Common schema

Any text is valid in terms of syntax of the Opt2Code language but the code processing by the Opt2Code tool depends on the special syntax elements:

1) Core key syntax elements are "**[[**", "**]]**" and "**][**". To screen these elements in a text, the symbol "**|**" is used. The expression "**||**" is converted to "**|**", one symbol "**|**" converted to the empty string "", so that the expression "**|||**" converted into "**|**"", **||||**" into "**||**" and so forth. For instance, to screen expression "**[[**" one should use the expression "**[|[**".

2) **Operator (section)** and **argument expressions** are defined by an expression of the form "*[[nameOfOperator][arg1expr][arg2expr]...[argNexpr]]*", where "nameOfOperator" and "argXexpr" don't contain core syntax elements "**]]**", "**][**" and the symbol "**]**". So, the **name of operator (section)** and argument expressions (of this operator) are defined by expressions *nameOfOperator, arg1expr, .. , argNexpr* respectively, taken without the leading and trailing whitespaces. The expression of the form "*[[nameOfOperator][arg1expr][arg2expr]...[argNexpr]]*" is called an operator (section) **declaration**.

3) The operator (section) declaration is a (system) *reserved* one if it is one of the following expressions: "*[[sConfStart]]*", "*[[sTagTranslation]]*", "*[[sPropTranslation]]*","*[[sUserTagTranslation]]*", "*[[sUserPropTranslation]]*", "*[[sTagPropDeclaration]]*", "*[[sTagDef]]*", "*[[sDefDesc]]*","

[[sConfEnd]]". Otherwise, we call it a *user-defined* operator (section) declaration. Accordingly, the **operator (section)** is a (system) *reserved* one if it is defined by a *reserved* operator declaration. If not, we call it a *user-defined* **operator**.

User-defined operators are usually declared in the main body of the text, system *reserved* operators should be defined after the main text and they are used for setting translation **rules** and other settings.

According to paragraphs 2-3, we see that not embedded *user-defined* operator declarations in the text do not overlap since they do not contain the expression "**]]**" inside. To eliminate ambiguity and for some other reasons, the algorithm chooses the longest declaration of the *user-defined* operator.

4) The Opt2Code tool, to analyze the text, first, splits it into text fragments by system *reserved* declarations. The received first fragment of the text we call a *user-defined* part of the document. The fragment of the text between "*[[sConfEnd]]*" and the next system *reserved* declaration is also considered as a *user-defined* part of the document. The others we call *configuration* parts of the text. So, fragments of the text between *user-defined* parts of the document we call **configuration blocks** (they don't contain a *user-defined* part of the document). Each *user-defined* part of the document with the following **configuration block** (if has) are processed **independently** by the Opt2Code tool and results are joined.

Hereafter, without limiting generality, we will describe the algorithm for a document with a single *user-defined* part, unless otherwise stated. In the following paragraphs (5-9) we give a general scheme of the algorithm, which will be described in more depth in this document.

5) Each fragment of the text going after the *reserved* operator declaration till the next one or the end of the **configuration block** is considered as the **body** of the first operator (declaration). After analyzing **rules** defined in the **bodies** of *reserved* operators, the *user-defined* part of the document is splitted by *user-defined* operator declarations. Analogously, each fragment of the text going after an *user-defined* operator declaration till the next one (or the end of the *user-defined* part of the document) is considered as the **body** of the first operator (declaration).

6) For each *user-defined* operator the algorithm calculates its hierarchy level *hLevel*, a non negative integer, using the order of operator declarations and their values of the predefined properties: *sTagId, sPriorityLevel, sPrevSiblingId, sParentTagId, sOnlyPrevElemByPriority, sOnlyChildren, sHierarchyIgnored, sToTopLevel, sToSubLevel, sToTheSameLevel, sNoChild*. The default values of these properties for each operator can be redefined in the section "**sTagPropDeclaration**" and overridden by argument expressions of an operator.

