Lab 1. 表达式

目录

1. 概述

在本次实验中，你将使用仓颉语言实现 njucj 语言的解释器的表达式计算器，支持解释和执行变量定义、赋值、基本运算、条件表达式和循环表达式等。

你将需要处理运行时类型检查、变量作用域与生存期管理、循环跳转等内容。

njucj 语言是仓颉语言的一个子集，未来将会包含函数、闭包、类等功能。

将代码文本解析为抽象语法树（AST）的工作已经由实验框架完成，你只需要实现表达式的动态语义。

2. 要求

2.1. 支持的类型

为了简化，njucj 语言只要求支持四种类型：

- **Int64**: 64 位整数，
- **String**: 任意长度字符串
- **Bool**: 布尔值
- **Unit**: 单位值

2.2. 支持的表达式

以下是本次实验要求支持的表达式子集：

(程序) $P ::= \textbf{main}()\ blk$

(语句块) $blk ::= \{el\}$

(表达式列表) $el ::= vd \mid e \mid vd\ \text{end}\ el \mid e\ \text{end}\ el$

(变量声明) $vd ::= \textbf{var}\ x : t = e \mid \textbf{var}\ x = e \mid \textbf{let}\ x : t = e \mid \textbf{let}\ x = e$

(分隔符) $end \in \text{NL} \mid ;$

(变量) $x, y, z \in ...$

(整数字面量) $n \in ...$

(字符串字面量) $s \in ...$

(表达式) $e ::= n \mid z \mid s \mid \textbf{true} \mid \textbf{false} \mid () \mid x \mid -e \mid x = e$
$\mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1\ /\ e_2 \mid e_1 \% e_2 \mid e_1\ **\ e_2$
$\mid e_1 < e_2 \mid e_1 <= e_2 \mid e_1 > e_2 \mid e_1 >= e_2 \mid e_1 == e_2 \mid e_1\ !=\ e_2$
$\mid e_1 \mid\mid e_2 \mid e_1\ \&\&\ e_2 \mid !e$
$\mid \textbf{if}\ (e)\ blk \mid \textbf{if}\ (e)\ blk\ \textbf{else}\ blk \mid \textbf{while}\ (e)\ blk \mid \textbf{break} \mid \textbf{continue}$

(类型) $t ::= \textbf{Bool} \mid \textbf{Int64} \mid \textbf{String} \mid \textbf{Unit}$

严格的 BNF 文法参见 小节 7.3

2.3. 输出

实验框架已经为你准备了一部分的代码。你需要解释和执行 `main` 中的代码块，并将最后一个表达式的结果打印在**标准输出** (stdout) 上。

3. 静态语义

njucj 语言尽可能与仓颉语言保持一致，但为了简化实验要求，存在一些修改。

3.1. 类型

在本次实验中, 我们不要求处理静态类型检查和类型推导。只有在运行时遇到的类型错误才应当报错。

这意味着，在本次实验中，一部分将无法通过静态类型检查的程序是合法的。例如，以下代码：

```
true || "string"
```

被认为是合法的的语句，因为 `||` 运算符具有短路特性，你不应该计算 `"string"` 的值，因此在本次试验中无法得知此处的 `||` 运算符右操作数的类型。

因此上述表达式的值为 `true`，类型为 **Bool**。

3.2. 作用域

所有的变量定义均使用**静态作用域** (*static scope*, or *lexical scope*)。变量的作用域从其声明或定义处开始，到其所在的静态作用域结束。

在 njucj 中不允许同一静态作用域内定义同名变量。

使用变量时，递归地从当前静态作用域开始，向上寻找该变量的定义。

与仓颉一样，`while` 循环体会创建一个新的静态作用域。 `if` 和 `else` 分支体会创建一个新的静态作用域。

4. 动态语义

4.1. 错误

在表达式计算过程中，有时会遇到类型错误或语义错误。此时应报错并终止程序执行，以 exit code 1 退出。

在出现报错时，你的程序应该在**标准错误输出** (stderr) 的最后输出一行错误信息，格式如下：

```
Error at line X: [ERROR_CODE]: your custom error message
```

其中，`X` 是出现错误时的行号，`ERROR_CODE` 是错误代码 (我们将会为每一处可能的报错定义错误代码，见 小节 7.1)。 `your custom error message` 是你自定义的错误信息，可以简短描述错误原因，方便调试。

