2nd Nov 2020

Studio 12
Meta-Circular Evaluator

CS1101S AY20/21 SEM 1
Studio 03A

Chen Xihao
Year 2 Computer Science

chenxihao@u.nus.edu
@BooleanValue

Studio 12

Agenda

- Admin
- Recap: Meta-Circular Evaluator
- Studio Sheet
- In-class Studio Sheet

Studio 12

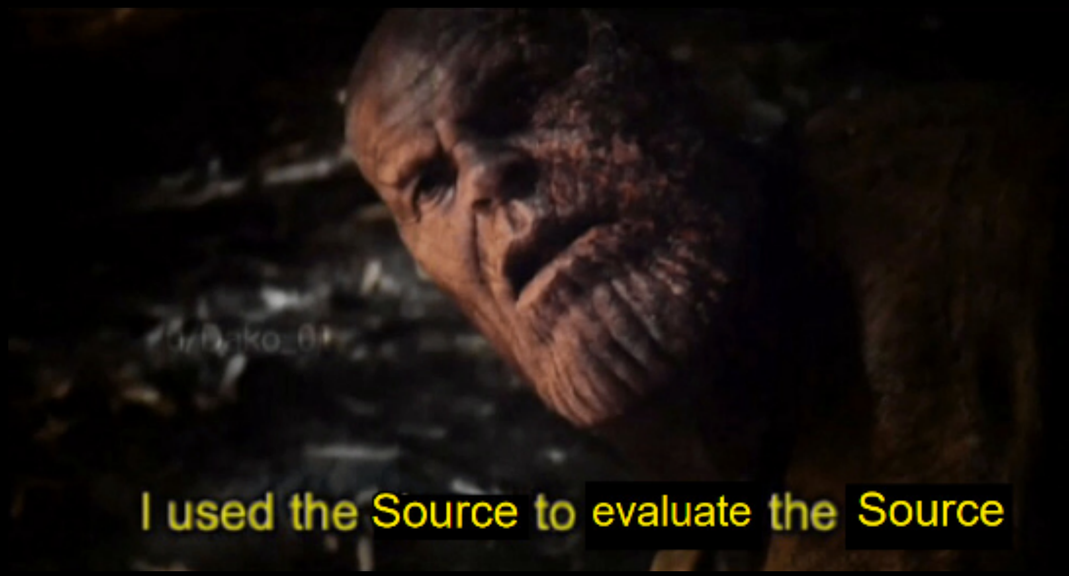
Admin

- Good news: end of CS1101S syllabus! :D
- Sad news: exams are coming
- Practical practice:
  - I’m thinking of some questions for extra practice!

Recap: Meta-Circular Evaluator

Recap
Meta-Circular Evaluator

- What is it?
  - A programme
  - That runs a source programme



Recap

Meta-Circular Evaluator

- Basis of MCE:
  - We are programming the environment model
- Editing the MCE:
  - We are writing our own programming language
- By allowing you to see the MCE, studying it will allow us to better understand Source and how it works!

Recap

Meta-Circular Evaluator

- Key idea:
  - MCE is just another programme!
  - We use this programme to evaluate another Source programme!
- Actually:
  - All programming languages are just programmes that can run some other programmes!
  - If you are “gud enuf”, you can change the MCE to evaluate another language too!

Recap
Meta-Circular Evaluator

- Understanding the MCE:
  - Defining the structure of the language
    - Decisions on the syntax (don’t have to worry about this)
  - Parsing the programme
    - Converts your programme string into an abstract syntax tree
    - Done for you too
  - Defining a method of processing the tree structure
    - Your job!

