МЦСТ

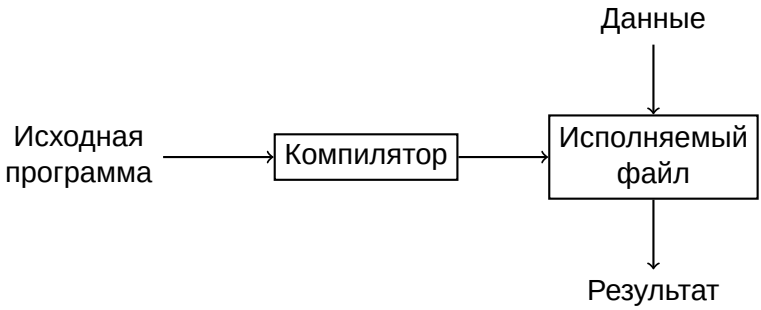
Устройство современных компиляторов

Лекция 1.
Введение в компиляторы.

Маркин А. Л.
alexanius@gmail.com

Часть 1.
Что такое компилятор.

Компилятор



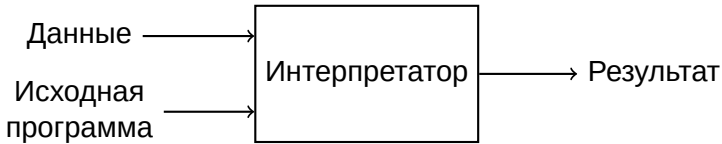
Компилятор — программа, принимающая на вход текст на одном (исходном) языке программирования, и возвращающая текст на другом (целевом) языке.

Компилятор

Современные компиляторы:

- GCC (GNU Compiler Collection) — набор компиляторов для различных языков программирования, разработанный в рамках проекта GNU;
- LLVM (Low Level Virtual Machine) — универсальная система анализа, трансформации и оптимизации программ, реализующая виртуальную машину с RISC-подобными инструкциями;
- Intel C++ compiler — оптимизирующий компилятор, разрабатываемый фирмой Intel для процессоров семейств x86, x86-64 и IA-64;
- Elbrus Compiler Collection — оптимизирующий компилятор, разрабатываемый фирмой МЦСТ для процессоров семейств Эльбрус и Sparc;

Интерпретатор



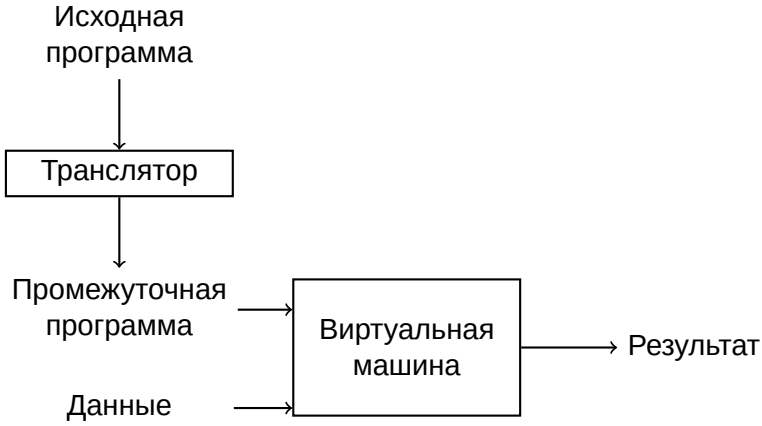
Интерпретатор — программа, принимающая на вход текст на одном (исходном) языке программирования, и выполняющая поданные на вход операции;

Интерпретатор

Современные интерпретаторы:

- Bash — командный процессор, работающий в интерактивном режиме. Используется для UNIX-подобных систем, в т.ч. для GNU/Linux;
- Perl — интерпретатор для языка программирования Perl;
- CPython — основной интерпретатор языка Python. Транслирует программу в промежуточное представление, а затем исполняет её.

Динамический компилятор



Динамический компилятор JIT (Just–In–Time) — программа, принимающая на вход текст на одном (исходном) языке программирования, транслирующая его в байт–код, исполняемый на виртуальной машине.

