**MOSDEX Syntax**

**Version 1-2**

Dr. Jeremy A. Bloom  
[jeremyblmca@gmail.com](mailto:jeremyblmca@gmail.com)  
September 1, 2019

**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 objects, 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 objects.
* 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 *Problems*, 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 objects, 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 objects are largely fixed by the requirements of an optimization solver. A Table can have either *instance* or *recipe* form. Instance form tables contain data while recipe 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 objects 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.

1. **JSON and Primitive Types**

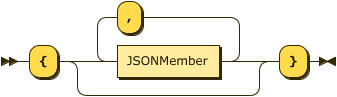
MOSDEX is a derivative of JavaScript Object Notation (JSON), and therefore, MOSDEX files adhere to the JSON standard (see <http://json.org/>). In fact, MOSDEX is specified using a standard JSON Schema (see <http://json-schema.org/>).

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, the field name is always either an *identifier* or 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:**



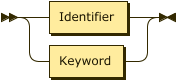
JSONObject ::= '{' ( JSONMember ( ',' JSONMember )\* )? '}'

Figure 2: **JSONMember:**



JSONMember ::= FieldName ':' JSONElement

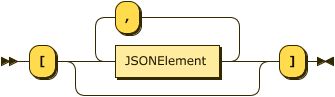
Figure 3: **FieldName:**



FieldName ::= Identifier | Keyword

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.

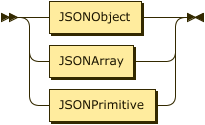
Figure 4: **JSONArray:**



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

In both objects and arrays, the elements are either primitive types or other objects or arrays.

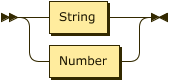
Figure 5: **JSONElement**:



JSONElement ::= JSONObject | JSONArray| JSONPrimitive

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

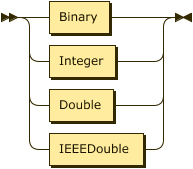
Figure 6: **JSONPrimitive:**



JSONPrimitive ::= String | Number

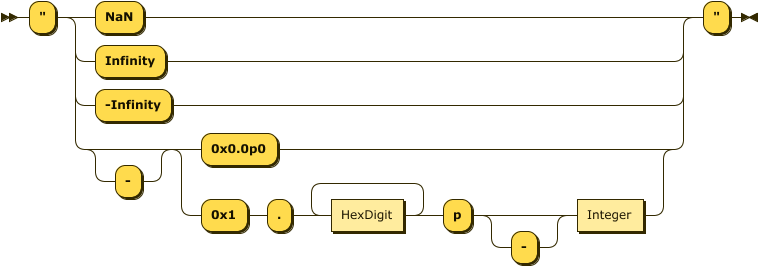
Additionally, MOSDEX allows two other number types. Binary (zero or one) values are a subtype of integers used in mathematical expressions, especially as decision variables in optimization models; note that the binary type differs from the boolean type, which is used in logical expressions. IEEE Doubles 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; MOSDEX parsers should convert 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:**



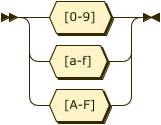
Number ::= Binary | Integer | Double | IEEEDouble

Figure 8: **IEEEDouble:**



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

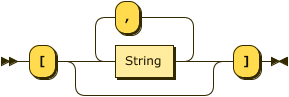
Figure 9: **HexDigit:**



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.

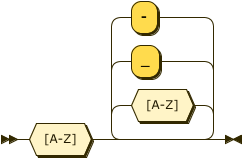
The JSON standard does not support comments, but many parsers support them. MOSDEX allows two kinds of C or Java-style comments:

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

1. **Keywords and Identifiers**

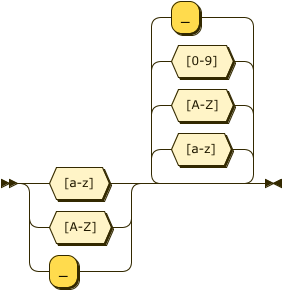
MOSDEX field names in JSON Objects are either *keywords*, assuming that no keyword is repeated in a JSON Object, or they are *identifiers*. An identifier generally follows the rules for identifiers in JavaScript. Usually a keyword denotes the type of the item which follows it. Where a keyword names an 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.

Figure 11: **Keyword:**



Keyword ::= [A-Z] ([A-Z]|'\_'|'-')\*

Figure 12: **Identifier:**



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

In MOSDEX, the element following a field name is frequently, but not always, a JSON Object. If the field name is a keyword, the element is called a *keyword* *object*, and its type is the keyword, which is parsed by the corresponding rule. On the other hand, if the field name is an identifier, the element is called a *named* *object*, and its type must be deduced from its context; the parser cannot read its type before it has been parsed, so all named objects within a MOSDEX object must be parsed by the same rule. (Named objects typically have a CLASS field, but it is only read while the object is being parsed.) For this reason, the only named objects allowed at the top level in a MOSDEX File are Problems, and the only named types allowed at the top level in a Problem are Tables. However, a Table can also contain keyword objects at the top level, as long as only one instance of a particular keyword object appears in a Table.

1. **Problems**

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 named Problem objects. It is useful to think of a Problem as a self-contained presentation of the data and modeling objects 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 Problems without modeling objects. A decomposition of a mathematical optimization problem can be presented as a collection of MOSDEX Problems linked in a modular structure. Although not addressed in this document, a syntax to include a simulation model could be defined within a MOSDEX Problem to accommodate a combined optimization/simulation model. The term Problem 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 objects may be presented in either recipe or instance form. The MOSDEX File includes a mandatory SYNTAX field which specifies the version of the JSON Schema for MOSDEX applicable to the file.

