PTC

Design Note

Association Type Identifier Role Processing

Jeff Clark

02/05/14

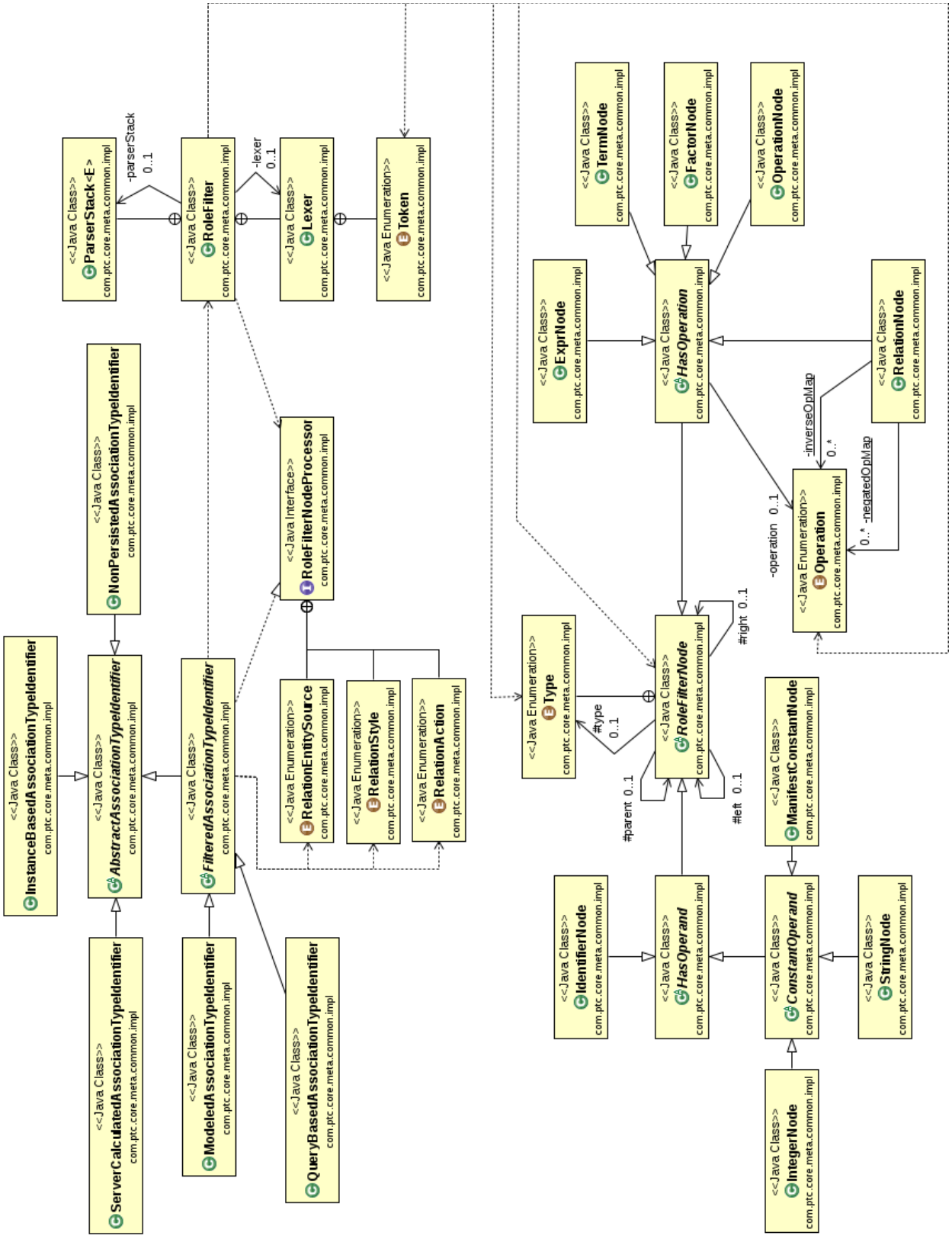
Summary:

This paper describes the processing of Association Type Identifier “roles”. In particular, it describes the theory and design of the parser for “role filter” strings that are used in the external form for some subtypes of AssociationTypeIdentifier. It also describes the requirements and implementation for inverting these filter string representations, and the processing of the internal (parse tree) form of these filters to produce expression evaluators (aka AttributeContainerSets).