Динамический компилятор

Современные динамический компиляторы:

- JVM (Java Virtual Machine) — виртуальная машина для исполнения байт-кода языка Java (а также Clojure, Groovy, Scala и т.д.);
- CLR (Common Language Runtime) — виртуальная машина стека Microsoft .NET.

Сравнение применимости

	Компилятор	Интерпретатор	Динамический компилятор
Время компиляции	Долгое	Очень быстрое	Быстрое
Исполнение программы	Очень быстрое	Долгое	Быстрое
Дополнительных ресурсов на исполнение	Нет	Средне	Много

Схема работы компилятора

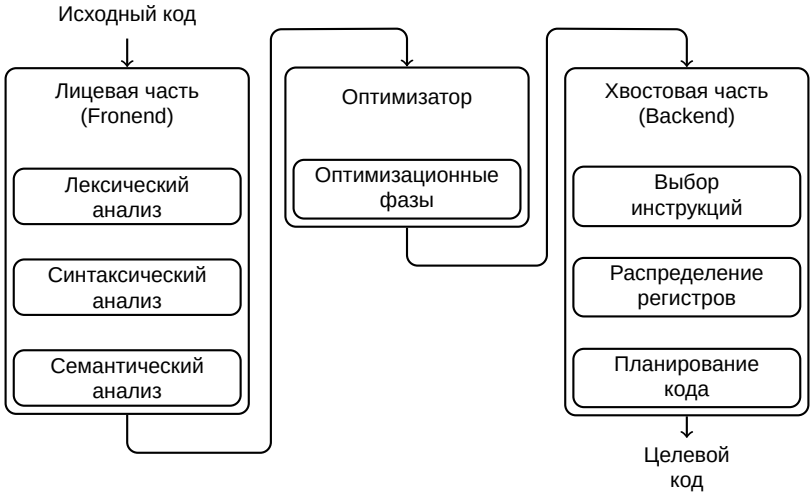


Схема работы компилятора

Лицевая часть проводит первоначальный разбор входной программы преобразование её к структурированному виду:

- **Лексический анализ** — преобразование потока символов в поток классифицированных слов
- **Синтаксический анализ** — определение принадлежности входного потока данному языку, построение дерева разбора программы
- **Семантический анализ** — более строгие проверки на соответствие правилам языка, проверка типов, построение таблицы символов и дерева абстрактного синтаксиса программы

Схема работы компилятора

Оптимизатор проводит анализы и преобразования программы с целями её ускорения, проверок на ошибки, уменьшения размера:

- **Оптимизационная фаза** — подпрограмма, анализирующая или модифицирующая входную программу

Схема работы компилятора

Хвостовая часть проводит преобразование программы из промежуточного представления компилятора к целевому языку:

- **Выбор инструкций** — преобразование инструкций промежуточного представления в инструкции целевой машины.
- **Распределение регистров** — преобразование кода для использования конечного числа регистров.
- **Планирование** — перестановка инструкций для ускорения исполнения целевой программы.

Схема работы компилятора

Какие программы вызывает `gcc`:

1. **cc1** — непосредственно компилятор. Преобразует входной файл в ассемблер
2. **as** — ассемблер. Преобразует файл с ассемблером в объектный файл
3. **collect2** — компоновщик. Производит связывание объектных файлов в итоговую программу.

Вывод: компилятор должен не только преобразовывать программу в целевой ассемблер, но и предоставлять инструменты для дальнейшей работы с ней (или уметь взаимодействовать с готовыми инструментами).

Схема работы компилятора

Инструменты, взаимодействующие с компилятором:

- интерпретатор промежуточного кода
- реализация стандартной библиотеки языка
- дополнительные библиотеки
- отладчик
- компоновщик
- профилировщик
- анализатор покрытия
- анализатор производительности
- прочие инструменты

Часть 2.
Программа внутри компилятора.

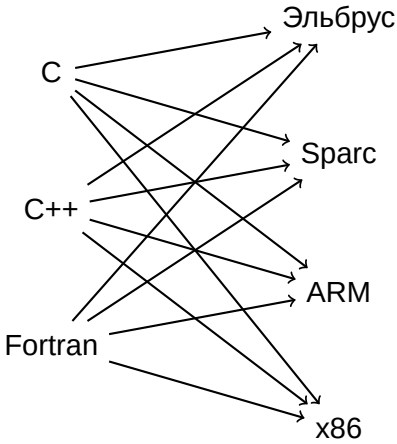
Промежуточное представление

Результатом работы лицевой части компилятора становится дерево абстрактного синтаксиса (AST - Abstract Syntax Tree):

```
-FunctionDecl 0x55ece71ed110 <t.c:1:1, line:4:1> line:1:5 main 'int (void)'
 `-CompoundStmt 0x55ece71ed250 <line:2:1, line:4:1>
   `-ReturnStmt 0x55ece71ed240 <line:3:5, col:12>
     `-IntegerLiteral 0x55ece71ed220 <col:12> 'int' 0
```

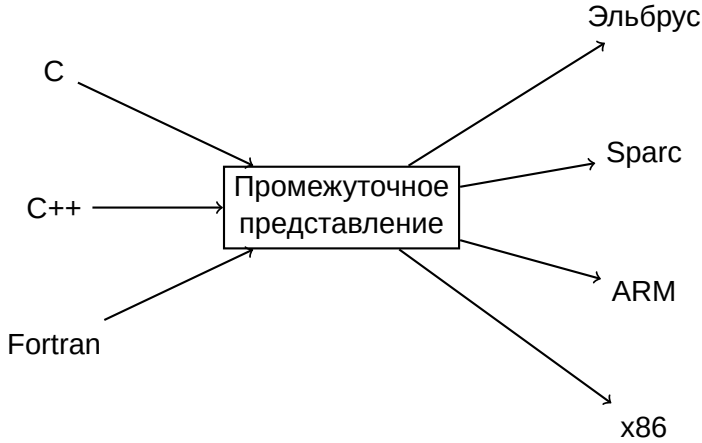
Промежуточное представление

Для каждого языка создаётся своя лицевая часть, поэтому для создания компиляторов M языков под N платформ могло бы понадобиться $M * N$ компиляторов:



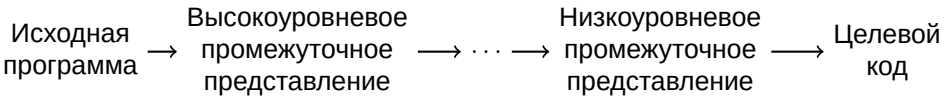
Промежуточное представление

Поэтому была выработана схема, которая предполагает общее независимое от языка **промежуточное представление**, в которое переводятся все программы всех поддерживаемых компилятором языков:



Промежуточное представление

В процессе трансляции компилятор может строить последовательность различных промежуточных представлений:



На высоком уровне возможно делать оптимизации и анализы, основанные на свойствах языка программирования.

На низком уровне возможно делать оптимизации, основанные на особенностях целевой машины

Промежуточное представление

Примеры одного и того же участка кода в представлениях gcc:

GENERIC

```
{
  return 0;
}
return 0;
```

GIMPLE

```
main ()
{
  int D.1909;

  {
    D.1909 = 0;
    return D.1909;
  }
  D.1909 = 0;
  return D.1909;
}
```

RTL

```
(note 3 0 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(note 2 3 9 2 NOTE_INSN_FUNCTION_BEG)
(insn 9 2 10 2 (set (reg/i:SI 0 ax)
        (const_int 0 [0])) "t.c":4:1 67 {*movsi_internal}
     (nil))
(insn 10 9 0 2 (use (reg/i:SI 0 ax)) "t.c":4:1 -1
     (nil))
```

Промежуточное представление

Для примеров мы будем использовать неформальную нотацию представления, обладающую свойствами трёхадресного кода. Таким образом обычный код:

```
a = b + c - d
```

Будет представлен в виде выражений:

```
t1 <- sub c, d
a  <- add t1, b
```

Промежуточное представление

При наличии ветвления код в представлении имел бы следующую форму:

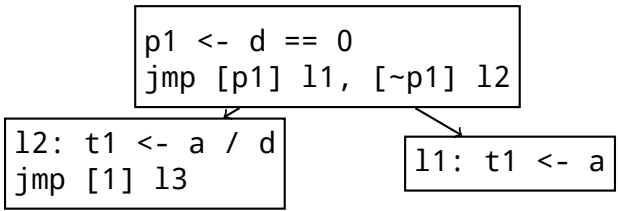
```
    p1 <- d == 0
    jmp [p1] l1, [~p1] l2
l2: t1 <- div a, d
    jmp [1] l3
l1: t1 <- a
l3:
```

Промежуточное представление

При наличии ветвления код в представлении имел бы следующую форму:

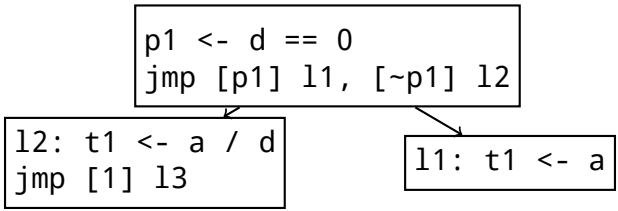
```
    p1 <- d == 0
    jmp [p1] l1, [~p1] l2
l2: t1 <- div a, d
    jmp [1] l3
l1: t1 <- a
l3:
```

Для упрощения работы с таким кодом принято строить **граф потока управления**.



Промежуточное представление

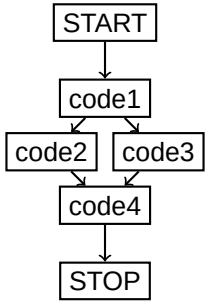
Узлами графа потока управления являются линейные участки — такие участки кода, попасть в которые можно только через начальную операцию, а выйти из которых только через конечную.



Т.е. это участок кода, начинающийся с метки перехода, и заканчивающиеся первой встреченной операцией перехода.

Промежуточное представление

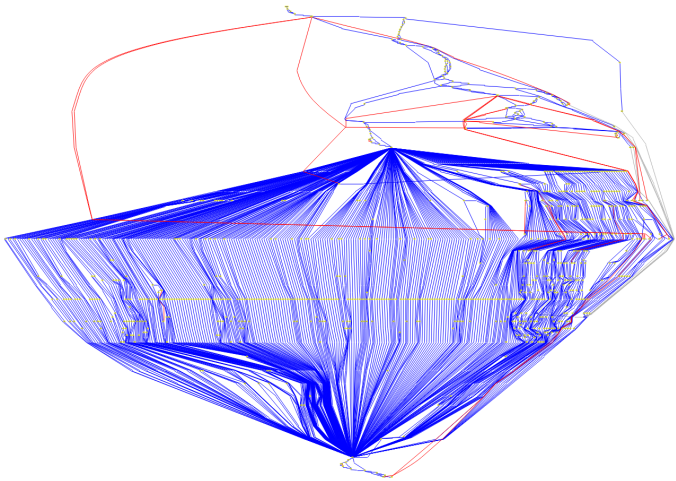
Дугами в графе потока управления будут возможные переходы от одного линейного участка к другому.



Для каждой процедуры строится свой собственный граф. Он всегда начинается с узла START и завершается узлом STOP.

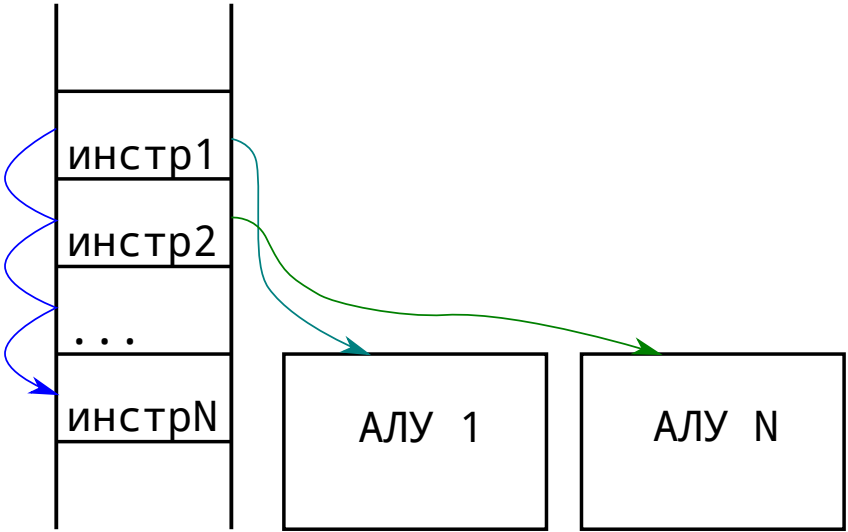
Промежуточное представление

Пример реального графа потока управления:



Часть 3.
Модель исполнения программы.

Модель исполнения программы



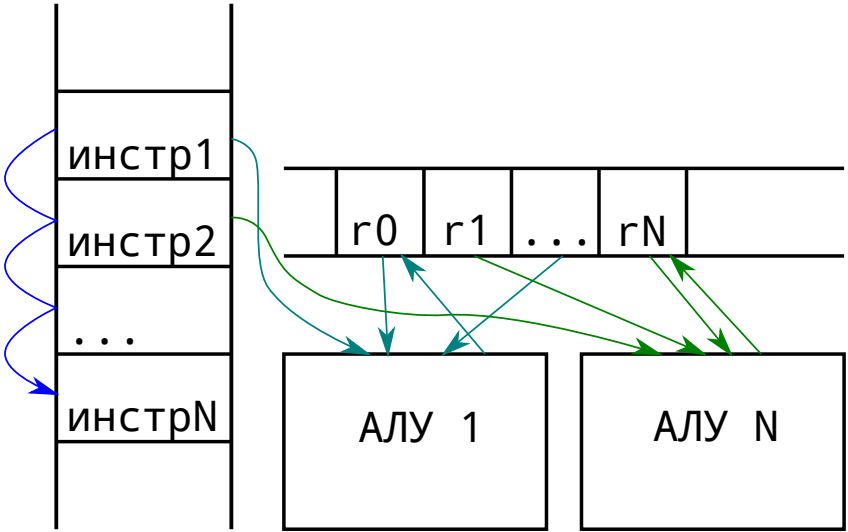
Модель исполнения программы

Программа — набор последовательных инструкций. Во время исполнения процессор идёт по каждой инструкции и отправляет её на исполняющее устройство.

Некоторые инструкции могут исполняться независимо друг от друга, поэтому в ядре процессора существует несколько исполняющих устройств для распараллеливания вычислений.

Такой вид параллелизма называется параллелизмом уровня инструкций (ILP — Instruction Level Parallelism). За него отвечают как процессор, так и компилятор.

Модель исполнения программы



Модель исполнения программы

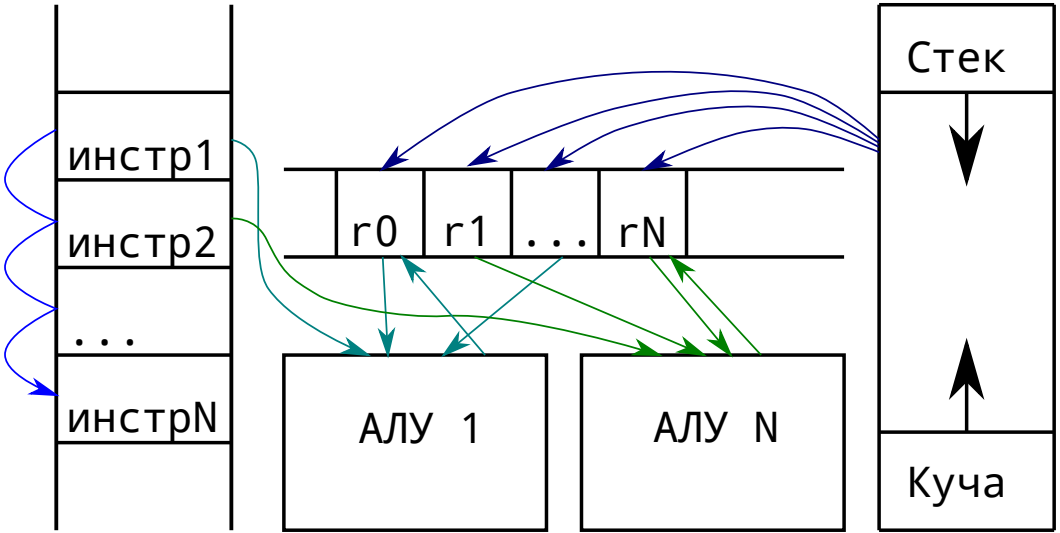
Регистры — самая память с самым быстрым доступом. Аргументы инструкций находятся именно в них.

Количество доступных пользователю универсальных регистров:

- x86_64: 16
- ArmV8: 31
- Эльбрус: 256

Сколько бы регистров ни предоставлял процессор — их всегда будет не хватать!

Модель исполнения программы



Модель исполнения программы

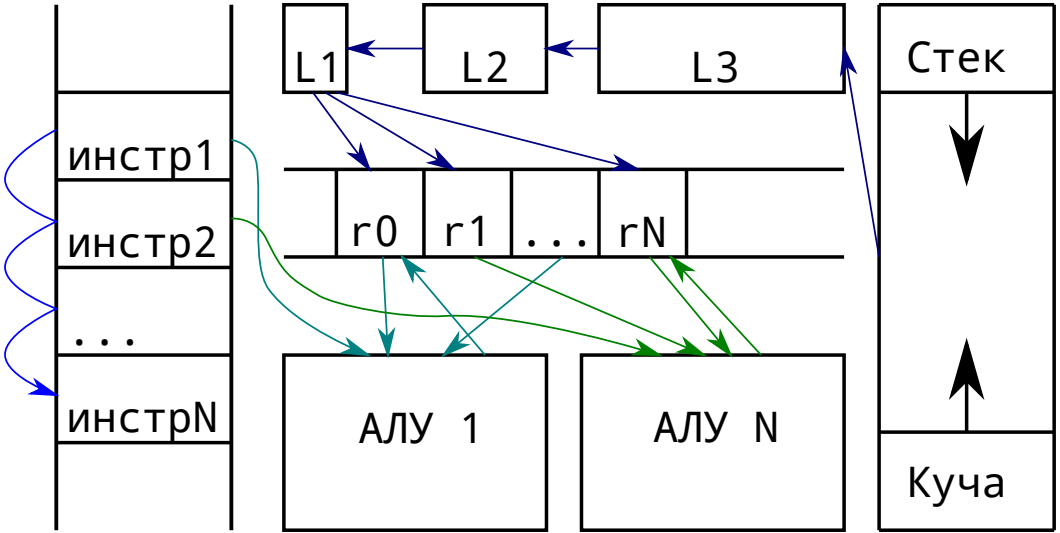
Процесс имеет много памяти, однако доступ к ней довольно длительный, может занимать десятки и стони тактов.

Память на стеке выделяется для каждой вызванной процедуры и содержит её локальные данные. После выхода из процедуры эти данные считаются удалёнными.

Память в куче выделяется и удаляется программистом при помощи вызовов `malloc`/`free` и им подобных.

Данные из памяти сначала передаются в регистры, а потом используются в качестве операндов инструкций. Чем чаще идут обращения в память тем больше время ожидания для подготовки операндов.

Модель исполнения программы



Модель исполнения программы

Для ускорения работы с памятью была придумана «сверхоперативная память», или **кэш**.

Кэши обладают малым временем доступа, но и при этом малым объёмом. Каждый следующий уровень кэша медленнее предыдущего. Примерные задержки извлечения данных из кэшей:

- L1: 3-5 тактов
- L2: 9 тактов
- L3: 25 тактов
- Оперативная память: >80 тактов

Объёма кэша всегда не хватает!

Модель исполнения программы

Чем более ценен ресурс тем больший дефицит испытает его потребитель.

Задача компилятора и процессора распределять ресурсы оптимальным образом, однако программисты стараются сделать всё чтобы усложнить компилятору эту задачу.

Понимание работы компилятора и грамотные подсказки могут помочь избежать потерь производительности и даже ускорить программу. Неправильные в свою очередь могут всё испортить.

Заключение.

Заключение

Компилятор — один из набора сложных инструментов для трансляции текста программы в исполняемый код.

В задачи компилятора помимо непосредственно трансляции входит анализ обрабатываемого кода на наличие ошибок, вывод максимально понятных сообщений об ошибках и ускорение компилируемой программы.

Каким бы хорошим ни был компилятор, он не поможет если код программиста будет плохим: будет запутанным, неоптимальным, несоответствующим стандарту.