Оглавление

Введение

Данный язык написан на Java, и многие конструкции позаимствованы из языка JavaScript. При описании работы я буду ссылаться на термины других языков и на особенности языка Java.

Краткое описание языка

Язык lang2 по сути представляет небольшой простой язык с набором переменных :) не давай те не так, я опишу лучше тезисно:

- Типы данных в языке динамические
  - Имеются числовые типы данных
  - Логические типы
  - Строки
  - Списки
  - Объекты / карты
  - Функции
- Имеются основные конструкции общие для многих языков
  - Комментарии
  - Последовательный блок инструкций
  - Ветвление
  - Циклы
  - Функции
  - Объекты
- Память используемая для хранения значений
  - Является динамической и управляется сборщиком мусора
  - Все ссылки на объекты, списки и функции это именно ссылки

Hello world

Создаем файл helloworld.l2:

```
# coding: utf-8
println( "Hello world!" );
```

Первая строка (# coding: utf-8) указывает на кодировку файла.

Строки в начале файла начинающие с символа решетки считаются комментариями, такие комментарии допускаются только в начале файла. В них допускается использовать после решетки любые символы. Эти комментарии могут носить служебный характер для загрузчика файла, в частности указывают на кодировку файла.

Далее из командной строки запускаем

```
[user@debian samples]$ ./lang2 helloworld.l2
Hello world!
```

Bash/Shell скрипты

Структура файлов позволяет использовать в качестве UNIX Shell Скриптов. Создадим файл printargs.l2 cj со следующим содержанием:

```
#!/bin/sh
exec lang2 -f "$0" -- "$@"
!#
# coding: utf-8

println( "print arguments" );
for( a in arguments ){
   println( a );
}
```

Выставим права на исполнение файла

```
chmod a+x printargs.l2
```

После запустим на выполнение с парой аргументов

```
[user@debian samples]$ ./printargs.l2 "first argument" "second argument"
print arguments
./printargs.l2
first argument
second argument
```

Теперь рассмотрим подробнее исходный код.

```
#!/bin/sh
exec lang2 -f "$0" -- "$@"
!#
```

Эти первые три строки являются специальным UNIX заголовком, в котором объясняется как запускать файл.

- **`#!/bin/sh`** — указывает что должна использоваться программа /bin/sh, путь до программы sh должен быть указан абсолютно, на большинстве UNIX систем

программа sh располагается в каталоге /bin. В данном случаи программа sh построчно читает содержимое файла и выполняет прочитанное.

- **`exec lang2 -f "$0" -- "$@"`** — на этом шаге программа sh:
  - передает управление программе **`lang2`** (**`exec`**), путь до которой прописан в переменных окружения и завершает свою работу
  - **`-f "$0"`** — указывает исполняемый файл, в данном случаи это будет сам исполняемый файл (./printargs.l2)
  - **`-- "$0"`** — передает принятые аргументы (first argument и second argument)
- **`!#`** — это строка отмечает конец специального UNIX заголовка

Следующая строка **`# coding: utf-8`** указывает программе **`lang2`** кодировку файла (**`utf-8`**), а последующий текст представляет из себя исходный код программы, который интерпретируется движком L2.

Комментарии которые начинаются с решетки введены для совместимости с UNIX и должны быть расположены только в начале программы, в грамматике языка они не описаны.

Примеры

```
if( notExistsVar ){
   1;
} else {
   2;
}
```

```
2
```

Блок операций

```
{
   1+2;
   2+2
}
```

```
4
```

```
{
   var a=1;
   {
       var a=2
   }
   a
}
```

```
1
```

Коментарии к функции

```
var testFun = /** testFun comment */ function( a, b ){ a + b };
testFun.comment
```

```
testFun comment
```

```
var sumFun = /** sumFun comment */ function( a, b ){ a + b };
/** mulFun comment */ var mulFun = function( a, b ){ a * b };
sumFun.comment + mulFun.comment
```

```
sumFun commentmulFun comment
```

Операции сравнения

```
1==1
```

```
true
```

```
2<>1
```

```
true
```

```
2>1
```

```
true
```

```
2<1
```

```
false
```

```
2>=2
```

```
true
```

```
2>=1
```

```
true
```

```
1<=2
```

```
true
```

```
1<=1
```

```
true
```

```
"abc" < "def"
```

```
true
```

```
"abc" == "def"
```

```
false
```

Унарный минус

```
-1+1
```

```
0
```

Умножение, сложение, ...

```
(2+2-3)*4/2
```

```
2
```

```
"abc" + "def"
```

```
abcdef
```

```
13 % 10
```

```
3
```

Логические операции

```
true and true
```

```
true
```

```
true and false
```

```
false
```

```
true and not false
```

```
true
```

```
true or true
```

```
true
```

```
true or false
```

```
true
```

```
true xor true
```

```
false
```

```
true xor false
```

```
true
```

```
false xor true
```

```
true
```

```
false xor false
```

```
false
```

