编译原理研讨课实验 PR001 实验报告

一、任务说明

本次实验的任务为：

1.熟悉 Clang 的安装和使用

1）掌握如何从源代码编译安装 LLVM 和 Clang

2）了解如何生成和查看 C 程序对应的 AST

2.通过修改前端#pragma 到 AST 的信息传递，使编译器能够识别#pragma elementWise，并在 AST 中增加相应元素。

对于添加了制导的源程序 *.c ，按照规则在编译时打印每个函数的名称，该函数是否在制导范围内。 对于一个函数的是否在制导范围内的定义:

1） 区域以函数/过程的定义（不是声明）为单位

2） 一个制导总是匹配在其后出现的，离它最近的一个函数定义

3） 一个制导只能匹配最多一个函数定义

二、成员组成

李昊宸 2017K8009929044

李颖彦 2017K8009929025

陆润宇 2017K8009929027

三、实验设计

将实验分为：parse（语法解析），sema（语义），AST（语法树）三个部分来逐一实现。

对于给定的代码框架，采用 gdb 跟随 clang 基础部分以及 TraverseFunctionDecls.cpp 运行时的函数调用流程，可以定位代码框架中为识别#pragma elementWise 而需要进行补充的地方，进而完成实验所需的内容。

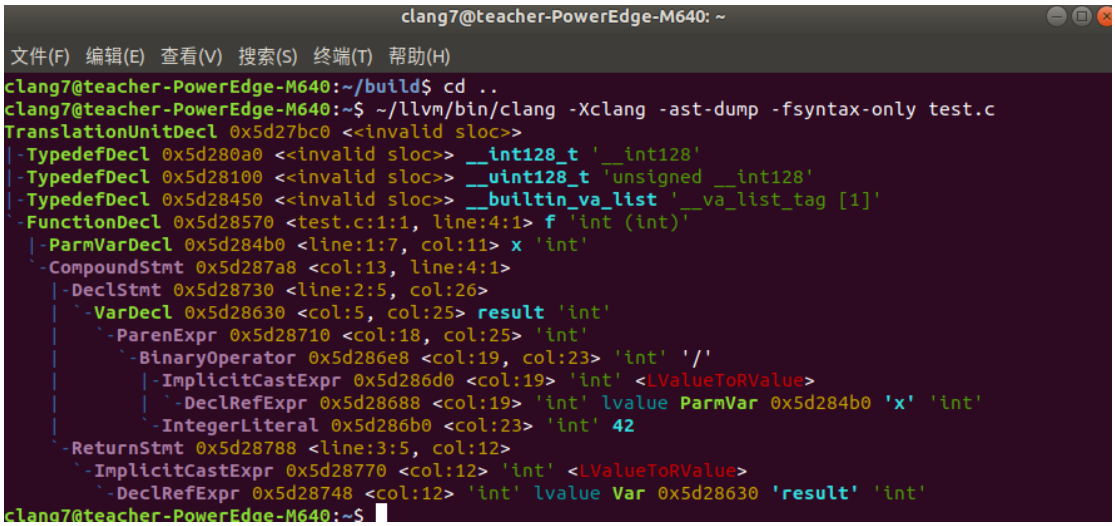
四、设计思路

熟悉 Clang 的安装和使用：

1. 准备一个 C 程序 test.c

```
int f(int x) {
  int result = (x / 42);
  return result;
}
```

2. 使用`clang`将 AST 给 dump 出来：`~/llvm-install/bin/clang -Xclang -ast-dump -fsyntax-only test.c`



图一 test.c 的 AST

添加 element wise 的制导：

1. 语法阶段

首先通过 parse 的构造函数来对 elementWise 的 PragmaHandler 进行初始化。之后，设计在 HandlerPragma()中被调用的 PragmaElementWiseHandler()，将 token 赋为 annot_pragma_elementWise 的类型，重新放回 token 流中。

之后，在对 Declaration 的解析当中，依照 token 类型 annot_pragma_elementWise，调用 HandlerPragmaElementWise()函数进行处理，它再调用 ActOnPragmaElementWise()函数，目的是对 elementWise 进行实质上的功能实现。

2. 语义阶段

在 prj1 中，我们只需要实现当#pragma elementWise 出现时，将对应的函数打印成 1 并输出，所以 ActOnPragmaElementWise()的内容即为 ElementWiseContext 置 1。

