L3: Refresher on Elements and Functions

Structure and Interpretation of Computer Programs

Martin Henz

July 6, 2023

1. What is on offer?
   - Some goals
   - Today
   - Team today
   - Book
2. Module 1: Elements of programming
3. Module 2: Introduction to functions

1. What is on offer?
   - Some goals
   - Today
   - Team today
   - Book
2. Module 1: Elements of programming
3. Module 2: Introduction to functions

Some goals

- A taste of
  the *structure and interpretation of computer programs*

Some goals

- A taste of
  the *structure and interpretation of computer programs*
- Learn to write simple programs

Some goals

- A taste of
  the *structure and interpretation of computer programs*
- Learn to write simple programs
- Discover a mental model for computational processes

Some goals

- A taste of
  the *structure and interpretation of computer programs*
- Learn to write simple programs
- Discover a mental model for computational processes
- Fun with graphics and sound

What we are **not** doing

- Teach the language JavaScript
- Teach how to program websites
- Introduce object-oriented programming

1. What is on offer?
   - Some goals
   - Today
   - Team today
   - Book

2. Module 1: Elements of programming

3. Module 2: Introduction to functions

Components

Components

- **L**ecture: 10–12: slides available in Canvas

Components

- **L**ecture: 10–12: slides available in Canvas
- **P**ath: will be discussed during lecture

Components

- **L**ecture: 10–12: slides available in Canvas
- **P**ath: will be discussed during lecture
- **Q**uests: will be discussed in afternoon session: 2-6

This morning

- Modules 1 and 2 refresher
    - 10:05–10:45: Elements of programming
    - 10:45–11:00: break; getting to know each other
    - 11:00–11:45: Functional abstraction
    - 11:45–12:00: more getting to know each other

1. What is on offer?
   - Some goals
   - Today
   - Team today
   - Book
2. Module 1: Elements of programming
3. Module 2: Introduction to functions

Team

Instructor

Martin Henz, Associate Professor at the National University of Singapore

Team

Instructor

Martin Henz, Associate Professor at the National University of Singapore

Teaching Assistant Brian will join at 2pm

- answering your questions
- clarifying concepts

1. What is on offer?
   - Some goals
   - Today
   - Team today
   - Book
2. Module 1: Elements of programming
3. Module 2: Introduction to functions

SICP JS

Online version

https://sourceacademy.nus.edu.sg/sicpjs

SICP JS

Online version

https://sourceacademy.nus.edu.sg/sicpjs

PDF version

https://sicp.sourceacademy.org/sicpjs.pdf

SICP JS

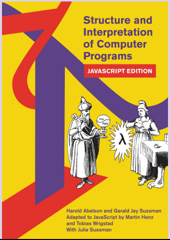
Online version

https://sourceacademy.nus.edu.sg/sicpjs

PDF version

https://sicp.sourceacademy.org/sicpjs.pdf

paper version



Processes

Processes

Definition (from Wikipedia)

A process is a set of activities that interact to produce a result. The activities unfold according to patterns that *de*scribe or *pre*scribe the process.

Processes

Definition (from Wikipedia)

A process is a set of activities that interact to produce a result. The activities unfold according to patterns that *de*scribe or *pre*scribe the process.

Examples

Processes are everywhere. They permeate our nature and culture:

- Galaxies and solar systems
- Metabolic pathways in our bodies
- Political parties, legislature, courts
- Industrial production

Computational processes

Definition

A *computational process* is a set of activities in a computer, designed to achieve a desired result.

Computational processes

Definition

A *computational process* is a set of activities in a computer, designed to achieve a desired result.

Our task

Here we are concerned about how this *design* happens.

Computational processes

Definition

A *computational process* is a set of activities in a computer, designed to achieve a desired result.

Our task

Here we are concerned about how this *design* happens.

Our design method

We use *programs* to prescribe how computational processes unfold.

What is programming?

What is programming?

People in focus

As the complexity of computer systems increases, *communication* between affected people becomes more and more important.

What is programming?

People in focus

As the complexity of computer systems increases, *communication* between affected people becomes more and more important.