Присваивание

```
d = c = a * 3 + b
```

```
5.0
```

Проверка массивов

```
var lst = [ "str", "ing" ];
lst[0] + lst[1]
```

```
string
```

```
var lst = [ 1, 2 ];
lst.length
```

```
2
```

Определение переменной

```
var newVar = 1 + 2
```

```
3
```

Условный оператор

```
1<2 ? "yes"
```

```
yes
```

```
1>2 ? "yes" : "no"
```

```
no
```

Цикл

```
var i=0;
while( i<5 ) i=i+1;
i
```

```
5
```

Вызов функции

```
print( 1+2, "abc" )
```

```
null
```

```
summa( 1, 2, 3 )
```

```
6.0
```

```
external.add( 1, 2 )
```

```
3.0
```

```
external.add( "con", "cat" )
```

```
concat
```

Вызов списка как функции

```
var list = [ "a", "b", "c"];
list( 0 );
```

```
a
```

```
var list = [ "a", "b", "c"];
list( 1 );
b
```

```
var list = [
   [ "a", "a.1" ],
   [ "b", "b.1" ],
   [ "c", "c.1" ]
];
list( 0,1 );
```

```
a.1
```

```
var list = [
   [ "a", "a.1" ],
   [ "b", "b.1" ],
   [ "c", "c.1" ]
];
list( 1,1 );
```

```
b.1
```

Конвертирование в java примитивные типы (boolean, int, ...)

```
external.And( true, true )
```

```
true
```

```
external.summa( 1, 2, 3, 4, 5, 6 )
```

```
21.0
```

```
external.concat( "abc", "def" )
```

```
ad
```

Тестирование создания функций

```
var f1 = function( a, b ) a+b;
f1( 1,2 )
```

```
3
```

```
var f2 = function( a ) a > 2 ? recursion( a-1 ) * a : a;
f2( 4 )
```

```
24
```

Тестирование создания объектов

```
var obj = {
   a : 1+2,
   b : "aa",
   c : true,
   d : function ( a, b ) { a + b }
};
obj.a
```

```
3
```

```
var obj = {
   a : 1+2,
   b : "aa",
   c : true,
   d : function ( self, a, b ) { a + b }
};
obj.b
```

```
aa
```

```
var obj = {
   a : 1+2,
   b : "aa",
   c : true,
   d : function ( self, a, b ) { a + b }
};
obj.c
```

```
true
```

```
var obj = {
   a : 1+2,
   b : "aa",
   c : true,
   d : function ( self, a, b ) { a + b }
};
obj.d( 2, 3 )
```

```
5
```

```
var obj = {
   a : "str",
   b : function ( self, c ){ self.a + c }
};
obj.b( "ing" )
```

```
string
```

Цикл, рекурсия, ...

```
var a = 1;
while( true ){
   a = a + 1;
   a > 5 ? break
}
a
```

```
6
```

```
var a = 1;
var b = [];
while( true ){
   a = a + 1;
   a > 20 ? break;
   a > 5 ? continue;
   b[b.length] = a
}
b
```

```
[2, 3, 4, 5]
```

```
var f1 = function( a ){
   a > 5 ? return a + a;
   a < 0 ? return a – a;
   a * a
}
f1( 6 ) + f1( -2 ) + f1( 1 )
```

```
13
```

Тестирование try / catch

```
try {
   error( "message" );
}
catch( e ) {
   print( e.getMessage() );
   e.getMessage();
}
```

```
message
```

```
try {
   throw "abc";
}
catch( e ) {
   print( e );
   "ok:"+e;
}
```

```
ok:abc
```

```
var fact = function( x ) {
   if ( x < 0 ) throw "factorial ("+x+") error";
   if ( x > 1 ) recursion( x-1 ) * x else 1;
};
try {
   fact( -2 );
}
catch( e ) {
   print( e );
   "ok";
}
```

```
ok
```

FOR Тестирование итераций

```
var src = [ 1 , 2 , 3 ];
var sum = 0;
for( v in src ) {
   sum = sum + v;
}
sum
```

```
6
```

```
var src = {
   idx : 0,
   hasNext : function( self ) { self.idx < 5 },
   next : function( self ) {
       self.idx = self.idx + 1;
```

```
        self.idx
    }
};
var sum = 0;
for( v in src ) {
    sum = sum + v;
}
sum
```

```
15
```

