编译原理实验6&7 实验报告

1120180488 王梓丞

实验目的

1. 熟悉 C 语言的语义规则，了解编译器语义分析的主要功能；
2. 掌握语义分析模块构造的相关技术和方法，设计并实现具有一定分析功能的 C 语言语义分析模块；
3. 掌握编译器从前端到后端各个模块的工作原理，语义分析模块与其他模块之间的交互过程。
4. 掌握语义分析模块构造的相关技术和方法，设计并实现具有一定分析功能的 C 语言语义分析模块；
5. 掌握编译器从前端到后端各个模块的工作原理，语义分析模块与其他模块之间的交互过程。

实验内容

对于语义分析，将语法分析所得结果进行语义分析并生成对应的符号表。如果存在非法的结果，请将结果报告给用户，其中语义检查的内容主要包括：

1. 未定义检测
2. 重复定义
3. 未在循环内使用break
4. 函数调用参数不匹配
5. 操作类型不匹配
6. 数组越界
7. goto标签不存在
8. 函数缺少return

对于中间代码生成，以自行完成的语义分析阶段的抽象语法树为输入，或者以 BIT-MiniCC的语 义分析阶段的抽象语法树为输入，针对不同的语句类型，将其翻译为中间代码序列。

实验过程

本次实验的实现将6和7合并为一个实验——即在生成中间代码的同时进行错误检查。

在上一次的实验中，我们生成了一个AST（抽象语法树），因此本次实验将在上次实验的基础上，利用语法分析结束后获得的AST，即对`ASTCompilationUnit`进行信息收集，上层节点通过下层节点的信息，利用综合属性生成代码——即S属性文法，收集信息以生成`LLVM-IR`。

LLVM-IR

本次实验的中间代码表示使用了`LLVM-IR`，一条`LLVM`指令通常是三操作数的形式出现的，事实上在使用上与四元式并没有显著的区别，在实验过程中需要注意的问题主要有：

- C语言类型到`LLVM`类型的映射
- 基本块的标号和虚拟寄存器的数值的关系

`LLVM`使用了`SSA`的机制使得寄存器的值只能用一次，通过顺序增加寄存器的值降低了生成中间代码的难度，但由于标号与寄存器值同样是递增的，因此控制语句对基本块编号的改变增加了复杂性，其原则如下：

基本块的组成：

- 开头的标签
- 一系列指令

- 结尾终结指令

终结指令：指改变程序执行顺序的指令：

- `ret`
- `br`

如果一个块没有基本块，就会自动赋一个标签（比如进入一个函数后，通常`%0`寄存器是用不了的，是因为`0`赋给了这个基本块，因此此时标号要递增）

由于基本块的引入，使得`LLVM`的中间代码表示经常会出现如下的一些无意义的顺序跳转（只跳转一条语句）：

```
call void @f2()
br label %37
37:
br label %38
38:
%39 = alloca i32
store i32 0, i32* %39
br label %40
40:
%41 = load i32, i32* %39
```

因此为了满足`LLVM`的需求，在每次插入跳转语句的时候都需要做额外的判断以产生合法的基本块标号。

符号表的建立

在符号表的建立上，为了对作用域进行支持，同时拥有较快的查表速度，采用了哈希表的方法建立符号表，不同作用域的符号表以栈的方式组合，即：

`private Stack<HashMap<String, IdentifierSymbol>> SymbolTableStack;`

每一个符号表是一个由`（符号名字,IdentifierSymbol）`组成的键值对，`IdentifierSymbol`中存了变量加载所需的各种信息，如地址等。

通过使用哈希表的栈，使得全局变量永远在栈底部，可以通过

```
 SymbolTableStack.firstElement().put(global_name, new IdentifierSymbol("@" +
global_name, type));
```

的方法很容易的得到全局变量。

而对于局部变量，如果需要在当前作用域判断一个变量是否**重复定义**，则从栈顶的表检测即可。

在进入作用域时，向栈中推入一个新表；在退出作用域时，将顶层符号表推出。这样就保证了作用域访问的局部性。

需要访问变量时，使用从栈顶到栈底遍历的访问顺序，使得变量总是优先从当前作用域取值，如果遍历到栈底后仍然未取到所需的值，即为**变量未声明使用**。

考察语法树的顶层节点：

```
Program -> FunctionDefine | Declaration
```

我们会发现整个源程序最外层语句可能由`FunctionDefine`或`Declaration`节点组成，即全局变量声明以及函数声明。`Declaration`和`FunctionDefine`将向符号表添加记录。其中`Declaration`负责全局变量的声明；而`FunctionDefine`的中的参数将变量添加到局部符号表中，函数名将添加到全局表中，在`FunctionDefine`内部嵌套出现的`Declaration`节点也将把对应变量添加到局部符号表中。