我们会使用正则表达式

```
^Error at line (\d+): \[(\w+)\]:.*$
```

来匹配你的错误信息，并根据行号和错误代码来判断你的程序是否正确地报错。

例如，以下程序：

```
main() {
    let x = 1
    let y = "hello"
    x + y
    x + x
    y + y
}
```

一个正确的报错是：

```
Error at line 4: [ADD_TYPE_MISMATCH]: cannot add Int64(1) and String("hello")
```

你可以假设，不会出现有错误的表达式被拆行，因此出错的行号可以从出错的表达式中的任意 Token 计算而来。

对于 `break` 和 `continue`，出错的行号即 `break` 和 `continue` 所在的行号。

4.2. 表达式块

对于表达式块

```
expr1
expr2
...
exprN
```

你需要依次计算 `expr1`, `expr2`, ..., `exprN` 的值，然后抛弃前面其他所有表达式的值，只返回 `exprN` 的值作为整个表达式块的值。

4.3. 二元运算符

在接下来的说明中，我们用 `operator func op(a: T1, b: T2): T3` 表示在 `a` 类型为 `T1`，`b` 类型为 `T2` 时，表达式 `a op b` 的值为 `T3` 的情形。

大部分的二元运算符和仓颉语言保持语义上的一致，但是少数运算符略有不同，请仔细阅读。

4.3.1. 加法

`operator func +(a: Int64, b: Int64): Int64` 被实现为两数的和。溢出报错 `ADD_OVERFLOW`。

`operator func +(a: String, b: String): String` 被实现为两个字符串的拼接。

其余类型报错 `ADD_TYPE_MISMATCH`。

4.3.2. 减法

`operator func -(a: Int64, b: Int64): Int64` 被实现为两数的差。溢出报错 `SUB_OVERFLOW`。

其余类型报错 `SUB_TYPE_MISMATCH`。

4.3.3. 乘法

`operator func *(a: Int64, b: Int64): Int64` 被实现为两数的乘积。溢出报错 `MUL_OVERFLOW`.

`operator func *(a: String, b: Int64): String` 和 `operator func *(b: Int64, a: String): String` 将字符串 `a` 重复 `b` 次。特别的，当 `b` 等于 `0` 时，结果为空字符串 `""`

其余类型报错 `MUL_TYPE_MISMATCH`.

4.3.4. 除法

`operator func /(a: Int64, b: Int64): Int64` 被实现为两数整除的商。当 `b` 等于 `0` 时报错 `DIV_BY_ZERO`。

其余类型报错 `DIV_TYPE_MISMATCH`。

4.3.5. 取余

`operator func %(a: Int64, b: Int64): Int64` 被实现为求 `a / b` 的余数。当 `b` 等于 `0` 时报错 `MOD_BY_ZERO`。

其余类型报错 `MOD_TYPE_MISMATCH`。

4.3.6. 乘方

`operator func **(a: Int64, b: Int64): Int64` 被实现为 `a` 的 `b` 次幂。当 `b` 小于 `0` 时报错 `EXP_NEGATIVE_POWER`。溢出时报错 `EXP_OVERFLOW`。

其余类型报错 `EXP_TYPE_MISMATCH`。

4.3.7. 大于，小于，大于等于，小于等于

`operator func <(a: Int64, b: Int64): Bool` 被实现为小于比较运算。

`operator func <(a: String, b: String): Bool` 被实现为字典序比较运算。当 `a` 字典序小于 `b` 时结果为 `true`，否则结果为 `false`。

其余类型报错 `CMP_TYPE_MISMATCH`。

`<=`, `>`, `>=` 同理。报错代码同上。

4.3.8. 相等与不等

`operator func ==(a: Int64, b: Int64): Bool` 求两整数是否相等。`a` 和 `b` 的值相等时结果为 `true`，否则结果为 `false`。

`operator func ==(a: String, b: String): Bool` 求两字符串是否相等。`a` 和 `b` 的值相等时结果为 `true`，否则结果为 `false`。

`operator func ==(a: Bool, b: Bool): Bool` 求两布尔值是否相等。`a` 和 `b` 的值相等时结果为 `true`，否则结果为 `false`。

`operator func ==(a: Unit, b: Unit): Bool` 总是返回 `true`。

其他类型报错 `EQ_TYPE_MISMATCH`。

`!=` 运算符是 `==` 的反面。类型不匹配时，报错 `NEQ_TYPE_MISMATCH`。

4.3.9. 逻辑与

`operator func &&(a: Bool, b: Bool): Bool` 被实现为逻辑与运算。`a` 和 `b` 均为 `true` 时结果为 `true`，否则结果为 `false`。

