```
/*************************************************************************
 * Copyright 2007-2020 - General Electric Company, All Rights Reserved
 *
 * Project: SADL
 *
 * Description: The Semantic Application Design Language (SADL) is a
 * language for building semantic models and expressing rules that
 * capture additional domain knowledge. The SADL-IDE (integrated
 * development environment) is a set of Eclipse plug-ins that
 * support the editing and testing of semantic models using the
 * SADL language.
 *
 * This software is distributed "AS-IS" without ANY WARRANTIES
 * and licensed under the Eclipse Public License - v 1.0
 * which is available at http://www.eclipse.org/org/documents/epl-v10.php
 *
 ************************************************************************/
grammar com.ge.research.sadl.SADL hidden(WS, ML_COMMENT, SL_COMMENT) //with with
org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate sADL "http://www.ge.com/research/sadl/SADL"

SadlModel :
    'uri' baseUri=STRING ('alias' alias=ID)? ('version' version=STRING)?
      annotations+=SadlAnnotation* EOS
    imports+=SadlImport*
    elements+=SadlModelElement*;

SadlAnnotation :
      ','? '(' type=('alias'|'note'|'see') contents+=STRING (',' contents+=STRING)* ')'
;

SadlImport :
    'import' importedResource=[SadlModel|STRING] ('as' alias=ID)? EOS;

// The various kinds of elements that make up the body of a model.
SadlModelElement :
             SadlStatement EOS
      |      ExpressionStatement => EOS
      |      RuleStatement => EOS
      |      QueryStatement => EOS
      |      UpdateStatement => EOS
      |      TestStatement => EOS
      |      PrintStatement => EOS
      |      ReadStatement => EOS
      |      ExplainStatement => EOS
      |      StartWriteStatement
      |      EndWriteStatement => EOS
      |   EquationStatement => EOS
      |   ExternalEquationStatement => EOS
;

EquationStatement:
```

```
	'Equation' EquationSignature
	(body = Expression)?
	('return' retval = Expression)?
	('where' where=Expression)?
;

ExternalEquationStatement:
	'External' EquationSignature
	uri = STRING
	('located' 'at' location=STRING)?
	('where' where=Expression)?
;

fragment EquationSignature returns AbstractSadlEquation:
	name=SadlResource (annotations+=NamedStructureAnnotation)* '('
(parameter+=SadlParameterDeclaration (',' parameter+=SadlParameterDeclaration)* )? ')'
		('returns' (returnType+=SadlReturnDeclaration (','
returnType+=SadlReturnDeclaration)*))? ':'
;

SadlParameterDeclaration:
		type=SadlPrimaryTypeReference name=SadlResource ('('
augtype=ExpressionParameterized<false,false>  ('{' units+=UNIT (',' units+=UNIT)* '}')?
')')?
//		type=SadlPrimaryTypeReference ('(' augtype=PropOfSubject<false,false>
('{' units+=UNIT (',' units+=UNIT)* '}')? ')')? name=SadlResource
//		('(' type=PropOfSubject<false,false>  ('{' units+=UNIT (',' units+=UNIT)*
'}')? ')' | type=SadlPrimaryTypeReference) name=SadlResource
	|	unknown='--'
	|	ellipsis='...'
;

SadlReturnDeclaration:
		type=SadlPrimaryTypeReference ('('
augtype=ExpressionParameterized<false,false>  ('{' units+=UNIT (',' units+=UNIT)* '}')?
')')?
//		('(' type=PropOfSubject<false,false> ('{' units+=UNIT (',' units+=UNIT)*
'}')? ')' | type=SadlPrimaryTypeReference)
	|	none ='None'
	|	unknown='--'
;

// These are the things that translate directly to OWL.
SadlStatement returns SadlStatement:
		SadlResource
		({SadlClassOrPropertyDeclaration.classOrProperty+=current} 'is' 'a'
			('top-level'? 'class'
			| 'type' 'of' superElement=SadlPrimaryTypeReference
facet=SadlDataTypeFacet?)
