CS1101S Discussion Group Week 12: *Meta-circular Evaluator*

Niu Yunpeng

niuyunpeng@u.nus.edu

November 7, 2017

Overview

1. More about interpreter
   - Revisit language processing
   - How does interpreter work
2. Meta-circular evaluator
   - Basic evaluator
   - Loop & assignment
   - OOP evaluator
   - Lazy evaluator
   - Memoized evaluator

More About Interpreter

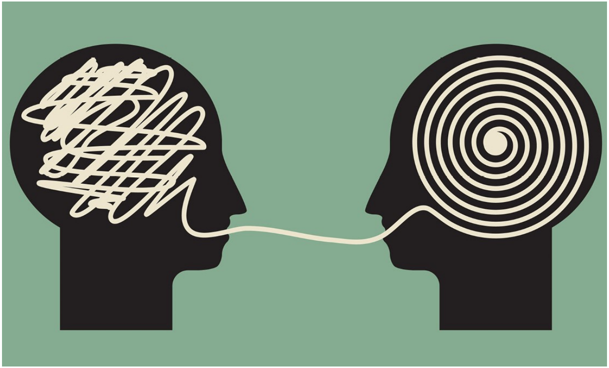
Interpreter

- An interpreter is a program that executes another program.
- *Source language*: the language in which the interpreter is written.
- *Target language*: the language in which the programs are written which the interpreter can execute.

More About Interpreter

Interpreter

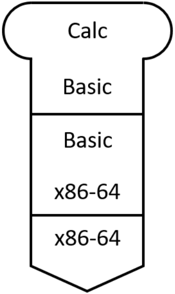
- Usually, an interpreter can execute each statement written in high-level language by converting it to a lower-level language.

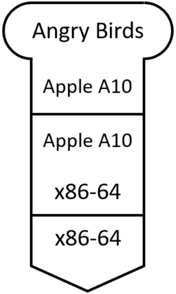


More About Interpreter

T-diagram for interpreter

- Programs written in high-level language can be executed on a CPU using an interpreter.





(Hardware emulation)

More About Interpreter

How to use an interpreter

- Interpreter is also a *program*.
- To use an interpreter is similar to call a function:
  - Supply the function parameters with input;
  - Evaluate the function body;
  - Get the return value as output.

What is the "output"?

- The output is the program being executed.
- It is *just* a string.

More About Interpreter

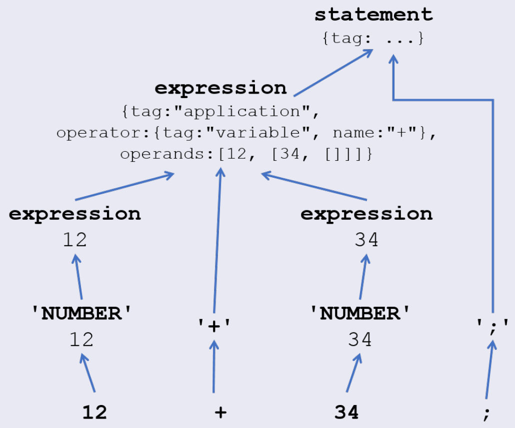
The working process of an interpreter

Only applicable to Abstract Syntax Tree (AST) Interpreter:

- Parse the source code string:
    - Run lexical analysis using regular expression;
    - Build the Abstract Syntax Tree (AST);
    - Run syntactic checking using Backus-Naur Form (BNF).
- Perform the behaviours directly.

More About Interpreter

Abstract Syntax Tree (AST)



Overview

1. More about interpreter
   - Revisit language processing
   - How does interpreter work
2. Meta-circular evaluator
   - Basic evaluator
   - Loop & assignment
   - OOP evaluator
   - Lazy evaluator
   - Memoized evaluator

Meta-circular Evaluator

Meta-circular evaluator

- Meta-circular evaluator is a special kind of interpreter.
- Its source language is the same as its target language.
- However, the source language is usually written in a more basic implementation than the target language.