Recap
Meta-Circular Evaluator

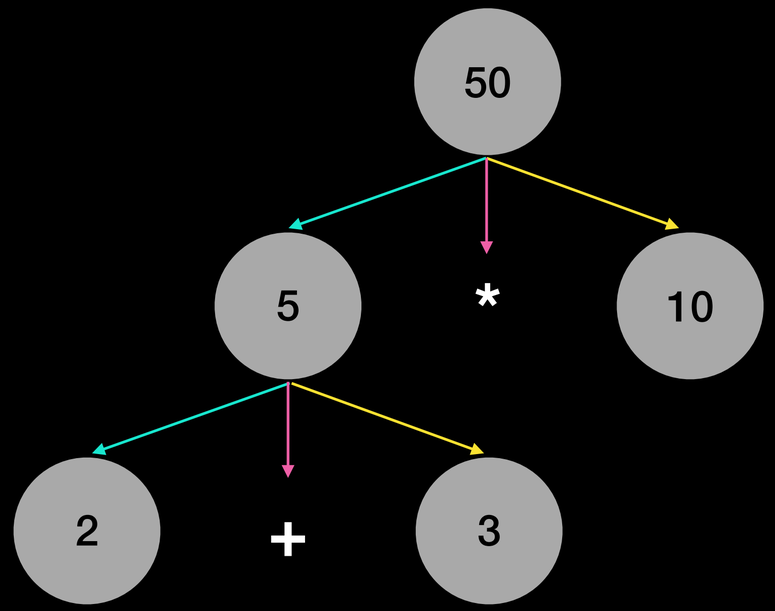
- If you try to parse some string:
  - It’s a mess (?)
  - To understand this better, let’s visit history!

```
parse("const x = y => y + 1;");
```

```
[ "constant_declaration",
[ ["name", ["x", null]],
[ [ "lambda_expression",
  [ [["name", ["y", null]], null],
  [ [ "return_statement",
    [ [ "operator_combination",
      ["+", [["name", ["y", null]], [["literal", [1, null]], null]]]],
    null]],
  null]]],
null]]]
```

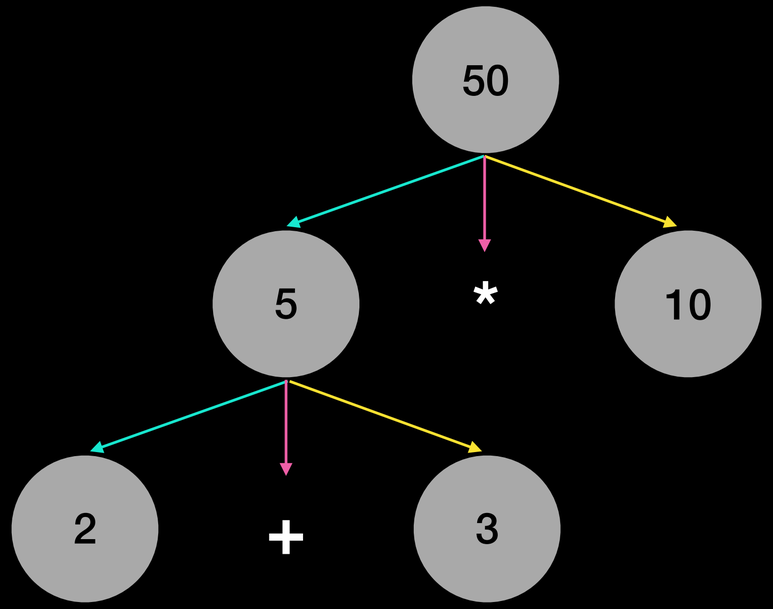
Recap
Meta-Circular Evaluator

- Recall: Abstract Syntax Trees!
  - AST of (2 + 3) * 10



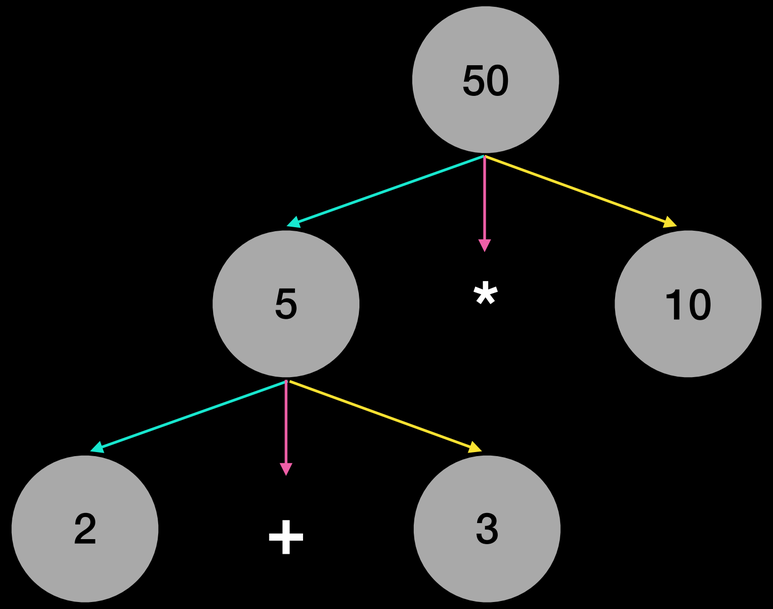
Recap
Meta-Circular Evaluator

- Structure:
  - (<ast> <op> <ast>)
- An AST is either a number, or the expression (<ast> <op> <ast>), where <ast> is an abstract syntax tree and <op> is a binary operator



Recap
Meta-Circular Evaluator

- We can represent this using a tree:
  - `list(list(2, “+”, 3), “*”, 5);`
  - (a binary arithmetic expression or something like it)
  - this is just an example!