Конвертирование java чисел

```
var a = 1;
a.intValue()
```

```
1
```

```
var a = 2;
a.doubleValue()
```

```
2.0
```

castTest1

```
var l = [ "a", "b", "c" ];
l.remove( 0 );
l.size;
```

```
2
```

mapExtender

```
var m = { a:1, b:2, c:3 };
m.size;
```

```
3
```

```
var m = { a:1, b:2, c:3 };
var k = m.keys;
k.size;
```

```
3
```

```
var m = { a:1, b:2, c:3 };
var k = m.keys;
k[0];
```

```
a
```

```
var m = { a:1, b:2, c:3 };
var k = m.keys;
k[1];
```

```
b
```

```
var m = { a:1, b:2, c:3 };
var k = m.keys;
k[2];
```

```
c
```

listExtender

```
var l = [ "a", "b", "c" ];
l.size;
```

```
3
```

readJREProperties

```
obj.field
```

```
123
```

```
obj.prop
```

```
abc
```

```
obj.prop = "12345";
obj.prop
```

```
12345
```

Взаимодействие с Java средой

Вызов из Java

Исходный код Java

```
// Импорт L2Engine класса
import xyz.cofe.lang2.parser.L2Engine;

public class JavaInteractSample {
   public void sampleInteract(){
       // Создание объекта L2Engine
       L2Engine scriptEngine = new L2Engine();

       // Исходный код программы l2
       String code = "1 + 2";
       Object result = scriptEngine.eval(code);

       System.out.println("source code:");
       System.out.println(code);
       System.out.println("result:");
       System.out.println(result);
   }
}
```

Результат

```
source code:
1 + 2
result:
3
```

Передача переменных в L2Engine

Исходный код Java

```
public void passingVariables(){
   L2Engine scriptEngine = new L2Engine();

   scriptEngine.getMemory().put("name", "John");
   scriptEngine.getMemory().put("summ", 1200);

   String code = "\"Hello \" + name + \" you win \" + summ + \"$\"";
   Object result = scriptEngine.eval(code);

   System.out.println("source code:");
   System.out.println(code);
   System.out.println("result:");
   System.out.println(result);
}
```

Результат

```
source code:
"Hello " + name + " you win " + summ + "$"
result:
Hello John you win 1200$
```

Передача функций в L2Engine

```
import xyz.cofe.lang2.vm.Callable;

...

public void passingFunctions(){
   L2Engine scriptEngine = new L2Engine();

   Callable sinFn = new Callable() {
       @Override
       public Object call(Object... arguments) {
           if( arguments!=null
               && arguments.length>0
               && (arguments[0] instanceof Number) )
               {
                   double n = ((Number)arguments[0]).doubleValue();
```

```
                return Math.sin(n);
            }
            return null;
        }
    };

    scriptEngine.getMemory().put("num1", 0.5);
    scriptEngine.getMemory().put("sin", sinFn);

    String code = "sin( num1 )";
    Object result = scriptEngine.eval(code);

    System.out.println("source code:");
    System.out.println(code);
    System.out.println("result:");
    System.out.println(result);
}
```

```
source code:
sin( num1 )
result:
0.479425538604203
```

Передача объектов Java в L2Engine

```
public static class PassObject {
    public int numField = 100;

    protected String strProperty = "some text";

    public String getStrProperty() {
        return strProperty;
    }
    public void setStrProperty(String strProperty) {
        this.strProperty = strProperty;
    }

    public String concat( String arg1, String arg2 ){
        StringBuilder strBldr = new StringBuilder();
        strBldr.append(arg1);
        strBldr.append(arg2);
        return strBldr.toString();
    }
}

public void passingObjects(){
    L2Engine scriptEngine = new L2Engine();

    PassObject passObj = new PassObject();
    scriptEngine.getMemory().put("inobj", passObj);

    String code = "inobj.concat( inobj.numField.toString(), inobj.strProperty )";
    Object result = scriptEngine.eval(code);
```

```
    System.out.println("source code:");
    System.out.println(code);
    System.out.println("result:");
    System.out.println(result);
}
```

```
source code:
inobj.concat( inobj.numField.toString(), inobj.strProperty )
result:
100some text
```

Регистрация стандартных функций

```
public void cliFunctions() {
    L2Engine scriptEngine = new L2Engine();

    CLIFunctions cliFunctions = new CLIFunctions(scriptEngine);
    scriptEngine.getMemory().putAll(cliFunctions.getMemObjects());

    String code = "println( \"test println function\" )";

    System.out.println("source code:");
    System.out.println(code);
    System.out.println("output:");

    scriptEngine.eval(code);
}
```

```
source code:
println( "test println function" )
output:
test println function
```

Создание Java объекта

```
public void createJavaObj() {
    L2Engine scriptEngine = new L2Engine();

    CLIFunctions cliFunctions = new CLIFunctions(scriptEngine);
    scriptEngine.getMemory().putAll(cliFunctions.getMemObjects());

    String code =
            "var file = java( \"java.io.File\", \".\" );\n"
        +   "println( file.getAbsolutePath() );\n"
        +   "file";

    System.out.println("source code:");
    System.out.println(code);
```

```
   System.out.println("output:");

   Object result = scriptEngine.eval(code);

   System.out.println("result:");
   System.out.println(result);
   System.out.println("result instanceof java.io.File = "+(result instanceof
java.io.File));
}
```

```
source code:
var file = java( "java.io.File", "." );
println( file.getAbsolutePath() );
file
output:
/home/user/code/dev/proj/lang2/.
result:
.
result instanceof java.io.File = true
```

