编译原理实验二：语义分析

林搏海 1190202128
哈尔滨工业大学 物联网工程

日期：2022 年 4 月 14 日

摘　　要

我们在 Lad1 实现的基于 Flex 的词法分析和基于 Bison 的语法分析的基础上，实现了语法制导的语义翻译过程，对于 c--代码的常见语义错误进行了检测和错误提示输出，除了对于实验指导书上基本语法错误的检测之外，我们还实现了选做二的错误检测，即对于变量作用域的限制，并且输出相应的错误信息。为了实现这些新增功能，我们建立了基于十字链表的哈希表结构用于存储和查询符号，使其能够在对于符号进行高效插入删除和检索的基础之上，还能够支持对于符号作用域的标注和限制。

关键词: 语法分析树，Flex，Bison，十字链表，哈希表

1　语义分析

1.1　语义分析过程概述

我们通过遍历实验一中的语法分析树来进行语义分析，语义分析主要是对于出现的符号进行一系列的类型检测工作，因此，当我们遇到一个需要处理的符号的时候，此时遍历语法分析树的过程一定是进入了 ExtDef 之后，因此我们每一次遇到 ExtDefen 就会进行对应的语法分析，然后对其递归地展开，就可以对于所有的非终结符进行处理，并且对于符号表进行操作和相应的语义错误检测。

1.2　符号表原理及其实现

在遍历语法分析树进行语义分析的时候，我们需要通过符号表来记录所有的符号出现过的位置，类型等信息，并且符号表还需要支持高效的查找和插入删除，同时，我们还希望符号表可以区分不同作用域的符号，以此来完成实验选做二。因此，我们选用了哈希表加上十字链表的机制来实现符号表，其中，通过哈希表来实现符号的快速增删查，通过把同一作用域的符号串联起来的形式来实现对于符号作用域的识别。对于符号表的哈希结构和十字链表，我们进行了如下的定义如图1

此外，为了方便区分函数符号以及结构体，我们对于结构体和函数信息进行了存储，对于匿名结构体，因为词法分析中规定了 id 不能为纯数字，所以此处我们用数字编号作为匿名结构体的名字，可以避免与其他符号的命名冲突，数字编号通过在 table 结构体中记录匿名结构体的个数来获得编号。

```c
typedef struct fieldList {
    char* name;       // 域的名字
    pType type;       // 域的类型
    pFieldList tail;  // 下一个域
} FieldList;

typedef struct tableItem {
    int symbolDepth;
    pFieldList field;
    pItem nextSymbol;  // same depth next symbol, linked from stack
    pItem nextHash;    // same hash code next symbol, linked from hash table
} TableItem;

typedef struct hashTable {
    pItem* hashArray;
} HashTable;

typedef struct stack {
    pItem* stackArray;
    int curStackDepth;
} Stack;
```

图 1: 符号表定义

我们对于 LR(1) 文法中的产生式编写了过程，在检测到对应的错误的时候会输出错误信息。如图2

```c
while (structField != NULL) {
    // then we have to check
    if (!strcmp(payload->name, structField->name)) {
        //出现重定义，报错
        char msg[100] = {0};
        sprintf(msg, "Redefined field \"%s\".",
                decitem->field->name);
        pError(REDEF_FEILD, node->lineNo, msg);
        deleteItem(decitem);
        return;
    } else {
        last = structField;
        structField = structField->tail;
    }
}
```

图 2: 错误输出

此外，为了进行空间管理，防止出现内存泄漏的情况，我们对于哈希表空间进行了严格的释放流程，当一个域结束之后，我们释放当前域中声明的符号，当程序运行结束的时候，我们释放所有的空间。图3

```
void deleteFieldList(pFieldList fieldList) {
    assert(fieldList != NULL);
    if (fieldList->name) {
        free(fieldList->name);
        fieldList->name = NULL;
    }
    if (fieldList->type) deleteType(fieldList->type);
    fieldList->type = NULL;
    free(fieldList);
}
```

图 3: 空间释放

2 主程序模块

为了方便编译和调试，我们将 main 函数放在了 main.c 文件中，其具体逻辑入下：首先，读入我们的 cmm 文件，调用 yyparse() 函数对其进行词法分析和语法分析，并且建立相应的语法分析树。如果词法分析阶段或者是语法分析阶段检测到错误，那么我们就调用我们自己重写的 error 函数进行报错；如果此过程没有错误，那么我们就初始化符号表，并且从根节点开始遍历整个语法分析树，并且执行对应的分析语句，如果出现语义错误，则输出相应的错误类型，如果整个程序没有任何的词法语法或者是语义错误，那么就不输出任何信息，并且在退出程序时释放语法分析树和符号表的空间。

3 编译执行方式

程序的正确编译执行需要以下八个文件：enum.h, node.h, lexical.l, main.c, syntax.y, makefile, semantic.h, semantic.c, 只需要确保他们处在同一个目录下，然后执行 make 命令调用 makefile 文件执行自动编译即可。