At the time of this writing, the only subclasses of AbstractAssocationTypeIdentifier that use role filters are the ModeledAssociationTypeIdentifier and the QueryBasedAssociationTypeIdentifier.

Design Note: AssociationTypeIdentifier Role Filter Parsing

The classes involved in processing role filters are shown in the class diagram below:



Background

QueryBasedAssociationTypeIdentifiers are defined in the paper “Multi-language support for attribute values in Windchill: Approach”, by Joe McKinley, March, 26, 2013.

Role Filter

ModeledAssociationTypeIdentifers and QueryBasedAssociationTypeIdentifiers have roles which may contain filter specifications that define the association. An example of an MBA with a role filter is:

```
WCTYPE|wt.part.WTPartMaster~MBA|masterReference[(name='foo')AND
(iterationInfo.latest=TRUE)]@WCTYPE|wt.part.WTPart
```

This MBA has a role named 'masterReference'. The text enclosed within the '[ ]' is the role filter. (The '[' and ']' characters are *not* part of the role filter.)

An example QBA with a role filter is:

```
WCTYPE|wt.part.WTPart~QBA|
[(foo=masterText)AND(authoringLanguage=sourceLanguage)AND(123456789=dictionaryReferen
ce.key.id)AND('en_us'=targetLanguage)AND(thePersistInfo.modifyStamp>effectiveStart)AN
D((NULL=effectiveEnd)OR(thePersistInfo.modifyStamp<=effectiveEnd))]@WCTYPE|
com.ptc.core.td.server.dictionary.TranslationDictionaryEntry
```

The filter in the QBA does not have a name component, but it is still enclosed in '[' and ']' characters.

Language Grammar and Lexical Rules

Role filter strings are described by the following grammar:

```
 1. <expression>  ::= <term>
 2. <expression>  ::= <expression> 'OR' <term>
 3. <term>        ::= <factor>
 4. <term>        ::= <factor> 'AND' <term>
 5. <factor>      ::= '(' <expression> ')'
 6. <factor>      ::= <relation>
 7. <factor>      ::= 'NOT' <factor>
 8. <relation>    ::= '(' <id> <relop> <id> ')'
 9. <relation>    ::= '(' <id> <relop> <constant> ')'
10. <relation>    ::= '(' <constant> <relop> <id> ')'
```

The lexical tokens of this language are:

```
(           Left parenthesis
)           Right parenthesis
NULL        Manifest constant
TRUE        Manifest constant
FALSE       Manifest constant
AND         Boolean operator
```

```
OR          Boolean operator
NOT         Boolean operator
<           Relational operator
<=          Relational operator
=           Relational operator
!=          Relational operator
>=          Relational operator
>           Relational operator

'<legal character sequence>'  Literal String
[0-9]+                        Integer Constant
[a-zA-Z0-9_\$\[\]\.\-]+       Identifier
```

Literal strings are sequences of characters delimited by single quote characters ('). String literals may *not* include the following characters:

```
'~'   tilde
'@'   at sign
'^'   circumflex
'\''  single quote
' '   spaces
'\t'  tabs (or any other “white space” characters)
```

Integer constants are any sequence of decimal digits.

Identifiers are sequences of letters, digits, and the following special characters:

```
'_'   underscore
'$'   dollar sign
'['   left square bracket
']'   right square bracket
'.'   dot or period
'-'   dash or hyphen
```

An identifier may neither start nor end with a '.', nor may it contain two (or more) consecutive '.' characters.

The following are some examples of legal role filter strings:

- `(A.1=B)AND(TRUE=D)`
- `(A>B)AND(C!=NULL)`
- `(A!=B)AND(NOT(C=D))`

- `(A=FALSE)AND((C=D)OR(E='654321'))`
-
```
(foo=masterText)AND(authoringLanguage=sourceLanguage)
      AND(123456789=dictionaryReference.key.id)
      AND('en_us'=targetLanguage)
      AND(thePersistInfo.modifyStamp>effectiveStart)
      AND((NULL=effectiveEnd)
            OR(thePersistInfo.modifyStamp<=effectiveEnd))
```

Note that role filter strings *may not* contain any whitespace characters. The last example above is split across multiple lines for readability, but neither whitespace nor newline characters are allowed in role filter strings.

