MLIR 编译框架的使用与探索

——编译原理课程大作业实验报告

实验环境

- 虚拟机软件：VirtualBox 6.1.32
- 操作系统：Ubuntu 20.04
- 内核版本：linux kernel 5.13.0-35-generic
- 软件：
  - git 2.25.1
  - cmake 3.16.3
  - clang 10.0.0-4ubuntu1
    - Target: x86_64-pc-linux-gnu
    - Thread model: posix
  - lld
  - ninja 1.10.0

实验过程

构建实验环境

前置工具

- git，cmake，clang，lld，ninja
- 在 */home/username* 目录下安装
  - 我的 *username* 为 *thousanrance* 。

```
sudo apt install git
sudo apt install cmake
sudo apt-get install clang lld
sudo apt install ninja-build
```

安装MLIR

- 下载MLIR项目文件

```
git clone https://github.com/Jason048/llvm-project.git
mkdir llvm-project/build
cd llvm-project/build
cmake -G Ninja ../llvm \
   -DLLVM_ENABLE_PROJECTS=mlir \
   -DLLVM_BUILD_EXAMPLES=ON \
   -DLLVM_TARGETS_TO_BUILD="X86;NVPTX;AMDGPU" \
   -DCMAKE_BUILD_TYPE=Release \
   -DLLVM_ENABLE_ASSERTIONS=ON \
   -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ -DLLVM_ENABLE_LLD=ON
```

- 编译MLIR

```
cmake --build . --target check-mlir
```

安装tiny_project

- 下载tiny_project项目文件

```
git clone https://github.com/Jason048/tiny_project.git
mkdir build
cd build

# 将下面的三行命令在build目录下执行
LLVM_DIR=/home/username/llvm-project/build/lib/cmake/llvm \
MLIR_DIR=/home/username/llvm-project/build/lib/cmake/mlir \
cmake -G Ninja ..
```

基础部分：词法分析与语法分析

词法分析器

- 对关键词和变量名进行分析。
- 文件路径：*/tiny_project/tiny/include/tiny/Lexer.h*
- 在 *Lexer.h* 搜索 "TODO"，可以看到需要补充的代码位置。 实现以下功能：
  - 能够识别“return”、“def”和“var”三个关键字
  - 按照如下要求识别变量名：
    - 变量名以字母开头
    - 变量名由字母、数字和下划线组成
    - 变量名中有数字时，数字应该位于变量名末尾
    - 例如：有效的变量名可以是 a123, b_4, placeholder 等。

具体实现

- 在枚举类型 Token 中添加 tok_invalid_identifier 来代表不合法的变量名。（已用 //basic part 注释）

```
enum Token : int {
  tok_semicolon = ';',
  tok_parenthese_open = '(',
  tok_parenthese_close = ')',
  tok_bracket_open = '{',
```

```
  tok_bracket_close = '}',
  tok_sbracket_open = '[',
  tok_sbracket_close = ']',

  tok_eof = -1,

  // commands
  tok_return = -2,
  tok_var = -3,
  tok_def = -4,
  tok_struct = -5,

  // primary
  tok_identifier = -6,
  tok_number = -7,

  //basic part
  tok_invalid_identifier = -8,
};
```

