编译原理与技术课程设计：Pascal-S 编译器的设计与实现
实验报告

李康童	班建龙	毛子恒	谢澳伦	李俊辉	乔永琦
2019211408	2019211368	2019211397	2019211377	2019211415	2019211372

北京邮电大学 计算机学院（国家示范性软件学院）

日期：2022 年 6 月 1 日

目录

1 需求分析

我们设计并实现了 *s1mple-compiler*，一个 Pascal-S 语言编译器。

1.1 源语言说明

Pascal-S 是 Pascal 语言的真子集，它保留了 Pascal 语言的大部分功能。其文法描述见A节。

1.2 目标语言说明

LLVM 是一种基于静态单一分配 (Static Single Assignment，SSA) 的中间表示 (IR)，它提供类型安全、低级操作、高灵活性以及清晰地表示所有高级语言的能力。它是 LLVM 编译的所有阶段使用的通用代码表示。

LLVM IR 以三种不同的形式出现：作为内存编译器的 IR、作为磁盘上的二进制码表示以及作为人类可读的汇编语言表示。LLVM IR 为高效的编译器转换和分析提供强大的中间表示，同时实现自然的调试和可视化。三种不同形式的 LLVM IR 都是等价的。

LLVM IR 的语法细节见LLVM Language Reference Manual。

1.3 功能描述

s1mple-compiler 以 Pascal-S 语言源程序（.pas 文件）作为输入，生成 LLVM 中间代码和可执行文件。

1.4 实验环境

- Docker version 20.10.14
- Ubuntu 20.04.4 LTS
- ANTLR Parser Generator Version 4.9.3
- Ubuntu LLVM version 13.0.1
- Ubuntu clang version 13.0.1
- cmake version 3.16.3

1.5 工具说明

Docker 为了实现组内快速的开发环境配置，我们构建了Docker镜像，在 Docker 容器内进行项目的构建和运行。利用 VSCode 的Remote - Container插件和C/C++ Extension Pack插件，可以实现在 VSCode 内进入容器环境，以及代码高亮和自动补全等功能。

ANTLR ANTLR是基于 LL(*) 算法实现的语法解析器生成器，使用自上而下（top-down）的递归下降 LL 剖析器方法。我们采用 ANTLR 实现了词法分析和语法分析，并生成了语法树。

LLVM LLVM是一套编译器基础设施项目，包含一系列模块化的编译器组件和工具链。我们利用 LLVM 组件实现了中间代码生成、代码优化、可执行文件生成等功能。

其他工具　使用Git进行版本管理，使用Doxygen生成 API 文档，使用CMake构建项目。

2　总体设计

2.1　技术路线

前端部分采用 ANTLR 语法解析器，通过预定义的 Pascal-S 语法范式对输入的 Pascal 程序进行词法分析、语法分析，生成语法树；然后借由 ANTLR 的 Visitor 模式遍历语法树，调用 LLVM 接口生成 LLVM 中间代码；最后由 clang 优化中间代码、并且生成对应的可执行文件。

2.2　实现方法

我们的代码实现主要由以下三个部分构成：

1. 通过 ANTLR 的语法规则定义 Pascal-S 的语法范式。
2. 编写 C++ 代码，通过 Visitor 模式访问语法树，利用 LLVM IRBuilder 构造 LLVM IR。
3. 利用 shell 脚本简单封装，实现自动化编译过程。

2.3　整体结构

s1mple-compiler 的整体结构如图 1。

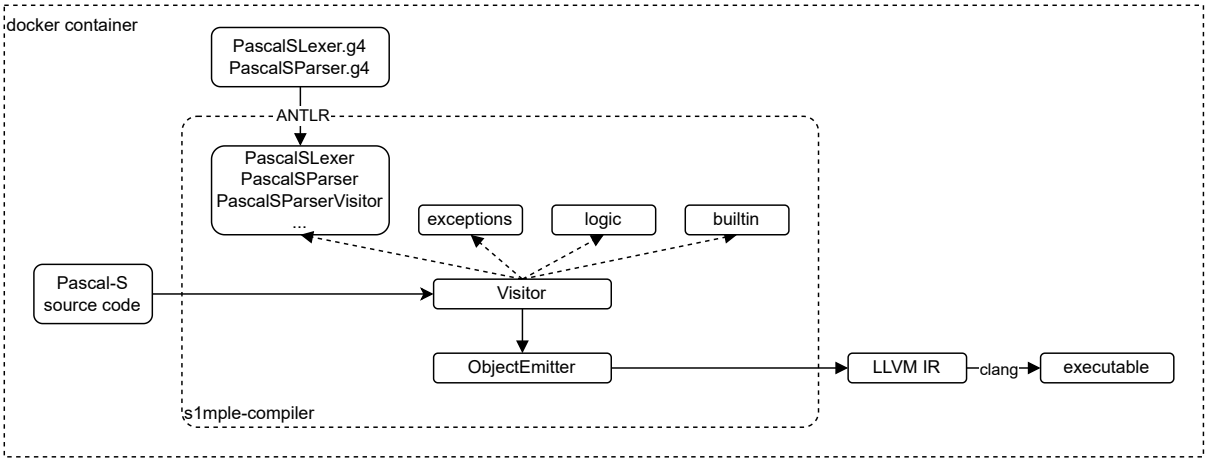

图 1: 整体结构

3　详细设计

3.1　词法、语法分析模块

我们采用 ANTLR 的格式来描述A节的 Pascal-S 的词法和语法，分别见PascalSLexer.g4和PascalSParser.g4。

ANTLR 对词法和语法文件进行分析后，生成 PascalSParserVisitor 库，在项目中链接此库，即可调用接口实现对 Pascal-S 程序的词法和语法分析，并且可以采用 Visitor 方法对生成的语法树进行遍历。

3.2 内置函数和过程

我们实现了`readln`和`writeln`内置过程，可以对标准变量类型进行输入和输出。

每个内置函数和过程的实现有原型构造函数和参数构造函数两部分，在函数和过程调用时，如果在局部和全局作用域中都查找不到变量，则会在内置函数和过程库中查找。如果查找到，则依次调用原型构造和参数构造函数，再生成函数调用。

可以对此模块进行扩展以支持更多的内置函数和过程。详细 API 文档见StandardProcedure.h 文件参考。

3.3 变量作用域

由于 Pascal-S 的嵌套作用域特性，我们实现了Scope 类，即从标识符（字符串）到`llvm::Value*`值的映射。在Visitor 类中有作用域栈实现对变量的查找和访问。

3.4 异常

我们实现了一些异常类，用于调试和报出语法分析过程中发现的错误，这些异常类位于`exceptions`目录下。

3.5 Visitor 模块

采用 ANTLR 的 Visitor 模式访问语法树，并且利用 LLVM 的 IRBuilder 构造中间代码，具体的函数接口见Visitor 类参考，本节对各部分的实现进行简单说明。

3.5.1 变量、常量、类型声明

常量声明 常量的定义如下：

```
constant
    : unsignedNumber          #ConstUnsignedNumber
    | sign unsignedNumber     #ConstSignedNumber
    | identifier              #ConstIdentifier
    | sign identifier         #ConstSignIdentifier
    | string                  #ConstString
    ;
```

访问到如上这些节点时，返回值为对应常量的值，由`auto value = visitConst...(Context);`接收，之后将其作为 Pascal-S 程序的全局常量存储起来，以便后续调用，并且防止常量被修改。

```
1 auto value = visitConst...(Context);
2 module->getOrInsertGlobal(id, value->getType());
3 auto global = module->getNamedGlobal(id);
4 global->setInitializer(value);
5 global->setConstant(true);
```

简单类型变量声明 有如下四种简单类型：

```
simpleType
    : (CHAR | BOOLEAN | INTEGER | REAL)
    ;
```

```
for (const auto &id : idList)
{
     // 对于每个声明的变量，都为其分配空间，并插入到变量表中
     auto addr = builder.CreateAlloca(type, nullptr);
     builder.CreateStore(llvm::UndefValue::get(type), addr);
     scopes.back().setVariable(id, addr);
}
```

同 const 常量声明类似，在访问到这些节点时，在对应的访问函数中得到其具体类型的 LLVM 表示，作为返回值转入到父节点，在 `visitVariableDeclaration()` 函数中，标识符和变量类型汇聚到同一父节点，此时为对应标识符的变量类型分配空间，并且在变量表中插入该变量。

数组变量声明 核心代码如下：

```
auto ranges = visitPeriods(context->periods()); // 数组的下标范围（每两个数字代表着一个数组的维度范围）
int eleNum = 1;
// 计算数组的元素个数
for (auto iter = ranges.begin(); iter != ranges.end(); iter++)
{
    auto v1 = *iter;
    auto v2 = *(++iter);
    eleNum *= (v2 - v1 + 1);
}
```

