Discussion and Reflection

The bulk of this project consists of a collection of five questions. You are to answer these questions after spending some amount of time looking over the code together to gather answers for your questions. Try to seriously dig into the project together--it is of course understood that you may not grasp every detail, but put forth serious effort to spend several hours reading and discussing the code, along with anything you have taken from it. Questions will largely be graded on completion and maturity, but the instructors do reserve the right to take off for technical inaccuracies (i.e., if you say something factually wrong). Each of these (six, five main followed by last) questions should take roughly at least a paragraph or two. Try to aim for between 100-500 words per question. You may divide up the work, but each of you should collectively read and agree to each other's answers.

[Question 1] For this task, you will write three new .irv programs. These are `ir-virtual?` programs in a pseudo-assembly format. Try to compile the program. Here, you should briefly explain the purpose of ir-virtual, especially how it is different than x86: what are the pros and cons of using ir-virtual as a representation? You can get the compiler to to compile ir-virtual files like so: racket compiler.rkt -v test-programs/sum1.irv (Also pass in -m for Mac)

myTest1.irv
```
((mov-lit r0 300)
 (mov-lit r1 20)
 (mov-lit r2 5)
 (mul r2 r1)
 (mul r2 r0)
 (print r2))
```

myTest2.irv
```
((mov-lit r0 15)
 (mov-lit r1 10)
 (mov-lit r2 5)
 (sub r2 r1)
 (sub r2 r0)
 (print r2))
```

myTest3.irv
```
((mov-lit r0 20)
 (mov-lit r1 2)
 (mov-lit r2 2)
 (div r0 r1)
 (div r0 r2)
 (print r0))
```

The purpose of ir-virtual seems to be to get instructions into a simplified format which throws them all into a virtual register which is significantly different from x86 where there are other allocations to instructions such as registers, constants, and memory addresses. It is far easier to implement than x86 because of the simplification of allocation since it reduces the sheer mass amount of code you have to type, however this creates significant inefficiencies since if the other allocation values aren't managed then there will be a significant amount of computer resources wasted. Also it is inefficient for the processing speed as a whole since as the documentation in the program describes, the code involves shuffling in and out registers which could definitely be significantly optimized with a more thorough algorithm of some sort.

[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.

The test-cases (aka .if programs):

1:
TRANSLATE THE RACKET CODE TO IfArith LANGUAGE IN ORDER TO GET THE CODE IN BITE SIZED PIECES FOR TRANSLATION TO IfArityTiny WHICH FURTHER BREAKS DOWN CODE

```
Input source tree in IfArith:
'(if true (+ 1 9) (- 1 9))
ifarith-tiny:
'(if true (+ 1 9) (- 1 9))
'(if true (+ 1 9) (- 1 9))
'true
'(+ 1 9)
1
9
'(- 1 9)
1
9
```

SIMPLIFY CODE INTO ADMINISTRATIVE NORMAL FORM (ANF) BY BREAKING UP SIMPLE INSTRUCTIONS INTO MANY LET STATEMENTS TO GIVE EXTRA "ADMINISTRATIVE" BINDINGS

```
anf:
'(let ((x1254 true))
   (if x1254
     (let ((x1255 1)) (let ((x1256 9)) (let ((x1257 (+ x1255 x1256))) x1257)))
     (let ((x1258 1))
       (let ((x1259 9)) (let ((x1260 (- x1258 x1259))) x1260)))))
```

TRANSLATE THE ANF CODE INTO A PSEUDO ASSEMBLY LANGUAGE (IRV TO PREP FOR TRANSFORMATION FROM IRV TO x86

```
ir-virtual:
'(((label lab1261) (mov-lit x1254 true))
  ((label lab1262) (mov-lit zero1271 0))
  (cmp x1254 zero1271)
  (jz lab1263)
  (jmp lab1267)
  ((label lab1263) (mov-lit x1255 1))
  ((label lab1264) (mov-lit x1256 9))
  ((label lab1265) (mov-reg x1257 x1255))
  (add x1257 x1256)
  (return x1257)
  ((label lab1267) (mov-lit x1258 1))
  ((label lab1268) (mov-lit x1259 9))
  ((label lab1269) (mov-reg x1260 x1258))
  (sub x1260 x1259)
  (return x1260))
```

