ModelCC

r2015 (October 2015)

User Manual

The ModelCC Development Team

www.modelcc.org

Contents

1 Introduction **3**

2 Model Specification **4**
- 2.1 Concatenation . . . 5
- 2.2 Selection . . . 5
- 2.3 Repetition . . . 6

3 Model Constraints **8**
- 3.1 Pattern Specification Constraints . . . 8
  - 3.1.1 The @Pattern annotation . . . 8
  - 3.1.2 The @Value annotation . . . 9
- 3.2 Delimiter Constraints . . . 10
  - 3.2.1 The @Prefix annotation . . . 10
  - 3.2.2 The @Suffix annotation . . . 11
  - 3.2.3 The @Separator annotation . . . 12
- 3.3 Cardinality Constraints . . . 12
  - 3.3.1 The @Optional annotation . . . 12
  - 3.3.2 The @Multiplicity annotation . . . 13
- 3.4 Evaluation Order Constraints . . . 14
  - 3.4.1 The @Associativity annotation . . . 14
  - 3.4.2 The @Composition annotation . . . 15
  - 3.4.3 The @Priority annotation . . . 16
- 3.5 Composition Order Constraints . . . 17
  - 3.5.1 The @Position Annotation . . . 18
  - 3.5.2 The @FreeOrder Annotation . . . 18
- 3.6 Reference Constraints . . . 19
  - 3.6.1 The @ID annotation . . . 19
  - 3.6.2 The @Reference annotation . . . 19
- 3.7 Custom constraints with the @Constraint Annotation . . . 20

4 Getting Started with ModelCC **21**
- 4.1 Setting up ModelCC . . . 21
  - 4.1.1 Downloading ModelCC . . . 21
  - 4.1.2 Setting up ModelCC in NetBeans . . . 21
  - 4.1.3 Setting up ModelCC in Eclipse . . . 21
- 4.2 Specifying a language model . . . 22
- 4.3 Generating and using a parser . . . 22
  - 4.3.1 Reading the language model . . . 22
  - 4.3.2 Generating a parser . . . 23
  - 4.3.3 Using the parser . . . 24
  - 4.3.4 Full example . . . 24

http://www.modelcc.org

1 Introduction

ModelCC is a model-based parser generator that decouples language specification from language processing. The idea behind model-based language specification is that, starting from a single abstract syntax model (ASM) representing the core concepts in a language, language designers would later develop one or several concrete syntax models (CSMs). These concrete syntax models would suit the specific needs of the desired textual or graphical representation. The ASM-CSM mapping could be performed, for instance, by annotating the abstract syntax model with the constraints needed to transform the elements in the abstract syntax into their concrete representation.

- Section 2 describes the building blocks used for model specification in ModelCC.
- Section 3 describes the model constraints supported by ModelCC, which declaratively define the features of the syntax of the formal language defined by an annotated abstract syntax model.
- Section 4 contains a tutorial on how to use ModelCC.

http://www.modelcc.org

2 Model Specification

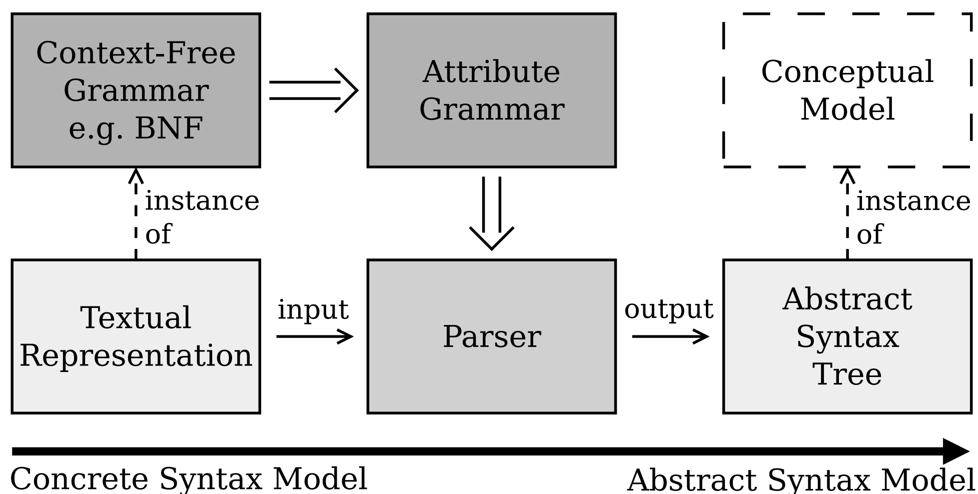

Figure 1: From the traditional approach...

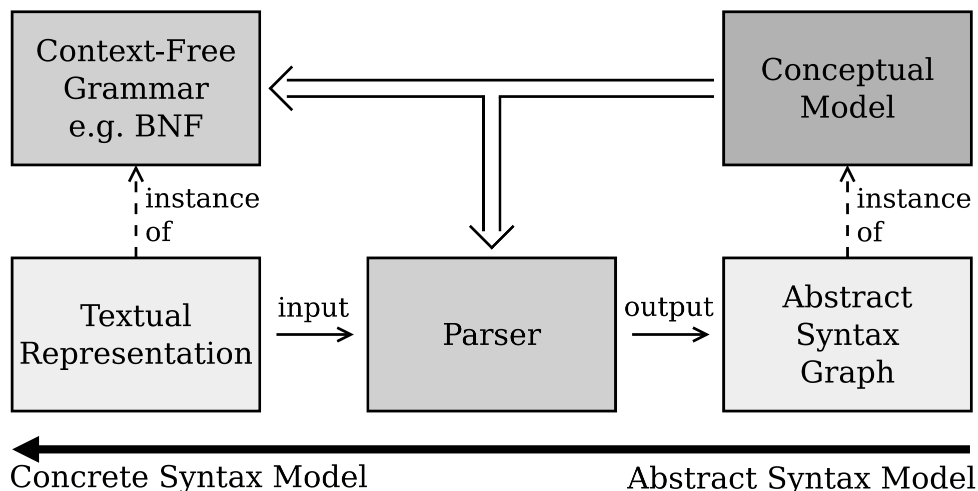

Figure 2: ... to ModelCC model-driven approach.

Any class implementing the *IModel* interface is considered a language element.

