деревья

хорошо

битовая карта

Игорь Алексеенко

училка в HTML Academy

Многие современные проблемы фронтендеров были решены ещё в 80-х 😎📼💾☎️👾🚀

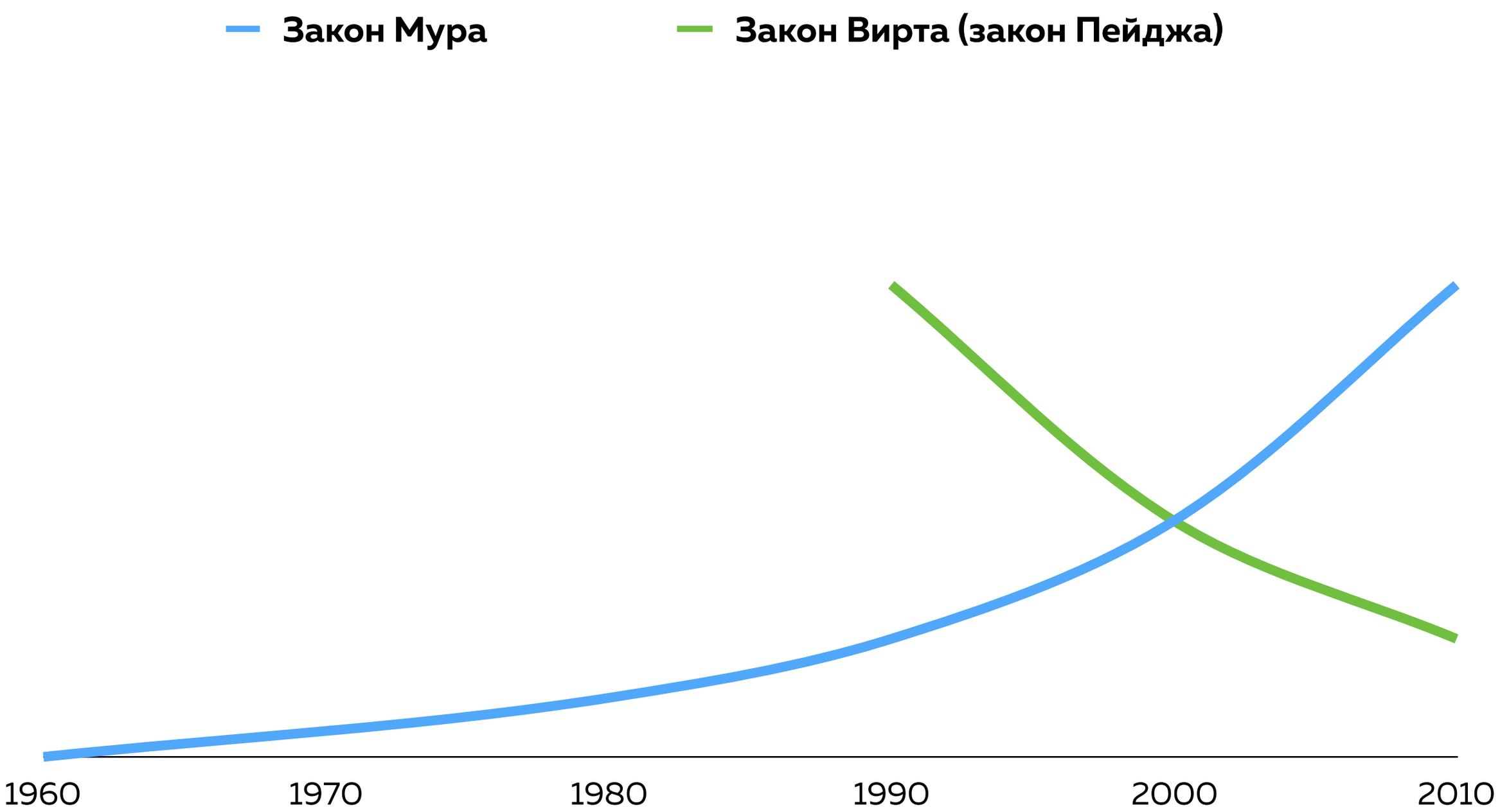
Просто мы об этом не знаем 👶 👶 👶

То, что мы разрабатываем, скорее, уже не сайты, а RIA

RIA —

(исландск. Rich Internet Application, насыщенное интернет-приложение), иногда IIA — Installable Internet Application — сайт, обладающий свойствами настольного приложения

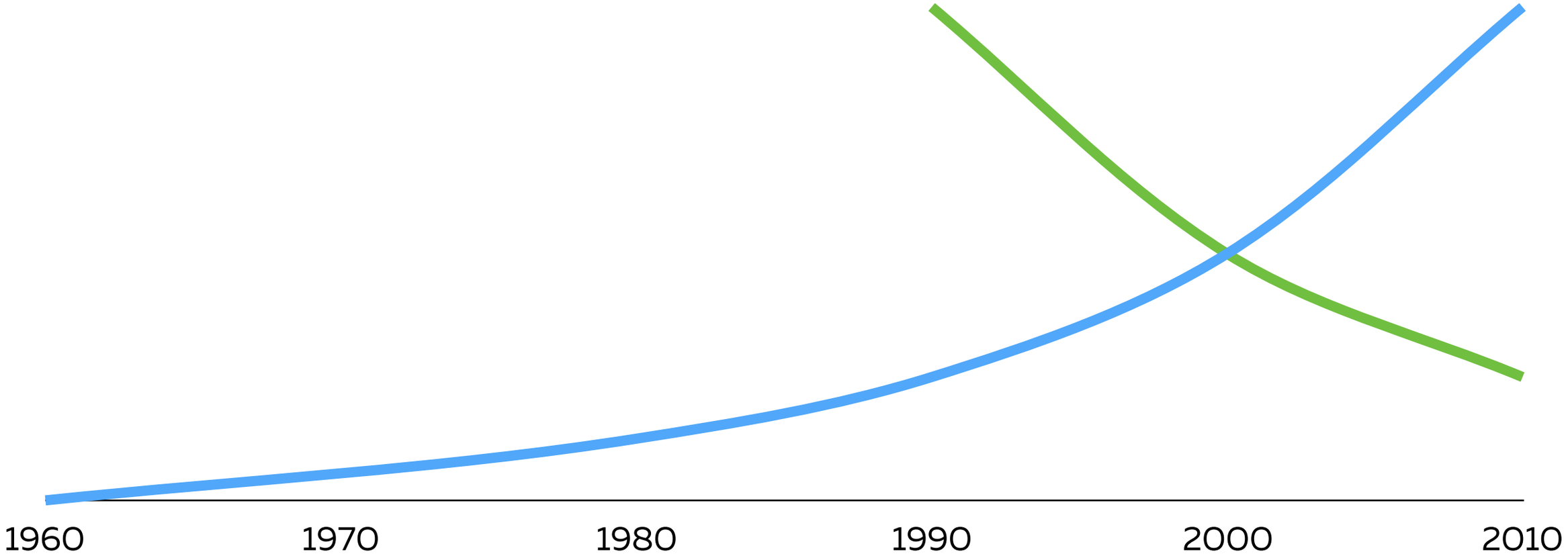
Законы развития ПО



Законы развития ПО

Закон Мура
1965
производительность компьютеров удваивается примерно каждые два года

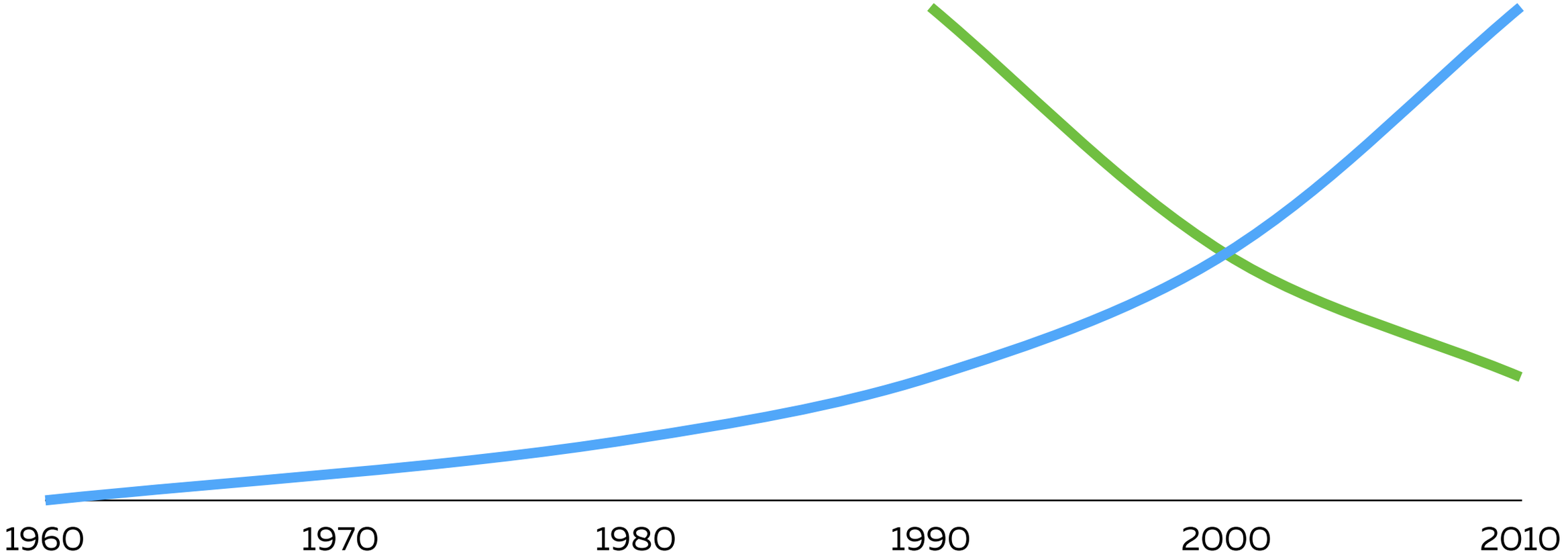
Закон Вирта (закон Пейджа)



Законы развития ПО

Закон Мура
1965
производительность компьютеров удваивается примерно каждые два года

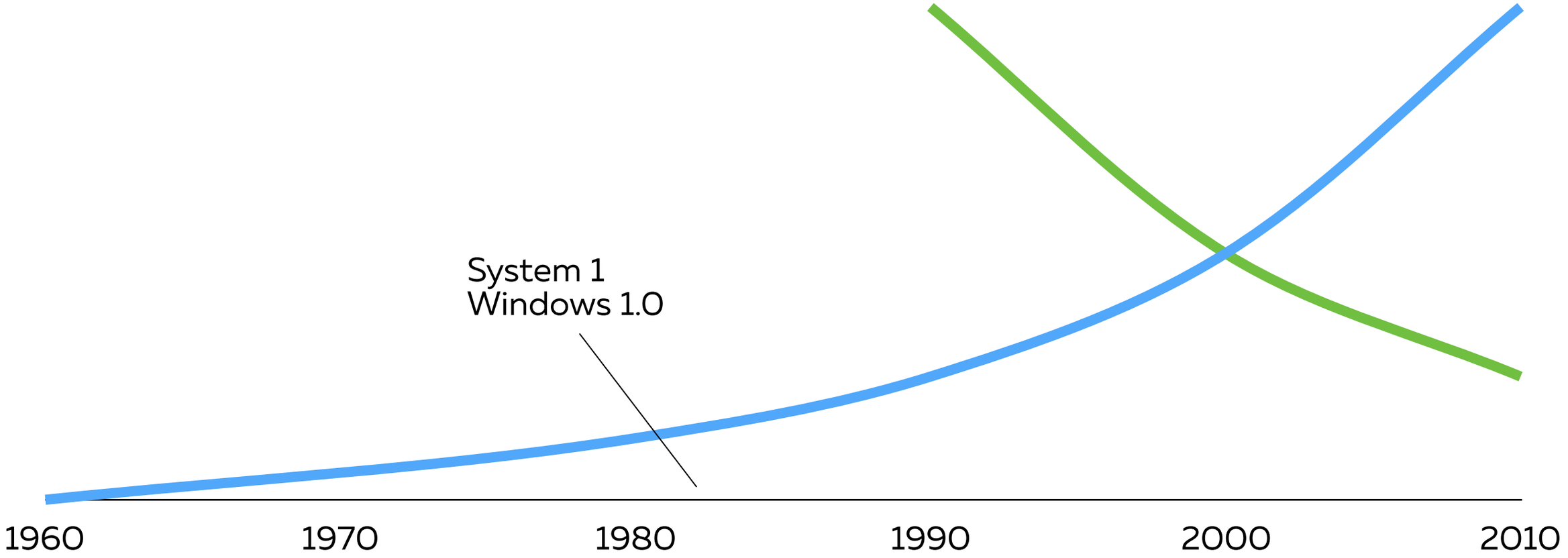
Закон Вирта (закон Пейджа)
~1995
медлительность программ возрастает быстрее чем производительность компьютеров



Законы развития ПО

Закон Мура
1965
производительность компьютеров удваивается примерно каждые два года

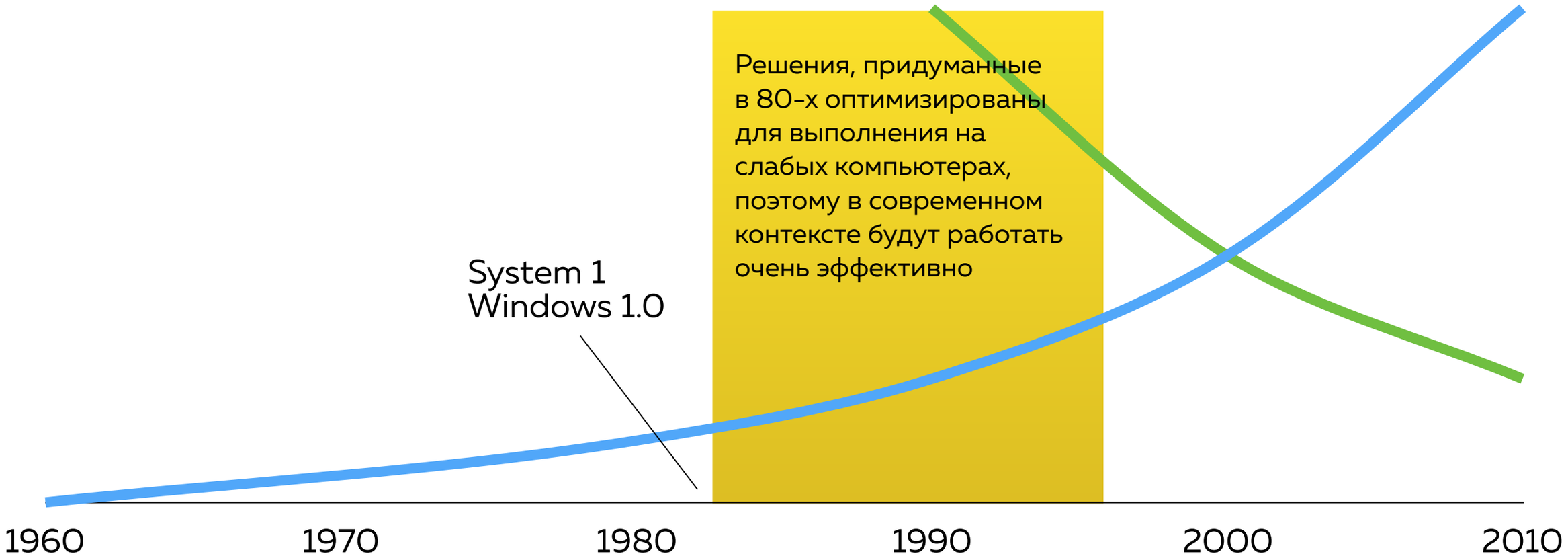
Закон Вирта (закон Пейджа)
~1995
медлительность программ возрастает быстрее чем производительность компьютеров



Законы развития ПО

Закон Мура
1965
производительность компьютеров удваивается примерно каждые два года

Закон Вирта (закон Пейджа)
~1995
медлительность программ возрастает быстрее чем производительность компьютеров



Компонента курильщика 🚬

Интерактивная демонстрация

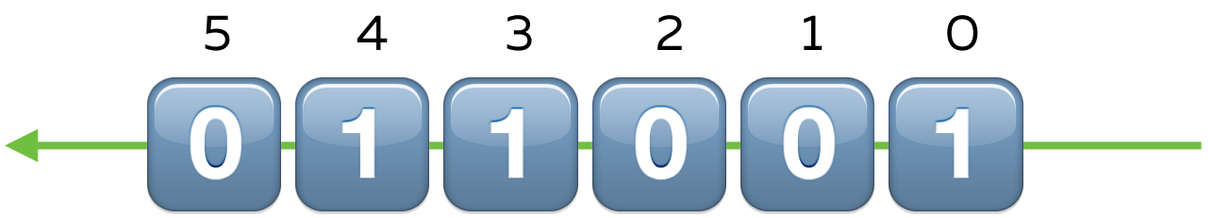
Битовые карты 

Двоичная запись

В памяти компьютера все числа хранятся в двоичном виде — как последовательность нулей и единиц. Разряды читаются справа налево. Нумерация разрядов начинается с нуля. Единица в разряде означает что итоговое число содержит двойку в степени номера разряда

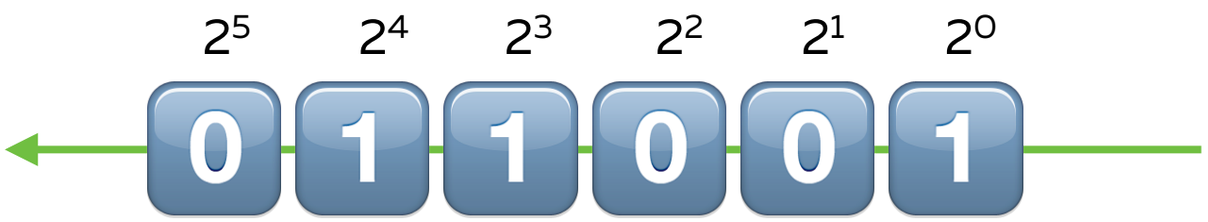
Двоичная запись

В памяти компьютера все числа хранятся в двоичном виде — как последовательность нулей и единиц. Разряды читаются справа налево. Нумерация разрядов начинается с нуля. Единица в разряде означает что итоговое число содержит двойку в степени номера разряда



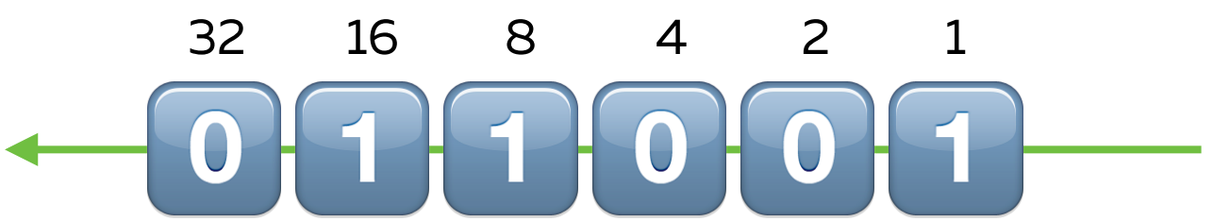
Двоичная запись

В памяти компьютера все числа хранятся в двоичном виде — как последовательность нулей и единиц. Разряды читаются справа налево. Нумерация разрядов начинается с нуля. Единица в разряде означает что итоговое число содержит двойку в степени номера разряда



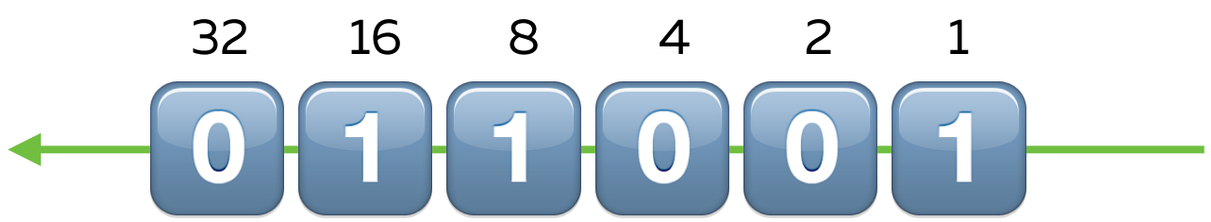
Двоичная запись

В памяти компьютера все числа хранятся в двоичном виде — как последовательность нулей и единиц. Разряды читаются справа налево. Нумерация разрядов начинается с нуля. Единица в разряде означает что итоговое число содержит двойку в степени номера разряда



Двоичная запись

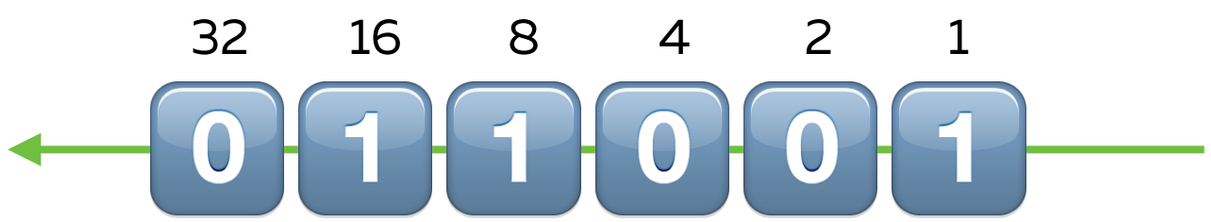
В памяти компьютера все числа хранятся в двоичном виде — как последовательность нулей и единиц. Разряды читаются справа налево. Нумерация разрядов начинается с нуля. Единица в разряде означает что итоговое число содержит двойку в степени номера разряда



0 + 16 + 8 + 0 + 0 + 1

Двоичная запись

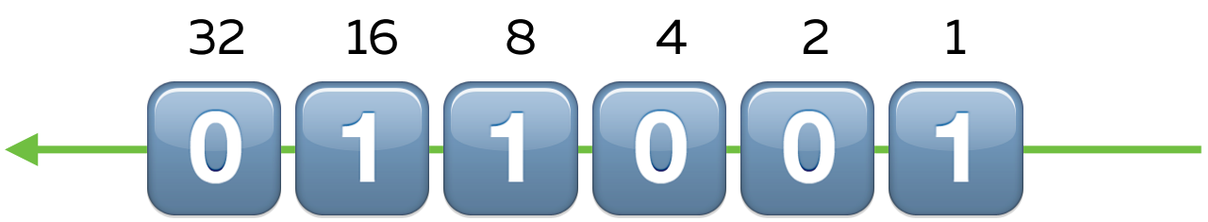
В памяти компьютера все числа хранятся в двоичном виде — как последовательность нулей и единиц. Разряды читаются справа налево. Нумерация разрядов начинается с нуля. Единица в разряде означает что итоговое число содержит двойку в степени номера разряда



0 + 16 + 8 + 0 + 0 + 1 =25

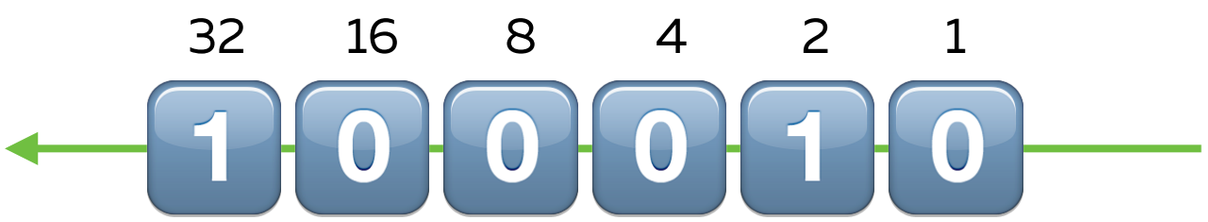
Двоичная запись

В памяти компьютера все числа хранятся в двоичном виде — как последовательность нулей и единиц. Разряды читаются справа налево. Нумерация разрядов начинается с нуля. Единица в разряде означает что итоговое число содержит двойку в степени номера разряда



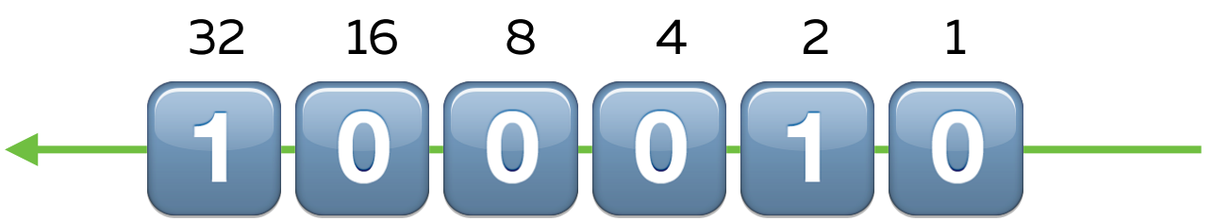
Двоичная запись

В памяти компьютера все числа хранятся в двоичном виде — как последовательность нулей и единиц. Разряды читаются справа налево. Нумерация разрядов начинается с нуля. Единица в разряде означает что итоговое число содержит двойку в степени номера разряда



Двоичная запись

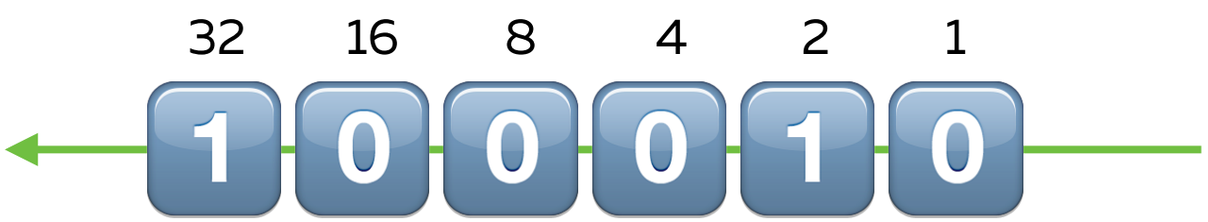
В памяти компьютера все числа хранятся в двоичном виде — как последовательность нулей и единиц. Разряды читаются справа налево. Нумерация разрядов начинается с нуля. Единица в разряде означает что итоговое число содержит двойку в степени номера разряда



32 + 0 + 0 + 0 + 2 + 0

Двоичная запись

В памяти компьютера все числа хранятся в двоичном виде — как последовательность нулей и единиц. Разряды читаются справа налево. Нумерация разрядов начинается с нуля. Единица в разряде означает что итоговое число содержит двойку в степени номера разряда



32 + 0 + 0 + 0 + 2 + 0 =34

Битовые карты —

(яп. — bitmap) они же битовые массивы (кор. — bitset, bitarray) последовательности бит, которые кодируют не степени двойки, а любую другую информацию. Хранятся в виде обычных чисел

Битовые карты

Битовой картой называется последовательность бит (нулей и единиц). В битовых картах может храниться не только число, но и любой другой набор данных

Битовые карты

Битовой картой называется последовательность бит (нулей и единиц). В битовых картах может храниться не только число, но и любой другой набор данных

- IP-адрес
  `192.168.1.1 / 255.255.255.0`

Битовые карты

Битовой картой называется последовательность бит (нулей и единиц). В битовых картах может храниться не только число, но и любой другой набор данных

- IP-адрес
  `192.168.1.1 / 255.255.255.0`

- цвет
  `#FACE8D`

Битовые карты

Битовой картой называется последовательность бит (нулей и единиц). В битовых картах может храниться не только число, но и любой другой набор данных

- IP-адрес
  `192.168.1.1 / 255.255.255.0`

- цвет
  `#FACE8D`

- права пользователя
  `chmod -R 777 .`

Битовые карты

Битовой картой называется последовательность бит (нулей и единиц). В битовых картах может храниться не только число, но и любой другой набор данных

- IP-адрес
  `192.168.1.1 / 255.255.255.0`

- цвет
  `#FACE8D`

- права пользователя
  `chmod -R 777 .`

- *карта уровня в игре* или пиксели изображения

Битовые карты

Битовой картой называется последовательность бит (нулей и единиц). В битовых картах может храниться не только число, но и любой другой набор данных

- IP-адрес
  `192.168.1.1 / 255.255.255.0`

- цвет
  `#FACE8D`

- права пользователя
  `chmod -R 777 .`

- *карта уровня в игре* или пиксели изображения

- состояние UX-компоненты

Компонента здорового человека

Интерактивная демонстрация. Кодирование состояний в битах

Битовые операции —

логические операции над цепочками битов, в которых биты выступают как значения `true` или `false`

Побитовое «или»

Складывает переданные значения и сохраняет включенные биты обоих операторов.
Идеально подходит для сложения нескольких состояний

```
  010
OR 100
```

Побитовое «или»

Складывает переданные значения и сохраняет включенные биты обоих операторов.
Идеально подходит для сложения нескольких состояний

```
  010
OR 100
---
  010
```

Побитовое «или»

Складывает переданные значения и сохраняет включенные биты обоих операторов.
Идеально подходит для сложения нескольких состояний

```
  010
OR 100
-----
  10
```

Побитовое «или»

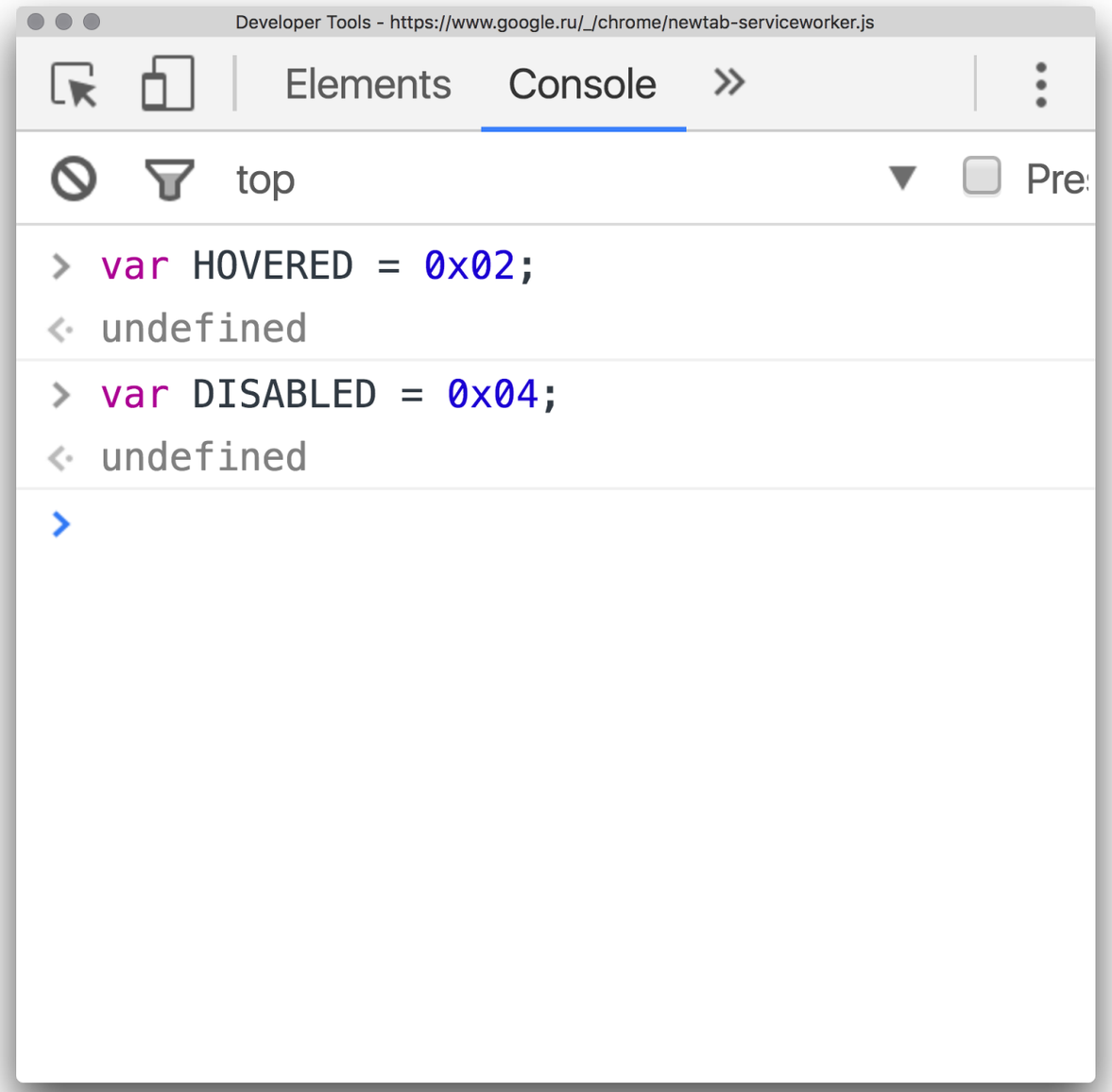
Складывает переданные значения и сохраняет включенные биты обоих операторов.
Идеально подходит для сложения нескольких состояний

```
  010
OR 100
  ---
  110
```

Побитовое «или»

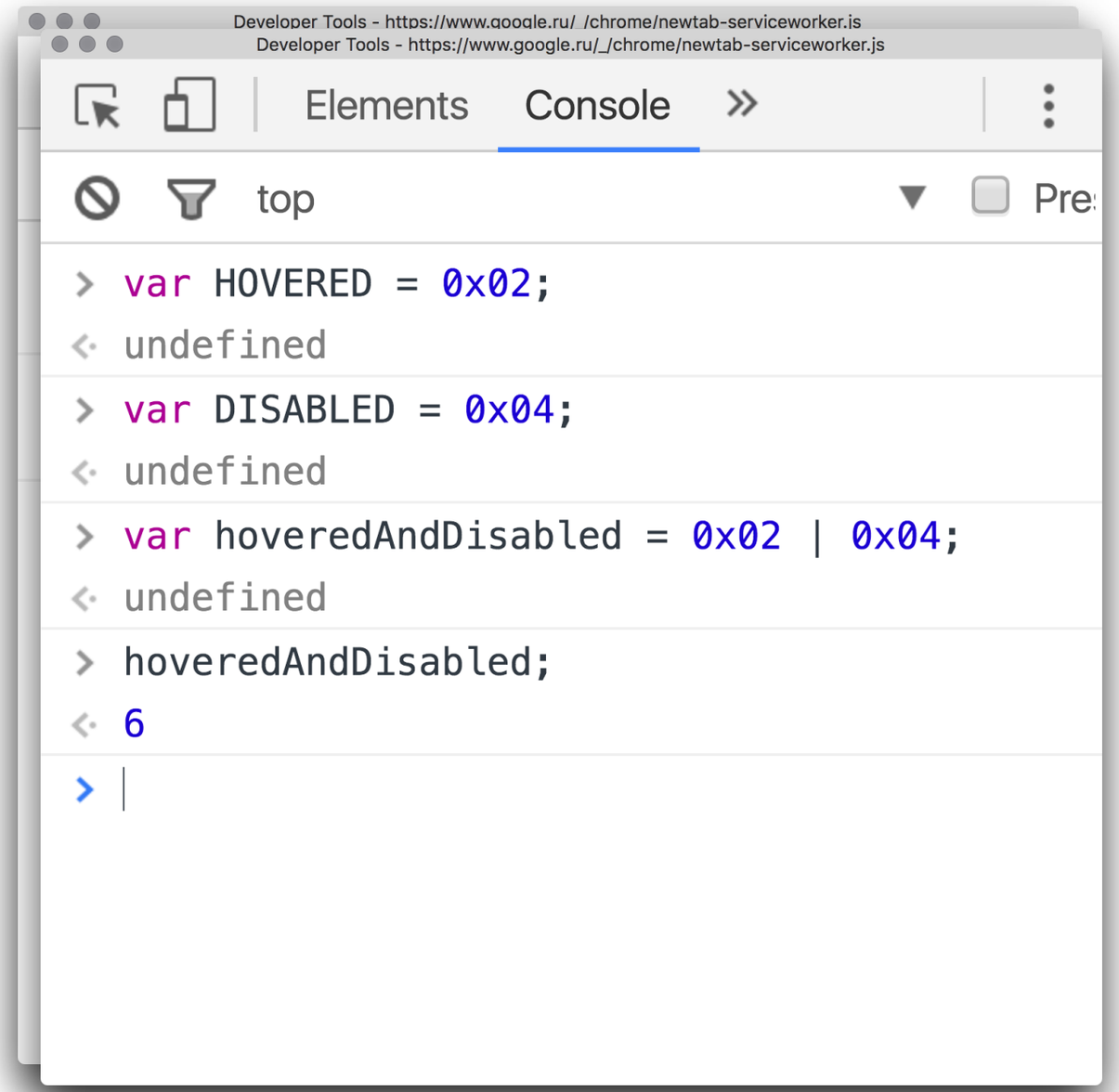
Складывает переданные значения и сохраняет включенные биты обоих операторов.

