目录

1 需求分析

1.1 程序功能

本实验要求实现一个简化版 C 语言的编译器，可以提供词法分析、语法分析、符号表管理、中间代码生成以及目标代码生成等功能。

本程序要求用户提供一个源程序文件，即可进行编译。编译过程中将输出词法分析、语法分析、中间代码及最终的可执行程序，或在解析出错时在控制台输出错误信息提示。

1.2 输入格式

本程序要求输入的源程序是一个简化版的 C 语言程序，该程序可定义普通变量、函数及数组，支持 if-else 分支语句和 while 循环语句，仅支持 int 型数据，不支持其它数据类型。

1.3 输出格式

对于一个格式正确的源程序，本程序将输出以下几个文件：

- 词法分析结果：以普通文本形式输出解析出的 token 流，包括 token 的类型、值与所在行列数
- 语法分析结果：以 Json 格式输出解析出的语法树，每个节点非叶子节点包含一个非终结符，叶子节点包含一个终结符
- 中间代码：以 LLVM IR 文本格式表示的中间代码，其中包含源程序生成的中间代码，以及为了展示结果而后期添加的 printf 系统调用代码
- 可执行文件：在 Windows 平台上运行的可执行文件，将会使用 printf 输出部分执行时的步骤以展示其运行过程

对于一个格式错误的源程序，本程序会在解析的不同阶段给出不同的错误提示：

- 若存在未知的词法符号，则会在词法分析阶段提示出现了未知的符号

- 若符号无法应用于任何产生式，则会在语法分析阶段提示某处缺少了何种符号
- 若生成中间代码过程中使用了未定义的符号，则会在中间代码生成时提示符号未定义

2 概要设计

本程序主要由以下四部分构成，每部分的产生的结果依次送入后续步骤处理。当其中某一步骤发生错误时，控制台将输出错误提示，并终止解析。

- 词法分析：由 TokenGenerator 根据原始输入文本，创建 CommonTokenStream 表示 Token 流，Token 流中记录了 Token 所属的类别、文本、行列数等信息
- 语法分析：由 GrammarGenerator 根据 Token 流，创建 ProgramContext 表示整个程序的语法上下文，再通过 DisplayTree 将上下文转换为 Json 格式的语法树
- 中间代码生成：由 IRGenerator 根据语法上下文，创建 LLVMModule 以内存形式表示 LLVM IR。然后可将 LLVMModule 转换为文本格式或二进制格式的中间代码。在构造 LLVMModule 的过程中，除了将原本的语法写入其中，还需要额外加入 printf 系统调用来展示程序的执行情况
- 可执行文件生成：由 ExecutableGenerator 根据二进制格式中间代码，创建目标文件，并将目标文件链接为可执行文件

3 详细设计

3.1 词法分析

在词法分析前首先需要定义词法，然后使用 ANTLR 生成语法分析类，再处理出错信息的输出格式。

ANTLR 是一个开源的词法/语法解析器生成工具，它需要特定的文法输入，其中的词法标识符定义要求以大写字母开头，并使用正则表达式表示词法规则。以下是本程序中使用的词法定义：

```
INT : 'int' ;   // 变量类型只有int
VOID : 'void' ;
IF : 'if' ;
ELSE : 'else' ;
WHILE : 'while' ;
RETURN : 'return' ;
ID : [a-zA-Z][a-zA-Z0-9]*;
NUM : [0-9]+ ;
ASSIGN : '=' ;
PLUS : '+' ;
MINUS : '-' ;
MULTIPLY : '*' ;
DIVIDE : '/' ;
EQUALS : '==' ;
GREATER : '>' ;
GREATER_EQUALS : '>=' ;
LESS : '<' ;
LESS_EQUALS : '<=' ;
NOT_EQUALS : '!=' ;
SEMICOLON : ';' ;
COMMA : ',' ;
COMMENT : '/*' .*? '*/' -> skip ;   // skip表示跳过与之匹配的内容 不生成TOKEN
LINE_COMMENT : '//' ~[\r\n]* -> skip ;
LEFT_PAREN : '(' ;
RIGHT_PAREN : ')' ;
LEFT_BRACE : '{' ;
RIGHT_BRACE : '}' ;
```

```
LEFT_BRACKET : '[' ;
RIGHT_BRACKET : ']' ;
WS : [ \t\r\n]+ -> skip ;
```

