(CLANG) WPARENTHESES 的一个 BUG

最近在看一点编译器的东西，感觉很神奇，本来想自己实现一个玩具的编译器，但是还是因为时间不足所以暂时搁置了，就一直在看一点有关 `LLVM` 的科普文。`LLVM` 真是个神奇的东西，这里推荐一个有趣教程 Kaleidoscope-Implementing a Language with LLVM，从简单的 `Lexer` 到 `AST` 再到后面的 `JIT` 都搞得十分有趣，特别值得一看。

今天在逛论坛时，看到了有人在 `2014` 年报了这样一个 `BUG`，原贴在Bug 18971 - Missing -Wparentheses warning

> (Update 2018/06/04) 我已经提交了 `PATCH`，目前等待有人帮我 `commit`: (fix: (Bug 18971) - Missing -Wparentheses warning)(https://reviews.llvm.org/D47687)

```
$ cat tmp/warning/a.cc
#include <cassert>
bool x;
int val;
void foo() {
  assert(x && val == 4 || (!x && val == 5));
}

$ gcc-4.8.2-inst/bin/g++ -c tmp/warning/a.cc -Wall -o a.o
In file included from /gcc-4.8.2-inst/include/c++/4.8.2/cassert:43:0,
                 from tmp/warning/a.cc:1:
tmp/warning/a.cc: In function ‘void foo()’:
tmp/warning/a.cc:5:12: warning: suggest parentheses around ‘&&’ within
‘||’ [-Wparentheses]
   assert(x && val == 4 || (!x && val == 5));
            ^
$ ./bin/clang++ -c tmp/warning/a.cc -Wall -o a.o
```

大致意思是因为在 `C++` 中 `&&` 的优先级要高于 `||`，为了让程序员留神自己的逻辑问题，所以在 `gcc` 中会给出一个 `Warning`，但是在 `clang` 中却没有给出一个 `Warning`，但其实根据我的分析，`clang` 理论上是会给出一个 `Warning`，比如我们这样来测试

```
void foo(bool b) {
  /* do nothing */
}

foo(x && val == 4 || (!x && val == 5));
```

这个的确是会抛出一个 `Warning`，而且很明确的建议我们加括号来强调一下我们的逻辑

```
a.cc:20:9: warning: '&&' within '||' [-Wlogical-op-parentheses]
  foo(x && val == 4 || (!x && val == 5));
      ~~^~~~~~~~~~~ ~~
a.cc:20:9: note: place parentheses around the '&&' expression to silence
this
      warning
  foo(x && val == 4 || (!x && val == 5));
        ^
      (             )
1 warning generated.
```

这就有点奇怪了，理论上应该是可以抛出一个 `Warning` 的呀！于是去看了下 `clang` 的这部分代码。我找到这部分比较关键的代码在 `clang/lib/Sema/SemaExpr.cpp`

```
/// DiagnoseBinOpPrecedence - Emit warnings for expressions with tricky
/// precedence.
static void DiagnoseBinOpPrecedence(Sema &Self, BinaryOperatorKind Opc,
                                    SourceLocation OpLoc, Expr *LHSExpr,
                                    Expr *RHSExpr){
//// Some codes ...

  // Warn about arg1 || arg2 && arg3, as GCC 4.3+ does.
  // We don't warn for 'assert(a || b && "bad")' since this is safe.
  if (Opc == BO_LOr && !OpLoc.isMacroID()/* Don't warn in macros. */) {
    DiagnoseLogicalAndInLogicalOrLHS(Self, OpLoc, LHSExpr, RHSExpr);
    DiagnoseLogicalAndInLogicalOrRHS(Self, OpLoc, LHSExpr, RHSExpr);
  }

//// Some other codes
}
```

注意到判断 `&&` 是否被包含于 `||` 的函数中的 `if (Opc == BO_LOr && !OpLoc.isMacroID()/* Don't warn in macros. */)` 也就是说，如果这玩意是在 `macro` 中，它就不会产生一个 `Warning` 了。而 `assert()` 函数的确是一个 `macro`，它的声明在这里 可以查看。

```
#ifdef NDEBUG
#define assert(condition) ((void)0)
#else
#define assert(condition) /*implementation defined*/
#endif
```

也就是说 `assert` 是一个在 `<cassert>` 中的一个 `macro`...这下就解释的通了，下面验证了一下，把一个函数定义到 `macro` 中，像这样

```
#define bar(x) \
  ( \
    ( std::cout << x ) \
  )

bar(x && val == 4 || (!x && val == 5));
```