Идеально подходит для сложения нескольких состояний



Побитовое «или»

Складывает переданные значения и сохраняет включенные биты обоих операторов.
Идеально подходит для сложения нескольких состояний




```
  010
OR 100
-----
  110
```

Побитовое «и»

Сохраняет только те биты, которые были включены в обоих операндах. Идеально подходит для проверки состояния

```
  1010
AND 1100
------
```

Побитовое «и»

Сохраняет только те биты, которые были включены в обоих операндах. Идеально подходит для проверки состояния

```
  1010
AND 1100
----------
  1000
```

Побитовое «и»

Сохраняет только те биты, которые были включены в обоих операндах. Идеально подходит для проверки состояния

```
  1010
AND 1100
------
  0000
```

Побитовое «и»

Сохраняет только те биты, которые были включены в обоих операндах. Идеально подходит для проверки состояния

```
  1010
AND 1100
----------
  1000
```

Побитовое «и»

Сохраняет только те биты, которые были включены в обоих операндах. Идеально подходит для проверки состояния

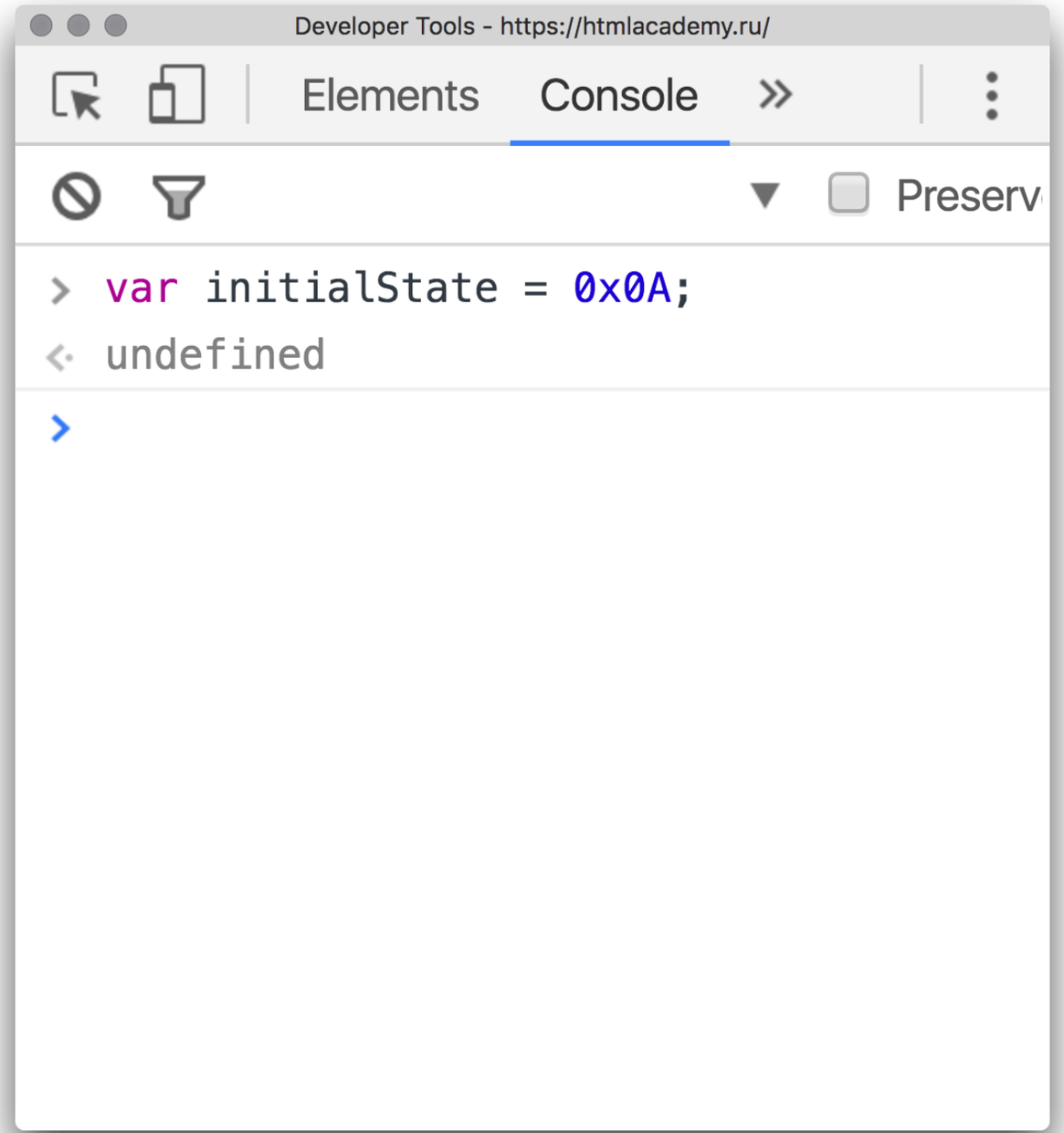
```
  1010
AND 1100
----------
  1000
```

Побитовое «и»

Сохраняет только те биты, которые были включены в обоих операндах. Идеально подходит для проверки состояния

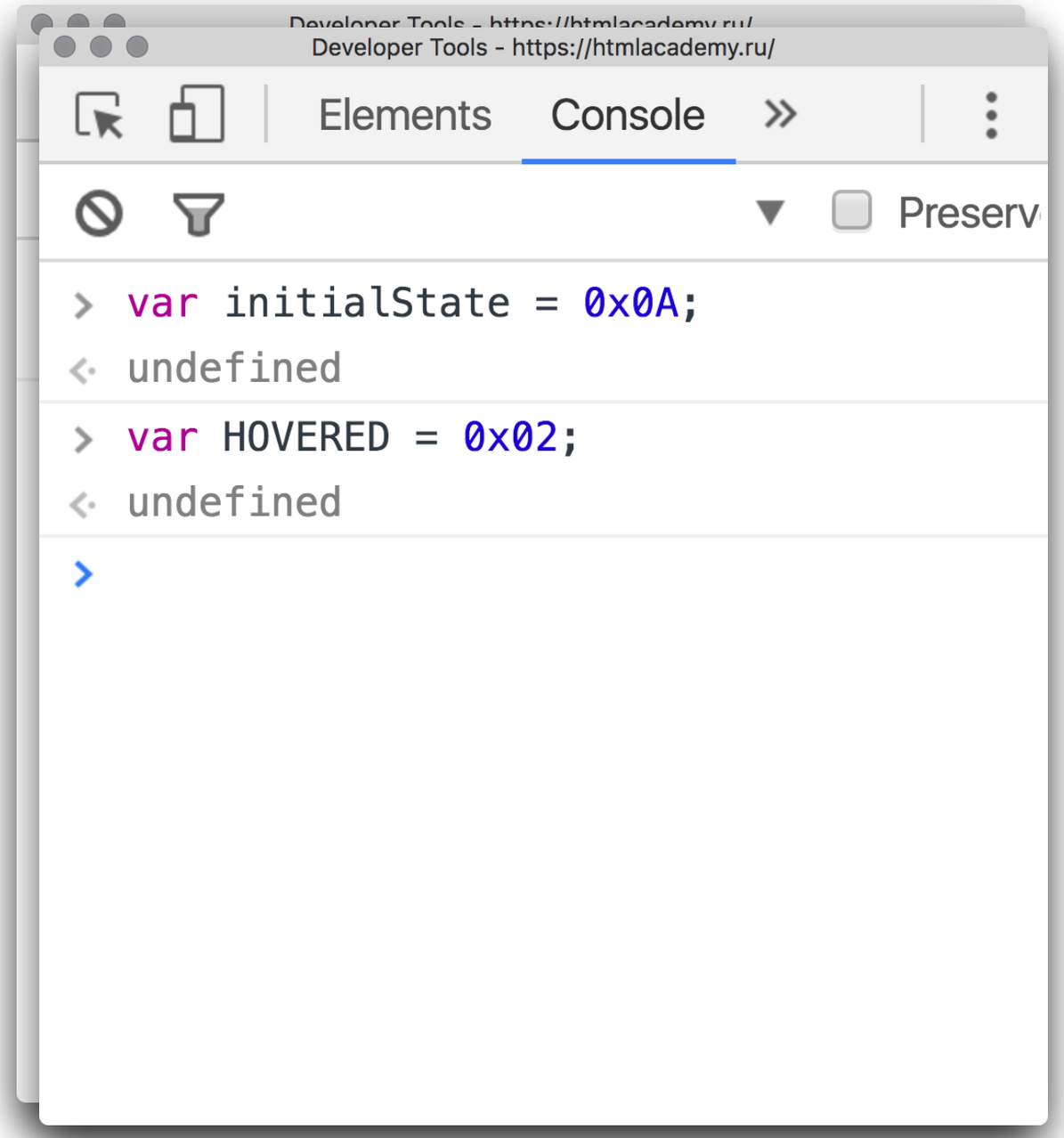
Побитовое «и»

Сохраняет только те биты, которые были включены в обоих операндах. Идеально подходит для проверки состояния



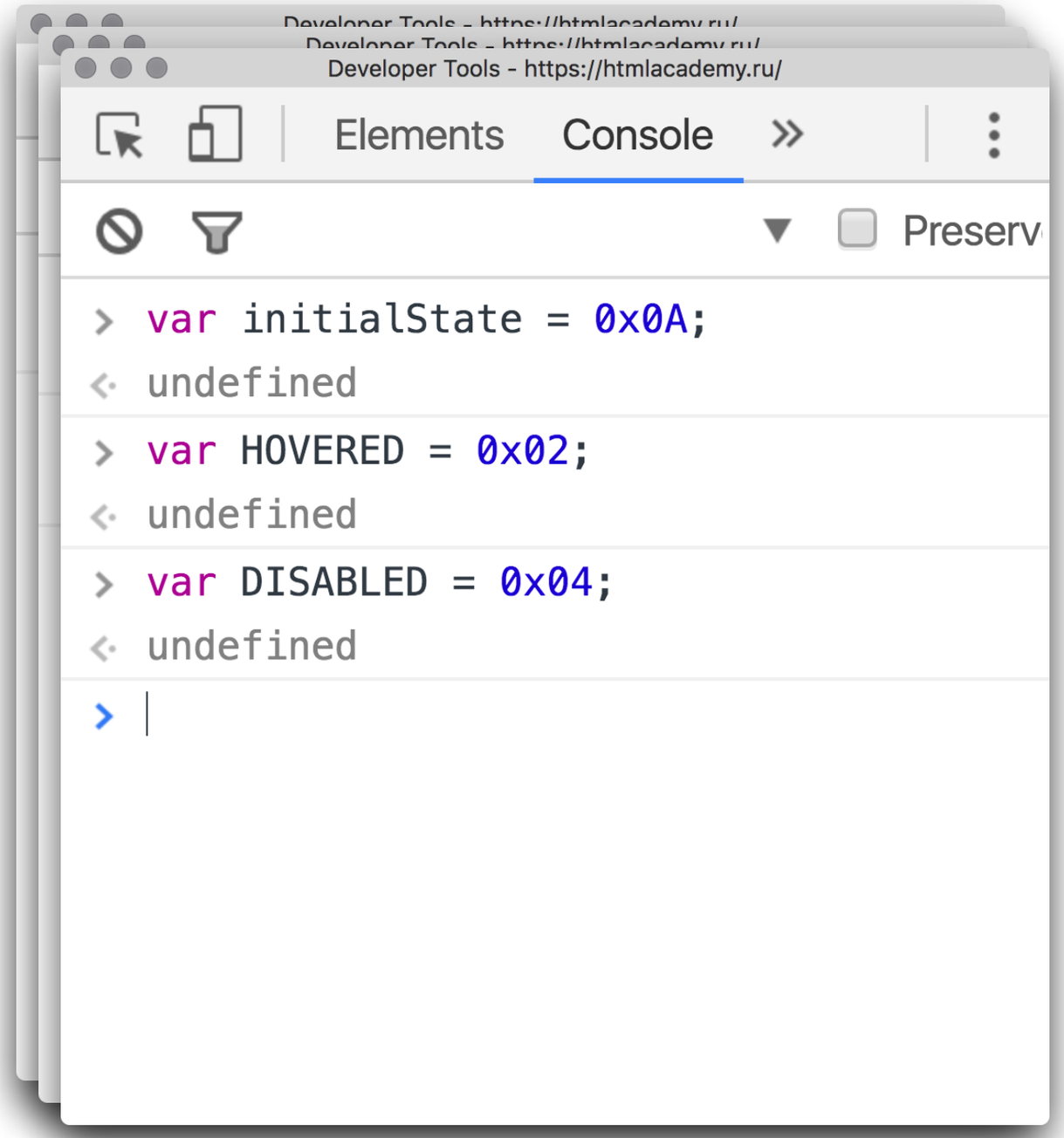
Побитовое «и»

Сохраняет только те биты, которые были включены в обоих операндах. Идеально подходит для проверки состояния



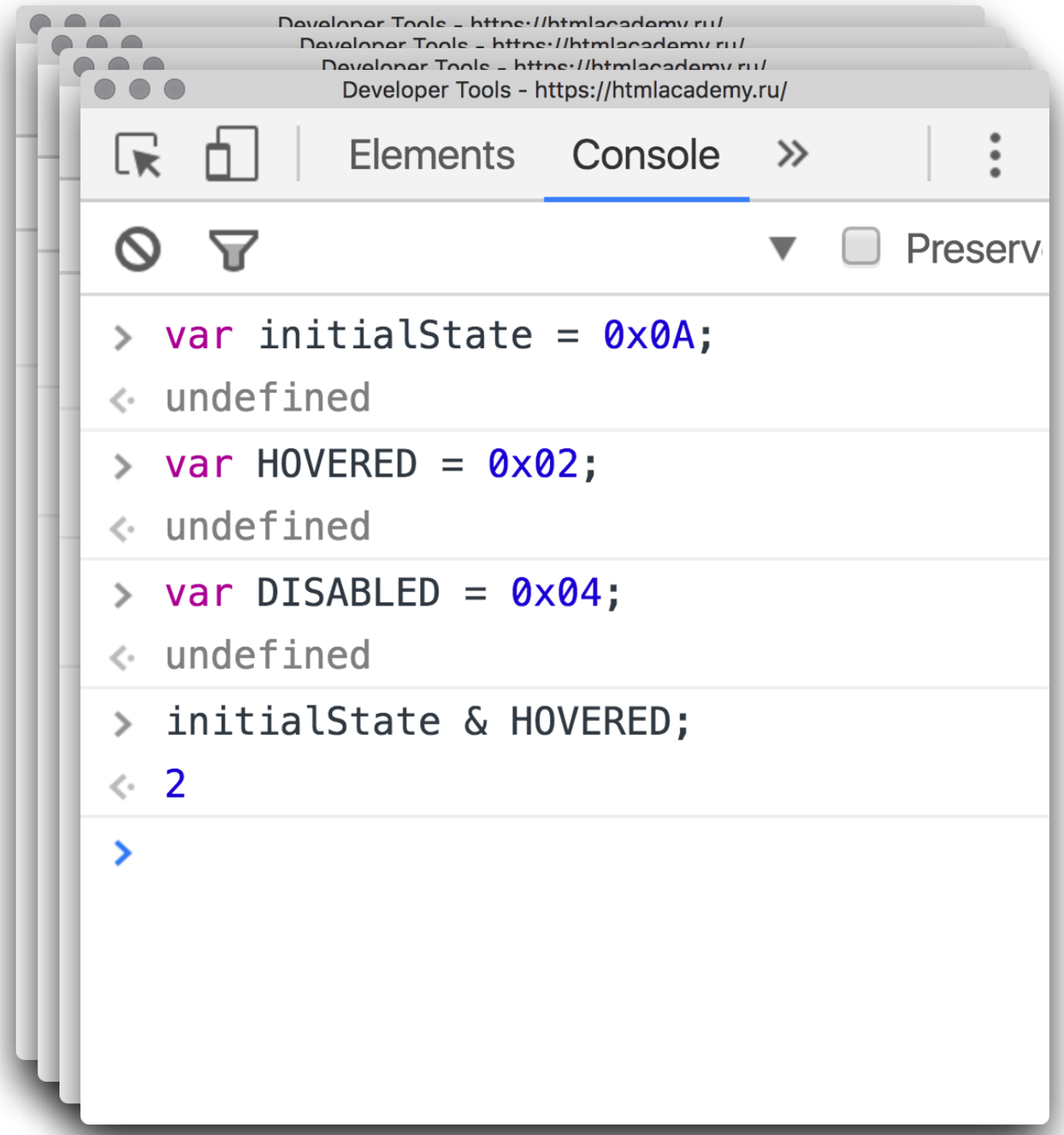
Побитовое «и»

Сохраняет только те биты, которые были включены в обоих операндах. Идеально подходит для проверки состояния



Побитовое «и»

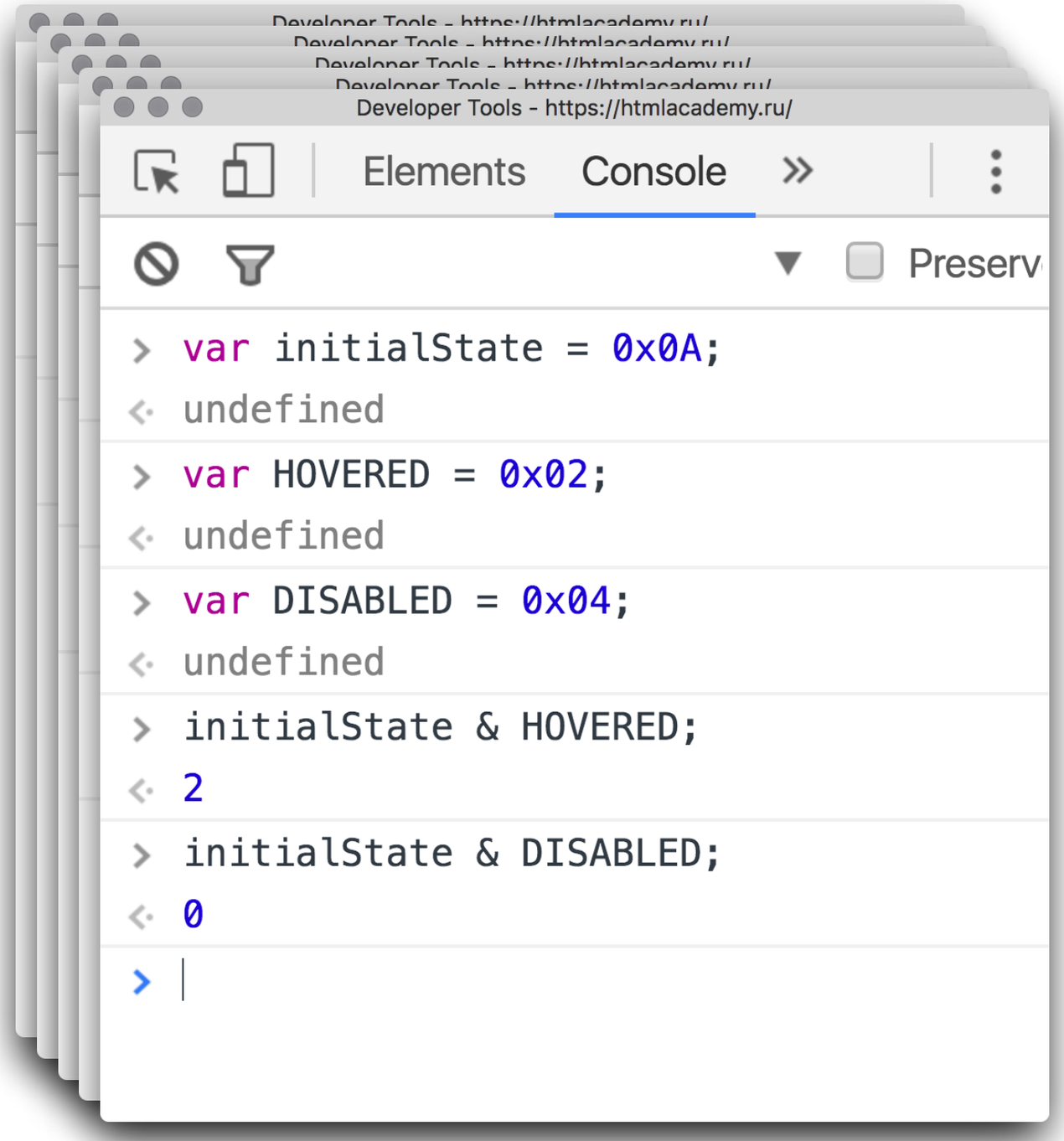
Сохраняет только те биты, которые были включены в обоих операндах. Идеально подходит для проверки состояния




```
  1010
AND 0010
  ----
  0010
```

Побитовое «и»

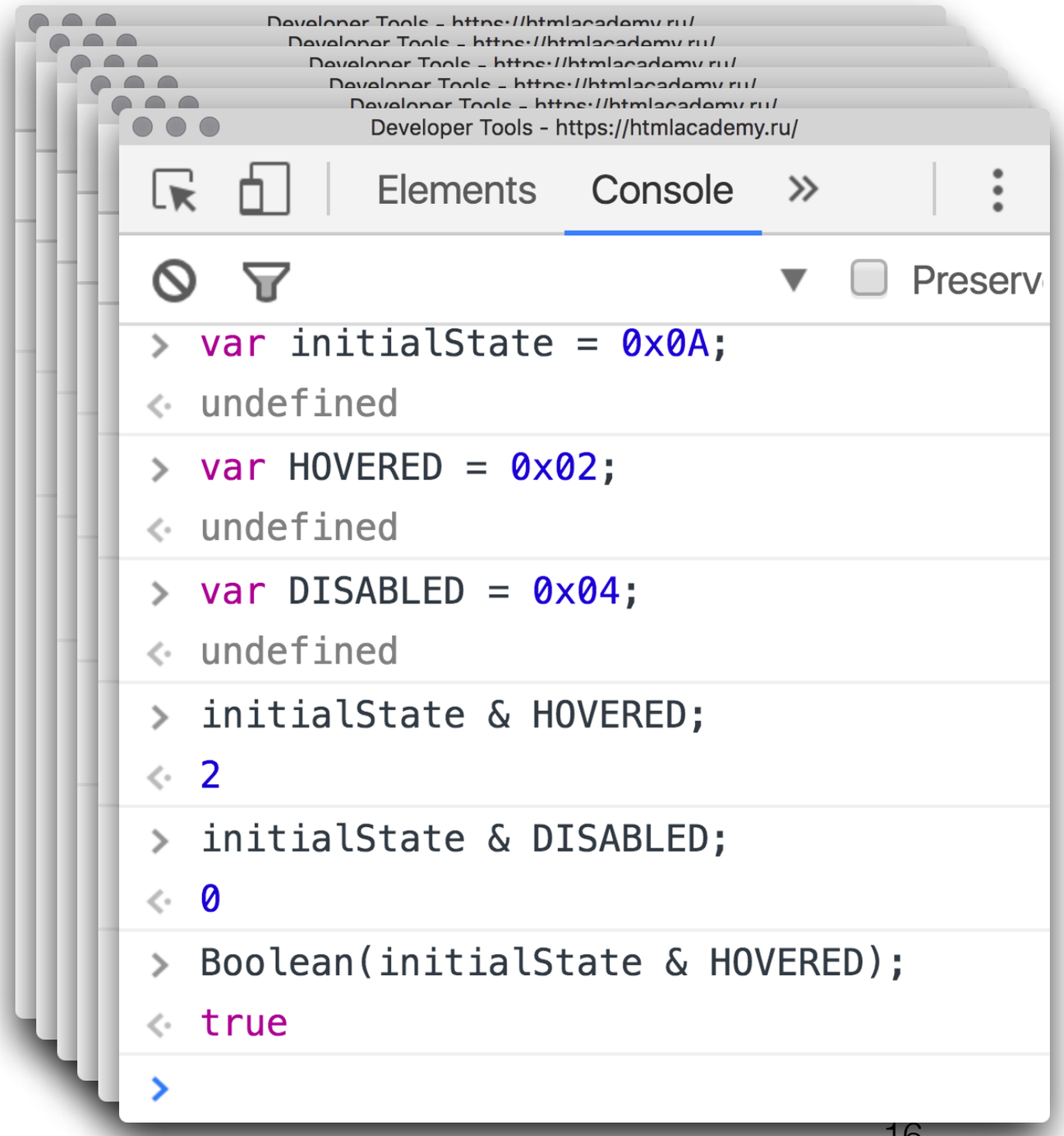
Сохраняет только те биты, которые были включены в обоих операндах. Идеально подходит для проверки состояния




```
  1010
AND 0100
  ----
  0000
```

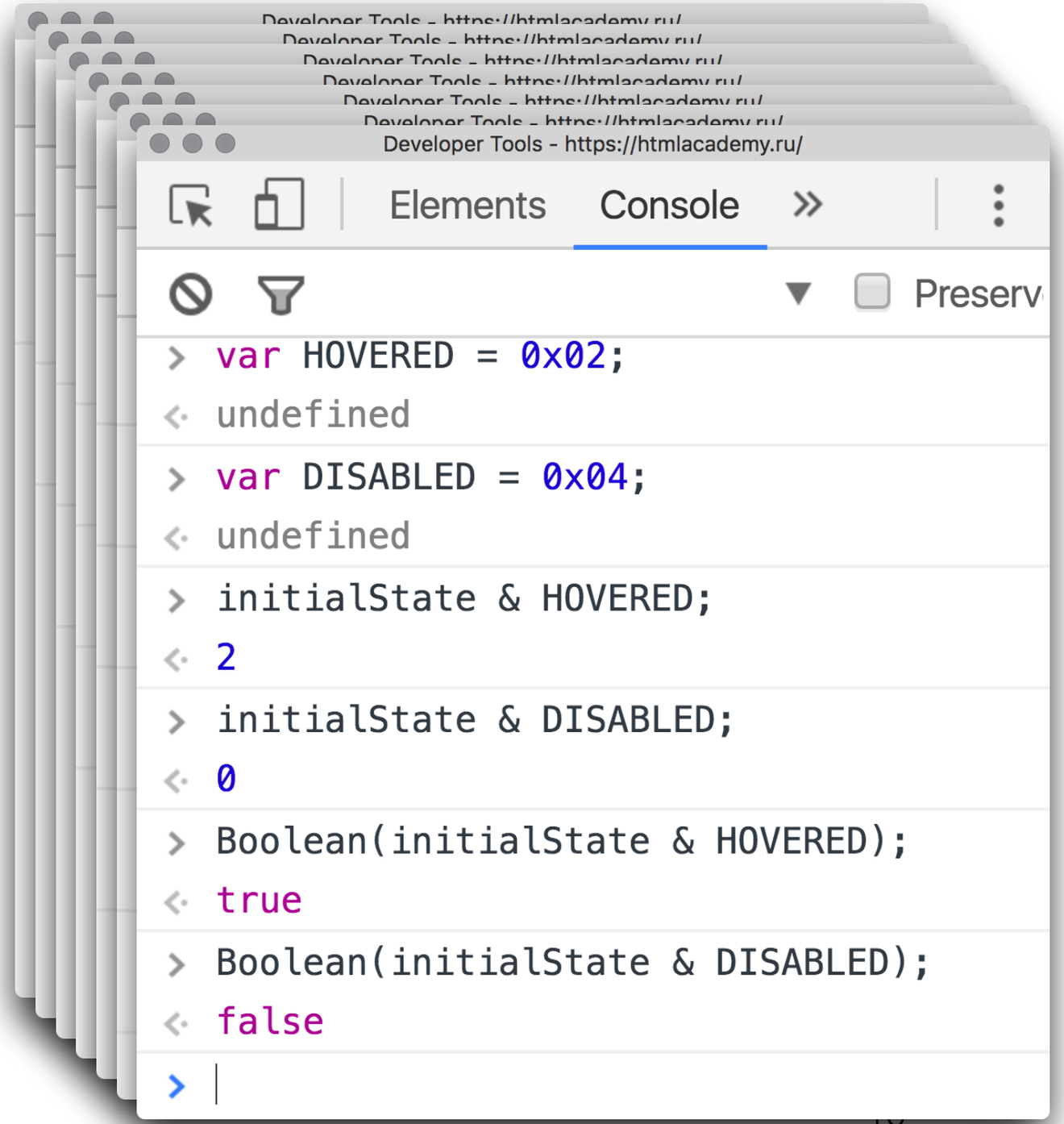
Побитовое «и»

Сохраняет только те биты, которые были включены в обоих операндах. Идеально подходит для проверки состояния



Побитовое «и»

Сохраняет только те биты, которые были включены в обоих операндах. Идеально подходит для проверки состояния



Компонента здорового человека

Интерактивная демонстрация. Продолжение

Декларативный стиль

Мы описываем не последовательность шагов (императивный стиль), а реакцию на изменение состояния

Полученный код содержит всего пять (одну) управляющих строк. Остальной код — статические словари. Вне зависимости от размеров словарей (количества состояний) размер управляющего кода меняться не будет

Использование состояний для описания поведения

code-and-magick

Это игра, где главного героя, которым вам предстоит управлять и изменять заклинаниями окружающий мир зовут Пендальф Синий. Вместе с ним вас ждет увлекательное приключение...

