«Анатомия» составных частей компилятора

ТЕРМИНОЛОГИЯ

- ***Транслятор*** *- это программа, которая переводит исходную программу в эквивалентную ей объектную программу. Если объектный язык представляет собой автокод или некоторый машинный язык, то транслятор называется компилятором.*
- ***Автокод*** *очень близок к машинному языку; большинство команд автокода - точное символическое представление команд машины.*
- ***Ассемблер*** *- это программа, которая переводит исходную программу, написанную на автокоде или на языке ассемблера (что, суть, одно и то же), в объектный (исполняемый) код.*
- ***Интерпретатор*** *принимает исходную программу как входную информацию и выполняет ее. Интерпретатор не порождает объектный код. Обычно интерпретатор сначала анализирует исходную программу (как компилятор) и транслирует ее в некоторое внутреннее представление. Далее интерпретируется (выполняется) это внутреннее представление.*

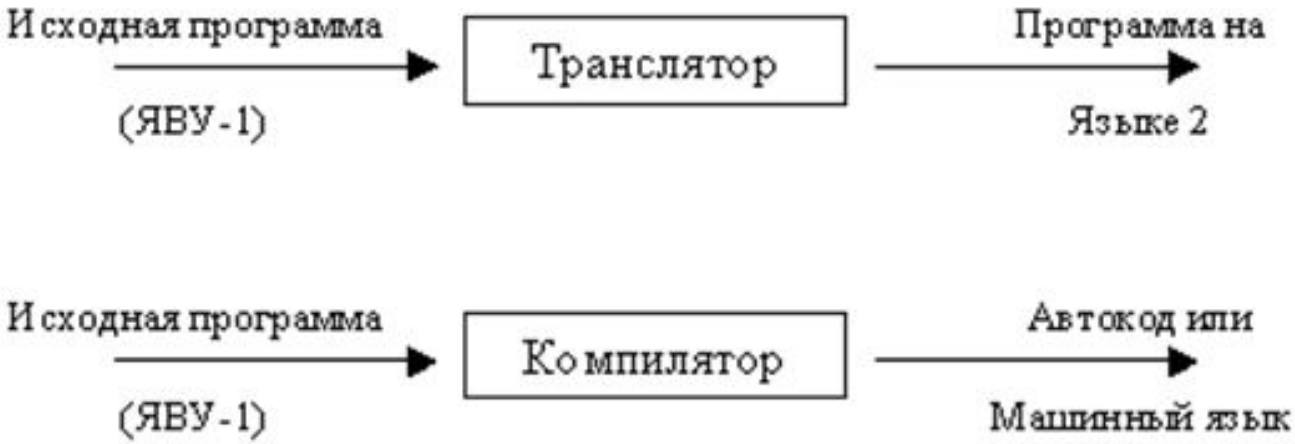
Препроцессор

- **Препроцессор** — это компьютерная программа — это компьютерная программа, принимающая данные на входе и выдающая данные, предназначенные для входа другой программы (например, компилятора). О данных на выходе препроцессора говорят, что они находятся в **препроцессированной** форме, пригодной для обработки последующими программами (компилятор). Результат и вид обработки зависят от вида препроцессора; так, некоторые препроцессоры могут только выполнить простую текстовую подстановку, другие способны по возможностям сравниться с языками программирования. Наиболее частый случай использования препроцессора — обработка исходного кода форме, пригодной для обработки последующими программами (компилятор). Результат и вид обработки зависят от вида препроцессора; так, некоторые препроцессоры могут только выполнить простую текстовую подстановку, другие способны по возможностям сравниться с языками программирования. Наиболее частый случай использования препроцессора — обработка исходного кода перед передачей его на следующий шаг компиляции. Языки программирования форме, пригодной

«Границы моего языка есть границы моего мира»

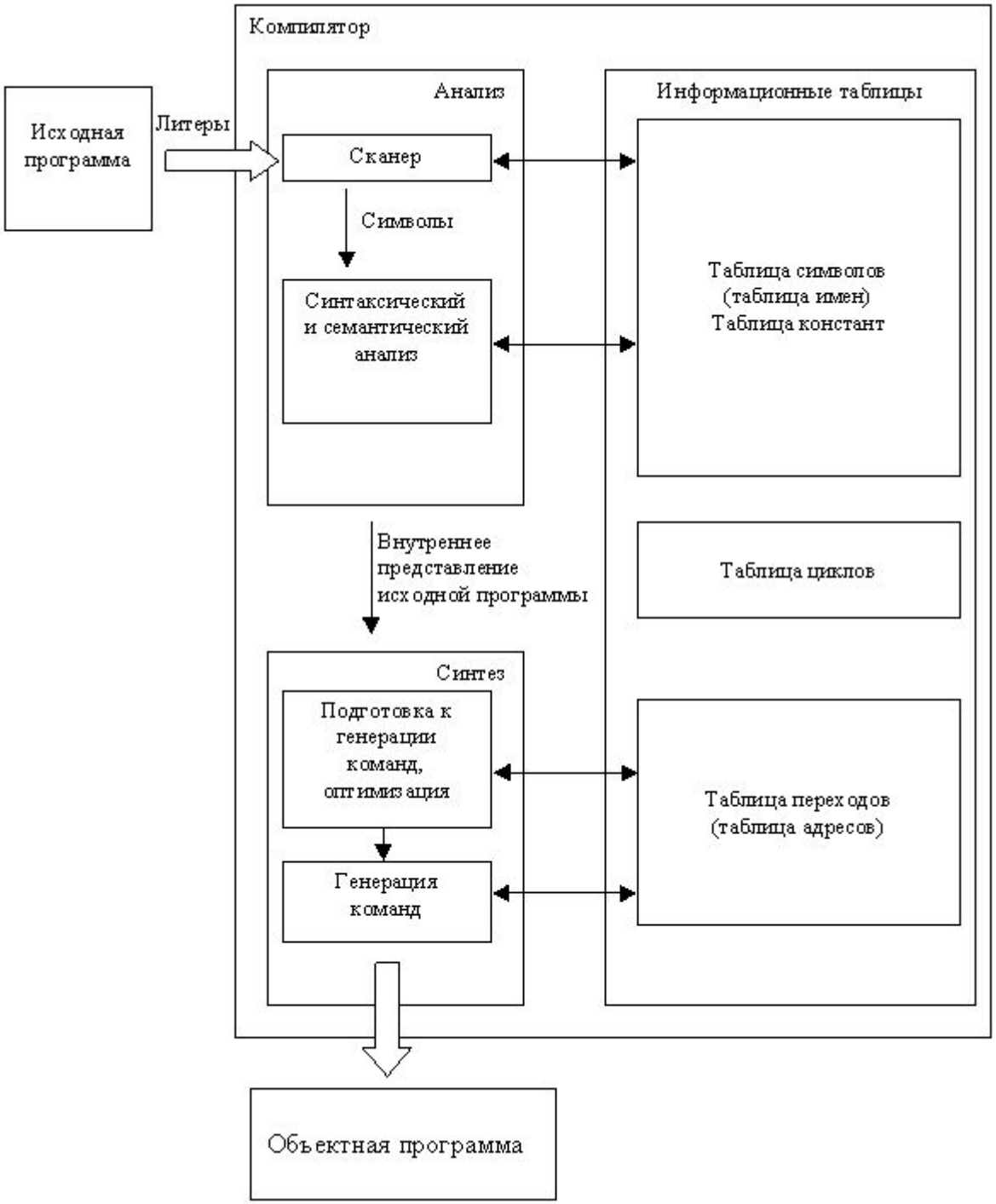
Л.Витгенштейн

Условно это можно изобразить следующим образом:



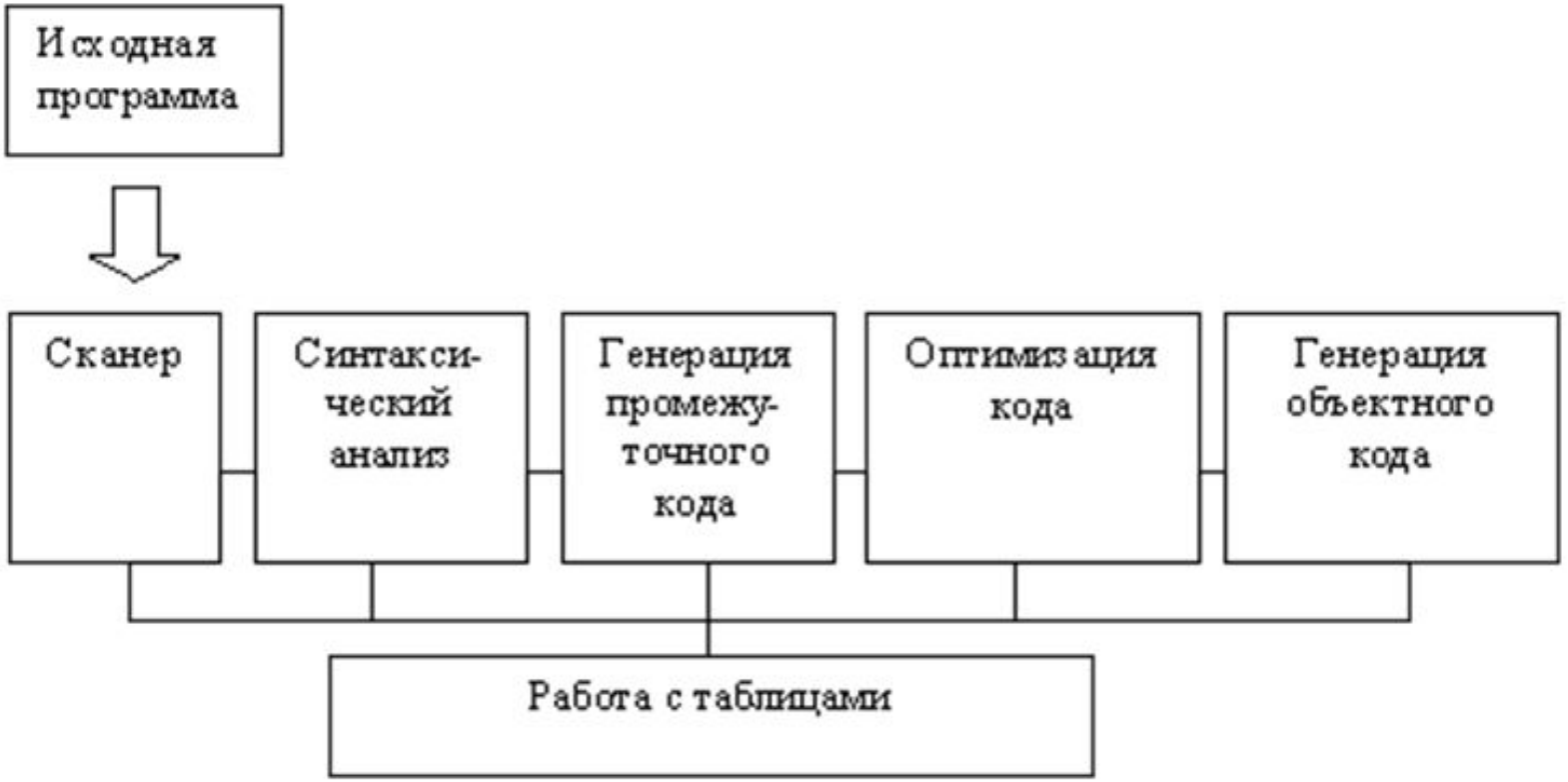
Компилятор компиляторов

- Компилятор компиляторов (КК) – система, позволяющая генерировать компиляторы; на входе системы - множество грамматик, а на выходе, в идеальном случае, - программа. Иногда под КК понимают язык программирования, в котором исходная программа - это описание компилятора некоторого языка, а объектная программа - сам компилятор для этого языка. Исходная программа КК - это просто формализм, служащий для описания компиляторов, содержащий, явно или неявно, описание лексического и синтаксического анализаторов, генератора кодов и других частей создаваемого компилятора.



Классическая структура компилятора

На всех этих этапах происходит работа с различного рода таблицами



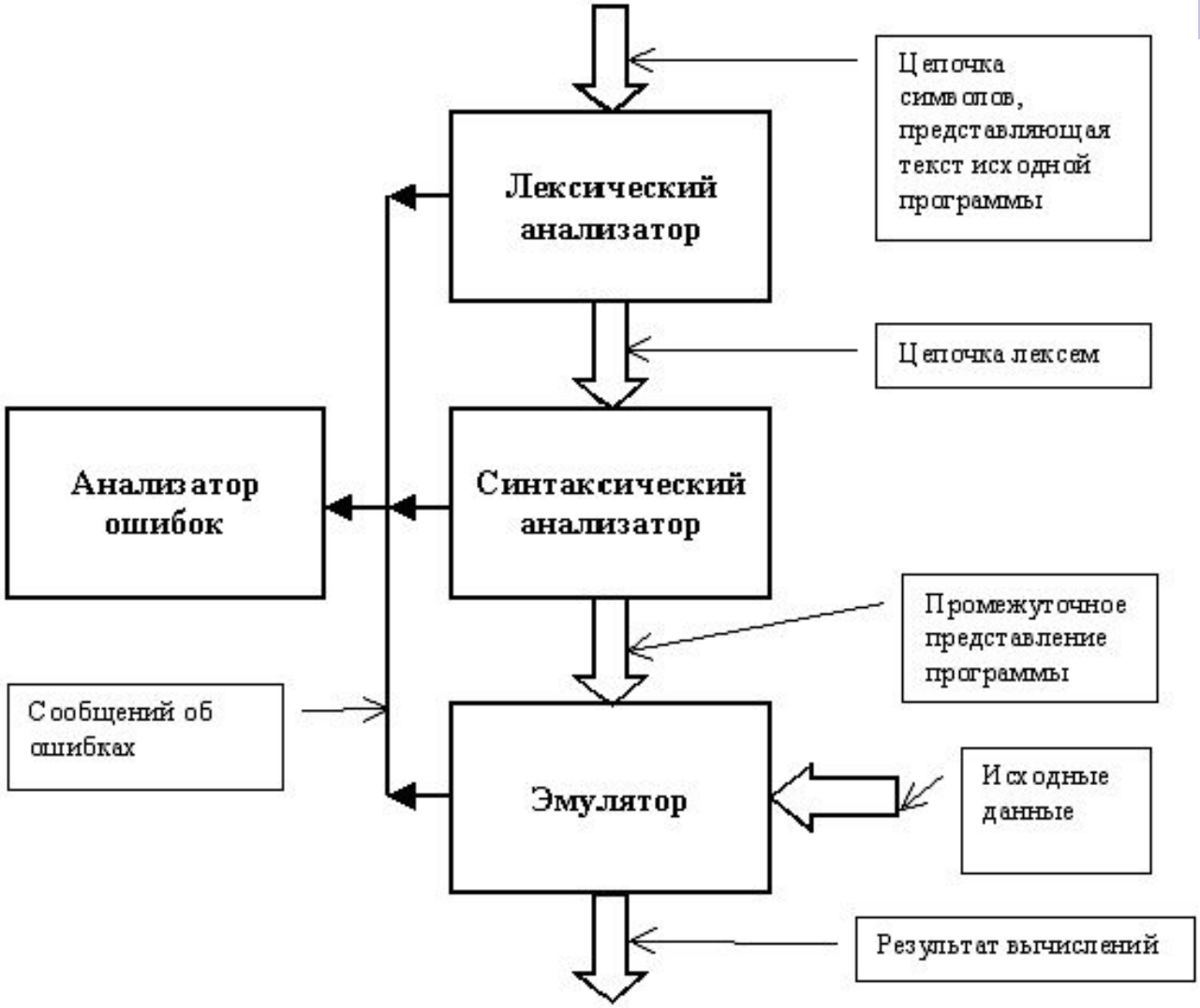


Рис. Обобщенная структура интерпретатора.

Синтаксис. Семантика. Синтаксический анализатор.

- **Синтаксис** - *совокупность правил некоторого языка, определяющих формирование его элементов.* Иначе говоря, это *совокупность правил образования семантически значимых последовательностей символов в данном языке.*
- **Семантика** - *правила и условия, определяющие соотношения между элементами языка и их смысловыми значениями, а также интерпретацию содержательного значения синтаксических конструкций языка.*
- **Синтаксический анализатор** - *компонента компилятора, осуществляющая проверку исходных операторов на соответствие синтаксическим правилам и семантике данного языка программирования.*

Лексический анализ (сканер)

- На входе сканера - цепочка символов некоторого алфавита (именно так выглядит для сканера наша исходная программа). При этом некоторые комбинации символов рассматриваются сканером как единые объекты. Лексический анализатор (ЛА) группирует определенные терминальные символы (т.е. входные символы) в единые синтаксические объекты - *лексемы*. В простейшем случае лексема - это пара вида <тип_лексемы, значение>.

Как будет интерпретироваться такая входная последовательность "567AB" ?

Прямой ЛА и непрямой ЛА

- Прямой ЛА определяет лексему, расположенную непосредственно справа от текущего указателя, и сдвигает указатель вправо от части текста, образующей лексему (ПЛА определяет тип лексемы, которая образована символами справа от указателя).
- Непрямой ЛА определяет, образуют ли знаки, расположенные непосредственно справа от указателя, лексему этого типа. Если да, то указатель передвигается вправо от части текста, образующей лексему. Иными словами, для него *заранее* задается тип лексемы, и он распознает символы справа от указателя и проверяет,

DO5I=1,10

- Фортран – это классический пример языка, использующего непрямой лексический анализатор. Все дело в том, что в этом языке игнорируются пробелы. Рассмотрим, например, конструкцию
- DO5I=1,10 …

- Итак, на выходе сканера - внутреннее представление имен, разделителей и т. п.

Работа с таблицами

- Таблица имен представляет собой структуру, подобную следующей:

Номер элемента	Идентификатор	Дополнительная информация (тип, размер и т.п.)
1	A	идент., тип = "строка"
…	…	…
N	3.1415	константа, число

- Механизм работы с таблицами должен обеспечивать:
- быстрое добавление новых идентификаторов и сведений о них;
- быстрый поиск информации.

Синтаксический и семантический анализ

- Синтаксический анализ - это процесс, в котором исследуется цепочка лексем и устанавливается, удовлетворяет ли она структурным условиям, явно сформулированным в определении синтаксиса языка. Это – самая сложная часть компилятора

- ***Синтаксический анализатор*** *расчленяет исходную программу на составные части, формирует ее внутреннее представление, заносит информацию в таблицу символов и другие таблицы. При этом производится полный синтаксический и, по возможности, семантический контроль программы. Фактически, это - синтаксически управляемая программа. При этом обычно стремятся отделить синтаксис от семантики насколько это возможно - когда синтаксический анализатор распознает конструкцию исходного языка, он вызывает семантическую процедуру, которая контролирует эту конструкцию, заносит информацию куда надо, проверяет на дублирование описания переменных, проверяет соответствие типов и т.п.*

Синтаксический анализатор можно разбить на следующие составляющие :

- распознаватель;
- блок семантического анализа;
- объектную модель, или промежуточное представление, состоящие из таблицы имен и синтаксической структуры.

Рис. 1. Обобщенная схема синтаксического анализатора

Предложения исходной программы обычно записываются в *инфиксной* форме

Три вида записи выражений

- Существуют три вида записи выражений:
  1. **инфиксная форма**, в которой оператор расположен между операндами (например, "a + b");
  2. **постфиксная форма**, в которой оператор расположен после операндов ("a b + ");
  3. **префиксная форма**, в которой оператор расположен перед операндами ("+ a b").

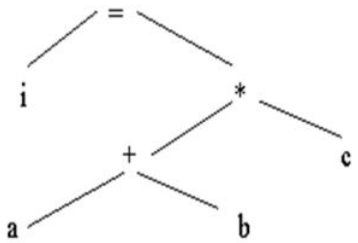
  Постфиксная и префиксная формы образуют т.н. польскую форму записи (польская нотация). Польская форма удобна, прежде всего, тем, что в ней

в честь польского математика Лукасевича

- Обычно под польской формой понимают именно постфиксную форму записи. Кроме того, используются и такие внутренние формы представления исходной программы, как *дерево* (синтаксическое) и *тетрады*.

Дерево.

- Допустим, имеется входная цепочка i=(a+b)*c. Тогда дерево будет выглядеть так:



- У каждого элемента дерева может быть только один “предок”. Дерево “читается” снизу вверх и слева направо. Дерево – это прежде всего удобная математическая абстракция.

- Тетрада- это четверка, состоящая из кода операции, приемника и двух операндов. Если требуется не два, а менее операндов, то в этом случае тетрада называется *вырожденной*

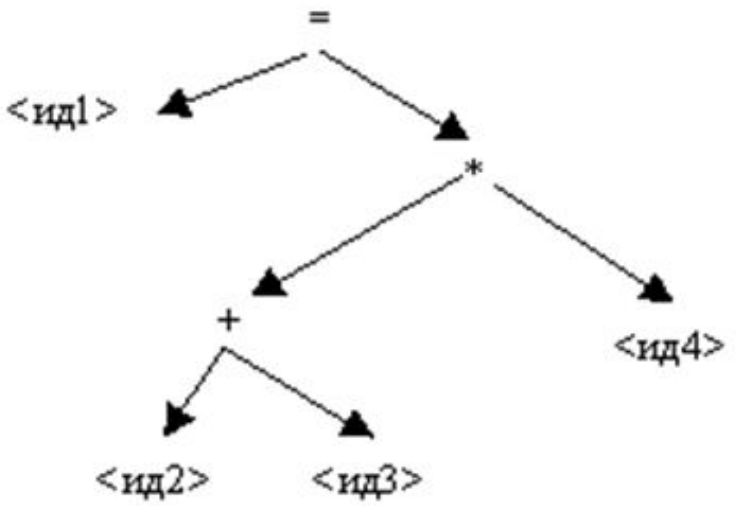
Исходное выражение	Код	Приемник	Операнд1	Операнд2
a+b→ T1	+	T1	a	b
T1∗ c→ T2	*	T2	T1	c
i=T2	=	I	T2	(вырожденная тетрада)

Пусть на вход синтаксического анализатора подаются выражения

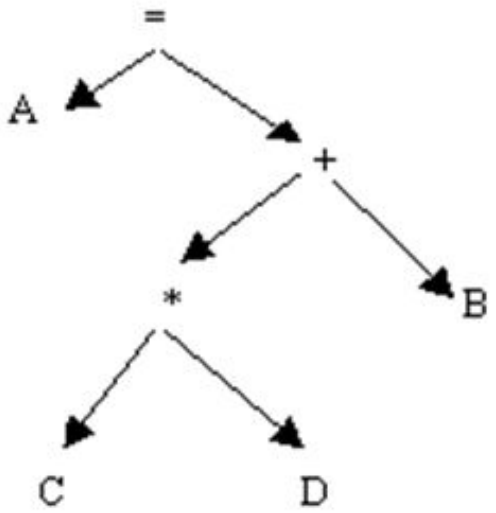
"<ид1>=(<ид2>+<ид3>)*<ид4>" и "A = B+C*D"

На выходе будем иметь:

Дерево для выражения "<ид1>=(<ид2>+<ид3>)*<ид4>"



Дерево для выражения "A = B+C*D"



Тетрады для "<ид1> = (<ид2>+<ид3>)*<ид4>"

1. +, <ид2>, <ид3>, T1
2. *, T1, <ид4>, T2
3. =, T2, <ид1>

Тетрады для "A = B+C*D"

1. *, C, D, T1
2. +, B, T1, T2
3. =, T2, A
4. (T1, T2 - временные переменные, созданные компилятором)

3

Польская форма для "<ид1> = (<ид2>+<ид3>)*<ид4>":

<ид1> <ид2> <ид3> + <ид4> * =

И еще один пример: польская форма для "A=B+C*D" будет выглядеть как "ABCD*+=".

Алгоритм вычисления польской формы записи очень прост:

- *Просматриваем последовательно символы входной цепочки. Если очередной символ является операндом (идентификатором или константой), то читаем дальше. Если символ является бинарным оператором, то извлекаем из цепочки два предыдущих операнда вместе с оператором, производим операцию и помещаем результат обратно в цепочку символов.*
- "ABCD*+=".

ТЕОРИЯ ФОРМАЛЬНЫХ ЯЗЫКОВ ГРАММАТИКИ

ФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ

- Общение на каком-либо языке – искусственном или естественном- заключается в обмене предложениями или, точнее, фразами. Фраза - это конечная последовательность слов языка. Фразы необходимо уметь строить и распознавать.
- Если существует механизм построения фраз и механизм признания их корректности и понимания (механизм распознавания), то мы говорим о существовании некоторой теории рассматриваемого языка.

Определение: Язык L - это множество цепочек конечной длины в алфавите Σ.

- Одним из наиболее эффективных способов описания языка является грамматика. Строго говоря, грамматика - это математическая система, *определяющая* язык. Грамматика пригодна не только для задания (или генерации) языка, но и для его *распознавания*.

$$L(G)=\{\omega \mid \omega \in \Sigma^{*}, S \Rightarrow^{*} \omega\}$$

- Если ∑ - множество (алфавит или словарь), то $\sum^{*}$ - замыкание множества ∑ , или, иначе, *свободный моноид*, порожденный ∑ , т.е. множество всех конечных последовательностей, составленных из элементов множества ∑ , включая и *пустую* последовательность.

Определение. *Грамматика* - это четверка G = (N, Σ ,P,S),

где

- N - конечное множество нетерминальных символов (синтаксические переменные или синтаксические категории);
- Σ - конечное множество терминальных символов (слов) (Σ ∩ N=∅);
- P - конечное подмножество множества
- $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- Элемент (α, β) множества P называется *правилом* или *продукцией* и записывается в виде $\alpha \rightarrow \beta$;
- S - выделенный символ из N ($S \in N$), называемый *начальным символом* (эта особая переменная называется также "начальной переменной" или "исходным символом").

http://www.masters.donntu.edu.ua/2006/fvti/svyezhentsev/library/article3.htm

- **http://www.chernyshov.com/SPPO_5/TRANS/Tr3-02.htm**

Нормальная форма Бэкуса-Наура

- <Буква> := A|B|C|…|Z
- <Цифра> := 0|1| … |9
- <Идент> := <Буква> {<Буква>|<Цифра>}

Грамматика целых чисел без знака:

- <число> := <цифра>|<цифра><число>
- <цифра> := 0|1|2|…|9

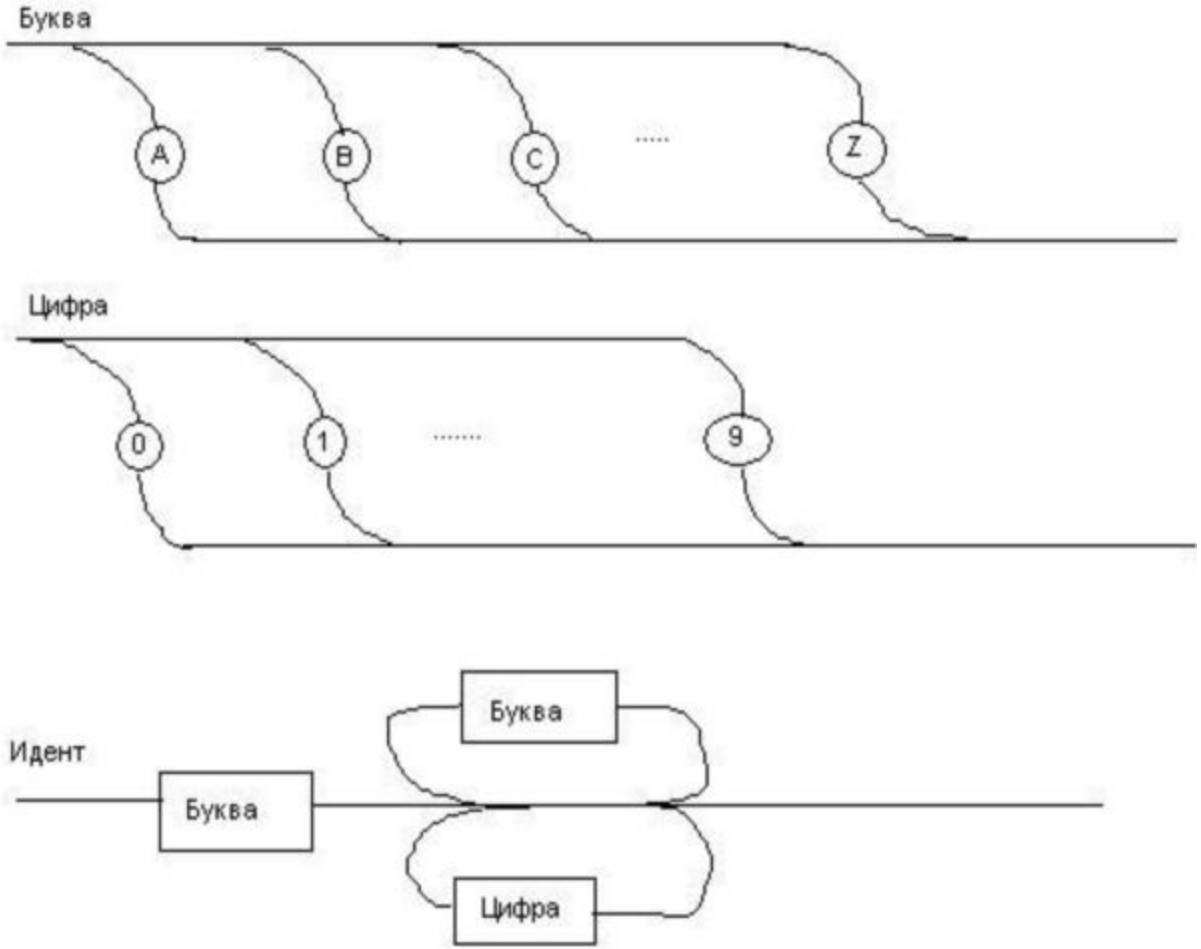
Формула с плюсами и минусами без скобок

- <формула> := <число>|<формула><знак><число>
- <знак> := +|-

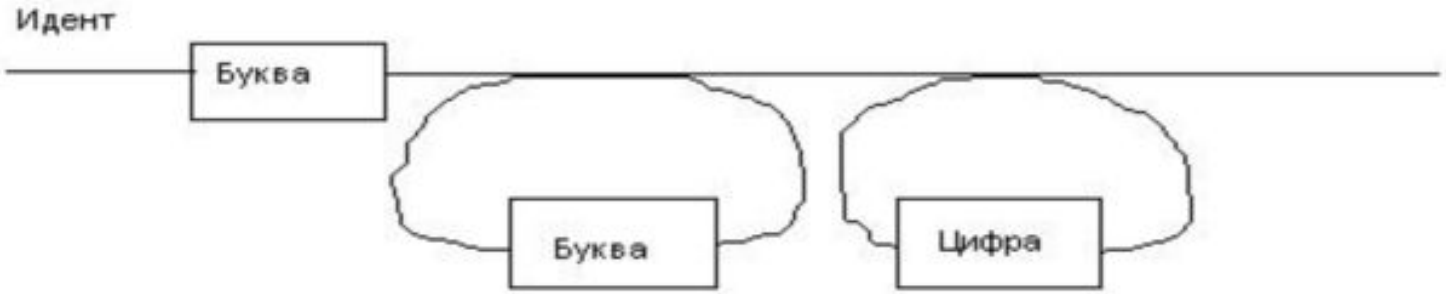
Диаграммы Вирта

<Буква> := A|B|C|…|Z
<Цифра> := 0|1| … |9
<Идент> := <Буква> |<Идент> <Буква>|<Идент> <Цифра>

Теперь – диаграмма из 3-х строк:



Отметим, что третью строку диаграммы можно нарисовать и так:



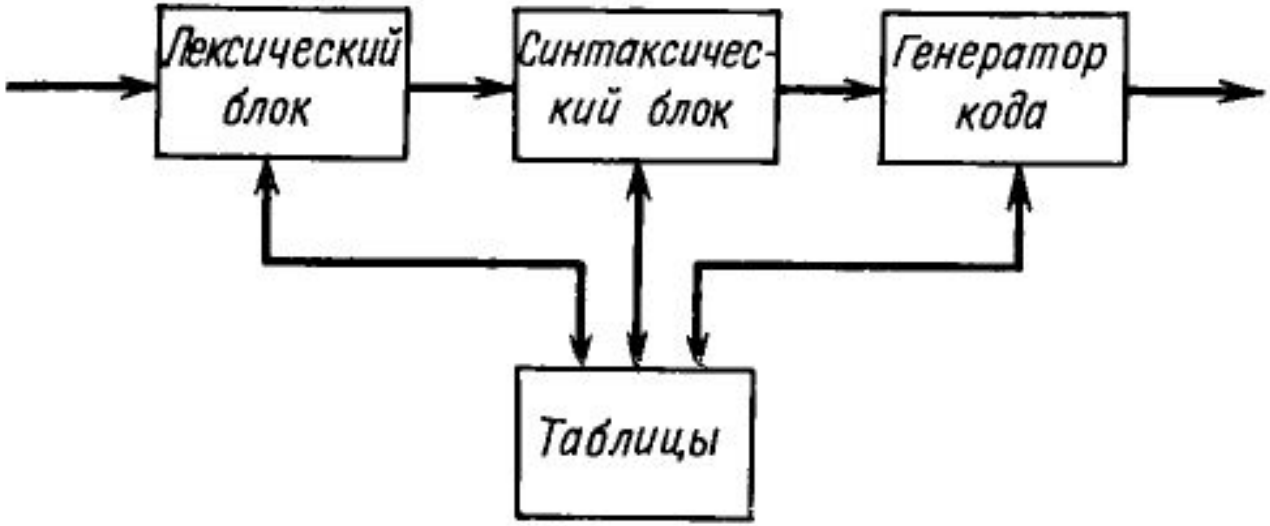
ИЕРАРХИЯ ХОМСКОГО

Тип 0 — неограниченные
Тип 1 — контекстно-зависимые
Тип 2 — контекстно-свободные
Тип 3 — регулярные

Иерархия Хомского — классификация формальных языков — классификация формальных языков и формальных грамматик — классификация формальных языков и формальных грамматик, согласно которой они делятся на 4 типа по их условной сложности. Предложена профессором Массачусетского технологического института, лингвистом *Ноамом Хомским*.

