

# 8

## An Expression Language

In this chapter we will implement a DSL for expressions, including arithmetic, boolean, and string expressions. We will do incrementally and in a test-driven way. Since expressions are by their own nature recursive, writing a grammar for this DSL requires some additional efforts, and this allows us to discover additional features of Xtext grammars.

You will also learn how to implement a type system for a DSL to check that expressions are correct with respect to types (for example, you cannot add an integer and a boolean). We will implement the type system so that it fits the Xtext framework and integrates correctly with the corresponding IDE tooling.

Finally, we will implement an interpreter for these expressions. We will use this interpreter to write a simple code generator that creates a text file with the evaluation of all the expressions of the input file, and also to show the evaluation of an expression in the editor.

### The Expressions DSL

In the DSL we develop in this chapter, which we call **Expressions DSL**, we want to accept input programs consisting of expressions and variable declarations with an initialization expression. Expressions can refer to variables and can perform arithmetic operations, compare expressions, use logical connectors (**and** and **or**), and concatenate strings. We will use **+** both for representing arithmetic addition and for string concatenation; when used with strings, the **+** will also have to automatically convert integers and booleans occurring in such expressions into strings.

Here is an example of a program that we want to write with this DSL:

```
i = 0
j = (i > 0 && (1) < (i+1))
k = 1
```

```
j || true
"a" + (2 * (3 + 5)) // string concatenation
(12 / (3 * 2))
```

For example, `"a" + (2 * (3 + 5))` should evaluate to the string `"a16"`.

## Creating the project

First of all, we will use the Xtext project wizard to create the projects for our DSL (following the same procedure explained in *Chapter 2, Creating Your First Xtext Language*).

Start Eclipse and perform the following steps:

1. Navigate to **File | New | Project...**, in the dialog navigate to the **Xtext** category and click on **Xtext Project**.
2. In the next dialog provide the following values:
  - **Project name:** `org.example.expressions`
  - **Name:** `org.example.expressions.Expressions`
  - **Extensions:** `expressions`
  - Uncheck the option **Create SDK feature project**

The wizard will create three projects and it will open the file `Expressions.xtext`, which is the grammar definition.

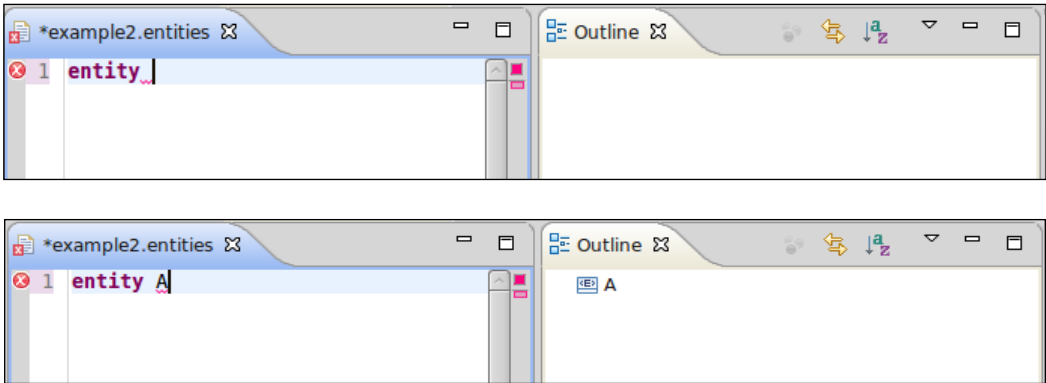
## Digression on Xtext grammar rules

Before writing the Xtext grammar for the Expressions DSL, it is important to spend some time to understand how the rules in an Xtext grammar and the corresponding EMF model generated for the AST are related.

From the previous chapters we know that Xtext, while parsing the input program, will create a Java object corresponding to the used grammar rule. Let us go back to our Entities DSL example and consider the rule:

```
Entity:
    'entity' name = ID ('extends' superType=[Entity])? '{'
        attributes += Attribute*
    '}' ;
```

When the parser uses this rule it will create an instance of the `Entity` class (that class has been generated by Xtext during the MWE2 workflow). However, the actual creation of such an instance will take place on the first assignment to a feature of the rule; in this example, no object will be created when the input only contains `entity`; the object will be created as soon as a name is specified, for example, when the input contains `entity A`. This happens because such an ID is assigned to the feature `name` in the rule. This is reflected in the outline view, as shown in the following screenshots:



This also means that the created `Entity` object is not "complete" at this stage, that is, when only a part of the rule has been applied. That is why when writing parts of the DSL implementation, for example, the validator, the UI customizations, and so on, you must always take into consideration that the model you are working on may have some features set to null.

The actual creation of the object of the AST can be made explicit by the programmer using the notation `{type name}` inside the rule; for example:

```
Entity:
  'entity' {Entity} name = ID ('extends' superType=[Entity])? '{'
    attributes += Attribute*
  '}' ;
```

If you change the rule as shown, then an `Entity` object will be created as soon as the `entity` keyword has been parsed, even if there has not been a feature assignment.

In the examples we have seen so far, the type of the object corresponds to the rule name; however, the type to instantiate can be specified using `returns` in the rule definition:

```
A returns B:
  ... rule definition ...
;
```

In this example, the parser will create an instance of `B`. `A` is simply the name of the rule, and there will be no generated `A` class. Indeed, the shape of the rule definitions we have used so far is just a shortcut for:

```
A returns A:
    ... rule definition ...
;
```

That is, if no `returns` is specified, the type corresponds to the name of the rule.

Moreover, the `returns` statement and the explicit `{type name}` notation can be combined:

```
A returns B:
    ... {C} ... rule definition ...
;
```

In this example, the parser will create an instance of `C` (and `C` is generated as a subclass of `B`). However, the object returned by the rule will have type `B`. Also in this case, there will be no generated `A` class.



When defining a cross-reference in the grammar, the name enclosed in square brackets refers to a type, not to a rule's name, unless they are the same.



When writing the grammar for the Expressions DSL, we will use these features.

## The grammar for the Expressions DSL

The DSL that we want to implement in this chapter should allow us to write lines containing either a variable consisting of an identifier and an initialization expression (the angle brackets denote non-terminal symbols):

```
<ID> = <Expression>
```

or an expression by itself:

```
<Expression>
```

If we write something like this

```
ExpressionsModel:
    variables += Variable*
    expressions += Expression*
;
```

We will not be able to write a program where variables and expressions can be defined in any order; we could only write variables first and then expressions.

To achieve the desired flexibility, we introduce an abstract class for both variable declarations and expressions; then, our model will consist of a (possibly empty) sequence of such abstract elements.

For the moment, we consider a very simple kind of expression: integer constants. These are the first rules (we skip the initial declaration parts of the grammar):

```
ExpressionsModel:
    elements += AbstractElement*;

AbstractElement:
    Variable | Expression ;

Variable:
    name=ID '=' expression=Expression;

Expression:
    value=INT;
```

The generated EMF classes for Variable and Expression will be subclasses of AbstractElement.

We are ready to write the first tests for this grammar. This chapter assumes that you fully understood the previous chapter about testing; thus, the code for testing we show here should be clear:

```
@RunWith( typeof (XtextRunner) )
@InjectWith( typeof (ExpressionsInjectorProvider) )
class ExpressionsParserTest {

    @Inject extension ParseHelper<ExpressionsModel>
    @Inject extension ValidationTestHelper

    @Test def void testSimpleExpression() {
        '''10'''.parse.assertNoErrors
    }

    @Test def void testVariableExpression() {
        '''i = 10'''.parse.assertNoErrors
    }
}
```

These methods test that both variable declarations and expressions can be parsed without errors.

We now continue adding rules; for example, we want to parse string and boolean constants besides integer constants.

We could write a single rule for all these constant expressions:

```
Expression:
(intvalue=INT) |
(stringvalue=STRING) |
(boolvalue=('true' | 'false'));
```

But this would not be good. It is generally not a good idea to have constructs that result in a single class that represents multiple language elements, since later when we are performing validation and other operations we cannot differentiate on class alone and instead have to inspect the corresponding fields (which, as you may recall, may not always be initialized due to partial parsing).