从 `period` 分支得到数组下标范围的 `vector`，计算得到每一维度数组的元素个数

```
auto type = visitType...(Context);

return llvm::ArrayType::get(type, eleNum);
```

访问数组定义的 `type` 分支，得到其类型，之后根据数组元素的类型和元素个数，创建对应的数组类型，向上返回

在 `visitVariableDeclaration()` 函数中，标识符和变量类型汇聚到同一父节点，此时为对应标识符的变量类型分配空间，并且在变量表中插入该变量。

record 变量声明 record 的定义如下：

```
recordType
    : RECORD recordField? END
    ;

recordField
    : variableDeclaration (SEMI variableDeclaration)* SEMI
    ;
```

可见，record 是包含多个 variableDeclaration 的自定义类型，因此可以使用 LLVM 中的结构体类型将其声明

```
std::vector<llvm::Type *> elements; // 存储多个 variable 类型

for (const auto &varDeclareCtx : context->variableDeclaration())
{
    auto e = visitVariableDeclaration(varDeclareCtx);
    elements.push_back(e);
}
for (auto id : idList)
{
    llvm::StructType *testStruct = llvm::StructType::create(*llvm_context,
    ↪ id);
    testStruct->setBody(elements); // 创建结构体类型代表当前 record 的声明
    return testStruct;
}
```

对于每个variableDeclaration中定义的类型，都将其存储起来，idList是从父节点得到的标识符名的列表；因此，结合类型和标识符名，可以直接在该节点创建该 record 类型，类型名即为传来的标识符名，之后向上返回创建的类型。

在 visitVariableDeclaration() 函数中，标识符和变量类型汇聚到同一父节点，此时为对应标识符的变量类型分配空间，并且在变量表中插入该变量。

类型声明 首先通过 visitTypeSimpleType 获取自定义变量所要表示的类型，然后创建 LLVM 中的 StructType 并使用 CreateAlloca 创建新的变量地址，然后在 Scope 中存入该变量类型。

3.5.2 过程和函数声明

在 visitProcedureAndFunctionDeclarationPart 函数中，获得名为procedureOrFunctionDeclaration的参数，判断：若为 ProOrFuncDecPro，则进入过程声明；若为 ProOrFuncDecFunc，则进入函数声明。

过程声明 在 visitProcedureDeclaration 中分析过程声明，函数实现如下：

1. visitIdentifier 获得过程名；
2. visitFormalParameterList 获得过程引用的参数的类型；
3. 根据得到的过程名、参数类型生成该过程对应的 llvm::function；
4. 为生成的 llvm::function 生成对应的基本块；
5. 为过程的参数设置对应的参数名，并将参数添加到 Scope 中；
6. 访问基本块。

函数声明 在 visitFunctionDeclaration 中分析函数声明，函数实现如下：

1. visitIdentifier 获得函数名；
2. visitSimpleType 获得该函数的返回值类型
3. visitFormalParameterList 获得函数引用的参数的类型；
4. 根据得到的函数名、参数类型生成该过程对应的 llvm::function；
5. 为生成的 llvm::function 生成对应的基本块；
6. 为函数的参数设置对应的参数名，并将参数添加到 Scope 中；
7. 访问基本块。

参数列表访问 在 visitFormalParameterList 函数中分析参数列表：

遍历 formalParameterSection，并判断：

- 若为传值调用，则调用 visitFormalParaSecGroup；
- 若为引用调用，则调用 visitFormalParaSecVarGroup。

在上述两个函数中：

1. visitSimpleType 获得参数类型；
2. visitIdentifierList 获取参数列表；
3. 添加到 paratypes 和 FormalParaIdList 中。

3.5.3 变量访问

在 Pascal-S 编译器中，获取变量地址是访问变量的第一步，我们实现了 visitVariable() 用于处理该过程。

visitVariable() 首先会在局部作用域和全局作用域中查询变量是否存在：存在则获取其（首）地址；不存在则抛出异常 VariableNotFoundException。

接下来，visitVariable() 将会根据上下文判断这个变量的类型：

接下来，程序会根据这个变量的上下文和局部作用域内容，对是否需要进一步处理进行判断：

1. 若变量以数组元素的形式被访问，程序会根据 Visitor 成员 arrayRanges 和数组下标计算目标元素的地址。选项this->arrayIndexFlag 将会在计算数组下标时被启用，此时无论变量是否作为readln的参数，visitFactorVar() 都将直接返回来自 visitVariable() 的地址而非值。详情参考章节 ** 表达式因子 ** 和 ** 函数和过程的调用 **。

2. 若获取的变量类型为函数类型 `llvm::Function`，这说明该变量是一个函数，而实际上我们需要的是该函数的返回变量的地址。此时程序需要在局部作用域中查询 `varName + "ret"`以获得该函数的返回变量的地址。
3. 其他情况直接返回已获取的地址。

值得特别指出的是，大部分情况下 PascalS 编译器访问变量的最终目的是获取变量的值而非地址。因此，只有少数情况下需要直接调用 `visitVariable()`（如赋值语句中对变量的访问）。

3.5.4 表达式

运算符与运算优先级 Pascal-S 中运算符按照运算优先级由低到高有：

1. Expression级：比较运算符`= <> > >= <= <`。
2. SimpleExpression级：算数运算符`+ -`和逻辑运算符`OR`。
3. Term级：算数运算符`* / DIV MOD`和逻辑运算符`AND`。
4. SignedFactor级：负号（MINUS）`-`。
5. Factor级：逻辑运算符`NOT`。

前 3 个级别的运算处理逻辑类似，以`SimpleExpression`级为例：

`SimpleExpression`的语法定义如下：

```
simpleExpression
    : term (additiveoperator term)?
    ;
```

相应的处理流程为：首先程序首先会计算优先级更高的表达式的值：

```
1  if (!context->additiveoperator())
2  {
3      return visitTerm(context->term(0));
4  }
5
6  auto L = visitTerm(context->term(0));
7  auto R = visitTerm(context->term(1));
```

然后根据运算符 `additiveoperator()` 的类型调用相应的运算函数 `visitOpxxx()` 进行计算并返回，或抛出异常：

```
1  if (auto plusContext = dynamic_cast<PascalSParser::OpPlusContext
   ↪  *>(context->additiveoperator()))
2  {
3      return visitOpPlus(plusContext, L, R);
4  }
5  else if (auto minusContext = dynamic_cast<PascalSParser::OpMinusContext
   ↪  *>(context->additiveoperator()))
6  {
```

```
    return visitOpMinus(minusContext, L, R);
}
else if (auto orContext = dynamic_cast<PascalSParser::OpOrContext
↪   *>(context->additiveoperator()))
{
    return visitOpOr(orContext, L, R);
}
else
{
    throw DebugException(NOW_FUNC_NAME + "Undefined Operator!");
}
```

大部分运算函数都为不同类型的操作数匹配了相应的 IRBuilder 接口。以 visitOpPlus() 为例：

```
llvm::Value *Visitor::visitOpPlus(PascalSParser::OpPlusContext *context,
↪   llvm::Value *L, llvm::Value *R)
{
    if (R->getType()->isFloatingPointTy() &&
↪   L->getType()->isFloatingPointTy())
        return builder.CreateFAdd(L, R);

    if (R->getType()->isFloatingPointTy() && L->getType()->isIntegerTy())
    {
        auto L_FP = builder.CreateSIToFP(L,
↪   llvm::Type::getFloatTy(*llvm_context));
        return builder.CreateFAdd(L_FP, R);
    }
    else if (L->getType()->isFloatingPointTy() && R->getType()->isIntegerTy())
    {
        auto R_FP = builder.CreateSIToFP(R,
↪   llvm::Type::getFloatTy(*llvm_context));
        return builder.CreateFAdd(L, R_FP);
    }
    else if (L->getType()->isIntegerTy() && R->getType()->isIntegerTy())
    {
        return builder.CreateAdd(L, R);
    }
    else
        throw DebugException(NOW_FUNC_NAME + "Unsupported Operands Type for
↪   Operator \'+\'");
}
```

逻辑运算函数 visitOpOr() 等少数运算函数无需限制操作数类型：

```
llvm::Value *Visitor::visitOpOr(PascalSParser::OpOrContext *context,
↪   llvm::Value *L, llvm::Value *R)
{
```

```
3       return builder.CreateOr(L, R);
4   }
```

SignedFactor级运算处理逻辑与前三者略有差别：

1. visitSignedFactor() 调用的是不同类型的表达式因子 visitFactorxxx() 而非运算处理函数。
2. 负号-的处理逻辑为：根据是否有 MINUS()，为不同类型的表达式因子的值选择相应的处理方案（乘-1 或按位取反）。