				(describedBy+=SadlPropertyDeclarationInClass+ | (','?
restrictions+=SadlPropertyRestriction)+)?
			|{SadlProperty.nameOrRef=current} primaryDeclaration?='is' 'a'
'property' (','? restrictions+=SadlPropertyRestriction)*
```

```
                    |{SadlProperty.nameOrRef=current} (','?
restrictions+=SadlPropertyRestriction)+
                    |{SadlSameAs.nameOrRef=current} 'is' 'the' 'same' 'as'
(complement?='not')? sameAs=SadlTypeReference
                    |{SadlDifferentFrom.nameOrRef=current} 'is' 'not' 'the' 'same' 'as'
notTheSameAs=SadlTypeReference
                    |{SadlInstance.nameOrRef=current} ('is' AnArticle
type=SadlTypeReference)? (listInitializer=SadlValueList |
propertyInitializers+=SadlPropertyInitializer+)?
                    |{SadlDisjointClasses.classes+=current} ('and'
classes+=SadlResource)+ 'are' 'disjoint'
                    )
      |     {SadlClassOrPropertyDeclaration}  '{' classOrProperty+=SadlResource (','
classOrProperty+=SadlResource)* '}' 'are'
                          (('top-level'? 'classes' | (oftype='types'  |
oftype='instances') 'of' superElement=SadlPrimaryTypeReference)
describedBy+=SadlPropertyDeclarationInClass*
                    |     {SadlDisjointClasses.types+=current} 'disjoint'
                    |     {SadlDifferentFrom.types+=current} (complement?='not')?
'the' 'same')
    | {SadlProperty} AnArticle? 'relationship' 'of' from=SadlTypeReference 'to'
to=SadlTypeReference 'is' property=SadlResource
    | AnArticle SadlTypeReference (
            {SadlInstance.type=current} instance=SadlResource?
propertyInitializers+=SadlPropertyInitializer*
      |     {SadlNecessaryAndSufficient.subject=current} 'is' AnArticle
object=SadlResource 'only' 'if' propConditions+=SadlPropertyCondition ('and'
propConditions+=SadlPropertyCondition)*)
;



SadlPropertyCondition :
      property=[SadlResource|QNAME] cond+=SadlCondition
;

SadlPropertyInitializer:
       ','? firstConnective=('with'|'has')? property=[SadlResource|QNAME]
(value=SadlExplicitValue | '(' value=SadlNestedInstance')')
//      ','? ('with'|'has')? (property=[SadlResource|QNAME] (value=SadlExplicitValue |
'(' value=SadlNestedInstance')')) | {SadlPropertyInitializer} ('data'
(valueTable=ValueTable | 'located' 'at' location=STRING))?
      |','? firstConnective='is' property=[SadlResource|QNAME] 'of'
type=[SadlResource|QNAME]
      |','? firstConnective='of' property=[SadlResource|QNAME] 'is'
(value=SadlExplicitValue | '(' value=SadlNestedInstance')')
;

SadlNestedInstance returns SadlInstance:
{SadlNestedInstance} (
            instance=SadlResource 'is' article=AnArticle type=SadlTypeReference
propertyInitializers+=SadlPropertyInitializer*
      |     article=AnArticle type=SadlTypeReference instance=SadlResource?
