Xbase Language Specification

Sven Efftinge, et al.

February 14, 2011

Contents

1 Preface **5**

2 Lexical Syntax **7**
2.1 Identifiers . . . 7
2.1.1 Escaped Identifiers . . . 7
2.1.2 Syntax . . . 7
2.1.3 Examples . . . 7
2.2 String Literals . . . 8
2.2.1 Syntax . . . 8
2.2.2 Examples . . . 8
2.3 Integer Literals . . . 8
2.3.1 Syntax . . . 8
2.4 Comments . . . 8
2.4.1 Syntax . . . 9
2.5 White Space . . . 9
2.6 Reserved Keywords . . . 9

3 Types **11**
3.1 Simple Type References . . . 11
3.1.1 Syntax . . . 11
3.1.2 Examples . . . 11
3.2 Function Types . . . 11
3.2.1 Syntax . . . 12
3.2.2 Examples . . . 12
3.3 Parameterized Type References . . . 12
3.3.1 Syntax . . . 12
3.3.2 Examples . . . 13
3.4 The type java.lang.Void . . . 13
3.5 Conformance Rules . . . 13
3.5.1 Common Super Type . . . 14

4 Expressions **15**
4.1 Literals . . . 15
4.1.1 String Literals . . . 15
4.1.2 Syntax . . . 15
4.1.3 Integer Literals . . . 15
4.1.4 Syntax . . . 15

4.1.5 Boolean Literals . . . 15
4.1.6 Syntax . . . 16
4.1.7 Null Literal . . . 16
4.1.8 Syntax . . . 16
4.1.9 Type Literals . . . 16
4.1.10 Syntax . . . 16
4.2 Type Casts . . . 16
4.2.1 Syntax . . . 16
4.2.2 Examples . . . 16
4.3 Infix Operators . . . 16
4.3.1 Property Assignment . . . 17
4.3.2 Short-Circuit Boolean Operators . . . 18
4.3.3 Examples . . . 18
4.4 Feature Calls . . . 18
4.4.1 Syntax . . . 19
4.4.2 Property Access . . . 19
4.4.3 Implicit 'this' variable . . . 19
4.4.4 Extensions . . . 19
4.4.5 Examples . . . 20
4.5 Spread Operator . . . 20
4.5.1 Syntax . . . 20
4.5.2 Examples . . . 20
4.6 Safe Navigation Featurecall . . . 21
4.6.1 Syntax . . . 21
4.6.2 Examples . . . 21
4.7 Closures . . . 21
4.7.1 Syntax . . . 21
4.7.2 Function Mapping . . . 21
4.7.3 Typing . . . 22
4.7.4 Examples . . . 22
4.8 If Expression . . . 22
4.8.1 Syntax . . . 23
4.8.2 Typing . . . 23
4.8.3 Examples . . . 23
4.9 Switch Expression . . . 23
4.9.1 Syntax . . . 23
4.9.2 Type guards . . . 24
4.9.3 Typing . . . 24
4.9.4 Examples . . . 24
4.10 Variable Declarations . . . 24
4.10.1 Syntax . . . 25
4.10.2 Typing . . . 25
4.11 Blocks . . . 25
4.11.1 Syntax . . . 25

4.11.2 Examples 26
4.12 While Loop 26
4.12.1 Syntax 26
4.12.2 Examples 26
4.13 Do-While Loop 26
4.13.1 Syntax 26
4.13.2 Examples 26
4.14 For Loop 27
4.14.1 Syntax 27
4.14.2 Typing 27
4.14.3 Examples 27
4.15 Constructor Call 27
4.15.1 Syntax 27
4.15.2 Example 27
4.16 Throwing Exceptions 28
4.16.1 Syntax 28
4.16.2 Typing 28
4.16.3 Example 28
4.17 Try, Catch, Finally 28
4.17.1 Syntax 28
4.17.2 Example 28

1 Preface

This document specifies the expression language Xbase. Xbase is a partial programming language implemented in Xtext and is meant to be embedded and extended within other programming languages and domain-specific languages (DSL) written in Xtext. Xtext is a highly extendable language development framework covering all aspects of language infrastructure such as parsers, linkers, compilers, interpreters and even full-blown IDE support based on Eclipse.

