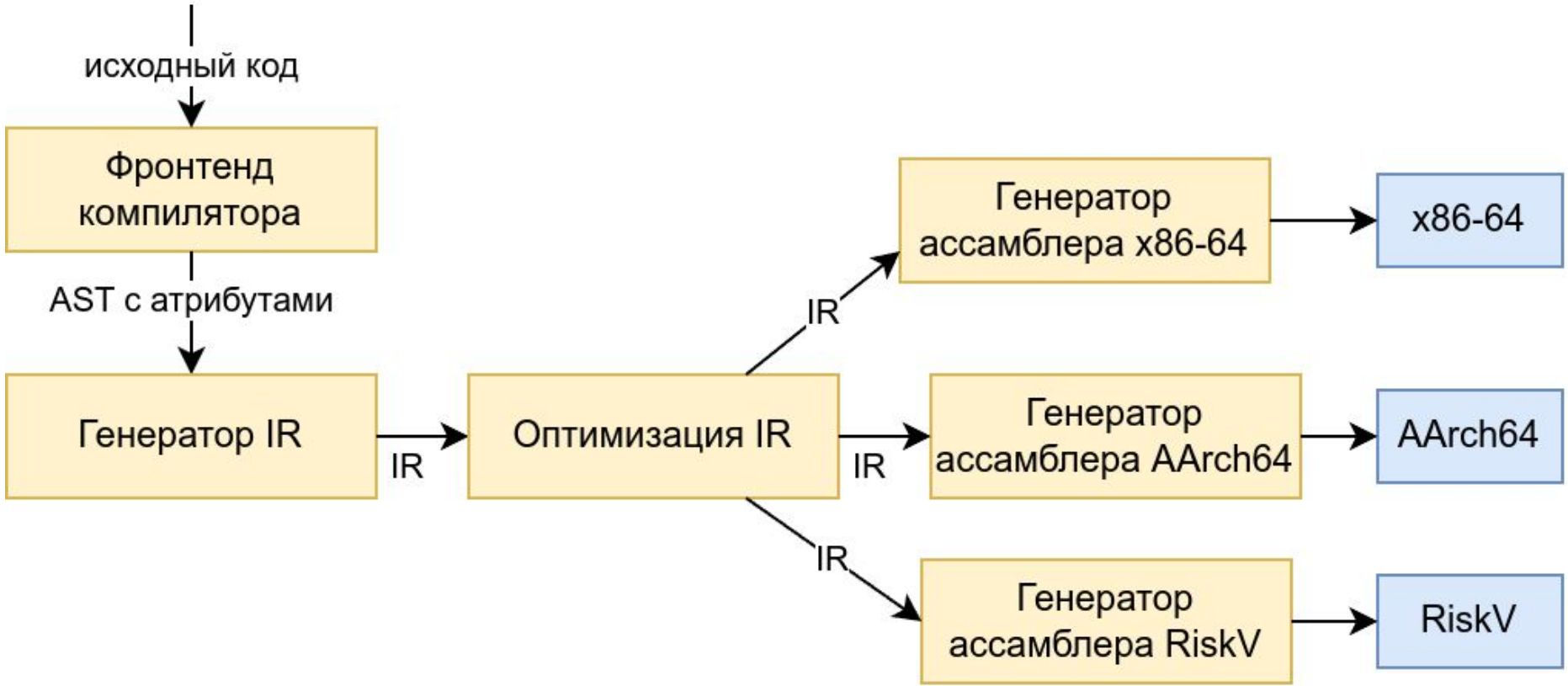
Промежуточный код

Лекция №3
Теория языков программирования

Оптимизации промежуточного кода



Сравнение

Виртуальная машина (VM)

- Компилятор создаёт байт-код
- Виртуальная машина исполняет байт-код (интерпретация)

Промежуточный код (IR)

- Компилятор создаёт промежуточный код
- Бэкенд компилирует IR в машинный код (компиляция)

Вопросы лекции

1. Как проектировать промежуточный код?
2. Как проектировать байт-код?