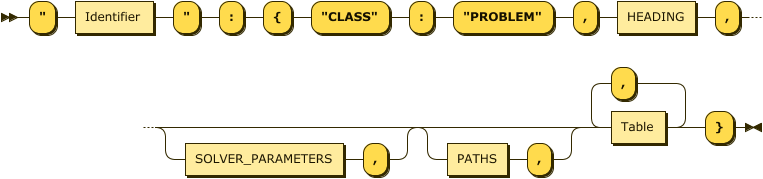
Figure 13: **MOSDEX\_File:**



MosdexFile::= '{' '"' SYNTAX '"' ':' String ( ',' PROBLEM )+ '}'

A MOSDEX Problem is a collection of named Table objects representing data and modeling objects, as well as several, mostly optional keyword objects.

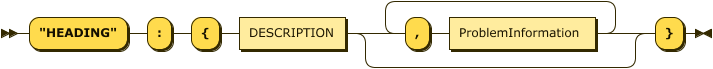
Figure 14: **PROBLEM:**



PROBLEM ::= '"' Identifier '"' ':' '{' '"CLASS"' ':' '"PROBLEM"' ',' HEADING ',' ( SOLVER\_PARAMETERS ',' )?   
( PATHS ',' )? Table ( ',' Table )\* '}'

A Heading collects information about a Problem or Table; it is mandatory for a Problem 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 object, 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:**



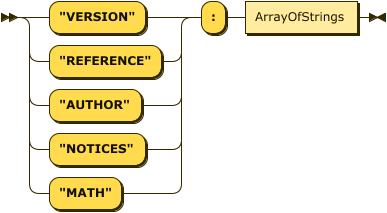
HEADING ::= '"HEADING"' ':' '{' DESCRIPTION ( ',' ProblemInformation )\* '}'

Figure 16: **Description:**



Description ::= '"DESCRIPTION"' ':' ArrayOfStrings

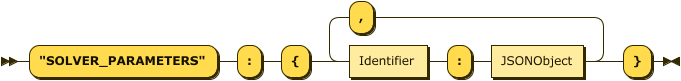
Figure 17: **ProblemInformation:**



ProblemInformation ::= ( '"VERSION"' | '"REFERENCE"' | '"AUTHOR"' | '"NOTICES"' | '"MATH"' ) ':'  
 ArrayOfStrings

Solver Parameters is an optional object, which if present in a Problem, provides solver-specific parameters in the form of *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 named objects within Solver Parameters are permitted, each tagged with an identifier denoting the target solver.

Figure 18: **SOLVER\_PARAMETERS**:



SOLVER\_PARAMETERS ::= '"SOLVER\_PARAMETERS"' ':' '{' Identifier ':' JSONObject   
( ',' Identifier ':' 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 Problem (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 19: **PATHS:**



PATHS ::= '"PATHS"' ':' ArrayOfStrings

1. **Table Instances and Recipes**

Table is the base class for almost all MOSDEX named objects (except, notably, for Problem). 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 modeling objects are subclasses of Table.

Figure 20: **Table:**

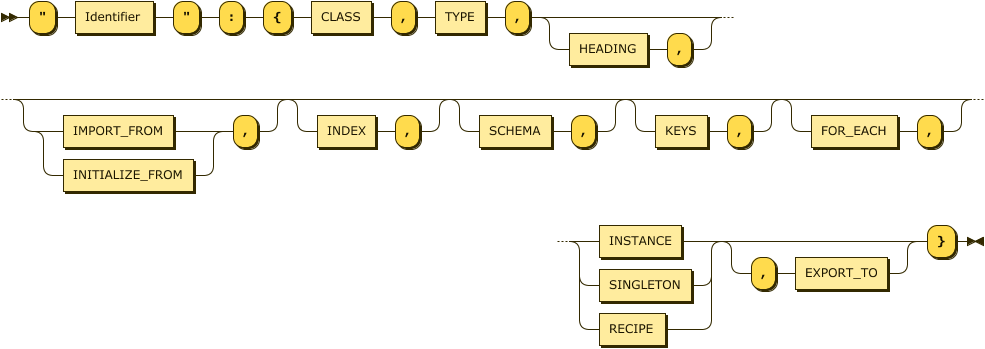
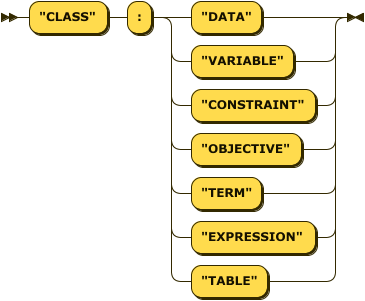


Table ::= '"' Identifier '"' ':' '{' CLASS ',' TYPE ',' ( HEADING ',' )?   
( ( IMPORT\_FROM | INITIALIZE\_FROM ) ',' )? ( INDEX ',' )? ( SCHEMA ',' )? ( KEYS ',' )?   
( FOR\_EACH ',' )? ( INSTANCE | SINGLETON | RECIPE ) ( ',' EXPORT\_TO )? '}'

