MOSDEX Syntax

Version 2-0

Dr. Jeremy A. Bloom
jeremyblmca@gmail.com
November 23, 2022

Synopsis of Changes In v2-0 vs. v1-2

- Eliminated named objects (Problems and Tables)
  - All objects now have a keyword as field name
  - The MOSDEX File has a Modules array
  - The MOSDEX Module has a Tables array
  - Problem has been renamed Module
- New syntax for schema
  - Eliminated named fields in favor of two arrays: FIELDS and TYPES
  - Horizontal layout aligns over the instance columns and eliminates a separate Fields array (this is not part of the standard; JSON doesn't care about the layout)
- Function calls
  - Needed for retrieving the solution
  - Define a data type for function calls for use in the schema
  - Parse the function call from a string in the MOSDEX File
- Recipe Object has been renamed Query.
- Type (schema) information added in SELECT Queries
  - MOSDEX supports data types (IEEEDOUBLE, Function calls) that are not supported in SQL.
  - MOSDEX needs to carry this type information through its data transformations from input -> model -> output
  - Solution: add the type information as comments (denoted by --) in the SQL
- Solver result calls are specified as part of the relevant modeling artifact
  - Results can be reformatted as OUTPUT DATA tables using SQL.
- UPDATE and APPEND query fields added to table to facilitate iterative modular structures
  - e.g. decomposition

Synopsis of Changes In v1-2 vs. v1-1

- Revised the Recipe object as an array of Clause objects, each of which specifies an SQL directive and its predicate. Fuller explanation of how SQL Recipes work.
- Added an optional Keys field to Table and its subclasses for use in queries with string substitution.
- All fields of Heading are arrays of strings, for consistency.
- Use #Solver prefix on calls to solver methods.

Synopsis of Changes In v1-1 vs. v1-0

- Implemented the MOSDEX syntax as a JSON Schema.
- Added a Syntax field to the MOSDEX file to verify the version of the JSON Schema used.
- Exposed the Table class as an general artifact that can be used in future extensions of MOSDEX
- Replaced the Role field of Table and its subclasses with a mandatory Type field, whose allowable values depend on the class, and put the role information in it; added a Type field to the class schemas.
- Added an Expression subclass of Table for specifying nonlinear problems using expression graphs.
- Renamed the Expression field in Heading as Math to avoid name conflict with the Expression class.
- Renamed the Coefficient subclass of Table as Term and created separate schemas for linear, quadratic and nonlinear types to accommodate the corresponding number and types of arguments.
- Minor typographical corrections.

Synopsis of Changes In v1-0 vs. v0-3

- MOSDEX syntax has been re-envisioned based on development of many examples. Many changes have been made to simplify and generalize the representation of optimization problems and to improve readability.
- Sections and type tags for Problems and Tables have been removed. A MOSDEX File now has Problems as its only named objects, and a Problem has Tables as its only named objects. As a result, the parser can infer the type of a named object before parsing it.
- A mathematical Expression section has been added to the Heading object, which was also renamed from Header.
- Arrays of strings are used extensively in place of ordinary strings in order to allow multi-line format and other segmentation of string information.
- Instance has replaced Rows as the data container in an instance form Table.
- Singleton provides a compact representation of an instance form table with a single record.
- SQL queries have separate clauses for Select, From, etc., to facilitate working with database managers that do not natively accept SQL query strings.
- Properties object has been removed from modeling artifacts, and the information it contained has been placed into fields of the object's schema.
- Fields for solver results have been added to the modeling artifacts.
- The Modules object has been removed, and its functions have been replaced by Import, Index, For Each, and other dispersed objects more closely identified with the objects they affect.
- Path names and string substitutions have been implemented.

1. Introduction and Overview

This paper defines a formal syntax for MOSDEX. We use extended Backus-Naur form (EBNF) (see https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form) and also present the corresponding syntax diagrams.

At the highest level, a *MOSDEX File* is a collection of optimization *Modules*, each of which in turn is a collection of *Tables*. A Table represents a table in a relational database. Subclasses of Table represent Data and modeling artifacts, such as Variables and Constraints. Every Table has a *Schema*, which defines its fields and their data types. Data tables can have any reasonable schema, while the schemas of the modeling artifacts are largely fixed by the requirements of an optimization solver. A Table can have either *instance* or *query* form. Instance form tables contain data while query form tables use SQL queries to reshape and populate data from other tables. MOSDEX can link problems together in a *modular structure*, to represent, for example, decomposition or stochastic programs. MOSDEX can also handle nonlinear problems using expression graphs.

The purpose of MOSDEX is to standardize data exchange with mathematical optimization solvers, of which there are many varieties. It is important to realize that while MOSDEX affords considerable expressive capabilities to represent general mathematical optimization problems, it can also be perceived as quite complicated. However, if the problem at hand does not require all of this generality, a minimal use of MOSDEX is quite feasible. In particular, if the application uses an algebraic modeling language, the modeling artifacts of MOSDEX are not required, and simply encoding the data as Data objects will suffice. Furthermore, if a database management system is not part of the application's software stack, instance form tables will suffice, provided that the problem is not too large or that the MOSDEX file is generated programmatically. Other advanced features, such as modular structures or expression graphs, can be ignored if one is solving a standard linear, quadratic, or mixed integer program. The value of MOSDEX is that it spans very simple, straightforward instances of mathematical optimization problems while supporting advanced capabilities within the same framework.

2. JSON and Primitive Types

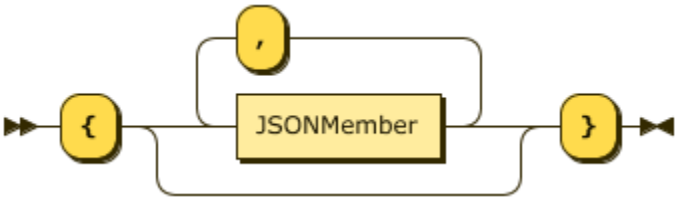
MOSDEX is a derivative of JavaScript Object Notation (JSON), and therefore, MOSDEX files adhere to the JSON standard (see http://json.org/). In fact, MOSDEX is specified using a standard JSON Schema (see http://json-schema.org/).

JSON has three fundamental elements: *objects, arrays,* and *primitives*.

An object is an unordered list of *key* : *element* pairs, or *members*, where each key, or *field name*, is a string; it is enclosed within curly braces, { and }, and a member's key and element are separated by a colon, with the object's members separated by commas. Keys must be unique within an object. In MOSDEX 2.0, the field name is always a *keyword*. These tokens are discussed in a subsequent section.

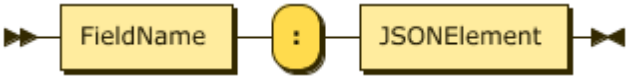
(Syntax Diagrams made with Railroad Diagram Generator by Gunther Rademacher at https://bottlecaps.de/rr/ui.)