对于逻辑与运算 `a && b`，若 `a` 的值为 `false`，则不计算 `b`，直接返回 `false` 作为整个表达式的值。

如果在计算后，逻辑与运算 `a && b` 中的 `a` 或 `b` 不是布尔类型，报错 `AND_TYPE_MISMATCH`。

4.3.10. 逻辑或

`operator func ||(a: Bool, b: Bool): Bool` 被实现为逻辑或运算。`a` 和 `b` 均为 `false` 时结果为 `false`，否则结果为 `true`。

对于逻辑或运算 `a || b`，若 `a` 的值为 `true`，则不计算 `b`，直接返回 `true` 作为整个表达式的值。

如果在计算后，逻辑或运算 `a || b` 中的 `a` 或 `b` 不是布尔类型，报错 `OR_TYPE_MISMATCH`。

4.4. 一元运算符

4.4.1. 取负

`operator func -(a: Int64): Int64` 的值等同于 `-a`。

类型不为 Int64，报错 `NEG_TYPE_MISMATCH`

发生溢出，报错 `NEG_OVERFLOW`

4.4.2. 取非

`operator func !(a: Bool): Bool` 被实现为逻辑非运算。`a` 为假值时结果为 `true`，否则结果为 `false`。

类型不为 Bool，报错 `NOT_TYPE_MISMATCH`

4.5. 条件表达式

4.5.1. if 表达式

`if (cond) { thenBranch }` 计算 `cond` 的值：

- 若 `cond` 的值为 `true`，则计算 `thenBranch`，抛弃其值，返回 `()` 作为整个表达式的值。
- 若 `cond` 的值为 `false`，则返回 `()` 作为整个表达式的值。

`if (cond) { thenBranch } else { elseBranch }` 计算 `cond` 的值：

- 若 `cond` 的值为 `true`，则计算 `thenBranch`，返回其值作为整个表达式的值。

- 若 `cond` 的值为 `false`，则计算 `elseBranch`，返回其值作为整个表达式的值。

对上述两种 `if`, 若 `cond` 的值不是布尔类型，报错 `IF_TYPE_MISMATCH`

4.6. 循环表达式

4.6.1. while 表达式

`while (cond) { body }` 重复执行以下步骤

1. 计算 `cond` 的值：
   - 若 `cond` 的值为 `false`，则结束循环，返回 `()` 作为整个表达式的值。
   - 若 `cond` 的值为 `true`，则继续执行下一步。
   - 若 `cond` 的值不是布尔类型，报错 `WHILE_TYPE_MISMATCH`
2. 计算 `body` 的值，抛弃其值。

循环结束后，返回 `()` 作为整个表达式的值。

4.6.1.1. break

在 `while` 循环体内，执行 `break` 会立即结束循环，跳转到循环体外继续执行后续代码。

`break` 语句总是匹配其**静态作用域**内最近的 `while` 循环。如果存在嵌套循环，`break` 只会结束内层循环。

如果 `break` 所属的静态作用域外层没有 `while` 表达式，报错 `BREAK_OUTSIDE_LOOP`

4.6.1.2. continue

在 `while` 循环体内，执行 `continue` 会立即结束本次循环迭代，跳转到下一次循环迭代的开始处。

`continue` 语句总是匹配其**静态作用域**内最近的 `while` 循环。如果存在嵌套循环，`continue` 只会影响内层循环。

如果 `continue` 所属的静态作用域外层没有 `while` 表达式，报错 `CONTINUE_OUTSIDE_LOOP`

4.7. 赋值

`a = b` 计算 `b` 的值，根据**静态作用域**递归地向上查找内变量 `a` 的定义：

- 当 `a` 是 `var` 定义的，检查 `a` 的旧值的类型。如果类型相同，将该值赋给变量 `a`。表达式 `a = b` 的值为 `()`。否则，报错 `ASSING_TYPE_MISMATCH`
- 当 `a` 是 `let` 定义，且 `a` 不是一个未初始化的变量，报错 `ASSGIN_IMMUT_VAR`。
- 当 `a` 未定义，报错 `UNDEFINED_VAR`。