- TODO：识别变量名与关键字
  - 读入第一个字母【isalpha()】时开始进行变量名与关键字的识别。
  - identifierStr用来记录这个关键字或变量名。
  - 接下来读入的字符只要是字母或者数字【isalnum()】，就拼在identifierStr后。
  - 布尔型变量containDigit用来记录identifierStr里是否含有数字；invalid用来记录变量名是否不合法。两者都初始化为false。
  - 在读入字符的过程中，如果读入的字符是数字，将containDigit置为true。
  - 在读入字符的过程中，如果containDigit为true，且新读入的字符是字母或下划线，则将invalid置为true。因为这意味着在数字之后又出现了字母或者下划线，说明数字没有位于变量名末尾，是不合法的变量名。
  - 如果读到不是字母、数字、下划线的字符，则结束继续读入字符，对当前已经读入的字符串进行判断。
  - 如果是关键字 return、var、def 中的某一个，则返回其对应的枚举类型。
  - 如果关键字，则为变量名。检查 invalid，返回 合法 / 非法 变量名对应的枚举类型。

```
Token getTok() {
    // Skip any whitespace.
    while (isspace(lastChar))
      lastChar = Token(getNextChar());

    // Save the current location before reading the token characters.
    lastLocation.line = curLineNum;
    lastLocation.col = curCol;

    //TODO: Here you need to identify:
    //      1. Keywords: "return", "def", "var"
    //      2. Variable names
    //      Note that a variable name should follow some rules:
    //      1) It should start with a letter.
    //      2) It consists of letters, numbers and underscores.
    //      3) If there are numbers in the name, they should be at the end of the
name.
    //      For example, these names are valid: a123, b_4, placeholder

    //Hints: 1. You can refer to the implementaion of identifying a "number" in the
same function.
    //       2. Some functions that might be useful:  isalpha(); isalnum();
```

```
    //       3. Maybe it is better for you to go through the whole lexer before you
get started.

    /*  Write your code here. */
    //Identify keywords and variable names:[a-zA-Z][a-zA-Z_0-9]
    if (isalpha(lastChar))
    {
        identifierStr = (char)lastChar;
        bool containDigit = false;
        bool invalid = false;
        while (isalnum((lastChar = Token(getNextChar()))) || lastChar == '_')
        {
            if(isdigit(lastChar))
            {
                containDigit = true;
            }
            if(containDigit && (isalpha(lastChar) || lastChar == '_'))
            {
                invalid = true;
            }
            identifierStr += (char)lastChar;
        }

        if (identifierStr == "return")
        {
            return tok_return;
        }
        if (identifierStr == "def")
        {
            return tok_def;
        }
        if (identifierStr == "var")
        {
            return tok_var;
        }

        if (invalid)
        {
            return tok_invalid_identifier;
        }
        return tok_identifier;
    }
    /*  Write your code above. */

    // Identify a number: [0-9] ([0-9.])*
    if (isdigit(lastChar)) {
      std::string numStr;
      do {
        numStr += lastChar;
        lastChar = Token(getNextChar());
      } while (isdigit(lastChar) || lastChar == '.');

      numVal = strtod(numStr.c_str(), nullptr);
      return tok_number;
    }

    if (lastChar == '#') {
      // Comment until end of line.
      do {
        lastChar = Token(getNextChar());
      } while (lastChar != EOF && lastChar != '\n' && lastChar != '\r');
```

```
      if (lastChar != EOF)
        return getTok();
    }

    // Check for end of file.  Don't eat the EOF.
    if (lastChar == EOF)
      return tok_eof;

    // Otherwise, just return the character as its ascii value.
    Token thisChar = Token(lastChar);
    lastChar = Token(getNextChar());
    return thisChar;
  }
```

- 在 *Parser.h* 中，对错误信息输出函数进行添加。（已用 //basic part 注释）
  - 如果当前Token是不合法的变量名对应的枚举类型，则输出 "[invalid variable name]" 信息。

```
/// Helper function to signal errors while parsing, it takes an argument
/// indicating the expected token and another argument giving more context.
/// Location is retrieved from the lexer to enrich the error message.
template <typename R, typename T, typename U = const char *>
std::unique_ptr<R> parseError(T &&expected, U &&context = "") {
    auto curToken = lexer.getCurToken();
    llvm::errs() << "Parse error (" << lexer.getLastLocation().line << ", "
        << lexer.getLastLocation().col << "): expected '" << expected
        << "' " << context << " but has Token " << curToken;
    /* basic part */
    if (curToken == tok_invalid_identifier)
    {
        llvm::errs() << "[invalid variable name]";
    }
    /* basic part above */
    if (isprint(curToken))
        llvm::errs() << " '" << (char)curToken << "'";
    llvm::errs() << "\n";
    return nullptr;
}
```

语法分析器

- 对 Tiny 语言中一维或二维 Tensor 的声明及定义进行语法分析。
- 文件路径：*/tiny_project/tiny/include/tiny/Parser.h*
- 在Parser.h搜索"TODO"，可以看到需要补充的代码位置。 实现以下功能：
  - 语法变量必须以“var”开头，后面为变量名及类型，最后为变量的初始化
  - 语法分析器已支持以下两种初始化形式，以一个二维矩阵为例：
    - var a = [[1, 2, 3], [4, 5, 6]]
    - var a<2,3> = [1, 2, 3, 4, 5, 6]
  - 需要额外支持第三种新的形式：
    - var a[2][3]= [1, 2, 3, 4, 5, 6]

具体实现

- TODO：额外支持第三种新的形式。

- 如果当前Token不是 '<' 或 '[' 则提示错误信息。
- 如果是，则分情况讨论。
- 如果是 '<'，照搬原代码。
- 如果是 '['，与上一种情况不同的地方是，分隔两个数字的Token不是一个逗号 ','，而是两个Token —— ']' 和 '[' 。所以在读入第一个 ']' 后，还要再尝试读入一个Token，并且希望它是 '[' ，所以要用布尔变量 ismedium（初始化为true）记录读入的 ']' 是否是在数字中间。只有 ismedium 为 true 时，才会尝试读入下一个 Token。
- 读入第一个 ']' 后会将 ismedium 置为false，当再次读到 ']' 时，因为 ismedium 为 flase，所以不会再尝试读入下一个Token，而是直接跳出循环，去进行type的结尾检查，因为实际上这个 ']' 是type的结尾。

```
/// type ::= < shape_list >
/// shape_list ::= num | num , shape_list
// TODO: make an extension to support the new type like var a[2][3] = [1, 2, 3, 4, 5, 6];
std::unique_ptr<VarType> parseType() {
    if (lexer.getCurToken() != '<' && lexer.getCurToken() != '[')
    {
        return parseError<VarType>("< or [", "to begin type");
    }

    auto type = std::make_unique<VarType>();

    if (lexer.getCurToken() == '<')
    {
        lexer.getNextToken(); // eat <

    while (lexer.getCurToken() == tok_number)
    {
        type->shape.push_back(lexer.getValue());
        lexer.getNextToken();
        if (lexer.getCurToken() == ',')
        {
            lexer.getNextToken();
        }
    }

    if (lexer.getCurToken() != '>')
    {
        return parseError<VarType>(">", "to end type");
    }
    }

    if (lexer.getCurToken() == '[')
    {
        lexer.getNextToken(); // eat [

        bool ismedium = true;
        while (lexer.getCurToken() == tok_number)
        {
            type->shape.push_back(lexer.getValue());
            lexer.getNextToken();
            if (lexer.getCurToken() == ']')
            {
                if(ismedium)
                {
                    lexer.getNextToken();
                    ismedium = false;
                    if (lexer.getCurToken() == '[')
                    {
```

```
                    lexer.getNextToken();
                }
            }
            else
            {
                break;
            }

        }
    }

    if (lexer.getCurToken() != ']')
    {
        return parseError<VarType>("]", "to end type");
    }
}

lexer.getNextToken(); // eat > or ]
return type;

}
```

- TODO：语法变量必须以“var”开头，后面为变量名及类型，最后为变量的初始化
  - 检查当前 Token 是否是关键字 var 对应的枚举类型，如果不是就输出错误信息，如果是就读入下一个Token。
  - 已经确定以"var" 开头正确，检查当前 Token 是否是合法变量名 identifier 对应的枚举类型，如果不是就输出错误信息。
- TODO：额外支持第三种新的形式。
  - 已经确定以“var”开头正确，后接变量名正确，开始检查类型。
  - 如果当前Token是 '<' 或 '['，则调用parseType()函数（即上一个函数）对类型进行语法分析。

```
// Parse a variable declaration,
// 1. it starts with a `var` keyword, followed by a variable name and initialization
// 2. Two methods of initialization have been supported:
//    (1) var a = [[1, 2, 3], [4, 5, 6]];
//    (2) var a <2,3> = [1, 2, 3, 4, 5, 6];
// You need to support the third method:
//    (3) var a [2][3] = [1, 2, 3, 4, 5, 6];
// Some functions may be useful:  getLastLocation(); getNextToken();
std::unique_ptr<VarDeclExprAST>
  parseVarDeclaration(bool requiresInitializer) {

    // TODO: check to see if this is a 'var' declaration
    //       If not, report the error with 'parseError', otherwise eat 'var'
    /* Write your code here. */
    if (lexer.getCurToken() != tok_var)
    {
        return parseError<VarDeclExprAST>("var", "to begin declaration");
    }
    auto loc = lexer.getLastLocation();
    lexer.getNextToken(); // eat var
    /* Write your code above */

    // TODO: check to see if this is a variable name
    //       If not, report the error with 'parseError'
    /* Write your code here. */
    if(lexer.getCurToken() != tok_identifier)
    {
        return parseError<VarDeclExprAST>("identified", "after 'var' declaration");
```

```
    }
    /* Write your code above */

    // eat the variable name
    std::string id(lexer.getId());
    lexer.getNextToken(); // eat id

    std::unique_ptr<VarType> type;
    // TODO: modify code to additionally support the third method: var a[][] = ...
    if (lexer.getCurToken() == '<' || lexer.getCurToken() == '[') {
      type = parseType();
      if (!type)
        return nullptr;
    }
    if (!type)
      type = std::make_unique<VarType>();

    std::unique_ptr<ExprAST> expr;
    if (requiresInitializer) {
      lexer.consume(Token('='));
      expr = parseExpression();
    }
    return std::make_unique<VarDeclExprAST>(std::move(loc), std::move(id),
                                            std::move(*type), std::move(expr));
  }
```

进阶部分：代码优化

- 进行冗余代码的消除：Tiny 语言内置的 transpose 函数会对矩阵进行转置操作。然而，对同一个矩阵进行两次转置运算会得到原本的矩阵，相当于没有转置。矩阵的转置运算是通过嵌套 for循环实现的，而嵌套循环是影响程序运行速度的重要因素。因此，侦测到这种冗余代码并进行消除是十分必要的。
- 文件路径：*/tiny_project/tiny/mlir/TinyCombine.cpp*
- 在TinyCombine.cpp搜索"TODO"，可以看到需要补充的代码位置。实现以下功能：
  - 将tiny dialect的冗余转置代码优化pass补充完整。最终实现冗余代码的消除。

具体实现

- TODO：实现冗余代码的消除
  - 获取当前转置操作输入的操作数。
  - 如果当前输入的操作数不是转置，说明没有冗余转置，则返回匹配失败，表达式不同进行优化。
  - 如果输入的操作数是转置，说明有冗余转置，所以重写表达式，消除冗余转置，最后返回匹配成功。

```
/// TODO Redundant code elimination
  mlir::LogicalResult
  matchAndRewrite(TransposeOp op,
                  mlir::PatternRewriter &rewriter) const override {

    // Step 1: Get the input of the current transpose.
    // Hint: For op, there is a function: op.getOperand(), it returns the parameter of
a TransposeOp and its type is mlir::Value.
    /* Write your code here. */
    mlir::Value transposeInput = op.getOperand();
```

```
    /* Write your code above. */

    // Step 2: Check whether the input is defined by another transpose. If not
defined, return failure().
    // Hint: For mlir::Value type, there is a function you may use:
    //       template<typename OpTy> OpTy getDefiningOp () const
      //       If this value is the result of an operation of type OpTy, return the
operation that defines it
    /* Write your code here. */
    /* if () return failure(); */
    TransposeOp transposeInputOp = llvm::dyn_cast_or_null<TransposeOp>
(transposeInput.getDefiningOp());
    if(!transposeInputOp)
    {
        return failure();
    }
    /* Write your code above. */

    // step 3: Otherwise, we have a redundant transpose. Use the rewriter to remove
redundancy.
    // Hint: For mlir::PatternRewriter, there is a function you may use to remove
redundancy:
    //       void replaceOp (mlir::Operation *op, mlir::ValueRange newValues)
    //       Operations of the second argument will be replaced by the first argument.
    /* Write your code here. */
    rewriter.replaceOp(op, {transposeInputOp.getOperand()});
    /* Write your code above. */

    return success();
  }
```

实验结果

词法分析器验证

- test_1 - test_4
- 以test_1为例： build tiny并执行下面测试用例以验证词法分析器是否能够检测出错误的词法单元。

```
# 在/home/username/tiny_project/build 目录下开始执行
cmake --build . --target tiny
cd ../
build/bin/tiny test/tiny/parser/test_1.tiny -emit=ast
```