Описание поведения

В качестве значений в словарях можно использовать не только обычные значения,
но и функции. В этом случае можно описывать поведение каждого состояния независимо

Описание поведения

В качестве значений в словарях можно использовать не только обычные значения,
но и функции. В этом случае можно описывать поведение каждого состояния независимо

```
var Direction = {
  RIGHT: 0×01
};
```

```
var DirectionBehaviour = {
  0×01: function(character) {
    return Object.assign({
      left: character.left + character.hSpeed
    }, character);
  }
});
```

Описание поведения

В качестве значений в словарях можно использовать не только обычные значения,
но и функции. В этом случае можно описывать поведение каждого состояния независимо

```
var Direction = {
  RIGHT: 0×01,
  TOP: 0×02
};
```

```
var DirectionBehaviour = {
  0×01: function(character) {},
  0×02: function(character) {}
};
```

Даже Redux!

Даже Redux!

Можно использовать битовые маски как параметр `type` для управляющих объектов `action`. Так, за одно обращение к хранилищу можно делать несколько операций с данными

Даже Redux!

Можно использовать битовые маски как параметр `type` для управляющих объектов `action`. Так, за одно обращение к хранилищу можно делать несколько операций с данными

```
const ActionType = {
  UPDATE_1: 0×01,
  UPDATE_2: 0×02
};
```

```
store.dispatch({
  type: ActionType.UPDATE_1 | ActionType.UPDATE_2
});
```

И кроме этого есть ещё

И кроме этого есть ещё

- **исключающие или XOR, ^**
  используется для переключения состояния *(toggle)*

И кроме этого есть ещё

- **исключающие или XOR, ^**
  используется для переключения состояния *(toggle)*

- **побитовый сдвиг влево <<**

И кроме этого есть ещё

- **исключающие или XOR, ^**
  используется для переключения состояния *(toggle)*
- **побитовый сдвиг влево <<**
- **побитовый сдвиг вправо >>**

И кроме этого есть ещё

- **исключающие или XOR, ^**
  используется для переключения состояния *(toggle)*
- **побитовый сдвиг влево <<**
- **побитовый сдвиг вправо >>**
- **сдвиг вправо с заполнением нулями >>>**

И кроме этого есть ещё

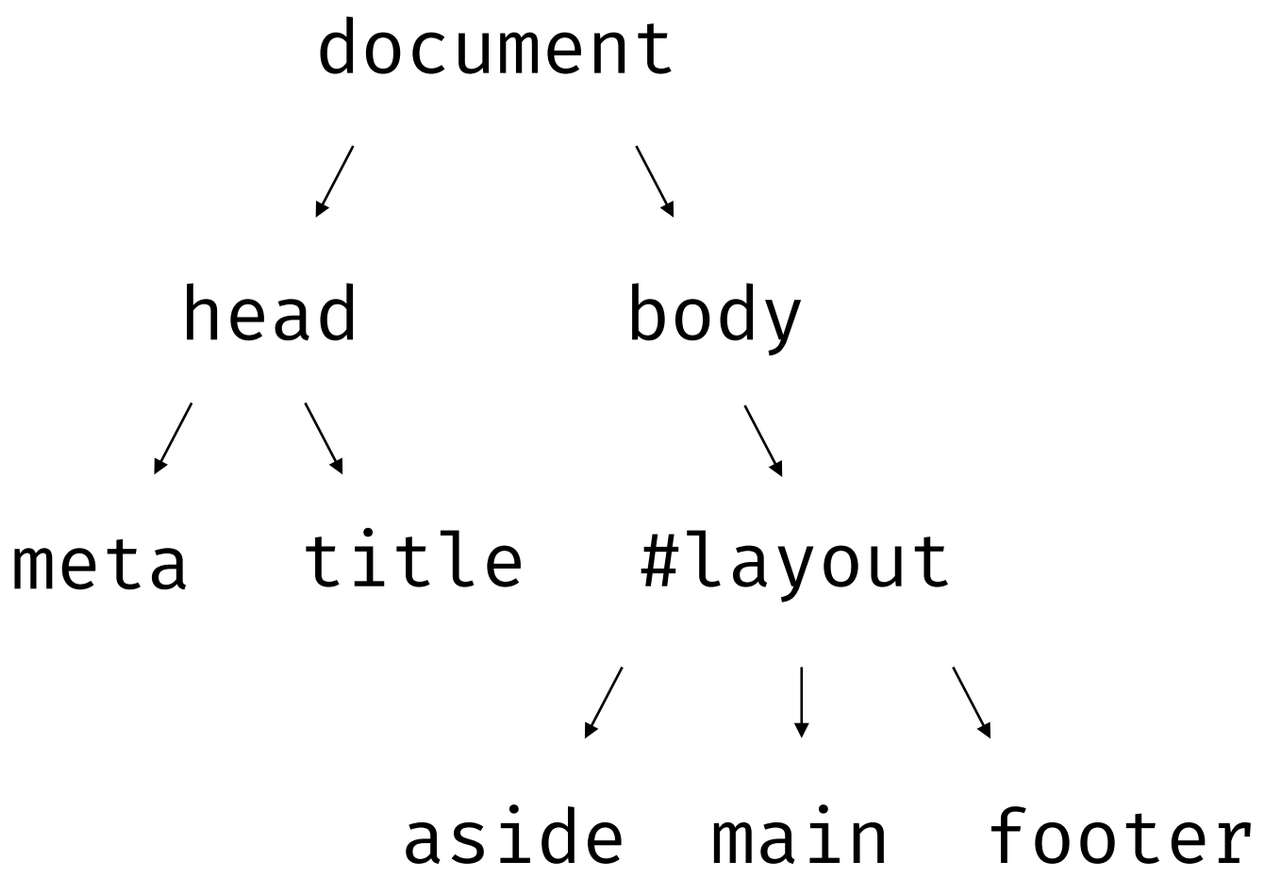
- **исключающие или XOR, ^**
  используется для переключения состояния *(toggle)*
- **побитовый сдвиг влево <<**
- **побитовый сдвиг вправо >>**
- **сдвиг вправо с заполнением нулями >>>**
- **битовые маски**
  когда одно состояние хранится в нескольких соседних битах используются маски, которые показывают единицами, в какие именно биты состояние записано *(IP-адрес и маска подсети)*

Зависимости в форме

 Деревья

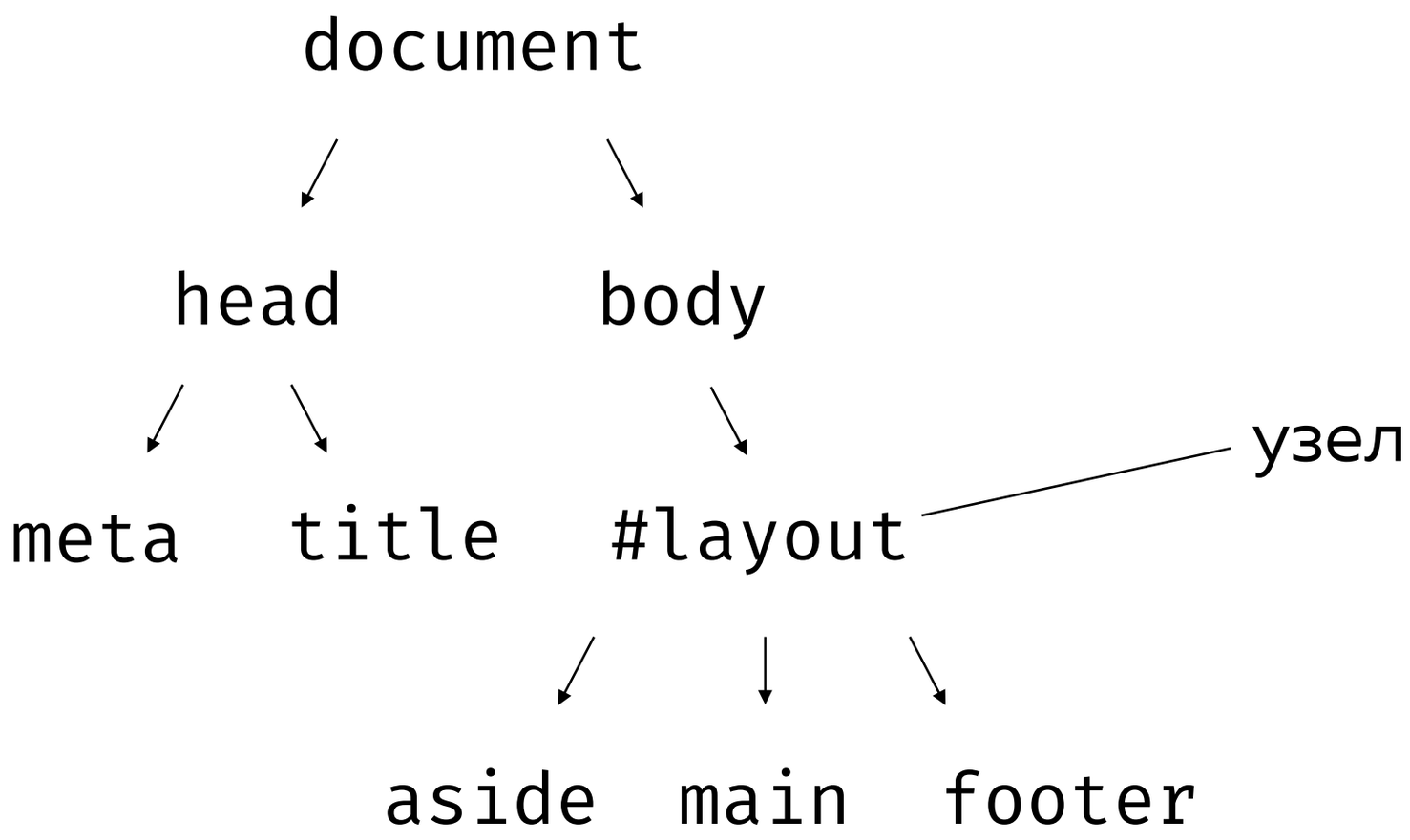
🌳 Деревья

Связанная структура данных, состоящая из вложенных друг в друга повторяющихся узлов. Самый наглядный пример, для фронтенд-разработчика — DOM-дерево



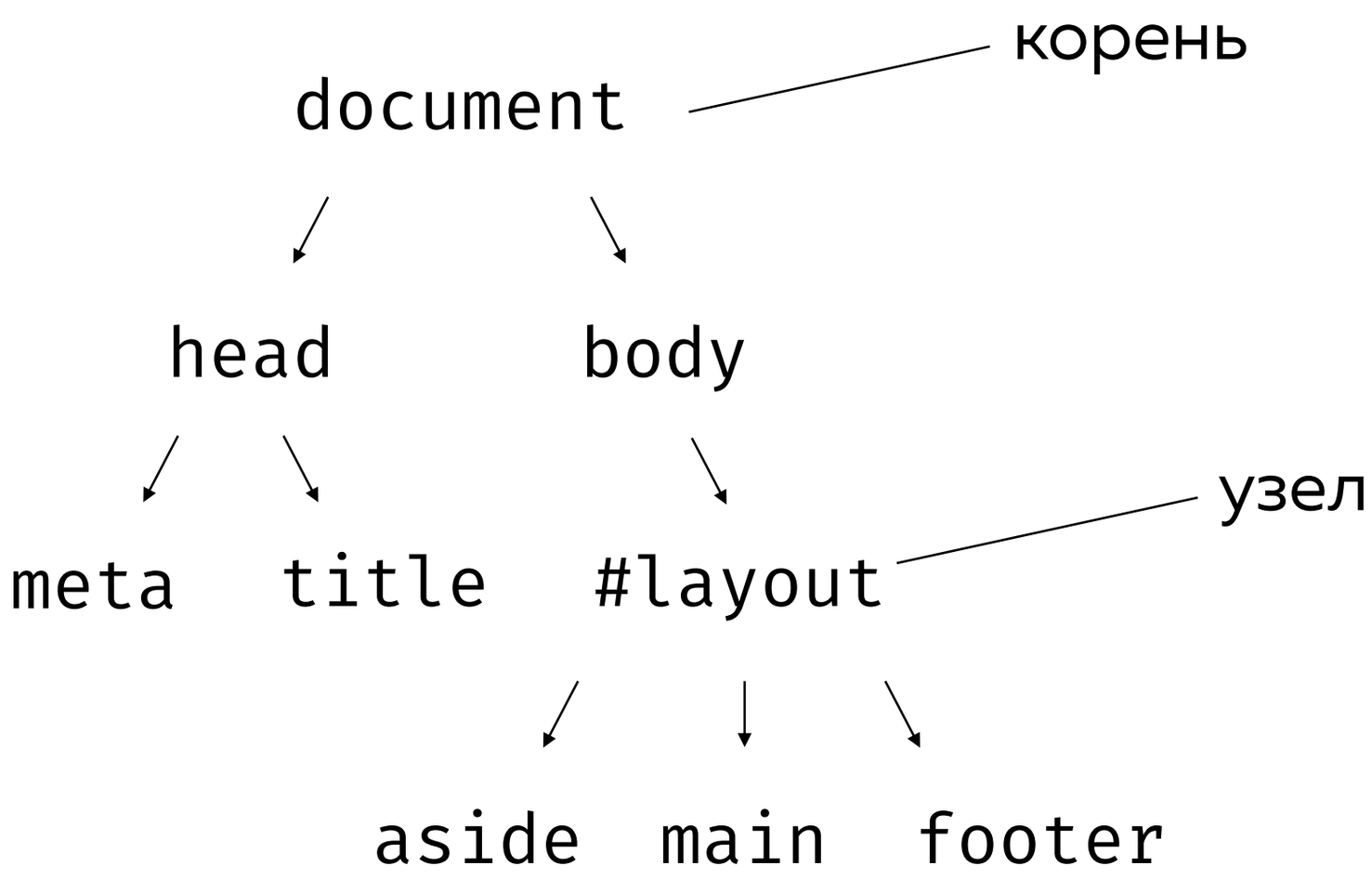
🌳 Деревья

Связанная структура данных, состоящая из вложенных друг в друга повторяющихся узлов. Самый наглядный пример, для фронтенд-разработчика — DOM-дерево



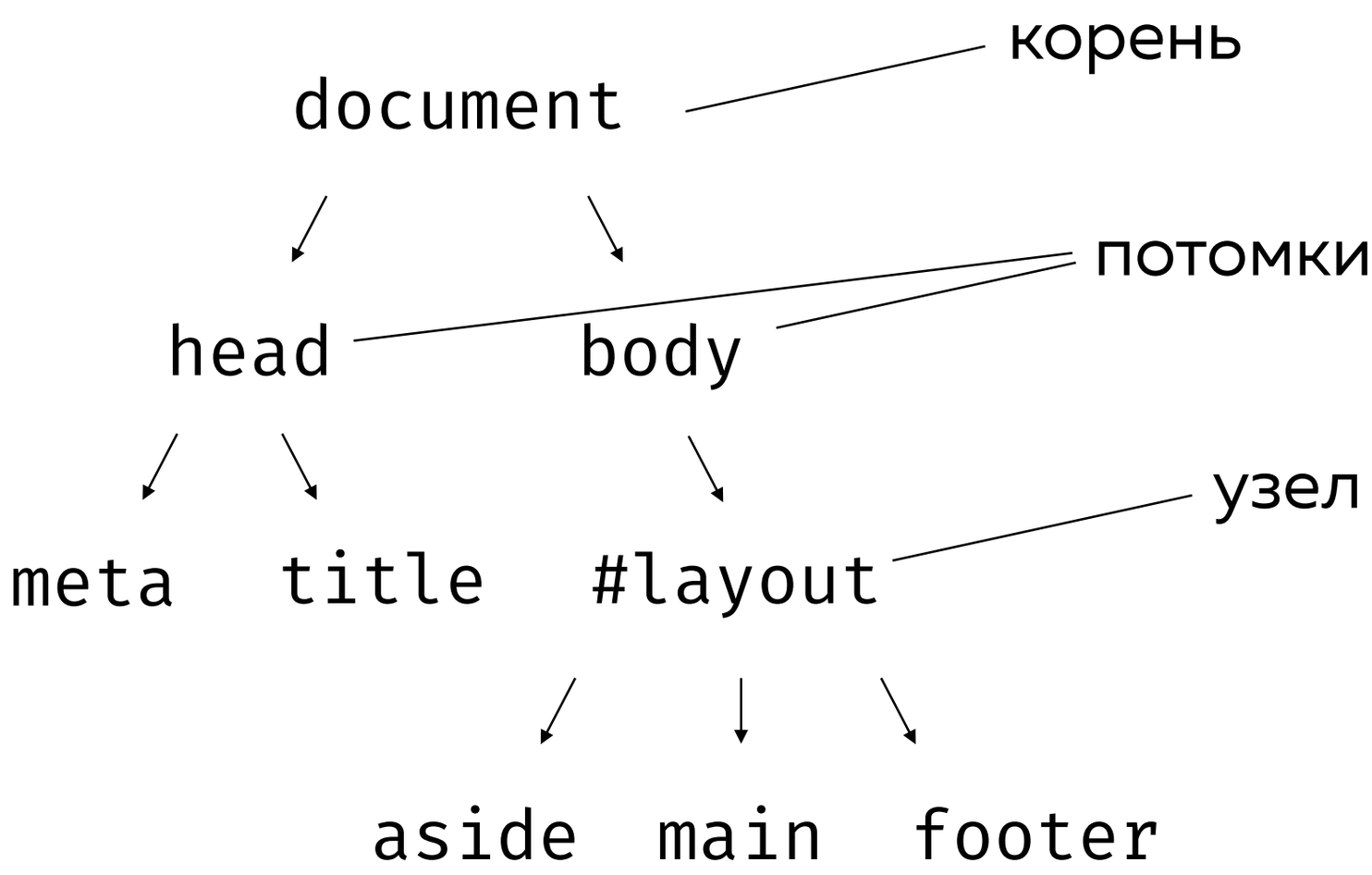
🌳 Деревья

Связанная структура данных, состоящая из вложенных друг в друга повторяющихся узлов. Самый наглядный пример, для фронтенд-разработчика — DOM-дерево



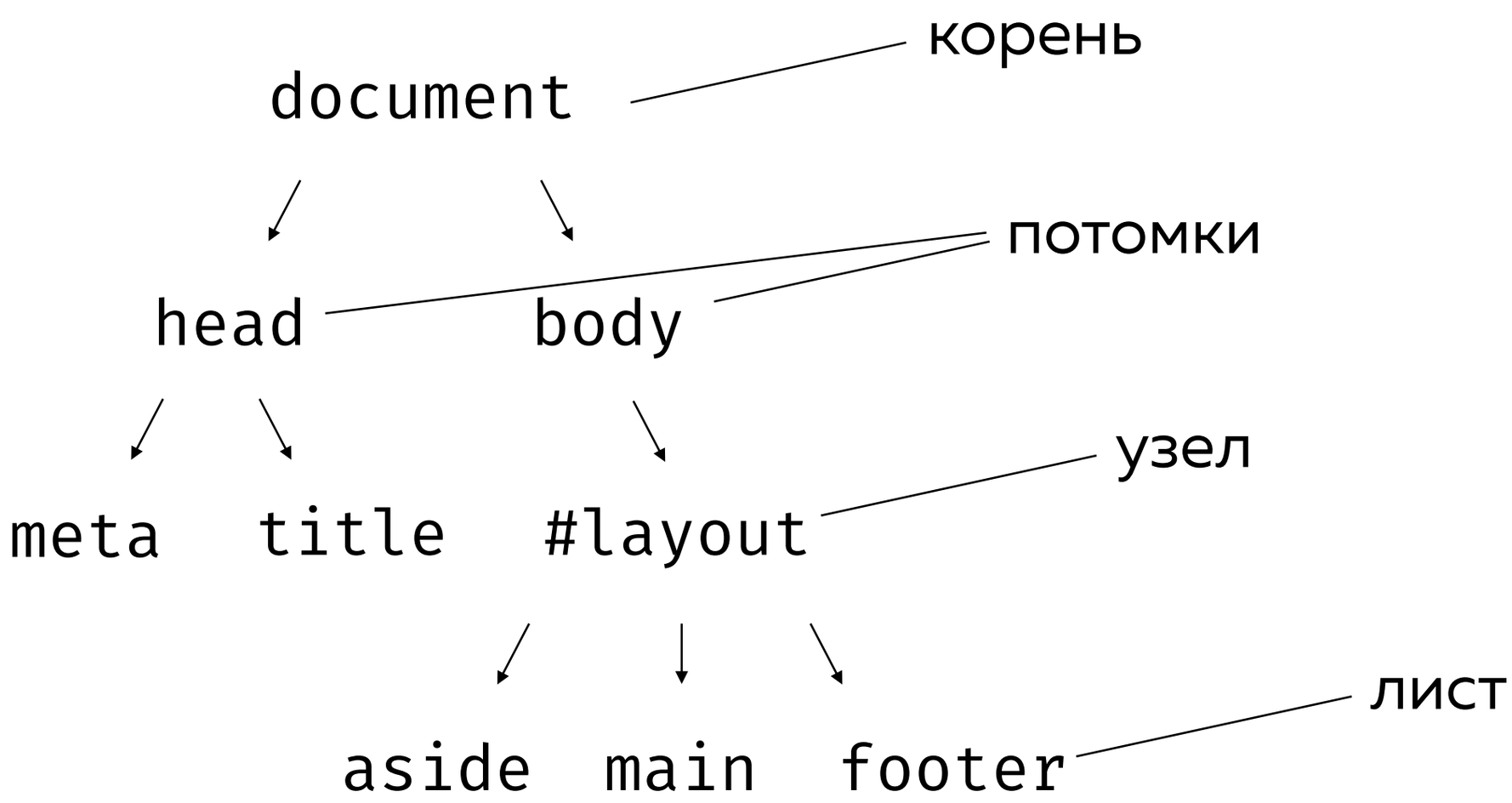
Деревья

Связанная структура данных, состоящая из вложенных друг в друга повторяющихся узлов. Самый наглядный пример, для фронтенд-разработчика — DOM-дерево



Деревья

Связанная структура данных, состоящая из вложенных друг в друга повторяющихся узлов. Самый наглядный пример, для фронтенд-разработчика — DOM-дерево



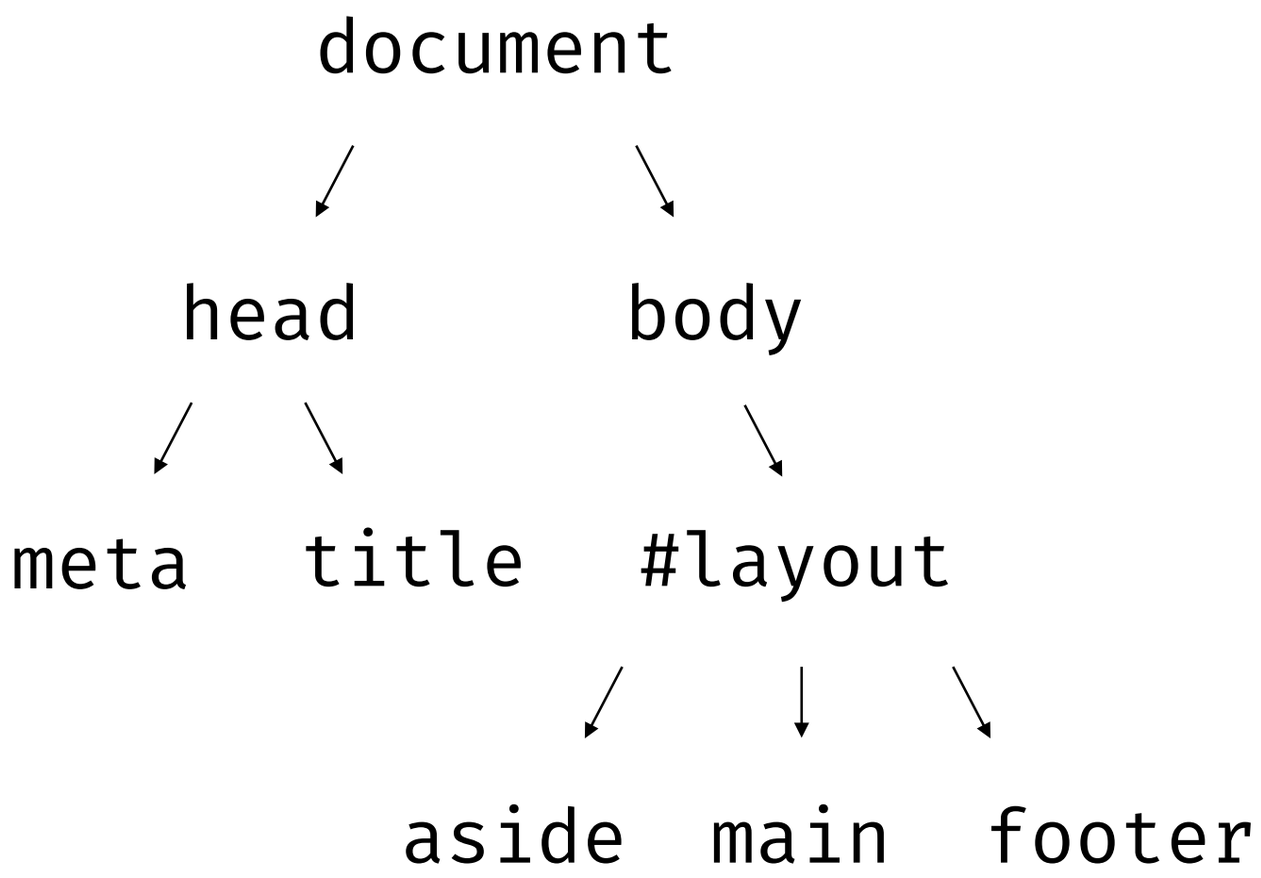
Деревья

Интерактивная демонстрация. Использование деревьев для описания зависимостей в форме

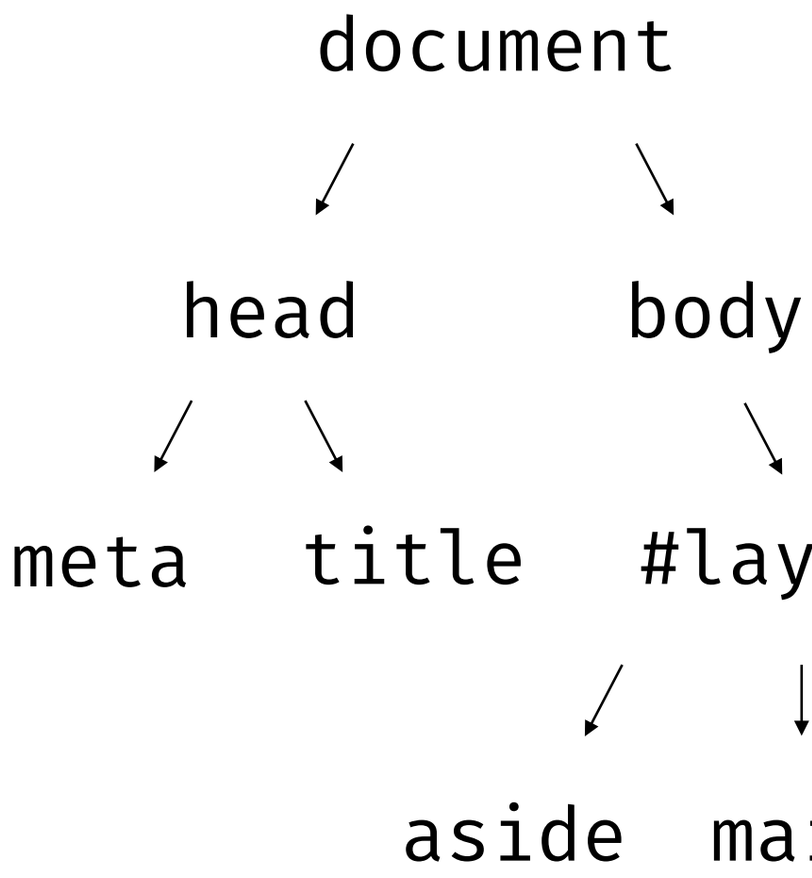
🚶 🌳 Обход дерева

Какими способами можно обходить дерево и на что может повлиять изменение способа обхода

Деревья

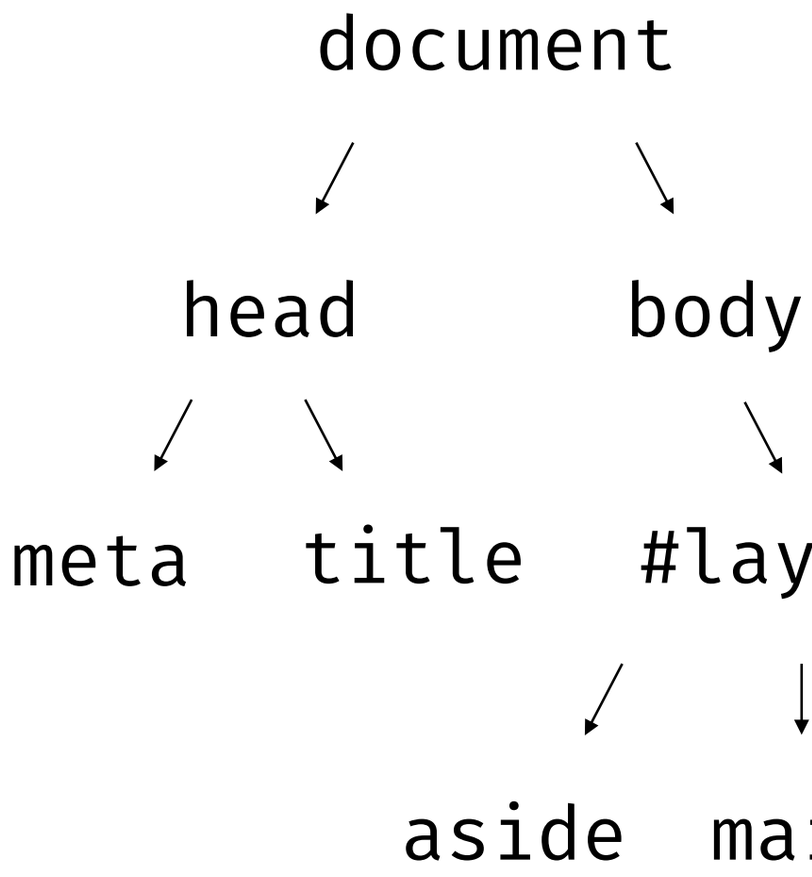


Деревья



Существует несколько способов перебора элементов дерева:

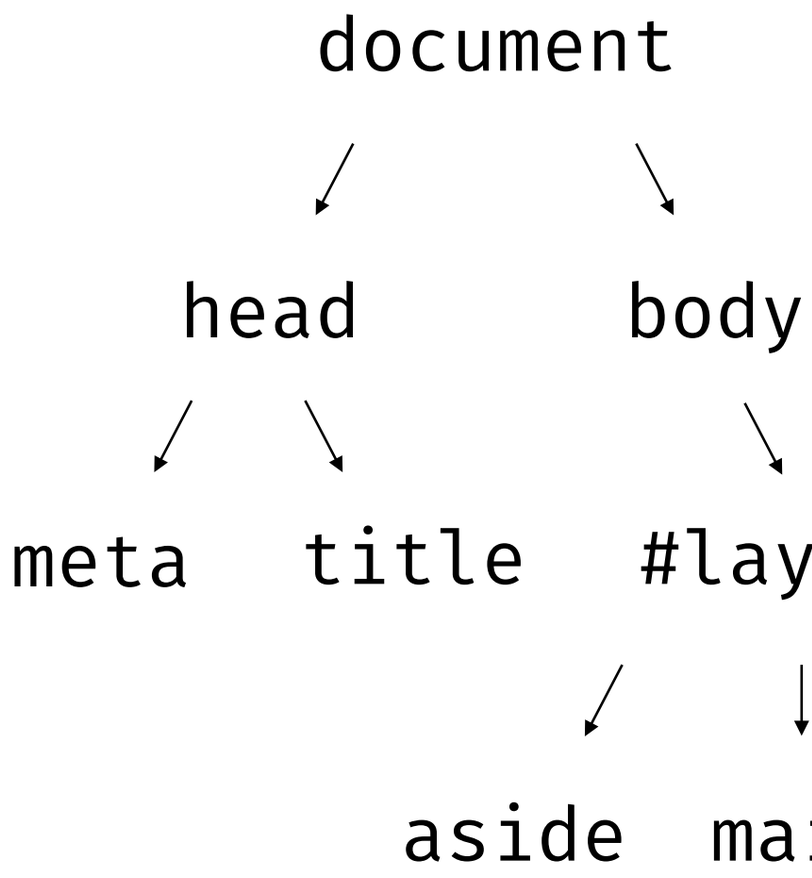
Деревья



Существует несколько способов перебора элементов дерева:

- DFS (поиск в глубину)

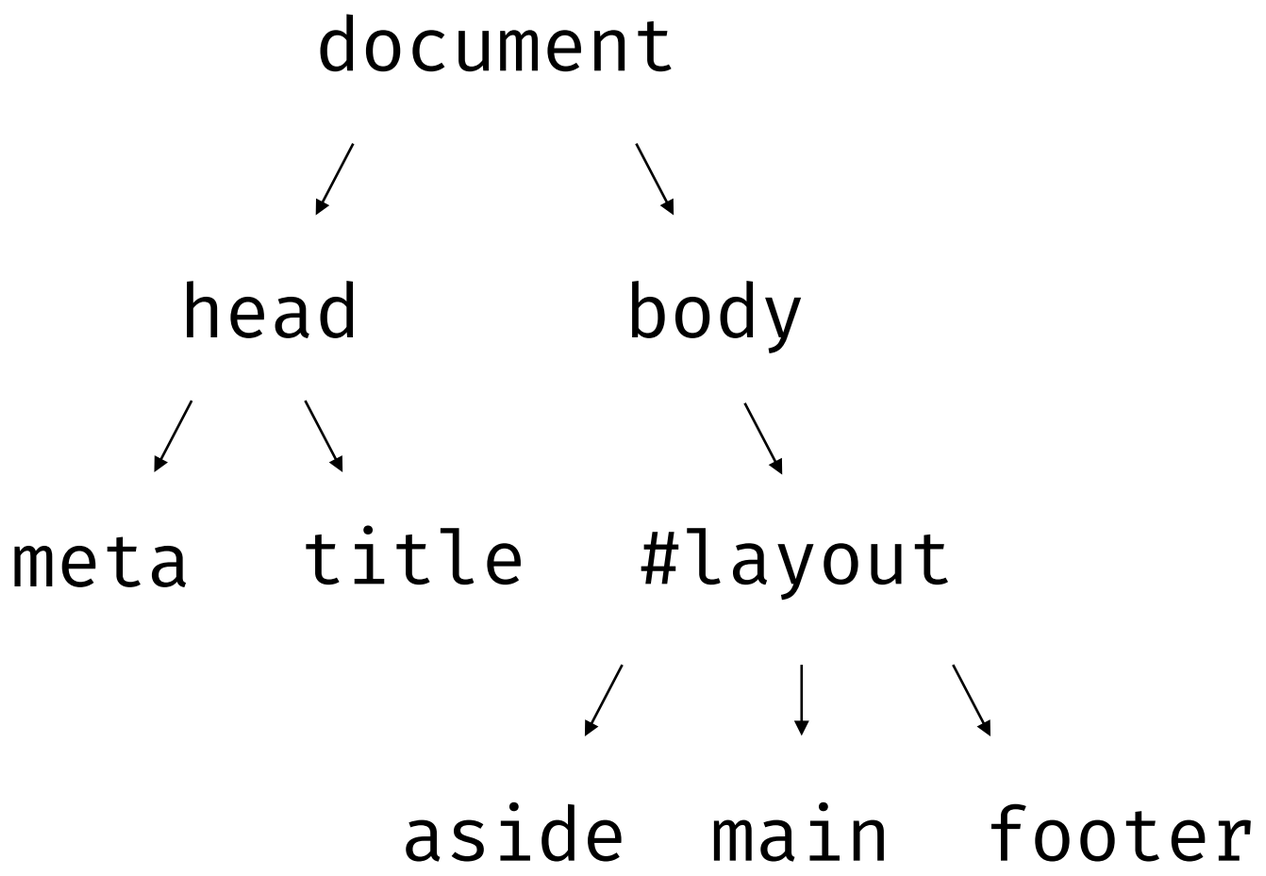
Деревья



Существует несколько способов перебора элементов дерева:

- DFS (поиск в глубину)
  - прямой

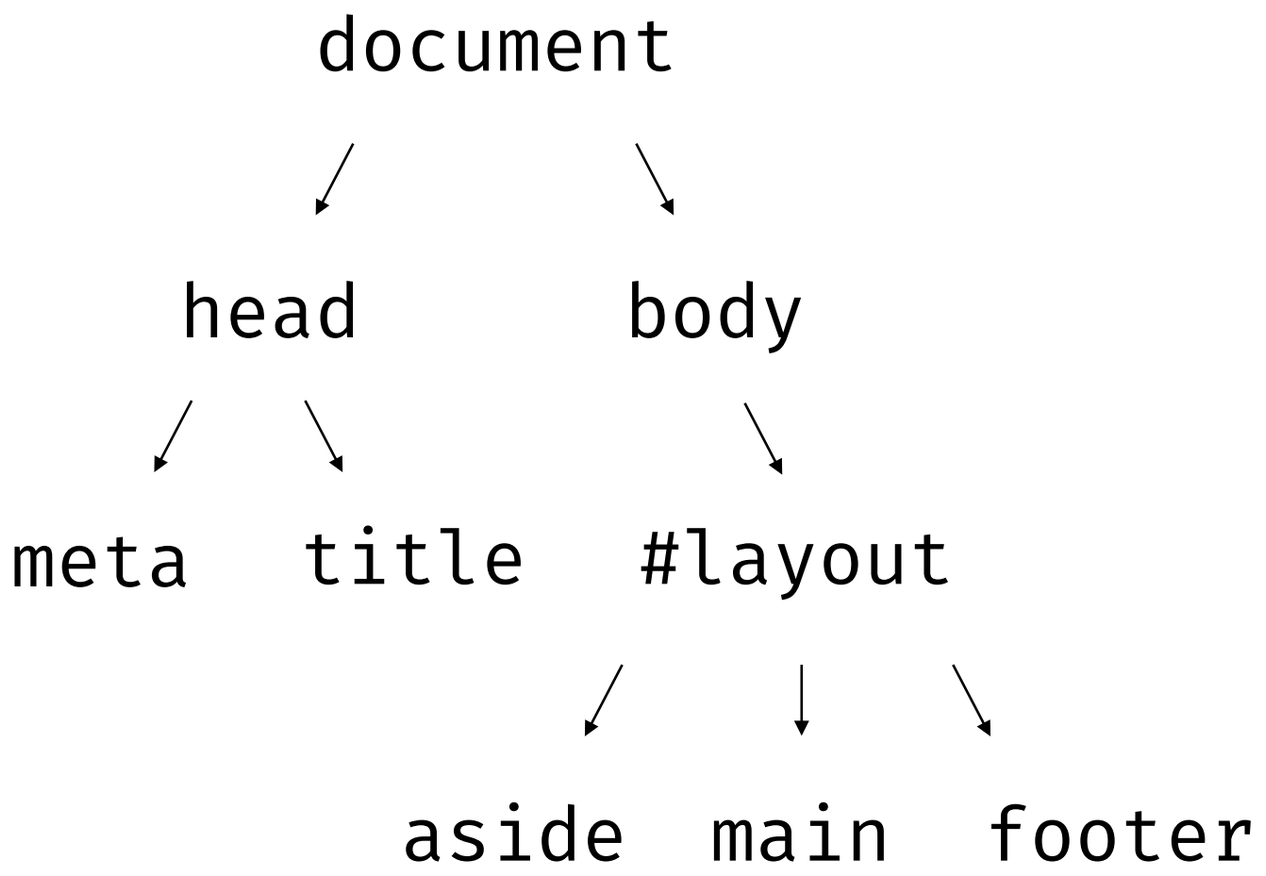
Деревья



Существует несколько способов перебора элементов дерева:

- DFS (поиск в глубину)
  - прямой
  - обратный

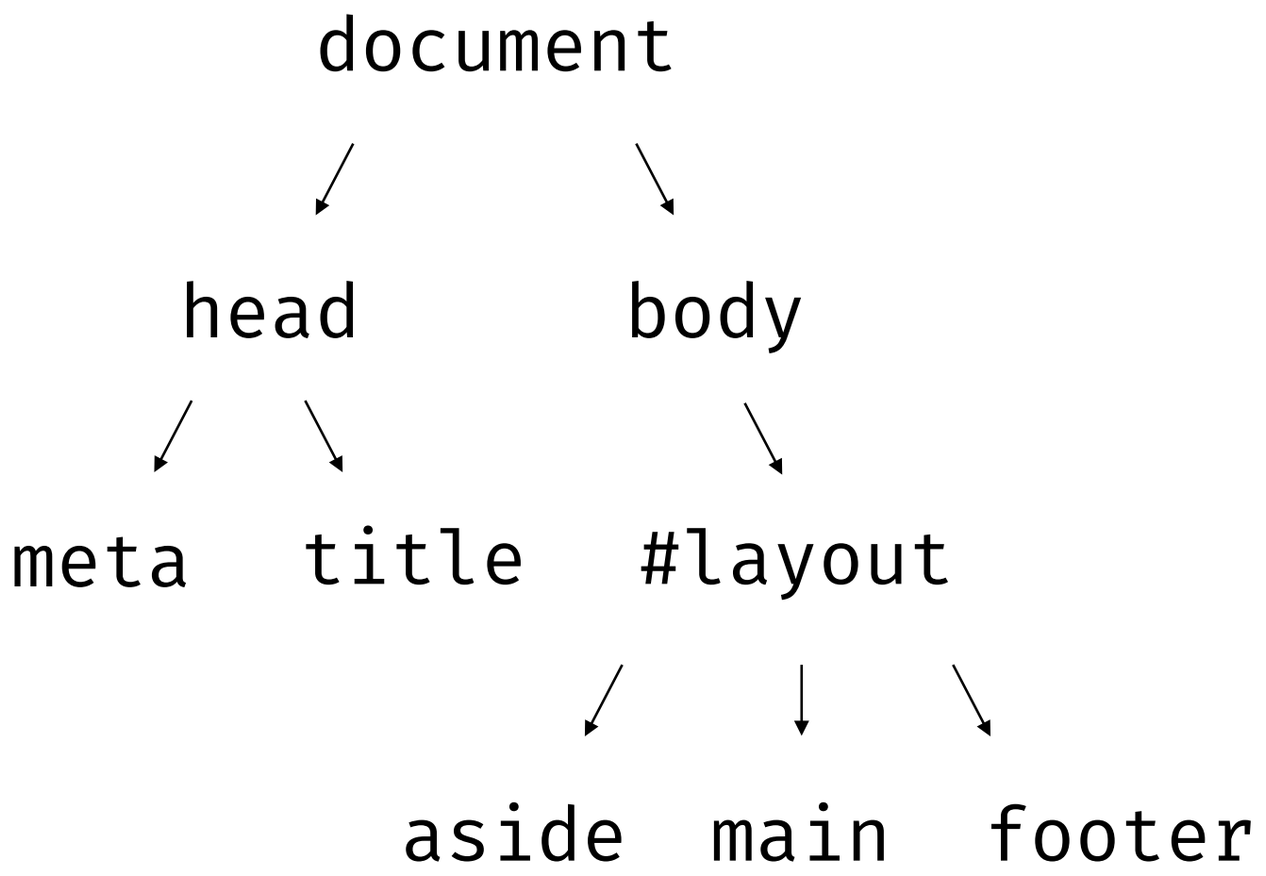
Деревья



Существует несколько способов перебора элементов дерева:

- DFS (поиск в глубину)
  - прямой
  - обратный
  - симметричный

Деревья



Существует несколько способов перебора элементов дерева:

- DFS (поиск в глубину)
  - прямой
  - обратный
  - симметричный
- BFS (поиск в ширину)

Demo

Игорь

localhost:8080

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="layout"></div>
</body>
</html>
```

CSS | LESS

```
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
```

JavaScript

```
body.onload = function() {
  console.log('jopa');
}
```

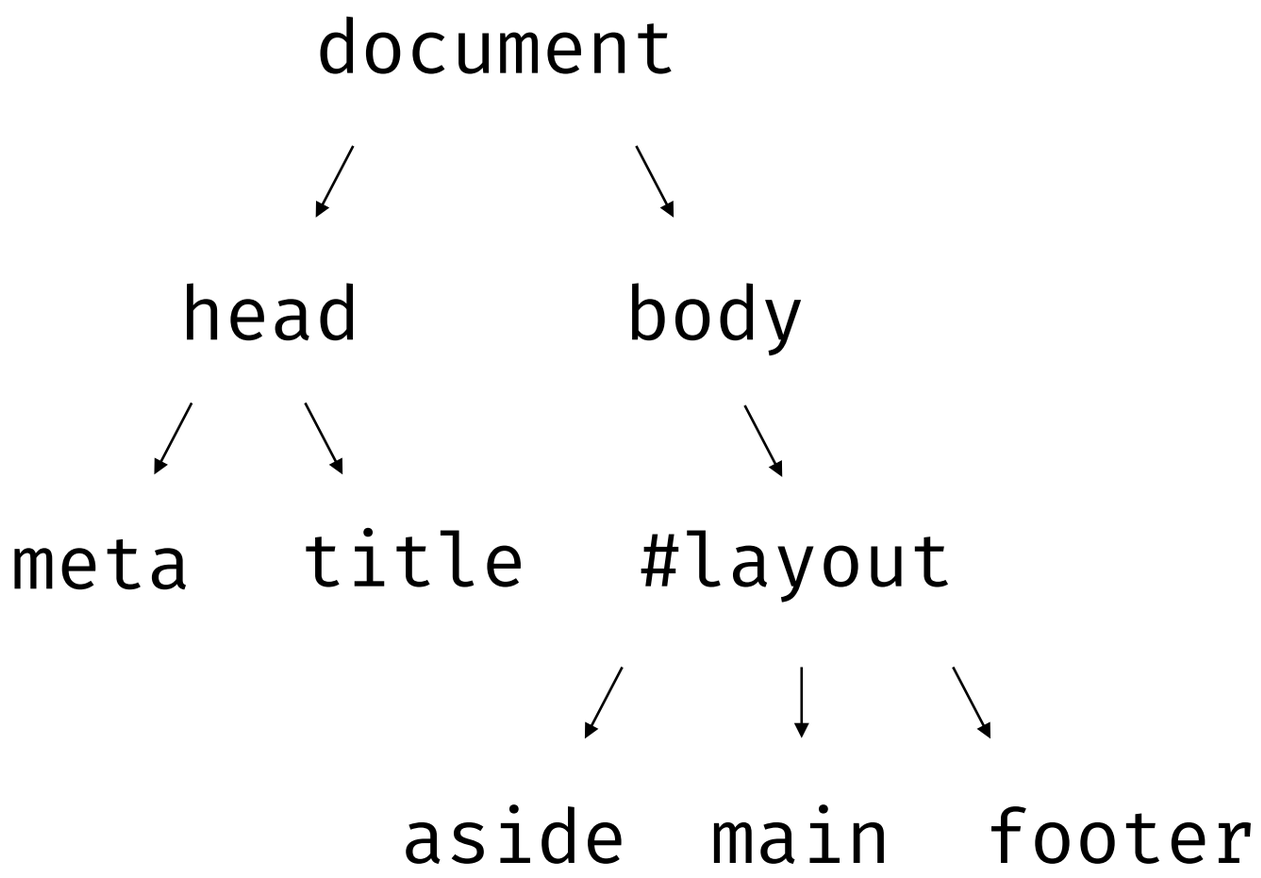
×1 ×0.5 ×0.75 ×1.5 400px 800px 1000px 1500px 1

Прямой поиск в глубину

(сербохорв. — Depth-first search, DFS) если у узла есть потомки, посещаются сначала они, а только потом соседний узел (всегда идём до самого конца)

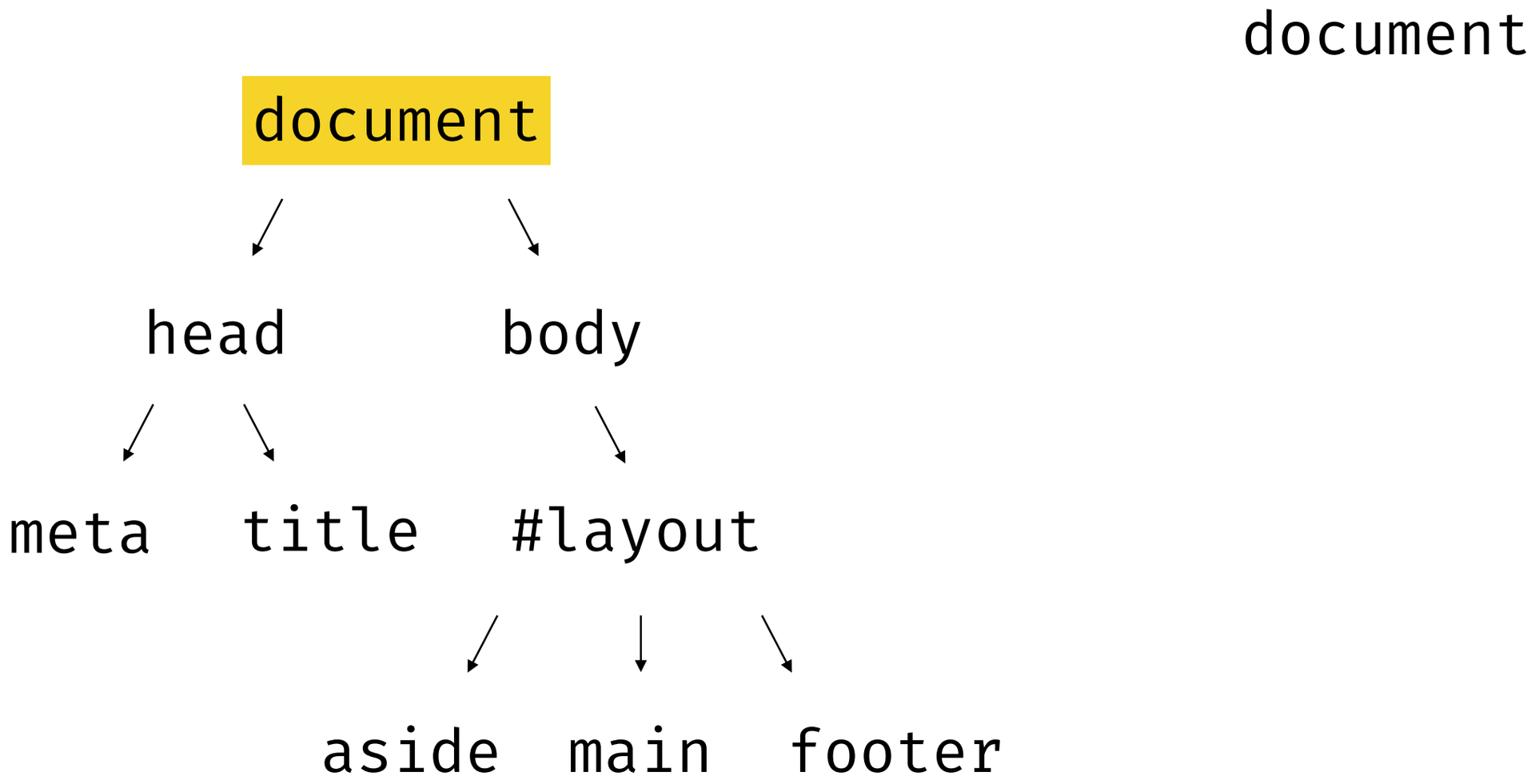
Прямой обход в глубину

Сначала посещается узел, а потом его потомки



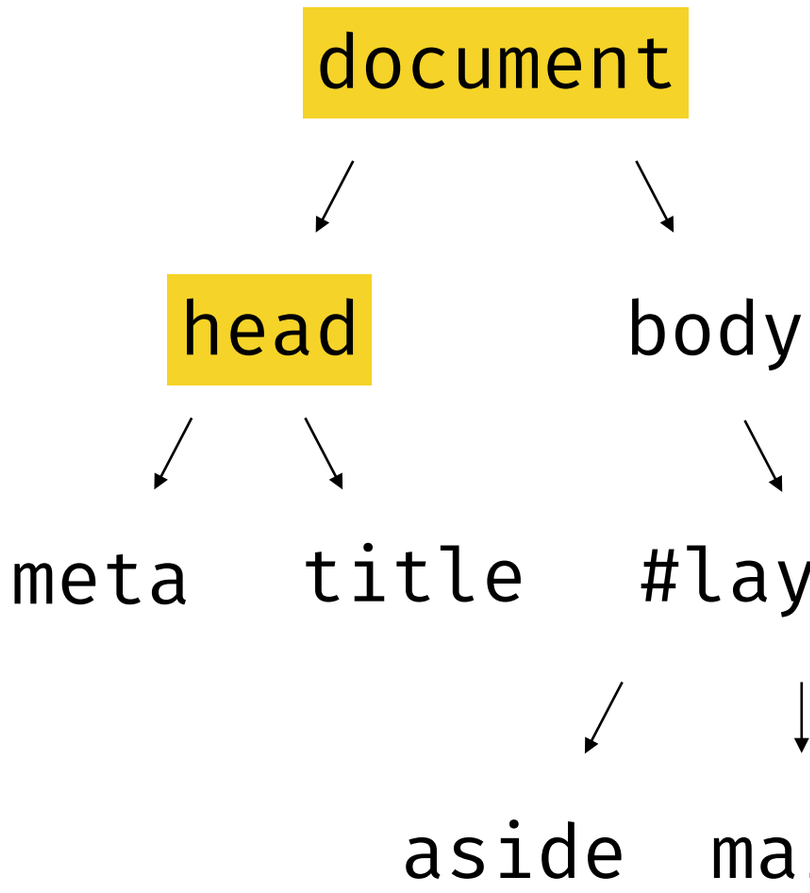
Прямой обход в глубину

Сначала посещается узел, а потом его потомки



Прямой обход в глубину

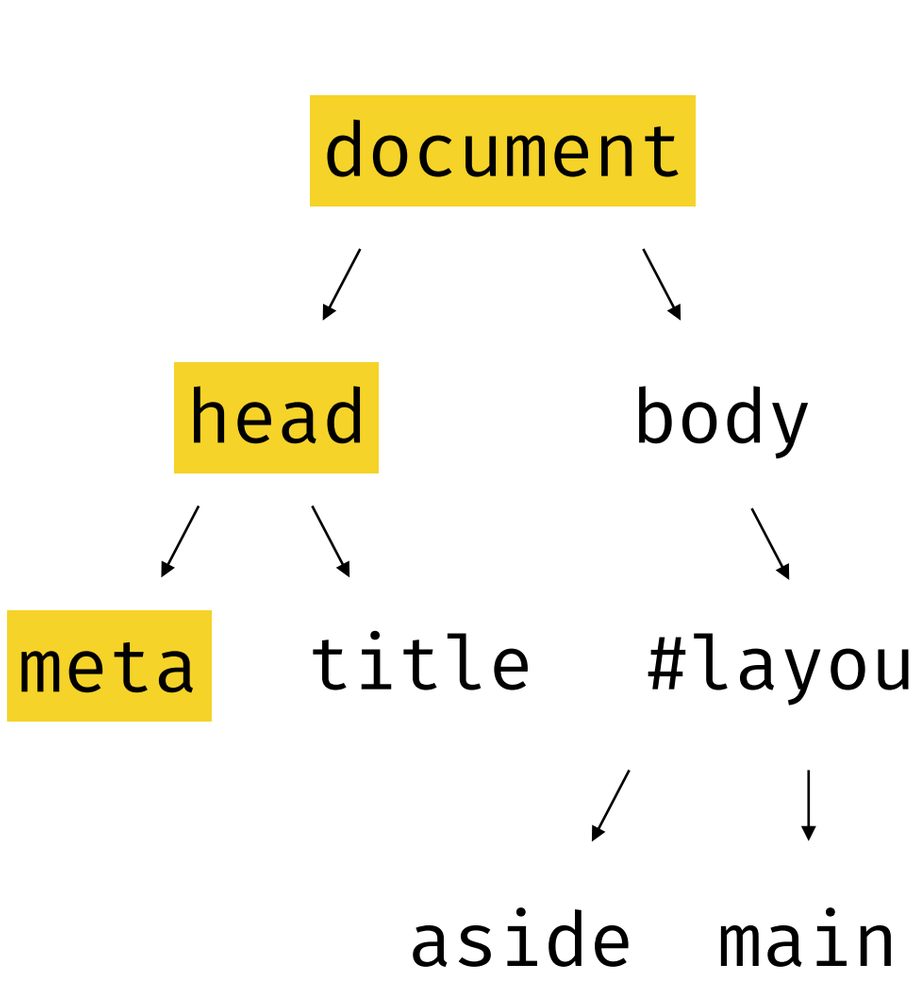
Сначала посещается узел, а потом его потомки




```
document
head
```

Прямой обход в глубину

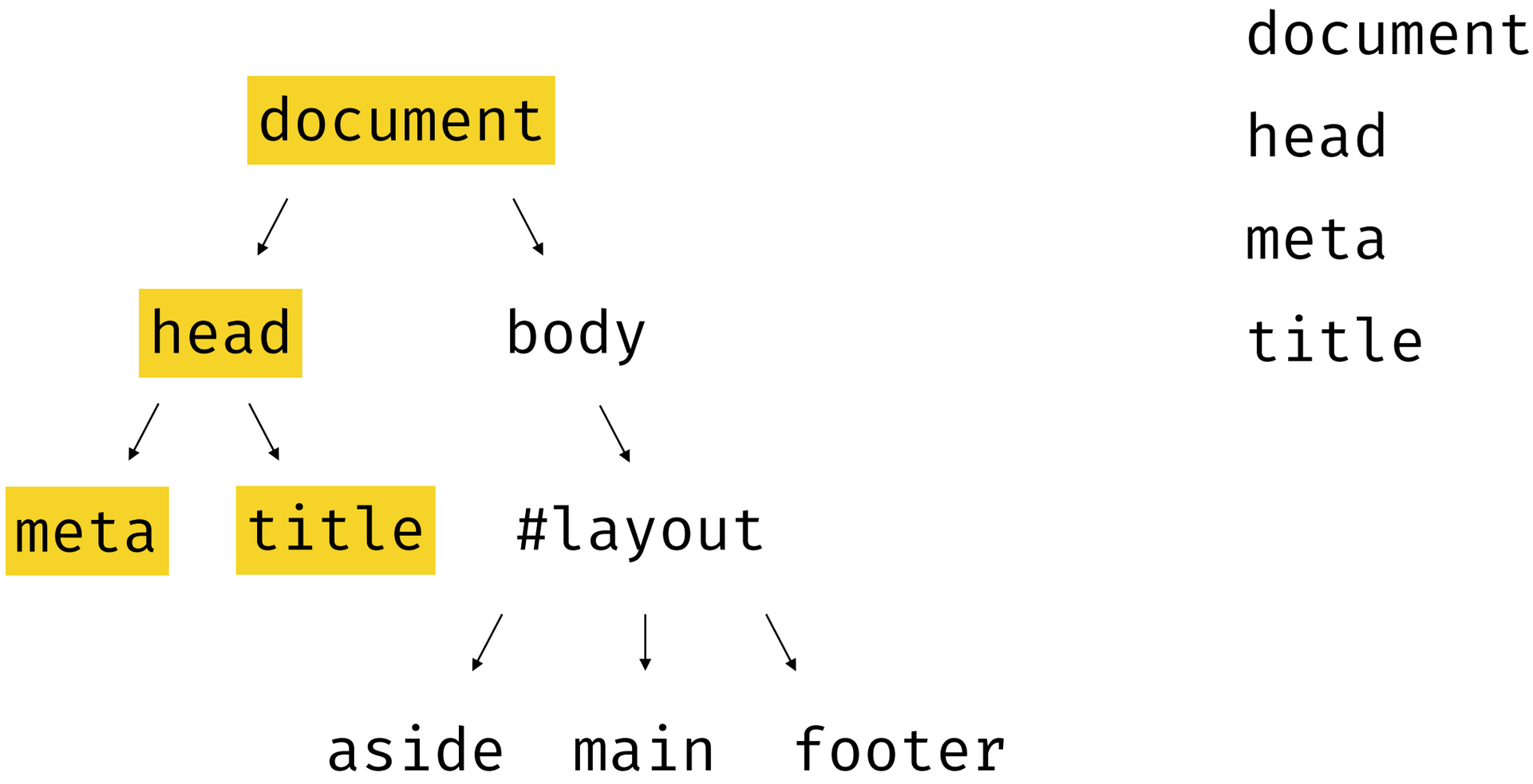
Сначала посещается узел, а потом его потомки




```
document
head
meta
```

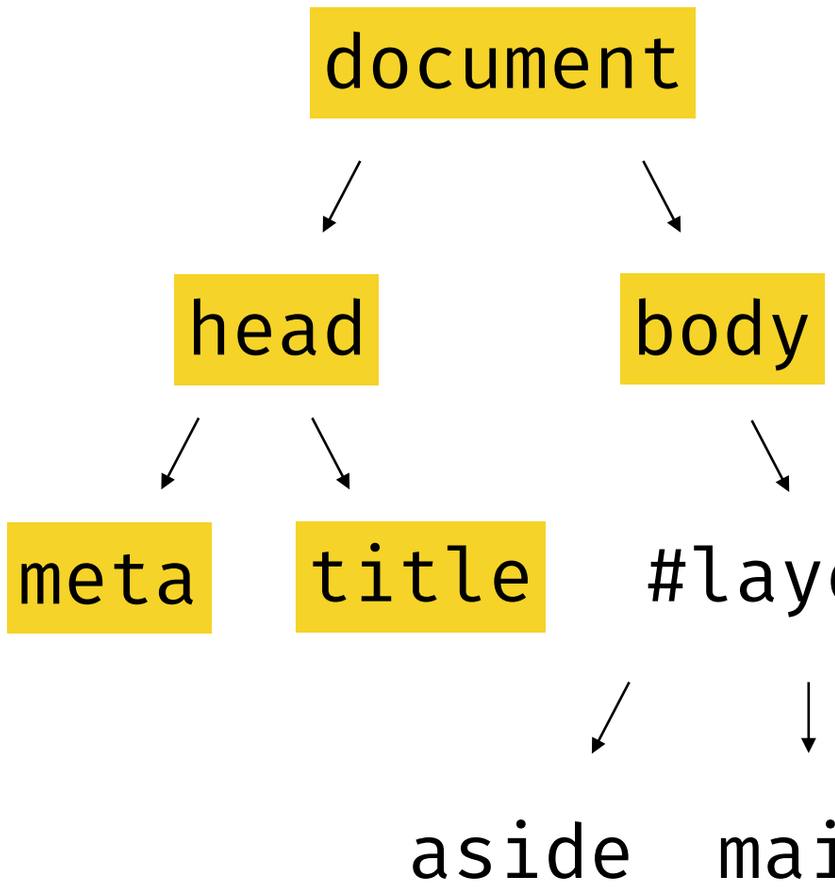
Прямой обход в глубину

Сначала посещается узел, а потом его потомки



Прямой обход в глубину

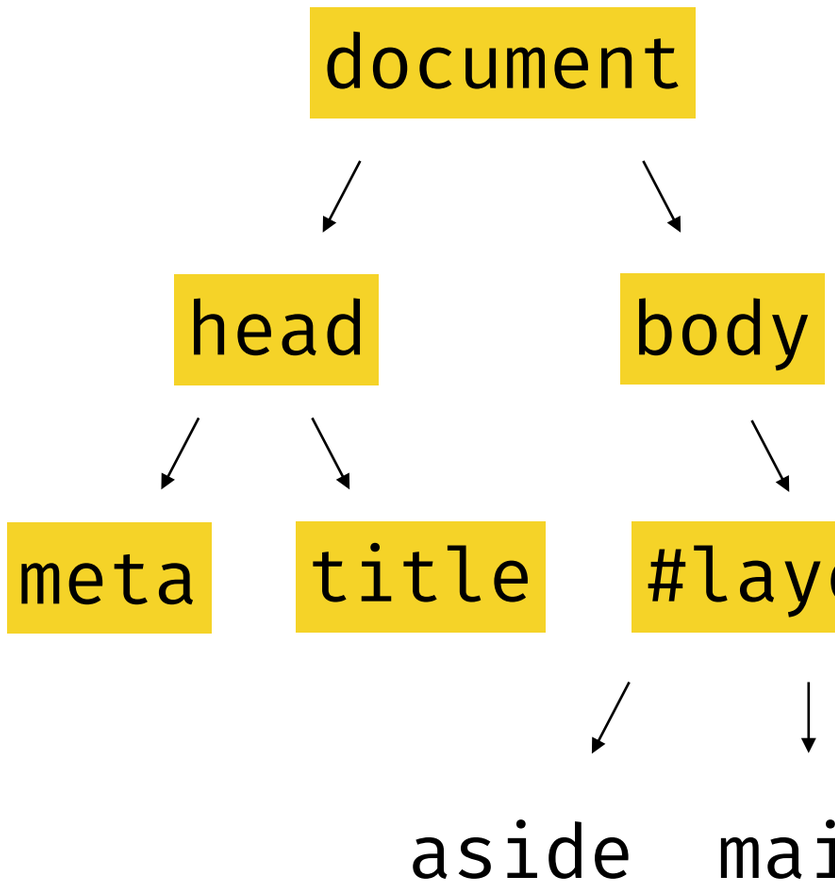
Сначала посещается узел, а потом его потомки




