编译原理课程实验报告

实验 3：语义分析

姓名	姚敏敏		院系	软件学院		学号	170720231	
任课教师	韩希先			指导教师	韩希先			
实验地点	研究院中 517			实验时间	2019 年 11 月 9 日			
实验课表现	出勤、表现得分			实验报告得分		实验总分		
	操作结果得分							

一、实验目的

要求：需分析本次实验的基本目的，并综述你是如何实现这些目的的？

1. 巩固对语义分析的基本功能的认识。

语义分析的基本功能是通过确定类型、类型检查、语义处理、以及静态语义检查将高级语言程序翻译成等价的中间语言代码。

2. 能够基于语法制导翻译的知识进行语义分析。

语法制导翻译是在语法分析过程中进行静态语义检查和中间代码生成的技术。语法制导翻译是在进行语法分析的同时，按照相应的语法成分的语义执行语义分析。

3. 掌握类高级语言中基本语句所对应的语义动作。

通过对控制流语句进行回填式翻译，可以生成完整的代码跳转序列。

4. 理解并处理语义分析中的异常和错误。

为了控制错误的出现，我们在语义分析的时候，会进行类型检查，从而降低了出错的概率。

二、实验内容

要求：对如下工作进行展开描述

1. 给出如下语言成分所对应的语义动作

- 函数定义(或过程定义)

  函数定义的基本文法：

  - P → prog id (input, output) D ; S

  函数定义对应的语义动作：

  - P → { offset := 0 } D

- 变量声明

  变量声明语句的文法：

  - P → prog id (input, output) D ; S
  - $D \rightarrow D$; D | $List$: T | proc id D ; S
  - $List \rightarrow List_1$, id | id
  - $T \rightarrow$ integer | real | array C of T_1 | ↑T_1 | record D
  - $C \rightarrow$ [num] C | ε

  其中 D 是程序说明部分的抽象、S 是程序体部分的抽象、T 是类型的抽象，需要表示成类型表达式、C 是数组下标的抽象。

根据翻译，可以得到以下语义动作：

- P→ { offset := 0 } *D*
- D→D ; D
- D→id : T {enter(id.name, T.type, offset); offset := offset + T.width}
- T→integer {T.type := integer; T.width := 4}
- T→real {T.type := real; T.width := 8}
- T→array[num] of T_1 {T.type := array(num.val, T_1.type);T.width := num.val * T_1.width}
- T→↑T_1 {T.type := pointer(T_1.type); T.width := 4}

- 赋值

简单赋值语句的文法如下：

- S → id := E
- E → E_1 + E_2
- E → –E_1
- E → (E_1)
- E → id

简单赋值语句翻译过后的语义动作如下：

- S → id := E {p := lookup(id.name);
  if p ≠ nil then gencode(p, ':=', E.addr)
  else error }
- E → E_1 + E_2 {E.addr :=newtemp;
  gencode(E.addr,':=',E_1.addr,'+',E_2.addr)}
- E → –E_1 {E.addr := newtemp;gencode(E.addr,':=','uminus',E_1.addr)}
- E → (E_1) {E.addr := E_1.addr }
- E → id {p := lookup(id.name);if p ≠ nil then E.addr := p else error }

- 表达式

布尔表达式的基本文法如下所示：

- B → B_1 or B_2 | B_1 and B_2 | not B_1 | (B_1) | E_1 relop E_2 | true | false

布尔表达式所对应的语义动作如下：

- B → B_1 or B_2 {B.addr := newtemp;
  gencode(B.addr':='B_1.addr'or'B_2.addr)}
- B → B_1 and B_2 {B.addr := newtemp;
  gencode(B.addr':='B_1.addr'and'B_2.addr)}
- B → not B_1 {B.addr := newtemp;
  gencode(B.addr':= ' 'not'B_1.addr)}
- B → (B_1) {B.addr:= B_1.addr}
- B → E_1 relop E_2 {B.addr := newtemp;
  gencode('if'E_1.addr relop.op E_2.addr'goto'nextquad+3);

gencode(B.addr ':= "0") ;gencode('goto'nextquad+2);
gencode(B.addr ':= "1")}

- B → true {B.addr:= newtemp; gencode(B.addr ':= "1')}
- B → false {B.addr:= newtemp; gencode(B.addr ':= "0')}

