XièXie
Programming Guide

A beginner guidance for the XièXie programming language

Lukas Hermanns

Updated on April 24, 2015

About the Author

My name is Lukas Hermanns (age-group 1990) and I started this project during my studies in 2014. By now I have over 12 years of experience in computer programming, started at the age of 12. I have been writing programs in Basic languages such as QBasic, PureBasic, and Blitz3D; in high level languages such as C, C++, C#, Objective-C, and Java; but also in scripting languages such as JavaScript and Python. I'm actually a preferred programmer in C++ (meanwhile C++11), low level stuff, and graphics programming with shading languages such as GLSL and HLSL.

The XièXie programming language is intended to be simple and not tuned for performance. It was originally designed to be used for scripting in video games, but can also be used for general purposes.

If you like, you can follow me on Twitter, YouTube, GitHub, or Bitbucket.

ToDo List

The compiler, and partially the virtual machine, are not yet completed. Some rules and explanations in this report may change over time. Here is a rough ToDo-list:

Parser	
Implementation of parser against grammar specification	done

Context Analyzer	
Expression Type Check	done
Cast Type Check	incomplete
Automatic Type Deduction	done
Procedure Overloading	done
Procedure Overriding	done
Named Parameters	done
Static and Non-Static Procedure Behavior	done
Class and Procedure Attributes	done

Code Generator	
Common CFG and TAC Generation from DAST	incomplete
CFG Simplification	done
If Statement	done
Switch Statement	incomplete
Return Statement	done
For Loop	done
Range-Based For Loop	done
Repeat Loop	done
While Loop	done
Do-While Loop	done

Optimizer	
Constant Folding	done
Local Constant Propagation	almost done
Local Copy Propagation	almost done
Local Variable Clean	almost done
Local Variable Reduction	incomplete

XASM Backend	
Final code generation for XASM	incomplete
Register Allocator	incomplete
Core Assembly File	incomplete

Contents

I The XièXie Programming Language 7

1 Introduction 8
- 1.1 What is XièXie? . . . 8
- 1.2 Why is it called "XièXie"? . . . 8
- 1.3 Motivation . . . 9

2 Syntax 10
- 2.1 Basics . . . 10
  - 2.1.1 Commentaries . . . 10
  - 2.1.2 Identifiers . . . 10
  - 2.1.3 Literals . . . 11
  - 2.1.4 String Interpolation . . . 12
- 2.2 Operators . . . 12
- 2.3 Type Denoters . . . 14
  - 2.3.1 Built-in Types . . . 14
  - 2.3.2 Objects . . . 14
  - 2.3.3 Arrays . . . 14
  - 2.3.4 Automatic Type Deduction . . . 14
- 2.4 Statements . . . 15
  - 2.4.1 Branch Statements . . . 16
  - 2.4.2 Loop Statements . . . 17
- 2.5 Expressions . . . 19
  - 2.5.1 Macros . . . 19

3 Classes 20
- 3.1 Getting Started . . . 20
- 3.2 Declaration Rules for Classes . . . 22
- 3.3 Procedures . . . 23
  - 3.3.1 Procedure Signature . . . 23
  - 3.3.2 Procedure Overloading . . . 23
  - 3.3.3 Procedure Overriding . . . 24

3.3.4 Named Arguments 25
3.4 Attributes 26
3.5 Anonymous Classes 27
3.6 Class Visibilities 28

4 Modules **30**
4.1 Using Modules 30
4.2 Creating Modules 30

II Developer Tools **32**

5 Compiler **33**
5.1 Command Line Tool 33
5.1.1 Output 33
5.1.2 Script 34

6 Virtual Machine **36**
6.1 Executing Programs 36

III Low-Level Programming **37**

7 Virtual Assembler **38**
7.1 Instruction Set 38
7.1.1 Registers 38
7.1.2 OpCodes 39
7.1.3 Memory Alignment 47
7.2 Procedure Calling 47
7.2.1 Calling Convention 47
7.2.2 Intrinsics 48

Part I

The XièXie Programming Language

Chapter 1

Introduction

First of all, "xièxie" is the chinese word for "thanks" or "thank you" (see hantrainerpro.com) and is roughly pronounced "Sh-eh sh-eh".

The design of the XièXie programming language is overall influenced by Java, C++, Swift, and Python.

1.1 What is XièXie?

The XièXie programming language is a high-level, object-oriented, scripting language with compiler and virtual machine. The XièXie compiler (XXC) translates the XièXie code (XX) to a virtual assembly (XASM), then assembles it to XièXie byte code (XBC), which can then be interpreted by the XièXie virtual machine (XVM).

1.2 Why is it called "XièXie"?

Some years, before I started with the development of this compiler, I already had some ideas about a name for it — at this time the compiler was intended to be a cross compiler, i.e. to compile the XièXie code to C++. The first idea I had in mind, was to call it "C+=2" or "C++++" because it should be a more comfortable and easier C++ variant. But this name looked and sounded strange. Another idea was to call it "C power of C" (in mathematical notation C^C). But that still was not what I was looking for. Then I remembered me to the Chinese word "Xièxie" which means "Thanks" in English and it sounds a little similar to "CC" which could be seen as a shortcut for C^C. That's the XièXie compiler's story about its name.

Or to make a long story short: I saw this word on a napkin in a Chinese restaurant and thought to my self: That's it! :-)

1.3 Motivation

There are many great programming languages out there. Some produce faster code than others, but some are simpler and have a better learning curve. Using a new language doesn't mean to give up a previous one, because every language has its own domain. For example, the performance of interpreted languages such as Python is totally sufficient for many applications. We don't need maximal performance for a script which does some socket connection or text processing for instance. But I would not write the compiler and interpreter for Python in a scripted language. There are also many ways to combine several languages: an interpreter could be used for scripting in video games for instance. But the game engine is written in C++. This is how it is done in Unity3D (see www.unity3d.com). They use Mono as compiler and interpreter framework for C#. But the engine itself is written in C++.