4.8. 变量定义

变量定义在文法上不属于表达式，但我们要求一个合法的变量定义语句具有值 `()`。

`let a = b` 计算 `b` 的值，然后在当前**静态作用域**内定义一个名为 `a` 的不可变变量，其值为 `b` 的值。

`var a = b` 计算 `b` 的值，然后在当前**静态作用域**内定义一个名为 `a` 的可变变量，其值为 `b` 的值。

在变量定义语句中，变量名不能与**当前**静态作用域内已有的变量名冲突，否则报错 `DUPLICATED_DEF`。

带有类型的变量定义稍复杂些：

`let a: T1 = b` 与 `let a = b` 有一样的效果，除了它会检查 `b` 的值的类型是否为 `T1`. 如果不是，报错 `DEF_TYPE_MISMATCH`. `var` 同理。

`var a: T1` 将在当前静态作用域内定义一个名为 `a`，类型为 `T1`，但尚未初始化的可变变量。

`let a: T1` 将在当前静态作用域内定义一个名为 `a`，类型为 `T1`，但尚未初始化的可变变量。**此后，`a` 被允许赋值一次，且仅允许赋值一次**

4.9. 变量

在表达式中使用变量名 `a` 会递归地，从当前**静态作用域**开始，向上寻找名为 `a` 的变量定义，并返回该变量的值。

变量定义和赋值表达式的等号左边除外。

如果在所有可见的静态作用域内均未找到名为 `a` 的变量定义，报错 `UNDEFINED_VAR`。

如果找到 `a` 的变量定义，但它还未初始化（如，`var a: Int64` 定义的变量），试图读取它的值将会报错 `UNINITIALIZED_VAR`。

```
// 正确的例子：
var a: Int64
a = 1 // 没有尝试读取 a 的值
```

```
// 错误的例子：
var a: Int64
a + 1 // UNINITIALIZED_VAR: 试图读取 a 的值
```

5. 输入输出

5.1. 样例

1. 样例 1. 斐波那契数列

   输入:

   ```
   main() {
       var a = 1
       var b = 1
       var i = 0
       while (i < 10) {
           let c = a
           a = a + b
           b = c
           i = i + 1
       }
       b
   }
   ```

   标准输出:

   ```
   89
   ```

   退出代码为 0

2. 样例 2. 错误的类型

   ```
   main() {
       var a = 1
       var b = 1
       var i = 0
       while (i - 10) {
           let c = a
           a = a + b
           b = c
           i = i + 1
       }
       b
   }
   ```

   标准错误输出:

   ```
   Error at line 5: [WHILE_TYPE_MISMATCH]: while condition must be Bool, but got Int64(-10)
   ```

   退出代码为 1

5.2. 测试用例保证

保证输入文件有且仅有一个函数定义，即 `main` 的定义。

保证 `main` 函数无参数。

`main` 函数内含有一个或多个表达式或变量定义语句。

保证不包含任何具有语法错误的内容。

6. 提交

提交方式 在智慧南雍平台（https://lms.nju.edu.cn）提交实验代码和实验报告
截止日期 2025 年 11 月 12 日, 23:59

6.1. 提交格式

将你的代码文件打包成 `lab1_[YOUR_ID]_[YOUR_NAME].zip`，例如，`lab1_123456789_张三.zip`，文件树如下

```
lab1_[YOUR_ID]_[YOUR_NAME].zip/
├─ report.pdf  # 实验报告
├─ cjpm.toml   # 仓颉项目文件
├─ cjpm.lock   # 仓颉项目文件
├─ src/        # 源代码文件夹
   ├─ main.cj
   ├─ ....
   ├─ 你的代码文件
```

请不要将 `.git`, `target` 等文件夹打包在内

6.2. 实验报告要求

实验报告命名为 `report.pdf` 并放置在打包的文件夹的根目录。该文件应该是 3~5 页的 PDF。

报告内容应包含以下内容:

1. **代码结构与设计**: 简要描述你采用的代码结构、设计思路。你无需描述实验框架已经给出的内容。你如何优雅地处理表达式？如何管理变量作用域？如何封装和抽象，从而保持代码整洁，在实验结束前仍然可以被你阅读？你的代码有没有什么亮点？你可以简要描述你认为比较重要或有趣的设计细节。
2. **遇到的问题与解决方案**: 简要描述你在实验过程中遇到的主要问题, 以及你是如何解决这些问题的。如果没有遇到问题，可以简要描述你认为的实验难度，和可以改进的地方。
3. **已知 Bug（可选）**: 如果你的代码中存在你已知但未解决的 Bug，可以简要描述。