3. 构建 AST 阶段

为每个函数声明新增一个变量 isElementWise，每当一个函数定义时，若 ElementWiseContext 的值为 1，则将这个定义内部的 isElementWise 变量置为 1，再将 ElementWiseContext 清零。这样，在输出插件遍历函数声明时，便可以判断具体函数是否是 elememtWise 的。

五、实验实现

1. 语法阶段

/home/clang7/llvm/tools/clang/include/clang/Parse/Parser.h
借助模板定义 ElementWiseHandler

```
144     OwningPtr<PragmaHandler> ElementWiseHandler;
```

在 class parse 中声明:

```
/// \brief Handle the annotation token produced for
/// #pragma elementWise
void HandlePragmaElementWise();
```

/home/clang7/llvm/tools/clang/lib/Parse/Parser.cpp
在 parse 的构造函数中，对 elementwise 的 PragmaHandler 进行初始化

```
ElementWiseHandler.reset(new PragmaElementWiseHandler());
PP.AddPragmaHandler(ElementWiseHandler.get());
```

这里注册了 PragmaElementWiseHandler，使下面的 FindHandler 函数可以根据 token 的 name 来找到 PragmaElementWiseHandler 的入口地址

```
PP.RemovePragmaHandler(ElementWiseHandler.get());
ElementWiseHandler.reset();
```

这里移除 elementwise 的 pragmahandler 并重置

/home/clang7/llvm/tools/clang/lib/Lex/Pragma.cpp
在函数
void PragmaElementWiseHandler::HandlePragma(Preprocessor &PP,
PragmaIntroducerKind Introducer,
Token &ElementWiseTok)
中通过调用

```
PragmaHandler *Handler
  = FindHandler(Tok.getIdentifierInfo() ? Tok.getIdentifierInfo()->getName()
                                        : StringRef(),
                /*IgnoreNull=*/false);
```

发现 handler 为 PragmaElementWiseHandler，并进行调用

```
  // Otherwise, pass it down.
  Handler->HandlePragma(PP, Introducer, Tok);
}
```

/home/clang7/llvm/tools/clang/lib/Parse/ParsePragma.h

```
class PragmaElementWiseHandler : public PragmaHandler {
public:
  explicit PragmaElementWiseHandler() : PragmaHandler("elementWise") {}

  virtual void HandlePragma(Preprocessor &PP, PragmaIntroducerKind Introducer,
                            Token &FirstToken);
};
```

/home/clang7/llvm/tools/clang/include/clang/Basic/TokenKinds.def

```
//Annotation for #pragma elementWise    Clang 7 [3 hours ago] • 5.20 commit
ANNOTATION(pragma_elementWise)
```

/home/clang7/llvm/tools/clang/lib/Parse/ParsePragma.cpp

```
void PragmaElementWiseHandler::HandlePragma(Preprocessor &PP,
                                            PragmaIntroducerKind Introducer,
                                            Token &ElementWiseTok) {
  PP.CheckEndOfDirective("pragma elementWise");
  Token *Toks =
    (Token*) PP.getPreprocessorAllocator().Allocate(
      sizeof(Token) * 1, llvm::alignOf<Token>());
  new (Toks) Token();
  Toks[0].startToken();
  Toks[0].setKind(tok::annot_pragma_elementWise);
  Toks[0].setLocation(ElementWiseTok.getLocation());
  Toks[0].setAnnotationValue(NULL);
  PP.EnterTokenStream(Toks, 1, /*DisableMacroExpansion=*/true,
                      /*OwnsTokens=*/false);
}
```

这里调用了
/home/clang7/llvm/tools/clang/lib/Lex/PPDirectives.cpp
中的
void Preprocessor::CheckEndOfDirective(const char *DirType, bool EnableMacros)
检查后缀 pragma elementwise

并通过
Toks[0].setKind(tok::annot_pragma_elementWise);
设置 token 的类型为 annot_pragma_elementWise

/home/clang7/llvm/tools/clang/lib/Parse/Parser.cpp
在 Parser::ParseExternalDeclaration 函数中，根据 tok 的不同类型来选择调用相应的函数，如果 tok 是 elementwise 的类型，则调用 HandlePragmaElementWise()

```
case tok::annot_pragma_elementWise:
  HandlePragmaElementWise();
  return DeclGroupPtrTy();
```

