Федеральное агентство связи
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»

Факультет ИВТ
Кафедра ВС

Теория языков программирования и методы трансляции

Молдованова Ольга Владимировна

График учебного процесса

Номер недели	Лабораторные работы	Курсовая работа
1	Лабораторная работа №1. Формальные языки, грамматики и их свойства	
2		
3		Выдача задания
4		Выполнение курсовой работы
5	Лабораторная работа №2. Лексический анализатор	
6		
7		
8		
9	Лабораторная работа №3. Таблицы идентификаторов	
10		
11		
12		
13	Лабораторная работа №4. Синтаксический анализатор	
14		
15		
16		Защита курсовой работы
17	Сдача долгов	

Содержание курса

- ✓ Языки и грамматики
- ✓ Трансляторы и компиляторы, методы и средства их разработки

Список литературы

Основная литература

1. Молчанов А.Ю. Системное программное обеспечение: Учебник для вузов. 3-е изд. – СПб.: Питер, 2010. – 400 с.

2. Ахо А.В., Лам М.С., Сети Р., Ульман Дж.Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд. : Пер. с англ. – М.: Изд. дом Вильямс, 2008. – 1184 с.

3. Cooper K.D., Torczon L. Engineering a Compiler, 2nd ed. – Elsevier, Inc., 2012. – 825 p.

4. Levine J.R. Flex & bison. – O'Reilly Media, Inc., 2009. – 274 p.

5. Niemann T. A Compact Guide To Lex & Yacc. Режим доступа: http://epaperpress.com/lexandyacc/index.html

6. Столяров А.В. Программирование на языке ассемблера NASM для ОС UNIX: Уч. пособие. – 2-е изд. – М.: МАКС Пресс, 2011. – 188 с.

7. Молдованова О.В. Языки программирования и методы трансляции: Учебное пособие. – Новосибирск: СибГУТИ, 2012. – 134 с.

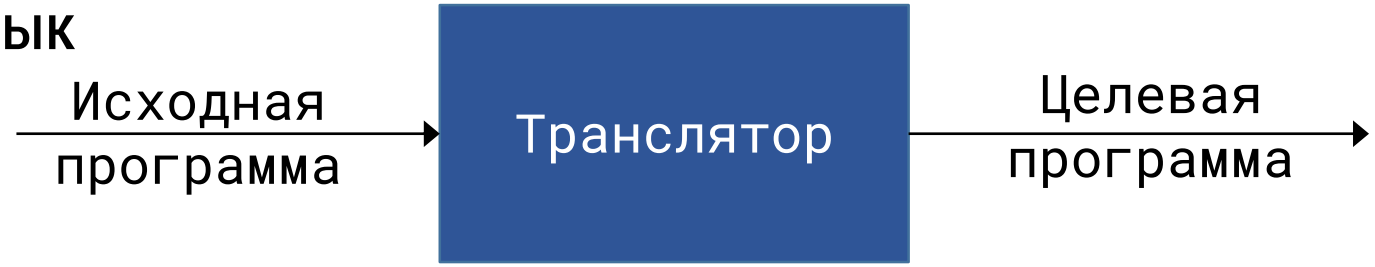
Список литературы

Дополнительная литература

1. Опалева Э., Самойленко В. Языки программирования и методы трансляции. — СПб. : БХВ-Петербург, 2010 г. — 480 с.
2. Волкова И.А., Руденко Т.В. Формальные грамматики и языки. Элементы теории трансляции: Учебное пособие. – М.: Издательский отдел факультета вычислительной математики и кибернетики МГУ им. М.В. Ломоносова, 1999. – 62 с.
3. Вирт Н. Построение компиляторов. – М.: ДМК-Пресс, 2010. – 192 с.
4. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т. 1: Синтаксический анализ: Пер. с англ. – М. : Мир, 1978. – 613 с.
5. Хопкрофт Дж. Э., Мотвани Р., Ульман Дж. Д. Введение в теорию автоматов, языков и вычислений, 2-е изд. : Пер. с англ. – М. : Издательский дом «Вильямс», 2002. – 528 с.
6. Aaby A.A. Compiler Construction using Flex and Bison. – Walla Walla College, 2005. – 96 p. – URL: www-lt.ls.fi.upm.es/compiladores/Software/compiler.pdf

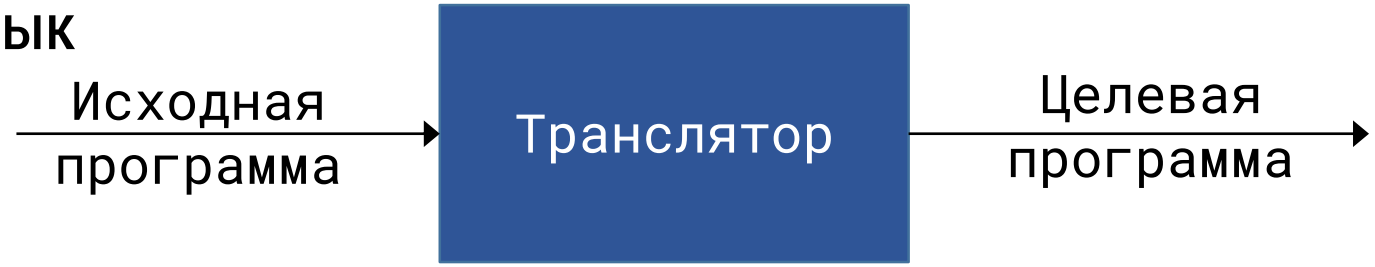
Транслятор, компилятор, интерпретатор

- Транслятор – это программа, которая транслирует программный код, написанный на одном языке, на другой язык



