INTRODUCTION TO

EXPRESSION LANGUAGES
WITH XTEXT

KARSTEN THOMS
EXPRESSIONS

Additive
+  -

Multiplicative
*  /  %

Boolean
&&  ||

Relational
>  <  >=  <=  ==  !=

Unary
!x

Postfix
x++

Atomic
„Hello“
CHALLENGES

• Recursion
• Precedence
• Associativity
• Ambiguity
• Typing
We can embed Xbase in our language!
But... if Xbase cannot be used?
What can we learn or reuse from Xbase?
Let’s look at some Xtext grammar patterns for expression languages in Xbase...
Language's meta model usually manually defined.
XExpression returns XExpression:
    XAssignment;

XAssignment returns XExpression:
    {XAssignment} feature=[types::JvmIdentifiableElement|FeatureCallID] OpSingleAssign value=XAs
    XORExpression (=>>({XBinaryOperation.leftOperand=current} feature=[types::JvmIdentifiableElement|OpMulti
    )?);
Every expression rule returns the same expression super type.
Rules first call the rule with the next higher priority.
...followed by an optional part starting with a syntactic predicate
...followed by a Tree Rewrite Action
Operators of an expression type listed in a datatype rule.
Operators of an expression type listed in a datatype rule
Rules for supported literals like Boolean, String, Number etc.

```
XBooleanLiteral returns XExpression:
    {XBooleanLiteral} ('false' | isTrue?='true');

XNullLiteral returns XExpression:
    {XNullLiteral} 'null';

XNumberLiteral returns XExpression:
    {XNumberLiteral} value=Number;

XStringLiteral returns XExpression:
    {XStringLiteral} value=STRING;

XTypeError Literal returns XExpression:
    {XTypeError Literal} 'typeof' ;
```
All literals collected by an abstract rule, being the rule with the highest priority.
A rule is left-recursive when the first symbol is a non-terminal and refers to the rule itself.

Expression :  
  Expression \( + \) Expression |  
  ( Expression |  
  Number;  

Number :  
  value = INT;
2 + 3

Sum

Number

Number
**SUM**

```
left=Number ('+' right=Sum)?;
```

**NUMBER**

```
value = INT;
```

**Example:**

```
1
```
```
left
```
```
1
```

```
Example: 2 + 3
```
```
left
```
```
2
```
```
right
```
```
3
```
**Addition returns Expression:**
Number (\{Sum.left = current\} ' + ' right=Number)*;

**Number:**
value = INT;

**Example:**
1

**Example:**
2 + 3
PRECEDENCE

Addition < Multiplication < Literal

Priority n < Priority n+1 < Priority n+2
SYNTACTIC PREDICATES

• Expressed by $\Rightarrow$ or $\rightarrow$ in front of a grammar element

• Give the parser hint to follow certain paths in case of ambiguity

  „if you see these tokens, then follow this path“

• Recommended instead of using backtracking
To which if element does the else part belong?
IfStatement returns Expression:
  'if'
  '(' condition=BoolExpression ')' 
  thenPart=Expression  
  (=>'else' elsePart=Expression)?
  ;
IfStatement returns Expression:
  'if' '(' condition ')' 
  thenPart=Expression 
  (=>'else' elsePart=Expression)? 
;
TYPES
IMPLEMENTING A TYPE SYSTEM

1. What is the Actual Type of an expression?

2. What is the Expected Type of an expression?

3. Is some type conformant with another type?
IMPLEMENTING A TYPE SYSTEM

1. What is the Actual Type of an expression?
   
   T getActualType (Expression expr)

2. What is the Expected Type of an expression?
   
   T getExpectedType (Expression expr)

3. Is some type conformant with another type?
   
   boolean isConformant (T expected, T actual)
class TypeValidator extends AbstractMyDSLValidator {
    @Inject ITypeComputer typeComputer

    @Check def void checkType(Expression expr) {
        val actualType = typeComputer.getActualType(expr)
        val expectedType = typeComputer.getExpectedType(expr)

        if (expectedType === null || actualType === null)
            return; // nothing to check
        if (!typeComputer.isConformant(expectedType, actualType)) {
            error("Incompatible types. Expected '" + expectedType.name + "'
                     but was '" + actualType.name + "'",
                     null,
                     IssueCodes.INCOMPATIBLE_TYPES);
        }
    }
}

class MyDSLRuntimeModule extends AbstractMyDSLRuntimeModule {
    ...
    @SingletonBinding(eager=true)
    def bindTypeValidator () { TypeValidator }
}
• Unit Tests are crucial
• Test Driven Development
Evaluate the Sessions

Sign in and vote at eclipsecon.org

-1 0 +1