

深圳大学实验报告

课程名称： 编译原理

实验项目名称： 语法制导翻译和中间代码生成

学院： 计算机与软件学院

专业： 软件工程

指导教师： 蔡树彬

报告人： 郑彦薇 学号： 2020151022 班级： 软件工程 01 班

实验时间： 2023 年 6 月 13 日至 6 月 18 日

实验报告提交时间： 2023 年 6 月 14 日

教务处制

实验目的与要求：

目的：

通过设计、实现赋值等语句的中间代码生成程序，熟练掌握语法制导翻译模式及其应用。

要求：

第一部分：基于简单算术表达式的赋值语句的翻译与中间代码生成

输入文件：由多条使用简单算术表达式形成的赋值语句组成的源代码，语句间用“；”隔开

输出文件：三地址码（或四元组）序列

例如：输入文件内容：a=b+c*e/g; d=f*g ...

则输出文件内容可以为：

- t1=c*e;
- t2=t1/g;
- t3=b+t2;
- a=t3;
- t4=f*g;
- d=t4
-

第二部分：（选做）条件语句的翻译与中间代码生成

输入文件：包含条件语句和赋值语句

输出文件：对应的三地址码序列

第三部分：（选做）Maple IR 中间代码生成

输入文件：第一部分的输入

输出文件：方舟 Maple IR 对应的中间代码

方法、步骤：

要完成本实验，依据实验要求进行分解，需要完成的实验步骤是：

1. 你为简单算术表达式、赋值语句、顺序结构语句设计的产生式和语义规则分别是什么？

（1）简单算术表达式：

一个算术表达式可以包含常量、变量和四则运算。在实验中默认操作数全是小写字母，忽略对常量的直接运算，可以设计简单算术表达式的产生式为：

$\langle \text{expr} \rangle := \langle \text{term} \rangle \mid \langle \text{term} \rangle \langle \text{addop} \rangle \langle \text{expr} \rangle$

$\langle \text{term} \rangle := \langle \text{factor} \rangle \mid \langle \text{factor} \rangle \langle \text{mulop} \rangle \langle \text{factor} \rangle$

$\langle \text{factor} \rangle := \langle \text{letter} \rangle$

$\langle \text{addop} \rangle := '+' \mid '-'$

$\langle \text{mulop} \rangle := '*' \mid '/'$

上述产生式中， $\langle \text{letter} \rangle$ 表示表达式中的小写字母。

定义语义规则如下：

对于小写字母，使用符号表查找其对应的值，并将其视为操作数；对于加减法，直接进行运算；对于乘除法，采用中间代码生成中的临时变量机制，将结果存放到一个新的临时变量 t_i 中，并将其地址返回。

（2）赋值语句：

赋值语句是将一个表达式的值赋给一个变量。可以设计产生式为：

$\langle \text{assign} \rangle := \langle \text{variable} \rangle '=' \langle \text{expr} \rangle$ 其中 $\langle \text{variable} \rangle$ 表示变量, $\langle \text{expr} \rangle$ 表示表达式。

定义语义规则如下:

使用符号表查找左侧变量对应的内存地址, 然后对右侧表达式进行求值, 得到其结果, 最后将右侧表达式的结果存储到左侧小写字母对应的值中。

(3) 顺序结构语句:

顺序结构语句是指按照代码的书写顺序依次执行的语句。可以设计产生式为:

$\langle \text{stmt} \rangle := \langle \text{assign} \rangle | \langle \text{expr} \rangle$ 其中 $\langle \text{expr} \rangle$ 表示表达式, $\langle \text{assign} \rangle$ 表示赋值语句。在这里将表达式视为一种语句, 是因为表达式可以出现在语句中的任何位置。

定义语法规则如下:

对于赋值语句, 按照赋值语句的语义规则进行求值; 对于表达式, 按照表达式的语义规则进行求值但忽略其结果。(因为实验中默认操作数全是字母, 所以相应的每一部分都是一个表达式而不进行直接求值过程)。

2. 翻译输出过程中, 需要生成多个临时变量, 你是如何管理、控制这些临时变量的?