Now XièXie is aimed to be used for small scripting purposes, with a gentle learning curve. The great thing about its interpreter is, that it is very tiny and can easily be integrated into existing C or C++ code. This virtual machine only consists of a single code file (written in C99) and can simply be included into any C99 compliant project.

Chapter 2

Syntax

We start out with the syntax.

2.1 Basics

2.1.1 Commentaries

Commentaries are a fundamental part of programming languages and they are nearly identical to those in JAVA:

```
// Single-line comment

/* Single-line comment */

/*
Multi-line comment
*/
```

Although they are very similar to the commentaries in JAVA, nested multi-line comments are allowed as well:

```
/*
Outer comment
/* Nested comment */
*/
```

2.1.2 Identifiers

Identifiers (for variables, classes, etc.) must only contain alpha-numeric characters and the underscore. They must also begin with a letter or an underscore. But there is a small exception: they must not begin with `__xx__`, because all identifiers with this prefix are reserved for internal use of the compiler.

Valid identifiers are:

- `_name_`
- `FooBar`
- `number_of_wheels`
- `Customer01`
- ...

Invalid identifiers are:

- `naïve`
- `Foo.Bar`
- `number-of-wheels`
- `3over2`
- ...

2.1.3 Literals

These are the kinds of literals:

- Boolean Literal: `true`, `false`
- Integer Literal (Binary): e.g. `0b11001`, `0b0000`
- Integer Literal (Octal): e.g. `0o24`, `0o01234567`
- Integer Literal (Decimal): e.g. `3`, `12`, `999`, `1234567890`
- Integer Literal (Hexa-Decimal): e.g. `0xff`, `0x00`, `0xaB29`
- Float Literal: e.g. `0.0`, `3.5`, `12.482`
- String Literal: e.g. `"Foo Bar"`, `"Hello, World"`, `"\n\t"`, `"1st fragment"␣"2nd fragment`
- Verbatim String Literal: e.g. `@"\home\test"`, `@"a ""b"" c"`

Integer and float literals may optionally contain the single quotation mark as digit separator, for better readability. If it's used, all separators must satisfy the following rules:

1. For decimal literals, the separators must be **three steps apart** from each other, beginning at the dot.
   Example: `12'345`, `3.141'592'654`, or `-27'836.283'74`.

2. For non-decimal literals, the separators must be **four steps apart** from each other, beginning at the dot.
   Example: `0xff0'214b`, `0b100'1101'1011`, `0o1234'5670`

3. A separator must not appear at the beginning or the end of the literal.
   Counterexample: `'123'456'`.

4. No valid separator must be omitted.
   Counterexample: `1234'567'890`.

2.1.4 String Interpolation

The syntax for *string interpolation* is equal to that in Swift and allows you to easily concatenate strings with other objects or primitive elements, such as integers or floating-point values. If you want to concatenate strings for yourself, it may look like this:

```
var x := 5, y := -3
var s1 := "x = ".append(x).append(", y = ").append(y)
```

This will write "`x = 5, y = -3`" into `s1`. The same result can be achieved with the following string interpolation:

```
var s2 := "x = \(x), y = \(y)"
```

With the escape character '\', everthing in the parenthesis will be read an arbitrary expression, which is appended (or concatenated) to the string on its left. Thus, the string literal `s2` is equivalent to `s1`.

2.2 Operators

The most operators as in Java or C++ are also available in XièXie:

```
/* Arithmetic Operators */
a + b    // Addition
a - b    // Substration
a * b    // Multiplication
a / b    // Division
a % b    // Modulo
```

```
a << b   // Left Shift
a >> b   // Right Shift
-a       // Negate

/* Bitwise Operators */
a & b    // Bitwise AND
a | b    // Bitwise OR
a ^ b    // Bitwise XOR
~a       // Bitwise NOT

/* Boolean Operators */
not a    // Logic NOT
a and b  // Logic AND
a or b   // Logic OR

/* Relation Operators */
a = b    // Equality
a != b   // Inequality
a < b    // Less
a <= b   // Less Or Equal
a > b    // Greater
a >= b   // Greater Or Equal
```

As the interested reader may have noticed, the *equality operator* is different to that in the most languages. This is because the *copy assignment* operators is := and not =.

```
x := 5        // OK, set x to 5
a, b, c := 4  // OK, set a, b, and c to 4
a := b := 3   // Error, b := 3 is not an expression
```

In the above example `a := b := 3` is invalid, because on the right hand side of the first `:=` must be an expression. But per definition in XièXie `b := 3` is not an expression, but a statement! The variable list assignment (`a, b, c := ...`) is a comfort functionality, which is only supported for the copy assignment.

The modify-assign operators are available as well:

```
a += b
a -= b
a *= b
a /= b
a %= b
a <<= b
a >>= b
a &= b
a |= b
a ^= b
```

However, they are also not allowed inside another expression.