The specification of a language in ModelCC starts with the definition of basic language elements, which might be viewed as the tokens in the model-driven specification of a language. Basic language elements can be modeled as simple classes in an object-oriented programming language annotated with pattern constraints, as described in Section 3.1. The ASM-CSM mapping of those basic elements will establish their correspondence to the tokens that appear in the concrete syntax of the language whose ASM we design in ModelCC.

The metadata annotation *@Setup* can be used to specify the method, if any, that has to be run to initialize the language element instances.

Then, ModelCC provides the necessary mechanisms to combine those basic elements into more complex language constructs, which correspond to the use of concatenation, selection, and repetition in the syntax-driven specification of languages.

In the following subsections, we describe the mechanisms provided by ModelCC to implement the three main constructs that let us specify complete abstract syntax models on top of basic language elements.

2.1 Concatenation

Concatenation is the most basic construct we can use to combine sets of language elements into more complex language elements.

In ModelCC, concatenation is achieved by object composition. The resulting language element is the composite element and its members are the language elements the composite element collates.

The following code shows an example of object composition, in which an *AssignmentStatement* is composed of an *Identifier* and an *Expression*:

```
import org.modelcc.*;

public class AssignmentStatement implements IModel {

   Identifier id;

   Expression exp;

}
```

2.2 Selection

Selection allows the specification of alternative elements in language constructs.

In ModelCC, selection is achieved by subtyping. Specifying inheritance relationships among language elements in an object-oriented context corresponds to defining 'is-a' relationships. The language element we wish to establish alternatives for is the superelement, whereas the different alternatives are represented as subelements.

The following code shows an example of selection, in which an *Expression* can be either an *UnaryExpression*, a *BinaryExpression*, or a *ParenthesizedExpression*:

```
import org.modelcc.*;

public class Expression implements IModel {

}
```

```
import org.modelcc.*;

public class UnaryExpression extends Expression implements IModel {

}
```

```
import org.modelcc.*;

public class BinaryExpression extends Expression implements IModel {

}
```

```
import org.modelcc.*;

public class ParenthesizedExpression extends Expression
                                          implements IModel {

}
```

2.3 Repetition

Repetition allows the specification of a language element that appears several times in a given language construct.

In ModelCC, repetition is also achieved through object composition, just by allowing different multiplicities in the associations that connect composite elements to their constituent elements, that is, by declaring container members such as arrays, Lists/ArrayLists, or Sets/HashSets. The cardinality constraints described in Section 3.3 can be used to annotate ModelCC models in order to establish specific multiplicities for repeatable language elements.

The following five pieces of code show examples of repetition, in which an *OutputStatement* can contain an array, a List, an ArrayList, a Set, or an HashSet of *Expressions*.

```
import org.modelcc.*;

public class OutputStatement extends Statement implements IModel {

  Expression[] exp;

}
```

```
import org.modelcc.*;

public class OutputStatement extends Statement implements IModel {

  List<Expression> exp;

}
```

```
import org.modelcc.*;

public class OutputStatement extends Statement implements IModel {

  ArrayList<Expression> exp;

}
```

```
import org.modelcc.*;

public class OutputStatement extends Statement implements IModel {

  Set<Expression> exp;

}
```

```
import org.modelcc.*;

public class OutputStatement extends Statement implements IModel {

  HashSet<Expression> exp;

}
```

http://www.modelcc.org

3 Model Constraints

Once we have examined the mechanisms that let us create abstract syntax models in ModelCC, we now proceed to describe how constraints can be imposed on such models in order to establish the desired ASM-CSM mapping. As soon as that ASM-CSM mapping is established, ModelCC is able to generate the suitable parser for the concrete syntax defined by the CSM.

ModelCC allows the definition of constraints using metadata annotations or a domain-specific language.

In ModelCC, which supports language composition without being scannerless, a first set of constraints is used for pattern specification, a necessary feature for defining the lexical elements of the concrete syntax model, i.e. its tokens (Subsection 3.1).

A second set of constraints is employed for defining delimiters in the concrete syntax model, whose use is common for eliminating language ambiguities or just as syntactic sugar in many languages (Subsection 3.2).

A third set of ModelCC constraints lets us impose cardinalities on language elements, which control element repeatability and optionality (Subsection 3.3).

A fourth set of constraints lets us impose evaluation order on language elements, which are employed to declaratively resolve further ambiguities in the concrete syntax of a textual language by establishing associativity, precedence, and composition policies, the latter employed, for example, for resolving the ambiguities that cause the typical shift-reduce conflicts in LR parsers (Subsection 3.4).

A fifth set of constraints lets us specify the element constituent order in composite elements. (Subsection 3.5).

A sixth set of constraints lets us specify referenceable language elements and references to them (Subsection 3.6).

Custom constraints let us provide specific lexical, syntactic, and semantic constraints that take into consideration context information (Subsection 3.7).

3.1 Pattern Specification Constraints

Pattern specification constraints allow the specification of the lexical elements in a concrete syntax model, i.e. the different token types defined for the concrete syntax of a textual language. It should be noted that, once a language element is annotated with pattern specification constraints, it cannot be a composite element since, as a lexical element, it cannot be composed of other elements.

The above constraint, which forces lexical elements to be basic language elements in a ModelCC ASM, does not reduce the flexibility of ModelCC for language composition. Language composition, typically achieved by scannerless parser generators, can also be achieved if the scanner supports lexical ambiguities. When lexical ambiguities are allowed, the same string or even overlapping strings might correspond to several tokens, which will later be processed by the parser consuming the output of the scanner supporting lexical ambiguities.

3.1.1 The @Pattern annotation

The *@Pattern* annotation allows the specification of the pattern that will be used to match a basic language element in the input string. Two mutually exclusive mechanisms are provided for pattern specification in ModelCC: regular expressions and user-defined pattern matching classes. Regular expressions can be specified in ModelCC to build standard lexers, whereas custom pattern matching classes allow the language designer to use any custom-defined matching element to recognize basic language elements in the input string. The custom pattern matching class can be anything, since it works as a black box for ModelCC. It might even be a complete ModelCC-generated parser, which could be used for the specification of modular languages, a coarse form of

@Pattern(regExp="[a-zA-Z][_a-zA-Z0-9]*")

Identifier

Figure 3: Pattern specification example: Regular expression.

```
[a-zA-Z][_a-zA-Z0-9]*  return Identifier;
```

Figure 4: Implementation of Figure 3 in lex.

@Pattern(matcher=JavaDocRecognizer,args="simple")

JavaDoc

Figure 5: Pattern specification example: Custom pattern matching.

language composition (e.g. think of the JavaScript scripts and CSS stylesheets within the HTML in a web page).

