1 Preface

This document specifies the programming language library Xbase. Xbase is a partial programming language implemented in Xtext and is ment to be used and extended within other programming languages and domain-specific languages (DSL) which are also implemented 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.

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.

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 choosen 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.

2.1.1 Syntax

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

Identifiers start with any alphabetic character or an undercore followed by any number of alphabetic characters, underscores or numbers. 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 had introduced a keyword 'service' in your language but want at some point call a Java property 'service'. In such cases you use an escaped identifier ^service to reference the Java property.

2.1.2 Examples

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

2.2 String Literals

In Xbase string literals can span multiple lines. In addition there are two different terminals you can use which allows for avoiding escape sequences as far as possible. Note that there are no character literals in Xbase.

2.2.1 Syntax

```
terminal STRING :
    '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'"'|"'"|'\\') | !('\\'|'"'
        ) )* '"' |
    "'" ( '\\' ('b'|'t'|'n'|'f'|'r'|'"'|"'"|'\\') | !('\\'|"'"
        ) )* "'"
;
```

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 consists of one or more digits.

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 to the one from Java:

2.4.1 Syntax

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

2.5 White Space

Xbase ignores all white space.

2.6 Reserved Keywords

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

1. extends
2. instanceof
3. new
4. null
5. false
6. true
7. if
8. else
9. switch
10. case
11. default
12. while
13. def
14. class

However, in case some of the keywords have to be used as identifiers at times, the escape character of identifiers (2.1) come 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 only types defined in the Java language, such as classes, interfaces, annotations and enums. It also resembles Java generics and shares the known syntax. In addition to Java, Xbase comes with the notion of function types, that is the type of a function.

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's no rule for a simple type reference, as it is expressed as a parameterized type references without paramters.

3.1.2 Examples

- java.lang.String
- String

3.2 Function Types

Xbase introduces closures, which need a special kind of type. On the Jvm-Level a closures (or more generally any function object) is just an instance of one of the types in `org.eclipse.xtext.xbase.lib.Function*`, dependening on the number of arguments. However, as closures are a very important language feature, and it would be very inconvenient to write `Function3<String,String,String,Boolean>` the syntax for funtcion types has been introduced. So instead of writing `Function<String,Boolean>` one can write `(String)=>Boolean` instead.

3.2.1 Syntax

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

3.2.2 Examples

- =>Boolean // predicate without parameters
- (String)=>Boolean // One argument predicate
- (Mutable)=>Void // A method doing sideffects on an instance of Mutable and returns null
- (List<String>)=>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 exactly the same as in Java, so please refer to the third edition of the Java Language Specification which is available online for free.

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

3.3.1 Syntax

```
JvmTypeReference:
  JvmParameterizedTypeReference | XFunctionTypeRef;

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

JvmParameterizedTypeReference:
  type=[JvmType|QualifiedName] ('<' arguments+=JvmTypeArgument
       (',' arguments+=JvmTypeArgument)* '>')?;

JvmTypeArgument:
  JvmReferenceTypeArgument | JvmWildcardTypeArgument;

JvmReferenceTypeArgument :
  typeReference=JvmTypeReference;
```

```
JvmWildcardTypeArgument:
  {JvmWildcardTypeArgument} '?' (constraints+=JvmUpperBound |
      constraints+=JvmLowerBound)?;

JvmLowerBound :
  'super' typeReference=JvmTypeReference;

JvmUpperBound :
  'extends' typeReference=JvmTypeReference;
```

3.3.2 Examples

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

4 Expressions

Expressions are the main language constructs which are used to express behavior and computation of values. Xbase doesn't 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. That is by default it is assumed that languages making use of Xbase provide enough static context infromation to allow for static type analysis, which is the basis of a lot of IDE features coming with Xbase for free. However, the static typing is not mandatory and might be completely skipped if not wished. The openess of the compiler even allows to change the generation of concrete feature invocations to reflective calls, so that the language can be fully dynamically typed.

4.1 Literals

A literal denotes a fixed unchangeable value. Xbase comes with the following literals

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 Integer Literals

An integer literal as defined in section 2.3 creates an instance of `Integer` (TODO discuss how we want to go about the built-in types of Java).

4.1.3 Boolean Literals

There are two boolean literals, **true** and **false** which correspond to the respective Java values of the native type **boolean**

4.1.4 Syntax

```
XBooleanLiteral:
    {XBooleanLiteral} 'false' | isTrue?='true';
```

4.1.5 Null Literal

The null pointer literal again is like in Java: **null** .

4.1.6 Syntax

```
XNullLiteral:
    {XNullLiteral} 'null';
```

4.1.7 Type Literals

Also type literals are written like in Java, that is it it consists of a reference to a raw type suffixed with a dot and the keyword **class** .

