Compiler 21 Review

Sample Programs

Test Program 1: Factorial

This test program calculates the factorial of the input number. Here is the program:

```
(let ((x input) (acc 1))
    (loop
        (if (= x 0)
            (break acc)
            (block
                (set! acc (* acc x))
                (set! x (sub1 x))
            )
        )
    )
)
```

This test program features two different binary operators (`+` and `*`), as well as an `input` expression and a loop that runs many times, depending on user input.

Code Snippet 1

```
        Expr::Loop(e) => {
            let startloop = new_label(l, "loop");
            let endloop = new_label(l, "loopend");
            v.push(Instr::LABEL(startloop.to_string()));
            compile_expr(&e, v, env, si, &endloop, l);
            v.push(Instr::JMP(startloop.to_string()));
            v.push(Instr::LABEL(endloop.to_string()));
        }
        Expr::Break(e) => {
            if brake.is_empty() {
                panic!("break")
            }
            compile_expr(&e, v, env, si, brake, l);
            v.push(Instr::JMP(brake.to_string()));
        }
```

This code snippet relates to my first test program because it handles loop-based control flow.

Code Snippet 2

```
enum Instr {
...
    LABEL(String),
...
}
```

I have elided most of the `Instr` enum to highlight what I find interesting about this snippet. This snippet is relevant to my test program because it pertains to how the author represents labels internally, which are necessary to implement all of the control flow present in my test program.

Code Snippet 3

```
                Op1::Sub1 => {
                    v.push(Instr::TEST(Val::Reg(Reg::RAX), Val::Imm(1))); // need
this if assignment operators contains both
                    v.push(Instr::JNZ("throw_error_i".to_string()));
                    v.push(Instr::ISub(Val::Reg(Reg::RAX), Val::Imm(2)));
                    v.push(Instr::JO());
                }
```

This code snippet is (obviously) the implementation of `sub1`, which is relevant because I use `sub1` in my test program. I think it is interesting because it shows the implementation of the runtime numerical type check that this author used. It is different from mine in that it uses the `test` instruction with `0b1`. Conversely, in my compiler, I chose to use the `bt` instruction to directly move the tag bit into the carry register and jump based on the carry register's value. In this case, I believe that neither implementation is superior -- just different ways to accomplish the same thing.

Test Program 2

```
(let ((x 0))
    (loop
        (block
            (if (isbool x)
                (break 1)
                (set! x (+ 1 x))
            )
            (if (< x 0)
                (break 2)
                (set! x (* 1 x))
            )
        )
    )
)
```

This test is slightly different, in that if it is running correctly, it should not exit (realistically, unless left to run for a very long time). The point of this code is to attempt to fuss out edge cases / unintended behavior with the author's tagging implementation by repeatedly adding to and trivially multiplying a numberic value, each iteration checking to ensure that it hasn't somehow become a boolean. For this compiler, I have not observed it exiting, implying that the implementation has no easily-reachable defects or edge cases.

Code Snippet 1

```
                    Op2::Times => {
                        v.push(Instr::SAR(Val::Reg(Reg::RAX)));
                        v.push(Instr::IMul(
                            Val::Reg(Reg::RAX),
```

```
                    Val::RegOffset(Reg::RSP, si * 8),
                ));
                v.push(Instr::JO());
            }
```

This is the author's implementation of the `*` binary operator. It is relevant to my second test program because my second test program repeatedly multiplies a value by 1 using the binary operator `*`. More generally, it is related to my test program because it showcases the simple implementation of `*` (this is a good thing) which contains no bugs or edge cases which are obvious to me.

Code Snippet 2

```
        Op1::IsBool => {
            let end_label = new_label(l, "ifend");
            let else_label = new_label(l, "ifelse");
            v.push(Instr::XOR(Val::Reg(Reg::RAX), Val::Imm(1)));
            v.push(Instr::TEST(Val::Reg(Reg::RAX), Val::Imm(1)));
            v.push(Instr::JNZ(else_label.to_string()));
            v.push(Instr::IMov(Val::Reg(Reg::RAX), Val::Imm(3)));
            v.push(Instr::JMP(end_label.to_string()));
            v.push(Instr::LABEL(else_label.to_string()));
            v.push(Instr::IMov(Val::Reg(Reg::RAX), Val::Imm(1)));
            v.push(Instr::LABEL(end_label.to_string()));
        }
```

This is the author's implementation of the `isbool` unary operator. It is relevant to my second test program because my second test program repeatedly calls this operator to detect if an incorrect type change has occurred. This snippet is interesting to be because the author has implemented the functionality using jumps and a conditional, rather than a simple conditional move into `RAX` (as is used elsewhere in compiler 21).

