**Компилятор** – программа, которая считывает текст программы на входном языке и транслирует его в текст программы на целевом языке с сохранением семантики, а также с указанием ошибок в программе на входном языке.

**Основные фазы**

*Поток символов* 🡪 Лексический анализатор 🡪 *Поток токенов* 🡪 Синтаксический анализатор 🡪 *Синтаксическое дерево* 🡪 Семантический анализатор 🡪 *Синтактическое дерево* 🡪 Генератор промеж. кода 🡪 *Промеж. представление* 🡪 Машинно-независимый оптимизатор 🡪 *Промеж. представление* 🡪 Генератор кода 🡪 *Код целевой машины* 🡪 Машинно-зависимый оптимизатор кода 🡪 *Код целевой машины*

**Статический компилятор (ahead of time - AOT)** – выполняет полную трансляцию программы до её выполнения.

**Динамический компилятор (just in time)** – выполняет динамическую трансляцию промежуточного кода в исполняемый код целевой архитектуры.

**Промежуточные представления.**

Входная программа содержит лишь поток символов. После лексического анализа формируются токены – таблица символов (<имя\_токена, токен>). После синтаксического анализа формируется АСТ. Далее, АСТ может быть модифицировано на этапе семантического анализа, который проверяет программу на семантическую согласованность с определением языка. После семантического анализа генератор промежуточного кода преобразовывает АСТ в чаще всего трёхадресный код.После генерации последнего, оптимизатор преобразовывает код различными методами, чаще всего сокращая количество инструкций. Далее, генератор кода преобразовывает оптимизированный код в целевой язык, а затем, машинно-зависимый оптимизатор используя аппаратные возможности преобразует код последний раз.

**Формальный язык** – множество L-конечных слов над конечным алфавитом А.

**Синтаксис формального языка** – набор правил, описывающих корректный вид его программ.

**Семантика формального языка** – набор правил, описывающих “смысл” программ (области видимости, совместимость типов данных, наследование, ...).

**Регулярные выражения для описания языков** – средство, с помощью которого можно описать некоторые формальные языки, с учётом того, что язык сематически не поддерживает сбалансированность аттрибутов. Область применения регулярных выражений – определение языковых лексем на этапе лексического анализа. Альтернативой регулярным выражениям являются продукции.

**Формальная грамматика языка** – способ описания формального языка, выделение подмножества L из множества слов некоторого алфавита А. Грамматика языка состоит из: G = {V­T, VN, P, S}. Языку принадлежат только те строки, которые можно получить с помощью грамматики.

**Эквивалентные грамматики** – такие грамматики, которые описывают один и тот же язык. То есть, все строки, которые можно получить с помощью одного языка, можно получить и с помощью ему эквивалентного языка.

**Иерархия Хомского** – классификация формальных языков, описывающая их условную сложность. Регулярные 🡪 Контекстно-свободные 🡪 Контекстно-зависимые 🡪 Неограниченные. Для описания простейших конструкций языка используются регулярные грамматики, для описания синтаксиса языка – контекстно-свободные. При этом один и тот же язык может быть задан несколькими типами грамматик, в таком случае следует относить его к более лёгкому типу.

**Форма Бэкуса-Наура** – компьютерное описание контекстно-свободных грамматик.

**A close-up of a computer code

Description automatically generated**

**Синтаксически управляемая трансляция** выполняется путём присоединения правил или программных фрагментов к продукциям грамматики. Для неё используются атрибуты – некоторые величины, связанные с программной конструкцией (тип данных, количество команд, ...). А также, схемы трансляции – запись присоединённых к продукциям грамматики программных фрагментов.

Атрибуты назначаются для терминалов и нетерминалов для присоединения к продукциям грамматики правила, которые описывают, каким именно образом происходит вычисление атрибутов в узлах дерева разбора.

Для данной входной строки строится дерево разбора, а затем, для вычисления атрибутов в каждом узле применяются семантические правила. Такое дерево, с назначенными атрибутами, называется аннотированным.

**Дерево разбора** – древовидное представление порождения строки языка из стартового символа грамматики. Такое дерево содержит корень (стартовый символ), листовые узлы (терминалы или пустые строки), а также внутренние узлы (нетерминалы). Если внутренний узел A имеет дочерние узлы X1,2,3,…, то должна существовать продукция A 🡪 X1 X2 X3, … .