2.3 Type Denoters

2.3.1 Built-in Types

There are only the following three built-in data types:

```
bool  // Boolean type; can be 'true' or 'false'
int   // 32-bit signed integral type
float // 32-bit floating-point type
```

2.3.2 Objects

Types for class objects (more about classes in chapter 3) are written as follows:

```
// Empty string
String s

// List of strings
String s1 := "Hello, World", s2 := "Foo", s3 := "Bar"
```

2.3.3 Arrays

The only generic way for lists are the built-in arrays:

```
// Declare array objects with initializer lists
int[] intArray := { 1, 2, 3 }
float[][] floatArrayArray := { { 0.0, 1.5 }, { 3.5, 1.23 } }
String[] stringArray := { "a", "b", "c" }

// Access array elements
String s1 := stringArray[0]
String s2 := stringArray[intArray[0]]
float[] floatArray := floatArrayArray[1]
```

2.3.4 Automatic Type Deduction

Whereas automatic type deduction in C++11 is a very extensive language feature, in XièXie it can be summarized in this section. There are two keywords for automatic type deduction: `var` and `const`. As the name implies `var` denotes a variable type and `const` denotes a constant type. The latter type is the only way to define constants in XièXie. Here are a few examples:

```
var    i  := 1          // i is from type 'int'
var    f  := 3.5        // f is from type 'float'
var    s  := "."        // s is from type 'String'
var    a  := { 5 }      // a is from type 'array of int'
var    aa := {          // aa is from type 'array of array of String'
```

```
  { "test" },
  { "a", "b" }
}

const ci := 5        // ci is a constnat int with value 5
const cj := ci*2     // cj is a constant int with value 10
const cf := 3.14     // cf is a constant float
const cb := ci > cj // cb is a constant bool with value 'false'
```

2.4 Statements

Unlike C++ and JAVA, there are no semicolons in XIÈXIE to terminate statements. Only the regular for-statement has two semicolons, to separate the initializer statement, the condition expression, and the increment statement. This means the compiler (or rather the *parser*, which reads the source code) always knows when a statement or expression is complete. But it also means that you — the programmer — can do weird things with this syntax. Consider the following code sample:

```
int a:=3
-4
,b:=-
5+2 int c
```

This is valid XIÈXIE code and it contains only two statements! If we write it in a more common convention, it may look like this:

```
int a := 3-4,
    b := -5+2
int c
```

Hence, the readability of your code is up to you and your programming style :-). The only language I've worked with, which forces you to practice better readability is PYTHON. Actually a great principle, but with XIÈXIE you have complete freedom.

The absence of statement terminators is the reason for the *double paren* syntax of attributes:

```
[[attribute]]
```

To understand why this is the case, let's assume attributes are written with a single paren and consider the following class declaration:

```
class Widget {
    const c := 0 // Initialize member constant 'c' with 0
    int   v := c // Initialize member variable 'v' with constant 'c'
    [final]      // Mark next procedure with attribute 'final'
    void proc() {}
```

```
}
```

Now this doesn't seem very complex. But the parser runs into trouble when reading `[final]`. This is because the parser reads it as follows:

```
int v := c[final]
```

But c is not an array. This is why attributes are written with a double paren, because array accesses never begin with '`[[`'. They may end with '`]]`', but this is not important for the parser.

In the following sections, we will see several types of statements, which are:

- *Branch* Statements: **`if`**, **`switch`**.
- *Loop* Statements: **`for`**, **`foreach`**, **`while`**, **`do-while`**, **`repeat`**.
- *Control Transfer* Statements: **`break`**, **`continue`**, **`return`**.

2.4.1 Branch Statements

A branch statement represents a branch in the control flow of a program, i.e. the program may pass through different paths during execution, depending on which branch is taken.

`if` Branch

```
// If x is greater than y, then execute the following code block
if x > y {
    doSomethingUseful()
}

// Nested boolean expression: x must be equal to y,
// and the condition inside the parentheses must be 'false'
if x = y and not ( x != 3 or y = -7 ) {
    thenDoThis()
} else {
    otherwiseDoThat()
}

// Classic if/elseif/else statement
if conditionA {
    /* ... */
} else if conditionB {
    /* ... */
} else if conditionC {
    /* ... */
} else {
    /* ... */
}
```

switch Branch

```
// Switch statement with many cases (only integers are allowed)
int idx := getSwitchIndex()
switch idx {
    case 1:
        print("case 1") // idx = 1
    case 2:
        print("case 2") // idx = 2
    case 3 .. 10:
        print("case 3") // idx >= 3 and idx <= 10
    case 11, 20 .. 25, 30:
        print("case 4") // idx = 11 or (idx >= 20 and
                        // idx <= 25) or idx = 30
    default:
        print("default case")
}

// Switch with break statements
const caseIndex := 11
int idx := getSwitchIndex()
switch idx {
    case 1:
        int x := 42 // each case has its own scope
    case 2 .. 10:
        int x := idx*2 // declares a new variable in this scope
        if x > 10 {
            break // jump out of switch statement
        }
        print("hi there!")
    case caseIndex:
        print("case \(caseIndex)")
}
```

2.4.2 Loop Statements

It follows several examples of loop statements.

for Loop

```
// This regular for loop iterates 'i' from 0 to 9 (similar to Java)
for int i := 0 ; i < 10 ; i++ {
    // Infinite loop (also similar to Java)
    for ;; {
        if i >= 0 {
            // Break infinite loop
            break
        }
    }
```

```
    // Inner iteration variable 'j' is implicit initialized to 0.0
    float j
    for ; j < 3.5 ; {
        j += 0.5
    }
}
```

Range-Based for Loop

```
// Print numbers 1, 2, 3, and 4
for i : 1 .. 4 {
    // Print value of 'i' (this 'i' is not mutable!)
    print(i)
}

// Print numbers 10, 7, 4, 1, -2, -5, and -8
const range := 10
for i : range .. -range -> 3 {
    print(i)
}
```

foreach Loop

```
// Iterate over array with elements 1, 2, and 3
foreach i : { 1, 2, 3 } {
    print(i)
}

// Iterate over a 'superList' from type (array of arrays of strings)
String[][] superList := { { "a", "b" }, { "c" } }
foreach list : superList {
    // Iterate over all elements in the current sub list
    foreach str : list {
        // Do something with this string
        print(str)
    }
}
```

repeat Loop

```
// Repeat for unconditional iterations
// (This is internally a "ForEver"-loop)
repeat {
    // Condition to break the loop
    if magicFunction() {
        // Break loop
        break
```

```
    }
}

// Do something 10 times (with invisible index variable)
// -> This is internally a 'range-based for loop',
//    i.e. equivalent to 'for i : 1 .. 10'
repeat 10 {
    doSomething()
}
```

while Loop

```
// Regular while loop
while magicFunction() {
    doSomething()
}
```

do/while Loop

```
// Regular do-while loop
do {
    doSomething()
} while magicFunction()
```

