Lab 2 — 函数和闭包

目录

1. 引言

在本次实验中，你需要对之前实现的 njucj 语言（简称 njucj ）解释器进行扩展，在表达式计算的基础上，支持函数定义、函数调用、嵌套函数定义（带有闭包）、高阶函数等功能。

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，你只需要实现语言的动态语义。

注意，请不要在你的代码中导入或使用 `std.database`、`std.net`、`std.posix`、`std.process`、`std.runtime` 包中的任何功能，否则你的代码将无法通过测试。

请记得查看附录中的更新说明部分，以了解本次实验框架相对于实验 1 的变更内容，以及如何升级实验框架。

2. 要求

2.1. 支持的语言子集语法

以下是本次实验要求支持的语言子集。表达式的优先级和运算符的结合性与仓颉语言一致。

(程序) $P \coloneqq tld^*\ md\ tld^*$

(程序入口声明) $md \coloneqq \texttt{main()}\ blk$

(顶层声明) $tld \coloneqq vd \mid fd$

(表达式块) $blk \coloneqq \texttt{\{}\ expr_1\ end\ expr_2\ end\ \ldots\ end\ expr_n\ \texttt{\}}$

(声明或表达式) $expr \coloneqq vd \mid fd \mid e$

(变量声明) $vd \coloneqq \texttt{var}\ x\ [\texttt{:}\ t]\ \texttt{=}\ e \mid \texttt{let}\ x\ [\texttt{:}\ t]\ \texttt{=}\ e$

(函数声明) $fd \coloneqq \texttt{func}\ x\ \texttt{(}\ pd_1\ \texttt{,}\ pd_2\ \texttt{,}\ \ldots\ \texttt{,}\ pd_n\ \texttt{)}\ blk$

(函数形参定义) $pd \coloneqq x\ \texttt{:}\ t$

(分隔符) $end \in \text{NL} \mid \texttt{;}$

(变量) $f, g, x, y \in \ldots$

(整数字面量) $n \in \ldots$

(字符串字面量) $s \in \ldots$

(表达式) $e \coloneqq n \mid s \mid \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid x \mid \texttt{(}\ e\ \texttt{)} \mid x\ \texttt{=}\ e \mid \texttt{-}\ e$
$\mid e_1\ \texttt{+}\ e_2 \mid e_1\ \texttt{-}\ e_2 \mid e_1\ \texttt{*}\ e_2 \mid e_1\ \texttt{/}\ e_2 \mid e_1\ \texttt{\%}\ e_2 \mid e_1\ \texttt{**}\ e_2$
$\mid e_1\ \texttt{<}\ e_2 \mid e_1\ \texttt{<=}\ e_2 \mid e_1\ \texttt{>}\ e_2 \mid e_1\ \texttt{>=}\ e_2 \mid e_1\ \texttt{==}\ e_2 \mid e_1\ \texttt{!=}\ e_2$
$\mid e_1\ \texttt{||}\ e_2 \mid e_1\ \texttt{\&\&}\ e_2 \mid \texttt{!}\ e \mid \texttt{if}\ \texttt{(}\ e\ \texttt{)}\ blk\ [\texttt{else}\ blk]$
$\mid \texttt{while}\ \texttt{(}\ e\ \texttt{)}\ blk \mid \texttt{break} \mid \texttt{continue} \mid \texttt{return}\ e$
$\mid e_f\ \texttt{(}\ e_1\ \texttt{,}\ e_2\ \texttt{,}\ \ldots\ \texttt{,}\ e_n\ \texttt{)}$

(类型) $t \coloneqq \texttt{Bool} \mid \texttt{Int64} \mid \texttt{String} \mid \texttt{Unit}$
$\mid \texttt{(}\ t_1\ \texttt{,}\ t_2\ \texttt{,}\ \ldots\ \texttt{,}\ t_n\ \texttt{)}\ \texttt{->}\ t_{ret}$

