L2: Substitution Model, Recursion

CS1101S: Programming Methodology

Martin Henz

August 18, 2021

Before we start...

Where are we in CS1101S?

What is CS1101S? SICP JS, Source Academy, Studios, Missions, Piazza, Paths

Before we start...

Where are we in CS1101S?

What is CS1101S? SICP JS, Source Academy, Studios, Missions, Piazza, Paths

Getting into gears

Before we start...

Where are we in CS1101S?

What is CS1101S? SICP JS, Source Academy, Studios, Missions, Piazza, Paths

Getting into gears

- Path P2A: out today, to complete by tomorrow

Before we start...

Where are we in CS1101S?

What is CS1101S? SICP JS, Source Academy, Studios, Missions, Piazza, Paths

Getting into gears

- Path P2A: out today, to complete by tomorrow
- Reflection R2: sheet out on LumiNUS, please prepare before your reflection session tomorrow

Before we start...

Where are we in CS1101S?

What is CS1101S? SICP JS, Source Academy, Studios, Missions, Piazza, Paths

Getting into gears

- Path P2A: out today, to complete by tomorrow
- Reflection R2: sheet out on LumiNUS, please prepare before your reflection session tomorrow
- Mission M2A: out today after lecture

Before we start...

Where are we in CS1101S?

What is CS1101S? SICP JS, Source Academy, Studios, Missions, Piazza, Paths

Getting into gears

- Path P2A: out today, to complete by tomorrow
- Reflection R2: sheet out on LumiNUS, please prepare before your reflection session tomorrow
- Mission M2A: out today after lecture
- Quest Q2A: out today evening

Before we start...

Where are we in CS1101S?

What is CS1101S? SICP JS, Source Academy, Studios, Missions, Piazza, Paths

Getting into gears

- Path P2A: out today, to complete by tomorrow
- Reflection R2: sheet out on LumiNUS, please prepare before your reflection session tomorrow
- Mission M2A: out today after lecture
- Quest Q2A: out today evening
- Mission M2B: out tomorrow

Before we start...

Where are we in CS1101S?

What is CS1101S? SICP JS, Source Academy, Studios, Missions, Piazza, Paths

Getting into gears

- Path P2A: out today, to complete by tomorrow
- Reflection R2: sheet out on LumiNUS, please prepare before your reflection session tomorrow
- Mission M2A: out today after lecture
- Quest Q2A: out today evening
- Mission M2B: out tomorrow
- Quest Q2B: out on Friday

Before we start...

Where are we in CS1101S?

What is CS1101S? SICP JS, Source Academy, Studios, Missions, Piazza, Paths

Getting into gears

- Path P2A: out today, to complete by tomorrow
- Reflection R2: sheet out on LumiNUS, please prepare before your reflection session tomorrow
- Mission M2A: out today after lecture
- Quest Q2A: out today evening
- Mission M2B: out tomorrow
- Quest Q2B: out on Friday

Registration for Reflections and Studios

Registration for Reflections and Studios

- Attend the session you have secured in ModReg

Registration for Reflections and Studios

- Attend the session you have secured in ModReg
- Find Zoom links and venues in the "CS1101S Lectures and Events" calendar

Registration for Reflections and Studios

- Attend the session you have secured in ModReg
- Find Zoom links and venues in the "CS1101S Lectures and Events" calendar
- If not secured any Reflection session yet, go to any ONLINE session **tomorrow**

Registration for Reflections and Studios

- Attend the session you have secured in ModReg
- Find Zoom links and venues in the "CS1101S Lectures and Events" calendar
- If not secured any Reflection session yet, go to any ONLINE session **tomorrow**
- Use ModReg appeals to secure a Reflection or Studio session for **Week 3 onwards**

Earning Experience Points

- Missions (basic homework): your primary source of XP

Earning Experience Points

- Missions (basic homework): your primary source of XP
- Quests (optional homework): supplement your XP

Earning Experience Points

- Missions (basic homework): your primary source of XP
- Quests (optional homework): supplement your XP
- Early Mission/Quest submission bonus: 75 XP within 3 days; bonus decays after that

Earning Experience Points

- Missions (basic homework): your primary source of XP
- Quests (optional homework): supplement your XP
- Early Mission/Quest submission bonus: 75 XP within 3 days; bonus decays after that
- Avengers can “unsubmit” your submission, within reasonable limits

Earning Experience Points

- Missions (basic homework): your primary source of XP
- Quests (optional homework): supplement your XP
- Early Mission/Quest submission bonus: 75 XP within 3 days; bonus decays after that
- Avengers can "unsubmit" your submission, within reasonable limits
- If you submit less than 2 days after release of Mission/Quest, Avengers will give feedback less than 4 days after release.

Earning Experience Points

- Missions (basic homework): your primary source of XP
- Quests (optional homework): supplement your XP
- Early Mission/Quest submission bonus: 75 XP within 3 days; bonus decays after that
- Avengers can "unsubmit" your submission, within reasonable limits
- If you submit less than 2 days after release of Mission/Quest, Avengers will give feedback less than 4 days after release.
- Path: XP for submission

Earning Experience Points

- Missions (basic homework): your primary source of XP
- Quests (optional homework): supplement your XP
- Early Mission/Quest submission bonus: 75 XP within 3 days; bonus decays after that
- Avengers can "unsubmit" your submission, within reasonable limits
- If you submit less than 2 days after release of Mission/Quest, Avengers will give feedback less than 4 days after release.
- Path: XP for submission
- Early path submission: 25 XP within 1 day; bonus decays after that

Earning Experience Points

- Missions (basic homework): your primary source of XP
- Quests (optional homework): supplement your XP
- Early Mission/Quest submission bonus: 75 XP within 3 days; bonus decays after that
- Avengers can "unsubmit" your submission, within reasonable limits
- If you submit less than 2 days after release of Mission/Quest, Avengers will give feedback less than 4 days after release.
- Path: XP for submission
- Early path submission: 25 XP within 1 day; bonus decays after that
- No "unsubmit" for paths

1. Substitution model, 1.1.3 and 1.1.5
2. Recursion and iteration, 1.2.1 and 1.2.2

1. Substitution model, 1.1.3 and 1.1.5
   - Evaluating combinations, 1.1.3
   - Evaluating function application, 1.1.5
   - Applicative vs normal order reduction, 1.1.5
2. Recursion and iteration, 1.2.1 and 1.2.2

Review: Evaluating arithmetic expressions, see 1.1.3

```
1 + 2 * 3 + 4
```

Review: Evaluating arithmetic expressions, see 1.1.3

```
1 + 2 * 3 + 4
```

Replace "eligible" subexpression with result

```
(1 + (2 * 3)) + 4
```

Review: Evaluating arithmetic expressions, see 1.1.3

```
1 + 2 * 3 + 4
```

Replace “eligible” subexpression with result

```
(1 + (2 * 3)) + 4

(1 + 6) + 4
```

Review: Evaluating arithmetic expressions, see 1.1.3

```
1 + 2 * 3 + 4
```

Replace “eligible” subexpression with result

```
(1 + (2 * 3)) + 4

(1 + 6) + 4

7 + 4
```

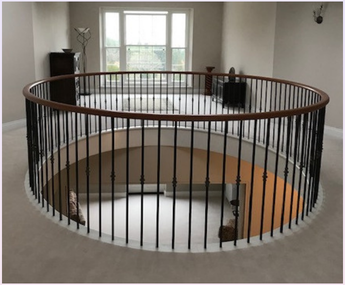
Review: Evaluating arithmetic expressions, see 1.1.3

```
1 + 2 * 3 + 4
```

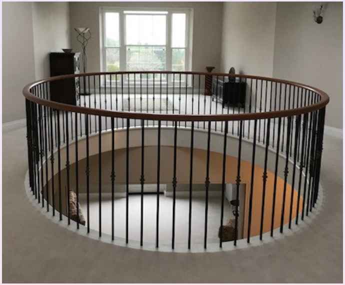
Replace "eligible" subexpression with result

```
(1 + (2 * 3)) + 4

(1 + 6) + 4

7 + 4

11
```

Example for substitution model 1.1.5



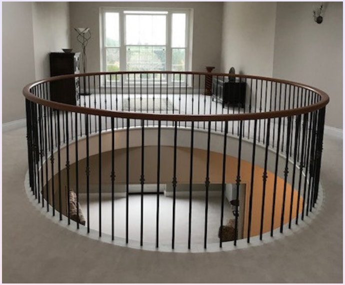
Example for substitution model 1.1.5



Problem

You need to find the cost of a circular handrail, knowing that the cost of the rail per meter is 199.50 dollars, and the radius of the circle is 2.1 meters.