4.1.8 Syntax

```
XTypeLiteral:
    type=[types::JvmType|QualifiedName] '.' 'class';
```

4.2 Infix Operators

Xbase supports the usual infix operators as well as some additional operators known from other languages. In contrast to Java, the operators are not fixed to operations on certain types, but 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 method signatures.

<Exp1>.someProp = <Exp2>	<Exp1>.setSomeProp(<Exp2>)
<Exp1> += <Exp2>	<Exp1>.add(<Exp2>)
<Exp1> || <Exp2>	<Exp1>.or(<Exp2>)
<Exp1> && <Exp2>	<Exp1>.and(<Exp2>)
<Exp1> instanceof RawTypeRef	<Exp1> instanceof RawTypeRef /* direct translation to Java */
<Exp1> == <Exp2>	<Exp1>.equals(<Exp2>)
<Exp1> != <Exp2>	<Exp1>.notEquals(<Exp2>)
<Exp1> < <Exp2>	<Exp1>.lessThan(<Exp2>)
<Exp1> > <Exp2>	<Exp1>.greaterThan(<Exp2>)
<Exp1> <= <Exp2>	<Exp1>.lessEqualsThan(<Exp2>)
<Exp1> >= <Exp2>	<Exp1>.greaterEqualsThan(<Exp2>)
<Exp1> -> <Exp2>	<Exp1>.mappedTo(<Exp2>)
<Exp1> .. <Exp2>	<Exp1>.upTo(<Exp2>)
<Exp1> + <Exp2>	<Exp1>.plus(<Exp2>)
<Exp1> - <Exp2>	<Exp1>.minus(<Exp2>)
<Exp1> * <Exp2>	<Exp1>.multiply(<Exp2>)
<Exp1> / <Exp2>	<Exp1>.divide(<Exp2>)
<Exp1> <Exp1> ** <Exp2>	<Exp1>.power(<Exp2>)
! <Exp1>	<Exp1>.not()
- <Exp1>	<Exp1>.minus()
<Exp1>[<Exp2>]	<Exp1>.apply(<Exp2)

The table above also defines the operator precedence (fom low to high precedence). The separator lines indicate a precedence level. The two assignment operators = and += are right-to-left associative, 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.

4.2.1 Property Assignment

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

```
myObj.myProperty = "foo"
```

It 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 operands type, a method called setMyProperty 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 (usually null i.e. type java.lang.Void). As a result the compiler translates to :

```
myObj.setMyProperty("foo")
```

4.2.2 Short-Circuit Boolean Operators

If the operators || and && are used in a context where both operands are of type boolean, the operation supports short circuit. That is

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** .

4.2.3 Examples

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

4.3 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.

4.3.1 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)* ')')
       ?)*
```

4.3.2 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 4.2.1 applies. That is, for the following expression

```
    myObj.myProperty
```

the compiler first looks for an accessible field in the static type of myObj. If no such field exists it looks for a method called getMyProperty() and binds to that if found. Otherwise such an expression is not bound which results in a compilation error.

4.3.3 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
```

if a valid expression

```
    myProperty
```

is valid as well and is equivalent.

4.3.4 Examples

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

4.4 Closures

A closure is a literal to create anonymous functions. A closure also captures the current scope, so that variables and parameters visible at contruction time can be referred to in the closure's expression.

4.4.1 Syntax

```
XClosure returns XExpression:
    {XClosure} (params+=JvmFormalParameter (',' params+=
        JvmFormalParameter)*)? '|' expression=XExpression;

JvmFormalParameter returns types::JvmFormalParameter:
    nonfinal?='nonfinal'? (parameterType=JvmTypeReference)?
        name=ID;
```

4.4.2 Function Mapping

An Xbase closure is a Java object of one of the Function interfaces shipped with the runtime library of Xbase. There's an interface for ech 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 seven parameters,

To be discussed: In order to allow seamless integration with Google Guava (formerly known as Google Collect) and Google Guice the Function1 extends com.google.common .base.Function<F, T> and the *Function0* extends com.google.common.base.Supplier< T> as well as com.google. inject . Provider<T> .

4.4.3 Type Inference

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 types of 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,Boolean>
    predicate)
```

the type of the parameter can be inferred. Which allows users to write:

```
getFirst(arrayList("Foo","Bar"), e|e=="Bar")
```

instead of

```
getFirst(arrayList("Foo","Bar"), String e|e=="Bar")
```

4.4.4 Examples

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

4.5 If Expression

The If expressions are used to choose two different values, based on a predicate. They are like if statements in java, but they are expressions. This means they always return a value and have a return type. Also this allows to use if clauses deeply nested within expressions.

4.5.1 Syntax