Получение класса Java

```
public void getJavaType(){
   L2Engine scriptEngine = new L2Engine();

   CLIFunctions cliFunctions = new CLIFunctions(scriptEngine);
   scriptEngine.getMemory().putAll(cliFunctions.getMemObjects());

   String code =
           "var clzfile = java.type( \"java.io.File\", \".\" );\n"
       +   "println( clzfile );\n"
       +   "clzfile";

   System.out.println("source code:");
   System.out.println(code);
   System.out.println("output:");

   Object result = scriptEngine.eval(code);

   System.out.println("result:");
   System.out.println(result);

   if( result instanceof Class ){
       Class cls = (Class)result;
       System.out.println("class name: "+cls.getName());
   }
}
```

```
source code:
var clzfile = java.type( "java.io.File", "." );
println( clzfile );
clzfile
```

```
output:
class java.io.File
result:
class java.io.File
class name: java.io.File
```

Создание Java массива

```
public void getJavaArray(){
   L2Engine scriptEngine = new L2Engine();

   CLIFunctions cliFunctions = new CLIFunctions(scriptEngine);
   scriptEngine.getMemory().putAll(cliFunctions.getMemObjects());

   String code =
           "var arr = java.array( \"java.lang.String\", 3 );\n"
       +   "arr[0] = \"first\";\n"
       +   "arr[1] = \"second\";\n"
       +   "arr[2] = \"thrid\";\n"
       +   "println( arr );\n"
       +   "arr";

   System.out.println("source code:");
   System.out.println(code);
   System.out.println("output:");

   Object result = scriptEngine.eval(code);

   System.out.println("result:");
   System.out.println(result);

   if( result!=null && result.getClass().isArray() ){
       int arrLen = Array.getLength(result);
       System.out.println("array length: "+arrLen);
   }
}
```

```
source code:
var arr = java.array( "java.lang.String", 3 );
arr[0] = "first";
arr[1] = "second";
arr[2] = "thrid";
println( arr );
arr
output:
[ first, second, thrid ]
result:
[Ljava.lang.String;@5749b290
array length: 3
```

Реализация интерфейса Runnable средствами L2

```
public void implementInterface(){
   L2Engine scriptEngine = new L2Engine();

   CLIFunctions cliFunctions = new CLIFunctions(scriptEngine);
   scriptEngine.getMemory().putAll(cliFunctions.getMemObjects());

   String code =
           "var obj = {"
       +   "  a : \"message\", \n"
       +   "  run : function( self ){\n"
       +   "    println( self.a )\n"
       +   "  }\n"
       +   "};\n"
       +   "java.implement( java.type( \"java.lang.Runnable\" ), obj )";

   System.out.println("source code:");
   System.out.println(code);
   System.out.println("output:");

   Object result = scriptEngine.eval(code);

   System.out.println("result:");
   System.out.println(result);

   if( result!=null && result instanceof Runnable ){
       Runnable run = (Runnable)result;

       System.out.println("run:");
       run.run();
   }
}
```

```
source code:
var obj = {  a : "message",
  run : function( self ){
    println( self.a )
  }
};
java.implement( java.type( "java.lang.Runnable" ), obj )
output:
result:
xyz.cofe.lang2.lib.gen.java_lang_Runnable@4929b0e1
run:
message
```

Реализация интерфейса возвращающего значение

```
public void implementInterface2(){
   L2Engine scriptEngine = new L2Engine();

   CLIFunctions cliFunctions = new CLIFunctions(scriptEngine);
   scriptEngine.getMemory().putAll(cliFunctions.getMemObjects());

   String interfaceName = SumItf.class.getName();

   String code =
           "var obj = {"
       +   "  summa : function( self, a, b ){\n"
       +   "    a + b\n"
       +   "  }\n"
       +   "};\n"
       +   "java.implement( java.type( \""+interfaceName+"\" ), obj )";

   System.out.println("source code:");
   System.out.println(code);
   System.out.println("output:");

   Object result = scriptEngine.eval(code);

   System.out.println("result:");
   System.out.println(result);

   if( result!=null && result instanceof SumItf ){
       SumItf sum = (SumItf)result;

       System.out.println("summa( 3, 4 ):");
       System.out.println(sum.summa(3, 4));
   }
}
```

```
source code:
var obj = {  summa : function( self, a, b ){
    a + b
  }
};
java.implement( java.type( "xyz.cofe.lang2.samples.SumItf" ), obj )
output:
result:
xyz.cofe.lang2.lib.gen.xyz_cofe_lang2_samples_SumItf@26b496d
summa( 3, 4 ):
7
```

Командная строка

Вместе с языком идут программы для работы из командной строки, в частности lang2 и lang2.bat, первая для запуска в ОС LINUX (bash скрипт), вторая в WINDOWS, для запуска необходима установленная JAVA версии 1.6 или выше. Ниже дается описание как запускать.