Programs as communication devices

Since programs prescribe the computational processes in these systems, they allow us to communicate their construction and operation.

What is programming?

People in focus

As the complexity of computer systems increases, *communication* between affected people becomes more and more important.

Programs as communication devices

Since programs prescribe the computational processes in these systems, they allow us to communicate their construction and operation.

A central theme of SICP

Programming is *communicating computational processes*.

Our programming environment: Source Academy

Our programming environment: Source Academy

Website designed for SICP JS

Has just what you need for understanding the *structure and interpretation of computer programs*.

Our programming environment: Source Academy

Website designed for SICP JS

Has just what you need for understanding the *structure and interpretation of computer programs*.

Research

Source Academy is also a *educational research tool*: We want to study what happens when people like you learn how to program.

Our programming environment: Source Academy

Website designed for SICP JS

Has just what you need for understanding the *structure and interpretation of computer programs*.

Research

Source Academy is also a *educational research tool*: We want to study what happens when people like you learn how to program.

Source Academy and Source Academy @ NUS

Source Academy (`https://sourceacademy.org`) is public.
We use mostly Source Academy @ NUS, which supports courses and requires login (`https://sourceacademy.nus.edu.sg`).

Elements of programming

Elements of programming

Every powerful language provides...

Elements of programming

Every powerful language provides...

- Primitive values

Elements of programming

Every powerful language provides...

- Primitive values
- Combination

Elements of programming

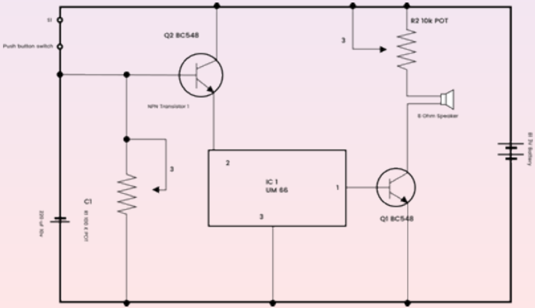
Every powerful language provides...

- Primitive values
- Combination
- Means of abstraction

Elements of programming

Every powerful language provides...

- Primitive values
- Combination
- Means of abstraction



Primitive expressions (1.1.1)

Primitive expressions (1.1.1)

- Numerals: 0, -42, 486

Primitive expressions (1.1.1)

- Numerals: 0, -42, 486
- Numerals use decimal notation

Primitive expressions (1.1.1)

- Numerals: 0, -42, 486
- Numerals use decimal notation
- Our interpreter can evaluate numerals, resulting in the numbers they represent

A detail

Expressions are not programs in Source

Instead they can be turned into programs using a semicolon.

A detail

Expressions are not programs in Source

Instead they can be turned into programs using a semicolon.

Example

`486;` is a program.

A detail

Expressions are not programs in Source

Instead they can be turned into programs using a semicolon.

Example

`486;` is a program.

What process does the program prescribe?

A detail

Expressions are not programs in Source

Instead they can be turned into programs using a semicolon.

Example

`486;` is a program.

What process does the program prescribe?

The process is *trivial*:
There is nothing to do. The result is already explicit.

Means of Combination: Operators (1.1.1)

Examples

```
5 * 99;

25 - (4 + 2) * 3;
```

Means of Combination: Operators (1.1.1)

Examples

```
5 * 99;

25 - (4 + 2) * 3;
```

Notation as usual

operator between operands:
infix notation with *precedences*

Evaluating Operator Combinations

But exactly what happens...

...when we evaluate a program like this

```
(2 + 4 * 6) * (3 + 12);
```

???

Evaluation of expressions (1.1.3)

Demonstration: evaluate

```
(2 + 4 * 6) * (3 + 12);
```

Means of Abstraction: Naming (1.1.2)

Example

```
const size = 2;
5 * size;
```

Path 1: Expressions and names

See Source Academy https://sourceacademy.nus.edu.sg

Pre-declared names

Pre-declared constants

Source has a few names pre-declared. For example, the name `math_PI` refers to the constant π

Pre-declared names

Pre-declared constants

Source has a few names pre-declared. For example, the name `math_PI` refers to the constant π

Predeclared functions

Source has *predeclared functions*.

