课程设计报告

1. 课程设计目标和要求

2. 需求分析

2.1 数据流图

2.2 功能

1) 词法分析

2) 语法分析

3) 语义分析

在语法分析的基础上，根据其产生式序列和以归约顺序重排的词法序列建立和维护符号表，检查是否存在重复定义，类型不一致等错误，并为代码生成提供对于符号表信息的查询接口。

4）代码生成

3. 总体设计

3.1 数据结构设计

1) 符号表 IdTable

设定符号表为多层次结构；

建立主符号表，用于存储全局的标识符；

以及子符号表用于存储局部的标识符；

- *father* 指向上层符号表，主符号表指向空，子符号表指向主符号表
- *nameToId* 使用了 C++ 的 STL 库中的 *unordered_map*, 为将一个字符串映射至一个标识符的存储方式。
- *now_id_table* 指向当前块所在的符号表；

```
struct Id_Table
{
        Id_Table *father;
        map<string, Id> name_to_id;
} * now_id_table;
```

- 标识符 **Id**

属性	类型	含义
name	string	标识符名字
dataType	DataType	数据类型
idType	IdType	标识符类型
paramList	vector<Param>paramList	参数列表
retDataType	DataType	返回值类型
isError	bool	是否有错误

- 参数 (**Param**)

属性	类型	含义
name	string	标识符名字
dataType	DataType	数据类型

- 数据类型 (**DataType**)

属性	类型	含义
basicType	BasicType	基础数据类型
constVal	int	整数型常量数值
dimension	int	数组维度

属性	类型	含义
upperBound	vector	数组上限
lowerBound	vector	数组下限
paramType	ParamType	参数类型

- 枚举类型 **BasicType**

```
enum BasicType : int
{
        _integer = 1,
        _real,
        _boolean,
        _char,
};
```

对应四种支持的数据类型

- 枚举类型 **ParamType**

```
enum ParamType : int
{
_reference = 1,
_value
};
```

对应引用参数，传值参数

- 枚举类型 **IdType**

```
enum IdType : int
{
  _constant = 1,
  _variable,
  _procedure,
  _function
};
```

对应 5 种标识符类型

4. 总体设计

4.3 语义分析

1) 接口描述

输入：语法分析生成的依照 SLR 分析归约顺序的产生式序列和记号序列。

输出：供代码生成的符号表，给语义分析提供所需的列表。

2) 功能描述

- 检索标识符

  对于给定的标识符进行查找，如果当前表不存在，则查找上一层符号表, 直至主符号表。

- 插入标识符

  在符号表中新建对应的项。

- 定位

  建立新的子符号表，将指针指向当前符号表

- 重定位

  删除当前符号表，根据指针找到上层符号表。

- 错误检查

  对于作用域，赋值语句，函数和过程调用，运算符使用进行检查。

3) 所用数据结构说明

- **idStack**(vector<Id>)

用于记录标识符信息的栈。

- **idList**(vector<Id>)

  用于变量定义时存储标识符列表。

- **opStack**(vector<string>)

  用于记录运算表达式操作符。

- **paramList**(vector<Id>)

  记录函数和过程的参数列表

- **varList**(vector<Id>)

  记录调用 read 函数的参数列表

- **exprListStack** (vector<vector<Id>>)

  记录 expressionList 的列表，用于函数的调用。

- **nowIdTable** (IdTable*)

  当前所在符号表的指针。

4) 详细设计

整体来看，程序根据产生式的序号，以及该产生式所归约的词法符号，决定如何进行对于符号表的修改和

- 常量定义

  $const\_value \rightarrow ...$

  idStack 根据产生式记录下常量类型.

  $const\_declaration \rightarrow id\ relop\_e\ const\_value$

  根据 *idStack* 中常量类型和 *id*，将其插入符号表中。

- 变量定义

$type \rightarrow ...$

$idStack$ 根据产生式记录下变量类型.