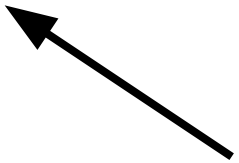
Конечные автоматы

- В основе всех процессов обработки языков лежит теория автоматов и формальных языков.



Лексический блок

IFB1=13GOTO4



Здесь 6 лексем

Каждая лексема состоит из 2-х частей: класса и значения

Синтаксический блок

- Этот блок переводит лексическую последовательность в другую последовательность.

Генератор кода

Генератор кода. Этот блок «развертывает» атомы, построенные синтаксическим блоком, в последовательность команд вычислительной машины, которые выполняют соответствующие действия.

Семантическая обработка

Та часть работы компилятора, которая связана со смыслом лексем называется *семантической обработкой*. Семантика идентификатора, например, может включать его тип, а в случае, если это массив,— его размерность. Один из видов семантической обработки включает занесение в таблицу имен свойств отдельных идентификаторов по мере их выявления. Другой вид включает действия, зависящие от типа данных.

Оптимизация

Та часть работы компилятора, которая, строго говоря, не является необходимой, но позволяет получать более эффективные объектные программы, часто называется *оптимизацией*.

Блоки и проходы компилятора

Рассмотрим, например, взаимодействие между лексическим и синтаксическим блоками. Здесь возможен выбор по крайней мере из двух типов взаимодействия.

Один тип предполагает, что каждый раз, когда лексический блок выдает лексему, управление передается синтаксическому блоку для обработки этой лексемы. Когда возникает необходимость в следующей лексеме, управление возвращается в лексический блок.

При другом типе взаимодействия лексический блок выдает всю цепочку лексем до того, как управление передается синтаксическому блоку. В этом случае говорят, что работа лексического блока образует отдельный проход.

Логика языка

Иногда сам исходный язык наводит на мысль о том, что компилятор должен иметь не менее двух проходов. Такая потребность возникает, если в какой-то момент компилятору нужна информация из еще не просмотренной части программы. Например, если описание идентификатора или переменной может появляться в тексте программы после их использования, то может случиться, что код нельзя выдать до тех пор, пока не будет частично обработана вся исходная программа. В этом случае для генерации кода требуется отдельный проход.

Оптимизация кода

Иногда объектный код получается более эффективным, если генератору кода доступна информация обо всей программе. Например, согласно некоторым методам оптимизации, нужно знать все те места программы, где используются переменные и где могут изменяться их значения. Поэтому, прежде чем начать оптимизацию, необходимо просмотреть всю программу до конца.

Экономия памяти

Обычно многопроходные компиляторы занимают в памяти меньше места, чем компиляторы с одним проходом, так как код каждого прохода может вновь использовать память, занимаемую кодом предыдущего прохода.

Каждый проход компилятора можно организовать в виде одного блока или комбинации нескольких блоков. С точки зрения теории построения компиляторов блок — это просто часть компилятора, которая мыслится и строится как одно целое.

Конечные автоматы

В литературе по теории автоматов существует несколько различных формальных определений. Общим в этих определениях является то, что они моделируют вычислительные устройства с фиксированным и конечным объемом памяти, которые читают последовательности входных символов, принадлежащих некоторому конечному множеству.

Конечные распознаватели

Конечный распознаватель — это модель устройства с конечным числом состояний, которое отличает правильно образованные или «допустимые» цепочки от недопустимых. Хотя это понятие чисто математическое, определяемое в терминах множеств, последовательностей (цепочек) и функций, лучше представлять его себе в виде вычислительной машины.

Контроль нечетности

Примером задачи распознавания может служить проверка нечетности числа единиц в произвольной цепочке, состоящей из нулей и единиц. Соответствующий конечный автомат будет «допускать» все цепочки, содержащие нечетное число единиц, и «отвергать» цепочки с четным их числом. Назовем этот автомат «контролёром нечетности».

Входной алфавит

Входной алфавит контролера нечетности состоит из двух символов: 0 и 1.

Представим себе, что в каждый момент времени конечный автомат имеет дело лишь с одним входным символом, а информацию о предыдущих символах входной цепочки сохраняет с помощью конечного множества состояний. Согласно этому представлению, автомат помнит о прочитанных ранее символах только то, что при их обработке он перешел в некоторое состояние, которое и является памятью автомата о прошлом.

Множество состояний нашего автомата содержит два состояния, которые мы будем называть ЧЕТ и НЕЧЕТ.

Переходы

При чтении очередного входного символа состояние автомата меняется, причем новое его состояние зависит только от входного символа и текущего состояния. Такое изменение состояния называется *переходом*. Может оказаться, что новое состояние совпадает со старым.

Функция переходов

$$\delta\,(s_{\text{тек}},\ x) = s_{\text{нов}}$$

Учитывая, что состояния ЧЕТ и НЕЧЕТ означают соответственно четное и нечетное число прочитанных единиц, определим функцию переходов контролера нечетности следующим образом:

$$\delta\,(\text{ЧЕТ},\ 0) = \text{ЧЕТ}$$
$$\delta\,(\text{ЧЕТ},\ 1) = \text{НЕЧЕТ}$$
$$\delta\,(\text{НЕЧЕТ},\ 0) = \text{НЕЧЕТ}$$
$$\delta\,(\text{НЕЧЕТ},\ 1) = \text{ЧЕТ}$$

Эта функция переходов отражает тот факт, что четность меняется тогда и только тогда, когда на входе читается единица.

Конечный автомат

Конечный автомат задается:

1) конечным множеством входных символов,
2) конечным множеством состояний,
3) функцией переходов δ, которая каждой паре, состоящей из входного символа и текущего состояния, приписывает некоторое новое состояние,
4) состоянием, выделенным в качестве начального, и
5) подмножеством состояний, выделенных в качестве допускающих или заключительных.

$$S_{\text{тек}} \xrightarrow{x} S_{\text{нов}}$$

Для контролера нечетности, например, можно написать

$$\text{НЕЧЕТ} \xrightarrow{1} \text{ЧЕТ}$$

Аналогичным образом можно изобразить последовательность переходов. Например, запись

$$\text{ЧЕТ} \xrightarrow{1} \text{НЕЧЕТ} \xrightarrow{1} \text{ЧЕТ} \xrightarrow{0} \text{ЧЕТ} \xrightarrow{1} \text{НЕЧЕТ}$$

показывает, как автомат в состоянии ЧЕТ применяется к цепочке 1101. Первый символ 1 меняет состояние ЧЕТ на НЕЧЕТ, так как δ(ЧЕТ, 1)=НЕЧЕТ. Следующая единица меняет НЕЧЕТ на ЧЕТ. Нуль оставляет автомат в состоянии ЧЕТ. Последняя единица изменяет состояние на НЕЧЕТ. Так как ЧЕТ — начальное, а НЕЧЕТ — допускающее состояние, цепочка 1101 допускается нашим автоматом.

1101

Регулярные множества

Иногда мы хотим говорить о множестве всех цепочек, распознаваемых некоторым конечным автоматом. Например, контролер нечетности распознает множество цепочек, состоящих из нулей и единиц и содержащих нечетное число единиц. Такие множества обычно называют «регулярными».

Регулярным множеством называется множество цепочек, которое распознается некоторым конечным распознавателем.

Таблица переходов

1. Столбцы помечены входными символами.
2. Строки помечены символами состояний.
3. Элементами таблицы являются символы новых состояний, соответствующих входным символам столбцов и состояниям строк.
4. Первая строка помечена символом начального состояния.
5. Строки, соответствующие допускающим (заключительным) состояниям, помечены справа единицами, а строки, соответствующие отвергающим состояниям, помечены справа нулями.

	0	1	
ЧЕТ	ЧЕТ	НЕЧЕТ	0
НЕЧЕТ	НЕЧЕТ	ЧЕТ	1

Таким образом таблица переходов создаёт конечный автомат:

входное множество = {0, 1},
множество состояний = {ЧЕТ, НЕЧЕТ},
переходы δ (ЧЕТ, 0) = ЧЕТ и т. д.,
начальное состояние = ЧЕТ,
допускающие состояния = {НЕЧЕТ}.

Ещё один автомат

	x	y	z	
1	1	3	4	1
2	2	1	3	0
3	2	4	4	1
4	3	3	3	0

входное множество $= \{x, y, z\}$

множество состояний $= \{1, 2, 3, 4\}$

переходы $\delta(1, x) = 1, \delta(1, y) = 3$ и т.д.