Developing DSLs has become incredibly easy with Xtext. Structural languages which introduce new coarse-grained concepts, such as services, entities, value objects or statemachines can be developed in minutes. However, software systems do not consist of structures solely. At some point a system needs to show some behavior, which is usually specified using so called *expressions*. Expressions are the heart of every programming language and are not easy to get right. That is why most people do not add support for expressions in their DSL, but try to solve this differently. The most often used workaround is to define only the structural information in the DSL and add behavior by modifying or extending the generated code. It is not only unpleasant to write, read and maintain information which closely belongs together in two different places, abstraction levels and languages. Also, modifying the generated source code comes with a lot of additional problems. But as of today this is the preferred solution since adding support for expressions (and a corresponding execution environment) for your language is hard - even with Xtext.

Xbase serves as a language library providing a common expression language bound to the Java platform (i.e. Java Virtual Machine). It ships in form of an Xtext grammar, as well as reusable and adaptable implementations for the different aspects of a language infrastructure such as an AST structure, a compiler, an interpreter, a linker, and a static analyzer. In addition it comes with implementations to integrate the expression language within an Xtext-based Eclipse IDE. Default implementations for aspects like content assistance, syntax coloring, hovering, folding and navigation can be easily integrated and reused within any Xtext based language.

Conceptually and syntactically, Xbase is like Java statements+expressions, with the following differences:

- Runs on the JVM
- No checked exceptions
- Object-oriented
- Everything is an expression, there are no statements

- Closures
- Type inference
- Properties
- Simple operator overloading
- Powerful switch expressions

2 Lexical Syntax

Xbase comes with a small set of lexer rules, which can be overridden and hence changed by users. However the default implementation is carefully chosen and it is recommended to stick with the lexical syntax described in the following.

2.1 Identifiers

Identifiers are used to name all constructs, such as types, methods and variables. Xbase uses the default Identifier-Syntax from Xtext - compared to Java, they are slightly simplified to match the common cases while having less ambiguities. They start with a letter *a-z*, *A-Z* or an underscore followed by more of these characters or a digit *0-9*.

2.1.1 Escaped Identifiers

Identifiers may not have the same spelling as any reserved keyword. However, identifiers starting with a ^ are so called escaped identifiers. Escaped identifiers are used in cases when there is a conflict with a reserved keyword. Imagine you have introduced a keyword `service` in your language but want to call a Java property *service* at some point. In such cases you use an escaped identifier `^service` to reference the Java property.

2.1.2 Syntax

```
terminal ID:
    '^'? ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
;
```

2.1.3 Examples

- Foo
- Foo42
- FOO
- _42
- _foo
- ^extends

2.2 String Literals

String literals can either use single quotes (') or double quotes (") as their terminals. When using double quotes all literals allowed by Java string literals are supported. In addition new line characters are allowed, that is in Xbase all string literals can span multiple lines. When using single quotes the only difference is that single quotes within the literal have to be escaped and double quotes do not.

See § 3.10.5 String Literals

In contrast to Java, equal string literals within the same class do not neccessarily refer to the same instance at runtime.

2.2.1 Syntax

//TODO

2.2.2 Examples

- 'Foo Bar Baz'
- "Foo Bar Baz"
- " the quick brown fox
  jumps over the lazy dog."
- 'Escapes : \' '
- "Escapes : \" "

2.3 Integer Literals

Integer literals consist of one or more digits. Only decimal literals are supported and they always result in a value of type java.lang.Integer (it might result in native type int when translated to Java, see Types (§3)). The compiler makes sure that only numbers between 0 and *Integer.MAX* (0x7fffffff) are used.

There is no negative integer literal, instead the expression −23 is parsed as the prefix operator − applied to an integer literal.

2.3.1 Syntax

```
terminal INT returns ecore::EInt:
    ('0'..'9')+
;
```

2.4 Comments

Xbase comes with two different kinds of comments: Single-line comments and multi-line comments. The syntax is the same as the one known from Java (see § 3.7 Comments).

2.4.1 Syntax

```
terminal ML_COMMENT :
    '/*' -> '*/'
;
terminal SL_COMMENT :
    '//' !('\n'|'\r')* ('\r'? '\n')?
;
```

2.5 White Space

The white space characters ' ', '\t', '\n', and '\r are allowed to occur anywhere between the other syntactic elements.

2.6 Reserved Keywords

The following list of words are reserved keywords, thus reducing the set of possible identifiers:

1. extends
2. super
3. instanceof
4. as
5. new
6. null
7. false
8. true
9. val
10. var
11. if
12. else
13. switch
14. case
15. default
16. do

17. while
18. for
19. typeof
20. throw
21. try
22. catch
23. finally

However, in case some of the keywords have to be used as identifiers, the escape character for identifiers (§2.1.1) comes in handy.

3 Types

Xbase binds to the Java Virtual Machine. This means that expressions written in Xbase refer to Java types and Java type members. Xbase itself uses types defined in the Java language, such as classes, interfaces, annotations and enums. It also supports Java generics and shares the known syntax. In addition to Java, Xbase comes with the notion of function types.

Xbase does not allow to define arrays or references to arrays. Instead, any references to arrays will be transparently converted to lists and vice versa. The return type of an external function that calculates an array *int[]* can be directly assigned to a variable of type *java.util.List<java.lang.Integer>* (in short *List<Integer>*). Any function, that takes an array as argument can be invoked with a *List* instead.

3.1 Simple Type References

A simple type reference only consists of a *qualified name*. A qualified name is a name made up of identifiers which are separated by a dot (like in Java).

3.1.1 Syntax

```
QualifiedName:
  ID ('.' ID)*
;
```

There is no parser rule for a simple type reference, as it is expressed as a parameterized type references without parameters.

3.1.2 Examples

- java.lang.String
- String

3.2 Function Types

Xbase introduces *closures*, and therefore an additional function type signature. On the JVM-Level a closure (or more generally any function object) is just an instance of one of the types in org.eclipse.xtext.xbase.lib.Function*, depending on the number of arguments. However, as closures are a very important language feature, a special sugared syntax for function types has been introduced. So instead of writing Function1<String,Boolean> one can write (String)=>Boolean.

For more information on closures see section 4.7.

3.2.1 Syntax

```
XFunctionTypeRef:
  ('('JvmTypeReference (',' JvmTypeReference)*')')?
    '=>' JvmTypeReference;
```

3.2.2 Examples

- =>Boolean // predicate without parameters
- (String)=>Boolean // One argument predicate
- (Mutable)=>Void // A method doing side effects only – returns null
- (List<String>, int)=>String

3.3 Parameterized Type References

The general syntax for type references allows to take any number of type arguments. The semantics as well as the syntax is almost the same as in Java, so please refer to the third edition of the Java Language Specification.

The only difference is that in Xbase a type reference can also be a function type. In the following the full syntax of type references is shown, including function types and type arguments.

3.3.1 Syntax

```
JvmTypeReference:
  JvmParameterizedTypeReference |
  XFunctionTypeRef;

XFunctionTypeRef:
  ('(' JvmTypeReference (',' JvmTypeReference)* ')')?
    '=>' JvmTypeReference;

JvmParameterizedTypeReference:
  type=QualifiedName ('<' JvmTypeArgument (',' JvmTypeArgument)* '>')?;

JvmTypeArgument:
  JvmReferenceTypeArgument |
  JvmWildcardTypeArgument;

JvmReferenceTypeArgument :
  JvmTypeReference;

JvmWildcardTypeArgument:
  '?' (JvmUpperBound | JvmLowerBound)?;
```

JvmLowerBound :
 'super' JvmTypeReference;

JvmUpperBound :
 'extends' JvmTypeReference;

3.3.2 Examples

- String
- java.lang.String
- List<?>
- List<? extends Comparable<? extends FooBar>
- List<? super MyLowerBound>
- List<? extends =>Boolean>

3.4 The type java.lang.Void

The null reference is the only valid value of the type Void, which gets some special treatment in Xbase. Every Java method which is declared *void* (i.e. without a return value) is translated to a method with return type *java.lang.Void*. At runtime such method invocations will result in *null*. The specialty is that while it is allowed to pass null everywhere

discuss use of nullable annotation

instead of any other value, this does not mean that *java.lang.Void* is a subtype of any other type. The instanceof operator as well as the type matchers in the section 4.9 do not match null.

3.5 Conformance Rules

Conformance is used in order to find out whether some expression can be used in a certain situation. For instance when assigning a value to a variable, the type of the right hand expression needs to conform to the type of the variable.