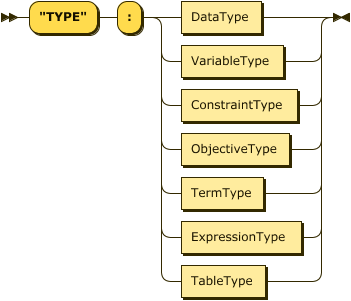
A Table has a Class field that identifies the type of the object it embodies. Since the MOSDEX parser reads a Table before knowing its Class, the Class does not affect how the Table is parsed. However, once a Table is parsed, MOSDEX will check its validity, so for instance, certain fields must be present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

Figure 21: **CLASS:**



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

Figure 22: **TYPE:**



TYPE ::= '"TYPE"' ':' ( DataType | VariableType | ConstraintType | ObjectiveType | TermType | ExpressionType | TableType )

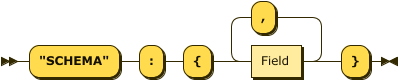
The Data and Table classes have no specific type or schema, but each of the modeling classes, Variable, Constraint, Objective, Term, and Expression, does have its own types and schema, as discussed below.

For now, we defer discussion of the other, subsidiary keyword fields of Table, to focus on the most important content.

Every Table has a *schema* which defines the names and types of its fields. Schema fields must have primitive data types; neither objects nor arrays are permitted as table 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:

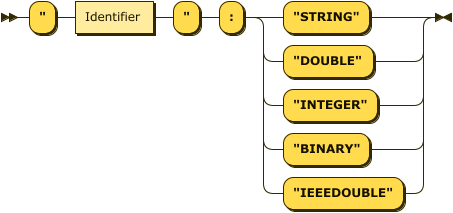
1. The Table is imported or initialized from another Problem.
2. The Table is specified by an SQL Recipe, in which case its schema is generated by the query.
3. The Table is specified as a Singleton instance, in which case, the schema is inferred from the data as explained below.

Figure 23: **SCHEMA:**



SCHEMA ::= '"SCHEMA"' ':' '{' Field ( ',' Field )\* '}'

Figure 24: **Field:**



Field ::= '"' Identifier '"' ':' ( '"STRING"' | '"DOUBLE"' | '"INTEGER"' | '"BINARY"' | '"IEEEDOUBLE"' )

Among the fields of the schema of a Table, the fields that uniquely identify a record are called its *keys*. Keys are often superfluous, and some database management systems, including Apache Spark, do not keep track of them. However, in some circumstances, it is desirable to identify the keys. For example, the modeling objects (variables, constraints, etc.) may use the keys in their mappings to a solver’s internal data structures. For that reason, MOSDEX tables can include an optional KEYS object.

Figure 25: **KEYS:**

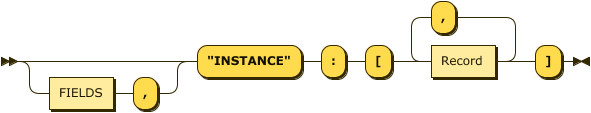


KEYS::= '"KEYS"' ':' ArrayOfStrings

A Table may be specified in either of two forms, *Instance* or *Recipe*. In Instance form, the Table directly specifies the data that it encompasses. In Recipe 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 Problem, but an individual Table must have one form or the other. A Singleton form Table is a subtype of Instance that specifies a single record instance compactly.

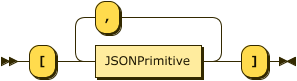
An Instance is a JSON Array of Records, each of which is itself a JSON Array that contains actual data. In an Instance object, the individual items in each Record are unlabeled and can only be parsed using the Table’s schema. The array of Records may be preceded by a Fieldsarray, which repeats the schema’s field names, and can be used as a visual guide for a human reader, but it does not convey any new information beyond the Schema and is not parsed by MOSDEX.

Figure 26: **Instance:**



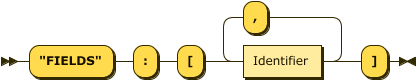
INSTANCE ::= '"INSTANCE"' ':' '{' SCHEMA ( ',' FIELDS? ',' ROWS )? '}'

Figure 27: **Record:**



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

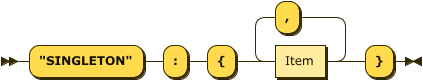
Figure 28: **FIELDS:**



FIELDS ::= '"FIELDS"' ':' '[' Identifier ( ',' Identifier )\* ']'

A Singleton object is a subtype of Instance that provides a simplified, compact representation of a table having a single record, a common special case that warrants separate treatment. A Singleton object is a JSON Object, so it has labels on the items in the record. The labels constitute the field names of the Singleton’s schema. The field types (which again are limited to JSON Primitives) are inferred by the parser.

Figure 29: **SINGLETON:**



SINGLETON ::= '"SINGLETON"' ':' '{' Item ( ',' Item )\* '}'

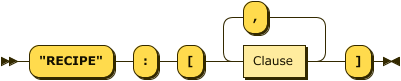
Figure 30: **Item:**



Item ::= '"' Identifier '"' ':' JSONPrimitive

As an alternative to Instance form, a MOSDEX Table can also be specified in Recipe form, which uses SQL to specify how the data are constructed from other Tables or from an external database. Using Recipe 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 Recipe in MOSDEX is an array of Clause objects, each of which has a Directive field and a Predicate field. The Directive is usually an SQL command, such “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. This formulation affords considerable flexibility in formulating queries. In its simplest form, a Recipe is simply a QUERY directive followed by the text string (or array of strings) of the query in the Predicate. A Recipe can also be specified in multiple clauses that break the query into its component directives. The general rule is that the directive and predicate must be interpreted directly as valid SQL by the database engine. It is recommended that such 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 Recipe object in MOSDEX also allows breaking a 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.

