Курс «Компиляторные технологии»

Лекция 1

Введение в компиляцию

Курносов Михаил Георгиевич

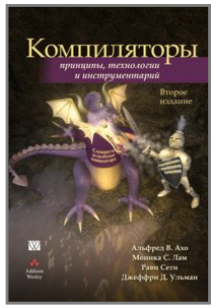
www.mkurnosov.net

Сибирский государственный университет телекоммуникаций и информатики
Осенний семестр

Литература по курсу

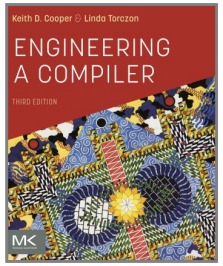
Основной источник (весна)

- [DragonBook] Ахо А., Сетхи Р., Ульман Дж. **Компиляторы: принципы, технологии и инструменты** (2-е изд., 2008)



Дополнительная литература

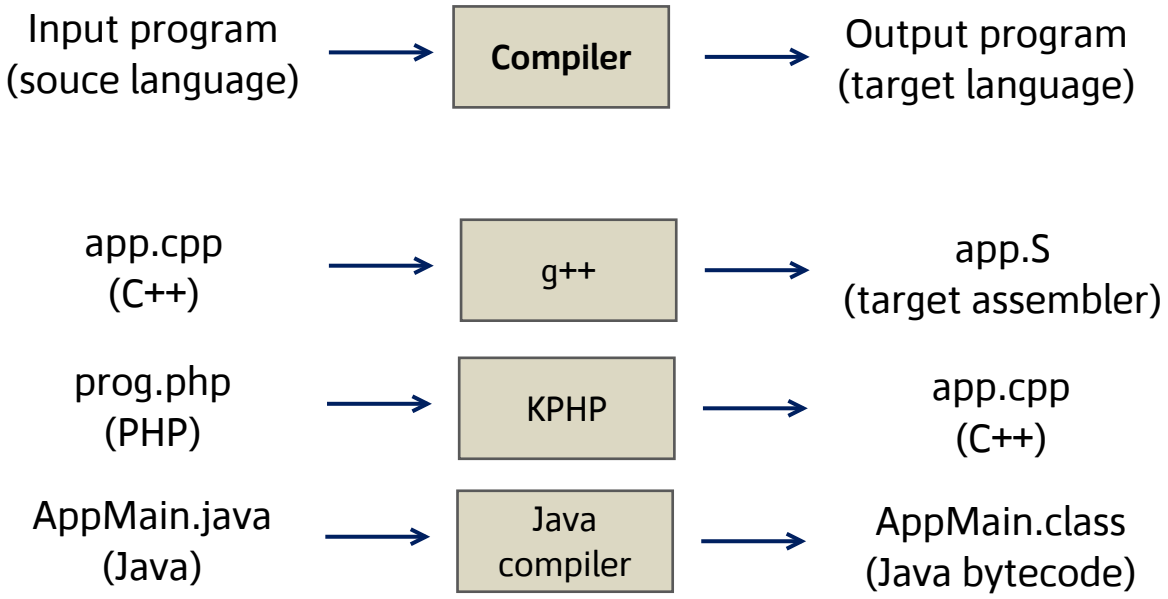
- Keith D. Cooper, Linda Torczon. **Engineering a Compiler** (3ed edition, 2022)
- Хантер Р. **Основные концепции компиляторов**. — 2002
- Вирт Н. **Построение компиляторов**. — 2010
- Appel A. **Modern Compiler Implementation in {C, Java}**. — 1998
- https://llvm.org/docs/
- https://gcc.gnu.org/onlinedocs/
- Stanford CS143 Compilers



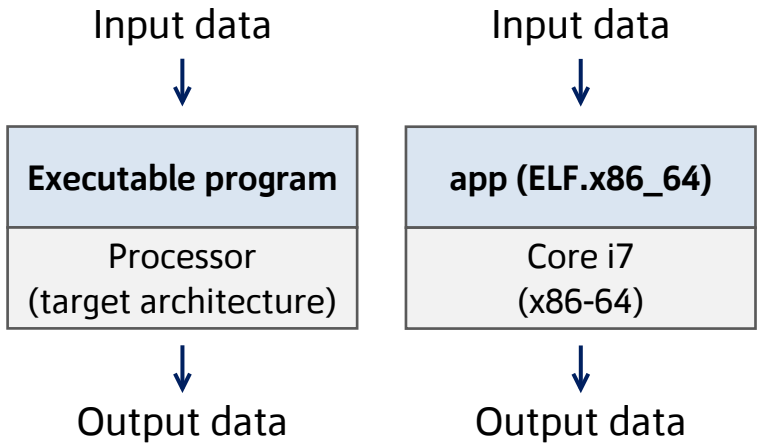
Компилятор (compiler)

- **Компилятор** (compiler) — программа, транслирующая код на *исходном языке* (source launguage) в текст программы на *целевом языке* (target language) с сохранением семантики
- **Исполняемая программа** (program) — файл на носителе информации в исполняемом формате (ELF, PE, Mach-O) с секциями кода на целевой архитектуре (target architecture)

