编译原理实验二报告

刘心悦
161220085

一、 功能简述

实验二的任务是在实验一的基础上，对 c--源代码进行语义分析和类型检查，并打印分析结果。在完成实验二的基础要求之外，我还完成了**要求 2.1** 和**要求 2.3** 的实现。要求 2.2 由于较为复杂，加之时间仓促，故未能完成。

二、 编译方法

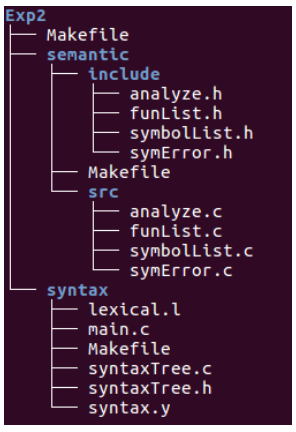
本次实验的源代码分布在具有层级结构的多个文件中，所以采用了 Makefile 进行递归编译。只需要在根目录下执行 **make** 指令，即可在根目录下得到可执行文件 parser。然后执行 **./parser <testfile.cmm>** 对 testfile.cmm 进行语义分析和类型检查。

三、 程序结构

在实验一中，我们已经得到了一颗语法树。在实验一的基础上，实验二在 analyze 函数中对语法树进行了一次遍历，并在检测到特定节点时对其进行分析，判断是否存在语义错误。这一部分功能实现在 analyze 文件中。

要实现语义分析，还需要借助符号表。由于没有实现要求 2.2，所以只需要用到 3 张全局符号表：函数表、结构体表和全局变量表。符号表数据结构及其对其进行相关操作的函数（包括类型检查）实现在 funList 和 symbolList 文件中。

如果检测到语义错误，则将其存入存放语义错误的链表中，语义分析结束后，将错误按照出错行数排序，再将所有错误打印出来。这一部分功能实现在 symError 文件中。本次实验的文件结构如图所示：

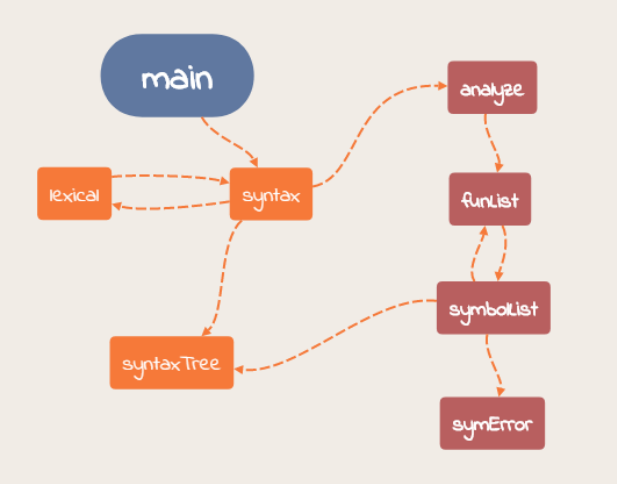


实验一的代码在 syntax 文件夹中，实验二的代码在 semantic 文件夹中。semantic 文件夹中包含 include 和 src 两个文件夹，分别存放头文件和源文件。Semantic 中各文件功能如表所示：

文件	功能
analyze	遍历语法树进行语义分析
funList	存放函数表

symbolList	存放全局变量表
	存放结构体表
	类型识别和检查
symError	存放和打印错误信息

文件结构如下图所示：（其中橙色表示实验一的文件，红色表示实验二的文件，箭头从 A 指向 B 表示"A include B"）



四、 功能详述

实验二的代码较为冗杂，下面将会主要按照错误类型的顺序对语义分析的实现方法进行讲述。

类型检查：

类型变量按照要求以嵌套链表（即二叉树）的形式存放在程序中，能够有效表示 int, float, array 和 struct 类型，链表节点名为 TypeNode，定义在 symbolList 文件中。同时还实现了多个与类型检查相关的函数：

函数名	功能
getType(specifier)	根据 Specifier 节点生成一颗 Type 树，并返回其根节点的指针
getExpType(exp)	根据 Exp 节点的类型生成一颗 Type 树，并返回其根节点的指针
TypeCheck(TypeNode* a, TypeNode* b)	检查两棵 Type 树是否相同

因为实现了要求 2.3，所以对两个结构体进行 TypeCheck 时依然按照树的结构进行判断，而非结构体的名字。

错误类型 1：变量在使用时未经定义

要判断这类错误，需要用到全局变量符号表，其数据结构实现在 symbolList 文件中，名为 ExtSList 的链表。当识别到 **Exp -> ID** 时，需要遍历 ExtSList 链表检查是否存在与此 ID 同名的已声明变量。若不存在，则表示该 ID 未经定义。

错误类型 2：函数在调用时未经定义

此类错误和和错误 1 类似，需要用到函数表，其数据结构实现在 funList 文件中，名为 funList 的链表。当识别到 **Exp -> ID LP Exp RB** 时，需要遍历 funList 链表检查是否存在与此 ID 同名的已声明函数。若不存在，则表示该函数未经定义。

错误类型 3：变量出现重复定义或变量与前面定义过的结构体重复

在 symbolList 文件中有一个名为 addExtNode 的函数，用于添加一个全局变量至链表。在添加前会遍历全局变量表和结构体表来检查该是否存在重名定义。

错误类型 4：函数出现重复定义

当识别到 **ExtDef -> Specifier FunDec CompSt** 时，会使用 addFunNode 函数将该函数定义加入函数表中。在 addFunNode 的函数中，在添加到表中前，会遍历函数表中的定义项来检查该函数是否是否存在重名定义。（FunListNode 分为声明项和定义项，用 FunListNode 结构体中的 funtype 属性来区分）

错误类型 5：赋值号两边的表达式类型不匹配

当识别到 **Dec -> VarDec ASSIGNOP Exp** 时，会检查 Dec 的 Specifier 和 Exp 的类型是否相同；

当识别到 **Exp -> Exp ASSIGNOP Exp** 时，会检查 ASSIGNOP 两端的 Exp 的类型是否相同。

错误类型 6：赋值号左边出现一个只有右值的表达式

当识别到 **Exp -> Exp ASSIGNOP Exp** 时，会检查 ASSIGNOP 左边的 Exp 是否是变量、数组或结构体，即该 Exp 是否推出 ID 或 Exp LB Exp RB 或 Exp DOT ID。若不是则报错。

错误类型 7：操作数类型不匹配或操作数类型与操作符不匹配

当识别到 Exp 推出的四则运算（PLUS MINUS STAR DIV）和 RELOP 运算时，会检查操作数 Exp 类型是否匹配而且是 int 或 float，若不是则报错；

当识别到 Exp 推出的逻辑运算（AND OR NOT）时，会检查操作数 Exp 类型是否匹配而且只能是 int，若不是则报错；

当识别到 Exp 推出的条件语句（IF WHILE）时，会检查括号中的 Exp 类型是否是 int，若不是则报错。

错误类型 8：return 语句的返回类型与函数定义的返回类型不匹配

当识别到 **ExtDef -> Specifier FunDec CompSt** 时，会以 CompSt 为根在语法树上遍历找到 CompSt 中所有的 RETURN 节点，然后将 Specifier 的类型保存在 RETURN 节点的 extra 属性中。

当识别到 **Stmt -> RETURN Exp SEMI** 时，会检查 Exp 的类型是否和 RETURN 中的 extra 属性匹配，若不是则报错。

错误类型 9：函数调用时实参与形参的数目或类型不匹配

当识别到 **Exp -> ID LP Args RP** 时，会遍历函数表找到与 ID 重名的函数声明或定义，然后检查 Args 中参数的类型和数量是否和函数表中的记录相符。