生成项目时，ANTLR 将会根据以上的词法定义，生成相应的词法解析器 CMinusMinusLexer（因为此文法的名称为 CMinusMinus）。通过子类 TokenGenerator 重写该解析器的 GetErrorDisplay 函数，即可重新定义词法解析出错时输出的信息。

使用 TokenGenerator 即可生成 CommonTokenStream，其中包含一系列实现了 IToken 接口的对象，可用于输出词法分析结果以及后续的语法分析。

3.2 语法分析

在进行语法分析前，首先要定义能够被 ANTLR 解析的语法，即非终结符的产生式，然后使用 ANTLR 生成语法解析器，再将解析得到的上下文转为 Json 语法树。

产生式可以使用扩展巴斯科范式进行定义，使用先前定义的词法符号作为终结符，而非终极符需要以小写字母开头。以下是本程序中使用的语法定义：

```
// 整个程序由多个声明构成
program : declaration*;
// 声明
declaration : variableDeclaration
            | functionDeclaration
            | arrayDeclaration
            ;
// 变量声明（不可赋初值）
variableDeclaration : INT ID SEMICOLON ;
// 函数声明
functionDeclaration : INT ID LEFT_PAREN parameterList RIGHT_PAREN block
                    | VOID ID LEFT_PAREN parameterList RIGHT_PAREN block;
// 数组声明
arrayDeclaration : INT ID LEFT_BRACKET NUM RIGHT_BRACKET (LEFT_BRACKET NUM
    RIGHT_BRACKET)* SEMICOLON;
// 参数列表（不能为空，只能写void）
parameterList : parameter (COMMA parameter)*
              | VOID
              ;
```

```
// 参数
parameter : INT ID ;
// 语句块（不能省略大括号）
block : LEFT_BRACE (innerDeclaration | statement)* RIGHT_BRACE ;
// 内部声明
innerDeclaration : variableDeclaration
                 | arrayDeclaration
                 ;
// 表达式语句
statement : ifStatement
          | whileStatement
          | returnStatement
          | assignmentStatement
          ;
// 赋值语句
assignmentStatement : ID ASSIGN expression SEMICOLON
                    | array ASSIGN expression SEMICOLON
                    ;
// 返回语句
returnStatement : RETURN expression? SEMICOLON ;
// while语句
whileStatement : WHILE LEFT_PAREN expression RIGHT_PAREN block ;
// if语句
ifStatement : IF LEFT_PAREN expression RIGHT_PAREN block (ELSE block)?;
// 比较表达式
expression : additiveExpression (relop additiveExpression)? ;
// 加减表达式
additiveExpression : multipleExpression (PLUS multipleExpression | MINUS
   multipleExpression)* ;
// 乘除表达式
multipleExpression : factor (MULTIPLY factor | DIVIDE factor)* ;

factor : NUM
       | LEFT_PAREN expression RIGHT_PAREN
       | ID call?
       | array
       ;
// 实参列表
call : LEFT_PAREN argument RIGHT_PAREN ;
```

```
// 实参
argument : expression (COMMA expression)*
         | // 空
         ;
// 数组元素
array : ID LEFT_BRACKET expression RIGHT_BRACKET
      | array LEFT_BRACKET expression RIGHT_BRACKET
      ;
// 比较运算符
relop : LESS
      | LESS_EQUALS
      | GREATER
      | GREATER_EQUALS
      | EQUALS
      | NOT_EQUALS
      ;
```

生成项目时，ANTLR 将会根据以上的语法定义，生成相应的语法解析器 CMinusMinusParser。通过更改其中的 ErrorListener，即可重新定义语法出错时的提示信息。

使用继承 CMinusMinusParser 的 GrammarGenerator 即可生成语法上下文 ProgramContext，其中包含一系列实现了 IParseTree 接口的对象，表示语法树中的节点。使用先序遍历搜索整棵语法树，即可从中得到所有非终结符或终结符，并将其以 Json 格式输出到文件。

3.3 中间代码生成

接下来是本程序实现过程中的重点内容，使用 LLVM 提供的 API 将语法上下文转为 LLVM IR。另外，在转换过程中还需要维护一个多级符号表，确保使用的符号在先前已经被定义。

符号表由 SymbolTable 类表示，其成员定义如下。符号表以栈的形式组织，能够记录任意层级的作用域中的符号，每个符号需要记录其值和类型。

```
1 internal class SymbolTable
2 {
3     private readonly Stack<Dictionary<string, (LLVMValueRef value,
          LLVMTypeRef type)>> m_ScopeStack = new();
4     public SymbolTable()
```

```
    {
        // 初始时就有一个全局符号表
        m_ScopeStack.Push(new Dictionary<string, (LLVMValueRef, LLVMTypeRef)
            >());
    }

    // 进入新作用域
    public void EnterScope();
    // 离开作用域
    public void ExitScope();

    // 添加符号
    public void AddSymbol(string name, LLVMValueRef value, LLVMTypeRef type);

    // 在所有符号表中获取符号
    public LLVMValueRef GetValue(string name);
    // 在最上层的符号表中获取符号
    public LLVMValueRef GetValueInTop(string name);
    /// 在所有符号表中获取符号的类型
    public LLVMTypeRef GetType(string name);
}
```

生成中间代码前，首先需要初始化一个 LLVMModule 来表示生成的中间语言程序，并初始化一个 LLVMBuilder 来管理当前中间代码写入位置与中间代码符号名称。另外，还需要在起始处引入 printf 函数的定义，并定义一些格式字符串，用于后续输出程序执行过程。

在生成中间代码的过程中，需要对每个非终结符判断其应用的是哪一条产生式，然后进行相应的处理。非终结符以上下文类表示，其中包含该非终结符可能使用的产生式中所包含的所有符号。例如赋值语句非终结符 assignmentStatement 的上下文类定义如下：

```
// ParserRuleContext是所有上下文类的基类，其中包含有获取上下文和Token的方法
public class AssignmentStatementContext : ParserRuleContext
{
    // ITerminalNode是所有终结符都继承的接口，其中包含相应的Token和产生该终结
        符的非终结符
    public ITerminalNode ID();
    public ITerminalNode ASSIGN();
    // 其它上下文类的命名同样以Context结尾
```

```
    public ExpressionContext expression();
    public ITerminalNode SEMICOLON();
    public ArrayContext array();
}
```

处理赋值语句的函数如下，其它处理函数命名同样以 Visit 开头。此函数开始时通过判断产生式中是否直接包含 ID，从而判断该赋值语句是为普通变量还是数组赋值，即判断所使用的产生式，然后根据不同的产生式进行不同的处理。在赋值语句的最后添加了 printf 函数，用于在可执行文件中展示此次赋值的过程。

```
private void VisitAssignmentStatement(AssignmentStatementContext context)
{
    LLVMValueRef lhs;
    // 获取普通变量
    if (context.ID() != null)
    {
        string name = context.ID().GetText();
        if (m_SymbolTable.GetValue(name).Handle == IntPtr.Zero)
            throw new Exception($"变量 {name} 未定义");
        lhs = m_SymbolTable.GetValue(name);
    }
    // 获取数组变量
    else
    {
        lhs = VisitArray(context.array());
    }

    // 计算右值
    LLVMValueRef rhs = VisitExpression(context.expression());

    // 生成赋值指令
    PrintF(PrintfType.Assign, rhs);
    m_Builder.BuildStore(rhs, lhs);
}
```

