PL/0 编译器

本题分为两个部分，分别是编译器的语义分析+目标代码生成部分，以及假想栈式计算机的解释器部分。

编译原理课程实验 语义分析器以及目标代码生成

概述

你需要完成一个PL/0语言的编译器。这个实验分为若干部分。在这个部分中，你需要完成PL/0语言的语义分析器以及目标代码生成部分。

你应当已经正确完成了词法分析器和语法分析器。如果没有，请先完成之前的实验。

词法分析器会向语法分析器提供词语流，而语法分析器则按规范推导/规约生成一棵语法树。

在本实验中，我们将用语法制导翻译的方式，完成给定语法的语义分析以及目标代码生成。

你需要修改你的语法生成器，使其在生成语法树的同时，进行语义分析以及目标代码生成。

你可以将本实验与下一个实验"目标代码解释器"一同完成。它们可以互相完成测试，从而简化你的调试过程。

这一部分不进行单独的自动评测，将与下一个实验"目标代码解释器"一同进行自动评测。

处理说明部分

对每个PL/0语言过程的说明部分（包括主程序）制作名字表，填写所在层次、属性，并分配相对地址。

主程序的层次是0，在主程序中定义的过程则为1层。PL/0语言不允许层数超过3，具体解释请参见语法分析器实验。

例如，对于以下的过程说明部分

```
const a=35, b=49;
var c,d,e;
procedure p;
var g;
```

应当生成如下名字表

NAME	KIND	PARAMETER1	PARAMETER2
a	CONSTANT	VAL:35	–
b	CONSTANT	VAL:49	–
c	VARIABLE	LEVEL: LEV	ADR: DX
d	VARIABLE	LEVEL: LEV	ADR: DX+1
e	VARIABLE	LEVEL: LEV	ADR: DX+2
p	PROCEDURE	LEVEL: LEV	ADR:
g	VARIABLE	LEVEL: LEV+1	ADR: DX

其中，LEVEL给出的是层次，DX是每一层局部量的相对地址。

对于过程名的地址，需要等待目标代码生成后再填写。

考虑到需要存储调用信息、返回信息等维护函数运行的数据，在本实验中，DX取3.

处理语句以及代码生成

对语句逐句分析，如果语法和语义正确则生成目标代码。处理标识符的引用时应当查询上文的名字表，检查是否存在正确的定义。

生成的目标代码是一种假想栈式计算机的汇编语言，其格式如下：

```
f l a
```

其中f为功能码，l代表层次差，a代表位移量。

这种假想栈式计算机有一个无限大的栈，以及四个寄存器IR,IP,SP,BP。

IR，指令寄存器，存放正在执行的指令。

IP，指令地址寄存器，存放下一条指令的地址。

SP，栈顶寄存器，指向运行栈的顶端。

BP，基址寄存器，指向当前过程调用所分配的空间在栈中的起始地址。

共有8种目标码

- LIT: l域无效，将a放到栈顶
- LOD: 将于当前层层差为l的层，变量相对位置为a的变量复制到栈顶
- STO: 将栈顶内容复制到于当前层层差为l的层，变量相对位置为a的变量
- CAL: 调用过程。l标明层差，a表明目标程序地址
- INT: l域无效，在栈顶分配a个空间
- JMP: l域无效，无条件跳转到地址a执行
- JPC: l域无效，若栈顶对应的布尔值为假（即0）则跳转到地址a处执行，否则顺序执行
- OPR: l域无效，对栈顶和栈次顶执行运算，结果存放在次顶，a=0时为调用返回

你可以自行定义OPR中的运算和a的对应关系。

PL/0的语义细节

可调用范围

函数可以调用比自己先声明的同层次函数，被自己直接包含的函数，函数本身，以及包含自己的高层次函数。

示例中，

```
const k=10;
var a, b;
procedure f1;
    var b,c;
    procedure f2;
        var c,d;
        procedure f3;
            var d,e;
            begin
            <0>
            end;
        begin
        <1>
        end;
    procedure f4;
        var c, e;
        begin
        <2>
        end;
    begin
    <3>
    end;
begin
<4>
end.
```

<0>处可以调用f1,f2,f3

<1>处可以调用f1,f2,f3

<2>处可以调用f1,f2,f4

<3>处可以调用f1,f2,f4

<4>处可以调用f1

变量访问

总是访问层次低于此层的变量，如果出现同名变量则取层次最靠近自己的那个变量。例如上例中，

<0>处可以访问常量k，主程序声明的变量a，f1中声明的变量b，f2中声明的变量c，以及f3中声明的变量d,e

<1>处可以访问常量k，主程序声明的变量a，f1中声明的变量b，f2中声明的变量c,d

<2>处可以访问常量k，主程序声明的变量a，f1中声明的变量b，f4中声明的变量c,e

<3>处可以访问常量k，主程序声明的变量a，f1中声明的变量b,c

<4>处可以访问常量k，主程序声明的变量a,b

请注意递归调用的情况中，访问的变量是递归过程中最接近的变量。例如：

```
var n, ans;
procedure fact;
    var depth;
    begin
        if n > 1 then
        begin
            depth := n;
            n := n - 1;
            call fact;
            ans := depth * ans;
        end;
    end;
begin
    ans := 1;
    read(n);
    call fact;
    write(ans)
end.
```