A type *T1* conforms to a type *T2* if

- T1==T2
- T1==java.lang.Void
- T1 is a subtype of T2

T1<T1P,..T1Pn> conforms to *T2<T2P, ... T2Pn>* if *T1* conforms to *T2* and each upper bound of a *T1Pn* conforms to the corresponding upper bound of *T2Pn*.

Is the type conformance for generic types correct? List<String> does not conform to Collection<CharSequence>

3.5.1 Common Super Type

For a set *[T1,T2,...Tn]* of types the common super type is computed by using the linear type inheritance sequence of *T1* and is iterated until one type conforms to each *T2,..,Tn*. The linear type inheritance sequence of *T1* is computed by ordering all types which are part if the type hierarchy of *T1* by their specificity. A type *T1* is considered more specific than *T2* if *T1* is a subtype of *T2*. Any types with equal specificity will be sorted by the maximal distance to the originating subtype. *CharSequence* has distance 2 to *StringBuilder* because the supertype *AbstractStringBuilder* implements the interface. Even if *StringBuilder* implements *CharSequence* directly, the interface gets distance 2 in the ordering because it is not the first class in the type hierarchy that implements the interface. If the distances for to classes are the same int he hierarchy, their qualified name is used to ensure deterministic results.

4 Expressions

Expressions are the main language constructs which are used to express behavior and computation of values. Xbase does not support the concept of a statement, but instead comes with powerful expressions to handle situations in which the imperative nature of statements are a better fit. An expression always results in a value (might be the value 'null' though). In addition expressions can be statically typed. By default it is assumed that languages making use of Xbase provide enough static context information for static type analysis, which is the basis of a lot of IDE features coming with Xbase.

4.1 Literals

A literal denotes a fixed unchangeable value. Xbase comes with literals for string, integers, booleans, null and types.

4.1.1 String Literals

A string literal as defined in section 2.2 is a valid expression and returns an instance of java.lang.String of the given value.

4.1.2 Syntax

```
XStringLiteral:
    STRING;
```

4.1.3 Integer Literals

An integer literal as defined in section 2.3 creates an int. There is no signed int. If you put a minus operator in front of an int literal it is taken as a UnaryOperator with one argument (the positive int literal).

4.1.4 Syntax

```
XIntegerLiteral:
    INT;
```

4.1.5 Boolean Literals

There are two boolean literals, true and false which correspond to their Java counterpart of type *boolean*.

4.1.6 Syntax

XBooleanLiteral:
    'false' | 'true';

4.1.7 Null Literal

The null pointer literal is, like in Java, null. It is the only value of the type *java.lang.Void* which has a special meaning in Xbase (see section 3.4).

4.1.8 Syntax

XNullLiteral:
    'null';

4.1.9 Type Literals

Type literals are specified using the keyword *typeof*, e.g.

- typeof(java.lang.String) which yields java.lang.String.class

4.1.10 Syntax

XTypeLiteral:
    'typeof' '(' QualifiedName ')';

4.2 Type Casts

Type cast behave like casts in Java, but have a slightly more readable syntax. Type casts bind stronger than any other operator but weaker than feature calls.

4.2.1 Syntax

XCastedExpression:
    Expression 'as' JvmTypeReference;

4.2.2 Examples

- my.foo as MyType
- (1 + 3 * 5 * (− 23)) as BigInteger

4.3 Infix Operators

Xbase supports a couple of predefined infix operators. In contrast to Java, the operators are not fixed to operations on certain types. Instead Xbase comes with an operator to method mapping, which allows users to redefine the operators for any type just by implementing the corresponding method signature. The following defines the operators and the corresponding Java method signatures / expressions.

e1 += e2	e1._operator_add(e2)
e1 || e2	e1._operator_or(e2)
e1 && e2	e1._operator_and(e2)
e1 == e2	e1._operator_equals(e2)
e1 != e2	e1._operator_notEquals(e2)
e1 < e2	e1._operator_lessThan(e2)
e1 > e2	e1._operator_greaterThan(e2)
e1 <= e2	e1._operator_lessEqualsThan(e2)
e1 >= e2	e1._operator_greaterEqualsThan(e2)
e1 -> e2	e1._operator_mappedTo(e2)
e1 .. e2	e1._operator_upTo(e2)
e1 + e2	e1._operator_plus(e2)
e1 - e2	e1._operator_minus(e2)
e1 * e2	e1._operator_multiply(e2)
e1 / e2	e1._operator_divide(e2)
e1 % e2	e1._operator_modulo(e2)
e1 ** e2	e1._operator_power(e2)
! e1	e1._operator_not()
- e1	e1._operator_minus()

The table above also defines the operator precedence in ascending order. The blank lines separate precedence levels. The assignment operator += is right-to-left associative in the same way as the plain assignment operator = is. That is a = b = c is executed as a = (b = c), all other operators are left-to-right associative. Parenthesis can be used to adjust the default precedence and associativity.