FINAL TRANSFORMATION DONE FROM IRV TO x86 ASSEMBLY LANGUAGE THROUGH STACK ALLOCATION OF ALL VIRTUAL REGISTERS FROM THE IRV CODE

```
x86:
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, 176
        mov esi, true
        mov [rbp-40], esi
        mov esi, 0
        mov [rbp-24], esi
        mov edi, [rbp-24]
        mov eax, [rbp-40]
        cmp eax, edi
        mov [rbp-40], eax
        jz lab1263
        jmp lab1267
lab1263:        mov esi, 1
        mov [rbp-32], esi
        mov esi, 9
        mov [rbp-16], esi
        mov esi, [rbp-32]
        mov [rbp-88], esi
        mov edi, [rbp-16]
        mov eax, [rbp-88]
        add eax, edi
        mov [rbp-88], eax
        mov rax, [rbp-88]
        jmp finish_up
lab1267:        mov esi, 1
        mov [rbp-80], esi
        mov esi, 9
        mov [rbp-72], esi
        mov esi, [rbp-80]
        mov [rbp-64], esi
        mov edi, [rbp-72]
        mov eax, [rbp-64]
        sub eax, edi
        mov [rbp-64], eax
        mov rax, [rbp-64]
        jmp finish_up
finish_up:      add rsp, 176
        leave
        ret
```

2:

```
Input source tree in IfArith:
'(let* ((x 2) (y 8) (z 5)) (* z (+ x y)))
ifarith-tiny:
'(let ((x 2)) (let ((y 8)) (let ((z 5)) (* z (+ x y)))))
'(let ((x 2)) (let ((y 8)) (let ((z 5)) (* z (+ x y)))))
2
'(let ((y 8)) (let ((z 5)) (* z (+ x y))))
8
'(let ((z 5)) (* z (+ x y)))
5
'(* z (+ x y))
'z
'(+ x y)
'x
'y
anf:
```

```
'(let ((x1254 2))
   (let ((x x1254))
     (let ((x1255 8))
       (let ((y x1255))
         (let ((x1256 5))
           (let ((z x1256))
             (let ((x1257 (+ x y))) (let ((x1258 (* z x1257))) x1258))))))))
ir-virtual:
'(((label lab1259) (mov-lit x1254 2))
  ((label lab1260) (mov-reg x x1254))
  ((label lab1261) (mov-lit x1255 8))
  ((label lab1262) (mov-reg y x1255))
  ((label lab1263) (mov-lit x1256 5))
  ((label lab1264) (mov-reg z x1256))
  ((label lab1265) (mov-reg x1257 x))
  (add x1257 y)
  ((label lab1266) (mov-reg x1258 z))
  (imul x1258 x1257)
  (return x1258))
x86:
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, 128
        mov esi, 2
        mov [rbp-24], esi
        mov esi, [rbp-24]
        mov [rbp-32], esi
        mov esi, 8
        mov [rbp-16], esi
        mov esi, [rbp-16]
        mov [rbp-40], esi
        mov esi, 5
        mov [rbp-8], esi
        mov esi, [rbp-8]
        mov [rbp-48], esi
        mov esi, [rbp-32]
        mov [rbp-64], esi
        mov edi, [rbp-40]
        mov eax, [rbp-64]
        add eax, edi
        mov [rbp-64], eax
        mov esi, [rbp-48]
        mov [rbp-56], esi
        mov edi, [rbp-64]
        mov eax, [rbp-56]
        imul eax, edi
        mov [rbp-56], eax
        mov rax, [rbp-56]
        jmp finish_up
finish_up:      add rsp, 128
        leave
        ret
```

