2022秋compiler–lab2

学号: 201220154　姓名: 王紫萁*

Department of Computer Science, Nanjing University

2022 年 10 月 23 日

1　实验目标

1.1　语义分析

本次实验的目标是实现语义分析，并构造符号表。为了实现这一目标，我们需要完成以下几个任务：确定翻译模式，确定符号表结构，填表查表与错误检查。不过在此之前，还需要完成一套可复用的数据结构库，包括list，array，stack，multimap等基本数据结构

2　实验内容

2.1　实验环境

- Ubuntu 18.04.6 LTS on WSL2
- GCC version 7.5.0
- GNU Flex version 2.6.4
- GNU Bison version 3.0.4

2.2　完成的任务

基础任务和全部三个扩展任务。

2.3　确定翻译模式

在理论学习中，我们知道语法制导的翻译（SDT）可以通过在产生式中添加语义动作实现一遍扫描的翻译。然而，实验一中使用的bison采用的是LALR（look ahead LR）的语法分析。要想在LR语法分析中通过一边扫描得到各变量（variable ）的属性值，每一个变量只能是综合属性。虽然可以在语法分析阶段自底向上传递每个变量的综合属性，但这增大了实现的难度，降低了代码的组织结构的模块化，不利于维护。因此，我们选择在语法分析生成的语法树上执行语义分析。分析的过程类似L属性的SDT，用到了节点的综合属性和继承属性。

L属性的SDT适合递归下降的语法分析，而在语法树上采用深度优先处理节点和这种翻译模式类似。关于为何不考虑采用广度优先，个人认为原因有二：

1. 空间开销过高

   虽然深度优先会随着递归不断展开分配大量栈空间，但是这样的展开深度不超过树高。而广度优先最坏情况下需要分配大小与节点数相等的队列作为调度器。前者的空间复杂度是对数级别的而后者为线性的。因此从空间上考虑深度优先更有优势。

2. 需要记录的中间属性过多

   这也是最主要的原因。本节的一个重要目的是构建符号表，而为了让语义分析器可以跨作用域分析，会使用到一个栈来记录作用域信息。我们知道，深度优先与栈犹如孪生兄弟，而广度优先与队列的关系密不可分。当我们将广度优先与栈配合会出现作用域混杂不清的问题。得到的结果也是错误的。

综上所述，本次实验采用在生成的语法分析树上用类似L属性SDT的方式进行语义分析。它的理

*E-mail: ziqiwang.wayne@gmail.com

论基础是对于每一个变量F（也叫非终结符。为方便起见下文均称“变量”），我们都有

$$syn = F(inh)$$

即一个变量的综合属性由其父节点和左侧兄弟节点给予他的继承属性计算而来。落实到具体实现中，syn就是返回值+具有引用性质的形参（或者返回值是综合属性数组），继承属性作为常量形参。

2.4 确定符号表结构

为了支持跨作用域的分析。我们的符号表采用链表和哈希表结合的方式实现。其中链表串联了每一个在自己的作用域中记录的变量，hash表为加速查找提供结构支持。这里的hash表是一个MultiMap，即一个槽内可能包含多个有相同名字但作用域不同的变量。

具体实现中有三个重要的全局变量及其作用分别是:

- **SymbolTable**：用于按名索引的符号表
- **ScopeTable**：管理串联同一作用域中符号表项节点的链表
- **ScopeStack**：扫描语法树过程中记录作用域拓扑序

一个符号表项（**SymbolInfo**）目前由以下几部分构成：

- *name*：符号名
- *SymbolKind*：符号种类
  - *SYM_VAR*：变量
  - *SYM_FUNC*：函数
  - *SYM_STRUCT*：有名结构体
- *id_scope*：作用域编号
- *type*：符号类型或函数返回值类型
- *detail*：详细信息。变量名的详细信息包括：是否为函数参数，如果是的话还包括它归属的函数名。函数的详细信息包括：函数是否被定义、函数体作用域id、第一次声明/定义的位置、参数列表

每次进入一对{}组成的语句块时，都会新建一个空的**ScopeList**,并将其扩展到 ScopeTable末尾，他的`scope_id`就是该作用域在整个语义分析中的**拓扑序！**再将其压入ScopeStack，深度记录为栈当前的大小。每次离开语句块时，就将作用域从栈顶弹出。值得注意的是，结构体定义的{}语句块也形成特殊的作用域，该作用域里，我们只将在其内部定义的带名结构体的名字记录在符号表中而不将成员名加入符号表，成员名单独记录在结构体类型（Type）的FieldList 内部。这样的设计是为了防止在结构体内部出现有名结构体定义的名字冲突问题，例如

```
struct{
    struct Piggy{
        int age;
    }p;
    struct Piggy{   //redefined name
        int weight;
    }q;
}
```

2.5 填表查表与错误检查

2.5.1 填表

填表与查表本身并不难，但和作用域结合会出现很多细节上的难点。容易出问题的地方是函数+作用域以及结构体名+作用域。

1. 函数+作用域

函数定义占有两个作用域，分别是名作用域（name scope）和体作用域（body scope）。函数名的作用域属于调用者，函数体和形式参数共同构成体作用域。另外，函数声明也是一个小坑。c允许函数声明出现在函数定义中，而函数定义只能出现在顶层作用域。函数声明和局部变量的重名机制也有区别。变量重名只会在当前作用域检查，而函数声明会从当前作用域向上寻找。如果该函数被声明或者定义过，就不再将该声明加入作用域，否则就将声明加入当前作用域。c–对于未定义函数处理和c不同，如果在程序体结束时只声明而未定义，c–也会报出函数未定义的错误，而在c语言当中不会出现这样的行为。

2. 结构体+作用域

考虑到执行效率，我们为结构体定义也加入了作用域，实际上这样一来我们的c–也可支持struct内定义成员函数或者成员变量（语法允许的前提下）。否则，为了报告结构体内定义的重名结构体错误，需要遍历所有定义过的成员名，最差的时间复杂度为$O(n^2)$。然而当我们为结构体加入作用域后，所有结构体内的重名问题都会随之解决。

2.5.2 查表与错误检查

符号定义阶段会出现重名错误。一个普通的变量只需要在当前作用域链表中查找该变量是否已被定义过。对于函数定义，还需要另外检查当前作用域是不是顶层作用域（这种情况没有对应的错误标号，我们将该错误记为20），对于函数声明，需要检查是否在其上层已经有同名函数。变量和结构体、函数重名也是易错点，因为变量不能和其上层的任何结构体或者函数重名，但只需要不与当前作用域的变量重名，因此需要分开判断。

符号使用阶段的错误类型较多，但主要可分为两类，一种是未定义错误，一种是类型不匹配错误，包括变量类型不匹配，结构体不等价，函数签名不匹配等。这些错误都是在Exp产生式中发现。还有一个隐藏较深的小坑，即双目操作符的结果类型与值的类型相同，但bool表达式的结果类型是int类型。

3 总结与改进

本次实验在语法树上完成了细节繁多的语义分析部分，并运用`void*`指针和回调函数实现了一套可复用的容器模板。虽然代码层次性增强了，但也为内存管理带来了挑战。即使已经在必要的地方对各种数据结构进行了析构，但仍可能存在内存泄漏的问题。未来实验的目标就是修补内存泄漏的代码，并进一步优化代码的层次结构。

参考文献

[1] 《编译原理实践与指导教程》 许畅等