这下，我们的 `bar()` 也跟 `assert()` 一样了，编译并不会给我们 `Warning` 了。

Solution

解决办法呢，很简单，~~删掉源码的 `&& !OpLoc.isMacroID()` 就好了。看了下记录好像是 `2010` 年左右的一个没有经过 `review` 直接 `commit` 的一段代码...~~

貌似是因为在编译一些 `function-like-macros` 时，有一些不合适的位置产生了这些 `warning` ，十分烦人，所以被B掉了... 比如说

```
#define VOID_(op0, op1, x, y, z) ( (void)(x op0 y op1 z) )
```

我的解决方法是，判断操作数与操作符的来源，比如说：

> 操作符和操作数都来自同一个位置的宏参数，那么这个表达式你是可以自己判断优先级并添加合适的括号来防止误判。所以编译器是可以产生一个 `warning` 来提醒你

```
VOID_(&&, ||, x && y || z, y, z)
```

因为你可以自己决定操作数的结合方式添加合适的括号，比如

```
VOID_(&&, ||, (x && y || z), y, z)
```

> 操作符和操作数没有来自同一个位置的宏参数，那么这个表达式是不应该直接的被武断的认为是逻辑问题，因为操作符位置是可以使用其他的操作符的，并不一定是逻辑操作符，所以编译器不应该抛出烦人的 `warning`

```
VOID_(&&, ||, x, y, z)
```

因为如果想添加括号，就只能在函数宏的函数体内部加，像这样

```
#define VOID_(op0, op1, x, y, z) ( (void)((x op0 y) op1 z) )
```

但是这样对复用就产生了影响，比如 `VOID_(+, *, 1, 2, 3)` 就会被展开为 `(1 + 2) * 3`，而你可能只想要 `1 + 2 * 3`

所以，解决方法就是判断操作符和操作数的位置，修改一下 `clang` 判断是否生成 `warning` 的逻辑

```
if ((OR操作符来自宏参数 && 左操作数来自宏参数) ||
    (OR操作符来自宏参数 && 右操作数来自宏参数)) {
  if (OR操作符的参数位置 == 左操作数的参数位置) {
    检查左操作数
  }
  if (OR操作符的参数位置 == 右操作数的参数位置) {
    检查右表达式
  }
}
```

```
/// DiagnoseBinOpPrecedence - Emit warnings for expressions with tricky
/// precedence.
static void DiagnoseBinOpPrecedence(Sema &Self, BinaryOperatorKind Opc,
                                    SourceLocation OpLoc, Expr *LHSExpr,
                                    Expr *RHSExpr){
//// Some codes ...

  // Warn about arg1 || arg2 && arg3, as GCC 4.3+ does.
  // We don't warn for 'assert(a || b && "bad")' since this is safe.
  if (Opc == BO_LOr) {
    if (!OpLoc.isMacroID()) {
      // Operator is not in macros
      DiagnoseLogicalAndInLogicalOrLHS(Self, OpLoc, LHSExpr, RHSExpr);
      DiagnoseLogicalAndInLogicalOrRHS(Self, OpLoc, LHSExpr, RHSExpr);
    } else {
      // This Expression is expanded from macro
      SourceLocation LHSExpansionLoc, OpExpansionLoc, RHSExpansionLoc;
      if ((Self.getSourceManager().isMacroArgExpansion(OpLoc,
                                                       &OpExpansionLoc)
&&
           Self.getSourceManager().isMacroArgExpansion(LHSExpr-
>getExprLoc(),
                                                       &LHSExpansionLoc))
||
           (Self.getSourceManager().isMacroArgExpansion(OpLoc,
                                                       &OpExpansionLoc)
&&
```

```
          Self.getSourceManager().isMacroArgExpansion(RHSExpr-
>getExprLoc(),

 &LHSExpansionLoc))) {
        if (OpExpansionLoc == LHSExpansionLoc) {
          DiagnoseLogicalAndInLogicalOrLHS(Self, OpLoc, LHSExpr,
RHSExpr);
        }
        if (OpExpansionLoc == RHSExpansionLoc) {
          DiagnoseLogicalAndInLogicalOrRHS(Self, OpLoc, LHSExpr,
RHSExpr);
        }
      }
    }
  }
//// Some other codes
}
```

Reference

(0)(Bug 18971 - Missing -Wparentheses warning)(https://bugs.llvm.org/show_bug.cgi?id=18971)

(1)(assert - cppreference)(http://en.cppreference.com/w/cpp/error/assert)