对于无名字符串，需要将其添加到全局变量中。

对于数组，会将其送到专门的函数`ArrayDeclaratorHandler()`进行信息收集，它将收集数组维度等信息，并将维度信息放入一个`LinkedList`中，以对后续数组访问时进行**数组越界**的检测。

Function Define

每个`FunctionDefine`都负责了一个函数内部区域的管理。在这里，将会进行局部环境的初始化并生成对应的中间代码。在初始化过程中，主要有：

- 虚拟寄存器计数的归零
- 建立新的指令缓冲区

虚拟寄存器计数是`LLVM-IR`中重要的一环，它将负责基本块标号和虚拟寄存器的递增，而每个函数内部，这个计数都是重新从0开始的，因此在每次进入新的函数时需要对其进行初始化。

对于指令缓冲区，这里采取的策略是一个函数作用域内的函数全部存入缓冲区，并在函数退出时将其打印。使用指令缓冲区的目的是方便跳转指令的反填。同时，由于指令反填的不可知性，我们很难知道一个标号是否被使用，因此我们可以通过指令缓冲区的方式在函数退出时检测**goto语句的标号是否存在。**

同时，中间代码生成类中维护一个`boolean`变量，以判断在本函数是否出现过`return`语句，以进行**函数是否有返回**的错误判断。

Statement

每个`Statement`可以认为是信息收集的基本单位，这样做的好处如下：

- `Statement`处于语法树偏上层的位置，因此较好遍历。
- `Statement`通常本身是有结构的，因此我们应该最大程度的保留这种结构信息。

`Statement`系列有以下几种情况：

- `labeledStatement`
- `compoundStatement`
- `expressionStatement`
- `selectionStatement`
- `iterationStatement`
- `jumpStatement`
    - `GotoStatement`
    - `ContinueStatement`
    - `BreakStatement`
    - `ReturnStatement`

在它们内部通常都会有跳转结构、对表达式的调用以及`Statement`间的互相调用。我们定义了`ExpressionHandler()`以对表达式结果进行收集，`StatementHandler()`进行`Statement`种类的判断并对对应信息进行收集。进入`Statement`后通过对这两个函数的调用，即可收集到各种信息。

`labeledStatement`对应于跳转的目的地址，在这里，我们需要：

- 向符号表添加一个标号
    - 如果符号表已有标号，并且标号有特别标记，则要对标号进行拉链反填
- 向指令栈中添加一个标号
    - 虽然标号并不是一个真实的指令，但是它确实是需要被打印的，因此我们依然需要将它添加到指令栈中

`Jumpstatement`中有各种跳转语句，我将其分为`GotoStatement`、`ContinueStatement`、`BreakStatement`和`ReturnStatement`进行处理。

对于`goto`，要进行目的地的判断，如果标号已经出现，并且无特殊标记，则填入`标号`，

对于`continue`语句，结构体维护了上一次循环开始的地址，为`continue`的跳转语句填入这个值。

对于`break`语句，将跳转语句在填入符号栈时并不知道目的地址，因此我选择将`break`语句推入队列，在上层`iterationStatement`进行信息收集阶段时再对其进行反填。同时，类也维护了一个对循环语句嵌套层数的计数，如果嵌套层数为0，则说明`break`语句不在循环内使用，可以进行**break循环内使用**的检查。

对于`return`语句，首先要调用`ExpressionHandler()`进行`expression`的收集，接着对`expression`的结果做出`ret`操作

同时，每次向符号栈添加一个跳转指令时，需要进行

- 上一条指令是否是跳转语句的检查
    - 如果是，则需要添加标号，并多添加一条无意义的跳转指令
- 虚拟寄存器计数自增1

通过上述步骤，以符合基本块的标准。

而对于`iterationDeclaredStatement`、`iterationStatement`、`SelectionStatement`和`compoundStatement`都应该有一个专属的局部符号表作为临时作用域，因此我们需要在每次进入这两个式子时向符号表栈中推入一个新的符号表，在退出时弹出之前的符号表。