需要注意的是：llvm::Type 中使用位宽为 1 的整型存储bool型数据，因此 visitSignedFactor() 通过位宽度区分表达式因子中不同类型的值。

Factor 级运算符 NOT 请参考下一节。

表达式因子 表达式因子Factor是 Pascal-S 表达式运算中的最小单元，其语法定义为：

```
factor
    : variable                  #FactorVar
    | LPAREN expression RPAREN  #FactorExpr
    | functionDesignator        #FactorFunc
    | unsignedConstant          #FactorUnsConst
    | NOT factor                #FactorNotFact
    | bool_                     #FactorBool
    ;
```

- FactorVar：获取变量的地址或值。调用了 visitVariable()。默认返回变量值，仅当构造readln参数时返回地址：

```
1   llvm::Value *Visitor::visitFactorVar(PascalSParser::FactorVarContext
    ↪  *context)
2   {
3       auto varName = visitIdentifier(context->variable()->identifier(0));
4       auto varAddr = visitVariable(context->variable());
5
6       // 为 readln 构造参数时需要传递地址而非值
7       if (readlnArgFlag == true && arrayIndexFlag == false)
8       {
9           return varAddr;
10      }
11      else
12      {
13          return
    ↪   builder.CreateLoad(varAddr->getType()->getPointerElementType(),
    ↪   varAddr);
14      }
15  }
```

- FactorExpr：计算内部表达式的值。调用了 visitExpression()。
- FactorFunc：调用函数并获取返回值。调用了函数声明 FunctionDesignator。见3.5.5节。
- FactorUnsConst：获取无符号常量值。包括无符号常量字符串、无符号常量整型和无符号常量浮点型。
- FactorNotFact：实现逻辑运算符 NOT。右递归文法，visitFactorNotFact() 如下：

```
1  llvm::Value
   ↪ *Visitor::visitFactorNotFact(PascalSParser::FactorNotFactContext
   ↪ *context)
2  {
3      llvm::Value *value;
4      if (auto factorVarCtx = dynamic_cast<PascalSParser::FactorVarContext
   ↪ *>(context->factor()))
5      {
6          value = visitFactorVar(factorVarCtx);
7      }
8      else if (auto factorExprCtx =
   ↪ dynamic_cast<PascalSParser::FactorExprContext *>(context->factor()))
9      {
10         value = visitFactorExpr(factorExprCtx);
11     }
12     else if (auto factorFuncCtx =
   ↪ dynamic_cast<PascalSParser::FactorFuncContext *>(context->factor()))
13     {
14         value = visitFactorFunc(factorFuncCtx);
15     }
16     else if (auto factorUnsConstCtx =
   ↪ dynamic_cast<PascalSParser::FactorUnsConstContext
   ↪ *>(context->factor()))
17     {
18         value = visitFactorUnsConst(factorUnsConstCtx);
19     }
20     else if (auto factorNotFactCtx =
   ↪ dynamic_cast<PascalSParser::FactorNotFactContext
   ↪ *>(context->factor()))
21     {
22         value = visitFactorNotFact(factorNotFactCtx);
23     }
24     else if (auto factorBoolCtx =
   ↪ dynamic_cast<PascalSParser::FactorBoolContext *>(context->factor()))
25     {
26         value = visitFactorBool(factorBoolCtx);
27     }
28     else
29     {
30         throw DebugException(NOW_FUNC_NAME);
31     }
32
33     if (context->NOT())
34     {
```

```
35          return builder.CreateNot(value);
36      }
37      else
38      {
39          return value;
40      }
41  }
```

- `FactorBool`：获取`bool`类型的常量值。

3.5.5 简单语句

赋值语句 `visitAssignmentStatement()` 实现了变量的赋值功能，它的思路非常简单：

1. 计算右侧表达式的值。
2. 获取左侧变量的地址（调用 `visitVariable()`）。
3. 将值加载至地址对应的空间中。

需要注意的是：由于 Pascal-S 的语法是强类型的，表达式的值的类型和变量的类型必须保持严格一致。

过程调用 在 LLVM 中，无论是过程（Procedure）还是函数（Function）都使用 `llvm::Function` 进行构造和调用。因此我们将过程视为不产生返回值的特殊函数进行处理。

函数和过程的调用分 3 步完成：

1. 获取原型：程序首先会根据函数名在作用域中查找 `llvm::Function` 类型的变量，若未找到则抛出异常：

```
1  auto function =
   ↪  llvm::dyn_cast_or_null<llvm::Function>(getVariable(funcName))
```

2. 构造参数表：然后程序会通过 `visitParameterList()` 遍历构造函数参数表：

```
1  auto paraList = visitParameterList(context->parameterList());
2  llvm::ArrayRef<llvm::Value *> argsRef(paraList);
```

3. 调用函数：最后程序会通过 `IRBuilder` 创建一个函数调用指令：

```
1  builder.CreateCall(procedure, argsRef);
```

函数和过程的调用分别实现于：`visitFunctionDesignator()` 和 `visitProcedureStatement()`。

需要特别说明的是：在为内置函数和过程（参见3.2节）构造参数表时，某些特别的处理将被启用：

- `this->readlnArgFlag = True`：调用 Pascal-S 输入`readln`时启用，此项将使 `visitFactorVar()` 返回变量（参数）的地址而非值，由此额外为`readln`构造一组变量地址参数并插入至正常参数中。这是由函数原型 `scanf` 决定的。

- `changeFP = True`：调用 Pascal-S 输入`readln`或 PascalS 输出`writeln`时启用，此项将使 `visitParameterList()` 在遍历参数表时将所有的 `llvm::FloatTy` 转换为 `llvm::DoubleTy`。这是由函数原型 `scanf` 和 `printf` 决定的。

3.5.6 复合语句

分支 在 `visitIfStatement` 函数中实现分支语句的构造，具体过程如下：

1. `visitExpression` 获取条件判断值；
2. 创建 `then`、`end` 基本块，并根据 `statement().size()` 的值判断是否生成`else`基本块，若为 2 则生成，为 1 则不生成，用于`if`的控制流；
3. 若 `statement().size()` 为 1，则根据条件判断值判断执行 `then` 还是 `end`；
4. 访问 `then` 基本块的内容；
5. 跳转至 `end` 块；
6. 若 `statement().size()` 为 2，则根据条件判断值判断执行 `then` 还是`else`；
7. 访问`else`块的内容；
8. 跳转至 `end` 块；
9. 设置 `blockInsertpoint` 为 `end` 块。

循环 以`for`循环为例，其定义如下：

```
forStatement
    : FOR identifier ASSIGN forList DO statement
    ;


forList
    : initialValue (TO | DOWNTO) finalValue
    ;
```

其中`forList`分支可以解析得到循环变量的初始值和结束值；`statement`就是`for`循环中的语句部分；

循环的实现可以分为三个代码块：

```
1 // 创建循环的基本块
2 // 判断循环是否完成的块
3 auto while_count = llvm::BasicBlock::Create(*llvm_context, "while_count",
  ↪  function, 0);
4 // 循环体代码块
5 llvm::BasicBlock *while_body = llvm::BasicBlock::Create(*llvm_context,
  ↪  "while_body", function, 0);
6 // 结束循环后的块
7 llvm::BasicBlock *while_end = llvm::BasicBlock::Create(*llvm_context,
  ↪  "while_end", function, 0);
```

首先创建循环变量，为其分配空间：

```
auto con_1 = llvm::ConstantInt::get(llvm::Type::getInt32Ty(*llvm_context), 1);
auto addr = builder.CreateAlloca(llvm::Type::getInt32Ty(*llvm_context),
    ↪ nullptr);
builder.CreateStore(initial, addr);
```

之后从原本的代码块调转到判断循环是否完成的块，取出循环变量的值，判断是否满足循环退出条件，根据判断结果跳转到不同的基本块：

```
builder.CreateBr(while_count); // 跳转语句
builder.SetInsertPoint(while_count); // 为基本块添加语句
auto tmp_i = builder.CreateLoad(llvm::Type::getInt32Ty(*llvm_context), addr);
auto cmp = builder.CreateICmpSLE(tmp_i, final);
builder.CreateCondBr(cmp, while_body, while_end); // 比较，跳转
```

在循环体中，每次执行完语句后，循环变量的值要加一，之后跳转到循环判断块，重复执行判断：

```
builder.SetInsertPoint(while_body);
...
// 循环中的循环变量，每次循环加一
auto i = builder.CreateLoad(llvm::IntegerType::getInt32Ty(*llvm_context),
    ↪ addr);
auto tmp = builder.CreateAdd(i, con_1);
builder.CreateStore(tmp, addr);

builder.CreateBr(while_count);
```