**Семантическое правило в контексте дерева разбора** – узел, с описанием действия, которое надо выполнить при его посещении. Схема трансляции должна гарантировать, что семантические действия будут выполняться в том же порядке, в котором они встречаются при обходе дерева разбора в обратном порядке. Данные семантическое правила назначаются продукции для вычисления значения атрибутов.

**Нисходящий анализ** – метод разбора дерева разбора, действия в котором происходят от корня до листьев дерева. При многократных повторениях действий выбора одной из продукций и построения дочерних узлов из её правой части и построения узлов для следующего найденного узла осуществляется построение дерева разбора.

**Анализ методом рекурсивного спуска** – представляет собой способ нисходящего анализа, при котором для обработки входной строки используется большое количество рекурсивных процедур. Предиктивный анализ основан на информации о первых символах, которые могут быть сгенерированы телом продукции. FIRST(a) – множество терминалов, которые могут появиться в качестве первого символа одной или нескольких строк. Такой вид анализа не требует отката, в отличие от обычного рекурсивного анализа.

**Левая рекурсия** – ситуация зацикливания, которая может возникнуть в случае анализа методом рекурсивного спуска. Такая проблема возникает при леворекурсивных продукциях наподобие expr 🡪 expr + term. Так как сканируемый символ изменяется только при встрече терминала в теле продукции, между рекурсивными вызовами не происходит никаких изменений. Для устранения таких рекурсий существует способ введения нового нетерминала.

A black arrows pointing to different letters

Description automatically generated with medium confidence

**АСТ** – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены с операторами языка, а листья, с соответствующими операндами. АСТ представляет собой более высокоуровневое представление программа, которое выделяет основные структурные элементы и их отношения.

**Таблица символов** – структура данных, которая используется компилятором для хранения информации о конструкциях исходной программы. Информация накапливается на этапе анализа. Записи в этой таблице представляют собой информацию об идентификаторах. Для согласованности областей видимости используется метод цепочек таблиц символов, более вложенная указывает на более общую. При этом обращение к идентификатору происходит с помощью правила поиска ближайшего по вложенности.

**Лексический анализ** – первая фаза компилятора, на которой происходит чтение входных символов исходной программы, их группирования в лексемы и вывод последовательностей токенов. Такой поток токенов пересылается синтаксическому анализатору. Также, лексический анализатор при встрече идентификатора, заносит его в таблицу символов.

**Регулярные выражения** – средство, позволяющая компилятору на этапе семантического анализа выявлять различные программные структуры, такие как идентификаторы, числа, ключевые слова и многое другое.

**Диаграммы переходов** – средство, которое содержит ряд узлов или кружков, именуемых состояниями, между ними также формируются отношения – дуги. Дуги представляют собой направленные линии из одного состояния в другое. Диаграмма переходов необходима для точного определения, какие состояния далее доступны из текущего символа.

**Лексический анализатор на основе диаграммы переходов:**

Существует несколько способов построения:

* Последовательное испытание диаграммы переходов для каждого токена.
* Параллельная работа с различными диаграммами перехода.
* Предпочтительный подход: все диаграммы переходов обьединяются в одну.

**Конечные автоматы** – средство, позволяющее для входной строки точно ответить на вопрос да или нет. Существует 2 вида конечных автоматов – ДКА и НКА. НКА не имеет ограничений на дуги. В свою очередь ДКА в своих состояниях имеет только одну выходящую дугу. Как ДКА, так и НКА способны разбирать регулярные языки.

**НКА** – граф переходов, который содержит: множество состояний, входные символы, функции переходов, начальное состояние, конечные состояния.

A diagram of a diagram

Description automatically generated

**ДКА** – частный случай НКА, который не содержит переходов для входа E. Для каждого состояния имеется только одна выходящая дуга. При построении лексического анализатора необходим именно ДКА, так что необходимо преобразование НКА в ДКА.

**Переход от НКА в ДКА** – необходимо построение таблицы переходов Dtran для D. Каждое состояние D – множество состояний НКА. Dtran строится так, чтобы параллельно моделировать все возможные переходы, которые N может выполнить для данной входной строки.

**Моделирование НКА** – на входе: входная строка с некоторым начальным состоянием, принимающими состояниями, и функцией перехода move. На выходе – ответ да или нет для строки на входе. При считывании очередного входного символа строки, для него вычисляются сначала всевозможные состояния, а затем, замыкания.

**Построение НКА** – синтаксически управляемый алгоритм, рекурсивно работающий с деревом разбора регулярного выражения. Для каждого подвыражения в регулярном выражении строится НКА с единственным принимающим состоянием.

**Синтаксический анализ** – фаза компилятора, на которой происходит получение токенов от лексического анализатора, и проверяется, может ли эта строка имен токенов порождаться грамматикой исходного языка. Также на этом этапе происходит обработка (вывод) ошибок. Концептуально, при корректности программы, синтаксический анализатор строит синтаксическое дерево.

**Методы разбора** – методы разбора синтаксического дерева. Основные типы: универсальные (Кока-Янгера-Касами, Эрли) – они могут работать с любой грамматикой, но неэффективны, нисходящие и восходящие. Предиктивный метод разбора работает только с LL-грамматиками. В то время как синтаксические анализаторы для большего класса LR-грамматик работают с помощью автоматизированных инструментов.

**Неоднозначная грамматика** – грамматика, которая даёт больше одного дерева разбора. Существуют правила устранения неоднозначности, отбрасывающие нежелательные деревья разбора.

**Левая факторизация** – преобразование грамматики в пригодную для предиктивного, или нисходящего синтаксического анализа. Является методом разрешения неоднозначностей грамматики.

**A blue arrow pointing to the right

Description automatically generated**

**Нисходящий синтаксический анализ** – построение дерева для входной строки, начиная с корня до листьев. Классической вариацией данного анализа является анализ методом рекурсивного спуска без предиктивности. В отличие от предиктивного типа анализа, классический вариант использует метод отката, так как не пользуется предпросмотром символов входной строки, он использует только предпросмотр одного символа. Класс грамматик, который можно разобрать таким способом, называется LL(1)-грамматика. Существуют функции, которые используются как для нисходящих, так и для восходящих грамматик – FIRST, FOLLOW – они позволяют выбрать применяемую продукцию на основании очередного символа входного потока.

**LL1-грамматики** – класс грамматик, для которых может быть построен анализатор, использующий метод рекурсивного спуска без возврата. Первая L – сканирование потока слева-направо. Вторая L – получение левого порождения. 1 – количество символов для предпросмотра. Такая грамматика не может содержать ни левой рекурсии, ни неоднозначности.

**Нерекурсивный предиктивный анализ с таблицей:** явно используется стек и таблица синтаксического анализа, имитируется левое порождение. Синт. анализатор рассматривает символ на вершине стека Х и текущий входной символ а, далее, если Х является нетерминалом, синтакс. анализатор выбирает Х-продукцию в соотв. с записью таблицы синтаксического анализа. Если же Х является терминалом В – проверяется соответствие между терминалом Х и текущим входным символов а.

**Восходящий синтаксический анализ** – анализ, при котором обход дерева разбора происходит начиная с листьев и заканчивая в корне. Обычно используется стандартный алгоритм анализа типа перенос/свёртка. Такой алгоритм действий использует стек. В стек помещаются символы до тех пор, пока содержимое стека не будет свёрнуто в какую-либо строку грамматики. Перенос – перенос очередного входного символа на вершину стека.

**Свертка строки к стартовому символу продукции** – на каждом этапе свёртки определяется подстрока, соответствующая телу продукции, заменяется нетерминалом из заголовка этой продукции. Ключевыми решениями во веремя восходящего анализа являются действия определения, когда применять свёртку и к какой продукции. Свертка представляет собой шаг, обратный порождению.

**Обрезка основ.** Основа –подстрока,которая соответствует телу продукции и свертка которой представляет один шаг правого порождения в обратном порядке. Обрезка основ – алгоритм, который помогает поддерживать информацию о том, какие символы уже считаны из входной цепочки, какие ожидаются, а также, какие были уже пройдены.

**A black and white sheet of paper with text

Description automatically generated**

**Конфликты в процессе ПС-анализа –** ситуации, возникающие при анализе в некоторых КС грамматиках. При работе с такой грамматикой такой анализ просто не может принять решение о дальнейших действиях.

**Простой LR-анализатор** – наиболее распространённый тип восходящих синтаксических анализаторов. L – сканирование входного потока слева-направо. R – построение правого порождения. k – количество препросматриваемых символов входного потока. LR-грамматики могут описывать намного больше языков, чем LL. Основной недостаток LR-грамматики – построение такого анализатора намного сложнее вручную и требует большого обьема работы. Для решения такой задачи используются такие инструменты, как Bison/Yacc.

**Алгоритм LR-анализа**: анализатор читает по одному символу предпросмотра, далее, функция считывает значение из считанных данных состояния и терминала, а затем, происходит выбор: перенос, свёртка, принятие или ошибка.

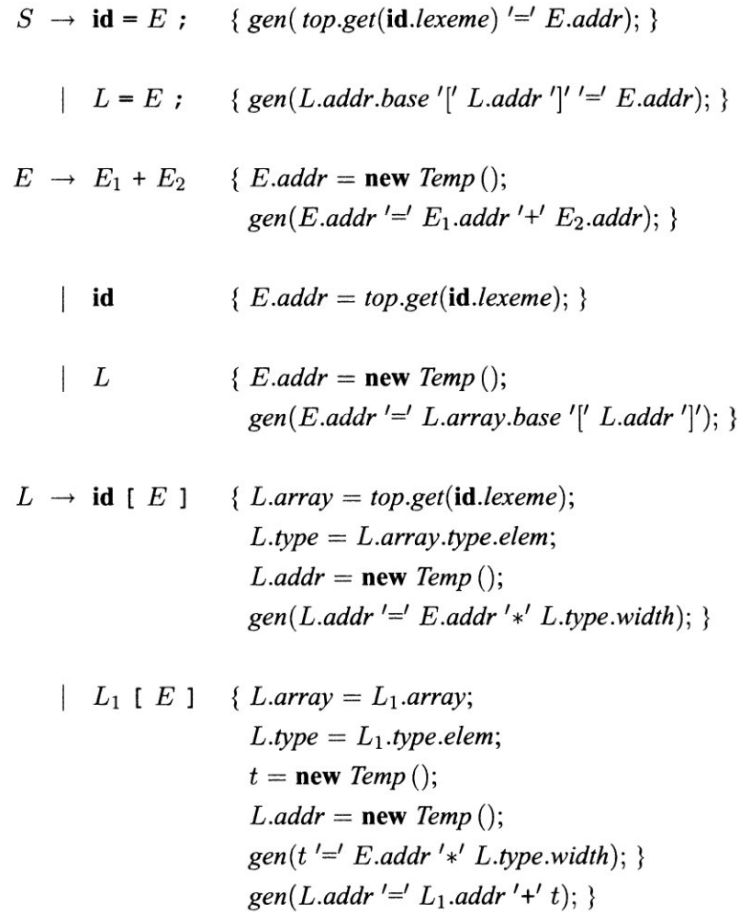
**Промежуточное представление** – архитектура набора команд АВМ, в который легко транслировать синтаксическое дерево, над которым легко выполняются оптимизации, а также генерировать код для целевой архитектуры.

**Трёхадресный код** – в каждой команде 3 операнда. Является линеаризованным представлением синтаксического дерева или ориентированного графа.

**SSA** (static single-assignment) – форма промежуточного представления в виде статических единственных присваиваний, упрощает некоторые формы оптимизации. В отличие от трёхадресного кода, работает принцип единственного присваивания, то есть, каждой переменной соответствует единственная операция присваивания.

**Трансляция выражений в трёхадресный код** – происходит инкрементно. Сначала транслируется правая часть по переменным, а затем, присваивается к левой части выражения. Инкрементная трансляция. Используется функция gen(), которая конструирует трёхадресную команду и добавляет её к последовательности уже сгенерированных команд.

**Трансляция обращений к массиву**. Нетерминал L имеет 3 синтезируемых атрибута: L.addr – временная переменная для вычесления смещения для обращения к массиву, L.array – указатель на запись таблицы символов для имени массива, L.type – тип подмассива.

 A diagram of a mathematical equation

Description automatically generated

**Вычисление булевых выражений по сокращенной схеме** – булевы операторы &&, ||, ! транслируются в условные переходы. То есть, сами операторы в коде отсутствуют. Также, если семантикой языка предусмотрена ситуация, где без разбора всего булева выражения известен результат – дальнейшие проверки прекращаются (побочный эффект от трансляции в условные переходы).

**Трансляция if-then-else** – (S 🡪 if (B) S1) если B истинно, управление переходит к первой команде S1.code, а если B ложно, управление переходит непосредственно к команде за S1.code.

**Трансляция цикла while** – представляет собой связность условия и повторных переходов по метке. В метке begin содержится условие для следующей итерации цикла, а переход при его выполнении осуществляется на метку, следующую после.