/home/clang7/llvm/tools/clang/lib/Parse/ParsePragma.cpp

```
void Parser::HandlePragmaElementWise() {
  assert(Tok.is(tok::annot_pragma_elementWise));
  ConsumeToken();
  Actions.ActOnPragmaElementWise();
}
```

调用 Sema 阶段的函数 ActOnPragmaElementWise

2. 语义阶段

/home/clang7/llvm/tools/clang/include/clang/Sema/Sema.h

```
int ElementWiseContext;
```

```
void ActOnPragmaElementWise();
```

/home/clang7/llvm/tools/clang/lib/Sema/Sema.cpp
在 sema 的构造函数中，将 ElementWiseContext 的初始值赋值为 0

```
Sema::Sema(Preprocessor &pp, ASTContext &ctxt, ASTConsumer &consumer,
           TranslationUnitKind TUKind,
           CodeCompleteConsumer *CodeCompleter)
  : ElementWiseContext(0),
```

/home/clang7/llvm/tools/clang/lib/Sema/SemaAttr.cpp
ActOnPragmaElementWise()的功能为被调用时将 ElementWiseContext 赋值为 1

```
void Sema::ActOnPragmaElementWise() {
  ElementWiseContext = 1;
}
```

/home/clang7/llvm/tools/clang/lib/Sema/SemaDecl.cpp
在函数
Decl *Sema::ActOnFinishFunctionBody(Decl *dcl, Stmt *Body,
bool IsInstantiation)
中有

```
if (FD) {
  FD->setBody(Body);
  if(ElementWiseContext) {
    FD->setElementWise(true);
    ElementWiseContext = 0;
  }
  else
    FD->setElementWise(false);
```

如果 ElementWiseContext 为 1，则 setElementWise 函数会将 IsElementWise 赋值为 true，并将 ElementWiseContext 清零；否则 IsElementWise 置为 false

3. AST

/home/clang7/llvm/tools/clang/include/clang/AST/Decl.h
声明 IsElementWise 的位域为 1：

```
bool IsElementWise: 1;
```

构建 setElementWise 函数

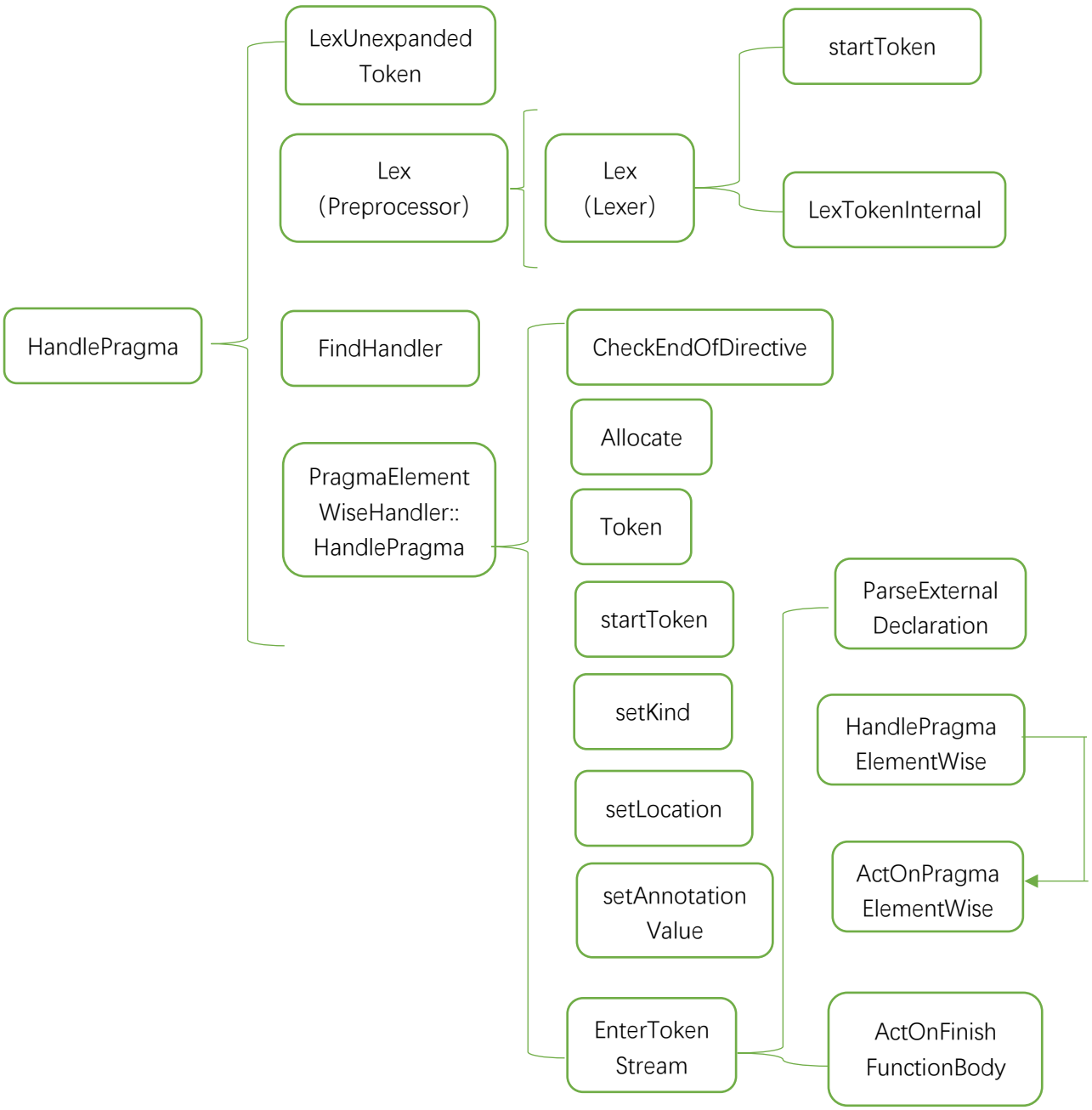
```
bool isElementWise() const { return IsElementWise; }

void setElementWise(bool I) {
  assert(doesThisDeclarationHaveABody());
  IsElementWise = I;
}
```