It is much better to write a separate rule for each element as shown in the following code snippet, and this will lead to the generation of separate classes:

```
Expression:
  IntConstant | StringConstant | BoolConstant;

IntConstant: value=INT;
StringConstant: value=STRING;
BoolConstant: value=('true' | 'false');
```

Note that, although `IntConstant`, `StringConstant`, and `BoolConstant` will all be subclasses of `Expression`, the field `value` will not be part of the `Expression` superclass; for `IntConstant` the field `value` will be of type integer, while for the other two it will be of type string; thus it cannot be made common.

At this point you must run the MWE2 workflow and make sure that the previous tests still run successfully; then you should add additional tests for parsing a string constant and a boolean constant (this is left as an exercise). We can write the aforementioned rules in a more compact form, using the notation `{type name}` that we introduced in the previous section, *Digression on Xtext grammar rules*:

```
Expression:
  {IntConstant} value=INT |
  {StringConstant} value=STRING |
  {BoolConstant} value=('true' | 'false');
```

Again, run the workflow and make sure tests still pass.

We add a rule which accepts a reference to an existing variable as follows:

```
Expression:
  {IntConstant} value=INT |
  {StringConstant} value=STRING |
  {BoolConstant} value=('true'|'false') |
  {VariableRef} variable=[Variable];
```

To test this last modification, we need an input with a variable declaration and an expression that refers to that variable:

```
@Test def void testVariableReference() {
  val e =
    '''
    i = 10
    i
    '''.parse
  e.assertNoErrors;

  (e.elements.get(1) as VariableRef).variable.
    assertSame(e.elements.get(0))
}
```

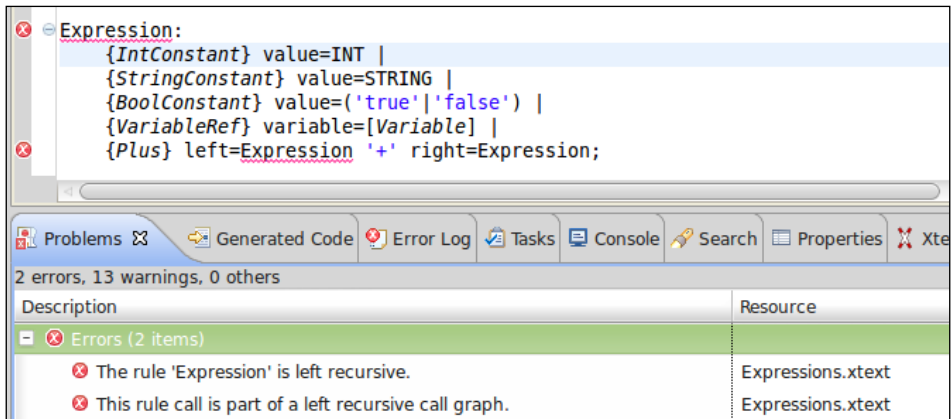
Note that we also test that the variable reference actually corresponds to the declared variable; `assertSame` comes from `org.junit.Assert`, whose static methods have been imported as extension methods.

## Left recursive grammars

When moving on to a more complex expression, such as addition, we need to write a recursive rule since the left and right parts of an addition are expressions themselves. It would be natural to express such a rule as follows:

```
Expression:
  ... as above
  {Plus} left=Expression '+' right=Expression;
```

However, this results in an error from the Xtext editor as shown in the following screenshot:



Xtext uses a parser algorithm that is suitable for use for interactive editing due to its better handling of error recovery. Unfortunately this parser algorithm does not deal with left recursive rules. A rule is **left recursive** when the first symbol of the rule is non-terminal and refers to the rule itself. The preceding rule for addition is indeed left recursive and is rejected by Xtext.



Xtext generates an ANTLR parser (Parr 2007), which relies on an LL(\*) algorithm; we will not go into details about parsing algorithms; we refer the interested reader to Aho et al. 2007. Such parsers have nice advantages concerning debugging and error recovery, which are essential in an IDE to provide a better feedback to the programmer. However, such parsers cannot deal with left recursive grammars.

The good news is that we can solve this problem, the bad news is that we have to modify the grammar to remove the left recursion, using a transformation referred to as **left-factoring**.

The parser generated by ANTLR cannot handle left recursion since it relies on a top-down strategy. Bottom-up parsers do not have this problem, but they would require to handle **operator precedence** (which determines which sub-expressions should be evaluated first in a given expression) and **associativity** (which determines how operators of the same precedence are grouped in the absence of parentheses). As we will see in this section, left-factoring will allow us to implicitly define operator precedence and associativity.



You may be tempted to remove the left recursion by simply adding a token before the recursion, for instance, parentheses:

```
Expression:
... as above
{Plus} '(' left=Expression '+' right=Expression '');
```

But clearly this is not a solution; our DSL would accept addition expressions only if enclosed in parentheses. We will add parentheses to the grammar later, for the purpose of allowing expression grouping as a means to influence precedence -- for example,  $10/(5+5) \Rightarrow 1$  instead of  $10/5+5 \Rightarrow 7$ .

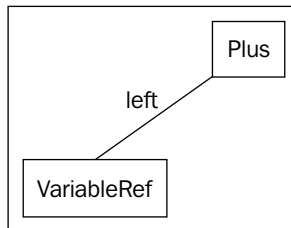
Instead, we remove the left recursion using a standard technique: we introduce a rule for expressions which are **atomic** and we state that an addition consists of a left part, which is an atomic expression, and an optional right part, which is recursively an expression (note that the right recursion does not disturb the ANTLR parser):

```
Expression:
{Plus} left=Atomic ('+' right=Expression)?;
```

```
Atomic returns Expression:
{IntConstant} value=INT |
{StringConstant} value=STRING |
{BoolConstant} value=('true' | 'false') |
{VariableRef} variable=[Variable];
```

Remember that the rule name is *Atomic*, but objects in the AST created by this rule will be of type *IntConstant*, *StringConstant*, and so on, according to the alternative used (indeed, no *Atomic* class will be generated from the preceding rule). Statically, these objects will be considered of type *Expression*, and thus the *left* and *right* fields in the *Plus* class will be of type *Expression*.

The preceding solution still has a major drawback, that is, additional useless nodes will be created in the AST. For example, consider an atomic expression such as a variable reference; when the atomic expression is parsed using the preceding rule, the AST will consist of a *Plus* object where the *VariableRef* object is stored in the *left* feature:



This indirection tends to be quite disturbing. For instance, the previous test method `testVariableReference` will now fail due to a `ClassCastException` and must be modified as follows:

```
@Test def void testVariableReference() {  
    ...  
  
    ((e.elements.get(1) as Plus).left as VariableRef).variable.  
        assertSame(e.elements.get(0))  
}
```

What we need is a way of telling the parser to:

- Try to parse an expression using the `Atomic` rule
- Search for an optional `+` followed by another expression
- If the optional part is not found then the expression is the element parsed with the `Atomic` rule
- Otherwise, instantiate a `Plus` object where `left` is the previously parsed expression with `Atomic` and `right` is the expression parsed after the `+`

All these operations can be expressed in an Xtext grammar as follows:

```
Expression:  
    Atomic ({Plus.left=current} '+' right=Expression)? ;
```

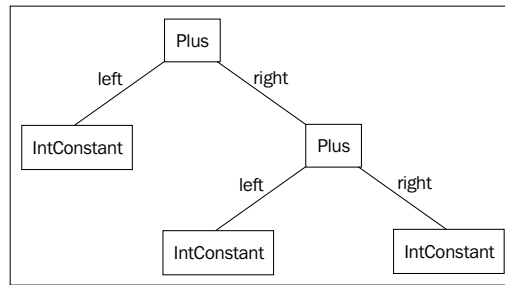
The part `{Plus.left=current}` is a **tree rewrite action**, and it does what we want: if the part `(...)?` can be parsed, then the resulting tree will consist of a `Plus` object where `left` is assigned the subtree previously parsed by the `Atomic` rule and `right` is assigned the subtree recursively parsed by the `Expression` rule.

Now, the test method `testVariableReference` can go back to its original form, since parsing an atomic expression does not result in an additional `Plus` object.

## Associativity

**Associativity** instructs the parser how to build the AST when there are several infix operators with the same precedence in an expression. It will also influence the order in which elements of the AST should be processed in an interpreter or compiler.

What happens if we try to parse something like `10 + 5 + 1`? The parsing rule invokes the rule for `Expression` recursively; the rule is right recursive, and thus we expect the preceding expression to be parsed in a **right-associative** way, that is, `10 + 5 + 1` will be parsed as `10 + (5 + 1)`. In fact, the optional part `(...)?` can be used only once; thus, the only way to parse `10 + 5 + 1` is to parse `10` with the `Atomic` rule and the rest with the optional part.

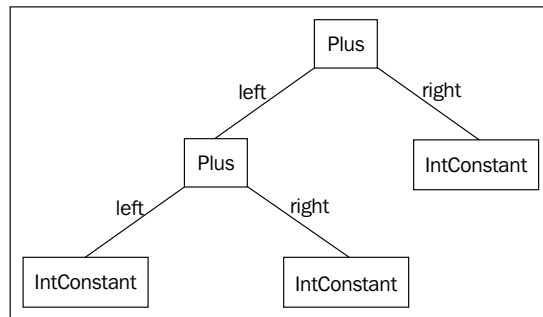


If, on the contrary, we write the rule as follows:

Expression:

```
Atomic ({Plus.left=current} '+' right=Atomic)* ;
```

We will get **left associativity**, that is,  $10 + 5 + 1$  will be parsed as  $(10 + 5) + 1$ . In fact, the optional part  $(\dots)^*$  can be used many times; thus, the only way to parse  $10 + 5 + 1$  is to apply that part twice (note that `right=Atomic` and not `right=Expression` as in the previous section).



It is important to check that the associativity of the parsed expressions is as expected. A simple way to check the result of associativity is to generate a string representation of the AST where non-atomic expressions are enclosed in parentheses.

```
def stringRepr(Expression e) {
  switch (e) {
    Plus:
      '''(«e.left.stringRepr» + «e.right.stringRepr）」'''
    IntConstant: '''«e.value»'''
    StringConstant: '''«e.value»'''
    BoolConstant: '''«e.value»'''
    VariableRef: '''«e.variable.name»'''
  }.toString
}
```

Then we write a method `assertRepr(input, expected)` that checks that the associativity of the input corresponds to the expected representation:

```
def assertRepr(CharSequence input, CharSequence expected) {
    input.parse => [
        assertNoErrors;
        expected.assertEquals(
            (elements.last as Expression).stringRepr
        )
    ]
}
```

We will use this method in the rest of the section to test the associativity of expressions. For instance, for testing the associativity of an addition, we write:

```
@Test def void testPlus() {
    "'10 + 5 + 1 + 2'".assertRepr("((10 + 5) + 1) + 2")
}
```

We add a rule for parsing expressions in parentheses:

```
Atomic returns Expression:
    '(' Expression ')' |
    {IntConstant} value=INT |
    ... as above
```

Note that the rule for parentheses does not perform any assignment to features; thus, given the parsed text `(exp)`, the AST will contain a node for `exp`, not for `(exp)`. This can be verified by this test:

```
@Test def void testParenthesis() {
    "(10)".parse.elements.get(0) as IntConstant
}
```

Although parentheses will not be part of the AST, they will influence the structure of the AST:

```
@Test def void testPlusWithParenthesis() {
    "( 10 + 5 ) + ( 1 + 2 )".assertRepr("((10 + 5) + (1 + 2))")
}
```

For the sum operator, left associativity and right associativity are equivalent (indeed, sum is an **associative operation**); the same holds for multiplication (with the possible exception of overflows or loss of precision that depends on the order of evaluation). This does not hold for subtraction and division.

In fact, how we parse and then evaluate such operations influences the result:  $(3 - 2) - 1$  is different from  $3 - (2 - 1)$ .

In these cases, you can disable associativity by using this pattern for writing the grammar rule:

```
Expression:
    Atomic ({Operation.left=current} '-' right=Atomic)? ;
```

In fact the ? operator (instead of \*) does not allow to parse an Atomic on the right more than once. This way, the user will be forced to explicitly use grouping expressions for the operations with no associativity when there are more than two operands.

The alternative solution is to choose an associativity strategy for the parser and implement the evaluator accordingly. Typically arithmetic operations are parsed and evaluated in a left-associative way (this holds in Java for instance), and we prefer this solution since users will get no surprises with this default behavior. Of course, parentheses can always be used for grouping. We modify the grammar for dealing with subtractions; we want to use a dedicated class for a subtraction expression, for example, Minus. We then modify the rule for Expression as follows:

```
Expression:
    Atomic (
        ({Plus.left=current} '+' | {Minus.left=current} '-')
        right=Atomic
    ) * ;
```

Note that the tree rewrite action inside the rule is selected according to the parsed operator. Though we are not doing that in this section (due to lack of space), you should write a test with a subtraction operation; of course, you must also update the `stringRepr` utility method for handling the case for Minus.

## Precedence

We can write the case for addition and subtraction in the same rule because they have the same arithmetic operator precedence. If we add a rule for multiplication and division we must handle their precedence with respect to addition and subtraction.

To define the precedence we must write the rule for the operator with less precedence in terms of the rule for the operator with higher precedence. This means that in the grammar, the rules for operators with less precedence are defined first. Since multiplication and division have higher precedence than addition and subtraction, we modify the grammar as follows:

```
Expression: PlusOrMinus;

PlusOrMinus returns Expression:
```

```
MulOrDiv (
  ({Plus.left=current} '+' | {Minus.left=current} '-')
  right=MulOrDiv
)* ;
```

MulOrDiv **returns** *Expression*:

```
Atomic (
  ({MulOrDiv.left=current} op=('*' | '/'))
  right=Atomic
)* ;
```

In the preceding rules, we use `returns` to specify the type of the created objects; thus, the features `left` and `right` in the corresponding generated Java classes will be of type `Expression`.

We added a main rule for `Expression` which delegates to the first rule to start parsing the expression; remember that this first rule (at the moment `PlusOrMinus`) concerns the operators with lowest precedence. There will be no class called `PlusOrMinus` since only objects of class `Plus` and `Minus` will be created by this rule. On the contrary in the rule `MulOrDiv`, we create objects of class `MulOrDiv`. In this rule we also chose another strategy: we have a single object type, `MulOrDiv`, both for multiplication and division expressions. After parsing, we can tell between the two using the operator which is saved in the feature `op`. Whether to have a different class for each expression operator or group several expression operators into one single class is up to the developer; both strategies have their advantages and drawbacks, as we will see in the next sections.

We now test the precedence of these new expressions:

```
@Test
def void testPlusMulPrecedence() {
  "10 + 5 * 2 - 5 / 1".assertRepr("((10 + (5 * 2)) - (5 / 1))")
}

def stringRepr(Expression e) {
  switch (e) {
    Plus:
      '''(«e.left.stringRepr» + «e.right.stringRepr»)'''
    Minus:
      '''(«e.left.stringRepr» - «e.right.stringRepr»)'''
    MulOrDiv:
      '''(«e.left.stringRepr» «e.op» «e.right.stringRepr»)'''
  }
  ... as before
}
```

Now we add boolean expressions and comparison expressions to the DSL; again, we have to deal with their precedence, which is as follows, starting from the ones with less precedence:

1. **boolean or** (operator `||`)
2. **boolean and** (operator `&&`)
3. **equality** and **dis-equality** (operators `==` and `!=`, respectively)
4. **comparisons** (operators `<`, `<=`, `>`, and `>=`)
5. **addition** and **subtraction**
6. **multiplication** and **division**

Following the same strategy for writing the grammar rules, we end up with the following expression grammar:

```
Expression: Or;

Or returns Expression:
    And ({Or.left=current} "||" right=And)* ;

And returns Expression:
    Equality ({And.left=current} "&&" right=Equality)* ;

Equality returns Expression:
    Comparison (
        {Equality.left=current} op=("=="|"!=")
        right=Comparison
    )* ;

Comparison returns Expression:
    PlusOrMinus (
        {Comparison.left=current} op(">="|"<="|">"|<")
        right=PlusOrMinus
    )* ;

... as before
```

When writing tests for these new expressions, we need to test for the correct precedence for the new operators both in isolation and in combination with other expressions (remember to update the `stringRepr` method to handle the new classes). We leave these tests as an exercise; they can be found in the sources of the examples. While adding rules for new expressions, we also added test methods in our parser test class, and we run all these tests each time: this will not only ensure that the new rules work correctly, but also that they do not break existing rules.

As a final step, we add a rule for boolean negation (operator "!"); this operator has the highest precedence among all the operators seen so far. Therefore, we can simply add a case in the `Atomic` rule:

```
Atomic returns Expression:
    '(' Expression ')' |
    {Not} "!" expression=Atomic |
    ... as before
```

We can now write a test with a complex expression and check that the parsing takes place correctly:

```
@Test def void testPrecedences() {
    "!true || false && 1 > (1/3+5*2)".
    assertRepr
        ("(!true) || (false && (1 > ((1 / 3) + (5 * 2))))")
}
```

Now might be a good time to refactor the rule `Atomic`, since it includes cases that are not effective atomic elements. We introduce the rule `Primary` for expressions with the highest priority that need some kind of evaluation; the rule `MulOrDiv` is refactored accordingly:

```
MulOrDiv returns Expression:
    Primary (
        {MulOrDiv.left=current} op=('*' | '/')
        right=Primary
    ) *
;
```

```
Primary returns Expression:
    '(' Expression ')' |
    {Not} "!" expression=Primary |
    Atomic
;
```

```
Atomic returns Expression:
    {IntConstant} value=INT |
    {StringConstant} value=STRING |
    {BoolConstant} value=('true' | 'false') |
    {VariableRef} variable=[Variable]
;
```



# The complete grammar

We sum up the section by showing the complete grammar of the Expressions DSL:

```

grammar org.example.expressions.Expressions with
    org.eclipse.xtext.common.Terminals

generate expressions
    "http://www.example.org/expressions/Expressions"

ExpressionsModel:
    elements += AbstractElement*;

AbstractElement:
    Variable | Expression ;

Variable:
    name=ID '=' expression=Expression;

Expression: Or;

Or returns Expression:
    And ({Or.left=current} "|" right=And)*
;

And returns Expression:
    Equality ({And.left=current} "&&" right=Equality)*
;

Equality returns Expression:
    Comparison (
        {Equality.left=current} op=("=="|"!=")
        right=Comparison
    ) *
;

Comparison returns Expression:
    PlusOrMinus (
        {Comparison.left=current} op(">="|"<="|">"|<")
        right=PlusOrMinus
    ) *
;

PlusOrMinus returns Expression:
    MulOrDiv (

```

```
    ({Plus.left=current} '+' | {Minus.left=current} '-')
    right=MulOrDiv
  ) *
;

MulOrDiv returns Expression:
  Primary (
    {MulOrDiv.left=current} op=('*' | '/')
    right=Primary
  ) *
;

Primary returns Expression:
  '(' Expression ')' |
  {Not} "!" expression=Primary |
  Atomic
;

Atomic returns Expression:
  {IntConstant} value=INT |
  {StringConstant} value=STRING |
  {BoolConstant} value=('true' | 'false') |
  {VariableRef} variable=[Variable]
;
```

## Forward references

You should know by now that parsing is only the first stage when implementing a DSL and that it cannot detect all the errors from the programs. We need to implement additional checks in a validator.

One important thing we need to check in our Expressions DSL is that an expression (including the initialization expression of a variable) does not refer to a variable defined after the very expression. Using an identifier before its declaration is usually called a **forward reference**.

Therefore, this program should not be considered valid:

```
i = j + 1
j = 0
```

since the initialization expression of `i` refers to `j`, which is defined after. Of course, this is a design choice: since we want to interpret the expressions, it makes sense to interpret them in the order they are defined.

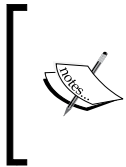
This strategy also avoids possible mutual dependency problems:

```
i = j + 1
j = i + 1
```

A variable which is initialized referring to itself is a special case of the preceding:

```
i = i + 1
```

We want to avoid this because our interpreter would enter an endless loop when evaluating expressions; this choice is also consistent with the design choice that variables, once initialized, cannot be further assigned.



Restricting the visibility of references can also be implemented with a custom `ScopeProvider`; the subject of **Scoping** will be detailed in *Chapter 10, Scoping*. However, **visibility** and **validity** are not necessarily the same mechanism. Therefore, it also makes sense to implement this checking in the validator.

We write a `@Check` method in the `ExpressionsValidator` class to report such problems.

To summarize, given a variable reference inside an expression, we need to:

- Get the list of all the variables defined before the containing expression
- Check that in the list there is a variable with the name of the reference

Note that we need to do that only if the variable is correctly bound; that is, if the cross-reference has already been resolved by `Xtext`, otherwise an error has already been reported.

The harder part is getting the list of all the variables defined before the containing expression. First of all, given an expression, we must get the `AbstractElement` containing such expression. For example, in an isolated expression, say `10`, such element is the expression itself; if we have `k = (10 + 5) < (j * 2)`, and we want to validate the variable reference `j`, then the element we need is the whole `k = (10 + 5) < (j * 2)`. This functionality only deals with model traversing, and we isolate it into a utility class, `ExpressionsModelUtil` (we have already seen this technique in *Chapter 7, Testing*, section *Testing and Modularity*). We will also reuse this utility class when implementing other parts of this DSL.

We implement the static method `variablesDefinedBefore` (`AbstractElement e`). In this method, we get the containing root model object using the method `org.eclipse.xtext.EcoreUtil2.getContainerOfType`. From the root, we get the list of all the elements. We then need to find the element containing `e`, that is, either `e` itself, or the element which contains `e`. We use the static method `org.eclipse.emf.ecore.util.EcoreUtil.isAncestor` to get that. Then, we get the sublist of variables. This is implemented in Xtend as follows:

```
import static extension org.eclipse.emf.ecore.util.EcoreUtil.*
import static extension org.eclipse.xtext.EcoreUtil2.*

class ExpressionsModelUtil {
  def static variablesDefinedBefore(AbstractElement e) {
    val allElements =
      e.getContainerOfType(typeof(ExpressionsModel)).elements
    val containingElement =
      allElements.findFirst[isAncestor(it, e)]
    allElements.subList(0,
      allElements.indexOf(containingElement)).
      typeSelect(typeof(Variable))
  }
}
```



This algorithm is potentially slow when dealing with large models and complex logic since it always searches from the top element down. Depending on your DSL, you might want to implement other strategies, such as bottom-up search, creating an index upfront, and so on. Once you have a bunch of tests for the simple implementation, you can experiment with optimizations and make sure that the tests still succeed.

We can now test this class in a separate JUnit test class (we show only a snippet) to make sure it does what we expect:

```
@Test def void variablesBeforeVariable() {
  '''
  true      // (0)
  i = 0      // (1)
  i + 10     // (2)
  j = i      // (3)
  i + j      // (4)
  '''.parse => [
    assertVariablesDefinedBefore(0, "")
    assertVariablesDefinedBefore(1, "")
    assertVariablesDefinedBefore(2, "i")
  ]
}
```

```

        assertVariablesDefinedBefore(3, "i")
        assertVariablesDefinedBefore(4, "i,j")
    ]
}

def void assertVariablesDefinedBefore(ExpressionsModel model,
    int elemIndex, CharSequence expectedVars) {
    expectedVars.assertEquals(
        model.elements.get(elemIndex).variablesDefinedBefore.
        map[name].join(",")
    )
}

```

Writing the `@Check` method in the validator is easy now:

```

import static extension org.example.expressions.typing.
ExpressionsModelUtil.*

class ExpressionsValidator extends AbstractExpressionsValidator {
    public static val FORWARD_REFERENCE =
        "org.example.expressions.ForwardReference";

    @Check
    def void checkForwardReference(VariableRef varRef) {
        val variable = varRef.getVariable()
        if (variable != null &&
            !varRef.variablesDefinedBefore.contains(variable))
            error("variable forward reference not allowed: '"
                + variable.name + "'",
                ExpressionsPackage::eINSTANCE.variableRef_Variable,
                FORWARD_REFERENCE, variable.name)
    }
}

```

It is also easy to test it (we show only a snippet):

```

@Test
def void testForwardReferenceInExpression() {
    '''i = 1 j+i j = 10'''.parse => [
        assertError(ExpressionsPackage::eINSTANCE.variableRef,
            ExpressionsValidator::FORWARD_REFERENCE,
            "variable forward reference not allowed: 'j'"
        )
        // check that it is the only error
        1.assertEquals(validate.size)
    ]
}

```

Note that in this test we also make sure that there is only one error concerning `j` and not `i`, which is correctly used after its declaration. We test this using the list of issues returned by `ValidationTestHelper.validate`. We issue an error in case of a forward reference, but we did not customize the way Xtext resolves references, thus the user of the Expressions DSL can still jump to the actual declaration of the variable, also in case of a forward reference error. This is considered a good thing in the IDE; for example, in JDT if you try to access a private member of a different class, you get an error, but you can still jump to the declaration of that member.

At the same time, though, the content assist of our DSL will also propose variables defined after the context where we are writing in the editor; this should be avoided since it is misleading. We can fix this by customizing the content assist. Xtext already generated a stub class in the UI plug-in project, for customizing this; in this example, it is `org.example.expressions.ui.contentassist.ExpressionsProposalProvider` in the `src` folder. Also this class relies on a method signature convention as shown in the following code snippet:

```
public void complete{RuleName}_{FeatureName} (
    EObject element, Assignment assignment,
    ContentAssistContext context,
    ICompletionProposalAcceptor acceptor)
public void complete_{RuleName} (
    EObject element, RuleCall ruleCall,
    ContentAssistContext context,
    ICompletionProposalAcceptor acceptor)
```

In the signatures, `RuleName` is the name of the rule in the grammar and `FeatureName` is the name of the feature (with the first letter capitalized) assigned in that rule. The idea is to use the first method signature to customize the proposals for a specific feature of that rule and the second one for customizing the proposals for the rule itself. In our case, we want to customize the proposals for the variable feature in the `Atomic` rule (that is, the rule which parses a `VariableRef`); thus we define a method called `completeAtomic_Variable`. The stub class extends a generated class in the `src-gen` folder, `AbstractExpressionsProposalProvider` that implements all of these `complete` methods. You can inspect the base class to get the signature right. You should also use the `@Override` annotation so that you get an error if the name of the rule or of the feature changes in the grammar. Most of the time, you will only use the first parameter, which is the `EObject` object representing the object corresponding to the rule being used and the acceptor to which you will pass your custom proposals.

In the method `completeAtomic_Variable`, we get the variables defined before the passed `EObject` and create a proposal for each variable:

```
override completeAtomic_Variable(EObject elem,
    Assignment assignment,
```

```

ContentAssistContext context,
ICompletionProposalAcceptor acceptor) {
if (!(elem instanceof AbstractElement))
    return; // no proposal

(elem as AbstractElement).variablesDefinedBefore.forEach[
    variable |
    acceptor.accept(createCompletionProposal
        (variable.name,
         variable.name + " - Variable", null,
         context));
]
}

```

Proposals are created using the `createCompletionProposal` method; you need to pass the string which will be inserted in the editor, the string shown in the content assist menu, a default image, and the context you received as a parameter. Each proposal must be passed to the acceptor.



Remember that for a given offset in the input program file, there can exist several possible grammar elements. Xtext will dispatch to the method declarations for any valid element, and thus many complete methods may be called.

In the previous chapter, we showed you how to test the content assist; we will now test our custom implementation (we only show the relevant parts):

```

@Test
def void testVariableReference() {
    newBuilder.append("i = 10 1+").
    assertText('!', '"Value"', '(', '+', '1', 'false', 'i', 'true')
}

@Test
def void testForwardVariableReference() {
    newBuilder.append("k= 0 j=1 1+ i = 10 ").
    assertTextAtCursorPosition("+", 1,
        '!', '"Value"', '(', '+', '1', 'false', 'j', 'k', 'true')
}

```

In the first method, we verify that we did not remove variable proposals that are valid; in the second one, we verify that only the variables that are defined before the current context are proposed. In particular, in the second test, we used an `assert` method passing the character in the input to set the (virtual) cursor at and an additional offset: we ask for the proposals right after the first `+` in the input string. We test that `j` and `k` are proposed, but not `i`.

## Typing expressions

In the Expressions DSL, types are not written explicitly by the programmer. However, due to the simple nature of our expressions, we can easily deduce the type of an expression by looking at its shape. In this DSL we have a fixed set of types: string, integer, and boolean. The mechanism of deducing a type for an expression is usually called **type computation** or **type inference**.

The base cases for type computation in the Expressions DSL are constants; trivially, an integer constant has type integer, a string constant has type string, and a boolean constant has type boolean.

As for composed expressions, besides computing a type, we must also check that its sub-expressions are correct with respect to types. This mechanism is usually called **type checking**. For example, consider the expression `!e`, where `e` is a generic expression. We can say that it has type boolean, provided that, recursively, the sub-expression `e` has type boolean; otherwise, the whole expression is not **well-typed**.

All the type mechanisms are part of the **type system** of the language. The type system depends on the semantics, that is, the meaning that we want to give to the elements of the DSL. For the Expressions DSL we design a type system that reflects the natural treatment of arithmetic and boolean expressions, in particular:

- If in a `Plus` expression one of the sub-expressions has type string, the whole expression is considered to have type string; if they have both type integer, then the whole expression has type integer; two boolean expressions cannot be added
- `Equality` can only act on sub-expressions with the same type
- `Comparison` can only act on sub-expressions with the same type, but not on booleans

Of course, a type system should also be consistent with code generation or interpretation; this is typically formally proved, but this is out of the scope of the book (we refer the interested reader to Hindley 1987, Cardelli 1996, and Pierce 2002). For instance, in our DSL, we allow expressions of the shape `1 + "a"` and `"a" + true`; we consider these expressions to have type string, since we use `+` also for string concatenation and implicit string conversion. If in a `Plus` expression the two sub-expressions have both type integer, the whole expression will have type integer, since that will be considered as the arithmetic addition. During interpretation, we must interpret such expressions accordingly.



Typically, type computation and type checking are implemented together and, as just seen, they are recursive. In general, the type of an expression depends on the type of the sub-expressions; the whole expression is not well-typed if any of its sub-expressions are not well-typed. Here are some examples: `j * true` is not well-typed since multiplication is defined only on integers; `true == "abc"` is not well-typed since we can only compare by equality expressions of the same type, `true < false` is not well-typed since comparison operators do not make sense on booleans, and so on.

When implementing a type system in an Xtext DSL, we must take into consideration a few aspects: Xtext automatically validates each object in the AST, not only on the first level elements. For instance, if in our validator we have these two `@Check` methods:

```
@Check
public void checkType(And and)
```

```
@Check
public void checkType(Not not)
```

and the input expression is `!a && !b` then Xtext will automatically call the second method on the `Not` objects corresponding to `!a` and `!b` and the first method on the object `And` corresponding to the whole expression. Therefore, it makes no sense to perform recursive invocations ourselves inside the validator's methods. If an expression contains some sub-expressions which are not well-typed, it should be considered not well-typed itself; however, does it make sense to mark the whole expression with an error? For example, consider this expression:

```
(1 + 10) < (2 * (3 + "a"))
```

If we mark the whole expression as not well-typed, we would provide useless information to the programmer. The same holds true if we generate additional errors for the sub-expressions `(2 * (3 + "a"))` and `(3 + "a")`. Indeed, the useful information is that `(3 + "a")` has type string while an integer type was expected by the multiplication expression.

Due to all the aforementioned reasons, we adopt the following strategy:

- Type computation is performed without recurring on sub-expressions unless required in some cases; for example, a `MulOrDiv` object has type integer independently of its sub-expressions. This is implemented by the class `ExpressionsTypeProvider`.

- In the validator, we have a `@Check` method for each kind of expression, excluding the constant expressions, which are implicitly well-typed; each method checks that the types of the sub-expressions, obtained by using `ExpressionsTypeProvider`, are as expected by that specific expression. For example, for a `MulOrDiv`, we check that its sub-expressions have both type integer, otherwise, we issue an error on the sub-expression that does not have type integer.

As we will see, this strategy avoids checking the same object with the validator several times, since the type computation is delegated to `ExpressionsTypeProvider`, which is not recursive. It will also allow the validator to generate meaningful error markers only on the problematic sub-expressions.

## Type provider

Since we do not have types in the grammar of the Expressions DSL, we need a way of representing them. Since the types for this DSL are simple, we just need an interface for types, for example, `ExpressionsType`, and a class implementing it for each type, for example, `StringType`, `IntType`, and `BoolType`. These classes implement a `toString` method for convenience, but they do not contain any other information.



We write the classes for types and for the type provider in the new Java sub-package `typing`. If you want to make its classes visible outside the main plug-in project, you should add this package to the list of exported packages in the **Runtime** tab of the `MANIFEST.MF` editor.

In the type provider, we define a static field for each type. Using singletons will allow us to simply compare a computed type with such static instances.

```
class ExpressionsTypeProvider {  
    public static val stringType = new StringType  
    public static val intType = new IntType  
    public static val boolType = new BoolType
```

We now write a method, `typeFor`, which, given an `Expression`, returns an `ExpressionsType` object. We use the `dispatch` methods for special cases and `switch` for simple cases. For expressions whose type can be computed directly, we write:

```
def dispatch ExpressionsType typeFor(Expression e) {  
    switch (e) {  
        StringConstant: stringType  
        IntConstant: intType  
        BoolConstant: boolType
```

```

    Not: boolType
    MulOrDiv: intType
    Minus: intType
    Comparison: boolType
    Equality: boolType
    And: boolType
    Or: boolType
  }
}

```

We now write a test class for our type provider with several test methods (we only show some of them):

```

import static extension org.junit.Assert.*

@RunWith(typeof(XtextRunner))
@InjectWith(typeof(ExpressionsInjectorProvider))
class ExpressionsTypeProviderTest {
  @Inject extension ParseHelper<ExpressionsModel>
  @Inject extension ExpressionsTypeProvider

  @Test def void intConstant() { "10".assertIntType }
  @Test def void stringConstant() { "'foo'".assertStringType }
  ...
  @Test def void notExp() { "!true".assertBoolType }

  @Test def void multiExp() { "1 * 2".assertIntType }
  @Test def void divExp() { "1 / 2".assertIntType }
  ...
  def assertStringType(CharSequence input) {
    input.assertType(ExpressionsTypeProvider::stringType)
  }
  ...
  def assertType(CharSequence input,
    ExpressionsType expectedType) {
    expectedType.assertSame
      (input.parse.elements.last.typeFor)
  }
}

```

We wrote the method `assertType` that does most of the work: it parses the passed input and computes the type of the last element of the program.

Then, we can move on to more elaborate type computations, which we implement in a dispatch method for better readability.

```
def dispatch ExpressionsType typeFor(Plus e) {  
  val leftType = e.left?.typeFor  
  val rightType = e.right?.typeFor  
  if (leftType == stringType || rightType == stringType)  
    stringType  
  else  
    intType  
}
```

For `Plus`, we need to compute the type of sub-expressions, since if one of them has type `string`, the whole expression is considered to be a string concatenation (with implicit conversion to `string`); thus, we give it a `string` type. Otherwise, it is considered to be the arithmetic sum and we give it type `integer`. In this type system, this is the only case where type computation depends on the types of sub-expressions.

Remember that the type provider can also be used on a non-complete model, and thus we use the null-safe operator `?.`, described in *Chapter 3, The Xtend Programming Language*.

We can now test this case for the type provider:

```
@Test def void numericPlus() { "1 + 2".assertIntType }  
@Test def void stringPlus() { "'a' + 'b'".assertStringType }  
@Test def void numAndStringPlus() { "'a' + 2".assertStringType }  
@Test def void numAndStringPlus2() { "2 + 'a'".assertStringType }  
@Test def void boolAndStringPlus() { "'a' + true".assertStringType }  
@Test def void boolAndStringPlus2() { "false+'a'".assertStringType }
```

Also, the case for variable reference requires some more work:

```
def dispatch ExpressionsType typeFor(VariableRef varRef) {  
  if (varRef.variable == null ||  
      !(varRef.variablesDefinedBefore.contains(varRef.variable)))  
    return null  
  else  
    return varRef.variable.expression?.typeFor  
}
```

We must check that the reference concerns a variable defined before the current expression, otherwise we might enter an infinite loop; to this aim, we reuse the `ExpressionsModelUtil` class whose static methods are here imported as extension methods. We must also consider the case when the reference is not resolved. In both cases, we simply return `null`, and we know that an error has already been reported (by our own validator in the former case and by the default validator for cross-references in the latter case). Otherwise, the type of a variable reference is the type of the referred variable, which, in turn, is the type of its initialization expression. We must check that the initialization expression is not `null`, in case of an incomplete program. The test for this case of the type provider is as follows:

```
@Test def void varRef() { "i = 0 i".assertIntType }
```



Remember that you should follow the **Test Driven Development** strategy illustrated in *Chapter 7, Testing*; first, write the implementation for a single kind of expression, write a test for that case then execute it; then, proceed with the implementation for another kind of expression, write a test for that case and run all the tests, and so on. As an exercise, you should try to re-implement everything seen in this section from scratch yourself following this methodology.

## Validator

We are ready to write the `@Check` methods in the `ExpressionsValidator`.



It is possible that some of the tests for the parser you have previously written fail now due to the validator methods and due to the fact that when you wrote those test expressions, you were not considering types. This is a standard situation in test driven development, and there is nothing to worry about. You can either modify the test expressions in the parser tests so that they are correct also with respect to types, or simply remove the call to `assertNoErrors`. In the latter case, the tests still make sense: when checking the associativity and precedence by traversing the AST, if the parsing failed due to a syntax error, the AST would be incomplete and the tests would fail with null pointer exception.

In the existing validator, we inject an instance of `ExpressionsTypeProvider` and we write some reusable methods which perform the actual checks. Thanks to these methods, we will be able to write the `@Check` methods in a very compact form.

```
class ExpressionsValidator extends
  AbstractExpressionsValidator {

  public static val WRONG_TYPE =
```

```
"org.example.expressions.WrongType";

@Inject extesion ExpressionsTypeProvider

def private checkExpectedBoolean(Expression exp,
                                EReference reference) {
    checkExpectedType(exp,
        ExpressionsTypeProvider::boolType, reference)
}

def private checkExpectedInt(Expression exp,
                              EReference reference) {
    checkExpectedType(exp,
        ExpressionsTypeProvider::intType, reference)
}

def private checkExpectedType(Expression exp,
                               ExpressionsType expectedType, EReference reference) {
    val actualType = getTypeAndCheckNotNull(exp, reference)
    if (actualType != expectedType)
        error("expected " + expectedType +
            " type, but was " + actualType,
            reference, WRONG_TYPE)
}

def private ExpressionsType getTypeAndCheckNotNull(
    Expression exp, EReference reference) {
    var type = exp?.typeFor
    if (type == null)
        error("null type", reference, WRONG_TYPE)
    return type;
}...
```



Although we present the reusable methods first, the actual strategy we followed when implementing the type checking in the validator is to first write some `@Check` methods, and then refactor the common parts. Before refactoring, we wrote some tests; after refactoring, we execute the tests to verify that refactoring did not break anything.

It should be straightforward to understand what the aforementioned methods do. The methods are parameterized over the EMF feature to use when generating an error; remember that this feature will be used to generate the error marker appropriately.

Let us see some @Check methods:

```
@Check def checkType(Not not) {
  checkExpectedBoolean(not.expression,
    ExpressionsPackage$Literals::NOT__EXPRESSION)
}

@Check def checkType(And and) {
  checkExpectedBoolean(and.left,
    ExpressionsPackage$Literals::AND__LEFT)
  checkExpectedBoolean(and.right,
    ExpressionsPackage$Literals::AND__RIGHT)
}
```

For Not, And, and Or, we check that the sub-expressions have type boolean and we pass the EMF features corresponding to the sub-expressions. (The case for Or is similar to the case of And, and it is therefore not shown).

Following the same approach, it is easy to check that the sub-expressions of Minus and MultiOrDiv both have integer types (we leave this as an exercise, but you can look at the sources of the example).