Синтаксис

```
Запуск         ::=  Вызов_java {опции} [скрипт_файл] ['--'] [аргументы]
Вызов_java     ::=  'lang2' | 'java' указание_ср
указание_ср    ::=  '-cp' указание_пути_к_классам главный_java_класс
главный_java_класс ::= 'xyz.cofe.lang2.cli.CLI'
указание_пути_к_классам ::= путь_к_jar { разделитель путь_к_jar }
указание_пути_к_jar ::= путь_к_jar
разделитель    ::=  ';' Для ОС Windows
                |   ':' Для остальных ОС
путь_к_jar     ::= путь к файлу/файлам jar входящих в состав дистрибутива
```

Опции

`-h` `-help`	Выводит справку по запуску
`-e` *Код* `--exp=`*Код*	Выражение которое необходимо выполнить
`-f` *Файл* `--file=`*Файл*	Файл с исходным кодом который необходимо выполнить
`-l` *Файл* `--log=`*Файл*	Файл куда будет записан лог
`--cs=`*Кодировка*	Указывает кодировку которая используется для чтения файлов со скриптами и записи лог файлов
`--charsets`	Выводит список доступных кодировок
`--endl=`*default* | *windows* | *linux* | *mac* | *other*	Указывает символы перевода строк (CR+LF[1]) используемые при выводе на консоль (STDIO) и в лог. Возможны следующие значения: windows, linux, mac, other, default
`-i` `--interactive`	Интерактивный режим работы
`--skipHello`	Отключает вывод подсказки в интерактивном режиме

1 Символы обозначающие перевод строк, обычно \n на unix системах, и \r\n на windows. см. http://ru.wikipedia.org/wiki/Перевод_строки

```
--DynamicCL=true | false
```

```
--AddDynCP=Путь
```

```
-c Файл
--config=Файл
```

```
--userInitScripts=true|false
```

```
--consoleClass=java_класс
```

Опции можно не указывать, после опций можно указать файл со скриптом что бы его запустить на исполнение.

Примеры

Интерактивный режим

```
[user@debian samples]$ lang2 -i
Для завершения работы наберите exit() и нажмите ENTER
Для справки наберите help( "help" ) и нажмите ENTER
```

```
L2> 1 + 2
```

```
3
```

```
L2> exit()
```

```
null
```

Объекты командной строки

cli

Свойство	Тип	Описание
evalCommand	Строка	Указывает строку после введения которой начнется исполнение, либо null для немедленного выполнения введенного кода.
prompt	Строка	Подсказка отображаемая при введение кода,

		обычно это «L2> »

Грамматика lang2

Введение

Грамматика языка lang2 представлена в формальном виде, используемом в средстве antlr[2].
Грамматика используется для сопоставления текущего набора символов с набором шаблонов/правил по которым пишется программа, и если есть сопоставление, то этот набор символов считается программой.

Грамматика представлена в виде именованных набора правил разбора последовательности символов.

Для пример возьмем правило expressions:

```
expressions
   :   expression
   (   ';' expression ) *
   (   ';' ) ?
   ;
```

Сначала идет название правила (expressions), потом двоеточие (:), потом описание и заканчивается точкой с запятой (;). Символы перевода и пробелы в описании правила не играют роль (за исключением описания текстовых констант).

В описании правила возможно **ссылаться на другие правила**, в примере есть две ссылки на правило expression, и на непосредственно ожидаемую последовательность символов заключенную в одинарные кавычки, в примере это точка с запятой `';'`.

Последовательность разбора начинается слева на право, так если в описании будет встречено `'while' '('` expression `')'` expression ;

1. то сначала будет проверено совпадение слова while;
2. затем открывающей круглой скобки после совпадения while;
3. затем совпадение согласно правилу expression, после предыдущего пункта;
4. затем закрывающей круглой скобки, после предыдущего пункта;
5. затем совпадение согласно правилу expression, после предыдущего пункта;

2 ANTLR — генератор парсеров, позволяющий автоматически создавать программу-парсер.

6. точка с запятой в конце описания завершает проверку, и если все пункты последовательно совпадают, то данная часть символов будет отнесена к определенному правилу.

В описании можно использовать **группировку** — это круглые скобки.

Если встречается **символ звездочка ***, то считается что последнее правило, группа или текстовая константа — то она может быть повторена 0 или более раз.

Если встречается **символ вопроса ?**, то считается что последнее правило, группа или текстовая константа — может быть отсутствовать.