注：上述文法使用 EBNF 范式，其中

- `abc` 表示终结符号，即源码中的字符串 `abc`；
- NL 表示换行符；
- [...] 表示可选；
- (...)* 表示重复零或多次；
- 为表达简便易懂，上述定义中的若干列表直接使用了省略号 ... 表示可以特定分隔符重复零或多次的部分，而没有写成 EBNF 的形式。这些列表包括表达式块的参数列表、函数定义的形参列表、函数调用的实参列表、函数类型的参数类型列表等。

严格的 BNF 文法参见 小节 6.4 。具体程序的 AST 结构请使用实验框架中提供的 `ast` 命令打印查看。

3. 语义

3.1. 函数

njucj 的函数是一等公民，可以作为参数传递给其他函数，也可以作为其他函数的返回值。

njucj 中可以在顶层或函数体中定义函数。函数定义视为定义一个不可变的变量，其值为函数值，因此其作用域规则和使用 `let` 声明并当场初始化的不可变变量定义一致，不允许函数重载。如果函数名与**同一静态作用域内**已定义的变量或函数同名（即使函数参数类型或个数不同），则应当在 `FuncDecl` 处**报错** `DUPLICATED_DEF`。

本实验不涉及接口和抽象类，故函数必须有函数体。为了在后续实验中支持接口和抽象类中的抽象方法，在语法分析阶段允许函数体为空，但在解释器解析到存在该问题的函数定义时，应当在 `FuncDecl` 处**报错** `FUNC_MISSING_BODY`。

函数的形式参数是不可变的，其所在的作用域为函数体表达式块的作用域。

本实验中不要求静态类型检查和类型推断，但需要在运行时进行类型检查，因此在本次实验中**函数参数类型和返回类型均不可省略**。但为了在后续实验中支持类型推断，在语法分析阶段仍然允许省略返回类型的语法形式存在，故需要在本次实验中进行主动检查。若函数定义中省略了返回类型，则解析到该函数定义时应当在 `FuncDecl` 处**报错** `FUNC_MISSING_RETURN_TYPE`。

由于目前引入的类型间无子类型关系，故函数类型间也暂无子类型关系。

函数中可以使用 `return` 表达式返回值。执行 `return e` 时，计算表达式 `e` 的值，并将该值作为函数体表达式块的值（也就是函数调用表达式的值），结束函数的执行。如果函数体执行过程中没有遇到 `return` 表达式，则函数体表达式块的值为其最后一条表达式的值；如果函数体为空，则函数体表达式块的值为 `()`。

程序入口 `main` 中也可使用 `return` 表达式结束程序执行，并返回一个值作为程序的返回值。

3.2. 嵌套函数和闭包

njucj 中函数支持嵌套定义，此时内层函数等效于捕获了外层函数**当前已定义的**变量，构成闭包。注意，内层函数访问非局部变量（指外层函数中定义的变量，不包括全局变量）时应当忽略其**定义时**外层函数中尚未定义的变量，例如：

```
main() {
    let x = 42
    func middle(): Unit {
        func inner1(): Unit {
            println(x) // prints 42
        }
        func inner2(): Unit {
            println(y) // Error at line 8: [UNDEFINED_VAR]: y is not defined
        }
        let x = 2
        let y = 100
        println(x) // prints 2
        inner1()
        inner2()
    }
    middle()
}
```

建议在函数定义时记录下可访问的非局部变量列表，以便在函数调用时进行访问。

注意，内层函数可以访问自己，从而进行递归调用。请确保内层函数本身也被包含在其可访问的非局部变量列表中。

闭包保证了内层函数离开其静态作用域后，调用该函数时仍然可以访问外层函数的局部变量（这些变量对于内层函数来说是非局部变量）。不过，为防止捕获了可变变量的闭包逃逸，内层函数不允许访问可变非局部变量（外层函数或更外层函数中使用 `var` 声明的变量）。如有尝试访问，则应在对应的 `RefExpr` 处**报错** `FUNC_USE_MUTABLE_NONLOCAL`。

