# Компилятор

1. Компилятор – это программа, которая берет исходный файл с кодом и с помощью вспомогательных программ (препроцессора, линковщика, асемблера ) превращает его в файл с машинным кодом эквивалентный программе написанной в исходном файле. После чего файл с машинным кодом запускается вместо исходного и принимает входные данные, обрабатывает их, возвращает что то итд.
2. Берет весь исходный код программы и превращает его в отдельный исполняемый файл (машинный код).

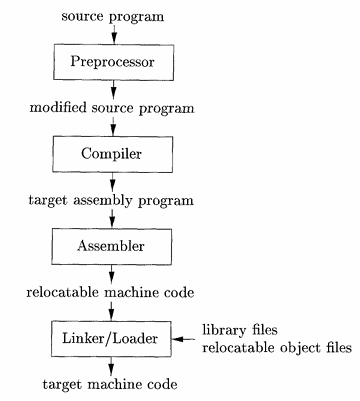
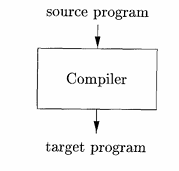
После компиляции сам исходный код больше не нужен для запуска программы — запускается уже сгенерированный файл.

Использует вспомогательные этапы: препроцессор, ассемблер, линкер.  
  
Пример: C, C++ (gcc, clang).

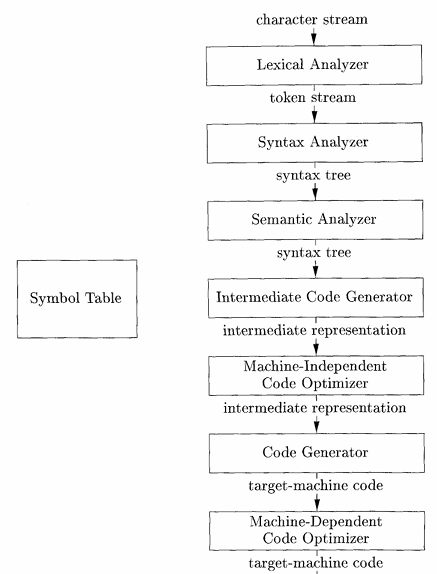
1. Компилятор релоцируемый генерирует код (ассемблер или машинный код).  
     
   Релоцируемый код — это код, в котором адреса функций и переменных могут быть изменены линкером при объединении нескольких объектных файлов в единый исполняемый модуль.

Релоцируемый код – код, в котором адреса функций и переменных заданы не как фиксированные значения, а как метки или относительные смещения. Благодаря этому при компоновке (linking) линкер может изменить эти адреса, чтобы согласовать их с кодом и данными из других объектных файлов и библиотек.

# Компиляция



1. Фазы компиляции



# Интерпретатор

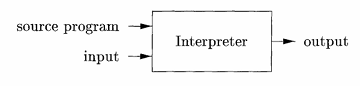
1. Интерпретатор - это программа, которая начинает работу при запуске исходного файла с кодом. Она берет исходный файл, входные данные и каждую строчку исходного файла разбирает на команды и выполняет.   
     
   В конце интерпретатор возвращает полученные значения в конце инструкций исходного файла с кодом.
2. Не создает отдельного исполняемого файла.

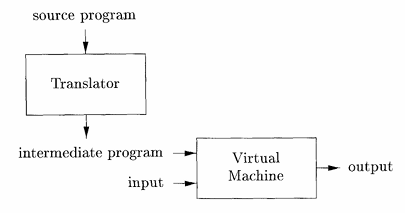
Запускает исходный код напрямую, построчно или поблочно, анализируя и выполняя команды “на лету”.

Каждый запуск программы требует интерпретатора.

Пример: Python, JavaScript (Node.js), Ruby.

# Интерпретация



1. Гибридная компиляция в Java.  
   

**Компилятор**:

* Берет весь исходный код программы и превращает его в отдельный исполняемый файл (машинный код).
* После компиляции сам исходный код больше не нужен для запуска программы — запускается уже сгенерированный файл.
* Использует вспомогательные этапы: препроцессор, ассемблер, линкер.
* Пример: C, C++ (gcc, clang).

**Интерпретатор**:

* Не создает отдельного исполняемого файла.
* Запускает исходный код напрямую, построчно или поблочно, анализируя и выполняя команды “на лету”.
* Каждый запуск программы требует интерпретатора.
* Пример: Python, JavaScript (Node.js), Ruby.

**Главная разница:**

* Компилятор — «собирает» программу заранее, результат можно запускать многократно без повторной компиляции.
* Интерпретатор — «читает и выполняет» код каждый раз при запуске.

# Pros & Cons

Преимущества компилятора над интерпретатором:

1. **Быстрее выполнение программы** — после компиляции создается машинный код, который процессор выполняет напрямую.
2. **Независимость исполняемого файла** — программу можно запускать без исходного кода или компилятора.
3. **Оптимизация кода** — компилятор может улучшить производительность, переписывая код, объединяя инструкции и удаляя ненужные операции.
4. **Защита исходного кода** — пользователь не видит исходные текстовые файлы, а только машинный код.

Преимущества интерпретатора над компилятором:

1. **Быстрое тестирование и отладка** — ошибки выявляются сразу во время выполнения, можно быстро проверять маленькие участки кода.
2. **Портативность исходного кода** — один и тот же файл можно запускать на разных платформах, где есть интерпретатор, без перекомпиляции.
3. **Динамическое выполнение** — можно выполнять код, который создается во время работы программы (например, eval в Python или JavaScript).
4. **Отсутствие этапа сборки** — код можно сразу запустить, не тратя время на компиляцию.

# Компиляция в асемблер

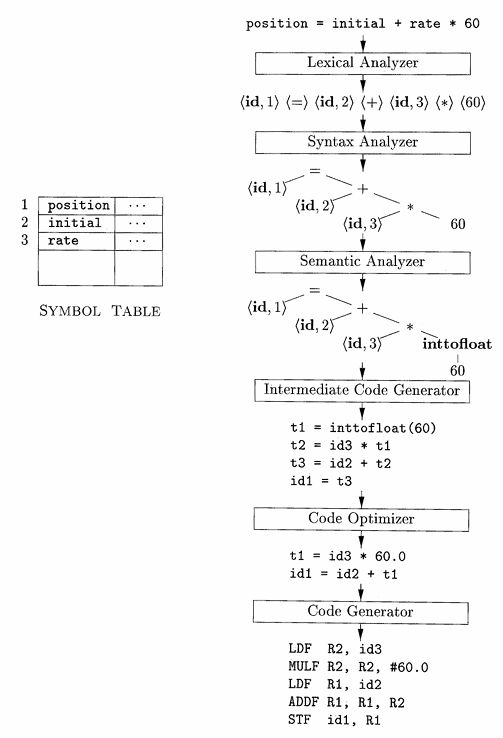
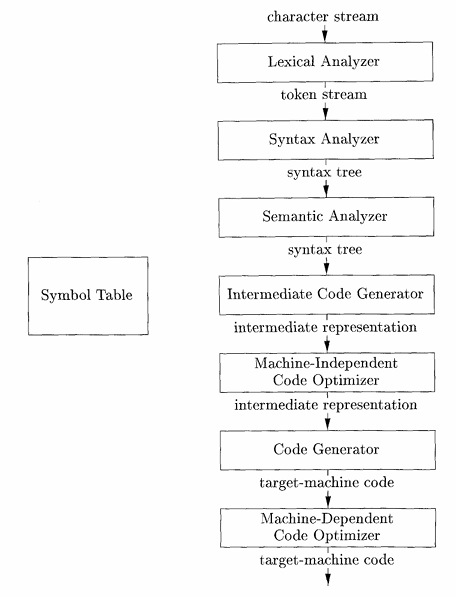
**Преимущества генерации ассемблера вместо машинного кода**

1. **Универсальность и переносимость** — как ты написал, один и тот же компилятор может работать на разных архитектурах, если он генерирует ассемблер под конкретный процессор, а локальный ассемблер уже соберёт машинный код.
2. **Упрощённая отладка** — ассемблерный код легче читать и связывать с исходным кодом, чем машинные инструкции, особенно при поиске ошибок.
3. **Оптимизация локальным инструментарием** — локальный ассемблер или линкер могут дополнительно оптимизировать код под конкретное окружение, чего компилятор “в вакууме” сделать не сможет.

# Задачи ассемблера

1. **Чтение и разбор исходного кода** — анализ синтаксиса ассемблера, определение команд, директив, меток, комментариев.
2. **Преобразование мнемоник в машинные инструкции** — перевод, например, MOV AX, 5 в соответствующий бинарный опкод и операнды.
3. **Обработка меток и символов** — создание *таблицы символов* с адресами функций, переменных и меток для последующей подстановки.
4. **Подстановка значений констант и вычисление адресов** — обработка выражений, смещений и относительных адресов в командах.
5. **Разрешение ссылок на внешние символы** — если код ссылается на функции или переменные из других файлов, ассемблер помечает их для последующей работы линкера.
6. **Генерация объектного файла** — формирование секций кода, данных и отладочной информации в формате, который понимает линкер (ELF, COFF, Mach-O и др.).
7. **Выдача диагностических сообщений** — сообщение об ошибках синтаксиса, неопределённых метках, неправильных операндах.

# Фазы компиляции

1. 
2. При анализе синтаксиса кода учитываются правила грамматики языка.   
     
   Все языки имеют строгие правила – **формальную грамматику**.  
     
   Однако, бывают разные типы грамматики.  
   Языки классифицируются в том числе по классам граматик (Context – Free Grammars and Languages, … )  
     
   Об этом в разделе ОСНОВЫ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ.