**Структура компилятора**

**Компилятор** – программа, которая для заданной программы, написанной на одном языке строит эквивалентный код на другом, целевом языке (исходный код в исполняемый).

Существует гибрид компилятора и интерпретатора (JAVA)

Лексический анализатор разбивает текст на отдельные слова. Слова называются лексемами. Лексический анализатор должен классифицировать эти лексемы.

Модуль текста программы - те свойства программы, которые можно выяснить, не выполняя программу (например, объявлена ли данная переменная; определена ли вызываемая функция). Этот модуль также выполняет проверку типов. На выходе получаем, корректная или некорректная программа. В результате проверки типов получаем атрибутированное синтаксическое дерево.

Генератор промежуточного кода строит эквивалентную программу на некотором универсальном машинном языке.

**Лексический анализ**

Формальный язык – множество последовательностей символов.

L = {“x”,”=”,”+”,”1”,”0”}

С языком обязательно связан алфавит (любое конечное множество символов).

∑ - алфавит.

Из символов мы можем строить строки, то есть последовательности символов. Формально, строка – кортеж из символов. Если строка имеет ноль символов, то она называется пустой и обозначается ε. Основная операция над строками – конкатенация. Для конкатенации пустая строка является нейтральный элементом.

**Операции над языками**

1) Объединение L и M

L U M = {s| s € L v S € M}

… (дальше в тетради)

Используя рассмотренные операции можно строить выражение, которые определяют регулярные языки.

**Регулярные выражение**

Позволяют языки определять алгебраически. Регулярные выражения определяются по индукции, то есть строятся из элементарных выражений, и дальше с помощью операций, которые позволяют из заданных языков получать более сложные языки.

Базовые случаи (дальше в тетради)

Общие случаи (дальше в тетради)

**Lex** – генератор лексических анализаторов

С помощью этой системы можно задать лексемы регулярными выражениями

**Распознаватели для регулярных языков**

Наиболее эффективным распознавателем для регулярных языков является конечный автомат. Теоретиками было доказано, что множества языков, определяемых регулярными выражениями и конечным автоматом совпадают.

Строка допускается конечным автоматом, если она составляет путь из начального состояния в некоторое финальное состояние.

Существует алгоритм, который позволяет для каждого регулярного выражения построить эквивалентный недетерминированный конечный автомат (НКА).

Идею предложил Кеннет Томпсон.

Этот алгоритм определяется рекурсивной функцией, которая действует по структуре выражения и строит результирующий конечный автомат.

Эпсилон состояние отображает множество состояний на множество состояний

**Множество достижимых состояний**

Определим все множества состояний, в которых может находиться автомат в данный момент. Для этого нам понадобится определить расширенную функцию переходов. Такая функция будет описывать переходы из множества состояний на множество состояний.

*- расширенная функция перехода*

*Базовый случай*

=

*Общий случай*

=…

**(дальше в тетради)**

**Преобразование недетерминированного конечного автомата в детерминированный**

Мы заранее строим все множества состояний, в которых может находиться автомат в данный момент. Недетерминированный автомат взять раньше по лекции. Цифрами обозначаются состояния.

**Алгоритм преобразования недетерминированного конечного автомата в детерминированный**

1) Входные данные: недетерминированный конечный автомат (N = (S, ∑, δ, S0, F)).

2) Выходные данные: ДКА (D – Dstates (множество состояний), Dtrans (множество переходов))

3) Метод

s – Состояние автомата N;

T – Множество состояний из N;

ε-closure(s)

ε-closure(T)

Move (T, a)

Состояние, для которого переходы уже найдены, будем считать помеченным.

**Алгоритм**

1) Вычисляем ε-closure(s0) и заносит это множество в Dstates, как непомеченное

2) Пока в Dstates есть хотя бы одно непомеченное состояние, выполнить:

а) помечаем Т;

б) Для каждого символа а, который принадлежит ∑ выполнить:

- U= ε-closure (Move (T, a))

- Если U не принадлежит Dstates, то заносим это состояние в Dstates непомеченным

-заносим переход Dtrans [T, a] := U

Состояние ε-closure(s0) является начальным

Состояние детерминированного конечного автомата будет финальным, если в него входит хотя бы одно состояние НКА.

