CS1101S Studio Session Week 12: *Object & Meta-circular Evaluator*

Niu Yunpeng

niuyunpeng@u.nus.edu

November 3, 2018

Overview

1. Object-oriented Programming
   - Literal objects
   - Object-oriented paradigm *(optional)*
2. More about interpreter
   - Revisit language processing
   - How does interpreter work
3. Meta-circular evaluator
   - Basic evaluator
   - Loop & assignment
   - OOP evaluator *(optional)*
   - Lazy evaluator *(optional)*
   - Memoized evaluator *(optional)*

From Last Week

Object

- Object is a collection of key-value pairs;
- Object is a built-in implementation of “table”;
- Key is string, value can be anything (function, data structure)

Object as “table”

- Still remember how we would generalize `memoize` function?

Literal Objects

Object accessor

- Using object is really similar to using array/table.

```
const obj = {"aa": 4,
             bb: true,
             "cc": x => x * x};

obj["aa"];
obj["bb"];
obj["cc"](5); // returns 25
```

Literal Objects

Dot operator

- Dot operator is a shortcut for object accessor.

```
let obj = {"aa": 4,
           bb: true,
           "cc": x => x * x};
obj["dd"] = "someone";

obj.aa;
obj.bb;
obj.cc(5);  // returns 25
```

Literal Objects

Quotation marks

- You must use quotation marks when
  - Access the attribute of an object using square bracket notation.
  - Add new attribute(s) to an existing object declared using `let`.
- Quotation marks are optional when
  - Declare attributes of a new object inside curly braces "{}".
- You cannot use quotation marks when
  - Access the attribute of an object using dot notation.

Object-oriented Programming

Our world...

- Our world is only a collection of objects.
- They have various states and behaviours.
- They belong to their own classes.
- Objects in the same class are similar.
- ...

Object-oriented Programming

Object-oriented Programming

Terminology

- Class
- Object
- Instance
- Field
- Attribute
- Method
- Constructor
- Inheritance
- Polymorphism
- Override
- ...

Object-oriented Programming

Class, object & instance

- Class: a blueprint/template of all the things of one type.
- Object: a particular thing of one type.
- Instance: a unique copy of information for an object in memory.

Relationship

- $\langle$*class_name*$\rangle$ **includes many** $\langle$*object_name*$\rangle$s.
- $\langle$*object_name*$\rangle$ **is a** $\langle$*class_name*$\rangle$.

Object-oriented Programming

Example

- Class: `Country`
- Objects: `Singapore`, `China`, `Russia`,...

Relationship

- `Country` includes `Singapore`, `China` and `Russia`.
- `Singapore` is a `Country`.
- `China` is a `Country`.
- `Russia` is a `Country`.

Object-oriented Programming

To describe an object

- Use adjectives: how large? how long? how old? ...
  or equivalent to:
  Use nouns: size, length, age, ...
- Use verbs: can jump? can swim? can speak?

Thus...

- Use *adjectives/nouns* to describe **states**;
- Use *verbs* to describe **behaviours**.

Object-oriented Programming

Property & method

- Property: variables that describe states of an object;
- Method: functions that operate on an object.

Relationship

- ⟨*class_name*⟩ or ⟨*obj_name*⟩ **has many** properties.
- Fields/attributes/methods **describes** ⟨*class_name*⟩ or ⟨*obj_name*⟩.

Example

- Class: `Student`
- Properties: `name`, `age`, `major`, ...
- Methods: `study`, `play`, ...

Relationship

- `name`, `age` and `major` describes a `Student`.
- A `Student` can `study` and `play`.

Constructor

- Constructor: to create a new instance of a class and perform related initialization actions.
  - Usually, the constructor will set the initial values of compulsory fields.

Relationship

- We **use** the constructor to **instantiate** a copy of ⟨*class_name*⟩ to get a new ⟨*obj_name*⟩.