3:

```
Input source tree in IfArith:
'(if 0 1 (let* ((x 10)) (+ x 10)))
ifarith-tiny:
'(if 0 1 (let ((x 10)) (+ x 10)))
'(if 0 1 (let ((x 10)) (+ x 10)))
0
1
'(let ((x 10)) (+ x 10))
10
'(+ x 10)
'x
10
anf:
'(let ((x1254 0))
   (if x1254
     (let ((x1255 1)) x1255)
     (let ((x1256 10))
       (let ((x x1256))
         (let ((x1257 10)) (let ((x1258 (+ x x1257))) x1258))))))
ir-virtual:
'(((label lab1259) (mov-lit x1254 0))
  ((label lab1260) (mov-lit zero1268 0))
  (cmp x1254 zero1268)
  (jz lab1261)
  (jmp lab1263)
  ((label lab1261) (mov-lit x1255 1))
  (return x1255)
  ((label lab1263) (mov-lit x1256 10))
  ((label lab1264) (mov-reg x x1256))
  ((label lab1265) (mov-lit x1257 10))
  ((label lab1266) (mov-reg x1258 x))
  (add x1258 x1257)
  (return x1258))
x86:
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, 144
        mov esi, 0
        mov [rbp-40], esi
        mov esi, 0
        mov [rbp-8], esi
        mov edi, [rbp-8]
        mov eax, [rbp-40]
        cmp eax, edi
        mov [rbp-40], eax
        jz lab1261
        jmp lab1263
Lab1261: mov esi, 1
        mov [rbp-32], esi
        mov rax, [rbp-32]
        jmp finish_up
Lab1263: mov esi, 10
        mov [rbp-24], esi
        mov esi, [rbp-24]
        mov [rbp-16], esi
        mov esi, 10
        mov [rbp-72], esi
        mov esi, [rbp-16]
        mov [rbp-64], esi
        mov edi, [rbp-72]
        mov eax, [rbp-64]
        add eax, edi
        mov [rbp-64], eax
        mov rax, [rbp-64]
        jmp finish_up
Finish_up: add rsp, 144
        leave
        ret
```

The .ifa files are:

```
(if true (+ 1 9) (- 1 9))
(let* ([x 2] [y 8] [z 5]) (* z (+ x y)))
(if 0 1 (let* ([x 10]) (+ x 10)))
```

[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.

Stage 1: IfArith (High-Level Language)

This stage processes IfArith, a high-level language with constructs like if, and, or, and arithmetic operations. An IfArith expression might look like (if (> x 0) x (- x)), which checks if x is positive, returning x if true, or its negative if false. Essential for defining the basic functional requirements and semantics of the program.

Stage 2: IfArithTiny (Subsurface Level)

Simplifies IfArith by desugaring complex constructs into simpler forms, such as converting let* into a series of let statements. The expression (let* ([x 5] [y (+ x 1)]) y) might be simplified to nested let expressions. This stage is crucial for reducing complexity before more detailed translation steps, ensuring that later stages can operate on a normalized form.

Stage 3: ANF (Administrative Normal Form)

Transforms the code to ensure that each operation has a very simple structure, facilitating easier translation to assembly-like forms. An expression like (+ 1 (* 2 3)) would be transformed into (let ([v (* 2 3)]) (+ 1 v)). This pass is vital for breaking down complex expressions into simpler steps that are closer to machine operations.

Stage 4: IR-Virtual (Virtual Assembly)

Represents the program as a series of instructions using virtual registers, simplifying register management and operation sequencing. Arithmetic operations are directly translated to operate on virtual registers, like (add r1 r2). While it adds a layer of abstraction, it's fundamental for a clean transition to actual machine instructions, allowing for more straightforward optimization and register allocation.

Stage 5: x86 (Concrete Assembly)

Converts IR-Virtual into actual x86 assembly instructions, dealing with concrete register allocation and instruction selection. Virtual register operations are mapped to physical x86 registers, and instructions like add are directly translated to ADD EAX, EBX. This is the final necessary translation step to produce executable code, integrating all previous simplifications and optimizations into a format that can be executed by the machine.

Evaluation of Redundancy and Necessity for More Passes

- **Redundancy**: Each pass in your compiler serves a distinct purpose, focusing on gradually lowering the level of abstraction. There does not appear to be redundancy, as each stage simplifies and translates the code in necessary steps to bridge high-level constructs and machine code.
- **Potential for More Passes**: While the current passes cover the essentials from high-level to assembly, potential improvements could include:
  - **Optimization Pass**: After the IR-Virtual stage and before the final translation to x86, incorporating an optimization pass could enhance performance. This could focus on peephole optimizations, constant folding, or more advanced techniques like loop unrolling.
  - **Error Handling and Debug Information**: Adding passes for better error reporting, debug information embedding, or static analysis could improve developer experience and code reliability.