Example for substitution model 1.1.5



Problem

You need to find the cost of a circular handrail, knowing that the cost of the rail per meter is 199.50 dollars, and the radius of the circle is 2.1 meters.

Solution

Evaluation of function application (1.1.5)

```
const cost_per_meter = 199.95;
function circumference(radius) {
    return 2 * math_PI * radius;}
function cost_of_circular_handrail(r) {
    return cost_per_meter * circumference(r);}
```

Evaluation of function application (1.1.5)

```
const cost_per_meter = 199.95;
function circumference(radius) {
    return 2 * math_PI * radius;}
function cost_of_circular_handrail(r) {
    return cost_per_meter * circumference(r);}

cost_of_circular_handrail(2.1)
```

Evaluation of function application (1.1.5)

```
const cost_per_meter = 199.95;
function circumference(radius) {
    return 2 * math_PI * radius;}
function cost_of_circular_handrail(r) {
    return cost_per_meter * circumference(r);}
```

```
cost_of_circular_handrail(2.1)
```

```
-> 199.5 * circumference(2.1)
```

Evaluation of function application (1.1.5)

```
const cost_per_meter = 199.95;
function circumference(radius) {
    return 2 * math_PI * radius;}
function cost_of_circular_handrail(r) {
    return cost_per_meter * circumference(r);}
```

```
cost_of_circular_handrail(2.1)
```

```
-> 199.5 * circumference(2.1)
```

```
-> 199.5 * ((2 * 3.141592) * 2.1)
```

Evaluation of function application (1.1.5)

```
const cost_per_meter = 199.95;
function circumference(radius) {
    return 2 * math_PI * radius;}
function cost_of_circular_handrail(r) {
    return cost_per_meter * circumference(r);}

cost_of_circular_handrail(2.1)

-> 199.5 * circumference(2.1)

-> 199.5 * ((2 * 3.141592) * 2.1)

-> 199.5 * (6.283185 * 2.1)
```

Evaluation of function application (1.1.5)

```
const cost_per_meter = 199.95;
function circumference(radius) {
    return 2 * math_PI * radius;}
function cost_of_circular_handrail(r) {
    return cost_per_meter * circumference(r);}
```

```
cost_of_circular_handrail(2.1)

-> 199.5 * circumference(2.1)

-> 199.5 * ((2 * 3.141592) * 2.1)

-> 199.5 * (6.283185 * 2.1)

-> 199.5 * 13.19468
```

Evaluation of function application (1.1.5)

```
const cost_per_meter = 199.95;
function circumference(radius) {
    return 2 * math_PI * radius;}
function cost_of_circular_handrail(r) {
    return cost_per_meter * circumference(r);}

cost_of_circular_handrail(2.1)

-> 199.5 * circumference(2.1)

-> 199.5 * ((2 * 3.141592) * 2.1)

-> 199.5 * (6.283185 * 2.1)

-> 199.5 * 13.19468

-> 2632.340
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}
```

```
f(5)
```

```
->  sum_of_sqs(5 + 1, 5 * 2)
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}
```

```
f(5)

->   sum_of_sqs(5 + 1, 5 * 2)

->   sum_of_sqs(6, 10)
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}
```

```
f(5)

->  sum_of_sqs(5 + 1, 5 * 2)

->  sum_of_sqs(6, 10)

->  sq(6) + sq(10)
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)

->  sum_of_sqs(5 + 1, 5 * 2)

->  sum_of_sqs(6, 10)

->  sq(6) + sq(10)

->  (6 * 6) + (10 * 10)
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)

->  sum_of_sqs(5 + 1, 5 * 2)

->  sum_of_sqs(6, 10)

->  sq(6) + sq(10)

->  (6 * 6) + (10 * 10)

⋮

->  136
```

Normal order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}
```

Normal order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)
```

Normal order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)

->  sum_of_sqs(5 + 1, 5 * 2)
```

Normal order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)

->  sum_of_sqs(5 + 1, 5 * 2)

->  sq(5 + 1) + sq(5 * 2)
```

Normal order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)

->  sum_of_sqs(5 + 1, 5 * 2)

->  sq(5 + 1) + sq(5 * 2)

->  ((5 + 1) * (5 + 1)) +
    ((5 * 2) * (5 * 2))
```

Normal order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)

->  sum_of_sqs(5 + 1, 5 * 2)

->  sq(5 + 1) + sq(5 * 2)

->  ((5 + 1) * (5 + 1)) +
    ((5 * 2) * (5 * 2))

⋮

->  136
```

Substitution model for applicative-order languages

Primitive expressions : take the value

Substitution model for applicative-order languages

Primitive expressions : take the value

Operator combinations : evaluate operands, apply operator

Substitution model for applicative-order languages

Primitive expressions : take the value

Operator combinations : evaluate operands, apply operator

Constant declaration : evaluate the value expression and replace the name by value

Substitution model for applicative-order languages

Primitive expressions : take the value

Operator combinations : evaluate operands, apply operator

Constant declaration : evaluate the value expression and replace the name by value

Function application : evaluate component expressions

Substitution model for applicative-order languages

Primitive expressions : take the value

Operator combinations : evaluate operands, apply operator

Constant declaration : evaluate the value expression and replace the name by value

Function application : evaluate component expressions

- if function is primitive: apply the primitive function

Substitution model for applicative-order languages

Primitive expressions : take the value

Operator combinations : evaluate operands, apply operator

Constant declaration : evaluate the value expression and replace the name by value

Function application : evaluate component expressions

- if function is primitive: apply the primitive function
- if function is compound: substitute argument values for parameters in body of declaration

Substitution model for runes

```
function turn_upside_down(rune) {
  return quarter_turn_right(
           quarter_turn_right(rune));
}
function quarter_turn_left(rune) {
  return quarter_turn_right(
           turn_upside_down(rune));
}
```

Substitution model for runes

```
function turn_upside_down(rune) {
  return quarter_turn_right(
           quarter_turn_right(rune));
}
function quarter_turn_left(rune) {
  return quarter_turn_right(
           turn_upside_down(rune));
}

quarter_turn_left(heart)
```

1. Substitution model, 1.1.3 and 1.1.5
2. Recursion and iteration, 1.2.1 and 1.2.2
   - `stackn` and `repeat_pattern`
   - Simple factorial and its recursive process, 1.2.1
   - Fibonacci function, 1.2.2

A new predeclared combination: `stack_frac`

`stack_frac(r, heart, sail)`

splits available bounded box such that
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)
```

Computers will follow our orders

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

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.

A Recipe

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

A Recipe

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

Recipe for recursion

A Recipe

```
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*

A Recipe

```
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 $n - 1$.

A Recipe

```
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?

Remember pre-defined 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?

Remember pre-defined 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);
```

Iterative process

With applicative order reduction,
pat function is applied *before*
the recursive call.
There is no deferred operation.

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

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

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

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

Translation into Source

Remember the base case

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

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
```

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
```

Notice the deferred operations.

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
```

Notice the deferred operations.
Recursive process.

A Closer look at performance

Dimensions of performance

A Closer look at performance

Dimensions of performance

- Time:
  how long does the program run

A Closer look at performance

Dimensions of performance

- Time:
  how long does the program run
- Space:
  how much memory do we need
  to run the program

Time for calculating n!

Number of operations

grows linearly proportional to n.

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

Space for calculating n!

Deferred operations: Number of "things to remember"

grows linearly proportional to n.

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

Can we write an iterative factorial?