在识别每一个简单算术表达式的过程中, 定义一个计数器并初始化为 1, 在识别过程中, 每获得一个子式 (例如 $c * e$), 计数器就加一。因为实验中使用栈存放变量, 于是用 t 表示的每一个子式也是最终表达式的一部分, 所以对于临时变量的存放把它们也压入存放变量的栈就可以了。

```
# 取出一个简单表达式
op = op_stack.pop()
val2 = val_stack.pop()
val1 = val_stack.pop()
temp_var = 't{}'.format(temp_count)
temp_count += 1
val_stack.append(temp_var)
```

实验过程及内容:

基于简单算术表达式的赋值语句的翻译与中间代码生成

1. 预处理

读取输入文件 input.txt, 然后首先根据分号将一整行的字符串分割成多个算术表达式

```
with open('input.txt', 'r') as f:
    input_str = f.read()

statements = input_str.strip().split(';')
```

初始化临时变量计数器、中间代码

```
code_str = ''
temp_count = 1
```

定义运算符优先级

定义优先级

```
def precedence(op):  
    if op == '(':  
        return 0  
    elif op == '+' or op == '-':  
        return 1  
    elif op == '*' or op == '/':  
        return 2  
    else:  
        return 3
```

2. 生成中间代码

(1) 定义两个栈，分别用于存放运算符和变量。

```
op_stack = [] # 运算符栈  
val_stack = [] # 变量栈
```

然后对右侧表达式进行识别：

(2) 如果识别到单个字母，用一个临时变量进行存储，并把它压入变量栈。

```
if ch.isalpha() or ch.isdigit():  
    temp_var += ch  
    val_stack.append(temp_var)
```

(3) 对于识别到的运算符，首先是对左右括号进行判断。如果存在左括号，因为它的优先级最高而且它需要跟右括号进行匹配，所以此时无条件压入栈。

```
if ch == '(':  
    op_stack.append(ch)
```

如果识别到右括号，则查找栈中的左括号，如果在找到左括号之前存在其他符号，则取出对应的子式。如果找到左括号，直接出栈。这是因为会生成中间代码，所以实际上我们是不需要对左右括号进行输出的。

取出子式的方式：弹出运算符栈的栈顶元素以及变量栈中的两个变量，组成一个子式。每次取出一个子式，就生成一个新的临时变量进行表示，并将临时变量压入变量栈。

```
elif ch == ')':  
    while op_stack[-1] != '(':  
        # 取出一个简单表达式  
        op = op_stack.pop()  
        val2 = val_stack.pop()  
        val1 = val_stack.pop()  
        temp_var = 't{}'.format(temp_count)  
        temp_count += 1  
        val_stack.append(temp_var)  
        code_str += '{}={}{}};\n'.format(temp_var, val1, op, val2)  
    op_stack.pop()
```

(4) 如果是其他运算符，首先获取其优先级，然后判断当前运算符栈中的运算符的优先级是否大于当前正在被识别的运算符。如果是，说明需要先进行操作了。操作方法同上述识别

方法相同。

```
else:
    while len(op_stack) > 0 and precedence(op_stack[-1]) >= precedence(ch):
        op = op_stack.pop()
        val2 = val_stack.pop()
        val1 = val_stack.pop()
        temp_var = 't{}'.format(temp_count)
        temp_count += 1
        val_stack.append(temp_var)
        code_str += '{}={}{}}{};\n'.format(temp_var, val1, op, val2)
    op_stack.append(ch)
```

3. 处理剩余的运算符和变量

上述判断完成后，如果栈还未空，说明表达式还未处理完成，此时按顺序取出栈中的元素与变量栈中的头两个元素组成子式，直到栈空。操作方法与上述操作相同。

4. 写入输出文件

对于生成的中间代码，将其写入输出文件。

```
with open('output.txt', 'w') as f:
    f.write(code_str)
```

5. 添加两种错误判断

上述代码中，没有对任何错误情况进行判断，当输入的表达式不是合法的表达式时，对栈的处理自然存在错误。这里简单添加了对所给输入表达式的错误判断，在生成中间代码操作开始前，需要先判断是否存在这两种错误。

```
def judge(right):
    # 如果右侧有左括号，看看是否有右括号，如果没有则出错
    if '(' in right and ')' not in right:
        print('括号不匹配，表达式错误! \n')
        return 1

    # 看看运算符是否连续，因为情况较少，所以直接定义一个错误列表进行列举，再看right里面有没有
    for i in error_list:
        if i in right:
            print('运算符连续，表达式错误!\n')
            return 1
```