[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.

1. Functional Programming

The entire structure of this compiler is built in Racket, a dialect of Scheme, which inherently supports functional programming. The use of functions, higher-order functions, and closures is prevalent throughout the codebase. Each stage of the compiler uses functions to transform code from one representation to another, showcasing the power of functional programming in handling complex transformations in a clear and modular way.

2. Pattern Matching and Quasiquoting

This compiler heavily utilizes Racket’s pattern matching and quasiquoting features. These are used to destructure and manipulate syntax trees easily. For example, transformation functions like ifarith->ifarith-tiny employ pattern matching to recognize different syntactic forms (like arithmetic operations and control structures) and rewrite them into simpler forms. This approach directly ties into discussions on quasiquoting and pattern matching from coursework, emphasizing their utility in syntactic transformations and compiler design.

3. Structural Recursion over Inductively-Defined Lists

The transformation stages of the compiler often recurse over structures like abstract syntax trees (ASTs). This is an application of structural recursion, where functions are defined to handle base cases and recursive cases, typical of inductively defined data structures like lists or trees. This concept is integral to understanding how compilers traverse and manipulate code represented as data structures.

4. Lambda Calculus and Closure-Creating Interpreters

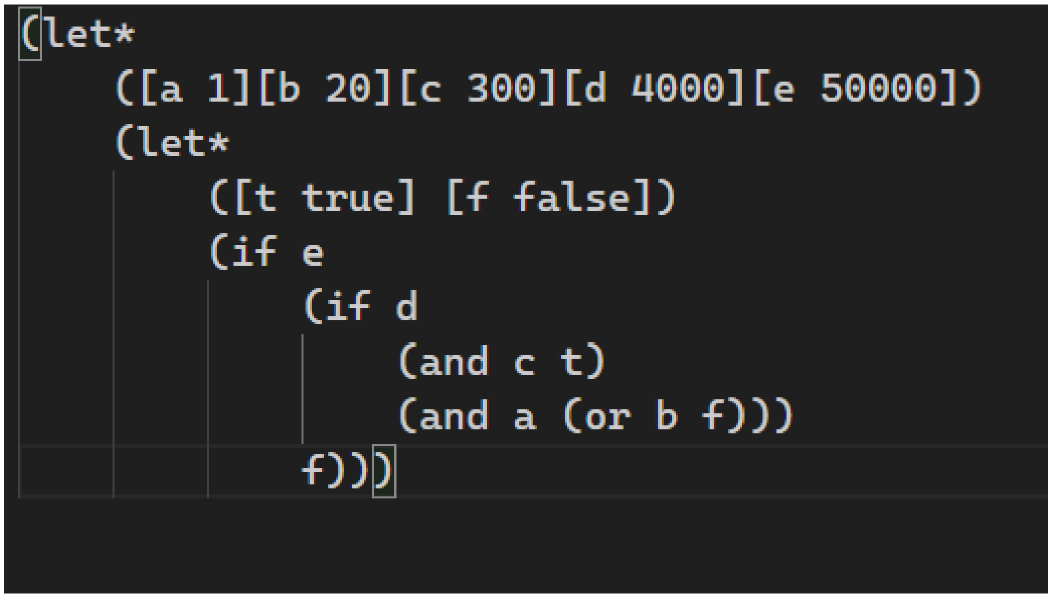
The theoretical foundation of this project includes lambda calculus, as the transformations between stages could conceptually be seen as applying lambda calculus reductions. The handling of environments and closures, though more implicit, resonates with the discussions on closure-creating interpreters, crucial for supporting functions in programming languages.

[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).