Так на примере правила expressions, его можно описать словами так:

Сначала должно следовать символы удовлетворяющие правилу expression, затем через точку с запятой могут несколько раз следовать символы согласно правилу expression, и за последним expression может присуствовать символ точка с запятой.

А если встречается **вертикальная черта**, то это обозначается что возможен другой вариант. Например в описании правила block возможны несколько вариантов, пример:

```
block
   : '{' ( block | expressions ) '}'
   ;
```

Описывая словами это правило получается следующее:

1. Сначала следует символ открывающей фигурной скобки
2. Затем
    1. Либо еще раз правило block — т. е. Еще одна фигурная скобка — `'{'`
    2. Либо набор выражений соответ. expressions (т. е. Возможно там будет, block или другие
3. Заканчивается правило символом закрывающей фигурной скобки — `'}'`.

В описании лексем вам может встретиться еще два специальных символа: тильда ~ и две точки подряд. Тильда используется для обозначения, что следующие символы не должны совпадать следующему правилу, если будет совпадение, то всё правило не будет примнено. Пример правило STRING:

```
STRING
   : '"'
   ( ESC_SEQ
   | ~ ( '\\' | '"' )
   ) *
   '"'
```

```
    ;
```

здесь тильда указывает что должны за ней следовать символы кроме косой черты или двойных кавычек.

Две точки подряд используется что бы сократить перечисление символов, пример правило HEX_DIGIT:

```
HEX_DIGIT
    : '0' .. '9'
    | 'a' .. 'f'
    | 'A' .. 'F'
    ;
```

Так '0' .. '9' соответственно обозначает вот такое выражение: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.

Резюме

1. Любое правило начинается с идентификатора, затем двоеточие отделяется от описания и заканчивается точкой с запятой;

   Пример: expression : varDefine | whileCycle | forCycle | assign | block | flow ;
2. **Последовательность** (что за чем следует) определяется также как она зада в описании правила;
3. Если встречается **символ звездочка ***, то считается что последнее правило, группа или текстовая константа — то она может быть повторена 0 или более раз;
4. Если встречается **символ вопроса ?**, то считается что последнее правило, группа или текстовая константа — может быть отсутствовать;
5. **Вертикальная черта** обозначает наличие альтернативного описания правила;
6. **Круглые скобки** используются для группировки ;
7. **Одинарные кавычки** используются для определения какие символы ожидать;
8. Если между двумя последовательностями символов заданных одинарными кавычками находятся **две точки подряд** — это обозначает последовательность возможных вариантов. Пример:

   Так '0' .. '9' соответственно обозначает вот такое выражение: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.

9. Символ **тильда** — обозначает правило отрицания следующего правила, пример:

   ~ ('\\' | '"')

   Указывает что должны за ней следовать символы кроме косой черты или двойных кавычек.

10. **Ссылка на правило** задается простым написанием его идентификатора;

expressions

```
expressions
   :   expression
   (   ';' expression ) *
   (   ';' ) ?
   ;
```

Это начальное правило описывающее всю грамматику языка lang2, т. е. весь язык. Так любая программа (lang2) состоит из набора выражений разделенных точкой с запятой.

Результатом вычисления этого набора выражений является последний вычисленный результат, в данном случаи подразумеваться последнее выражение (expression), но не обязательно, возможно в процессе вычислений появится внештатная ситуация, например конструкция return, throw см. flow.

expression

```
expression
    :   varDefine
    |   whileCycle
    |   forCycle
    |   assign
    |   block
    |   flow
    ;
```

Эта конструкция описывает класс основных выражений языка как определение переменной, цикл, присваивание и т.д.

varDefine

```
varDefine
```

```
    : 'var' ID ( '=' expression ) ? ;
```

Эта конструкция определяет переменную в памяти.

whileCycle

```
whileCycle
    : 'while' '(' expression ')' expression ;
```

Эта конструкция определяет обычный цикл с условием. Цикл будет выполняться пока первое выражение возвращает значение Истина в первом выражении. Сначала вычисляется выражение в скобках, и если оно истинно, то вычисляет второе выражение, затем снова происходит повтор.

Пример

```
var a=1;
while( true ){
   a = a + 1;
   a > 5 ? break
}
a
```

```
6
```

В данном примере цикл выполняется столько раз, пока переменная a не достигает значения 6.

```
var a=1;
var b=[];
while( true ){
   a = a + 1;
   a > 20 ? break;
   a > 5 ? continue;
   b[b.length]=a
}
b
```

```
[2, 3, 4, 5]
```

В данном случаи определяется 2 переменных a и b. И выполняется цикл. Цикл выполняется пока переменная a не достигнет значения 21, после чего выполнение цикла завершается. Переменная b, объявленная как массив заполняется значениями от 2 до 5.