When used with regular expressions, the *@Pattern* annotation includes an argument representing the regular expression. This regular expression, specified as the *regExp* field of the annotation, corresponds to the traditional token type definition in lex-like scanners.

When used with custom pattern matching classes, the *@Pattern* annotation is used to specify the name of the class implementing the matching algorithm and its argument string. If this pattern specification mechanism is used, ModelCC will need to resort to a scanning algorithm that supports pattern matcher classes.

For example, the *@Pattern* annotation in Figure 3 defines the typical *Identifier* token in programming languages, which can be specified by the following regular expression: *[a-zA-Z][_a-zA-Z0-9]**. This specification corresponds to the lex token definition shown in Figure 4.

The example in Figure 5 illustrates the use of custom pattern matching classes in ModelCC. In this case, the *@Pattern* annotation refers to the *JavaDocRecognizer* class, which will be responsible for recognizing *JavaDoc* comments as basic language elements in the shown ASM. Arguments can also be specified when using the *matcher* field of the *@Pattern* annotation with the help of its optional *args* field (*"simple"* in Figure 5).

As mentioned above, the ModelCC use of custom pattern matching algorithms for defining basic language elements has no counterpart in traditional lexer generators, hence an equivalent lex-like definition cannot be provided.

3.1.2 The @Value annotation

The *@Value* annotation can be used in ModelCC to indicate the location where the recognized token value will be stored in the abstract syntax graph, so that value can be used once the input string has been parsed.

Associated to a field of the class defining a basic language element, that field will contain the value obtained from the input string that matches the token type pattern specification.

When a numeric or boolean field is annotated with the *@Value* annotation, it is not necessary to specify the corresponding *@Pattern* annotation for recognizing the numeric or boolean tokens. When the *@Value*-annotated field is not numeric nor boolean (e.g. a string, a single character, or any non-primitive data type), the use of the *@Pattern* annotation is mandatory.

Elements from the ASM that contain a *@Value*-annotated numeric or boolean field are transformed into their corresponding token type lex-like definitions, even when no *@Pattern* annotation is present. ModelCC will always perform the proper assignment of the recognized token to the *@Value*-annotated field.

For example, the model in Figure 6 defines an *IntegerLiteral* language element that recognizes long integer literals. The proper regular expression for such literals will be employed and, whenever an integer literal is found in the input string, its integer value will be stored in the *@Value*-annotated *value* field of the *IntegerLiteral* class. If we had used lex & yacc, we would have had to type the lexical definition and semantic action shown in Figure 4.

As another example, Figure 8 shows how the *@Value* annotation would be used in conjunction with the *@Pattern* annotation to define string literals surrounded by double quotes. A *StringLiteral* will be recognized whenever a pair of double quotes encloses a string not containing a double quote, a constraint that can be

IntegerLiteral
~ @Value value : long

Figure 6: Value field specification example: Integer literals.

```
[0-9]+ {
  yylval.value = atoi(yytext);
  return INTEGERLITERAL;
}
```

Figure 7: Implementation of Figure 6 using lex & yacc.

@Pattern(regExp="\"[^\"]*\"")
StringLiteral
~ @Value text : String

Figure 8: Value field specification example: Double-quoted string literals.

```
\"[^\"]*\" {
  int size,csize;
  size = strlen(yytext);
  csize = sizeof(char)*(size+1);
  yylval.pval = malloc(csize);
  strncpy(yylval.pval,yytext,size);
  return STRINGLITERAL;
}
```

Figure 9: Implementation of Figure 8 using lex & yacc.

specified by the following regular expression: *\“[ˆ\“]*\“*. The lex & yacc implementation of this string literal token type definition would be slightly more complex than the previous example, as shown in Figure 9.

3.2 Delimiter Constraints

Delimiter constraints allow the specification of language element delimiters in a concrete syntax model. Delimiters include prefixes, suffixes, and separators. Such kinds of delimiters are often used to eliminate language ambiguities and facilitate parsing, but they can appear just as syntactic sugar to make languages more readable.

Usually, reserved words in programming languages act just as delimiters. As such, they will not appear in the language abstract syntax model. They will be specified as metadata annotations in the ASM-CSM mapping corresponding to the concrete syntax of the language.

It should be noted that delimiters should always be specified as constraints on the ASM-CSM mapping rather than language elements in the ASM, since they do not provide any relevant information from the perspective of the abstract syntax model, even though they might be necessary to define unambiguous textual languages.

3.2.1 The @Prefix annotation

The *@Prefix* annotation allows the specification of prefixes for language elements and specific constituents in composite language elements.

The *value* field of the *@Prefix* annotation is used to specify the list of regular expressions that define the prefixes that precede the corresponding language element (or a specific constituent element within a composite element) in the concrete syntax of a textual CSM.

When converting the ASM into a textual CSM, ModelCC will include the specified prefixes in every production where the annotated element appears in the textual CSM grammar, just before the appearance of the annotated element. When the annotation is associated to a constituent element within a composite language element, the sequence of prefixes will be included only in the productions that correspond to the composite language element, preceding the annotated constituent element within their right-hand side.

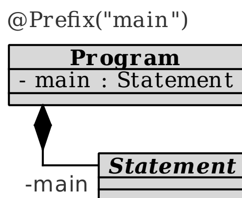

Figure 10: *@Prefix* annotation example.

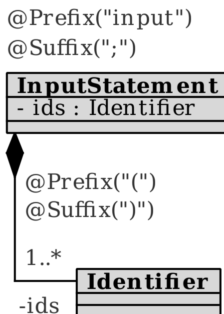

Figure 11: *@Suffix* annotation example.

It should be noted that, when the annotated element is repeatable, the sequence of prefixes appear only once, preceding the first instance of the annotated element. Prefixes will also be included in the CSM even when no elements appear in a repetition language construct (e.g. as when the opening parenthesis appears before an empty list of arguments in a parameterless C-like function call), but not when the repetition language construct is optional and is not present.

For example, the model in Figure 10 specifies that the textual representation of a *Program* will always be preceded by a "main" keyword prefix. The grammar defining the textual CSM for this simple example is *<ProgramMain> ::= "main" <Statement>*.

3.2.2 The @Suffix annotation

The *@Suffix* annotation allows the specification of suffixes for language elements and specific constituents in composite language elements.

