Source Style Guide—Version 2.1

Martin Henz

National University of Singapore
School of Computing

October 21, 2025

This is the style guide for the language Source, the JavaScript sublanguage used in the book *Structure and Interpretation of Computer Programs*, JavaScript Adaptation (SICP JS). In the beginning of a first-year course, instructors may just *encourage* adherence to the style, and then gradually enforce adherence in order to improve communication between learners and staff.

Line Length and tabs

Newline characters should be indented such that lines should not be longer than 80 characters. The book uses at most 70 characters per line.

Blocks

The body of blocks are indented by four characters more than their surrounding program.

```
// good example
const x = 1;
{
    const y = 2;
    display(x + y);
    {
        const z = 3;
        display(x + y + z);
    }
}
```

Function declarations

The brace that starts the body of a function declaration is placed at the end of the line that starts with the keyword **`function`**. It is separated from the parameter list by one space. Example:

```
function make_adder(x) {
    function the_adder(y) {
        return x >= 0 ? x + y : - x + y;
    }
    return the_adder;
}
```

JavaScript syntax enforces that there cannot be a newline in a return statement between the keyword **`return`** and the return expression.

```
// illegal JavaScript
function square(x) {
    return
        x * x;
}
```

Conditional Statements

The brace that starts the consequent block of a conditional statement is placed at the end of the line that starts with the keyword `if`. The keyword `else` is in the same line as the brace that ends the consequent block and has one space before and after the keyword, followed by the brace that starts the alternative block.

```
// good example
function make_abs_adder(x) {
    function the_adder(y) {
        if (x >= 0) {
            return x + y;
        } else {
            return - x + y;
        }
    }
    return the_adder;
}
```

Always use braces... even if the block consists of only one statement. This is required in Source, and recommended in JavaScript.

```
// good example
if (<predicate>) {
    return x + y;
} else {
    return x * y;
}

// bad example (illegal in Source)
if (<predicate>)
    return x + y;
else
    return x * y;

// even worse
if (<predicate>) return x + y;
else return x * y;
```

SICP JS always uses `else`... even if the alternative block is empty. An empty block is written as `{}` with no space or newline between the braces.

```
// good example
function signal_error(p, x) {
    if (p) {
        error(x, "fatal error, bad value found:");
    } else {}
}

// bad example (space in empty block)
function signal_error(p, x) {
    if (p) {
        error(x, "fatal error, bad value found:");
    } else {
    }
}
```

Conditional expressions

The consequent and alternative expressions of conditional expressions are either in the same line as the predicate (if they fit), or they are aligned under the beginning of the predicate. Examples:

```
// good style
function abs(x) {
    return x >= 0 ? x : - x;
}

// good style
function abs(x) {
    return x > 0
           ? x
           : x === 0
           ? 0
           : - x;
}

// good style
const aspect_ratio = landscape ? 4 / 3 : 3 / 4;

// bad style: wasted lines
const aspect_ratio = landscape
                     ? 4 / 3
                     : 3 / 4;
```

Avoid parentheses around the predicate. Note that conditional expressions have lower precedence than all other operators in Source (except the assignment operator `=`, so parentheses around the predicate are rarely needed.

```
// good style
function abs(x) {
    return x >= 0 ? x : - x;
}

// bad style
function abs(x) {
    return (x >= 0) ? x : - x;
}
```

If the *consequent-expression* or *alternative-expression* are lengthy, use indentation, but align all clauses of a conditional expression without further indentation.

```
// good style
function A(x,y) {
    return y === 0
           ? 0
           : x === 0
           ? 2 * y
           : y === 1
           ? 2
           : A(x - 1, A(x, y - 1));
}

// bad style: line too long
function A(x,y) {
    return y === 0 ? 0 : x === 0 ? 2 * y : y === 1 ? 2 : A(x - 1, A(x, y - 1));
}
```

```
// bad style: too much indentation
function A(x,y) {
    return y === 0
           ? 0
           : x === 0
             ? 2 * y
             : y === 1
               ? 2
               : A(x - 1, A(x, y - 1));
}
```