For an Equality expression, we must check that the two sub-expressions have the same type. This holds true also for a Comparison expression, but in this case, we also check that the sub-expressions do not have type boolean, since in our DSL, we do not want to compare two boolean values. The implementation of these @Check methods are as follows, using two additional reusable methods:

```
@Check def checkType(Equality equality) {
  val leftType = getTypeAndCheckNotNull(equality.left,
    ExpressionsPackage$Literals::EQUALITY__LEFT)
  val rightType = getTypeAndCheckNotNull(equality.right,
    ExpressionsPackage$Literals::EQUALITY__RIGHT)
  checkExpectedSame(leftType, rightType)
}

@Check def checkType(Comparison comparison) {
  val leftType = getTypeAndCheckNotNull(comparison.left,
    ExpressionsPackage$Literals::COMPARISON__LEFT)
  val rightType = getTypeAndCheckNotNull(comparison.right,
    ExpressionsPackage$Literals::COMPARISON__RIGHT)
  checkExpectedSame(leftType, rightType)
  checkNotBoolean(leftType,
    ExpressionsPackage$Literals::COMPARISON__LEFT)
  checkNotBoolean(rightType,
```

```
        ExpressionsPackage$Literals::COMPARISON__RIGHT)
    }

    def private checkExpectedSame(ExpressionsType left,
                                   ExpressionsType right) {
        if (right != null && left != null && right != left) {
            error("expected the same type, but was "+left+", "+right,
                  ExpressionsPackage$Literals::EQUALITY.getEIDAttribute(),
                  WRONG_TYPE)
        }
    }

    def private checkNotBoolean(ExpressionsType type,
                                 EReference reference) {
        if (type == ExpressionsTypeProvider::boolType) {
            error("cannot be boolean", reference, WRONG_TYPE)
        }
    }
}
```

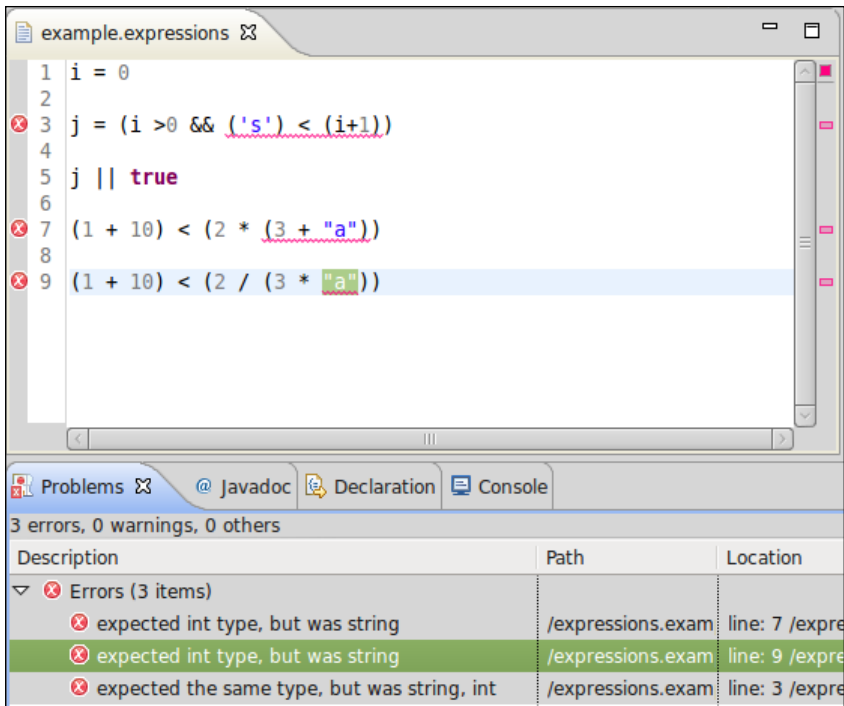
The final check concerns the `Plus` expression; according to our type system, if one of the two sub-expressions has type `string`, everything is fine, and therefore all these combinations are accepted as valid: `string+string`, `int+int`, `string+boolean`, and `string+int` (and the corresponding specular cases). We cannot add two boolean expressions or an integer and a boolean. Therefore, when one of the two sub-expressions has type `integer` or when they both have a type different from `string`, we must check that they do not have type `boolean`:

```
@Check def checkType(Plus plus) {
    val leftType = getTypeAndCheckNotNull(plus.left,
        ExpressionsPackage$Literals::PLUS__LEFT)
    val rightType = getTypeAndCheckNotNull(plus.right,
        ExpressionsPackage$Literals::PLUS__RIGHT)
    if (leftType == ExpressionsTypeProvider::intType
        || rightType == ExpressionsTypeProvider::intType
        || (leftType != ExpressionsTypeProvider::stringType &&
            rightType != ExpressionsTypeProvider::stringType)) {
        checkNotBoolean(leftType,
            ExpressionsPackage$Literals::PLUS__LEFT)
        checkNotBoolean(rightType,
            ExpressionsPackage$Literals::PLUS__RIGHT)
    }
}
```



Of course, while writing these methods, we also wrote test methods in the `ExpressionsValidatorTest` class. Due to lack of space, we are not showing these tests, and instead we refer you to the source code of the Expressions DSL.

Let's try the editor and look at the error markers as shown in the following screenshot:



The error markers are placed only on the sub-expression that is not well-typed; it is clear where the problem inside the whole expression is. If we did not follow the preceding strategy for computing and checking types, in a program with some not well-typed expressions, most of the lines would be red, and this would not help. With our implementation, the expression `j || true` does not have error markers, although the initialization expression of `j` contains an error; our type provider is able to deduce that `j` has type boolean anyway. Formally, also `j`, and in turn `j || true`, are not well-typed; however, marking `j || true` with an error would only generate confusion.

# Writing an interpreter

We will now write an interpreter for our Expressions DSL. The idea is that this interpreter, given an `AbstractElement`, returns a Java object which represents the evaluation of that element. Of course, we want the object with the result of the evaluation to be of the correct Java type; that is, if we evaluate a boolean expression, the corresponding object should be a Java boolean object.

Such an interpreter will be recursive, since to evaluate an expression, we must first evaluate its sub-expressions and then compute the result.

When implementing the interpreter we make the assumption that the passed `AbstractElement` is valid. Therefore, we will not check for null sub-expressions; we will assume that all variable references are resolved and we will assume that all the sub-expressions are well-typed; for example, if we evaluate an `And` expression, we assume that the objects resulting from the evaluation of its sub-expressions are Java `Boolean` objects.

For constants, the implementation of the evaluation is straightforward:

```
class ExpressionsInterpreter {  
    def dispatch Object interpret(Expression e) {  
        switch (e) {  
            IntConstant: e.value  
            BoolConstant: Boolean::parseBoolean(e.value)  
            StringConstant: e.value  
        }  
    }  
}
```

Note that the feature value for an `IntConstant` object is of Java type `int` and for a `StringConstant` object, it is of Java type `String`, and thus we do not need any conversion. For a `BoolConstant` object the feature value is also of Java type `String`, and thus we perform an explicit conversion using the static method of the Java class `Boolean`.

As usual, we immediately start to test our interpreter, and the actual assertions are all delegated to a reusable method:

```
class ExpressionsInterpreterTest {  
    @Inject extension ParseHelper<ExpressionsModel>  
    @Inject extension ValidationTestHelper  
    @Inject extension ExpressionsInterpreter  
  
    @Test def void intConstant() { "1".assertInterpret(1) }  
    @Test def void boolConstant() { "true".assertInterpret(true) }  
    @Test def void stringConstant() { "'abc'".assertInterpret("abc") }  
  
    def assertInterpret(CharSequence input, Object expected) {  
        input.parse => [  
            expected  
        ]  
    }  
}
```

```

        assertNoErrors
        expected.assertEquals(elements.last.interpret)
    }
}...

```

Note that, in order to correctly test the interpreter, we check that there are no errors in the input (since that is the assumption of the interpreter itself) and we compare the actual objects, not their string representation. This way, we are sure that the object returned by the interpreter is of the expected Java type.

Then, we write a case for each expression. We recursively evaluate the sub-expressions, and then apply the appropriate Xtend operator to the result of the evaluation of the sub-expressions. For example, for And:

```

switch (e) {
...
And: {
    (e.left.interpret as Boolean) && (e.right.interpret as Boolean)
}

```

Note that the method `interpret` returns an `Object`, and thus we need to cast the result of the invocation on sub-expressions to the right Java type. We do not perform an instanceof check because, as hinted previously, the interpreter assumes that the input is well-typed.