在生成中间代码的过程中，需要进行的操作包括注册符号到符号表、进入/退出作用域、为变量分配存储空间等。在操作 LLVM 时，大部分符号都表现为 LLVMValue 对象，而

类型则表现为 LLVMType 对象。在生成中间代码时，需要使用 LLVMBuilder 来创建指令，该对象能够管理当前创建指令的位置，并可根据需要调整其创建位置（例如在实现分支语句时）。另外，还需要在函数（同样是 LLVMValue 对象）中添加以 LLVMBasicBlock 表示的基本块，用于实现函数的逻辑控制。

生成中间代码后，需要使用 LLVMModule 的 Verify 方法检查生成的代码是否有错误，例如 LLVMValue 的类型是否与各处需要使用的类型相符，可以通过这种方式调试中间代码生成器。

3.4 可执行文件生成

可执行文件需要通过已经构造好的 LLVMModule 来生成，并且需要安装 LLVM 的配套工具 clang 和 lld-link，将其加入环境变量以便主程序调用。

首先将 LLVMModule 以二进制格式输出到文件中，然后使用 clang 加载该文件并生成目标文件，接下来再使用 lld-link 将自行编译得到的目标文件与 C 标准库链接起来（因为使用了 C 标准库中的 printf 函数），即可得到在此平台上可用的可执行文件。

如果需要生成在其他平台上可执行的文件，需要在构造 LLVMModule 时就设置目标平台，否则 clang 会自动识别当前工作的平台，并采用其作为生成的目标文件执行平台。

4 测试结果

4.1 测试用例

以下是测试过程中使用的输入源文件，后续展示的正确运行结果将以此文件内容为准，在进行出错提示的测试时会对其进行少量修改：

```
int program(int a,int b,int c)
{
        int i;
        int j;
        i=a;
        j = b;
        if(a>(b + c))
        {
                j=a+(b*c+1);
        }
        else
        {
                j=c;
        }
        while(i<=100)
        {
                i=i*2 + j;
        }
        return i;
}

int demo(int a)
```

```
{
        a=a+2;
        return a*2;
}
void main(void)
{
        int a[2][2];
        a[0][0]=3;
        a[0][1]=a[0][0]+1;
        a[1][0]=a[0]+a[0][1];
        a[1][1]=program(a[0][0],a[0][1],demo(a[1][0]));
        return;
}
```

4.2 词法分析

运行程序后生成的词法分析结果以 Token 流文本的形式表示，部分内容如下：

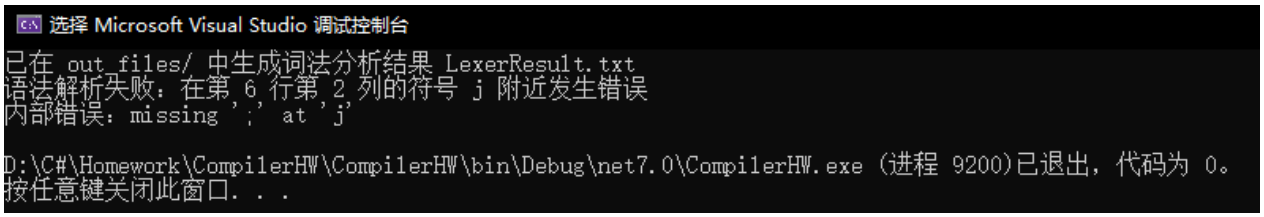
```
第 1  个token: int     , 类型: INT         , 行数: 1  , 列数: 0
第 2  个token: program  , 类型: ID          , 行数: 1  , 列数: 4
第 3  个token: (       , 类型: LEFT_PAREN   , 行数: 1  , 列数: 11
第 4  个token: int     , 类型: INT         , 行数: 1  , 列数: 12
第 5  个token: a       , 类型: ID          , 行数: 1  , 列数: 16
第 6  个token: ,       , 类型: COMMA       , 行数: 1  , 列数: 17
第 7  个token: int     , 类型: INT         , 行数: 1  , 列数: 18
第 8  个token: b       , 类型: ID          , 行数: 1  , 列数: 22
第 9  个token: ,       , 类型: COMMA       , 行数: 1  , 列数: 23
第 10 个token: int     , 类型: INT         , 行数: 1  , 列数: 24
第 11 个token: c       , 类型: ID          , 行数: 1  , 列数: 28
第 12 个token: )       , 类型: RIGHT_PAREN  , 行数: 1  , 列数: 29
第 13 个token: {       , 类型: LEFT_BRACE   , 行数: 2  , 列数: 0
第 14 个token: int     , 类型: INT         , 行数: 3  , 列数: 1
第 15 个token: i       , 类型: ID          , 行数: 3  , 列数: 5
第 16 个token: ;       , 类型: SEMICOLON    , 行数: 3  , 列数: 6
第 17 个token: int     , 类型: INT         , 行数: 4  , 列数: 1
第 18 个token: j       , 类型: ID          , 行数: 4  , 列数: 5
第 19 个token: ;       , 类型: SEMICOLON    , 行数: 4  , 列数: 6
第 20 个token: i       , 类型: ID          , 行数: 5  , 列数: 1
```

当在源文件中加入非法字符（如 @）后，程序会提示无法解析该字符，并终止执行：

4.3 语法分析

运行程序后生成的语法分析结果是 Json 格式的语法树，部分内容如下（原始内容没有换行与缩进，以下是经过 VSCode 优化表现后的结果）：

当在源文件中留下语法错误（如删除必要的分号）后，程序会提示语法分析有误，并终止执行：

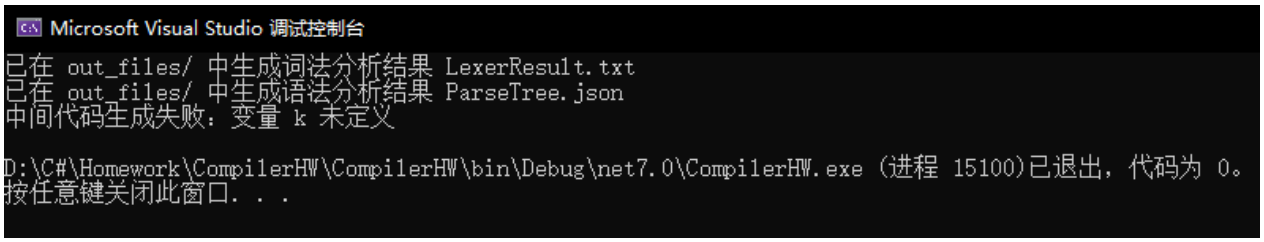


4.4 中间代码生成

运行程序后生成的中间代码为 LLVM IR 的文本形式，部分内容如下（开始的内容是用于 printf 的字符串常量的定义）：

```
; ModuleID = 'CMinusMinus'
source_filename = "CMinusMinus"

@DeclareVarFormat = private unnamed_addr constant [19 x i8] c"\E5\AE\9A\E4\B9\89\E5\8F\98\E9\87\
@DeclareArrayFormat = private unnamed_addr constant [19 x i8] c"\E5\AE\9A\E4\B9\89\E6\95\B0\E7\B
@CallFormat = private unnamed_addr constant [19 x i8] c"\E8\B0\83\E7\94\A8\E5\87\BD\E6\95\B0\EF\
@AssignFormat = private unnamed_addr constant [16 x i8] c"\E8\B5\8B\E5\80\BC\E4\B8\BA\EF\BC\9A%
@ReturnFormat = private unnamed_addr constant [19 x i8] c"\E5\87\BD\E6\95\B0\E8\BF\94\E5\9B\9E\E
@ReturnVoidFormat = private unnamed_addr constant [27 x i8] c"\E5\87\BD\E6\95\B0\E8\BF\94\E5\9B\
@ArgumentFormat = private unnamed_addr constant [13 x i8] c"\E5\8F\82\E6\95\B0\EF\BC\9A%d\0A\(
@string = private unnamed_addr constant [2 x i8] c"i\00", align 1
@string.1 = private unnamed_addr constant [2 x i8] c"j\00", align 1
@string.2 = private unnamed_addr constant [2 x i8] c"a\00", align 1
@string.3 = private unnamed_addr constant [5 x i8] c"demo\00", align 1
@string.4 = private unnamed_addr constant [8 x i8] c"program\00", align 1

declare i32 @printf(...)

define void @global() {
entry:
  ret void
}

define i32 @program(i32 %a, i32 %b, i32 %c) {
entry:
  %arg_a = alloca i32, align 4
  store i32 %a, ptr %arg_a, align 4
  %arg_b = alloca i32, align 4
  store i32 %b, ptr %arg_b, align 4
  %arg_c = alloca i32, align 4
  store i32 %c, ptr %arg_c, align 4
  %i = alloca i32, align 4
  %print = call i32 (...) @printf(ptr @DeclareVarFormat, ptr @string)
  %j = alloca i32, align 4
  %print1 = call i32 (...) @printf(ptr @DeclareVarFormat, ptr @string.1)
  %a_value = load i32, ptr %arg_a, align 4
  %print2 = call i32 (...) @printf(ptr @AssignFormat, i32 %a_value)
```

