First experience with Xtext

In this post I describe my experience with the recent version (2.1) of Xtext (http://www.eclipse.org/Xtext/), a framework for developing domain specific languages (DSLs).

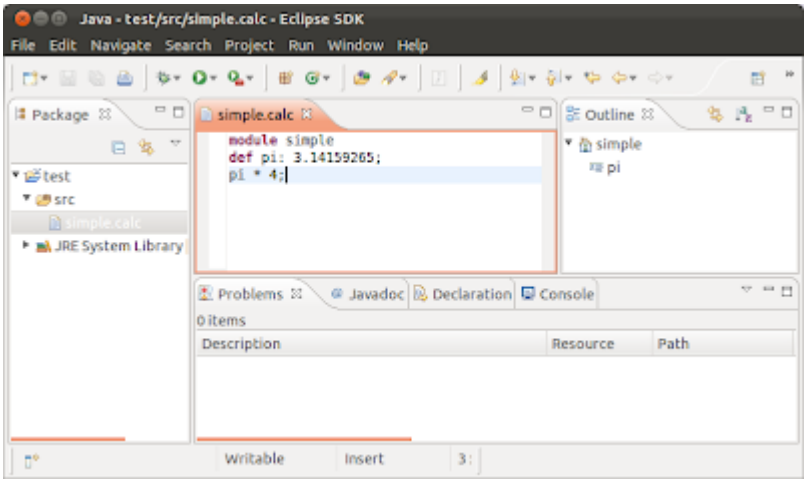
The main input for Xtext is a grammar file written in an elegant little language (http://git.eclipse.org/c/tmf/org.eclipse.xtext.git/tree/plugins/org.eclipse.xtext/src/org/eclipse/xtext/Xtext.xtext). The grammar language uses a variation of Extended Backus–Naur Form (http://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form) to define rules for both terminals and nonterminals. So far this is very similar to parser generators like GNU bison (http://www.gnu.org/s/bison/) or ANTLR (http://www.antlr.org/) which can create parsers out of similar grammar definitions. I have a lot of experience with parser generators and have become a bit sceptical about them. A hand-written parser is often easier to debug and maintain compared to a generated one. From my experience other parts of the language development process like semantic analysis or code generation are usually more complicated and at the same time these receive much less attention from the developers of automated tools. What makes Xtext different is that it not only generates a parser and a lexer but also a full-blown Eclipse IDE tailored for your language. So I decided to give Xtext a try and develop a toy language with it. I take a shortcut and start from one of the examples that come with Xtext. Choosing `File -> New -> Example...` and selecting `Xtext Simple Arithmetic Example` under the `Xtext Examples` category creates three projects:

- org.eclipse.xtext.example.arithmetics (main)
- org.eclipse.xtext.example.arithmetics.tests (tests)
- org.eclipse.xtext.example.arithmetics.ui (ui)

The main project contains the grammar file `Arithmetic.xtext` for the language. Following is an example of input that it accepts:

```
module simple
def pi: 3.14159265;
pi * 4;
```

Here is how it looks in Eclipse:



(http://3.bp.blogspot.com/-Fpffgjy0c3E/TtsWbHcZKRI/AAAAAAAACIg/_3mEXmFHUmM/s1600/eclipse-calc.png)

Syntax colouring, content assist, the Outline and Problems views - all work nicely. Let's now extend this language. First I add support for scientific notation (http://en.wikipedia.org/wiki/Scientific_notation), like `6.02e23`, in floating-point literals. The following modification of the `NUMBER` terminal does the job:

```
terminal NUMBER returns ecore::EBigDecimal:
    (('0'..'9')+ ('.' ('0'..'9')*)? | '.' ('0'..'9')+)
    (('e' | 'E') ('+' | '-')? ('0'..'9')+)?;
```

Next I add support for string expressions. To check for errors like trying to divide a string by a number I introduce a simple type system that consists of three types - `NUMBER`, `STRING` and `INVALID` with the latter used in case of errors. The type is represented by the following simple enum added to the main project:

```
public enum Type {
    INVALID,
    NUMBER,
    STRING
}
```

Then I add a string concatenation expression which uses the & operator and replace multiple classes for binary expressions with a single class `BinaryExpr` to make writing validation checks easier:

```
...
Expression returns Expr:
    Concatenation;

Concatenation returns Expr:
    Addition
    ({BinaryExpr.left=current} op=('&') right=Addition)*;

Addition returns Expr:
    Multiplication
    ({BinaryExpr.left=current} op=('+' | '-') right=Multiplication)*;

Multiplication returns Expr:
    PrimaryExpression
    ({BinaryExpr.left=current} op=('*' | '/') right=PrimaryExpression)*;

PrimaryExpression returns Expr:
    '(' Expression ')' |
    {NumberLiteral} value=NUMBER |
    {StringLiteral} value=STRING |
    {FunctionCall} func=[AbstractDefinition]
        ('(' args+=Expression (',' args+=Expression)* ')')?;
...
```

The next step is to add the type checks to the validator. Unfortunately Xtext validator does pre-order traversal of the AST while to check expression types we need post-order, because the type of an expression depends on the types of its subexpressions. It is easy to implement such traversal manually:

```
...
public class ArithmeticsJavaValidator
        extends AbstractArithmeticsJavaValidator {
    @Inject
    private Calculator calculator;

    private Map<Expr, Type> types = new HashMap<Expr, Type>();

    private boolean checkNumeric(Expr parent, Expr e,
            EStructuralFeature feature, int index) {
        Type type = checkType(e);
        if (type == Type.NUMBER)
            return true;
        if (type != Type.INVALID) {
            error("Expected numeric expression.", parent, feature,
                index);
        }
        return false;
    }

    private boolean checkString(Expr parent, Expr e,
      EStructuralFeature feature) {
        Type type = checkType(e);
        if (type == Type.STRING)
            return true;
        if (type != Type.INVALID) {
            error("Expected numeric expression.", parent, feature,
                    ValidationMessageAcceptor.INSIGNIFICANT_INDEX);
        }
        return false;
    }

    private boolean checkNumeric(Expr parent, Expr e,
      EStructuralFeature feature) {
        return checkNumeric(parent, e, feature,
                ValidationMessageAcceptor.INSIGNIFICANT_INDEX);
    }

    public Type checkType(Expr expr) {
        Type type = types.get(expr);
        if (type != null)
            return type;

        if (expr instanceof NumberLiteral) {
            type = Type.NUMBER;
        } else if (expr instanceof StringLiteral) {
            type = Type.STRING;
        } else if (expr instanceof BinaryExpr) {
            BinaryExpr be = (BinaryExpr) expr;
            type = Type.INVALID;
            Expr left = be.getLeft(), right = be.getRight();
            if (!be.getOp().equals("&")) {
                if (checkNumeric(expr, left,
```

```
                ArithmeticsPackage.Literals.BINARY_EXPR__LEFT)
                  && checkNumeric(expr, right,
                      ArithmeticsPackage.Literals.BINARY_EXPR__RIGHT))
                  type = Type.NUMBER;
          } else {
              if (checkString(expr, left,
                      ArithmeticsPackage.Literals.BINARY_EXPR__LEFT)
                  && checkString(expr, right,
                      ArithmeticsPackage.Literals.BINARY_EXPR__RIGHT))
                  type = Type.STRING;
          }
      } else if (expr instanceof FunctionCall) {
          FunctionCall call = (FunctionCall) expr;
          EList<Expr> args = call.getArgs();
          AbstractDefinition func = ((FunctionCall) expr).getFunc();
          if (func instanceof Definition)
              type = checkType(((Definition) func).getExpr());
          else
              type = Type.NUMBER;
          for (int i = 0, n = args.size(); i < n; i++) {
              if (!checkNumeric(expr, args.get(i),
                  ArithmeticsPackage.Literals.FUNCTION_CALL__ARGS, i))
                  type = Type.INVALID;
          }
      }
      types.put(expr, type);
      return type;
  }

  @Check
  public void check(Module m) {
      types.clear();
  }

  public final static String NORMALIZABLE = "normalizable-expression";

  @Check
  public void check(Expr expr) {
      Type type = checkType(expr);
      if (type != Type.NUMBER)
          return;

      // ignore literals
      if ((expr instanceof NumberLiteral) ||
          (expr instanceof FunctionCall))
          return;
      // ignore evaluations
      if (EcoreUtil2.getContainerOfType(expr, Evaluation.class)!=null)
          return;

      TreeIterator<EObject> contents = expr.eAllContents();
      while(contents.hasNext()) {
          EObject next = contents.next();
          if (next instanceof FunctionCall) {
```

```
                return;
            }
        }
        BigDecimal decimal = calculator.evaluateNumeric(expr);
        if (decimal.toString().length()<=8) {
            warning(
                "Expression could be normalized to constant '" +
                    decimal + "'",
                null,
                ValidationMessageAcceptor.INSIGNIFICANT_INDEX,
                NORMALIZABLE,
                decimal.toString());
        }
    }
}
```