1) Компиляция программы



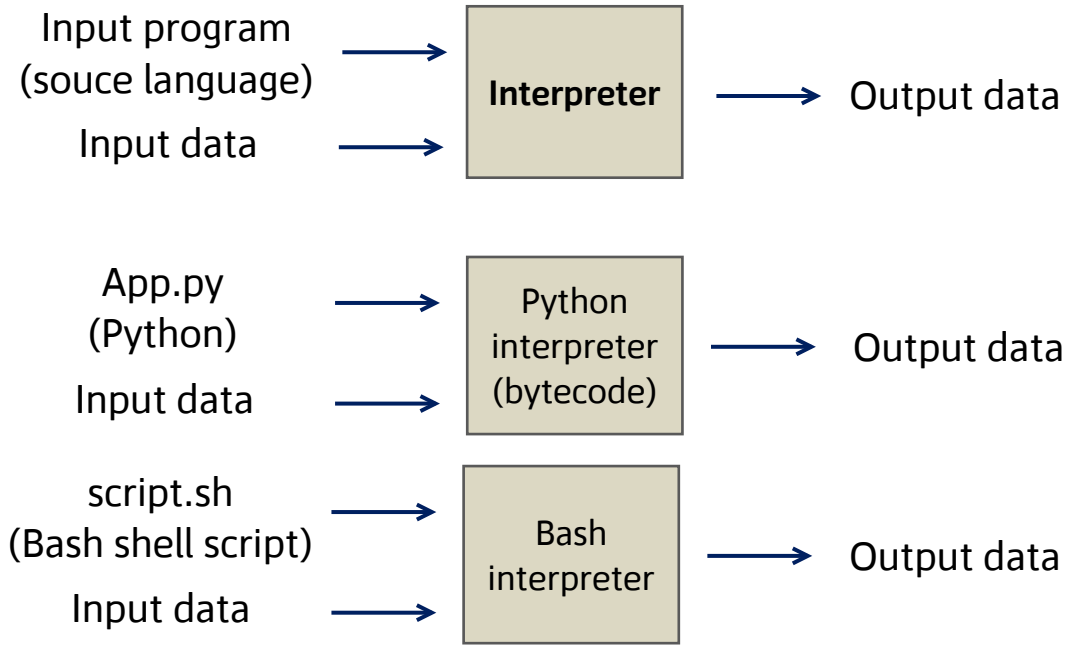
2) Выполнение программы



Интерпретатор (interpreter)

- **Интерпретатор** (interpreter) — программа, выполняющая программу на *исходном языке* (source launguage) для заданных входных данных
- Отличия от компилятора: не генерирует программу на целевом языке, сразу получает входные данные для выполнения программы

Интерпретация



Статические и динамические компиляторы

- **AOT-компилятор** (ahead-of-time) — выполняет полную трансляцию программы до её выполнения (статическая компиляция)
- **JIT-компилятор** (just-in-time) — выполняет динамическую трансляцию промежуточного кода (байт-кода) в исполняемый код целевой архитектуры
  - Трудоемкие операции выполнены статическим транслятором: синтаксический анализ, оптимизация кода
  - Плюсы: позволяет задействовать аппаратные возможности (векторизация, счетчики производительности, оптимизации доступа к кеш-памяти), встраивание кода (inlining), оптимизация для заданных входных данных, байт код обеспечивает переносимость между платформами
  - Минусы: затраты на динамическую компиляцию (warm-up time, startup delay)
- **Гибридный компилятор** — выполняет статическую трансляцию в промежуточный код (bytecode, p-code) и динамическую JIT-компиляцию отдельных частей программы из промежуточного кода в код целевой архитектуры
  - Java Bytecode (Java VM) — стековая виртуальная машина
  - Google Dalvik — регистровая Java VM
  - Microsoft .NET — стековая VM (CIL, MSIL)
  - JavaScript V8 bytecode — регистровая VM
  - CPython bytecode — стековая VM

Гетерогенные компиляторы и кросс-компиляция

- **Гетерогенный компилятор** — выполняет трансляцию программы с частями на разных языках (диалектах) для разных целевых архитектур
- Примеры
  - C/C++/Fortran код + директивны #pragma omp parallel для выгрузки (offload) на ускоритель
  - C/C++ и NVIDIA CUDA
  - C/C++ и OpenCL

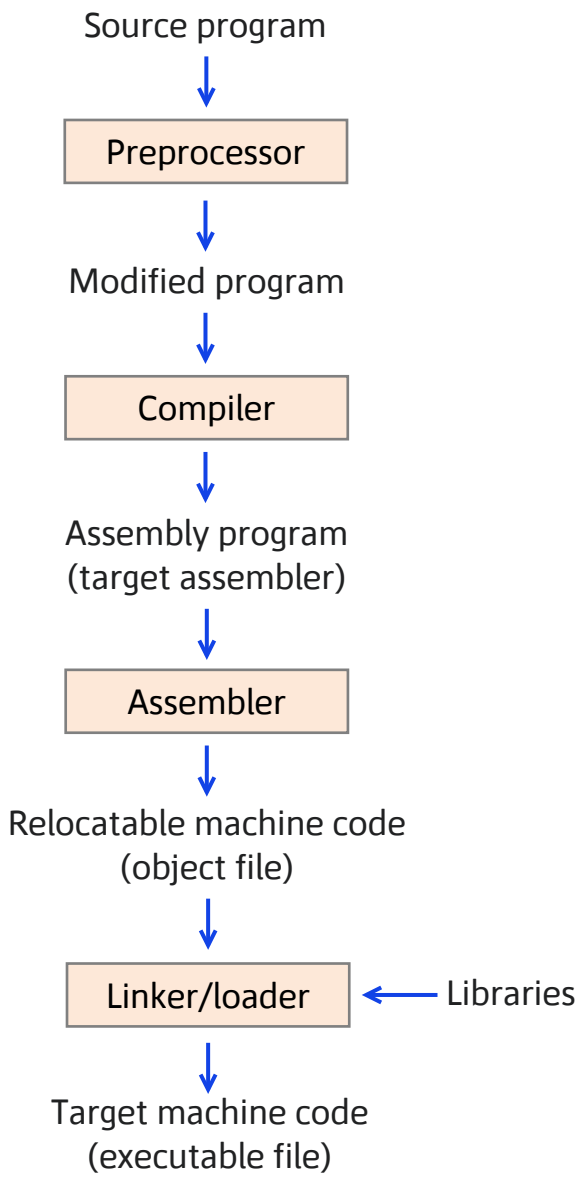
- **Кросс-компиляция** — трансляция программы под архитектуру процессора отличающуюся от архитектуры процессора, на котором выполняется компиляция
  - Сборка на x86-64 для ARMv8
  - Сборка на x86-64 (little-endian) для s390x (big-endian)
  - Сборка на x86-64 для RISC-V
- Вопрос — как при кросс-компиляции транслятор вычисляет constexpr блоки C++ (x86-64 —> s390x) ?

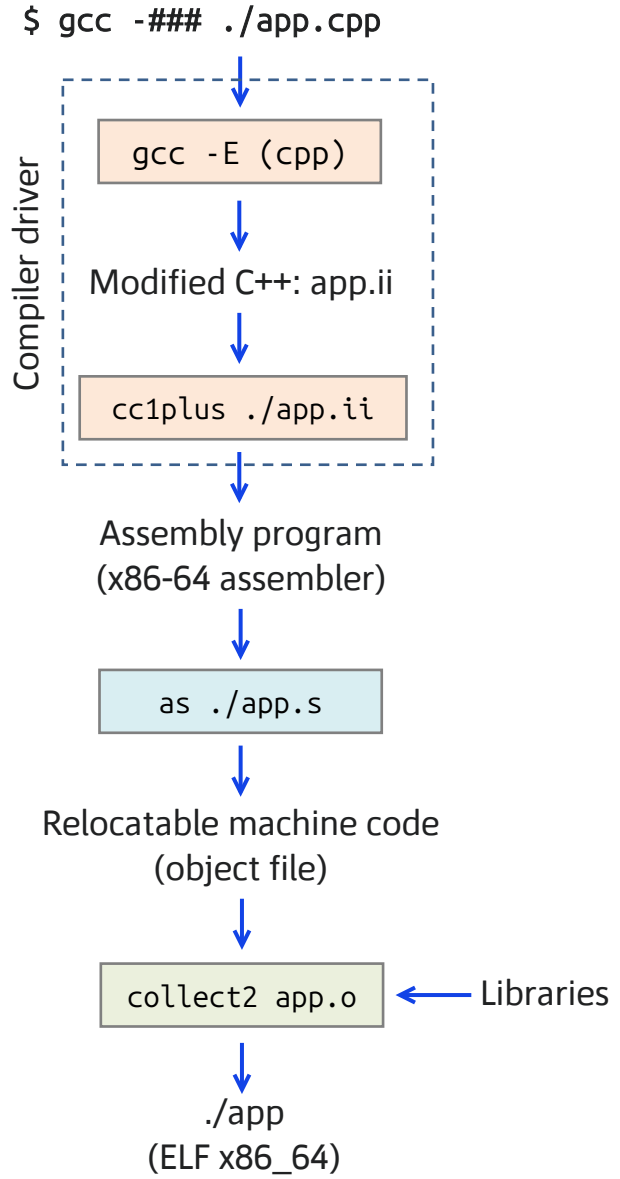
Двоичная трансляция

- **Двоичная трансляция** (binary translation) — трансляция машинного кода программы заданной архитектуры в машинный код целевой архитектуры
- **Динамическая бинарная трансляция** (dynamic binary translation) — трансляция машинного кода программы заданной архитектуры в машинный код целевой архитектуры в ходе ее выполнения
- Примеры
  - Apple: динамическая трансляция из M68K в PowerPC
  - Эльбрус: Lintel x86-64 —> Elbrus E2K
  - Intel: IA-32EL x86 —> Intel Itanium (VLIW)
  - Transmeta: x86 -> Transmeta Crusoe (VLIW)
  - QEMU

Статическая компиляция

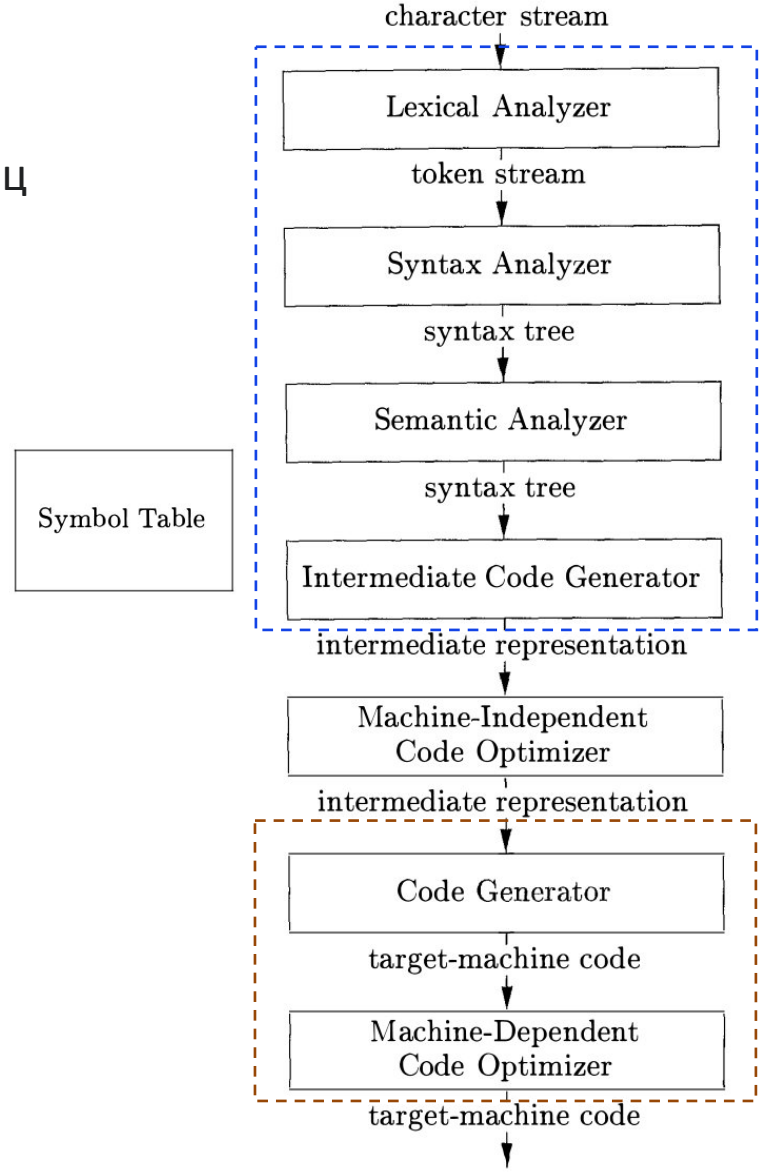
- **Препроцессор** (preprocessor) — обрабатывает директивы (#include, #define, #ifdef)
- **Компилятор** (compiler) — программа, транслирующая код на *исходном языке* (source launguage) в текст программы на *целевом языке* (target language) с сохранением семантики
- **Ассемблер** (assembler) — транслятор с ассемблера в машинный код
- **Компоновщик** (linker) — программа объединяющая объектные файлы в исполняемый файл
- **Исполняемая программа** (program) — файл на носителе информации в исполняемом формате (ELF, PE, Mach-O) с секциями кода для целевой архитектуры (target architecture)
- **Загрузчик** (loader) — загружает секции исполняемого файла в память, загружает требуемые библиотеки динамической компоновки, передает управление на точку старта





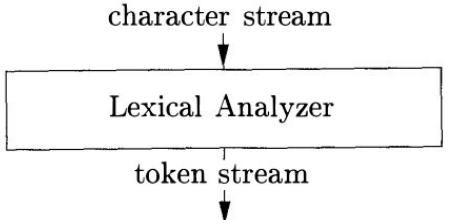
Структура компилятора

- **Фаза анализа** (frontend, начальная стадия) —
  разбивает программу на последовательность минимально значимых единиц языка, накладывает на них грамматическую структуру языка, обнаруживает синтаксические и семантические ошибки, формирует таблицу символов, генерирует промежуточное представление программы
- **Фаза синтеза** (backend, заключительная стадия) —
  транслирует программу на основе таблицы символов и промежуточного представления в код целевой архитектуры
- **Общий процесс компиляция включает фазы** (phases):
  - лексический анализ
  - синтаксический анализ
  - семантический анализ
  - генерация (синтез) промежуточного представления
  - машинно-независимая оптимизация промежуточного представления
  - генерация машинного кода
  - машинно-зависимые оптимизации кода



[*] Aho A., Lam M., Sethi R., Ullman J. **Compilers: principles, techniques, and tools**. — 2nd ed., 2007

Лексический анализ




```
<ID,0>, <ASSIGN>,
<ID,1>, <PLUS>,
<ID,2>, <MUL>,
<STR,3>,
<SEMICOLON>
```

- **Лексический анализатор** (lexical analyzer, lexer, scanner) — разбивает входную программу на последовательность *лексем* (lexeme), минимально значимых единиц входного языка
- Тип допустимых лексем определяется описанием языка
- Игнорирует пробельные символы, комментарии, отслеживает номер текущей строки для корректного информирования о положении возможных ошибок

```
// Увеличить сумму
globalSum = localSum + r * 16;
```

Лексемы: «globalSum», «=», «localSum», «+», «r», «*», «16», «;»

Symbol table	
ID	Symbol
0	globalSum
1	localSum
2	r
3	16

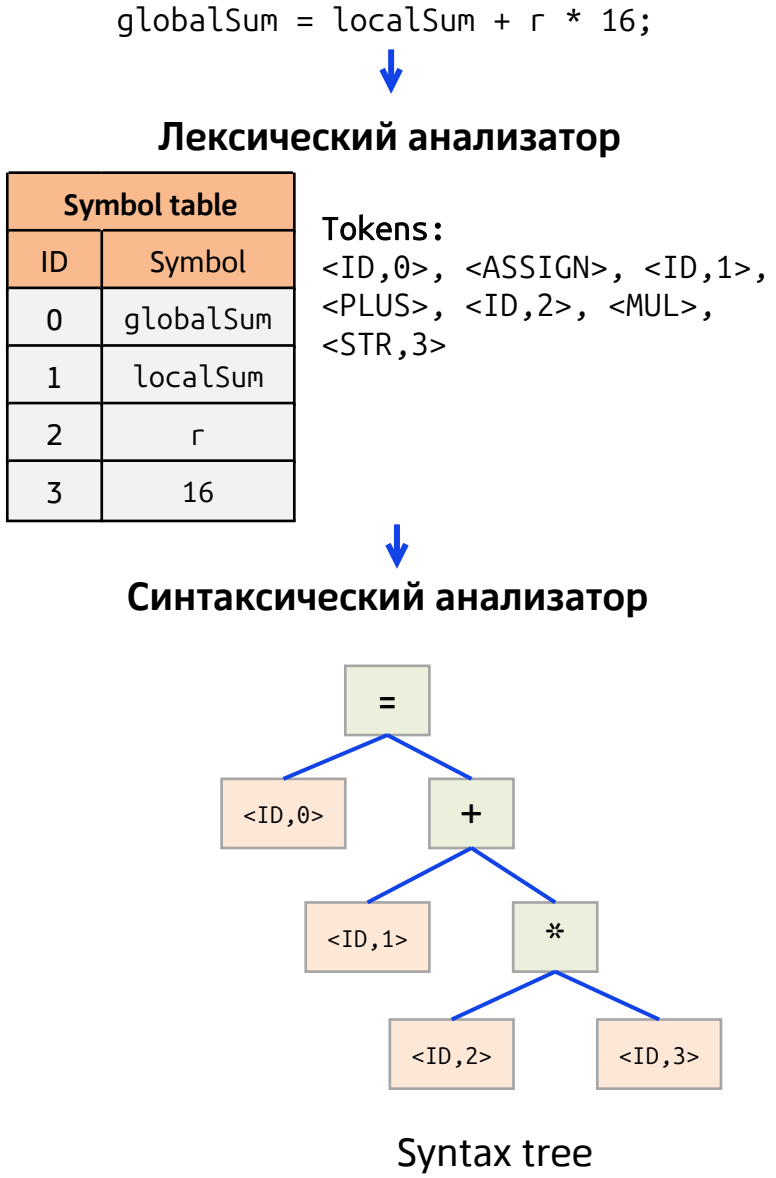
- Для каждой найденной лексемы анализатор формирует *токен* (token) — пара <*имя-токена*, *значение-атрибута*>, *имя-токена* — тип/класс лексемы, *значение-атрибута* — непосредственно лексема или ссылка на запись в таблице символом

Tokens: <ID,0>, <ASSIGN>, <ID,1>, <PLUS>, <ID,2>, <MUL>, <STR,3>, <SEMICOLON>

Token-names: ID, ASSIGN, PLUS, MUL, STR, SEMICOLON

Синтаксический анализ

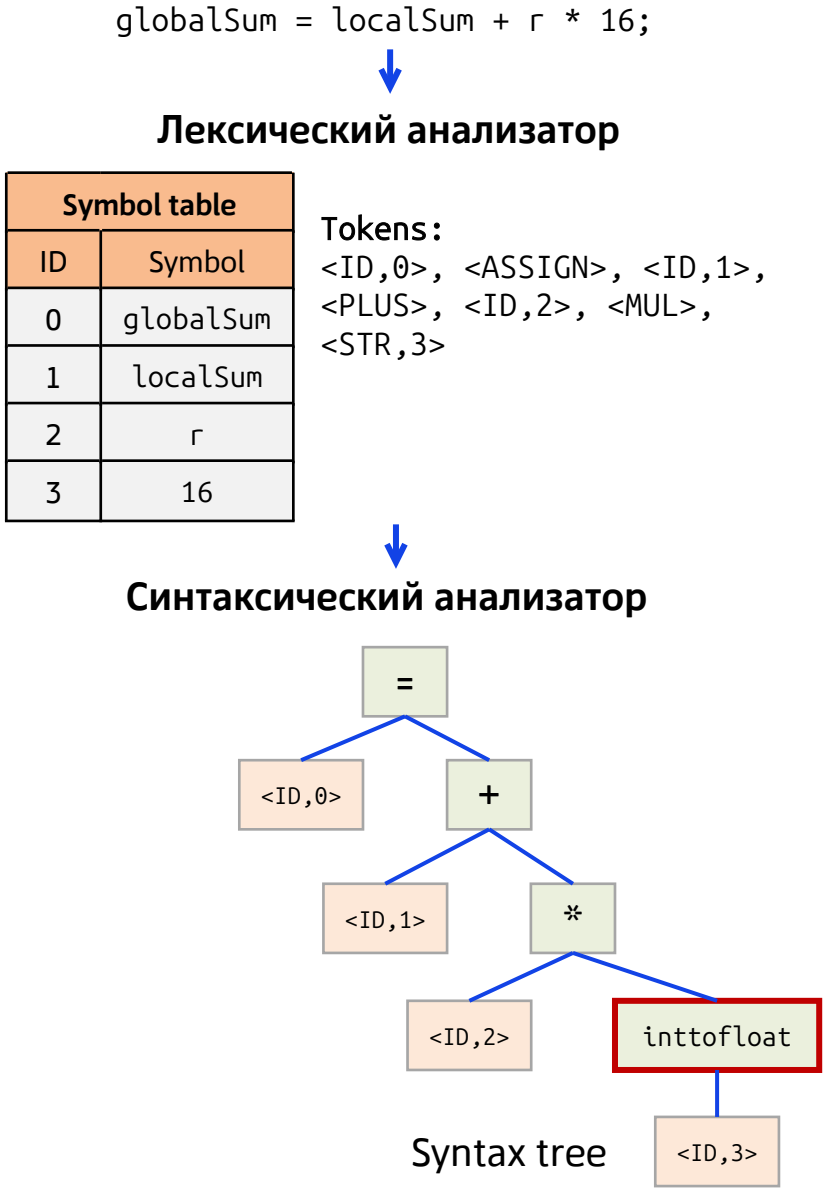
- **Синтаксический анализ** (разбор, parsing) — процесс проверки соответствия входного потока токенов (текста программы) синтаксису входного языка и построения синтаксического дерева
- Синтаксический анализатор строится на основе синтаксиса входного языка, который описывается *формальной грамматикой языка*
- **Синтаксическое дерево** (syntax tree) — каждый внутренний узел дерева представляет операцию языка, дочерние узлы — аргументы операции



Syntax tree

Семантический анализ

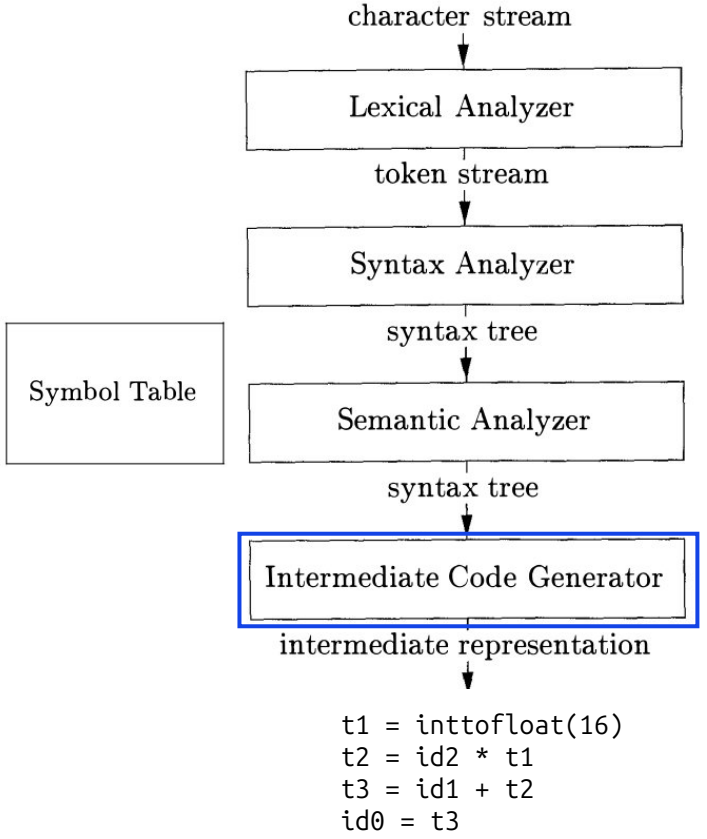
- **Семантический анализатор** проверяет исходную программу (синтаксическое дерево) на семантическую согласованность с определением языка
- Проверка типов данных (операнды, аргументы), отсутствие циклов в графе наследования классов (и др.), проверка существования имен объектов в области видимости
- Дополняет синтаксическое дерево и таблицу символов информацией о типах данных, добавляет преобразования типов
- Синтаксическое дерево — форма промежуточного представления фазы анализа

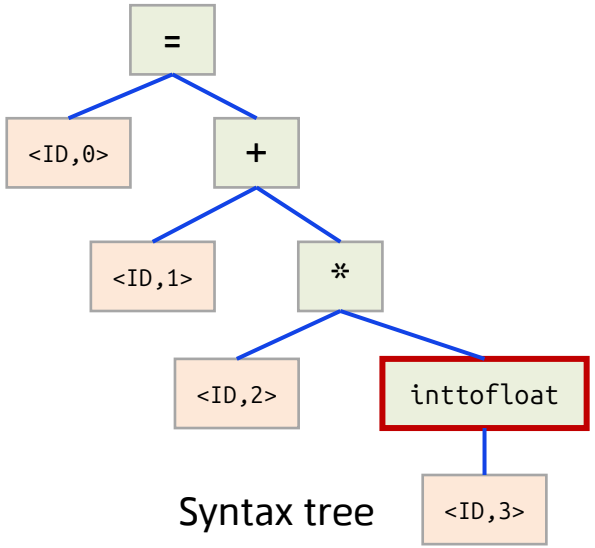


Генерация кода в промежуточное представление

- **Промежуточное представление** (intermediate representation, IR) — архитектура набора команд (ISA) абстрактной вычислительной машины, в который легко транслировать синтаксическое дерево, над которым легко выполнять оптимизации и трансформации и генерировать машинный код для целевой архитектуры

- Трехадресный код — в каждой команде 3 операнда
- Стековые и регистровые машины





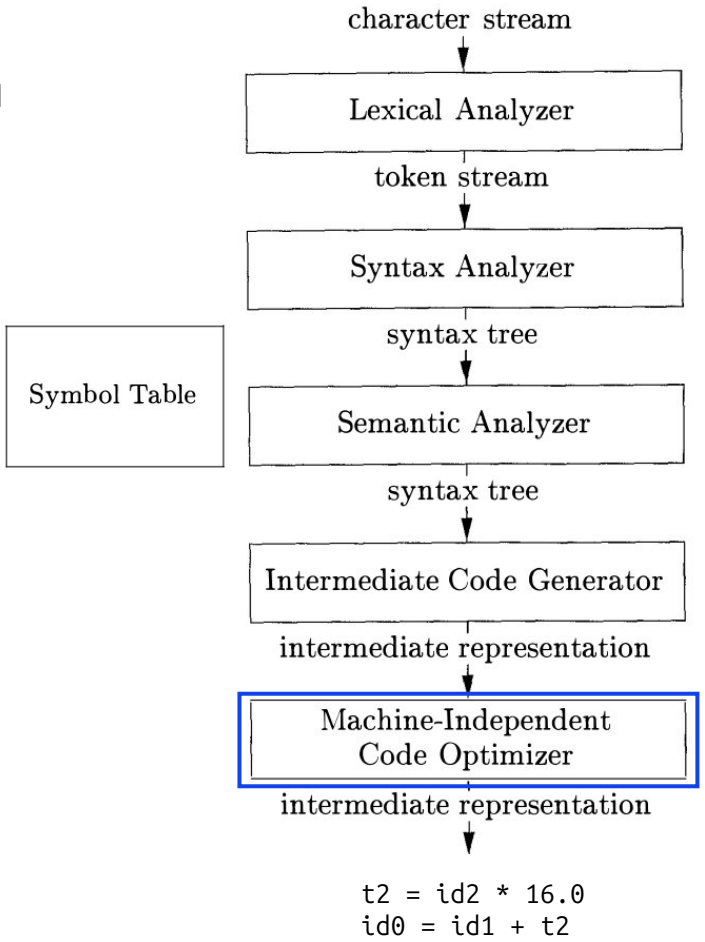
Syntax tree

Трехадресный код (IR)

```
t1 = inttofloat(16)
t2 = id2 * t1
t3 = id1 + t2
id0 = t3
```

Оптимизации на уровне промежуточного представления

- **Оптимизации на уровне промежуточного представления** — совокупность применяемых алгоритмов машинно-независимых оптимизаций (проходы, passes)
- Цели оптимизации: минимизация времени, минимизация размера кода, минимизация использования ресурсов
- Оптимизирующие преобразования — длительная фаза компиляции
- Область оптимизации: базовый блок, функция, файл (единица трансляции)



Трехадресный код (IR)

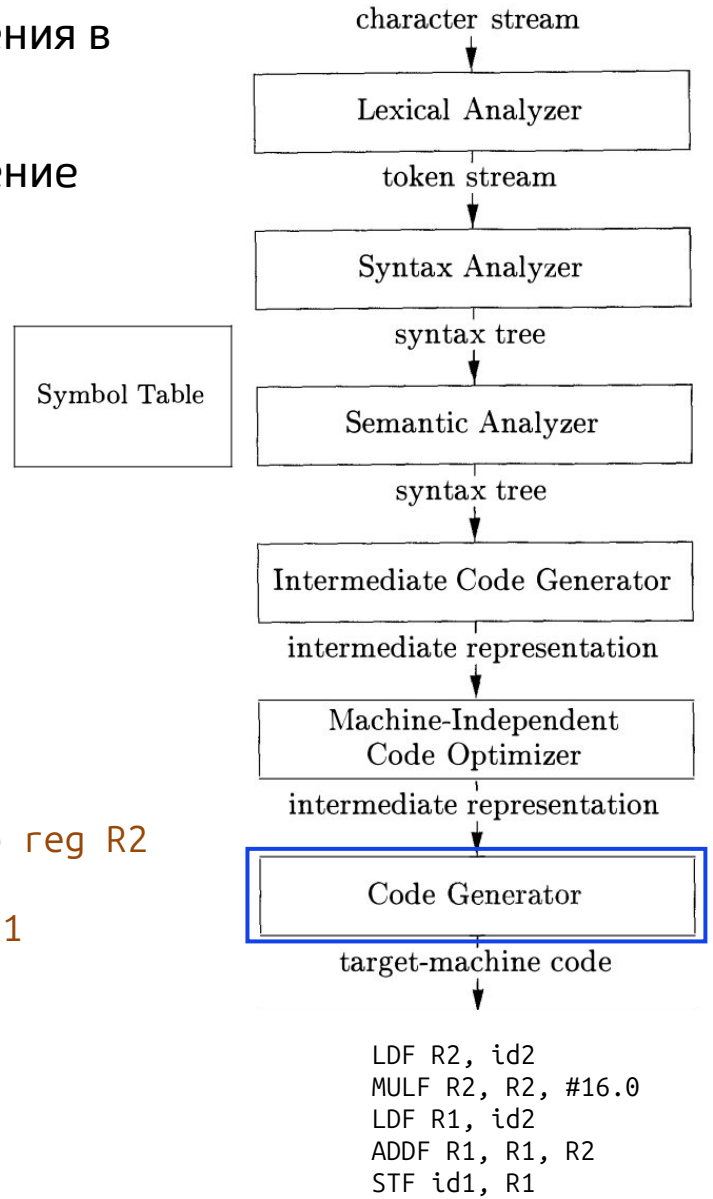
```
t1 = inttofloat(16)
t2 = id2 * t1
t3 = id1 + t2
id0 = t3
```

Трехадресный код (IR)

```
t2 = id2 * 16.0
id0 = id1 + t2
```

Генерация кода

- **Генерация кода** (code generation) — трансляция промежуточного представления в язык целевой системы (ассемблер)
- Решаются задачи распределения ресурсов целевой архитектуры: распределение регистров, управление стеком, соблюдение ABI
- Компилятор должен иметь описание целевой системы:
  описание набора команд, их временные характеристики, число регистров, соглашение ABI и др.



Трехадресный код (IR)

```
t2 = id2 * 16.0
id0 = id1 + t2
```

→

Код целевой системы (ассемблер)

```
LDF R2, id2          # Load float id2 to reg R2
MULF R2, R2, #16.0   # R2 = R2 * 16.0
LDF R1, id2          # Load id2 to reg R1
ADDF R1, R1, R2      # R1 = R1 + R2
STF id1, R1          # Store R1 to id1
```

Инструменты построения компиляторов

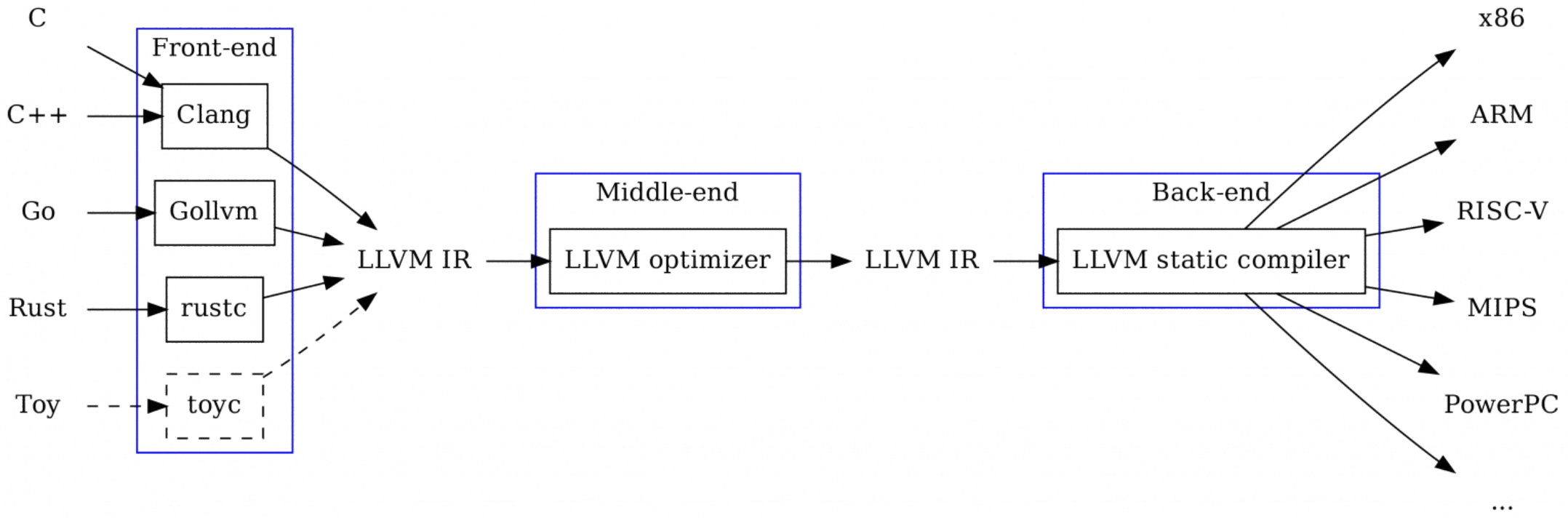
- **Генераторы лексических анализаторов**
  - Lex
  - Flex
  - Ragel

- **Генераторы синтаксических анализаторов**
  - Yacc
  - Bison
  - ANTLR
  - Coco/R

- https://en.wikipedia.org/wiki/Comparison_of_parser_generators

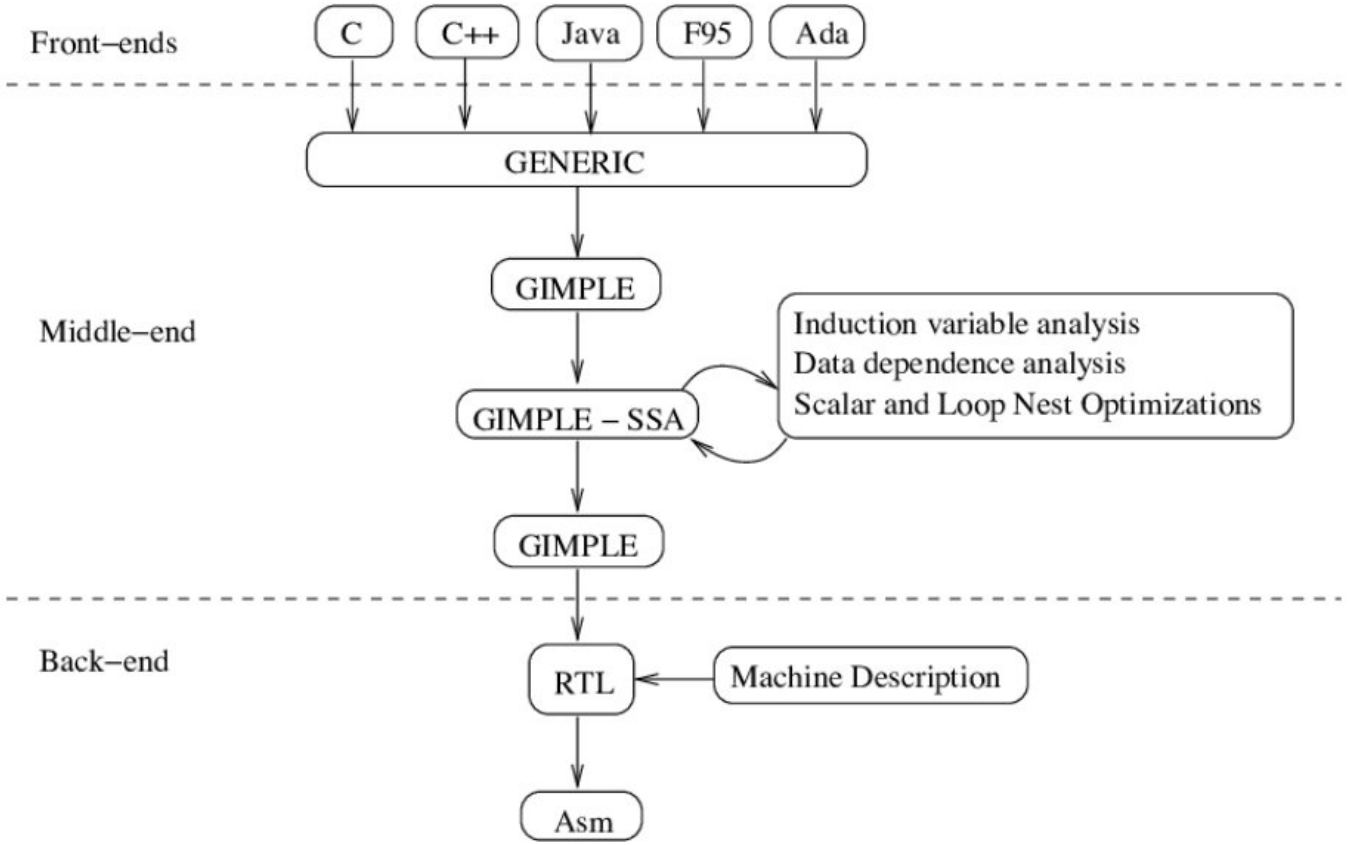
Clang LLVM

- Lexer: https://clang.llvm.org/doxygen/classclang_1_1Lexer.html
- Parser: https://clang.llvm.org/doxygen/classclang_1_1Parser.html
- IR: LLVM IR (SSA), LLVM MLIR



GCC

- Parser: https://github.com/gcc-mirror/gcc/blob/master/gcc/c/c-parser.h
- IR: GIMPLE (three-address IR)



IDE, Static Analyzers

Static Analayzers

- Lint ==> «линтеры»
- Clang Static Analyzer
- Synopsis Coverity
- Cppcheck
- Cpplint
- GCC -fanalyzer
- PVS-Studio

```
uninitStructMember  7459      rex.reg_match = NULL;
7434                7460      rex.reg_mmatch = rmp;
uninitStructMember  7461      rex.reg_buf = curbuf;       /* always works on the current buffer! */
7434                7462      rex.reg_firstlnum = lnum;
uninitStructMember  7463      rex.reg_maxline = curbuf->b_ml.ml_line_count - lnum;
7434                7464      rex.reg_line_lbr = FALSE;
uninitStructMember  7465      result = vim_regsub_both(source, NULL, dest, copy, magic, backslash);
7434                7466
uninitStructMember  7467      rex_in_use = rex_in_use_save;
7434                7468      if (rex_in_use)
uninitStructMember  7469          rex = rex_save; <--- Uninitialized struct member: rex_save.reg_match
7434                7470
uninitStructMember  7471      return result;
7469                7472  }
uninitStructMember  7473
7469                7474      static int
uninitStructMember  7475  vim_regsub_both(
7469                7476      char_u      *source,
uninitStructMember  7477      typval_T    *expr,
                    7478      char_u      *dest,
                    7479      int         copy,
                    7480      int         magic,
                    7481      int         backslash)
```