propertyInitializers+=SadlPropertyInitializer*)
```

```
;

SadlResource:
	name=[SadlResource|QNAME] annotations+=SadlAnnotation*
;

SadlDataTypeFacet :
		{SadlDataTypeFacet} ('(' | minInclusive?='[') min=FacetNumber? ','
max=FacetNumber? (maxInclusive?=']' | ')')
	|	regex=STRING
	|	'length' (len=FacetNumber | minlen=FacetNumber '-'
maxlen=(FacetNumber|'*'))
	|	'{' values+=FacetValue (','? values+=FacetValue)* '}'
;

FacetNumber :
	'-'? AnyNumber
;

FacetValue :
	STRING | FacetNumber
;

// TypeReferences
SadlTypeReference returns SadlTypeReference:
    SadlUnionType
;

SadlUnionType returns SadlTypeReference:
	SadlIntersectionType ({SadlUnionType.left=current} ('or')
right=SadlIntersectionType)*
;

SadlIntersectionType returns SadlTypeReference:
	SadlPrimaryTypeReference ({SadlIntersectionType.left=current} ('and')
right=SadlPrimaryTypeReference)*
;

SadlPrimaryTypeReference returns SadlTypeReference:
		{SadlSimpleTypeReference} type=[SadlResource|QNAME] list?='List'?
	|	{SadlPrimitiveDataType} primitiveType=SadlDataType list?='List'?
	|	{SadlTableDeclaration} 'table' '[' (parameter+=SadlParameterDeclaration
(',' parameter+=SadlParameterDeclaration)* )?']'
			('with' 'data' (valueTable=ValueTable | 'located' 'at'
location=STRING))?
	|	'(' SadlPropertyCondition ')'
	|	'{' SadlTypeReference '}'
;

// Built-in primitive data types
enum SadlDataType :
    string | boolean | decimal | int | long | float | double | duration | dateTime | time
| date |
```

```
    gYearMonth | gYear | gMonthDay | gDay | gMonth | hexBinary | base64Binary | anyURI
|
    integer | negativeInteger | nonNegativeInteger | positiveInteger | nonPositiveInteger
|
    byte | unsignedByte | unsignedInt | anySimpleType;

// Class declarations may also describe the class's properties.
SadlPropertyDeclarationInClass returns SadlProperty:
      ','? 'described' 'by' nameDeclarations+=SadlResource
restrictions+=SadlPropertyRestriction*;

SadlPropertyRestriction :
      SadlCondition
    |   {SadlTypeAssociation} ('describes'|'of') domain=SadlTypeReference
    |   {SadlRangeRestriction} ('has'|'with') ('a' singleValued?='single'
'value'|'values') 'of' 'type' ((typeonly=('class'|'data')) |
range=SadlPrimaryTypeReference facet=SadlDataTypeFacet?)
    |   {SadlIsInverseOf} 'is' 'the' 'inverse' 'of' otherProperty=[SadlResource|QNAME]
    |   {SadlIsTransitive} 'is' 'transitive'
    |   {SadlIsSymmetrical} 'is' 'symmetrical'
    |   {SadlIsAnnotation} 'is' 'a' 'type' 'of' 'annotation'
    |   {SadlDefaultValue} 'has' ('level' level=NUMBER)? 'default'
defValue=SadlExplicitValue
    |   {SadlIsFunctional} 'has' 'a' 'single' (inverse?='subject' | 'value')?
    |   {SadlMustBeOneOf} 'must' 'be' 'one' 'of' '{' values+=SadlExplicitValue (','
values+=SadlExplicitValue)* '}'
    |   {SadlCanOnlyBeOneOf} 'can' 'only' 'be' 'one' 'of' '{' values+=SadlExplicitValue
(',' values+=SadlExplicitValue)* '}'
;

SadlCondition :
            SadlAllValuesCondition
      |   SadlHasValueCondition
    |   SadlCardinalityCondition
;

SadlAllValuesCondition :
      'only' ('has'|'with') 'values' 'of' 'type' type=SadlPrimaryTypeReference
facet=SadlDataTypeFacet?;

SadlHasValueCondition :
      'always' ('has'|'with') 'value' (restriction=SadlExplicitValue | '('
restriction=SadlNestedInstance ')');

SadlCardinalityCondition :
      ('has'|'with')
      ('at' operator=('least'|'most') |'exactly')?