Operator combinations

Leave a single space between operators and their operands.

```
// good style
const x = 1 + 1;
return x === 0 ? "zero" : "not zero";
const negative_x = - x;

// bad style
const x=1+1;
return x === 0?"zero":"not zero";
const negative_x = -x;
```

Arguments and parameters

The arguments of function applications and the parameters of function declarations and lambda expressions are aligned either horizontally (if they fit in one line) or vertically (if they don't fit in one line). The first argument of of a function application either appears immediately after the open parenthesis that encloses the arguments or is indented by four characters compared to the beginning of the function expression.

```
// good example:
// note how arguments are aligned vertically when necessary
// indented below the first argument or by four characters
function execute_application(fun, args, succeed, fail) {
    return is_primitive_function(fun)
           ? succeed(apply_primitive_function(fun, args),
                     fail)
           : is_compound_function(fun)
           ? function_body(fun)(
                 extend_environment(
                     function_parameters(fun),
                     args,
                     function_environment(fun)),
                 (body_result, fail2) =>
                   succeed(is_return_value(body_result)
                           ? return_value_content(body_result)
                           : undefined,
                           fail2),
                 fail)
           : error(fun, "unknown function type -- execute_application");
}
```

```
// good example:
// the arguments of make_execution_function
// are ok here; five newlines would be excessive here
function update_insts(insts, labels, machine) {
    const pc = get_register(machine, "pc");
    const flag = get_register(machine, "flag");
    const stack = machine("stack");
    const ops = machine("operations");
    return for_each(inst => set_instruction_execution_fun(
                                inst,
                                make_execution_function(
                                    instruction_source(inst),
                                    labels, machine, pc,
                                    flag, stack, ops)),
                    insts);
}

// good example:
// the parameters of make_execution_function
// are ok here; six newlines would be excessive here
function make_execution_function(inst, labels, machine,
                                 pc, flag, stack, ops) {
    return type(inst) === "assign"
           ? make_assign_ef(inst, machine, labels, ops, pc)
           : type(inst) === "test"
           ? make_test_ef(inst, machine, labels, ops, flag, pc)
           : type(inst) === "branch"
           ? make_branch_ef(inst, machine, labels, flag, pc)
           : type(inst) === "go_to"
           ? make_go_to_ef(inst, machine, labels, pc)
           : type(inst) === "save"
           ? make_save_ef(inst, machine, stack, pc)
           : type(inst) === "restore"
           ? make_restore_ef(inst, machine, stack, pc)
           : type(inst) === "perform"
           ? make_perform_ef(inst, machine, labels, ops, pc)
           : error(inst, "unknown instruction type -- assemble");
}
```

When calling or declaring a function with multiple parameters, leave a space after each comma. There should be no spaces before your parameter list.

```
// good style
function my_function(arg1, arg2, arg3) {
    ...
}

// bad style
function my_function (arg1, arg2, arg3) {
    ...
}

// good style
my_function(1, 2, 3);

// bad style
my_function(1,2,3);

// bad style
my_function (1, 2, 3);
```

There should be no spaces after opening parentheses and before closing parentheses.

```
// good style
function my_function(arg1, arg2, arg3) {
    ...
}

// good style
my_function(1, 2, 3);

// good style
if (x === 1) {
    ...
} else {}

// bad style
function my_function( arg1, arg2, arg3 ) {
    ...
}

// bad style
my_function( 1, 2, 3 );

// bad style
if ( x === 1 ) {
    ...
```

Lambda expressions

Keep the parameters and the body expression of a lambda expression in one line, if they are short enough.

If they are lengthy, use indentation. The indentation of lambda expressions with expression bodies is two characters and the indentation of lambda expressions with block bodies is four characters.

JavaScript does not allow a newline between the parameters and the => symbol.

```
// good style
function count_buttons(garment) {
    return accumulate((sleaves, total) => sleaves + total,
                      0,
                      map(jacket =>
                            is_checkered(jacket)
                            ? count_buttons(jacket)
                            : 1,
                          garment));
}

// good style
function count_buttons(garment) {
    return accumulate(
               (sleaves, total) =>
                 delicate_calculation(sleaves + total),
               0,
               map(jacket =>
                     is_checkered(jacket)
                     ? count_buttons(jacket)
                     : 1,
                   garment));
}
```

```
// good style
function count_buttons(garment) {
    return accumulate((sleaves, total) => {
                          return sleaves + total;
                      },
                      0,
                      map(jacket => {
                              return is_checkered(jacket)
                                     ? count_buttons(jacket)
                                     : 1;
                          },
                          garment));
}

// good style
function count_buttons(garment) {
    return accumulate(
               (sleaves, total) => {
                   return delicate_calculation(sleaves + total);
               },
               0,
               map(jacket => {
                       return is_checkered(jacket)
                              ? count_buttons(jacket)
                              : 1;
                   },
                   garment));
}

// bad style: too little indentation
function count_buttons(garment) {
    return accumulate((sleaves, total) =>
               delicate_calculation(sleaves + total),
               0,
               map(jacket =>
               is_checkered(jacket)
               ? count_buttons(jacket)
               : 1,
               garment));
}

// illegal JavaScript: no newline allowed between parameters and =>
function count_buttons(garment) {
    return accumulate(
        (sleaves, total)
            => delicate_calculation(sleaves + total),
        0,
        map(jacket
            => is_checkered(jacket)
               ? count_buttons(jacket)
               : 1,
            garment));
}
```

Import directives

Import directives enclose the declared names in braces `{ ... }`, and the names are indented by four characters.

```
// good style
import {
    beside, heart,
    show
} from "rune";

show(beside(heart, heart));
```

Trailing whitespace

There should be no whitespace (spaces or tabs) before a newline, because they cause problems in tools that handle program text such as version control systems. Clean up all such *trailing whitespace* before submitting your program.

Names

When naming constants or variables, choose names that are meaningful to your reader and that make sense in the application domain. Names should only contain lower case letter of the roman alphabet, digits, and underscores. Use underscores to separate words.

Good examples: `my_variable`, `x_1`, `x1`, `draw_connected_proportional`

Bad examples: π, `myVariable`, `Draw_Connected_Proportional`, 神秘的名字

Nested scope

Do not use the same name in a nested scope. Examples:

```
// bad program
const x = 1;
function f(x) {
    // here, the name x declared using const
    // is ''shadowed'' by the formal parameter x
    ...
}

// another bad program
function f(x) {
    return x => ...;
    // here, the formal parameter x of f is ''shadowed''
    // by the formal parameter of the returned function
}

// illegal in JavaScript (and therefore Source)
function f(x) {
    const x = 1;
    // in JavaScript, a parameter of a function cannot be
    // redeclared in the body block of the function
    ...
}
```

Finally, the worst case would be a (surely accidental) use of the same variable name for two parameters of a function. In this case, the second variable is not visible; it is “shadowed” by the first. This is illegal in Source and JavaScript (strict mode).

```
// illegal in Source and JavaScript strict mode
function f(x, x) {
    ...
}
```

Comments

Comments should be used to describe and explain statements that might not be obvious to a reader. Redundant comments should be avoided.

An example for bad commenting (no added information):

```
// square computes the square of the argument x
function square(x) {
    return x * x;
}
```

For multi-line comments, use `/* ... */` and for single line comments, use `//`.

As in any technical writing, comments should be precise and concise. Put yourself in the shoes of the reader of the program. What information would be helpful to the reader to understand your program? Ideally, the program should be written such that little comment is needed. Use comments to explain ideas that are not obvious when just reading the program.

Comments and names

The comment in the following program might be useful because it explains what `x` and `y` stands for and what type of object is meant.

```
// area of rectangle with sides x and y
function area(x, y) {
    return x * y;
}
```

The programmer has decided to use the short word `area` as the name of the function, which is ok, as long as it is clear that the geometric objects that the program deals with are always rectangles. Alternatively, a longer name of the function makes the comment unnecessary.

```
function area_of_rectangle(x, y) {
    return x * y;
}
```