# 编译原理研讨课实验PR002实验报告

## 任务说明

本实验PR002在PR001的基础上，要求对能够支持的'+','*','='三种操作有正确的语法分析和错误检测。

## 1. 实验目的

- 扩展AST的表示已支持element-wise的操作
- 操作匹配：类型匹配（静态数组，类型相同），大小匹配（大小相等）
- 生成合法的AST
- 不破坏原有C语言代码的语义

## 2. 实验具体要求

- 支持'+','*','='三种操作
- 支持C语言标准的int类型
- 操作数应为静态大小的一维数组
- 操作数类型匹配：类型相同且大小相同
- 编译器可以直接编译符合规范的源代码文件生成二进制文件并正确执行

# 成员组成

骆嵩涛 2018K8009907001
毛啸腾 2018K8009907033
陈彦帆 2018K8009918002

# 实验设计

## 设计思路

### 具体目标

Parser生成AST时，每个函数（定义或声明）会生成一个FunctionDecl结点，我们在PR001中，已经完成了对该函数是否被`#pragma elementWise`标注定义了一个标记属性（IsElementWise)。之后在生成AST的过程中，原本不支持数组直接的加法、乘法和赋值（会报出对于操作无效的类型），现在我们需要他支持上述三种运算，在不会报错的同时生成正确的AST语法树。同时对于其他类对数组的操作我们依然不能支持，同时对于如类型不匹配，大小不匹配等问题都需要在生成我们被标记了IsElementWise函数时报出准确的错误。（具体错误的对应可以参见测试程序部分）

### 实现思路

在不支持elementWise的情况下，当我们对于二元运算符的分析方式是如下的调用顺序：`Sema::ActOnBinOp`-> `Sema::BuildBinOp`-> `Sema::CreateBuiltinBinOp`之后处理不同的运算符，生成AST语法分析树。所以我们要在`Sema::CreateBuiltinBinOp`函数中，对于需要修改的二元操作符（+，*，=）进行支持elementWise的修改。

注：

- `Sema::ActOnBinOp`：开始对于二元运算符进行分析。初步分析二元运算符左右是否都有左和右表达式左表达式和右表达式（具体报错与测试用例12相同）。
- `Sema::BuildBinOp`：可以调用`checkPseudoObjectAssignment`检查左对象是不是一个伪对象，`BuildOverloadedBinOp`创建重载二元运算符（重载运算符即为一个运算符可以根据输入类型不同等，将有不同的含义），`CreateBuiltinBinOp`创建非重载二元运算符，我们专注地部分在此）
- `Sema::CreateBuiltinBinOp`：即创建非重载二元运算符，在此函数中，对于每一种不同的二元运算符都有独特的检测函数，我们就此修改`CheckAssignmentOperands`、`CheckAdditionOperands`和`CheckMultiplyDivideOperands`的规则，以此来完成支持elementWise的二元运算的检验。

**"="赋值的修改：**

单赋值符号(不包括"+="等)的语法检查的流程是`CheckAssignmentOperands`->`CheckSingleAssignmentConstraints`，在检查赋值操作的约束的时候，我们会检查二元运算符的左表达式和右表达式是否是相容的（Compatible）或者不相容的（Incompatible）。由于我们要支持elementWise的数组的赋值，因此我们对于相容的情况，还要加入一种左表达式和右表达式都有可能是相同大小的int类型数组。

同时这里还需要值得注意的是，在`CheckSingleAssignmentConstraints`之前，他还会通过`CheckForModifiableLvalue`检测左侧是否是modifiable的左值。例如：如果我们用const修饰的变量即是不可修改的。而实际上数组结构也不是modifiable的，所以通常处理会被报错。我们的修改方式有很多种，这里我们认为修改的越表层越不容易出错，因为我们不修改数组的标记属性modifiable，而我们是直接跳过了报错流程，让编译继续执行，这样就可以成功的生成最后的AST语法树。

相关函数：isIntType　CheckAssignmentOperands

**"+"加法的修改：**

在处理加法的时候，`CreateBuiltinBinOp`会直接调用`CheckAdditionOperands`。为了使支持elementWise加法，我们需要对该函数进行适当的修改，使之可以在elementWise标记下，支持左表达式和右表达式可以是大小相同的int数组。同时如果出现类型不匹配，大小不匹配等情况，我们也应该正确的给出报错。

相关函数：isIntType　CheckElementWiseConstraints　CheckAdditionOperands

**"*"乘法的修改：**

在处理加法的时候，`CreateBuiltinBinOp`会直接调用`CheckMultiplyDivideOperands`为了使支持elementWise乘法，我们需要对该函数进行适当的修改，使之可以在elementWise标记下，支持左表达式和右表达式可以是大小相同的int数组。同时如果出现类型不匹配，大小不匹配等情况，我们也应该正确的给出报错。

相关函数：isIntType　CheckElementWiseConstraints　CheckMultiplyDivideOperands

# 实验实现

## 1. 添加`isIntType()`：实现对于int这一单独类型的判断函数

首先我们通过参考`isIntergerType()`的类似写法写出同样的类型判断函数:

`./tools/clang/include/clang/AST/Type.h`

```
inline bool Type::isIntegerType() const {
  if (const BuiltinType *BT = dyn_cast<BuiltinType>(CanonicalType))
    return BT->getKind() >= BuiltinType::Bool &&
           BT->getKind() <= BuiltinType::Int128;
  if (const EnumType *ET = dyn_cast<EnumType>(CanonicalType)) {
    // Incomplete enum types are not treated as integer types.
    // FIXME: In C++, enum types are never integer types.
    return IsEnumDeclComplete(ET->getDecl()) &&
      !IsEnumDeclScoped(ET->getDecl());
  }
  return false;
}
```

我们知道integer实际上就是int的全程，那么我们有自己定义一个`isIntType()`是否多此一举？我们可不可以就通过`isIntergerType()`来判断是否是int类型？答案是否定的。

通过观察可以发现实际上这里的interger实际上所指的范围非常广泛，具体有哪些内容，可以看下面的ASTContext.h的代码段中的所有类型。而就本实验而言，我们所要关注的只有int这一种类型，所以我们要单独的写出`isIntType()`.