2.5 Expressions

2.5.1 Macros

XièXie does not support the declaration of macros. However, there are a few built-in macros, which represent additional reserved identifiers:

- **__FILE__** String which contains the current filename.
- **__CLASS__** String which contains the current class name.
- **__PROC__** String which contains the current procedure name.
- **__LINE__** Integer which contains the current line number.
- **__DATE__** String which contains the current date and time.
- **__VERSION__** Integer which contains the compiler version number (e.g. 200 for version "2.00").

Chapter 3

Classes

A XièXie program can only consist of imports, modules, and class declarations. And classes can only be defined in the global scope. That means every procedure must be defined inside a class and inner classes are currently not supported.

3.1 Getting Started

To get started, take a look at the following example program which prints the classical phrase "Hello, World!" onto the standard output:

```
1 // XieXie Hello World Program
2 import System
3 class HelloWorld {
4     static void main() {
5         System.out.writeLine("Hello, World!")
6     }
7 }
```

This merely writes the line "`Hello, World!`" to the standard output. Let's take a closer look at each line.

Line 2 imports the "System.xx" file from the XièXie standard library:

```
import System      // either this ...
import "System.xx" // ... or this
```

If the imported file is in another directory, the string version of `import` is the only choice. This can also be omitted if the file is added to the compilation process. The files for the classes `Object`, `String`, `Array`, and `Intrinsics` are always implicit imported, because they are 'internal' classes the compiler knows generally. Note that verbatim strings are allowed wherever string literals are allowed, i.e. the following example is valid XièXie code:

```
import "C:\\Program Files\\Test1.xx" // either this ...
import @"C:\Program Files\Test1.xx"  // ... or this
```

The `import` keyword is different to that in Java and also different to the `#include` directive in C++. Although it takes a filename as parameter (like C++'s `#include`), it does not *include* the file in place. Whenever an `import` is read by the *parser*, the filename is added to the set of import files. After all source files have been read, which were passed as input to the compiler, all import files will be read next. This will be repeated until no new files are added to the set. Consider this is a *set* of files, i.e. several `import` commands may occur with the same filename, but it will be read only once. This is why the above sample is valid XièXie code. It also means that recursive imports are allowed as well:

```
// File1.xx
import "File2.xx"
```

```
// File2.xx
import "File1.xx"
```

Line 3 declares the class `HelloWorld` which implicit inherits from the base class `Object`, like it is done in Java:

```
3 class HelloWorld { /* ... */ }
```

To inherit from other classes, just write a colon and the identifier of the base class:

```
class SubClass : BaseClass { /* ... */ }
```

There is no multiple inheritance like in C++ or interfaces like in Java!

The next line declares the procedure `main`.

```
4 static void main() {
```

This is the main entry point for the program. There can only be one main entry point, but there are several possible signatures for this procedure:

```
// No return value, no arguments
static void main() { /* ... */ }

// No return value, arguments
static void main(String[] args) { /* ... */ }

// Return value, no arguments
static int main() { /* ... */ }

// Return value, arguments
static int main(String[] args) { /* ... */ }
```

The last line of code prints the message to the standard output:

```
System.out.writeLine("Hello, World!")
```

`System` is a class from the standard XièXie library, `out` is a *static* member from the type 'OutputStream', and `writeLine` is a function which takes a string as input.

3.2 Declaration Rules for Classes

In XièXie there is no need for *forward declarations*. Everything can be declared in the respective scope and is accessible throughout the entire program (except private scope). This is why the following code is valid:

```
// First declare sub class
class SubClass : BaseClass { /* ... */ }

// Then declare base class
class BaseClass { /* ... */ }
```

The same applies for procedure declarations:

```
class B {
    static void procB1() {
        A.procA()  // no forward declaration required ...
        B.procB2() // ... same here
    }
    static void procB2() { /* ... */ }
}
class A {
    static void procA() { /* ... */ }
}
```

This works because the *context analyzer* of the compiler works in several phases:

1. Class symbols are registered in global scope.
2. Class signatures are analyzed (attributes and base class).
3. Class inheritance is verified (check for cycles).
4. Member procedures are registered in respective class scope.
5. Member variables are registered in respective class scope.
6. Run-time type information (RTTI) for the entire class hierarchy is generated.
7. Procedure code is analyzed.

3.3 Procedures

In XièXie we talk about *procedures* (somethings called *methods*), because in the strict sense *functions* have no side effects. Functions have only input parameters which are calculated to a result. But in XièXie every procedure can have side effects, meaning that they can modify the program state (with static variables for instance).

3.3.1 Procedure Signature

A *procedure signature* is the extended identification of a procedure beyond its identifier string. The signature consists of the identifier string, its parameter list, and the procedure's return type. However, the return type is never used for identification.

3.3.2 Procedure Overloading

XièXie supports overloading of procedures. This means the same identifier can be used several times for procedures inside a class declaration (including its inheritance hierarchy). This requires that the following rules are satisfied:

> All procedures with the same identifier inside a class declaration can be distinguished by their parameter count or parameter types.