4.3.1 Property Assignment

The translation rule for the simple assignment operator = is a bit more complicated. Given the expression

myObj.myProperty = "foo"

The compiler first looks up whether there is an accessible Java Field called myProperty on the type of myObj. If there is one it translates to the following Java expression :

myObj.myProperty = "foo";

Remember in Xbase everything is an expression and has to return something. In the case of simple assignments the return value is the value returned from the corresponding

Java expression, which is the assigned value.

If there is no accessible field on the left operand's type, a method called setMyProperty(OneArg) (JavaBeans setter method) is looked up. It has to take one argument of the type (or a super type) of the right hand operand. The return value will be whatever the setter method returns (which usually is null). As a result the compiler translates to :

myObj.setMyProperty("foo")

if the return type is null, a = b = c will assign null to a.

4.3.2 Short-Circuit Boolean Operators

If the operators || and && are used in a context where the left hand operand is of type boolean, the operation is evaluated in short circuit mode, which means that the right hand operand might not be evaluated at all in the following cases:

1. in the case of || the operand on the right hand side is not evaluated if the left operand evaluates to true.
2. in the case of && the operand on the right hand side is not evaluated if the left operand evaluates to false.

4.3.3 Examples

- my.foo = 23
- myList += 23
- x > 23 && y < 23
- x && y || z
- 1 + 3 * 5 * (− 23)
- !(x)
- my.foo = 23
- my.foo = 23

4.4 Feature Calls

A feature call is used to invoke members of objects, such as fields and methods, but also can refer to local variables and parameters, which are made available for the current expression's scope.

4.4.1 Syntax

The following snippet is a simplification of the real Xtext rules, which cover more than the concrete syntax.

```
FeatureCall :
    ID |
    Expression ('.' ID ('(' Expression (',' Expression)* ')')?)*
```

4.4.2 Property Access

Feature calls are directly translated to their Java equivalent with the exception, that for calls to properties an equivalent rule as described in subsection 4.3.1 applies. That is, for the following expression

```
myObj.myProperty
```

the compiler first looks for an accessible field in the type of `myObj`. If no such field exists it looks for a method called `myProperty()` before it looks for the getter methods `getMyProperty()`. If none of these members can be found the expression is unbound and a compiliation error is thrown.

4.4.3 Implicit 'this' variable

If the current scope contains a variable named `this`, the compiler will make all its members available to the scope. That is if `this.myProperty` is a valid expression `myProperty` is valid as well and is equivalent, as long as there is no local variable 'myProperty' on the scope, which would have higher precedence.

4.4.4 Extensions

Languages extending Xbase might want to contribute to the feature scope. Besides that one can of course change the whole implementation as it seems fit there is a special hook, which can be used to add so called extension methods to existing types.

Xbase itself comes with a standard library of such extension methods adding support for various operators for the common types, such as java.lang.String, java.util.List, etc.

These extension methods are declared in separate Java classes. There are various ways how extension methods can be added. The simplest is, that the language designer predefines, which extension methods are available. This means, that language users cannot add additional library functions using this mechanism.

Another alternative is to have them looked up by a certain naming convention. Also for more general languages it is possible to let users add extension methods using imports or the like. This approach can be seen in the language Xtend2, where extension methods are lexically imported through static imports and/or dependency injection.

The precedence of extension methods is always lower than real member methods, that is you cannot override member features. Also the extension members are not invoked polymorphic. If you have two extension methods on the scope (`foo(Object)` and `foo(String)`) the expression `(foo as Object).foo` would bind and invoke `foo(Object)`.