6.3. 代码提示

- 创建一个本地的 Git 仓库来跟踪你的更改是一个不错的选择。你也可以将你的代码上传到 GitHub 等平台的私有仓库。保持提交记录的整洁有利于在代码变得复杂时跟踪你的每一行代码是何时编写、为何编写。
- 适当的编写注释有利于你在一段时间后仍能快速理解你编写的代码。对于复杂的代码, 注释也有助于理清自己的思路。
- 在提交代码前，你可以执行 `cjfmt -d .` 来格式化代码。好的代码风格能方便你和助教的阅读。使用合理的、适当的变量名，避免过长的函数和代码块。

6.4. 学术诚信 Academic Integrity

学术诚信是所有从事学术活动的学生和学者最基本的职业道德底线，本课程将不遗余力的维护学术诚信规范，违反这一底线的行为将不会被容忍。

作业完成的原则: 署你名字的工作必须是你个人的贡献。在完成作业的过程中, 允许讨论, 前提是讨论的所有参与者均处于同等完成度。但关键想法的执行、以及作业文本的写作必须独立完成,

并在报告中致谢（acknowledge）所有参与讨论的人。不允许其他任何形式的合作——尤其是与已经完成作业的同学“讨论”。

本课程将对剽窃行为采取零容忍的态度。在完成作业过程中，对他人工作（出版物、互联网资料、其他人的作业等）直接的文本抄袭和对关键思想、关键元素的抄袭，按照 ACM Policy on Plagiarism 的解释，都将视为剽窃。剽窃者成绩将被取消。如果发现互相抄袭行为， 抄袭和被抄袭双方的成绩都将被取消。因此请主动防止自己的作业被他人抄袭。

学术诚信影响学生个人的品行，也关乎整个教育系统的正常运转。为了一点分数而做出学术不端的行为，不仅使自己沦为一个欺骗者，也使他人的诚实努力失去意义。让我们一起努力维护一个诚信的环境。

7. 附录

7.1. 错误代码

这里列出这次实验中，所有可能的错误代码和简要解释。

`ADD_TYPE_MISMATCH` 加法两侧类型不受支持。

`ADD_OVERFLOW` 整数加法结果超出 `Int64` 表示范围。

`SUB_TYPE_MISMATCH` 减法两侧类型不受支持。

`SUB_OVERFLOW` 整数减法结果超出 `Int64` 表示范围。

`MUL_TYPE_MISMATCH` 乘法两侧类型不受支持。

`MUL_OVERFLOW` 整数乘法结果超出 `Int64` 表示范围。

`DIV_TYPE_MISMATCH` 除法两侧类型不受支持。

`DIV_BY_ZERO` 整数除法的除数为 `0`。

`MOD_TYPE_MISMATCH` 取余两侧类型不受支持。

`MOD_BY_ZERO` 取余运算的除数为 `0`。

`EXP_NEGATIVE_POWER` 指数为负数的幂运算不被支持。

`EXP_OVERFLOW` 幂运算结果超出 `Int64` 表示范围。

`EXP_TYPE_MISMATCH` 幂运算两侧类型不受支持。

`CMP_TYPE_MISMATCH` 比较运算两侧的类型不受支持。

`EQ_TYPE_MISMATCH` 比较两个不一样的类型。

`NEQ_TYPE_MISMATCH` 比较两个不一样的类型。

`AND_TYPE_MISMATCH` 在计算中遇到逻辑与的任一操作数不是 `Bool`。

`OR_TYPE_MISMATCH` 在计算中遇到逻辑或的任一操作数不是 `Bool`。

`NOT_TYPE_MISMATCH` 逻辑非的操作数不是 `Bool`。

`NEG_TYPE_MISMATCH` 一元取负的操作数不是 `Int64`。

`NEG_OVERFLOW` 一元取负运算发生溢出。

`IF_TYPE_MISMATCH` if 条件表达式的结果不是 `Bool`。

`WHILE_TYPE_MISMATCH` while 条件表达式的结果不是 `Bool`。

`BREAK_OUTSIDE_LOOP` 在非循环体内使用 `break`。

`CONTINUE_OUTSIDE_LOOP` 在非循环体内使用 `continue`.