```
XIfExpression:
    'if' '(' p=XExpression ')'
        e1=XExpression
    ('else' e2=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'
```

4.5.2 Type Inference

The type of an if expression is calculated by the return types T1 and T2 of the two expression e1 and e2 .

1. If the T1 == T2 the type of the if expression is T1
2. If one T1==java.lang.Void the type of if expression is T2
3. If one T2==java.lang.Void the type of if expression is T1
4. If both T1!=T2, the expected type T3 of the current context is used and it is checked whether T1 and T2 are both assignable to T3

4.5.3 Examples

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

4.6 Switch Expression

4.6.1 Syntax

```
XSwitchExpression:
    'switch' main=XExpression? '{'
        ((cases+=XCasePart)+ |
         (cases+=XTypeCasePart)+)
        ('default' ':' default=XExpression)?
    '}';

XCasePart:
    'case' case=XExpression ':' then=XExpression;

XTypeCasePart:
    'instanceof' type=JvmTypeReference ':' then=XExpression;
```

The switch statement is a bit different then 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 instead can be used for any object reference. For a switch expression

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

first the main expression `e` is evaluated and then each case sequentially. If a case expression `en` evaluates to something such that `e == en` the result of the whole switch expression is `ern` . If non of the case expressions `e1` ... `en` was equal to the result of the main expression `e` , the result of the default part `er` is returned.

4.6.2 Leaving out the main expression

It is possible to leave out the main expression. Then the case expressions `e1` ... `en` have to be of type Boolean. They are evaluated in the specified order and as soon as one predicate `ex` evaluates to true the corresponding then expression `ex` is the result of the switch expression.

4.6.3 Inline polymorphic dispatch

Xbase allows to use class clauses instead of case clauses. The effect will be an inlined polymorphic dispatch, that is a value is taken depending on the runtime type of the main expression `e` . At compile time this has the effect that the type of `e` if used in some `erx` will be `tx` instead of the statically computed type of `e` .

Example:

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

4.6.4 Type inference

The return type of a switch expression is computed similarly to how the type of an if expression is computed.

1. If all types of `er1 ... ern` are the same or some are of type `java.lang.Void` , the type of the switch expression is the type of the first one which is not `java.lang.Void` .
2. if the types are different the context type is used and it is ensured at compilation time, that each `erx` is assignable to that type.

4.7 Variable Declarations

Variable Declarations are only allowed within a Block (4.8).

4.7.1 Syntax

```
XVariableDeclaration:
    nonfinal?='nonfinal'? 'var' type=JvmTypeReference? name=ID
        '=' right=XExpression;
```

By default all local variables are considered final which is unlike as it is in Java where variables or non-final by default. In order to make a variable non-final, such that it is possible to use them on the left hand side of an assignment subsection 4.2.1 one has to add the keyword 'nonfinal' to the declaration.

```
nonfinal var i = 0;
```

4.7.2 Type Inference

The type of a variable declaration would only be interesting in case the declaration is the last expression of a block, which doesn't make sense and results in a compile time error. However, the type of the variable can be defined in two way. It can be explicitly declared like in the following:

```
var List<String> msg = new ArrayList<String>();
```

In such cases, the right hand expression's type must be assignable to the type on the left hand side.

Alternatively the type can be left off and corresponds to the type of the initialization expression:

```
var msg = new ArrayList<String>(); // msg will be of type
    ArrayList<String>
```

4.8 Blocks

The block expression allows to simulate imperative code sequences. It conists 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. Variable declarations are only possible within blocks.

4.8.1 Syntax

```
XBlockExpression:
    '{'
        (expressions+=XExpressionInsideBlock ';')+
    '}';
```

A block expression is surrounded by curly braces and contains at least one expression. Each expression is terminated with a semicolon.

4.8.2 Examples

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

4.9 While Loop

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

4.9.1 Syntax

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

4.9.2 Examples

- while (true) \ doSideEffect("foo"); \

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

4.10 Do-While Loop

A do-while loop **do** e **while** (p) is used to execute a certain expression e unless a given predicate p is evaluated to **false** . The difference to the while loop (4.9) 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** .

4.10.1 Syntax

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

4.10.2 Examples

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

4.11 For Loop

The for loop **for** (T1 var : iterableOfT1) e is used to execute a certain expression e for each element of an java.lang.Iterable . The return type of a for loop is java.lang.Void and the return value is **null** .

4.11.1 Syntax

```
XForExpression:
    'for' '(' var=JvmFormalParameter ':' iterable=XExpression
       ')'
        body=XExpression
    ;
```

4.11.2 Type Inference

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.

4.11.3 Examples

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

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

4.12 Constructor Call

4.12.1 Syntax

```
XConstructorCall:
    'new' type=JvmTypeReference '(' (params+=XExpression (','
       params+=XExpression)*)? ')' ('as' alias=ID)?
        initializer=XBlockExpression?;
```

4.12.2 Example

```
new Foo() as f in {
    f.foo += new OtherFoo() {
        parent = f;
    };
}
```