错误类型 10：对非数组型变量使用 […] 操作符

当识别到 **Exp -> ID LB Exp RB** 时，会遍历全局变量表找到与 ID 重名的全局变量，若找到了，但该全局变量的类型不是 array（即 int, float 或 struct）则报错。

错误类型 11：对普通变量使用 () 或 (…) 操作符

当识别到 **Exp -> ID LP Args RP** 时，会遍历函数表找到与 ID 重名的函数声明或定义，若没找到，再在全局变量表中进行查找，若找到同名的全局变量则报错。

错误类型 12：数组访问操作符 […] 中出现非整数

当识别到 **Exp -> ID LB Exp RB** 时，会检查括号间的 Exp 的类型是否是整数。

错误类型 13：对非结构体型变量使用 “.” 操作符

当识别到 **Exp -> Exp DOT ID** 时，会检查 DOT 前的 Exp 的类型是否是整数。

错误类型 14：访问结构体未定义过的域

当识别到 **Exp -> Exp DOT ID** 时，先调用 getExpType 得到 DOT 前 Exp 的类型变量，再检查类型变量中是否有和 ID 同名的域，若没有则报错。

错误类型 15：在定义时对域进行初始化

当识别到由 StructSpecifier 推导得到的 DefList 中有 ASSIGNOP 时报错。

由于所有变量的作用域是全局的，所以当域被重复定义时，会被识别为错误类型 3。

错误类型 16：结构体的名字与前面定义过的结构体或变量名字重复

此类错误和和错误 1 类似，需要用到结构体表，其数据结构也实现在 symbolList 文件中，名为 structList 的链表。当识别到 **StructSpecifier -> STRUCT OptTag LC DefList RC** 时，如果 OptTag 推出 ID，那么需要遍历 structList 和 symbolList 链表检查是否存在与此 ID 同名的已定义结构体或全局变量。若不存在，则调用 addStructNode 函数将该结构体加入到结构体表中；若存在，则报错。

错误类型 17：直接使用未定义的结构体来定义变量

当识别到 **StructSpecifier -> STRUCT Tag** 时，遍历 structList 检查是否有和 Tag 推导得到的 ID 同名的已定义结构体。若不存在，则报错。

错误类型 18：函数进行了声明，但没有定义

在 syntax 文件中，当 analyze 函数完成对语法树的遍历后，调用 checkNoDef 函数对函数表进行遍历，检查每个声明项能否找到与其同名且类型相符的定义项。若出现无定义的声明，则报错。

错误类型 19：函数多次声明相互冲突，或声明与定义之间相互冲突

当识别到 **ExtDef -> Specifier FunDec SEMI** 时，调用 addFunNode 函数将该函数声明加入到函数表中。在 addFunNode 的函数中，在添加到表中前，会遍历函数表来检查该函数的重名声明或定义中，是否存在返回类型，形参数量、类型的冲突，若存在则报错。

五、 写在最后

这次实验全程使用 C 语言进行编写。虽不复杂，但各种链表操作还是稍显繁琐。如果在一开始选择 C++语言进行编写，那么 C++中的容器和类封装功能会使代码结构能加清晰，检错过程更加简洁明了，那么要求 2.2 也能更加轻易地实现。

在进行具有层次文件结构的编译时，需要先调用子文件夹中的 Makefile 将其中的的 .c .y .l 文件编译成 .o 文件，再通过根目录下的 Makefile 将所有的.o 文件链接起来。如果按照实验一中所说执行 **flex lexical.l** 命令将 lexical.l 转换为 lex.yy.c，在链接时会报 multiple definition of **的错误，原因是 lex.yy.c 被重复包含。解决办法是通过 **flex --header-file=lex.yy.h lexical.l** 命令为 lexical.l 同时生成 .c 和 .h 文件，由于.h 文件中有条件编译选项，所以不会出现重复定义的错误。