начальное состояние $= 1$

допускающие состояния $= \{1, 3\}$

Входная цепочка *xyzz* допускается этим автоматом, так как

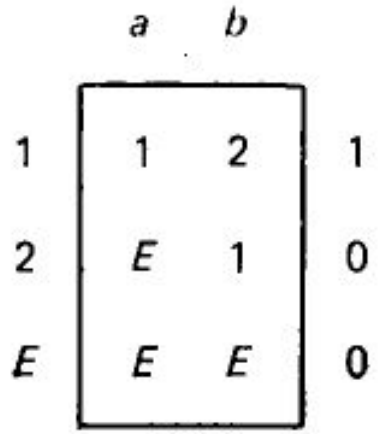
$$1 \xrightarrow{x} 1 \xrightarrow{y} 3 \xrightarrow{z} 4 \xrightarrow{z} 3$$

и 3 является допускающим состоянием, тогда как цепочка *zyx* отвергается, потому что

$$1 \xrightarrow{z} 4 \xrightarrow{y} 3 \xrightarrow{x} 2$$

Концевые маркеры и выходы из распознавания

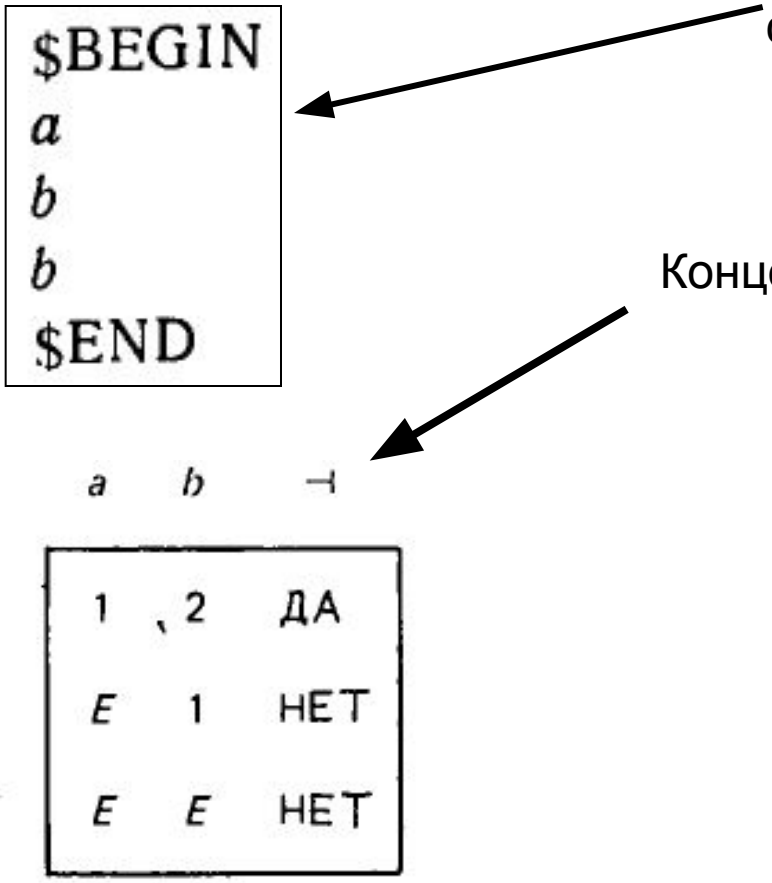
Конечный распознаватель лежит в основе процессов распознавания цепочек в компиляторе. Один из способов использования такого распознавателя — поставить его под контроль некоторой управляющей программы, которая определяет момент, когда входная цепочка прочитана, и по состоянию распознавателя выясняет, допустима она или нет.



	a	b	
1	1	2	1
2	*E*	1	0
E	*E*	*E*	0

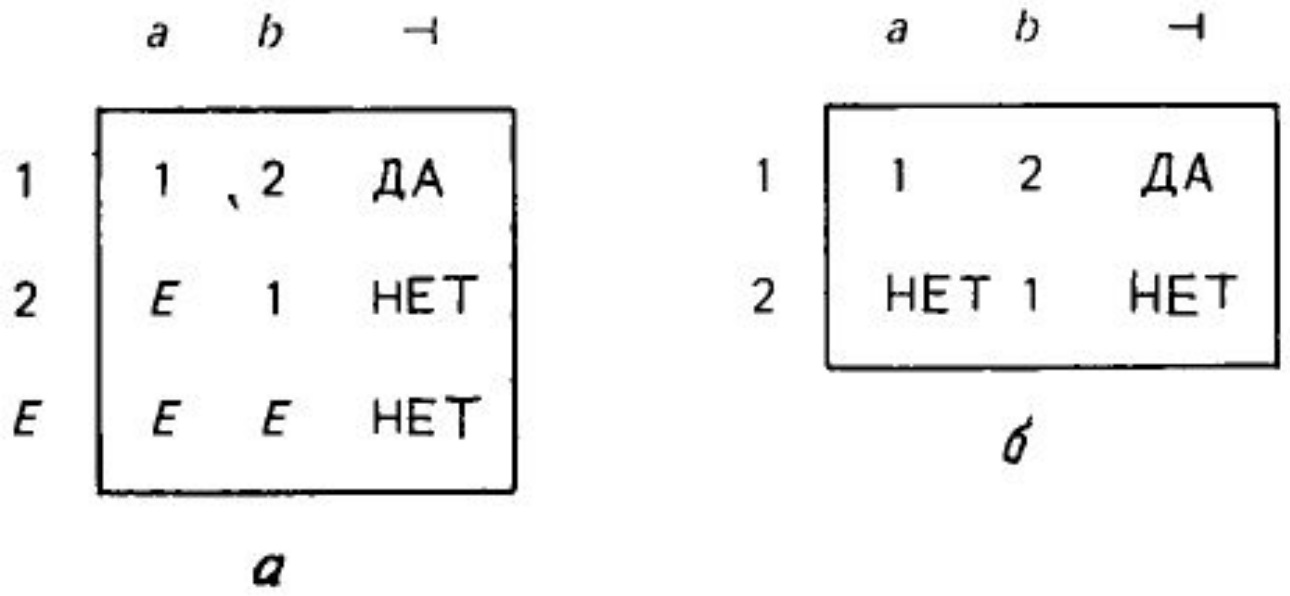
Автомат допускает множество цепочек в алфавите $\{a, b\}$, таких, что символы b в них либо не встречаются, либо встречаются парами. Например, этот автомат допускает цепочки *abb*, *abba*, *aaa*, *abbbbabb*, *bba*, но отвергает *baa*, *abbb* и *abbab*. Состояние 1 «помнит», что обработанная часть цепочки допустима. Если после допустимой части цепочки следует символ b, автомат переходит в состояние 2. В состояние E он переходит, когда входная цепочка окончательно испорчена вхождением символа a вслед за «неспаренным» b. Таким образом, состояние E можно назвать «состоянием ошибки», которое запоминает, что обнаружена ошибка.

Получение обрабатывающего автомата из распознающего



а Любая техника построения конечных распознавателей потенциально применима и при построении обрабатывающих автоматов (процессоров).

Если решили прервать процесс…



Мы допускаем теперь, что любой элемент таблицы переходов конечного процессора может быть не переходом, а выходом из распознавания. Этот аспект процесса обработки входных цепочек назовем *обнаружением* (или *детекцией*). Автомат *обнаруживает* некоторую ситуацию до того, как прочитана вся цепочка, и прекращает свою работу.

языка транслятора (работа по курсу «Теория вычислительных

Процесс трансляции с алгоритмического языка можно условно разбить на три этапа:
лексический анализ, грамматический разбор и генерацию машинного кода.
Рассмотрим задачу построения лексического анализатора входного текста транслятора.

Лексический анализ

- Под лексическим анализом понимают процесс предварительной обработки исходной программы, на котором основные лексические единицы программы - лексемы: ключевые слова, идентификаторы, метки, константы приводятся к единому формату и заменяются условными кодами или ссылками на соответствующие таблицы, а комментарии исключаются из текста программы.
  Результатом лексического анализа является список лексем-дескрипторов и таблицы. В таблицах хранятся значения выделенных в программе лексем.

Дескриптор

- Дескриптор- это пара вида: (. < указатель>), где - это, как правило, числовой код класса лексемы, который означает, что лексема принадлежит одному из конечного множества классов слов, выделенных в языке программирования;
- - это может быть либо начальный адрес области основной памяти, в которой хранится адрес этой лексемы, либо число, адресующее элемент таблицы, в которой хранится значение этой лексемы.

- После проведения успешной идентификации лексемы формируется её образ - дескриптор, он помещается в выходные данные лексического анализатора. В случае неуспешной идентификации формируется сообщение об ошибках в написании слов программы.
- В ходе лексического анализа могут выполняться и другие виды лексического контроля, в частности, проверяться парность скобок и других парных символов, наличие метки у оператора, следующего за GOTO и т.д.
- Результаты работы сканера передаются в последствии на вход синтаксического анлизатора. Имеется две возможности их связи: раздельная связь и нераздельная связь.

- PROGRAM PRIMER;
- VAR X,Y,Z : REAL;
- BEGIN
- X:=5;
- Y:=6;
- Z:=X+Y;
- END;

Применим следующие коды для типов лексем:

- К1- ключевое слово;
- К2- разделитель;
- К3- идентификатор;
- К4- константа.

Лексический анализ можно производить, если нам задан алфавит, список ключевых слов языка и служебных символов.
Пусть всё это имеется. Тогда внутренние таблицы сканера примут следующий вид.

Таблица 4. Ключевые слова.

№	Ключевое слово
1	PROGRAM
2	BEGIN
3	END
4	FOR
5	REAL
6	VAR

Таблица 5. Разделители.

№	Разделители
1	;
2	,
3	+
4	-
5	/
6	*
7	:
8	=
9	.

Таблица 6. Идентификаторы.

№	Идентификаторы
1	PRIMER
2	X
3	Y
4	Z

Таблица 7. Константы.

№	Знач. констант
1	5
2	6

На основании составленных таблиц можно записать входной текст через введённые дескрипторы (дескрипторный текст):

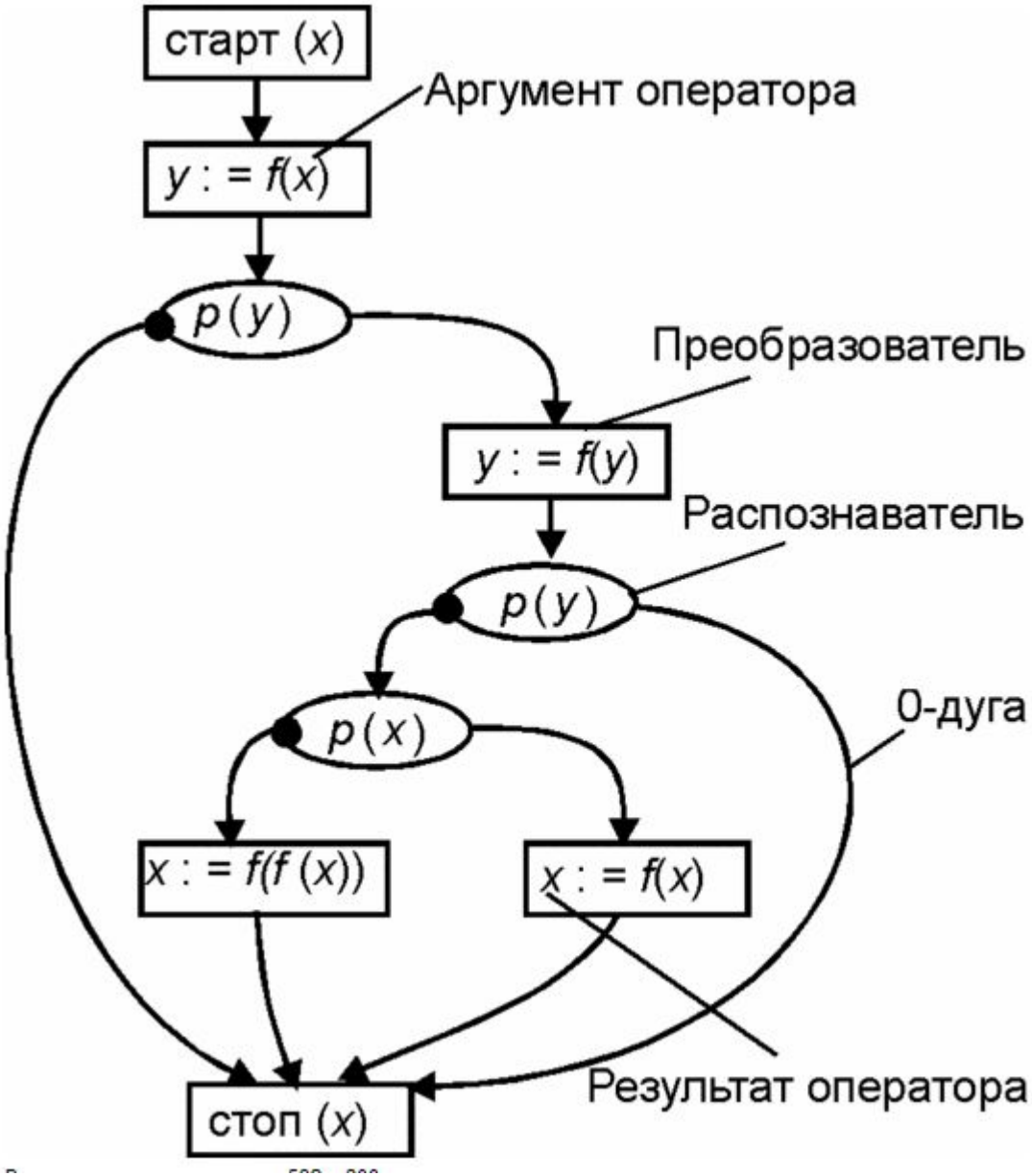
- (К1, 1) (К3, 1) (К2, 1)
- (К1, 6) (К3, 2) (К2, 2) (k3, 3) (К2, 2) (К3, 4) (К2, 7) (К1, 5) (К2, 1)
- (К1, 2)
- (К3, 2) (К2, 7) (К2, 8) (К4, 1) (К2, 1)
- (К3, 3) (К2, 7) (К2, 8) (К4, 2) (К2, 1)
- (К3, 4) (К2, 7) (К2, 8) (К3, 2) (К2, 3) (К3, 3) (К2, 1)
- (К1, 3) (К2, 9).