*Figure 1**: JSONObject:***



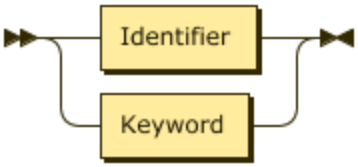
JSONObject ::= '{' (JSONMember (',' JSONMember)*)? '}'

*Figure 2: **JSONMember:***



JSONMember ::= FieldName ':' JSONElement

*Figure 3: **FieldName:***

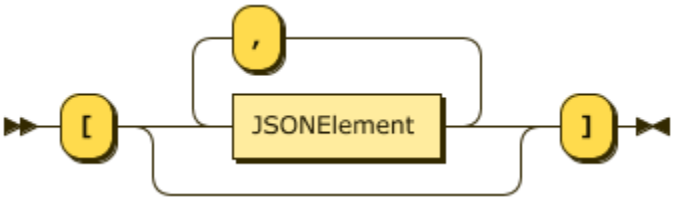


FieldName ::= Identifier | Keyword

Note: In MOSDEX 2.0, field names are always keywords.

An array is an ordered list of elements; it is enclosed in square brackets, [and], and the elements are separated by commas. Array elements may be of mixed types.

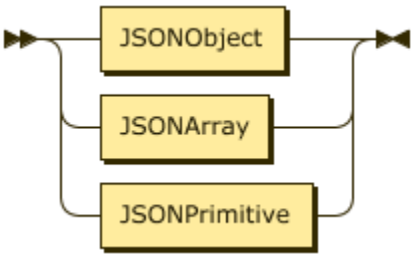
*Figure 4: **JSONArray:***



JSONArray ::= '[' (JSONElement (',' JSONElement)*)? ']'

In both Objects and Arrays, an element is either a Primitive or another Object or Array.

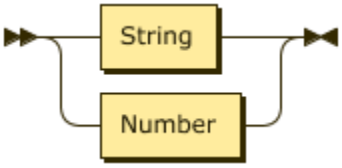
Figure 5: ***JSONElement****:*



JSONElement ::= JSONObject | JSONArray| JSONPrimitive

JSON supports the following Primitive types: *strings* of Unicode characters enclosed in double quotes, decimal *integers*, decimal floating point numbers (MOSDEX assumes these are *doubles*) written with or without an exponent, *boolean* (true or false), and *null*. Boolean is not used in MOSDEX.

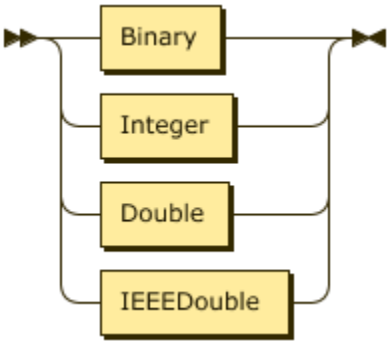
Figure 6: ***JSONPrimitive:***



JSONPrimitive ::= String | Number

Additionally, MOSDEX allows two other number types. *Binary* (zero or one) values are a subtype of integers used in mathematical expressions, especially as decision variables in optimization models; note that the binary type differs from the boolean type, which is used in logical expressions. *IEEE Doubles* are represented as strings of hexadecimal digits, as well as the special values +/- infinity and NaN, according to the IEEE 754 standard; IEEE doubles are represented as JSON strings, since neither standard JSON parsers nor most databases support this type; the MOSDEX parser converts them to ordinary doubles, and most programming languages support this conversion. Because optimization solvers use this format, it provides the most precise way to exchange numerical data with a solver.

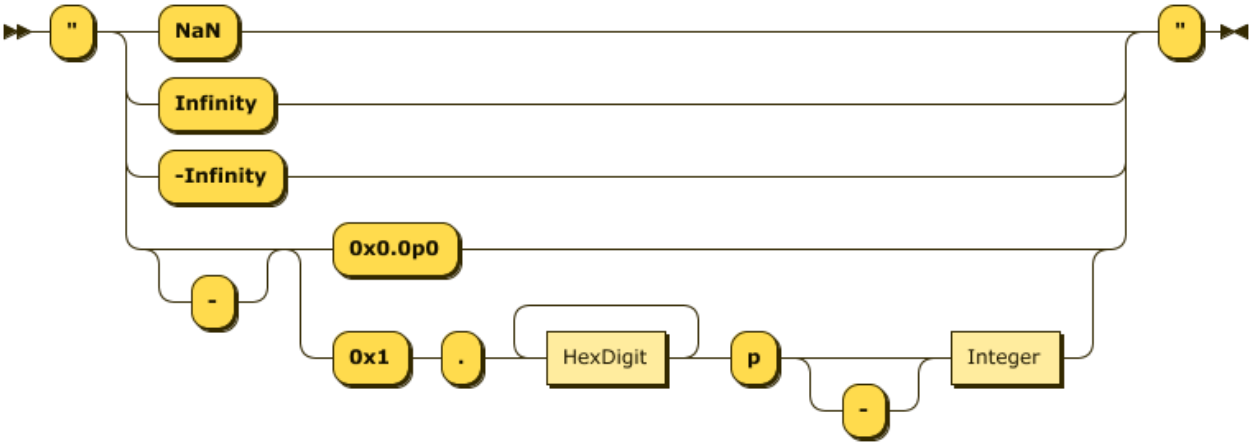
Figure 7: ***Number:***



Number ::= Binary | Integer | Double | IEEEDouble

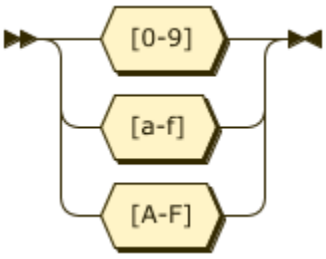
Note: MOSDEX 2.0 does not use Binary numbers.

Figure 8: ***IEEEDouble:***



IEEEDouble ::= '"' ('NaN' | 'Infinity' | '-Infinity' | '-'? ('0x0.0p0' | '0x1' '.' HexDigit+ 'p' '-'? Integer)) '"'

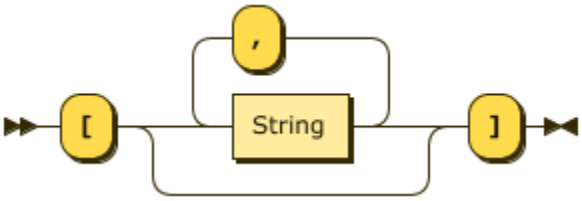
Figure 9: ***HexDigit:***



HexDigit ::= [0-9a-fA-F]

JSON and MOSDEX do not support multiline string literals; however, they both support arrays of strings, which serve the same purpose. A MOSDEX parser will generally convert an array of strings into an ordinary string by concatenation, usually embedding a context appropriate separator, such as newline, between the components.

Figure 10: ***ArrayOfStrings****:*



ArrayOfStrings ::= '[' (String (',' String)*)? ']'

JSON generally ignores white space between tokens.