```
document
head
meta
title
body
```

Прямой обход в глубину

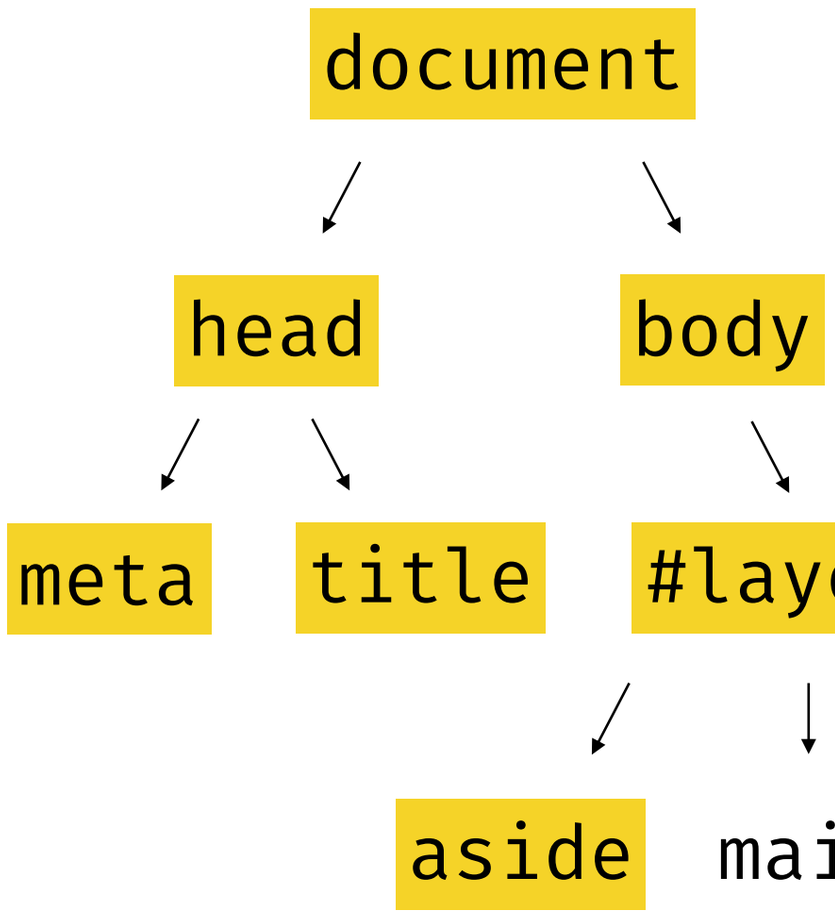
Сначала посещается узел, а потом его потомки




```
document
head
meta
title
body
#layout
```

Прямой обход в глубину

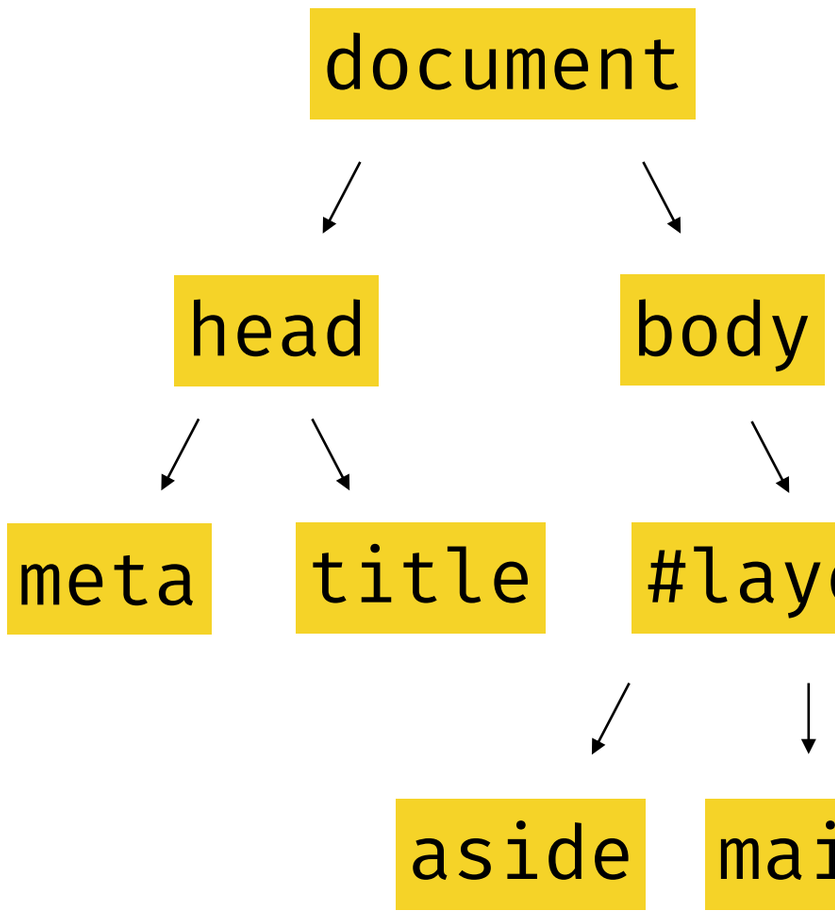
Сначала посещается узел, а потом его потомки




```
document
head
meta
title
body
#layout
aside
```

Прямой обход в глубину

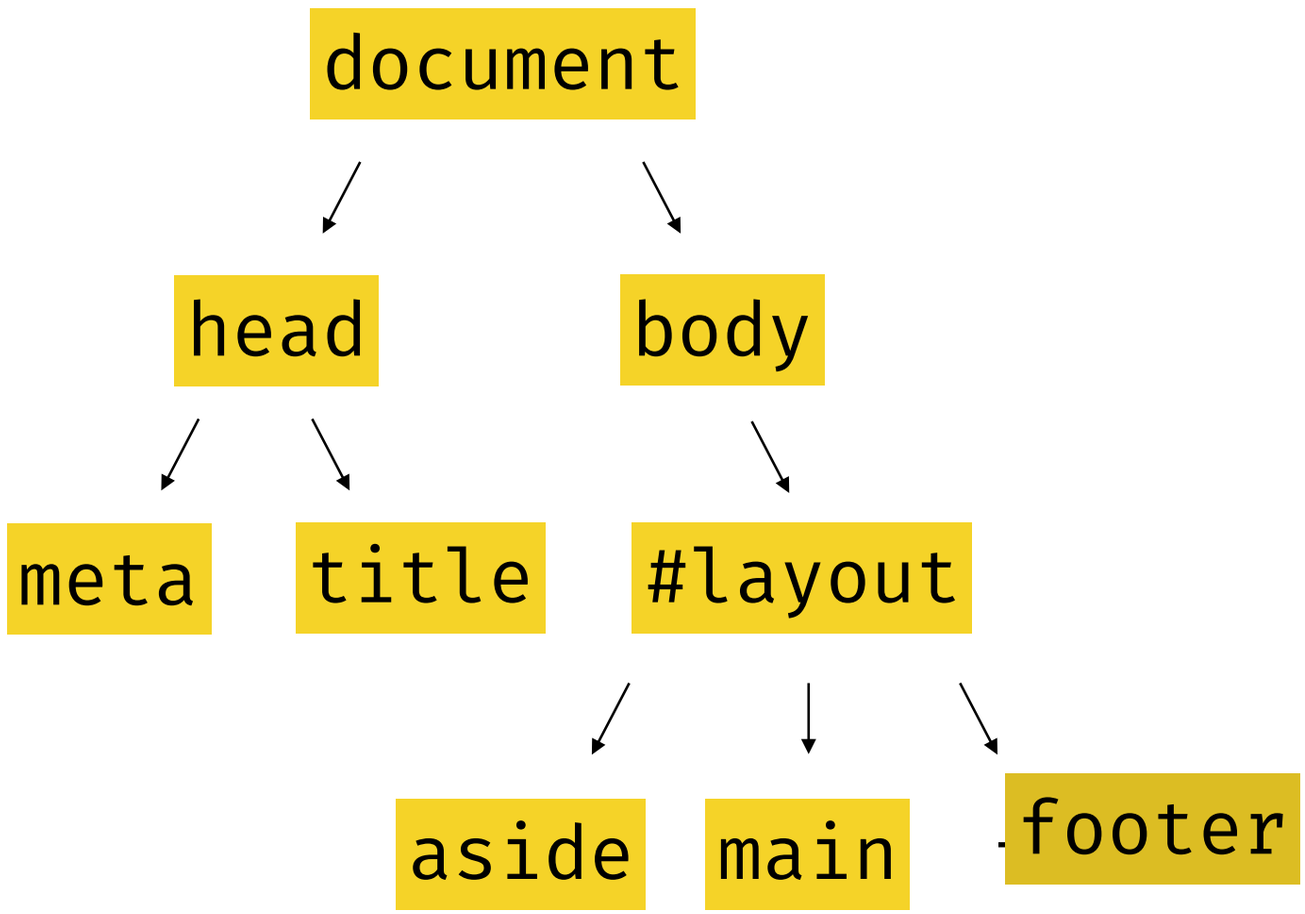
Сначала посещается узел, а потом его потомки




```
document
head
meta
title
body
#layout
aside
main
```

Прямой обход в глубину

Сначала посещается узел, а потом его потомки




```
document
head
meta
title
body
#layout
aside
main
 footer
```

Demo

Игорь

localhost:8080

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="layout"></div>
</body>
</html>
```

CSS | LESS

```
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
```

JavaScript

```
body.onload = function() {
  console.log('jopa');
}
```

×1 ×0.5 ×0.75 ×1.5 400px 800px 1000px 1500px 1

Demo

Игорь

localhost:8080

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="layout"></div>
</body>
</html>
```

CSS | LESS

```
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
```

JavaScript

```
body.onload = function() {
  console.log('jopa');
}
```

×1 ×0.5 ×0.75 ×1.5 400px 800px 1000px 1500px

Demo

Игорь

localhost:8080

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="layout"></div>
</body>
</html>
```

CSS | LESS

```
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
```

JavaScript

```
body.onload = function() {
  console.log('jopa');
}
```

×1 ×0.5 ×0.75 ×1.5 400px 800px 1000px 1500px 1

Demo

Игорь

localhost:8080

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="layout"></div>
</body>
</html>
```

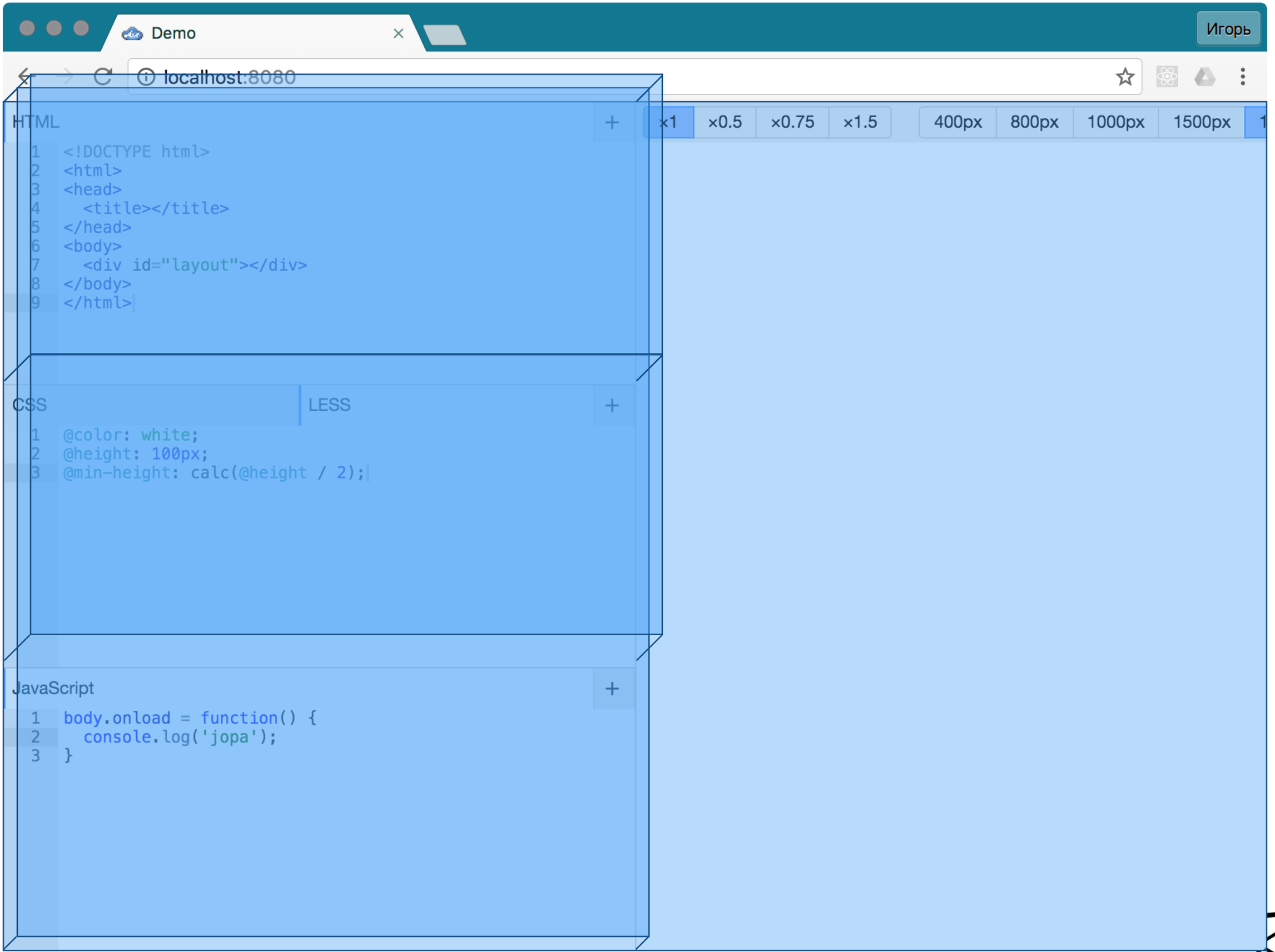
CSS | LESS

```
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
```

JavaScript

```
body.onload = function() {
  console.log('jopa');
}
```

×1 | ×0.5 | ×0.75 | ×1.5 | 400px | 800px | 1000px | 1500px | 1


Demo
Игорь
localhost:8080
HTML
<!DOCTYPE html>
<html>
<head>
<title></title>
</head>
<body>

</body>
</html>
CSS
LESS
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
JavaScript
body.onload = function() {
console.log('jopa');
}
×1
×0.5
×0.75
×1.5
400px
800px
1000px
1500px

Demo

Игорь

localhost:8080

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="layout"></div>
</body>
</html>
```

CSS LESS

```
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
```

JavaScript

```
body.onload = function() {
  console.log('jopa');
}
```

×1 ×0.5 ×0.75 ×1.5 400px 800px 1000px 1500px 1

Demo

Игорь

localhost:8080

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="layout"></div>
</body>
</html>
```

CSS LESS

```
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
```

JavaScript

```
body.onload = function() {
  console.log('jopa');
}
```

×1 ×0.5 ×0.75 ×1.5 400px 800px 1000px 1500px

Demo

Игорь

localhost:8080

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="layout"></div>
</body>
</html>
```

CSS LESS

```
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
```

JavaScript

```
body.onload = function() {
  console.log('jopa');
}
```

×1 ×0.5 ×0.75 ×1.5 400px 800px 1000px 1500px 1

Demo

Игорь

localhost:8080

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="layout"></div>
</body>
</html>
```

×1 ×0.5 ×0.75 ×1.5 400px 800px 1000px 1500px

CSS LESS

```
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
```

JavaScript

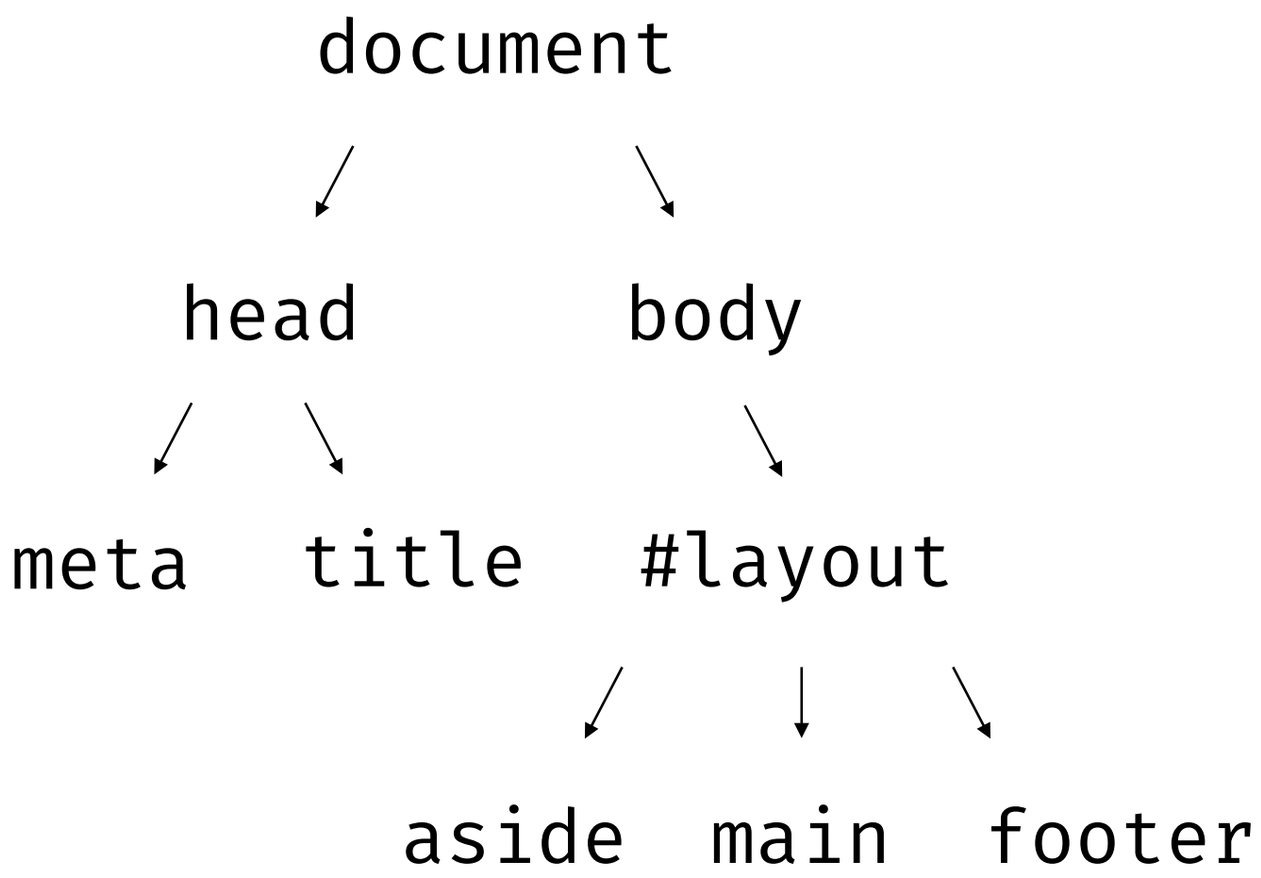
```
body.onload = function() {
  console.log('jopa');
}
```

Поиск в ширину

(древнеарамейск. Breadth-first search, BFS) узлы посещаются по уровням, будто срезаются слои с торта (смотрим по верхам)

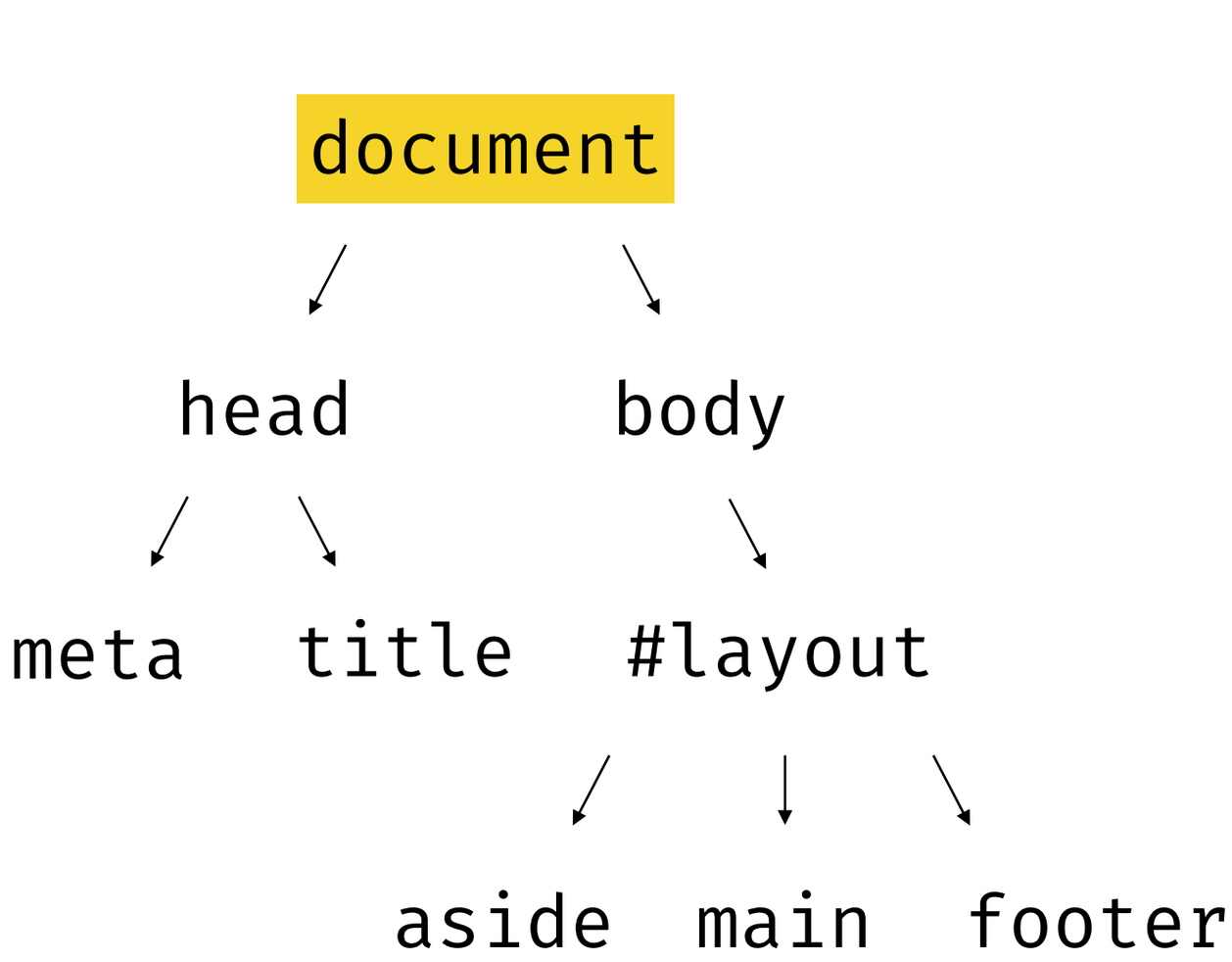
Обход в ширину

Сначала посещаются все узлы на уровне, потом их потомки и так до самого низа



Обход в ширину

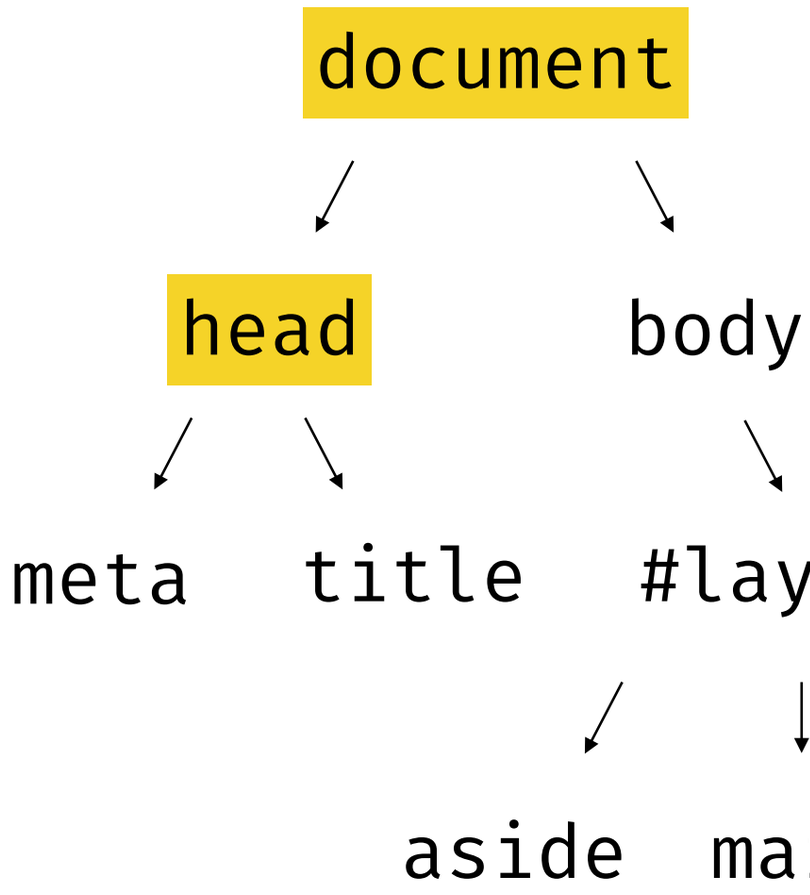
Сначала посещаются все узлы на уровне, потом их потомки и так до самого низа



document

Обход в ширину

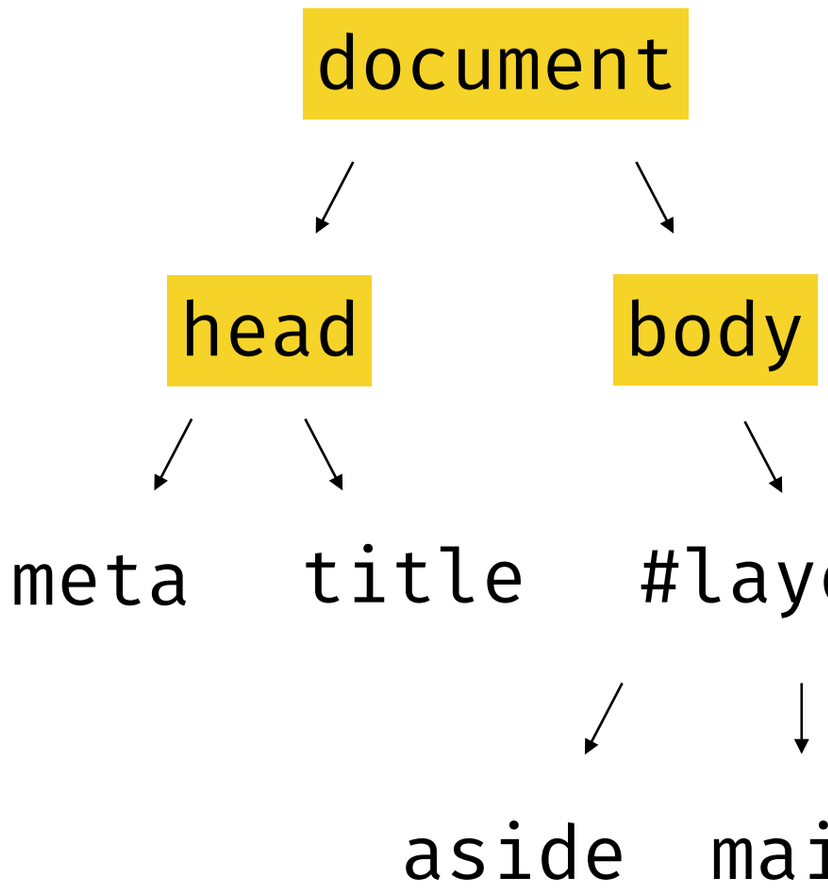
Сначала посещаются все узлы на уровне, потом их потомки и так до самого низа




```
document
head
```

Обход в ширину

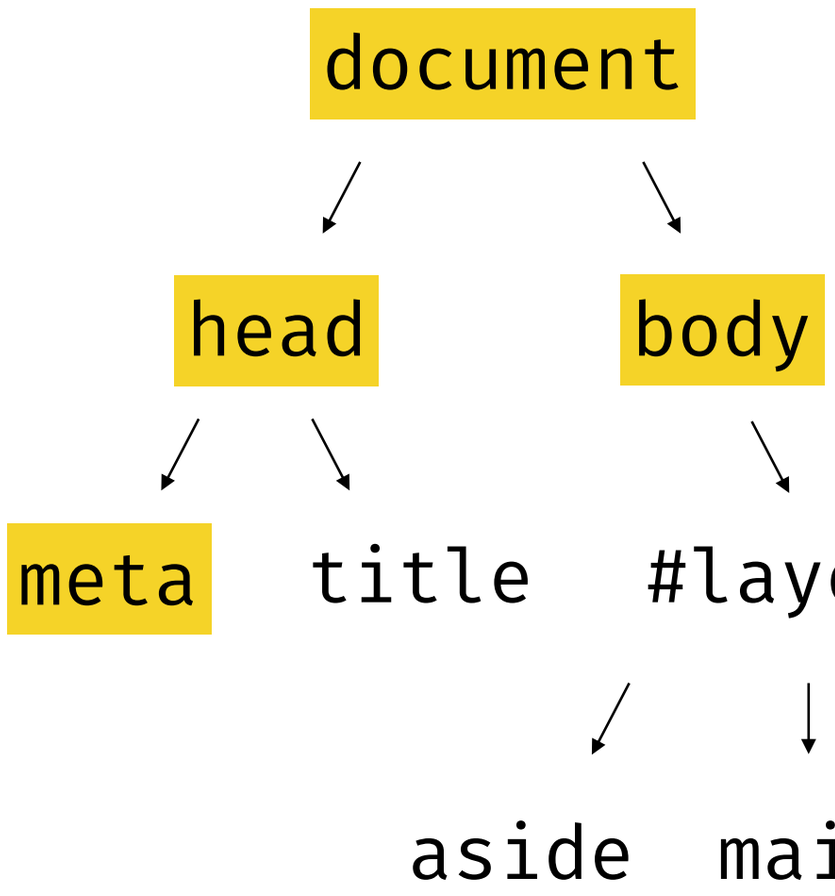
Сначала посещаются все узлы на уровне, потом их потомки и так до самого низа