Meta-circular Evaluator

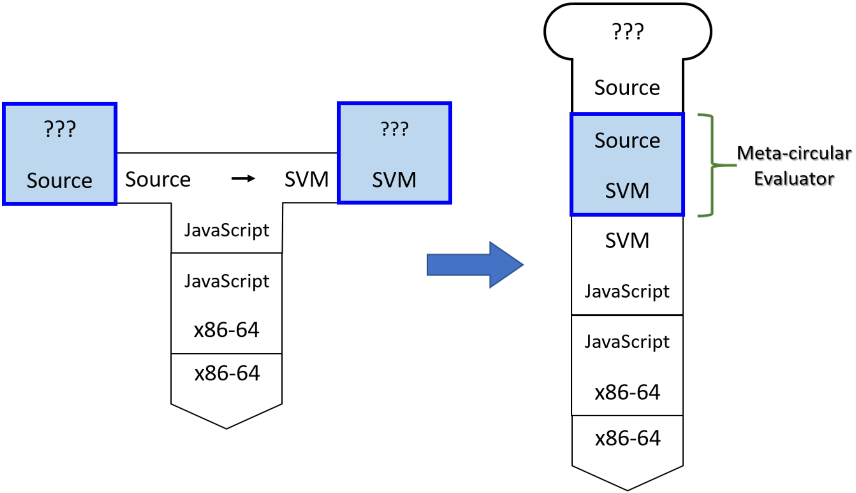
Meta-circular evaluator for the Source language

- Meta-circular evaluator is the kernel for our textbook, *Structure and Interpretation of Computer Programs* (SICP).
- A similar evaluator is also implemented for the Source language.

Fallbacks

- It does not include the parser component.
- It does not support tail recursion.
  - It does support recursion, though.

Meta-circular Evaluator



Revisit - components of programming language

- Primitives:
  The smallest constituent unit of a programming language.
- Combination:
  Ways to put primitives together.
- Abstraction:
  The method to simplify the messy combinations.
  - To abstract data: use naming;
  - To abstract procedures: use functions.
  - Sometimes, naming and functions are combined together.

Meta-circular Evaluator

Primitives in meta-circular evaluator

- Primitives include primitive data and primitive operators.
  - Primitive data: numeral, boolean, string;
  - Primitive operators: $+, -, \times, \div, \%, \ldots$
- Primitives are *self-evaluating*.
  - Primitive data are applied directly;
  - Primitive operators are defined in the global environment.

Meta-circular Evaluator

Primitive data in meta-circular evaluator

```
function is_self_evaluating(stmt) {
    return is_number(stmt) || is_string(stmt) ||
           is_boolean(stmt);
}

function evaluate(stmt) {
    if (is_self_evaluating(stmt)) {
        return stmt;
    } else {
        error("Unknown expression type -- evaluate: " +
              stmt);
    }
}
```

Meta-circular Evaluator

Primitive operators in meta-circular evaluator

```
function is_tagged_object(stmt, the_tag) {
    return is_object(stmt) && stmt.tag === the_tag;
}

function is_primitive_function(func) {
    return is_tagged_object(func, "primitive");
}
function primitive_implementation(func) {
    return func.implementation;
}

function apply_primitive_function(func, argument_list) {
    return apply_in_underlying_javascript(
        primitive_implementation(func), argument_list);
}
```

Meta-circular Evaluator

Primitive operators

```
var primitive_functions = list(
    pair("+", function(x, y) { return x + y; }),
    pair("-", function(x, y) { return x - y; }),
    pair("*", function(x, y) { return x * y; }),
    pair("/", function(x, y) { return x / y; }),
    pair("%", function(x, y) { return x % y; }),
    pair("===", function(x, y) { return x === y; }),
    pair("!==", function(x, y) { return x !== y; }),
    pair("<", function(x, y) { return x < y; }),
    pair(">", function(x, y) { return x > y; }),
    pair("<=", function(x, y) { return x <= y; }),
    pair(">=", function(x, y) { return x >= y; }),
    pair("!", function(x) { return !x; }),
    ...
);
```

Meta-circular Evaluator

Combination & abstraction

- Combination and abstraction are evaluated recursively until all the things left are primitives.
- For naming (variables):
  - Use a list to represent the frames;
  - Use a table to represent the binding of names and their values;
  - Search in the list to find the value of a variable.
- For functions:
  - Append the list to extend the enclosing environment;
  - Evaluate all the actual arguments;
  - Evaluate the function body sequentially.