Example: `math_sqrt` is the square root function.

Pre-declared names

Pre-declared constants

Source has a few names pre-declared. For example, the name `math_PI` refers to the constant π

Predeclared functions

Source has *predeclared functions*.

Example: `math_sqrt` is the square root function.

Function application expressions...

...use the usual mathematical notation:

```
math_sqrt(15);
```

applies `math_sqrt` to 15.

Import declarations

Import syntax

Source allows *import declarations*. Each of a list of names is imported from a given module. Example:

```
import { heart, show } from 'rune';
```

Now the name `heart` refers to a "rune" (shape) defined by the module `'rune'` and `show` refers to a rune function.

```
show(heart);  // shows heart rune in a tab
```

Import declarations

Import syntax

Source allows *import declarations*. Each of a list of names is `imported from` a given module. Example:

```
import { heart, show } from 'rune';
```

Now the name `heart` refers to a "rune" (shape) defined by the module `'rune'` and `show` refers to a rune function.

```
show(heart);  // shows heart rune in a tab
```

Modules

The list of all modules is given in
https://source-academy.github.io/modules/documentation/

Quest 1A: Playing with Runes

See Source Academy https://sourceacademy.nus.edu.sg

1. What is on offer?
2. Module 1: Elements of programming
   - Expressions and names, 1.1.1 and 1.1.2
   - Predeclared names
   - Functional abstraction, 1.1.4
   - Predicates and conditional expressions, 1.1.6
3. Module 2: Introduction to functions

Means of Abstraction: Compound functions (1.1.4)

Example

```
function square(x) {
    return x * x;
}
```

Means of Abstraction: Compound functions (1.1.4)

Example

```
function square(x) {
    return x * x;
}
```

What does this statement mean?

This *function declaration* declares a name, here `square`. The value associated with `square` is a function that takes an argument `x` and produces (returns) the result of multiplying `x` by itself.

Review: Function application

Recall from the `math_sqrt` example

We apply a function by supplying its arguments in parentheses.

Review: Function application

Recall from the `math_sqrt` example

We apply a function by supplying its arguments in parentheses.

Example

```
function square(x) {
    return x * x;
}
square(7);
```

The core of the substitution model

```
function square(x) {
    return x * x;
}
square(7);
```

The core of the substitution model

```
function square(x) {
    return x * x;
}
square(7);
```

Function application

To apply a compound function to arguments,
evaluate the return expression of the function
with each parameter *replaced* by the corresponding argument.

More examples

```
square(21);

square(2 + 5);

square(square(3));
```

1. What is on offer?
2. Module 1: Elements of programming
   - Expressions and names, 1.1.1 and 1.1.2
   - Predeclared names
   - Functional abstraction, 1.1.4
   - Predicates and conditional expressions, 1.1.6
3. Module 2: Introduction to functions

Boolean values (1.1.6)

Boolean values

The two boolean values `true` and `false` represent *answers* to yes-no questions

Boolean values (1.1.6)

Boolean values

The two boolean values `true` and `false` represent *answers* to yes-no questions

Predicates

A *predicate* is a function or an expression that returns or evaluates to a boolean value.

Boolean values (1.1.6)

Boolean values

The two boolean values `true` and `false` represent *answers* to yes-no questions

Predicates

A *predicate* is a function or an expression that returns or evaluates to a boolean value.

Examples

```
true;
false;
x >= 1;
```

Declare predicate with compound function

Example

```
function adult(x) {
    return x >= 18;
}
```

Conditional expressions

Purpose

We would like to check something, e.g. answer a yes/no question, and return a different value, depending on the answer

Conditional expressions

Purpose

We would like to check something, e.g. answer a yes/no question, and return a different value, depending on the answer

Example:

```
adult(my_age)
? enter_club()
: go_to_movie()
```

Conditional expressions

Purpose

We would like to check something, e.g. answer a yes/no question, and return a different value, depending on the answer

Example:

```
adult(my_age)
? enter_club()
: go_to_movie()
```

Another example: abs function

Check if the number is greater or equal 0.
If yes, return the number unchanged,
and if no, return the number negated.