Here is an example of procedure overloading.

```
class Widget {
    int f() {
        return 1
    }
    int f(int x) {
        return x*2
    }
    float f(float x) {
        return x*3.0
    }
    void caller() {
        int   a := f()     // a is 1
        int   b := f(1)    // b is 2
        float c := f(2.0)  // c is 6.0
    }
}
```

When overloading procedures, try to avoid ambiguities with default arguments. Adding default arguments to all parameters of all overloaded procedures is allowed, but procedure calls may be ambiguous for the compiler:

```
class Widget {
    int f() { /* ... */ }
    int f(int x := 0) { /* ... */ }
    float f(float x := 0.0) { /* ... */ }

    void caller() {
        int   a := f()     /* Error, could be f(), f(int x := 0),
                              or f(float x := 0.0) */
        int   b := f(1)    /* OK, argument is from type 'int' */
        float c := f(2.0)  /* OK, argument is from type 'float' */
    }
}
```

3.3.3 Procedure Overriding

XièXie supports overriding of procedures. This means the same procedure signature can be used inside a class and its base class. The procedure calls of overloaded and overridden procedures require that the following rules are satisfied:

> If a procedure with identifier P is declared inside a class C, a procedure with the same identifier is declared in its base class B but with another signature, and another procedure inside class C calls the procedure P from class B, then the identifier super must be specified in front of the call.

To better understand this awkward definition, take a look at this example:

```
class B {
    int f(int x, int y) {
        return 1
    }
    int g() {
        return 2
    }
}
class S : B {
    int f(int x) {
        return 3
    }
    void caller() {
        int a := f(0)          /* Equivalent to 'this.f(0)' */
        int b := super.f(0, 0) /* 'super' is required,
                                  due to overloaded procedure 'f' */
        int c := g()           /* No need for 'super',
                                  because 'g' is not overloaded */
    }
}
```

3.3.4 Named Arguments

Named arguments, like in C#, are supported as well. They also have the same rules as in C#, i.e. if you started to write named arguments in a procedure call, you need to finish it till the last argument. Here is an example and counter-example:

```
// Example:
class Widget {
    void f(int x := 1, int y := 2, int z := 3)
    void call() {
        int y := 12
        f()                    // Use all default arguments
        f(y: -5)               // Equivalent to f(1, -5, 3)
        f(y: y)                // Equivalent to f(1, y, 3)
        f(-1, z: 0)            // Equivalent to f(-1, 2, 0)
        f(z: 1, y: 2, x: 3) // Equivalent to f(3, 2, 1)
    }
}

// Counter-example:
class Widget {
    void f(int x := 1, int y := 2, int z := 3)
    void call() {
        f(1, x: 1)     /* Error, parameter 'x' already assigned */
        f(1, y: 2, 3) /* Error, can not start with named
                          arguments and continue with unnamed */
    }
}
```

But what are named arguments good for? Consider the following function signature:

```
// User account class.
class UserAccount {
    init(
        String name := "<unnamed>",
        int age := 0,
        String location := "",
        String city := "",
        int postalCode := 0
        int flags := 0
    )
    /* ... */
}

// Creates an account, only initialize with basic information.
class AccountManager() {
    // Alternative 1
    UserAccount createAccountAlt1(String name, int flags := 0) {
        return new UserAccount(name: name, flags: flags)
    }

```

```
21      // Alternative 2
22      UserAccount createAccountAlt2(String name, int flags := 0) {
23          return new UserAccount(name, 0, "", "", 0, flags)
24      }
25  }
```

In this example alternative 2 (Line 23) is not much more to type. But alternative 1 (Line 18) is much better to read :-). You see immediately which arguments are passed to which parameters. No matter if the parameter order will change, your procedure call will still work as suggested. Moreover, the default arguments in the procedure declaration may change, but your procedure call will still use them correctly.

3.4 Attributes

There are several attributes for class-, procedure-, and variable declarations:

```
// Class A is marked as 'deprecated'.
[[deprecated]]
class A {
    // Declare some procedure
    void p() {}
}

/* Class B is marked as 'deprecated' with a hint.
   Since B is deprecated as well, the deprecation
   of A is ignored here. */
[[deprecated("hint...")]]
class B {
    /* Mark 'p' to override 'A.p'. If 'p' does not match the
       signature of procedure 'A.p', the compiler will
       throw an error message. */
    [[override]]
    void p() {}
}

/* Class C is 'final', i.e. no class can inherit from C.
   Because A is deprecated but not C,
   the compiler will throw a warning. */
[[final]]
class C : A {}
```

Here is a summary of all attributes:

- **`deprecated`**, **`deprecated(String hint)`** Marks a *class*, *procedure*, or *member-variable* as deprecated.
- **`export`**, **`export(String label)`** Exports the address of a *static procedure*. The default label is `"CLASS.PROCEDURE"`, where `CLASS` is the class

name, and `PROCEDURE` is the procedure name (without name mangling). This can be used to start the program from a specific static procedure. This should be some kind of an entry point!

- **`bind`**, **`bind(String name)`** Binds a *module* of the specified name. The default name is: `"NAME/NAME"`, where `NAME` is the name of the module. Binding a module will make it loaded automatically when the program is started within the `xvm`.
- **`override`** Overrides a procedure from a class in its inheritance hierarchy.
- **`final`** Makes a class final, which disables to inherit from this class.

3.5 Anonymous Classes

XièXie supports the declaration of anonymous classes. This can be quite comfortable when a class needs to be instantiated only once:

```
class Hello {
    /* No code block after procedure declaration inside
       a non-extern class implies an abstract procedure */
    String say()
}

class EnglishHello : Hello {
    [[override]]
    String say() { return "Hello" }
}

class Main {
    static void main() {
        /* Allocate class "EnglishHello" */
        var englishHello = new EnglishHello()

        /* Allocate anonymous class
          (internal name "__xx__AnonymousClass0") */
        var germanHello = new Hello() {
            [[override]]
            String say() { return "Hallo" }
        }
    }
}
```

Now it should be clear why identifiers with the prefix "__xx__" are reserved.