      cardinality=CardinalityValue ('value'|'values')
      ('of' 'type' type=SadlPrimaryTypeReference  facet=SadlDataTypeFacet?)?;

CardinalityValue :
      NUMBER | 'one'
;
```

```
SadlExplicitValue:
	SadlExplicitValueLiteral |
	=>({SadlUnaryExpression} operator=('-' | 'not') value=SadlExplicitValueLiteral)
;

SadlExplicitValueLiteral:
		SadlResource		// e.g., George
	|	{SadlNumberLiteral} literalNumber=AnyNumber (-> unit=UNIT)?
    |   {SadlStringLiteral} literalString=STRING
    |   {SadlBooleanLiteral} (truethy?='true'|'false')
    |   SadlValueList
	|	{SadlConstantLiteral} term=('PI'|'e'|'known')
;

UNIT: STRING | ID;

SadlValueList:
	{SadlValueList} '[' (explicitValues+=SadlExplicitValue (','
explicitValues+=SadlExplicitValue)*)? ']'
;

// These articles can appear before the property name and are indicative of the
functionality of the property or
//     the cardinality of the property on the class
AnArticle :
	IndefiniteArticle | DefiniteArticle;

IndefiniteArticle :
	'A'|'a'|'An'|'an'|'any'|'some'|'another';

DefiniteArticle :
	'The'|'the';

Ordinal :
		'first'
	|	'second' | 'other'
	|	'third'
	|	'fourth'
	|	'fifth'
	|	'sixth'
	|	'seventh'
	|	'eighth'
	|	'ninth'
	|	'tenth'
;

// This is primarily for debugging purposes. Any expression can be given after "Expr:"
to see if it is valid.
// Such an expression has no meaning in translation.
ExpressionStatement returns ExpressionScope :
	{ExpressionStatement}
	'Expr:' expr=Expression ('=>' evaluatesTo=STRING)?;

NamedStructureAnnotation :
```

```
	','? '(' type=SadlResource contents+=SadlExplicitValue (','
contents+=SadlExplicitValue)* ')'
;

RuleStatement returns ExpressionScope :
	{RuleStatement}
	('Stage' stage=NUMBER)? 'Rule' name=SadlResource
(annotations+=NamedStructureAnnotation)* ':'? ('given' ifs+=Expression)? ('if'
ifs+=Expression)? // (','? ifs+=Expression)*
	                                    'then' thens +=Expression // (','?
thens+=Expression)*
;

QueryStatement returns ExpressionScope :
	{QueryStatement}
	start = ('Ask'|'Graph') ((name=SadlResource
(annotations+=NamedStructureAnnotation)* )? ':'
	        (expr=(ConstructExpression | AskExpression | Expression) (':'
'['parameterizedValues = ValueRow']')?)
	  | srname=SadlResource
	)
;

UpdateStatement returns ExpressionScope :
	{UpdateStatement}
	'Update' ((name=SadlResource (annotations+=NamedStructureAnnotation)* )? ':'
expr=(UpdateExpression | Expression)
	        | name=SadlResource
	)
;

ConstructExpression returns Expression :
	{ConstructExpression}
	'construct' subj=SadlResource ','? pred=SadlResource ','? obj=SadlResource
'where' whereExpression=Expression
;

AskExpression returns Expression :
	{AskExpression}
	'ask' 'where' whereExpression=Expression
;

UpdateExpression returns Expression :
	{UpdateExpression}
	('delete' dData='data'? deleteExpression=Expression)?
	('insert' iData='data'? insertExpression=Expression)?
	('where' whereExpression=Expression)?
;

TestStatement returns ExpressionScope :
	{TestStatement}
	'Test:' tests+=Expression
;
```

```
PrintStatement :
	'Print:'
			(displayString=STRING
		|	model='Deductions'
		|	model='Model');

ExplainStatement :
	'Explain:'
			(expr=Expression
		|	'Rule' rulename=SadlResource)
			;

StartWriteStatement :
	write='Write:' (dataOnly='data')? '{'
;

EndWriteStatement :
	'}' 'to' filename=STRING
;

ReadStatement :
	'Read:' 'data' 'from' filename=STRING ('using' templateFilename=STRING)?