最后，在输出插件
/home/clang7/llvm/tools/clang/examples/TraverseFunctionDecls/TraverseFunctionDecls.cpp 中，根据 isElementWise 的值来决定相应的输出

```
namespace {
class TraverseFunctionDeclsVisitor
    : public RecursiveASTVisitor<TraverseFunctionDeclsVisitor> {
public:
    explicit TraverseFunctionDeclsVisitor(ASTContext *Context)
        : Context(Context) {}

    bool TraverseDecl(Decl *DeclNode) {
        if (DeclNode == NULL) {
            return true;
        }
        if (const FunctionDecl *FD = dyn_cast<FunctionDecl>(DeclNode)) {
            std::string name = FD -> getNameAsString();
            bool IsElementWise = FD -> isElementWise();

            if(IsElementWise) {
                funcNamesToIsElementWise[name] = true;
            } else {
                std::map<std::string, bool>::iterator it = funcNamesToIsElementWise.find(name);
                if(it == funcNamesToIsElementWise.end())
                    funcNamesToIsElementWise[name] = FD -> isElementWise();
            }
        }
        return RecursiveASTVisitor<TraverseFunctionDeclsVisitor>::TraverseDecl(DeclNode);
    }
    void OutputIsElementWise() {
        for(std::map<std::string, bool>::iterator it = funcNamesToIsElementWise.begin(); it != funcName
            llvm::outs() << it -> first << ": " << it -> second << "\n";
        }
    }
```

```
private:
    ASTContext *Context;
    std::map<std::string, bool> funcNamesToIsElementWise;
};

class TraverseFunctionDeclsConsumer : public ASTConsumer {
public:
    explicit TraverseFunctionDeclsConsumer(ASTContext *Context)
        : Visitor(Context) {}

    virtual void HandleTranslationUnit(ASTContext &Context) {
        Visitor.TraverseDecl(Context.getTranslationUnitDecl());
        Visitor.OutputIsElementWise();
    }
private:
    TraverseFunctionDeclsVisitor Visitor;
};
```

4. 主要调用流程图



六、总结

本次实验主要是围绕 clang 的词法和语法方面，首先识别#pragma elementWise，并调用相关函数来对 token 进行处理。处理之后，将变量 ElementWiseContext 置 1。随后 AST 处理时若 ElementWiseContext 为 1，则将该函数声明的内部值 IsElementWise 置 1，将 ElementWiseContext 清零。如此即可实现对于 elementWise 的检测和标识。

本次实验成功实现了任务要求，且代码书写切近源码风格，较为清晰。

以下是几组个人测试结果:

测试一:

```
clang7@teacher-PowerEdge-M640:~/lhc$ cat ~/lhc/example.c
#pragma elementWise
int main(){
    return 1;
}
int second(){
    return 0;
}
clang7@teacher-PowerEdge-M640:~/lhc$ ~/lhc/llvm-install/bin/clang -cc1 -load
~/lhc/build/lib/TraverseFunctionDecls.so -plugin traverse-fn-decls ~/lhc/exam
ple.c
main: 1
second: 0
```

正常情况下，制导范围包括第一个遇到的函数定义 main，不包括 second。

测试二:

```
clang7@teacher-PowerEdge-M640:~/lhc$ cat ~/lhc/example.c
#pragma elementWise
int main(){
    return 1;
}
#pragma elementWise
int second(){
    return 0;
}
clang7@teacher-PowerEdge-M640:~/lhc$ ~/lhc/llvm-install/bin/clang -cc1 -load
~/lhc/build/lib/TraverseFunctionDecls.so -plugin traverse-fn-decls ~/lhc/exam
ple.c
main: 1
second: 1
```

正常情况下，第一个制导范围包括第一个遇到的函数定义 main，第二个制导范围包括 second。

测试三：

```
clang7@teacher-PowerEdge-M640:~/lhc$ cat ~/lhc/example.c

int main(){
    return 1;
}
#pragma elementWise
int second(){
    return 0;
}
clang7@teacher-PowerEdge-M640:~/lhc$ ~/lhc/llvm-install/bin/clang -cc1 -load
~/lhc/build/lib/TraverseFunctionDecls.so -plugin traverse-fn-decls ~/lhc/exam
ple.c
main: 0
second: 1
```

正常情况下，制导范围包括第一个遇到的函数定义 second，不包括之前的函数定义 main。

测试四：

```
clang7@teacher-PowerEdge-M640:~/lhc$ cat ~/lhc/example.c
#pragma elementWise
#define u8 char
int main(){
    return 1;
}
#pragma elementWise
int second(){
    return 0;
}
clang7@teacher-PowerEdge-M640:~/lhc$ ~/lhc/llvm-install/bin/clang -cc1 -load
~/lhc/build/lib/TraverseFunctionDecls.so -plugin traverse-fn-decls ~/lhc/exam
ple.c
main: 1
second: 1
```

制导与函数定义间加入其他语句（如宏定义），不影响对函数定义的制导

测试五：

```
clang7@teacher-PowerEdge-M640:~/lhc$ cat example.c
#pragma elementWise
#define u8 char
int foo();

int second(){
    return 0;
}

int foo(){
    return 0;
}

int main(){
    return 1;
}
clang7@teacher-PowerEdge-M640:~/lhc$ ~/lhc/llvm-install/bin/clang -cc1 -load
~/lhc/build/lib/TraverseFunctionDecls.so -plugin traverse-fn-decls ~/lhc/exam
ple.c
foo: 0
main: 0
second: 1
```

制导与函数定义之间增加函数声明，制导后第一个遇到的函数定义 second 在制导范围内，函数声明 foo 并不在制导范围内。

七、成员总结

李昊宸：本次实验负责 clang 代码的书写，以及实验报告的修改和补充。第一次处理大工程代码，起初无从下手，还好老师给出了处理 pack 制导的样例，得以比较快的了解 clang 代码的运行过程，并且实现了仿照 pack 完成的 element wise 制导定义。每个调用的函数名都较为直观，可以快速理解函数实现的功能，但是在不同的分析过程下，经常会出现同名函数（但是不在同一名字空间），导致起初较难理解调用过程。

李颖彦：

本次实验负责实验报告的书写。因为本次实验是第一次接触 clang 源代码，所以理解 clang 源码的结构和功能便花了很长的时间，同时还需要有一定 C++的面向对象的基础。前期的准备工作做的很长。在理解了 clang 源码基础上，实验则变得简单了很多。

陆润宇：

本次实验负责实验报告的修改和补充。在 clang 源代码中，用到了 C++类相关的功能，与我之前进行的其他实验相比，我觉得这种方式屏蔽了更多底层的细节，使得代码更加清晰明了。在这次实验中，使用 gdb 跟踪运行起到了很关键的作用，这使我们能够沿着程序运行轨迹逐步补充需要的内容，减少了不必要的代码阅读量，也帮助我们有效地理解了 clang 代码的部分运行机制。