Quest 1B: Rune Trials

See Source Academy `https://sourceacademy.nus.edu.sg`

Racap: Elements of Programming (1.1)

Racap: Elements of Programming (1.1)

- Primitives: things like −42

Racap: Elements of Programming (1.1)

- Primitives: things like `-42`
- Combination: things like `-42 * 7`

Racap: Elements of Programming (1.1)

- Primitives: things like `-42`
- Combination: things like `-42 * 7`
- Name abstraction: things like `const size = 2;`

Racap: Elements of Programming (1.1)

- Primitives: things like `-42`
- Combination: things like `-42 * 7`
- Name abstraction: things like `const size = 2;`
- Functional abstraction: ...

A function can be a "black box"

Example

`math_sqrt`

A function can be a "black box"

Example

`math_sqrt`

- Input: any number
- Output: a number whose square is the input number

A function can be a "black box"

Example

`math_sqrt`

- Input: any number
- Output: a number whose square is the input number

Function application

```
math_sqrt(15);
```

Means of Abstraction: Compound functions

Example

```
function square(x) {
    return x * x;
}
```

Means of Abstraction: Compound functions

Example

```
function square(x) {
    return x * x;
}
```

What does this statement mean?

Like constant declarations, this *function declaration* declares a name, here `square`. The value associated with the name is a function that takes an argument `x` and produces (returns) the result of multiplying `x` by itself.

Substitution model

Substitution model

- Keep making small steps until the simplest possible form is reached

Substitution model

- Keep making small steps until the simplest possible form is reached
- Reduce statement by statement, in the given order

Substitution model

- Keep making small steps until the simplest possible form is reached
- Reduce statement by statement, in the given order
- Reduce operands of operator combinations and arguments of function applications, before applying the operator/function

Substitution model

- Keep making small steps until the simplest possible form is reached
- Reduce statement by statement, in the given order
- Reduce operands of operator combinations and arguments of function applications, before applying the operator/function
- Reduce the predicate of conditional expressions until it is true or false

Substitution model

- Keep making small steps until the simplest possible form is reached
- Reduce statement by statement, in the given order
- Reduce operands of operator combinations and arguments of function applications, before applying the operator/function
- Reduce the predicate of conditional expressions until it is true or false
- If predicate is true/false, keep consequent/alternative

Substitution model

- Keep making small steps until the simplest possible form is reached
- Reduce statement by statement, in the given order
- Reduce operands of operator combinations and arguments of function applications, before applying the operator/function
- Reduce the predicate of conditional expressions until it is true or false
- If predicate is true/false, keep consequent/alternative
- When the arguments of a function application are *values*, replace the application by the return expression of the function, where parameters are replaced by the argument values.

A new predeclared combination: `stack_frac`

`stack_frac(r, heart, sail)`

splits available bounded heart occupies top fraction `r` of box and sail occupies remaining `1 - r` of box

Examples

`stack_frac(87 / 100, heart, sail)`

splits available bounded box such that heart occupies the top 87% of box and sail occupies remaining 13% of box

Examples

stack_frac(87 / 100, heart, sail)

splits available bounded box such that heart occupies the top 87% of box and sail occupies remaining 13% of box

Trisection of the heart

```
stack_frac(
    1 / 3,
    heart,
    stack_frac(
        1 / 2,
        heart,
        heart));
```

Can we define stackn in Source?

Trisection of the heart

```
stack_frac(1 / 3, heart,
  stack_frac(1 / 2, heart, heart));
```

Can we define stackn in Source?

Trisection of the heart

```
stack_frac(1 / 3, heart,
  stack_frac(1 / 2, heart, heart));
```

Quadrisection of the heart

```
stack_frac(1 / 4, heart,
  stack_frac(1 / 3, heart,
    stack_frac(1 / 2, heart,
                heart)));
```

Can we define stackn in Source?