最后是循环结束块，在此语句之后的语句，都将成为循环结束后执行的语句：

```
builder.SetInsertPoint(while_end);
```

其他两种循环：`repeat`，`while`的结构类似。特别的，`repeat`循环需要注意跳转逻辑的改变，与其他循环不同，当满足循环条件时应该结束循环，不满足时应该继续执行循环体。在循环过程中需要注意循环条件变量的获取时机应该在 body 部分，以此来避免循环条件不能及时改变导致的跳转错误情况的出现。

3.6 中间代码生成

ObjectEmitter 类获取当前目标机器的信息，并且生成二进制格式的 LLVM IR 文件。

3.7 代码优化

我们对中间代码使用 clang 的O3级别的优化，我们选取了部分优化选项解释其意义：

`-instcombine` 冗余指令合并。

实现基本块内部的优化，比如将下面的内容：

```
%Y = add i32 %X , 1
%Z = add i32 %Y , 1
```

优化为：

```
%Z = add i32 %X , 2
```

还包括：

1. 如果二元运算符有一个常量操作数，则将其移至右侧。
2. 将具有常量操作数的位运算符分组，以便首先执行移位，然后是或运算，然后是与运算，然后是异或运算。
3. 如果可能，把比较指令从 <、>、≤ 或 ≥ 转换为 = 或 ≠。
4. 将布尔值的 cmp 指令都将替换为逻辑运算。
5. 将`add X, X`替换为`shl X, 1`。
6. 将为 2 的幂次的乘法转换为移位。

此过程还可以简化对特定常用的函数调用（例如运行时库函数）的调用。例如，在 `main()` 函数中发生的调用 `exit(3)` 可以简单地转换为`return 3`。

`-simplifycfg` 简化 CFG。

执行死代码消除和基本块合并。具体来说：

1. 删除没有前置块的基本块。
2. 如果该基本块没有后续块，且其前置块只有一个后续块，则将该基本块合并到其前置块中。
3. 消除具有单个前置块的基本块的 PHI 节点。
4. 消除仅包含无条件分支的基本块。

此过程还可以简化对特定常用的函数调用（例如运行时库函数）的调用。例如，在 `main()` 函数中发生的调用 `exit(3)` 可以简单地转换为`return 3`。

`-strip-dead-prototypes` 删除未使用的函数原型。

在输入模块中循环遍历所有函数，查找死声明并删除它们。死声明是指没有被实现的函数的声明或是未使用的库函数的声明。

`-mem2reg` 将内存引用优化为为寄存器引用。它优化了仅具有加载和存储功能的`alloca`指令。通过使用 PHI 节点来优化`alloca`，然后以深度优先的方法遍历函数以适当地重写加载和存储。

`-functionattrs` 一个简单的过程间传递，它遍历调用图，寻找不访问或只读取非本地内存的函数，并将它们标记为 readnone 或 readonly。

`-globaldce` 消除程序中无法访问的内部全局变量。它搜索已知存在的全局变量。在找到所需的所有全局变量后，它会删除剩余的全局变量。

`-jump-threading` 通过基本块尝试查找正在运行的不同的控制流。此传递将查看具有多个前置任务和多个后续任务的块。如果可以证明块的一个或多个前置任务总是导致程序跳转到其中一个后续任务，此优化将通过复制此块的内容从而将控制流从前置任务转移到后续任务。

例如：

```
if()
{
    ...
    X = 4;
}
if(X < 3)
{
```

在这种情况下，第一个`if`末尾的无条件分支可以重新定向到第二个`if`的右侧。

`-lcssa` 此优化通过在循环边界上处于活动状态的所有值的循环末尾放置 PHI 节点来转换循环。

例如，它将代码：

```
for(...)
    if(c)
        x1 = ...
    else
        x2 = ...
    x3 = phi(x1, x2)
    ...
... = x3 + 4
```

转换为以下代码：

```
for(...)
    if(c)
        x1 = ...
    else
        x2 = ...
    x3 = phi(x1, x2)
x4 = phi(x3)
... = x4 + 4
```

这仍然是有效的LLVM代码,多余的PHI节点纯粹是冗余的,将被简单地通过InstCombine消除。这种转换的主要好处是它使许多其他循环优化更简单。

-indvars　规范化迭代变量。

-indvars将迭代变量（以及源于它们的计算）分析并转换为便于后续操作（分析与转换）的简单形式。

-indvars对于每一个具有可识别迭代变量的循环都将应用如下规则:

1. 循环被转换为具有单个规范化迭代变量（Single Canonical Induction），该迭代变量从 0 开始且步长为 1。2. 这个规范化迭代变量保证是循环头部块（Loop Header Block）中的第一个 PHI 结点。3. 任何指针算术递归（Pointer Arithmetic Recurrences）都将被提取以使用数组下标。

如果循环的行程计数（Trip Count）是可计算的，则-indvars会额外应用如下更改:

循环的退出条件将被规范化以对比循环的归纳值（Induction Value）和退出值。

例如循环:

```
for(i = 7; i*i < 1000; ++i)
```

将被转换为:

```
for(i = 0; i != 25; ++i)
```

对于迭代变量派生表达式（Expression Derived From The Induction Variable）的任何循环外部使用（Use Outside Of The Loop），-indvars都将改变它们以计算循环外派生值，从而消除对迭代变量的退出值的依赖性。如果循环的唯一目的是计算某些派生表达式的退出值，则-indvars将使循环失效。

-loop-unroll-and-jam　展开和阻塞循环。

实现了简单的经典循环展开、阻塞优化过程。

例如循环:

```
for i.. i+= 1
    for j..
      code(i, j)
```

将被转换为:

```
for i.. i+= 4
    for j..
        code(i, j)
        code(i+1, j)
```

```
5           code(i+2, j)
6           code(i+3, j)
7           remainder loop
```

这可以看作是展开外环并将内环融合（Fusing）为一个。当变量或负载可以在新的内部循环中共享时，此优化可以显著提高性能。`-loop-unroll-and-jam`使用依赖分析（Dependence Analysis）来证明这些转换是安全的。

`-loop-unswitch` 展开和阻塞循环。

`-loop-unswitch`将那些包含循环不变条件分支（Branches On Loop-Invariant Conditions）的循环转换为多个循环。

例如循环：

```
1  for (...)
2      A
3      if (lic)
4          B
5      C
```

将被转换为：

```
1  if (lic)
2      for (...)
3          A; B; C
4  else
5      for (...)
6          A; C
```

`-loop-unswitch`会指数级地增加代码数（每次取消切换循环时将其加倍），因此我们仅在结果代码小于阈值时才取消切换。

`-loop-unswitch`运行之前推荐使用`-licm`将循环不变条件提出循环，使得取消切换与否更加显而易见。

4 用户指南

4.1 构建说明

4.1.1 构建 Docker 镜像

有以下三种方式获取 Docker 镜像：

通过 GitHub Packages 下载 执行如下命令：

```
docker pull ghcr.io/xqmmcqs/s1mple-compiler:0.11
docker tag ghcr.io/xqmmcqs/s1mple-compiler:0.11 s1mple-compiler
```

通过归档文件加载 获取到`s1mple-compiler.tar`，执行如下命令：

```
docker load -i s1mple-compiler.tar
```

自行构建镜像 在项目目录下执行如下命令：

```
docker build . -t s1mple-compiler
```

4.1.2 创建 Docker 容器

启动 Docker，在项目目录下执行：

```
./dev.sh
```

以创建 Docker 容器，容器将在推出后自动删除。

4.1.3 构建项目

在 Docker 环境内执行：

```
./build.sh
```

来构建项目。

4.2 使用说明

执行：

```
./compile.sh ./test.pas
```

来编译`test.pas`文件，若编译成功，则会生成名为`test`的可执行文件和名为`test.bc`的中间代码。

若有调试需求，可以执行：

```
./disas.sh ./test.bc
```

来将`test.bc`转化为可读的汇编文件，文件名为`test.ll`。

5 调试总结

5.1 测试结果

5.1.1 测试集 1

测试功能 调用标准过程（包括readln和writeln两个标准过程）的参数构造器和原型。

测试程序 见test_stdPro.pas。

```
1  Program test_stdPro;
2
3  Var
4    intVar: integer;
5    realVar: real;
6    a: array[-4..4, -4..4] Of real;
7  Begin
8    {test readln}
9    readln(intVar, realVar, a[-2, -3]);
10   {test writeln}
11   writeln(intVar);
12   writeln(realVar);
13   writeln(a[-2, -3]);
14 End.
```