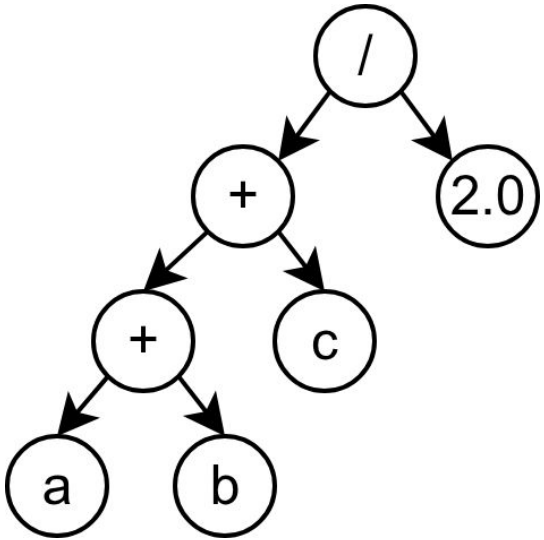
От исходного кода к AST

Площадь треугольника по формуле Герона

```
float triangleSquare(
  float a, float b, float c
) {
  float p = (a + b + c) / 2.0;
  return sqrtf(
    p * (p - a)
    * (p - b) * (p - c)
  );
}
```

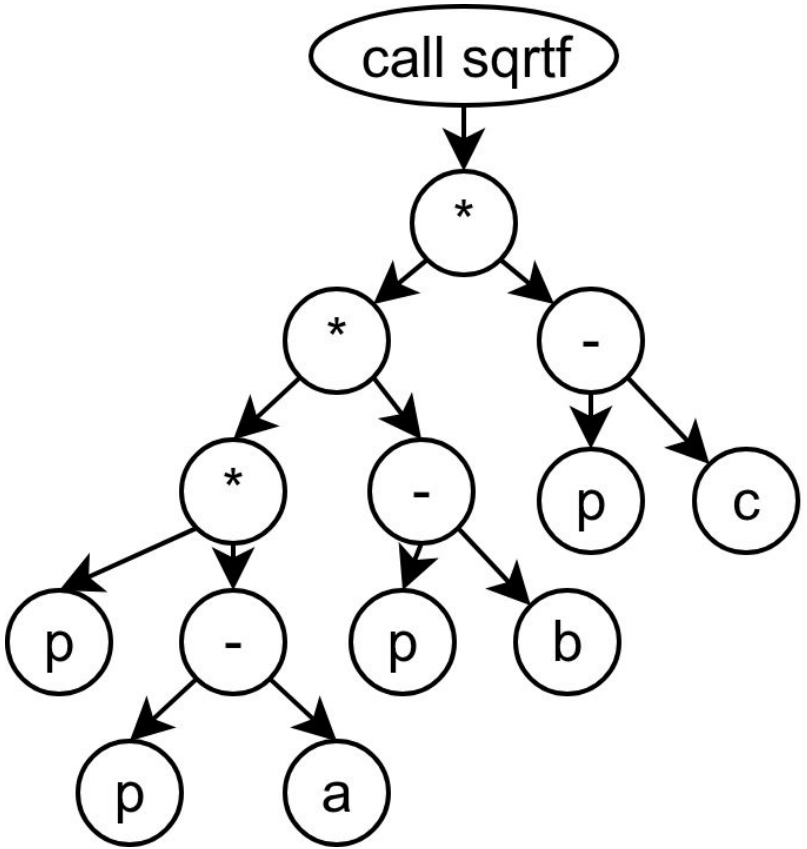
Площадь треугольника по формуле Герона

```
float triangleSquare(
  float a, float b, float c
) {
  float p = (a + b + c) / 2.0;
  return sqrtf(
    p * (p - a)
    * (p - b) * (p - c)
  );
}
```



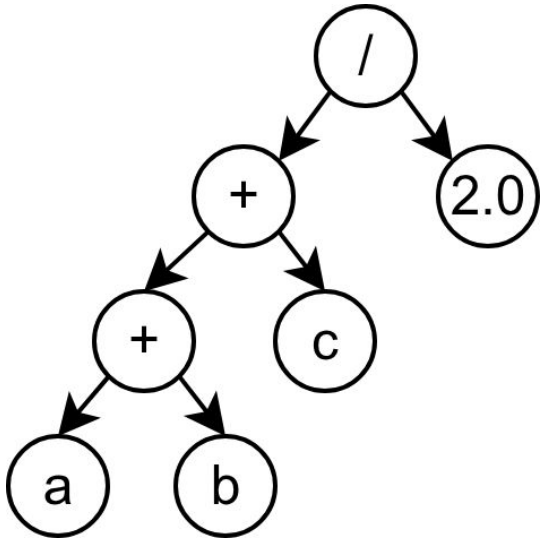
Площадь треугольника по формуле Герона

```
float triangleSquare(
  float a, float b, float c
) {
  float p = (a + b + c) / 2.0;
  return sqrtf(
    p * (p - a)
    * (p - b) * (p - c)
  );
}
```



Трёхадресный код

Трёхадресный код



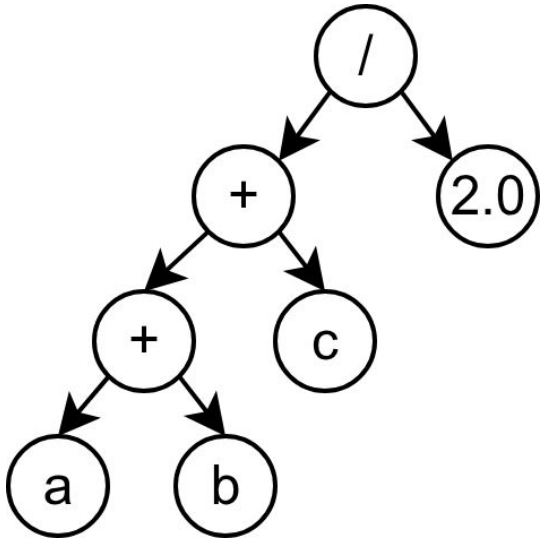

```
float t1 = a + b;
float t2 = t1 + c;
float p = t2 / 2.0;
```

Трёхадресный код

Одна инструкция содержит:

1. Один оператор
2. До двух операндов
3. Один результат

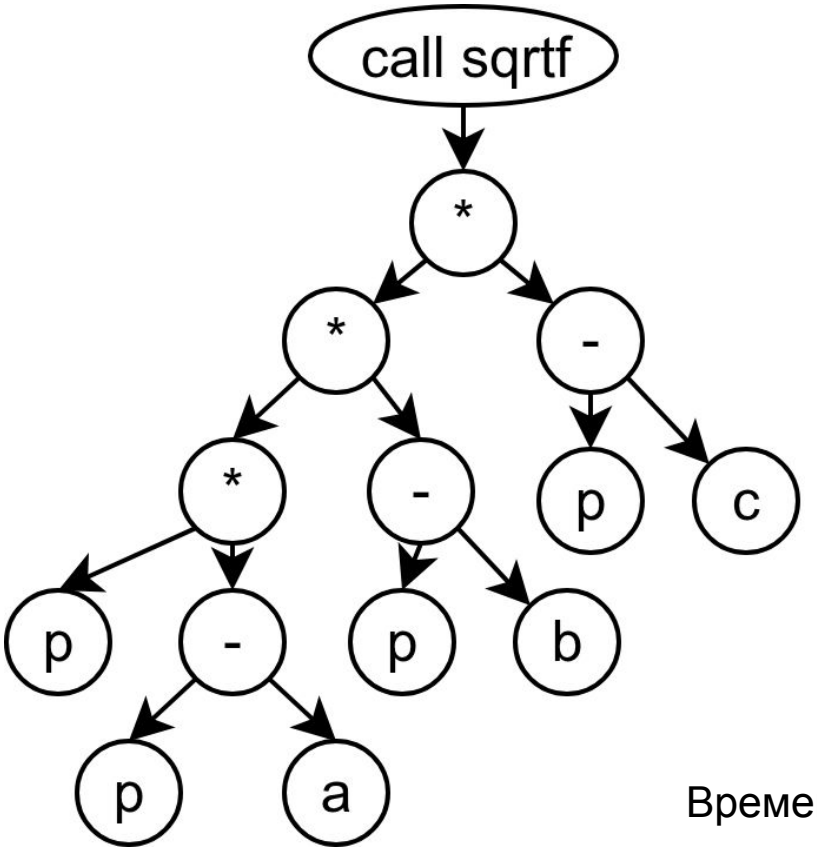
Трансляция в трёхадресный код




```
float t1 = a + b;
float t2 = t1 + c;
float p = t2 / 2.0;
```

Временные переменные генерируются компилятором

Трансляция в трёхадресный код




```
float t4 = p - a;
float t5 = p - b;
float t6 = p - c;

float t7 = p * t4;
float t8 = t7 * t5;
float t9 = t8 * t6;

float t10 = sqrt(t9);
```

Временные переменные генерируются компилятором

Структура трёхадресного кода — четвёрки

```
typedef union Operand {
    Name name;
    Constant constant;
} Operand;

typedef struct Instruction {
    OpCode operation;
    Operand left;
    Operand right;
    Name result;
} Instruction;
```

- OpCode — перечислимый тип
- Constant — значение литерала
- Name — имя

Компромиссы трёхадресного кода

Как представить вызов функции?

```
float triangleSquare(float a, float b, float c)
```

Неудачный вариант:

```
t := call a, b, c
```

Компромиссы трёхадресного кода

Как представить вызов функции?

```
float triangleSquare(float a, float b, float c)
```

Решение №1 — дополнительная инструкция:

```
param a
param b
param c
t := call
```

Компромиссы трёхадресного кода

Как представить вызов функции?

```
float triangleSquare(float a, float b, float c)
```

Решение №2 — отойти от строгих правил:

```
t := call(a, b, c)
```

Двухадресный код

Двухадресные команды

```
# Вычисляем p = (a + b + c) * 0.5
movaps    %xmm3, %xmm0       # xmm0 = a
addss     %xmm4, %xmm0       # xmm0 = a + b
addss     %xmm5, %xmm0       # xmm0 = a + b + c
movss     .LC0(%rip), %xmm6  # xmm6 = 0.5
mulss     %xmm6, %xmm0       # xmm0 = p
```

Двухадресные команды

```
# Вычисляем p = (a + b + c) * 0.5
movaps   %xmm3, %xmm0       # xmm0 = a
addss    %xmm4, %xmm0       # xmm0 = a + b
addss    %xmm5, %xmm0       # xmm0 = a + b + c
movss    .LC0(%rip), %xmm6  # xmm6 = 0.5
mulss    %xmm6, %xmm0       # xmm0 = p
```

Четвёрки: `(opcode, operand, operand, result)`

Тройки: `(opcode, operand, operand)`

Двухадресные команды

```
# Вычисляем p = (a + b + c) * 0.5
movaps    %xmm3, %xmm0       # xmm0 = a
addss     %xmm4, %xmm0       # xmm0 = a + b
addss     %xmm5, %xmm0       # xmm0 = a + b + c
movss     .LC0(%rip), %xmm6  # xmm6 = 0.5
mulss     %xmm6, %xmm0       # xmm0 = p
```

Четвёрки: `(opcode, operand, operand, result)`

Тройки: `(opcode, operand, operand)`

Вопрос №1: а где будет результат операции?

Двухадресные команды

```
# Вычисляем p = (a + b + c) * 0.5
movaps   %xmm3, %xmm0       # xmm0 = a
addss    %xmm4, %xmm0       # xmm0 = a + b
addss    %xmm5, %xmm0       # xmm0 = a + b + c
movss    .LC0(%rip), %xmm6  # xmm6 = 0.5
mulss    %xmm6, %xmm0       # xmm0 = p
```

Четвёрки: `(opcode, operand, operand, result)`

Тройки: `(opcode, operand, operand)`

Вопрос №2: почему современные ассемблеры — двухадресные?