特别的，对于`CompoundStatement`在大多数情况下的语句（除了全局变量的声明外）都处于`CompoundStatement`的`blockItems`中，所有`Statement`的调用的根源入口都是来自`FunctionDefine`的`CompoundStatement`的入口。

Expression

通过调用`ExpressionHandler()`我们可以完成对一个`Expression`的内容的收集。`ExpressionHandler()`通常会返回一个存着部分结果的虚拟寄存器。对其下的参数递归调用`ExpressionHandler()`，配合返回的寄存器，就可以完成对表达式语句中间代码的生成。

常见的寄存器分配、变量的加载等均在`ExpressionHandler()`处发生。通过对变量的加载操作，可以确定寄存器所存数的类型，从而也可以判断后续操作此寄存器的新寄存器类型的判断。在操作时对寄存器类型的记录进行检测，以完成**表达式类型相容**的语义检测。

`Expression`系列有：

- `UnaryExpression`
- `UnaryTypename`
- `BinaryExpression`
- `PostfixExpression`

函数调用也属于`Expression`的范畴，在进行函数调用时，会进行参数的判断，以检测**传入参数和函数调用不匹配**的错误。

在访问数组时，会将对应的`ASTArrayAccess`送入`ArrayAccessHandler()`函数中进行处理。考虑到多维数组的情况，传入的参数是各个维度对应的列表，通过和符号表中对应数组的信息进行比对，以完成**数组访问越界**的语义检测。

`Expression`遍历时会依赖当前局部作用域的环境，比如：

- 符号表
- 虚拟寄存器数值

另外，由于LLVM的特殊性质，条件判断生成的寄存器宽度为1，而一般运算则是寄存器的宽度。因此对于条件表达式，和一般生成数值的表达式需要采取不同的策略，在上层`SelectionStatement`或者`IterationStatement`以条件表达式用途进行调用时会对其用法进行区别。

实验结果

语义分析

1. 变量/函数未定义：

```
C:\Users\I_Rin\AppData\Local\JetBrains\Toolbox\apps\IDEA-C\ch-0\211.7142.45\jbr\bin\java.exe ...
Start to compile ...
LLVM-IR Generating..
AST Output: test/semantic_testcases/0_var_not_defined.ast.json
ES01 >> Function: f is not defined.

ES01 >> Identifier: a is not defined.

LLVM-IR Generated!
LLVM-IR Output: test/semantic_testcases/0_var_not_defined.ll
Compiling completed!

Process finished with exit code 0
```

2. 变量/函数作用域内重复定义：

```
C:\Users\I_Rin\AppData\Local\JetBrains\Toolbox\apps\IDEA-C\ch-0\211.7142.45\jbr\bin\java.exe ...
Start to compile ...
LLVM-IR Generating..
AST Output: C:\Users\I_Rin\Desktop\BIT-MiniCC\test\semantic_testcases\1_var_defined_again.ast.json
ES02 >> Function: f has been declared.

ES02 >> Declaration: a has been declared.

LLVM-IR Generated!
LLVM-IR Output: C:\Users\I_Rin\Desktop\BIT-MiniCC\test\semantic_testcases\1_var_defined_again.ll
Compiling completed!

Process finished with exit code 0
```

3. break未在循环内使用：

```
C:\Users\I_Rin\AppData\Local\JetBrains\Toolbox\apps\IDEA-C\ch-0\211.7142.45\jbr\bin\java.exe ...
Start to compile ...
LLVM-IR Generating..
AST Output: test/semantic_testcases/2_break_not_in_loop.ast.json
ES3 >> BreakStatement: must be in a LoopStatement.

ES3 >> BreakStatement: must be in a LoopStatement.

LLVM-IR Generated!
LLVM-IR Output: test/semantic_testcases/2_break_not_in_loop.ll
Compiling completed!

Process finished with exit code 0
```

4. 函数调用参数不匹配：

```
C:\Users\I_Rin\AppData\Local\JetBrains\Toolbox\apps\IDEA-C\ch-0\211.7142.45\jbr\bin\java.exe ...
Start to compile ...
LLVM-IR Generating..
AST Output: test/semantic_testcases/3_func_arg_not_match.ast.json
ES4 >> FunctionCall:f's param type is not matched.
LLVM-IR Generated!
LLVM-IR Output: test/semantic_testcases/3_func_arg_not_match.ll
Compiling completed!

Process finished with exit code 0
```

5. 表达式操作不相容：

```
C:\Users\I_Rin\AppData\Local\JetBrains\Toolbox\apps\IDEA-C\ch-0\211.7142.45\jbr\bin\java.exe ...
Start to compile ...
LLVM-IR Generating..
AST Output: test/semantic_testcases/4_opnd_not_match.ast.json
ES5 >> BinaryExpression:(<< >> & | ^) expression's should be int.

LLVM-IR Generated!
LLVM-IR Output: test/semantic_testcases/4_opnd_not_match.ll
Compiling completed!

Process finished with exit code 0
```

6. 数组访问越界：

```
C:\Users\I_Rin\AppData\Local\JetBrains\Toolbox\apps\IDEA-C\ch-0\211.7142.45\jbr\bin\java.exe ...
Start to compile ...
LLVM-IR Generating..
AST Output: test/semantic_testcases/5_arrayaccess_out_of_bounds.ast.json
ES6 >> ArrayAccess:Out of Bounds.
```

7. 标号未定义：

```
C:\Users\I_Rin\AppData\Local\JetBrains\Toolbox\apps\IDEA-C\ch-0\211.7142.45\jbr\bin\java.exe .
Start to compile ...
LLVM-IR Generating..
AST Output: test/semantic_testcases/6_goto_label_not_exist.ast.json
ES7 >> Label:end is not defined.
LLVM-IR Generated!
LLVM-IR Output: test/semantic_testcases/6_goto_label_not_exist.ll
Compiling completed!

Process finished with exit code 0
```

8. 函数无return：

```
C:\Users\I_Rin\AppData\Local\JetBrains\Toolbox\apps\IDEA-C\ch-0\211.7142.45\jbr\bin\java.exe ...
Start to compile ...
LLVM-IR Generating..
AST Output: test/semantic_testcases/7_func_lack_of_return.ast.json
ES08 >> Function: main must have a return in the end.

LLVM-IR Generated!
LLVM-IR Output: test/semantic_testcases/7_func_lack_of_return.ll
Compiling completed!

Process finished with exit code 0
```

中间代码生成

对`test.c`

部分结果如下：

`example_test.c`：

myLLVM.ll

```
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

define i32 @main() #0 {
  %1 = alloca i32
  %2 = alloca i32
  %3 = alloca i32
  store i32 0, i32* %1
  store i32 1, i32* %2
  store i32 2, i32* %3
  %4 = load i32, i32* %1
  %5 = load i32, i32* %2
  %6 = add i32 %4, %5
  %7 = load i32, i32* %3
  %8 = add i32 %7, 3
  %9 = add i32 %6, %8
  store i32 %9, i32* %3
  ret i32 0
}

```

和通过`clang -S -emit-llvm myLLVM.c -o myLLVM_Target.ll`生成的进行对比：

myLLVM.ll

```
target datalayout = "e-m:e-p270:32
target triple = "x86_64-pc-linux-g

define i32 @main() #0 {
  %1 = alloca i32
  %2 = alloca i32
  %3 = alloca i32
  store i32 0, i32* %1
  store i32 1, i32* %2
  store i32 2, i32* %3
  %4 = load i32, i32* %1
  %5 = load i32, i32* %2
  %6 = add i32 %4, %5
  %7 = load i32, i32* %3
  %8 = add i32 %7, 3
  %9 = add i32 %6, %8
  store i32 %9, i32* %3
  ret i32 0
}
```

myLLVM_Target.ll

```
source_filename = "myLLVM.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  store i32 1, i32* %3, align 4
  store i32 2, i32* %4, align 4
  %5 = load i32, i32* %2, align 4
  %6 = load i32, i32* %3, align 4
  %7 = add nsw i32 %5, %6
  %8 = load i32, i32* %4, align 4
  %9 = add nsw i32 %8, 3
  %10 = add nsw i32 %7, %9
  store i32 %10, i32* %4, align 4
  ret i32 0
}
```