Trisection of the heart

```
stack_frac(1 / 3, heart,
  stack_frac(1 / 2, heart, heart));
```

Quadrisection of the heart

```
stack_frac(1 / 4, heart,
  stack_frac(1 / 3, heart,
    stack_frac(1 / 2, heart,
                 heart)));
```

Can we generalise this idea?

A *Recursive* Function, first try

```
function stackn(n, rune) {
    return stack_frac(1 / n,
                      rune,
                      stackn(n - 1, rune));
}

stackn(3, heart);
```

A *Recursive* Function, first try

```
function stackn(n, rune) {
    return stack_frac(1 / n,
                      rune,
                      stackn(n - 1, rune));
}

stackn(3, heart);
```

Not working! Why?

A *Recursive* Function, first try

```
function stackn(n, rune) {
    return stack_frac(1 / n,
                      rune,
                      stackn(n - 1, rune));
}

stackn(3, heart);
```

Not working! Why? Computers will follow our orders

We need to *precisely* describe *how* a computational process should evolve.

The correct version

```
function stackn(n, rune) {
    return n === 1
           ? rune
           : stack_frac(1 / n,
                        rune,
                        stackn(n - 1, rune));
}
```

The correct version

```
function stackn(n, rune) {
    return n === 1
           ? rune
           : stack_frac(1 / n,
                        rune,
                        stackn(n - 1, rune));
}
```

Observation

Solution for n computed using solution $n-1$,

solution for $n-1$ is computed using solution $n-2$, ...

until we reach trivial case.

"Wishful thinking"

```
function stackn(n, rune) {
    return n === 1
           ? rune
           : stack_frac(1 / n,
                        rune,
                        stackn(n - 1, rune));
}
```

"Wishful thinking"

```
function stackn(n, rune) {
    return n === 1
           ? rune
           : stack_frac(1 / n,
                        rune,
                        stackn(n - 1, rune));
}
```

Recipe for recursion

"Wishful thinking"

```
function stackn(n, rune) {
    return n === 1
           ? rune
           : stack_frac(1 / n,
                        rune,
                        stackn(n - 1, rune));
}
```

Recipe for recursion

- Figure out trivial *base case*

"Wishful thinking"

```
function stackn(n, rune) {
    return n === 1
           ? rune
           : stack_frac(1 / n,
                        rune,
                        stackn(n - 1, rune));
}
```

Recipe for recursion

- Figure out trivial *base case*
- Assume you know how to solve problem for $n - 1$.

"Wishful thinking"

```
function stackn(n, rune) {
    return n === 1
           ? rune
           : stack_frac(1 / n,
                        rune,
                        stackn(n - 1, rune));
}
```

Recipe for recursion

- Figure out trivial *base case*
- Assume you know how to solve problem for $n - 1$. How can we solve problem for n?

Can we define repeat_pattern in Source?

Consider the function repeat_pattern in module rune

```
repeat_pattern(3, make_cross, sail)
// should lead to
make_cross(make_cross(make_cross(sail)))
```

Can we define repeat_pattern in Source?

Consider the function repeat_pattern in module rune

```
repeat_pattern(3, make_cross, sail)
// should lead to
make_cross(make_cross(make_cross(sail)))
```

Another example

```
function square(x) { return x * x; }
repeat_pattern(3, square, 2);
// should lead to
square(square(square(2)));
```

repeat_pattern, our first version

```
function repeat_pattern(n, pat, init) {
  return n === 0
         ? init
         : pat(repeat_pattern(n - 1, pat, init));
}
```

repeat_pattern, our first version

```
function repeat_pattern(n, pat, init) {
  return n === 0
         ? init
         : pat(repeat_pattern(n - 1, pat, init));
}
repeat_pattern(3, square, 2);
```

repeat_pattern, our first version

```
function repeat_pattern(n, pat, init) {
  return n === 0
         ? init
         : pat(repeat_pattern(n - 1, pat, init));
}
repeat_pattern(3, square, 2);
```

Recursive process

The applications of pat
accumulate as result of recursive calls.
They are *deferred* operations.

