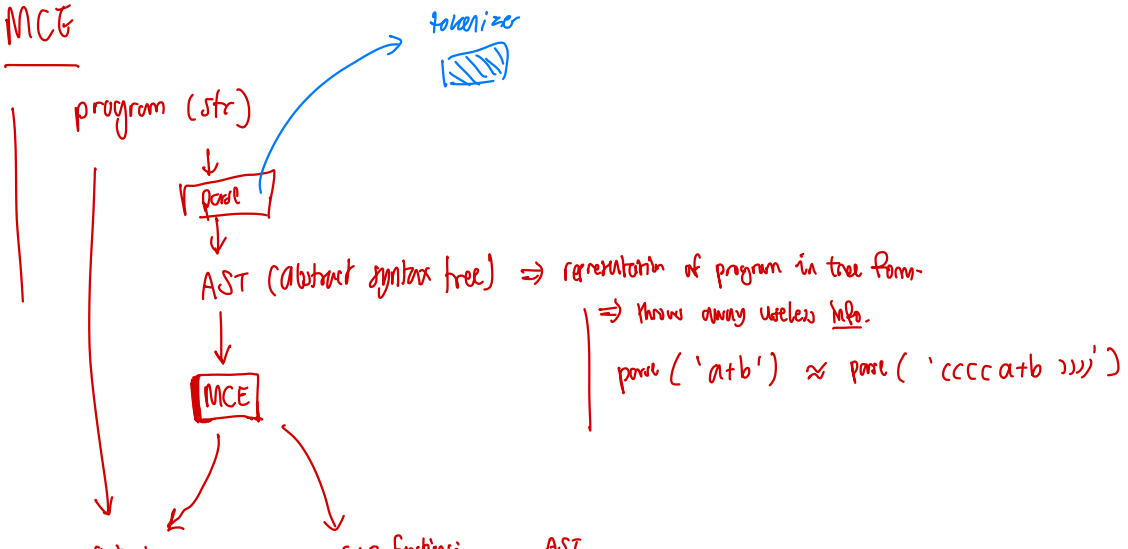
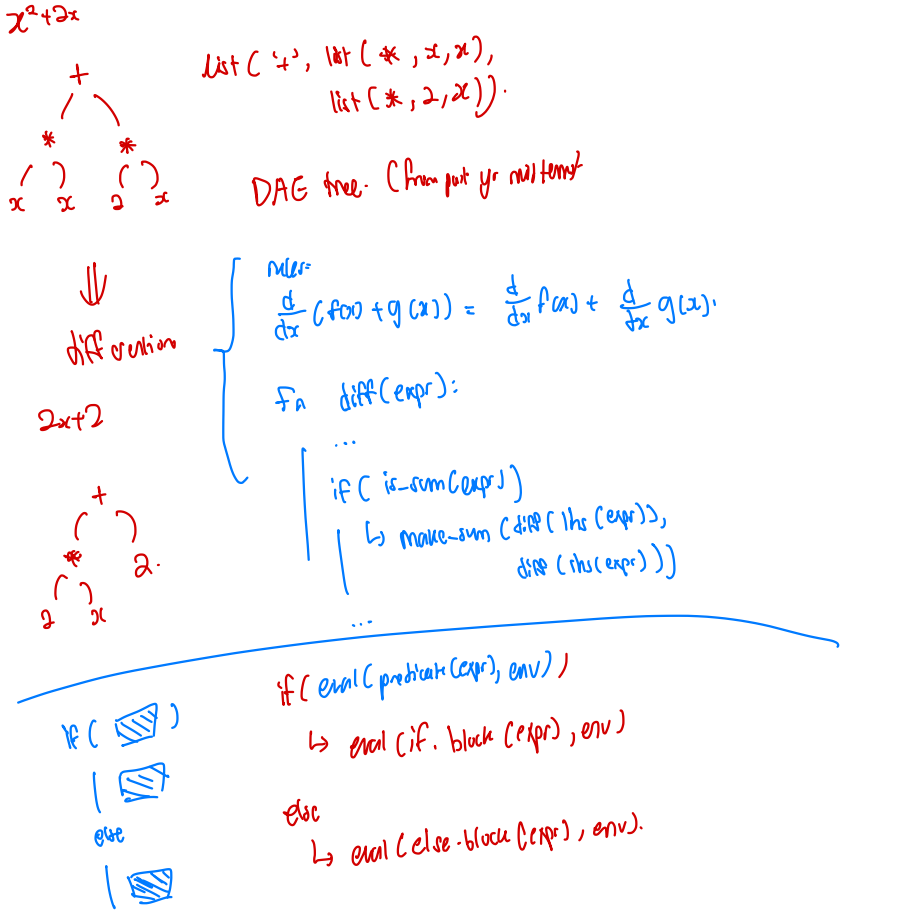
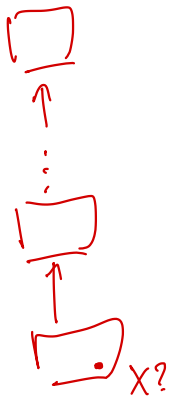
MCE