forCycle

```
forCycle
    : 'for' '(' ID 'in' expression ')' expression ;
```

Пример

```
var src = [ 1 , 2 , 3 ];
var sum = 0;
for( v in src ) {
   sum = sum + v;
}
sum
```

```
6
```

Пример

```
var src = {
   idx : 0,
   hasNext : function() {
       this.idx < 5
   },
   next : function(){
       this.idx = this.idx + 1;
       this.idx
   }
};
var sum = 0;
for( v in src ) {
   sum = sum + v;
}
sum
```

```
15
```

assign

```
assign
   : or
   ( '=' expression
   | '?' expression ( ':' expression ) ?
   ) ?
   ;
```

block

```
block
   : '{' ( block | expressions ) '}'
   ;
```

Пример

```
{
   1+2;
   2+2
}
```

```
4
```

```
{
   var a=1;
   {
       var a=2
   }
a
}
```

```
1
```

flow

```
flow
   :   'if' '(' expression ')' expression ( 'else' expression ) ?
   |   'return' ( expression ) ?
   |   'break' ( expression ) ?
   |   'continue' ( expression ) ?
   |   'throw' expression
   |   'try' expression 'catch' '(' ID ')' expression
   ;
```

Приостановка или изменение последовательности исполнения программы:

- return — завершение выполнения функции.
- break — завершение выполнения цикла.
- continue — завершение выполнения итерации цикла, и переход к следующей итерации.

- throw — генерация исключения.
- try .. catch — «отлов» исключения.

Пример

```
var a=1;
while( true ){
  a = a + 1;
  a > 5 ? break
}
a
```

```
6
```

```
var a=1;
var b=[];
while( true ){
  a = a + 1;
  a > 20 ? break;
  a > 5 ? continue;
  b[b.length]=a
}
b
```

```
[2, 3, 4, 5]
```

```
var f1 = function( a ){
  a > 5 ? return a + a;
  a < 0 ? return a - a;
  a * a
}
f1( 6 ) + f1( -2 ) + f1( 1 )
```

```
13
```

```
try {
  throw "abc";
}
catch( e ) {
  "ok:"+e;
}
```

```
ok:abc
```

or

```
or
   : xor ( ( '|' | 'or' ) xor ) *
   ;
```

Логическая операция ИЛИ.

Примеры логической операции ИЛИ

```
true or true
```

```
true
```

```
true or false
```

```
true
```

xor

```
xor
   : and ( ( '^' | 'xor' ) and ) *
   ;
```

Логическая операция НЕ ИЛИ.

Примеры логической операции НЕ ИЛИ

```
true xor true
```

```
false
```

```
true xor false
```

```
true
```

```
false xor true
```

```
true
```

```
false xor false
```

```
false
```

and

```
and
   : compare ( ( '&' | 'and' ) compare ) *
   ;
```

Логическая операция И.

Примеры логической операции И

```
true and true
```

```
true
```

```
true and false
```

```
false
```

```
true and not false
```

```
true
```

compare

```
compare
   : addition
   ( ( '<>' | '!=' ) addition
   | '<=' addition
   | '>=' addition
   | '==' addition
   | '<' addition
   | '>' addition
   ) *
   ;
```

Операция сравнения.

Примеры сравнения

```
1==1
```

```
true
```

```
2<>1
```

```
true
```

```
2>1
```

```
true
```

```
2<1
```

```
false
```

```
2>=2
```

```
true
```

```
2>=1
```

```
true
```

```
1<=2
```

```
true
```

```
1<=1
```

```
true
```

```
"abc" < "def"
```

```
true
```

```
"abc" == "def"
```

```
false
```

addition

```
addition
   : ( '-' ) ?
         multiple
   ( '+' multiple
   | '-' multiple
   ) *
   ;
```

Сложение.

Примеры

```
(2+2-3)*4/2
```

```
2.0
```

```
"abc" + "def"
```

```
abcdef
```

```
-1+1
```

```
0
```

multiple

```
multiple
   : postfix ( '*' postfix | '/' postfix | '%' postfix ) *
   ;
```

Умножение.

Примеры

```
(2+2-3)*4/2
```

```
2.0
```

postfix

```
postfix
   : value
   ( '.' ID
   | '[' expression ']'
   | '(' ( expression ( ',' expression ) * ) ? ')'
```

```
   ) *
   ;
```

.ID - Доступ к элементу массива/объекта.

[expression] - Доступ к элементу массива/символу в строке.

(expression (, expression)*) - Вызов функции/метода.