4.4.5 Examples

- foo
- my.foo
- my.foo(x)
- oh.my.foo(bar)

4.5 Spread Operator

Xbase provides a special sugared notation to invoke a certain feature on each element of an iterable and obtain the results as a list. This is essentially a short cut notation for a higher order function on java.lang.Iterable. If you for instance want the names of all persons in a List you could write the following using a higher order function and a closures:

```
listOfPersons.collect(e|e.name)
```

However because this situation is so common Xbase provides a special sugared expression for that. So alternatively to the code above you can write:

```
listOfPersons*.name
```

Note that we decided to come up with an explicit operator (*.) because overloading the '.' operator as it is the case in Xpand, has caused a lot of surprises in the past. With an explicit operator the tooling as well as the user always can distinguish whether you invoke a feature on the iterables elements or on the iterable itself. Also note that this really is just a shortcut for collect. It doesn't do any flattening, when you invoke it on iterables containing iterables.

The operator works on all members of java.lang.Iterable. the return type is always an java.util.List<T> where T is the return type of the called feature.

TODO: Discuss, whether we want do an eager transformation of the iterable (and thereby provide a suiteable return type such as list if 'this' is a list or 'collection' if this is a collection). Otherwise we'll have to restrict the parameters of the spread operation to final values (capture context as we do for closures).

4.5.1 Syntax

```
XIterableFeatureCall:
    PrimaryExpression ('*.') FeatureCall;
```

4.5.2 Examples

- myListOfPersons*.name

4.6 Safe Navigation Featurecall

Checking for null references can make code very unreadable. In many situations it is ok for an expression to return null if a receiver was null. Xbase supports the safe navigation operator ?. to make such code more readable.

Instead of writing

```
if (myRef==null) null else myRef.doStuff()
```

one can write

```
myRef?.doStuff()
```

4.6.1 Syntax

```
XSafeFeatureCall:
    PrimaryExpression ('?.') FeatureCall;
```

4.6.2 Examples

- person?.address?.name

4.7 Closures

A closure is a literal that defines an anonymous function. A closure also captures the current scope, so that any final variables and parameters visible at construction time can be referred to in the closure's expression.

4.7.1 Syntax

```
XClosure:
    '[' (JvmFormalParameter (',' JvmFormalParameter)*)? '|' XExpression ']';

JvmFormalParameter:
    JvmTypeReference? ID;
```

The surrounding square brackets are optional if the closure is directly passed to function call.

```
myList.find(e|e.name==null)
myList.find([e|e.name==null])
```

But in all other cases the square brackets are mandatory:

```
val func = [String s| s.length>3]
```

4.7.2 Function Mapping

An Xbase closure is a Java object of one of the *Function* interfaces shipped with the runtime library of Xbase. There is an interface for each number of parameters. The names of the interfaces are

- *Function0<ReturnType>* for zero parameters,

- *Function1<Param1Type, ReturnType>* for one parameters,
- *Function2<Param1Type, Param2Type, ReturnType>* for two parameters,
- ...
- *Function6<Param1Type, Param2Type, Param3Type, Param4Type, Param5Type, Param6Type, ReturnType>* for six parameters,

In order to allow seamless integration with existing libraries such as Google Guava (formerly known as Google Collect) closures are auto coerced to expected types if those types declare only one method (methods from java.lang.Object don't count).

As a result given the method java.util.Collections.sort(List<T>, Comparator<? super T>) is available as an extension method, it can be invoked like this

```
newArrayList('aaa','bb','c').sort(
    e1,e2| if e1.length>e2.length
                −1
           else if e1.length<e2.length
                1
           else
                0)
```

4.7.3 Typing

Closures are expressions which produce function objects. The type is a function type (§3.2), consisting of the types of the parameters as well as the return type. The return type is never specified explicitly but is always inferred from the expression. The parameter types can be inferred if the closure is used in a context where this is possible.

For instance, given the following Java method signature: public T <T>getFirst(List<T> list, Function0<T,Boo
the type of the parameter can be inferred. Which allows users to write: getFirst(arrayList("Foo","Bar"), e|e=="B
instead of getFirst(arrayList("Foo","Bar"), String e|e=="Bar")

4.7.4 Examples

- | "foo" // closure without parameters
- String s | s.toUpperCase() // explicit argument type
- a,b,a | a+b+c // inferred argument types

4.8 If Expression

An if expression is used to choose two different values based on a predicate. While it has the syntax of Java's if statement it behaves like Java's ternary operator (predicate ? thenPart : elsePart), i.e. it is an expression that returns a value. Consequently, you can use if expressions deeply nested within expressions.