repeat_pattern, second version

```
function repeat_pattern(n, pat, rune) {
  return n === 0
         ? rune
         : repeat_pattern(n - 1, pat, pat(rune));
}
```

repeat_pattern, second version

```
function repeat_pattern(n, pat, rune) {
  return n === 0
         ? rune
         : repeat_pattern(n - 1, pat, pat(rune));
}

repeat_pattern(3, square, 2);
```

repeat_pattern, second version

```
function repeat_pattern(n, pat, rune) {
  return n === 0
         ? rune
         : repeat_pattern(n - 1, pat, pat(rune));
}

repeat_pattern(3, square, 2);
```

Difference

pat function is applied *before* the recursive call.

There is no deferred operation.

Summary

- We started with primitive constants.

Summary

- We started with primitive constants.
- No idea how they are rendered!

Summary

- We started with primitive constants.
- No idea how they are rendered! Example: `heart`

Summary

- We started with primitive constants.
- No idea how they are rendered! Example: `heart`
- We introduced primitive combinations.

Summary

- We started with primitive constants.
- No idea how they are rendered! Example: `heart`
- We introduced primitive combinations.
- No idea how the primitive combinations work!

Summary

- We started with primitive constants.
- No idea how they are rendered! Example: `heart`
- We introduced primitive combinations.
- No idea how the primitive combinations work!
  Example: `quarter_turn_right`

Summary

- We started with primitive constants.
- No idea how they are rendered! Example: `heart`
- We introduced primitive combinations.
- No idea how the primitive combinations work!
  Example: `quarter_turn_right`
- Yet we can use primitives and combinations
  to generate complex runes.

Summary

- We started with primitive constants.
- No idea how they are rendered! Example: `heart`
- We introduced primitive combinations.
- No idea how the primitive combinations work! Example: `quarter_turn_right`
- Yet we can use primitives and combinations to generate complex runes.
- **Abstractions** to conquer complexity:

Summary

- We started with primitive constants.
- No idea how they are rendered! Example: `heart`
- We introduced primitive combinations.
- No idea how the primitive combinations work!
  Example: `quarter_turn_right`
- Yet we can use primitives and combinations
  to generate complex runes.
- **Abstractions** to conquer complexity:
  Naming

Summary

- We started with primitive constants.
- No idea how they are rendered! Example: `heart`
- We introduced primitive combinations.
- No idea how the primitive combinations work!
  Example: `quarter_turn_right`
- Yet we can use primitives and combinations
  to generate complex runes.
- **Abstractions** to conquer complexity:
  Naming and **functional abstraction**

Summary

- We started with primitive constants.
- No idea how they are rendered! Example: `heart`
- We introduced primitive combinations.
- No idea how the primitive combinations work!
  Example: `quarter_turn_right`
- Yet we can use primitives and combinations
  to generate complex runes.
- **Abstractions** to conquer complexity:
  Naming and **functional abstraction**
- **Recursion**: describe the solution to a problem
  by using a solution to a (slightly) smaller problem.
  ("wishful thinking")

Quest 2A: Rune Reading

See Source Academy `https://sourceacademy.nus.edu.sg`

Another example: Factorial 1.2.1

Another example: Factorial 1.2.1

Factorial

$n\,! = n \ \ (n-1)(n-2)\cdots 1$

Another example: Factorial 1.2.1

Factorial

$$n\,! = n \quad (n-1)(n-2)\cdots 1$$

Grouping

$$n\,! = n\ ((n-1)(n-2)\cdots 1)$$

Another example: Factorial 1.2.1

Factorial

$$n\,! = n \;\; (n-1)(n-2)\cdots 1$$

Grouping

$$n\,! = n\;((n-1)(n-2)\cdots 1)$$

Replacement

$$n\,! = n\;(n-1)\;!$$

Another example: Factorial 1.2.1

Factorial

$$n\,! = n\ \ (n-1)(n-2)\cdots 1$$

Grouping

$$n\,! = n\,((n-1)(n-2)\cdots 1)$$

Replacement

$$n\,! = n\,(n-1)\,!$$

Remember the base case

$$\begin{array}{ll} n\,! = 1 & \text{if } n = 1 \\ n\,! = n\,(n-1)\,! & \text{if } n > 1 \end{array}$$

Translation into Source

Remember the base case

$$
\begin{aligned}
n\,! &= 1 && \text{if } n = 1 \\
n\,! &= n\,(n-1)\,! && \text{if } n > 1
\end{aligned}
$$

Factorial in Source

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

Example execution using Substitution Model

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

```
factorial(4)
4 * factorial(3)
4 * (3 * factorial(2))
4 * (3 * (2 * factorial(1)))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24
```

Calculating 4!

```
factorial(4)
4 * factorial(3)
4 * (3 * factorial(2))
4 * (3 * (2 * factorial(1)))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24
```

Calculating 4!

```
factorial(4)
4 * factorial(3)
4 * (3 * factorial(2))
4 * (3 * (2 * factorial(1)))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24
```

Accumulating deferred operations: *recursive* process

Absence of deferred operations: *iterative* process

Just curious: Can we write an iterative factorial?

Just curious: Can we write an iterative factorial?

```
function factorial(n) {
  return iter(1, 1, n);
}
function iter(product, counter, n) {
  return counter > n
         ? product
         : iter(counter * product,
                counter + 1,
                n);
}
```

Rick the Rabbit

Rick the Rabbit

- Rick the rabbit needs to climb a flight of stairs.

Rick the Rabbit

- Rick the rabbit needs to climb a flight of stairs.
- Given: Rick can **hop** (1 step), **skip** (2 steps) or **jump** (3 steps).

Rick the Rabbit

- Rick the rabbit needs to climb a flight of stairs.
- Given: Rick can **hop** (1 step), **skip** (2 steps) or **jump** (3 steps).
- Let's consider the problem of how many different ways can Rick climb a flight of `n` stairs?

Rick the Rabbit

Rick the Rabbit

- Consider the case of `n = 2` (two stairs):

Rick the Rabbit

- Consider the case of `n = 2` (two stairs):
  hop hop

Rick the Rabbit

- Consider the case of `n = 2` (two stairs):
  hop hop
  skip

Rick the Rabbit

- Consider the case of `n = 2` (two stairs):
  hop hop
  skip

- How about 3 stairs?

Rick the Rabbit

- Consider the case of `n = 2` (two stairs):
  hop hop
  skip

- How about 3 stairs?
  hop hop hop

Rick the Rabbit

- Consider the case of `n = 2` (two stairs):
  hop hop
  skip
- How about 3 stairs?
  hop hop hop
  skip hop

Rick the Rabbit

- Consider the case of `n = 2` (two stairs):
  hop hop
  skip

- How about 3 stairs?
  hop hop hop
  skip hop
  hop skip

Rick the Rabbit

- Consider the case of `n = 2` (two stairs):
  hop hop
  skip

- How about 3 stairs?
  hop hop hop
  skip hop
  hop skip
  jump

Rick the Rabbit

- Consider the case of `n = 2` (two stairs):
  hop hop
  skip
- How about 3 stairs?
  hop hop hop
  skip hop
  hop skip
  jump
- What about 0 stairs?
- What about -1 stairs?

Rick the Rabbit

Rick the Rabbit

- If Rick hops, we have `n - 1` stairs remaining

Rick the Rabbit

- If Rick hops, we have `n - 1` stairs remaining
- If Rick skips, we have `n - 2` stairs remaining

Rick the Rabbit

- If Rick hops, we have `n - 1` stairs remaining
- If Rick skips, we have `n - 2` stairs remaining
- If Rick jumps, we have `n - 3` stairs remaining

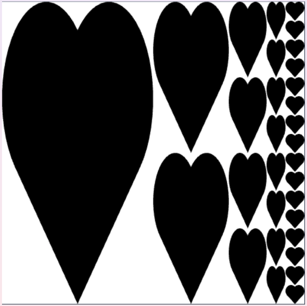
We now have a smaller problem (the number of stairs is decreasing)

Rick the Rabbit Source

```
function rabbit_ways(n) {
    return n < 0
           ? 0
           : n === 0
           ? 1
           : rabbit_ways(n - 1) // Rick hops
             +
             rabbit_ways(n - 2) // Rick skips
             +
             rabbit_ways(n - 3); // Rick jumps
}
```

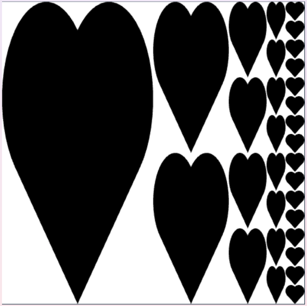
Example from Quest 2A “Rune Reading”: `fractal`

Define a function `fractal` that returns pictures like this:



Example from Quest 2A "Rune Reading": `fractal`

Define a function `fractal` that returns pictures like this:



when we call it like this: `fractal(heart, 5);`

Example from "Rune Reading": fractal, Solution 1

```
function fractal_1(rune, n) {
    return n === 1
        ? rune
        : beside(rune,
                 stack(fractal_1(rune, n - 1),
                       fractal_1(rune, n - 1)));
}
```

Tree recursion

```
function fractal_1(rune, n) {
    return n === 1
        ? rune
        : beside(rune,
                 stack(fractal_1(rune, n - 1),
                       fractal_1(rune, n - 1)));
}
```

Tree recursion

```
function fractal_1(rune, n) {
    return n === 1
        ? rune
        : beside(rune,
              stack(fractal_1(rune, n - 1),
                    fractal_1(rune, n - 1)));
}
```

Question

Can we implement this function with linear recursion?

Is this a good idea?

```
function fractal_2(rune, n) {
    const sub_frac = fractal_2(rune, n - 1);
    return n === 1
        ? rune
        : beside(rune, stack(sub_frac, sub_frac));
}
```

Is this a good idea?

```
function fractal_2(rune, n) {
    const sub_frac = fractal_2(rune, n - 1);
    return n === 1
        ? rune
        : beside(rune, stack(sub_frac, sub_frac));
}
```

Can we declare a const...

Is this a good idea?

```
function fractal_2(rune, n) {
    const sub_frac = fractal_2(rune, n - 1);
    return n === 1
        ? rune
        : beside(rune, stack(sub_frac, sub_frac));
}
```

Can we declare a const...

...**just for the alternative of the conditional**?

Conditional *statements* (see SICP JS 1.3.2)

```
function fractal_3(rune, n) {
    if (n === 1) {
        return rune;
    } else {
        const f = fractal_3(rune, n - 1);
        return beside(rune, stack(f, f));
    }
}
```

Conditional *statements* (see SICP JS 1.3.2)

```
function fractal_3(rune, n) {
    if (n === 1) {
        return rune;
    } else {
        const f = fractal_3(rune, n - 1);
        return beside(rune, stack(f, f));
    }
}
```

- Each branch of the conditional is a *block*.

Conditional *statements* (see SICP JS 1.3.2)

```
function fractal_3(rune, n) {
    if (n === 1) {
        return rune;
    } else {
        const f = fractal_3(rune, n - 1);
        return beside(rune, stack(f, f));
    }
}
```

- Each branch of the conditional is a *block*.
- A block can have local names, only visible inside the block.

Conditional *statements* (see SICP JS 1.3.2)

```
function fractal_3(rune, n) {
    if (n === 1) {
        return rune;
    } else {
        const f = fractal_3(rune, n - 1);
        return beside(rune, stack(f, f));
    }
}
```

- Each branch of the conditional is a *block*.
- A block can have local names, only visible inside the block.
- Remember to return a result *in each branch*.
  (Otherwise `undefined` is returned.)

A “divine” solution

```
function fractal_4(rune, n) {
    return n === 1
           ? rune
           : beside(rune,
                    fractal_4(stack(rune, rune),
                              n - 1));
}
```

Quest 2B: Beyond the Second Dimension

See Source Academy https://sourceacademy.nus.edu.sg