- 循环

  循环语句的基本文法：

  - S → while B do S1
  - S → begin Slist end
  - Slist → Slist; S | S

  循环语句翻译后的语义动作：

  - S → while B do S_1

    {S.begin:= newlabel;
    B.true := newlabel;
    B.false := S.next;
    S_1.next := S.begin;
    S.code:=gencode(S.begin,':')||B.code||
    gencode(B.true,':')||S_1.code||
    gencode('goto',S.begin)}

- 分支

  分支语句的基本文法：

  - S → if B then S1 | if B then S1 else S2

  分支语句翻译后的语义动作：

  - S → if B then S_1

    {B.true := newlabel;
    B.false := S.next;
    S_1.next := S.next;
    S.code:=B.code||gencode(B.true,':')||S_1.code }
  - S → if B then S_1

    {B.true := newlabel;
    B.false := S.next;
    S_1.next := S.next;
    S.code:=B.code||gencode(B.true,':')||S_1.code }

2. 语义动作具体加入所涉及的数据结构及其物理实现

   数据结构：

   我们采用 Java 来实现语义分析，所以，并没有定义结构体类型的数据据结构，而是把数据结构定义封装在类之中。通过访问控制符来控制对这些数据结构的访问。而且我们还使用了 Java 中封装好的 `ArrayList`，更加方便了我们代码的编写。

我们定义了线性表 List<Name>类型的符号表，用来存储符号表 nameTable。我们还定义了用来存储中间代码的线性表 List<String>类型的 intermediateCode。

```
private List<Name> nameTable = new ArrayList<>();//符号表 nameTable
private List<String> intermediateCode = new ArrayList<>(); //中间代码
```

对于这些数据结构，我们通过基本的操作函数来加以实现：

- 函数 genCode(String... s)用于产生一条中间代码。
- 函数 merge(List<Integer> p1, List<Integer> p2)用于回填时将两个符号表进行合并。
- 函数 backPatch(List<Integer> p, int i)用于将一个变量 i 回填到 list p 中
- 函数 backPatchAll()将所有变量 i 回填到 list p 中

3. 语义属性的分析、设计和实现
   - 文法变量 T(类型)的语义属性的分析
     - type： 类型(表达式)
     - width：类型所占用的字节数
   - 语义属性的设计

     首先设置基地址 baseQuad 为 100，然后定义偏移量 offset，在生成跳转地址的同时与基地址 baseQuad 相加，得到实际的地址。然后我们定义在回填的时候使用的下一条四元式序号 nextQuad，用来在语义分析时生成四元式时查找下一个四元式。
   - 语义属性的实现

```
private final int baseQuad = 100;
private int offset;
private int nextQuad = 0;
```

4. 符号表的相关处理

符号表的数据结构：

符号表在 Java 中采用静态类的 List 线性表的方法来实现，该静态类的名称是 Name，主要属性有 String 类型的 type，符号的类型；有 String 类型的 name，是符号的名称；有 int 类型的 offset，是符号的偏移量，有 int 类型的 width，是符号的宽度。然后，我们可以通过构造函数来初始化该表项。

最后，我们建立一个 Name 类型的 List，来储存该符号表。因为 Java 中提供了 List 类，所以对该表操作起来比较简单。

对于符号表的数据结构，我们通过一些基本函数来加以操作：

- 函数 makeList(int i)用于创建一个新的符号表，并返回这个新表
- 函数 enter(String name, String type, int width)用于在符号表中为名字 name 添加一个表项。
- 函数 newTemp(String type, int width)用于产生一个新的临时变量，并且把这个变量存到符号表中，返回这个临时变量的偏移量。

- 函数 lookup(String name)用于检查符号表中是否抽面某个同样的名字 name。

5. 错误处理
   - 如果在语义分析中查找符号表，却没有在符号表中找到我们想找到的符号 name，就报错。
   - 如果在语义分析中没有符合语义规则的情况出现，则报错

三、实验结果

要求：将实验获得的结果进行描述，基本内容包括：

1. 针对测试程序输出其语义分析结果；
   - 测试程序如下图所示：

```
int main()
{
    float x;
    x=60;
    int y;
    y=0;
    if(x>=0&&x<60)
    {
        x=60;
    }
    else
    {
        x=100;
    }
    while(x>60)
    {
        y=y+1;
    }
}
$
```

   - 测试程序输出的语义分析的结果如下所示：
     - 测试程序翻译过后生成的四元式如图所示

       其中，单引号括起来的部分代表程序中出现的数字，没有单引号括起来的数组代表程序中出现的变量的 offset，这里为了简便考虑，没有给每个临时变量命名，如果要给每个临时变量命名的话，需要设立全局变量，但是这样子在 Java 里面是不符合规范的。或者我们可以把临时变量单独做成一个类，对这个类进行操作。

```
================================
100:    12 := '60'
101:    4 := 12
102:    20 := '0'
103:    16 := 20
104:    24 := '0'
105:    if 4 >= 24 goto 107
106:    goto 113
107:    28 := '59'
108:    if 4 < 28 goto 110
109:    goto 113
110:    32 := '60'
111:    4 := 32
112:    goto 115
113:    36 := '100'
114:    4 := 36
115:    40 := '60'
116:    if 4 > 40 goto 118
117:    goto 122
118:    44 := '1'
119:    48 := 16 + 44
120:    16 := 48
121:    goto 115
122:    end.

Process finished with exit code 0
```

2. 输出针对此测试程序经过语义分析后的符号表。
   - 符号表如下图所示，所有的临时变量用 t 来代替。

```
================================
Name    Type    Width   Offset
return  int     4       0
x       real    8       4
t       int     4       12
y       int     4       16
t       int     4       20
t       int     4       24
t       int     4       28
t       int     4       32
t       int     4       36
t       int     4       40
t       int     4       44
t       int     4       48
================================
```

四、实验中遇到的问题总结

要求：主要阐述两方面的问题

一、 实验过程中遇到的问题如何解决的？

在编写语义分析程序的时候，如果我们只按照书上 7.5 节的方法对控制流语句进行翻译的话，则无法完全输出语句，但是如果我们能够采取回填的办法对控制流语句进行翻译，则可以正确地输出翻译后的代码。

在一开始理解这些语句翻译的时候有些困难，但是如果采取建立注释分析树的方式，就可以更加快速地理解程序的内在逻辑。

在进行回填的时候，产生了两个 *list:truelist* 和 *falselist*，对于这两个 *list*，一开始并不知道采用什么方式来实现它，但是，后来经过考虑，发现我们可以直接在现有的写好的

hash map 中添加以 *truelist* 和 *falselist* 为类型的对象，这样不仅仅节省了空间，而且 *hash map* 的运用加快了程序运行的速度。

二、 思考题的思考与分析

思考题 1：你编写的程序能否和语义动作完全无关，即无论什么样的语义动作，都不需要修改你编写的程序，说明要达到完全无关需满足什么样的条件？

不能和语义动作完全无关，必须按照我们确定的语义规则来编写程序。如果想达到完全无关，则必须包括所有的语义动作，或者通过自动机的方式来自动识别新的语义规则，或者取决于采用的文法，是否为上下文无关文法。

思考题 2：如果你采用的自顶向下的语法分析，你是如何处理综合属性的，如果你采用的是自底向上的语法分析，你是如何处理继承属性的？

我采用的是自底向下的语法分析，继承属性自上而下传递，直至传递至最底层，在程序中使用二叉树的方式来实现，在语法分析的时候，先进行建树，然后对建好的树，将继承属性自上而下地传递下去，直至二叉树的最后一层。

思考题 3：你产生的结果(四元组)还需经过什么样的处理后就可以等价于汇编程序了？

首先将语法转换为汇编的语法，然后确定分配给这段程序的内存的基地址，再然后将基地址与偏移量相加，临时变量需要确定存放在哪个寄存器的什么位置，数据需要放到数据段中，就可以将四元组或者三元式转换成汇编程序。

五、实验体会

编译原理的三次实验让我深刻地理解了词法分析、语法分析、语义分析三者之间的关系。词法分析是进行编译的基础，我们首先必须确定等待编译的代码的词法是否符合规范，如果有错误就第一时间对单词进行报错，避免以后进行语法分析和语义分析的时候产生错误。

语法分析是比较难以实现的部分，由于语法分析种类众多，而且所涉及的算法实现起来难度较大，所以在这方面花费了不少的时间，语法分析实现的重点是移进规约，移进规约的时候涉及到很多栈的操作，在 Java 中实现的时候，可以使用自带的 *Stack* 类，也可以直接使用线性表来实现栈。可以节省很多编码的时间。

至于语义分析，最难的部分就是回填了，在进行回填操作的时候，我们要对 *truelist* 和 *falselist* 进行操作，这一部分需要搞清楚这些操作之间的内在逻辑。其次就是符号表的处理，如果我们使用 Java 实现符号表的处理时，通过 *Arraylist* 类可以实现对符号表的快速操作。

指导教师评语：

日期：