**Минимизация числа состояний ДКА**

Было доказано, что для каждого ДКА существует единственный эквивалентный ДКА с минимальным числом состояний. Единственный с точностью до нумерации состояний.

Существуют два основных метода минимизации ДКА.

Первый метод основан на объединении эквивалентных состояний. Второй подход – разбиение множества состояний автомата на классы эквивалентности.

Два состояния эквиваленты тогда и только тогда, когда выполняются вместе два условия:

1) Условие подобия: оба состояния должны быть финальными или не финальными.

2) Условие преемственности: для всех входных символов эти состояния должны иметь переходы в эквивалентные состояния.

**(схемы в тетради)**

Во втором методе используется утверждение о неразличимости состояний.

Два состояния S1 и S2 называются неразличимыми,

если для всех W ((Δ(S1, w) принадлежит F ⸧ Δ(s2,w) принадлежит F)

если для всех W Δ(S1, w) не принадлежит F ⸧ Δ(s2,w) не принадлежит F)

Если существует такая строка W, которая принадлежит ∑^\*

Δ(s1,w) принадлежит F и Δ(s2,w) не принадлежит F или

Δ(s1,w) не принадлежит F и Δ(s2,w) принадлежит F,

то состояния различимы

Все состояния автомата разбиваются на финальные и не финальные

{0, 1, 2, 3, 4} – не финальные (1 группа)

{5} – финальные (2 группа)

Множества будем разбивать на части:

Символ «с»

{0, 2, 4} переход во 2 группу

{1, 3} переход в себя

**Структура лексического анализатора**

Входной буфер – исходный текст программы (массив символов)

В таблице действий указывается, как для каждого класса лексем сформировать лексический знак.

В таблице переходов нужно заставить работать вместе несколько распознавателей.

Каждому регулярному выражению соответствует свой ДКА

Можно взять одно состояние в качестве стартового и с помощью ε переходы получить автомат с N финальными состояниями. Этот подход имеет ряд недостатков:

А) языки, распознаваемые отдельными конечными автоматами могут пересекаться. Решением может выступать определение приоритетов для классов лексем (зарезервированные слова имеют приоритет для идентификаторов).

Б) если есть фрагмент строки «if12», то ее можно разобрать двумя способами.

1 вариант – идентификатор

2 вариант – зарезервированное слово и литерал

При втором подходе используется N ДКА, работающих параллельно. Используется один указатель на текущий символ. Прочитав входной символ, автомат переходит в новое состояние. Если какой-либо автомат перейдет в финальное состояние, и он будет задан приоритетным по отношению к другим автомат, перешедшим в финальное состояние, то сохраняется в глобальной переменной номер этого автомата и текущая позиция. Когда все автоматы потерпят неудачу, разбор лексемы завершается. Результат определяется по последнему сохраненному номеру автомата.

**Синтаксический анализ**

Синтаксический анализатор строится на основе контекста свободных грамматик (КСГ). КСГ свободны описать более широкий языков, чем регулярные выражения. Они свободны различать рекурсивные вложенные структуры. Формально КСГ – четверка объектов: (N,T,S,P).

N – Множество нетерминалов

T – Множество терминалов (алфавит)

S – Стартовый нетерминал

P – Множество продукций (пар объектов)

P подмножество N\*(N U T)^(\*)

(1+2)\*3 <- (expr+2)\*3 <- (expr+expr)\*3 <- expr\*3 <- expr\*expr <- expr

Expr -> litera

Expr -> expr+expr

Expr -> expr\*expr

Expr -> (expr)

Слово Expr обозначает то, что должно сформироваться в процессе синтаксического разбора.