As you can see from the above code the type checks aren't really pretty with all the `instanceof` s and casts. If I had more control over the generated AST classes I would apply the Visitor pattern (http://en.wikipedia.org/wiki/Visitor_pattern) and implemented the type checker as a visitor. However this can be mitigated with the help of the poorly documented PolymorphicDispatcher (http://download.eclipse.org/modeling/tmf/xtext/javadoc/2.0.0/org/eclipse/xtext/util/PolymorphicDispatcher.html).

Next I modify the `Calculator` class that implements evaluation to handle string concatenation. Also the evaluation methods should return `Object` instead of `BigDecimal` because the value of an expression can be a string or a number:

```
...
public class Calculator {

    private PolymorphicDispatcher<Object> dispatcher =
        PolymorphicDispatcher.createForSingleTarget(
            "internalEvaluate", 2, 2, this);

    public Object evaluate(Expr obj) {
        return evaluate(obj, ImmutableMap.<String,Object>of());
    }

    public BigDecimal evaluateNumeric(Expr obj) {
        return (BigDecimal)evaluate(obj);
    }

    public Object evaluate(Expr obj,
            ImmutableMap<String, Object> values) {
        Object invoke = dispatcher.invoke(obj, values);
        return invoke;
    }

    public BigDecimal evaluateNumeric(Expr obj,
            ImmutableMap<String, Object> values) {
        return (BigDecimal)evaluate(obj, values);
    }

    protected Object internalEvaluate(Expr e,
            ImmutableMap<String, Object> values) {
        throw new UnsupportedOperationException(e.toString());
    }

    protected Object internalEvaluate(NumberLiteral e,
            ImmutableMap<String, Object> values) {
        return e.getValue();
    }

    protected Object internalEvaluate(StringLiteral e,
            ImmutableMap<String, Object> values) {
        return e.getValue();
    }

    protected Object internalEvaluate(FunctionCall e,
            ImmutableMap<String, Object> values) {
        if (e.getFunc() instanceof DeclaredParameter) {
            return values.get(e.getFunc().getName());
        }
        Definition d = (Definition) e.getFunc();
        Map<String,Object> params = Maps.newHashMap();
        for (int i = 0; i < e.getArgs().size(); i++) {
            DeclaredParameter declaredParameter = d.getArgs().get(i);
            Object evaluate = evaluate(e.getArgs().get(i), values);
            params.put(declaredParameter.getName(), evaluate);
        }
```

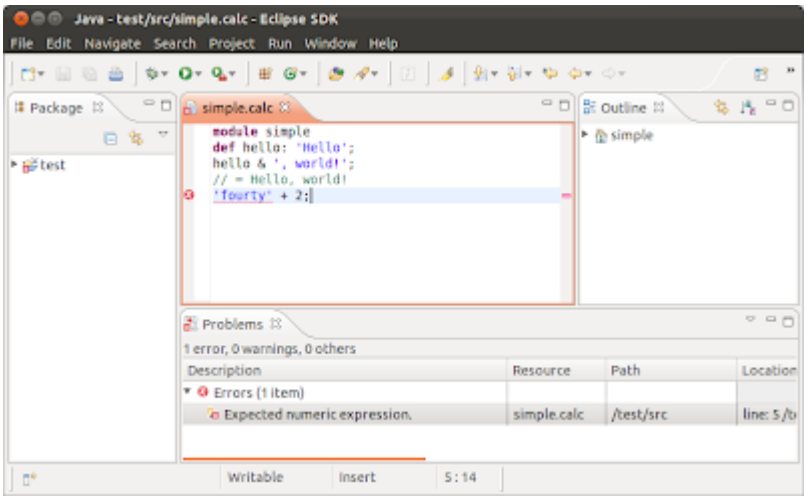
```
        return evaluate(d.getExpr(),ImmutableMap.copyOf(params));
    }

    protected Object internalEvaluate(BinaryExpr e,
            ImmutableMap<String, Object> values) {
        Resource res = e.eResource();
        if (!res.getErrors().isEmpty())
            return null;
        char op = e.getOp().charAt(0);
        if (op == '&') {
            return (String)evaluate(e.getLeft(), values) +
                evaluate(e.getRight(), values);
        }
        BigDecimal left = evaluateNumeric(e.getLeft(), values);
        BigDecimal right = evaluateNumeric(e.getRight(), values);
        switch (op) {
        case '+': return left.add(right);
        case '-': return left.subtract(right);
        case '*': return left.multiply(right);
        case '/': return left.divide(right, 20, RoundingMode.HALF_UP);
        }
        throw new UnsupportedOperationException(e.getOp());
    }
}
```