运算符连续的情况较少，所以在编程中直接定义一个列表对这些错误进行存放：

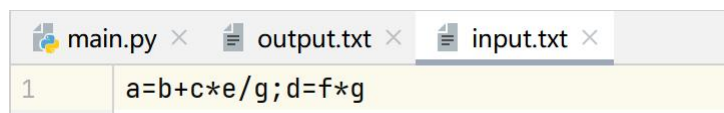
```
error_list = ['**', '*-', '**+', '*/', '+++', '+-', '**+', '+/', '-+', '--', '-*', '-/', '/+', '/-', '/*', '///']
```

然后在生成中间代码操作开始之前，对是否存在错误进行判断：

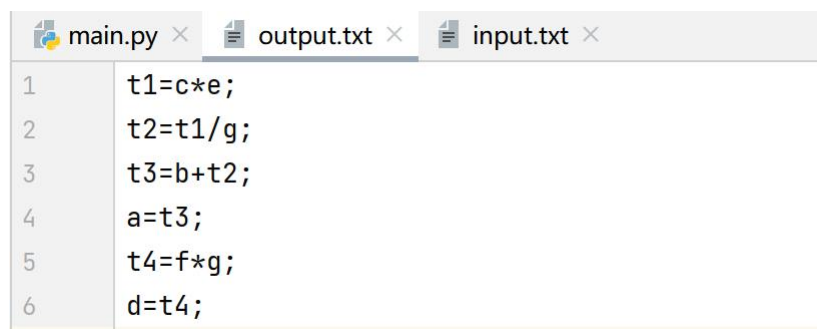
```
if (judge(right) != 1):
    for ch in right:
```

实验结论：

1. 在输入文件中输入题目所给样例

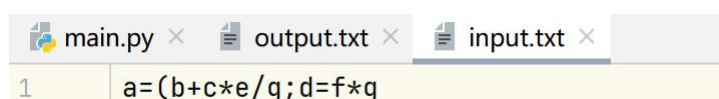


得到输出文件为:



```
main.py × output.txt × input.txt ×
1      t1=c*e;
2      t2=t1/g;
3      t3=b+t2;
4      a=t3;
5      t4=f*g;
6      d=t4;
```

2. 在输入文件中任意位置加左括号, 对第一种错误判断进行检验



```
main.py × output.txt × input.txt ×
1      a=(b+c*e/g;d=f*g
```

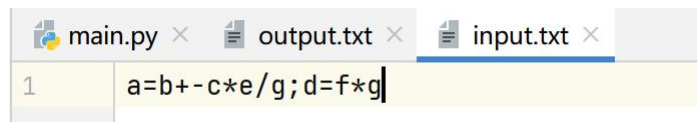
可以看到运行结果为:

5\test5\main.py

括号不匹配, 表达式错误!

进程已结束, 退出代码0

3. 在输入文件中任意位置设置连续运算符, 对第二种错误进行检验



```
main.py × output.txt × input.txt ×
1      a=b+-c*e/g;d=f*g|
```

可以看到运行结果为:

5\test5\main.py

运算符连续, 表达式错误!

进程已结束, 退出代码0

心得体会:

1. 通过本次实验, 掌握了如何利用栈对表达式进行处理, 并生成中间代码对表达式的运算过程进行表示。在这个过程中, 还掌握了如何存储临时变量, 借助临时变量表示运算执行过程。能够通过该实验对语法分析有较好的学习, 并能够结合数据结构、算法等思维对分析过程进行编程实现。

2. 对语法制导翻译的理解:

一种将源语言翻译成目标语言程序的方法, 其特点是翻译过程中与语法分析过程相结合, 即翻译过程中考虑源语言程序的语法结构, 同时生成目标语言程序的语法结构。在语法制导翻译中, 翻译规则与语法规则相对应, 这些规则可以在语法分析时被执行, 以生成目标语言程序的语法结构。通过语法制导翻译, 可以实现源语言程序向目标语言程序的无缝切换, 从而提高程序的可读性和可维护性。

3. 中间代码生成的理解:

编译器中的一个重要阶段，它将源代码翻译成一种中间表示形式，以便于后续优化和代码生成。中间代码通常是一种低级别的语言，比如三地址码、四元式等，它们比高级语言更接近机器语言，但又比机器语言更容易被编译器处理和优化。

中间代码生成的主要任务是将源代码转换为中间代码表示形式，并且保留源代码的语义信息。在这个过程中，编译器会进行符号表的构建和类型检查等操作，以确保生成的中间代码是正确的。通常，中间代码生成的结果会被传递给后续的优化和代码生成阶段，以最大程度地提高程序的执行效率。

指导教师批阅意见:

成绩评定：

指导教师签字：蔡树彬
2023 年 6 月 22 日

备注: