "{[env] 10 < exec avg close_price from env where sector=`Tech} "; "Found
 error Non-unary lambda at line 3 and column 29"];`self))
josecambronero orderly-external$ orderly --generate --optimize examples.o | tail -n 5
// generated by Orderly
`order_book upsert flip `side`volume`units`ticker`px`cond`client!flip (
        (`sell;1000.0;`shares;`IBM;50.0;wrapDate "D"$ "05/16/2016" ;`BAML);
        (`buy;1000000.0;`usd;`AAPL;100.0;{$[isUnary x;x;'y]}[value "{[env] 10 < exec avg close_price
from env where sector=`Tech} "; "Found error Non-unary lambda at line 3 and column 29"];`self)
        )
```

Figure 15: External: Compare non-optimized vs optimized orderly. Activation clearly indicated as command line argument

References I

[]

Cyrille Artho, Klaus Havelund, Rahul Kumar, and Yoriyuki Yamagata.
Domain-specific languages with scala.
In *International Conference on Formal Engineering Methods*, pages 1–16. Springer, 2015.

Jeremy Gibbons and Nicolas Wu.
Folding domain-specific languages: deep and shallow embeddings (functional pearl).
In *ACM SIGPLAN Notices*, volume 49, pages 339–347. ACM, 2014.

Alberto Lerner.
Querying Ordered Databases with Aquery.
PhD thesis, Ph.D. Thesis, Ecole Nationale Superieure de Telecommunications, ENST-Paris, 2003.

References II

Alberto Lerner and Dennis Shasha.
Aquery: Query language for ordered data, optimization techniques, and experiments.
In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 345–356. VLDB Endowment, 2003.

Martin Odersky, Lex Spoon, and Bill Venners.
Programming in scala.
Artima Inc, 2008.

Arthur Whitney.
Abridged Q Language Manual, 2009 (accessed November 6, 2015).