`./tools/clang/include/clang/AST/ASTContext.h`

```
  CanQualType BoolTy;
  CanQualType CharTy;
  CanQualType WCharTy;  // [C++ 3.9.1p5], integer type in C99.
  CanQualType WIntTy;   // [C99 7.24.1], integer type unchanged by default
promotions.
  CanQualType Char16Ty; // [C++0x 3.9.1p5], integer type in C99.
  CanQualType Char32Ty; // [C++0x 3.9.1p5], integer type in C99.
  CanQualType SignedCharTy, ShortTy, IntTy, LongTy, LongLongTy, Int128Ty;
```

那么根据我们的需求，首先我们只需要类型是int型，因此我们将getkind的结果仅与int对比。同时我们不需要对enum型对比，因此我们就把对于enum类型的判断去掉即可（//TODO:@cyf加一个测例带有char bool enum型，然后报错，然后从这里链接过去）

具体代码如下：

`./tools/clang/include/clang/AST/Type.h`

```
inline bool Type::isIntType() const {
  if (const BuiltinType *BT = dyn_cast<BuiltinType>(CanonicalType))
    return BT->getKind() == BuiltinType::Int;
  return false;
}
```

注：CanonicalType意为规范类型，指与任何语法糖或者typedef类型无关的类型。

## 2.添加函数CheckElementWiseConstraints：对于函数有elementWise标记的时候，我们对于操作符左右的限制要有特殊的判断方式

当我们的函数被elementWise标记之后，我们实际上就修改了对于我们需要支持的二元操作运算符的约束有了改变。因此我们要对这种特殊的约束设计一个函数，来判断左表达式和右表达式是否符合我们的规定。

而实际上这里我们定义的函数，是在已知左表达式和右表达式均为数据的情况下，首先对于他的数组类型是不是int类型进行比较，之后我们在对他的大小进行比较。

之后我们还要根据C99 6.3.2.1p2，将左值转化成右值。

这里非常值得注意的是我们在这里我们的左表达式和右表达式的类型只要是int类型即可，并不需要左表达式和右表达式的类型完全相同，间接相同(CanonicalType相同)也可，这里具体可见测例13，如果我们将两边的type直接进行==比较，那么在测例13中就会报错，这也是我们为什么建议把测例13加入统一测例。

最后如果都符合要求，我们返回对应左右表达式的类型，否则返回QualType()。（QualType()表示其真正类型还未被确定，有待后续检查）

`./tools/clang/lib/Sema/SemaExpr.cpp`

```
QualType Sema::CheckElementWiseConstraints(ExprResult &LHS, ExprResult &RHS) {
  // Now we assert LHS and RHS are constantarray type.
  const ConstantArrayType *LHSCAT = Context.getAsConstantArrayType(LHS.get()-
>getType());
```

```
  const ConstantArrayType *RHSCAT = Context.getAsConstantArrayType(RHS.get()-
>getType());
  // Check if the element types of array are both Int type.
  if (LHSCAT->getElementType()->isIntType() &&
      RHSCAT->getElementType()->isIntType() &&
      LHSCAT->getSize() == RHSCAT->getSize()) {
    // From c99 6.3.2.1p2, do lvalue conversion
    // We do not need array-to-pointer conversion
    LHS = DefaultLvalueConversion(LHS.take());
    if(LHS.isInvalid())
      return QualType();
    RHS = DefaultLvalueConversion(RHS.take());
    if(RHS.isInvalid())
      return QualType();
    // We ignore any qualifiers.
    return Context.getCanonicalType(LHS.get()->getType()).getUnqualifiedType();
  }
  return QualType();
}
```

实际上这部分逻辑并非我们独立想出，而是我们参考了llvm中`Sema::CheckVectorOperands`的一系列操作方式，比如为什么要将需要将左右表达式都转化左值，为什么需要去限定，转化规范型，实际上我们都是参照下面的代码段才得到了启发。在我们的测试中去掉左值的转换并不会影响生成树的生成，但是到PR003具体到生成执行代码是，有可能由于不符合llvm的标准而出现bug和达不到预期。对于左值的检测我们也是有测例的，分别是测例4和测例6。

`./tools/clang/lib/Sema/SemaExpr.cpp`

```
QualType Sema::CheckVectorOperands(ExprResult &LHS, ExprResult &RHS,
                                   SourceLocation Loc, bool IsCompAssign) {
  if (!IsCompAssign) {
    LHS = DefaultFunctionArrayLvalueConversion(LHS.take());
    if (LHS.isInvalid())
      return QualType();
  }
  RHS = DefaultFunctionArrayLvalueConversion(RHS.take());
  if (RHS.isInvalid())
    return QualType();

  // For conversion purposes, we ignore any qualifiers.
  // For example, "const float" and "float" are equivalent.
  QualType LHSType =
    Context.getCanonicalType(LHS.get()->getType()).getUnqualifiedType();
  QualType RHSType =
    Context.getCanonicalType(RHS.get()->getType()).getUnqualifiedType();

  // If the vector types are identical, return.
  if (LHSType == RHSType)
    return LHSType;

  // Handle the case of equivalent AltiVec and GCC vector types
  if (LHSType->isVectorType() && RHSType->isVectorType() &&
      Context.areCompatibleVectorTypes(LHSType, RHSType)) {
    if (LHSType->isExtVectorType()) {
      RHS = ImpCastExprToType(RHS.take(), LHSType, CK_BitCast);
      return LHSType;
    }
```

```
    if (!IsCompAssign)
      LHS = ImpCastExprToType(LHS.take(), RHSType, CK_BitCast);
    return RHSType;
  }

  ...
}
```

### 3.修改CheckAdditionOperands：使加法符号左表达式和右表达式可以支持数组类型而不报错

这里加法的逻辑即检查加法左右表达式的类型如果是数组，那么他是否符合对应elementWise的要求。因此在检测出数组后，我们直接调用我们已经写好的CheckElementWiseConstraints函数即可

`./tools/clang/lib/Sema/SemaExpr.cpp`

```
QualType Sema::CheckAdditionOperands( // C99 6.5.6
    ExprResult &LHS, ExprResult &RHS, SourceLocation Loc, unsigned Opc,
    QualType* CompLHSTy) {
  checkArithmeticNull(*this, LHS, RHS, Loc, /*isCompare=*/false);

...

  //PROJ2
  if (LHS.get()->getType()->isConstantArrayType() &&
      RHS.get()->getType()->isConstantArrayType() &&
      getCurFunctionDecl() && getCurFunctionDecl()->isElementWise()) {
    QualType compType = CheckElementWiseConstraints(LHS, RHS);
    if(!compType.isNull())
      return compType;
  }

...
}
```