3.3. 顶层定义

在程序顶层可以定义全局变量和全局函数。

全局变量必须带有初始化表达式，否则应当在 `VarDecl` 处**报错** `GLOBAL_NO_INITIALIZER`。

全局变量的初始化表达式执行过程中可以访问任何时候定义的全局函数，但只能访问**先前定义**的全局变量，不能访问之后定义的全局变量，否则应当在使用时的 `RefExpr` 处**报错** `UNDEFINED_VAR`。

顶层定义的初始化顺序为：先将定义的全局函数加入全局作用域，然后按源代码顺序初始化全局变量并加入全局作用域。

3.4. 赋值

njucj 中的赋值表达式 `x = e` 仅允许对使用 `var` 定义的可变变量或尚未初始化的 `let` 定义的变量进行赋值操作。使用 `func` 定义的函数和使用 `let` 定义并当场初始化的不可变变量一样，不可被赋值。

尝试对使用 `func` 定义的函数进行赋值操作时，与尝试对已初始化的不可变变量赋值时一样，也应当在 `AssignExpr` 处**报错** `ASSIGN_IMMUT_VAR`。

函数的形式参数也是不可变的。若在函数体中尝试对形参赋值，则应当在对应的 `AssignExpr` 处**报错** `ASSIGN_IMMUT_VAR`。

3.5. 作用域

在实验 1 的作用域规则基础上，实验 2 对作用域规则进行了如下扩展：

- 存在一个全局作用域。全局作用域中预定义了一些内置函数（见 小节 3.7 ）。在程序顶层定义的变量和函数属于全局作用域，不可与内置函数重名，也不可互相重名，否则在发现重名的定义（`VarDecl` 或 `FuncDecl`）处**报错** `DUPLICATED_DEF`。
- 函数定义视为定义一个不可变的变量，其值为函数值，因此也不可以与当前静态作用域中的其他变量（或函数）同名。注意，函数不允许重载。
- 每个函数（以及程序入口 `main`）会创建一个函数级别的作用域，该作用域的父作用域为定义该函数时所在的静态作用域（对于 `main`，其作用域的父作用域为全局作用域）。
- 使用变量或函数时，递归地从当前静态作用域开始，向上寻找该变量或函数的定义。
  - 若在当前静态作用域中找到定义，使用该定义。
  - 若在某个祖先作用域中找到定义，则根据该定义的类型进行如下处理：
    - 该定义是局部变量或函数，则使用该定义。这种情况会出现在 `if` 或 `while` 等语句的语句块中。
    - 该定义是不可变非局部变量或函数，且位于当前作用域所在函数定义之前，则使用该定义。

  - 该定义是不可变非局部变量或函数，且位于当前作用域所在函数定义之后，则忽略该定义，继续前往上层静态作用域寻找。
  - 该定义是可变非局部变量，则在使用变量的 `RefExpr` 处**报错** `FUNC_USE_MUTABLE_NONLOCAL`。
  - 该定义是全局变量或函数，则使用该定义。
- 若未找到定义，在使用变量的 `RefExpr` 处**报错** `UNDEFINED_VAR`。

3.6. 函数调用

函数调用使用 `f(a1, a2, ... an)` 的形式，其中 `f` 是函数，`a1, a2, ... an` 是实参表达式列表。

由于函数是一等公民，函数调用表达式中的 `f` 可以是任意返回函数值的表达式，如使用 `func` 定义的函数、使用 `let` 或 `var` 定义的变量、该表达式所在函数的参数、返回函数的另一个函数调用表达式等。