Common patterns between different classes

- We know there are a lot of common patterns within a class.
- However, different classes may also have common patterns.

Problem...

- How can we share common patterns between different classes?

Object-oriented Programming

Inheritance

- Inheritance: abstract the common patterns into one superclass, and keep the specification within each subclass.

Polymorphism

- Polymorphism: the same method may behave in different ways due to different and potentially heterogeneous implementations.
- Polymorphism in OOP is usually achieved via method override.

Three terms

- Override
- Overwrite
- Overload

Your task today

- Find out the difference between these three terms.

Object-oriented Programming

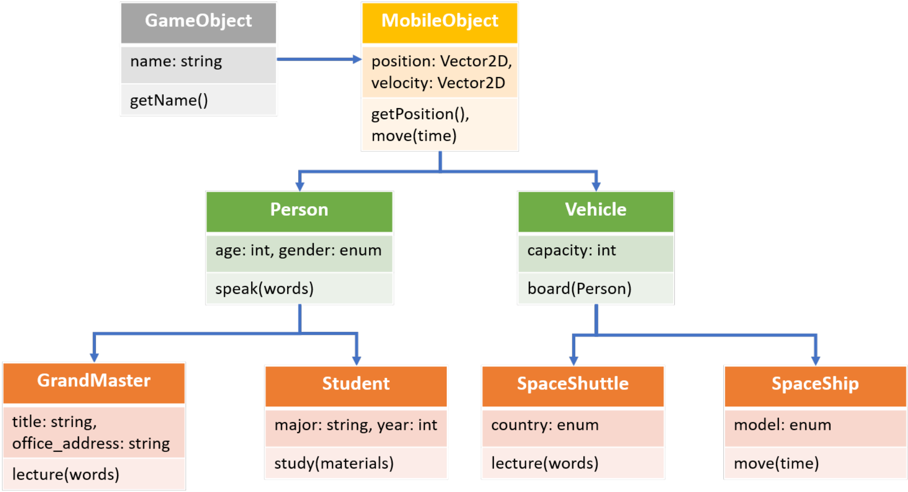
Relationship

- A ⟨*super_class_name*⟩ **has many** ⟨*sub_class_name*⟩s.
- A ⟨*sub_class_name*⟩ **belongs to** a ⟨*super_class_name*⟩.
- A ⟨*sub_class_name*⟩ **inherits from** its ⟨*super_class_name*⟩.

Diagram

- We can draw a diagram to visualize the hierarchy relationship between all the superclasses and subclasses.
- The diagram is going to be a ***tree***.

Object-oriented Programming



Overview

1. Object-oriented Programming
   - Literal objects
   - Object-oriented paradigm *(optional)*
2. More about interpreter
   - Revisit language processing
   - How does interpreter work
3. Meta-circular evaluator
   - Basic evaluator
   - Loop & assignment
   - OOP evaluator *(optional)*
   - Lazy evaluator *(optional)*
   - Memoized evaluator *(optional)*

More About Interpreter

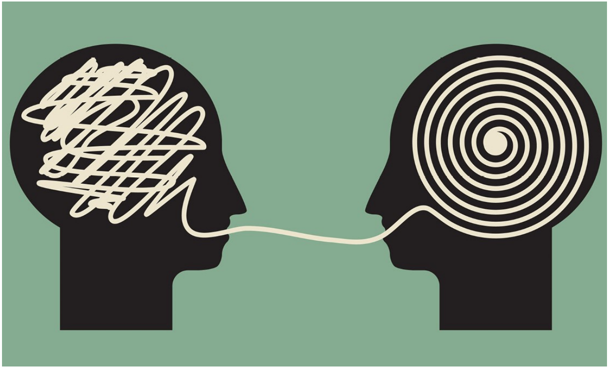
Interpreter

- 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

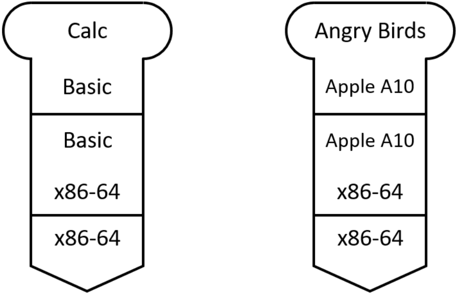
- Usually, an interpreter can execute each statement written in high-level language by converting it to a lower-level language.