Байт-код стековой машины

Байт-код с одним операндом

```
public class Instruction {

    public Instruction(InstructionCode code, Value value) {

        Code = code;

        Operand = value;

    }

    public InstructionCode Code { get; }

    public Value Operand { get; }

}
```

Вопрос: куда исчезли остальные операнды и результат?

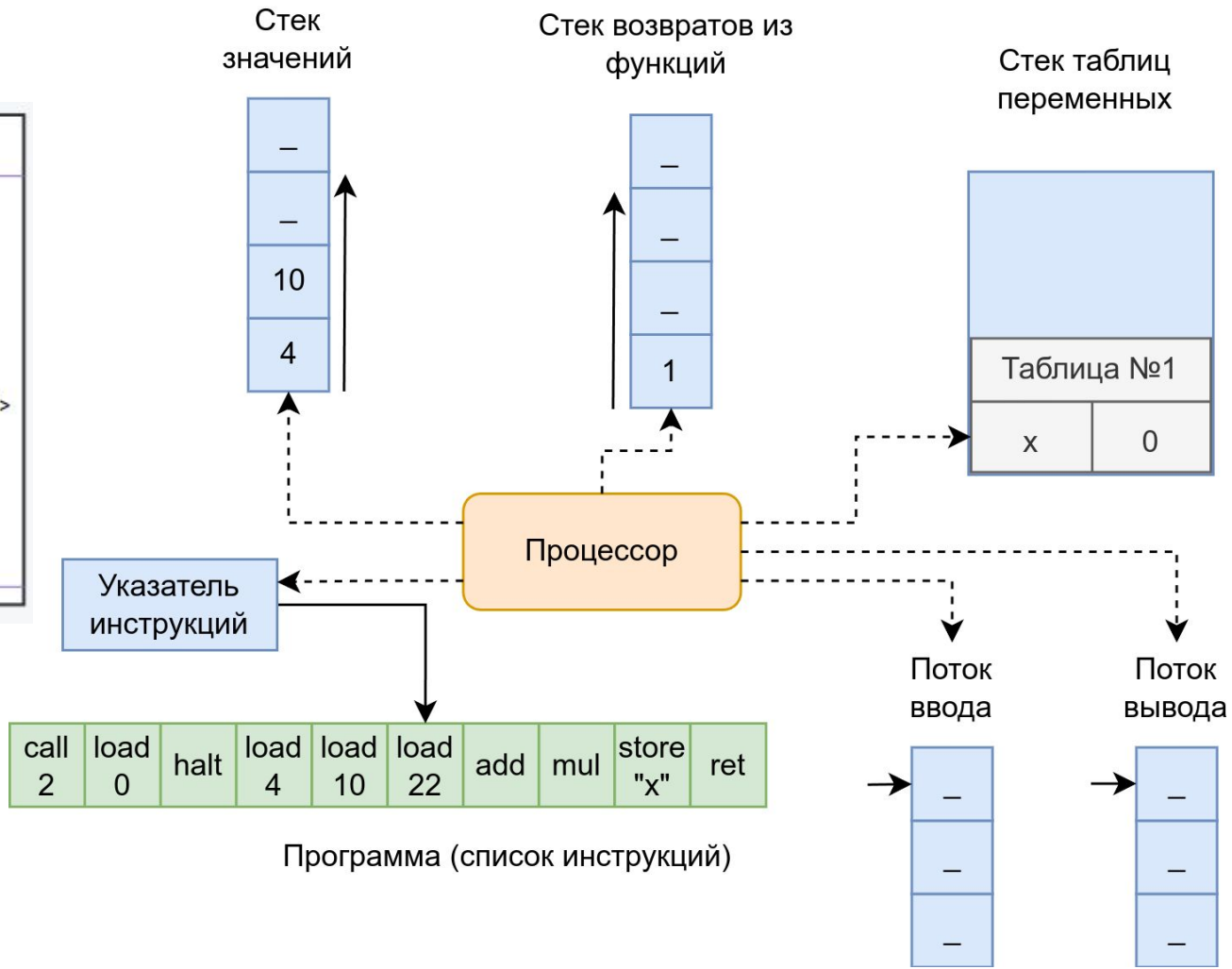
TigerVM

TigerVm

- _evaluationStack: Stack<Value>
- _returnStack: Stack<ReturnContext>
- _variables: VariablesTable?
- _instructionPointer: int
- _instructions: IReadOnlyList<Instruction>
- _builtinFunctions: BuiltinFunctions
- _exitCode: int
- _result: Value

VariablesTable

- _parent: VariablesTable
- _variables: Dictionary<string, Value>
- _depth: int



Категории инструкций в байт-коде

1. Операции со стеком
2. Доступ к переменными
3. Бинарные и унарные операции
   - арифметические
   - сравнения
   - логические
4. Безусловный переход
5. Условные переходы
6. Вызовы функций и возврат
7. Доступ к элементам массива
8. Доступ к полям структуры
9. Управление областями видимости

Примеры инструкций

Операции со стеком:

1. `Push <operand>`
2. `Pop`
3. `Duplicate`

Доступ к переменным:

1. `DefineVar (или Alloca)`
2. `StoreVar`
3. `LoadVar`

Примеры инструкций

Арифметические операции:

1. Add
2. Subtract
3. Multiply
4. Divide
5. Modulo
6. Power
7. Sqrt
8. Negate

Логические операции:

1. And
2. Or
3. Xor
4. Not

Примеры инструкций