- 执行结果：

test_1：变量名以下划线开头。

```
thousanrance@thousanrance-VirtualBox:~/tiny_project$ build/bin/tiny test/tiny/parser/test_1.tiny -emit=ast
Parse error (5, 8): expected 'identified' after 'var' declaration but has Token 95 '_'
Parse error (5, 8): expected 'nothing' at end of module but has Token 95 '_'
```

test_2：变量名中包含数字但是没有在结尾。

```
thousanrance@thousanrance-VirtualBox:~/tiny_project$ build/bin/tiny test/tiny/parser/test_2.tiny -emit=ast
Parse error (5, 8): expected 'identified' after 'var' declaration but has Token -8[invalid variable name]
Parse error (5, 8): expected 'nothing' at end of module but has Token -8[invalid variable name]
```

test_3：变量名中包含数字但是没有在结尾。

```
thousanrance@thousanrance-VirtualBox:~/tiny_project$ build/bin/tiny test/tiny/parser/test_3.tiny -emit=ast
Parse error (5, 8): expected 'identified' after 'var' declaration but has Token -8[invalid variable name]
Parse error (5, 8): expected 'nothing' at end of module but has Token -8[invalid variable name]
```

test_4：变量名中包含数字但是没有在结尾。

```
thousanrance@thousanrance-VirtualBox:~/tiny_project$ build/bin/tiny test/tiny/parser/test_4.tiny -emit=ast
Parse error (5, 9): expected 'identified' after 'var' declaration but has Token -8[invalid variable name]
Parse error (5, 9): expected 'nothing' at end of module but has Token -8[invalid variable name]
```

语法分析器验证

- test_5 - test_6
- 测试test_5: 验证语法分析器能否识别出三种声明及初始化方式，输出AST（-emit=ast）：

```
build/bin/tiny test/tiny/parser/test_5.tiny -emit=ast
```

- 执行结果：

第三种类型 —— d[2][3] ——已成功识别。

```
thousanrance@thousanrance-VirtualBox:~/tiny_project$ build/bin/tiny test/tiny/parser/test_5.tiny -emit=ast
Module:
  Function
    Proto 'multiply_transpose' @test/tiny/parser/test_5.tiny:2:1
    Params: [a, b]
    Block {
      Return
        BinOp: * @test/tiny/parser/test_5.tiny:3:25
          Call 'transpose' [ @test/tiny/parser/test_5.tiny:3:10
            var: a @test/tiny/parser/test_5.tiny:3:20
          ]
          Call 'transpose' [ @test/tiny/parser/test_5.tiny:3:25
            var: b @test/tiny/parser/test_5.tiny:3:35
          ]
    } // Block
  Function
    Proto 'main' @test/tiny/parser/test_5.tiny:6:1
    Params: []
    Block {
      VarDecl a<> @test/tiny/parser/test_5.tiny:7:3
        Literal: <2, 3>[ <3>[ 1.000000e+00, 2.000000e+00, 3.000000e+00], <3>[ 4.000000e+00, 5.000000e+00, 6.000000e+00]] @test/tiny/parser/test_5.tiny:7:11
      VarDecl b<2, 3> @test/tiny/parser/test_5.tiny:8:3
        Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00] @test/tiny/parser/test_5.tiny:8:17
      VarDecl c<> @test/tiny/parser/test_5.tiny:9:3
        Call 'multiply_transpose' [ @test/tiny/parser/test_5.tiny:9:11
          var: a @test/tiny/parser/test_5.tiny:9:30
          var: b @test/tiny/parser/test_5.tiny:9:33
        ]
      VarDecl d<2, 3> @test/tiny/parser/test_5.tiny:10:3
        Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00] @test/tiny/parser/test_5.tiny:10:17
      VarDecl e<> @test/tiny/parser/test_5.tiny:11:3
        Call 'multiply_transpose' [ @test/tiny/parser/test_5.tiny:11:11
          var: b @test/tiny/parser/test_5.tiny:11:30
          var: c @test/tiny/parser/test_5.tiny:11:33
        ]
      VarDecl f<> @test/tiny/parser/test_5.tiny:12:3
        Call 'multiply_transpose' [ @test/tiny/parser/test_5.tiny:12:11
          Call 'transpose' [ @test/tiny/parser/test_5.tiny:12:30
            var: a @test/tiny/parser/test_5.tiny:12:40
          ]
          var: c @test/tiny/parser/test_5.tiny:12:44
        ]
    } // Block
```