Figure 31: **RECIPE:**



RECIPE ::= '"RECIPE"' ':' '[' Clause ( ',' Clause )\* ']'

Figure 32: **Clause:**



Clause ::= '{' '"DIRECTIVE"' ':' ArrayofStrings ',' '"PREDICATE"' ':' ArrayofStrings '}'

The use of arrays of strings as the arguments to Clauses in the Recipe affords a great deal of flexibility because MOSDEX does not parse the SQL. It permits segmenting the clause’s arguments into logical groups; the segmentation is somewhat arbitrary, as long as their concatenation results in a valid SQL statement. The segmentation can be used, for instance, to improve readability or to map the SQL into another language’s query methods.

A QUERY directive is used to create a general query, while other directives can be used to break the query into identifiable components. SELECT, FROM and WHERE directives map directly to the corresponding SQL clauses. JOIN and UNION directives serve as containers for the corresponding SQL clauses and may include multiple Joins or Unions, but they must have arguments that result in valid SQL.

Returning to the discussion of the other, subsidiary keyword fields, a Table may have an optional Heading for documentation. Headings are discussed under the Problems section.

A caveat concerning the use of Instance form Data tables with type Output: Since output usually depends on the solution of the Problem obtained by the solver, it is not possible to create an output instance when MOSDEX is parsed, which must occur before the solver acts in order to create the optimization objects it uses. Thus, even in Instance form, an output table is essentially a recipe for construction. However, since one cannot depend on the solver including an SQL engine, the “recipe” for an output instance table must be very simple:

1. a From directive is limited to a single table,
2. a Select directive can only include fields that are present in that table (possibly relabeled),
3. a Where directive can only include simple logical constraints
4. Joins and Unions are not allowed.

In essence, the recipe would be implemented by a simple iterator over a single solver object.

MOSDEX accommodates data/model separation and modular structures by enabling a Table to refer to objects external to the Problem of which it is a member. Import copies the table from a location specified by its argument string. Initialize performs the same function, but enables updates on subsequent uses in an iterative solver procedure. Such processes are discussed in the section on Modular Structure. Export performs the opposite function, exposing a Table to another Problem. The strings in the element portion of these objects may use string substitution to access Tables in MOSDEX files identified by a Path name connected by the MOSDEX caller at run-time.

Figure 33: **IMPORT\_FROM:**



IMPORT\_FROM ::= '"IMPORT\_FROM"' ':' ArrayOfStrings

Figure 34: **INITIALIZE\_FROM:**



INITIALIZE\_FROM ::= '"INITIALIZE\_FROM"' ':' ArrayOfStrings

Figure 35: **EXPORT\_TO:**



EXPORT\_TO ::= '"EXPORT\_TO"' ':' ArrayOfStrings

The Index and ForEach fields of Table enable functions of Modular Structure and are discussed there.

1. **Modeling Objects and their Standard Schemas**

The MOSDEX modeling classes, subclasses of Table, are Variable, Constraint, Objective, Term, and Expression; they can be given in either instance or recipe form. While a MOSDEX Data object may have any reasonable schema, each type of MOSDEX modeling object has a *standard schema* that enables realizing the object in a solver. 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. Recipe form modeling objects do not specify a schema 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 object.

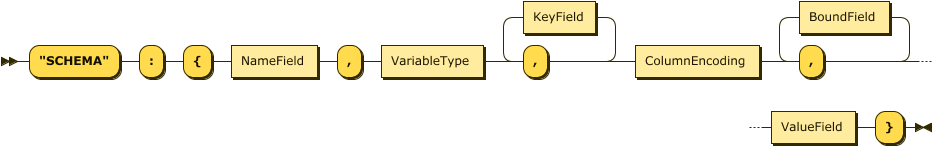
While each type of modeling object has unique fields, there are a number of commonalities. First, in most cases, a modeling object represents an entire category, or *block*, rather than a singular occurrence, because most large optimization problems have a block structure. That is, for instance, a network 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 objects 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 objects 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 schemas of the modeling objects 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 the modeling object’s schemas 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 to null, which are populated in a post-processing phase of the solution process.

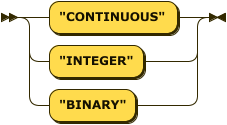
Finally, since the solver will read the MOSDEX modeling objects to create its own internal representation of the Problem, each record in a Table should contain the information required to construct the internal representation. While each solver has its own syntax, there is enough commonality among most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

Figure 36: **VariableSchema:**



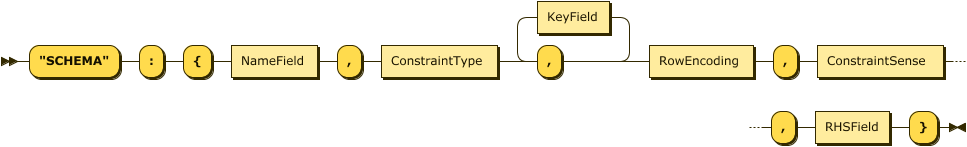
VariableSchema ::= '"SCHEMA"' ':' '{' NameField ',' VariableType ',' ( KeyField ',' )\* ColumnEncoding ','   
( BoundField ',' )\* ValueField '}'