`ASSGIN_IMMUT_VAR` 试图给 `let` 定义的不可变变量赋值。

`ASSING_TYPE_MISMATCH` 试图给变量赋值成不同的类型

`UNDEFINED_VAR` 试图使用未定义的变量。

`DUPLICATED_DEF` 试图在同一静态作用域内定义同名变量。

`DEF_TYPE_MISMATCH` 变量定义时，赋值的类型与声明的类型不匹配。

`UNINITIALIZED_VAR` 试图读取未初始化的变量的值。

7.2. Tokens

见 仓颉语言文档 - `TokenKind`

7.3. Grammar

对于 parser 能处理的所有文法，详见实验框架中给出的 `currentGrammar.md` 文件

具体而言，本次实验可能的输入，将会是满足以下文法的程序：

```
## VARIABLE DEFINITION

variableDeclaration
    : (LET | VAR | CONST) NL* patternsMaybeIrrefutable
        ( (NL* COLON NL* type)? (NL* ASSIGN NL* expression) | (NL* COLON NL* type) )
    ;

patternsMaybeIrrefutable
    : varBindingPattern
    ;

varBindingPattern
    : identifier
    ;

## EXPRESSION

expression
    : assignmentExpression
    ;

assignmentExpression
    : leftValueExpression NL* ASSIGN NL* logicDisjunctionExpression
    | logicDisjunctionExpression
    ;

leftValueExpression
    : leftValueExpressionWithoutWildCard
    ;

leftValueExpressionWithoutWildCard
    : identifier

logicDisjunctionExpression
    : logicConjunctionExpression (NL* OR NL* logicConjunctionExpression)*
    ;

logicConjunctionExpression
    : bitwiseDisjunctionExpression (NL* AND NL* bitwiseDisjunctionExpression)*
    ;

bitwiseDisjunctionExpression
    : bitwiseConjunctionExpression (NL* BITOR NL* bitwiseConjunctionExpression)*
    ;
```

```
bitwiseConjunctionExpression
    : equalityComparisonExpression (NL* BITAND NL* equalityComparisonExpression)*
    ;

equalityComparisonExpression
    : comparisonOrTypeExpression (NL* equalityOperator NL* comparisonOrTypeExpression)?
    ;

comparisonOrTypeExpression
    : additiveExpression (NL* comparisonOperator NL* additiveExpression)?
    ;

additiveExpression
    : multiplicativeExpression (NL* additiveOperator NL* multiplicativeExpression)*
    ;

multiplicativeExpression
    : exponentExpression (NL* multiplicativeOperator NL* exponentExpression)*
    ;

exponentExpression
    : prefixUnaryExpression (NL* exponentOperator NL* prefixUnaryExpression)*
    ;

prefixUnaryExpression
    : prefixUnaryOperator* postfixExpression
    ;

postfixExpression
    : atomicExpression
    | postfixExpression callSuffix
    | postfixExpression NL* DOT NL* identifier
    ;

callSuffix
    : LPAREN NL* (valueArgument (NL* COMMA NL* valueArgument)* NL*)? RPAREN
    ;

valueArgument
    : expression
    ;

atomicExpression
    : literalConstant
    | identifier
    | ifExpression
    | loopExpression
    | jumpExpression
    | parenthesizedExpression
    ;

literalConstant
    : IntegerLiteral
    | booleanLiteral
    | stringLiteral
    | unitLiteral
    ;

booleanLiteral
```

```
    : TRUE
    | FALSE
    ;

stringLiteral
    : lineStringLiteral
    ;

lineStringContent
    : LineStrText
    ;

lineStringLiteral
    : QUOTE_OPEN (lineStringContent)* QUOTE_CLOSE
    ;

unitLiteral
    : LPAREN NL* RPAREN
    ;

ifExpression
    : IF NL* LPAREN NL* expression NL* RPAREN NL* block
    (NL* ELSE (NL* ifExpression | NL* block))?
    ;

loopExpression
    : whileExpression
    ;


whileExpression
    : WHILE NL* LPAREN NL* expression NL* RPAREN NL* block
    ;

jumpExpression
    : RETURN (NL* expression)?
    | CONTINUE
    | BREAK
    ;

parenthesizedExpression
    : LPAREN NL* expression NL* RPAREN
    ;

block
    : LCURL expressionOrDeclarations RCURL
    ;

expressionOrDeclarations
    : end* (expressionOrDeclaration (end+ expressionOrDeclaration?)*)?
    ;

expressionOrDeclaration
    : expression
    | varOrfuncDeclaration
    ;

varOrfuncDeclaration
    | variableDeclaration
    ;
```

```
assignmentOperator
    : ASSIGN
    ;

equalityOperator
    : NOTEQUAL
    | EQUAL
    ;

comparisonOperator
    : LT
    | GT
    | LE
    | GE
    ;

additiveOperator
    : ADD | SUB
    ;

exponentOperator
    : EXP
    ;

multiplicativeOperator
    : MUL
    | DIV
    | MOD
    ;

prefixUnaryOperator
    : SUB
    | NOT
    ;
```