- 测试test_6: 执行以下指令查看输入程序的运行结果（-emit=jit）：

```
build/bin/tiny test/tiny/parser/test_6.tiny -emit=jit
```

- 执行结果：

```
thousanrance@thousanrance-VirtualBox:~/tiny_project$ build/bin/tiny test/tiny/parser/test_6.tiny -emit=jit
1.000000 8.000000 27.000000
64.000000 125.000000 216.000000
```

代码优化验证

- 对于test_7中的例子：

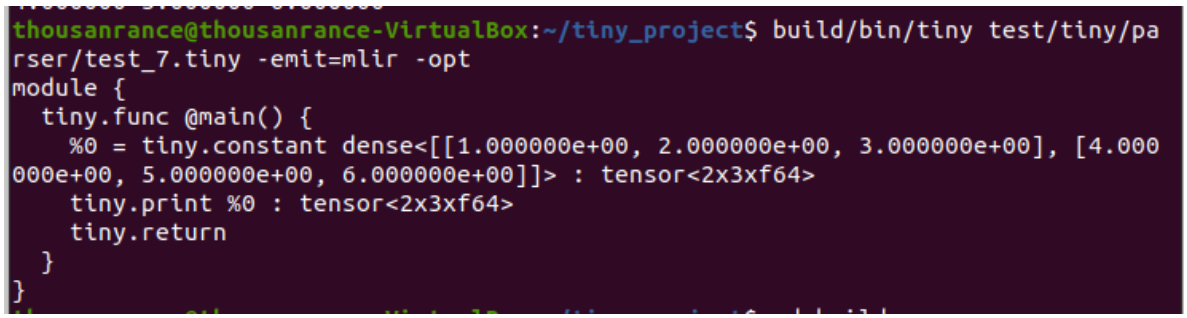
```
def transpose_transpose(x) {
  return transpose(transpose(x));
}

def main() {
  var a<2, 3> = [[1, 2, 3], [4, 5, 6]];
  var b = transpose_transpose(a);
  print(b);
}
```

- 执行以下指令，输出转换后的tiny dialect（即转换后的代码表示），查看输出结果可判断是否成功消除冗余转置：

```
build/bin/tiny test/tiny/parser/test_7.tiny -emit=mlir -opt
```

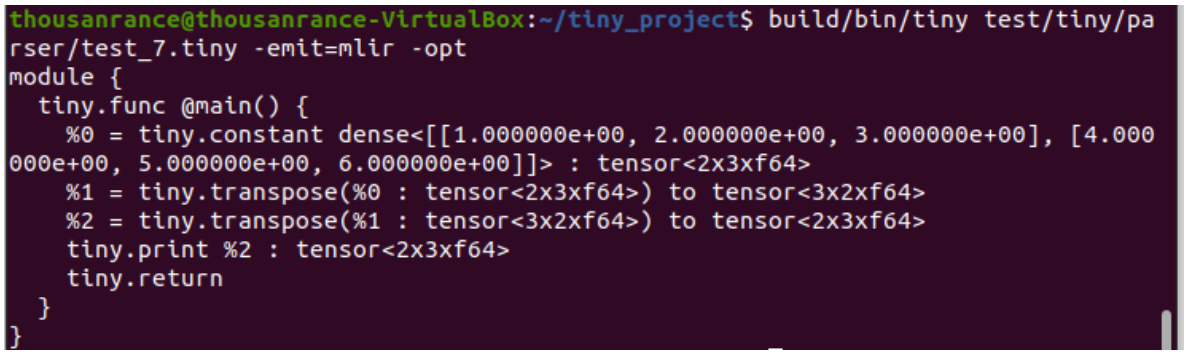
- 执行结果：已消除转置操作。



- 注释掉刚刚在TinyCombine.cpp中添加的代码，执行指令查看未优化的输出，对比优化前后输出的差异：

```
build/bin/tiny test/tiny/parser/test_7.tiny -emit=mlir -opt
```