### 4.修改CheckMultiplyDivideOperands：使乘法符号左表达式和右表达式可以支持数组类型而不报错

乘法与加法的处理逻辑是完全类似的，但这里还有一点细节值得注意。我们仅要求了对于加法和乘法的elementWise支持，因此我们对于除法是不支持的，所以当出现对于数组除法的时候我们是应该报错的。由于在这里，乘除被放在了同一个函数里做检查，所以我们在检测两端都是数组的时候，要对于除法的情况进行排除，所以我们这里加了一个额外的!isDiv逻辑。如果不加，那么将在测例11无法成功报出正确的错误。

`./tools/clang/lib/Sema/SemaExpr.cpp`

```
QualType Sema::CheckMultiplyDivideOperands(ExprResult &LHS, ExprResult &RHS,
                                           SourceLocation Loc,
                                           bool IsCompAssign, bool IsDiv) {
  checkArithmeticNull(*this, LHS, RHS, Loc, /*isCompare=*/false);
...

  // PROJ2
  if (!IsDiv && LHS.get()->getType()->isConstantArrayType() &&
```

```
      RHS.get()->getType()->isConstantArrayType() &&
      getCurFunctionDecl() && getCurFunctionDecl()->isElementWise()) {
    QualType compType = CheckElementWiseConstraints(LHS, RHS);
    if(!compType.isNull())
      return compType;
  }

...
}
```

## 5.修改CheckForModifiableLvalue，使跳过对于数组的检查

首先我们为什么要修改CheckForModifiableLvalue，在下面两个相关文件及注释中已经表明，一个左值是包括数组类型的，从isModifiableLvalue也可以看出，通过获取类别，对于某一个类型是否是可修改左值已经有了一个明确的映射。

`./tools/clang/include/clang/AST/Expr.h`

```
  /// isModifiableLvalue - C99 6.3.2.1: an lvalue that does not have array type,
  /// does not have an incomplete type, does not have a const-qualified type,
  /// and if it is a structure or union, does not have any member (including,
  /// recursively, any member or element of all contained aggregates or unions)
  /// with a const-qualified type.
  ///
  /// \param Loc [in,out] - A source location which *may* be filled
  /// in with the location of the expression making this a
  /// non-modifiable lvalue, if specified.
  isModifiableLvalueResult isModifiableLvalue(ASTContext &Ctx,
                                              SourceLocation *Loc = 0) const;
```

`./tools/clang/lib/AST/ExprClassification.cpp`

```
Expr::isModifiableLvalue(ASTContext &Ctx, SourceLocation *Loc) const {
  SourceLocation dummy;
  Classification VC = ClassifyModifiable(Ctx, Loc ? *Loc : dummy);
  switch (VC.getKind()) {
  case Cl::CL_LValue: break;
  case Cl::CL_XValue: return MLV_InvalidExpression;
  case Cl::CL_Function: return MLV_NotObjectType;
  case Cl::CL_Void: return MLV_InvalidExpression;
  case Cl::CL_AddressableVoid: return MLV_IncompleteVoidType;
  case Cl::CL_DuplicateVectorComponents: return MLV_DuplicateVectorComponents;
  case Cl::CL_MemberFunction: return MLV_MemberFunction;
  case Cl::CL_SubObjCPropertySetting: return MLV_SubObjCPropertySetting;
  case Cl::CL_ClassTemporary: return MLV_ClassTemporary;
  case Cl::CL_ArrayTemporary: return MLV_ArrayTemporary;
  case Cl::CL_ObjCMessageRValue: return MLV_InvalidMessageExpression;
  case Cl::CL_PRValue:
    return VC.getModifiable() == Cl::CM_LValueCast ?
      MLV_LValueCast : MLV_InvalidExpression;
  }
  assert(VC.getKind() == Cl::CL_LValue && "Unhandled kind");
  switch (VC.getModifiable()) {
  case Cl::CM_Untested: llvm_unreachable("Did not test modifiability");
  case Cl::CM_Modifiable: return MLV_Valid;
  case Cl::CM_RValue: llvm_unreachable("CM_RValue and CL_LValue don't match");
```

```
  case Cl::CM_Function: return MLV_NotObjectType;
  case Cl::CM_LValueCast:
    llvm_unreachable("CM_LValueCast and CL_LValue don't match");
  case Cl::CM_NoSetterProperty: return MLV_NoSetterProperty;
  case Cl::CM_ConstQualified: return MLV_ConstQualified;
  case Cl::CM_ArrayType: return MLV_ArrayType;
  case Cl::CM_IncompleteType: return MLV_IncompleteType;
  }
  llvm_unreachable("Unhandled modifiable type");
}
```

但是由于我们要支持对于数组的赋值，所以我们相当于要跳过对于数组类型是否是可修改左值的检查，因此我们相当于“开后门”的方式，让数组即使被看出来不是可修改左值，也可以继续执行编译程序。

具体修改如下：

```
static bool CheckForModifiableLvalue(Expr *E, SourceLocation Loc, Sema &S) {
  ...
  case Expr::MLV_ArrayType:
  case Expr::MLV_ArrayTemporary:
    if(S.getCurFunctionDecl() && S.getCurFunctionDecl()->isElementWise())
      return false; //ADD THIS IN PROO2
    Diag = diag::err_typecheck_array_not_modifiable_lvalue;
    NeedType = true;
    break;
  ...
}
```

我们在识别到函数的elementWise属性之后，将跳过对于数组的检查，直接返回函数。

## 6.修改CheckSingleAssignmentConstraints：使赋值符号左表达式和右表达式可以支持数组类型而不报错

这里实际上赋值操作的检查逻辑和加法以及乘法知识稍有不同。首先我们修改的CheckSingleAssignmentConstraints实际上是CheckAssignmentOperands的一个调用好的封装函数，因此逻辑细节和加乘有所不同，所以这里我们没有直接调用我们写好的CheckElementWiseConstraints，而是再写了一遍对于elementWise检验的要求，同时这里值得注意的是，我们对于等号右侧进行了右值的转化，这与clang处理标量赋值的逻辑是相同的。