3.6 Class Visibilities

Like in C++ and JAVA, XIÈXIE provides visibilites for all members within a class. The syntax for these visibilites are a mixture of C++ and JAVA. You can declare the visibility for each class member like in JAVA and/or for a whole bunch of members like in C++:

```
class C {
public:
              int      publicVar0
              float    publicVar1
    private   int      privateVar0
              String   publicVar2
    protected int      protectedVar0
    private   float[]  privateVar1
              int[]    publicVar3
private:
              int      privateVar2
    public    bool     publicVar4
              bool[]   privateVar3
}
```

Unlike in common object-oriented languages, the default visibility in XIÈXIE is `public`. This is because there are only classes and no structures like in C++. So a small class, which only has variables and acts like a data structure can be written with as less effort as possible:

```
// Examples of simple data structures
class Vector3 { float x, y, z }
class FileHeader {
    int magicNumber
    int formatVersion
}
```

Like in C++, there are also *'friends'*. And like in C++ as well, friendship is not inherited:

```
class Companion {
    protected:
        static int a
        friend MyFriend
    private:
        static int b
}
class MyFriend {
    static void f() {
        var a := Companion.a /* OK, "MyFriend" is a
                                'protected friend' of "Companion" */
        var b := Companion.b /* Error, "MyFriend" is not a
                                'private friend' of "Companion" */
    }
}
```

```
class Rival {
    static void f() {
        var a := Companion.a /* Error, "Rival" is not a
                               'protected friend' of "Companion" */
    }
}
```

Chapter 4

Modules

A *module* consists of a shared library (`*.dll` on Windows and `*.so` on GNU/Linux) and a XièXie module file. They are similar to Python modules.

4.1 Using Modules

Module declarations are similar to external classes. But they can only have static procedures. Here is an example:

```
module MyModule {
    static void doSomething(int x)
}
```

Using this in your XièXie code will then look like this:

```
import MyModule
class MyClass {
    static void main() {
        MyModule.doSomething(42)
    }
}
```

4.2 Creating Modules

Modules must be written in plain C or C++. A minimal module would consist of three files. Supposed our example module is named "MyModule" we would have these files:

- **`MyModule.c`** Module code written in C.

- **MyModule.dll** (Windows)/ ***.so** (GNU/Linux) Shared library which runs the module code.

- **MyModule.xx** XièXie code file which declares the module interface.

The C code file must implement the following function interfaces to be a valid XièXie module:

```
// Returns the number of module procedures.
int xx_module_proc_count();

// Returns the procedure for the respective index.
XVM_INVOCATION_PROC xx_module_fetch_proc(int index);

// Returns the procedure identifier for the respective index.
const char* xx_module_fetch_ident(int index);
```

Our simple module example from above could be implemented as follows:

```
// Include XieXie module header
#include <xiexie/xx_module.h>

// This is our module procedure "soSomething",
// all module procedures must have this interface
void doSomething(XVM_Env env) {
    // Get 1st parameter from XVM environment
    int x = XVM_ParamInt(env, 1);

    // Do something with 'x'
    printf("input parameter 'x' is %i\n", i);

    // Pop arguments from stack (1 for the single parameter 'x')
    XVM_ReturnVoid(env, 1);
}

// Implement the module interface
static XVM_INVOCATION_PROC procList[] = {
    { "doSomething", doSomething },
};

XVM_IMPLEMENT_MODULE_INTERFACE(procList);
```

Part II

Developer Tools

Chapter 5

Compiler

5.1 Command Line Tool

The compiler can be used as command line tool. The appropriate program is named `xxc`. Enter `xxc help` in a command line to see the manual pages.

In contrast to most other command line tools, the commands for `xxc` don't have the '-' prefix. Instead the command options use this prefix. Here is an example:

```
xxc C -f FILE1 -f FILE2 -O
```

The above command line uses the `C` command (also `compile`) with the flags '-f' and '-O'.

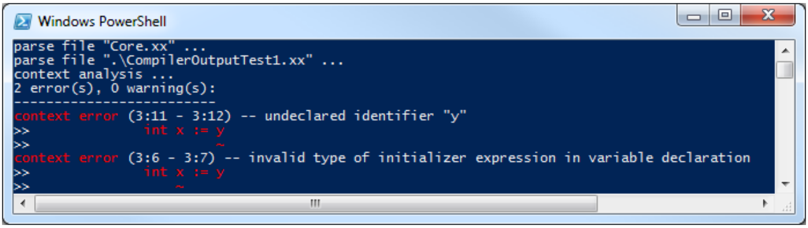
5.1.1 Output

The compiler tries to show you where an error or warning occurred. If possible, it is highlighted with a *line marker*. Consider the following code sample:

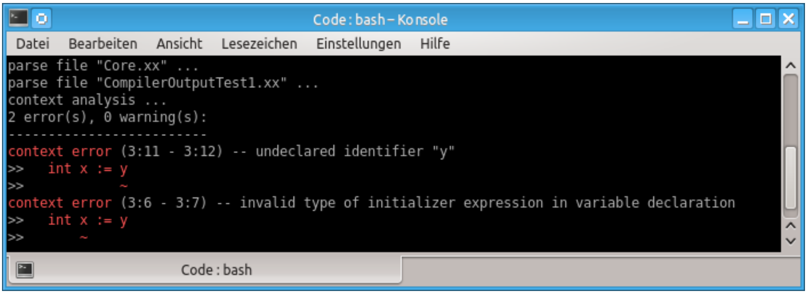
```
class Foo {
    void Bar() {
        int x := y
    }
}
```

Now consider the compiler processes this code with the following command line:

```
xxc C -f Core.xx -f Foo.xx
```



(a) Output in PowerShell on Windows 7



(b) Output in Terminal on Kubuntu GNU/Linux

Figure 5.1: Compiler output for the above code sample.