4.8.1 Syntax

```
XIfExpression:
    'if' '(' XExpression ')'
        XExpression
    ('else' XExpression)?;
```

An expression if (p) e1 else e2 results to either the value e1 or e2 depending on whether the predicate p evaluates to true or false. The else part is optional which is a shorthand for else null. That is if (foo) x // is the same as 'if (foo) x else null'

4.8.2 Typing

The type of an if expression is calculated by the return types T1 and T2 of the two expression e1 and e2. It uses the rules defined in subsection 3.5.1.

4.8.3 Examples

- if (isFoo) this else that
- if (isFoo) this else if (thatFoo) that else other
- if (isFoo) this

4.9 Switch Expression

4.9.1 Syntax

```
XSwitchExpression:
    'switch' (ID ':')? XExpression '{'
        XCasePart+
        ('default' ':' XExpression))?
    '}';

XCasePart:
    JvmTypeReference? ('case' XExpression)? ':' XExpression );
```

The switch statement is a bit different from the one in Java. First, there is no fall through which means only one case is evaluated at most. Second, the use of switch is not limited to certain values but can be used for any object reference instead. For a switch expression

```
switch e {
    case e1 : er1
    case e2 : er2
    ...
    case en : ern
    default : er
}
```

the main expression e is evaluated first and then each case sequentially. If the switch expression contains a variable declaration using the syntax known from section 4.14, the value is bound to the given name. Expressions of type java.lang.Boolean are not allowed in a switch expression.

The guard of each case clause is evaluated until the switch value equals the result of the case's guard expression or if the case's guard expression evaluates to true. Then the right hand expression of the case evaluated and the result is returned.

4.9.2 Type guards

In addition to the case guards one can add a so called *Type Guard* which is syntactically just type reference (§3.1) in front of the case keyword. The compiler will use that type for the switch expression in subsequent expressions. Example:

```
{
var Object x = ...;
switch x {
    String case x.length()>0 : x.length()
    List<?> : x.size()
    default : -1
}
```

Only if the switch value passes a type guard, i.e. an instanceof operation returns true, the case's guard expression is executed using the same semantics explained in previously. If the switch expression contains an explicit declaration of a local variable or the expression references a local variable, the type guard acts like a cast, that is all references to the switch value will be of the type specified in the type guard.

4.9.3 Typing

The return type of a switch expression is computed using the rules defined in subsection 3.5.1. The set of types from which the common super type is computed corresponds to the types of each case's result expression. In case a switch expression's type is computed using the expected type form the context, it is sufficient to return the expected type if all case branches types conform to the expected type.

4.9.4 Examples

- switch foo { Entity : foo.superType.name Datatype : foo.name default : throw new IllegalStateException }
- switch x : foo.bar.complicated('hello',42) { case "hello42" : ... case x.length<2 : ... default : .... }

4.10 Variable Declarations

Variable declarations are only allowed within blocks (§4.11) and switch expressions (§4.9). They are visible in any subsequent expressions in the block.

4.10.1 Syntax

```
XVariableDeclaration:
    ('val' | 'var') JvmTypeReference? ID '=' XExpression;
```

Xbase resembles the keywords val and var known from Scala (The Scala Language Specification 2.8). A variable declaration starting with the keyword val denotes a so called value, which is essentially a final (i.e. unsettable) variable. In rare cases, one needs to update the value of a reference. In such situations the variable needs to be declared with the keyowrd var, which stands for 'variable'.

```
{
    var i = 0
    while (i>MAX) {
        print("Hi there!")
    }
}
{
    val myFoo = my.complex(expression)
    myFoo.call(myFoo)
}
```

4.10.2 Typing

The return type of a variable declaration expression is always java.lang.Void. The type of the variable itself can either be explicitly declared or be inferred from the right hand side expression. Here is an example for an explicitly declared type: `var List<String> msg = new ArrayList<String>();` In such cases, the right hand expression's type must conform (§3.5) to the type on the left hand side.

