

Collaborators: None

Name: Jyotirmay Zamre

1 File Structure & Compilation

The file structure is as follows:

1. **Asm**: This is the target directory that will contain the output assembly files for the test cases
2. **include**: This directory contains all the header files. As opposed to the previous assignment, the headers are all separated into logical units
3. **Quads**: This is the target directory that will contain the output quads for the test cases
4. **src**: This directory contains all the source code for the assignment. The main function is in Jyotirmay_Zamre_PA4_compiler.cpp.
5. **Tests**: This directory contains all the test cases
6. **Others**: The .l, .y, and makefile are all in the root directory

In order to compile the compiler, run **make** from the root directory. In order to generate the quads and the target code, run **make test** from the root directory. To cleanup output and intermediate files, run **make clean** from the root directory.

2 Lexer

The lexer is the same as the one from Programming Assignment 3.

3 Parser

The parser is the same as the one from Programming Assignment 3, along with a few updates. The symbol class now has 2 new fields - **initValue** and **isLiteral**. Constants and literals are now stored as such in the global symbol table (with initial values updated). This is needed in order to declare literals in the data segment in the generated assembly file.

4 Memory Binding

I am using an x86-64 instruction set (System V ABI) specifically. `%rip` is the current instruction pointer and is used to load values from addresses.

Global data

The `.data` section contains global variables, constants and literals. Since everything is 8 bytes, the `.quad` keyword is used for integers and the `.asciz` keyword is used for string literals. Each label becomes a named memory address in the data section.

Local Variables

Local variables are stored on the stack using negative offsets from the saved base pointer. The `__retval` variable is also stored as a local variable at a negative offset from the `rbp`.

Parameters

Parameters are stored in registers. The first 6 params are stored in the following register - `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`. If there are more, then the remaining parameters are allocated on the stack

The return address from the caller is stored at a positive offset from the saved `rbp`.

5 Target code generation

I made a new class called **asmEmitter**. This class contains all the relevant functions used for target code generation.

Separating quads

Once the quads are generated in the global quad array, a function called **separateQuads** is run. This function separates the quads for each function and the main program and remaps the destination of all jumps to match local quad indices instead of global quad indices. This is required in order to be able to generate each function's prologue, body, epilogue and correctly map the jump targets. I couldn't find a way to generate quads into function specific quad arrays while parsing so for now, I am using an inefficient method to parse all the quads in the global quad array.

Data segment

Once this is done, the data segment of the assembly file is generated first. I loop through the global symbol table (where all the globals, constants, and literals reside) and emit the required declarations for globals, constants, and literals (using `.quad` and `.asciz`).

Text segment

After the data segment is generated, the text segment is generated. Using the global table list, I loop through each symbol table and generate the function prologue, function body, and function epilogue for each function, including main.

The function prologue first stores the old base pointer at the top of the stack. It then sets the base of the new stack frame and reserves the required number of bytes local vars, temporaries, etc.

The function body is processed by looping through the quad array of the function. For each kind of quad, a handler is run which emits the specific instructions required for that kind of quad. A label is also emitted for each quad to make handling jump targets much easier, even if it is not the most optimized solution.

1. **Assignment** - First lookup the symbols in the function's symbol table. Compute the memory addresses operands. Call `mov` or `lea` in accordance with the scope of the source symbol.
2. **Arithmetic** - Arithmetic is fairly straightforward. Lookup and get the memory addresses of `arg1`, `arg2`, and `result`. Based on the operation, call `add`, `sub`, `imul`, `idiv` and handle the case where one of the args is a literal or a constant.
3. **Conditionals** - Conditionals are also straightforward. For each kind of operation, the relevant jump instruction is emitted with the correct label.
4. **Params** - Params are a bit tricky because of the fact that only the first 6 are stored in registers and the remaining are stored on the stack. A param index is used to keep track of the number of params for a particular function call. This param index is reset once a function call quad is reached.
5. **Function calls** - The number of params is stored in the quad and the stack cleanup instructions are also emitted incase the number of params is greater than 6.
6. **Function returns** - This returns a jump instruction that jumps to the function's epilogue

The function epilogue restores the stack pointer and the caller's frame pointer and returns.

6 Test cases

There are 5 provide test cases and I am able to generate assembly for each test case. I have tried executing the assembly as well and it seems to be working. The only edge case that might not work is if a function has more than 6 params. I don't think I have handled the exact details of that process correctly. However, the test cases provided seem to be working fine.

7 Makefile

make test loops through the files in the Tests directory and executes the compiler while passing the file name as an argument.