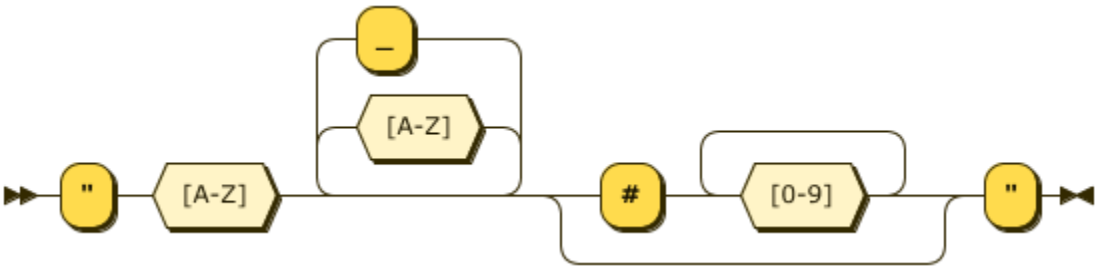
The JSON standard does not support comments, but many parsers support them. MOSDEX allows two kinds of C or Java-style comments:

- // ignores the rest of a line, and
- the pair /* and */ ignores everything in between including line breaks.

3. Keywords and Identifiers

MOSDEX 2.0 field names in JSON Objects are always *keywords*, and no keyword is repeated in a JSON Object. A keyword generally follows the rules for identifiers in JavaScript; since it is a JSON String, a keyword is always surrounded by double quotes. Usually a keyword denotes the type of the item which follows it; for example, the keyword TABLES denotes the JSON Array of Table Objects in a Module. Where a keyword names an element, it is also used as the name of the corresponding parsing rule. MOSDEX keywords are strings written in all capital letters and can include underscore and hyphen characters. However, in certain circumstances, the number sign '#' and digits are allowed. One such circumstance arises when multiple SQL Directives of the same kind appear in a single Query Object; for example, where there are multiple joins, they are denoted "JOIN#1", "JOIN#2", etc., in order to avoid having multiple fields with the same key in the Query Object.

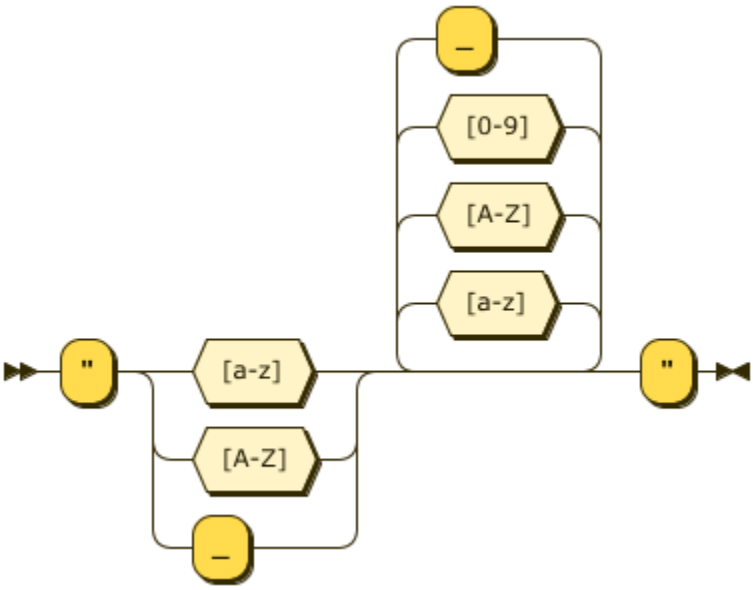
Figure 11: ***Keyword:***



Keyword ::= '"' [A-Z] [A-Z_]* ('#' [0-9]+)? '"'

Identifiers that are not keywords can occur in MOSDEX as the names of Fields in a Schema, in the Predicate of a Query Clause, and in other places in which MOSDEX does not prescribe a particular semantic. However, even in such places, the requirements of a target for the MOSDEX data, such as a solver or database, may dictate the semantic meaning of a particular identifier instance. For example, as discussed below, the names of some of the fields of the Schema for a Variable artifact, such as "Lower_Bound," may be required for interaction with a target solver's API. Since it is a JSON String, an identifier is always surrounded by double quotes

Figure 12: ***Identifier:***



Identifier ::= '"' [a-zA-Z_] [a-zA-Z0-9_]* '"'

4. File and Modules

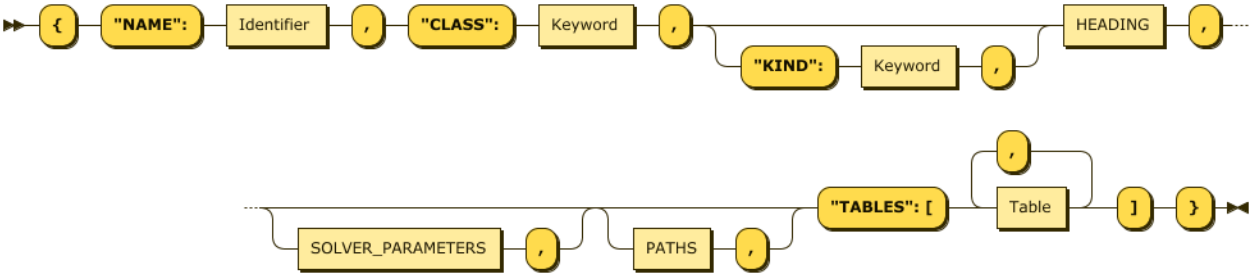
At the highest level, a MOSDEX File (we use the term *file* generically for any input source) is a JSON Object consisting of a collection of Module Objects. It is useful to think of a Module as a self-contained presentation of the data and the variables, constraints, objectives and terms (collectively the *modeling artifacts*) for a mathematical optimization problem; however, MOSDEX can actually accommodate more general structures. When model/data separation is used, the data may be presented in one or more separate Modules without modeling artifacts. A decomposition of a mathematical optimization problem can be presented as a collection of MOSDEX Modules linked in a modular structure. Although not addressed in this document, a syntax to include a simulation model could be defined within a MOSDEX Module to accommodate a combined optimization/simulation model. The term Module was chosen, as opposed to Model or Instance, because a variety of specifications are permitted in MOSDEX; a model may or may not be present in a MOSDEX File, and data tables and modeling artifacts may be presented in either query or instance form. The MOSDEX File also includes a mandatory SYNTAX field which specifies the version of the JSON Schema for MOSDEX applicable to the File.

Figure 13: ***MOSDEX_File:***

MosdexFile ::= '{' '"SYNTAX":' String ',' '"MODULES":' '[' Module (',' Module)* ']' '}'

A MOSDEX Module is a collection of Table Objects representing data and modeling artifacts, as well as several other, mostly optional Objects. At present, MOSDEX recognizes two keywords for a Module's CLASS field (although others could be added): DATA denotes a Module with only data Tables, while MODEL denotes a Module with modeling artifact Tables and possibly data Tables as well. It is recommended that a MODEL Module include all the modeling artifacts that constitute a well-formed optimization model valid for a solver; internally, the MOSDEX Reference Architecture assigns a separate solver instance to each MODEL Module. The keywords allowed for a Module's KIND field depend on the CLASS. At present MOSDEX recognizes two keywords for a DATA Module, INPUT or OUTPUT, although neither is mandatory, while no KIND keywords are recognized for a MODEL Module; other KIND keywords could be added.

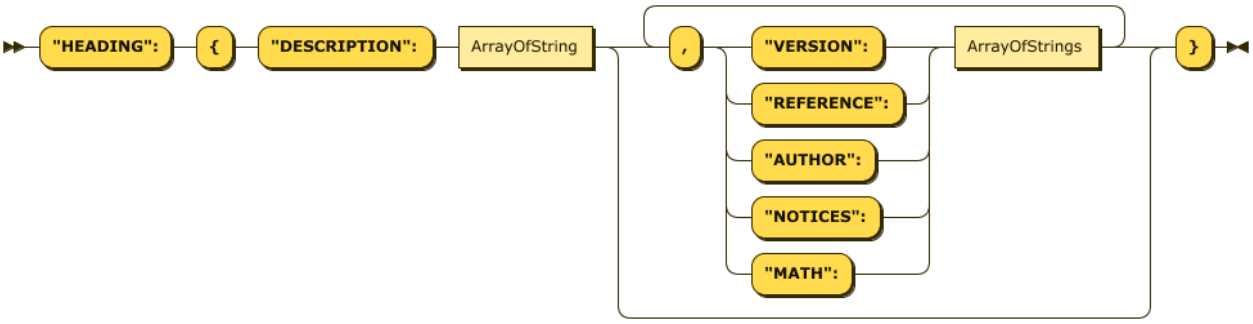
Figure 14: ***Module:***



Module ::= '{' '"NAME":' Identifier ',' '"CLASS":' Keyword ',' ('"KIND":' Keyword ',')? HEADING ',' (SOLVER_PARAMETERS ',')? (PATHS ',')? '"TABLES": [' Table (',' Table)* ']' '}'

A Heading collects information about a Module or Table; it is mandatory for a Module and optional for a Table. Of the information in the Heading, only the Description is mandatory. The other items in the Heading are a Version number, a Reference, which is typically a link or a citation to the source of the problem, the Author's name and contact information, legal Notices, such as a copyright or a license, and a mathematical expression of the problem or modeling artifact, in any suitable language, such as OPL, AMPL, or GAMS. Note that the information in the Heading, including the Math expression, are not otherwise processed by MOSDEX.

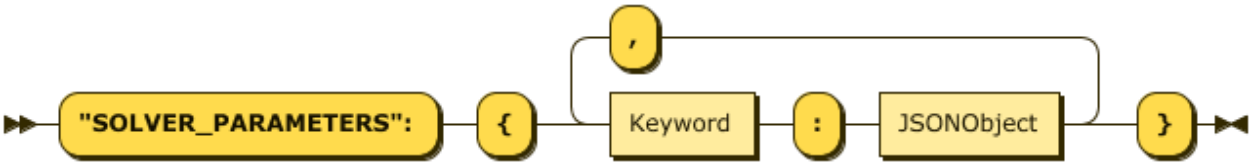
Figure 15: ***HEADING:***



HEADING ::= '"HEADING":' '{' '"DESCRIPTION":' ArrayOfStrings (',' ('"VERSION":' | '"REFERENCE":' | '"AUTHOR":' | '"NOTICES":' | '"MATH":') ArrayOfStrings)* '}'

Solver Parameters is an optional object, which if present in a Module, provides solver-specific parameters in the form of JSON Object *key*: *value* pairs. MOSDEX does not prescribe the parameter names nor their allowed values, as they can vary among solvers, but it passes them to the solver without parsing them. Multiple JSON objects within Solver Parameters are permitted, each tagged with a keyword denoting the target solver.

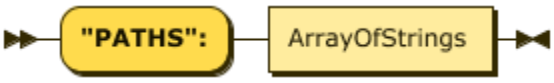
Figure 16: ***SOLVER_PARAMETERS****:*



SOLVER_PARAMETERS ::= '"SOLVER_PARAMETERS":' '{' Keyword ':' JSONObject (',' Keyword ':' JSONObject)* '}'

Paths, also an optional object, provides a means to reference external files in a MOSDEX File. The Paths object specifies a list of variable names that can be used in string substitution within a MOSDEX Module (see the section on Modular Structure below). The calling program of MOSDEX will pass the actual values of these name variables to the MOSDEX parser. Usually the actual values will be paths or URLs appropriate for the operating system running the application that uses MOSDEX. Thus, for instance, the data for an optimization problem could be supplied in a separate MOSDEX File that is connected at run time.

Figure 17: ***PATHS:***



PATHS ::= '"PATHS":' ArrayOfStrings

5. Tables, Instances and Queries

Table is the base class for data, modeling artifacts, and other MOSDEX objects (except for, notably, Module). Conceptually, a Table is a two-dimensional object with a fixed number of columns, or *fields*, and an indefinite number of rows, or *records*; think of a table in a relational database. Data and the modeling artifacts are subclasses of Table.

Figure 18: ***Table:***

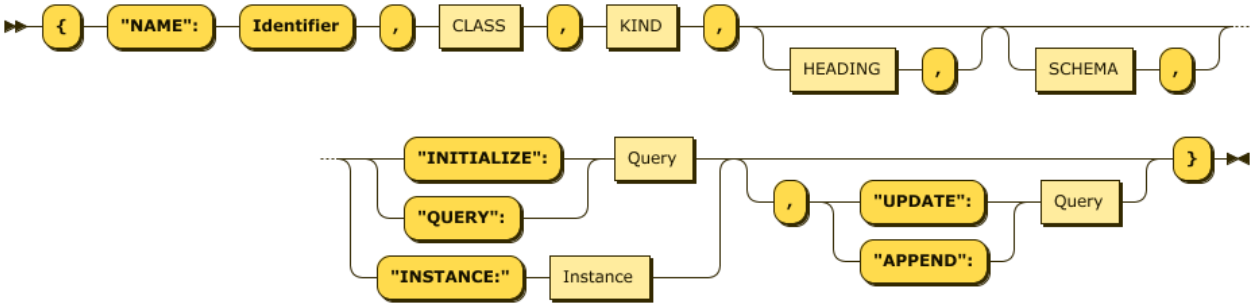
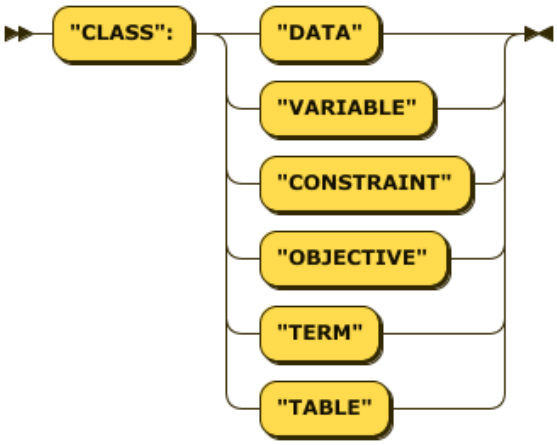


Table ::= '{' '"NAME":' 'Identifier' ',' CLASS ',' KIND ',' (HEADING ',')? (SCHEMA ',')? (('"INITIALIZE":' | '"QUERY":') Query | '"INSTANCE:"' Instance) (',' ('"UPDATE":' | '"APPEND":') Query)? '}'