```
document
head
body
```

Обход в ширину

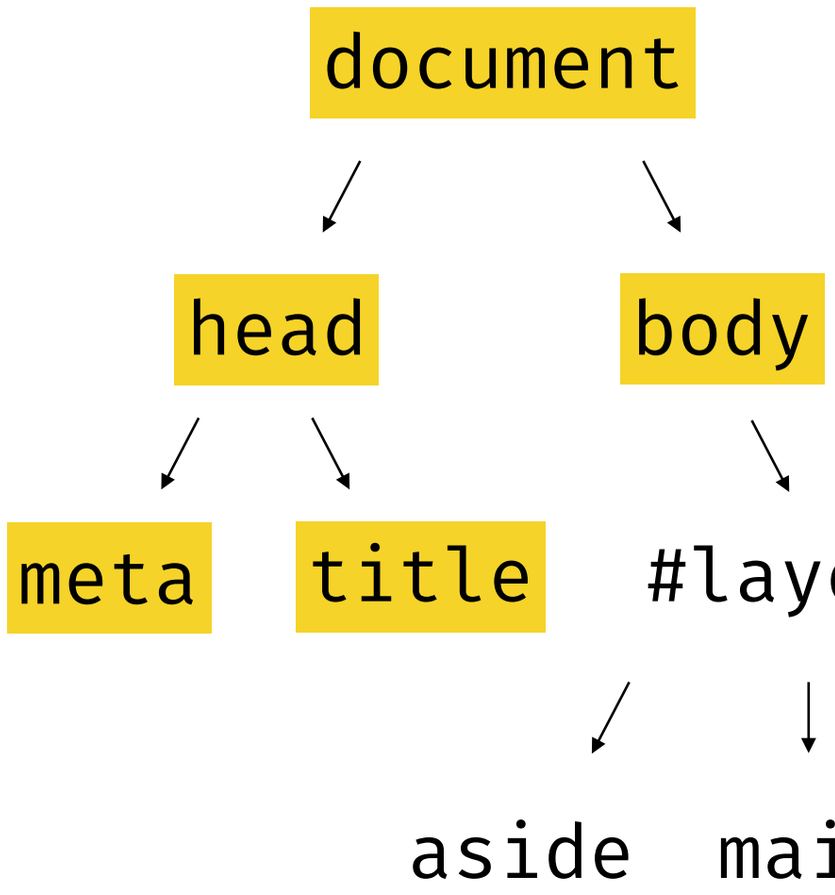
Сначала посещаются все узлы на уровне, потом их потомки и так до самого низа




```
document
head
body
meta
```

Обход в ширину

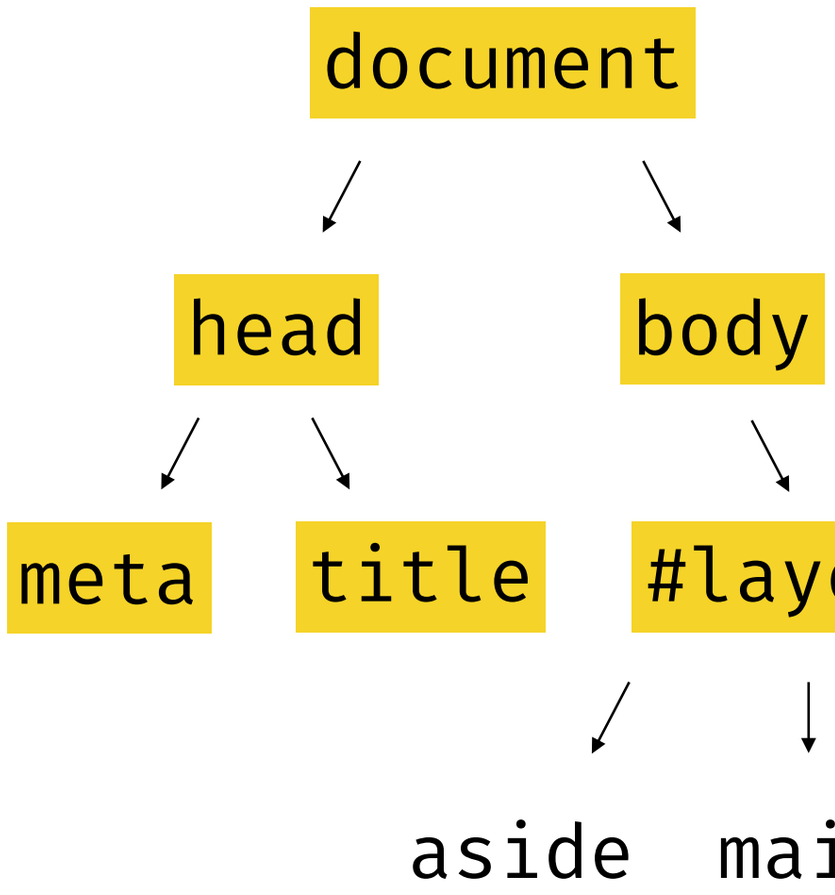
Сначала посещаются все узлы на уровне, потом их потомки и так до самого низа




```
document
head
body
meta
title
```

Обход в ширину

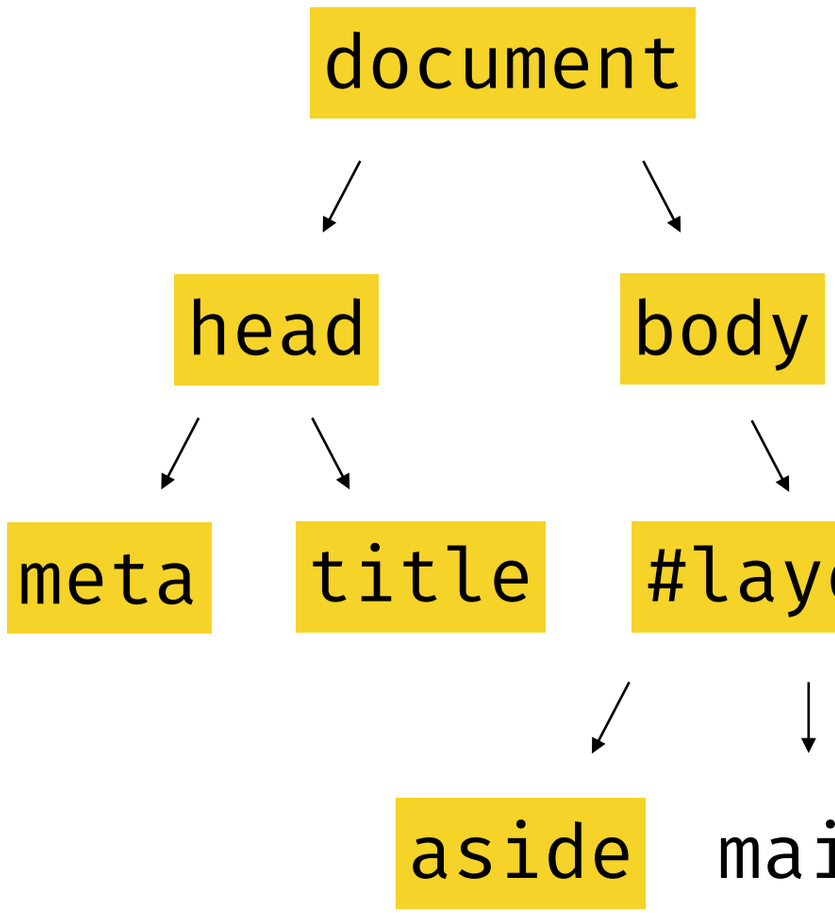
Сначала посещаются все узлы на уровне, потом их потомки и так до самого низа




```
document
head
body
meta
title
#layout
```

Обход в ширину

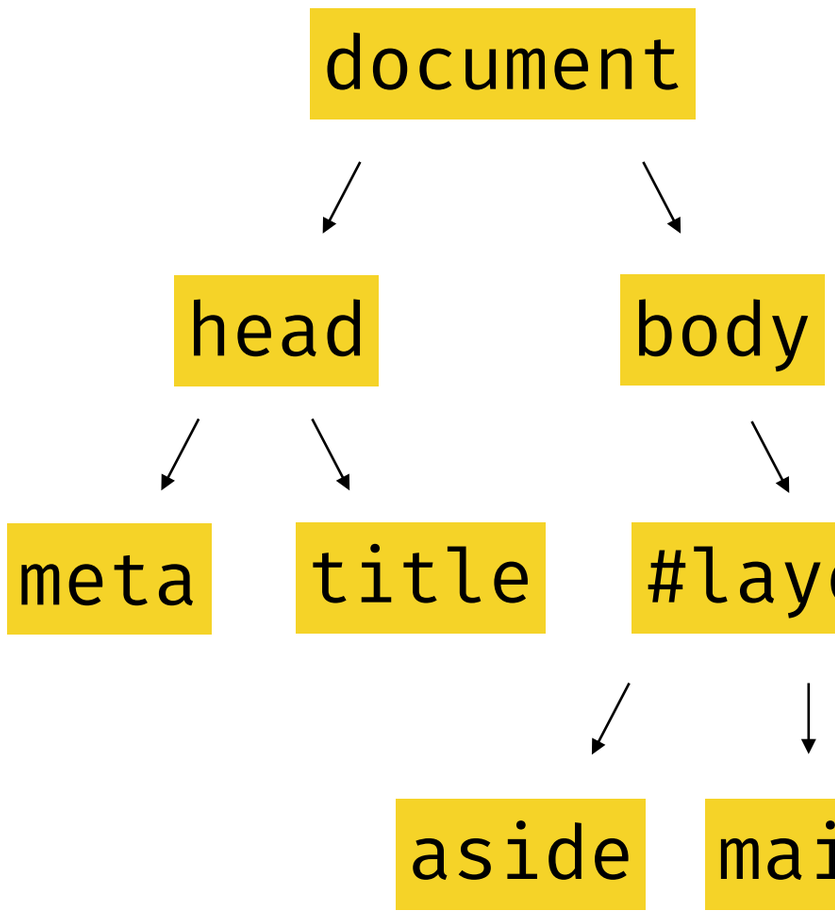
Сначала посещаются все узлы на уровне, потом их потомки и так до самого низа




```
document
head
body
meta
title
#layout
aside
```

Обход в ширину

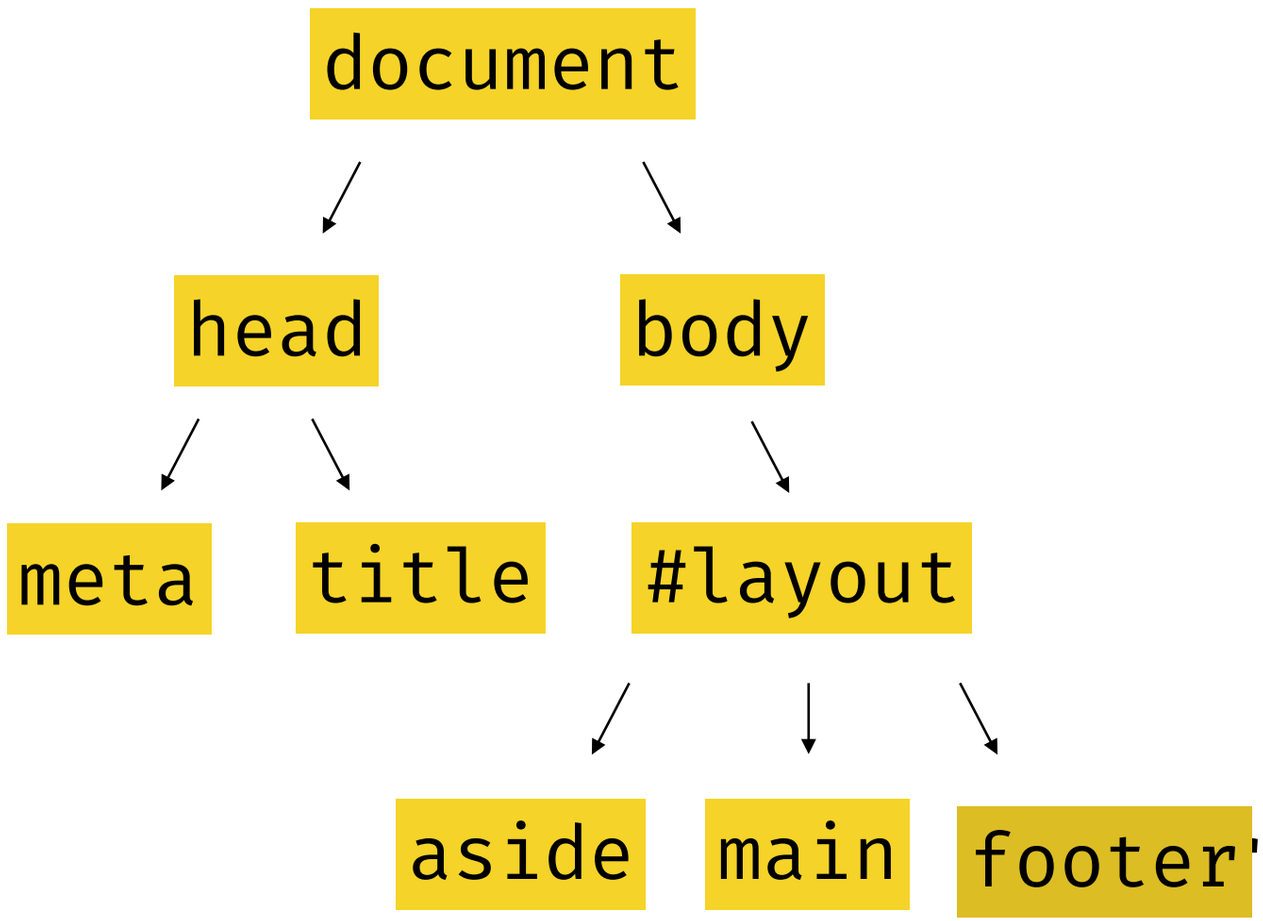
Сначала посещаются все узлы на уровне, потом их потомки и так до самого низа




```
document
head
body
meta
title
#layout
aside
main
```

Обход в ширину

Сначала посещаются все узлы на уровне, потом их потомки и так до самого низа




```
document
head
body
meta
title
#layout
aside
main
 footer
```

Demo

Игорь

localhost:8080

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="layout"></div>
</body>
</html>
```

CSS | LESS

```
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
```

JavaScript

```
body.onload = function() {
  console.log('jopa');
}
```

×1 ×0.5 ×0.75 ×1.5 400px 800px 1000px 1500px 1

Demo

Игорь

localhost:8080

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="layout"></div>
</body>
</html>
```

CSS | LESS

```
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
```

JavaScript

```
body.onload = function() {
  console.log('jopa');
}
```

×1 ×0.5 ×0.75 ×1.5 400px 800px 1000px 1500px 1

Demo

Игорь

localhost:8080

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="layout"></div>
</body>
</html>
```

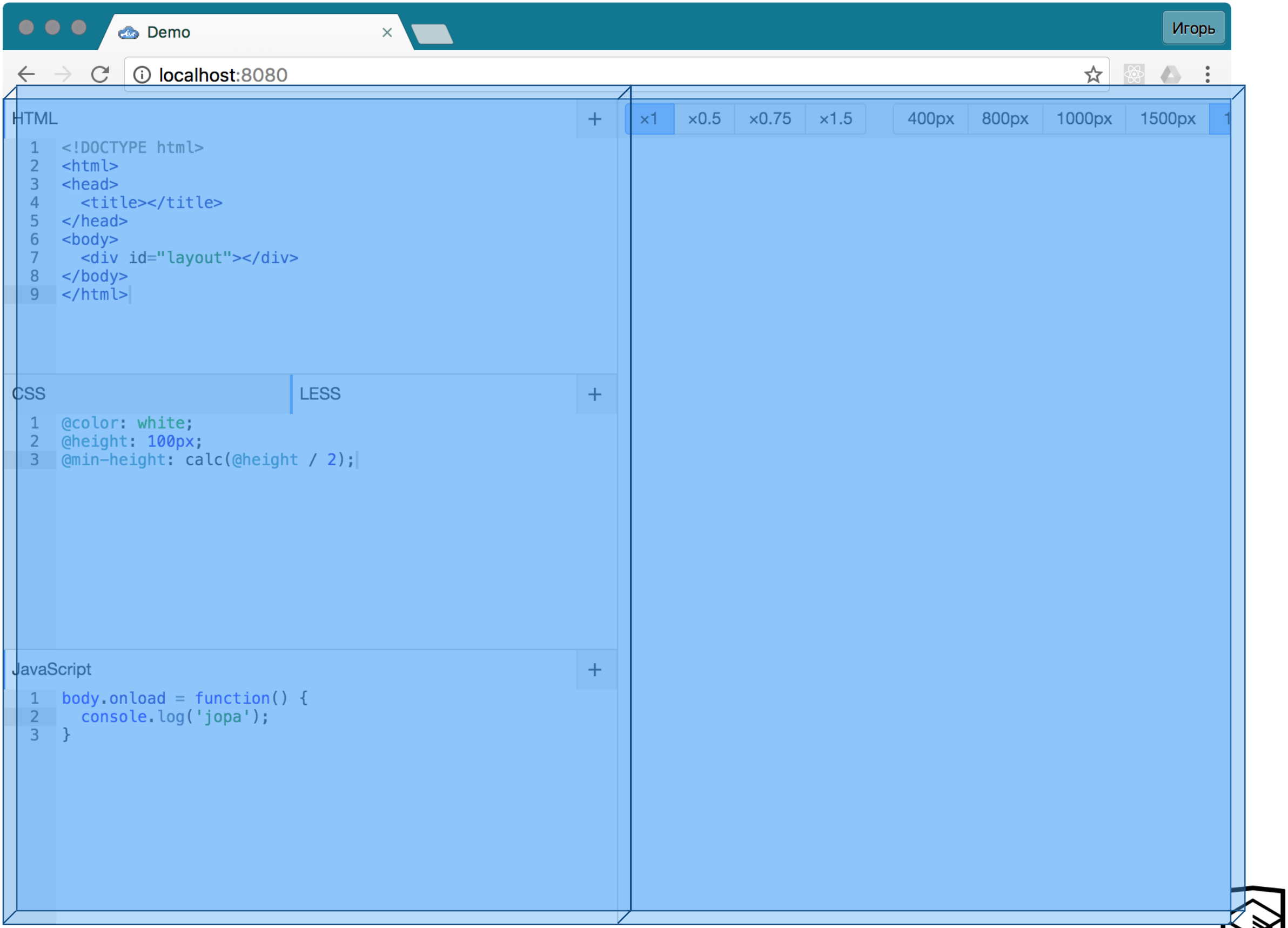
CSS | LESS

```
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
```

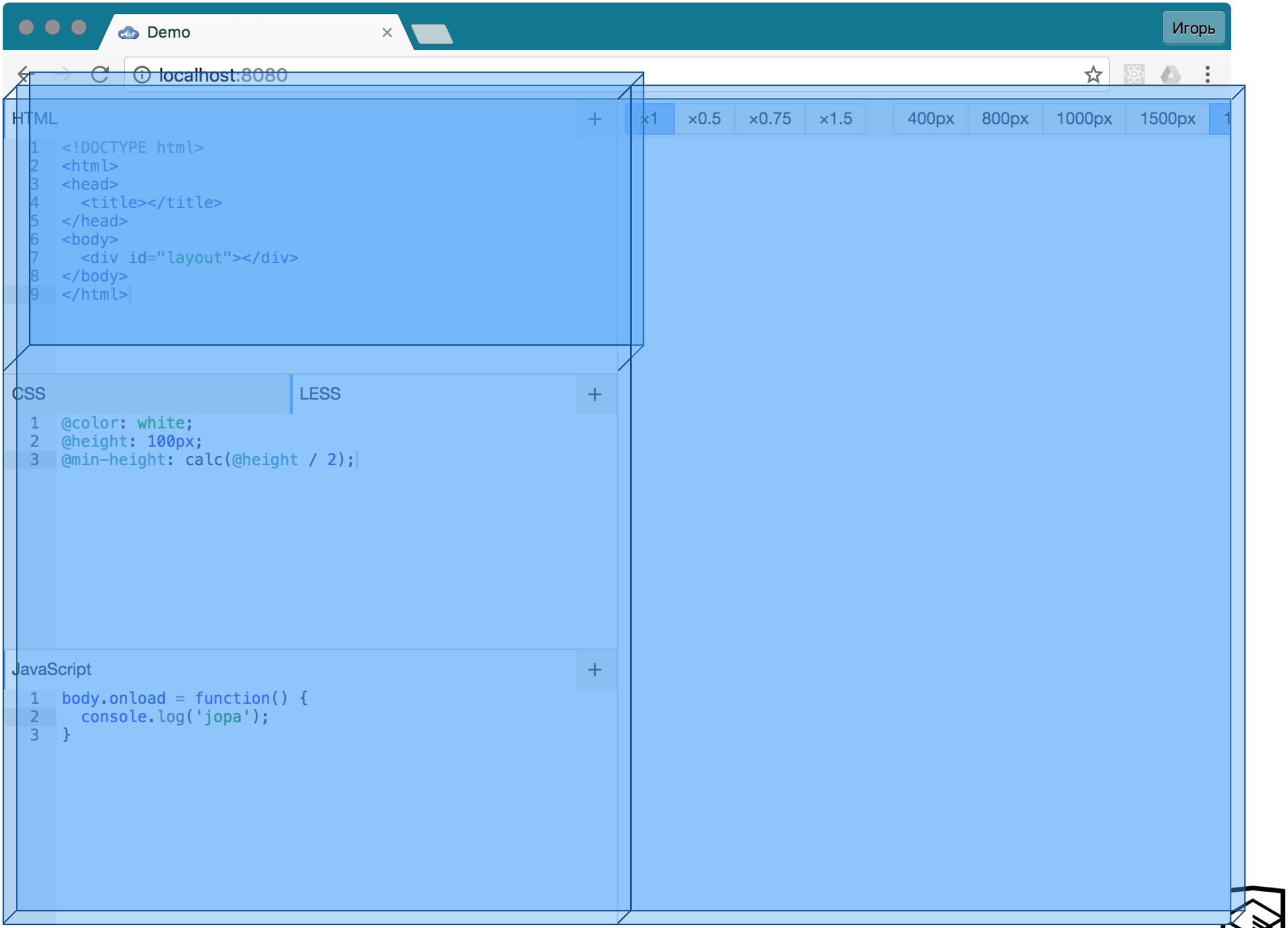
JavaScript

```
body.onload = function() {
  console.log('jopa');
}
```

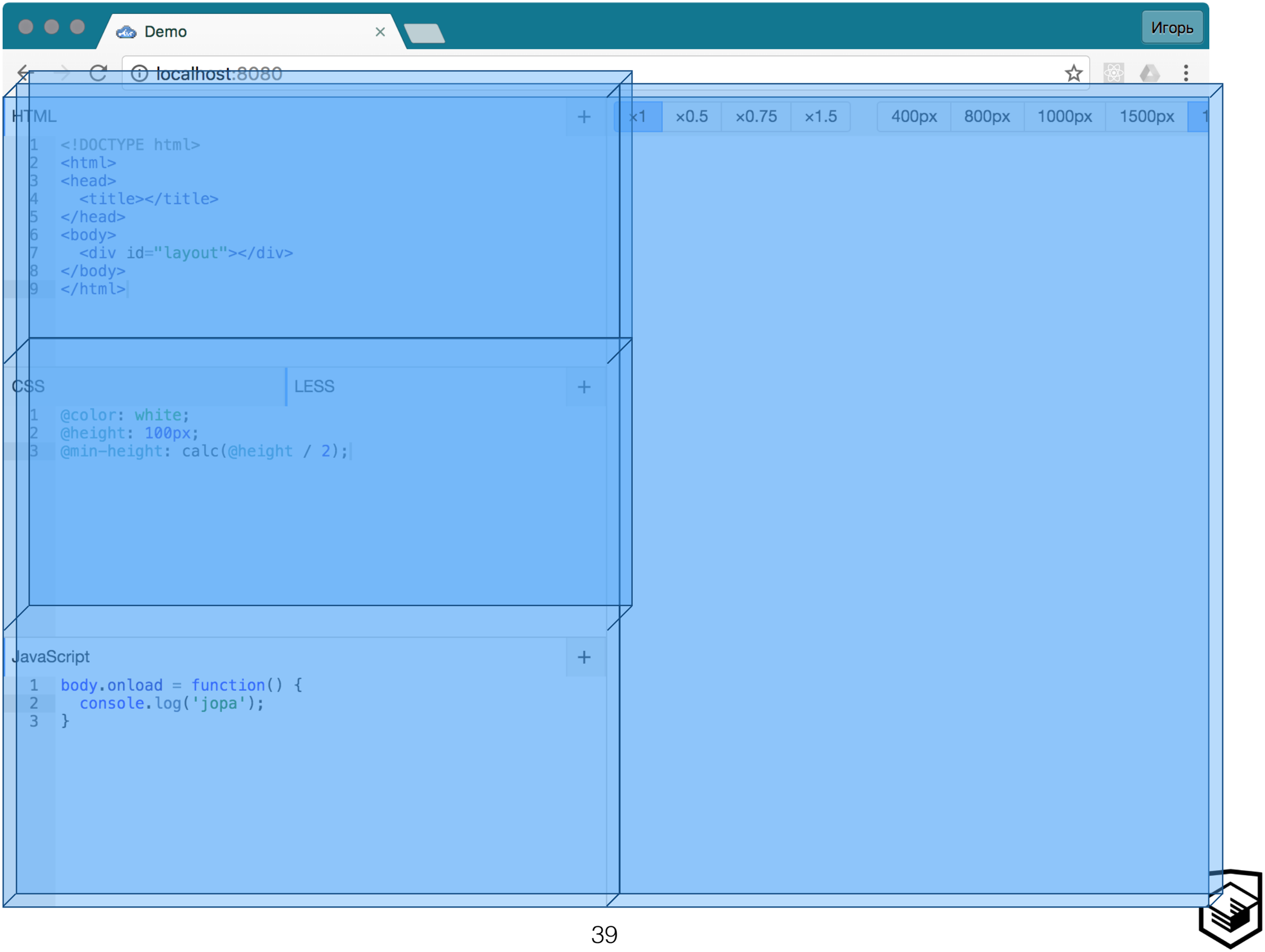
×1 ×0.5 ×0.75 ×1.5 400px 800px 1000px 1500px 1


Demo
Игорь
localhost:8080
HTML
<!DOCTYPE html>
<html>
<head>
<title></title>
</head>
<body>

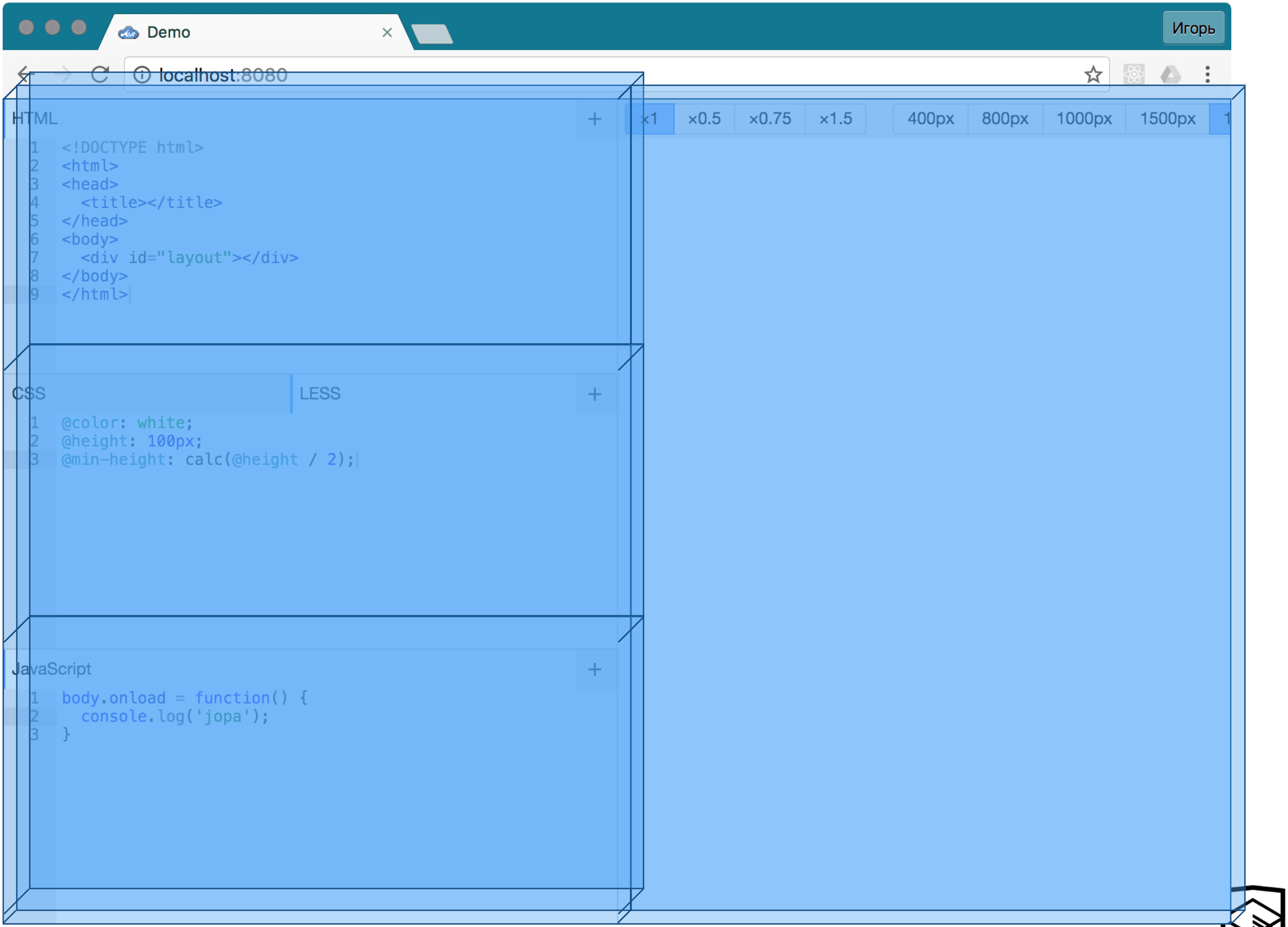
</body>
</html>
CSS
LESS
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
JavaScript
body.onload = function() {
console.log('jopa');
}
×1
×0.5
×0.75
×1.5
400px
800px
1000px
1500px


Demo
Игорь
localhost:8080
HTML
<!DOCTYPE html>
<html>
<head>
<title></title>
</head>
<body>