With the same strategy, we implement all the other cases. We show here only the most interesting ones. For `MulOrDiv`, we will need to check the actual operator, stored in the feature `op`:

```

switch (e) {
...
MulOrDiv: {
    val left = e.left.interpret as Integer
    val right = e.right.interpret as Integer
    if (e.op == '*')
        left * right
    else
        left / right
}

```

For `Plus`, we need to perform some additional operations: since we use `+` both as the arithmetic sum and as string concatenation, we must know the type of the sub-expressions. We use the type provider and write:

```

class ExpressionsInterpreter {
    @Inject extension ExpressionsTypeProvider

```

```
def dispatch Object interpret(Expression e) {  
  switch (e) {  
    ...  
    Plus: {  
      if (e.left.typeFor.isString || e.right.typeFor.isString)  
        e.left.interpret.toString + e.right.interpret.toString  
      else  
        (e.left.interpret as Integer) +  
        (e.right.interpret as Integer)  
    } ...  
  }
```

The method `isString` is a utility method that we added to `ExpressionsTypeProvider` to avoid doing the comparison with string types.

Finally, we deal with the case of variable and variable reference:

```
def dispatch Object interpret(Expression e) {  
  switch (e) {  
    ...  
    VariableRef: e.variable.interpret  
    ...  
  }  
  
  def dispatch Object interpret(Variable v) {  
    v.expression.interpret  
  }
```

## Using the interpreter

Xtext allows us to customize all UI aspects, as we saw in *Chapter 6, Customizations*. We can provide a custom implementation of **text hovering** (that is, the pop-up window that comes up when we hover for some time on a specific editor region) so that it shows the type of the expression and its evaluation. We refer to the Xtext documentation for the details of the customization of text hovering; here, we only show our implementation (note that we create a multi-line string using HTML syntax):

```

class ExpressionsEObjectHoverProvider extends
    DefaultEObjectHoverProvider {
    @Inject extension ExpressionsTypeProvider
    @Inject extension ExpressionsInterpreter
    override getHoverInfoAsHtml (EObject o) {
        if (o instanceof AbstractElement && o.programHasNoError) {
            val elem = o as AbstractElement
            return '''
                <p>
                    type  : <b>«elem.typeFor.toString»</b> <br>
                    value : <b>«elem.interpret.toString»</b>
                </p>
                '''
        } else
            return super.getHoverInfoAsHtml(o)
    }

    def programHasNoError(EObject o) {
        Diagnostician::INSTANCE.validate(o.rootContainer).
            children.empty
    }
}

```

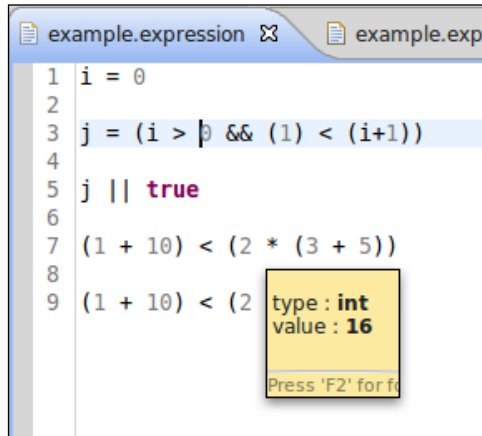
Remember that our interpreter is based on the assumption that it is invoked only on an EMF model that contains no error. We invoke our validator programmatically using the EMF API, that is, the `Diagnostician` class; we must validate the entire AST; thus, we retrieve the root of the EMF model using the method `EcoreUtil.getRootContainer` and check that the list of validation issues is empty. We need to write an explicit bind method for our custom implementation of text hovering in the `ExpressionsUiModule`:

```

public Class<? extends IEObjectHoverProvider>
    bindIEObjectHoverProvider() {
    return ExpressionsEObjectHoverProvider.class;
}

```

In the following screenshot, we can see our implementation when we place the mouse over the `*` operator of the expression `2 * (3 + 5)`: the pop-up window shows the type and the evaluation of the corresponding multiplication expression:



Finally, we can write a code generator which creates a text file (by default, it will be created in the directory `src-gen`):

```
import static extension
    org.eclipse.xtext.nodemodel.util.NodeModelUtils.*

class ExpressionsGenerator implements IGenerator {
    @Inject extension ExpressionsInterpreter

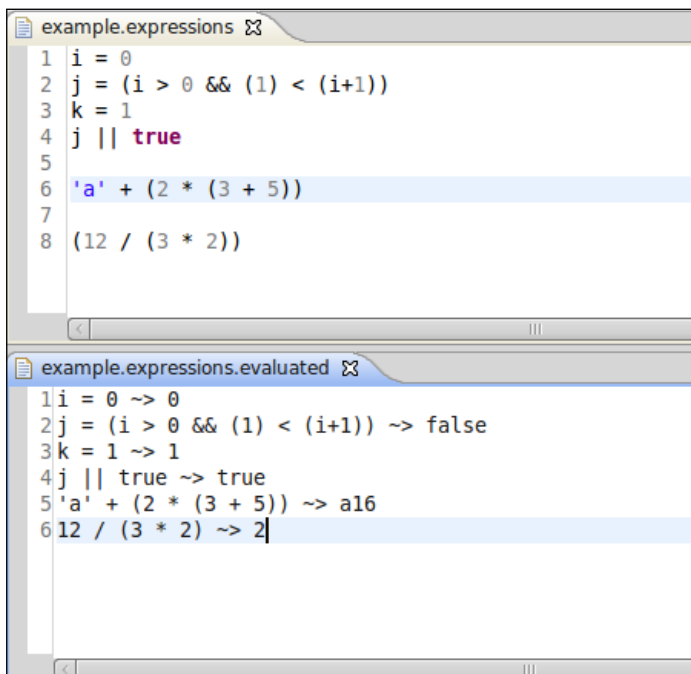
    override void doGenerate(Resource resource,
                              IFileSystemAccess fsa) {
        resource.allContents.toIterable()
            .filter(typeof(ExpressionsModel)).forEach[
                fsa.generateFile
                    (''«resource.URI.lastSegment».evaluated'',
                     interpretExpressions)
            ]
    }

    def interpretExpressions(ExpressionsModel model) {
        model.elements.map[
            '''«getNode.getTokenText» ~> «interpret»'''
        ].join("\n")
    }
}
```

Differently from the code generator we saw in *Chapter 5, Code Generation*, here we generate a single text file for each input file (an input file is represented by an EMF Resource); the name of the output file is the same as the input file (retrieved by taking the last part of the URI of the resource), with an additional `evaluated` file extension.

Instead of simply generating the result of the evaluation in the output file, we also generate the original expression. This can be retrieved using the Xtext class `NodeModelUtils`. The static utility methods of this class allow us to easily access the elements of the **node model** corresponding to the elements of the AST model. (Recall from *Chapter 6, Customizations* that the node model carries the syntactical information, for example, offsets and spaces of the textual input.) The method `NodeModelUtils.getNode(EObject)` returns the node in the node model corresponding to the passed `EObject`. From the node of the node model, we retrieve the original text in the program corresponding to the `EObject`.

An example input file and the corresponding generated text file are shown in the following screenshot:



```

example.expressions
1 i = 0
2 j = (i > 0 && (1) < (i+1))
3 k = 1
4 j || true
5
6 'a' + (2 * (3 + 5))
7
8 (12 / (3 * 2))

example.expressions.evaluated
1 i = 0 ~> 0
2 j = (i > 0 && (1) < (i+1)) ~> false
3 k = 1 ~> 1
4 j || true ~> true
5 'a' + (2 * (3 + 5)) ~> a16
6 12 / (3 * 2) ~> 2
  
```

## Summary

In this chapter, we implemented a DSL for expressions; this allowed us to explore some techniques for dealing with recursive grammar rule definitions in Xtext grammars and some simple type checking. We also showed how to write an interpreter for an Xtext DSL.

In the next chapter we will develop a small object-oriented DSL. We will use this DSL to show some advanced type checking techniques that deal with object-oriented features such as inheritance and subtyping (type conformance).