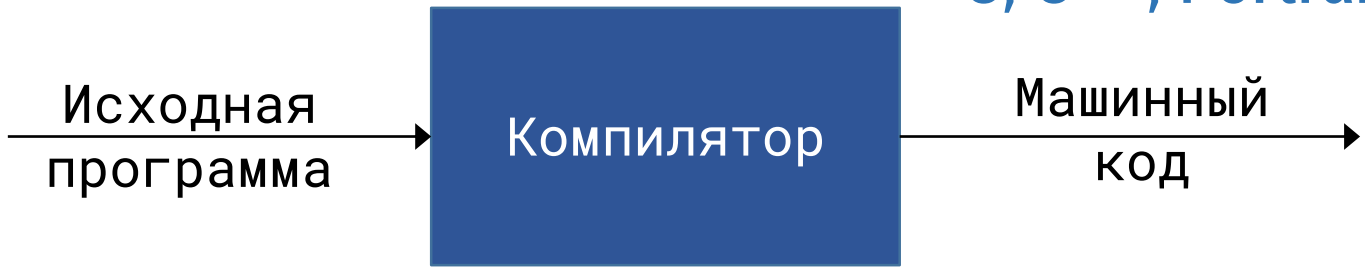
Транслятор, компилятор, интерпретатор

- Транслятор – это программа, которая транслирует программный код, написанный на одном языке, на другой язык



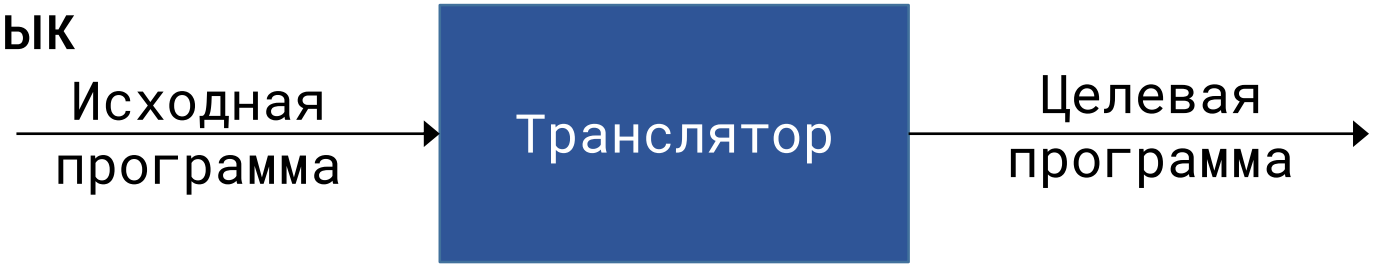
- Компилятор – это программа, которая транслирует программный код, написанный на языке программирования в машинный код или ассемблер

C, C++, Fortran, …



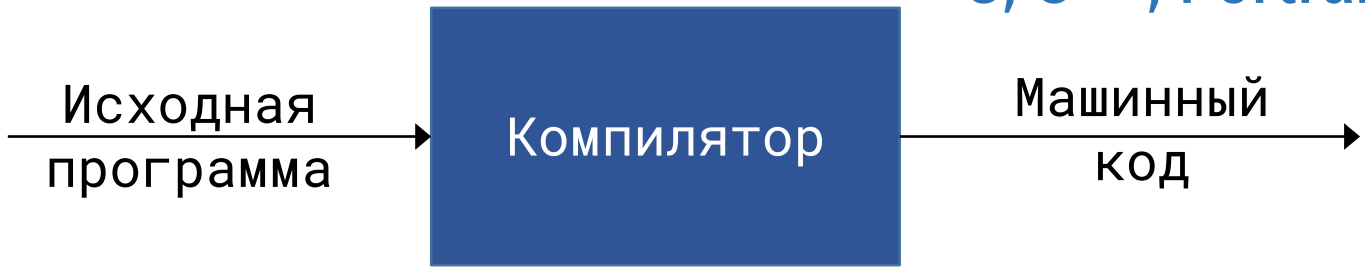
Транслятор, компилятор, интерпретатор

- Транслятор – это программа, которая транслирует программный код, написанный на одном языке, на другой язык



- Компилятор – это программа, которая транслирует программный код, написанный на языке программирования в машинный код или ассемблер

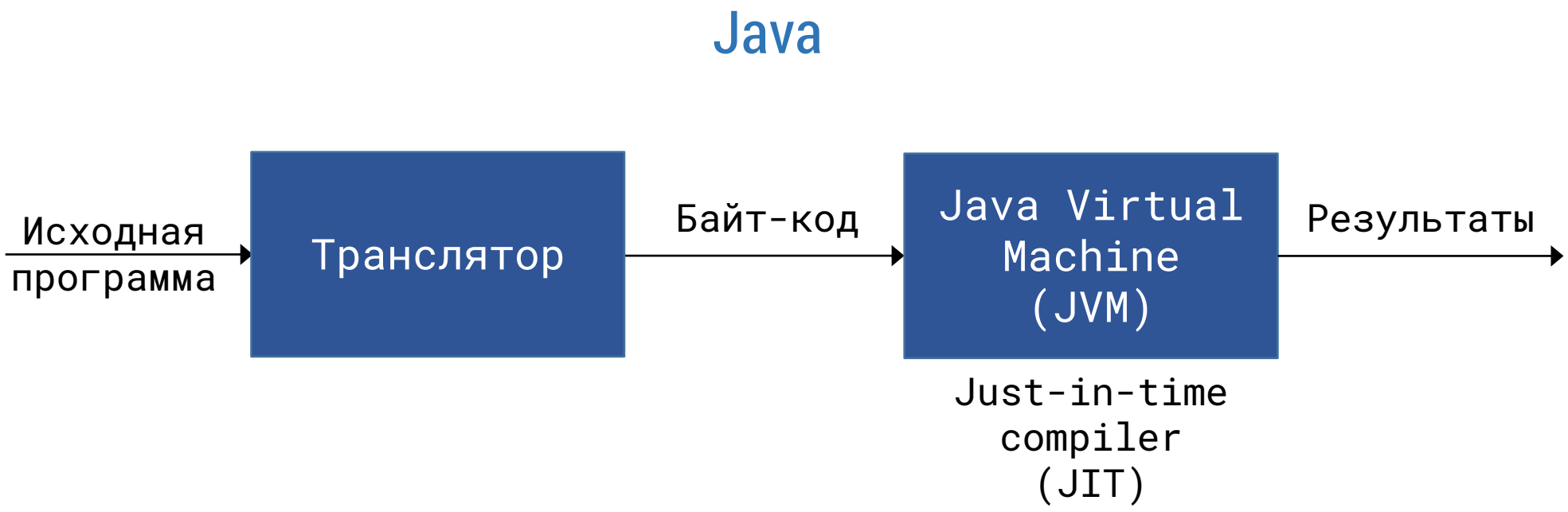
C, C++, Fortran, …



- Интерпретатор – это программа, которая воспринимает исходную программу на входном (исходном) языке и выполняет её

Perl, Schema, …

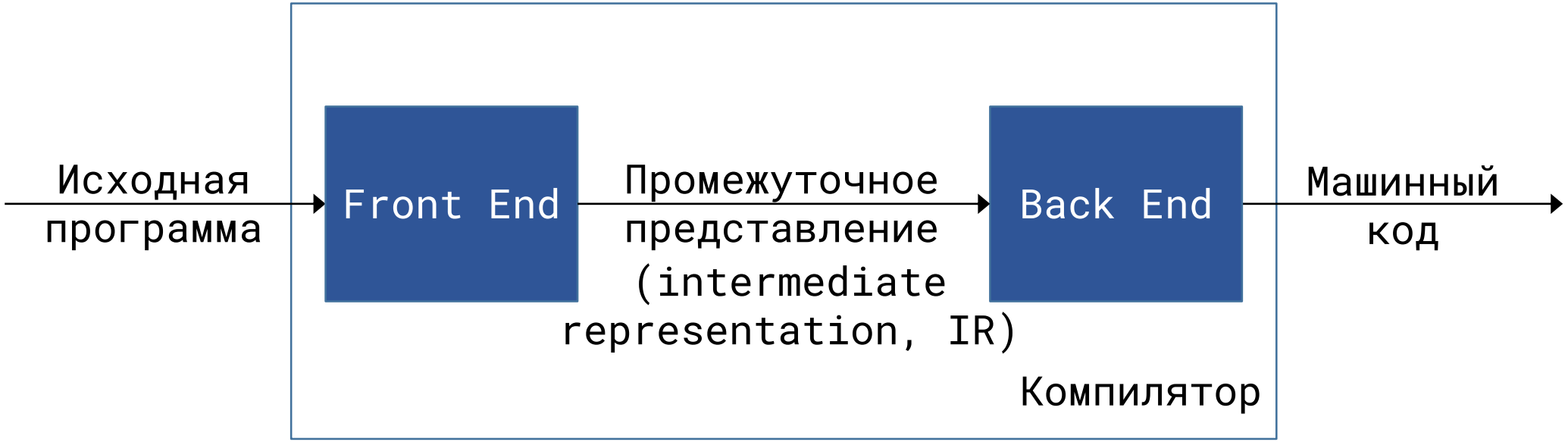
Гибридный компилятор



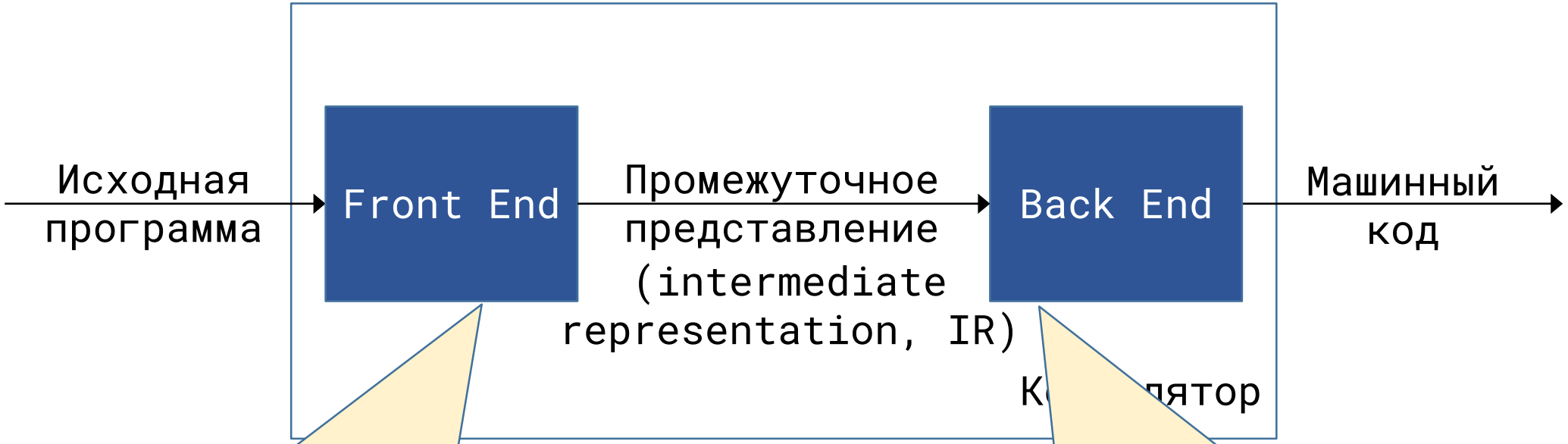
Компилятор – большая и сложная программа

- ✓ Жадные (greedy) алгоритмы (распределение регистров)
- ✓ Эвристические алгоритмы поиска
- ✓ Алгоритмы на графах (удаление неиспользуемого (dead) кода)
- ✓ Динамическое программирование (выбор инструкций)
- ✓ Конечные автоматы (лексический анализ)
- ✓ Автоматы с магазинной памятью (синтаксический анализ)

Двухфазная структура компилятора



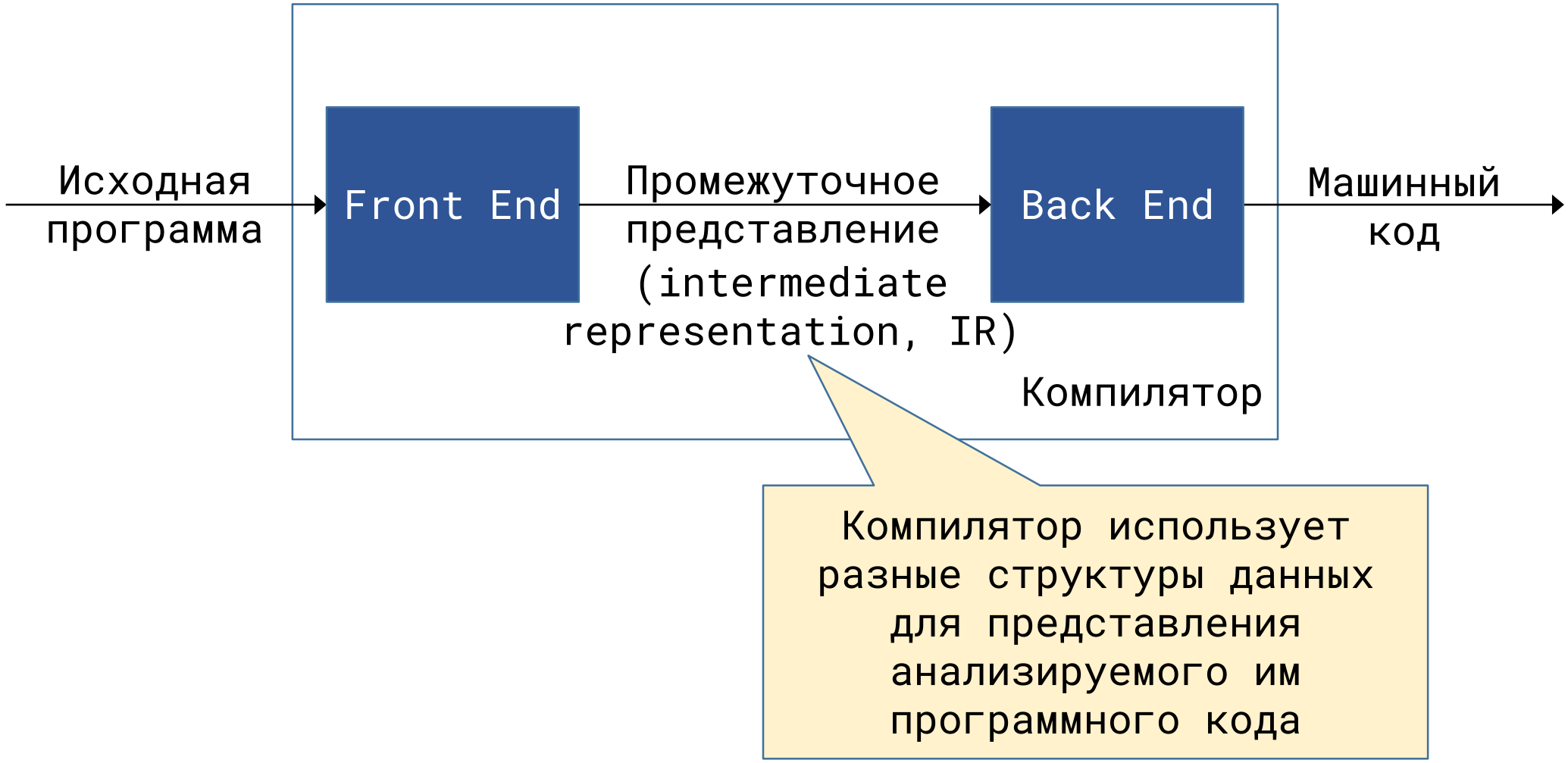
Двухфазная структура компилятора



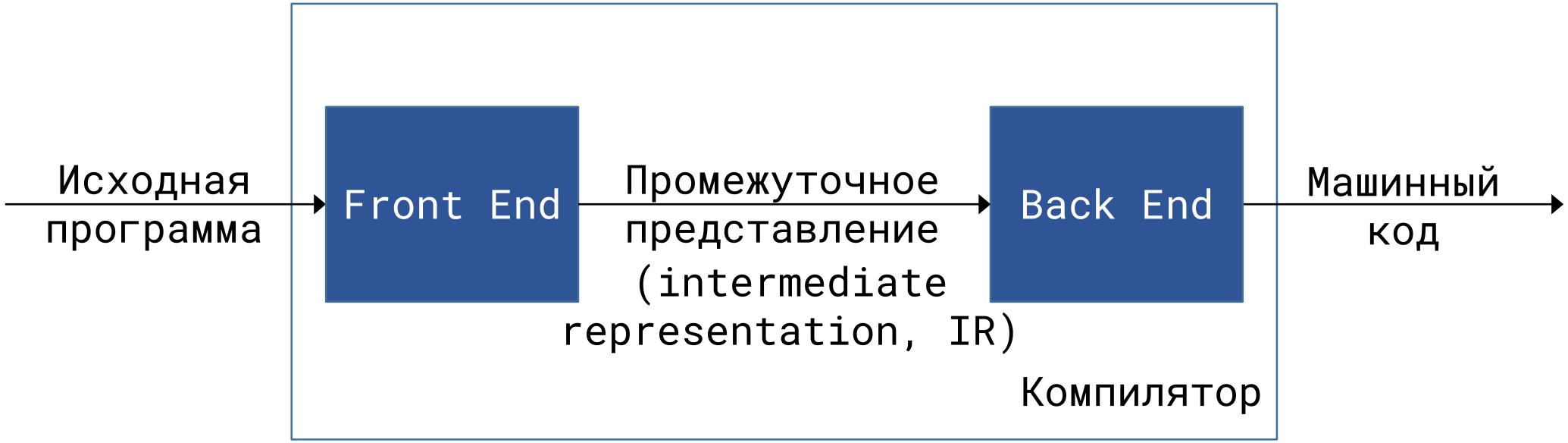
- препроцессор
- анализатор исходного кода
- начальная стадия
- распознавание

- постпроцессор
- генератор исполняемого кода
- заключительная стадия
- порождение

Двухфазная структура компилятора

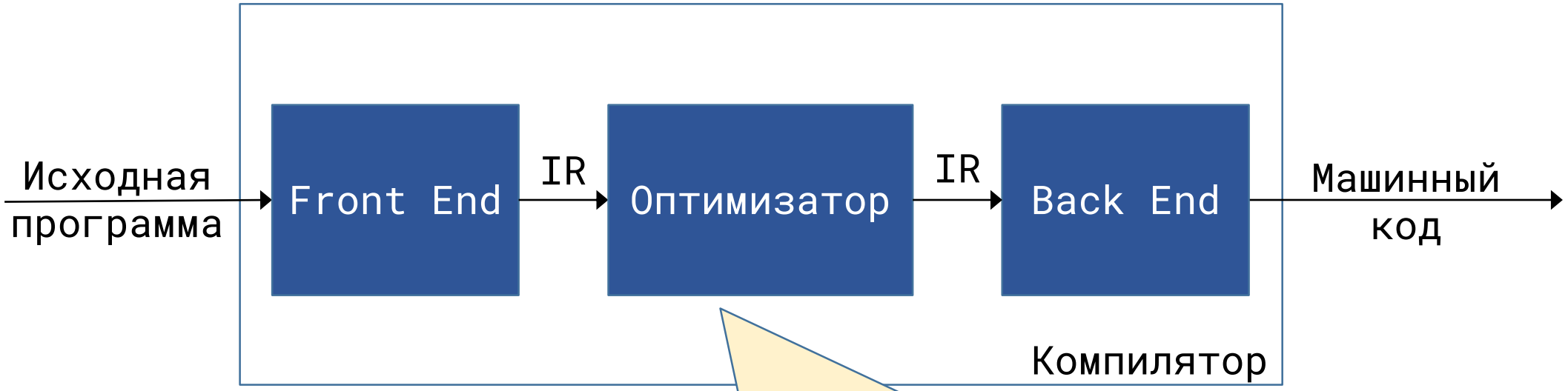


Двухфазная структура компилятора



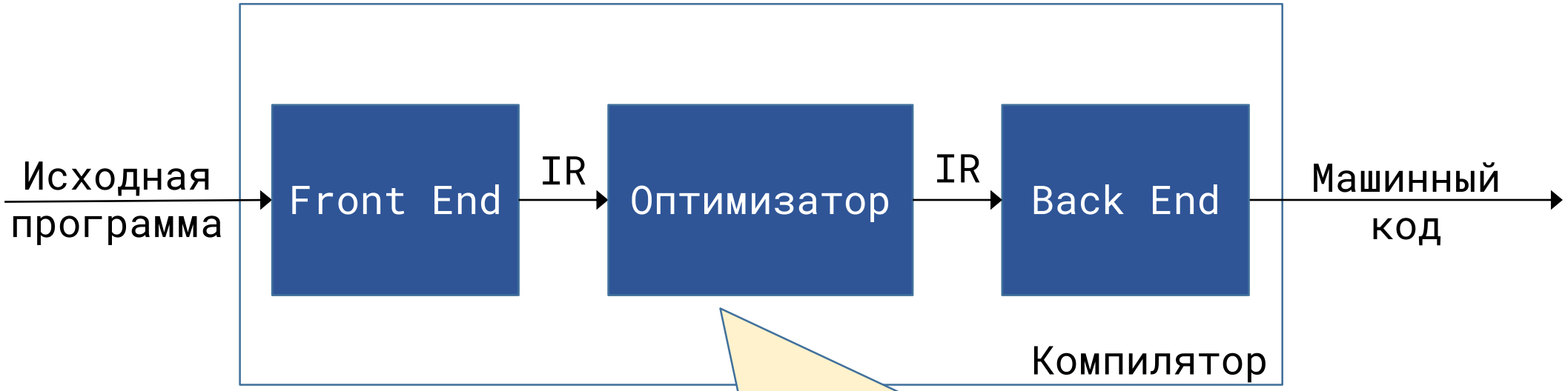
- упрощает процесс переориентации компилятора на другую целевую ВС
- предоставляет возможность реализации нескольких back-end’ов для одного front-end’а или нескольких front-end’ов для одного back-end’а
- предоставляет возможность добавления других фаз компиляции

Трехфазная структура компилятора



- Старается улучшить промежуточное представление: время выполнения и размер программы
- Выполняет один или несколько проходов, анализируя промежуточное представление и перезаписывая его

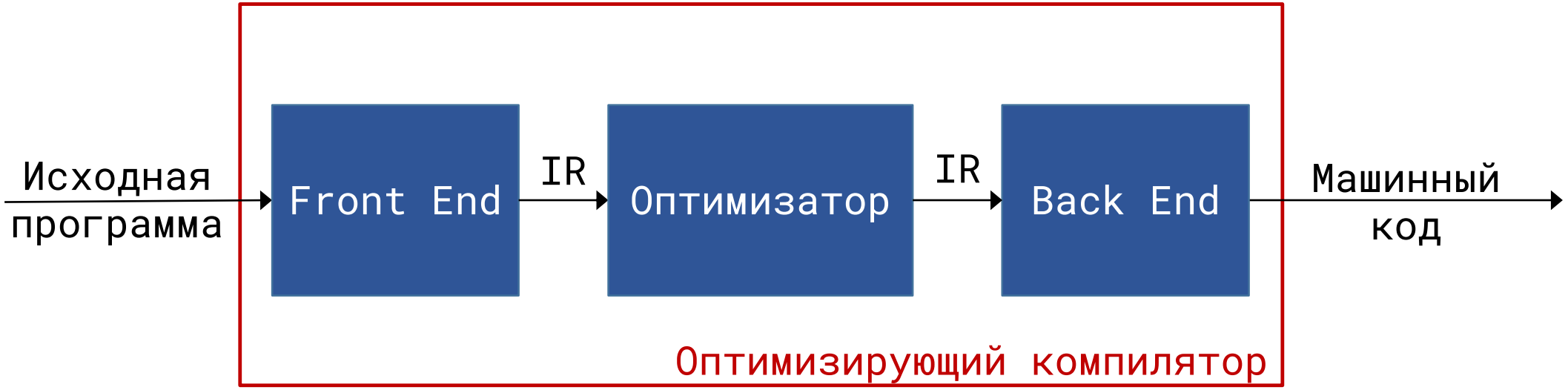
Трехфазная структура компилятора



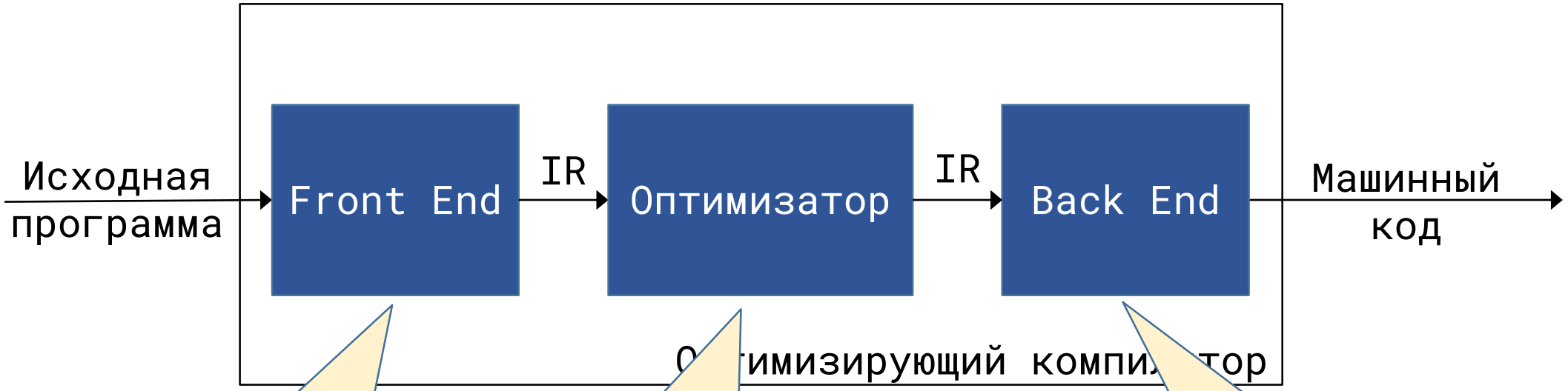
Проход (pass) – процесс последовательного чтения компилятором данных из памяти, их обработки и помещения результата работы в память