Note that the surrounding "()" are required. For example, this string:

- `A=B  -- Warning: Illegal`

... is not a legal role filter string.

Shift-Reduce Parsing

A role filter string is used to create a RoleFilter object. When the RoleFilter is created, the role filter string is parsed into an internal (tree) representation that is easy to process.

Parsing a simple expression language like the one describing role filters is usually performed using *shift-reduce parsing*. A subtype of shift-reduce parsing, known as *operator precedence parsing* is particularly suitable for this purpose. The usual approach is to use a table-driven parser; the table is known as the *operator precedence matrix* (or table). Because the grammar for this language is so simple, it is not necessary to use a table-driven parser or compute the operator precedence matrix.

The only ambiguity in this grammar arises from productions #3 and #4 (see above). When the top of the parse stack contains a <factor>, should the parser reduce it to a <term> or wait to see if the next input token is an AND? This ambiguity is easily resolved by using a one-token "look-ahead" when the parser is deciding whether to *shift* or *reduce*. If the parser sees a <factor> on the top of the stack and the next input token is 'AND', it shifts the 'AND' onto the stack; otherwise, it will reduce the <factor> to a <term>.

In all other cases, the rule is "if you can reduce the top of the stack, do so; otherwise, shift."

Design of the Parser

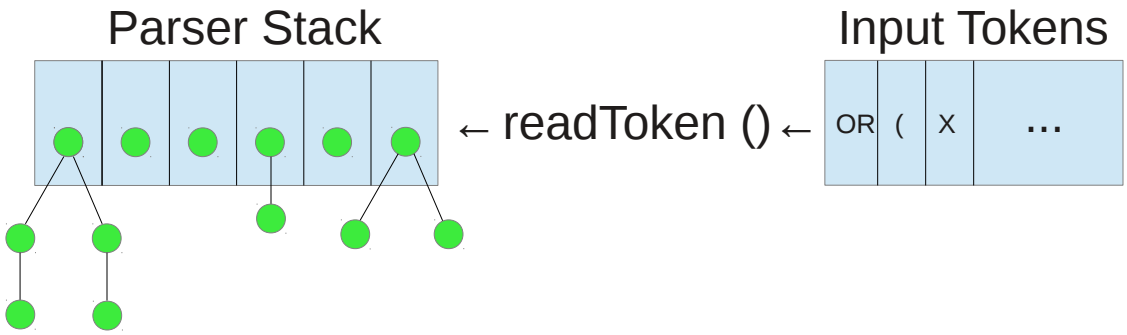
The parser operates in a simple loop, described by the Java code fragment below:

```
// Do shift/reduce parsing
Node currentToken = readToken ();
while (currentToken != null)
{
    parserStack.push (currentToken); // Shift current token onto stack
    Node nextToken = readToken (); // Get the "Lookahead" token
    doReductions (nextToken); // Perform all possible reductions
    currentToken = nextToken;
}
```

```
// Input is exhausted; perform all possible reductions
doReductions (null);

// Was the parse successful?
if (isAcceptingState ())
{
    parserStack.peek ().cleanTree (false);
    return top; // SUCCESS !!!
}
```

The parser reads lexical tokens from the input string and converts them to “Nodes”, which are pushed onto the *parser stack.* It then attempts to *reduce* the Nodes on the top of the stack using one of the productions from the grammar above.



For example, if the five Nodes on the top of the stack (from bottom to top) represent the input tokens '(', 'A', '=', 'B', and ')', then the parser will “reduce” these five nodes to just one (a <relation> Node), using production #8. Those five Nodes will be removed from the stack and replaced with the new <relation> Node.