当在源文件中使用了未定义/离开作用域的符号时，程序会提示中间代码生成失败，并终止执行：



4.5 可执行文件生成

运行程序后生成的可执行文件执行结果如下。因为输入的源程序没有任何输出，所有该程序使用 printf 展示了程序的执行过程（需要将控制台编码改为 UTF-8 才可正确显示内容）：

5 总结

在完成本次课程设计的过程中，我学习了如何使用 ANTLR4 与 LLVM 搭建一个完整的简化版 C 语言编译器，深入理解了构建一门新的上下文相关语言，并将其编译为可执行文件的过程。在实现程序的过程中，我遇到了如下的几点困难：

- 对 ANTLR 工作原理不熟悉：这是我第一次使用 ANTLR 构建词法分析器和语法分析器，尽管最终实现上述功能的代码比较简单，但在实现过程初期，我并没有理解 ANTLR 在构建编译器的过程中扮演的角色。在参考了官方案例（尤其是以 $C\#$ 实现的案例），以及现有可用的文法文件后，我逐渐理解了文法文件、ANTLR 生成代码与我需要编写的代码之间的关系，最终完成了词法分析和语法分析。

- 对 ANTLR 数据结构不熟悉：ANTLR 使用的数据结构，有许多以接口的形式出现，而某些接口又继承了上层接口，使我在研究其数据结构时常常找不到我需要的功能。例如在构造 Json 语法树时，我得到的 IParseTree 对象中并没有直接包含该节点的文本，我需要从它所继承 ITree 中得到其所含内容，并将该内容（object 类型）转为我所需要的类型。这些过程对于初学者而言，常常容易犯错，但在研究 ANTLR 数据结构的同时，我也进一步地理解了 ANTLR 的工作原理。

- 对 LLVM 使用方式不熟悉：这同样是我第一次使用 LLVM 实现中间代码生成与可执行文件生成，而由于我使用 $C\#$ 编码，我不能直接操作 LLVM 对象，而要通过许多名为 LLVMXXXRef 的结构体来间接引用与操作对象。这种间接引用使我无法直接得到对象的类型，也大大增加了调试的难度（我不确定使用 C++ 编码是否能够更易于调试）。因此，我在中间代码生成的部分耗费了不少精力来理解 LLVM 的使用方式，才最终实现了中间代码生成。

经过此次课程设计，我掌握了实现一门新语言的编译的基本技术。尽管我不打算参与到主流编程语言的编译器开发中，但我知道在某些特定项目中，需要定义一门简单的新语言，来帮助开发人员（乃至非程序员）进行开发，或者将这门新语言集成到该项目的编辑器中。我相信我在本次课程设计中学习到的技术，能够在未来对我的工作有所帮助。