中，每一次调用过程中的depth变量是不同的。

标识符的使用

你应当只给声明为变量的标识符赋值、输入。

你应当只调用声明为函数的标识符。

你不应当使用声明为函数的标识符作为表达式的项或将其作为输出语句的标识符。

测试

由于可以自定义假想计算机的实现细节，这一个实验部分不进行单独的自动评测。我们会将其于下一个实验，假想计算机代码的解释器共同评测。

届时，你可以于SDU OJ上自动测试这两个实验的正确性。如果通过测试，这两个实验的正确性分数将同时自动评定为满分。

如果你不能或不愿自动测试这两个实验，请联系助教进行人工验收。我们会人工给出这两个实验的正确性分数。

编译原理课程实验 目标代码解释器

你需要完成一个PL/0语言的编译器。这个实验分为若干部分。在这个部分中，你需要完成PL/0语言的目标代码解释部分。

完成了目标代码生成部分后，你已经可以将一个PL/0语言程序生成为目标代码了。但是，假想计算机的机器指令并不能在现代计算机上直接运行。你还需要为此完成一个解释器来执行对应的机器指令。

在正确完成这一部分实验内容后，祝贺你，你的程序已经能够正确运行PL/0语言程序了。你完成了一个自己的编译器。

概述

请参阅"语义分析器以及目标代码生成"实验部分，以及回忆你对该虚拟机的设计来了解目标代码解释器所需要处理的指令和寄存器。

编译器从标准输入读取PL/0源程序，并将编译后的机器码输出到标准输出。

解释器需要从**program.code**读取机器码，从标准输入读取read指令对应的数字，并将write指令对应的数字输出到标准输出。

输入时，有若干行，每行一个数字，保证输入的数字不少于你应当读取的个数。

输出若干行，每行一个数字，代表write指令的输出。

测试时，保证输入、输出和中间计算结果可以使用32位带符号整数表示。

测试

如果你不能或不愿自动测试这两个实验，请联系助教进行人工验收。我们会人工给出这两个实验的正确性分数。

提交一个zip文件，包括了你的PL/0语言的源代码，PL/0解释器的源代码，以及一个build.sh文件来告诉我们如何构建你的工程。

执行build.sh后，应当生成两个文件Compiler和Interpreter，分别代表编译器和解释器。

对于Java来说，你提供的Java文件编译后需要可以被java Compiler执行以及java Interpreter执行（即这两个类中包括main函数）

我们会提供若干组编写好的PL/0程序，使用你的编译器编译出目标机器代码，再用若干组测试数据，使用你的解释器运行测试。

你的编译器的输出需要满足输出的是合法的假想栈式虚拟机机器码

如果输入的程序包含词法、语法或语义错误，你的编译器应当返回非0值。

build.sh的编写

类似于之前的build.sh，执行后应当生成两个文件Compiler和Interpreter，分别代表编译器和解释器，区分大小写。

以下是示例:

C

```
#! /bin/bash
gcc -c compiler1.c -o compiler1.o -O2
gcc -c compiler2.c -o compiler2.o -O2
gcc -c tools.c -o tools.o -O2
gcc tools.o compiler1.o compiler2.o -o Compiler -lm

gcc -c interpreter.c -o interpreter.o -O2
gcc tools.o interpreter.o -o Interpreter -lm
```

C++

```
#! /bin/bash
g++ -c compiler1.cpp -o compiler1.o -O2
g++ -c compiler2.cpp -o compiler2.o -O2
g++ -c tools.cpp -o tools.o -O2
g++ tools.o compiler1.o compiler2.o -o Compiler -lm

g++ -c interpreter.cpp -o interpreter.o -O2
g++ tools.o interpreter.o -o Interpreter -lm
```

Java

```
#! /bin/bash
javac *.java
```

样例

样例1

PL/0代码

```
var a,b;
begin
    read(a,b);
    a := a+b;
    write(a)
end.
```

测试数据 输入

```
233
666
```

输出

```
899
```

样例2

PL/0代码

```
const c=10;
var a,b;
begin
    read(a,b);
    c := a+b;
    write(c)
end.
```

编译器返回非零值（编译失败，给常数赋值）

这里还有一组很复杂的程序（真的很复杂）spfa.zip

展望

完成本实验后，这门课程的实验内容就已经全部结束。如果你还意犹未尽，可以思考以下问题：

- 如何优化生成的目标代码
- 如何扩展PL/0语法以使其更实用（例如，增加数组支持）
- 尝试生成可以直接在x86计算机上运行的PE文件或ELF文件
- and so on

展望部分不是实验内容，不计入成绩。