- Старается улучшить промежуточное представление: время выполнения и размер программы
- Выполняет один или несколько проходов, анализируя промежуточное представление и перезаписывая его

Трехфазная структура компилятора



Трехфазная структура компилятора

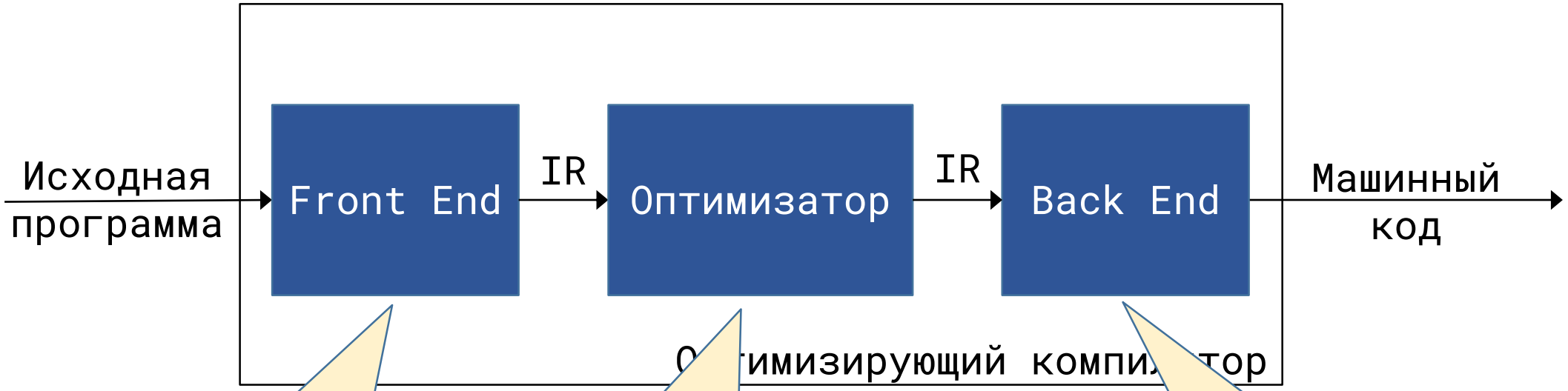


- Лексический анализ
- Синтаксический анализ
- Семантический анализ

- Оптимизация 1
- Оптимизация 2
- …
- Оптимизация *n*

- Выбор команд
- Планирование команд
- Выделение регистров

Трехфазная структура компилятора



- Лексический анализ
- Синтаксический анализ
- Семантический анализ

- Оптимизация 1
- Оптимизация 2
- …
- Оптимизация *n*

- Выбор команд
- Планирование команд
- Выделение регистров

Количество проходов в фазе оптимизации зависит от реализации конкретного компилятора

Краткий обзор процесса компиляции

```
a = a * 2 * b * c * d
```

Краткий обзор процесса компиляции

```
a = a * 2 * b * c * d
```

Front End

<u>Лексический анализ</u>

- читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся *лексемами*
- для каждой лексемы строит выходной *токен* (token) вида:

```
<имя_токена, значение_токена>
```

```
<id,a> <=> <id,a> <*> <int,2> <*> <id,b> <*> <id,c> <*> <id,d>
```

Краткий обзор процесса компиляции

```
a = a * 2 * b * c * d
```

Front End

<u>Лексический анализ</u>

- читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся *лексемами*
- для каждой лексемы строит выходной *токен* (token) вида:

```
<имя_токена, значение_токена>
```

```
<id,a> <=> <id,a> <*> <int,2> <*> <id,b> <*> <id,c> <*> <id,d>
```

1	a	…
2	2	…
3	b	…
…	…	…

Краткий обзор процесса компиляции

```
a = a * 2 * b * c * d
```

Front End

Лексический анализ

- читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся *лексемами*
- для каждой лексемы строит выходной *токен* (token) вида:

```
<имя_токена, значение_токена>
```

```
<id,a> <=> <id,a> <*> <int,2> <*> <id,b> <*> <id,c> <*> <id,d>
```

Обычно реализуется с помощью хэш-таблицы

1	a	…
2	2	…
3	b	…
…	…	…

Таблица идентификаторов (Symbol Table)

Краткий обзор процесса компиляции

```
<id,a> <=> <id,a> <*> <int,2> <*> <id,b> <*> <id,c> <*> <id,d>
```

Front End

<u>Синтаксический анализ</u>

- использует первые компоненты токенов, полученных при лексическом анализе, для создания *синтаксического дерева* (Abstract Syntax Tree, AST), которое описывает грамматическую структуру потока токенов

Краткий обзор процесса компиляции

```
<id,a> <=> <id,a> <*> <int,2> <*> <id,b> <*> <id,c> <*> <id,d>
```

Front End

Синтаксический анализ

- использует первые компоненты токенов, полученных при лексическом анализе, для создания *синтаксического дерева*, которое описывает грамматическую структуру потока токенов

```
<Arithmetic_Expr> → <Identifier>=<Identifier><Expr> | <Identifier><Expr>
<Expr> → *<Number> | *<Identifier> | *<Number><Expr> | *<Identifier><Expr>
<Identifier> → a | b | c | d
<Number> → 2
```

