Лекция 13
БГТУ, ФИТ, ПОИТ, 3 семестр
Конструирование программного обеспечения

Лексический анализатор: проектирование, принципы построения и реализации

Цель: ознакомление с назначением и принципами работы лексических анализаторов (сканеров), получение практических навыков построения лексического анализатора на примере заданного простейшего входного языка.

I Общая схема работы лексического анализатора

1. Первая фаза работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором (сканером).

   Вход лексического анализатора: последовательность символов входного языка.

   Выход лексического анализатора: таблица лексем (ТЛ) и таблица идентификаторов (ТИ)

   Лексический анализатор выделяет в этой последовательности простейшие конструкции языка, которые называют лексическими единицами (токенами). Примеры лексических единиц: идентификаторы, числа, символы операций, служебные слова и т.д.

   Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, с целью создания промежуточного представления исходной программы.

   Каждой лексеме сопоставляется ее тип и для некоторых лексем создается запись в таблице идентификаторов, в которой сохраняется дополнительная информация.

   Таблица лексем (ТЛ) и таблица идентификаторов (ТИ) являются входом для следующей фазы компилятора – синтаксического анализа (разбора, парсера).

   Пример 1: лексеме ***v***, соответствует одна из операций (**+ – * /**), определенных в учебном языке.

   Пример 2: идентификатору учебного языка ***sc*** соответствует лексеме ***i***. Для нее необходимо сохранить имя идентификатора (***sc***), его тип (string) и т.д.

Лексический анализ важен для процесса компиляции по следующим причинам:

- лексический анализ уменьшает длину программы, устраняя из ее исходного представления несущественные пробелы и комментарии;
- замена в программе идентификаторов, констант, разделителей, ключевых слов лексемами делает представление программы более удобным для дальнейшей обработки;
- упрощает разработку (для выделения и анализа лексем применяются эффективные и простые алгоритмы разбора);
- избавляет синтаксический анализатор от работы с текстом исходный программы.

Функции лексического анализатора:

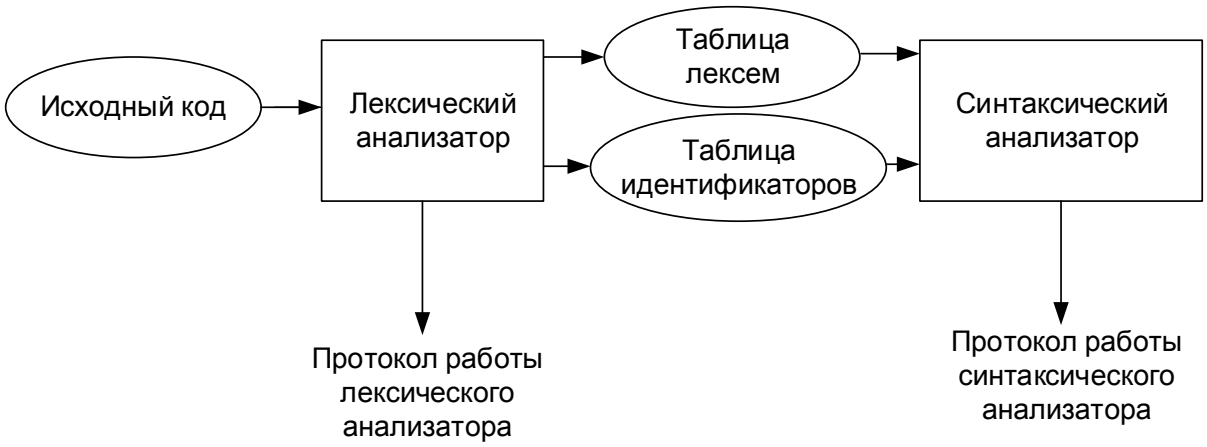
- удаление «пустых» символов и комментариев. Если «пустые» символы (пробелы, знаки табуляции и перехода на новую строку) и комментарии будут удалены лексическим анализатором, синтаксический анализатор никогда не столкнется с ними (альтернативный способ, состоящий в модификации грамматики для включения «пустых» символов и комментариев в синтаксис, достаточно сложен для реализации);
- распознавание идентификаторов и ключевых слов;
- распознавание констант;
- распознавание разделителей и знаков операций.