tokenizer
program (str)
parse
AST (abstract syntax tree) ⇒ representation of program in tree form.
⇒ throws away useless info.
parse('a+b') ≈ parse('((((a+b))))')
MCE
Output.
Core functions:
AST
fn evaluate(comp, env)
wish that:
I can evaluate ANY subprogram.
and produce a result.
checks for type of comp & dispatches.
to correct handler.


x^2+2x
+
* *
x x 2 x
list('+', list(*, x, x),
list(*, 2, x)).
DAG tree. (from part yr midterm)
differentiation
rules:
$\frac{d}{dx}(f(x)+g(x)) = \frac{d}{dx}f(x) + \frac{d}{dx}g(x)$.
fn diff(expr):
...
if(is_sum(expr))
↳ make_sum(diff(lhs(expr)),
diff(rhs(expr)))
...
2x+2
+
* 2.
2 x
if( )
else
if(eval(predicate(expr), env))
↳ eval(if.block(expr), env)
else
↳ eval(else.block(expr), env).



fn lookup(name, env)

if env is null / undef
↳ error.

if x is in env:
↳ value of x in env.

↳ lookup(name, parent(env)).

---

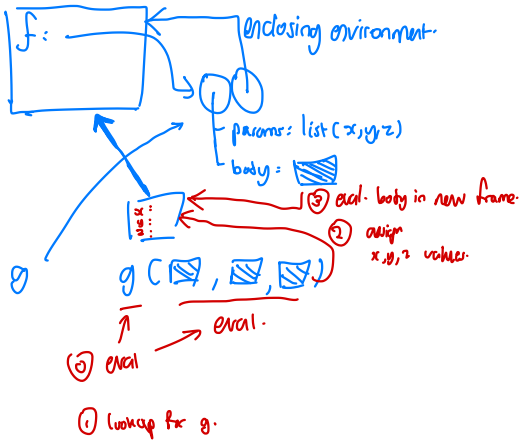
fn f(params...)

Const f = (params) ⇒ 

① Const declaration.

② λ expr.

Const f = (x,y,z) ⇒ 



National University of Singapore
School of Computing
CS1101S: Programming Methodology
Semester I, 2021/2022

R11
Metacircular Evaluator

Getting Started

For your convenience, the final evaluator given in class is available here.

It may be useful to look at the parser output when working on some of the following problems. You can enter `display_list(parse("<statement>"));` to do so.

The evaluator linked above includes the following convenient function:

```
function parse_and_evaluate(program) {
    return evaluate(make_block(parse(program)),
                    the_global_environment);
}
```

When applied to a given string `input`, it performs the following steps:

- parse `input`
- wrap program in a block called `implicit_top_level_block`
- evaluate `implicit_top_level_block` using the global environment.