Грамматика языка в форме Бэкуса-Наура

Каждая строка представляет собой одно или несколько синтаксических правил

Краткий обзор процесса компиляции

```
<id,a> <=> <id,a> <*> <int,2> <*> <id,b> <*> <id,c> <*> <id,d>
```

Front End

Синтаксический анализ

- использует первые компоненты токенов, полученных при лексическом анализе, для создания *синтаксического дерева*, которое описывает грамматическую структуру потока токенов

```
<Arithmetic_Expr> → <Identifier>=<Identi
<Expr> → *<Number> | *<Identifier> | *<Nu
<Identifier> → a | b | c | d
<Number> → 2
```

Вывод предложения с помощью правил грамматики

```
<Arithmetic_Expr>
<Identifier>=<Identifier><Expr>
a=a<Expr>
a=a*<Number><Expr>
a=a*2<Expr>
a=a*2*<Identifier><Expr>
a=a*2*b<Expr>
…
a=a*2*b*c*d
```

Краткий обзор процесса компиляции

`Abstract Syntax Tree + Symbol Table`

Front End

<u>Генерация промежуточного представления кода</u>

- разные компиляторы используют разные виды промежуточного представления в зависимости от исходного языка программирования, целевого языка и выполняемых преобразований (например, в фазе оптимизации)

```
t0 =  a × 2
t1 = t0 × b
t2 = t1 × c
t3 = t2 × d
 a = t3
```

Краткий обзор процесса компиляции

Промежуточное представление кода

Оптимизатор

- сокращение времени выполнения исполняемого кода
- сокращение размера исполняемого кода
- сокращение энергопотребления процессора при выполнении исполняемого кода
- …

```
b = ···
c = ···
a = 1
for i = 1 to n
      read d
      a = a × 2 × b × c × d
end
```

Краткий обзор процесса компиляции

Промежуточное представление кода

Оптимизатор

- сокращение времени выполнения исполняемого кода
- сокращение размера исполняемого кода
- сокращение энергопотребления процессора при выполнении исполняемого кода
- …

```
b = ···
c = ···
a = 1
for i = 1 to n
      read d
      a = a × 2 × b × c × d
end
```

$4n$ операций умножения

Краткий обзор процесса компиляции

Промежуточное представление кода

Оптимизатор

- сокращение времени выполнения исполняемого кода
- сокращение размера исполняемого кода
- сокращение энергопотребления процессора при выполнении исполняемого кода
- …

```
b = ···
c = ···
a = 1
for i = 1 to n
      read d
      a = a × 2 × b × c × d
end
```

$4n$ операций умножения

```
b = ···
c = ···
a = 1
t = 2 × b × c
for i = 1 to n
      read d
      a = a × d × t
end
```

Краткий обзор процесса компиляции

Промежуточное представление кода

Оптимизатор

- сокращение времени выполнения исполняемого кода
- сокращение размера исполняемого кода
- сокращение энергопотребления процессора при выполнении исполняемого кода
- …

```
b = ···
c = ···
a = 1
for i = 1 to n
      read d
      a = a * 2 * b * c * d
end
```

$4n$ операций умножения

```
b = ···
c = ···
a = 1
t = 2 * b * c
for i = 1 to n
      read d
      a = a * d * t
end
```

$2n + 2$ операций умножения

Краткий обзор процесса компиляции

Промежуточное представление кода

Оптимизатор

- сокращение времени выполнения исполняемого кода
- сокращение размера исполняемого кода
- сокращение энергопотребления процессора при выполнении исполняемого кода
- …

- Анализ потока данных
- Анализ зависимостей по данным
- …

```
b = ···
c = ···
a = 1
for i = 1 to n
      read d
      a = a * 2 * b * c * d
end
```

$4n$ операций умножения

```
b = ···
c = ···
a = 1
t = 2 * b * c
for i = 1 to n
      read d
      a = a * d * t
end
```

$2n + 2$ операций умножения

Краткий обзор процесса компиляции

Промежуточное представление кода

Back End

- выбор команд
- планирование команд
- выделение регистров
- …

```
t0 =  a × 2
t1 = t0 × b
t2 = t1 × c
t3 = t2 × d
 a = t3
```

```
movq  %rsp, %rbp
subq  $16, %rsp
movl  -4(%rbp), %eax
addl  %eax, %eax
imull -8(%rbp), %eax
imull -12(%rbp), %eax
movl  -16(%rbp), %edx
imull %edx, %eax
movl  %eax, -4(%rbp)
movl  -4(%rbp), %eax
```

Краткий обзор процесса компиляции

Промежуточное представление кода

Back End

- выбор команд
-
-
-

Ассемблерный код, сгенерированный компилятором GCC 7.1.1 для программы:

```
int main()
{
    int a, b, c, d;
    a = a * 2 * b * c * d;
    printf("\n%d", a);
    return 0;
}
```

```
movq  %rsp, %rbp
subq  $16, %rsp
movl  -4(%rbp), %eax
addl  %eax, %eax
imull -8(%rbp), %eax
imull -12(%rbp), %eax
movl  -16(%rbp), %edx
imull %edx, %eax
movl  %eax, -4(%rbp)
movl  -4(%rbp), %eax
```

Инструментарий для создания компиляторов

1. Генераторы синтаксических анализаторов (YACC, Bison)
2. Генераторы лексических анализаторов (LEX, FLEX)
3. Средства синтаксически управляемой трансляции
4. Генераторы генераторов кода
5. Средства работы с потоком данных (Datalog)
6. Наборы для построения компиляторов