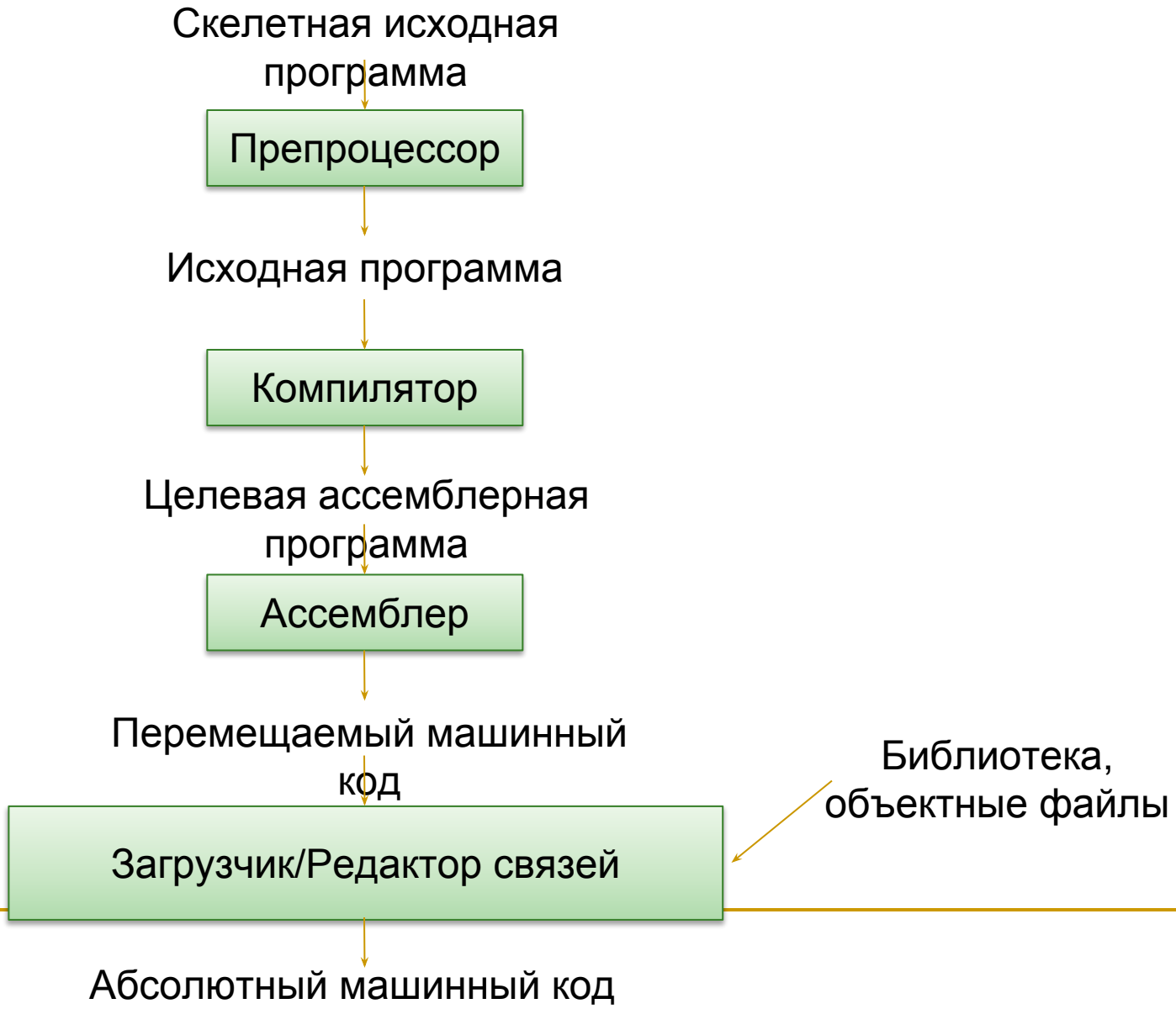
Общая схема работы компилятора. Задачи семантического анализа

Контекст компилятора



Контекст компилятора

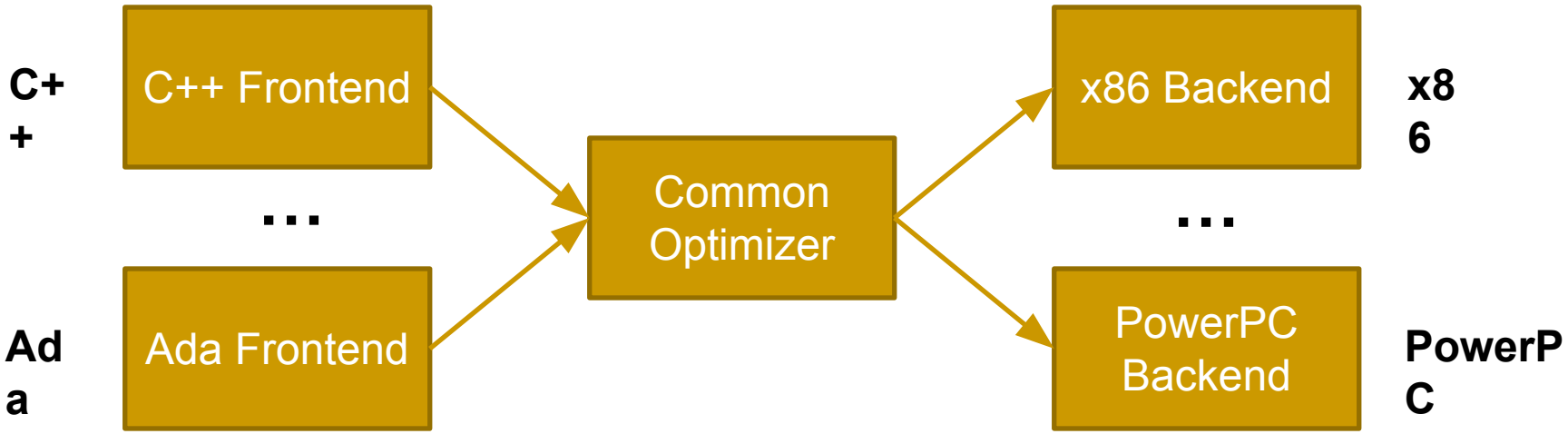
- **Препроцессоры -** создают входной поток информации для компилятора
  - обработка макросов;
  - включение файлов;
  - «интеллектуальные" препроцессоры;
  - языковые расширения…

Структура компилятора

- Frontend
  - парсинг исходного кода
  - синтаксический и семантический анализ
  - построение синтаксического дерева
- Oprimizer
  - преобразование представления с целью устранения избыточных действий (архитектура не учитывается)
- Backend
  - преобразование оптимизированного кода в машинное представление

Структура компилятора

- Возможность поддержки нескольких языков и нескольких платформ
- Добавление нового языка только новый Frontend
- Добавление новой архитектуры – новый Backend



Этапы компиляции

- Лексический анализ
- Синтаксический анализ
- Семантический анализ

Абстрактное синтаксическое дерево

- Генерация промежуточного кода
- Генерация машинного кода
- Профит

Генерация кода

- Конвертирование синтаксически корректной программы в последовательность исполняемых инструкций
- Может быть два этапа:
  - генерация промежуточного кода
  - генерация кода для целевой архитектуры

Генерация кода

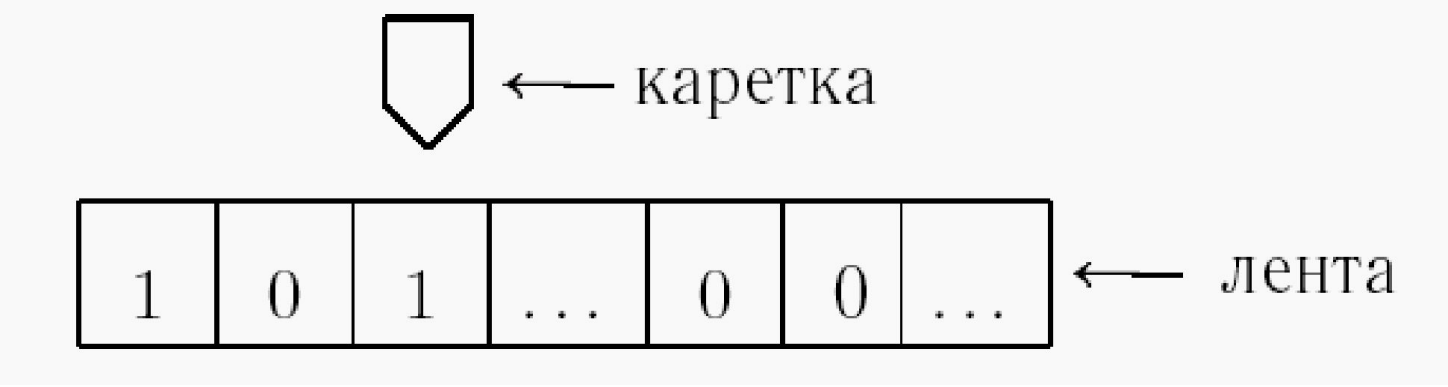
- Генерация промежуточного кода:
  - код для абстрактной машины (часто **трехадресной**)
  - идеализированный ассемблерный язык
  - бесконечное количество регистров (**MHP**)
- Генерация кода для целевой архитектуры:
  - выбор инструкций с учетом системы команд процессора
  - назначение регистров в качестве операндов для инструкций

Генерация кода

- **Абстрактной машиной** (**abstract machine**) называют математическую формализацию, которая моделирует правила выполнения программы (или, иначе, алгоритмы) для реальной вычислительной машины (компьютера).
- В настоящее время при практической реализации различных классов языков программирования, в частности функциональных и объектно-ориентированных языков, широко используются аналоги абстрактных машин в форме так называемых **виртуальных машин** (**virtual machine**).
- Виртуальные машины представляют собой средство создания промежуточного (следующего за текстом программы на высокоуровневом языке программирования) кода (именуемого в различных реализациях Java-кодом, MSIL-кодом и т.д.), который затем транслируется в машинный код.

Описание машины Тьюринга

- Машина Тьюринга - это воображаемое вычислительное устройство, имеющее следующие составные части.
- Оно имеет ленту, разбитую на ячейки, и каретку, расположенную в каждый конкретный момент работы
- м



Описание машины Тьюринга

- Каждая ячейка содержит ровно один из символов 0 или 1. Лента представляется конечной, но дополняемой в любой момент ячейками слева и справа для записи новых символов 0 или 1.
- Эта ситуация может возникнуть при сдвиге каретки влево или вправо за край ленты. Тогда наращивается новая клетка с содержимым 0.
- Это соглашение отражает идею о сколь угодно большой, но конечной памяти.
- Если каретка, расположена над некоторой ячейкой с символом 0 (с символом 1), то говорим, что каретка обозревает символ 0 (обозревает символ 1).

Описание машины Тьюринга

- Машина Тьюринга имеет программу.
- Это конечная последовательность инструкций q1, q2, ... ,qn, каждая из которых является строкой из 5 компонент:

(i, a, x, y, z),

где i – номер инструкции;

a =0, или a=1;

x=0, или x=1;

y= L или y=R;

z – номер инструкции.

Машина с неограниченными регистрами

- **Машина с неограниченными регистрами** (МНР) – это абстрактная машина, более сходная с реальным компьютером по сравнению с машиной Тьюринга.
- Она имеет следующие составные части:
- **1)** Регистры R_1, R_2, ..., в которых содержатся соответственно натуральные числа r_1, r_2, ...
- Число регистров бесконечно, но только конечное множество регистров R_1, R_2, ... R_k содержит числа, отли[illegible]
- Все [illegible] и.

Машина с неограниченными регистрами

- **2)** Программа машины – это конечная последовательность $I_1, I_2, \ldots, I_s$ из следующих четырех типов команд:

Z(n), S(n), T(m, n), J(m, n, q),

где m, n, q - {1,2, ... }.

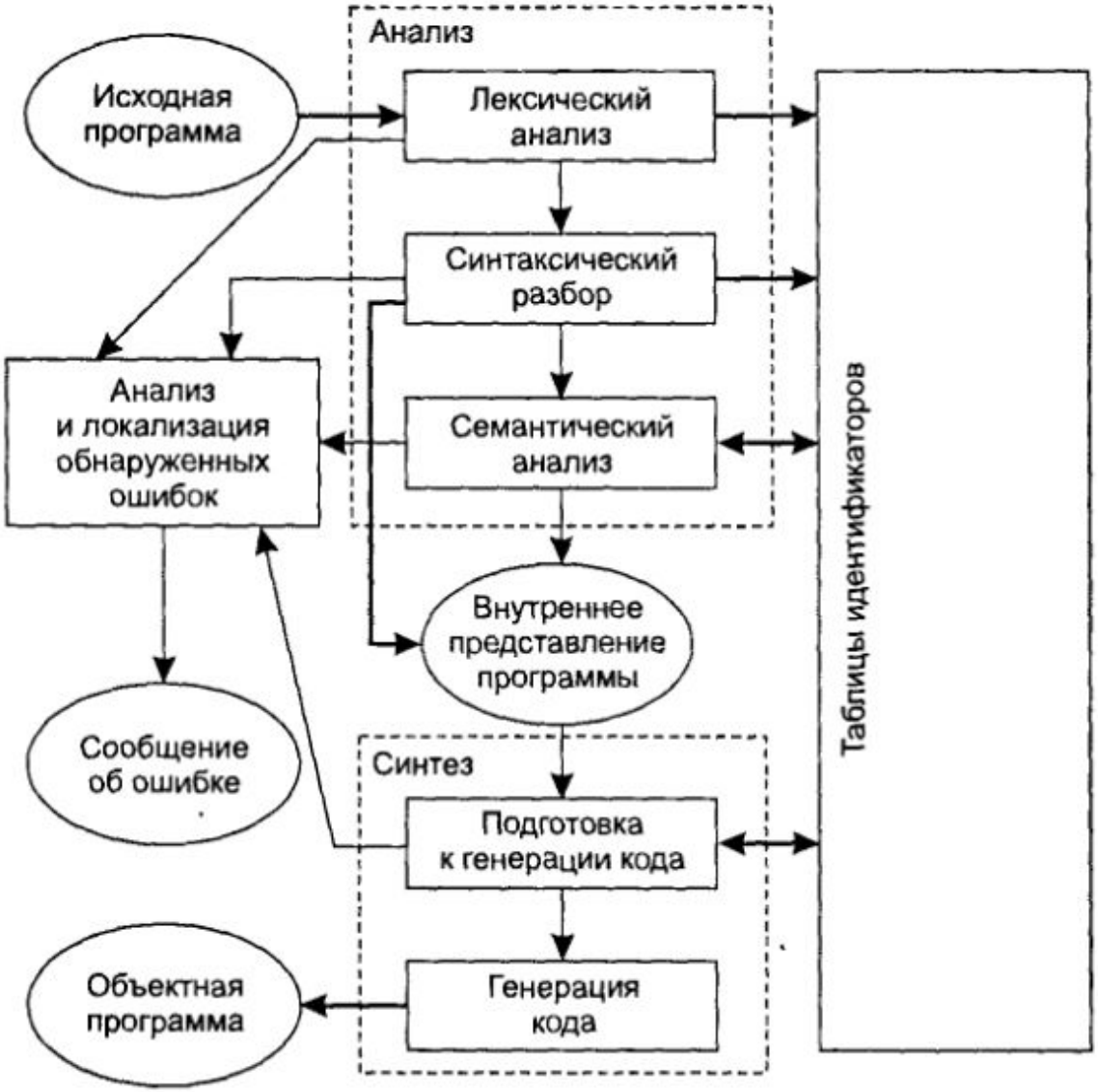
Эти команды выполняют следующие действия.