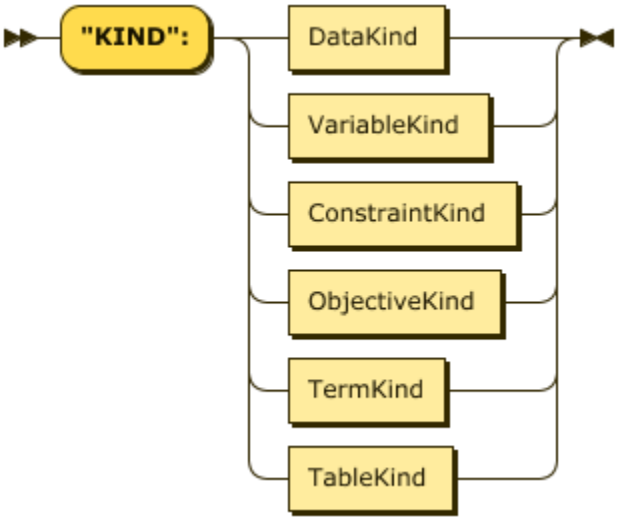
A Table has a CLASS field that identifies the type of the object it embodies. Since the MOSDEX parser reads a Table before knowing its Class, the Class does not affect how the Table is parsed. However, once a Table is parsed, MOSDEX will check its validity, so for instance, certain fields must be present in the Schema of a Variable while others must be present for a Constraint. In addition to specific classes for data and modeling artifacts, MOSDEX permits a general TABLE class with minimal structure that could be used for future expansion. A Table also has a KIND field, which depends on its Class, to specify how it is realized in the application. The base Table permits general strings as Kinds, but the others each have a specific enumeration of allowed Kinds.

Figure 19: ***CLASS:***



CLASS ::= '"CLASS":' ('"DATA"' | '"VARIABLE"' | '"CONSTRAINT"' | '"OBJECTIVE"' | '"TERM"' | '"TABLE"')

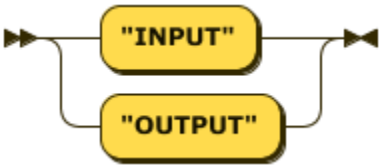
Figure 20: ***KIND:***



KIND ::= '"KIND":' (DataKind | VariableKind | ConstraintKind | ObjectiveKind | TermKind | TableKind)

The TABLE class have no specific Kind or Schema, while the DATA class has Kind of INPUT or OUTPUT. Each of the modeling artifacts, VARIABLE, CONSTRAINT, OBJECTIVE, and TERM does have its own Kind and Schema, as discussed below.

Figure 21: DataKind



DataKind ::= '"INPUT"' | '"OUTPUT"'

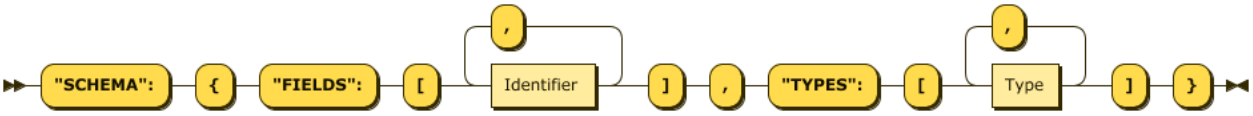
A Table may have an optional HEADING field for documentation. Headings are discussed under the Modules section.

Every Table has a *Schema* which defines the names and types of its fields. Schema Fields must have JSON Primitive data Types; neither JSON Objects nor JSON Arrays are permitted as Schema elements. The Schema of a Table may be specified explicitly by a SCHEMA object; however, under certain circumstances, an explicit Schema is not required, although one may be supplied for clarity. The circumstances when a Schema is optional include:

1. The Table is specified by an SQL Query (either a QUERY or INITIALIZE object), in which case its Schema is generated by the Query as discussed below.
2. The Table is updated as part of an iterative solver process (either an UPDATE or APPEND object) since the Schema is established when the Table is initialized.

A SCHEMA object consists of two parallel JSON Arrays, one containing the Field names and the other their Types. Both must have the same number of elements. Additional parallel arrays may be added, to denote, for example, whether or not the field is a key.

Figure 22: ***SCHEMA:***

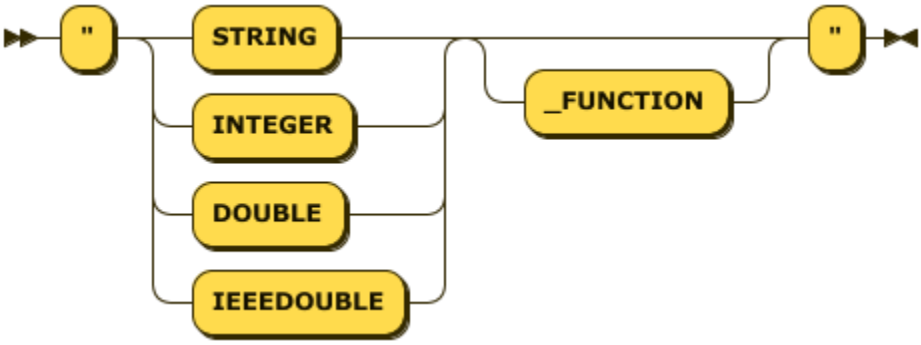


SCHEMA ::= '"SCHEMA":' '{' '"FIELDS":' '[' Identifier (',' Identifier)* ']' ',' '"TYPES":' '[' Type (',' Type)* ']' '}'

The Type keywords distinguish the data types supported by MOSDEX. Those with the suffix "_FUNCTION" denote *function calls* that get data from the solver. These function calls are discussed further in the section on Interactions with Solvers, below.

.

Figure 23: ***Type:***

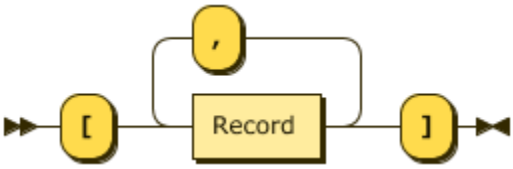


Type ::= '"' ('STRING' | 'INTEGER' | 'DOUBLE' | 'IEEEDOUBLE') '_FUNCTION'? '"'

A Table may be specified in either of two forms, *Instance* or *Query*. In Instance form, the Table directly specifies the data that it encompasses. In Query form, the Table uses SQL to specify how the data are constructed from other Tables or from an external database. Both forms of Tables may coexist in a MOSDEX Module, but an individual Table must have one form or the other. (Note: in a modular model that allows an iterative algorithm to apply Update or Append Queries on successive iterations, a Table may be Initialized by either a Query or an Instance.)

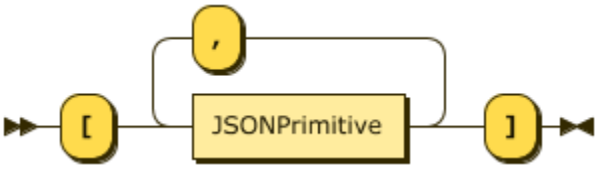
An Instance is a JSON Array of Records, each of which is itself a JSON Array that contains actual data in the form of JSON Primitive value (JSON Objects and Arrays are not permitted as elements in MOSDEX Records). In an Instance object, the individual items in each Record are unlabeled and can only be parsed using the Table's Schema.