The *value* field of the *@Suffix* annotation is used to specify the list of regular expressions that define the suffixes that follow the corresponding language element (or a specific constituent element within a composite element) in the concrete syntax of a textual CSM.

When converting the ASM into a textual CSM, ModelCC will include the specified suffixes in every production where the annotated element appears in the textual CSM grammar, just after the appearance of the annotated element. When the annotation is associated to a constituent element within a composite language element, the sequence of suffixes will be included only in the productions that correspond to the composite language element, following the annotated constituent element within their right-hand side.

It should be noted that, when the annotated element is repeatable, the sequence of suffixes appear only once, just after the last element in the sequence of repetitions. Suffixes will also be included in the CSM even when no elements appear in a repetition language construct (e.g. as when the closing parenthesis appears at the end of an empty list of arguments in a parameterless C-like function call), but not when the repetition language construct is optional and is not present.

For example, the model in Figure 11 specifies that the textual representation of an *InputStatement* is preceded by an "input" keyword prefix and followed by a semicolon (";") as its suffix. It also contains a sequence of *Identifier*s delimited by opening and closing parentheses: "(" as the prefix of the *ids* constituent of the *InputStatement* composite and ")" as its suffix. The grammar defined by the ASM-CSM mapping specified by the annotations in Figure 11 is *<InputStatement> ::= "input" "(" <IdentifierList> ")" "';"; <IdentifierList> ::= <Identifier> <IdentifierList> | <Identifier>*.

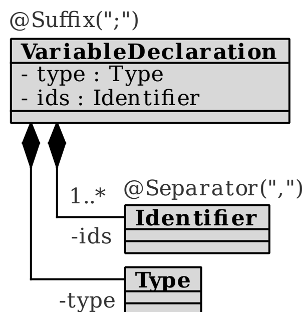

Figure 12: Default *@Separator* example.

3.2.3 The @Separator annotation

The *@Separator* annotation allows the specification of separators between consecutive instances of elements within a repetition. Separators can be defined in ModelCC by annotating a language element in the ASM or just its appearance within a particular repetition construct. In the first case, a default separator is established for the language element: the specified separator will be used for separating consecutive instances of the annotated language element whenever a sequence of such language elements appears in a textual CSM. In the second case, an *ad hoc* separator is defined: the specified separator will be used only when consecutive instances of the language element appear within the context of the annotated repetition construct.

The *ad hoc* definition of separators with the *@Separator* annotation within repetition constructs can be used to override or disable the default sequence of separators associated to the repeatable element in a repetition construct.

Therefore, default separators are specified for language elements by the *@Separator* annotation. When converting the ASM into a textual CSM, ModelCC will include the sequence of regular expressions defining those default separators in every recursive production rule generated from a repetition where the annotated element is repeatable. When the *@Separator* annotation accompanies a repeatable element within a particular repetition construct, i.e. the *ad hoc* case, separators will only appear in the recursive production rule derived from that particular repetition construct, but not in other constructs where the constituent element might also be repeatable.

As an example of defining a default separator, Figure 12 illustrates how a comma (",") can be used as the default separator for *Identifier*s. Whenever a list of *Identifier*s is needed within the language, a comma will separate consecutive identifiers. In the example, since a *VariableDeclaration* contains a *Type* and a set of *Identifier*s, they will be separated by "," in the textual CSM derived from the language ASM. The grammar of the resulting CSM will include the following productions: *<VariableDeclaration> ::= <Type> <IdentifierList> ";"* and *<IdentifierList> ::= <Identifier> "," <IdentifierList>* | *<Identifier>*.

As an example illustrating the use of *ad hoc* separators, consider the model in Figure 13. Here, *Identifier*s are also separated by commas, but only within *InputStatement*s, i.e. "," is the *ad hoc* separator for identifiers within input statements, but lists of identifiers might employ different separators elsewhere. The grammar associated to the textual CSM derived from Figure 13 would include the following productions: *<InputStatement> ::= "input" "(" <InputStatementIdentifierList> ")" ";"* and *<InputStatementIdentifierList> ::= <Identifier> "," <InputStatementIdentifierList>* | *<Identifier>*.

3.3 Cardinality Constraints

Cardinality constraints let us impose cardinalities on composite language elements, which control element repeatability and optionality.

3.3.1 The @Optional annotation

The *@Optional* annotation allows the specification of optional elements in textual CSMs.

Optional elements naturally appear in language specifications and optionality could always be modeled by means of selection constructs. However, the declarative specification of the optionality constraints is necessary to avoid unnecessary duplication in the language model.

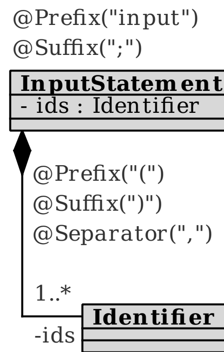

Figure 13: *Ad hoc @Separator* example.

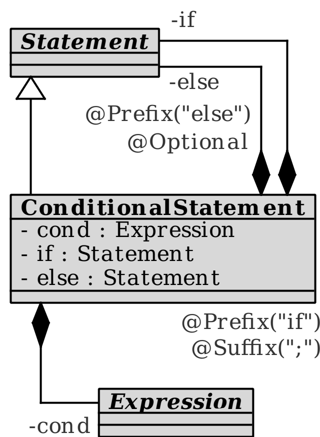

Figure 14: *@Optional* element example: if-then-else statement.

When one of the constituent elements within a composite language element is optional, the textual representation of the composite element might include the optional element along with its corresponding delimiters, or not. In the latter case, the missing element delimiters are not included in the textual representation either, even though a prefix and a suffix might have been defined for the missing constituent element.

If we performed a naive transformation of a composite language into a set of CFG production rules and the composite element includes i optional elements, 2^i production rules would result with the composite element in their left-hand side and all the possible combinations of optional elements in their right-hand side. A more reasonable transformation employs just $2i$ ancillary production rules that represent whether if each optional element is to be matched or not.

For example, the model in Figure 14 shows that a *ConditionalStatement* contains an *Expression*, the *Statement* that will be run when the *Expression* evaluates to true, and, optionally, the *Statement* that will be run when the *Expression* evaluates to false. The grammar resulting from the model transformation into a textual CSM will include the following two productions: *<ConditionalStatement> ::= "if" <Expression> <Statement> <OptionalElse>* and *<OptionalElse> ::= "else" <Statement>* | ϵ.

3.3.2 The @Multiplicity annotation

The *@Multiplicity* annotation, depicted as minimum and maximum multiplicity constraint in standard UML notation, allows the specification of the lower and upper bound for the multiplicity of repeatable language elements within repetition constructs. By default, the lower bound is 0 and the upper bound is infinite.

It should be noted that, when the minimum multiplicity is 0, no elements might appear in a particular instance of the repetition. However, delimiters would still be represented in the textual CSM unless the *@Optional* annotation were explicitly employed.

ModelCC generates semantic predicates that check if specific bounds are met for a list of elements, in which

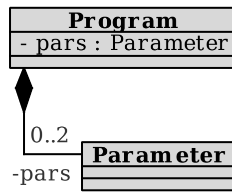

Figure 15: Maximum multiplicity example.

case the generated parser would not recognize the list of elements as such, since it does not satisfy the cardinality constraint imposed by the multiplicity annotation.

Other parser generators would require the explicit generation of a grammar representing the minimum and maximum cardinality constraint, i.e. when an element must appear at least i times within a repetition, two production rules would be necessary: *<MinIElements> ::= <Element>... i times... <Element> <ElementList>* and *<ElementList> ::= <Element> <ElementList>* | ϵ.

For example, the model in Figure 15 indicates that a *Program* might have from 0 to 2 *Parameter*s. Here, the multiplicity annotation is inferred from the multiplicity of the UML association. The grammar corresponding to the textual CSM derived from Figure 15 would include the following production rules: *<Program> ::= <OptionalParameterList>; <OptionalParameterList> ::= <ParameterList>* | ϵ*; and <ParameterList> ::= <Parameter> <ParameterList>* | *<Parameter>*, where the last production would be accompanied by a semantic predicate that would check whether the maximum multiplicity constraint holds. If such feature were not available in our parsing algorithm generator, this last production would have to be replaced by a much more explicit (and potentially verbose) set of equivalent productions incorporating the maximum multiplicity constraint: *<ParameterList> ::= <Parameter>* | *<Parameter> <Parameter>*. This approach poses no problems in this simple example, but it might get much more complicated (no problem yet whenever the resulting grammar is automatically generated by a model-driven language specification tool).

3.4 Evaluation Order Constraints

Evaluation order constraints let us declaratively resolve syntactic ambiguities in the concrete syntax of a textual language by establishing associativity, composition, and precedence constraints for CSMs.

3.4.1 The @Associativity annotation

The *@Associativity* annotation allows the specification of the operator-like associativity of language elements. ModelCC supports the following options for specifying associativity constraints:

- *UNDEFINED*, when no associativity is declared (by default), all possibilities are considered.
- *LEFT_TO_RIGHT*, for left-associative operations (e.g. subtraction, division, or function application).
- *RIGHT_TO_LEFT*, for right-associative elements (e.g. exponentiation and function definition).
- *NON_ASSOCIATIVE*, for non-associative elements (e.g. cross of three vectors).

The specification of associativity constrains help us resolve ambiguities that might appear in recursive compositions (i.e. when using the composite design pattern for modeling the ASM for operations without explicit delimiters), where different interpretations of the input string could be given unless the associativity constraints impose an order on the reductions that can be performed (either left-to-right or right-to-left).

For each production in the CSM grammar where the nonterminal of a language element with associativity constraints is preceded and/or followed by the nonterminal that appears on the left-hand side the production or any of its superclasses in the ASM, ModelCC generates a semantic predicate that enforces that associativity constraint. The associativity constraint can be implemented by inhibiting the reduction of a production in three situations:

- Whenever the element that follows a left-to-right associative element was generated by a reduction of the same production.

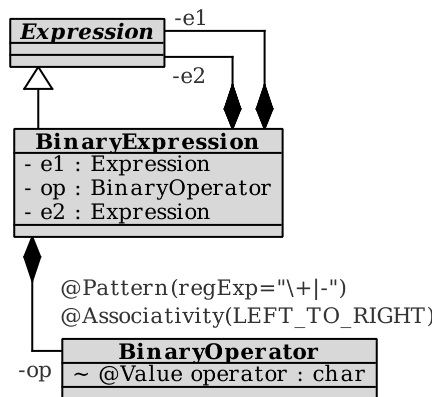

Figure 16: Associativity constraint example.

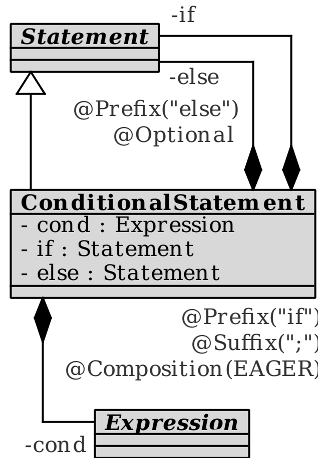

Figure 17: Composition constraint example.

- Whenever the element that precedes a right-to-left associative element was generated by a reduction of the same production.
- Whenever an element that precedes or follows a non-associative element was generated by a reduction of the same production.

For example, the model in Figure 16 establishes that *BinaryOperators* are left-associative. The grammar for the resulting textual CSM would include the productions *<Expression> ::= <BinaryExpression>* and *<BinaryExpression> ::= <Expression> <BinaryOperator> <Expression>*, where associativity is not explicit. Left-associativity will be imposed by the corresponding parser semantic predicate.

3.4.2 The @Composition annotation

The *@Composition* annotation allows the specification of the suitable order of evaluation of compositions represented in a CSM, a situation that appears whenever the composite design pattern is present in the ASM, no delimiters are employed to eliminate potential ambiguities, and the composite contains several consecutive components of the same type of the composite. When such composites are nested, different interpretations are possible unless we specify composition constraints in the CSM. This is the case of the typical shift-reduce conflicts that appear in LR parsers when parsing nested if-then-else statements.

Hence, a specific constraint on element composition must be used to enforce a particular interpretation of such nested compositions in the ASM-CSM mapping. ModelCC supports the following options for composition constraints:

```
@Separator(",")
@Pattern(regExp="[a-zA-Z][_a-zA-Z0-9]*")
Identifier
~ @Value name : String

@Priority(precedes={Identifier})
@Pattern(regExp="func_[a-zA-Z0-9]*")
FunctionName
~ @Value name : String
```

Figure 18: Relative priority resolved by the lexical analyzer.