- Команда обнуления **Z(n)** делает содержимое регистра Rn равным нулю.
- Команда прибавления единицы **S(n)** к содержимому регистра R_n прибавляет число 1.
- Команда переадресации T(m, n) заменяет содержимое регистра R_n на содержимое регистра R_m.

Машина с неограниченными регистрами

- Команда условного перехода J(m, n, q) сравнивает содержимое регистров R_m и R_n.
- При $r_m = r_n$ в качестве следующей команды выполняется команда с номером q, в противном случае выполняется следующая по порядку команда программы.
- Команды обнуления, прибавления единицы и переадресации называются **арифметическими командами**.

Общая схема работы компилятора



Основные <u>этапы</u> компиляции:

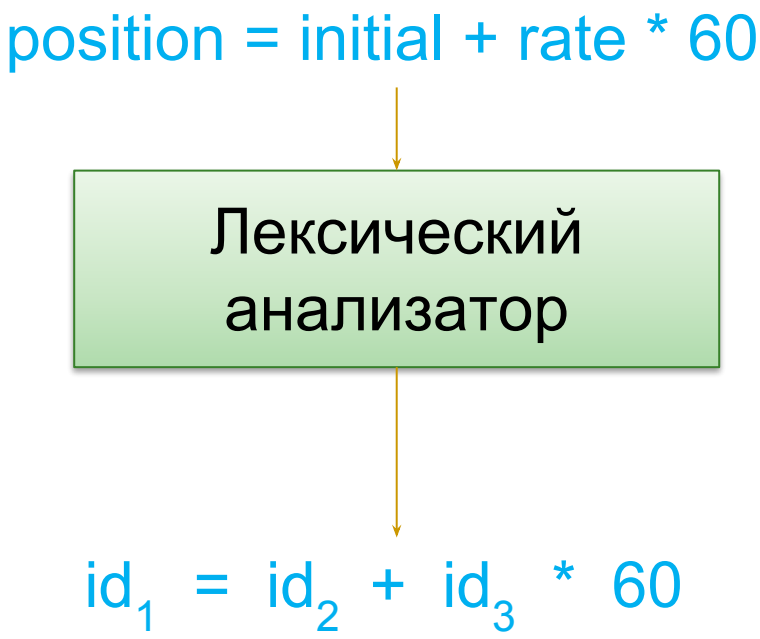
- анализ
- синтез

Основные <u>фазы</u> компиляции:

- лексический анализ
- синтаксический разбор
- семантический анализ
- подготовка к генерации кода
- генерация кода

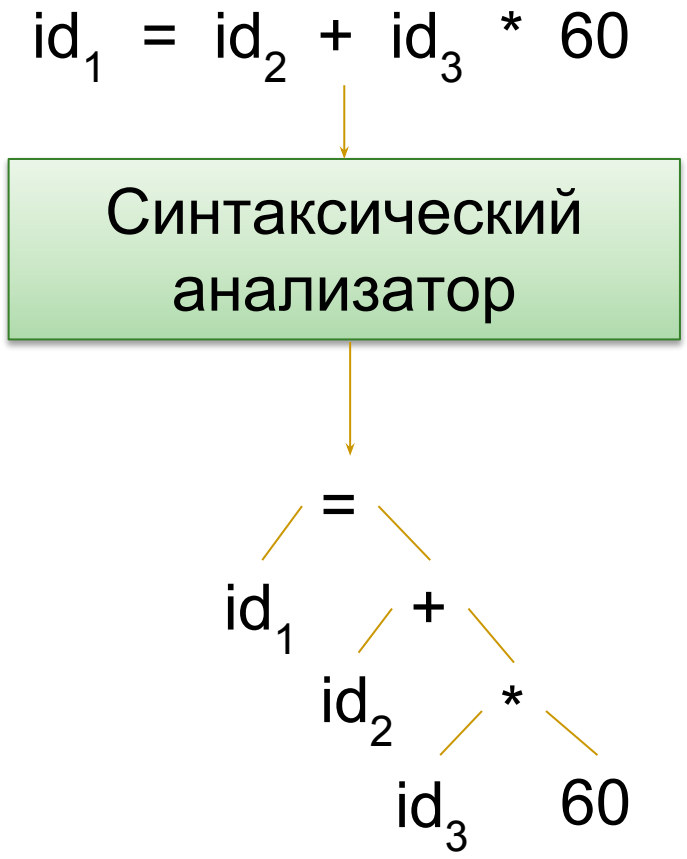
Лексический анализ

- При **лексическом анализе** символы исходной программы считываются (слева направо) и группируются в поток **токенов** (token), в котором каждый токен представляет логически связанную последовательность символов: идентификатор, ключевое слово (if, while и т. п.), символ пунктуации,…



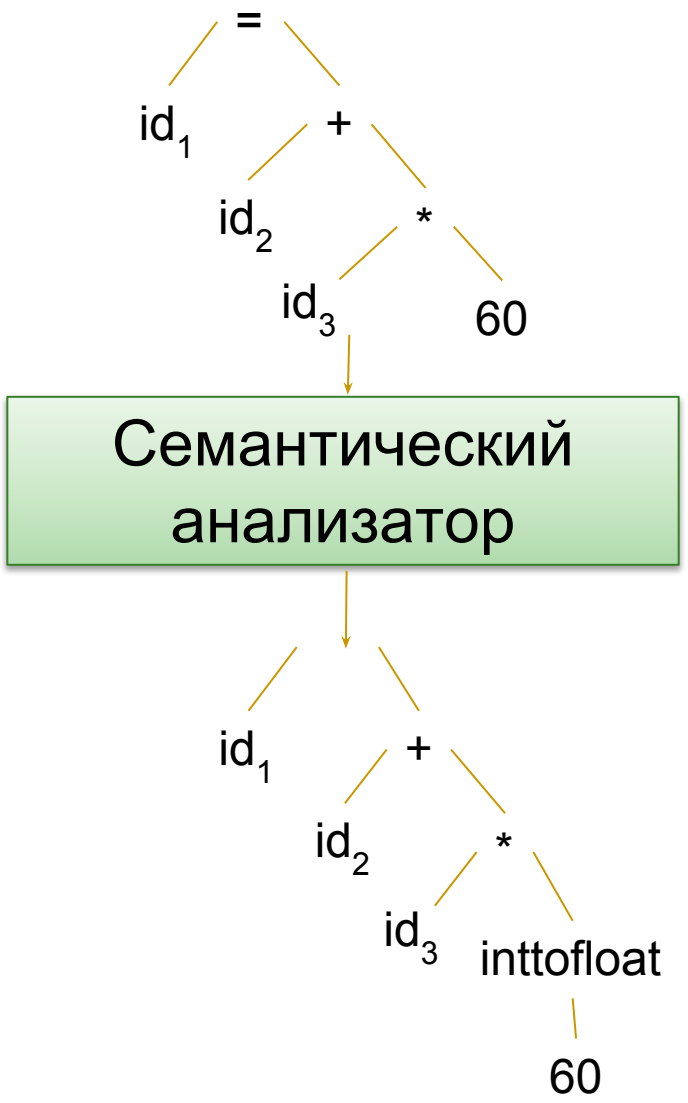
Синтаксический анализ

- **Синтаксический разбор** – основная часть компилятора на этапе анализа:
  - выполняется выделение и иерархическое группирование синтаксических конструкций в тексте исходной программы, обработанном лексическим анализатором
  - проверяется синтаксическая правильность программы



Семантический анализ

- В процессе **семантического анализа:**
  - проверяется наличие семантических ошибок в исходной программе и накапливается информация о типах
  - производится идентификации операторов и операндов выражений и инструкций