Figure 24: ***Instance:***



Instance ::= '[' Record (',' Record)* ']'

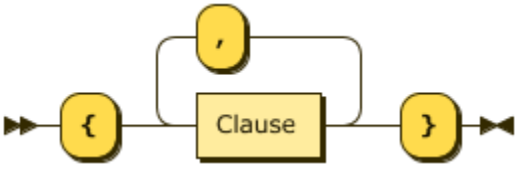
Figure 25: ***Record:***



Record ::= '[' JSONPrimitive (',' JSONPrimitive)* ']'

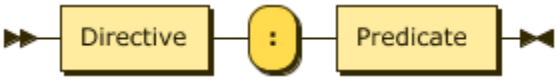
As an alternative to Instance form, a MOSDEX Table can also be specified in Query form, which uses SQL to specify how the data are constructed from other Tables or from an external database. Using Query form Tables presupposes that a database engine is present in the software stack that supports the optimization application; MOSDEX does not supply database management software. A Query in MOSDEX is a JSON Object of Clause members, each of which has a Directive as its key and a Predicate as its value. The Directive is usually an SQL command, such as “SELECT” or “FROM”; the Predicate is the argument list of the command, such as a list of fields in a Select Query or a list of Tables in a From Clause. A Predicate can also contain a subquery. This formulation affords considerable flexibility in specifying MOSDEX Queries. The general rule is that the Directive and Predicate must be interpreted directly as valid SQL by the database engine. It is recommended that MOSDEX Queries adhere as closely as possible to ANSI standard SQL, as many database engines support it. However, not all database management systems natively support SQL of any flavor (e.g. Python Pandas), so using a Query object in MOSDEX breaks an SQL query into its component Clauses in order to enable adapting to such systems. MOSDEX does not actually parse SQL, leaving that task to the database engine; however, MOSDEX does read the From, Join, and Union Clauses to identify dependencies among the Tables.

Figure 26: ***Query:***



Query ::= '{' Clause (',' Clause)* ‘}'

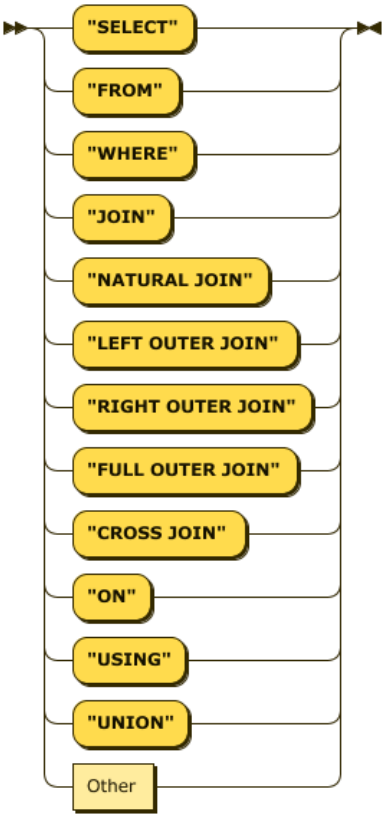
Figure 27: ***Clause:***



Clause ::= Directive ':' Predicate

Directives are SQL commands, the most common of which used in MOSDEX are listed below. However, since parsing SQL is one function of a database engine, MOSDEX can accommodate any other directive that the database supports. Note that a particular MOSDEX Query may require multiple Directives of the same name, especially where multiple joins are used. Since JSON Objects must have unique keys (i.e. field names), multiple Clauses with the same Directive will be flagged as an error by most JSON parsers. To avoid that concern, MOSDEX permits appending a number sign (#) followed by an integer to a keyword to create unique field name. For example, JOIN#1 … USING#1(…), JOIN#2… USING#2(…), etc. Internally, the MOSDEX parser will strip the suffix before sending the Query to the SQL parser.

Figure 28: Directive



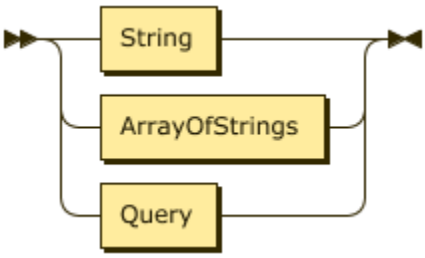
Directive ::= ('"SELECT"' | '"FROM"' | '"WHERE"' | '"JOIN"' | '"NATURAL JOIN"' | '"LEFT OUTER JOIN"' | '"RIGHT OUTER JOIN"' | '"FULL OUTER JOIN"' | '"CROSS JOIN"' | '"ON"' | '"USING"' | '"UNION"' | Other)

A Predicate generally consists of a string or an array of strings, depending on the associated Directive. Some Directives (e.g. FROM) can accept a subquery as a predicate; MOSDEX does not attempt untangle

such complex Queries, except to find dependencies on other Tables, but the general rule is that a parsed MOSDEX Query must be acceptable to the database engine as valid SQL.

There is one aspect of the MOSDEX syntax for Predicates that bears further explanation, however. Query-form Tables in MOSDEX do not require an explicit SCHEMA object; however, the type of each field in the result of the Query needs to be defined nonetheless, since SQL may not automatically assign a data item to a relevant type for the purpose of optimization modeling (e.g. SQL could assign a number as a Big Decimal, when the optimization model requires a Double). So MOSDEX specifies the schema as part of the Query. Specifically, the fields are named in the AS part of each item of a the SELECT Clause Predicate, and their types are specified following a "--" symbol (double hyphen), which SQL reads as a comment (as distinguished from comments in JSON, which are not part of its standard).

Figure 29: Predicate



Predicate ::= (String | ArrayOfStrings | Query)

6. Modeling Artifacts and their Standard Schemas

The MOSDEX modeling artifact classes, subclasses of Table, are Variable, Constraint, Objective, and Term; they can be given in either instance or query form. While a MOSDEX Data object may have any reasonable schema, each type of MOSDEX modeling artifact has a *standard Schema* that enables realizing the object in a solver's API. While there is no standard for a solver's modeling API, most solvers have similar APIs, so the MOSDEX standard Schemas should suffice in most cases. Query form modeling artifacts do not have to specify a Schema object explicitly, but rather the Schema is inferred from the Query that generates it. Thus, each Query should mirror the standard Schema of the given modeling artifact.

While each type of modeling artifact has unique fields, there are a number of commonalities. First, in most cases, a modeling artifact represents an entire family, or *block*, rather than a singular occurrence, because most large optimization problems have a block structure. That is, for instance, a network model has nodes and arcs, with flow variables defined over the arcs and balance constraints defined over the nodes. In traditional mathematical notation, the flow variables would have an index, or *key*, representing the arcs, and the balance constraints would have a key representing the nodes. Algebraic modeling languages realize this notation in their syntax. In MOSDEX, the modeling artifacts represent these indexes (which can have multiple components) by a set of *key fields* drawn from one or more Tables representing the index sets. The key fields and their corresponding index sets differ according to the particulars of the model, but the Schemas of the modeling artifacts include them.