该表达式语义如下：

1. 计算表达式 `f` 的值。如果得到的不是函数值，在函数调用表达式 `CallExpr` 处**报错** `CALLEE_NOT_FUNCTION`。否则，记得到的函数值为 `F`。
2. 检查函数值 `F` 的形参个数是否与实参个数 `n` 相等，若不相等，在函数调用表达式 `CallExpr` 处**报错** `CALL_ARG_COUNT_MISMATCH`。
3. 依次计算实参表达式列表 `a1, a2, ... an` 实参 `ai` 的值，并检查其与函数值 `F` 的形参类型是否匹配，若不匹配，在对应的参数表达式处**报错** `CALL_ARG_TYPE_MISMATCH`。注意，每计算一个实参表达式 `ai`，都应当立即检查其类型是否与对应形参类型匹配，而不是等到所有实参表达式都计算完毕后再进行检查。
5. 创建一个新的函数级别作用域 `S`，其父作用域为函数值 `F` 定义时所在的静态作用域。
6. 在作用域 `S` 中，依次为函数值 `F` 的每个形参定义一个不可变变量，并将其值设为对应实参的值。
7. 依次计算函数值 `F` 的函数体表达式块的值，返回该值作为整个函数调用表达式的值。

完成函数执行时，若返回值类型与函数声明的返回类型不匹配，则在函数定义 `FuncDecl` 处**报错** `FUNC_RETURN_TYPE_MISMATCH`。

3.7. 内置函数

在 njucj 中，需要与外部交互的操作无法使用语言本身实现，如各类 I/O 操作等，因此我们引入了内置作用域和内置函数来完成这些操作。为避免引入子类型关系，内置函数暂时不能作为一等公民使用。

本实验中，解释器应当实现 1 个内置函数：`println`。

3.7.1. println 函数

`println(x: Any): Unit`：打印参数 `x` 的值到标准输出，并在末尾添加换行符。该函数的返回值为 `()`。我们在语法中没有定义 `Any` 类型，这里的 `Any` 也仅作示意。`println` 函数的参数类型可以接受任意类型的值，因此对 `println` 进行函数调用时不必检查参数类型。

对各类参数值的打印格式如下：

- 对于 `Bool` 类型，打印 `true` 或 `false`。
- 对于 `Int64` 类型，打印对应的十进制整数表示。
- 对于 `String` 类型，打印字符串内容（不使用引号包裹）。

- 对于 `Unit` 类型，打印 `()`。
- 对于函数，打印由尖括号 `<>` 包裹的包含 `Function` 或 `function` 字样的字符串，
  如 `<Function>`、`<Function f>`、`<Function f(a1, a2, ..., an)>`、`<Built-in function println>` 等。
  你可以按自己的喜好选择函数打印格式，但必须在尖括号内包含 `Function` 或 `function` 字样。

该函数和实验 1 框架代码中的 `Value.toValueString()` 方法类似，但对于字符串的打印不使用引号包裹，也不对引号进行转义。

4. 输入输出

4.1. 样例

1. 递归计算 Fibonacci 数列

   输入：

   ```
   func fib(n: Int64): Int64 {
       if (n ≤ 1) {
           n
       } else {
           fib(n - 1) + fib(n - 2)
       }
   }

   main() {
       println(fib(4))
       println(fib(6))
       println(fib(8))
   }
   ```

   标准输出：

   ```
   3
   8
   21
   ()
   ```

   退出代码为 0。

   注意，最后的 `()` 是程序入口 `main` 返回值的输出。你应当和 Lab 1 一样，在程序执行结束后通过 `println(val.toValueString())` 输出程序的返回值。

2. 全局变量 输入：

   ```
   let immut = initImmut()
   var counter = 0

   func initImmut(): Int64 {
       var ans = 0
       var x = 0
       while (x < 10) {
           ans += x
       }
       ans
   }

   func increment(): Unit {
       counter = counter + 1
   }

   main() {
       increment()
       increment()
       counter
   }
   ```

   标准输出：

```
45
2
```

退出代码为 0。

3. 嵌套函数和闭包

   输入：

```
main() {
    let x = 10
    func getClosure(): () -> Int64 {
        func closure(): Int64 {
            x * 2
        }
        closure
    }
    getClosure()()
}
```

   标准输出：

```
20
```

   退出代码为 0。

4. 函数调用参数数目不匹配

   输入：

```
func add(x: Int64, y: Int64): Int64 {
    return x + y
}

main() {
    add(5)
}
```

   标准错误输出：

```
Error at line 6: [CALL_ARG_COUNT_MISMATCH]: Function `add` expects 2 arguments, but got
1.
```