Code Snippet 3

```
            Op2::LessEqual => {
                let end_label = new_label(l, "ifend");
                let else_label = new_label(l, "ifelse");
                v.push(Instr::CMP(
                    Val::Reg(Reg::RAX),
                    Val::RegOffset(Reg::RSP, si * 8),
                ));
                v.push(Instr::JLE(else_label.to_string()));
                v.push(Instr::IMov(Val::Reg(Reg::RAX), Val::Imm(1)));
                v.push(Instr::JMP(end_label.to_string()));
                v.push(Instr::LABEL(else_label.to_string()));
                v.push(Instr::IMov(Val::Reg(Reg::RAX), Val::Imm(3)));
                v.push(Instr::LABEL(end_label.to_string()));
            }
```

This is the author's implementation of the `<=` binary operator. It is relevant to my second test program because my second test program uses it to check whether or not any incorrect integer overflow /

wraparound has occurred, which would not be correct behavior.

Bugs, Missing Features, Design Decisions

I have been unable to find any examples of programs which do not execute correctly when compiled using compiler_21, aside from one small parsing bug. This compiler does not check for number overflow in compiled constants properly. It accepts the number 4611686018427387904 > 4611686018427387903, which is out of bounds for an i63, as a valid program. This should be simple to fix, however -- some additional logic in `compile_expr` under the `Expr::number` match case should be enough to fix this bug. Besides this, I believe that compiler_21 correctly implements Cobra. I feel confident about this because it passes my own tests, in addition to all of the autograder tests and the (extremely!) comprehensive test suite implemented by compiler_57.

Lessons and Advice

Answer the following questions:

```
Identify a decision made in this compiler that's different from yours. Describe one
way in which it's a better design decision than you made.
Identify a decision made in this compiler that's different from yours. Describe one
way in which it's a worse design decision than you made.
What's one improvement you'll make to your compiler based on seeing this one?
What's one improvement you recommend this author makes to their compiler based on
reviewing it?
```

1. One thing that I liked about this compiler was that the author directly included the most recent loop exit label in the compilation context (see sample program 1, snippet 1). Comparatively, my implementation passes a "label index" corresponding to the current loop's exit label, rather than directly passing down the label. Overall, I think this author's approach is more flexible, and therefore better.

2. One thing that I like better about my compiler is the internal representation of assembly labels (see: sample program 1, code snippet 2). The author of this compiler has chosen to represent labels as a type of instruction, which I find interesting -- in my compiler, I chose to wrap the `Instr` enum inside of another enum:

   ```
   enum AssemblyLine {
       Instructure(Instr),
       Label(String)
   }
   ```

   In this case, I believe that my implementation is overall better. Although it leads to a slightly messier codebase, it is more explicit and could possibly prevent certain bugs through the Rust type checker.

3. One improvement which I will make to my compiler after seeing this one is my implementation of the `*` binary operator. My implementation does not shift one operand prior to multiplying, which leads to error in the result by a factor of two. It is a simple mistake, but I did not realize I was making it until I read this compiler's implementation of `*`.

4. One improvement that I would recommend to this author after reading their compiler is to move more of their code into functions. Essentially all of the compilation logic is present inside of one giant `match...` within `compile_expr`, and this makes it a bit hard to read and follow the specific control flow. I believe that refactoring much of this code into specific functions would make it both more readable and more maintainable / extensible.

Compiler 34 Review

Sample Programs

Test Program 1: Factorial

This test program calculates the factorial of the input number. Here is the program:

```
(let ((x input) (acc 1))
    (loop
        (if (= x 0)
            (break acc)
            (block
                (set! acc (* acc x))
                (set! x (sub1 x))
            )
        )
    )
)
```

This test program features two different binary operators (`+` and `*`), as well as an `input` expression and a loop that runs many times, depending on user input.