`./tools/clang/lib/Sema/SemaExpr.cpp`

```
Sema::CheckSingleAssignmentConstraints(QualType LHSType, ExprResult &RHS,
                                       bool Diagnose) {
...

  // 6301-6318: New in PROO2
  QualType RHSType = RHS.get()->getType();
  if (LHSType->isConstantArrayType() &&
      RHSType->isConstantArrayType() &&
      getCurFunctionDecl() && getCurFunctionDecl()->isElementWise()) {
    const ConstantArrayType *LHSCAT = Context.getAsConstantArrayType(LHSType);
    const ConstantArrayType *RHSCAT = Context.getAsConstantArrayType(RHSType);
    QualType LHSElementType = LHSCAT->getElementType().getUnqualifiedType();
    QualType RHSElementType = RHSCAT->getElementType().getUnqualifiedType();
    if (LHSElementType->isIntType() && RHSElementType->isIntType() &&
        LHSCAT->getSize() == RHSCAT->getSize()) {
```

```
          RHS = DefaultLvalueConversion(RHS.take());
          return Compatible;
        }
    else return Incompatible;
  }

...
}
```

### 7. 补充上述实现中省略的声明或定义

`./tools/clang/include/clang/AST/Type.h`

```
...
bool isIntType() const;
...
```

`./tools/clang/include/clang/Sema/Sema.h`

```
...
QualType CheckElementWiseConstraints(ExprResult &LHS, ExprResult &RHS);
...
```

# 总结

## 实验结果总结

**测试用例1：**

- 源码：

```
#pragma elementWise

void foo1(){
    int A[1000];
    int B[1000];
    int C[1000];
    C = A + B;
    C = A * B;
    C = A;
}
```

- 终端结果：

```
test result 1:
TranslationUnitDecl 0x6487e80 <<invalid sloc>>
|-TypedefDecl 0x6488360 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x64883c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x6488710 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x64887b0 <test1.c:3:1, line:10:1> foo1 'void ()'
  `-CompoundStmt 0x64b4ef0 <line:3:12, line:10:1>
    |-DeclStmt 0x6488928 <line:4:5, col:16>
    | `-VarDecl 0x64888d0 <col:5, col:15> A 'int [1000]'
    |-DeclStmt 0x64889d8 <line:5:5, col:16>
    | `-VarDecl 0x6488980 <col:5, col:15> B 'int [1000]'
    |-DeclStmt 0x6488a88 <line:6:5, col:16>
    | `-VarDecl 0x6488a30 <col:5, col:15> C 'int [1000]'
    |-BinaryOperator 0x6488b70 <line:7:5, col:13> 'int [1000]' '='
    | |-DeclRefExpr 0x6488aa0 <col:5> 'int [1000]' lvalue Var 0x6488a30 'C' 'int [1000]'
    | `-BinaryOperator 0x6488b48 <col:9, col:13> 'int [1000]' '+'
    |   |-ImplicitCastExpr 0x6488b18 <col:9> 'int [1000]' <LValueToRValue>
    |   | `-DeclRefExpr 0x6488ac8 <col:9> 'int [1000]' lvalue Var 0x64888d0 'A' 'int [1000]'
    |   `-ImplicitCastExpr 0x6488b30 <col:13> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x6488af0 <col:13> 'int [1000]' lvalue Var 0x6488980 'B' 'int [1000]'
    |-BinaryOperator 0x64b4e50 <line:8:5, col:13> 'int [1000]' '='
    | |-DeclRefExpr 0x64b4d80 <col:5> 'int [1000]' lvalue Var 0x6488a30 'C' 'int [1000]'
    | `-BinaryOperator 0x64b4e28 <col:9, col:13> 'int [1000]' '*'
    |   |-ImplicitCastExpr 0x64b4df8 <col:9> 'int [1000]' <LValueToRValue>
    |   | `-DeclRefExpr 0x64b4da8 <col:9> 'int [1000]' lvalue Var 0x64888d0 'A' 'int [1000]'
    |   `-ImplicitCastExpr 0x64b4e10 <col:13> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x64b4dd0 <col:13> 'int [1000]' lvalue Var 0x6488980 'B' 'int [1000]'
    `-BinaryOperator 0x64b4ec8 <line:9:5, col:9> 'int [1000]' '='
      |-DeclRefExpr 0x64b4e78 <col:5> 'int [1000]' lvalue Var 0x6488a30 'C' 'int [1000]'
      `-DeclRefExpr 0x64b4ea0 <col:9> 'int [1000]' lvalue Var 0x64888d0 'A' 'int [1000]'
0
```

- 原因分析：

符合要求

**测试用例2：**

- 源码：

```
void foo1(){
    int A[1000];
    int B[1000];
    int C[1000];
    C = A + B;
    C = A * B;
    C = A;
}
```

- 终端结果：

```
test result 2:
test2.c:6:11: error: invalid operands to binary expression ('int *' and 'int *')
    C = A + B;
        ~ ^ ~
test2.c:7:11: error: invalid operands to binary expression ('int *' and 'int *')
    C = A * B;
        ~ ^ ~
test2.c:8:7: error: array type 'int [1000]' is not assignable
    C = A;
    ~ ^
TranslationUnitDecl 0x53c2e80 <<invalid sloc>>
|-TypedefDecl 0x53c3360 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x53c33c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x53c3710 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x53c37b0 <test2.c:2:1, line:9:1> foo1 'void ()'
  `-CompoundStmt 0x53f0178 <line:2:12, line:9:1>
    |-DeclStmt 0x53c3928 <line:3:5, col:16>
    | `-VarDecl 0x53c38d0 <col:5, col:15> A 'int [1000]'
    |-DeclStmt 0x53c39d8 <line:4:5, col:16>
    | `-VarDecl 0x53c3980 <col:5, col:15> B 'int [1000]'
    `-DeclStmt 0x53c3a88 <line:5:5, col:16>
      `-VarDecl 0x53c3a30 <col:5, col:15> C 'int [1000]'
3 errors generated.
1
```

- 原因分析：

不符合要求。没有elementWise标记。

**测试用例3：**

- 源码：

```
#pragma elementWise
void foo3(){
  int A[1000];
  int B[1000];
  int C[1000];
  int *D;
  C = D;
}
```