   退出代码为 1。

5. 嵌套函数中访问非局部可变变量

   输入：

```
main() {
    var x = 10
    func inner(): Int64 {
        x * 2
    }
    inner()
}
```

   标准错误输出：

```
Error at line 5: [FUNC_USE_MUTABLE_NONLOCAL]: Cannot access mutable nonlocal variable `x`
in nested function.
```

退出代码为 1。

4.2. 测试用例保证

保证不包含任何具有语法错误的内容。

保证输入一定包含一个程序入口 `main`（事实上已由语法保证）。

保证 `main` 无参数。

保证 `main` 内含有一个或多个表达式或变量/函数定义，即 `main` 的表达式块不为空。

保证不出现未定义的类型，即所有显示写出的类型一定是基本类型（`Int64`、`Bool`、`Unit`）、字符串类型（`String`），以及这些类型组合成的函数类型（或再次组合出的函数类型）。

5. 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 12/3, 23:59

5.1. 提交格式

将你的代码文件打包成 `lab2_[YOUR_ID]_[YOUR_NAME].zip`，例如，`lab2_123456789_张三.zip`，文件树如下

```
lab2_[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

请不要在代码中导入或使用 `std.database`、`std.net`、`std.posix`、`std.process`、`std.runtime` 包中的任何功能

5.2. 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。你如何设计函数类型的值? 如何实现函数的调用栈? 如何处理函数的静态作用域? 如何确保嵌套函数忽略其定义时尚未声明的外层局部变量? 如何确定嵌套函数能否访问非局部可变变量? 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。

5.3. 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。

5.4. 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。

6. 附录

6.1. 更新说明

6.1.1. 更新内容

本次实验框架更新主要包括：

1. 词法分析
   - 修复了 `_` 开头，后接单字符的标识符无法正确识别的问题。
   - 修复了分号后接换行符时后续 `Token` 行号错误的问题。
2. 语法分析
   - 修复了无参函数类型无法正确识别的问题。
   - 支持了函数类型中包含换行符的情况。
   - 修复了行首的负号表达式被错误识别为与上一行连接的二元减法表达式的问题。
   - 修复了负数字面量被拆分为操作数为字面量的负号表达式的问题，现在字面量可能包含负数了，**请确保你对 `LitConstExpr` 的处理允许负数字面量**。
3. AST 打印
   - 修复了字面量打印时一定添加引号的问题，并额外打印了字面量的具体类别 `kind`。
   - 修复了函数参数打印时不打印参数名 `name` 的问题。
   - 添加了打印 `Token` 时附加行号和列号注释的功能。
   - 添加了字符串自动添加引号和转义功能，使得包含冒号、减号、空格、换行符、制表符等特殊字符的字符串可以正确输出为 YAML 格式。
4. 错误代码（`src/executor/Errors.cj`）
   - 修正了错误代码的拼写错误，**请注意修正错误代码使用处的拼写**。
   - 添加了本次实验所需的错误代码定义。
5. 其他
   - 在 `src/test/Parser_test.cj` 中补全了缺失的 `import` 语句。不过该文件只是语法分析的测试文件，并没有被实验框架使用，对完成实验也没有帮助。

6.1.2. 升级方式

在升级实验框架前，请务必备份你当前的实验代码。本次实验提供两种升级方式：

1. 使用 `patch -p 1 --merge -i lab2_update.diff` 命令应用补丁文件 `lab2_update.diff`。请关注命令行输出，以解决可能出现的冲突。
2. 或者，解压缩 `lab2.zip`，并将实验 1 中你修改或创建的文件复制到实验 2 的文件夹中。

6.2. 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。

ASSIGN_IMMUT_VAR 试图给 `let` 定义的不可变变量或 `func` 定义的函数赋值。

UNDEFINED_VAR 试图使用未定义的变量或函数。

DUPLICATED_DEF 试图在同一静态作用域内定义同名变量。

GLOBAL_NO_INITIALIZER 全局变量定义缺少初始化表达式。

FUNC_MISSING_RETURN_TYPE 函数定义中省略了返回类型。

FUNC_MISSING_BODY 函数定义中缺少函数体。

CALLEE_NOT_FUNCTION 试图调用一个非函数值。

CALL_ARG_COUNT_MISMATCH 函数调用时实参个数与形参个数不匹配。

CALL_ARG_TYPE_MISMATCH 函数调用时实参类型与形参类型不匹配。

FUNC_USE_MUTABLE_NONLOCAL 试图访问可变非局部变量。

FUNC_RETURN_TYPE_MISMATCH 函数返回值类型与声明的返回类型不匹配。

以上是与本次实验内容相关的错误代码。除此之外，你依然需要处理实验 1 中定义的错误代码：

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`。

ASSIGN_TYPE_MISMATCH 试图给变量赋值成不同的类型。

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。

6.3. Tokens

见 仓颉语言文档 - `TokenKind`

6.4. Grammar

详见 `currentGrammar.md` 文件

具体而言，本次实验要求可以处理以下文法的程序：