Second, the standard form, or *tableau*, of an optimization problem that is submitted to a solver has a two-dimensional structure, with columns representing variables and rows representing constraints. An important challenge is how to encode the multidimensional keys of a model into a two-dimensional tableau; automating this encoding is an advantage of adopting an algebraic modeling language. MOSDEX does not explicitly define how this encoding is to be done (although as discussed in the section on Interactions with Solvers below, MOSDEX suggests a method), but it does specify that the Schema of a modeling artifact must contain fields corresponding to the tableau entry for each object's key. MOSDEX does not parse the components of an encoding.

Third, the solver will compute a solution (or more than one) for the problem, and a modeling artifact's Schema should give the solver fields in which to report the solution and auxiliary information, such as the value of each variable, its reduced cost and basis status, the dual value (for continuous problems), and slack for each constraint, the value of the objective function, and so on. The solution fields are initially set with a type representing a *function call*, which is populated with a value in a post-processing phase of the solution process.

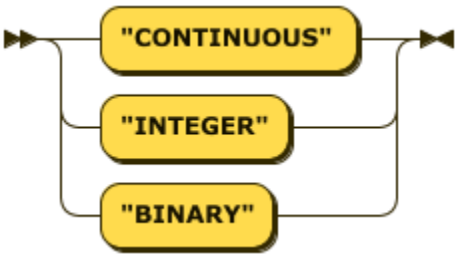
Finally, since the solver will read the MOSDEX modeling artifacts to create its own internal representation of the Module, each record in a Table should contain the information required to construct the internal representation. While each solver has its own syntax, there is enough commonality among most solvers to establish a standard schema for each type of modeling artifact in MOSDEX.

Figure 30: ***VariableSchema:***

FIELDS/Description	**Token Type**	**TYPES**	**Multiplicity**	**Default for KIND**
"Name"	Literal	STRING		
key field name	Identifier	STRING or INTEGER	*	
"Column"	Literal	STRING or INTEGER		
"LowerBound"	Literal	Number	?	0 for BINARY or INTEGER 0.0 for CONTIUOUS
"UpperBound"	Literal	Number	?	1 for BINARY 'infinity' for INTEGER or CONTIUOUS
solution value	Identifier	Keyword_FUNCTION	*	

VariableSchema ::= '"FIELDS":' '[' '"Name"' ',' (KeyField ',')* '"Column"' ',' ('"LowerBound"' ',')? ('"UpperBound"' ',')? SolutionValue* ']'

*Figure 31: **VariableKind:***



VariableKind ::= '"CONTINUOUS"' | '"INTEGER"' | '"BINARY"'

*Figure 32: **ConstraintSchema:***

FIELDS/Description	**Token Type**	**TYPES**	**Multiplicity**	**Default for KIND**
“Name”		STRING		
key field name	Identifier	STRING or INTEGER	*	
“Row”	Literal	STRING or INTEGER		
“Sense”	Literal	“GE”, “EQ”, “LE”*		
“RHS”	Literal	Number		
solution value	Identifier	Keyword_FUNCTION	*	

* Also allowed for “Sense”: “GE”, “=>”, “<=”,“EQ”, “==”,“LE”, “<=”, “=<”

ConstraintSchema ::= '"FIELDS":' '[' '"Name"' ',' (KeyField ',')* '"Row"' ',' '"Sense"' ',' '"RHS"' ','
SolutionValue* ']'

*Figure 33: **ObjectiveSchema:***

FIELDS/Description	**Token Type**	**TYPES**	**Multiplicity**	**Default for KIND**
“Name”	Literal	STRING		
key field name	Identifier	STRING or INTEGER	*	
“Row”	Literal	STRING or INTEGER		
“Sense”	Literal	“Min”, “Max”*		
“Constant”	Literal	Number		0.0
solution value	Identifier	Keyword_FUNCTION	*	

* Also allowed for “Sense”: "Minimize", "MINIMIZE", "Min", "MIN", "Maximize", "MAXIMIZE", "Max",
"MAX"

ObjectiveSchema::= '"FIELDS":' '[' '"Name"' ',' (KeyField ',')* '"Row"' ',' '"Sense"' '"Constant"' ','
SolutionValue* ']'

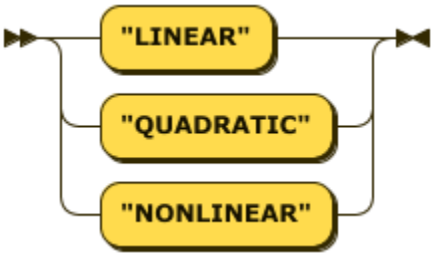
*Figure 34: **TermSchema:***

FIELDS/Description	**Token Type**	**TYPES**	**Multiplicity**	**Default for KIND**
“Row”	Literal	STRING or INTEGER		
“Column”	Literal	STRING or INTEGER		
“Column2”	Literal	STRING or INTEGER	?	

“Coefficient”	Literal	Number		

TermSchema ::= '"FIELDS":' '[' '"Row"' ',' '"Column"' ',' ('"Column2"' ',')? '"Coefficient"' ']'

Figure 35: ***TermKind:***



TermKind ::= '"LINEAR"'| '"QUADRATIC"' | '"NONLINEAR"'

The Term schema permits one or more column index fields, representing the possibility that it could involve more than one variable, to enable MOSDEX to accommodate quadratic and other nonlinear optimization problems.

7. Interactions with Solvers

As a standard for data exchange, MOSDEX is designed to interoperate with many different mathematical optimization solvers in many different programming languages. Its design therefore necessarily abstracts the features of the particular solvers, so that each solver will need a set of *bridges* between MOSDEX and its own API. Because most of the solvers of interest have similar APIs, the software that creates these bridges, called a *solver factory*, will usually be fairly similar and straightforward to write, especially because MOSDEX handles much of their structure already. Generally, many solvers already have a *modeling API*, a set of classes that are used to construct the internal representations of the variables, constraints, and objectives of an optimization problem. Since MOSDEX has similar objects, the MOSDEX parser would need to call the factory to map its modeling artifacts onto those of the target solver. Some solvers have a low level interface that uses a tableau (row and column) representation of an optimization problem, either as its main interface or as a supplement to the modeling interface. With a fairly straightforward factory, MOSDEX can accommodate a tableau representation as well.

In either case, modeling API or tableau, a critical decision is how to encode the rows and columns, as discussed in the section on MOSDEX’s modeling artifacts and their schemas. MOSDEX does not specify how this encoding is done, to leave some freedom to adapt to a particular solver. The example problems that have been developed to guide the syntax formulation have used strings of the form

artifactName_key1_...keyN

where the artifactct name is the name of the variable, constraint, or objective, and the keys are the values of the key fields of the particular object. This form has the advantage of providing readable column and row identifiers for debugging purposes. However, some solvers many not allow strings or may limit the length of strings used for that purpose. As an alternative, MOSDEX users might want to consider assigning consecutive non-negative integers to the rows and columns. In addition to the

difficulty of interpreting the codes, counting the items in large data set is problematic. However, if necessary, MOSDEX can accommodate either strings or integers to encode rows and columns.