Операции сравнения:

1. `Less`
2. `LessOrEqual`
3. `NotEqual`
4. `Equal`

Безусловный переход:

1. `Jump <адрес>`

Условные переходы:

2. `JumpIfTrue <адрес>`
3. `JumpIfFalse <адрес>`

Примеры инструкций

Поддержка функций:

1. `CallBuiltin <имя>`
2. `Call <адрес>`
3. `Return`

Поддержка областей видимости:

4. `PushVars <число>`
5. `PopVars`

Управление программой:

1. `StoreResult`
2. `Halt`

Поддержка составных типов

Поддержка массивов:

```
1. CreateArray    –   intArray[numsCount] of 7
2. LoadArray      –   printi(nums[i])
3. StoreArray     –   nums[0] := 2;
```

Поддержка структур:

```
4. InitField      –   Point{x = 10, y = 20}
5. LoadField      –   printi(p.x);
6. StoreField     –   points[0].x := 1;
```

Выбор таких инструкций сильно зависит от целевого языка

Инструкции перехода

Инструкции перехода

```
if x then
  print("True branch")
else
  print("False branch")
```

```
0.  LoadVar, "x"
1.  JumpIfTrue, 5
2.  Push, "False branch"
3.  CallBuiltin, "print"
4.  Jump, 7
5.  Push, "True branch"
6.  CallBuiltin, "print"
7.  Halt
```

Инструкции перехода

```
if x then
  print("True branch")
else
  print("False branch")
```

Допустим, x=0

```
0.  LoadVar, "x"
1.  JumpIfTrue, 5
2.  Push, "False branch"
3.  CallBuiltin, "print"
4.  Jump, 7
5.  Push, "True branch"
6.  CallBuiltin, "print"
7.  Halt
```

Инструкции перехода

```
if x then
  print("True branch")
else
  print("False branch")
```