;


Expression returns Expression: // (1)
	SelectExpression<true, true>
;

NestedExpression returns Expression: // (1)
	SelectExpression<true, false>
;

SelectExpression<EnabledWith, EnableComma> returns Expression :
	->({SelectExpression}
	'select' distinct?='distinct'? ('*' | selectFrom+=SadlResource (','?
selectFrom+=SadlResource)*) 'where'
whereExpression=ExpressionParameterized<EnabledWith, EnableComma> (orderby='order'
'by' orderList+=OrderElement ->(','? (orderList+=OrderElement))*)?)
	| ExpressionParameterized<EnabledWith, EnableComma>
;

OrderElement :
	('asc' | desc?='desc')? orderBy=SadlResource;

// Real expressions start here
ExpressionParameterized<EnabledWith, EnableComma> returns Expression: // (1)
    {Sublist} AnArticle? 'sublist' 'of' list=OrExpression<EnabledWith, EnableComma>
'matching' where=OrExpression<EnabledWith, EnableComma>
    | OrExpression<EnabledWith, EnableComma>;

OrExpression<EnabledWith, EnableComma> returns Expression:
	AndExpression<EnabledWith, EnableComma> ({BinaryOperation.left=current} op=OpOr
right=AndExpression<EnabledWith, EnableComma>)*;
```

```
OpOr:
	'or' | '||';

AndExpression<EnabledWith, EnableComma> returns Expression:
	EqualityExpression<EnabledWith, EnableComma> ({BinaryOperation.left=current}
op=OpAnd right=EqualityExpression<EnabledWith, EnableComma>)*;

OpAnd:
	'and' | '&&';

EqualityExpression<EnabledWith, EnableComma> returns Expression:
	RelationalExpression<EnabledWith, EnableComma> ({BinaryOperation.left=current}
op=InfixOperator right=RelationalExpression<EnabledWith, EnableComma>)*;

InfixOperator :
	'=='
	| '!='
	| '='
	| 'is' ('not'? 'unique' 'in')?
	| 'contains'
	| 'does' 'not' 'contain'
;

RelationalExpression<EnabledWith, EnableComma> returns Expression:
	Addition<EnabledWith, EnableComma> ->({BinaryOperation.left=current}
=>op=OpCompare right=Addition<EnabledWith, EnableComma>)*;

OpCompare:
	'>=' | '<=' | '>' | '<';

Addition<EnabledWith, EnableComma> returns Expression:
	Multiplication<EnabledWith, EnableComma> ({BinaryOperation.left=current}
op=AddOp right=Multiplication<EnabledWith, EnableComma>)*;

AddOp:
	'+' | '-'
;

Multiplication<EnabledWith, EnableComma> returns Expression:
	Power<EnabledWith, EnableComma> ({BinaryOperation.left=current} op=MultiOp
right=Power<EnabledWith, EnableComma>)*;

MultiOp :
	'*'|'/'|'%'
;

Power<EnabledWith, EnableComma> returns Expression:
	PropOfSubject<EnabledWith, EnableComma> ({BinaryOperation.left=current} op='^'
right=PropOfSubject<EnabledWith, EnableComma>)*;

PropOfSubject<EnabledWith, EnableComma> returns Expression:
	ElementInList<EnabledWith, EnableComma> ->(
```

```
                ({PropOfSubject.left=current} of=('of'|'for'|'in')
right=PropOfSubject<EnabledWith, EnableComma>)
          | ->({SubjHasProp.left=current} (<EnableComma> comma?=',')?
(<EnabledWith>'with'|'has')? prop=SadlResource ->right=ElementInList<EnabledWith,
EnableComma>?)+)?