测试结果 中间代码见test_stdPro.ll。

```
1 2.34 3.28345
1
2.340000
3.283450
```

5.1.2 测试集 2

测试功能 为函数和过程传入参数并调用。

测试程序 见test_funcCall.pas。

```
1  Program test_funcCall;
2
3  Function testFunc1(intVarTem:integer): integer;
4  Begin
5    writeln(intVarTem+1);
6  End;
7
8  Function testFunc2: integer;
```

```
Begin
  writeln(3.14159);
End;

Procedure testPro1(intVarTem:integer);
Begin
  writeln(intVarTem-1);
End;

Procedure testPro2;
Begin
  writeln(114514);
End;

Begin
  testFunc1(2022);
  testFunc2;
  testPro1(2022);
  testPro2;
End.
```

测试结果 中间代码见test_funcCall.ll。

```
2023
3.141590
2021
114514
```

5.1.3 测试集 3

测试功能 赋值语句。

测试程序 见test_assign.pas。

```
Program test_assign;

Const conInt = 27;

Const conReal = 3.14;

Var
  varInt1, varInt2 : integer;
  varReal1, varReal2 : real;
  a: array[-4..4, -4..4] Of real;

Begin
  varInt1 := conInt * conInt;
```

```
14    varInt2 := 2 * conInt;
15    varReal1 := conReal * varInt1;
16    varReal2 := conReal * varInt2;
17    a[-3, -2] := varReal1 * varReal2;
18    writeln(varInt1);
19    writeln(varInt2);
20    writeln(varReal1);
21    writeln(varReal2);
22    writeln(a[-3, -2]);
23  End.
```

测试结果 中间代码见test_assign.ll。

```
729
54
2289.060059
169.560013
388133.062500
```

5.1.4 测试集 4

测试功能 比较运算符（类型：整型与浮点型）。

测试程序 见test_cmp.pas。

```
1   Program test_cmp;
2
3   Var
4     int1,int2: integer;
5     real1,real2: real;
6   Begin
7     int1 := 5;
8     int2 := 6;
9     writeln(int1=int2);
10    writeln(int1<int2);
11    writeln(int1<=int2);
12    writeln(int1<=int1);
13    writeln(int1>=int2);
14    writeln(int2>=int2);
15    writeln(int1>int2);
16
17    real1 := 3.141592;
18    real2 := real1-1;
19    writeln(real1=real2);
20    writeln(real1<real2);
21    writeln(real1<=real2);
22    writeln(real1<=real1);
```

```
23    writeln(real1>=real2);
24    writeln(real2>=real2);
25    writeln(real1>real2);
26
27    writeln(int1=real2);
28    writeln(int1<real2);
29    writeln(int1<=real2);
30    writeln(int1<=real1);
31    writeln(int1>=real2);
32    writeln(int2>=real2);
33    writeln(int1>real2);
34  End.
```

测试结果 中间代码见test_cmp.ll。

```
0
1
1
1
0
1
0
0
0
0
1
1
1
1
0
0
0
0
1
1
1
```

5.1.5 测试集 5

测试功能 整型与浮点型值的四则运算。

测试程序 见test_calculate.pas。

```
1  Program test_calculate;
2
3  Const conInt = 27;
4
```

```
5  Const conReal = 3.14;
6
7  Var
8    varInt1, varInt2 : integer;
9    varReal1, varReal2 , divResult: real;
10   varBool1, varBool2 : boolean;
11
12 Begin
13   varInt1 := conInt * conInt;
14   varInt2 := conInt + conInt;
15   varReal1 := conReal * varInt1;
16   varReal2 := conReal * varInt2;
17   divResult := ((varInt1 + conReal) * conReal) / ((varInt2 / varReal1) -
18                varReal2);
19   writeln(varInt1);
20   writeln(varInt2);
21   writeln(varReal1);
22   writeln(varReal2);
23   writeln(divResult);
24 End.
```

测试结果 中间代码见test_calculate.ll。

```
729
54
2289.060059
169.560013
-13.560034
```

5.1.6 测试集 6

测试功能 for循环。

测试程序 见test_for.pas。

```
1  Program test_for;
2
3  Var n,i: integer;
4  Begin
5    For i:=5 Downto 1 Do
6      writeln(i);
7    For i:=1 To 5 Do
8      writeln(i);
9  End.
```

测试结果 中间代码见test_for.ll。

```
5
4
3
2
1
1
2
3
4
5
```

5.1.7 测试集 7

测试功能 `while`循环。

测试程序 见test_while.pas。

```
1  Program test_while;
2
3  Var
4    a: integer;
5  Begin
6    a := 10;
7    While  a < 20  Do
8      Begin
9        writeln(a);
10       a := a + 5;
11     End;
12 End.
```

测试结果 中间代码见test_while.ll。

```
10
15
```

5.1.8 测试集 8

测试功能 `repeat`循环。

测试程序 见test_repeat.pas。

```
1  Program test_repeat;
2
```

```
3  Var c,b: integer;
4  Begin
5    c := 1;
6    b := 1;
7    Repeat
8      c := c+1;
9      b := b*2;
10     writeln(114451);
11   Until c>5;
12   writeln(b);
13 End.
```

测试结果 中间代码见test_repeat.ll。

```
114451
114451
114451
114451
114451
32
```

5.1.9 测试集 9

测试功能 以bool作为循环变量的循环。

测试程序 见test_boolLoop.pas。

```
1  Program test_boolLoop;
2
3  Var c,b: integer;
4  Begin
5    c := 1;
6    b := 1;
7    Repeat
8      c := c+1;
9      b := b*2;
10     writeln(114451);
11   Until TRUE;
12   While FALSE Do
13     writeln(114415);
14   writeln(b);
15 End.
```

测试结果 中间代码见test_boolLoop.ll。

```
114451
2
```

5.1.10 测试集 10

测试功能 无限循环。

测试程序 见test_infiLoop.pas。

```
1  Program test_infiLoop;
2
3  Var c,b: integer;
4  Begin
5    While TRUE Do
6      writeln(114415);
7    writeln(b);
8  End.
```

测试结果 中间代码见test_infiLoop.ll。

```
114415
114415
114415
114415
114415
114415
...
```

5.1.11 测试集 11

测试功能 if then else语句。

测试程序 见test_ifThenElse.pas。

```
1  Program test_ifThenElse;
2
3  Var b: integer;
4  Function testfun(n:integer): integer;
5  Begin
6    {一个实现斐波那契数列的函数}
7    If n>2 Then testfun := testfun(n-1)+testfun(n-2)
8    Else testfun := 1;
9  End;
10 Begin
```

```
11    b := 10;
12    b := testfun(b);
13    writeln(b);
14  End.
```

测试结果 中间代码见test_ifThenElse.ll。

输出斐波那契数列第十项。

```
55
```

5.1.12 测试集 12

测试功能 if then语句。

测试程序 见test_ifThen.pas。

```
1   Program test_ifThen;
2
3   Var a: integer;
4
5   Var b: real;
6
7   Var c: boolean;
8   Begin
9     a := 1;
10    b := 1.5;
11    c := true;
12    If c Then writeln(111);
13    If a>b Then writeln(222);
14    writeln(333)
15  End.
```

测试结果 中间代码见test_ifThen.ll。

```
111
333
```

5.1.13 测试集 13

测试功能 各种变量定义。

测试程序 见test_varDec.pas。

```
Program test_varDec;

Type m = integer;

Const
    c0 = '123';
    c1 = 1.12345;
    c2 = 3;
    c3 = -c2;

Var
    n: integer;
    a: array[-1..4, -1..4] Of integer;
    r: real;
    s: char;
    b: boolean;

Var Book1, Book2: Record
    title: array [1..50] Of char;
    author: array [1..50] Of char;
    subject: array [1..100] Of char;
    book_id: integer;
End;
Begin

End.
```

测试结果 中间代码见test_varDec.ll。

5.1.14 测试集 14

测试功能 快速排序。

测试程序 见qsort.pas。

```
Program qsort;

Var
  n,i: integer;
  a: array[0..10000] Of integer;

Procedure kp(l,r:integer);

Var
  i,j,x: integer;
Begin
```

```
12    If l<r Then
13      Begin
14        i := l;
15        j := r;
16        x := a[i];
17        While i<j Do
18          Begin
19            While (i<j) And (a[j]>=x) Do
20              j := j-1;
21            a[i] := a[j];
22            While (i<j) And (a[i]<=x) Do
23              i := i+1;
24            a[j] := a[i];
25          End;
26        a[i] := x;
27        kp(l,i-1);
28        kp(i+1,r);
29      End;
30  End;
31
32  Begin
33    readln(n);
34    For i:=0 To n-1 Do
35      readln(a[i]);
36    kp(0,n-1);
37    For i:=0 To n-1 Do
38      writeln(a[i]);
39  End.
```