Figure 37: **VariableType:**



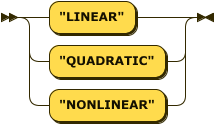
VariableType ::= '"CONTINUOUS"' | '"INTEGER"' | '"BINARY"'

Figure 38: **ConstraintSchema:**



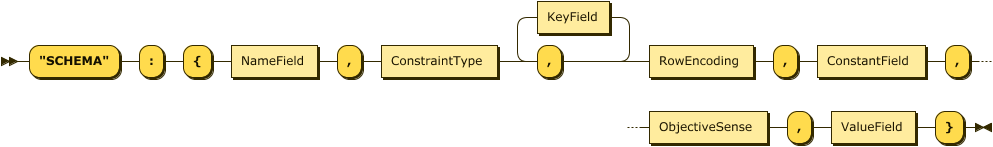
ConstraintSchema ::= '"SCHEMA"' ':' '{' NameField ',' ConstraintType ',' ( KeyField ',' )\* RowEncoding ',' ConstraintSense ',' RHSField '}'

Figure 39: **ConstraintType, ObjectiveType, and TermType:**



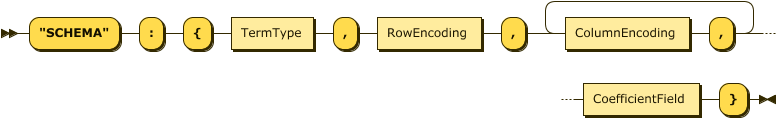
(ConstraintType | ObjectiveType | TermType) ::= '"LINEAR"'| '"QUADRATIC"' | '"NONLINEAR"'

Figure 40: **ObjectiveSchema:**



ObjectiveSchema ::= '"SCHEMA"' ':' '{' NameField ',' ConstraintType ',' ( KeyField ',' )\* RowEncoding ',' ConstantField ',' ObjectiveSense ',' ValueField '}'

Figure 41: **TermSchema:**



TermSchema::= '"SCHEMA"' ':' '{' TermType ',' RowEncoding ',' ( ColumnEncoding ',' )+ CoefficientField '}'

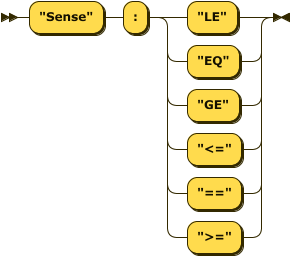
The Term schema permits one or more column index fields, representing the possibility that it could involve more than one variable, to enable MOSDEX to accommodate quadratic and other nonlinear optimization problems.

Figure 42: **NameField:**



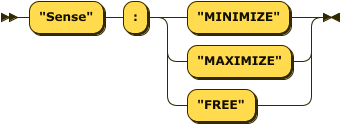
NameField ::= '"' Identifier '"' ':' '"STRING"'

Figure 43: **ConstraintSense:**



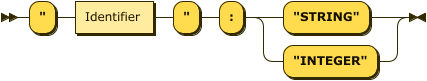
ConstraintSense ::= '"Sense"' ':' ( '"LE"' | '"EQ"' | '"GE"' | '"<="' | '"=="' | '">="' )

Figure 44: **ObjectiveSense:**



ObjectiveSense ::= '"Sense"' ':' ( '"MINIMIZE"' | '"MAXIMIZE"' | '"FREE"' )

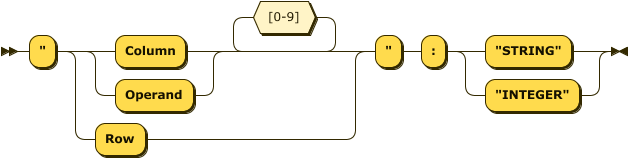
Figure 45: **KeyField:**



KeyField ::= '"' Identifier '"' ':' ( '"STRING"' | '"INTEGER"' )

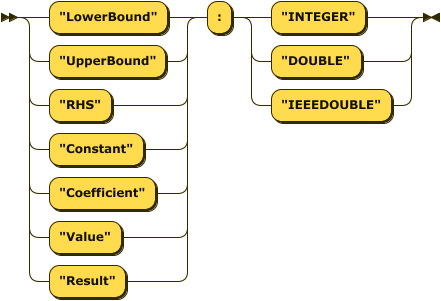
Tableau Encodings come in two varieties, Row Encoding and Column Encoding. Note that, as discussed above, there may be multiple column indexes, which are distinguished by an integer suffix.

Figure 46: **RowEncoding, ColumnEncoding, and OperandEncoding:**



(RowEncoding | ColumnEncoding | OperandEncoding)::= '"' ( 'Row' | ( 'Column' | 'Operand' ) [0-9]\* ) '"' ':' ( '"STRING"' | '"INTEGER"' )

Figure 47: **NumberField:**



NumberField ::=   
( '"LowerBound"' | '"UpperBound"' | '"RHS"' | '"Constant"' | '"Coefficient"' | '"Value"' | '"Result"' ) ':' ( '"INTEGER"' | '"DOUBLE"' | '"IEEEDOUBLE"' )

1. **Modular Structure**

Modular structures occur when a Problem, called the *parent*, embeds other Problems, called the *children*. Modular structures arise in such model constructs as block structures, decomposition algorithms, and stochastic programs. A child problem may be part of the same MOSDEX File as the parent, or it may be part of a linked file. In MODEX, the parent is responsible for mapping its tables onto the corresponding tables of the child; the child usually has little or no knowledge of the parent’s structure. In fact, the child could in most cases be optimized as a stand-alone problem.

It is important to understand that MOSDEX itself does not provide an algorithm for solving modular problems; that is the province of a solver. For example, setting up a column generation structure in MOSDEX will not be sufficient for an ordinary solver of linear programs to execute the decomposition. The solver part of the software stack must have a control structure that alternately solves the master problem and subproblems, checks the convergence criterion, and terminates the solve process. Furthermore, the solver algorithm’s requirements will determine, in part, how the problems are set up in MOSDEX. What MOSDEX does provide are standards to specify the modules that the solver executes, how they exchange data and coordinate during execution.

MOSDEX provides several constructs that enable building modular structures. First, as discussed above, a Table can be Imported from, Initialized from, or Exported to another Problem. If the target is a Table in another Problem within the same MOSDEX File, a Table reference of the form

*ProblemName.TableName*

is used. However, if the target Table is in a separate file, the form of the Table reference is

*Path/ProblemName.TableName*

where the Path is a path string or URL in the relevant operating system (most programming languages provide facilities for managing paths). Good programming practice also discourages hard-wiring properties such as path strings in code, so MODEX provides a Paths object to define *symbolic* names for paths that can be bound to actual path strings by the caller of the MOSDEX File. The actual path strings are substituted for these names by the MOSDEX parser using *string* *substitution*.

As the second construct for supporting modular structure, MOSDEX provides a limited capability to use *string* *variables*. In MOSDEX, a string beginning with a $ sign denotes a string variable. The parser will substitute a string variable with an actual string as it processes a MOSDEX file. MOSDEX permits string variables in two situations:

1. To substitute a path string for a path name in a Table reference, which usually occurs in an Import, Initialization, or Export or in the From or Join clause of an SQL query. In this case, the path name must be declared (without the $ sign) in a Paths object.
2. To substitute an actual value for an index variable, which usually occurs in a Select, Where, or On clause of an SQL query. In this case, the index variable must be declared (without the $ sign) in Index object within a Table. Indexes are discussed below.

When a string variable is used, it should appear as a stand-alone component in an array of strings; the parser does not attempt to locate string variables embedded within strings.

As the third construct for supporting modular structure, MOSDEX provides a capability to use *Index* *variables*. It is frequently the case that the parent problem will want to incorporate multiple versions of a child problem. For example, in a multi-commodity flow problem, there could be multiple versions of a minimum cost flow problem, one for each product. Index variables deal with this common situation. An index variable is defined by an Index object within a Table and is bound to that Table as its *index* *set*; the index variable represents the records of its index set.

An index variable is called by For Each objects in other Tables. The For Each object in effect creates a loop in which the host Table is reproduced for each different index value, that is, for each record in the index set corresponding to the index variable. Usually the reproduced tables will be collected by a Union or Outer Join clause in an SQL query, since MOSDEX does not support arrays of Tables; as mentioned above, string substitution is used to refer to an index variable in an SQL query. When multiple Tables all use a For Each object with the same index variable, the MOSDEX parser in effect executes them in parallel, so that each table is reproduced for the given index value before the parser moves on to the next value.

Figure 48: **INDEX:**

****

INDEX ::= '"INDEX"' ':' Identifier

Figure 49: **FOR\_EACH:**

****

FOR\_EACH ::= '"FOR\_EACH"' ':' Identifier

1. **Nonlinear Problems and Expressions**

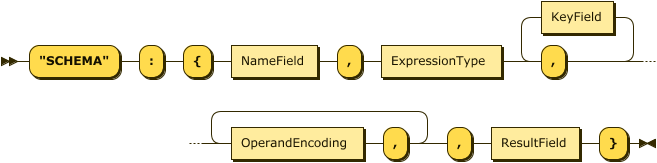
This experimental aspect of MOSDEX is an attempt to support nonlinear models. It would require a suitable optimization solver and a software bridge capable to construct the nonlinear functions specified by MOSDEX expressions. The MOSDEX representation is based on the JSON ADFun proposal by Brad Bell at <https://coin-or.github.io/CppAD/doc/json_ad_graph.htm> and <https://coin-or.github.io/CppAD/doc/to_json.cpp.htm>.

The nonlinear extension of MOSDEX is embodied in the Expression class, a subclass of Table, which embodies a node of an expression graph, analogously to the ADFun graph construct created by Brad. Expressions themselves behave like other Tables (i.e. they each have a defined schema, and they can have Instance or Recipe form). The main difference between MOSDEX and JSON ADFun is that in the latter, a node is essentially a scaler object while in MOSDEX, an Expression has dimensionality as specified by its key fields. Thus MOSDEX implements families of Expressions that correspond to the structure of the optimization problem that it represents.

Expression is a subclass of Table in MOSDEX; as named JSON Objects, multiple Expression objects can coexist within a MOSDEX Problem. Typically, a linked collection of Expressions comprises a nonlinear expression in the Problem; multiple collections of Expressions can represent different nonlinear expressions in the Problem. (Note: there is no explicit object representing such a collection; instead, the collection is represented by linkages among the Expressions through their operand fields.) Each Expression is one of 4 types: Independent Variable, Dependent Variable, Parameter, or operation. Independent and Dependent Variables link to a Variable modeling object; the distinction between them is that Independent Variables are inputs to the expression and Dependent Variables are outputs. Parameters link to data items and do not interact with the solver. Finally, operations represent operators that combine Independent Variables and Parameters; typically they represent elementary math operations such as add or multiply, or math functions, such as exp or log. Currently MOSDEX offers unary (one operand) and binary (two operands) operators, but a future extension will also permit operators with an indeterminate number of operands (e.g. sum) which are needed to support model/data separation. Expressions combine in a graph-like fashion that enables building complex expressions from elementary operations. The use of expression graphs is common to representing nonlinear optimization problems, and it facilitates automatic differentiation used by nonlinear solvers. The Expressions should take the form of an acyclic directed graph; however, MOSDEX will not check this condition; that is the responsibility of the service that evaluates the expression when called by the solver.

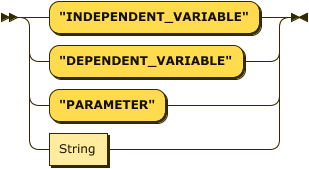
Each Expression is a named Table object, and as such, its name can be any valid identifier. The schema of each Expression has fields for its Operation (which specifies the operator it embodies), its keys, its Operands, and its Result. The result (e.g. Node\_0.Result) is usually used as an operand in the parent expression. An Expression object also must contain either an Instance or Recipe specifying its connections to other Tables in the Problem. In the case of an Independent Variable, the link is often through a call to the Solver which retrieves the current value of the variable (assuming that the Solver repeatedly needs to evaluate the expression for different trial solutions). A Dependent Variable links back to a Constraint or Objective through a Term object.

Figure 50: **ExpressionSchema:**



ExpressionSchema::= '"SCHEMA"' ':' '{' NameField ',' ExpressionType ',' ( KeyField ',' )\* ( OperandEncoding ',' )+ ',' ResultField '}'

Figure 51: **ExpressionType:**



ExpressionType::= '"INDEPENDENT\_VARIABLE"' | '"DEPENDENT\_VARIABLE"' | '"PARAMETER"' | String

The Expression schema permits one or more operand fields, representing linkages to other expressions, or to variables or parameters, thus realizing a graph structure.

1. **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 bridge between MOSDEX and its own API. Because most of the solvers of interest have similar APIs, the bridge software 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 bridge to map its modeling objects 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 bridge, 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 objects 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

*objectName\_key1\_...keyN*

where the object 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. MOSDEX offers two ways to accomplish this task. First, as discussed in the section on modeling objects and their schemas, each object 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.

As an alternative, MOSDEX presumes the existence of methods in the bridge classes that return values from the solver. The form of these method calls is

*#Solver.getXXX(code)*

where XXX stands for Variable, Objective, Dual, etc., and code is the relevant column or row identifier. In the examples, these calls are inserted as strings with a # prefix in the Instance or Recipe representing the output table.

1. **Conclusion**

To be written

**Appendix 1: MOSDEX Examples**

The JSON Schema for MOSDEX is found in the following document:

MOSDEXSchemaV1-2.json

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

Volsay\_1-2.json – a simple 2-variable, 3-constraint linear program illustrating instance form tables

net1a\_1-2.json – a network flow linear program illustrating recipe form tables

net1b\_1-2.json – the same network flow linear program using instance form tables

sailco\_1-2.json – a production planning linear program illustrating lagged inventory decision variables

warehousing\_1-2.json – a facility location mixed-integer linear program illustrating a large-scale, structured problem in recipe form

multicommodity\_1-2.json – a multi-commodity network flow linear program in extensive form

multinet\_1-2.json – the same multi-commodity flow problem using modular structure

cuttingStock\_1-2.json – a cutting-stock mixed-integer linear program illustrating use of modular structure for decomposition in a column generation algorithm

trafficNetworkQP\_1-2.json – a quadratic programming problem

trafficNetworkNLP\_1-2.json – the same problem formulated as a nonlinear program with expression graph

**Appendix 2: MOSDEX Parsing Rules Compilation**

The JSON Schema for MOSDEX is found in the following document:

MOSDEXSchemaV1-2.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. ArrayOfStrings::= '[' (String (',' String)\*)? ']'
6. JSONElement::= JSONObject | JSONArray |JSONPrimitive
7. JSONPrimitive::= String | Number
8. Number::= Binary | Integer | Double | IEEEDouble
9. IEEEDouble::= '"' ('NaN' | 'Infinity' | '-Infinity' | ('-')?'0x0.0p0' | (('-')? '0x1' '.' (HexDigit)+ 'p' ('-')? Integer)) '"'
10. HexDigit::= [0-9a-fA-F]
11. Identifier::= ([a-z]|[A-Z]|'\_') ([a-z]|[A-Z]|[0-9]|'\_')\*
12. Keyword::= [A-Z] ([A-Z]|'\_'|'-')\*
13. MosdexFile::= '{' '"' SYNTAX '"' ':' String ',' PROBLEM (',' PROBLEM)\* '}'
14. PROBLEM::= '"' Identifier '"' ':' '{' '"CLASS"' ':' '"PROBLEM"' ',' HEADING (',' SOLVER\_PARAMETERS)? (',' PATHS)? ',' Table (',' Table)\* '}'
15. HEADING::= '"HEADING"' ':' '{' DESCRIPTION (',' ProblemInformation)\* '}'
16. DESCRIPTION::= '"DESCRIPTION"' ':' ArrayOfStrings
17. ProblemInformation::= ('"VERSION"' | '"REFERENCE"' |'"AUTHOR"' | '"NOTICES"' | '"MATH"') ':' ArrayOfStrings
18. SOLVER\_PARAMETERS::= '"SOLVER\_PARAMETERS"' ':' '{' Identifier ':' JSONObject (',' Identifier ':' JSONObject)\* '}'
19. PATHS::= '"PATHS"' ':' ArrayOfStrings
20. Table::= '"' Identifier '"' ':' '{' CLASS ',' TYPE (',' HEADING)? (',' (IMPORT\_FROM | INITIALIZE\_FROM))? (',' INDEX)? (',' SCHEMA)? (',' KEYS)? (',' FOR\_EACH)? ',' (INSTANCE | SINGLETON | RECIPE)
21. (',' EXPORT\_TO) ?'}'
22. CLASS::= '"CLASS"' ':' ('"DATA"' | '"VARIABLE"' | '"CONSTRAINT"' | '"OBJECTIVE"' | '"TERM"' | '"EXPRESSION"' | '"TABLE"')
23. TYPE::= '"TYPE"' ':' (DataType | VariableType | ConstraintType | ObjectiveType | TermType | ExpressionType | TableType)
24. DataType::= ('"INPUT"' | '"OUTPUT"' |'"AUXILIARY"')
25. IMPORT\_FROM::= '"IMPORT\_FROM"' ':' ArrayOfStrings
26. INITIALIZE\_FROM::= '"INITIALIZE\_FROM"' ':' ArrayOfStrings
27. EXPORT\_TO::= '"EXPORT\_TO"' ':' ArrayOfStrings
28. INDEX::= '"INDEX"' ':' Identifier
29. FOR\_EACH::= '"FOR\_EACH"' ':' Identifier
30. SCHEMA::= '"SCHEMA"' ':' '{' Field (',' Field)\* '}'
31. Field::= '"' Identifier '"' ':' ('"STRING"' | '"DOUBLE"'| '"INTEGER"' | '"BINARY"' | '"IEEEDOUBLE"')
32. KEYS::= '"KEYS"' ':' ArrayOfStrings
33. INSTANCE::= (FIELDS ',')? '"INSTANCE"' ':' '[' Record (',' Record)\* ']'
34. FIELDS::= '"FIELDS"' ':' '[' Identifier (',' Identifier)\* ']'
35. Record::= '[' JSONPrimitive (',' JSONPrimitive)\* ']'
36. SINGLETON::= '"SINGLETON"' ':' '{' Item (',' Item)\* '}'
37. Item::= '"' Identifier '"' ':' JSONPrimitive
38. RECIPE::= '"RECIPE"' ':' '[' Clause (',' Clause)\* ']'
39. Clause::= '{''"DIRECTIVE"' ':' ArrayofStrings ',' '"PREDICATE"' ':' ArrayofStrings '}'
40. VariableSchema::= '"SCHEMA"' ':' '{' NameField ',' VariableType ',' (KeyField ',')\* ColumnEncoding ',' (BoundField ',')\* ValueField '}'
41. ConstraintSchema::= '"SCHEMA"' ':' '{' NameField ',' ConstraintType ',' (KeyField ',')\* RowEncoding ',' ConstraintSense ',' RHSField '}'
42. ObjectiveSchema::= '"SCHEMA"' ':' '{'' NameField ',' ConstraintType ',' (KeyField ',')\* RowEncoding ',' ConstantField ',' ObjectiveSense ',' ValueField '}'
43. TermSchema::= '"SCHEMA"' ':' '{' TermType ',' RowEncoding ',' (ColumnEncoding ',' )+ CoefficientField '}'
44. ExpressionSchema::= '"SCHEMA"' ':' '{' NameField ',' ExpressionType ',' (KeyField ',')\* (OperandField ',')+ ',' ResultField '}'
45. NameField::= '"' Identifier '"' ':' '"STRING"'
46. VariableType::= ('"CONTINUOUS"' | '"INTEGER"' | '"BINARY"')
47. ConstraintType::= ('"LINEAR"' | '"QUADRATIC"' | '"NONLINEAR"')
48. ObjectiveType::= ('"LINEAR"' | '"QUADRATIC"' | '"NONLINEAR"')
49. TermType::= ('"LINEAR"' | '"QUADRATIC"' | '"NONLINEAR"')
50. ExpressionType::= ('"INDEPENDENT\_VARIABLE"' | '"DEPENDENT\_VARIABLE"' | '"PARAMETER"' | String)
51. TableType::= String
52. ObjectiveSense::= '"Sense"' ':' ('"MINIMIZE"' | '"MAXIMIZE"' | '"FREE"')
53. ConstraintSense::= '"Sense"' ':' ('"LE"' | '"EQ"' | '"GE"' | '"<="' | '"=="' | '">="')
54. KeyField::= '"' Identifier '"' ':' ('"STRING"' | '"INTEGER"' )
55. TableauEncoding::= '"' ('Row' | 'Column'[0-9]\*| 'Operand'[0-9]\*) '"' ':' ('"STRING"' | '"INTEGER"')
56. NumberField::= ('"LowerBound"' | '"UpperBound"' | '"RHS"' | '"Constant"' | '"Coefficient"' | '"Value"' | '"Result"') ':' ('"INTEGER"' | '"DOUBLE"' | '"IEEEDOUBLE"')