$idlist \rightarrow id|idList, id$

$idList$ 记录下 id 名。

$var\_declaration- > idlist\ punc\_colon\ type$

根据 $idStack$ 中变量量类型和 id，将其插入符号表中。

- 运算表达式

  $x \rightarrow y\ op\ z$

  根据 op 类型以及 y 和 z 的类型得出 x 的类型，以及判断是否存在运算符的使用错误

运算符	类型	子表达式 y,z 类型限制	x 表达式类型
relop	关系运算符	y,z 类型相同，或者为 real 和 integer	bool
not	取非运算符	bool	bool
minus	取反运算符	integer 或 real	z 类型
”+”,”-”,”*”,”/”	算术运算符	integer 或 real	有 real
”div”,”mod”	算术运算符	integer	integer
”and”,”or”	与或运算符	bool	bool

- if 与 $else$ 语句

  $statement \rightarrow if\ expression\ then\ statementelse\_part$

  检查 $expression$ 的类型是否为 $bool$

- for 语句

  $statement \rightarrow for\ id\ assignop\ expression\ to\ expression\ do\ statement$

  检查 id 类型是否为 $integer$

- 数组使用

$period \rightarrow period\ punc\_comma\ digits\ punc\_point\ punc\_point\ digits$

记录数组维度增加一维，并且记录上限和下限，并判断上限是否小于等于下限。

$type \rightarrow array\ punc\_square\_left\ period\ punc\_square\_right\ of\ basic\_type$

将 $idStack$ 中记录维度和类型的信息合并。

$variable- > id\ id\_varpart$

查看数组，对比维度数量是否正确，以及 $varpart$ 均为 $integer$ 类型

- 过程及函数使用

  $parameter\_list \rightarrow parameter\_list\ punc\_semicolon\ parameter$

  用 $paramList$ 记录下参数列表.

  $subprogram\_head \rightarrow function\ id\ formal_parameter\ punc_colon\ basic\_type$

  将函数的属性 (参数列表，返回值类型) 计入符号表, 并且进行定位，将所有的参数和返回值加入子符号表中。

  $procedure\_call \rightarrow id\ punc\_round\_left\ expression\_list\ punc\_round\_right$

  调用函数，从符号表中查找到函数后，对比是否参数类型和参数数量相同，并且对于引用参数需要看到对应表达式是否是一个单独的变量。

- $read$ 与 $write$ 处理

  将 read 和 write 对应的所有标识符记录下来，提供给代码生成类型信息。

6. 程序测试

8.

- 体会与收获

兴淑鹏：这次我主要负责的是语义分析部分和测试的部分，在这次的程序编写过程我真正实际接触到了 **pascal** 语言的语义分析过程，真正实际的面对了语义分析中的实际问题。认识到了在完成一个比较复杂的代码，应该先进行由简单到难的分步，再依次逐步完成。像是在语义分析过程要处理的问题其实很多，最基本的是常量和变量的定义，再到运算表达式的判断，以及 if 语句和 for 语句等等。通过一步步完善这些功能，我对于实际的符号表维护和错误检查有了完整的理解。并且由于语义分析需要获取到词法分析和语法分析的结果并且为代码生成提供接口，需要小组成员之间互相足够的交流协助，我更加了解了小组工作中如何互相协同以便于个部分的对接。在测试的过程中，我更多的了解到了测试对于程序功能检测的重要性，因为在完成很大的工程任务时候，出现纰漏是难免的，这时就需要足够多的样例对于程序执行的各种可能情况进行测试，在测试过程中找出了各个部分的 bug 并修复。

- 遇到的主要问题以及解决方案

  兴淑鹏：遇到的主要问题：1. 如何解决对于函数递归中标识符重名问题，例如在计算 **gcd** 的函数中会遇到返回值对应标识符名和调用函数的标识符名相同，我最后选择增加了一个特殊的查找标识符的函数，由于在文法要求中函数定义是无法嵌套的，因此对于函数调用的查询一定是在主符号表中，所以增加在主符号表中查找标识符的函数专门对于过程和函数的调用进行标识符查找；2. 如何解决对于引用参数，调用该函数对应的参数必须是单一的实际变量问题，例如在 **swap** 函数中通常我们会将参数设置为引用参数，而调用该函数时如果调用时为 $swap(a+b, a+c)$ 显然这样是存在错误的。为此我对于记录 **id** 信息的 $idStack$ 栈中，当使用操作符将两个变量合并后，记录其对应的 $idType$ 由 $\_variable$ 修改为 $\_none$

- 修改建议

  建议增加可以对于语义错误的恢复功能，并且使得各种错误除了输出输出行号外，还能够输出列号等