- 终端结果：

```
test result 3:
test3.c:7:5: error: assigning to 'int [1000]' from incompatible type 'int *'
  C = D;
    ^ ~
TranslationUnitDecl 0x6d40e80 <<invalid sloc>>
|-TypedefDecl 0x6d41360 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x6d413c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x6d41710 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x6d417b0 <test3.c:2:1, line:8:1> foo3 'void ()'
  `-CompoundStmt 0x6d6e0c0 <line:2:12, line:8:1>
    |-DeclStmt 0x6d41928 <line:3:3, col:14>
    | `-VarDecl 0x6d418d0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x6d419d8 <line:4:3, col:14>
    | `-VarDecl 0x6d41980 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x6d41a88 <line:5:3, col:14>
    | `-VarDecl 0x6d41a30 <col:3, col:13> C 'int [1000]'
    `-DeclStmt 0x6d41b38 <line:6:3, col:9>
      `-VarDecl 0x6d41ae0 <col:3, col:8> D 'int *'
1 error generated.
1
```

- 原因分析：

不符合要求。等号右边是指针类型，左边是数组类型，类型不匹配。

**测试用例4：**

- 源码：

```
#pragma elementWise
void foo4(){
  int A[1000];
  int B[1000];
  int C[1000];
  int *D;
  (A + B) = C;
}
```

- 终端结果：

```
test result 4:
test4.c:7:11: error: expression is not assignable
  (A + B) = C;
  ~~~~~~~ ^
TranslationUnitDecl 0x544be80 <<invalid sloc>>
|-TypedefDecl 0x544c360 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x544c3c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x544c710 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x544c7b0 <test4.c:2:1, line:8:1> foo4 'void ()'
  `-CompoundStmt 0x5478e00 <line:2:12, line:8:1>
    |-DeclStmt 0x544c928 <line:3:3, col:14>
    | `-VarDecl 0x544c8d0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x544c9d8 <line:4:3, col:14>
    | `-VarDecl 0x544c980 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x544ca88 <line:5:3, col:14>
    | `-VarDecl 0x544ca30 <col:3, col:13> C 'int [1000]'
    `-DeclStmt 0x544cb38 <line:6:3, col:9>
      `-VarDecl 0x544cae0 <col:3, col:8> D 'int *'
1 error generated.
1
```

- 原因分析：

不符合要求。等号左边的（A+B)是右值不是左值。

**测试用例5：**

- 源码：

```
#pragma elementWise
void foo5(){
  int A[1000];
  int B[1000];
  int C[1000];
  int *D;
  C = A + D;
  C = D + A;
  C = D + D;
}
```

- 终端结果：

```
test result 5:
test5.c:7:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = A + D;
      ~ ^ ~
test5.c:8:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = D + A;
      ~ ^ ~
test5.c:9:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = D + D;
      ~ ^ ~
TranslationUnitDecl 0x5c00e80 <<invalid sloc>>
|-TypedefDecl 0x5c01360 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x5c013c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x5c01710 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x5c017b0 <test5.c:2:1, line:10:1> foo5 'void ()'
  `-CompoundStmt 0x5c2e270 <line:2:12, line:10:1>
    |-DeclStmt 0x5c01928 <line:3:3, col:14>
    | `-VarDecl 0x5c018d0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x5c019d8 <line:4:3, col:14>
    | `-VarDecl 0x5c01980 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x5c01a88 <line:5:3, col:14>
    | `-VarDecl 0x5c01a30 <col:3, col:13> C 'int [1000]'
    `-DeclStmt 0x5c01b38 <line:6:3, col:9>
      `-VarDecl 0x5c01ae0 <col:3, col:8> D 'int *'
3 errors generated.
1
```

- 原因分析：

不符合要求。等号右边是指针类型，左边是数组类型，类型不匹配。

**测试用例6：**

- 源码：

```
#pragma elementWise
void foo6(){
  int A[1000];
  int B[1000];
  int C[1000];
  C = A / B;
}
```

- 终端结果：

```
TranslationUnitDecl 0x6254f30 <<invalid sloc>>
|-TypedefDecl 0x6255410 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x6255470 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x62557c0 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x6255860 <negative/test5.c:2:1, line:10:1> foo5 'void ()'
  `-CompoundStmt 0x6282350 <line:2:12, line:10:1>
    |-DeclStmt 0x62559d8 <line:3:3, col:14>
    | `-VarDecl 0x6255980 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x6255a88 <line:4:3, col:14>
    | `-VarDecl 0x6255a30 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x6255b38 <line:5:3, col:14>
    | `-VarDecl 0x6255ae0 <col:3, col:13> C 'int [1000]'
    `-DeclStmt 0x6255be8 <line:6:3, col:9>
      `-VarDecl 0x6255b90 <col:3, col:8> D 'int *'
3 errors generated.
negative/test6.c:6:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = A / B;
      ~ ^ ~
```

- 原因分析：

不符合要求。不支持除法操作。

**测试用例7：**

- 源码：

```
#pragma elementWise
void foo7(){
  int A[1000];
  int B[1000];
  int C[1000];
  int *D;
  int E[10][100];
  E = A;
  E = A + B;
  E = A * B;
}
```

- 终端结果：

```
test result 7:
test7.c:8:5: error: assigning to 'int [10][100]' from incompatible type 'int [1000]'
  E = A;
    ^ ~
test7.c:9:5: error: assigning to 'int [10][100]' from incompatible type 'int [1000]'
  E = A + B;
    ^ ~~~~~
test7.c:10:5: error: assigning to 'int [10][100]' from incompatible type 'int [1000]'
  E = A * B;
    ^ ~~~~~
TranslationUnitDecl 0x5e1ce80 <<invalid sloc>>
|-TypedefDecl 0x5e1d360 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x5e1d3c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x5e1d710 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x5e1d7b0 <test7.c:2:1, line:11:1> foo7 'void ()'
  `-CompoundStmt 0x5e4a3e0 <line:2:12, line:11:1>
    |-DeclStmt 0x5e1d928 <line:3:3, col:14>
    | `-VarDecl 0x5e1d8d0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x5e1d9d8 <line:4:3, col:14>
    | `-VarDecl 0x5e1d980 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x5e1da88 <line:5:3, col:14>
    | `-VarDecl 0x5e1da30 <col:3, col:13> C 'int [1000]'
    |-DeclStmt 0x5e1db38 <line:6:3, col:9>
    | `-VarDecl 0x5e1dae0 <col:3, col:8> D 'int *'
    `-DeclStmt 0x5e49e88 <line:7:3, col:17>
      `-VarDecl 0x5e49e30 <col:3, col:16> E 'int [10][100]'
3 errors generated.
1
```

- 原因分析：

不符合要求。等号左边应该为一维数组类型。

**测试用例8：**

- 源码：