Code Snippet 1

```
fn compile_if(condition_expr: &Box<Expr>, then_expr: &Box<Expr>,
else_expr:&Box<Expr>, si: i32, env: &mut HashMap<String, i32>, current_break:
&String, label_count: &mut i32) -> Vec<Instr> {
    let mut vec: Vec<Instr> = vec![];
    let else_label = new_label(label_count, "ifelse");
    let end_label = new_label(label_count, "ifend");
    let cond_instrs = compile_to_instrs(condition_expr, si, env, current_break,
label_count);
    let then_instrs = compile_to_instrs(then_expr, si, env, current_break,
label_count);
    let else_instrs = compile_to_instrs(else_expr, si, env, current_break,
label_count);
    append_instr(&mut vec, cond_instrs);
    vec.push(Instr::Cmp(Val::Reg(Reg::RAX), Val::Imm(1)));
    vec.push(Instr::JE(Val::Label(else_label.clone())));
    append_instr(&mut vec, then_instrs);
    vec.push(Instr::JMP(Val::Label(end_label.clone())));
    vec.push(Instr::Label(Val::Label(else_label)));
    append_instr(&mut vec, else_instrs);
    vec.push(Instr::Label(Val::Label(end_label)));
    vec
}
```

This snippet is the implementation of the compilation of `if` -expressions. This is relevant to my test program because my program uses an `if` -expression to check if we have finished computing the factorial

we are looking for.

Code Snippet 2

```
fn compile_set(name: &String, expr: &Box<Expr>, si: i32, env: &mut HashMap<String,
i32>, current_break: &String, label_count: &mut i32) -> Vec<Instr> {
    if !env.contains_key(name) {
        panic!("Unbound variable identifier {}", name)
    }
    let mut vec: Vec<Instr> = vec![];
    append_instr(&mut vec, compile_to_instrs(expr, si, env, current_break,
label_count));
    vec.push(Instr::IMov(Val::RegOffset(Reg::RSP, *env.get(name).unwrap() * 8),
Val::Reg(Reg::RAX)));
    vec
}
```

This snippet is the implementation of the compilation of `set!` -expressions. This is relevant to my test program because by program uses two `set!` expressions to update the value of x and the accumulator each iteration.

Code Snippet 3

```
        Op1::Sub1 => {
            append_instr(&mut vec, compile_to_instrs(e, si, env, current_break,
label_count));
            check_is_number(&mut vec);
            vec.push(Instr::ISub(Val::Reg(Reg::RAX), Val::Imm(2)));
            check_overflow(&mut vec);
        }
```

This snippet is the implementatino of the compilation of the `sub1` unary operator. This is relevant to my test program because my program uses the `sub1` unary operator to decrement `x` on each iteration.

Test Program 2

```
(let ((x input) (curr 0) (last 1))
    (loop
        (if (= x 0)
            (break curr)
            (let ((temp curr))
                (block
                    (set! curr (+ last curr))
                    (set! last temp)
                    (set! x (sub1 x))
                )
            )
        )
    )
```

This test is different from my second test program for compiler_21. This test program computes the `n` th zero-indexed fibonacci number. It features a long-running loop, two different binary operators (one of which is not present in Test Program 1), and an `input` statement.

Code Snippet 1

```
        Expr::Id(id) => {
            if id == "input" {
                vec.push(Instr::IMov(Val::Reg(Reg::RAX), Val::Reg(Reg::RDI)));
                return vec
            }
            if !env.contains_key(id) {
                panic!("Unbound variable identifier {id}")
            }
            vec.push(Instr::IMov(Val::Reg(Reg::RAX), Val::RegOffset(Reg::RSP,
*env.get(id).unwrap() * 8)));
            vec
        }
```

This code snippet is relevant to my test program for two reasons -- it handles resolving bound variables, and it also handles the `input` statement, which is simply parsed to an identity by this compiler.

Code Snippet 2

```
        Op2::Plus => {
            append_instr(&mut vec, e1_instr);
            check_is_number(&mut vec);
            vec.push(Instr::IMov(Val::RegOffset(Reg::RSP, si * 8),
Val::Reg(Reg::RAX)));
            append_instr(&mut vec, e2_instr);
            check_is_number(&mut vec);
            vec.push(Instr::IAdd(Val::Reg(Reg::RAX), Val::RegOffset(Reg::RSP, si *
8)));
            check_overflow(&mut vec);
        }
```

This code snippet is relevant to my test program because it shows the implementation of the `+` binary operator, which my program (obviously) uses as part of the computation to find the `n` th fibonacci number.

