语义实验

实验目标

在语法实验的基础上，修改语义动作，输出SysY代码对应的x86 AT&T汇编代码。

实验指导

以下仅供参考，可以根据自己的喜好选择其他方法进行实验。

为了简化代码难度，以下提示都有一定程度上根据**SysY**的语法进行优化，不具备普适性。

（1）符号表

使用栈式结构储存符号表：

```
vector < map <string, var> *>
```

最外层的vector起到一个类似栈的作用（也可以使用C++的stack实现），每进入一个过程，就新创建一个map（相当于建立一张空的符号表）加入到vector的尾部。当离开一个过程时，就将vector的尾部删除。

var为结构体，其中储存了一个变量的有用的信息，包括该变量的类型（常量、数组、整形变量、函数等）、该变量的值（当且仅当为常量时，该值有效）、该变量在栈上的偏移量、该变量的维数（当且仅当该变量为数组时，该值有效）等。

此处使用map将变量的名称映射到一个var型的结构体上，可以快速地通过变量的名称，判断是否定义过这个变量、使用这个变量等。

（2）汇编指令

汇编指令可以通过vector进行存储：

```
vector < string >
```

好处在于对汇编指令的存储操作可以动态进行，不用纠结数组的大小。回填的时候，可以先向vector中push_back一个空字符串，后续再修改该字符串的内容。

关于汇编的提示：

- 输入输出

  可以直接使用__isoc99_scanf@PLT和printf@PLT等函数直接进行输出，在汇编转可执行文件时链接相应的库即可。需要注意的是，调用输入输出函数时，需要保证栈向**16**对齐，即%rsp为16的倍数。刚进入main函数时，可以认为此时%rsp已经向16对齐。

  此处对输入输出进行一些约定：

  ```
  scanf(x);       // 输入一个int类型的整数到x中
  printf(x);      // 输出x并换行
  ```

  以下给出一个示例：

  ```
  .LC0:
      .string "%d"                # 定义模式串
      .text
      .globl  main
      .type   main, @function
  main:
      pushq   %rbp
      movq    %rsp, %rbp

      subq    $16, %rsp           # 对齐
      leaq    -12(%rbp), %rsi     # 给定输入的地址
      leaq    .LC0(%rip), %rdi    # 给定输入形式
      call    __isoc99_scanf@PLT  # 输入

      movl    -12(%rbp), %esi     # 给定输出内容
      leaq    .LC0(%rip), %rdi    # 给定输出形式
      call    printf@PLT          # 输出

      addq    $16, %rsp
      popq    %rbp                # 还原
  ```

- 寄存器、内存、立即数

汇编中相当一部分多操作数的指令，要求操作数中需要有一个是寄存器。比如mov指令不能将数据从内存拷贝到内存。因此建议多操作数的指令，先将操作数移动到寄存器上，然后在寄存器上进行操作，最后将寄存器上操作得到的结果移动到正确的地方。

- 除法操作

  除法操作需要将被除数（int类型）放到%eax中（long long则存放在%rax），然后执行cltd指令（long long为cltq），接着对除数执行idiv指令。除法得到的商储存在%rax中，余数储存在%rdx中。

  以下是一段汇编代码的示例：

```
movl    $1, -16(%rbp)       # 被除数
movl    $2, -12(%rbp)       # 除数

movl    -16(%rbp), %eax     # 设置被除数
cltd
idivl   -12(%rbp)           # 进行除法
movl    %eax, -8(%rbp)      # 得到商

movl    -16(%rbp), %eax     # 设置被除数
cltd
idivl   -12(%rbp)           # 进行除法
movl    %edx, -4(%rbp)      # 得到余数
```