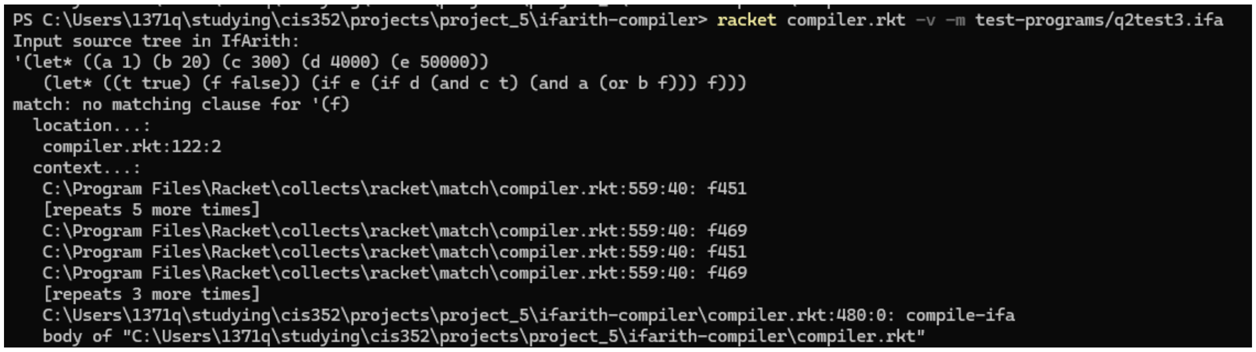
No functionality for (/ a b) in ifArith

No functionality for let in ifArith. Only in ifArithTiny

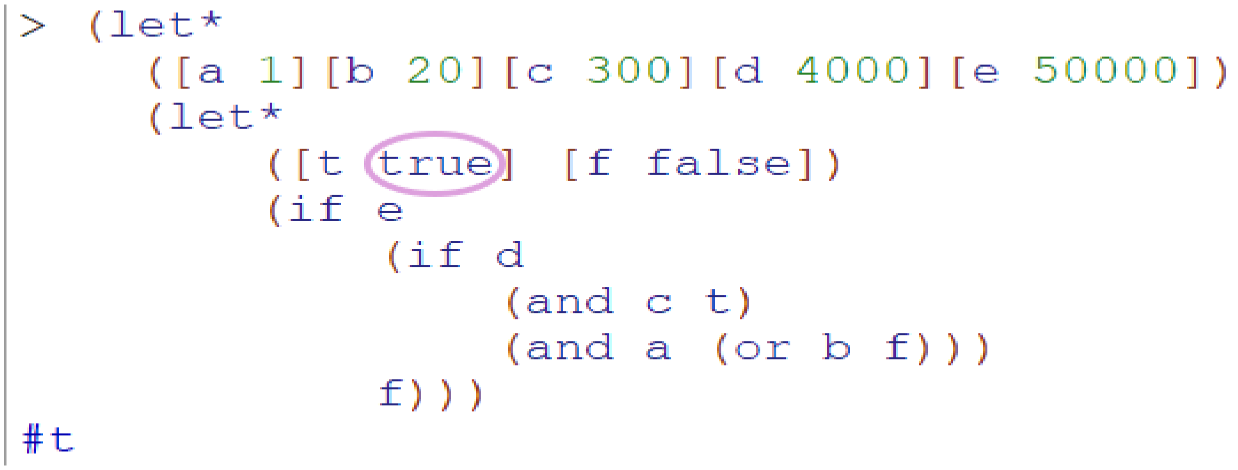
No functionality for this



```
(let*
    ([a 1][b 20][c 300][d 4000][e 50000])
    (let*
        ([t true] [f false])
        (if e
            (if d
                (and c t)
                (and a (or b f)))
            f)))
```