测试结果 中间代码见qsort.ll。

```
10
5 8 7 6 2 4 9 0 3 1
0
1
2
3
4
5
6
7
8
9
```

5.2 测试中遇到的问题和解决方案

5.2.1 Visitor 模式中类的继承关系

```
type_
    : simpleType      #TypeSimpleType
    | structuredType  #TypeStructuredType
    ;
```

对于上述的 g4 语法，ANTLR 生成代码时，将会把 TypeSimpleType 和 TypeStructuredType 作为 type_ 的子类，这时候，如果要获得子类的 context，可以从父类的 context 直接进行dynamic_cast操作，如下：

```
if (auto ...Context = dynamic_cast<PascalSParser::...Context *>(...Context))
```

由于有两个子类，所以要进行判断，转换类型符合才能成功进行转换；

从语法树父节点的 context 得到子节点的 context，要么使用dynamic_cast操作，要么调用父节点的类方法中对应的函数，返回值为子节点的 context，这样重复，我们就能对于语法树中的每一个节点编写相应的 LLVM 方法进行处理，生成对应的 LLVM IR；

5.2.2 变量声明部分遇到的问题

常量类型的创建问题 查阅资料得知，LLVM 中有创建全局变量的对应方法，并且可以设置全局变量为 constant，不能被修改；所以，我们的解决思路是，创建常量为全局变量，设置为 constant，保证其不能被修改，同时，把常量写入符号表中，通过符号表来控制常量的作用域；如下：

```
auto value = visitConstIdentifier(constIdentifierContext); // 获得常量的值
module->getOrInsertGlobal(id, value->getType()); // 创建常量
auto global = module->getNamedGlobal(id);
global->setInitializer(value); // 初始化常量
global->setConstant(true); // 设定为常量，不可更改
scopes.back().setVariable(id, global); // 写入符号表中
```

数组类型声明的下标问题 在数组声明时，得到的数组下标范围是 LLVM 表示，如何将其转换为int类型方便计算数组分配空间的大小?

查阅资料得知，LLVM::constantInt 类型可以转换为int类型，方式如下：

```
if (llvm::ConstantInt *CI = llvm::dyn_cast<llvm::ConstantInt>(value))
{
    constIntValue1 = CI->getSExtValue();
}
```

由于 Pascal-S 的语法中规定，数组声明时，下标的表示要么是数字整型常量，要么是之前以及声明过的，用标识符表示的整型常量，所以我们可以先判断将其转换 `llvm::ConstantInt` 类型，如果失败，证明这不是整型或者无法转换为整型；成功后我们直接调用 `getSExtValue()` 函数就可以得到对应的`int`值；

类型定义的问题 在`type`类型定义部分最初采取了和`var`声明相似的办法，首先 `CreateStore` 生成局部变量，然后对于这一类型的变量，为其分配空间，并在符号表中插入其对应的标识符和地址

我们发现自己对于 Pascal-S 语法中的`type`，声明理解错误，通过查阅资料，Pascal-S 语法支持自定义类型，可以通过`type`关键字将某些自定义标识符确立为对应的 Pascal 变量类型，因此需要存入变量表的并非其值，而是一个类型。

5.2.3 变量访问部分遇到的问题

变量表的查询顺序问题 函数无法访问全局数组：LLVM 报错：Invalid Record。

对于全局变量的处理策略发生变化后响应查询策略未及时更新。

原先将全局变量作为 `main` 函数的局部变量压入变量表 `scopes` 的最底层，后来利用 `llvm::global` 体系将全局变量移入 `model`，而对应原先的变量查询逻辑在 `visitFactorVar`，数组的解析逻辑却在其调用函数 `visitVariable` 中，这导致无法找到全局数组或变量未声明异常无法正确抛出。

在 `visitVariable` 中检查变量是否声明，在 `visitFactorVar` 中抛出相应异常。

数组类型变量的访问、赋值问题 无法使用变量作为数组的下标：抛出异常`NotImplementedException`。

找不到提取 `llvm::Inst` 类型中整数值的方法，无法正确地根据下标计算目标元素偏移量。

最开始测试数组类型的变量时，误以为整数值的提取方法适用于所有 `llvm::Value`，于是直接照搬声明中对数组范围值的提取方法(`llvm::value(数组下标)->uint_64(计算)->llvm::value(偏移量)`)。实际上声明中的方法仅仅适用于 `llvm::ConstInt`，不适用于变量 `llvm::Inst`（获取该值的 IR 语句类型）。

跳过提取环节，利用 IR 指令直接计算 `llvm::value` 类型偏移量。

5.2.4 算术表达式部分遇到的问题

整型与浮点型数据的比较、计算问题 整型与浮点型数值无法共同参与运算：LLVM 报错：Error Type/Type Mismatched。

在 `visitOpxx` 中直接默认类型相同进行处理，忽略了强类型体系中无法自动转换类型。

LLVM 是一个强类型体系，其中每一个 value 都具有 type 属性；这不同于编程语言的基本类型，无法也不应该自动转换；而 `IRBuilder.CreateOpxx` 要求操作数必须类型相同，因此产生了错误。

根据实际需要，在 `visitOpxx` 中添加了整型和浮点型复合运算处理逻辑（有浮点型则转换为浮点型）。

visitSignedFactor 添加符号问题 无法为变量声明、赋予一个负的浮点值：LLVM 报错：Error Type/Type Mismatched。

在 visitSignedFactor 中处理 context->MINUS() 时，对不同类型的值进行相同的处理：CreateMul(-1, value)。

visitSignedFactor 中需要根据 MINUS() 确定 Factorxx 值的正负号，必须对不同类型值做不同处理。

在 visitSignedFactor 中添加不同类型的负号处理逻辑（FP/32Int/1Int）。

5.2.5 过程调用部分遇到的问题

readln 输入参数构造问题 readln无法正常调用：报错 Segmentation Fault。

readln构造参数时需要传递参数的地址而非值，visitFactorVar 之前只传递参数的值而非地址。

添加了 readlnArgFlag 和 arrayIndexFlag，使得 visitFactorVar 在解析readln参数时能正确地返回地址或值。

浮点值的输入输出问题 无法正常读入或写出浮点值：报错 Segmentation Fault 或抛出异常 NotImplementedExce

llvm::FloatingPointTy 无法直接被 C++ 输入输出 (scanf/printf) 占位符%f 匹配，需要转换为 llvm::DoubleTy 后参与输入输出参数构造。

在 visitParameterList 中添加一个bool changeFP 参数，保证 llvm::FloatingPointTy 可以被转换为 llvm::DoubleTy。

5.2.6 函数和过程声明部分遇到的问题

函数返回值问题 因为 Pascal-S 中的返回值是一个赋值语句，将一个值赋值给与function的标识符同名的一个变量，于是选择在 visitFunctionDeclaration 这一函数中遍历完函数体后（即执行完 visitBlock 函数），再从变量表中查找出返回值，将其返回。

当时编写这个地方的时候，执行 CreateAlloca 为返回值分配地址，以及 CreateLoad 取出地址中的值的时候，程序运行出现 Invalid Instruction with no BB 报错。

执行上述语句时没有创建基本块（最开始以为函数声明不需要基本块，只有函数体内才需要基本块，事实证明这种想法是错误的），于是在执行 CreateAlloca 语句之前，创建一个函数声明所对应的基本块便解决了该问题。

此时又面临了一个问题，就是返回值变量的标识符与函数名相同，且两者都需要存储在变量表（identifier->value）中，但是同一个 identifier 无法对应两个值，为了解决此问题选择将返回值在变量表中的 identifier 加上一个"ret" 的后缀以作区分，即 identifier->(llvm *value)function，identifier+"ret"->(llvm *value)ret_vlaue，同时对 visitVariable 函数进行修改，因为该函数只用于取出传入的 identifier 对应的变量的值的地址，而调用函数时所要根据 identifier 取 (llvm *value)function 使用的时另外的函数，所以可以在 visitVariable 函数中进行判断，如果当前 identifier 对应的 value 是一个function类型，

那么就将当前的 identifier 转换成返回值对应的 identifier，即 identifier+"ret"，代码如下：

```
if (auto func = llvm::dyn_cast_or_null<llvm::Function>(addr))
{
    addr = getVariable(varName + "ret");
}
```

5.2.7 分支部分遇到的问题

分支跳转顺序混乱的问题 遇到分支的入口块，then块，else块与end块中对应内容的顺序混乱，以及对应的 preds 错误的问题。

```
ifStatement
    : IF expression THEN statement (: ELSE statement)?
    ;
```

if需要根据 expression 返回的 value 来进行跳转，在 expression 中，涉及到比较运算的地方所采用的代码均为 CreateFCmpXXX(L, R)，即创建的都是浮点数之间的比较，当 Pascal 代码中的 expression 为整数间的比较时，便会出现错误，产生上述的结果，在此处增加了关于比较运算的类型的判断后，便解决了上述 bug。

else 分支的基本块问题 当 Pascal-S 的分支语句中没有else部分时，此时只能声明一个else的基本块，但是不能直接对其实例化，需要根据判断（判断是否有else部分）结果来决定是否实例化。如果直接实例化，创建了一个else的 BasicBlock，却没有使用它，便会报错，即创建了的 BasicBlock 就必须使用，否则报错。

基本块创建缺少参数的问题 当测试代码出现if中嵌套if，while，for等模块时，程序报段错误，但是对于普通语句却是正常运行，起初以为是变量作用域中出现问题，导致获取变量的地址出现了问题，获取到了空指针，或是不正确的指针，后来反复验证确定在嵌套的if，while等语句中获取的变量地址无误。后来发现在if中嵌套的 StructuredStatement，均需要进行 createBasicBlock，根据 LLVM 的基本组成之间的关系可知，每一个 BasicBlock 都有其对应的function，然而在 visitStructuredStatement 的时候并没有将当前function作为参数传入，导致其后面部分在 createBasicBlock 时，function 为空指针，从而出现段错误。更改 visitStructuredStatement 的函数声明，为其增添 llvm::Function *function 参数后，程序即可正常运行，如图 2所示。

5.2.8 循环部分遇到的问题

循环的实现 对于循环语句，我们将其分成 3 个基本块，count 块，body 块和 end 块，在 count 块进行循环是否结束的判断，在 body 块是循环体，end 块是循环结束后跳转到的块，表示循环语句之后的语句；

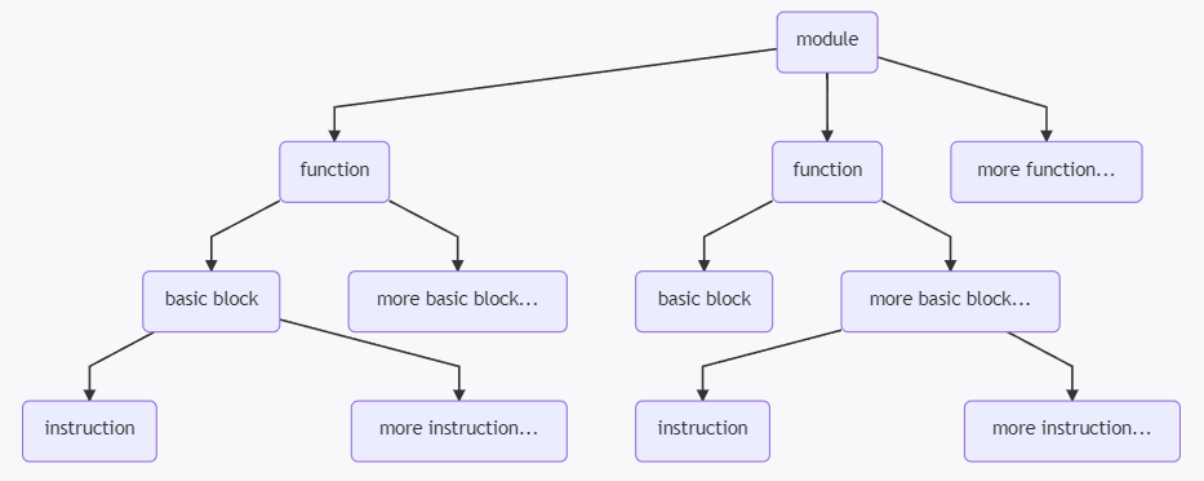

图 2: 基本块传参问题

当访问到循环语句的分支时，先进入 count 块（对于 repeat 语句是 body 块），判断循环是否结束，进入 body 块或者 end 块；body 块执行完毕后返回 count 块；

一个简单的实现如下：

```
1  auto while_count = llvm::BasicBlock::Create(*llvm_context, "while_count",
   ↪  function, 0); // 创建基本块
2
3  builder.CreateBr(while_count); // 无条件跳转
4  builder.SetInsertPoint(while_count); // 这个语句表示，此后的语句都将作为
   ↪  while_count 块的语句，除非跳转到另一块
5  auto cmp = builder.CreateICmpSLE(tmp_i, final); // 比较
6  builder.CreateCondBr(cmp, while_body, while_end); // 有条件跳转
```

repeat循环的判断过程中应该使用相反的跳转函数，即当满足条件时跳出循环，当不满足条件时继续执行循环体内容。

在开始构建while循环的代码块之前便获取了判断条件对应的值，导致在循环过程中若出现对于循环条件的改变，循环体并不能及时获取到对应的循环条件，导致循环体不能正确弹出，在后来的改进过程中，将循环体的条件获取部分延后至循环体的 body 部分，从而使得每次更新循环条件后都可以正确更新循环条件，使其正确结束循环。

在循环相关函数的初始参数中均存在 `llvm::Function *function` 部分，这部分代码在传入过程中因为疏忽导致了前面传入的部分为缺省值（指针为空），导致循环体始终出现段错误，在传入正确的指针后循环部分恢复正常。

5.2.9 段错误的处理

在代码实现以及测试阶段，我们常常会遇到段错误（Segmentation Fault），段错误究根到底就是访问了非法内存，在实际操作过程中，我们建议使用如下方式去排查段错误：

1. 检查数组是否越界
2. 检查是否使用了空指针来调用函数

3. 特别的，如果在测试时出现在 `createLoad` 和 `createStore` 操作时报告段错误，那么多半是因为你在某个地方用空指针调用了 LLVM 的函数；对于这样的情况，首先把语句附近所有的指针的值都打印出来，观察是否有空指针；然后对于使用到的作为参数传递的指针，你是否忘记在上一层进行传参，或者在定义函数时，将该参数缺省为 0；

这些是我们遇到的，导致段错误的原因，解决这些问题花费了我们不少时间，如果在写代码的时候更规范一些，我们就可以减少类似的问题。

6 实验总结

李康童 本次实验，通过对 ANTLR，LLVM 工具的学习，我们完成了一个 PASCAL-S 编译器的编写，在这个过程中，学习掌握了编译相关工具的使用（包括 ANTLR，LLVM），并且对于代码开发过程中的代码规范和代码管理的工具更加熟悉（如 Docker，GitHub）。

另外，我详细了解了一个编译器的运行过程，包括词法分析，语法分析，中间代码生成等，对编译原理课程的知识学以致用并且有了更加深刻的理解。

在编写代码的过程中，尤其是使用 LLVM 相关函数 API 时，遇到的很多问题都找不到比较详细的文档资料来解答，并且查找到的资料大多也是英文的，因此这次实验也提升了我们对于英文文献的阅读理解能力。

班建龙 这次课设让我学习到了 Docker，ANTLR 和 LLVM 的相关知识，同时也加强了我对 Git 的运用与掌握，了解了团队开发项目的流程和操作。

特别是在编写 LLVM 相关代码进行语义分析的过程中，我充分的体会到上学期编译原理课程中所学的理论知识，尤其是写到基本块内容以及它们之间的顺序执行，跳转执行时，感受尤深，这也使得我对控制流图的相关内容有了更深刻的理解，以及为后面对基本块进行优化时提供了理论支持。

而后期编写代码时阅读 LLVM 官方文档也增强了我的英文文档阅读能力，同时也使我对 LLVM 的基本组成部分：Module、Function、BasicBlock、Instruction 以及它们间的关系有了更全面的理解，走完整个课程设计的过程我觉得我对 LLVM 应该是有了初步的掌握，同时也对上学期所学的编译原理的理论知识进行了实践运用。

毛子恒 本次实验我们实现了一个简单的 Pascal-S 语言编译器，将上学期所学习的编译原理知识应用到实际项目中，使我对相关的原理知识有了更加深入的理解和体会。

在编程过程中，我们利用了成熟的开发工具，包括 Git、Docker、CMake 等，使开发效率和组内写作效率得到大幅提升，我在此期间收获良多。这些工具的熟练使用一定会对我将来的软件开发提供非常大的帮助。

最后，我们在 ANTLR 和 LLVM 代码编写过程中，参考许多国外论坛和英文文档，使得自己的英文文档阅读能力得到极大增强。

谢澳伦 通过这一次课程设计，我不但复习并巩固了上个学期理论课的内容并将一些疑问和想法在行动中践行并解决，对一个前后端分离型的编译器结构及其运作流程有了更加深刻的认识