- 寄存器和栈

  读写寄存器的速度比读写栈更快，因此多使用寄存器会有更高的效率。但是，为了简化编程难度，可以将临时变量都开在栈上，尽可能地减少寄存器的使用。

  使用寄存器的时候，需要将callee-saved的寄存器push到栈上，退出时再pop出来，以防止程序结束时发生崩溃。

如果汇编存在其他问题，可以自己编写相关的C/C++程序，将其编译成汇编后，和自己的代码进行对比。

使用如下命令将C程序编译为汇编代码（C代码文件为a.c），生成的汇编代码为a.s：

```
gcc -S a.c
```

使用如下命令编译汇编代码（代码文件为a.s，可执行文件为a）：

```
gcc a.s -o a
```

（3）常数处理

对于常数，很容易可以发现，当且仅当常数和常数的计算结果为常数，其他情况都不是常数。同时，常数可以在编译时确定它们的值。因此，对于一个变量，我们可以标记其是否为常数，如果是常数，直接计算出这个变量的值，在后续的运算中直接使用这个值即可，不用刻意把它放在内存上奇怪的地方。

（4）数组处理

数组可以利用匹配右括号、在语法树上进行回溯的时候处理。

- 对于数组定义，此时括号内部都是常数，直接计算即可。在匹配的时候可以记录数组每一维的维数，同时统计数组所需要占据的空间大小。
- 对于数组的使用，需要递归从最后一维开始计算该变量所在的地址。需要注意的是，这时候不一定每一维都是常数，不能直接通过计算得到，需要一步一步使用汇编实现地址的计算。
- 数组作为函数参数传递，本质是传递数组地址，可以直接在符号表中进行记录，而无需再汇编指令中进行传递。

（5）控制流语句

- 对于if语句，在条件判断的时候处理出truelist和falselist，然后在if的分支中打标记，使用回填的方法和处理truelist和falselist中指令的跳转位置。

  由于汇编代码需要保证栈中内容的对齐，所以在进入分支之前需要匹配一个空字符串，其语义动作将栈中的内容对齐，在离开分支的时候，需要将栈中的内容进行恢复，恢复到进入循环之前的状态。

- 对于while语句，由于其判断语句可能会执行多次，所以在进入判断语句之间就要对栈中的内容进行维护和对齐，退出时要恢复到原来的样子。

- 对于break和continue语句，需要要求其在while内才能使用。对于这两条语句，需要先对栈中内容进行恢复，才能进行跳转。否则最后程序结束时栈顶指针无法恢复到原来的状态。同时，需要记录break和continue针对的是哪个循环语句，需要弹出栈顶多少的内容。

  由于break和continue也需要跳转到循环的尾部，而跳转的时候循环尾部还没进行标记，所以需要在标记之后进行回填。

- 进入while和if时，进入了一个新的scope，需要对符号表进行相应的处理。

- 对于while和if的语法进行了一定程序上的修改，修改之后如下所示（仅供参考）：

```
IF BLL Cond BRL SetLabel Enter Stmt Exit SetLabel
IF BLL Cond BRL SetLabel Enter Stmt Exit SetLabel OutIF
SetLabel Enter Stmt
WHILE BLL SetWhileLabel Enter Cond SetOutLabel BRL SetLabel
Stmt SetLabel Exit
```

其中，BLL匹配左括号，Cond匹配条件语句，BRL匹配右括号，Stmt匹配if中需要执行的语句。同时，新增加一些非终结符：SetLabel为设置一个标签，用于程序的跳转，Enter为进入时备份先前的状态，Exit为退出时恢复原来的状态。对于while语句，由于其比较特别，所以有细微区别，各个终结符的作用与if语句的大致相同。这些新增加的非终结符都是匹配空字符串的。

（6）函数调用

函数调用的关键点在于参数传递、返回值、维护栈和寄存器三个部分。

- 对于无参的函数调用，直接使用call进行调用即可。
- 对于void型函数要判断其是否有返回值（有的话需要报错）。对于int型函数，要注意在函数返回的时候将返回值拷贝到%eax上。
- 对于有参的函数调用，在LALR匹配的过程中，记录下每一个参数。然后将这些参数储存到对应的寄存器上（为了简化代码，可以将这些参数都放在栈上），之后使用call调用函数。进入新的函数之后，再从栈中将这些变量取出来，取到当前的段上。

  需要注意的是，调用call指令会将下一条指令的地址压入栈中，使用栈上的变量的时候需要注意这条地址占用的内存空间。
- 为了方便起见，进入一个函数的时候需要保证此时栈顶是向16对齐的。从一个函数离开时，需要还原所有callee-saved的寄存器，同时需要恢复栈顶和栈底的指针。