**MOSDEX Syntax**

Dr. Jeremy A. Bloom  
[jeremyblmca@gmail.com](mailto:jeremyblmca@gmail.com)  
March 16, 2019

1. **Introduction**

This paper presents an initial attempt to define 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.

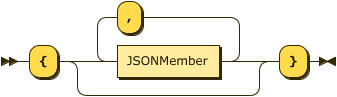
1. **JSON and Primitive Types**

MOSDEX is a derivative of JavaScript Object Notation (JSON), and therefore, MOSDEX files adhere to the JSON standard (see <http://json.org/>). JSON has two fundamental classes: *objects* and *arrays*.

An object is an unordered list of key:value pairs, or *members*, where each key is a string; it is enclosed within curly braces, { and }, and a member’s key and value are separated by a colon, with the object’s members separated by commas. Keys must be unique within an object.

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

Figure **: JSONObject:**



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

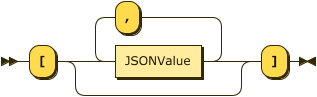
Figure : **JSONMember:**

C:\Users\Jeremy\Documents\Technical\Research\Optimization Modeling\MOSDEX Syntax Diagrams\JSON Diagrams\JSONMember.png

JSONMember::= String ':' JSONValue

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

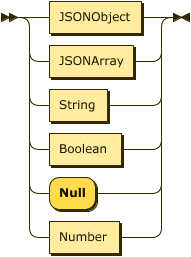
Figure : **JSONArray:**



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

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

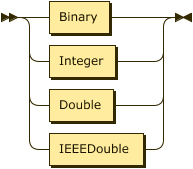
Figure : **JSONValue:**



JSONValue ::= JSONObject | JSONArray | String | Boolean | 'Null' | Number

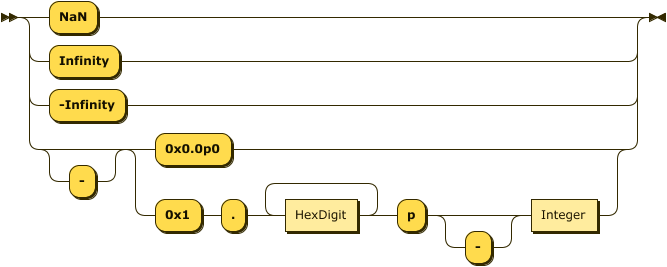
Additionally, MOSDEX allows two other primitive types. Binary (zero or one) values are a subtype of integers used in mathematical expressions, especially as decision variables in optimization models; note that the binary type differs from the boolean type, which is used in logical expressions. IEEE doubles 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. However, because optimization solvers use this format, it provides the most precise way to exchange numerical data with a solver.

Figure : **Number:**



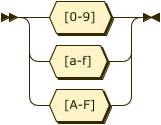
Number::= Binary | Integer | Double | IEEEDouble

Figure : **IEEEDouble:**



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

Figure : **HexDigit:**



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

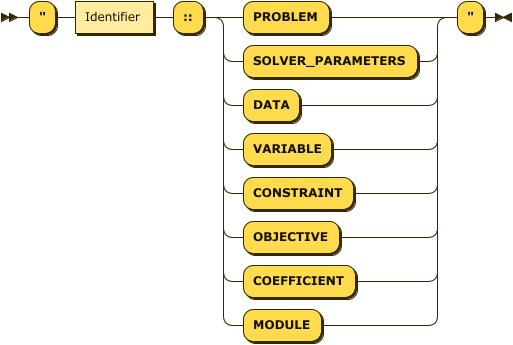
JSON and MOSDEX do not support multiline string literals; however, they both support arrays of strings, which serve the same purpose. JSON generally ignores white space between tokens. The JSON standard does not support comments, but many parsers support them. MOSDEX allows two kinds of C or Java-style comments:

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

1. **Keywords and Tags**

MOSDEX keywords are written in all capital letters and can include underscore characters. Keys in MOSDEX objects are either keywords, assuming that no keyword in repeated in an object, or they are tags of the form “identifier::KEYWORD” where the double quotes are essential to JSON recognizing the tag as a string. An identifier generally follows the rules for identifiers in JavaScript. Usually the keyword denotes the type of the value which follows the tag. In an extension of JSON, MOSDEX will parse a tag string to extract the item identifier and type.

Figure : **Tag:**

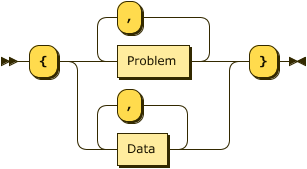


Tag ::= '"' Identifier '::' ( 'PROBLEM' | 'SOLVER\_PARAMETERS' | 'DATA' | 'VARIABLE' | 'CONSTRAINT' | 'OBJECTIVE' | 'COEFFICIENT' | 'MODULE' ) '"'

1. **Problems and Datasets**

At the highest level, a MOSDEX file (we use the term file generically for any input source) is a JSON Object. Two types of files are supported: a self-contained presentation of the data and modeling objects for one or more optimization Problems, or, when model/data separation is used, the data may be presented in one or more separate files. 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 MOSDEX and data and modeling objects may be presented in either recipe or instance form.

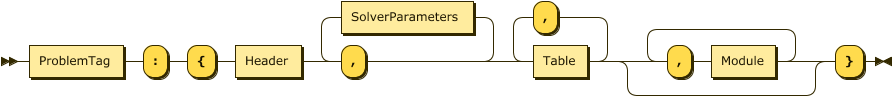
Figure : **MOSDEX:**



MOSDEX::= '{' ( Problem ( ',' Problem )\* | Data ( ',' Data )\* ) '}'

A MOSDEX Problem is a collection of tables representing data and modeling objects. A problem can also include one or more modules that represent calls to other problems, as discussed in the modules section below.

Figure : **Problem:**



Problem ::= ProblemTag ':' '{' Header ',' ( SolverParameters ',' )\* Table ( ',' Table )\* ( ',' Module )\* '}'

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 solver parameters objects are permitted, each tagged with an identifier denoting the target solver.

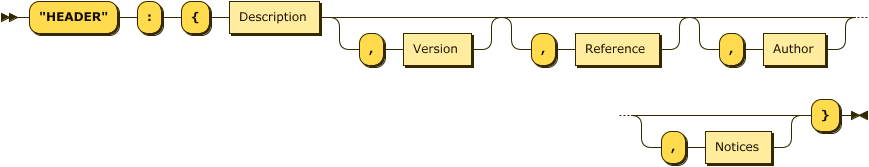
Figure : **SolverParameters:**



SolverParameters::= SolverParametersTag ':' JSONObject

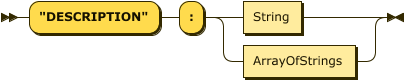
A header collects information about a problem or table; it is mandatory for a problem and optional for a table. Of the information in the header, only the description is mandatory, and it may consist of a single string or of multiple lines. The other items in the header are a version number, a reference, which is typically a link or a citation to the source of the problem, the author’s name and contact information, and legal notices, such as a copyright or a license.

Figure : **Header:**



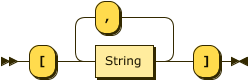
Header::= '"HEADER"' ':' '{' Description ( ',' Version )? ( ',' Reference )? ( ',' Author )? ( ',' Notices )? '}'

Figure : **Description:**



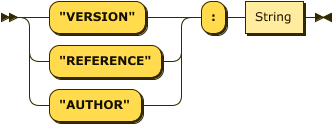
Description::= '"DESCRIPTION"' ':' ( String | ArrayOfStrings )

Figure : **ArrayOfStrings:**



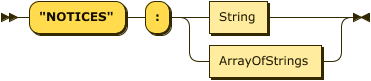
ArrayOfStrings::= '[' String ( ',' String )\* ']'

Figure : **ProblemInformation:**



ProblemInformation::= ( '"VERSION"' | '"REFERENCE"' | '"AUTHOR"' ) ':' String

Figure : **Notices:**



Notices::= '"NOTICES"' ':' ( String | ArrayOfStrings )

1. **Table Instances and Recipes and Data Objects**

Table is the base class for almost all MOSDEX objects (except for Problem and Module, to be discussed below). Conceptually a table is a two-dimensional object with a fixed number of columns, or *fields*, and an indefinite number of rows, or *tuples*; think of a table in a relational database. A Data object is a subclass of Table.

Figure : **Table:**

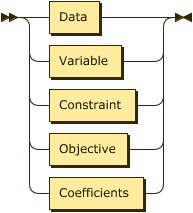
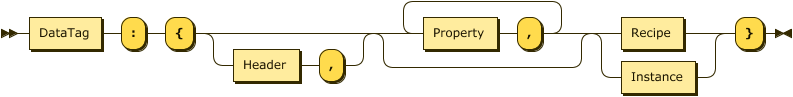


Table::= Data | Variable | Constraint | Objective | Coefficients

Figure : **Data:**

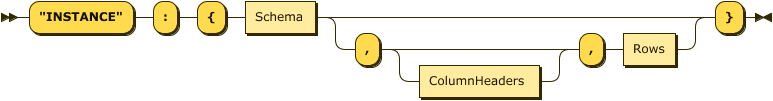


Data::= DataTag ':' '{' ( Header ',' )? ( Property ',' )\* ( Recipe | Instance ) '}'

A table may have an optional header for documentation. Headers are discussed under the Problems section. A table may optionally also have *properties*, each of which is keyword followed by a string, which may also be a keyword. The allowed properties for a table depend on the type of table; see the Modeling Objects section for further discussion.

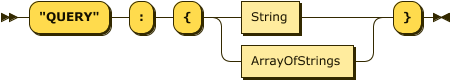
A table may be specified in either of two forms, *instance* or *recipe*. In instance form, the table directly specifies the data it encompasses. In recipe form, the table uses SQL to specify how the data are constructed from other tables or from an external database. Both forms of tables may coexist in a MOSDEX problem, but an individual table must have one form or the other.

Figure : **Instance:**



Instance::= '"INSTANCE"' ':' '{' Schema ( ',' ColumnHeaders? ',' Rows )? '}'

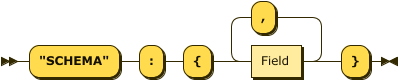
Figure : **Recipe:**



Recipe::= '"QUERY"' ':' '{' ( String | ArrayOfStrings ) '}'

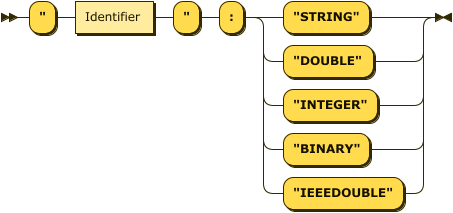
An instance form table must include a *schema*, which defines the names and types of its fields. Fields are primitive data types; neither objects nor arrays are permitted as table elements.

Figure : **Schema:**



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

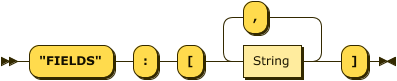
Figure : **Field:**



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

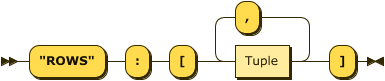
An instance may also have a *Rows* array that contains actual data; however, when the dataset is large, it may be presented in a separate file, and only the schema is present in the problem file. When Rows is present, it may be preceded by *ColumnHeaders* array, which repeats the schema field names, used as a visual guide for a human reader; it does not convey any new information beyond the schema and is not parsed by MOSDEX.

Figure : **ColumnHeaders:**



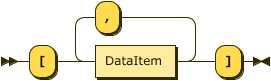
ColumnHeaders::= '"FIELDS"' ':' '[' String ( ',' String )\* ']'

Figure : **Rows:**



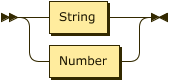
Rows::= '"ROWS"' ':' '[' Tuple ( ',' Tuple )\* ']'

Figure : **Tuple:**



Tuple::= '[' DataItem ( ',' DataItem )\* ']'

Figure : **DataItem:**

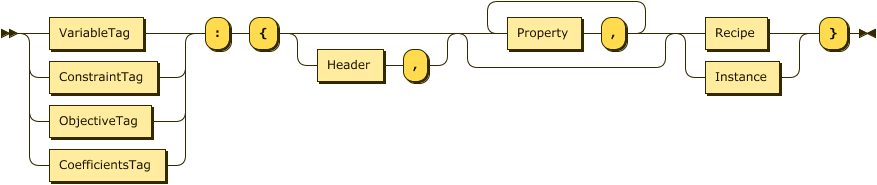


DataItem ::= String | Number

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

The MOSDEX modeling classes, subclasses of Table, are Variable, Constraint, DecisionExpression, and Coefficient; they can be given in either instance or recipe form.

Figure : **ModelingObject:**



ModelingObject::= ( VariableTag | ConstraintTag | ObjectiveTag | CoefficientsTag ) ':'   
'{' ( Header ',' )? ( Property ',' )\* ( Recipe | Instance ) '}'

Each class has some relevant properties, which differ among them, as shown in the table below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Role | Order | Type | Sense |
| Variable | "OPTIMZATION" | N/A | "CONTINUOUS" | "INTEGER" | "BINARY" | N/A |
| Constraint | "OPTIMZATION" | N/A | "LINEAR"\* | "LE" | "EQ" | "GE" | "<=" | "==" | ">=" |
| DecisionExpression | "OPTIMZATION" | N/A | "LINEAR"\* | N/A |
| Coefficient | "OPTIMZATION" | "NONE" | "ROW-WISE" | "COLUMN-WISE" | N/A | "MINIMIZE" | "MAXIMIZE" | "FREE" |

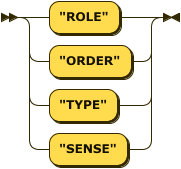
\* At present, only “LINEAR” is supported, but other types may be added in the future

Figure : **Property:**



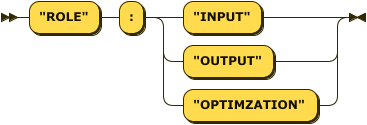
Property ::= PropertyKey ':' String

Figure : **PropertyKey:**



PropertyKey::= '"ROLE"' | '"ORDER"' | '"TYPE"' | '"SENSE"'

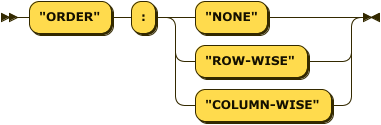
Figure : **Role:**



Role::= '"ROLE"' ':' ( '"INPUT"' | '"OUTPUT"' | '"OPTIMZATION"' )

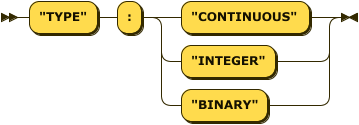
The Order property, used only in a Coefficients object or a Data table, specifies if the items have an order, which may be useful with certain solvers.

Figure : **Order:**



Order::= '"ORDER"' ':' ( '"NONE"' | '"ROW-WISE"' | '"COLUMN-WISE"' )

Figure : **VariableType:**



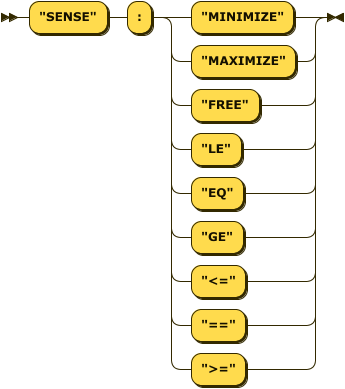
VariableType::= '"TYPE"' ':' ( '"CONTINUOUS"' | '"INTEGER"' | '"BINARY"' )

Figure : **ConstraintType:**



ConstraintType::= '"TYPE"' ':' '"LINEAR"'

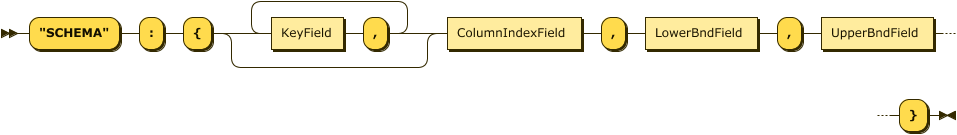
Figure : **Sense:**



Sense::= '"SENSE"' ':' ( '"MINIMIZE"' | '"MAXIMIZE"' | '"FREE"' |   
'"LE"' | '"EQ"' | '"GE"' | '"<="' | '"=="' | '">="' )

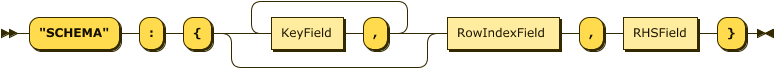
While a MOSDEX Data object may have any reasonable schema, each type of MOSDEX ModelingObject has a *standard schema* that enables realizing the object in a solver. While there is no standard for a solver’s modeling API, most solvers have similar APIs, so the MOSDEX standard schemas should suffice in 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.

Figure : **VariableSchema:**



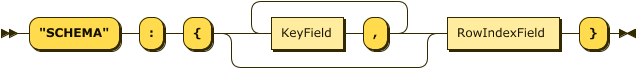
VariableSchema::= '"SCHEMA"' ':' '{' ( KeyField ',' )\* ColumnIndexField ',' LowerBndField ',' UpperBndField '}'

Figure : **ConstraintSchema:**



ConstraintSchema::= '"SCHEMA"' ':' '{' ( KeyField ',' )\* RowIndexField ',' RHSField '}'

Figure : **ObjectiveSchema:**



ObjectiveSchema::= '"SCHEMA"' ':' '{' ( KeyField ',' )\* RowIndexField '}'

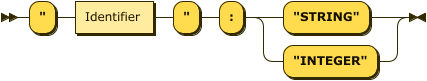
Figure : **CoefficientSchema:**



CoefficientSchema::= '"SCHEMA"' ':' '{' RowIndexField ',' ( ColumnIndexField ',' )+ EntryField '}'

The decision expression and coefficient schemas permit one or more column index fields, representing the possibility that both could involve more than one variable, to enable future MOSDEX extensions to quadratic and other non-linear optimization problems.

Figure : **KeyField:**

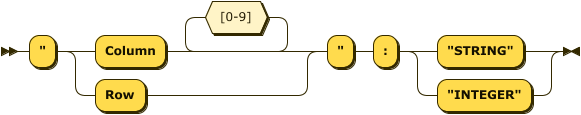


KeyField ::= '"' Identifier '"' ':' ( '"STRING"' | '"INTEGER"' )

The key field or fields uniquely identify each individual variable, constraint, or decision expression, based on the index sets of undelying model structure; index sets are usually multi-dimensional. To communicate with the solver, however, typically each individual variable, constraint, or decision expression must be mapped to a column or row of the two-dimensional tableau by constructing a unique identifier, an *index*, for each row or column. The exact form of the index may depend on the solver, since some will accept strings, which may have limits on the number of characters allowed, while others accept only integers. As a general rule, the authors recommend an index of the form variable or constraint name followed by its keys within the table which defines its index set, with each item separated by an underscore. Doing so will make the model instance readable for debugging. MOSDEX does not parse the components of an index.

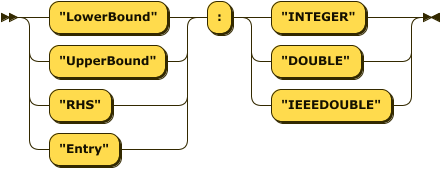
Index fields come in two varieties, RowIndexField and ColumnIndexField. Note that, as discussed above, there may be multiple column indexes, which are distinguished by an integer suffix.

Figure : **IndexField:**



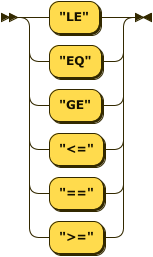
IndexField::= '"' ( 'Row' | 'Column' [0-9]\* ) '"' ':' ( '"STRING"' | '"INTEGER"' )

Figure : **NumberField:**



NumberField::= ( '"LowerBound"' | '"UpperBound"' | '"RHS"' | '"Entry"' ) ':' ( '"INTEGER"' | '"DOUBLE"' | '"IEEEDOUBLE"' )

Figure : **Direction:**



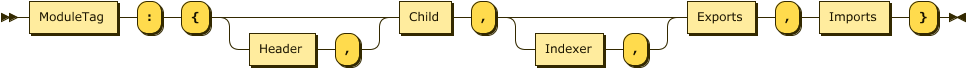
Direction::= '"LE"' | '"EQ"' | '"GE"'| '"<="'| '"=="' | '">="'

1. **Modules** (This section is tentative)

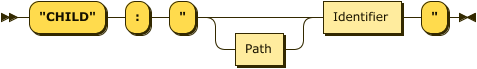
Modules permit problems to embed other problems, enabling such model constructs as block structures, decomposition algorithms, stochastic programs, and even special structures like indicator constraints. A module is thus a reference to another problem, called the child, which may be part of the same MOSDEX file as the parent or 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 has no knowledge of the parent’s structure. In fact, the child could in most cases be optimized as a stand-alone problem. Thus within a module, tables with an input role are mapped to inputs to the child, and tables with an output role are mapped to outputs from the child.

Figure : **Module:**



Module::= ModuleTag ':' '{' ( Header ',' )? Child ',' ( Indexer ',' )? Exports ',' Imports '}'



Child::= '"CHILD"' ':' '"' Path? Identifier '"'

If the child problem is present in the parent MOSDEX file, only the name is needed; otherwise the path string provides the location information.

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. To deal with this common situation, a module can define an index for its child problem.

Figure : **Indexer:**



Indexer::= '"FOREACH"' ':' '"' Identifier '"IN"' TableReference '"'

The identifier is the index name, the scope of which is limited to the module. The table reference is the index set.

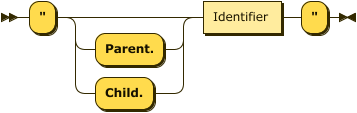
The exports object establishes the correspondences between the child’s tables and the parent’s. Similarly, the imports object establishes the correspondences between the parent’s tables and the child’s. Each correspondence is a pair of a table reference and a table definition, which can take one of three forms:

* another table reference if the two tables are defined identically in both the parent and child problems;
* a recipe if both tables are defined in the parent and child;
* a new table definition if only one of problems has a definition.

The use of a recipe enables such features as adding the index to an import or deleting the index from an export, since the index would not be present in the child, or renaming the fields of the transferred table.

A table reference identifies a table, which may disambiguated by prepending the table’s owner in the case of duplicate names in the parent and child.

Figure : **TableReference:**



TableReference::= '"' ( 'Parent.' | 'Child.' )? Identifier '"'

1. **Interactions with Solvers**

To be written

Discusses how the tables map to a solver’s native objects.

1. **Conclusion**

To be written