Not all of the Nodes that are removed from the stack are discarded. Refer to the diagram above. Each Node in the stack is the root of a tree of Nodes. (Some “trees” consist of just a single Node.) In this case, the right-most Node now represents the new <relation> Node that was just “pushed” onto the parser stack in place of the five Nodes that were removed. Its children will be Nodes representing the *identifiers* 'A' and 'B'. The <relation> node will have an *operation* of “EQ”, representing the “=” Node that was previously on the parser stack. The Nodes representing the parentheses are discarded, as they are no longer needed.

The parser will continue to examine the nodes on the top of the stack, attempting to reduce them according to the rules of the grammar. When it can no longer perform a reduction, it will *shift* the next token from the input string onto the stack.

The parser performs only four actions: shift, reduce, accept, reject. When the parser can no longer shift or reduce, it examines the top of the stack. If the stack contains just one Node, and that Node is an <expression>, then the input string is a legal sentence in the language defined by the grammar, and the parser accepts the input. The Node at the top of the stack is the root of the *parse tree* that is the result of the parse.

If the parser exhausts the input string and cannot perform any reductions, and the stack contains more than one Node or the single remaining Node is not an <expression>, then the input role filter string is illegal and the parser rejects it. It is also possible for the parser to reject the string earlier in the processing, if it discovers that the input string cannot be a legal sentence in the language.

Cleaning the Tree

Given the design of the role filter language grammar, the parse tree resulting from an accepted input string may contain some "useless" Nodes. These are nodes that have no *operation* and have only one child. For example a <factor> Node may have a single child that is a <relation> Node. If the <factor> Node has no operation, it is useless. When the parse is complete and the input string has been accepted, the parser performs a *cleaning pass* over the tree that will remove the useless <factor> Node by linking its parent Node to the <factor> Node's child <relation> Node.

Note that not all <factor> Nodes, nor all Nodes with a single child are useless. For example, the role filter fragment "... NOT (A=B)" will result in a <factor> node that has just a single <relation> child, but the operation of the <factor> Node will be "NOT". In this case, both the <factor> Node and its child <relation> Node will be retained.

There is one exception to the rule that no-operation nodes are removed from the tree: the root node of the parse tree for an accepted input string will always be an <expression> Node; that node may or may not have an operation, depending upon the input expression.

Implementation

The role filter parser is implemented in the RoleFilter class in the com.ptc.core.meta.common.impl package in Foundation/src. The lexer used by the parser to break the input string into lexical tokens is nested within the RoleFilter class, and is *package-private*, as it is not needed by clients of RoleFilter. The Node class is RoleFilterNode and is contained in the same package.

The tests for these classes are in RoleFilterTest, in the same package in FoundationTest/src_test.

Input-Output Example

Given this input string (note that spaces are *not* allowed; they are included here for readability):

```
(foo=masterText) AND  (authoringLanguage=sourceLanguage)
AND  (123456789=dictionaryReference.key.id)
AND  ('en_us'=targetLanguage)
AND  (thePersistInfo.modifyStamp>effectiveStart)
AND  ((NULL=effectiveEnd) OR (thePersistInfo.modifyStamp<=effectiveEnd))
```

The following parse tree will result after cleaning the tree (the numbers to the left of each Node indicate its "depth" in the tree):

```
0: EXPR:
    1: TERM:  Op = AND
        2: RELATION:  Op = EQ
            3: IDENTIFIER:  Value = "foo"
            3: IDENTIFIER:  Value = "masterText"
```

```
2: TERM:  Op = AND
    3: RELATION:  Op = EQ
        4: IDENTIFIER:  Value = "authoringLanguage"
        4: IDENTIFIER:  Value = "sourceLanguage"
    3: TERM:  Op = AND
        4: RELATION:  Op = EQ
            5: INTEGER:  Value = "123456789"
            5: IDENTIFIER:  Value = "dictionaryReference.key.id"
        4: TERM:  Op = AND
            5: RELATION:  Op = EQ
                6: STRING:  Value = "'en_us'"
                6: IDENTIFIER:  Value = "targetLanguage"
            5: TERM:  Op = AND
                6: RELATION:  Op = GT
                    7: IDENTIFIER:  Value = "thePersistInfo.modifyStamp"
                    7: IDENTIFIER:  Value = "effectiveStart"
                6: EXPR:  Op = OR
                    7: RELATION:  Op = EQ
                        8: CONSTANT:  Value = "NULL"
                        8: IDENTIFIER:  Value = "effectiveEnd"
                    7: RELATION:  Op = LE
                        8: IDENTIFIER:  Value = "thePersistInfo.modifyStamp"
                        8: IDENTIFIER:  Value = "effectiveEnd"
```

Parse Example

Given the input string:

```
(A=B)AND(C=D)
```

... the parser would perform the following steps to accept the input string and produce the parse tree result shown below.