Каждый распознаватель представляет собой детерминированный конечный автомат, совокупность всех распознавателей составляет основу сканера.
На уровне лексического анализатора определяются только некоторые ошибки, поскольку лексический анализатор рассматривает исходный текст программы в ограниченном контексте.

Лексический анализатор должен выдавать сообщения о наличии во входном тексте ошибок, если они будут обнаружены на этапе лексического анализа.

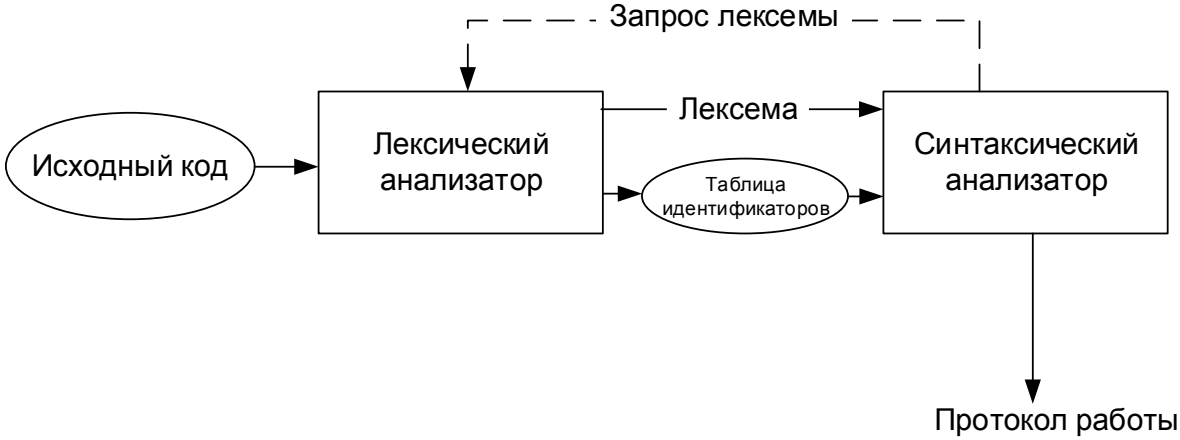
Взаимодействие лексического и синтаксического анализаторов может быть последовательным и параллельным.

2. Последовательное взаимодействие лексического и синтаксического анализаторов.



При последовательном варианте взаимодействия выполняется анализ всего текста исходной программы и выходом лексического анализатора является таблица лексем (ТЛ) и таблица идентификаторов (ТИ).

3. Параллельное взаимодействие лексического и синтаксического анализаторов.



4. Для описания лексики языка программирования обычно применяются регулярные грамматики.

С точки зрения лексического анализатора – язык программирования – это набор лексем (токенов), которые распознаются (классифицируются) лексическим анализатором по шаблонам, описывающим вид соответствующих лексем. Множество слов, или строк символов, которые соответствуют шаблону, называют языком. Для описания шаблонов используются регулярные выражения. Язык программирования (на уровне лексического анализа) представляет собой регулярный язык, заданный регулярным выражением (язык типа 3 иерархии Хомского).

II Напоминание (лекция 10).

1. Грамматика описывает множество правильных цепочек символов над заданным алфавитом. Лексику языка программирования описывает регулярная грамматика 3-го типа иерархии Хомского. Как правило, для описания регулярных языков не применяют грамматики в виду громоздкости записи, а используют другую форму – регулярные выражения.

2. Регулярное выражение описывает множество цепочек – формальный язык. Для записи регулярного выражения используются метасимволы.

Регулярные выражения – способ записи спецификации шаблонов лексем.

Множество цепочек, описанных регулярным выражением называется **регулярным множеством** (или регулярным языком).

3. Определение регулярного множества:

Пусть I – алфавит.
Регулярные выражения над алфавитом I и языки, представляемые ими, рекурсивно определяются следующим образом:

1) $\varnothing$ – регулярное выражение и представляет пустое множество;
2) λ – регулярное выражение и представляет множество $\{\lambda\}$, содержащее только пустую строку λ;
3) для каждого $a \in I$ символ a является регулярным выражением и представляет множество $\{a\}$;

   Операции:
4) если p – регулярное выражение, представляющее множество P, если q – регулярное выражение, представляющее множество Q, то $p+q$, pq, q^* являются регулярными выражениями и представляют множества $P \cup Q$ (объединение множеств), PQ (конкатенация множеств) и $P*$ (замыкание Клини) соответственно. P^+ – (положительное замыкание).
5) $pp^* = p^+$

Символы, применяемые для описания регулярных выражений, называются **метасимволами** или **символами-джокерами**. Символами-джокерами являются символы: $*, ^+, +, (,), \varnothing$.

Комментарий

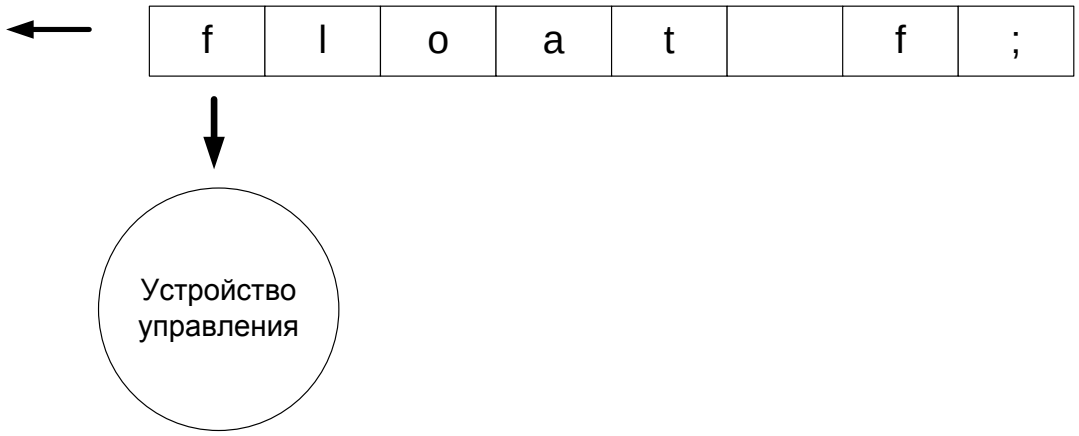
Для каждой лексемы надо разработать регулярное выражение.
Пример: для лексемы ***t*** регулярное выражение `integer` соответствует непосредственно строке символов `integer`.

При описании лексических структур полезно вводить имена регулярным выражениям и использовать эти имена для ссылки на них по имени.

Пример использования имен для регулярных выражений для множества идентификаторов (шаблон).

Letter = A|B|C|…|X|Y|Z|a|b|c|...|x|y|z|_		Letter = [A-Za-z_]
Digit = 0|1|...|9	==>	Digit = [0-9]
Identifier = Letter(Letter|Digit)*		Identifier = Letter(Letter|Digit)*

4. Схема работы лексического анализатора



Распознаватель – это алгоритм, позволяющий определить некоторое множество (входной язык).

Простейший распознаватель состоит из ***входной ленты*** (входная цепочка символов), ***управляющего устройства*** с конечной памятью.

Класс алгоритмов, соответствующих приведенной схеме, может быть записан в форме конечного автомата (КА).

Регулярные выражения описывают регулярные множества.
Для распознавания регулярных множеств служат конечные автоматы.

5. Определение конечного автомата (КА):

КА это пятерка $M = (S, I, \delta, s_0, F)$,
где
S – конечное множество состояний устройства управления;
I – алфавит входных символов;
δ – функция переходов, отображающая $S \times (I \cup \{\lambda\})$ в множество подмножеств S : $\delta(s,i) \subset S, s \in S, i \in I$;
$s_0 \in S$ – начальное состояние устройства управления;
$F \subseteq S$ – множество заключительных (допускающих) состояний устройства управления.

Если $\delta(s,\lambda) = \varnothing$ и $|\delta(s,a)| \leq 1$, то конечный автомат **детерминированный** (ДКА) иначе – конечный автомат **недетерминированный** (НКА).