Meta-circular Evaluator

Representation of environment & frames

```
function make_frame(variables, values) {
    if (is_empty_list(variables) && is_empty_list(values)) {
        return {};
    } else {
        var frame = make_frame(tail(variables), tail(values)
            );
        frame[head(variables)] = head(values);
        return frame;
    }
}

function add_binding_to_frame(variable, value, frame) {
    frame[variable] = value;
    return undefined;
}
```

Meta-circular Evaluator

Environment table lookup

```
function lookup_variable_value(variable, env) {
    function env_loop(env) {
        if (is_empty_environment(env)) {
            error("Unbound variable: " + variable);
        } else if (has_binding_in_frame(variable,
                                        first_frame(env))) {
            return first_frame(env)[variable];
        } else {
            return env_loop(enclosing_environment(env));
        }
    }
    return env_loop(env);
}
function has_binding_in_frame(variable, frame) {
    return has_own_property(frame, variable);
}
```

Meta-circular Evaluator

To extend environment

```
function extend_environment(vars, vals, base_env) {
    var var_length = length(vars);
    var val_length = length(vals);

    if (var_length === val_length) {
        return pair(make_frame(vars, vals), base_env);
    } else if (var_length < val_length) {
        error("Too many arguments supplied: " + vars +
              " " + vals);
    } else {
        error("Too few arguments supplied: " + vars +
              " " + vals);
    }
}
```

Meta-circular Evaluator

Function application

```
function apply(fun, args) {
    if (is_primitive_function(fun)) {
        return apply_primitive_function(fun, args);
    } else {
        error("Unknown function type -- apply: " + fun);
    }
}
function list_of_values(exps, env) {
    if (no_operands(exps)) {
        return [];
    } else {
        return pair(evaluate(first_operand(exps), env),
                    list_of_values(rest_operands(exps), env));
    }
}
```

`return` statement

- Function body may not have a `return` statement.
- `return` statement may appear in the middle of the function body.
  - Everything after should be ignored.
- `return` statement should not appear outside a function body.

Meta-circular Evaluator

return statement

```
...
var parameters = function_value_parameters(fun);

if (length(parameters) === length(args)) {
    var env = extend_environment(parameters, args,
                  function_value_environment(fun));
    var result = evaluate(function_value_body(fun), env);

    if (is_return_value(result)) {
        return return_value_content(result);
    } else {
        return undefined;
    }
}
...
```

Stateful programming

- We have already supported:
  - Variables
  - Functions
- That is almost enough for pure functional programming
- But what about *stateful* programming?
  - `while` & `for` loop
  - Assignment

while loop

```
function evaluate_while_statement(stmt,env) {
   if (is_true(evaluate(while_predicate(stmt),env))) {
       evaluate(while_body(stmt),env);
       evaluate_while_statement(stmt, env);
   } else {
       return undefined;
   }
}
```

Meta-circular Evaluator

for loop

```
function for_loop(predicate, body, finaliser, env) {
   if (is_true(evaluate(predicate,env))) {
       evaluate(body,env);
       evaluate(finaliser, env);
       for_loop(predicate, body, finaliser, env);
   } else {
       return undefined;
   }
}

function evaluate_for_statement(stmt,env) {
    evaluate(for_initialiser(stmt), env);
    for_loop(for_predicate(stmt), for_body(stmt),
             for_finaliser(stmt), env);
    return undefined;
}
```

Meta-circular Evaluator

Assignment

```
function set_variable_value(variable,value,env) {
    function env_loop(env) {
        if (is_empty_environment(env)) {
            error("Unbound variable -- assignment: " +
                variable);
        } else if (has_binding_in_frame(variable,first_frame(
             env))) {
            add_binding_to_frame(variable,value,first_frame(env
                ));
        } else {
            env_loop(enclosing_environment(env));
        }
    }
    env_loop(env);
    return undefined;
}
```

Assignment

```
function evaluate_assignment(stmt,env) {
   var value = evaluate(assignment_value(stmt),env);
   set_variable_value(variable_name(assignment_variable(stmt
      )),
                        value, env);
   return value;
}
```

