《编译原理》实验 2

实验报告

一、综述

本次实验完成了语义分析和中间代码生成的所有功能，在中间代码生成部分还超额完成了诸如结构体和数组相互嵌套或者多层嵌套的功能，并且在完成所有样例的基础上增加了一个包含各种规则融合的样例（example/2-s3.c），其中的语法是本次实验完成的所有内容。本程序能够正确对这个文件中的写法进行正确的类型检查和中间代码生成。

本次试验采用的分析方法是类似 LR 的分析方法，能够在建立语法树的同时进行翻译，不需要二次回溯。由于 bison 基本不支持继承属性，因此在实现 LR 分析的时候使用了各种全局变量用来在节点之间传值，以此来模拟继承属性的功能。在特殊的确实无法用全局变量处理的情况下，或者全局变量存在问题的时候，采用过在高层进行少量回溯、构建状态机、重写部分产生式、使用二级指针、使用栈结构储存全局变量等方法来解决问题。

二、实验内容

（一）语义分析和类型检查

1．实现思路

1) 建立符号表结构

本次实验一共建立了四个符号表，都是链表结构，分别是：

- 变量符号表 VarRec

  用于记录所有出现的变量（结构体、整数、浮点数、数组）的变量名、类型、在中间代码中分配的名字等。

- 函数符号表 FuncRec

  记录所有等译函数的函数名、返回值类型、参数个数、参数列表等。

- 数组符号表 ArrRec

  记录数组变量的名字、维度、类型、占用空间等。

- 结构体符号表 StRec

记录结构体变量的名字、参数列表、占用空间等。

与这些数据结构对应的是对他们的操作，例如增加、检查、对比等。

2) 添加符号表项目

在建立了符号表结构之后，下面需要做的就是如何将所需要的信息添加到符号表中。我采取的方式是将所有附属的信息汇集到 ID 节点上，之后将 ID 节点传递给建立符号表的函数。函数名 ID 举例，在 bison 进行 LALR 分析的时候，在其中加上合适的代码将函数的返回值类型，参数列表、参数个数等信息传递到 ID 节点，然后在需要的信息都添加完之后，将 ID 节点传递给添加函数符号表项目的函数，该函数用于从 ID 节点中获取需要的信息，建立函数符号表项目添加到函数符号表。其他的结构体 ID、数组 ID、基本类型变量 ID 同理。

在进行 ID 类别判断的时候，使用之前的产生式，并不能够使用 LR 方法区分数组类型和基本变量类型，因为它们定义时候的前缀都是一样的（e.g. int a 和 int a[2]，前一个是基本类型，后一个是数组类型），所以这里我改变了产生式结构，消除了左递归，使其在 LR 分析的时候有办法确定变量类型。改变后如下：

```
VarDec: ID VarDec_x
;
VarDec_x:
| LB INT RB VarDec_x
;
```

3) 检查错误

有了相应的符号表和每个类型的变量对应的主要数据，检查错误就变得相对的容易了，主要思路就是检查和匹配。出现冲突、不匹配或者查询不到就说明有错误。

2．完成样例

17 个样例可以全部成功检测出正确错误。注意因为中间代码生成和错误检测是同时进行的，所以在输出错误的时候也会输出部分生成的中间代码。

（二）中间代码生成

1．实现思路

1) 代码拼接

每一个语法树上的节点都有两个指针，分别指向中间代码链表的起点和终点。每一条中间代码是一个结构体（CodeBlock），中间代码之间使用链表数据结构。这样能够节约空间并且简化复制、拼接、插入的过程。四个工具函数（combineCode、combineNodeCode、addCode、copyCode）用作代码的拼接和赋值，getCodeblock 用于创建一个代码节点。

2) 冲突处理

因为不能使用继承属性，那么可能会在需要某个中间代码代码（比如某个上级节点的 trueLabel）的时候，可能这个标签还不存在（未被赋值），这个时候就想到了使用二级指针。只需要将代码块的指针指向所需要节点的某个标签（例如 trueLabel），而不是直接指向字符串，这种用法类似于一种引用，当目标节点的标签在之后被赋值的时候，代码节点也就能够正常获取了。

这里还需要注意的是，因为 CompSt（“{}”结构）是可以嵌套的，但是在外层使用的全局变量和内层使用的全局变量又是相同的，所以在处理内层的时候外层的变量可能会被污染，这个时候想到了使用栈结构。在进入内层之前，将外层某些会被污染的全局变量压栈，当内层结束之时将外层的全局变量数据弹栈，这样就实现了全局变量的隔离。

3) 结构优化

按照原本的产生式，如果要产生正确的标签，那么每一个 Stmt 之后都需要一个标签，标志着这个 Stmt 的出口，即 S.next。但是很多情况下不会使用到这个标签，例如普通的表达式语句（x=x+1;），只有 if、while 等控制流语句才会使用到，为了简化标签，使得和样例的输出结果更相符合，我改变了产生式的结构。如下：

```
StmtList:
| PStmt StmtList
;
PStmt: Stmt
| Exp SEMI
| CompSt
```

```
| RETURN Exp SEMI
;
Stmt: IF LP Exp RP PStmt %prec AFTER_ELSE
| IF LP Exp RP PStmt ELSE PStmt
| WHILE LP Exp RP PStmt
| error SEMI
;
```

加入了 PStmt 非终结符，用来区分控制流类语法和表达式类语法。从而更好的进行标签输出。总的代码的输出是在定义完一个函数之后进行，因为函数不能嵌套并且线性排列，作为输出的节点比较合适。

2．完成样例

最后样例能够全部跑通，并且为了完全反应本实验结果所能解析的语法，我另外写了一个更加复杂的样例（example/2-s3.c），其中包含语法定义中的所有控制流结构、数组和结构体的相互、多层嵌套。老师可以进行检查。

三、编译及验证方式

主要文件：

parser.y 语法分析 bison 文件

cminus.l 词法分析 flex 文件

funcs.h 功能函数头文件

funcs.c 功能函数定义文件

makefile 定义如何编译的文件

Makefile 文件内容：

```
cminus: parser.y cminus.l funcs.h funcs.c
    bison -d -o parser.c parser.y
    flex  -o cminus.lex.c cminus.l
    gcc -o $@ parser.c cminus.lex.c funcs.c  -lfl -lm
```

运行及检验方式：

打开终端，进入文件夹根目录，输入 make+回车 即可编译。生成的 cminus 即为最终程序，在命令行输入 ./cminus ./examples/1.c 即可检验样例。输入参数是一个文件的时候，该文件是需要编译的源码，输出会在标准输出 stdout 中，当输入参数是两个文件的时候，那么第二个文件是输出文件，所有输出在第二个文件中。