**Компиляторы**

**Компиляция** – процесс перевода программного кода в машинный без его запуска.

**Компиляция**:

1. **Анализ** – разбиение исходной программы на составные части и создание **промежуточного представления**. Создаются синтаксические деревья (**дерево** синтаксического **разбора**), в которых каждый узел – операция, дочерние узлы – аргументы операции
2. **Синтез** – конструирование целевой программы из промежуточного представления

**Анализ**

Состоит из трёх фаз:

1. Линейный анализ (Лексический) – поток символов читается слева направо и группируется в токены.
   1. Пробелы и комментарии отбрасываются
   2. После того, как символы сгруппируются в токен, они записываются в **таблицу символов** и удаляются из входного потока, начинается сканирование следующего токена (?происходит замена каждой лексемы на её тип и ссылку на её значение в таблице?)
   3. Если есть макросы и директивы препроцессора, то они также реализовываются
   4. Токен – терминальный символ грамматики (ключ слова, операторы, констант, ; и т.д.), лексема – последовательность символов, которая соответствует шаблону, шаблон – правило, описывающее набор лексем.
   5. Промежуточный шаг в создании такого анализатора – построение диаграммы переходов (дка): состояние 0, считали токен, если есть, то состояние 1, иначе ошибка
2. Иерархический анализ (Разбор, синтаксический анализ) – символы или токены иерархически группируются во вложенные конструкции
   1. Создается **дерево разбора** ((абстрактное) **синтаксическое дерево** – дерево разбора в сжатом виде – нет элементов, которые не влияют на семантику программы – разделители, группирующие скобки – иногда представляется в виде ациклического графа/трёхадресный код). АСД – один из вариантов промежуточного представления. Оно получается при LL и LR-анализе, при анализе методом рекурсивного спуска
   2. Определение иерархического расположения блоков
   3. \*для построения предсказывающего анализатора – FIRST(u) (множество терминалов, с которых начинаются строки, выводимые из u) и FOLLOW (множество нетерминалов, которые могут появиться непосредственно справа)
3. Семантический анализ
   1. проверяется наличие ошибок
   2. накапливается информация о **типах** для следующей стадии – генерации кода + проверяется соответствие типов: каждый оператор имеет операнды допустимого типа
   3. используются иерархические структуры, полученные во время синтаксического анализа

Заметки:

1. Разделение на лексический и синтаксический произвольно. Фактор: использование рекурсии – лексические правила не требуют рекурсии, а в синтаксических очень часто она нужна.
2. Синтаксическое дерево – сжатое дерево разбора, операторы размещены во внутренних узлах, операнды – дочерние ветви узла

**Фазы компилятора:**

1. Лексический анализатор
2. Синтаксический анализатор
3. Семантический анализатор
4. Генератор промежуточного представления
5. Оптимизация кода
6. Генератор кода

**Таблица символов**

1. \*формируется на этапе лексического анализа
2. Содержит инфу о:
   1. Об отведенной идентификатору памяти
   2. Тип идентификатора
   3. Его область видимости
   4. И т.д.
3. – СД, содержащая записи о каждом идентификаторе с полями для его атрибутов

**Ошибки**

1. В процессе синтаксического и семантического анализа обрабатывается большая часть
2. Лексический анализ – символы не образуют ни один из токенов языка
3. Поток токенов нарушает структурные правила (синтаксис) языка – синтаксический анализ
4. Конструкции, не имеющие смысла с т.з. операций – сложение названия массива и названия процедуры

**Препроцессор**

1. Создают входной поток информации для компилятора
   1. Обрабатывают макросы
   2. Включают файлы

**Двухпроходный ассемблер**

Два прохода (= считывание файла) по входному потоку:

1. Находятся все идентификаторы и размещаются в таблице символов, отличной от таблицы символов компилятора, им назначаются адреса памяти
2. Переводит:
   1. код операции - в последовательности битов, представляющих операцию на машинном языке;
   2. идентификатор – в адрес, назначенный ему в таблице символов
3. Итог: получается перемещаемый машинный код (может быть загружен в память с любого стартового адреса) = объектный модуль

**Загрузчик**

Выполняет две функции:

1. Загрузка
   1. Получение перемещаемого машинного кода
   2. Изменение перемещаемых адресов
   3. Размещение изменённых команд и данных по корректным адресам в памяти
2. Редактирование связей
   1. Собирает единую программу из нескольких файлов (файлы могут быть получены различными компиляциями, например, библиотечные файлы) с перемещаемым машинным кодом. Т.к. файлы используются вместе => есть внешние ссылки. Файл с перемещаемым машинным кодом должен содержать информацию о таблице символов

ЯП может быть определён с помощью описания того, как должна выглядеть программа (= синтаксис языка), для определения синтаксиса применяется запись, называемая контекстно-свободной грамматикой (BNF).

Грамматика – четверка:

1. Множество нетерминальный символов
2. Множество терминальных
3. Множество правил
4. Начальный символ грамматики

Дерево разбора

1. В процессе разбора определяется, может ли некоторая строка токенов быть сгенерирована данной грамматикой
2. Два метода разбора:
   1. Нисходящий (сверху вниз) – построение начинается от корня по направлению к листьям – популярный
      1. Пример: метод рекурсивного спуска (убираем левую рекурсию)
   2. Восходящий (снизу вверх) – от листьев по направлению к корню

Промежуточное представление –> код

1. Ll файл парсится, строится граф управления потоком (узлы – блоки кода, ребра указывают на поток управления между блоками)
2. Оптимизация кода – удаление мертвого кода, слияние блоков и т.д.
3. Распределение регистров
4. Генерация машинного кода с помощью llvm backend