Meta-circular Evaluator

Object-oriented programming

- Our basic evaluator does not support OOP yet.
- To support OOP in meta-circular evaluator:
  - Object lateral and property accessor
  - The `new` keyword
  - The `prototype` chain
  - Object method invocation

Meta-circular Evaluator

To create an object

```
function evaluate_object_literal(stmt,env) {
    var obj = {};

    for_each(function(p) {
      obj[head(p)] = evaluate(tail(p), env);
    }, pairs(stmt));

    return obj;
}
```

Meta-circular Evaluator

To access/set the property of an object

```
function evaluate_property_access(stmt,env) {
    var obj = evaluate(object(stmt), env);
    var prop = evaluate(property(stmt), env);
    return obj[prop];
}

function evaluate_property_assignment(stmt,env) {
    var obj = evaluate(object(stmt), env);
    var prop = evaluate(property(stmt), env);
    var val = evaluate(value(stmt), env);
    obj[prop] = val;
    return val;
}
```

To invoke the method of an object

```
function evaluate_object_method_application(stmt,env) {
    var obj = evaluate(object(stmt), env);
    var method_name = property(stmt);
    var method = obj[method_name];

    var first_arg = obj;
    var other_args = list_of_values(operands(stmt),
                                    env);

    return apply_compound_function(method,
                                   pair(obj, other_args));
}
```

Meta-circular Evaluator

The new keyword

```
function evaluate_new_construction(stmt,env) {
    var obj = {};
    var constructor = lookup_variable_value(type(stmt),env);

    // link to the prototype table
    obj.__proto__ = constructor.prototype;

    // apply constructor with obj as "this"
    apply_compound_function(constructor,
        pair(obj, list_of_values(operands(stmt), env)));

    // ignore the result value, and return the object
    return obj;
}
```

Meta-circular Evaluator

Laziness

- *General idea*: compute values only when they are needed.
- In the lazy evaluator, actual arguments are only evaluated when they are needed in the function body.

`thunk`

- We wrap each argument into a `thunk` to distinguish them.
- They will be unwrapped when needed in the function body.
- *The same idea as stream.*

Meta-circular Evaluator

When will expressions in `thunk` get evaluated?

- When they become parameters of a primitive function;
- When they become predicate of a conditional statement;
- When the variable referring to it get applied;
- When it is a statement in the global frame.
- ...

Meta-circular Evaluator

Lazy evaluation

```
function list_of_values(exps, env) {
    if (no_operands(exps)) {
        return [];
    } else {
        return pair(make_thunk(first_operand(exps), env),
            list_of_values(rest_operands(exps), env));
    }
}

function force(v) {
    return is_thunk(v) ? v
                       : force(
      evaluate(thunk_expression(v), thunk_environment(v)));
}
```

Memoization

- We can enable automatic memoization in the meta-circular evaluator.
- To achieve this, we can make use of `thunk`.
- Once the thunk has been forced to evaluated once, its value will be changed to the return value of the wrapping expression.
- Thus, the expression inside will always be evaluated ***once***.

Meta-circular Evaluator

Memoized evaluation 1

```
function make_thunk(expr, env) {
    return {
        tag: "thunk",
        expression: expr,
        environment: env,
        has_memoized_value: false,
        memoized_value: undefined
    };
}

function thunk_memoize(thunk, value) {
    thunk.has_memoized_value = true;
    thunk.memoized_value = value;
}
```

Meta-circular Evaluator

Memoized evaluation 2

```
function force(v) {
    if (is_thunk(v)) {
        if (thunk_has_memoized_value(v)) {
            return thunk_memoized_value(v);
        } else {
            var value = evaluate(thunk_expression(v),
                                 thunk_environment(v));
            thunk_memoize(v, value);
        }
    } else {
        return v;
    }
}
```

Meta-circular Evaluator

Memoized evaluation 3

```
function lookup_variable_value(variable, env) {
    function env_loop(env) {
        if (is_empty_environment(env)) {
            error("Unbound variable: " + variable);
        } else if (has_binding_in_frame(variable,
                                          first_frame(env))) {
            var value = force(first_frame(env)[variable]);
            first_frame(env)[variable] = value;
            return value;
        } else {
            return env_loop(enclosing_environment(env));
        }
    }
    return env_loop(env);
}
```

Let's discuss them now.

The End