- *UNDEFINED*, when no composition constraints are defined and potential ambiguities are taken into account.
- *EAGER*, when the matching of constituent elements is performed as soon as possible. This corresponds to the typical interpretation of nested if-then-else statements in programming languages, where the else clause is attached to the innermost if statement.
- *LAZY*, when the matching of constituent elements is deferred as much as possible. Then, a rightmost derivation is obtained; i.e. when an element might accompany any of two nested language constructs, it is associated to the outermost one.
- *EXPLICIT*, when no composition constraints are defined and any ambiguities should be resolved with the help of delimiters.

Composition order constraints are enforced by defining precedences for the productions in the grammar of the resulting textual CSM. Establishing such precedences is possible in most parsing tools, including all yacc derivatives and Fence. When composition is eager, shift operations will precede reduce operations. In contrast, when composition is lazy, reduce operations will have precedence over shift operations. Finally, when the composition order must be explicit in the CSM, the use of delimiters will determine whether shift or reduce operations are performed on a case by case basis.

For example, the model in Figure 17 represents typical if-then-else statements. In this case, the optional else *Statement* of the eager *ConditionalStatement* will always match the innermost if statement when such statements are nested, e.g. in "if E1 if E2 S1 else S2", the else clause will correspond to the E2 if statement. The grammar for the resulting CSM will include the following productions: *<ConditionalStatement> ::= "if" <Expression> <Statement> ";"* | *"if" <Expression> <Statement> "else" <Statement> ";"*. The parser will enforce the precedence of the second alternative over the first one, so that else clauses are parsed as usual.

3.4.3 The @Priority annotation

The *@Priority* annotation allows the specification of precedences among language elements for eliminating ambiguities in textual CSMs.

ModelCC implements two mechanisms to specify priority constraints in the ASM-CSM mapping:

- A relative one, where precedence relationships are established between particular language elements (a *precedes* declaration indicates which language elements have lower priority than the current element).
- An absolute one, where a numeric priority *value* determines the priority level for each language element (the lower the value, the higher the priority)

Unless specified otherwise, all language elements have the same priority. Precedences established among basic language elements are managed at the lexical analysis level by the corresponding lexer. Precedences established among non-basic language elements that appear in concatenation, selection, and repetition constructs within the CSM are managed at the syntactic analysis level by the corresponding parser.

For example, the model in Figure 18 establishes a (fictitious) relative priority constraint between function names and identifiers: a *FunctionName* will always precede an *Identifier*. In case a string like "func_power" is

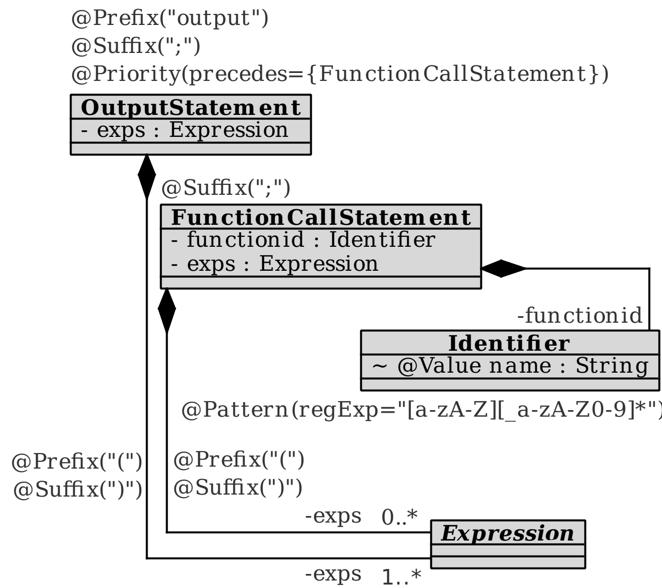

Figure 19: Relative priority resolved by the syntactic analyzer.

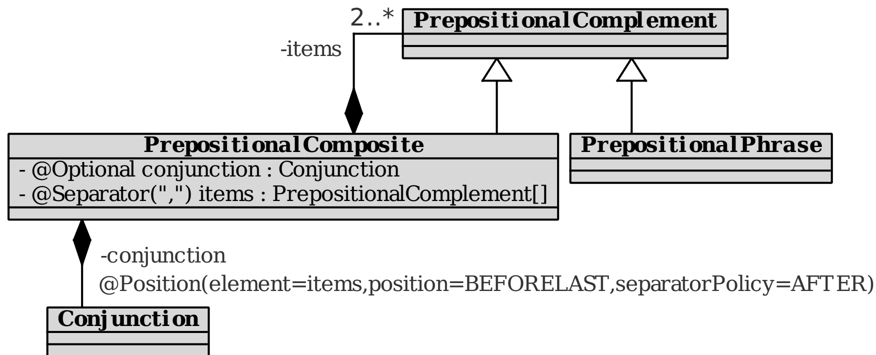

Figure 20: Position constraint example.

found in the input string, it will be recognized as a *FunctionName*, but not as an *Identifier*. Since the constraint is defined over basic language elements, the lexical analyzer generated by ModelCC will be responsible for identifying the right element in the input string.

Figure 19 shows another example. In this case, the model enforces relative priority constraints between composite language elements that will be resolved by the parser generated by ModelCC. Here, output statements precede function calls so that a string like “output(3+5,4+1);” will be recognized as an *OutputStatement* but not as an *FunctionCallStatement*, even when “output” would be a perfectly valid identifier. The ModelCC lexer, which supports lexical ambiguities, will consider “output” as an *Identifier* and also as a delimiter for output statements (i.e. its prefix keyword in the CSM).

3.5 Composition Order Constraints

The default order of the component elements in a composite element is given by the order in which the composition relationships were specified. However, composition order constraints let us override the element constituent order in composite elements.

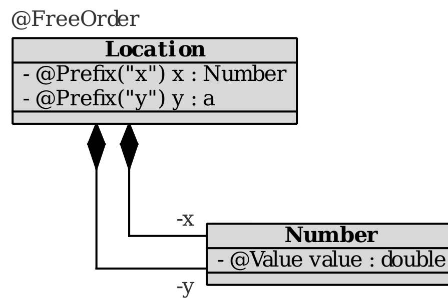

Figure 21: Free order constraint example.

3.5.1 The @Position Annotation

The *@Position* annotation allows the specification of the position of a component element relative to other component element. ModelCC supports combinations of the following options for specifying composition order:

- *BEFORE*, when the annotated element appears before the target element.
- *AFTER*, when the annotated element appears after the target element.
- *WITHIN*, when the annotated element appears as an embedded element in a repetition language construct.
- *BEFORELAST*, when the annotated element appears before the last element in a repetition language construct.

When the annotated element can appear as part of a repetitive construct (e.g. *WITHIN* or *BEFORELAST* in an enumeration), ModelCC allows the following separator policies:

- *BEFORE*, when the annotated element appears before the separator.
- *AFTER*, when the annotated element appears after the separator.
- *EXTRA*, when the annotated element appears between separators.
- *REPLACE*, when the annotated element replaces the corresponding separator.

The specification of position constraints allow the definition of CSMs in which element constituents are in different order than the composition relationships. Position constraints also allow embedding an element into a repetitive construct.

The implementation of position constraints in ModelCC consists in generating a set of grammar production that contains all the element orderings that comply with the constraints. When the target element in a position constraint is a repetitive construct, an ancillary production that includes the embedded element in the corresponding position is produced.

The model in Figure 20 presents an example in which a *PrepositionalComposite* is composed of two or more *PrepositionalComplement*s and an optional conjunction which, in case is present, precedes the last *PrepositionalComplement* in the construct. The grammar for the resulting textual CSM would include the production *<PrepositionalComplementList+Conjuntion> ::= <PrepositionalComplementList>* “,” *<Conjunction>* *<PrepositionalComplement>* and, finally, *<PrepositionalComplementList+Conjuntion> ::= <PrepositionalComplementList>* due to the optionality of the *Conjunction*.

3.5.2 The @FreeOrder Annotation

The *@FreeOrder* annotation allows the members of a language element to be shuffled in their textual representation.

When using the *@FreeOrder* annotation, ModelCC generates grammar productions for each of the valid orderings of constituent elements in the annotated element.

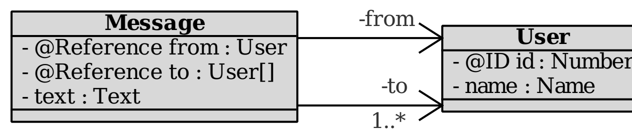

Figure 22: Reference constraint example.

As an example of the use of the @FreeOrder annotation, the model in Figure 21 specifies that a *Location* consists of an "x" coordinate and an "y" coordinate which may appear in any order. The grammar defining the textual CSM for this example contains *<Location> ::= "x" <Number> "y" <Number>* and *<Location> ::= "y" <Number> "x" <Number>*.

3.6 Reference Constraints

Reference constraints let us specify referenceable language elements and references to them.

References in ModelCC can be anaphoric, when they are preceded by the corresponding object definition, but also cataphoric, when the references precede the definition, and even recursive, when they appear within the definition they refer to.

3.6.1 The @ID annotation

The *@ID* annotation allows the specification of the identifier of referenceable language elements. This annotation is applied to a subset of the members of a language element model. This subset determines the syntax of references to particular instances of such elements in the concrete syntax of the corresponding language. That is, any appearance of the same set of values will be interpreted as a reference to the same instance of the referred language element.

It should be noted that the *@ID* annotation is incompatible with the *@Optional* ModelCC annotation, as null language element identifiers are not allowed, for the same reasons that attributes in a primary key are not nullable in a relational database. However, the *@ID* annotation can be used together with other ModelCC annotations, such as *@FreeOrder*, which allows the members of a language element to be shuffled in their textual representation, and *@Prefix* and *@Suffix*, which add syntactic sugar to the incarnation of the abstract syntax model as a concrete textual language.

The use of references is resolved in our implementation of ModelCC by the introduction of grammar productions that characterize such references and semantic predicates that map them to the corresponding language elements. The inadvertent definition of two entities of the same class with the same identifier results in a runtime warning produced by ModelCC when parsing its input.

In the model in Figure 22, the *@ID* annotation is employed to identify an *User* by a single number. The grammar for the resulting CSM will contain the traditional *<User> ::= <Number> <Name>* production with an associated semantic predicate that maps the *Number* identifier of the language element with a particular *User*, and also an additional *<UserRef> ::= <Number>* production with an associated semantic predicate that follows the mapping from the *Number* identifier to the particular referenced *User*, if possible.

3.6.2 The @Reference annotation

The *@Reference* annotation allows the specification of references to language elements. The *@Reference* annotation applies to individual members of any language element, provided that the referenced types contain at least one *@ID*-annotated member in their language model.

Whenever a language element member is annotated with *@Reference*, the corresponding grammar productions are modified so that they refer to the symbol corresponding to the element reference specification rather than the symbol that corresponds to its full specification. These productions are then associated to a semantic predicate that resolves the references at the end of the parsing process, in order to support cataphoric and recursive references, apart from the anaphoric references that could be resolved on the fly during the parsing process.

In the example model in Figure 22, the *@Reference* annotation is employed so that a *Message* contain references to the sender and the recipient *User*s. The grammar for the resulting CSM will contain, apart from

@Pattern(regExp="\"[^\"]*\"")

StringLiteral

~ @Value text : String

+ @Constraint checkEscapedCharacters()

Figure 23: Custom constraint example.

the aforementioned <*UserRef*> *::=* <*Number*> production and its associated semantic predicate, a <*Message*> *::=* <*UserRef*> <*UserRefList*> <*Text*> production, a recursive production of the form <*UserRefList*> *::=* <*UserRef*> <*UserRefList*> and a complementary production <*UserRefList*> *::=* <*UserRef*>.

3.7 Custom constraints with the @Constraint Annotation

While ModelCC supports a wide range of constraints, specific situations may require complex lexical, syntactic, or semantic constraints. ModelCC custom constraints allow the specification of actions that will determine, during the parsing process, whether an instance of a language element is valid or not.

The *@Constraint* annotation allows the definition of user-defined methods that are evaluated whenever an instance of the language element is created during the parsing. The constraining method can take into consideration any of the element members and must return a boolean from its evaluation. When the return value of the constraining method is "true", the element complies with the custom constraints and it is generated. Contrarily, when the return value of the constraining method is "false", the element does not comply with the custom constraints and it is inhibited from generating.

ModelCC directly translates custom constraints into lexer and parser runtime semantic predicates.

The model in Figure 23 presents an example in which the *@Constraint* annotation is employed so that the *checkEscapedCharacters* user-defined method checks that all escaped characters in a *StringLiteral* are valid (e.g. "\a" would yield a badly-escaped character error, since the character 'a' does not need escaping).

http://www.modelcc.org

4 Getting Started with ModelCC

This section explains how to use ModelCC.
Subsection 4.1 shows how to set ModelCC up.
Subsection 4.2 explains how to implement a language model.
Subsection 4.3 comments on how to generate and use a parser from a language model.
Section ?? enumerates ModelCC predefined types.