Схемы программ

- **Схема программ**— математическая модель программ, в которой такие понятия, как оператор, операнд, переменная, выполнение и т.д., являются обобщением соответствующих понятий существующих языков программирования. Понятие схемы программы принадлежит советскому математику А.А. Ляпунову, которое он ввел в 1953г., исходя из общей концепции необходимости и возможности формализации процесса программирования. В настоящее время теория схем программ — это широко разветвленная область исследования, которая имеет многочисленные выходы в практику программирования и содержит фундаментальные результаты не только по операторным методам программирования, определившим современное состояние автоматизации программирования, но и по признаваемым перспективными методам рекурсивного и параллельного программирования.

Стандартные схемы

- **Стандартные схемы** — схемы алголоподобных программ, исследование которых составляет основное содержание общей теории схем программ.
- Стандартные схемы учитывают разбиение памяти на переменные и позволяют исследовать более широкий класс преобразований программ, включающий уже и такие преобразования, как, например, экономия общих подвыражений.
- Стандартные схемы запрещают структурность операторов и переменных и, таким образом, моделируют лишь узкий класс реальных программ. Каждый оператор в такой схеме является либо *преобразователем* — оператором, изменяющим состояние памяти, либо *распознавателем* — оператором, осуществляющим выбор для исполнения одного из нескольких своих преемников. Преобразователь содержит одно обязательное присваивание и имеет одну исходящую дугу, а распознаватель не содержит присваиваний и имеет две исходящие дуги, одна из которых называется *плюс-стрелкой* (или *1-дугой*), а вторая — *минус-стрелкой* (или *0-дугой*).


старт (x)
Аргумент оператора
y : = f(x)
p(y)
Преобразователь
y : = f(y)
Распознаватель
p(y)
p(x)
0-дуга
x : = f(f (x))
x : = f(x)
Результат оператора
стоп (x)

Доказательство правильности программ

Принцип математической индукции

Принцип простой индукции

Обычно математическую индукцию вводят как метод доказательства утверждений о положительных числах.

Пусть $S(n)$ — некоторое высказывание о целом числе n и требуется доказать, что $S(n)$ справедливо для всех положительных чисел n.

Согласно простой математической индукции, для этого необходимо

1. Доказать, что справедливо высказывание $S(1)$.
2. Доказать, что если для любого положительного числа n справедливо высказывание $S(n)$, то справедливо и высказывание $S(n+1)$.

Принцип модифицированной простой индукции

- Иногда необходимо доказать, что высказывание $S(n)$ справедливо для всех целых $n \geq n_0$. Для этого можно довольно легко модифицировать принцип простой индукции. Чтобы доказать, что высказывание $S(n)$ справедливо для всех целых n, необходимо:
- **1.** Доказать, что справедливо $S(n_0)$.
- **2.** Доказать, что если $S(n)$ справедливо для всех целых $n \geq n_0$, то справедливо и $S(n+1)$.

ПРИМЕР. Для любого неотрицательного числа *n* доказать, что

$2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1.$

- Используя простую индукцию, мы должны
- **1**. Доказать, что $2^0 = 2^{0+1} - 1$. Это очевидно, так как
- $2^0 = 1 = 2^{0+1} - 1 = 2^1 - 1 = 2 - 1 = 1.$
- **2.** Доказать, что если для всех неотрицательных целых *n* справедлива формула
- $2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1,$
- то справедлива и формула
- $2^0 + 2^1 + 2^2 + \ldots + 2^n + 2^{n+1} = 2^{(n+1)+1} - 1.$
- Высказывание $2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$ называется гипотезой индукции. Второе положение доказывается следующим образом:
- $2^0 + 2^1 + 2^2 + \ldots + 2^n + 2^{n+1} = (2^0 + 2^1 + 2^2 + \ldots + 2^n) + 2^{n+1} =$
- $= (2^{n+1} - 1) + 2^{n+1} = (2^{n+1} + 2^{n+1}) - 1 = 2 \cdot 2^{n+1} - 1 = 2^{n+2} - 1 = 2^{(n+1)+1} - 1.$
- (По гипотезе индукции)
- Что и требовалось доказать.
- Поскольку мы доказали справедливость двух утверждений, то по индукции формула
- $2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1.$
- считается справедливой для любого неотрицательного числа *n.*

Иногда нужно доказать справедливость высказывания $S(n)$ для целых n, удовлетворяющих условию $n_0 \le n \le m_0$. Так как между n_0 и m_0 находится конечное число целых чисел, то справедливость $S(n)$ можно доказать простым перебором всех возможных вариантов. Однако легче, а иногда и необходимо (если, например, мы не знаем конкретных значений n_0 и m_0) доказать $S(n)$ по индукции. В этом случае можно воспользоваться одним из двух вариантов индукции

- *Простая нисходящая индукция*
- **1.** Доказать, что справедливо $S(n_0)$
- **2.** Доказать, что если справедливо $S(n)$, то для любых целых $n_0 \le n \le m_0 - 1$ справедливо и $S(n + 1)$.
-
- *Простая восходящая индукция*
- **1.** Доказать, что справедливо $S(m_0)$.
- **2.** Доказать, что если справедливо $S(n)$, то для любых целых $n_0 + 1 \le n \le m_0$ справедливо и $S(n - 1)$.
- Интуитивно понятно, что этого достаточно для доказательства справедливости $S(n)$ при любых n, удовлетворяющих условию $n_0 \le n \le m_0$.

ДОКАЗАТЕЛЬСТВО ПРАВИЛЬНОСТИ БЛОК-СХЕМ ПРОГРАММ

- Если мы хотим доказать, что некоторая программа правильна или верна, то прежде всего должны уметь описать то, что по предположению делает эта программа. Назовем такое описание *высказыванием о правильности* или просто *утверждением* и свяжем его с точкой на блок-схеме, предшествующей блоку останова STOP. В этом случае доказательство правильности состоит из доказательства того, что программа, будучи запущенной, в конце концов окончится (выполнится оператор STOP), и после окончания программы будет справедливо утверждение о правильности. Часто требуется, чтобы программа работала правильно только при определенных значениях входных данных. В этом случае доказательство правильности состоит из доказательства того, что если программа выполняется при соответствующих входных данных, то она когда-либо закончится, и после этого будет справедливо утверждение о правильности.

Высказывание о входных данных

- *Утверждение, относящееся к данным* (или *высказывание о входных данных*), описывающее ограничение на входные данные программы, приписывается точке блок-схемы сразу после начала (блок START) или ввода исходных данных.

Инвариант цикла

Инвариант цикла - это предикат, который выполняется на каждой итерации цикла. Вот, например, возьмем такой цикл:

Код:

```
q = 0;
while (a >= b)
{
  a = a - b;
  q++;
}
```

Инвариантом будет $P(a, b, q) \equiv (a + qb = a_0)$ (a_0 - начальное значение переменной a)

Инварианты используются для доказательства того, что программа действительно делает то, что надо. В нашем случае поскольку инвариант выполняется на каждой итерации цикла, то после выхода из цикла будет $a + qb = a_0$ (инвариант) и $a < b$ (условие выхода из цикла), т.е. q - неполное частное и a - остаток от деления a_0 на b. До полного доказательства не хватает утверждения о том, что цикл обязательно завершится. В нашем случае если $a_0 > 0$, $b > 0$, то цикл завершится, так как на каждой итерации значение a уменьшается.

Инвариант цикла

- Под инвариантом цикла будем понимать утверждение, связывающее переменные, изменяющиеся внутри тела цикла, которое принимает значение *истинно* при входе в цикл, при каждой итерации цикла и после окончания цикла. Данный факт отображается в логическом комментарии, который называется *комментарием инвариантного отношения* и приписывается к точке начала цикла.