```
#pragma elementWise
void foo8(){
  int A[1000];
  int B[1000];
  const int C[1000];
  C = A;
  C = A + B;
}
```

- 终端结果：

```
test result 8:
test8.c:6:5: error: read-only variable is not assignable
  C = A;
  ~ ^
test8.c:7:5: error: read-only variable is not assignable
  C = A + B;
  ~ ^
TranslationUnitDecl 0x6990e80 <<invalid sloc>>
|-TypedefDecl 0x6991360 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x69913c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x6991710 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x69917b0 <test8.c:2:1, line:8:1> foo8 'void ()'
  `-CompoundStmt 0x69bddb8 <line:2:12, line:8:1>
    |-DeclStmt 0x6991928 <line:3:3, col:14>
    | `-VarDecl 0x69918d0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x69919d8 <line:4:3, col:14>
    | `-VarDecl 0x6991980 <col:3, col:13> B 'int [1000]'
    `-DeclStmt 0x6991ac8 <line:5:3, col:20>
      `-VarDecl 0x6991a70 <col:3, col:19> C 'const int [1000]'
2 errors generated.
1
```

- 原因分析：

不符合要求。等号左边是const类型，不是可修改的。

**测试用例9：**

- 源码：

```
#pragma elementWise
void foo9(){
  int A[1000];
  const int B[1000];
  int C[1000];
  C = B;
  C = A + B;
}
```

- 终端结果：

```
test result 9:
TranslationUnitDecl 0x5d69e80 <<invalid sloc>>
|-TypedefDecl 0x5d6a360 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x5d6a3c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x5d6a710 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x5d6a7b0 <test9.c:2:1, line:8:1> foo9 'void ()'
  `-CompoundStmt 0x5d96e08 <line:2:12, line:8:1>
    |-DeclStmt 0x5d6a928 <line:3:3, col:14>
    | `-VarDecl 0x5d6a8d0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x5d6aa18 <line:4:3, col:20>
    | `-VarDecl 0x5d6a9c0 <col:3, col:19> B 'const int [1000]'
    |-DeclStmt 0x5d6aac8 <line:5:3, col:14>
    | `-VarDecl 0x5d6aa70 <col:3, col:13> C 'int [1000]'
    |-BinaryOperator 0x5d6ab30 <line:6:3, col:7> 'int [1000]' '='
    | |-DeclRefExpr 0x5d6aae0 <col:3> 'int [1000]' lvalue Var 0x5d6aa70 'C' 'int [1000]'
    | `-DeclRefExpr 0x5d6ab08 <col:7> 'const int [1000]' lvalue Var 0x5d6a9c0 'B' 'const int [1000]'
    `-BinaryOperator 0x5d96de0 <line:7:3, col:11> 'int [1000]' '='
      |-DeclRefExpr 0x5d6ab58 <col:3> 'int [1000]' lvalue Var 0x5d6aa70 'C' 'int [1000]'
      `-BinaryOperator 0x5d96db8 <col:7, col:11> 'int [1000]' '+'
        |-ImplicitCastExpr 0x5d96d88 <col:7> 'int [1000]' <LValueToRValue>
        | `-DeclRefExpr 0x5d6ab80 <col:7> 'int [1000]' lvalue Var 0x5d6a8d0 'A' 'int [1000]'
        `-ImplicitCastExpr 0x5d96da0 <col:11> 'const int [1000]' <LValueToRValue>
          `-DeclRefExpr 0x5d96d60 <col:11> 'const int [1000]' lvalue Var 0x5d6a9c0 'B' 'const int [1000]'
0
```

- 原因分析：

符合要求。

**测试用例10：**

- 源码：

```
#pragma elementWise
void foo10(){
  int A[1000];
  int B[1000];
  int C[1000];
  int D[1000];
  D = A + B + C;
  D = A * B + C;
  D = (D = A + B);
  D = (A + B) * C;
  D = (A + B) * (C + D);
}
```

- 终端结果：