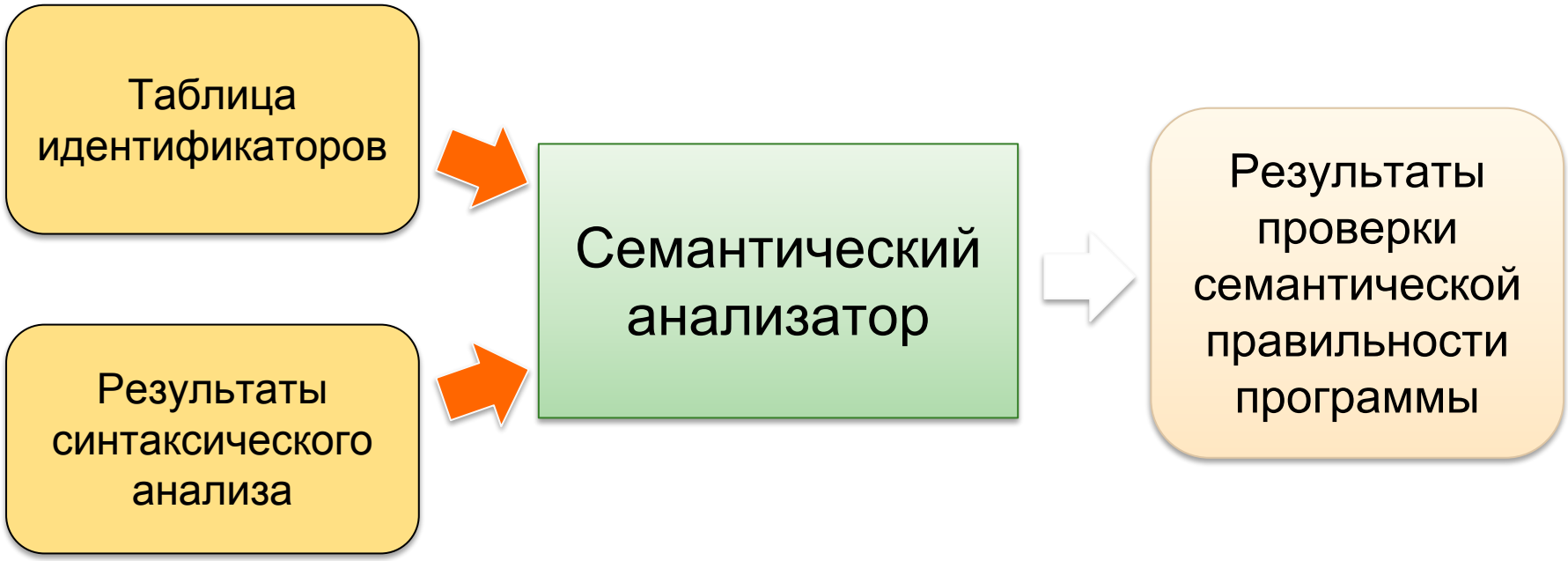


Назначение семантического анализатора

Семантический анализатор выполняет следующие основные действия:

- проверку соблюдения во входной программе **семантических соглашений** входного языка;
- **дополнение внутреннего представления программы** в компиляторе операторами и действиями, неявно предусмотренными семантикой входного языка;
- **проверку элементарных семантических** (смысловых) **норм** языков программирования, напрямую не связанных со входным языком;

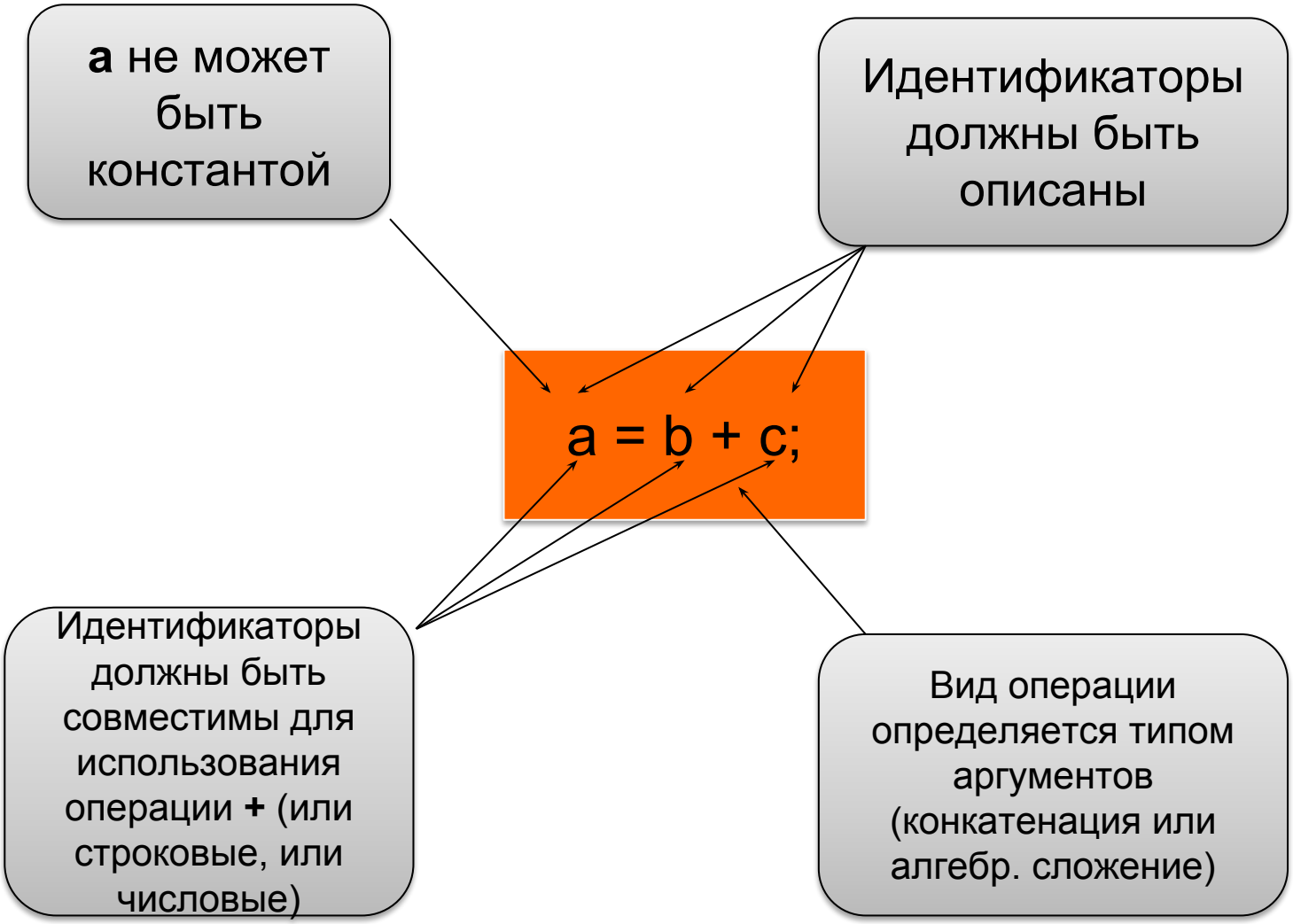
Схема работы



Проверка соблюдения семантических соглашений

- каждая метка, на которую есть ссылка, должна один раз присутствовать в программе;
- каждый идентификатор должен быть описан один раз и ни один идентификатор не может быть описан более одного раза (с учетом блочной структуры описаний);
- все операнды в выражениях и операциях должны иметь типы, допустимые для данного выражения или операции;
- типы переменных в выражениях должны быть согласованы между собой;
- при вызове процедур и функций число и типы фактических параметров должны быть согласованы с числом и типами формальных параметров;
- и т.д.

Проверка соблюдения семантических соглашений



Дополнение внутреннего представления программы

- Связано с добавлением в текст программы операторов и действий, неявно предусмотренных семантикой входного языка:
  - преобразование типов операндов в выражениях и при передаче параметров в процедуры и функции;
  - операции вычисления адреса, когда происходит обращение к элементам сложных структур данных.