Recap
Meta-Circular Evaluator

- But, this can get complicated real soon if the programme is complex
- So we can have some function to do this for us!
  - `parse_ast;`
- Now, we also need some function that evaluates this for us!
  - `eval_ast;`
- (Can’t be bothered to implement this… so you can do this as a challenge!)

Recap

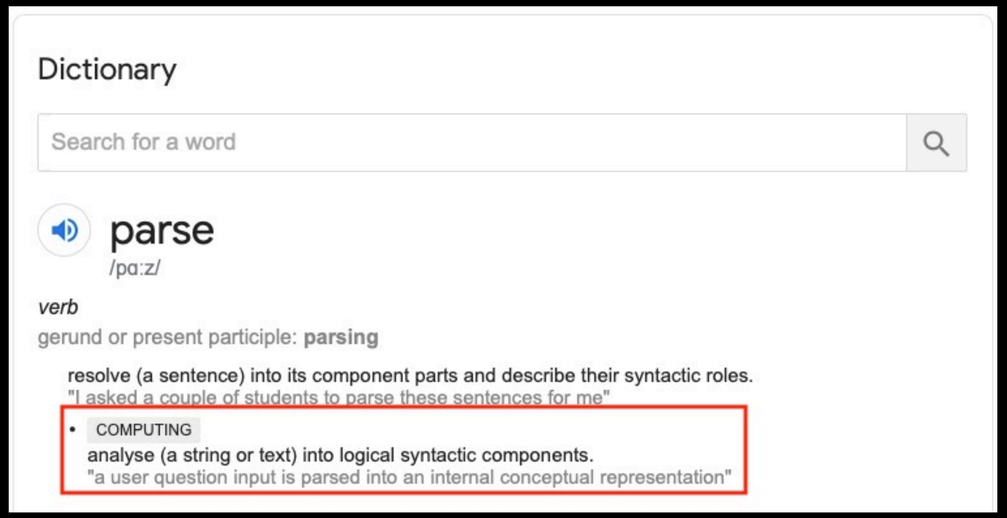
Meta-Circular Evaluator

- Now, we have a programme that can convert a arithmetic programme into an abstract syntax tree
- And then evaluate this abstract syntax tree
- `eval_ast(parse_ast(input));`

Recap
Meta-Circular Evaluator

- Let’s take a step back. What have we done here?
- With these two functions and our very simple tree structure:
  - We have defined the structure of our programme
  - We have a programme that converts a valid input into this structure
  - We have a programme that evaluates this structure to give us the correct output
- We have defined our own language! (if you write out the functions, that is…)

Recap
Meta-Circular Evaluator



- So what is parsing exactly?
  - Analyse into logical syntactic components
  - Divide the programme into parts to follow some logical structure
- In the example,
  - We assume the existence of `parse_ast` to magically convert our programme into an AST
- In the MCE, we have the function `parse`.
  - Use abstraction!

Recap
Meta-Circular Evaluator

- So is our AST programme an MCE?
  - Technically… yes
  - Recall: MCE is a programme that takes in a programme and evaluates it
- But aren’t we missing some things?
  - Primitive functions?
  - Declarations?
  - Data structures?