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 "intput"?

- The input is the program being executed.
- The input of an interpreter is the output of the parser.

More About Interpreter

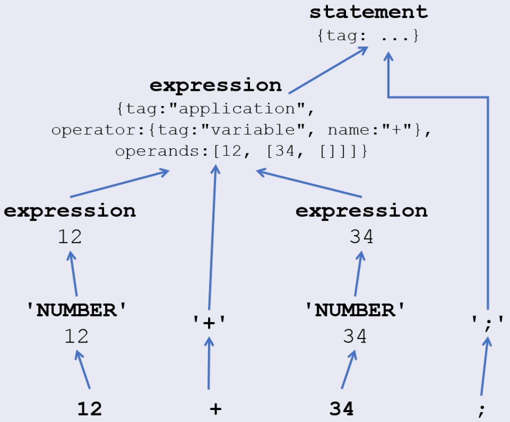
The working process of program execution

Only applicable to interpreter using Abstract Syntax Tree (AST) parser:

- 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.

More About Interpreter

Abstract Syntax Tree (AST)



Overview

1. Object-oriented Programming
   - Literal objects
   - Object-oriented paradigm *(optional)*
2. More about interpreter
   - Revisit language processing
   - How does interpreter work
3. Meta-circular evaluator
   - Basic evaluator
   - Loop & assignment
   - OOP evaluator *(optional)*
   - Lazy evaluator *(optional)*
   - Memoized evaluator *(optional)*

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 of the same language than the target language.

Meta-circular Evaluator

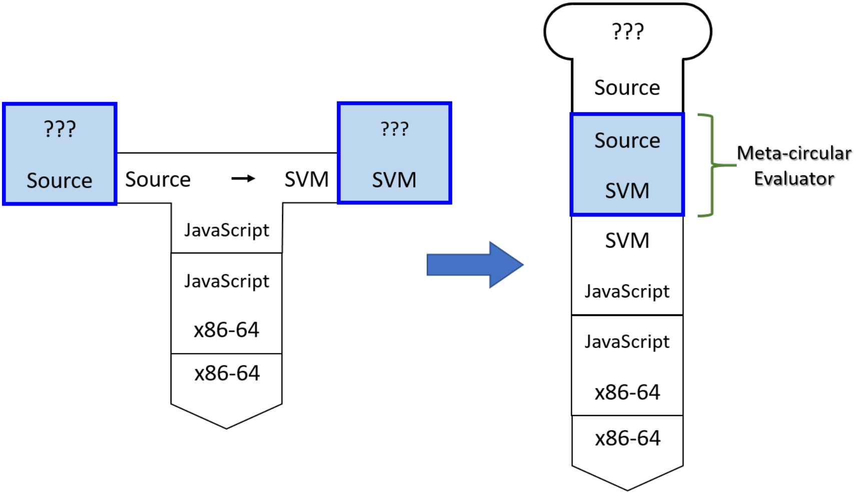
Meta-circular evaluator for the Source language

- 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_builtin_function(fun) {
   return is_tagged_object(fun,"builtin");
}

function builtin_implementation(fun) {
   return fun.implementation;
}

function make_builtin_function(impl) {
    return { tag: "builtin", implementation: impl };
}

function apply_builtin_function(fun, argument_list) {
    return apply_in_underlying_javascript(
```

Meta-circular Evaluator

Primitive operators