```
## TRANSLATION UNIT

translationUnit
    : end* (topLevelObject (end+ topLevelObject?)*)?
    (end+ mainDefinition)? NL* (topLevelObject (end+ topLevelObject?)*)? EOF
    ;

end
    : NL | SEMI
    ;

## MAIN ENTRY DEFINITION

mainDefinition
    : MAIN
    NL* LPAREN NL* RPAREN
    (NL* COLON NL* type)?
    NL* block
    ;

## TOP-LEVEL DEFINITION

topLevelObject
    : functionDefinition
    | variableDeclaration
    ;

## FUNCTION DEFINITION

functionDefinition
    :FUNC
        NL* identifier
        NL* functionParameters
```

```
    (NL* COLON NL* type)?
    (NL* block)?
    ;

functionParameters
    : (LPAREN (NL* unnamedParameterList)?
        NL* RPAREN NL*)
    ;

unnamedParameterList
    : unnamedParameter (NL* COMMA NL*  unnamedParameter)*
    ;

unnamedParameter
    : (identifier | WILDCARD) NL* COLON NL* type
    ;

## VARIABLE DEFINITION

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

patternsMaybeIrrefutable
    : wildcardPattern
    | varBindingPattern
    ;


varBindingPattern
    : identifier
    ;

wildcardPattern
    : WILDCARD
    ;

## EXPRESSION

expression
    : assignmentExpression
    ;

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

leftValueExpression
    : leftValueExpressionWithoutWildCard
    ;

leftValueExpressionWithoutWildCard
    : identifier

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

```
logicConjunctionExpression
    : equalityComparisonExpression (NL* AND NL* equalityComparisonExpression)*
    ;

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

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

additiveExpression
    : multiplicativeExpression (additiveOperator NL* multiplicativeExpression)*
    ;

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

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

prefixUnaryExpression
    : prefixUnaryOperator* postfixExpression
    ;

postfixExpression
    : atomicExpression
    | postfixExpression callSuffix
    ;

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
    | functionDefinition
    ;
```

```
assignmentOperator
    : ASSIGN
    ;

equalityOperator
    : NOTEQUAL
    | EQUAL
    ;

comparisonOperator
    : LT
    | GT
    | LE
    | GE
    ;

additiveOperator
    : NL* ADD | SUB
    ;

exponentOperator
    : EXP
    ;

multiplicativeOperator
    : MUL
    | DIV
    | MOD
    ;

prefixUnaryOperator
    : SUB
    | NOT
    ;

## TYPE

type
    : arrowType
    | atomicType
    ;

arrowType
    : arrowParameters NL* ARROW NL* type
    ;

arrowParameters
    : LPAREN NL* (type (NL* COMMA NL* type)* NL*)? RPAREN
    ;

atomicType
    : charLangTypes
    | userType
    | parenthesizedType
    ;

atomicType
    : charLangTypes
    | userType
    ;
```

```
charLangTypes
    : INT64
    | BOOLEAN
    | UNIT
    | Nothing
    ;

userType
    : identifier
    ;
```