!!! Лексический анализатор можно создать на базе регулярной грамматики, и построить эквивалентный ДКА, что снизит сложность разбора до O(N), где N – длина строки.

Определения.

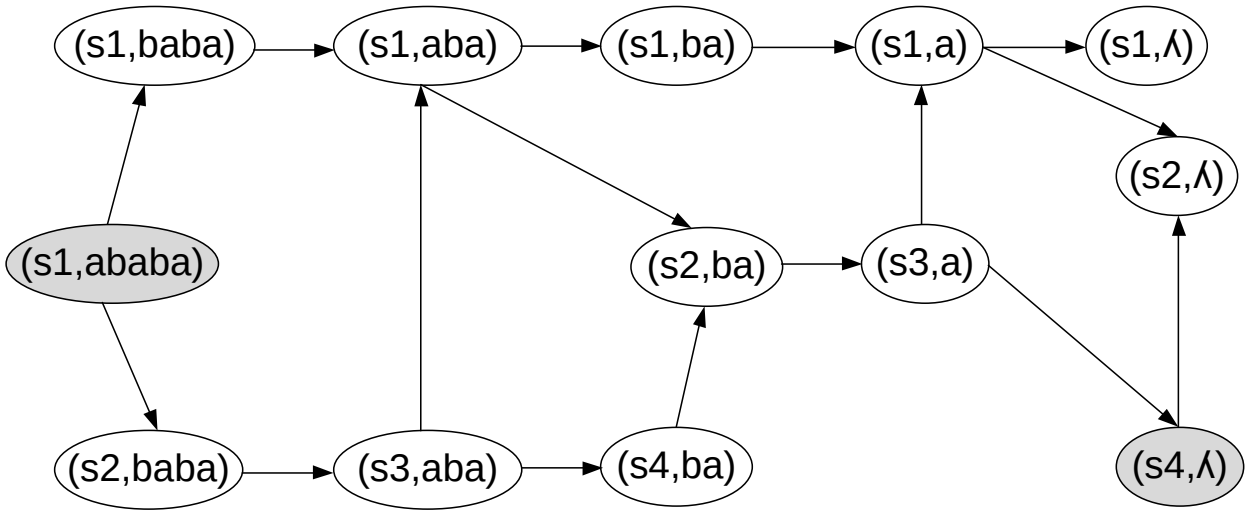
- ✓ Мгновенное описание КА является пара (s, w), где $s \in S$ – состояние КА, $w \in I^*$ – неиспользованная часть входной цепочки.
- ✓ (s_0, w_0) – начальное мгновенное описание КА,
  где $s_0 w_0$ – начальное состояние КА, w_0 – анализируемая цепочка.
- ✓ $(s_f, \lambda), s_f \in S$ – допускающее мгновенное описание КА.
- ✓ Если (s, aw) и $s' \in \delta(s,a)$, где $s', s \in S$, $a \in I \cup \lambda$, $w \in I^*$, то $(s, aw) \succ (s', w)$ – непосредственно следует.
- ✓ Если $(s_i, w_i) \succ (s_{i+1}, w_{i+1}) \succ (s_{i+2}, w_{i+2}) \succ \ldots \succ (s_k, w_k)$, то $(s_i, w_i) \succ *(s_k, w_k)$ – следует.
- ✓ Если $(s_0, w) \succ *(s_f, \lambda)$, а $s_0 \in S$ – начальное состояние и $s_f \in F$ – конечное состояние, то цепочка $w \in I^*$ допускается (распознается) КА.

Пример: пусть входная цепочка описывается регулярным выражением $w \in (a+b)^* aba$,

КА $M = (\{s_1, s_2, s_3, s_4\}, \{a, b\}, \delta, s_1, \{s_4\})$, где функция δ задана следующей таблицей:

	a	b	λ
s_1	$\{s_1, s_2\}$	$\{s_1\}$	$\varnothing$
s_2	$\varnothing$	$\{s_3\}$	$\varnothing$
s_3	$\{s_4\}$	$\varnothing$	$\{s_1\}$
s_4	$\varnothing$	$\varnothing$	$\{s_2\}$

Последовательность мгновенных описаний автомата, распознающего цепочку ababa:



Из мгновенного состояния за конечное количество шагов можно попасть в заключительное: $(s_1, abaaba) \succ^* (s_4, \lambda)$ – это значит, что автомат M допускает (или распознает) цепочку $abaaba$.

