Review of compiler_09

Tracing the Compiler

Code1

```
(let ((x 0) (s 0))
    (loop
       (if (>= x input)
          (break s)
          (block
             (set! x (add1 x))
             (set! s (+ s (* x x)))
          )
       )
    )
)
```

Snippet1

```
let my_u64 = input.parse::<u64>();
match my_u64 {
Ok(n) => {
let is_zero_64 = (n & (1 << 63));
let is_zero_63 = (n & (1 << 62));
if is_zero_64 != is_zero_63 {
eprintln!("invalid argument");
std::process::exit(1);
}
res = n << 1
},
Err(_) => res = 1,
}
```

I have encountered some bugs in the program's input parser, which I will discuss in detail later.

Snippet2

```
Expr::UnOp(Op1::Add1, subexpr) => {
v.append(&mut compile_to_instrs(subexpr, si, env, l,
break_target));
v.push(Instr::IAdd(Val::Reg(Reg::RAX), Val::Imm(2)));
v.push(Instr::IJo());
v
},
```

It's related to the add1 operator, where there is no type checking, which may cause a bug. For example, the result of `(isbool (add1 (isbool false)))` is true, which should have been an error.

Snippet3

```
v.push(Instr::IXor(Val::Reg(Reg::RBX), Val::RegOffset(Reg::RSP,
stack_offset)));
v.push(Instr::ITest(Val::Reg(Reg::RBX), Val::Imm(1)));
let error_label = format!("throw_error");
v.push(Instr::IJne(error_label));
```

It's related to how it detects error .

Code2

```
(let
    (
        (i 1)
        (prevPrevNum 0)
        (prevNum 0)
        (currNum 1)
    )
    (if (= input 0)
        0
        (loop
            (if (< i input)
                (block
                    (set! prevPrevNum prevNum)
                    (set! prevNum currNum)
```

```
                (set! currNum (+ prevPrevNum prevNum))
                (set! i (add1 i))
            )
            (break currNum)
        )
    )
)
```

Snippet1

```
Expr::If(cond, thn, els) => {
let end_label = new_label(l, "ifend");
let else_label = new_label(l, "ifelse");
v.append(&mut compile_to_instrs(cond, si, env, l, break_target));
v.push(Instr::ICmp(Val::Reg(Reg::RAX), Val::Imm(1)));
v.push(Instr::IJe(else_label.clone()));
v.append(&mut compile_to_instrs(thn, si, env, l, break_target));
v.push(Instr::IJmp(end_label.clone()));
v.push(Instr::ILabel(else_label.clone()));
v.append(&mut compile_to_instrs(els, si, env, l, break_target));
v.push(Instr::ILabel(end_label.clone()));
v
},
```

This is how it compiles if.

Snippet2

```
let asm_program = format!(
"
section .text
extern snek_error
global our_code_starts_here
throw_error:
mov rdi, 7
```

```
push rsp
call snek_error
overflow:
mov rdi, 5
push rsp
call snek_error
our_code_starts_here:
{}
ret
",
result
);
```

This is how it handles error. It’s fixed instead of flexible.

Snippet3

```
fn print_value(i:u64) {
if i % 2 == 0 {
let is_negative = (i & (1 << 63)) != 0;
if is_negative {
let mut result = !i;
result += 1;
println!("-{}", result/2);
} else {
println!("{}", i/2);
}
} else if i == 3 {
println!("true");
} else if i == 1 {
println!("false");
} else {
println!("Unknown: {}", i);
```

```
eprintln!("invalid argument");
std::process::exit(1);
}
}
```

This is how it print the result manually, which can be replaced by api.

Bugs, Missing Features, and Design Decisions

There are bugs in fn parse_input(). If the input is -1, it will be saved as false. That's because the input is parsed as u64 which can't represent negative number, and the algorithm to detect invalid argument is wrong. What's more, if input is and invalid argument like "abc", the input will be saved as false, where I think an error should be thrown.

The code should look like this:

```
match input {
   "true" => 3,
   "false" => 1,
   "" => 1,
   _ => {
   let t = i64::from_str_radix(&input, 10);
   match t {
   Ok(t) => {
   if t <= 4611686018427387903 && t >= -4611686018427387904 {(t
<< 1) as u64}
   else {panic!("invalid argument")}
   },
   Err(_) => panic!("invalid argument"),
   }
   },
}
```

Lessons and Advice

1. 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.

He compile operator equal individually, so that the code used to detect invalid argument error for other operators can be reused. I didn’t do that so my code is much longer.

2. 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.

To print the return value of our_code_start_here, he used if/else and manually printed the result, which is kind of complicated. We can accomplish that function easily with i.tostring() and match.

3. What's one improvement you'll make to your compiler based on seeing this one?

Because the code to detect error for equal is different from others, I will compile equal operation individually and reuse the code to detect invalid argument error for other operators.

4. What's one improvement you recommend this author makes to their compiler based on reviewing it?

Simplify print_value with i.tostring() and match.

Review of compiler_57

Tracing the Compiler

Code1

```
(let ((x 0) (s 0))
    (loop
        (if (>= x input)
            (break s)
            (block
                (set! x (add1 x))
                (set! s (+ s (* x x)))
            )
        )
    )
)
```

Snippet1

```
match op_type {
Op2::Arith(op) => {
// For arithmetic operators, we only want (number, number).
instructions.push(Instr::Compare(Val::Reg(Reg::R11),
Val::Imm(0)));
instructions.push(Instr::JumpEqual(type_check_success.to_owned
()));
call_error_fn(TYPE_MISMATCH, instructions);
instructions.push(Instr::Label(type_check_success));
```

It defines Op2::Arith and Op2::Comp, which is a good idea to reuse type check code when performing op2.

Snippet2

```
pub(crate) fn new_label(s: &str, label_counter: &mut u64) -> String
{
let curr = *label_counter;
```

```
*label_counter += 1;
format!("{s}_{curr}")
}
```

```
Expr::Loop(expr) => {
let loop_start = new_label("loop_start", counter);
let loop_done = format!("loop_break_done#{loop_start}");

instructions.push(Instr::Label(loop_start.to_owned()));
compile_helper(
expr,
si,
instructions,
environment,
counter,
Some(&loop_done),
)?;
instructions.push(Instr::Jump(loop_start));
instructions.push(Instr::Label(loop_done));
}
```

This is how loop and label is compiled.

Snippet3

```
pub extern "C" fn snek_error(errcode: i64) {
match errcode {
OVERFLOW_ERROR => eprintln!("[Runtime Error] overflow error."),
TYPE_MISMATCH => eprintln!("[Runtime Error] type mismatch error
(invalid argument)."),
_ => eprintln!("[Runtime Error] Unknown error code: {errcode}"),
};
```

```
std::process::exit(1);
}
```

It’s related to throwing an error, which is more flexible than compiler_09.

Code2

```
(let
    (
        (i 1)
        (prevPrevNum 0)
        (prevNum 0)
        (currNum 1)
    )
    (if (= input 0)
        0
        (loop
            (if (< i input)
                (block
                    (set! prevPrevNum prevNum)
                    (set! prevNum currNum)
                    (set! currNum (+ prevPrevNum prevNum))
                    (set! i (add1 i))
                )
                (break currNum)
            )
        )
    )
)
```

Snippet1

```
fn parse_input(input: &str) -> u64 {
if input == "true" {
0b11
} else if input == "false" {
0b01
} else if let Ok(val) = input.parse::<i64>() {
if val > (i64::MAX >> 1) {
panic!("[Input Error] Invalid input, {} overflow/overflow", val)
} else if val < (i64::MIN >> 1) {
```

```
panic!("[Input Error] Invalid input, {} overflow/underflow", val)
} else {
(val << 1) as u64
}
} else {
panic!("[Input Error] Unsupported input: `{}`", input);
}
}

fn main() {
let args: Vec<String> = env::args().collect();
let input = if args.len() == 2 { &args[1] } else { "false" };
let input = parse_input(&input);
```

It uses a very good way to parse input.

Snippet2

```
Sexp::List(list) => match list.as_slice() {
// (block <expr>+)
[Sexp::Atom(S(op)), expressions @ ..] if op == "block" => {
let mut parsed_statements = vec![];
for statement in expressions {
parsed_statements.push(parse_expr(statement)?);
}

if parsed_statements.is_empty() {
Err(ParseError::BlockEmpty)
} else {
Ok(Expr::Block(parsed_statements))
}
}
```

This is how it compiles block exp efficiently and robustly.

Snippet3

```
"section .text
extern snek_error
global our_code_starts_here

throw_error:
mov rdi, rax
push rsp
call snek_error
pop rsp
our_code_starts_here:
{}
ret
",
```

It's related to throwing error, which provide potential to handle error and come back. That's a good design.

Bugs, Missing Features, and Design Decisions

I think the compiler perfectly implements Cobra, because I use test cases to test not only every module, but also the output of program, which includes loop, break, set! and all other keywords of Cobra.

Lessons and Advice

1.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.

He wrote different modules in different files, and test every module, while I only wrote the test cases in lots of files, run them manually and test the output, without testing each module.

2.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.

He uses jump to return the result of compare, which can cause branching and slow down the performance. I use cmov instruction so that there is no branching.

3.What's one improvement you'll make to your compiler based on seeing this one?

Write test cases to test each module in my code.

4.What's one improvement you recommend this author makes to their compiler based on reviewing it?

Use cmov instruction to return the result of compare, so that you can avoid branching and improve performance.