Note that besides single and double quotes, JavaScript (and Source) provide a backtick that can stretch over multiple lines. This is convenient when the program strings that we give to the parser become longer. Example:

```
parse_and_evaluate(`
function factorial(n) {
    n === 1
    ? 1
    : n * factorial(n - 1);
}
factorial(5);`);
```

Note that the interpreted language covered in class does not have `return` statements. (We will cover these on Friday.)

Problems:

1. The target languages of the evaluators given in the lecture do not provide an exponentiation operator. Modify the evaluator program for the calculator language such that the operator `%` is now the exponentiation operator, i.e. `b % e` evaluates to the value b^e.

Does `b % e % n` evaluate to b^{e^n}? Why? If not, what needs to be changed in the evaluator?

2. The evaluator currently does not support the "lazy" logical composition operations; the following program leads to an error `Unknown component` in the metacircular evaluator, but evaluates to false in Source.

   ```
   false && 0();
   ```

   Modify the evaluator such that lazy logical composition operations are supported as described in the lectures.

   Depending on your approach, you might find the following functions useful, also available here.

   ```
   // the syntax predicate
   function is_logical_composition(component) {
       return is_tagged_list(component, "logical_composition");
   }
   // selectors
   function logical_symbol(comp) {
       return list_ref(comp, 1);
   }
   function logical_composition_first_component(comp) {
       return list_ref(comp, 2);
   }
   function logical_composition_second_component(comp) {
       return list_ref(comp, 3);
   }
   // helper to make a conditional expression
   function make_conditional_expression(pred, cons, alt) {
       return list("conditional_expression", pred, cons, alt);
   }
   // helper to make a literal value
   function make_literal(value) {
       return list("literal", value);
   }
   ```

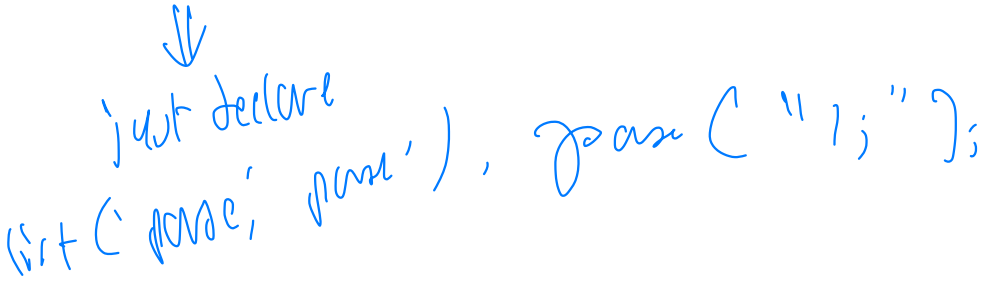
3. The evaluator currently does not support the function `parse`, and therefore we cannot write a meta-metacircular evaluator!

   - Add the function `parse` such that the following program evaluates to a list as expected:

     ```
     parse_and_evaluate(`parse('parse("1;");');`);
     // returns
     // list("application", list("name", "parse"), list(list("literal", "1;")))
     ```

   - In the program above you find four semicolons. Explain the meaning of each of them, in reverse order.
   - (If you have a day to spare and like this sort of thing) Apply the evaluator to itself, i.e. modify/extend your evaluator sufficiently so that it can be truly metacircular: You should be able to evaluate the evaluator as follows:

     ```
     // here goes your evaluator
     parse_and_evaluate(`// here goes your evaluator again
                         parse_and_evaluate("(x => x + 1)(7);");`);
     ```



Report your findings to your tutor: What problems did you encounter? How did you solve them?

```
fn evaluate_logical_composition (comp, env) {
  const symbol = logical_symbol(comp);
  const lhs = first_comp(comp);
  const rhs = second_comp(comp);
  if (symbol === '||') {
      return evaluate(lhs, env) ? true : evaluate(rhs, env);
  } else {
      return evaluate(lhs, env) ? evaluate(rhs, env) : false;
  }
```