Alternatively the type can be left out and will be inferred from the initialization expression: `var msg = new ArrayList<String>(); // –> type ArrayList<String>`

4.11 Blocks

The block expression allows to simulate imperative code sequences. It consists of a sequence of expressions, and returns the value of the last expression. The return type of a block is also the type of the last expression. Empty blocks return null. Variable declarations (§4.10) are only allowed within blocks and cannot be used as a block's last expression.

4.11.1 Syntax

```
XBlockExpression:
    '{'
        (XExpressionInsideBlock ';'?)*
    '}';
```

A block expression is surrounded by curly braces and contains at least one expression. It can optionally be terminated by a semicolon.

4.11.2 Examples

- { doSideEffect("foo") result }
- { var x = greeting(); if ((x.equals("Hello ")) { x+"World!"; } else { x; } }

4.12 While Loop

A while loop while (predicate) expression is used to execute a certain expression unless the predicate is evaluated to false. The return type of a while loop is java.lang.Void and the return value is null.

4.12.1 Syntax

```
XWhileExpression:
    'while' '(' predicate=XExpression ')'
        body=XExpression;
```

4.12.2 Examples

- ```
  while (true) {
      doSideEffect("foo");
  }
  ```
- ```
  while ((i=i+1)<max) doSideEffect("foo")
  ```

4.13 Do-While Loop

A do-while loop do expression while (predicate) is used to execute a certain expression unless the predicate is evaluated to false. The difference to the while loop (§4.12) is that the execution starts by executing the block once before evaluating the predicate for the first time. The return type of a do-while loop is java.lang.Void and the return value is null.

4.13.1 Syntax

```
XDoWhileExpression:
    'do'
        body=XExpression
    'while' '(' predicate=XExpression ')';
```

4.13.2 Examples

- ```
  do {
      doSideEffect("foo");
  } while (true)
  ```
  ```

- do doSideEffect("foo") while ((i=i+1)<max)

4.14 For Loop

The for loop for (T1 variable : iterableOfT1) expression is used to execute a certain expression for each element of an java.lang.Iterable. The local variable is final, hence canot be updated.

4.14.1 Syntax

```
XForExpression:
    'for' '(' JvmFormalParameter ':' XExpression ')'
        XExpression
    ;
```

4.14.2 Typing

The return type of a for loop is java.lang.Void and the return value is null. The type of the local variable can be left out. In that case it is inferred from the type of the java.lang.Iterable returned by the iterable expression.

4.14.3 Examples

- ```
  for (String s : myStrings) {
      doSideEffect(s);
  }
  ```

- ```
  for (s : myStrings)
      doSideEffect(s)
  ```

4.15 Constructor Call

Construction of objects is done by invoking Java constructors. Xbase uses the new keyword and the syntax is like the one known from Java.

4.15.1 Syntax

```
XConstructorCall:
  'new' QualifiedName
        ('<' JvmTypeArgument (',' JvmTypeArgument)* '>')?
        ('('(XExpression (',' XExpression)*)?')')?;
```

4.15.2 Example

new Foo()

4.16 Throwing Exceptions

Like in Java it is possible to throw java.lang.Throwable. The syntax is exactly the same as in Java.

4.16.1 Syntax

```
XThrow:
  'throw' XExpression;
```

4.16.2 Typing

The type of a throw expression is always java.lang.Void. The type of the expression after the throw keyword needs to conform to java.lang.Throwable.

4.16.3 Example

```
throw new RuntimeException()
```

4.17 Try, Catch, Finally

The try-catch-finally expression is used to handle exceptional situations gracefully. Xbase never forces you to catch exceptions, because there is no such concept like checked exceptions in Java. The syntax again is like the one known from Java.

4.17.1 Syntax

```
XTryCatchFinally:
  'try' XBlockExpression
  CatchClause*
  FinallyClause?;

CatchClause:
    'catch' XDeclaredParameter
        XBlockExpression

FinallyClause:
    'finally' XBlockExpression
```

4.17.2 Example

```
try {
    throw new RuntimeException()
} catch (NullPointerException e) {
    // handle e
} finally {
    // do stuff
}
```

List of External Links

http://java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.7
http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html
http://java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.10.5
http://www.scala-lang.org/docu/files/ScalaReference.pdf

Todo list

discuss use of nullable annotation 13
Is the type conformance for generic types correct? List<String> does not conform to Collection<CharSequence> 14
if the return type is null, a = b = c will assign null to a. 18