[Question 1]

The IR-Virtual instructions are translated into x86 assembly code. Each IR-Virtual instruction is mapped to one or more x86 assembly instructions. This step produces executable machine code that can be run on a computer. There were several pros and cons that our group discovered while implementing, testing, and running our code. We found the IR virtual to be much easier to read than assembly. It is difficult to understand programs for solving arithmetic questions in assembly, but it is very straightforward in IR virtual which was a positive. The con that we as a group struggled with is when we wrote our programs for the IR virtual and we were trying to test it, the test cases were not as straightforward as the other test cases we have dealt with previous classes and past projects in this class. Most of the test cases were in python which was easier to implement compared to when we were trying to implement test cases in IR virtual. Overall, we had a positive experience with IR virtual even though it was hard to implement the test cases as the programs themselves were easy to write and implement.

```
racket compiler.rkt -v test-programs/sum1.irv
two.irv: ((mov-lit r0 2) (print r0))
sum2.irv ((mov-lit r0 10)
  (mov-lit r1 6)
  (mov-lit r2 0)
  (add r2 r1)
  (add r2 r0)
  (print r2))
sum3.irv ((mov-lit r0 5)
  (mov-lit r1 3)
  (mov-lit r2 0)
  (add r2 r1)
  (add r2 r0)
  (print r2))
```

(Also pass in -m for Mac)

[Question 2]

For this task, you will write three new .ifa programs. Your programs must be correct, in the sense that they are valid. There are a set of starter programs in the test-programs directory now. Your job is to create three new .ifa programs and compile and run each of them. It is very much possible that the compiler will be broken: part of your exercise is identifying if you can find any possible bugs in the compiler.

For each of your exercises, write here what the input and output was from the compiler. Read through each of the phases, by passing in the -v flag to the compiler. For at least one of the programs, explain carefully the relevance of each of the intermediate representations.

For this question, please add your .ifa programs either (a) here or (b) to the repo and write where they are in this file.

```
Arith3.ifa: (* 10 10) Output: 100
Arith3.asm:
section .data
    int_format db "%ld",10,0
    global _main
        extern _printf
section .text
_start: call _main
        mov rax, 60
        xor rdi, rdi
        syscall
_main:  push rbp
        mov rbp, rsp
        sub rsp, 48
        mov esi, 10
        mov [rbp-24], esi
        mov esi, 10
        mov [rbp-16], esi
        mov esi, [rbp-24]
        mov [rbp-8], esi
        mov edi, [rbp-16]
        mov eax, [rbp-8]
        imul eax, edi
        mov [rbp-8], eax
        mov rax, [rbp-8]
        jmp finish_up
finish_up:      add rsp, 48
        leave
        ret
```

```
Cond2.ifa: (cond [(- 10 (* 2 3)) (print 4)]
      [(* 10 (+ 2 2)) (print 40)]
      [else (print 5)])
Cond2.asm:
section .data
    int_format db "%ld",10,0
    global _main
    extern _printf
section .text
_start: call _main
    mov rax, 60
    xor rdi, rdi
    syscall
_main:  push rbp
    mov rbp, rsp
    sub rsp, 304
    mov esi, 10
    mov [rbp-48], esi
    mov esi, 2
    mov [rbp-96], esi
    mov esi, 3
    mov [rbp-88], esi
    mov esi, [rbp-96]
    mov [rbp-32], esi
    mov edi, [rbp-88]
    mov eax, [rbp-32]
    imul eax, edi
    mov [rbp-32], eax
    mov esi, [rbp-48]
    mov [rbp-80], esi
    mov edi, [rbp-32]
    mov eax, [rbp-80]
    sub eax, edi
    mov [rbp-80], eax
    mov esi, 0
    mov [rbp-40], esi
    mov edi, [rbp-40]
    mov eax, [rbp-80]
    cmp eax, edi
    mov [rbp-80], eax
    jz lab1275
    jmp lab1277
lab1275:    mov esi, 4
    mov [rbp-16], esi
    mov esi, [rbp-16]
    lea rdi, [rel int_format]
    mov eax, 0
    call _printf
    mov rax, 0
    jmp finish_up
lab1277:    mov esi, 10
    mov [rbp-72], esi
    mov esi, 2
    mov [rbp-152], esi
    mov esi, 2
    mov [rbp-144], esi
    mov esi, [rbp-152]
    mov [rbp-136], esi
    mov edi, [rbp-144]
    mov eax, [rbp-136]
    add eax, edi
    mov [rbp-136], eax
    mov esi, [rbp-72]
    mov [rbp-128], esi
    mov edi, [rbp-136]
    mov eax, [rbp-128]
    imul eax, edi
    mov [rbp-128], eax
    mov esi, 0
    mov [rbp-104], esi
    mov edi, [rbp-104]
    mov eax, [rbp-128]
    cmp eax, edi
    mov [rbp-128], eax
    jz lab1283
    jmp lab1285
lab1283:    mov esi, 40
    mov [rbp-64], esi
    mov esi, [rbp-64]
    lea rdi, [rel int_format]
    mov eax, 0
    call _printf
    mov rax, 0
```

```
    jmp finish_up
lab1285:    mov esi, 5
    mov [rbp-120], esi
    mov esi, [rbp-120]
    lea rdi, [rel int_format]
    mov eax, 0
    call _printf
    mov rax, 0
    jmp finish_up
finish_up:  add rsp, 304
    leave
    ret
```

```
2587.ifa: 2587
2587.asm:
section .data
    int_format db "%ld",10,0
        global _main
        extern _printf
section .text
 _start:        call _main
        mov rax, 60
        xor rdi, rdi
        syscall
_main:  push rbp
        mov rbp, rsp
        sub rsp, 16
        mov esi, 2587
        mov [rbp-8], esi
        mov rax, [rbp-8]
        jmp finish_up
finish_up:      add rsp, 16
        leave
        ret
```

[Question 3]

Describe each of the passes of the compiler in a slight degree of detail, using specific examples to discuss what each pass does. The compiler is designed in series of layers, with each higher-level IR desugaring to a lower-level IR until ultimately arriving at x86-64 assembler. Do you think there are any redundant passes? Do you think there could be more?

In answering this question, you must use specific examples that you got from running the compiler and generating an output.

Arith to IfArith:

Explanation: This step simplifies the Arith-IfArith language by removing the let construct and replacing variable references with their corresponding values.

Example Call: Given an Arith-IfArith expression `(let ([x 3]) (* (+ x 2) (- x 1)))`.

Output: The equivalent IfArith expression `(* (+ 3 2) (- 3 1))`.

IfArith to IfArith-Tiny:

Explanation: IfArith-Tiny is a simplified version of IfArith where let expressions can only have literals as bindings. This step converts IfArith expressions to IfArith-Tiny by replacing all variable bindings with literals.

Example Call: Given an IfArith expression `(+ 2 (* 3 4))`.

Output: The IfArith-Tiny equivalent `(+ 2 (let ([temp1 (* 3 4)]) temp1))`.

IfArith-Tiny to ANF:

Explanation: Administrative Normal Form (ANF) is a form where all subexpressions are simple variable bindings or primitive operations. This step converts IfArith-Tiny expressions to ANF by ensuring that all expressions are in this normalized form.

Example Call: Starting with an IfArith-Tiny expression (let ([temp1 (* 3 4)]) (+ 2 temp1)).

Output: The ANF representation `(let ([temp1 (* 3 4)]) (let ([temp2 (+ 2 temp1)]) temp2))`.

ANF to IR-Virtual:

Explanation: IR-Virtual is an intermediate representation language that is closer to machine code. This step converts ANF expressions to IR-Virtual instructions, which are low-level operations that closely resemble assembly language instructions.

Example Call: Taking an ANF expression `(let ([temp1 (* 3 4)]) (let ([temp2 (+ 2 temp1)]) temp2))`.

Output: IR-Virtual instructions:

```
(mov-lit temp1 12)

(mov-reg temp2 temp1)

(add temp2 2)
```

IR-Virtual to x86:

Explanation: Finally, the IR-Virtual instructions are translated into x86 assembly code. Each IR-Virtual instruction is mapped to one or more x86 assembly instructions. This step produces executable machine code that can be run on a computer.

Example Call: Converting IR-Virtual instructions to x86 assembly.

Output: Corresponding x86 assembly code:

```
section .data
   int_format db "%ld",10,0

section .text

global _main
extern printf
```

```
_main:
   push rbp
   mov rbp, rsp
   sub rsp, 16
   mov esi, 12
   mov rax, 2
   add rax, rsi
   lea rdi, [rel int_format]
   mov eax, 0
   call printf
   add rsp, 16
   leave
   ret
```

[Question 4]

This is a larger project, compared to our previous projects. This project uses a large combination of idioms: tail recursion, folds, etc.. Discuss a few programming idioms that you can identify in the project that we discussed in class this semester. There is no specific definition of what an idiom is: think carefully about whether you see any pattern in this code that resonates with you from earlier in the semester.

In this project, the main similarities between it and projects 3 and 4 are that they are all compilers in Racket used to run specifically written code. An example of something we had to do in this project was rewriting built-in racket functions using other ones that exist. For instance, within ifarith-tiny, there are implementations of functions like let*, and list mutating functions that are all written out using if statements, the let function, and other conditionals. This reminded us of the lambda encoding project where we rewrote some racket functions and constants using lambdas. Additionally, the portion of the project that we had to code the most ourselves was like both projects 3 and 4, where we were writing the built-ins as a direct style application.

[Question 5]

In this question, you will play the role of bug finder. I would like you to be creative, adversarial, and exploratory. Spend an hour or two looking throughout the code and try to break it. Try to see if you can identify a buggy program: a program that should work, but does not. This could either be that the compiler crashes, or it could be that it produces code which will not assemble. Last, even if the code assembles and links, its behavior could be incorrect.

To answer this question, I want you to summarize your discussion, experiences, and findings by adversarially breaking the compiler. If there is something you think should work (but does not), feel free to ask me.

Your team will receive a small bonus for being the first team to report a unique bug (unique determined by me).

While working through the code and running the different types of files on the terminal, it was relatively easy to run through everything and our group was able to utilize linux commands to assist us with creating and writing files directly on the terminal. However, we did encounter some issues when trying to convert .irv and . Ifa files to assembly code. First of all, we would create these files through "nano" on the terminal and write them that way, and then turn them into assembly using the racket compiler. However, when we tried to run and link the newly created assembly files, we were having trouble using nasm and ld. Although we had homebrew and the nasm package installed for x86, the terminal was giving us an error saying that it did not exist when we tried running the new files right after they were created. To fix this, we had to restart the terminal and make sure to be in the right directory for it to work. Although this may not be a bug within the code itself, we still thought it was appropriate to share since given an outlet to do so here.

[High Level Reflection]

In roughly 100-500 words, write a summary of your findings in working on this project: what did you learn, what did you find interesting, what did you find challenging? As you progress in your career, it will be increasingly important to have technical conversations about the nuts and bolts of code, try to use this experience to think about how you would approach group code critique. What would you do differently next time, what did you learn?

The project helped me understand what the different compilers are and how they are different compared to each other. What was interesting is how easy it was to implement the programs and their physical codes in IR virtual compared to assembly but on the flip side how much harder it was to implement the test cases. As a group, we had discussions on how we could find the new compilers on a personal basis and then discussed if we as a group prefer one over the other. It was interesting to have those discussions because different people have different preferences, and this was helpful in knowing what each of us liked and whether we could adapt as a group based on our strengths and weaknesses.

None of us really had any prior knowledge on how to use the other compilers and how to implement the test cases for the specific compilers to test our programs, so it was a lot of researching and learning as a group to figure out how each compiler works and how the implementations of the test cases help. It helped that we had other examples already given to us which we could use as our springboard to come up with our own programs and test cases. What we would do differently the next time around is probably spend a bit more time playing around with the compilers and see if we can find ways to translate the way we implement stuff in one compiler to a different compiler. It probably is already possible but we as a group needed more time to play around with them to understand them better.