Code Snippet 3

```
fn complie_let(bindings: &Vec<(String, Expr)>, body: &Box<Expr>, si: i32, env: &mut
HashMap<String, i32>, current_break: &String, label_count: &mut i32) -> Vec<Instr> {
    let mut vec: Vec<Instr> = vec![];
    let mut set: HashSet<String> = HashSet::new();
    let mut current_env = env.clone();
    let length = bindings.len() as i32;
    for binding in bindings {
        if set.contains(&binding.0) {
            panic!("Duplicate binding")
```

```
        }
        if binding.0.eq("let") || binding.0.eq("add1") || binding.0.eq("sub1")
        ||
        binding.0.eq("break") || binding.0.eq("set!") || binding.0.eq("loop")
        ||
        binding.0.eq("if") || binding.0.eq("block") || binding.0.eq("input") {
            panic!("Invalid variable name, can't use the {} keyword", binding.0)
        }
        set.insert(binding.0.clone());
    }
    for (i, binding) in bindings.iter().enumerate() {
        let mut nenv = current_env.clone();
        append_instr(&mut vec, compile_to_instrs(&binding.1, si + i as i32, &mut
nenv, current_break, label_count));
        current_env.insert(binding.0.clone(), si + i as i32);
        vec.push(Instr::IMov(Val::RegOffset(Reg::RSP, (si + i as i32) * 8),
Val::Reg(Reg::RAX)));
    }
    append_instr(&mut vec, compile_to_instrs(body, si + length, &mut current_env,
current_break, label_count));
    vec
}
```

This code snippet shows the author's implementation of the code which compiles `let` expressions, which my program uses both to hold its overall state, and to save a temporary variable inside of the loop body.

Bugs, Missing Features, Design Decisions

This compiler (compiler_34) appears to correctly implement the Cobra specification. I have determined this through my own testing of several edge cases which came up during the debugging process for my compiler, and also because this compiler passes both the public Gradescope test suite and the comprehensive test suite laid out by compiler_57.

Lessons and Advice

Answer the following questions:

```
Identify a decision made in this compiler that's different from yours. Describe one
way in which it's a better design decision than you made.
Identify a decision made in this compiler that's different from yours. Describe one
way in which it's a worse design decision than you made.
What's one improvement you'll make to your compiler based on seeing this one?
What's one improvement you recommend this author makes to their compiler based on
reviewing it?
```

1. One decision made in this compiler which is different from mine is to pass the vector of `Instr` down as a reference to `compile_to_instrs`, and the associated family of functions. Conversely, my compiler has each invocation of `compile_to_instrs` return its own constructed `Vec<_>`. I believe it is better to pass the `Vec` as a reference, since this is more efficient. It also leads to more readable code -- when this compiler needs to insert a number check, all they have to do is call `check_is_number(&mut instrs)`, whereas in my compiler I need something like:

```
    let check = gen_num_check(Reg::RAX);
    instrs.extend(check)
```

This is much less declarative, and so I prefer the direction taken by the author of compiler_34.

2. Another decision made in this compiler which is different from mine is to parse the `input` expression as a string-based identity. Conversely, my compiler represents `input` as a variant under the `Instr` enum. I believe the decision to represent it as an `Id` is a poor design decision because it weakens the usefulness of the Rust type system in verifying program correctness -- since the compiler cannot actually check to make sure that we're exhaustively handling all matched expressions. Additionally, it makes the code less extensible and less readable (as it introduces a special case).
3. One improvement that I will make to my compiler after reading this one is changing the difference I outlined in (1.) to match the design of this compiler. I was very impressed with how readable the inline "assembly" (i.e., instantiation of the enum `Instr`) was in this compiler, and I think this would be a very positive change to my compiler.
4. One improvement that I would recommend to the author of this compiler is to change the difference I outlined in (2.) to use an enum variant to represent `Input`, rather than `Instr::Id(s)`. I believe this change will help prevent bugs in the future -- it is reasonable to expect that we will add more no-argument operators (like `input`) as we continue in the course. If they are all implemented as different special cases of `Id`, this will quickly become messy and bug-prone.