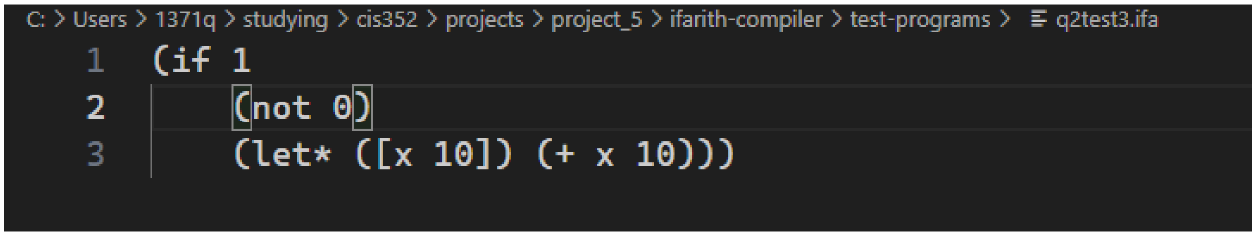
```
PS C:\Users\1371q\studying\cis352\projects\project_5\ifarith-compiler> racket compiler.rkt -v -m test-programs/q2test3.ifa
Input source tree in IfArith:
'(let* ((a 1) (b 20) (c 300) (d 4000) (e 50000))
   (let* ((t true) (f false)) (if e (if d (and c t) (and a (or b f))) f)))
match: no matching clause for '(f)
  location...:
   compiler.rkt:122:2
  context...:
   C:\Program Files\Racket\collects\racket\match\compiler.rkt:559:40: f451
   [repeats 5 more times]
   C:\Program Files\Racket\collects\racket\match\compiler.rkt:559:40: f469
   C:\Program Files\Racket\collects\racket\match\compiler.rkt:559:40: f451
   C:\Program Files\Racket\collects\racket\match\compiler.rkt:559:40: f469
   [repeats 3 more times]
   C:\Users\1371q\studying\cis352\projects\project_5\ifarith-compiler\compiler.rkt:480:0: compile-ifa
   body of "C:\Users\1371q\studying\cis352\projects\project_5\ifarith-compiler\compiler.rkt"
```



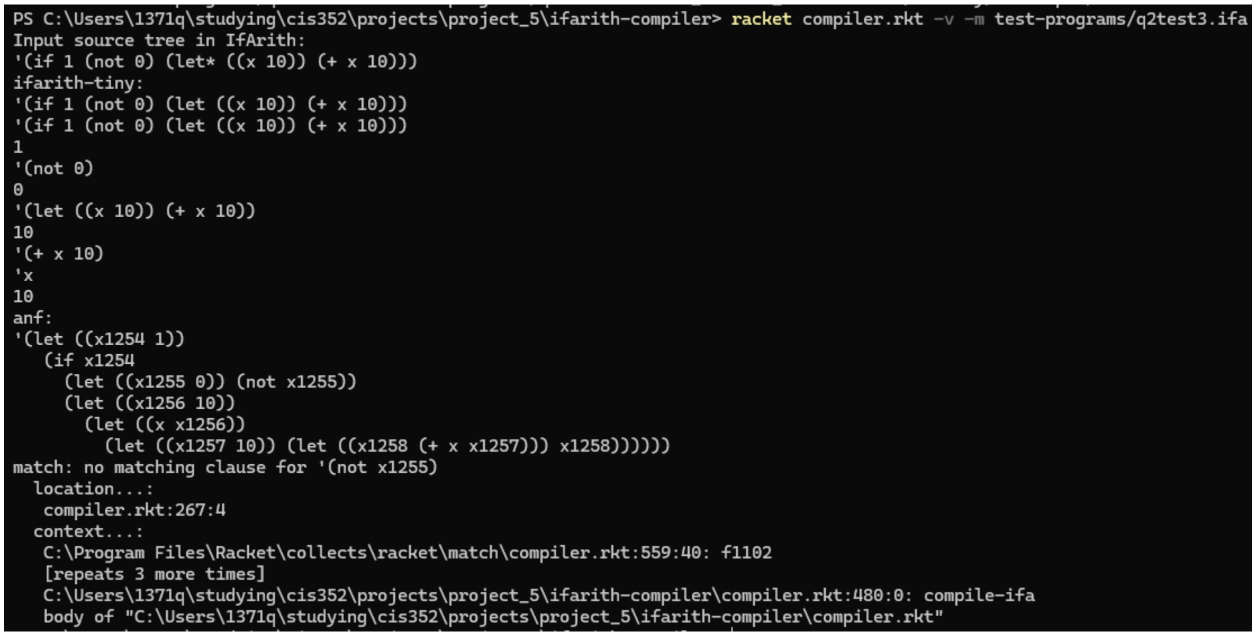
```
> (let*
     ([a 1][b 20][c 300][d 4000][e 50000])
     (let*
          ([t true] [f false])
          (if e
               (if d
                    (and c t)
                    (and a (or b f)))
               f)))
#t
```