```
test result 10:
TranslationUnitDecl 0x5b7be80 <<invalid sloc>>
|-TypedefDecl 0x5b7c360 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x5b7c3c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x5b7c710 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x5b7c7b0 <test10.c:2:1, line:12:1> foo10 'void ()'
  `-CompoundStmt 0x5ba9520 <line:2:13, line:12:1>
    |-DeclStmt 0x5b7c928 <line:3:3, col:14>
    | `-VarDecl 0x5b7c8d0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x5b7c9d8 <line:4:3, col:14>
    | `-VarDecl 0x5b7c980 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x5b7ca88 <line:5:3, col:14>
    | `-VarDecl 0x5b7ca30 <col:3, col:13> C 'int [1000]'
    |-DeclStmt 0x5b7cb38 <line:6:3, col:14>
    | `-VarDecl 0x5b7cae0 <col:3, col:13> D 'int [1000]'
    |-BinaryOperator 0x5ba8ea8 <line:7:3, col:15> 'int [1000]' '='
    | |-DeclRefExpr 0x5b7cb50 <col:3> 'int [1000]' lvalue Var 0x5b7cae0 'D' 'int [1000]'
    | `-BinaryOperator 0x5ba8e80 <col:7, col:15> 'int [1000]' '+'
    |   |-BinaryOperator 0x5ba8e18 <col:7, col:11> 'int [1000]' '+'
    |   | |-ImplicitCastExpr 0x5ba8de8 <col:7> 'int [1000]' <LValueToRValue>
    |   | | `-DeclRefExpr 0x5b7cb78 <col:7> 'int [1000]' lvalue Var 0x5b7c8d0 'A' 'int [1000]'
    |   | `-ImplicitCastExpr 0x5ba8e00 <col:11> 'int [1000]' <LValueToRValue>
    |   |   `-DeclRefExpr 0x5ba8dc0 <col:11> 'int [1000]' lvalue Var 0x5b7c980 'B' 'int [1000]'
    |   `-ImplicitCastExpr 0x5ba8e68 <col:15> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x5ba8e40 <col:15> 'int [1000]' lvalue Var 0x5b7ca30 'C' 'int [1000]'
    |-BinaryOperator 0x5ba9008 <line:8:3, col:15> 'int [1000]' '='
    | |-DeclRefExpr 0x5ba8ed0 <col:3> 'int [1000]' lvalue Var 0x5b7cae0 'D' 'int [1000]'
    | `-BinaryOperator 0x5ba8fe0 <col:7, col:15> 'int [1000]' '+'
    |   |-BinaryOperator 0x5ba8f78 <col:7, col:11> 'int [1000]' '*'
    |   | |-ImplicitCastExpr 0x5ba8f48 <col:7> 'int [1000]' <LValueToRValue>
    |   | | `-DeclRefExpr 0x5ba8ef8 <col:7> 'int [1000]' lvalue Var 0x5b7c8d0 'A' 'int [1000]'
    |   | `-ImplicitCastExpr 0x5ba8f60 <col:11> 'int [1000]' <LValueToRValue>
    |   |   `-DeclRefExpr 0x5ba8f20 <col:11> 'int [1000]' lvalue Var 0x5b7c980 'B' 'int [1000]'
    |   `-ImplicitCastExpr 0x5ba8fc8 <col:15> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x5ba8fa0 <col:15> 'int [1000]' lvalue Var 0x5b7ca30 'C' 'int [1000]'
    |-BinaryOperator 0x5ba9170 <line:9:3, col:17> 'int [1000]' '='
    | |-DeclRefExpr 0x5ba9030 <col:3> 'int [1000]' lvalue Var 0x5b7cae0 'D' 'int [1000]'
    | `-ParenExpr 0x5ba9150 <col:7, col:17> 'int [1000]'
    |   `-BinaryOperator 0x5ba9128 <col:8, col:16> 'int [1000]' '='
    |     |-DeclRefExpr 0x5ba9058 <col:8> 'int [1000]' lvalue Var 0x5b7cae0 'D' 'int [1000]'
    |     `-BinaryOperator 0x5ba9100 <col:12, col:16> 'int [1000]' '+'
    |       |-ImplicitCastExpr 0x5ba90d0 <col:12> 'int [1000]' <LValueToRValue>
    |       | `-DeclRefExpr 0x5ba9080 <col:12> 'int [1000]' lvalue Var 0x5b7c8d0 'A' 'int [1000]'
    |       `-ImplicitCastExpr 0x5ba90e8 <col:16> 'int [1000]' <LValueToRValue>
    |         `-DeclRefExpr 0x5ba90a8 <col:16> 'int [1000]' lvalue Var 0x5b7c980 'B' 'int [1000]'
    |-BinaryOperator 0x5ba92f0 <line:10:3, col:17> 'int [1000]' '='
    | |-DeclRefExpr 0x5ba9198 <col:3> 'int [1000]' lvalue Var 0x5b7cae0 'D' 'int [1000]'
    | `-BinaryOperator 0x5ba92c8 <col:7, col:17> 'int [1000]' '*'
    |   |-ParenExpr 0x5ba9268 <col:7, col:13> 'int [1000]'
    |   | `-BinaryOperator 0x5ba9240 <col:8, col:12> 'int [1000]' '+'
    |   |   |-ImplicitCastExpr 0x5ba9210 <col:8> 'int [1000]' <LValueToRValue>
    |   |   | `-DeclRefExpr 0x5ba91c0 <col:8> 'int [1000]' lvalue Var 0x5b7c8d0 'A' 'int [1000]'
    |   |   `-ImplicitCastExpr 0x5ba9228 <col:12> 'int [1000]' <LValueToRValue>
    |   |     `-DeclRefExpr 0x5ba91e8 <col:12> 'int [1000]' lvalue Var 0x5b7c980 'B' 'int [1000]'
    |   `-ImplicitCastExpr 0x5ba92b0 <col:17> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x5ba9288 <col:17> 'int [1000]' lvalue Var 0x5b7ca30 'C' 'int [1000]'
    `-BinaryOperator 0x5ba94f8 <line:11:3, col:23> 'int [1000]' '='
      |-DeclRefExpr 0x5ba9318 <col:3> 'int [1000]' lvalue Var 0x5b7cae0 'D' 'int [1000]'
      `-BinaryOperator 0x5ba94d0 <col:7, col:23> 'int [1000]' '*'
        |-ParenExpr 0x5ba93e8 <col:7, col:13> 'int [1000]'
        | `-BinaryOperator 0x5ba93c0 <col:8, col:12> 'int [1000]' '+'
        |   |-ImplicitCastExpr 0x5ba9390 <col:8> 'int [1000]' <LValueToRValue>
        |   | `-DeclRefExpr 0x5ba9340 <col:8> 'int [1000]' lvalue Var 0x5b7c8d0 'A' 'int [1000]'
        |   `-ImplicitCastExpr 0x5ba93a8 <col:12> 'int [1000]' <LValueToRValue>
        |     `-DeclRefExpr 0x5ba9368 <col:12> 'int [1000]' lvalue Var 0x5b7c980 'B' 'int [1000]'
        `-ParenExpr 0x5ba94b0 <col:17, col:23> 'int [1000]'
          `-BinaryOperator 0x5ba9488 <col:18, col:22> 'int [1000]' '+'
            |-ImplicitCastExpr 0x5ba9458 <col:18> 'int [1000]' <LValueToRValue>
            | `-DeclRefExpr 0x5ba9408 <col:18> 'int [1000]' lvalue Var 0x5b7ca30 'C' 'int [1000]'
            `-ImplicitCastExpr 0x5ba9470 <col:22> 'int [1000]' <LValueToRValue>
              `-DeclRefExpr 0x5ba9430 <col:22> 'int [1000]' lvalue Var 0x5b7cae0 'D' 'int [1000]'
0
```

- 原因分析：

符合要求。支持复合表达式。

**测试用例11：**

- 源码：

```
#pragma elementWise
void foo10(){
  int A[1000];
  int B[1000];
  int C[1000];
  C = A - B;
}
```

- 终端结果：