Дополнение внутреннего представления программы

Исходный текст

```
a = b + c;
```

```
double a;
int b;
float c;
```

Код, порождаемый компилятором

```
a = double(
float(b) + c);
```

(Без использования явного преобразования типов)

Вариант разработчика

```
a = double(b
+ trunc(c));
```

(С использованием явного преобразования типов)

Проверка смысловых норм языков программирования

Обеспечивает проверку компилятором соглашений, выполнение которых связано со смыслом как всей исходной программы в целом, так и отдельных ее фрагментов:

- каждая переменная или константа должна хотя бы один раз использоваться в программе;
- каждая переменная должна быть определена до ее первого использования при любом ходе выполнения программы (первому использованию переменной должно всегда предшествовать присвоение ей какого-либо значения);
- результат функции должен быть определен при любом ходе ее выполнения;
- каждый оператор в исходной программе должен иметь возможность хотя бы один раз выполниться;
- операторы условия и выбора должны предусматривать возможность хода выполнения программы по каждой из своих ветвей;
- операторы цикла должны предусматривать возможность завершения цикла;
- ...

Проверка смысловых норм языков программирования

```
int f_test(int a)
{    int b, c;
     b = 0;
     c = 0;
     if (b=1) { return a;}
     c = a + b;
}
```

Идентификация лексических единиц языков программирования

- **Идентификация** переменных, типов, процедур, функций и др. лексических единиц языков программирования – это установление однозначного соответствия между лексическими единицами и их именами в тексте исходной программы:
  - имена лексических единиц не должны совпадать как между собой, так и с ключевыми словами синтаксических конструкций языка;
  - локальные переменные имеют область видимости.

Идентификация лексических единиц языков программирования

На этапе семантического анализа каждой лексической единице языка дается уникальное имя в пределах всей исходной программы и потом используется при синтезе результирующей программы:

- имена локальных переменных дополняются именами тех блоков (функций, процедур), в которых эти переменные описаны;
- имена внутренних переменных и функций модулей исходной программы дополняются именами самих модулей;
- имена процедур и функций, принадлежащих объектам (классам) в объектно-ориентированных языках программирования дополняются наименованиями типов объектов (классов), которым они принадлежат;
- имена процедур и функций модифицируются в зависимости от типов их формальных аргументов и др.

Вспомогательные фазы компилятора

- **Управление таблицей символов**
  - **Таблица символов (идентификаторов)** - структура данных, содержащую записи о каждом идентификаторе и связанными с ними характеристиками в течение всего процесса компиляции (сведения об отведенной идентификатору памяти, его типе, области видимости, …), чтобы иметь возможность использовать их на различных фазах компиляции
- **Обнаружение ошибок и сообщение о них**

Таблицы идентификаторов

Назначение – хранение идентификаторов и их характеристик

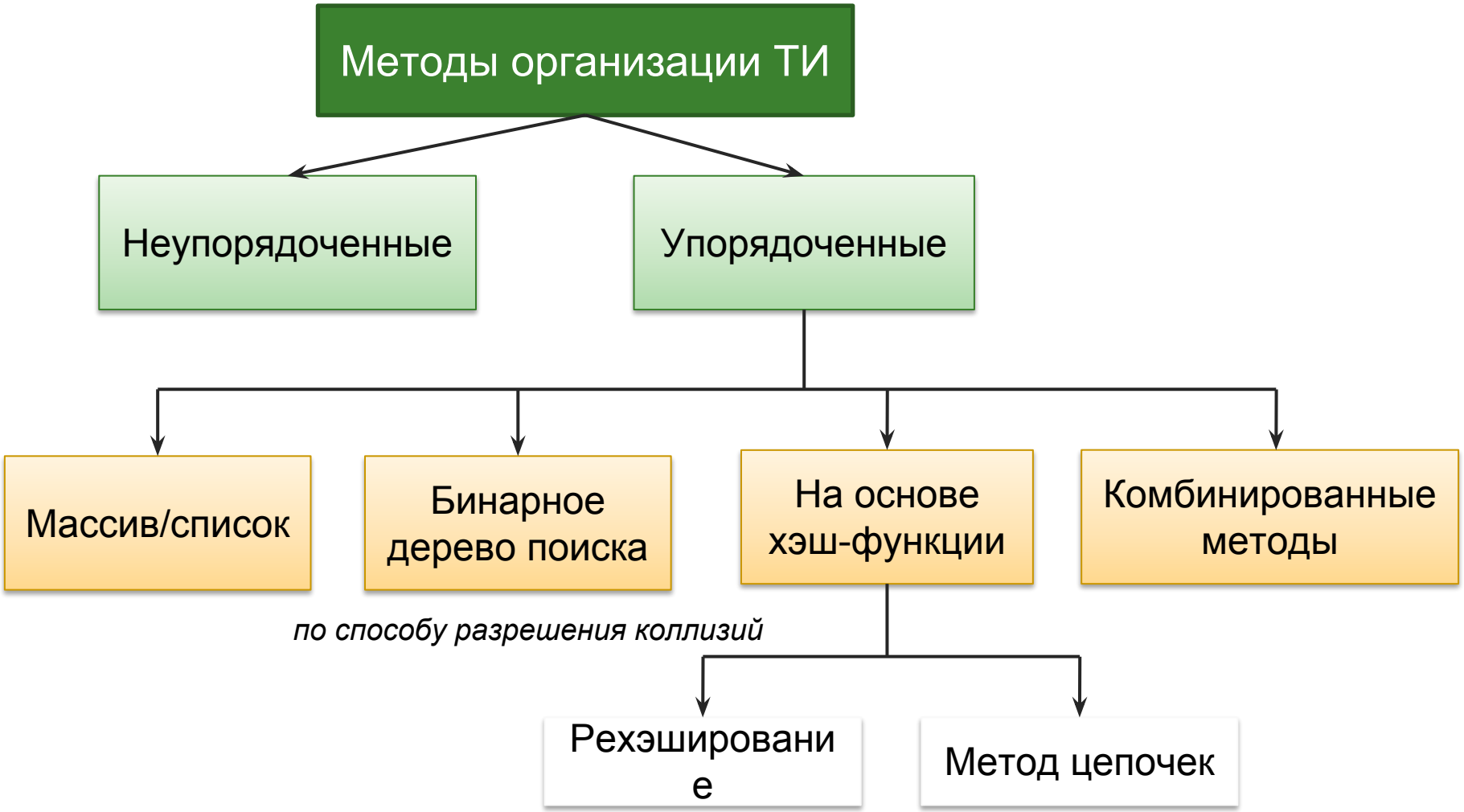
Особенности:

- возможен значительный объем хранимой информации;
- может быть одна ТИ или несколько (например, для различных модулей или различных типов элементов);
- состав информации, хранимой в ТИ, зависит от семантики входного языка и типа элемента;
- информация в ТИ заполняется не сразу, а по мере того, как становится известна – на различных фазах компиляции;
- принцип работы с ТИ – многократное обращение компилятора для поиска информации и записи новых данных на различных фазах компиляции;
- **ТИ должны быть организованы таким образом, чтобы обеспечить максимально быстрый поиск нужного элемента**

Таблицы идентификаторов

Имя объекта					*Описание объекта*
s	o	r	t		
a					
r	e	a	d		
i					

Таблицы идентификаторов

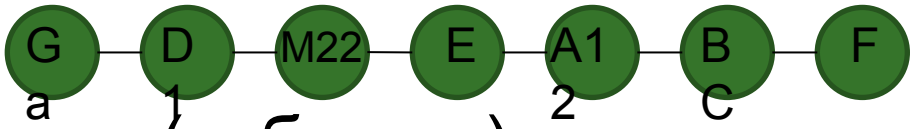


Состав информации, хранимой в ТИ

- для переменных
  - имя переменной;
  - тип данных переменной;
  - область памяти, связанная с переменной;
- для констант:
  - название константы (если оно имеется);
  - значение константы;
  - тип данных константы (если требуется);
- для функций:
  - имя функции;
  - количество и типы формальных аргументов функции;
  - тип возвращаемого результата;
  - адрес кода функции.

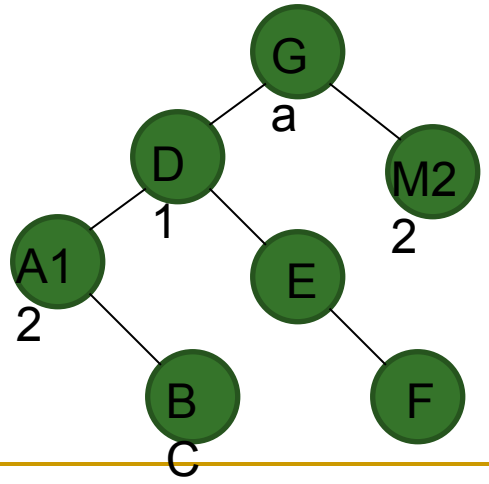
Простейшие методы построения ТИ

- Списки (таблицы)
  - $T_з \sim 0$; $T_п = O(N)$



- Упорядоченные списки (таблицы)
  - $T_п = O(\log_2 N)$

- Бинарное дерево (узел – элементы таблицы)
  - $T_з = N*O(\log_2 N)$; $T_п = O(\log_2 N)$



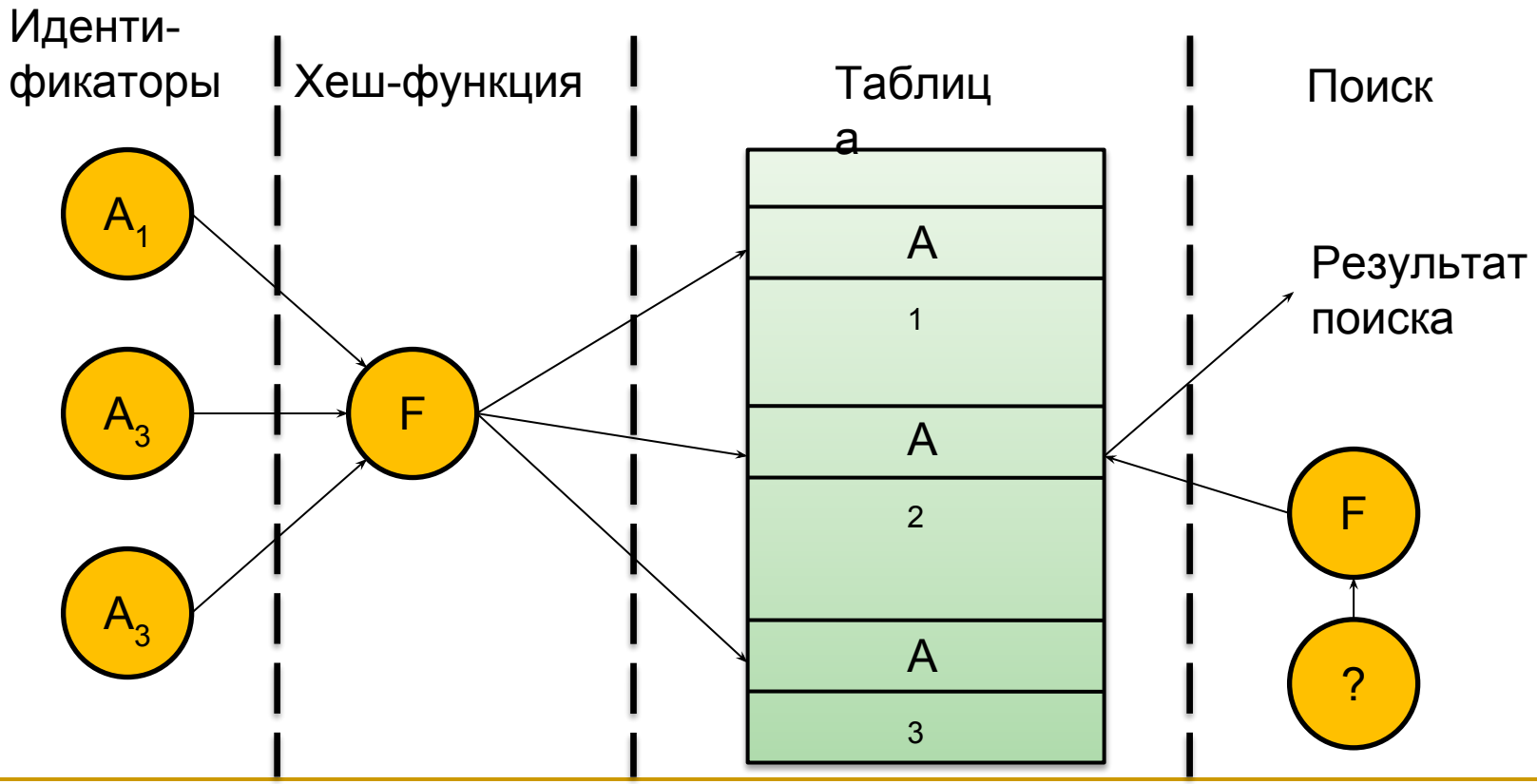
Последовательность идентификаторов:
GA, D1, M22, E, A12, BC, F

Хеш-функции

- **Хеш-функцией** F называется некоторое отображение множества входных элементов *R* на множество целых неотрицательных чисел *Z*: $F(r) = n, r \in R, n \in Z$
  - Область определения хеш-функции - множество допустимых входных элементов R
  - Множество значений хеш-функции - $M \subseteq Z$, содержащее все возможные значения, возвращаемые функцией F
- **Хеширование** - процесс отображения области определения хеш-функции на множество значений

Хеш-адресация

- **Хеш-адресация** заключается в использовании значения, возвращаемого хеш-функцией, в качестве адреса ячейки из некоторого массива данных

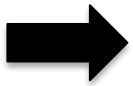


Построение ТИ на основе хеш-функций

- **Хеширование** обычно достигается за счет выполнения над цепочкой символов некоторых простых арифметических и логических операций:
  - = коду внутреннего представления в компьютере литеры символа
- Недостатки хеш-адресации:
  - Неэффективное использование объема памяти под таблицу идентификаторов;
  - Необходимость разумного выбора хэш-функции;
  - **Коллизии** - ситуация, когда двум или более идентификаторам соответствует одно и то же значение функции.

Разрешение коллизий: рехэширование (метод открытой адресации)

- если для элемента A адрес h(A), вычисленный с помощью хэш-функции h, указывает на уже занятую ячейку, то необходимо вычислять значения функций $n_i = h_i(A)$ до тех пор, пока не будет найдена свободная ячейка

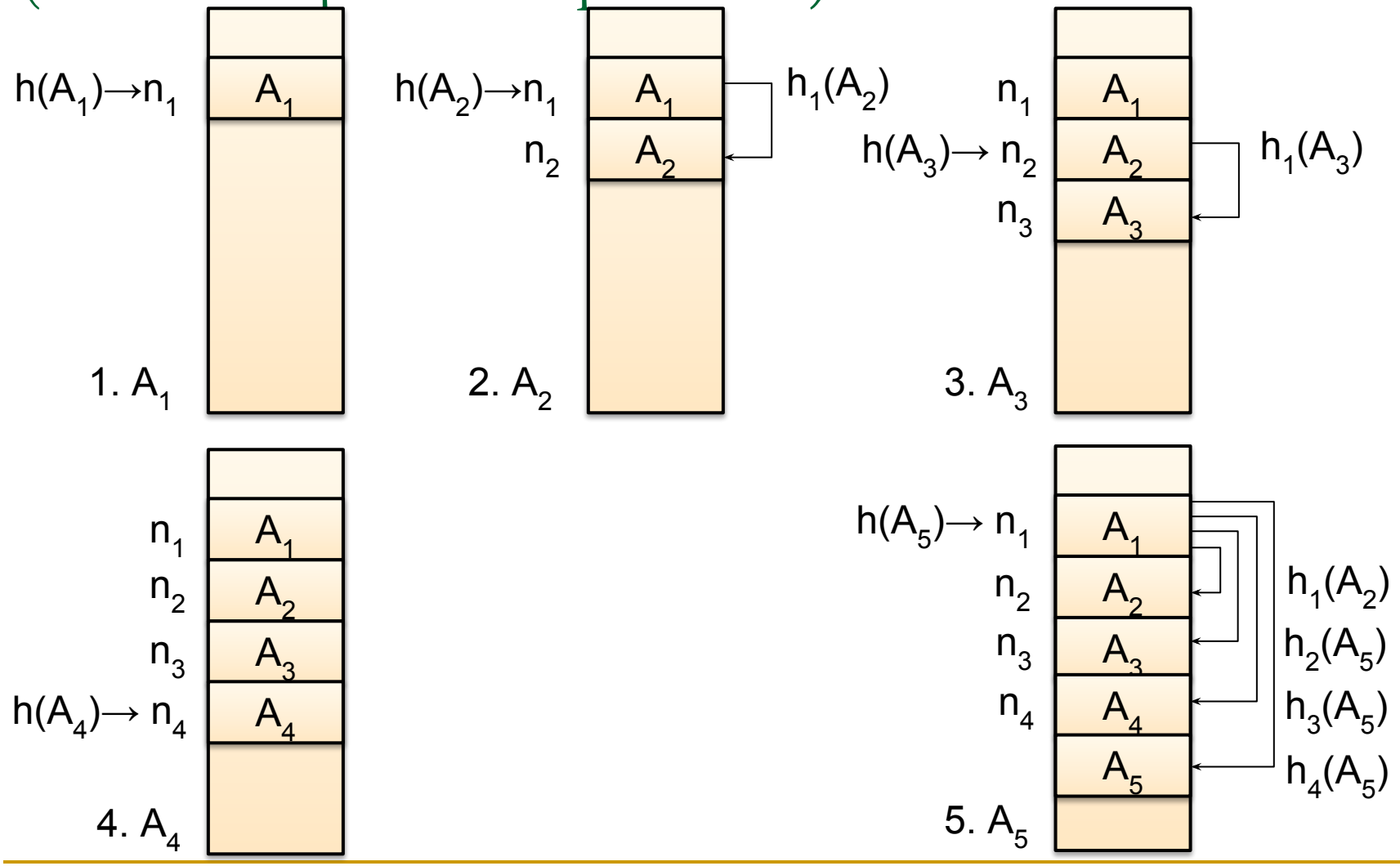
$h_i=(h(A)+p_i)\bmod N_m$  $h_i=(h(A)+i)\bmod N_m$

$h_i=(h(A)*i)\bmod N_m$

P_i – вычисляемое число; N_m - максимальное значение из области значений хеш-функции

$T_п=O((1-L_f/2)/(1-L_f))$

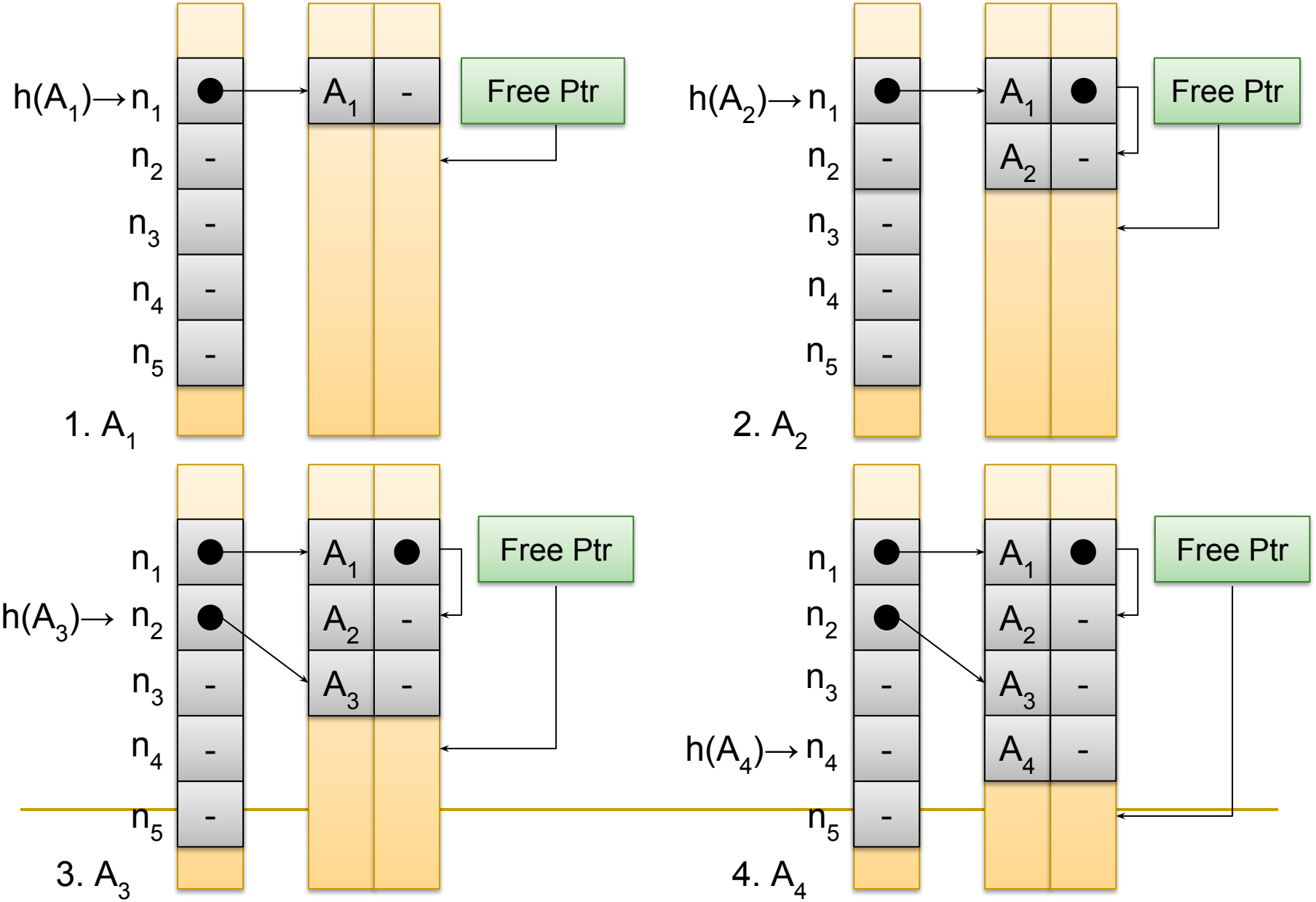
Разрешение коллизий: рехэширование (метод открытой адресации)



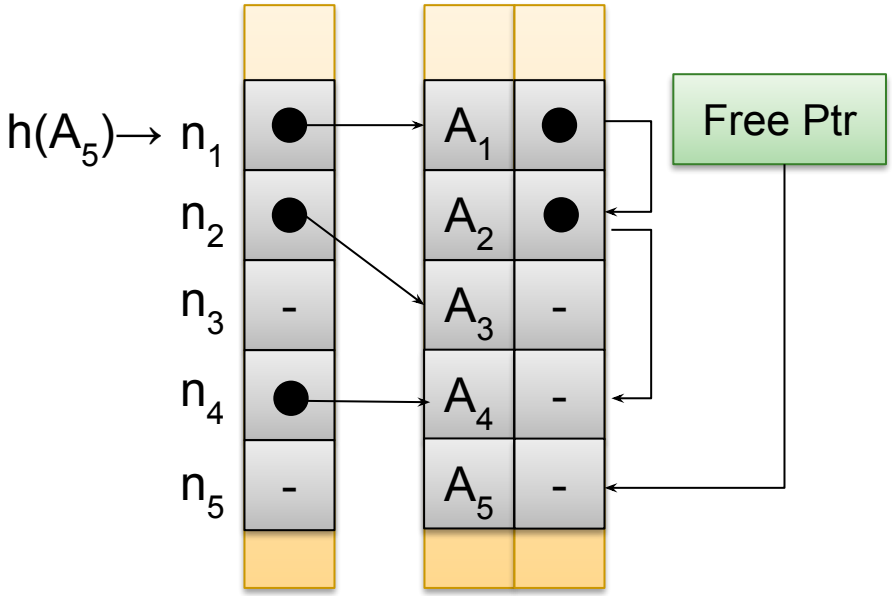
Метод цепочек

- Основная идея – использование «промежуточной» хеш-таблицы со значениями указателей на области памяти из основной таблицы идентификаторов