Пример

```
var lst = [ "str", "ing" ];
lst[0] + lst[1]
```

```
string
```

```
var obj = {
   a : 1+2,
   b : "aa",
   c : true,
   d : function ( self, a, b ) { a + b }
};
obj.a
```

```
3.0
```

```
var obj = {
   a : 1+2,
   b : "aa",
   c : true,
   d : function ( self, a, b ) { a + b }
};
obj.d( 2, 3 )
```

```
5.0
```

value

```
value
   : constPrimitive
   | ID
   | NOT expression
   | '(' expression ')'
```

```
   | functionDefine
   | objectDefine
   | arrayDefine
   ;
```

Указывает на значение, как то число, объект, функцию ...

constPrimitive

```
constPrimitive
   : 'true'
   | 'false'
   | 'null'
   | FLOAT
   | INT
   | STRING
   ;
```

Объявление константы.

Булево

```
true
false
```

Нулевая ссылка

```
null
```

Плавующее число

```
12.34
```

Целое число

```
56
```

Строка

```
"Последовательность символов"
"Перевод\nстроки"
```

Возможны следующие экранированные символы:

```
\n
\r
\t
\b
\f
\"
\'
\\
```

А также использование UNICODE значений:

```
"\u0411\u0443\u043A\u0432\u0430"
```

```
Буква
```

И использование 8-ричных значений:

```
"\0117\0143\0164\0141\0154"
```

```
Octal
```

functionDefine

```
functionDefine
   :
   FUNCTION
   '(' ( ID ( ',' ID ) * ) ? ')'
   expression
   ;
```

Объявление функции.

Пример

```
var f1 = function( a, b ) a+b; f1( 1,2 )
```

```
3.0
```

```
var f2 = function( a ) a > 2 ? recursion( a-1 ) * a : a; f2( 4 )
```

```
24.0
```

arrayDefine

```
arrayDefine
   :
   '['
   ( expression ( ',' expression ) * ) ?
   ']'
   ;
```

Объявление массива.

Пример

```
var lst = [ "str", "ing" ];
lst[0] + lst[1]
```

```
string
```

```
var lst = [ 1, 2 ];
lst.length
```

```
2
```

objectDefine

```
objectDefine
   :
   '{'
   ID ':' expression ( ',' ID ':' expression ) *
   '}'
   ;
```

Объявление объекта.

Пример

```
var obj = {
   a : 1+2,
   b : "aa",
   c : true,
   d : function ( self, a, b ) { a + b }
};
obj.a
```

```
3
```

```
var obj = {
   a : 1+2,
   b : "aa",
   c : true,
   d : function ( self, a, b ) { a + b }
};
obj.b
```

```
aa
```

```
var obj = {
   a : 1+2,
   b : "aa",
   c : true,
   d : function ( self, a, b ) { a + b }
};
obj.c
```

```
true
```

```
var obj = {
   a : 1+2,
   b : "aa",
   c : true,
   d : function ( self, a, b ) { a + b }
};
obj.d( 2, 3 )
```

```
5
```

```
var obj = {
   a : "str",
   b : function ( self, c ) {
       self.a + c
   }
};
obj.b( "ing" )
```

```
string
```

FUNCTION

```
FUNCTION : 'function' ;
```

NOT

```
NOT : '!' | 'not' ;
```

ID

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

Идентификатор, используется в качестве имен переменных и тд....

INT

```
INT : ( '0' .. '9' ) + ;
```

Целое число.

FLOAT

```
FLOAT
   :
```

```
   ( '0' .. '9' ) +
   '.' ( '0' .. '9' ) *
   ;
```

Рациональное число

COMMENT

```
COMMENT
   :
   '//' ~ ( '\n' | '\r' ) * '\r' ? '\n'
   | '/*' ( . ) * '*/'
   ;
```

// - Однострочный комментарий.

/* ... */ - Многострочный комментарий.

WS

```
WS
   : ' '
   | '\t'
   | '\r'
   | '\n'
   ;
```

STRING

```
STRING
   : '"'
   ( ESC_SEQ
   | ~ ( '\\' | '"' )
   ) *
   '"'
   ;
```

Строковая константа

CHAR

```
CHAR
   : '\''
```

```
   ( ESC_SEQ
   | ~ ( '\'' | '\\' )
   )
   '\''
   ;
```

EXPONENT

```
EXPONENT
   : ( 'e' | 'E' ) ( '+' | '-' ) ? ( '0' .. '9' ) +
   ;
```

HEX_DIGIT

```
HEX_DIGIT
   : '0' .. '9'
   | 'a' .. 'f'
   | 'A' .. 'F'
   ;
```

ESC_SEQ

```
ESC_SEQ
   : '\\' ( 'b'    | 't' | 'n' | 'f' | 'r' | '\"' | '\'' | '\\' )
   | UNICODE_ESC
   | OCTAL_ESC
   ;
```

OCTAL_ESC

```
OCTAL_ESC
   : '\\' ( '0' .. '3' ) ( '0' .. '7' ) ( '0' .. '7' )
   | '\\' ( '0' .. '7' ) ( '0' .. '7' )
   | '\\' ( '0' .. '7' )
   ;
```

UNICODE_ESC

```
UNICODE_ESC
```

```
    : '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
    ;
```