And finally I change the `InterpreterAutoEdit` class in the ui project to check types before evaluation:

```
...
public class InterpreterAutoEdit implements IAutoEditStrategy {

    public void customizeDocumentCommand(IDocument document,
            DocumentCommand command) {
        for (String lineDelimiter : document.getLegalLineDelimiters()) {
            if (command.text.equals(lineDelimiter)) {
                int line;
                int lineStart;
                try {
                    line = document.getLineOfOffset(command.offset);
                    lineStart = document.getLineOffset(line);
                    if (!document.get(lineStart, 3).equals("def")) {
                        Object computedResult = computeResult(document,
                                command);
                        if (computedResult != null) {
                            command.text = lineDelimiter + "// = " +
                                computedResult + lineDelimiter;
                        }
                    }
                } catch (BadLocationException e) {
                    // ignore
                }
            }
        }
    }

    private Object computeResult(IDocument document,
            final DocumentCommand command) {
        return ((IXtextDocument) document).readOnly(new
            IUnitOfWork<Object, XtextResource>() {
                public Object exec(XtextResource state)
                        throws Exception {
                    Evaluation stmt = findEvaluation(command, state);
                    if (stmt == null)
                        return null;
                    return evaluate(stmt);
                }
            });
    }

    protected Object evaluate(Evaluation stmt) {
        ArithmeticsJavaValidator validator =
            new ArithmeticsJavaValidator();
        Expr expr = stmt.getExpression();

        // Ignore all messages.
        validator.setMessageAcceptor(new ValidationMessageAcceptor() {

            public void acceptError(String message, EObject object,
                EStructuralFeature feature, int index, String code,
                String... issueData) {
            }
```

```
            public void acceptError(String message, EObject object,
                int offset, int length, String code,
                String... issueData) {
            }

            public void acceptWarning(String message, EObject object,
                EStructuralFeature feature, int index, String code,
                String... issueData) {
            }

            public void acceptWarning(String message, EObject object,
                int offset, int length, String code,
                String... issueData) {
            }

            public void acceptInfo(String message, EObject object,
                EStructuralFeature feature, int index, String code,
                String... issueData) {
            }

            public void acceptInfo(String message, EObject object,
                int offset, int length, String code,
                String... issueData) {
            }
        });

        return validator.checkType(expr) != Type.INVALID ?
                new Calculator().evaluate(expr) : "?";
    }

    ...
}
```

After implementing the above changes and re-generating Xtext artefacts, I get an Eclipse IDE for the language with simple string expressions and type checking:



(http://4.bp.blogspot.com/-EtJlFWF8hXI/Ttsv7K5-ZLI/AAAAAAAACls/Yilict9o-wI/s1600/eclipse-calc-with-strings.png)

Comments

asdasd
How to run the .calc file?
Victor Zverovich
Could be, but there is already an Eclipse-based IDE for AMPL called AMPLDev: http://bit.ly/KLuJYS . This was more of a test drive to get a better understanding of Xtext features.
Paul Rubin
Next step: use Xtext to create an Eclipse-based IDE for AMPL?

Related Posts

22 Jun 2021 » iOS SSID format string bug is preventable (/2021/06/22/preventing-format-string-bugs.html)
16 Jun 2021 » A quest for safe text formatting API (/2021/06/16/safe-formatting-api.html)
04 Aug 2020 » Writing files 5 to 9 times faster than fprintf (/2020/08/04/optimal-file-buffer-size.html)