Метод цепочек



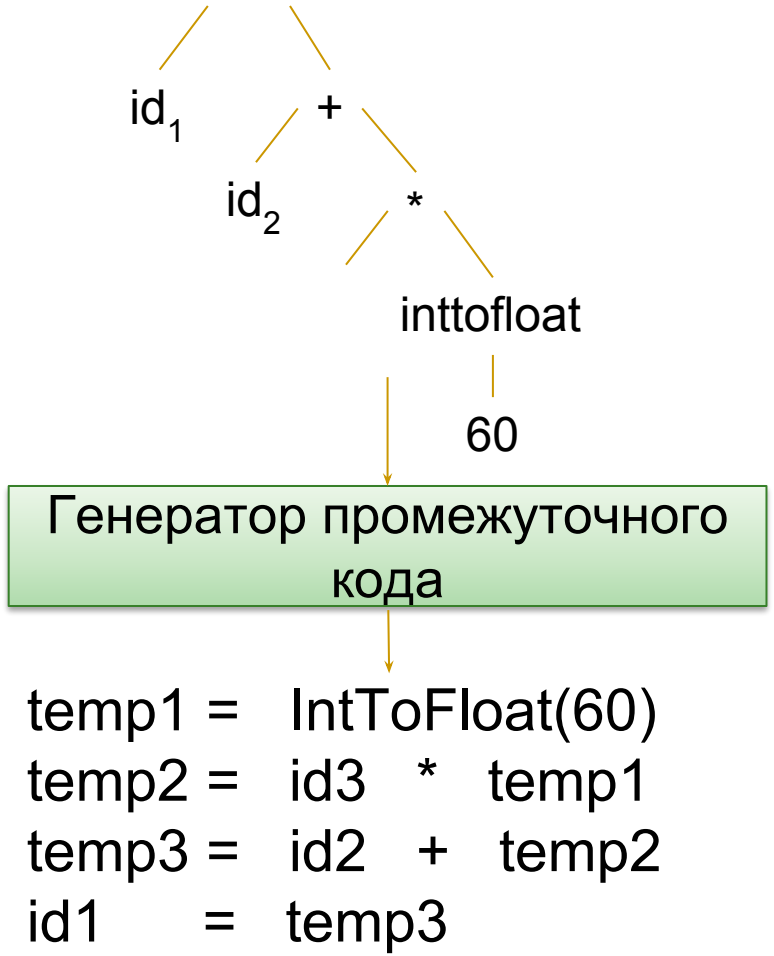
Метод цепочек



5. A_5

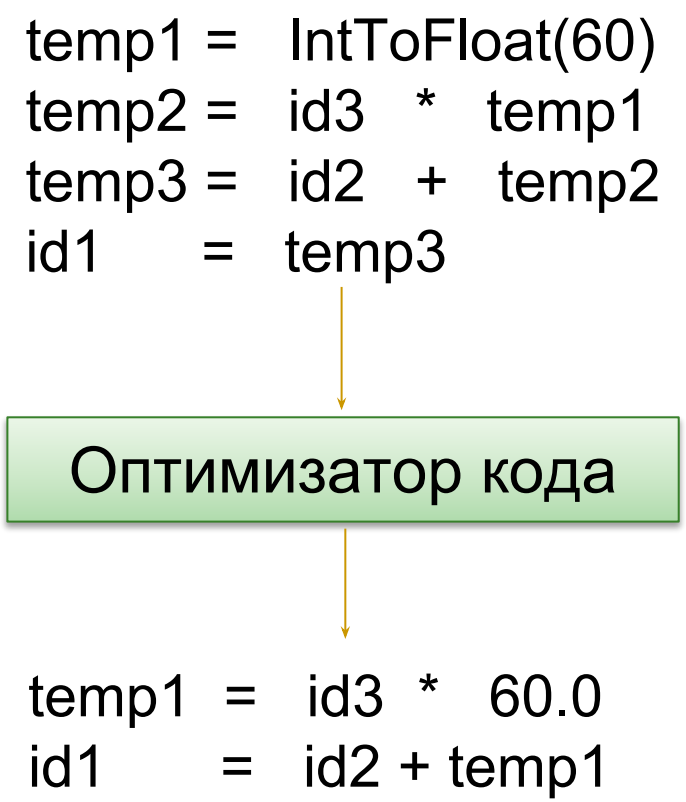
Генерация промежуточного кода

- Генерация явного промежуточного представления исходной программы (программа для абстрактной машины)



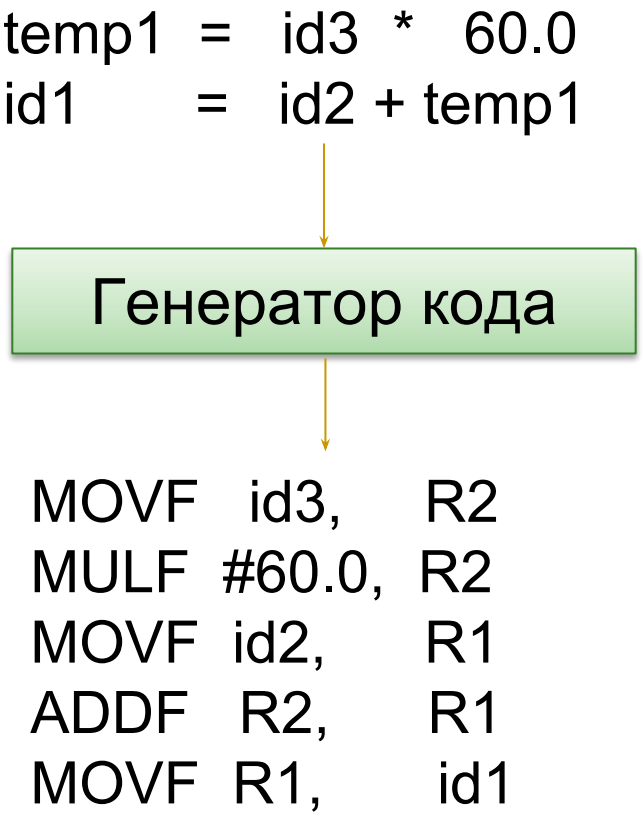
Оптимизация кода

- При оптимизации кода производятся попытки улучшить промежуточный код, чтобы получить более эффективный машинный код.



Генерация кода

- При оптимизации кода производятся попытки улучшить промежуточный код, чтобы получить более эффективный машинный код.



Группировка фаз

- **Начальная стадия** объединяет те фазы компилятора, которые зависят в первую очередь от исходного языка и практически не зависят от целевой машины
  - Лексический и синтаксический анализ, создание таблицы символов, семантический анализ и генерация промежуточного кода
- **Заключительная стадия** состоит из тех фаз компилятора, которые в первую очередь зависят от целевой машины, для которой выполняется компиляция, и, вообще говоря, не зависят от исходного языка (а только от промежуточного) + часть оптимизации кода и генерация выходного кода (сопровождаемые необходимой обработкой ошибок и работой с таблицей символов)

Проходы

- **Проход** – процесс последовательного чтения компилятором данных из внешней памяти, их обработки и помещения результата работы во внешнюю память
  - Чаще всего один проход включает в себя выполнение одной или нескольких фаз компиляции
  - Наиболее распространены 2х- и 3х-проходные компиляторы, например: 1-ый проход — лексический анализ, 2-ой - синтаксический разбор и семантический анализ, 3-ий — генерация и оптимизация кода
  - Желательно иметь компилятор с минимальным числом проходов

Проходы

Количество проходов – важная техническая характеристика компилятора.

Результат прохода – внутреннее представление исходной программы (кроме последнего), которое хранится в оперативной памяти или во временных файлах на диске.

- **Количество проходов зависит от грамматики и семантических правил исходного языка**

Проходы(пример)