和理解。

在代码过程中，我初步学习使用 LLVM 构建编译器、应用 Docker 和 Git 的相关知识、调整并完善了日常的 C++ 开发环境、使用非集成开发环境进行代码调试，

由于编写代码的过程中，遇到的问题几乎全部只有 LLVM 官网的自动生成文档可以参考，不仅我提高了自己的英文文档阅读水平和大型结构图的阅读理解能力，还更加积极的参与团队合作中相关问题的讨论。

李俊辉　通过本次编译原理课程设计，对于一套完整的编译器制作流程有了进一步的认识，从最开始的 ANTLR 生成的 Pascal-S 语法树，到后面通过 Visitor 模式的语法树遍历，然后通过 LLVM 工具将语法树转变为中间代码然后生成可执行文件，整个过程较好复现了编译原理这门课程对于一种语言实现的前端与后端，对于整体的实现流程有了更加深刻的见解。

对于 ANTLR 部分的语法树学会了通过 ANTLR 语法的 g4 文件描述 Pascal-S 语法，采用的类 BNF 范式较好描述了一种语言的语法规则，按照形式语言与自动机的理解方式，将这门语言较好的描述出来，不仅是对于 Pascal-S 语法的描述，后续对于其他已知语法的语言也可以采用相同的逻辑方式。

对于 Visitor 部分，LLVM 工具的使用让编译器实现了从前端语法树到后端中间代码的转变，从 Scope 变量表的存储到每一部分语法树的遍历，既是对 Pascal-S 语法的进一步深刻认知，也是对于自动机以及正则式的进一步理解，更好实现了不同学科之间的联系。

在整个实现过程中出现的错误大部分为指针错误（包括空指针，指向错误等问题），对于小型项目的代码安排有了更加深切的体会，同时小组合作的模式也让我对于团队合作实现项目有了更加好的锻炼，对于项目整体规划、功能模块分工合作有了更加深刻的认知。

乔永琦　通过本次编译原理课程设计，我加深了对编程语言底层实现的理解，对 LLVM、ANTLR、Docker 的掌握也更加完整。

本次课程设计的关键点在于 Pascal-S 语言的词法分析器、语法分析器的设计，分析过程中涉及的多种需要自己实现的功能，同时也对我们编写代码、调试代码的能力有更高的要求。与上学期编译的实验相比，对一种语言设计一个完整的编译器有更高的难度，如果对词法、语法、语义等知识点一知半解，则很难理解程序原理和调试。

在本次课程设计过程中，我们灵活运用多种程序调试方式：观察 LLVM IR 代码、通过添加输出信息、读取 LLVM 地址信息等 当然，其中最基础也是重要的还是查看 LLVM IR 的代码，检查函数、基本块、变量等是否出现异常。

调试程序、发现错误、找出根源的过程锻炼了我们遇到问题坚持不懈的品质，增强了我们面对程序时的耐心和毅力。在此过程中我们碰撞出了思维的火花，相互帮助学习，每一位成员都受益良多。

A　Pascal-S 语法

```
program
   => programHeading block DOT
```

```
programHeading
   => PROGRAM identifier (LPAREN identifierList RPAREN)? SEMI

identifier
   => IDENT

block
   => (constantDefinitionPart | typeDefinitionPart | variableDeclarationPart | procedureAndF

constantDefinitionPart
   => CONST (constantDefinition SEMI)+

constantDefinition
   => identifier EQUAL constant

constant
   => unsignedNumber
   | sign unsignedNumber
   | identifier
   | sign identifier
   | string

unsignedNumber
   => unsignedInteger
   | unsignedReal

unsignedInteger
   => NUM_INT

unsignedReal
   => NUM_REAL

sign
   => PLUS
   | MINUS

bool_
```

```
   => TRUE
   | FALSE

string
   => STRING_LITERAL

typeDefinitionPart
   => TYPE (typeDefinition SEMI)+

typeDefinition
   => identifier EQUAL type_

type_
   => simpleType
   | structuredType

simpleType
   => (CHAR | BOOLEAN | INTEGER | REAL)

structuredType
   => arrayType
   | recordType

arrayType
   => ARRAY LBRACK periods RBRACK OF type_
   | ARRAY LBRACK2 periods RBRACK2 OF type_

periods
   => period (COMMA period)*

period
   => constant DOTDOT constant

recordType
   => RECORD recordField? END

recordField
   => variableDeclaration (SEMI variableDeclaration)* SEMI
```

```
variableDeclarationPart
   => VAR variableDeclaration (SEMI variableDeclaration)* SEMI

variableDeclaration
   => identifierList COLON type_

procedureAndFunctionDeclarationPart
   => procedureOrFunctionDeclaration SEMI

procedureOrFunctionDeclaration
   => procedureDeclaration
   | functionDeclaration

procedureDeclaration
   => PROCEDURE identifier (formalParameterList)? SEMI block

formalParameterList
   => LPAREN formalParameterSection (SEMI formalParameterSection)* RPAREN

formalParameterSection
   => parameterGroup
   | VAR parameterGroup

parameterGroup
   => identifierList COLON simpleType

identifierList
   => identifier (COMMA identifier)*

constList
   => constant (COMMA constant)*

functionDeclaration
   => FUNCTION identifier (formalParameterList)? COLON simpleType SEMI block

statement
   => simpleStatement
```

```
   | structuredStatement

simpleStatement
   => assignmentStatement
   | procedureStatement
   | emptyStatement_

assignmentStatement
   => variable ASSIGN expression

variable
   => identifier (LBRACK expression (COMMA expression)* RBRACK | LBRACK2 expression (COMMA e

expression
   => simpleExpression (relationaloperator simpleExpression)?

relationaloperator
   => EQUAL
   | NOT_EQUAL
   | LT
   | LE
   | GE
   | GT

simpleExpression
   => term (additiveoperator term)?

additiveoperator
   => PLUS
   | MINUS
   | OR

term
   => signedFactor (multiplicativeoperator signedFactor)?

multiplicativeoperator
   => STAR
   | SLASH
```

```
   | DIV
   | MOD
   | AND

signedFactor
   => (PLUS | MINUS)? factor

factor
   => variable
   | LPAREN expression RPAREN
   | functionDesignator
   | unsignedConstant
   | NOT factor
   | bool_

unsignedConstant
   => unsignedNumber
   | string

functionDesignator
   => identifier LPAREN parameterList RPAREN

parameterList
   => actualParameter (COMMA actualParameter)*

procedureStatement
   => identifier (LPAREN parameterList RPAREN)?

actualParameter
   => expression parameterwidth*

parameterwidth
   => '=>' expression

emptyStatement_
   =>

empty_
```

```
   =>
   /* empty */

structuredStatement
   => compoundStatement
   | conditionalStatement
   | repetetiveStatement

compoundStatement
   => BEGIN statements END

statements
   => statement (SEMI statement)*

conditionalStatement
   => ifStatement
   | caseStatement

ifStatement
   => IF expression THEN statement (=> ELSE statement)?

caseStatement
   => CASE expression OF caseListElement (SEMI caseListElement)* (SEMI ELSE statements)? END

caseListElement
   => constList COLON statement

repetetiveStatement
   => whileStatement
   | repeatStatement
   | forStatement

whileStatement
   => WHILE expression DO statement

repeatStatement
   => REPEAT statements UNTIL expression
```

```
forStatement
   => FOR identifier ASSIGN forList DO statement

forList
   => initialValue (TO | DOWNTO) finalValue

initialValue
   => expression

finalValue
   => expression

recordVariableList
   => variable (COMMA variable)*

PLUS
   => '+'

MINUS
   => '-'

STAR
   => '*'

SLASH
   => '/'

ASSIGN
   => '=>='

COMMA
   => ','

SEMI
   => ';'

COLON
   => '=>'
```

```
EQUAL
   => '='

NOT_EQUAL
   => '<>'

LT
   => '<'

LE
   => '<='

GE
   => '>='

GT
   => '>'

LPAREN
   => '('

RPAREN
   => ')'

LBRACK
   => '['

LBRACK2
   => '(.'

RBRACK
   => ']'

RBRACK2
   => '.)'

DOT
```

```
   => '.'

DOTDOT
   => '..'

LCURLY
   => '{'

RCURLY
   => '}'

IDENT
   => ('a' .. 'z' | 'A' .. 'Z') ('a' .. 'z' | 'A' .. 'Z' | '0' .. '9' | '_')*

STRING_LITERAL
   => '\'' ('\'\'' | ~ ('\''))* '\''

NUM_INT
   => ('0' .. '9') +

NUM_REAL
   => ('0' .. '9') + (('.' ('0' .. '9') + (EXPONENT)?)? | EXPONENT)

EXPONENT
   => ('e') ('+' | '-')? ('0' .. '9') +
```