4.1 Setting up ModelCC

You can get started to ModelCC in a few very easy steps.

4.1.1 Downloading ModelCC

The most recent version of ModelCC can be downloaded from the ModelCC web site at `http://www.modelcc.org`. There, you can download the binary distribution package `modelcc-[version].zip` and read the ModelCC Shared Software License.

4.1.2 Setting up ModelCC in NetBeans

Whenever you want to use ModelCC in a NetBeans project, you should add the ModelCC library to your project:

1. Right click on your project.
2. Go to **Properties**.
3. Go to the **Libraries** tab.
4. In the **Compile** tab (which is open by default), click the **Add JAR/Folder...** button.
5. Go to the directory where you extracted **modelcc-[version].zip** and select **ModelCC.jar**. Do not select ModelCCExamples.jar, which is a stand-alone application for testing example languages.
6. Accept.

4.1.3 Setting up ModelCC in Eclipse

Whenever you want to use ModelCC in an Eclipse project, you should add the ModelCC library to your project:

1. Right click on your project.
2. Go to **Properties**.
3. Go to the **Java Build Path** section.
4. In the **Libraries** tab, click the **Add External JARs...** button.
5. Go to the directory where you extracted **modelcc-[version].zip** and select **ModelCC.jar**. Do not select ModelCCExamples.jar, which is a stand-alone application for testing example languages.
6. Accept.

4.2 Specifying a language model

The following class describes a very simple language. The entity **MySimpleLanguage** is a lexical entity that consists of an integer value, which is deducted from the **@Value** annotation that is assigned to an int data type. Any integer number pertains to this language.

```
import org.modelcc.*;

public class MySimpleLanguage implements IModel

    @Value
    private int data;

    public int getData()
        return data;
```

More complex example languages can be found in the Examples section of the ModelCC site (www.modelcc.org) and in the **ModelCCExamples** library, which is included in the source and binary distribution packages of ModelCC.

4.3 Generating and using a parser

The parser generation step consists of reading the model of a language and generating a parser from the model. Subsection 4.3.1 explains how to read the model of a language. Subsection 4.3.2 explains how to generate a parser from the language model. Subsection 4.3.3 explains how to use the parser.

4.3.1 Reading the language model

The *Model* class stores language models, which can be read by using a *ModelReader*.

ModelCC currently provides *JavaModelReader*, which is able to create a model from a set of annotated classes. The following code illustrates the usage of this model reader:

```
import org.modelcc.io.ModelReader;
import org.modelcc.io.java.JavaModelReader;
import org.modelcc.metamodel.Model;

[...]

//being MySimpleLanguage the main model class.

try {
  ModelReader jmr = new JavaModelReader(MySimpleLanguage.class);
  Model m = jmr.read();
} catch (Exception e) {
  System.out.println(e.getMessage());
}
```

When JavaModelReader is instantiated that way, it provides the *getWarnings()* method, which returns a list of Strings corresponding to the warnings generated during the model reading.

The following code illustrates another way to use this model reader:

```
import org.modelcc.io.java.JavaModelReader;
import org.modelcc.metamodel.Model;

[...]

//being MySimpleLanguage the main model class.

try {
  Model m = JavaModelReader.read(MySimpleLanguage.class);
} catch (Exception e) {
  System.out.println(e.getMessage());
}
```

4.3.2 Generating a parser

A *Parser* can be generated from a model.

ModelCC currently provides *ParserFactory*, which is able to produce a Fence parser with a subyacent Lamb lexer from a model and an optional *skip* model which determines the patterns that should be ignored by the lexer (i.e. comments). It should be noted that, in the current implementation, the Lamb lexer ignores any character that does not match a pattern, and the *skip* model can be used to ignore patterns that would be matched as other tokens. The following code illustrates the usage of this parser generator:

```
import org.modelcc.metamodel.Model;
import org.modelcc.parser.Parser;
import org.modelcc.parser.ParserFactory;

[...]

//being m the model.

try {
  Parser<MySimpleLanguage> parser = ParserFactory.create(m,ParserFactory.WHITESPACE);
} catch (Exception e) {
  System.out.println(e.getMessage());
}
```

The following code illustrates the usage of this parser generator with a *skip* model:

```
import org.modelcc.metamodel.Model;
import org.modelcc.parser.Parser;
import org.modelcc.parser.ParserFactory;

[...]

//being m the model.
//being skip the skip model.
//being MySimpleLanguage the main model class.

try {
  Parser<MySimpleLanguage> parser = ParserFactory.create(m,skip);
} catch (Exception e) {
  System.out.println(e.getMessage());
}
```

4.3.3 Using the parser

After the parser has been generated, it accepts strings or readers as input and produce instances of the main class. The parser provides several methods that produce different sets of results, considering one or several interpretation (in the case of ambiguities). The following code illustrates the usage of the parser:

```
import org.modelcc.parser.Parser;

[...]

//being parser a Parser<MySimpleLanguage>.
//being MySimpleLanguage the main model class.
//being input an input String or Reader.

MySimpleLanguage result = parser.parse(input);
```

The parser also provide the *parseAll* method, which return a *Collection* of instances of the main class, one for each possible interpretation; and the *parseIterator* method, which return an *Iterator* to the instances of the main class, which allows considering each possible interpretation.

4.3.4 Full example

The following code reads a model from a set of annotated Java classes, generates a parser, and uses it to generate an object instance from an input string:

```
import org.modelcc.io.java.JavaModelReader;
import org.modelcc.metamodel.Model;
import org.modelcc.parser.Parser;
import org.modelcc.parser.ParserFactory;

[...]

try {

  // Read the language model.
  Model m = JavaModelReader.read(Expression.class);

  // Generate a parser from the model.
  Parser<Expression> parser = ParserFactory.create(m);

  // Parse an input string.
  Expression result = parser.parse("3+(2+5)");

  // Print output.
  System.out.println(result.eval());

  // Print metadata.
  Map<String,Object> metadata = parser.getParsingMetadata(result);
  System.out.println("Start index: "+metadata.get("startIndex"));
  System.out.println("End index: "+metadata.get("endIndex"));

} catch (Exception e) {
  System.out.println(e.getMessage());
}
```