Конечность цикла

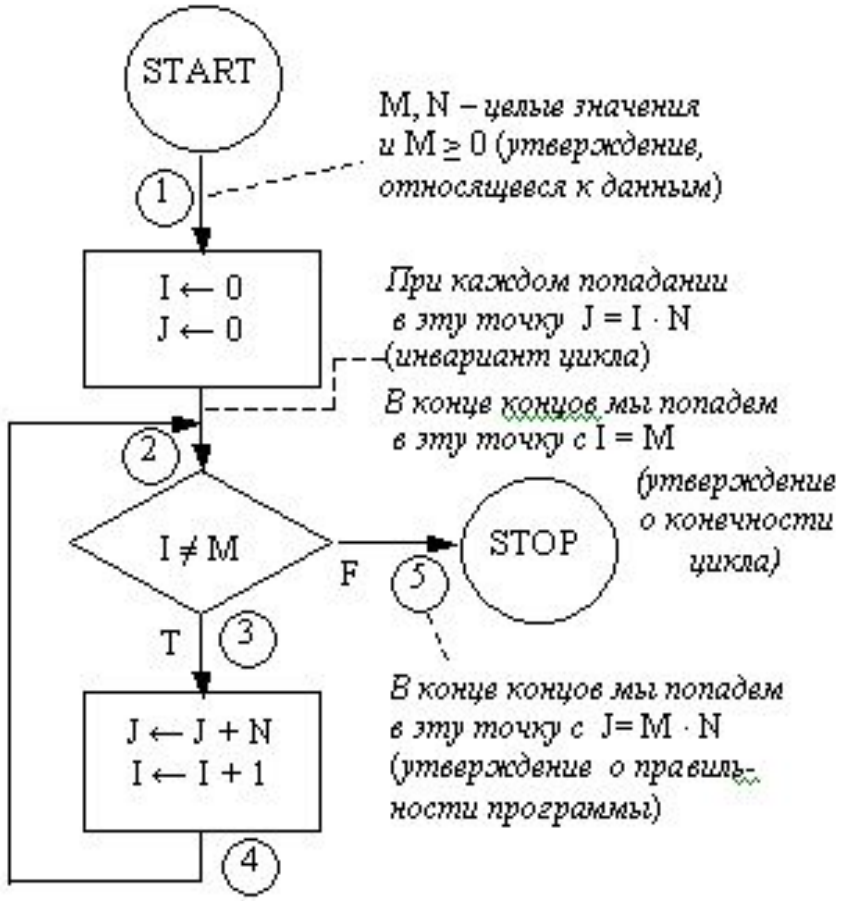
- Кроме того, припишем к этой же точке ключевое *утверждение о конечности цикла*, в котором говорится, что в конце концов мы попадем в эту точку со значениями переменных, обуславливающих прекращение выполнения тела цикла.

При доказательстве правильности блок-схем являются:

- 1) доказательство того, что при попадании во входную точку цикла всегда будет справедливо некоторое утверждение (инвариант цикла). Чтобы доказать это, мы показывали, во-первых, что это утверждение справедливо при первом попадании на вход цикла, и, во-вторых, если мы попали в эту точку и утверждение справедливо, то после выполнения цикла и возврата во входную точку утверждение будет оставаться справедливым;
- 2) доказательство того, что при одновременном выполнении утверждения, являющегося инвариантом цикла и утверждения о конечности цикла из инварианта цикла автоматически следует утверждение о правильности программы.

ПРИМЕР 2. Предположим, что нужно вычислить произведение двух любых целых чисел M, N, причем $M \geq 0$ и операцию умножения использовать нельзя. Для этого можно использовать операцию сложения и вычислить сумму из M слагаемых (каждое равно N). В результате получим $M \cdot N$. Рассмотрим блок-схему, реализующую такое вычисление. Нужно доказать, что приведенная программа правильно вычисляет произведение двух любых целых чисел M и N при условии $M \geq 0$, т. е. если программа выполняется с целыми числами M и N, где $M \geq 0$, то она в конце концов окончится (достигнув точки 5) со значением $J = M \cdot N$.

Доказательства правильности для блок-схемы очень простой программы



Докажем теперь, что приведенная на рис. блок-схема правильна, т. е. если ее начать выполнять с M и N, имеющими некоторые целые значения, причем $M \geq 0$, то выполнение в конечном итоге закончится с $J = M \cdot N$.

Вначале докажем, что при попадании в точку 2 $J = I \cdot N$.

1. При первом попадании в точку 2 при переходе из точки 1 имеем $I = 0$ и $J = 0$. Таким образом, утверждение $J = I \cdot N = 0 \cdot N = 0$ справедливо.

2. Предположим, что мы попали в точку 2 и утверждение $J = I \cdot N$ справедливо. Пусть I и J в этой точке принимают значения I_n и J_n, т.е. $J_n = I_n \cdot N$. Если $I \neq M$ ложно, то это уже обеспечивает конечный результат. Предположим теперь, что мы прошли по циклу (от точки 2 через точки 3, 4 вновь в точку 2), что возможно только при выполнении условия $I \neq M$. При возвращении в точку 2 I и J принимают новые значения I_{n+1} и J_{n+1}, которые можно представить следующим образом:

$$I_{n+1} = I_n + 1,$$

$$J_{n+1} = I_n \cdot N + N = (I_n + 1) \cdot N = I_{n+1} \cdot N.$$

(Так как $J_n = I_n \cdot N$)

Следовательно, при очередном попадании в точку 2 высказывание $J = I \cdot N$ вновь справедливо, что и требовалось доказать, т. е. при любом попадании в точку 2 справедливо высказывание $J = I \cdot N$.

Теперь докажем, что в конце концов попадем в точку 2 со значением $I=M$. При первом попадании в точку 2 имеем $I=0$. При последующих попаданиях, если таковые есть, I каждый раз увеличивается на 1. Так как значение M нигде в программе не изменяется и мы предположили, что $M \geq 0$, то очевидно, что в какой-то момент I станет равным M.

Если мы попадем в точку 2 при $I = M$, то будет верно и $J = I \cdot N = M \cdot N$. Отношение $I \neq M$ в этот момент ложно, и мы попадаем по стрелке с пометкой F (FALSE – ложь) в точку 5 с $J = M \cdot N$. На этом доказательство правильности программы заканчивается.

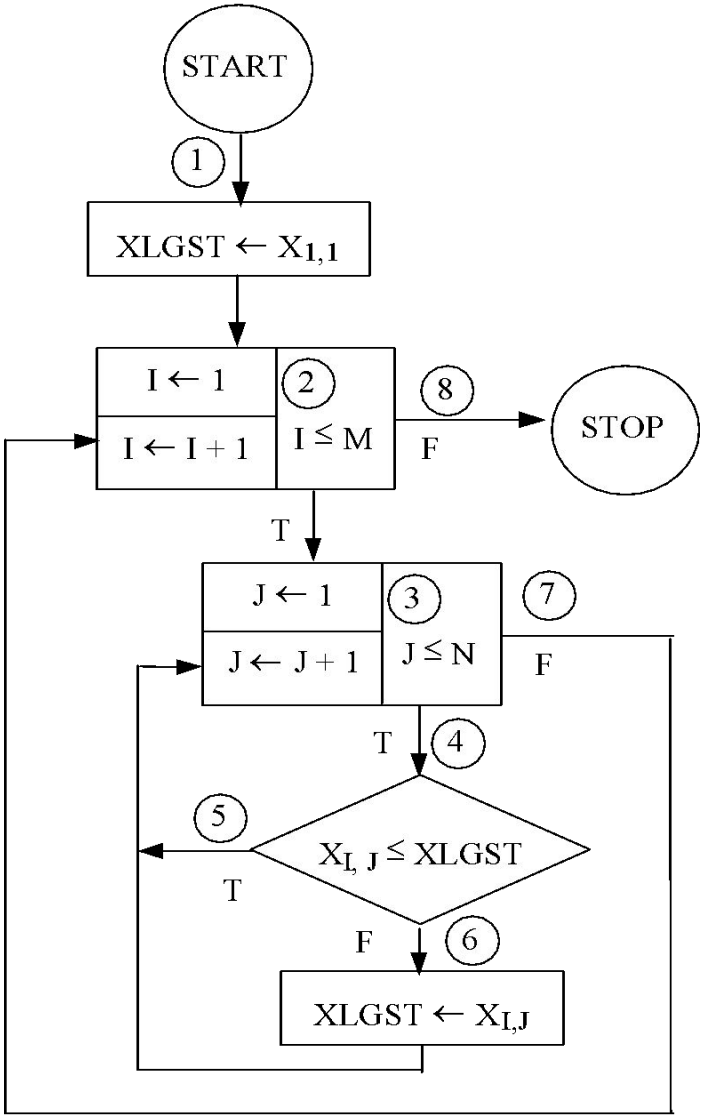
МЕТОД ИНДУКТИВНЫХ УТВЕРЖДЕНИЙ

- При доказательстве правильности программ методом индуктивных утверждений доказательство конечности программы проводится отдельно от доказательства справедливости некоторых ключевых утверждений при достижении соответствующих точек программы.
- Причем сначала необходимо доказать конечность программы, а после этого уже доказывать ее правильность, применяя метод индуктивных утверждений.

- Пусть A – некоторое утверждение, описывающее предполагаемые свойства данных в программе (блок-схеме), а C – утверждение, описывающее то, что мы по предположению должны получить в результате процесса выполнения программы (т. е. утверждение о правильности). Будем говорить, что программа правильна (по отношению к A и C), если программа заканчивается при всех данных, удовлетворяющих A, и каждом выполнении ее с данными, удовлетворяющими предположению A, будет справедливо утверждение C.

Метод индуктивных утверждений

- Свяжем утверждение **A** с началом программы, а утверждение **C** – с конечной точкой программы. Кроме этого, выявим некоторые закономерности, относящиеся к значениям переменных, и свяжем соответствующие утверждения с другими точками программы. В частности, свяжем такие утверждения по крайней мере с одной из точек в каждом из замкнутых путей в программе (в циклах). Для каждого пути в программе, ведущего из точки **i**, связанной с утверждением $\mathbf{A_i}$, в точку **j**, связанную с утверждением $\mathbf{a_j}$ (при условии, что на этом пути нет точек с какими-либо дополнительными утверждениями), докажем, что если мы попали в точку **i** и справедливо утверждение $\mathbf{A_i}$, а затем прошли от точки **i** до точки **j**, то при попадании в точку **j** будет справедливо утверждение $\mathbf{a_j}$. Для циклов точки **i** и **j** могут быть одной и той же точкой.


START
1
XLGST ← X1,1
I ← 1
I ← I + 1
2
I ≤ M
8
F
STOP
T
J ← 1
J ← J + 1
3
J ≤ N
7
F
T
4
5
XI, J ≤ XLGST
T
F
6
XLGST ← XI,J

Конечность программы

- Сначала надо удостовериться, что программа закончится. Отметим, что единственными местами в программе, где изменяются I и J, являются «изолированные» части двух итерационных блоков, управляющие увеличением параметра цикла. Так как значение J увеличивается на 1, а значение N при выполнении внутреннего цикла не изменяется (путь через точки 3, 4, 5, 3 или через точки 3, 4, 6, 3), то значение J должно в конце концов превысить значение N. Таким образом, попадая в точку 3, мы когда-нибудь (после конечного числа выполнений внутреннего цикла) должны попасть в точку 7, а затем в точку 2, При каждом попадании в точку 2 мы затем попадем либо в точку 8, и процесс закончится, либо в точку 3. Если мы попали в точку 3, мы только что видели, как в конце концов дойдем до точки 7 и вернемся в точку 2. При этом значение I будет увеличиваться на 1, а значение M останется неизменным. Следовательно, после конечного числа шагов значение I станет больше значения M. В этот момент мы перейдем из точки 2 в точку 8, и программа закончится. Таким образом, мы доказали конечность нашей программы.

Для доказательства правильности рассмотрим все пути

- 1. *Путь из точки 1 в точку 2.* Предположим, что мы находимся в точке 1 и связанное с ней утверждение справедливо, т. е. данные удовлетворяют исходному допущению. Перейдем из точки 1 в точку 2. Нужно показать, что после прихода в точку 2 связанное с этой точкой утверждение будет справедливо. Если мы попали в точку 2 из точки 1, то имеем I = 1 и XLGST = $X_{1,1}$. Так как M≥1, то очевидно, что 1 ≤ (I=1) ≤ M+1. Поскольку XLGST = $X_{1,1}$ и I=1, то утверждение о XLGST справедливо.