works ok in racket

“Not” doesn’t work



```
C: > Users > 1371q > studying > cis352 > projects > project_5 > ifarith-compiler > test-programs > q2test3.ifa
(if 1
    (not 0)
    (let* ([x 10]) (+ x 10)))
```



```
PS C:\Users\1371q\studying\cis352\projects\project_5\ifarith-compiler> racket compiler.rkt -v -m test-programs/q2test3.ifa
Input source tree in IfArith:
'(if 1 (not 0) (let* ((x 10)) (+ x 10)))
ifarith-tiny:
'(if 1 (not 0) (let ((x 10)) (+ x 10)))
'(if 1 (not 0) (let ((x 10)) (+ x 10)))
1
'(not 0)
0
'(let ((x 10)) (+ x 10))
10
'(+ x 10)
'x
10
anf:
'(let ((x1254 1))
   (if x1254
     (let ((x1255 0)) (not x1255))
     (let ((x1256 10))
       (let ((x x1256))
         (let ((x1257 10)) (let ((x1258 (+ x x1257))) x1258))))))
match: no matching clause for '(not x1255)
  location...:
   compiler.rkt:267:4
  context...:
   C:\Program Files\Racket\collects\racket\match\compiler.rkt:559:40: f1102
   [repeats 3 more times]
   C:\Users\1371q\studying\cis352\projects\project_5\ifarith-compiler\compiler.rkt:480:0: compile-ifa
   body of "C:\Users\1371q\studying\cis352\projects\project_5\ifarith-compiler\compiler.rkt"
```

[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 as a way to think about how you would approach doing group code critique. What would you do differently next time, what did you learn?

Mihir:
This compiler design project was a deep and enlightening exploration into how high-level code is transformed into machine-readable instructions. Developing and testing intermediate representations (IR) such as .irv and .ifa programs underscored the complexities involved in compiler architecture and showcased the significant differences in resource management between IR-Virtual and x86 assembly. This hands-on experience provided profound insights into the intricacies of compiler functionality and the challenges of efficient compilation. I enjoyed the debugging the most. The experience of being a bug hunter enhanced my analytical skills and taught me to desire a meticulous, adversarial approach to software testing. The activity also immersed me in understanding the compiler’s mechanisms; it highlighted the need for extensive

software testing.I also noted the need for precise communication during this initiative. The project involved several groups and a team. It emphasized the necessity for clearly defined roles and active participation throughout the activity. I have taken my commitment to work in any team. I learned to be adaptive and equally vocal. I also realized that facing software challenges requires extensive perseverance. The project was excellent for understanding that it is essential to break down complex problems into small and easier tasks for a more justifiable analysis. This is a crucial skill that transcends coding and applies to professional problem-solving.

Artem:
Working on this project has been a great experience for me in multiple ways.
Firstly, it was my first experience with a compiler-like app. Since most of the High-level Programming Languages are compiled (and/or interpreted), working on project like this one helped me understand what that process looks like. It allowed me to not dive deeper into how the code of high-level programming languages is prepared to be executed, but also to directly experience what the intermediate stages and the final assembly code can look like.
Secondly, this project, being a group project, allowed me to experience working in teams. I am considering becoming a software engineer. And most of the time the case is that multiple programmers would be working on one project (i.e. one application), it is important for me to learn how to efficiently divide the project into sub-tasks and distribute them among the team members, which is exactly what the project 5 made us do.
Thirdly, one of the questions involved writing test cases to see whether the compiler returns any kind of error upon running. I believe it made me improve my debugging skills, which are extremely important in coding, because they will allow me to understand where bugs come from and even spot them before even running the code.

Mike:
This project was an effective exercise for building up team communication skills in regards to the debriefing of code. Having to sit down and talk about what each section of code did in order to work together to create a full fledged idea of what is happening in the code was very beneficial for us as coders. Seeing the alterations of the code through the processes of putting code through filters which would continually translate the code to other languages was really interesting to see. It's a program very similar but of much larger scale to the church compiler project we did and it was really cool to see a far more complicated and applicable version of such a compiler. I'd say that if I was to do anything different it would perhaps be playing around with the code more and seeing what additions I could make would work. It was pretty clear we only needed to add code in one section (let* application) but it's definitely rewarding to make other various educated changes to the code to see what happens in order to get deeper insight of what's going on in the code.