Expr – нетерминал

Lit,+,\*,(,) – терминал

Каждое из 4 определений называется продукцией

a,b,c,d – используется для обозначения терминалов

A,B,C,D – обозначения нетерминалов

U,V,W,X,Y,Z – переменные, которые обозначают грамматический символ (терминал или нетерминал)

u,v,w,x,y,z – последовательности терминалов

альфа, бета, гамма, дельта – последовательности грамматических символов

Строка: альфа А бета => альфа гамма бета, если продукция А->гамма принадлежит p.

Альфа => \*бета, если его можно получить за 0 или более шагов

Альфа => Lбета из альфа можно получить бета за один шаг заменой самого левого нетерминала

Альфа => Rбета

Последовательность, которая порождается из стартового нетерминала называется сентенциальной формой.

Порождения последовательности, принадлежащей языку можно представить в виде дерева. **(дальше в тетради)**

Грамматика, которая позволяет для одной и той же входной последовательности построить два разных дерева разбора называется неоднозначной.

Недостаток неоднозначной грамматики: не до конца определена структура предложения.

Контекстно-свободная грамматики должен определять структуру предложения.

S->E

E->E+T

E->T

T->T\*Lit

T->Lit

**Дерево этого в тетради**

Упрощенное дерево называется абстрактным синтаксическим деревом.

Контекстно-свободно грамматика позволяет выявлять структуру предложения.

**Контекстно-свободные грамматики для регулярных языков**

Языки, которые можно распознавать контекстно-свободными грамматиками называется контекстно-свободными.

Класс контекстно-свободных языков включает в себя и регулярные языки.

Пусть регулярный язык задан конечным автоматом **(схема в тетради)**.

В качестве нетерминалов грамматики мы возьмем состояние автомата: А1, А2, А3, А4.

Алфавит автомата будет совпадать с множеством терминалов. Если в автомате есть переход из первого состояния во второе по символу а, то этот факт можно обозначить продукцией А1->aA2.

А1->bА3

Если состояние финальное, то для него нужно добавить эпсилон-продукцию.

Легко убедиться, что автомат и грамматика будут распознавать один и тот же язык.

**Методы грамматического разбора**

Все методы грамматического разбора разделяются на виды:

- сверху вниз (нисходящий)

- снизу вверх (восходящий)

**Нисходящий разбор**

Начинаем разбор со стартового нетерминала (S). Если он определяется несколькими продукциями, стоит вопрос выбора.

При нисходящем разборе используется левый вывод. Для нисходящего разбора можно использовать метод рекурсивного спуска.

**Недетерминированный метод рекурсивного спуска**

Для каждого нетерминала мы должны написать свою функцию разбора.

А-> альфа

А-> бета

А-> гамма

Функция должна проверить, порождает ли заданную строку альфа, бета или гамма

Пусть альфа = Х1, Х2…Хn

Если Xi – нетерминал, то проверка выполняется вызовом функции для этого нетерминала. Если Хi – терминал, то нужно проверить наличие этого терминала во входном буфере.

Этот метод неэффективен. Метод рекурсивного спуска не работает с леворекурсивными грамматиками.

Е->Е+Т – рекурсия стоит слева

Грамматика является леворекурсивной, если в ней есть нетерминал, который за один или более шагов порождает цепочку, начинающуюся с этого нетерминала.

Леворекурсивную грамматику можно преобразовать к эквивалентной грамматике, но без левой рекурсии.

Устранение левой рекурсии

Е->Е+Т – непосредственная левая рекурсия

**Е->Аα**

**А->Еβ**

**Косвенная левая рекурсия**

Рассмотри продукции

А->Аα

А->β

{β,βα,βαα…}

Полученный язык можно представить, используя язык регулярных выражений.

βα\*

А->βА’

A’->αA’

A’->ε

Если из грамматики устранили левую рекурсию, то алгоритм по методу рекурсивного спуска не уйдет в бесконечный цикл.

Имея две продукции:

А->α

A->β

Если будет простой тест, позволяющий выбрать правую часть, то получится алгоритм с линейной сложностью.

А->аγ

А->bγ

Здесь определить можно.

Грамматика, для которой можно сделать выбор между двумя продукциями по текущему символу буфера называется LL(1) грамматикой.

L – входной буфер рассматривается слева направо.

L – используется левый вывод

1 – достаточно прочитать один входной символ из входного буфера, чтобы выбрать продукцию для следующего шага разбора.

Существуют LL(k) грамматики.

**Задача – какую грамматику применить**

А->α

A->β

Если во входном буфере символ А, с которого может начинаться последовательность, выводимая из альфа, то применяем продукцию альфа.

Если А, стоит в начале последовательности, выводимой из бета, то применяем вторую продукцию.

Для такого определения будем использовать множество first

First(α) – множество символов, с которых может начинаться последовательность α.

Альфа принадлежит First(α), если порождает последовательность, которая начинается с бета.

Если альфа за 0 или более шагов дает эпсилон, то в эпсилон принадлежит First(α).

Для каждого нетерминала А можно определить множество FOLLOW(A), которое будет содержать те терминалы, которые могут встретиться вслед за А в какой-либо сентенциальной форме.

Пользуясь этими множествами, можно сформулировать условие, что грамматика входит в класс LL(1):

**Для всех пар продукций вида А->α | β выполняется условие FIRST(α), при условии что FIRST(α) /\ FOLLOW(β) = пустое множество**

Если эпсилон принадлежит FIRST(α), то эту продукцию мы будем выбирать на основе множества FOLLOW(А) при пересечении с FIRST(β) дает пустое множество.

**Левая факторизация**

S->if BE then S

| if BE then S else S

| other

if BE then S – общая часть

else S – вариативная часть

V->else S | ε

S->if BE then S V other

Общий вид операции факторизации:

А->αβ1 | αβ2 |…| αβn | γ

A-> αB | γ

B -> β1 | β2 |…| βn

BE -> BT \/ BE | BT

BT -> BF /\ BT | BF

BF -> E == E | E<E |…| (BE)

BE -> BT BE’

BE’ -> \/ BE | ε

**Таблично-управляемый LL(1) разбор**

Структура таблично-управляемого синтаксического анализатора (схема в тетради)

**Алгоритм детерминированного разбора**

Алгоритм начинает работу, когда в стеке находится стартовый нетерминал.

Push $

Push (S)

Token := nexttoken

X := Вершина стека

Repeat

If X-терминал

If X == token then

Pop X

Token := nexttoken

Else error

If X-нетерминал

If X->y1,y2,..,yk = M[X,token]

{ pop(X);

For (i=k;i>=0;i--)

Push(Yi)

Else error

Until X=$

If token <> $ eror

Ошибка: получен ошибочный символ токен, а ожидаются следующие символы.

Функция FIRST

FIRST (α)

1) FIRST (a) = {a}

2) FIRST (A) = { ε } если ест ьпродукция А-> ε

3) FIRST (X), если есть продукция {->Y1 Y2…Yk

Терминал а принадлежит FIRST(X), если а принадлежит FIRST(Yi) и ε принадлежит FIRST(Yj) для всех J от 1 до i-1

Множество FOLLOW

Это множество может встретиться после терминал A

Множество строится в цикле до тех пор, пока в эти множества можно что-то добавить.

1) Помечаем признак конца файла ($) в FOLLOW(S)

2) Цикл

а) Если есть продукция вида А->α, B β, то заносим в FOLLOW(В) FIRST(β) без символа ε

б) если есть продукция вида А->α B, то заносим в FOLLOW(B) множество FOLLOW(A)

в) если есть продукция вида А->α, B β и β может исчезнуть, то в FOLLOW(B) заносим FOLLOW(A)

Выполнение таблицы разбора для LL(1) грамматики

Для всех продукций А=α выполнить:

1) Для всех терминалов а, принадлежащих FIRST(α) заносим нашу продукцию А ->α в ячейку M[A,a]

2) Если ε принадлежит FIRST(α), то заносим продукцию А ->α в ячейку M[A,b] для всех b, принадлежащих FOLLOW(A)

3) Если ε принадлежит FIRST(α) и конец файла принадлежит FOLLOW(A), то заносим А ->α в ячейку M[A,$]

В каждую оставшуюся свободную ячейку заносим признак ошибки

1. Введение – написать, как важно изучать языки программирования и методы трансляции
2. Лексический анализ – определить, какие лексические единицы понадобятся. Регулярные выражения. Описание реализации.
3. Синтаксический анализ. Описание грамматики. Реализация синтаксического анализатора. Удаление левой факторизации и т.п. Описание того, как строили: привести описание работы программы.
4. Абстрактное синтаксическое дерево. Определение типа дерева. Синтаксически управляемое определение. Описать реализацию, как строится дерево.
5. Заключение. Формулировка результатов.

Объем страниц 25, но не меньше 20.

Нужно использовать язык С.

**Восходящий обзор**

E->E+E | E\*E | (E) | id

В восходящем разборе используется правый вывод

x+y\*z

E->E+E->E+E\*E->E+E\*id3 -> E+id2\*id3->id1+id2\*id3

Процесс нисходящего разбора соответствует правому выводу, записанному в обратном порядке.

Разбор снизу-вверх можно представить как обрезку дерева

**Метод сдвиг-свертка**

Синтаксический анализатор по этому методу для своей работы использует стек. Для работы также необходим буфер.

Стек Буфер Операция

$ id1+id2\*id3$ Сдвиг

$id1 +id2\*id3$ Свертка E->id

$E +id2\*id3$ Сдвиг

$E + id2\*id3$ Сдвиг

$E +id2 \*id3$ Свертка E->id

$E +E \*id3$ Сдвиг

$E +E\* id3$ Сдвиг

$E +E\* id3 $ Свертка E->id

$E +E\* E $ Свертка E->E\*E

$E +E $ Свертка E->E+E

$E $

Последняя конфигурация указывает на то, что разбор можно завершить

…называется продукция А->β и позиция, где β находится в γ, которую можно заменить на А, то получится право сентенциальная форма в выходе γ.

Основа всегда формируется в вершине стека, если используется метод сдвиг-свертка. Основу в стеке можно распознавать конечным автоматом.

**Структура LR-анализатора (схема в тетради)**

Таблица action определяет для каждого состояния и терминала действие, которое должен выполнить синтаксический анализатор. Этих действий может быть 4 вида:

1) Shift S

2) «Свертка» А->β

3) «Принимается»

4) ошибка

Таблица goto содержит переходы по нетерминалам

**Алгоритм LR-разбора**

S0 – стартовое состояние

Алгоритм разбора является циклом, который можно повторять бесконечно.

Пусть Sm – состояние в вершине стека, ai – текущий символ.

Если таблица action с индексами Sm ai стоит Shift S делаем:

Push(ai)

Push(S)

Читаем следующий символ

Если action с индексами Sm ai равно свертка, то

{пусть r – длина последовательности β, делаем pop 2\*r раз}

Находим новое состояние S=goto[Sm-r, A]

Push(A)

Push(S)

Если action (Sm,ai] = accept (“Принимается”)

Запускается завершение

Если action (Sm,ai] = accept (“Error”)

Запускается обработчик ошибочной ситуации

**Построение автомата для распознавания основ в стеке**

S-> XYZ

S-> .XYZ (. – процесс ожидания)

S-> X.YZ

S-> XY.Z

S-> XYZ.

…

Грамматика, у которой стартовый нетерминал не встречается в правой части никакой продукции называется пополненной.

Атрибутные грамматики или синтаксическое управляемое определение.

Грамматические символы могут иметь атрибуты.