CS4121 Pascal Junior Compiler Project 2: An Expression Evaluator
Due Date: Wed, Feb. 19, 2025 at *11:59pm*

Purpose

The purpose of this project is to gain experience in giving meaning to a programming language by generating x86-64 assembly for a subset of Pascal Junior (PJ). Specifically, you will be generating assembly for string output operations, integer I/O operations, integer arithmetic and logical expressions and assignment statements. Completely read this document.

Project Summary

In this project, you will add/change actions to the provided parser that will do the following.

1. Assign space for global integer and boolean variables.
2. Generate assembly pseudo-ops for string constants including the format strings in IO statements.
3. Generate assembly to print string constants with or without a formatting constraint.
4. Generate assembly to print integers with or without a formatting constraint..
5. Generate assembly to read integers.
6. Generate assembly to compute integer expressions.
7. Generate assembly to compute logical expressions.
8. Generate assembly to assign values to integer/boolean variables.

Prologue and Epilogue Code

Since you are converting a PJ program to x86-64 assembly, you must begin each assembly file with any data declarations and a declaration of where the main program begins. This is done with the following code:

```
        .section        .rodata
        #declare format strings and globals in this area
        .text
        .globl main
        .type main,@function
main:   nop
        pushq %rbp
        movq %rsp, %rbp
```

This code declares a data section, where you can declare the area for the globals and the read and write format strings, followed by a text section (instructions) containing a declaration of the main routine and instruction to set up the frame pointer (`%rbp`). Each x86-64 assembly file should begin with this sequence. If you assign space in the static data area for global variables, then the space may be allocated with directives after the `.section` directive and before the `.text` directive.

To end an x86-64 assembly routine, add the code

```
        leave
        ret
```

These instructions exit a program.

Assigning Variable Space

Memory for global variables declared in a PJ program may be allocated in the static area. The PJ declarations

```
VAR i, j, k : LONGINT;
```

require 12 bytes of space. We can add the following x86-64 declaration to the data section

```
.comm _gp, 12, 4
```

We will use these declarations for the remaining examples in this document. You can treat boolean variables as integer variables in this project. Variables may be addressed as a positive offset off of the global pointer, _gp. The first variable is located at the global pointer with offset 0, and each successive variable is located 4 bytes from that point. For example, one may load the value of `j` above with the following instructions:

```
leaq _gp(%rip), %rbx
addq $4, %rbx
movl (%rbx), %eax
```

String Constants

A PJ program may use string constants in write statements. These constants are declared in the data section using the `.string` pseudo-op. For the PJ statement,

```
WRITELN(‘Hello‘);
```

the following declaration must be added to the data section of the assembly file:

```
.string_const0: .string "Hello"
```

The label `.string_const0` is implementation dependent. You may name your string constants however you wish.

Printing Strings

Printing strings requires using the C library function `printf`. As an example, the code to implement the `WRITELN` statement in the previous section would be:

```
leaq .string_const1(%rip), %rdi
leaq .string_const0(%rip), %rsi
movl $0, %eax
call printf@PLT
```

You will need to declare the format string constant in the data section:

```
.string_const1: .string "%s\n"
```

Printing Integers

Printing integers involves a format string as the first argument and the actual integer passed as the second argument. As an example, to implement the statement

```
WRTIELN(7:10);
```

the following x86-64 assembly would need to be generated:

```
movl $7, %ebx
leaq .string_const2(%rip), %rdi
movl %ebx, %esi
movl  $0, %eax
call printf@PLT
```

You will need to declare the format string constant in the data section:

```
.string_const2: .string "%10d\n"
```

Reading Integers

To read an integer, use the C library routine `scanf`. As an example, to read the integer variable `j` declared above, we implement

```
READ(j);
```

using the following instructions:

```
leaq .string_const3(%rip), %rdi
leaq _gp(%rip), %rbx
addq $4, %rbx
movq %rbx, %rsi
movl $0, %eax
call scanf@PLT
```

You will need to declare the format string constant in the data section:

```
.string_const3: .string "%d"
```

Integer Arithmetic Expressions

In x86-64 assembly, operations can be done on registers. The best way to generate code is to store all intermediate values in x86-64 registers. Using the registers `%eax, %ebx, %ecx, %edx, %esi, %edi, %r8d, %r9d, %r10d, %r11d, %r12d, %r13d, %r14d` and `%r15d` is sufficient. You should not need any other temporary memory locations. Note that you will need to expand registers to their 64-bit correspondences for address calculation. For an operation, the operands should all be put into registers, the left operand should be the result register, the operations should be performed and then the right operand register should be released to be reused later. As an example, the statement

```
WRITELN(i+j:10);
```

might result in the code

```
leaq _gp(%rip), %rbx
addq $0, %rbx
movl (%rbx), %ecx
leaq _gp(%rip), %rbx
addq $4, %rbx
movl (%rbx), %r8d
addl %r8d, %ecx
leaq .string_const1(%rip), %rdi
movl %ecx, %esi
movl $0, %eax
call printf@PLT
```

Note that integer division is not as straightforward. See the notes from class on x86-64 assembly to figure out how to do integer division.

Logical Expressions

Logic expressions are similar to arithmetic expressions. For the x86-64, the value for `false` is 0 and the value for `true` is 1. You can use the bit-wise x86-64 logical instructions, `and`, `or`, `xor`, and `not` to implement logical expressions.

There are multiple ways to implement relational expressions. As an example, the statement

```
WRITE(i>j);
```

might result in the code

```
leaq _gp(%rip), %rbx
addq $0, %rbx
movl (%rbx), %ecx
leaq _gp(%rip), %rbx
addq $4, %rbx
movl (%rbx), %r8d
cmpl %r8d, %ecx
movl $0, %ecx
movl $1, %ebx
cmovg %ebx, %ecx
leaq .string_const0(%rip), %rdi
movl %ecx, %esi
movl $0, %eax
```

Note that instruction `cmpl %edx, %ecx` compares the values of the two registers and update the flags. Later, `cmovg %ebx, %ecx` will copy `%ebx` to `%ecx` if the flag is "greater than". Otherwise `%ecx` remains as 0. The other conditional move instructions you might need are `cmove, cmovne, cmovge, cmovl, cmovle` where `e` means equal, `g` greater than, and `l` less than.

Storing Integer Variables (Assignment)

To store a value in a variable, first compute the address and then store the value into that location. For example, the statement

```
i := 5;
```

could be implemented with

```
leaq _gp(%rip), %rbx
addq $0, %rbx
movl $5, %ecx
movl %ecx, (%rbx)
```

Requirements

Write all of your code in C or C++. It will be tested on CS lab machines or MTU provided remote Linux systems, and MUST work there. You will receive no special consideration for programs which "work" elsewhere.

Input. I have provided my solution to Project 1 as a Makefile project in `PJProject2.tgz`. Sample input is provided in the directory `PJProject2/input`. I will be happy to go over my code for students if they wish to use this version of the project – please reach out if you have questions. To run your compiler, use the command

```
pjc <file>.pas
```

which will output to `<file>.s` if you uncomment lines 723-730 and line 741 of PJParser.y.
To create an executable, run the following command on the assembly file

```
gcc -o <file> <file>.s
```

You may then directly run the executable. You can compare your output with the Project 1 solution output. They should be the same.

Submission. Your code should be well-documented. You will submit all of your files, by tarring up your working directory using the command

```
tar -czf PJProject2.tgz PJProject2
```

Submit the file `PJProject2.tgz` via Canvas. Make sure you do a 'clean' of your directory before executing the tar command. This will remove all of the '.o' files and make your tar file much smaller.

An Example

Given the following PJ program (proj2exp.pas):

```
PROGRAM project2Example;

VAR i,j,k,l : LONGINT;

BEGIN
        i := 1; k := 3; l := 4;
        j := i + l + k;
        WRITE('j = 1+3+4 =':15);
        WRITELN(j:4)
END.
```

it may be implemented with the following x86-64 assembly.

```
        .section .rodata
        .comm _gp, 16, 4
        .string_const0: .string "%15s"
        .string_const1: .string "j = 1+3+4 ="
        .string_const2: .string "%4d\n"
        .text
        .globl main
        .type main,@function
main: nop
        pushq %rbp
        movq %rsp, %rbp
        leaq _gp(%rip), %rbx
        addq $0, %rbx
        movl $1, %ecx
```

```
movl %ecx, (%rbx)
leaq _gp(%rip), %rbx
addq $8, %rbx
movl $3, %ecx
movl %ecx, (%rbx)
leaq _gp(%rip), %rbx
addq $12, %rbx
movl $4, %ecx
movl %ecx, (%rbx)
leaq _gp(%rip), %rbx
addq $4, %rbx
leaq _gp(%rip), %rcx
addq $0, %rcx
movl (%rcx), %r8d
leaq _gp(%rip), %rcx
addq $12, %rcx
movl (%rcx), %r9d
addl %r9d, %r8d
leaq _gp(%rip), %rcx
addq $8, %rcx
movl (%rcx), %r9d
addl %r9d, %r8d
movl %r8d, (%rbx)
leaq .string_const0(%rip), %rdi
leaq .string_const1(%rip), %rsi
call printf@PLT
leaq _gp(%rip), %rbx
addq $4, %rbx
movl (%rbx), %ecx
leaq .string_const2(%rip), %rdi
movl %ecx, %esi
movl $0, %eax
call printf@PLT
leave
ret
```

We will go through this example in class.