Step	Stack	Input	Action
1	*empty*	“(A = B) AND (C = D)”	Shift
2	(	“A = B) AND (C = D)”	Shift
3	(A	“= B) AND (C = D)”	Shift
4	(A=	“B) AND (C = D)”	Shift
5	(A=B	“) AND (C = D)”	Shift
6	(A=B)	“AND (C = D)”	Reduce 8
7	<relation>	“AND (C = D)”	Reduce 6
8	<factor>	“AND (C = D)”	Shift
9	<factor> AND	“(C = D)”	Shift
10	<factor> AND (	“C = D)”	Shift
11	<factor> AND (C	"= D)”	Shift
12	<factor> AND (C=	"D)”	Shift
13	<factor> AND (C=D	")”	Shift
14	<factor> AND (C=D)	“”	Reduce 8
15	<factor> AND <relation>	“”	Reduce 6
16	<factor> AND <factor>	“”	Reduce 3
17	<factor> AND <term>	“”	Reduce 4
18	<term>	“”	Reduce 1
19	<expression>	“”	Shift
20	<expression>	“”	Accept

The final *shift* action that occurs in step 19 recognizes the imaginary “end of input” token that is automatically a part of every input string submitted to the parser.

The resulting parse tree would be:

```
0: EXPR:
    1: TERM:  Op = AND
        2: RELATION:  Op = EQ
            3: IDENTIFIER:  Value = "A"
            3: IDENTIFIER:  Value = "B"
        2: RELATION:  Op = EQ
            3: IDENTIFIER:  Value = "C"
            3: IDENTIFIER:  Value = "D"
```

Inverting a Role

AssociationTypeIdentifier roles must be invertible. Different rules may apply when inverting ATI roles of differing subtypes, however, for all ATI subtypes the following condition must be true:

X.invert().invert() == X

where X represents the ATI's role.

Note that *equality* is defined in terms of *logical equality*, not *textual equality*. That is, two differing textual representations of a role may reduce to the same logical value, and these are considered “equal”. For example, the following role filter strings are equivalent:

- (A=B)AND(C=D)
- ((A=B)AND(C=D))
- (((A=B))AND(C=D))

When inverting the role, the relational operators are inverted and the operands are transposed. For example, the inverse of:

```
WCTYPE|wt.part.WTPartMaster~MBA|
masterReference[(name='foo')AND(iterationInfo.latest=TRUE)]
@WCTYPE|wt.part.WTPart
```

is:

```
WCTYPE|wt.part.WTPart~MBA|
[('foo'=name)AND(TRUE=iterationInfo.latest)]masterReference
^WCTYPE|wt.part.WTPartMaster
```

And the inverse of:

```
WCTYPE|wt.part.WTPart~QBA|
[(foo=masterText)
       AND(authoringLanguage=sourceLanguage)
       AND(123456789=dictionaryReference.key.id)
       AND('en_us'=targetLanguage)
       AND(thePersistInfo.modifyStamp>effectiveStart)
```

```
        AND((NULL=effectiveEnd)OR(thePersistInfo.modifyStamp<=effectiveEnd))]
@WCTYPE|com.ptc.core.td.server.dictionary.TranslationDictionaryEntry
```

is:

```
WCTYPE|com.ptc.core.td.server.dictionary.TranslationDictionaryEntry~QBA|
[(mastertext=foo)
        AND(sourceLanguage=authoringLanguage)
        AND(dictionaryReference.key.id=123456789)
        AND(targetLanguage='en_us')
        AND(effectiveStart<thePersistInfo.modifyStamp)
        AND((effectiveEnd=NULL)OR(effectiveEnd>=thePersistInfo.modifyStamp))]
^WCTYPE|wt.part.WTPart
```

Note that the “context” and “datatype” roles at the beginning and end of the TypeIdentifier have been swapped in the inverted form. This is true for all AssocationTypeIdentifiers. Also note that, for the MBA, the role name – “masterReference” – has been moved from the prefix position of the role filter to the postfix position.