;

ElementInList<EnabledWith, EnableComma> returns Expression:
       UnitExpression<EnabledWith, EnableComma> |
       {ElementInList} 'element' (before?='before'|after?='after')?
element=UnitExpression<EnabledWith, EnableComma>
;

UnitExpression<EnabledWith, EnableComma> returns Expression:
       UnaryExpression<EnabledWith, EnableComma> ({UnitExpression.left=current}
unit=STRING)?
;

UnaryExpression<EnabledWith, EnableComma> returns Expression:
       PrimaryExpression<EnabledWith, EnableComma> |
   {UnaryExpression} op=('not' | '!' | 'only' | '-' | ThereExists)
expr=PrimaryExpression<EnabledWith, EnableComma>
;

ThereExists :
       ('there' 'exists')
;

// primary expressions are the atom units of expression in the grammar
PrimaryExpression<EnabledWith, EnableComma> returns Expression:
       '(' SelectExpression<EnabledWith, EnableComma> ')'
       | Name
       | {Declaration} article=AnArticle ordinal=Ordinal? type=SadlPrimaryTypeReference
              (
                     '[' arglist+=NestedExpression? (',' arglist+=NestedExpression)* ']'
                     |'length' len=FacetNumber ->('-' maxlen=(FacetNumber|'*'))?
              )?
       | {StringLiteral} value=STRING
       | {NumberLiteral} value=AnyNumber
       | {BooleanLiteral} value=BooleanValue
       | {Constant} constant=Constants
       | {ValueTable} valueTable=ValueTable;

Name returns SadlResource:
       {Name} name=[SadlResource|QNAME] ->(function?='(' arglist+=NestedExpression?
(',' arglist+=NestedExpression)* ')')?
;

       // the truth table
ValueTable:
       '[' row=ValueRow ']'
       // e.g., [George, 23, "Purple", 38.186111]
       | '{' '[' rows+=ValueRow ']' (','? '[' rows+=ValueRow ']')* '}'
       // e.g., {[George, 23, "Purple", 38.186111], [Martha, 24, "Pink", 45.203]}
```

```
;

BooleanValue:
	'true' | 'false';


Constants:
	'PI' | 'known' | 'e' | '--' | 'None' | 'a'? 'type' | DefiniteArticle? 'length' |
'count' | DefiniteArticle? 'index' | ('first'|'last'|AnArticle Ordinal?) 'element' |
'value';

ValueRow:
	explicitValues+=NestedExpression (',' explicitValues+=NestedExpression)*; //
e.g., George, 23, "Purple", 38.186111

DNAME:
	ID;

AnyNumber returns ecore::EBigDecimal :
	DecimalNumber EXPONENT?;

DecimalNumber returns ecore::EBigDecimal :
	NUMBER ;

EXPONENT returns ecore::EBigDecimal:
	('e' | 'E') ('-' | '+')? DecimalNumber;

EOS:
	'.';

QNAME :
	QNAME_TERMINAL | ID
;

terminal NUMBER returns ecore::EInt :
	'0'..'9'+;

terminal WS:
	('\u00A0' | ' ' | '\t' | '\r' | '\n')+;

terminal ID:
	'^'? ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '-' | '%'
| '~')* ;

terminal QNAME_TERMINAL:
	ID ':' ID;

terminal STRING:
	'"' ('\\' ('b' | 't' | 'n' | 'f' | 'r' | 'u' | '"' | "'" | '\\') | !('\\' | '"'))*
'"' |
	"'" ('\\' ('b' | 't' | 'n' | 'f' | 'r' | 'u' | '"' | "'" | '\\') | !('\\' | "'"))*
"'";

terminal ML_COMMENT:
```

```
        '/*'->'*/';

terminal SL_COMMENT:
        '//' !('\n' | '\r')* ('\r'? '\n')?;

terminal ANY_OTHER:
        .;
```