Recap
Meta-Circular Evaluator

```
parse("const p = list(pair(1, 2), pair(3, 4));");
```

```
[ "constant_declaration",
[ ["name", ["p", null]],
[ [ "application",
  [ ["name", ["list", null]],
  [ [ [ "application",
      [ ["name", ["pair", null]],
      [[["literal", [1, null]], [["literal", [2, null]], null]], null]]],
    [ [ "application",
      [ ["name", ["pair", null]],
      [[["literal", [3, null]], [["literal", [4, null]], null]], null]]],
    null]],
  null]]],
null]]]
```

- In Source, we can have much more complicated programmes and expressions!
- The `parse` function handles everything for you
  - Convert a programme string into an AST
- Our job:
  - Understand the structure of parsing
  - Decide on changes to make to the MCE

Recap
Meta-Circular Evaluator

- Parsing in MCE - Tagged Lists:
  - The `parse` function returns a “tagged list”
  - A tagged list is (unofficially):
    - a pair whose head is a string (the tag),
    - and the tail is a list with relevant information associated to the tag (might be a tagged list too)

Recap
Meta-Circular Evaluator

- Tags: there are many tags
  - Name
  - Constant / Variable declaration
  - Conditional Expression
  - Function Definition
  - Primitive / Compound Function
  - Sequence
  - Application
  - Block
  - …

Recap
Meta-Circular Evaluator

- Tags are used to:
  - Specify what the tail list is
  - How to evaluate this list

Recap
Meta-Circular Evaluator

- E.g.
  - Sequence: sequence of expressions and statements, defines order of execution
  - Application: function applications, two properties (operator and operand(s))
  - Return Statement / Value: terminates function upon return, etc.

Recap
Meta-Circular Evaluator

- Key notes:
  - Abstraction is CRITICAL!

Recap
Meta-Circular Evaluator

- Frames in MCE:
  - Same as usual
  - Created if block has bindings
  - Not created if there are no bindings.
- ^ this might be a tip for the mission

Any questions?

Studio Sheet

Studio Sheet

Urm

- Disclaimer:
  - I’m slightly lost :’)
  - I will try mah best

1. There is a difference in the handling of function declarations between JavaScript on the one hand and the Source Academy and the MCE on the other hand: In JavaScript, the function declarations that appear anywhere in a statement sequence of a block are automatically moved to the beginning of the block. In JavaScript, the following program will produce the value 42:

```
{
    const x = f(8);
    function f(y) {
        return y + 34;
    }
    x;
}
```

because any JavaScript system will move the function declaration to the beginning of the block:

```
{
    function f(y) {
        return y + 34;
    }
    const x = f(8);
    x;
}
```

before the program is evaluated. Verify that this is not the case in the Source Academy or in the MCE. Modify the evaluator of Lecture L11 such that it behaves like JavaScript implementations in this respect.

Studio Sheet
Q1

- Let’s to understand how this works
- If we simply parse this block, this is what we get as the AST
- Notice the first tag is “sequence”

```
[ "sequence",
[ [ [ "constant_declaration",
    [ ["name", ["x", null]],
    [ ["application", [["name", ["f", null]], [[["literal", [8, null]], null], null]]],
    null]]],
  [ [ "function_declaration",
    [ ["name", ["f", null]],
    [ [["name", ["y", null]], null],
    [ [ "return_statement",
      [ [ "operator_combination",
        ["+", [["name", ["y", null]], [["literal", [34, null]], null]]]],
      null]],
    null]]]],
  [["name", ["x", null]], null]]],
null]]
```

Studio Sheet
Q1

- We see that in the evaluate function, it checks if the component is a sequence.
- And subsequently calls the `evaluate_sequence` function by passing in the tail-list of the component (the “relevant info”)

```
function evaluate(component, env) {
   return is_literal(component)
          ? literal_value(component)
          : is_name(component)
          ? lookup_symbol_value(symbol_of_name(component), env)
          : is_application(component)
          ? apply(evaluate(function_expression(component), env),
                  list_of_values(arg_expressions(component), env))
          : is_operator_combination(component)
          ? evaluate(operator_combination_to_application(component), env)
          : is_conditional(component)
          ? eval_conditional(component, env)
          : is_lambda_expression(component)
          ? make_function(lambda_parameter_symbols(component),
                          lambda_body(component), env)
          : is_sequence(component)
          ? eval_sequence(sequence_statements(component), env)
          : is_block(component)
          ? eval_block(component, env)
          : is_return_statement(component)
          ? eval_return_statement(component, env)
          : is_assignment(component)
          ? eval_assignment(component, env)
          : is_function_declaration(component)
          ? evaluate(function_decl_to_constant_decl(component), env)
          : is_declaration(component)
          ? eval_declaration(component, env)
          : error(component, "Unknown syntax -- evaluate");
}
```