Допустим, x=1

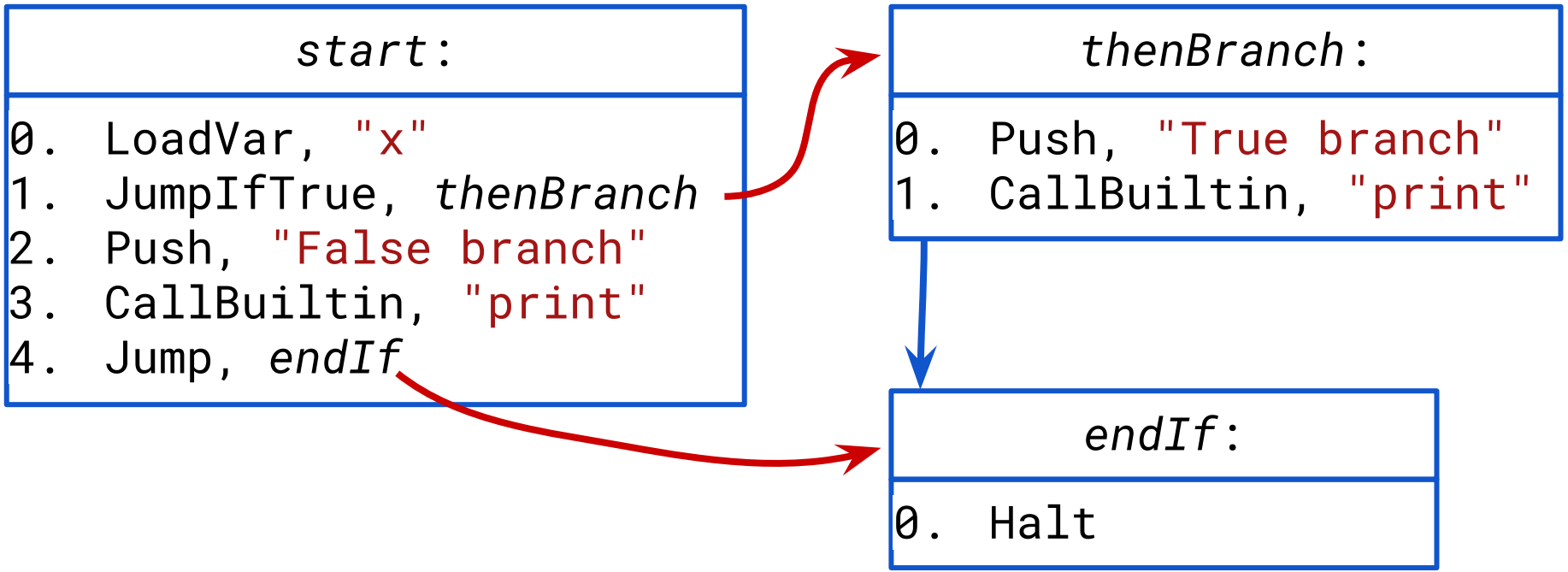
```
0.  LoadVar, "x"
1.  JumpIfTrue, 5
2.  Push, "False branch"
3.  CallBuiltin, "print"
4.  Jump, 7
5.  Push, "True branch"
6.  CallBuiltin, "print"
7.  Halt
```

Реализация переходов через метки (labels)

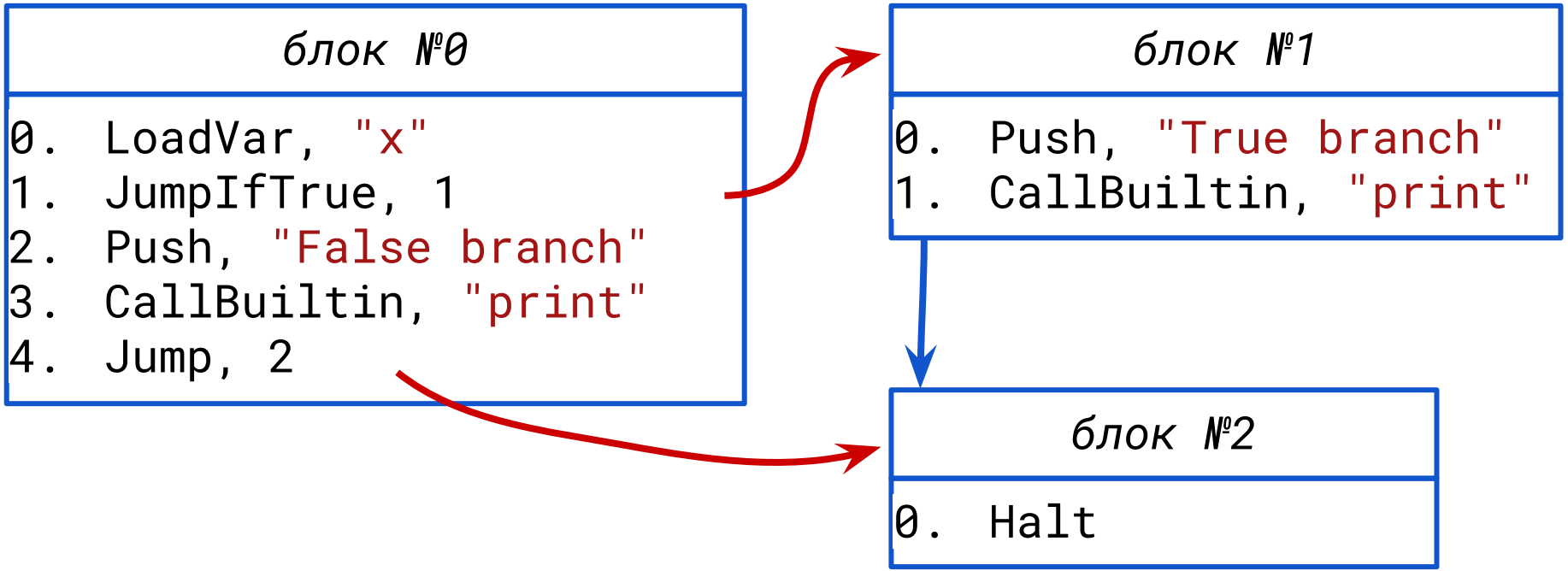
```
if x then
  print("True branch")
else
  print("False branch")
```

```
   0.  LoadVar, "x"
   1.  JumpIfTrue, thenBranch
   2.  Push, "False branch"
   3.  CallBuiltin, "print"
   4.  Jump, endIf

thenBranch:
   5.  Push, "True branch"
   6.  CallBuiltin, "print"

endIf:
   7.  Halt
```

Абстракция «базовый блок» — именованные блоки



Абстракция «базовый блок» — нумерованные блоки



Класс BasicBlock

```
public class BasicBlock(int id) {
    public int Id { get; } = id;
    public List<Instruction> Instructions { get; } = [];

    public void Append(Instruction instruction) {
        Instructions.Add(instruction);
    }
}
```

Класс InstructionsBuilder

```
public class InstructionsBuilder
{
    BasicBlock InsertPoint { get; set; }

    BasicBlock CreateBasicBlock();
    void Append(Instruction instruction);
    void AppendJump(OpCode code, BasicBlock target);
    List<Instruction> Finish();
}
```

Применение InstructionsBuilder

```
// Конструкция if ... then выполняется так:
// 1) Вычисляется условие
// 2) Если результат равен нулю, то перепрыгиваем через ветку then
BasicBlock finalBlock = _builder.CreateBasicBlock();
e.Condition.Accept(this);
_builder.AppendJump(InstructionCode.JumpIfFalse, finalBlock);
e.ThenBranch.Accept(this);
_builder.AppendJump(InstructionCode.Jump, finalBlock);
_builder.InsertPoint = finalBlock;
```

Зелёная дорожка

Этапы проекта «интерпретатор со своей VM»

1. Базовый уровень (3+)
    - Ввод-вывод
    - Выражения с операциями над числами и строками
    - Переменные
2. Контроль потока управления (4+)
    -

Пример для проекта «интерпретатор со своей VM»

Основной пример: https://sourcecraft.dev/sshambir-public/pstiger

Ветки:

08_arrays	Массивы и объявления типов
09_records	Структуры и nil
10_benchmarks	Бенчмарк программ на Tiger
11_virtual_machine	Виртуальная машина для языка Tiger
12_code_coverage	Сбор отчёта о покрытии кода тестами

Конец