```
const builtin_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(names, values) {
    const frame = {};
    while (!is_empty_list(names) && !is_empty_list(values))
        {
        add_binding_to_frame(head(names), head(values),
            frame, true);
        names = tail(names);
        values = tail(values);
    }
    return frame;
}

function add_binding_to_frame(name, value, frame, mutable) {
    // object field assignment
    frame[name] = make_denoted(value, mutable);
    return undefined;
}
```

Meta-circular Evaluator

Environment table lookup

```
function lookup_name_value(name, env) {
    function env_loop(env) {
        if (is_empty_environment(env)) {
            error("Unbound name: " + name);
        } else if (has_binding_in_frame(name, first_frame(
            env))) {
            return first_frame(env)[name].value;
        } else {
            return env_loop(enclosing_environment(env));
        }
    }
    return env_loop(env);
}

function has_binding_in_frame(name, frame) {
    return frame[name] !== undefined;
}
```

Meta-circular Evaluator

To extend environment

```
function extend_environment(names, vals, base_env) {
    if (length(names) === length(vals)) {
        return enclose_by(make_frame(names, vals),
                          base_env);
    } else if (length(names) < length(vals)) {
        error("Too many arguments: " + names + vals);
    } else {
        error("Too few arguments: " + names + vals);
    }
}
```

Meta-circular Evaluator

Function application

```
function apply(fun, args) {
    if (is_builtin_function(fun)) {
        return apply_builtin_function(fun, args);
    } else if (is_function_object(fun)) {
        return get_return_value(fun, args);
    } else {
        error("Unknown function type in apply: " + fun);
    }
}
```

Meta-circular Evaluator

`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

```
function get_return_value(fun, args) {
    const result = evaluate(
        function_object_body(fun),
        extend_environment(
            function_object_parameters(fun), args,
            function_object_environment(fun)));

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

Stateful programming

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

Meta-circular Evaluator

while loop

```
function evaluate_while_loop(stmt, env) {
    const pred = evaluate(while_loop_predicate(stmt), env);

    if (pred) {
        evaluate(while_loop_statements(stmt), env);
        return evaluate_while_loop(stmt, env);
    } else {
        return true;
    }
}
```

for loop

```
function evaluate_for_loop(stmt, env) {
    const init = for_loop_initialiser(stmt);
    const body = block_body(for_loop_statements(stmt));
    const fina = list(for_loop_finaliser(stmt));
    const loop = make_while_loop(for_loop_predicate(stmt),
                                 append(body, fina));

    return evaluate(make_block(list(init, loop)), env);
}
```

Meta-circular Evaluator

Assignment

```
function assign_name_to_value(name, value, env) {
    function env_loop(env) {
        if (is_empty_environment(env)) {
            error("Unbound name: " + name);
        } else if (has_binding_in_frame(name, first_frame(
            env))) {
            first_frame(env)[name].value = value;
        } else {
            return env_loop(enclosing_environment(env));
        }
    }

    return env_loop(env);
}
```

Meta-circular Evaluator

Assignment

```
function evaluate_assignment(stmt, env) {
    const value = evaluate(assignment_right_hand_side(stmt),
        env);
    assign_name_to_value(assignment_name(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) {
    const 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) {
    const obj = evaluate(object(stmt), env);
    const prop = evaluate(property(stmt), env);
    return obj[prop];
}

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

Meta-circular Evaluator

To invoke the method of an object

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

    const first_arg = obj;
    const 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) {
    const obj = {};
    const 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***.

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 {
            const value = evaluate(thunk_expression(v),
                                   thunk_environment(v));
            thunk_memoize(v, value);
        }
    } else {
        return v;
    }
}
```

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))) {
            const value = force(first_frame(env)[variable]);
            first_frame(env)[variable] = value;
            return value;
        } else {
            return env_loop(enclosing_environment(env));
        }
    }
    return env_loop(env);
}
```

Studio Group Problems

Let's discuss them now.

The End

Copyright