Studio Sheet
Q1

- By applicative order reduction, we know that `sequence_stmts(component)` will be evaluated first to give us the “relevant info”
    - Which contains the declarations and the function call to `f(8)`

```
function evaluate(component, env) {
   return is_literal(component)
          ? literal_value(component)
          : is_name(component)
          ? lookup_symbol_value(symbol_of_name(component), env)
          : is_application(component)
          ? apply(evaluate(function_expression(component), env),
                  list_of_values(arg_expressions(component), env))
          : is_operator_combination(component)
          ? evaluate(operator_combination_to_application(component), env)
          : is_conditional(component)
          ? eval_conditional(component, env)
          : is_lambda_expression(component)
          ? make_function(lambda_parameter_symbols(component),
                          lambda_body(component), env)
          : is_sequence(component)
          ? eval_sequence(sequence_statements(component), env)
          : is_block(component)
          ? eval_block(component, env)
          : is_return_statement(component)
          ? eval_return_statement(component, env)
          : is_assignment(component)
          ? eval_assignment(component, env)
          : is_function_declaration(component)
          ? evaluate(function_decl_to_constant_decl(component), env)
          : is_declaration(component)
          ? eval_declaration(component, env)
          : error(component, "Unknown syntax -- evaluate");
}
```

Studio Sheet
Q1

- Can we re-order this sequence of statements before calling `eval_sequence`?

```
function evaluate(component, env) {
   return is_literal(component)
          ? literal_value(component)
          : is_name(component)
          ? lookup_symbol_value(symbol_of_name(component), env)
          : is_application(component)
          ? apply(evaluate(function_expression(component), env),
                  list_of_values(arg_expressions(component), env))
          : is_operator_combination(component)
          ? evaluate(operator_combination_to_application(component), env)
          : is_conditional(component)
          ? eval_conditional(component, env)
          : is_lambda_expression(component)
          ? make_function(lambda_parameter_symbols(component),
                          lambda_body(component), env)
          : is_sequence(component)
          ? eval_sequence(sequence_statements(component), env)
          : is_block(component)
          ? eval_block(component, env)
          : is_return_statement(component)
          ? eval_return_statement(component, env)
          : is_assignment(component)
          ? eval_assignment(component, env)
          : is_function_declaration(component)
          ? evaluate(function_decl_to_constant_decl(component), env)
          : is_declaration(component)
          ? eval_declaration(component, env)
          : error(component, "Unknown syntax -- evaluate");
}
```

Studio Sheet
Q1

- Something like this:
- We added a function to reorder the statements before evaluating the sequence
- Your job now:
  - Write the `reorder_statements` function
  - Discuss!

```
function evaluate(component, env) {
    return is_literal(component)
           ? literal_value(component)
           : is_name(component)
           ? lookup_symbol_value(symbol_of_name(component), env)
           : is_application(component)
           ? apply(evaluate(function_expression(component), env),
                   list_of_values(arg_expressions(component), env))
           : is_operator_combination(component)
           ? evaluate(operator_combination_to_application(component), env)
           : is_conditional(component)
           ? eval_conditional(component, env)
           : is_lambda_expression(component)
           ? make_function(lambda_parameter_symbols(component),
                           lambda_body(component), env)
           : is_sequence(component)
           ? eval_sequence(reorder_statements(
                                sequence_statements(component)), env)
           : is_block(component)
           ? eval_block(component, env)
           : is_return_statement(component)
           ? eval_return_statement(component, env)
           : is_assignment(component)
           ? eval_assignment(component, env)
           : is_function_declaration(component)
           ? evaluate(function_decl_to_constant_decl(component), env)
           : is_declaration(component)
           ? eval_declaration(component, env)
           : error(component, "Unknown syntax -- evaluate");
}
```

2. The evaluator of Lecture L11 does not detect undeclared names. Therefore, the following program runs without error in the MCE:

```
false ? abracadabra(simsalabim) : 42;
```

The Source Academy, on the other hand, gives nice error messages for *any* name that is not declared. Modify the evaluator such that any undeclared name is detected and reported to the user as an error.

Hint: The function `scan_out_declarations` of the given MCE might come in handy.

Studio Sheet In-Class

1. Compare the Source Academy and the MCE with respect to their behaviour when evaluating the following program:

```
const x = y;
const y = 42;
const z = "***" + x + "***";
z;
```

Explain the behaviour of the MCE. Modify the evaluator of Lecture L11 such that it behaves like the Source Academy with respect to the situation exemplified in the program.

End of Recap

End of File