Then the compiler indicates that the variable 'y' was not declared (see Figure 5.1). As you can see, some errors may produce multiple outputs. In this case, 'y' is undeclared which produces two error outputs here:

1. The identifier 'y' is undeclared, so the appearance in the initializer expression is invalid.

2. The type of the initializer expression for the declaration of 'x' can not be deduced, so the type check fails.

5.1.2 Script

If `xxc` has no arguments, it will switch to the *immediate script* mode. This allows the user to immediately script XièXie code inside the command line (see Figure 5.2).

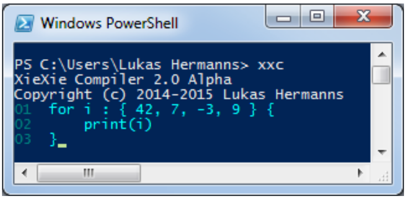


Figure 5.2: Compiler immediate script mode.

Chapter 6

Virtual Machine

The XièXie Virtual Machine (XVM) is a separated program (named `xvm`), written in pure C99.

6.1 Executing Programs

To execute (or run) a virtual program, just enter the filename of an `*.xbc` file into the `xvm`:

```
xvm HelloWorld.xbc
```

Part III

Low-Level Programming

Chapter 7

Virtual Assembler

The XièXie Virtual Assembler (XASM) is the low level language which is used as interface to the `xvm`. It's a very low level language, with similarities to ARM® Assembler.

7.1 Instruction Set

7.1.1 Registers

The `xvm` is a register machine, i.e. it uses registers as operands for its instructions instead of the stack. This commonly increases performance but is a little more tricky to use, when you run out of registers. Each register index inside an instruction is stored with 4 bits, thus there are 16 registers and this is the list:

1. **`$r0`** General Purpose Register 0.

   ⋮

25. **`$r24`** General Purpose Register 24.
26. **`$ar`** Argument Return.
27. **`$xr`** Extended Register.
28. **`$gp`** Global Pointer.
29. **`$cf`** Conditional Flag.
30. **`$lb`** Local Base Pointer.
31. **`$sp`** Stack Pointer.
32. **`$pc`** Program Counter.

All pointer registers and the program counter (i.e. `$gp`, `$lb`, `$sp`, and `$pc`) are all 'real' pointers to the memory in your operating system.

7.1.2 OpCodes

Each instruction stores its mnemonic (Greek ‘memory’ $\widehat{=}$ instruction identifier) in the first 6 bits. This provides a maximum of 64 instructions. However, only 52 mnemonics are currently in use.

3-Register Instruction OpCodes (00....)						
Mnemonic	OpCode 31.......26	Dest. 25.....21	LSource 20.....16	RSource 15.....11	Unused 10..................0	Description
AND	0 0 0 0 0 1	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Bitwise AND (D := L & R).
OR	0 0 0 0 1 0	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Bitwise OR (D := L | R).
XOR	0 0 0 0 1 1	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Bitwise XOR (D := L ^ R).
ADD	0 0 0 1 0 0	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Integer add. (D := L + R).
SUB	0 0 0 1 0 1	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Integer sub. (D := L - R).
MUL	0 0 0 1 1 0	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Integer mul. (D := L × R).
DIV	0 0 0 1 1 1	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Integer div. (D := L ÷ R).
MOD	0 0 1 0 0 0	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Modulo (D := L % R).
SLL	0 0 1 0 0 1	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Shift left (D := L « R).
SLR	0 0 1 0 1 0	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Shift right (D := L » R).
ADDF	0 0 1 0 1 1	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Float add. (D := L + R).
SUBF	0 0 1 1 0 0	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Float sub. (D := L - R).
MULF	0 0 1 1 0 1	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Float mul. (D := L × R).
DIVF	0 0 1 1 1 0	D D D D D	L L L L L	R R R R R	0 0 0 0 0 0 0 0 0 0 0	Float div. (D := L ÷ R).

2-Register Instruction OpCodes (01....)					
Mnemonic	OpCode 31.......26	Dest. 25.....21	Source 20.....16	Unused or Value 15..........................0	Description
MOV	0 1 0 0 0 0	D D D D D	S S S S S	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Move (D := S).
NOT	0 1 0 0 0 1	D D D D D	S S S S S	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Bitwise NOT (D := ~S).
FTI	0 1 0 0 1 0	D D D D D	S S S S S	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Float to integer (D := (int)S).
ITF	0 1 0 0 1 1	D D D D D	S S S S S	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Integer to float (D := (float)S).
AND	0 1 0 1 0 0	D D D D D	S S S S S	V V V V V V V V V V V V V V V V	Bitwise AND (D := S & V).
OR	0 1 0 1 0 1	D D D D D	S S S S S	V V V V V V V V V V V V V V V V	Bitwise OR (D := S | V).
XOR	0 1 0 1 1 0	D D D D D	S S S S S	V V V V V V V V V V V V V V V V	Bitwise XOR (D := S ˆ V).
ADD	0 1 0 1 1 1	D D D D D	S S S S S	V V V V V V V V V V V V V V V V	Modulo (D := S % V).
SUB	0 1 1 0 0 0	D D D D D	S S S S S	V V V V V V V V V V V V V V V V	Shift left (D := S « V).
MUL	0 1 1 0 0 1	D D D D D	S S S S S	V V V V V V V V V V V V V V V V	Shift right (D := S » V).
DIV	0 1 1 0 1 0	D D D D D	S S S S S	V V V V V V V V V V V V V V V V	Integer add. (D := S + V).
MOD	0 1 1 0 1 1	D D D D D	S S S S S	V V V V V V V V V V V V V V V V	Integer sub. (D := S - V).
SLL	0 1 1 1 0 0	D D D D D	S S S S S	V V V V V V V V V V V V V V V V	Integer mul. (D := S × V).
SLR	0 1 1 1 0 1	D D D D D	S S S S S	V V V V V V V V V V V V V V V V	Integer div. (D := S ÷ V).
CMP	0 1 1 1 1 0	X X X X X	Y Y Y Y Y	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Integer comparison (→ `$cf`).
CMPF	0 1 1 1 1 1	X X X X X	Y Y Y Y Y	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Float comparison (→ `$cf`).