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);
}
```

Fibonacci: Computing $F(n)$, first attempt

Fibonacci: Computing $F(n)$, first attempt

```
function fib(n) {
    return n <= 1
           ? n
           : fib(n - 1) + fib(n - 2);
}
```

Fibonacci: Computing $F(n)$, first attempt

```
function fib(n) {
    return n <= 1
           ? n
           : fib(n - 1) + fib(n - 2);
}
```

Time for exploring the tree

...grows with size of tree.

Fibonacci: Computing $F(n)$, first attempt

```
function fib(n) {
    return n <= 1
           ? n
           : fib(n - 1) + fib(n - 2);
}
```

Time for exploring the tree

...grows with size of tree.

Tree for `fib(n)` has $F(n+1)$ leaves.

Fibonacci: Computing $F(n)$, first attempt

```
function fib(n) {
    return n <= 1
           ? n
           : fib(n - 1) + fib(n - 2);
}
```

Time for exploring the tree

...grows with size of tree.

Tree for `fib(n)` has $F(n+1)$ leaves.

$$F(n) = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor, \text{where } \phi = \frac{1+\sqrt{5}}{2}$$

Fibonacci: Computing $F(n)$, first attempt

```
function fib(n) {
    return n <= 1
           ? n
           : fib(n - 1) + fib(n - 2);
}
```

Time for exploring the tree

...grows with size of tree.

Tree for `fib(n)` has $F(n+1)$ leaves.

$$F(n) = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor, \text{where } \phi = \frac{1+\sqrt{5}}{2}$$

Can we write an *efficient iterative* version?

Fibonacci: Computing $F(n)$, iterative solution

Fibonacci: Computing $F(n)$, iterative solution

```
function fib(n) {
    return fib_iter(1, 0, n);
}
function fib_iter(a, b, count) {
    return count === 0
           ? b
           : fib_iter(a + b, a, count - 1);
}
```

Fibonacci: Computing $F(n)$, iterative solution

```
function fib(n) {
    return fib_iter(1, 0, n);
}
function fib_iter(a, b, count) {
    return count === 0
           ? b
           : fib_iter(a + b, a, count - 1);
}
```

Time for exploring the tree

...grows proportional to n.

Summary

Summary

- Substitution allows us to trace the evaluation of expressions

Summary

- Substitution allows us to trace the evaluation of expressions
- Function applications are replaced by the return expression

Summary

- Substitution allows us to trace the evaluation of expressions
- Function applications are replaced by the return expression
- Examples for recursive solutions: `stackn`, `repeat_pattern`, `factorial`, `fib`

Summary

- Substitution allows us to trace the evaluation of expressions
- Function applications are replaced by the return expression
- Examples for recursive solutions: `stackn`, `repeat_pattern`, `factorial`, `fib`
- Recursive and iterative processes

Summary

- Substitution allows us to trace the evaluation of expressions
- Function applications are replaced by the return expression
- Examples for recursive solutions: `stackn`, `repeat_pattern`, `factorial`, `fib`
- Recursive and iterative processes
- Resources for computational processes:

Summary

- Substitution allows us to trace the evaluation of expressions
- Function applications are replaced by the return expression
- Examples for recursive solutions: `stackn`, `repeat_pattern`, `factorial`, `fib`
- Recursive and iterative processes
- Resources for computational processes: time and space

Summary

- Substitution allows us to trace the evaluation of expressions
- Function applications are replaced by the return expression
- Examples for recursive solutions: `stackn`, `repeat_pattern`, `factorial`, `fib`
- Recursive and iterative processes
- Resources for computational processes:
  time and space
  (more on this on Friday)

Summary

- Substitution allows us to trace the evaluation of expressions
- Function applications are replaced by the return expression
- Examples for recursive solutions: `stackn`, `repeat_pattern`, `factorial`, `fib`
- Recursive and iterative processes
- Resources for computational processes:
  time and space
  (more on this on Friday)
- Two versions for Fibonacci

Summary

- Substitution allows us to trace the evaluation of expressions
- Function applications are replaced by the return expression
- Examples for recursive solutions: `stackn`, `repeat_pattern`, `factorial`, `fib`
- Recursive and iterative processes
- Resources for computational processes:
  time and space
  (more on this on Friday)
- Two versions for Fibonacci:
  a bad one and a good one!