`test.c`的部分，并和clang生成的中间代码进行对比：

test.ll

```
}
define i32 @main() #0 {
  %1 = alloca i32
  store i32 1, i32* %1
  %2 = alloca i32
  store i32 2, i32* %2
  %3 = alloca i32
  %4 = load i32, i32* %1
  %5 = icmp ne i32 %4, 0
  %6 = xor i1 %5, true
  %7 = zext i1 %6 to i32
  store i32 %7, i32* %3
  %8 = load i32, i32* %1
  %9 = xor i32 %8, -1
  store i32 %9, i32* %3
  %10 = load i32, i32* %1
  %11 = load i32, i32* %2
  %12 = add i32 %10, %11
  store i32 %12, i32* %3
  %13 = load i32, i32* %1
  %14 = load i32, i32* %2
  %15 = srem i32 %13, %14
  store i32 %15, i32* %3
  %16 = load i32, i32* %1
  %17 = load i32, i32* %2
  %18 = shl i32 %16, %17
  store i32 %18, i32* %3
  %19 = load i32, i32* %1
  %20 = add i32 %19, 1
  store i32 %20, i32* %1
  store i32 %20, i32* %3
  %21 = load i32, i32* %1
```

test_Target.ll

```
; Function Attrs: noinline nounwind opt
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 1, i32* %2, align 4
  store i32 2, i32* %3, align 4
  %6 = load i32, i32* %2, align 4
  %7 = icmp ne i32 %6, 0
  %8 = xor i1 %7, true
  %9 = zext i1 %8 to i32
  store i32 %9, i32* %4, align 4
  %10 = load i32, i32* %2, align 4
  %11 = xor i32 %10, -1
  store i32 %11, i32* %4, align 4
  %12 = load i32, i32* %2, align 4
  %13 = load i32, i32* %3, align 4
  %14 = add nsw i32 %12, %13
  store i32 %14, i32* %4, align 4
  %15 = load i32, i32* %2, align 4
  %16 = load i32, i32* %3, align 4
  %17 = srem i32 %15, %16
  store i32 %17, i32* %4, align 4
  %18 = load i32, i32* %2, align 4
  %19 = load i32, i32* %3, align 4
  %20 = shl i32 %18, %19
  store i32 %20, i32* %4, align 4
  %21 = load i32, i32* %2, align 4
  %22 = add nsw i32 %21, 1
  store i32 %22, i32* %2, align 4
```

由于在Linux的C标准库中并没有`Mars_PrintStr`，因此我们将对应的函数替换成`printf`

因为在具体实现上有区别，所以我的LLVM中间代码生成工具在变量值加载的顺序性与clang有所区别，但结果与控制流均为正确的。

最后的中间代码都能通过编译并运行：

```
# ubuntu @ VM-8-7-ubuntu in ~/Code/LLVM_IR [12:34:26]
$ clang ./myLLVM.ll

# ubuntu @ VM-8-7-ubuntu in ~/Code/LLVM_IR [12:35:52]
$ ./a.out
```

心得体会

本次实验在实现原理上并不难，并且与课本的实际较为贴合，但由于在实验途中我看到了`LLVM-IR`的极大潜力，因此选择支持了许多特性，并令生成的中间代码可以直接送入clang编译为目标代码：

```
目前支持情况：
语义检查：
1. 未定义检测 ✅
2. 重复定义 ✅
3. 未在循环内使用break ✅
4. 函数调用参数不匹配 ✅
5. 操作类型不匹配 ✅
6. 数组越界 ✅
7. goto标签不存在 ✅
8. 函数缺少return ✅

中间代码生成：
1. 全局变量声明 ✅
2. 局部变量声明 ✅
3. 递归作用域 ✅
4. 无名字符串 ✅
6. 函数调用 ✅
7. 运算操作 ✅
8. 循环语句 ✅
9. 选择语句 ✅
10. 跳转语句 ✅
```

这增加了许多工作，但令我受益很多。`LLVM-IR`作为一种工业界广泛应用的中间代码表示让我学到了许多新的概念，通过将指令结果存入临时的寄存器，再将这个结果送入下一条指令的方法；以及通过判定和跳转生成循环和判断结构加深了我对高级语言生成对应底层机器代码的理解。