Inverting the role filter is easily performed on the parse tree representation generated by the role filter parser: simply perform a recursive walk of the tree and execute the appropriate node-type specific action for each node.

For <expr>, <term>, and <factor> nodes, the only action required is to recursively invert the left and right subtrees rooted at the node. When inverting a <relation> node, swap the left and right subtrees of the node, and invert the operator according to the rules in the table below. No other operators are allowed for <relation> nodes; if one appears, it is an error.

Operator	**Inverse**
<	>
<=	>=
=	*no change*
!=	*no change*
>=	<=
>	<

For all other node types (i.e., Operation, Identifier, Integer, String, and Constant), the invert operation is a no-op.

Note that Operation nodes should *not* be present in “cleaned” tree, and therefore should not be encountered when inverting the role filter. If one is found at this stage of processing, it is an error.

The inverse of a RoleFilter is obtained by invoking the invert() method on the root node of the parse tree. This will invert the parse tree. The inverted tree can then be used to obtain the string representation of the inverted filter, by calling its toString() method.

Generating Expression Evaluators (AttributeContainerSets)

Role filters are ultimately translated into *SQL Where clauses* that specify join and selection criteria. The input to the SQL generators are AttributeContainerSets (ACS). An ACS is a representation of a Boolean expression expressed in terms of constant values and attributes of object types. Invoking the RoleFilterNode.getFilter() method on the root node of the parse tree will generate and return the expression evaluator (i.e., AttributeContainerSet) that is represented by the tree.

Generating the expression evaluator represented by a parse tree is accomplished by a post-order traversal of the tree. As each node of the tree is visited, a method appropriate for that type of node is executed.

The parse tree is a binary tree of RoleFilterNodes. Each node in the tree has a field that marks it as a specific node type as defined by RoleFilterNode.Type. Some nodes are *operation* nodes, while others are *operand* nodes. Operation nodes have an *operator*, while operand nodes have a *value.*

For an operand node, this method generates a result that represents the value identified by that node. For example, the result for an IdentifierNode will be an AttributeTypeIdentifier. The result for a StringNode will be an object that represents the constant string value identified by the StringNode.

For an operator node, the method invoked when generating an expression evaluator will return an AttributeContainerSet. For example, the result for a RelationNode will be an ACS that represents the result of the comparison operation identified by the node applied to the expressions represented by the nodes's left and right child nodes. For ExprNodes, TermNodes, and FactorNodes, the result will be an ACS that represents the result of the Boolean operation identified by the node applied to the expression(s) represented by the node's left (and, if non-null, right) child node(s).

The processing that may apply to any given RoleFilterNode may be dependent upon the type of the AssociationTypeIdentifier in which the role filter occurs. The getFilter() method that walks the parse tree to produce the ACS it represents, accepts one parameter of type RoleFilterNodeProcessor. This is expected to be an object than can apply AssociationTypeIdentifier subtype-specific processing to nodes within the tree. The tree-walker will invoke methods on this object, passing it data obtained from the current node, and use the processor's results to generate appropriate values for the expression evaluator created for the node.

The AttributeContainerSet that results from the recursive application of this process to the root node of the parse tree is essentially another tree representation of the role filter. The nodes in this new tree have just two types: CompositeAttributeContainerSet (resulting from ExprNodes, TermNodes, and FactorNodes appearing in the parse tree), and SingleCriterionAttributeContainerSet nodes (resulting from the RelationNodes in the original parse tree). The results of the various *operand nodes* from the parse tree appear as operands of the AttributeContainerFunctions that are wrapped by the SingleCriterionAttributeContainerSet nodes that were generated from the parse tree's RelationNodes.