7) For each *user-defined* operator x with hierarchy level y, we define *user-defined* child operators. Each of these operators satisfies the following conditions:
1. The operator is declared after x
2. Its hierarchy level is equal to y+1
3. All operators declared between x and the given operator have hierarchy level higher then y.
The algorithm calculates list of child operators in the order of their declarations for each *user-defined* operator. As a result, we get list of trees ordered by declarations of their roots (operators with hierarchy level 0 are without parent).

8) For each *user-defined* operator, we match it with a function using its name. This function can be defined in the **body** of the *reserved* operator declaration "*[[sTagDef]]*" or predefined in the system (i.e. defined somewhere else). This function uses values of properties of the given operator and list of the call results of functions, which are corresponding to the child operators of the given operator, as an argument. The list of the call results is empty if there is no child operator for the given operator.

9) Using the priority of operators, the Opt2Code tool builds an ordered sequence of tree structures of *user-defined* operators and executes corresponding functions for each operator in the order of the tree structure sequence, starting from the functions corresponding to the leaf operators and up to the root. Joining the outputs of the function invocations corresponding to the operators with hierarchy level 0 (i.e. roots) is the output of the algorithm. It can be a HTML code or Markdown based text or a list of objects in the memory used further to display formatted text.

Review of predefined section formats

1) The section **sConfStart** defines the start of the configuration part of the text in Opt2Code. The content of the body of this section is ignored.

2) The section **sTagTranslation** is reserved for transforming Opt2Code code into *digital* form. The system define substitution rules for matching old names of *user-defined* operators with their new numeric names. The format of records is

...
[[newNumericNameOfOperator]=oldNameOfOperator

[[newNumericNameOfOperator2]=oldNameOfOperator2
...

3) The section **sPropTranslation** is also reserved for transforming Opt2Code code into digital form. The system define substitution rules for matching old names of argument expressions of *user-defined* operators with their new numeric names. The format of records is

...
][numericArgExpr1]=oldArgExpr1
][numericArgExpr2]=oldArgExpr2
...

4) The section **sUserTagTranslation** is used for matching names of *user-defined* operators with their *canonical* names, which can be used in the reserved section "sTagDef" for matching *user-defined* operators with mappings. The format of records is

...
[[nameOfOperator1]=functionName1
[[nameOfOperator2]=functionName2
...

5) The section **sUserPropTranslation** is used for matching argument expressions into others. The resulting expressions are used to extract values for properties of *user-defined* operators. The format of records is

...
][argExpr1]=propName1=value1
][argExpr2]=propName2=value2
][argExpr3]=anotherArgExpr3
...

6) The section **sTagPropDeclaration** is used for setting default values for properties of operators. By default, the property **sPriorityLevel**=100 and the other properties are not set for *user-defined* operators. However, these settings can be redefined there. The format of records is

...
[[canonicalNameOfOperator1][prop1=value1][prop2=value2]...
[propN=valueN]...[argumentExprX]]
[[canonicalNameOfOperator2][prop=value][propX=valueX]...
[propK=valueK]...[argumentExprY]]
...

7) The section **sTagDef** is used for defining user-defined rules in JavaScript. This section is not used if we use an converter based on embedded rules. The format of records is

...
[[canonicalNameOfOperator1]=functionBody1 /* body of function in JavaScript depending on one argument "it", where "it" is an JavaScript object containing properties of the corresponding operator, context properties and the call results of functions corresponding child operators of the current one (it.children).*/
[[canonicalNameOfOperator2]= functionBody2
...

8) The section **sDefDesc** for documentation. The recommended format of records is

...
[[nameOfOperator1]- description of operator1
[[nameOfOperator2]- description of operator2
...
][argExp1]- description of argExp1
][argExp2]- description of argExp2

...
Using the recommended notation, we provide that any operator can be easily renamed by replacing an expression "*[[oldName]*" by "*[[newName]*". As for an argument expression, to rename it we replace an expression "*][oldArgumentExpression]*" by "*][newArgumentExpression]*".

9) The section *sConfEnd* is used to indicate the end of configuration block.

Predefined properties

The property **sTagId** is used as an identifier of *user-defined* operator. Its value can be used as a value for properties **sPrevSiblingId**, **sParentTagId**. The value for this property is not set by default.
The property **sValue** is reserved for *predefined* rules and future use. The value for this property is not set by default.
The algorithm uses the other system properties to define hierarchy level *hLevel* of *user-defined* operators. It analyzes list of *user-defined* operators in the order of their declarations and for each operator defines its *hLevel* checking the following conditions one by one until the first success:
1) It is the first element. In this case, *hLevel*=0
2) The property **sParentTagId** is set to *parentId* and the *user-defined* operator *P* with sTagId=*parentId* is found. In this case, *hLevel=hLevelOfP*+1
3) The property **sPrevSiblingId** is set to *prevSiblingId* and the *user-defined* operator *S* with sTagId=*prevSiblingId* is found. In this case, *hLevel=hLevelOfS*
4) The property **sToTopLevel** is set to true. In this case, *hLevel*=0
5) The property **sNoChild** is set to true for the previous *user-*

defined operator. In this case, *hLevel*=0

6) The property **sOnlyChildren** is set for the previous *user-defined* operator. The format of this property is a canonical names of *user-defined* operators delimited by comma. If the canonical name of the current operator is not in that enumeration, then *hLevel*=0

7) The property **sOnlyPrevElemByPriority** is set. The format of this property is a canonical names of *user-defined* operators delimited by comma. If the canonical name of the previous *user-defined* operator is not in that enumeration, then *hLevel*=0

8) There is not a previously defined *user-defined* operator with the property **sHierarchyIgnored** not equaled to true. In this case, *hLevel*=0

Let *U* be the last previously defined *user-defined* operator with the property **sHierarchyIgnored** not equaled to true.

9) The property **sToSubLevel** is set to true. In this case, *hLevel=hLevelOfU*+1

10) The property **sToTheSameLevel** is set to true. In this case, *hLevel=hLevelOfU*

11) The value of **sPriorityLevel** is higher then the priority level of *U*. In this case, *hLevel=hLevelOfU*+1

12) The value of **sPriorityLevel** is equal to the priority level of *U*. In this case, *hLevel=hLevelOfU*

13) The value of **sPriorityLevel** is less than the priority level of *U*. If there is not parent operator of *U*, than *hLevel*=0. Otherwise, we repeat steps 11-13 for the parent operator of *U* (i.e. for *U=ParentOfU*)

Algorithm for parsing predefined sections

For all predefined sections, with rules of the form **X***key***Y***value***Z**, where **X** is *[[* or *][*, **Y** is *]=* and **Z** is the end of the predefined section or the next expression of the same form (**X***key***Y***value***Z**)

we assume that *key* doesn't contain **X** and *value* is screened by the symbol "|". For all such rules, the algorithm extracts key and value without leading and trailing whitespaces and puts them into the hashmap structure corresponding to the predefined section. As for the section **sTagPropDeclaration**, for each rule, the algorithm extracts the *canonical* name of an operator *key* and extract names of properties with their values from argument expressions into a hashmap structure *value* and puts them into the hashmap structure corresponding to the predefined section.

Algorithm for extracting properties with their values from an argument expression:

1) We remove all leading and trailing whitespaces from the argument expression.

2) If the resulting expression doesn't contain the symbol "=", the name of property we define as the position number of the argument expression in a rule or user-defined operator declaration. The value would be the resulting expression itself.

3) If the resulting expression start with the symbol "=", the name of property we also define as position number of the argument expression in a rule or user-defined operator declaration, but the value would be the resulting expression without first symbol "=" and leading whitespaces.

4) Otherwise, we split the resulting expression by the first symbol "=" into 2 strings and remove from them leading and trailing whitespaces. The resulting pair would be the property name with its value.

Algorithm for parsing a user-defined part of the document

The algorithm analyzes list of *user-defined* operators in the order of their declarations. For each user defined-operator, it creates the corresponding object *Info* for retention of all information relating to him. From the user-defined operator declaration it extracts its name and list of argument expressions, without leading and trailing whitespaces. If this name is contained as a key in the map *X* corresponding to **sTagTranslation**, it uses the value *X*[key] instead of the initial name for the operator. If the resulting name is also contained as a key in the map *Y* corresponding to **sUserTagTranslation**, it uses the value *Y*[key] instead of the previous name. We call it the **canonical** name of a *user-defined* operator. Analogously, for each argument expression, the algorithm checks if it is contained as a key in the map *Z* corresponding to **sPropTranslation**. If yes, it uses the value *Z*[key] instead of the initial argument expression. If the resulting argument expression is contained as a key in the map *W* corresponding to **sUserPropTranslation**, it uses the value *W*[key] instead of the previous argument expression for extracting property and its value for the *user-defined* operator. Note that all properties defined in a operator declaration redefine default properties. This information is added to *Info*. We also keep the body of a user-defined operator there. After that, we calculate hierarchy level *hLevel* of the operator. If *hLevel*=0, we add *Info* to the resulting list. Otherwise, we add *Info* to the list of children to the parent *Info* and save a link to the parent object in the child one. As a result, we get a list of tree-based objects. The Opt2Code algorithm uses the hashmap structure defined by **sTagDef** or a custom mapping defined in the code to match each object *Info* in the forest (the list of tree-based objects) with a mapping *F* by the *canonical* name of user-defined operator. The mapping *F* uses *Info* (or a normalized version of object *Info*) and list of the call results of functions, which are corresponding to child

objects *Info* of the given object *Info*, as an argument. The list of the call results is empty if there is no child object *Info* for the given *Info*. The algorithm invokes the corresponding function *F* for each object *Info* in the forest, for each tree in the order of declarations of root elements in the list and starting from the functions corresponding to the leaf objects *Info* and up to the root. Joining the outputs of the function invocations corresponding to the operators with hierarchy level 0 (i.e. roots) is the output of the algorithm.

JavaScript code in the section sTagDef

As already mentioned, the rule declarations in this section have the form

...
[[canonicalNameOfOperator1]=functionBody1
[[canonicalNameOfOperator2]=functionBody2
...

For the body of a function in JavaScript, we inject the variable *it*, where *it* is an JavaScript object containing properties of the corresponding operator, context properties and the call results of functions corresponding child operators of the current one. You have access to this information through the following property names of the object *it*:

1) The property **elems** is list of root elements corresponding to *Info*.

2) The property **rules** is the key value map corresponding to the **sTagDef** section.

3) The property **name** is the canonical name of the user-defined operator

4) The property **text** is the body of the user-defined operator

5) The property **children** is list of call results corresponding to child user-defined operators.

6) The property **seq** is the position number of the user-defined operator in the text.
7) The property **props** is the property map for the user-defined operator.

Tools for working with Opt2Code code

The list of tools in the Converter section:
1) "*Code by user-defined rules*" - for transforming Opt2Code code into a text using JavaScript rules defined in the section **sTagDef**.
2) "*HTML view by user-defined rules*" - for a preview of Opt2Code code transformation in a browser, JavaScript rules should be defined in the section **sTagDef**.
3) "*Digital form*" - adds sections **sTagTranslation**, **sPropTranslation** to Opt2Code code with substitution rules from new digital notation for *user-defined* operators and argument expressions to old ones and renames them using new notation.
4) "*User form*" - removes sections **sTagTranslation**, **sPropTranslation** in Opt2Code code and use substitution rules from these sections to rename user-defined operators and argument expressions.
5) "*Canonical form*" - removes sections **sTagTranslation**, **sPropTranslation**, **sUserTagTranslation**, **sUserPropTranslation** in Opt2Code code and use substitution rules from these sections to rename user-defined operators and argument expressions.
6) "*Code by HTML style embedded rules*" - uses embedded rules which match any user-defined operator and algorithm expression to corresponding HTML tag with properties by their names.
7) "*HTML view by HTML style embedded rules*" - for a preview

of the Opt2Code code transformation in a HTML view. It uses embedded rules which match any user-defined operator and algorithm expression to corresponding HTML tag with properties by their names.

8) "*Code by embedded rules*" - for transforming Opt2Code code into text using embedded rules defined in the system. It is provided only for educational purposes to run some examples of Opt2Code code.

9) "*HTML view by embedded rules*" - for a preview of the Opt2Code code transformation, based on embedded rules, in a browser. It is provided only for educational purposes to run some examples of Opt2Code code.