1-Register Instruction OpCodes (100...)				
Mnemonic	OpCode 31.......26	Reg. 25.....21	Unused, Value, or Offset 20....................................0	Description
PUSH	1 0 0 0 0 0	R R R R R	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Push register onto stack.
POP	1 0 0 0 0 1	R R R R R	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Pop register from stack.
INC	1 0 0 0 1 0	R R R R R	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Increment integer (D++).
DEC	1 0 0 0 1 1	R R R R R	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Decrement integer (D–).
MOV	1 0 0 1 0 0	R R R R R	V V V V V V V V V V V V V V V V V V V V V	Move integer (D := V).
LDA	1 0 0 1 0 1	R R R R R	O O O O O O O O O O O O O O O O O O O O O	Load address from program (D := ByteCode + R_{word} + O_{byte}).

Jump Instruction OpCodes (101...)				
Mnemonic	OpCode 31.......26	Reg. 25.....21	Offset 20...................................0	Description
JMP	1 0 1 0 0 0	R R R R R	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Jump.
JE	1 0 1 0 0 1	R R R R R	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Jump if equal.
JNE	1 0 1 0 1 0	R R R R R	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Jump if not-equal.
JG	1 0 1 0 1 1	R R R R R	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Jump if greater.
JL	1 0 1 1 0 0	R R R R R	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Jump if less.
JGE	1 0 1 1 0 1	R R R R R	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Jump if greater or equal.
JLE	1 0 1 1 1 0	R R R R R	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Jump if less or equal.
CALL	1 0 1 1 1 1	R R R R R	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Push dynamic link ($1b and $pc) onto stack; Set $1b to new stack frame; Jump to address.

Load/Store Instruction OpCodes (1100..)					
Mnemonic	OpCode 31.......26	Reg. 25.....21	Addr. 20.....16	Offset 15...........................0	Description
LDB	1 1 0 0 0 0	R R R R R	A A A A A	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Load byte from memory.
STB	1 1 0 0 0 1	R R R R R	A A A A A	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Store byte to memory.
LDW	1 1 0 0 1 0	R R R R R	A A A A A	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Load word from memory.
STW	1 1 0 0 1 1	R R R R R	A A A A A	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Store word to memory.

Special-1 Instruction OpCodes			
Mnemonic	OpCode 31.......26	Unused or Value 25...0	Description
STOP	0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Stop program execution.
PUSH	1 1 1 0 0 0	V V V V V V V V V V V V V V V V V V V V V V V V V V	Push integer onto stack.
INVK	1 1 1 0 0 1	V V V V V V V V V V V V V V V V V V V V V V V V V V	Invoke external procedure.
INSC	1 1 1 0 1 0	V V V V V V V V V V V V V V V V V V V V V V V V V V	Call intrinsic.

Special-2 Instruction OpCodes

Mnemonic	OpCode 31.......26	Result Size 25...............16	Argument Size 15...........................0	Description
RET	1 1 1 0 1 1	R R R R R R R R R R	A A A A A A A A A A A A A A A A	Pop R wrods from stack and buffer them; Pop current stack frame; Pop A words from stack; Push stored R words back onto stack; Restore dynamic link ($1b and $pc)

7.1.3 Memory Alignment

Some instructions use *byte* alignment and some others use *word* alignment.

7.2 Procedure Calling

7.2.1 Calling Convention

In XASM, the calling convention provides that the *callee* removes the procedure arguments from the stack. After any procedure call, the *dynamic link* is stored at the beginning of the *stack frame* and reserves the first 8 bytes.

dynamic link

Consists of two words: 32-bit value of the `$lb` register *before* the procedure call and 32-bit value of the `$pc` register *before* the procedure call.

Local variables can be store by increasing the *stack pointer* (`$sp`) and load/store instructions at the *local base pointer* (`$lb`) plus 8 bytes (after the *dynamic link*). Here is an example:

```
; void proc() { ... }
proc:
  ADD $sp, $sp, 8    ; Allocate memory on the stack ...
                     ; ... for two words (2*4 bytes)
  MOV $r0, 42        ; Store value 42 in register $r0
  STW $r0, ($lb) 8   ; Store register $r0 in local scope ...
                     ; ... at first position (after dynamic link)
  ADD $r0, $r0, 10   ; Increment value of register $r0
  STW $r0, ($lb) 12  ; Store register $r0 in local scope ...
                     ; ... at second position
  ...
```

The arguments for a procedure call are meant to be pushed onto the stack from *right-to-left*. In this way the arguments can be accessed from *left-to-right*. Here is an example:

```
; int proc(int x, int y) { return x+y }
proc:
  LDW $r0, ($lb) -4  ; Fetch first argument (x) from stack
  LDW $r1, ($lb) -8  ; Fetch second argument (y) from stack
  ADD $ar, $r0, $r1  ; Calculate ar := x + y
  RET 2              ; Return: $ar and pop 2 arguments from stack
```

7.2.2 Intrinsics

Intrinsics are the internal primitive procedures of the `xvm`. All intrinsics store the result (if they have one) in the `$ar` register. An intrinsic can be used like this:

```
; Allocate memory for 10 bytes.
; Pointer will be in $ar register.
PUSH 10
INSC AllocMem
```