```
TranslationUnitDecl 0x6955f30 <<invalid sloc>>
|-TypedefDecl 0x6956410 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x6956470 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x69567c0 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x6956860 <positive/test9.c:2:1, line:8:1> foo9 'void ()'
  `-CompoundStmt 0x6982f10 <line:2:12, line:8:1>
    |-DeclStmt 0x69569d8 <line:3:3, col:14>
    | `-VarDecl 0x6956980 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x6956ac8 <line:4:3, col:20>
    | `-VarDecl 0x6956a70 <col:3, col:19> B 'const int [1000]'
    |-DeclStmt 0x6956b78 <line:5:3, col:14>
    | `-VarDecl 0x6956b20 <col:3, col:13> C 'int [1000]'
    |-BinaryOperator 0x6956bf8 <line:6:3, col:7> 'int [1000]' '='
    | |-DeclRefExpr 0x6956b90 <col:3> 'int [1000]' lvalue Var 0x6956b20 'C' 'int [1000]'
    | `-ImplicitCastExpr 0x6956be0 <col:7> 'const int [1000]' <LValueToRValue>
    |   `-DeclRefExpr 0x6956bb8 <col:7> 'const int [1000]' lvalue Var 0x6956a70 'B' 'const int [1000]'
    `-BinaryOperator 0x6982ee8 <line:7:3, col:11> 'int [1000]' '='
      |-DeclRefExpr 0x6956c20 <col:3> 'int [1000]' lvalue Var 0x6956b20 'C' 'int [1000]'
      `-BinaryOperator 0x6982ec0 <col:7, col:11> 'int [1000]' '+'
        |-ImplicitCastExpr 0x6982e90 <col:7> 'int [1000]' <LValueToRValue>
        | `-DeclRefExpr 0x6982e40 <col:7> 'int [1000]' lvalue Var 0x6956980 'A' 'int [1000]'
        `-ImplicitCastExpr 0x6982ea8 <col:11> 'const int [1000]' <LValueToRValue>
          `-DeclRefExpr 0x6982e68 <col:11> 'const int [1000]' lvalue Var 0x6956a70 'B' 'const int [1000]'
test at positive pass!!
negative/test11.c:6:5: error: assigning to 'int [1000]' from incompatible type 'long'
  C = A - B;
```

- 原因分析：

不符合要求。不支持减号。此时等号右边是指针相减，结果为long类型，与左边类型不匹配。

**测试用例12：**

- 源码：

```
#pragma elementWise
void foo10(){
  int A[1000];
  A =;
  A +;
  + A;
  = A;
}
```

- 终端结果：

```
test result 12:
test12.c:4:6: error: expected expression
  A =;
     ^
test12.c:5:6: error: expected expression
  A +;
     ^
test12.c:6:3: error: invalid argument type 'int *' to unary expression
  + A;
  ^ ~
test12.c:7:3: error: expected expression
  = A;
  ^
TranslationUnitDecl 0x543ce80 <<invalid sloc>>
|-TypedefDecl 0x543d360 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x543d3c0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x543d710 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x543d7b0 <test12.c:2:1, line:8:1> foo10 'void ()'
  `-CompoundStmt 0x5469f48 <line:2:13, line:8:1>
    `-DeclStmt 0x543d928 <line:3:3, col:14>
      `-VarDecl 0x543d8d0 <col:3, col:13> A 'int [1000]'
4 errors generated.
1
```

- 原因分析：

不符合要求。二元操作符缺少左右表达式。

**测试用例13：**

- 源码：

```
#pragma elementWise
typedef int s32;
s32 a[100], b[100], c[100];
int e[100];
void f(){
    e = b + (c+e)*a;
}
```

- 原因分析：

符合要求。经过typedef的s32类型在进行elementWise的乘、加、赋值运算时应被视为int类型。类型匹配。

**测试用例14：**

- 源码：

```
#pragma elementWise
int a[100],b[100];
int* e;
void g(){
    e=a+b;
}
```

- 终端结果：

```
TranslationUnitDecl 0x5f5ee70 <<invalid sloc>>
```

```
|-TypedefDecl 0x5f5f350 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x5f5f3b0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x5f5f700 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
|-VarDecl 0x5f5f7d0 <test2.c:2:1, col:10> a 'int [100]'
|-VarDecl 0x5f5f870 <col:1, col:17> b 'int [100]'
|-VarDecl 0x5f5f920 <line:3:1, col:6> e 'int *'
`-FunctionDecl 0x5f5f9d0 <line:4:1, line:6:1> g 'void ()'
  `-CompoundStmt 0x5f5fb80 <line:4:9, line:6:1>
    `-BinaryOperator 0x5f5fb58 <line:5:5, col:10> 'int *' '='
      |-DeclRefExpr 0x5f5fa70 <col:5> 'int *' lvalue Var 0x5f5f920 'e' 'int *'
      `-ImplicitCastExpr 0x5f5fb40 <col:8, col:10> 'int *' <ArrayToPointerDecay>
        `-BinaryOperator 0x5f5fb18 <col:8, col:10> 'int [100]' '+'
          |-ImplicitCastExpr 0x5f5fae8 <col:8> 'int [100]' <LValueToRValue>
          | `-DeclRefExpr 0x5f5fa98 <col:8> 'int [100]' lvalue Var 0x5f5f7d0 'a'
'int [100]'
          `-ImplicitCastExpr 0x5f5fb00 <col:10> 'int [100]' <LValueToRValue>
            `-DeclRefExpr 0x5f5fac0 <col:10> 'int [100]' lvalue Var 0x5f5f870
'b' 'int [100]'
0
```

- 原因分析：

符合要求。等号右边经过elementWise运算后是int[100]类型，且为右值。根据C99 6.3.2.1p3：

```
Except when it is the operand of the sizeof operator or the unary & operator, or
is a string literal used to initialize an array, an expression that has type
‘‘array of type’’ is converted to an expression with type ‘‘pointer to type’’
that points to the initial element of the array object and is not an lvalue. If
the array object has register storage class, the behavior is undefined.
```

右边被隐式转换为int*类型，指向其第一个元素的地址。若右边是register类(比如register int d[100])，则行为是未定义的。

左边也是int* 类型，类型匹配。

# 分成员总结

## 陈彦帆

进行了一些测试，针对测试修改了代码中一系列不符合要求的bug。

## 骆嵩涛

主要负责本次报告的书写，同时对于代码加入一定的注释，设计了个别用例。

本次实验代码量不大，修改思路也清晰明了，本什么整个实验的目的其实是为了让我们理解llvm整个编译框架的整体思路。所在在这次实验主要了解到了语法书的生成中如何判断类型的不匹配，大小不匹配，操作数不匹配的等问题，最终生成对应的报错。

## 毛啸腾

撰写本次代码