MOSDEX also needs to capture and structure the solution information from the solver into its output Tables. As discussed in the section on modeling artifacts and their schemas, each artifact's Schema can define fields to be populated by the solver. For variables, typically these fields would include its optimal value, and possibly its reduced cost and basis status. For constraints, they could include its dual value (for continuous problems) and slack. For objectives, they would include the optimal value.

MOSDEX defines a set of types specifically for this purpose. The Type keywords with the suffix "_FUNCTION" denote function calls that get data from the solver. The value assigned to each such field, either in an Instance or in a Query, is a *function call* string of the form

"functionName(argumentField1, ...argumentFieldN)"

where the **argumentFields** are the names of the fields in the current record where the arguments are found (usually, there is only one argument field, a column or a row, in a function call).

The **functionName** links to a method of the solver's API that retrieves the relevant value from its solution. After the solve, MOSDEX creates a solution output table from the modeling artifact, in which that value is substituted for the function call string, and it replaces the function type with the corresponding return type in the output table's schema.

8. Conclusion

To be written

Appendix 1: MOSDEX Examples

The following example problems have been used in creating this MOSDEX syntax and should be read along with this specification document:

Simple linear program – https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/volsay_2-0.json

Network (instance form) – https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/net1b_2-1.json

Network (query form) – https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/net1a_2-1.json

LP with lagged variables – https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/sailco_2-0.json

Warehouse location (MIP) – https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/warehousing_2-0.json

Traffic Network (QP) – https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/trafficNetworkQP_2-0.json

Appendix 2: MOSDEX Parsing Rules Compilation

The JSON Schema for MOSDEX is found in the following document: https://github.com/JeremyBloom/MOSDEX-Reference-Architecture/blob/main/MOSDEXSchemaV2-0.json

The equivalent EBNF specification is as follows:

```
1. JSONObject::= '{' (JSONMember (',' JSONMember)*)? '}'

2. JSONMember::= FieldName ':' JSONElement

3. FieldName::= ('"' Identifier '"')  | ('"' Keyword '"')

4. JSONArray::= '[' (JSONElement (',' JSONElement)*)? ']'

5. JSONElement::= JSONObject | JSONArray |JSONPrimitive

6. JSONPrimitive::= String | Number

7. Number::= Binary | Integer | Double | IEEEDouble

8. IEEEDouble::= '"' ('NaN' | 'Infinity' | '-Infinity' | ('-')?'0x0.0p0' |      (('-
   ')? '0x1' '.' (HexDigit)+ 'p' ('-')? Integer)) '"'

9. HexDigit::= [0-9a-fA-F]

10. ArrayOfStrings::= '[' (String (',' String)*)? ']'

11. Keyword ::= '"' [A-Z] [A-Z_]* ( '#' [0-9]+ )? '"'

12. Identifier::= ([a-z]|[A-Z]|'_') ([a-z]|[A-Z]|[0-9]|'_')*

13. MosdexFile ::= '{' '"SYNTAX":' String ',' '"MODULES":' '[' Module ( ',' Module )*
    ']' '}'

14. Module ::= '{' '"NAME":' Identifier ',' '"CLASS":' Keyword ',' ( '"KIND":' Keyword
    ',' )? HEADING ',' ( SOLVER_PARAMETERS ',' )? ( PATHS ',' )? '"TABLES": [' Table
    (',' Table )* ']' '}'

15. HEADING ::= '"HEADING":' '{' '"DESCRIPTION":' ArrayOfStrings ( ',' ( '"VERSION":' |
    '"REFERENCE":' | '"AUTHOR":' | '"NOTICES":' | '"MATH":' ) ArrayOfStrings )*
    '}'HEADING::= '"HEADING"' ':' '{' DESCRIPTION (',' ProblemInformation)*  '}'

16. SOLVER_PARAMETERS ::= '"SOLVER_PARAMETERS":' '{' Keyword ':' JSONObject ( ','
    Keyword ':' JSONObject )* '}'SOLVER_PARAMETERS::= '"SOLVER_PARAMETERS"' ':' '{'
    Identifier ':' JSONObject (',' Identifier ':' JSONObject)* '}'

17. PATHS::= '"PATHS"' ':' ArrayOfStrings

18. Table ::= '{' '"NAME":' 'Identifier' ',' CLASS ',' KIND ',' ( HEADING ',' )? (
    SCHEMA ',' )? ( ( '"INITIALIZE":' | '"QUERY":' ) Query | '"INSTANCE:"' Instance ) (
    ',' ('"UPDATE":' | '"APPEND":' ) Query )? '}'

19. CLASS::= '"CLASS"' ':' ('"DATA"' | '"VARIABLE"' | '"CONSTRAINT"' | '"OBJECTIVE"' |
    '"TERM"' | '"TABLE"')
```

20. KIND ::= '"KIND":' (DataKind | VariableKind | ConstraintKind | ObjectiveKind | TermKind | TableKind)

21. SCHEMA ::= '"SCHEMA":' '{' '"FIELDS":' '[' Identifier (',' Identifier)* ']' ',' '"TYPES":' '[' Type (',' Type)* ']' '}'

22. Type ::= '"' ('STRING' | 'INTEGER' | 'DOUBLE' | 'IEEEDOUBLE') '_FUNCTION'? '"'

23. Instance ::= '[' Record (',' Record)* ']'

24. Record::= '[' JSONPrimitive (',' JSONPrimitive)* ']'

25. Query ::= '{' Clause (',' Clause)* `}'

26. Clause ::= Directive ':' Predicate

27. Directive ::= ('"SELECT"' | '"FROM"' | '"WHERE"' | '"JOIN"' | '"NATURAL JOIN"' | '"LEFT OUTER JOIN"' | '"RIGHT OUTER JOIN"' | '"FULL OUTER JOIN"' | '"CROSS JOIN"' | '"ON"' | '"USING"' | '"UNION"' | Other)

28. Predicate ::= (String | ArrayOfStrings | Query)

29. DataKind::= ('"INPUT"' | '"OUTPUT"')

30. VariableSchema ::= '"FIELDS":' '[' '"Name"' ',' (KeyField ',')* '"Column"' ',' ('"LowerBound"' ',')? ('"UpperBound"' ',')? SolutionValue* ']'

31. VariableKind ::= '"CONTINUOUS"' | '"INTEGER"' | '"BINARY"'

32. ConstraintSchema ::= '"FIELDS":' '[' '"Name"' ',' (KeyField ',')* '"Row"' ',' '"Sense"' ',' '"RHS"' ',' SolutionValue* ']'

33. ObjectiveSchema::= '"FIELDS":' '[' '"Name"' ',' (KeyField ',')* '"Row"' ',' '"Sense"' '"Constant"' ',' SolutionValue* ']'

34. TermSchema ::= '"FIELDS":' '[' '"Row"' ',' '"Column"' ',' ('"Column2"' ',')? '"Coefficient"' ']'

35. TermKind ::= '"LINEAR"'| '"QUADRATIC"' | '"NONLINEAR"'