- 执行结果：存在冗余转置。



- 此外，还可以利用不同指令，输出此测试用例在编译过程中出现的多种形式：

```
# 将.tiny文件转换为抽象语法树AST：
build/bin/tiny test/tiny/parser/test_7.tiny -emit=ast

# 或直接得到tiny程序的运行结果，是否进行消除冗余不会影响运行结果
build/bin/tiny test/tiny/parser/test_7.tiny -emit=jit
```

- 执行结果：

```
thousanrance@thousanrance-VirtualBox:~/tiny_project$ build/bin/tiny test/tiny/parser/test_7.tiny -emit=ast
  Module:
    Function
      Proto 'transpose_transpose' @test/tiny/parser/test_7.tiny:4:1
      Params: [x]
      Block {
        Return
          Call 'transpose' [ @test/tiny/parser/test_7.tiny:5:10
            Call 'transpose' [ @test/tiny/parser/test_7.tiny:5:20
              var: x @test/tiny/parser/test_7.tiny:5:30
            ]
          ]
      } // Block
    Function
      Proto 'main' @test/tiny/parser/test_7.tiny:8:1
      Params: []
      Block {
        VarDecl a<2, 3> @test/tiny/parser/test_7.tiny:9:3
          Literal: <2, 3>[ <3>[ 1.000000e+00, 2.000000e+00, 3.000000e+00], <3>[ 4.000000e+00, 5.000000e+00, 6.000000e+00]] @test/tiny/parser/test_7.tiny:9:17
        VarDecl b<> @test/tiny/parser/test_7.tiny:10:3
          Call 'transpose_transpose' [ @test/tiny/parser/test_7.tiny:10:11
            var: a @test/tiny/parser/test_7.tiny:10:31
          ]
        Print [ @test/tiny/parser/test_7.tiny:11:3
          var: b @test/tiny/parser/test_7.tiny:11:9
        ]
      } // Block
```

```
thousanrance@thousanrance-VirtualBox:~/tiny_project$ build/bin/tiny test/tiny/parser/test_7.tiny -emit=jit
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
```

总结

- 通过本次大作业，我对这种递归进行词法分析和语法分析的方法有了更深的理解。并且初步学习了MLIR编译框架，学到了如何通过MLIR编译框架生成语法树，学到了如何在MLIR框架下用C++编写表达式的匹配与重写函数来优化代码。
- 在进行本次实验的过程中，应特别注意路径问题，包括安装路径、各条指令执行的路径等。路径错误不是什么很有技术含量的错误，但会造成很多的时间浪费。
- 在实现词法分析器识别关键字与变量名时，我原本是这样写的：

```
if (isalpha(lastChar))
{
    identifierStr = (char)lastChar;
    while (isalpha((lastChar = Token(getNextChar()))) || lastChar == '_')
    {
        identifierStr += (char)lastChar;
    }

    if (identifierStr == "return")
    {
        return tok_return;
    }
    if (identifierStr == "def")
    {
        return tok_def;
    }
    if (identifierStr == "var")
    {
        return tok_var;
    }

    if (isdigit(lastChar))
    {
        while (isdigit((lastChar = Token(getNextChar()))))
        {
            identifierStr += (char)lastChar;
        }
        return tok_identifier;
    }
    return tok_identifier;
}
```

这样写的话，会在变量名为 a2b3 这类情况时，返回两个连续的identifier。我原本以为原有的语法分析器会识别这种错误，但是测试时发现语法分析器只会在函数consume()中用assert()去检测，然后报一个词法单元不匹配的错误。所以我修改了实现方式，将整个变量名读入后再判断是否合法，不合法则返回添加的新枚举类型，然后在原有的报错函数中添加相应的打印信息。

- 感谢老师与助教设计大作业，编写实验文档，讲解实验，让我们能够更清晰地了解实验具体要求、掌握实验所需技能。
- 参考：https://github.com/llvm/llvm-project/tree/main/mlir/examples/toy