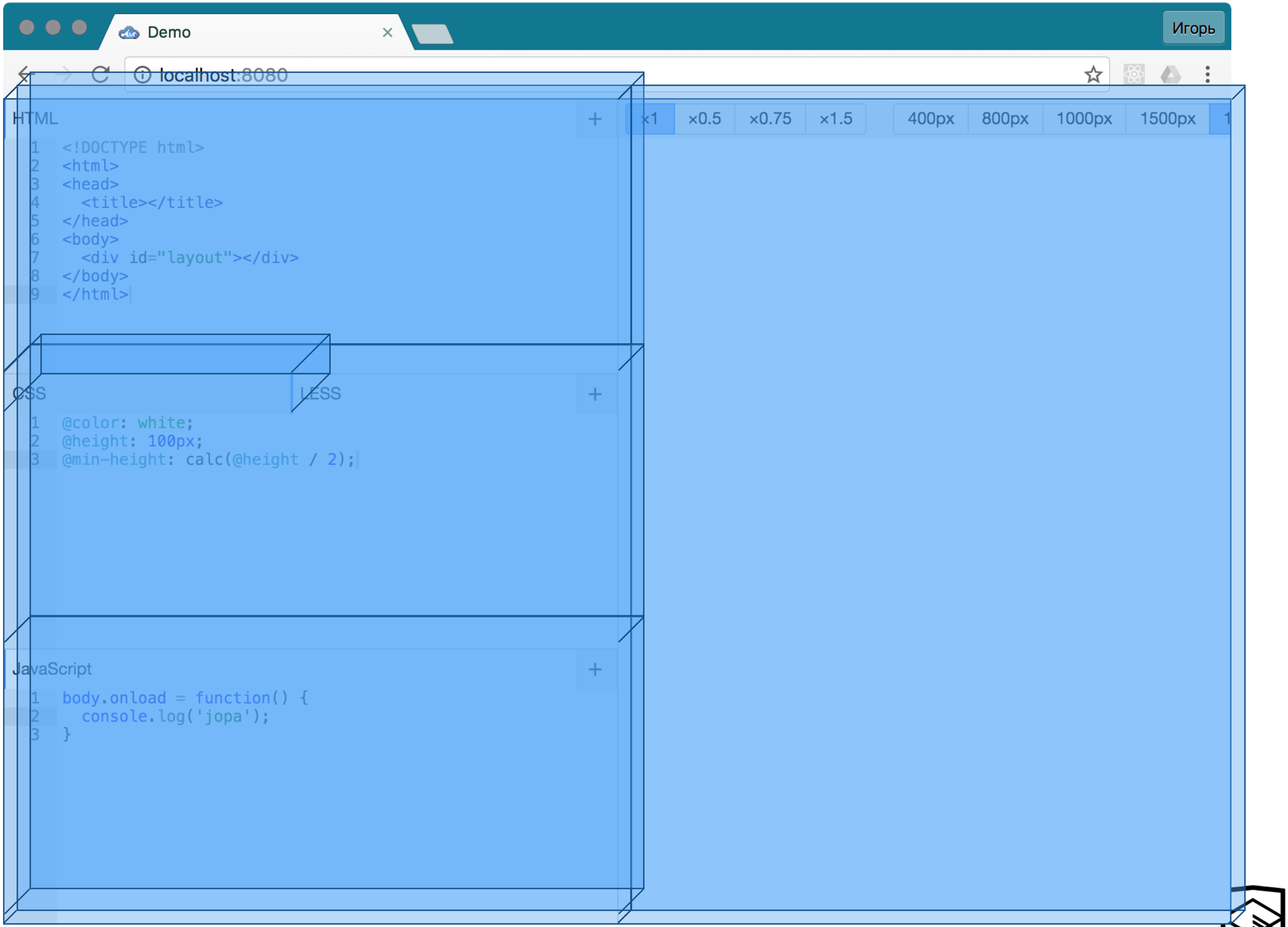
</body>
</html>
CSS
LESS
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
JavaScript
body.onload = function() {
console.log('jopa');
}
×1
×0.5
×0.75
×1.5
400px
800px
1000px
1500px


Demo
Игорь
localhost:8080
HTML
<!DOCTYPE html>
<html>
<head>
<title></title>
</head>
<body>

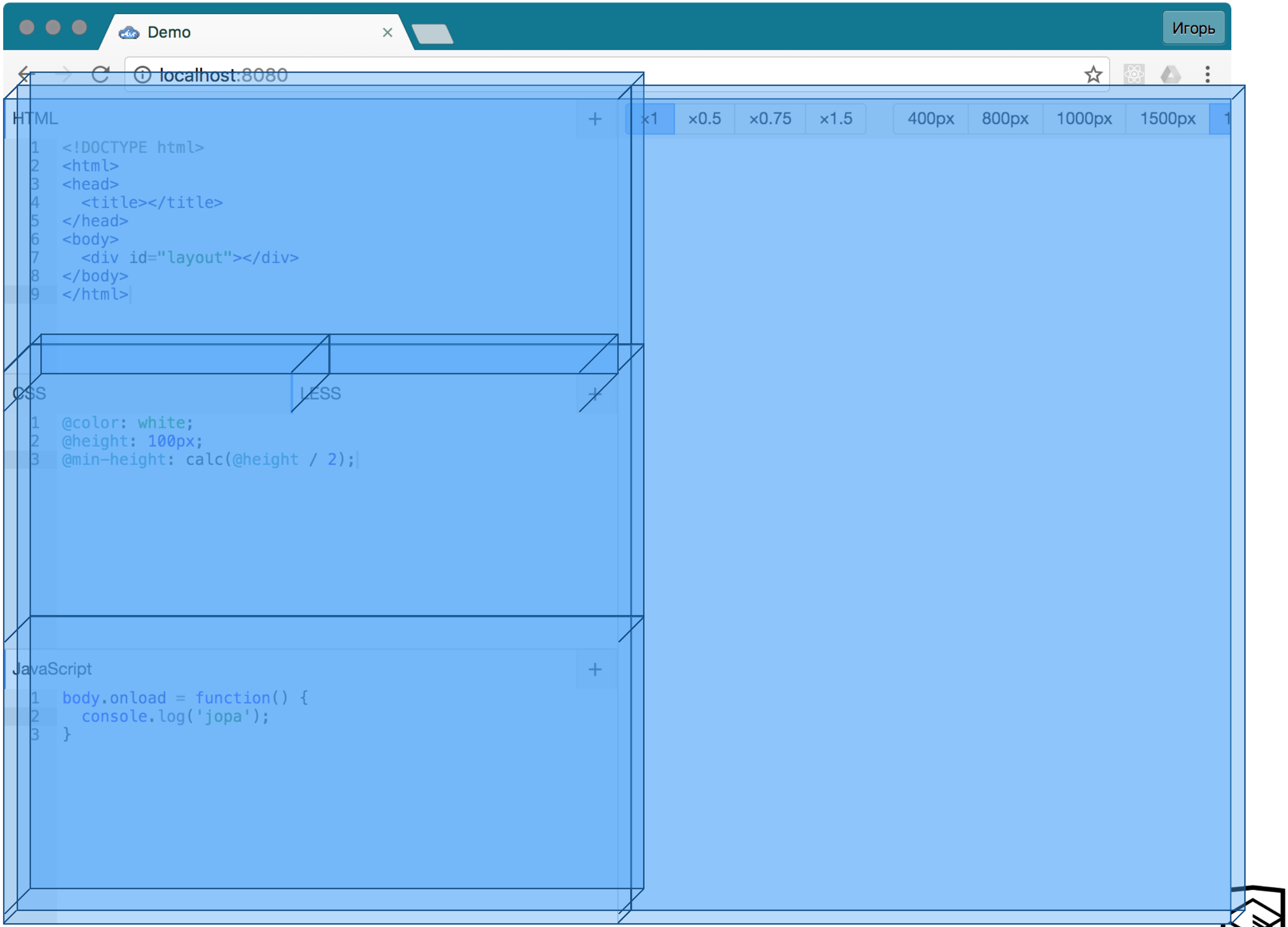
</body>
</html>
CSS
LESS
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
JavaScript
body.onload = function() {
console.log('jopa');
}
×1
×0.5
×0.75
×1.5
400px
800px
1000px
1500px


Demo
Игорь
localhost:8080
HTML
<!DOCTYPE html>
<html>
<head>
<title></title>
</head>
<body>

</body>
</html>
CSS
LESS
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
JavaScript
body.onload = function() {
console.log('jopa');
}
×1
×0.5
×0.75
×1.5
400px
800px
1000px
1500px


Demo
Игорь
localhost:8080
HTML
×1
×0.5
×0.75
×1.5
400px
800px
1000px
1500px
<!DOCTYPE html>
<html>
<head>
<title></title>
</head>
<body>

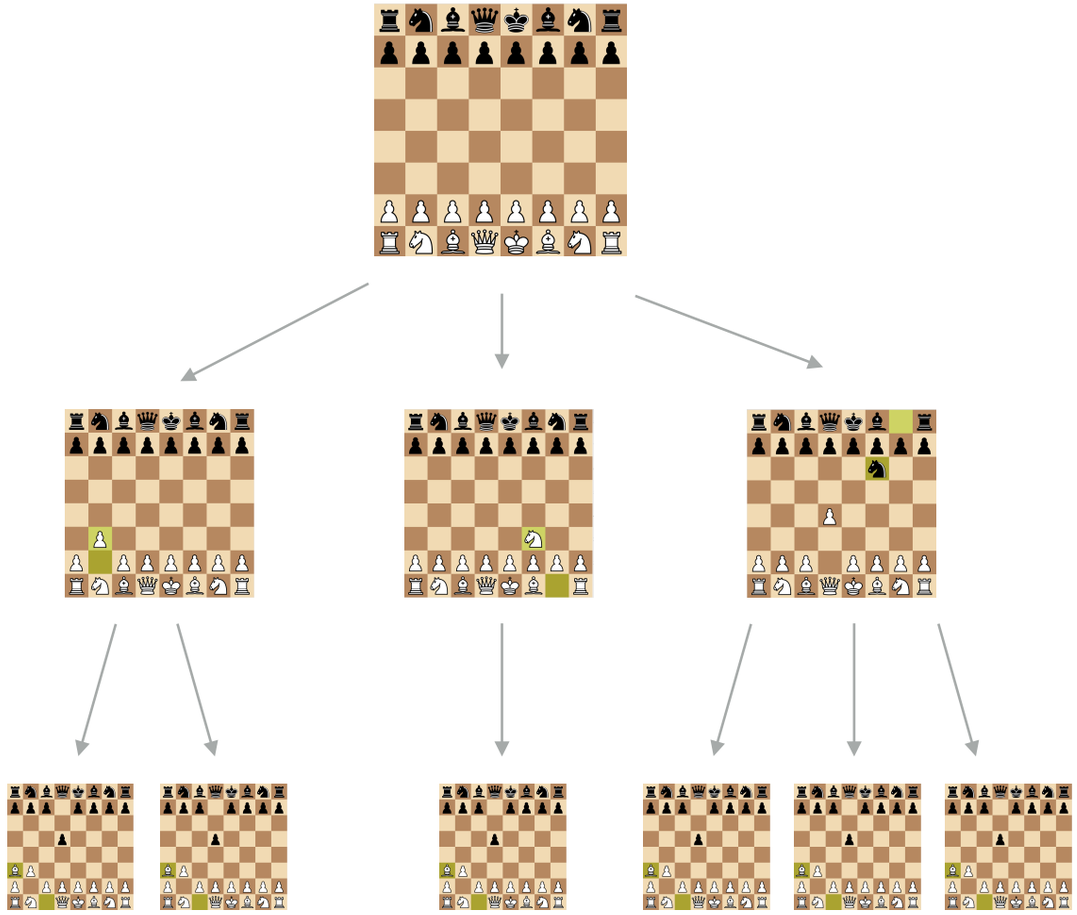
</body>
</html>
CSS
LESS
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
JavaScript
body.onload = function() {
console.log('jopa');
}


Demo
Игорь
localhost:8080
HTML
×1
×0.5
×0.75
×1.5
400px
800px
1000px
1500px
<!DOCTYPE html>
<html>
<head>
<title></title>
</head>
<body>

</body>
</html>
CSS
LESS
@color: white;
@height: 100px;
@min-height: calc(@height / 2);
JavaScript
body.onload = function() {
console.log('jopa');
}

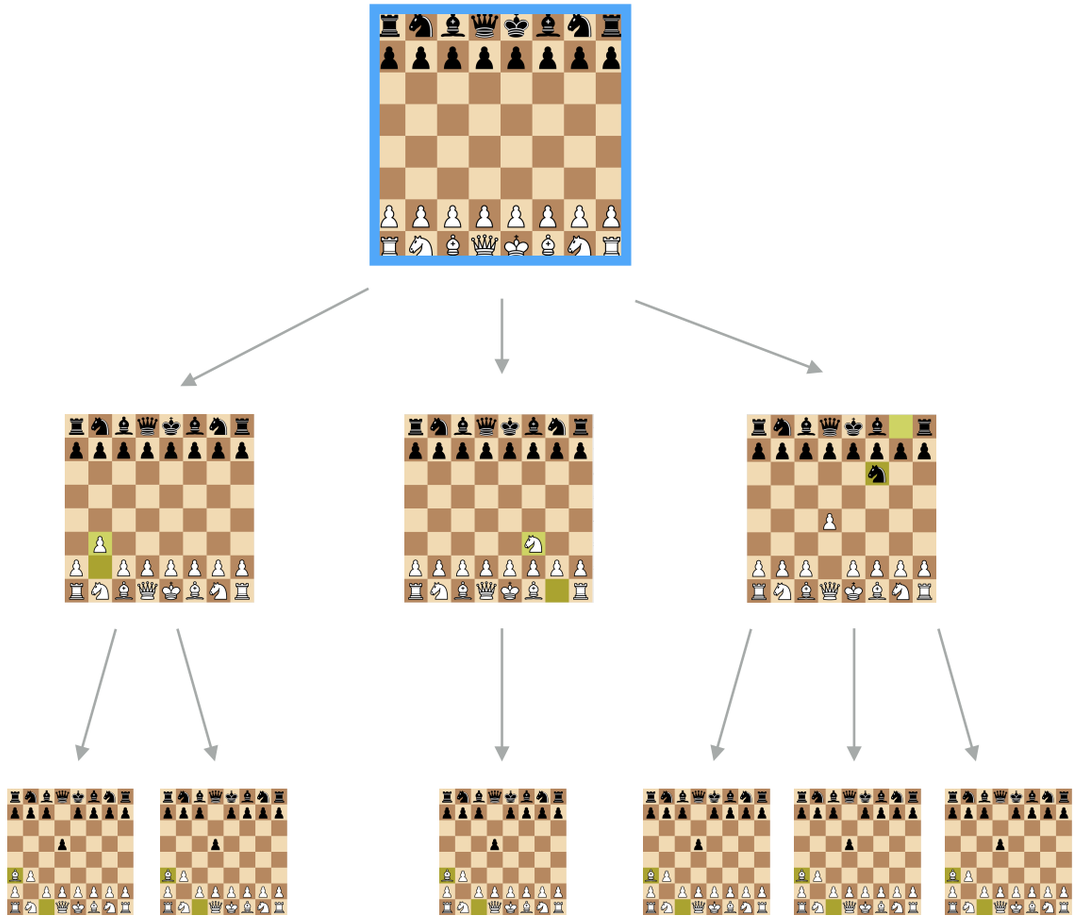
«Лёгкий компьютер» в играх

Обходит дерево сверху вниз и выбирает варианты, в которых не проиграет в ближайшие несколько шагов *(BFS)*



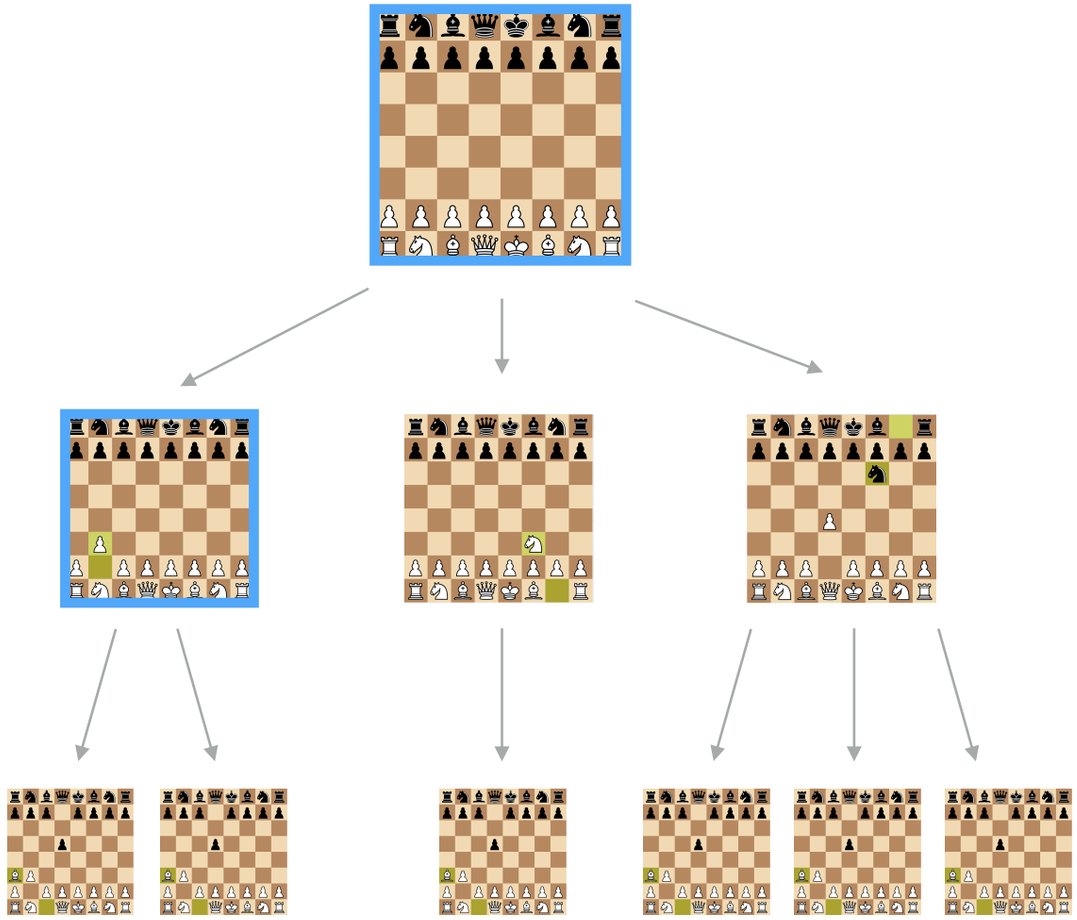
«Лёгкий компьютер» в играх

Обходит дерево сверху вниз и выбирает варианты, в которых не проиграет в ближайшие несколько шагов *(BFS)*



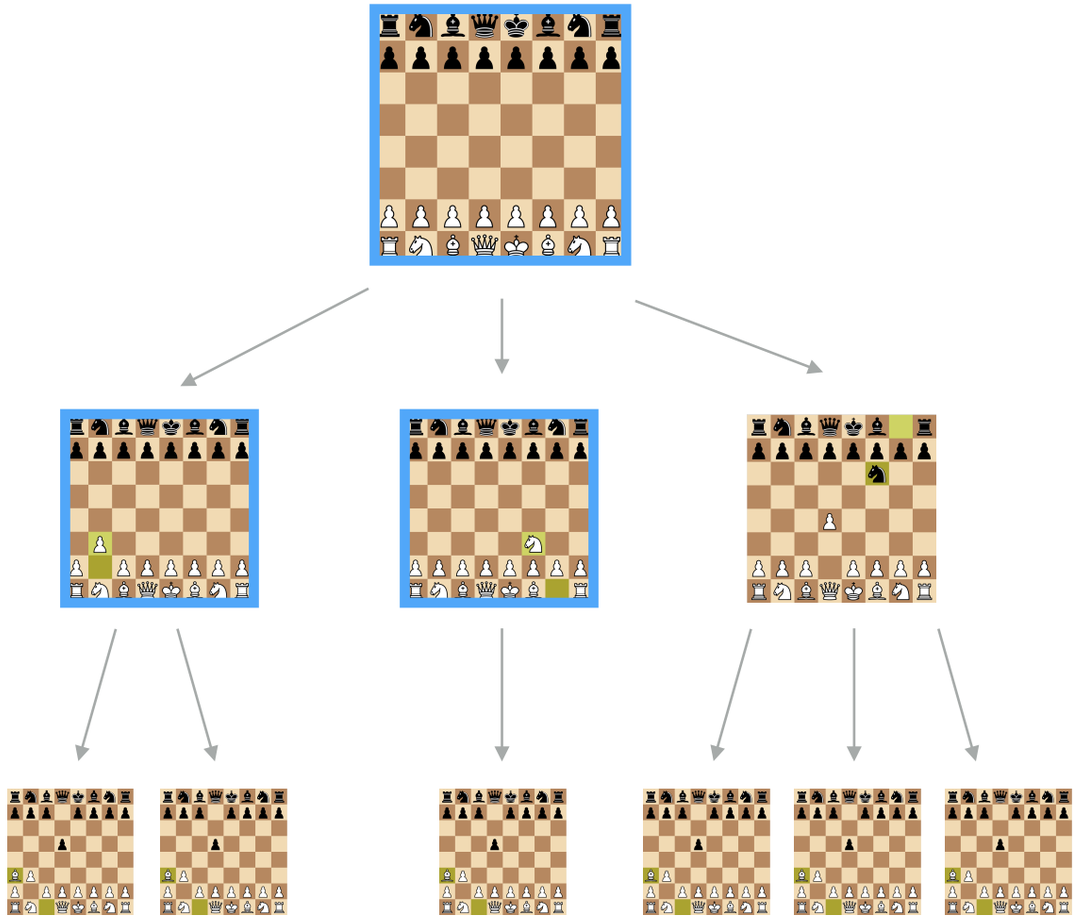
«Лёгкий компьютер» в играх

Обходит дерево сверху вниз и выбирает варианты, в которых не проиграет в ближайшие несколько шагов *(BFS)*



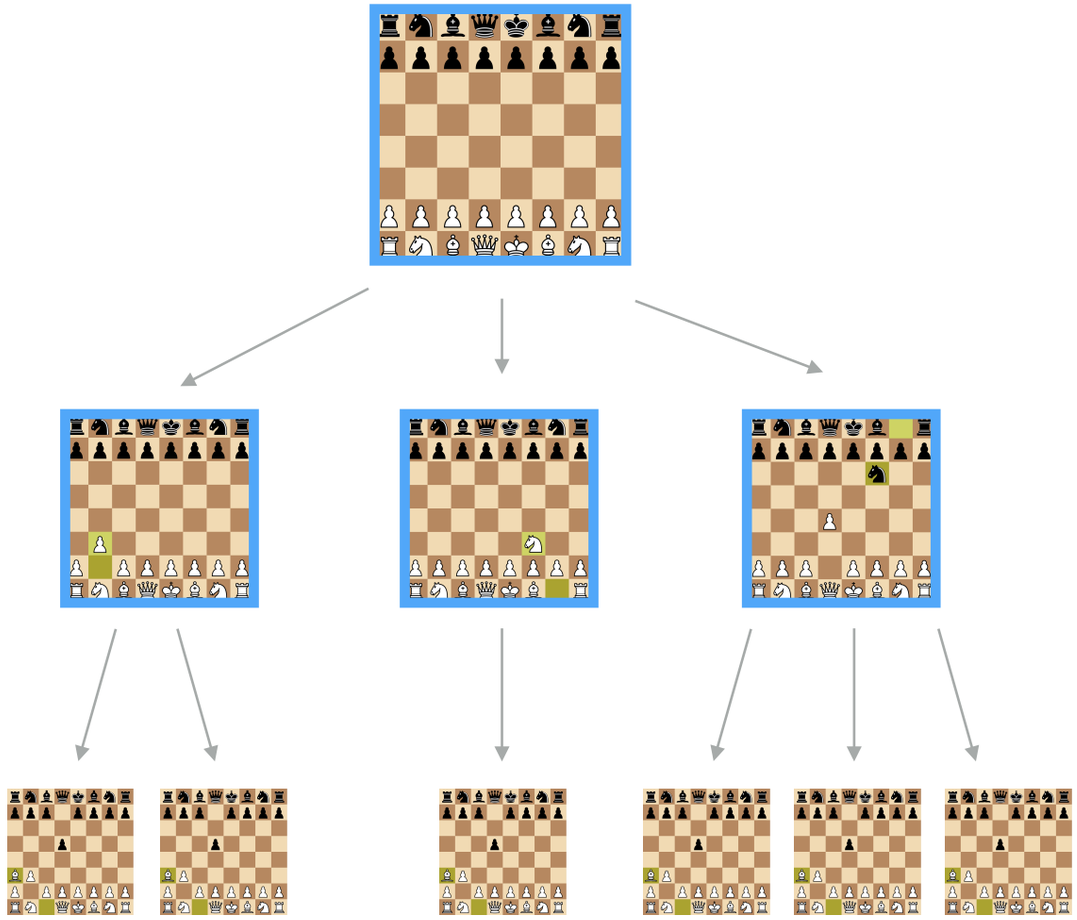
«Лёгкий компьютер» в играх

Обходит дерево сверху вниз и выбирает варианты, в которых не проиграет в ближайшие несколько шагов *(BFS)*



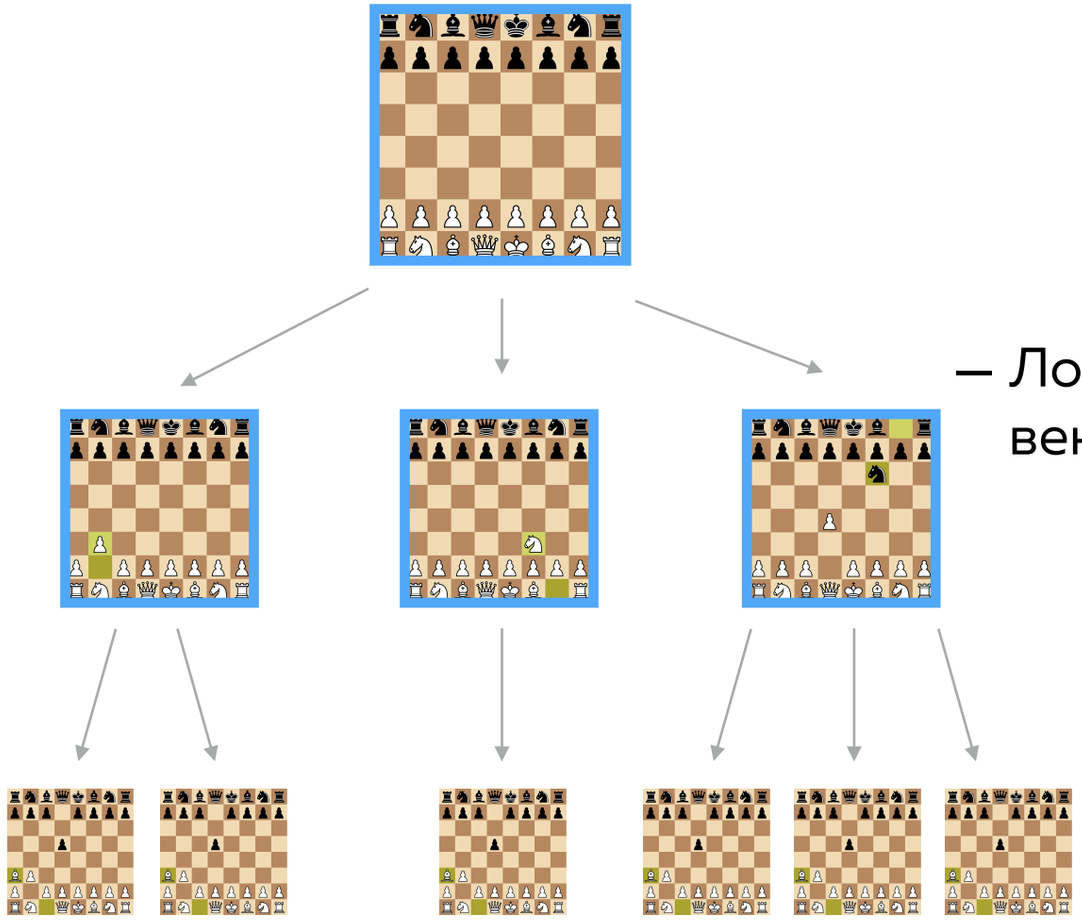
«Лёгкий компьютер» в играх

Обходит дерево сверху вниз и выбирает варианты, в которых не проиграет в ближайшие несколько шагов *(BFS)*



«Лёгкий компьютер» в играх

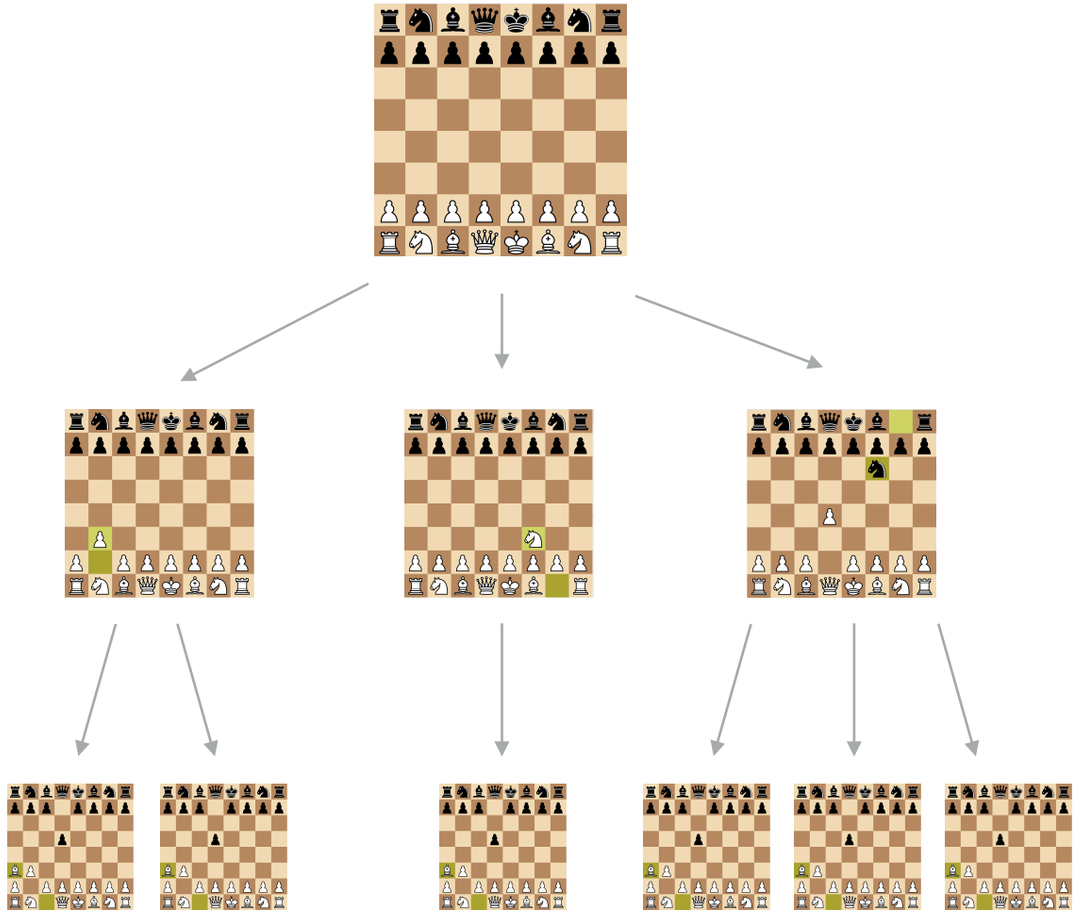
Обходит дерево сверху вниз и выбирает варианты, в которых не проиграет в ближайшие несколько шагов *(BFS)*



— Лошадью ходи, век воли не видать!