- 2. *Путь из точки 2 в точку 3.* Предположим, что мы находимся в точке 2 и справедливо связанное с нею утверждение. Перейдем из точки 2 в точку 3. Требуется показать, что при попадании в точку 3 будет справедливо утверждение, связанное с этой точкой. Если мы дошли до точки 3 (из точки 2), то имеем J = 1, а значение XLGST осталось неизменным. Так как N≥1, 1 ≤ (J=1) ≤ N+1, а значение I после точки 2 не изменялось, то 1 ≤ I ≤ M + 1. Однако если мы пришли из точки 2 в точку 3, то известно, что проверка 1≤M была истинной, и, комбинируя это отношение с 1 ≤ I ≤ M + 1, получаем 1 ≤ I ≤ M. Если I = 1и J = 1, то из утверждения 2 получим XLGST = $X_{1,1}$. В противном случае (т.е. при I≠1) из утверждения 2 получаем, что XLGST равно максимальному из значений элементов в первых I–1 строках массива X или, более точно, максимальному из значений элементов в первых I–1 строках и из значений первых J–1 = 1–1 = 0 элементов в I-й строке. Таким образом, очевидно, что при переходе из точки 2 в точку 3 утверждение, связанное с точкой 3, оказывается справедливым.

- *3. Путь из точки 3 через точки 4, 5 к точке 3.* Предположим, что мы находимся в точке 3 и справедливо утверждение, связанное с этой точкой. Пройдем через точки 4, 5 к точке 3. Нужно показать, что при возврате в точку 3 соответствующее утверждение останется справедливым. Пусть I и J в исходном положении в точке 3 принимают значения I_n и J_n. Мы имеем $1 \le I_n \le M$ и $1 \le J_n \le N+1$. При возврате в точку 3 через точки 4 и 5 получим $I_{n+1} = I_n$ и $J_{n+1} = J_n + 1$. Следовательно, опять имеем $1 \le I_{n+1} = I_n \le M$. Если мы проходим по этому пути, то проверка $J_n \le N$ была истинной. Из этого, а также из соотношения $1 \le J_n \le N+1$ получаем $1 \le J_n \le N$. Таким образом, при возврате в точку 3 снова имеем $1 < (J_{n+1} = J_n + 1) \le N+1$. На всем пути перехода в точку 3 значение XLGST не изменялось, и известно, кроме того, что истинна проверка ≤ XLGST. Если учесть истинность утверждения о XLGST до «прохода» по указанному пути, то данное утверждение остается истинным и при возврате в точку 3 с $I_{n+1} = I_n$ и $J_{n+1} = J_n + 1$. Так как ≤ XLGST, то неизменное значение XLGST снова будет максимальным из значений элементов в первых $I_{n+1} - 1 = I_n - 1$ строках и из значений первых $J_{n+1} - 1 = (J_n + 1) - 1 = J_n$ элементов в I_n-й строке массива X.

- *4. Путь из точки 3 через точки 4, 6 в точку 3.* Рассуждения для этого пути аналогичны приведенным для предыдущего пути, за исключением того, что при возврате в точку 3 XLGST будет иметь значения . Кроме того, так как выбран этот путь, проверка ≤ XLGST была ложной и, следова тельно, XLGST < . Таким образом, больше максимального из значений элементов первых I_n – 1 строк и J_n – 1 элементов в I_n-й строке. Отсюда следует, что при возврате в точку 3 значение XLGST остается максимальным из значений элементов первых I_{n+1} – 1 = I_n – 1 строк и первых J_{n+1} – 1 = (J_n + 1) – 1 = J_n элементов в I_{n+1} = I_n-й строке массива X.

- 5. *Путь из точки 3 через точку 7 в точку 2.* Предположим, что мы находимся в точке 3 и справедливо соответствующее утверждение. Перейдем из точки 3 через точку 7 в точку 2. Нужно показать, что при возврате в точку 2 будет справедливо связанное с нею утверждение. Из точки 3 в точку 7 мы попадем только тогда, когда ложна проверка $J \le N$. Но из утверждения, относящегося к точке 3, известно, что $1 \le J \le N+1$. Отсюда можно заключить, что $J = N+1$. Пусть I в точке 3 принимает значение I_n. Утверждение в точке 3 гласит:
  $1 \le I_n \le M$ и XLGST равно максимальному из значений элементов в первых $I_n - 1$ строках и первых $J - 1 = (N + 1) - 1 = N$ элементов в I_n-й строке массива X. Но так как в массиве X существует только N столбцов, то XLGST равно максимальному из значений элементов в I_n строках массива X. Значение XLGST между точками 3 и 2 не изменялось, а значение I изменилось и стало равным I + 1. Так как в точке 3 было справедливо отношение $1 \le I_n \le M$, то это означает, что $1 < (I_{n+1} = I_n + 1) \le M + 1$ справедливо в точке 2. Следовательно, при достижении точки 2 будет справедливо утверждение: XLGST равно максимальному из значений элементов в первых $I - 1 = (I_n + 1) - 1 = I_n$ строках массива X.

- *6. Путь от точки 2 в точку 8.* Предположим, что мы находимся в точке 2, справедливо соответствующее утверждение и мы переходим в точку 8. Необходимо показать, что при достижении точки 8 будет справедливо связанное с нею утверждение. Из точки 2 мы перейдем в точку 8, если была ложной проверка $I \leq M$. Однако в точке 2 было справедливо неравенство $1 \leq I \leq M+1$; следовательно, можно заключить, что $I = M+1$. Значение XLGST при переходе из 2 в 8 не изменялось, и, таким образом, из утверждения, относящегося к точке 2, можно заключить, что при попадании в точку 8 XLGST равно максимальному из значений элементов в первых $I - 1 = (M + 1) - 1 = M$ строках массива X. Но массив X содержит только M строк; следовательно, XLGST равно максимальному из значений элементов X.
- Если объединить это доказательство с предыдущим доказательством конечности программы, то отсюда следует полная правильность программы.

Основной метод при доказательстве правильности рекурсивных программ называется *методом структурной индукции*.

F(X1, ... , XN) ≡ IF *проверка 1* THEN *выражение 1*

ELSE IF *проверка 2* THEN *выражение 2*

:

ELSE IF *проверка m* THEN *выражение m*

ELSE OTHERWISE *выражение m + 1*

Такая программа вычисляет значение F(X1,... , XN).

функция F, определяемая в нашей программе, может входить как часть в любое из выражений или проверок в программе. Такое появление F называется рекурсивным обращением к этой функции. Для каждой такой рекурсивной программы нужно еще указать, для каких значений аргументов X1, ... , XN применима программа, т. е. нужно задать область определения функции. Выполнение программы заключается в применении ее к некоторым конкретным данным из ее области определения.

Пример 7.1. Рассмотрим рекурсивную программу, определенную для любого положительного целого числа X:

F(X) ≡ IF X = 1 THEN 1

ELSE OTHERWISE X•F(X–1)

Чтобы понять, как работает такая программа, выполним ее для некоторого конкретного значения аргумента X. Например,

F(4) = 4 • F(3) [Так как условие 4 = 1 ложно, то F(4)=4•F(3).

Теперь нужно вычислить F(3), т.е. рекурсивно

обратиться к F.]

= 4 • 3 • F(2) [Так как при вычислении F(3) условие 3 = 1 ложно,

то F(3) = 3•F(2).]

= 4 • 3 • 2 • F(1) [Так как условие 2 = 1 ложно, то F(2) = 2•F(1).]

= 4 • 3 • 2 • 1 [Так как условие 1 = 1 истинно, то F(1) = 1.]

= 24 = 4 !

Далее мы докажем, что F(X) = X! для любого положительного целого числа X.

- Чтобы сделать наш язык программирования точным, нам нужно задать правила вычислений для программ, определяющие последовательность вычислений. Будем считать, что правило вычислений отдает предпочтение *самому левому* и *самому внутреннему* обращению. Другими словами, в любой из моментов вычислений первым начинает вычисляться *левое* и *внутреннее* обращение (для простоты далее везде будем опускать слово «самое») к функции F, все аргументы которой уже не содержат F. Это правило – не обязательно наилучшее, иногда оно может приводить к неоканчивающейся последовательности вычислений, хотя другое правило дало бы конечную последовательность (пример 4.3). Однако во многих существующих языках программирования используется правило вычислений, подобное описанному выше. Кроме того, большинство рассматриваемых нами программ заканчивается при любых значениях аргументов независимо от того, какие правила вычислений мы используем, а следовательно, и результат в любых случаях будет одинаковым. Поэтому для определенности будем считать, что предпочтение отдается левому внутреннему обращению.

Структурная индукция

- Рекурсивные программы обычно строятся по следующей схеме: сначала в них явно определяется способ вычисления результата для наипростейших значений аргументов рекурсивной функции, затем способ вычисления результата для аргументов более сложного вида, причем используется сведение к вычислениям с данными более простого вида (с аргументами рекурсивного обращения к той же функции). Таким образом, вполне естественно попытаться доказать правильность таких программ следующим образом:
- 1) доказать, что программа работает правильно для простейших данных (аргументов функции);
- 2) доказать, что программа работает правильно для более сложных данных (аргументов функции) в предположении, что она работает правильно для более простых данных (аргументов функции).
- Такой способ доказательства правильности для рекурсивных функций естествен, поскольку он следует основной схеме вычислений в рекурсивных программах. Назовем его доказательством с помощью структурной индукции.

- Докажем правильность рекурсивной программы:
- F(X) ≡ IF X = 1 THEN 1
- ELSE OTHERWISE X•F(X – 1)
- Предполагается, что эта функция вычисляет факториал от аргумента. Нужно доказать, что F(M) = 1 • 2 • 3 • ... • N = N! для любого положительного числа N. При доказательстве с помощью структурной индукции используем простую индукцию по положительным целым числам:
- **1.** Докажем, что F(1)= 1!. Действительно, F(1)= 1=1!.
- **2.** Докажем (для любого положительного числа N), что если F(N) = 1 • 2 • ... • N = N!, то F(N + 1) = 1 • 2 • ... • N • (N + 1) = (N + 1)!. Следовательно, мы предполагаем, что N – положительное число, а F(N) = N! – гипотеза индукции. Так как N положительное число, то проверка N + 1 = 1 дает отрицательный ответ, и, прослеживая далее последовательность вычислений, получаем
- F(N + 1) = (N + 1) • F((N + 1) – 1) = (N + 1) • F(N) =
  = (N + 1) • (N!) = (N + 1) • (1 • 2 • ... • N) = 1 • 2 • ... • N • (N + 1) = (N + 1)!
- (По гипотезе
  индукции)
- что и требовалось доказать, т. е. F(N) = N! для любого положительного числа N.

Рекурсия и итерация

- ***Рекурсия*** **— это такой способ организации обработки данных, при котором программа вызывает сама себя непосредственно, либо с помощью других программ.**
- ***Итерация*** **— способ организации обработки данных, при котором определенные действия повторяются многократно, не приводя при этом к рекурсивным вызовам программ.**