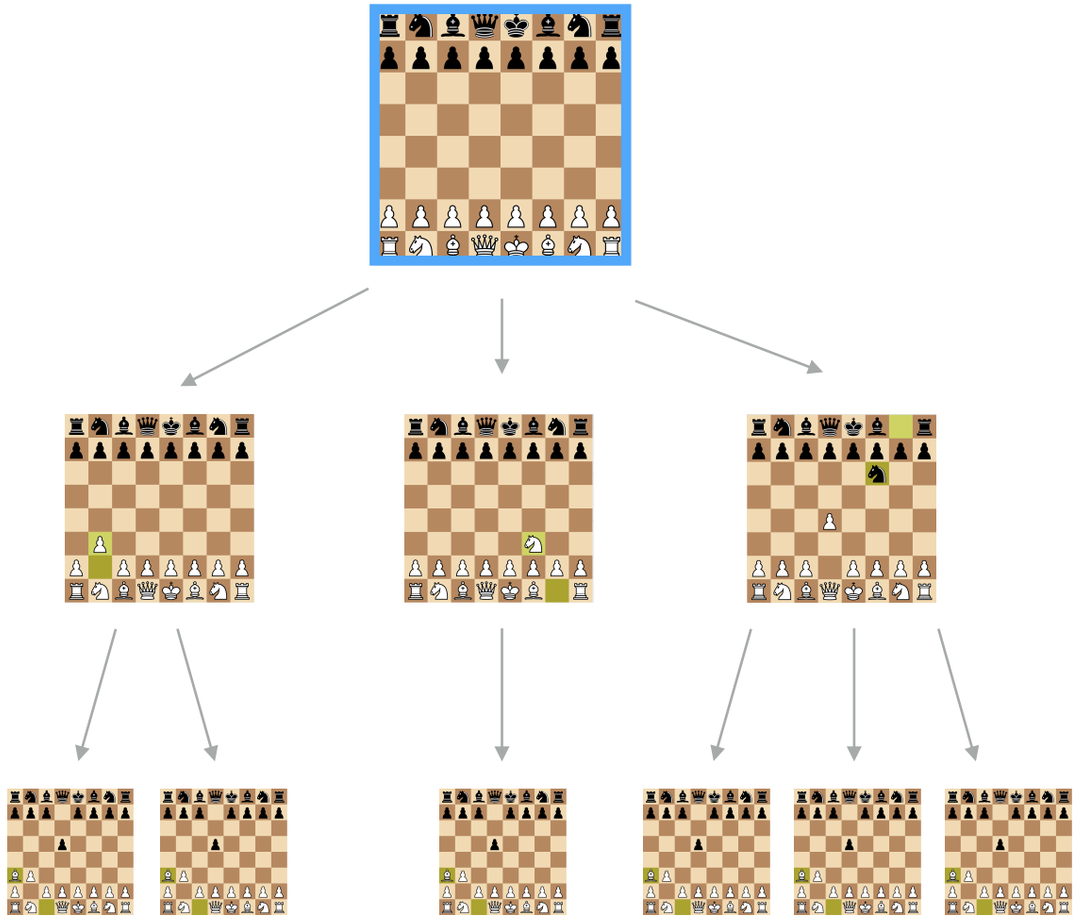
«Сложный компьютер» в играх

Обходит дерево как можно глубже и выбирает выигрышные варианты *(DFS)*



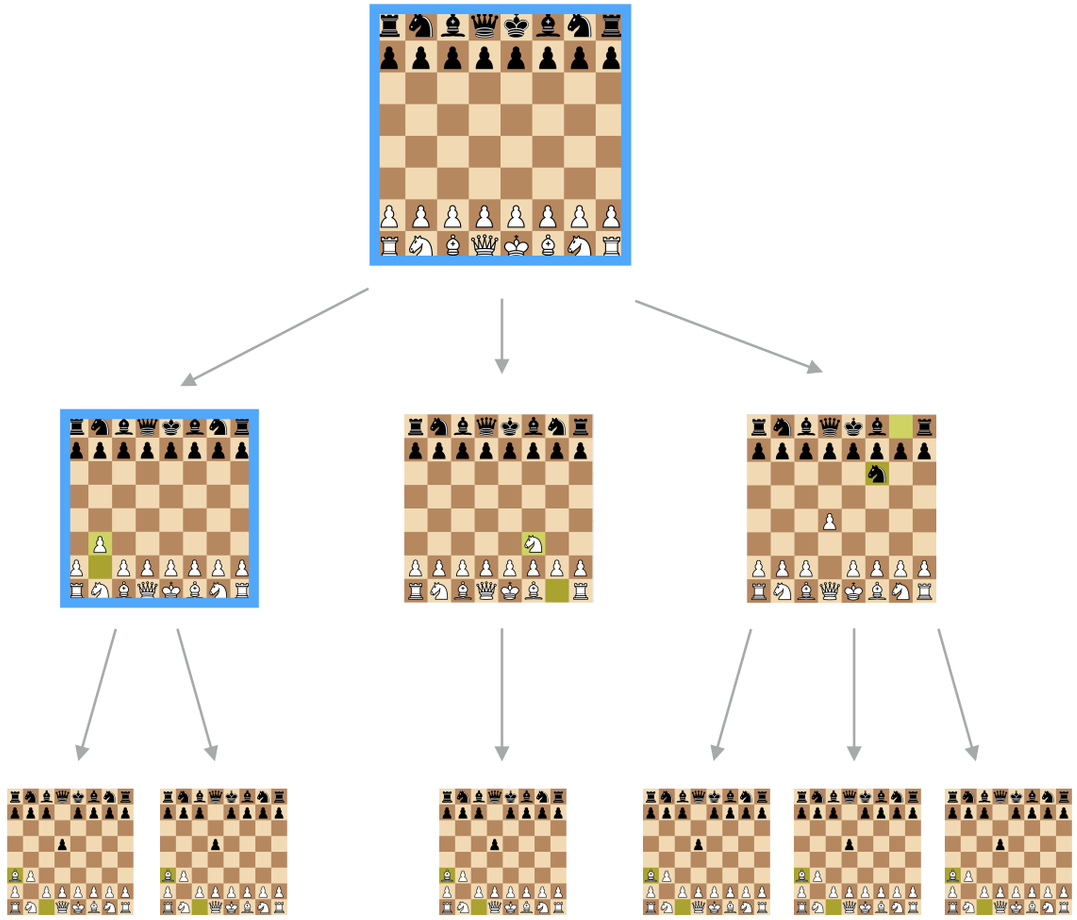
«Сложный компьютер» в играх

Обходит дерево как можно глубже и выбирает выигрышные варианты *(DFS)*



«Сложный компьютер» в играх

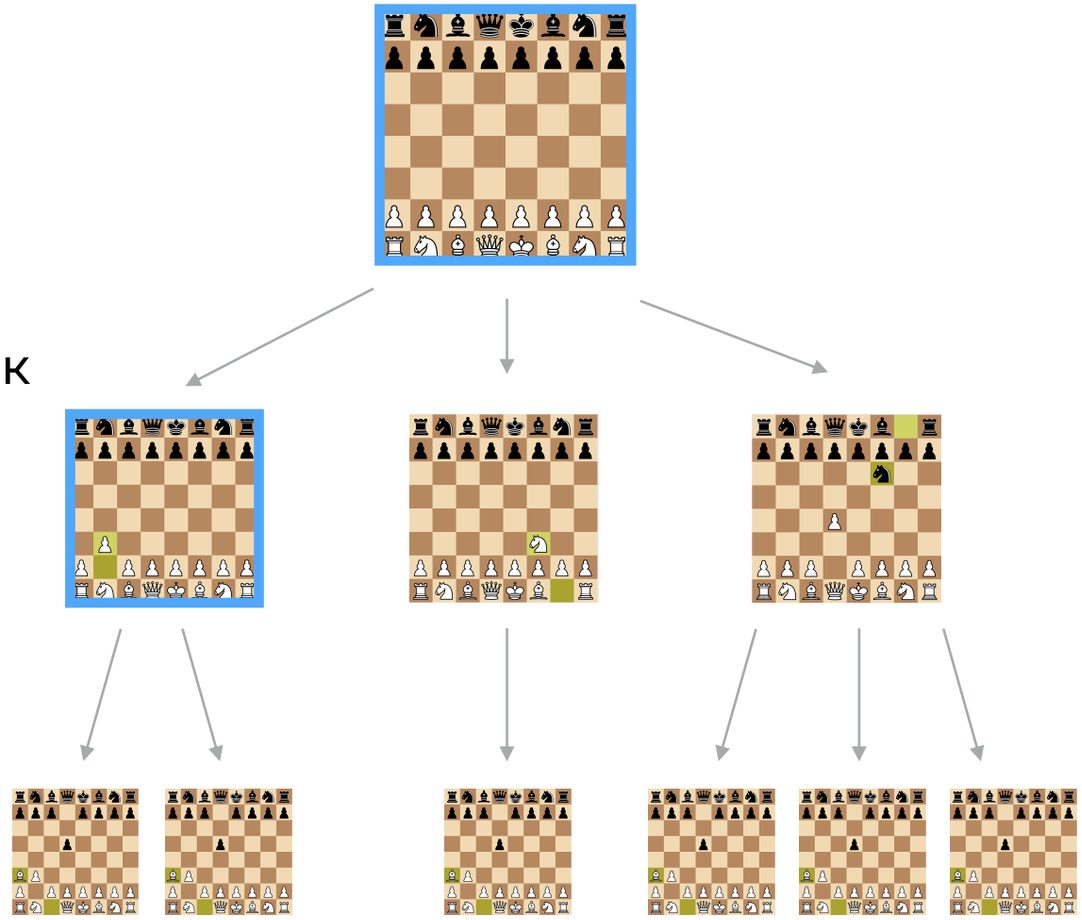
Обходит дерево как можно глубже и выбирает выигрышные варианты *(DFS)*



«Сложный компьютер» в играх

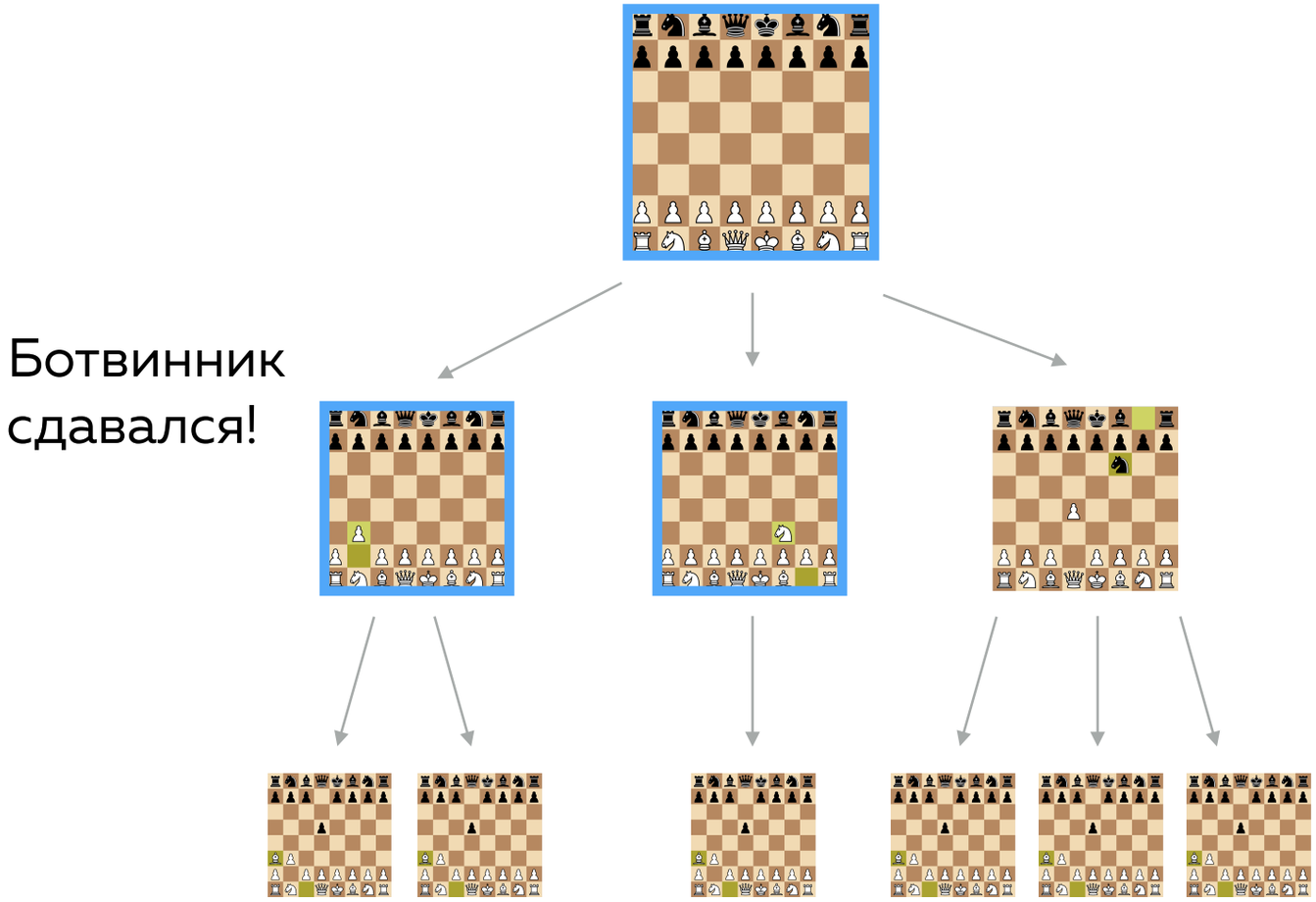
Обходит дерево как можно глубже и выбирает выигрышные варианты *(DFS)*

— Ботвинник сдавался!



«Сложный компьютер» в играх

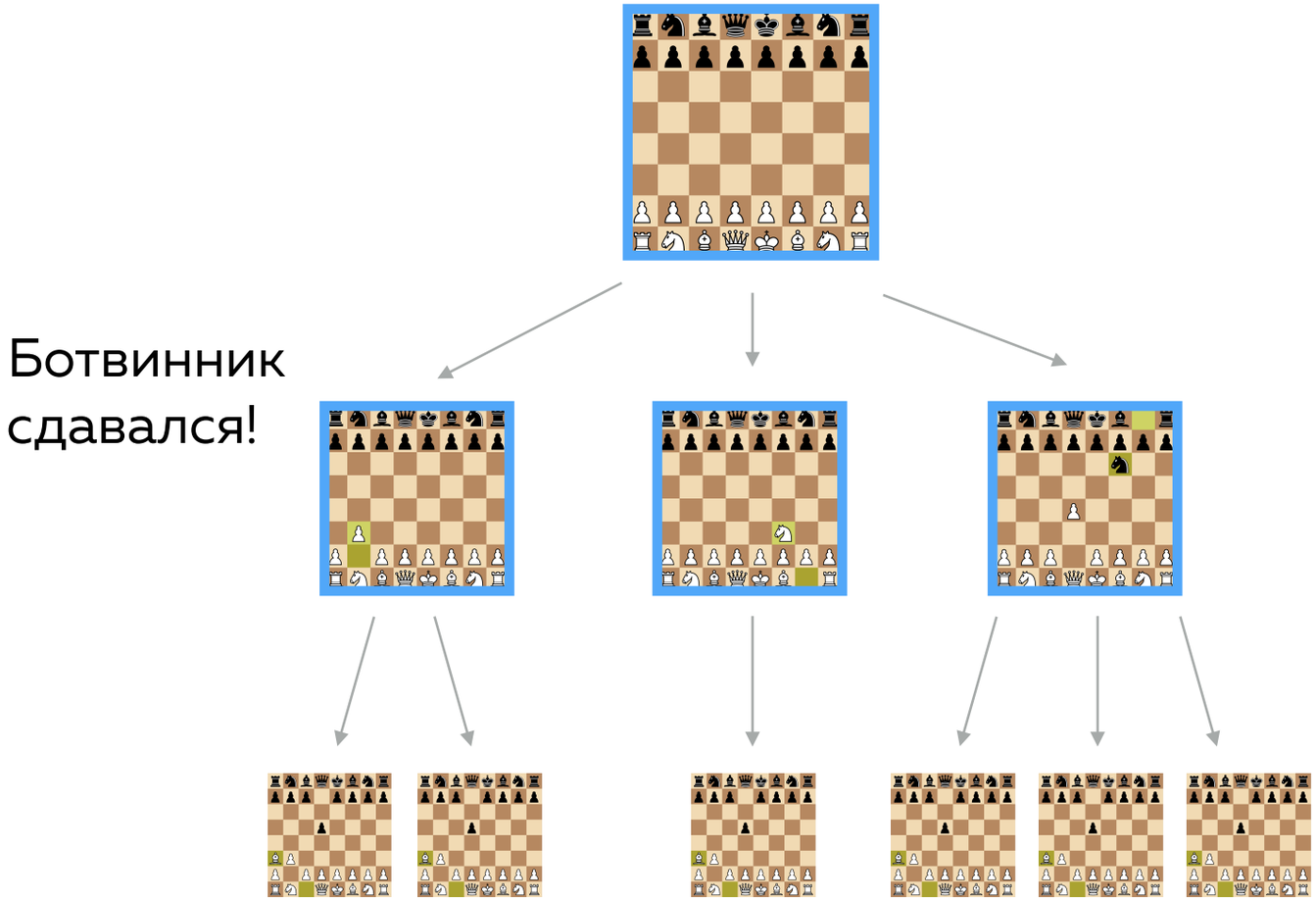
Обходит дерево как можно глубже и выбирает выигрышные варианты *(DFS)*



— Ботвинник сдавался!

«Сложный компьютер» в играх

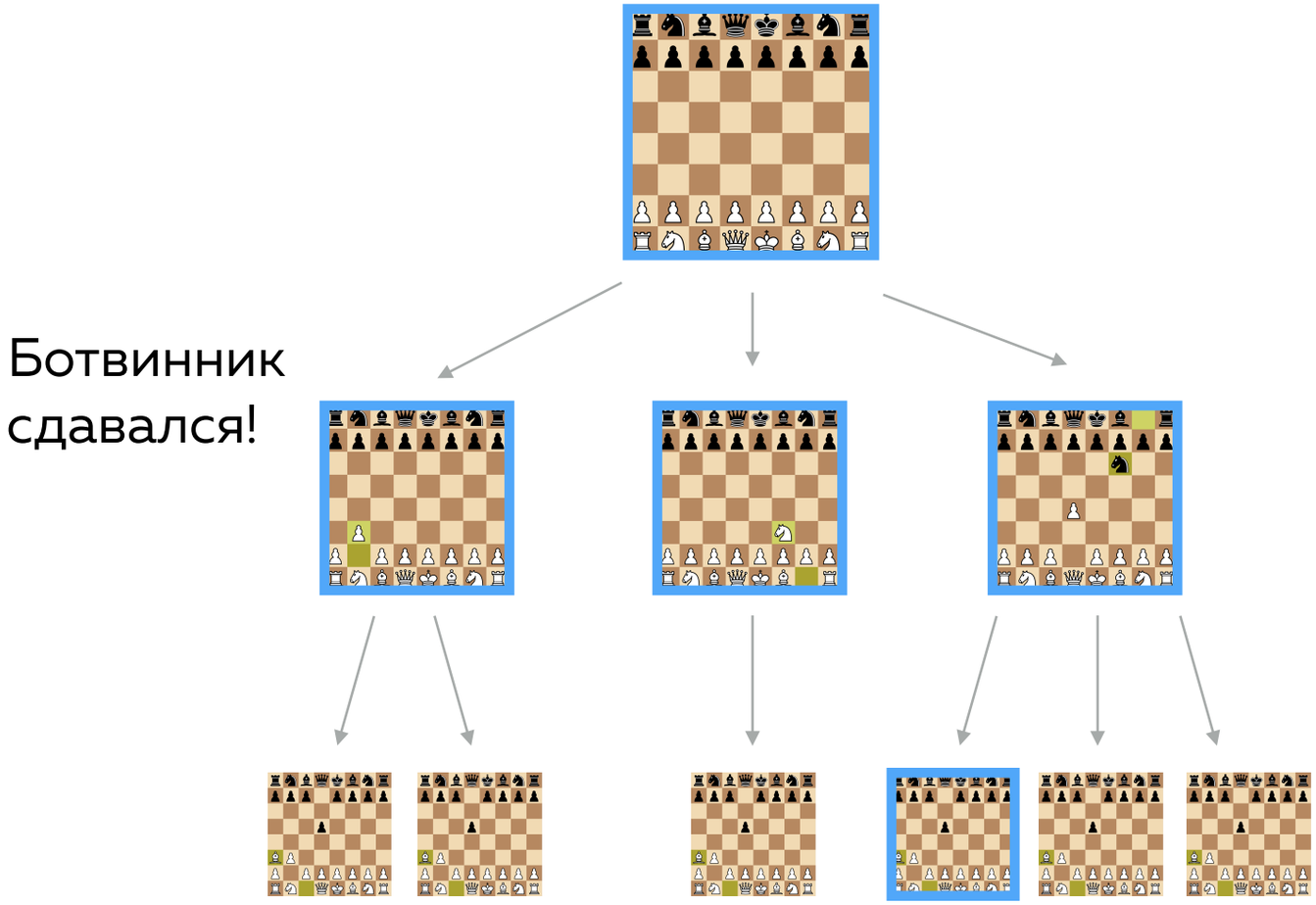
Обходит дерево как можно глубже и выбирает выигрышные варианты *(DFS)*



— Ботвинник сдавался!

«Сложный компьютер» в играх

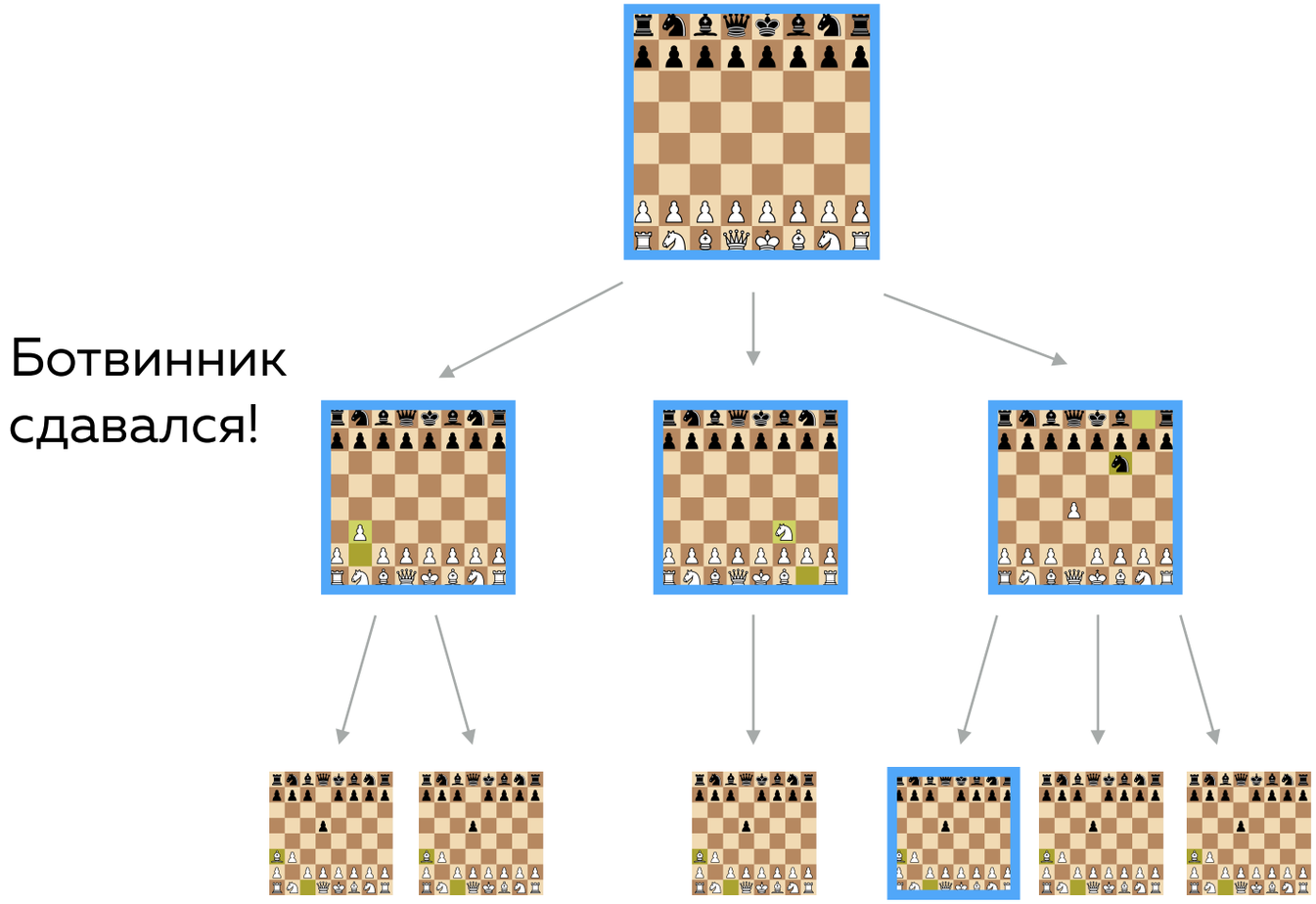
Обходит дерево как можно глубже и выбирает выигрышные варианты *(DFS)*



— Ботвинник сдавался!

«Сложный компьютер» в играх

Обходит дерево как можно глубже и выбирает выигрышные варианты *(DFS)*



— Ботвинник сдавался!

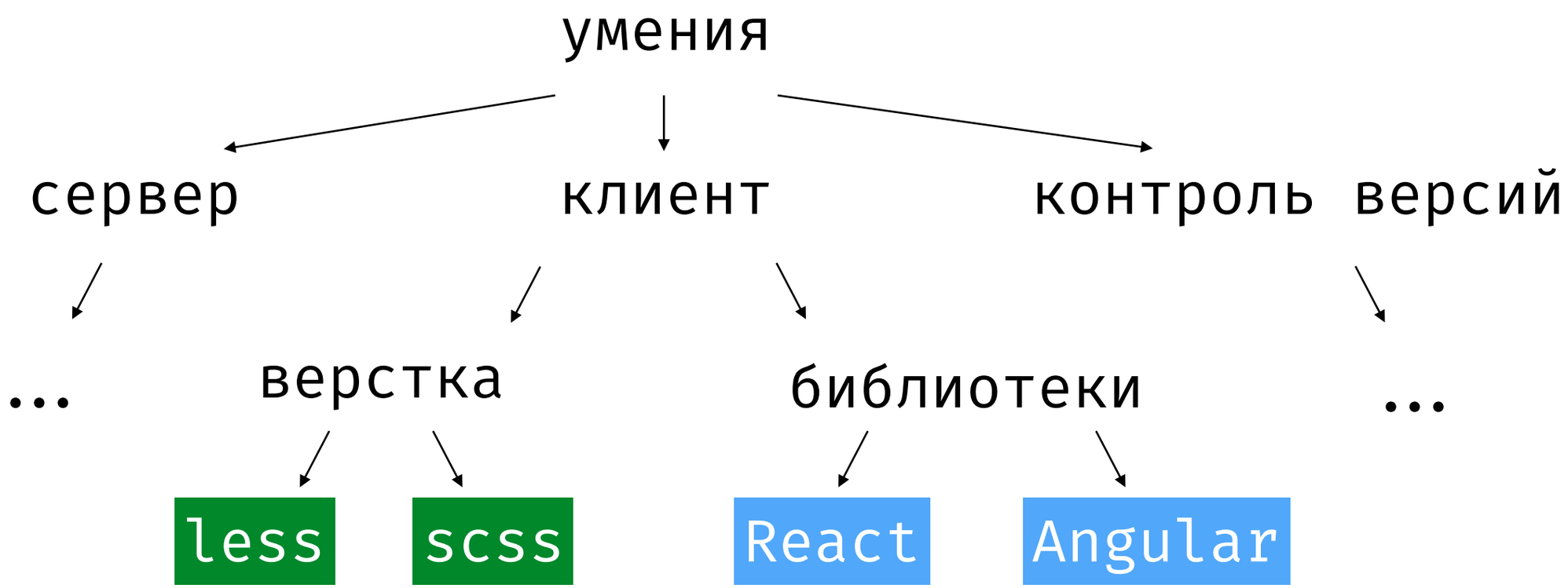
— Так-так, вот тут у меня больше шансов на победу

🌞🔦 Анализ рынка разработки

Статистика требований к разработчикам на основе вакансий «Моего Круга» с помощью библиотеки d3.js

Дерево требований к разработчикам

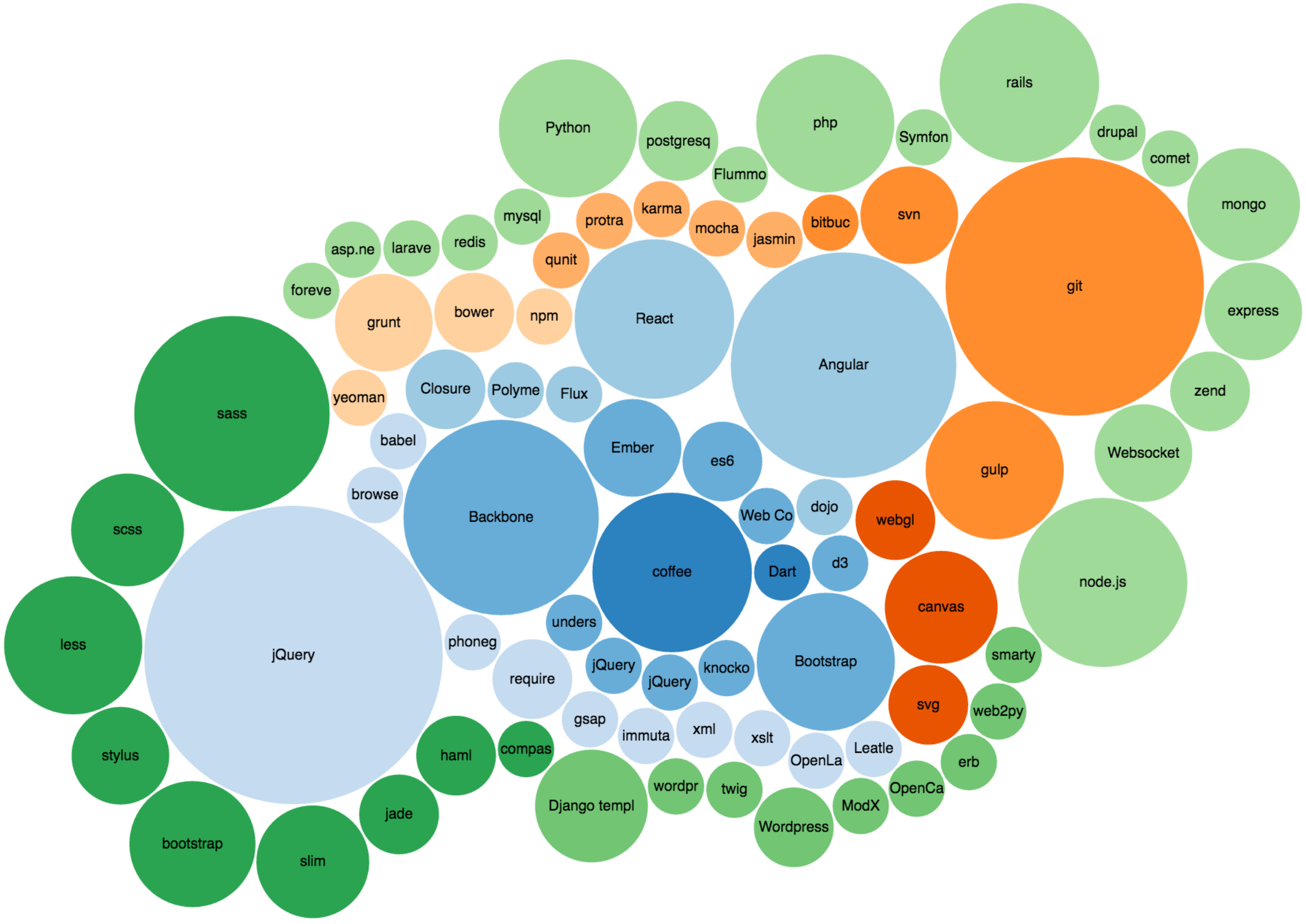
Листьями являются конкретные технологии, а узлами — группы



localhost:8080

Игорь

localhost:8080



Преимущества деревьев

 Преимущества деревьев

- сохраняются и передаются в виде строковых данных

 Преимущества деревьев

- сохраняются и передаются в виде строковых данных
- легко читаются

 Преимущества деревьев

- сохраняются и передаются в виде строковых данных
- легко читаются
- быстро выполняются *(нативный объект JS)*

 Преимущества деревьев

- сохраняются и передаются в виде строковых данных
- легко читаются
- быстро выполняются *(нативный объект JS)*
- не имеют чёткого формата описания, поэтому идеально подстраиваются под задачу

Ищите решения, а не инструменты

Через 20 лет не будет ни Бэйбеля, ни Реакта, ни Вебпака, ни Джаваскрипта. А деревья и битовые карты — будут

Что поизучать

Что поизучать

- `github.com/htmlacademy/bitset.js`
  библиотека от HTML Academy для работы с битовыми массивами в JS — обёртки для добавления, удаления, проверки и получения объектов с состоянием

Что поизучать

- github.com/htmlacademy/bitset.js
  библиотека от HTML Academy для работы с битовыми массивами в JS — обёртки для добавления, удаления, проверки и получения объектов с состоянием

- o0.github.io/trees
  интерактивные демки, примеры для разных областей применения, больше теории, исходники

Что поизучать

- github.com/htmlacademy/bitset.js
  библиотека от HTML Academy для работы с битовыми массивами в JS — обёртки для добавления, удаления, проверки и получения объектов с состоянием

- o0.github.io/trees
  интерактивные демки, примеры для разных областей применения, больше теории, исходники

- github.com/thejameskyle/itsy-bitsy-data-structures
  репозиторий, с интересным и подробным рассказом про алгоритмы и структуры данных на JS

– Спасибо!

tutors@htmlacademy.ru

Наставничество — отличный способ найти себе в команду новичка и воспитать его так, как считаешь должным на готовой продуманной инфраструктуре или просто поделиться опытом