Доказаны 4 утверждения:

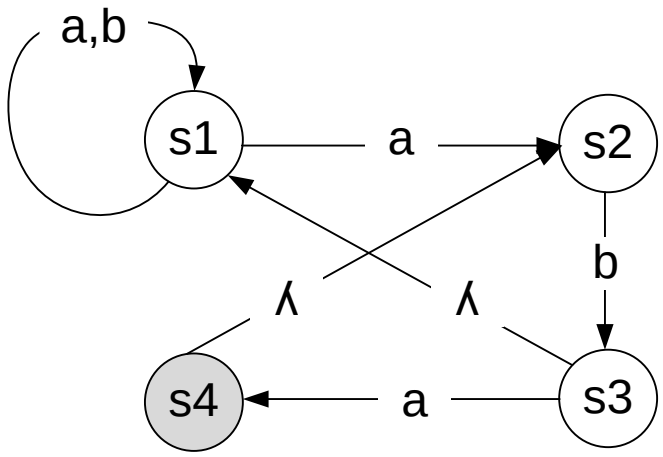
1) язык является регулярным множеством тогда и только тогда, когда он задан регулярной грамматикой;
2) язык может быть задан регулярной грамматикой (левосторонней или правосторонней) тогда и только тогда, когда язык является регулярным множеством;
3) язык является регулярным множеством тогда и только тогда, когда он задан конечным автоматом;
4) язык распознается с помощью конечного автомата тогда и только тогда, когда он является регулярным множеством.

Другими словами: любой ***регулярный язык*** может быть задан ***регулярной грамматикой***, ***регулярным выражением*** или ***конечным автоматом***.
Или: любой ***конечный автомат*** задает ***регулярный язык***, а значит ***регулярную грамматику*** или ***регулярное выражение***.

Определение

Графом переходов конечного автомата $M = (S, I, \delta, s_0, F)$ называется ориентированный граф $G = (S, E)$,
где
S – множество вершин графа совпадает с множеством состояний КА,
E – множество ребер (направленных линий, соединяющих вершины),
ребро $(s_i, s_j) \in E$, если $s_j \in \delta(s_i, a), a \in I \cup \lambda$.
Метка ребра (s_i, s_j) – все a, для которых $s_j \in \delta(s_i, a)$.

Пример



Конечный автомат может быть однозначно задается своим графом переходов.

Доказана теорема (А. Ахо, Дж. Хопкрофт, Дж. Ульман): пусть α – регулярное выражение, тогда найдется недетерминированный конечный автомат $M = (S, I, \delta, s_0, \{s_f\})$, допускающий автомат, представленный α, и обладающий следующими свойствами:

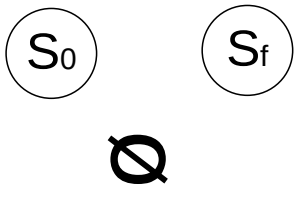
1) $|S| \le 2|\alpha|$;

2) $\forall a \in I \cup \{\lambda\} : \delta(s_f, a) = \varnothing$;

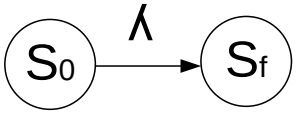
3) $\forall s \in S : \sum_{a \in I \cup \{\lambda\}} |\delta(s, a)| \le 2$.

6. Построение графа конечного автомата по регулярному выражению.

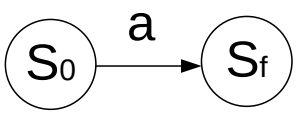
Алгоритм МакНотона-Ямады-Томпсона (McNaughton-Yamada-Thompson) для регулярного выражения в НКА.



Автомат для выражения ∅, обозначающего множество ∅.

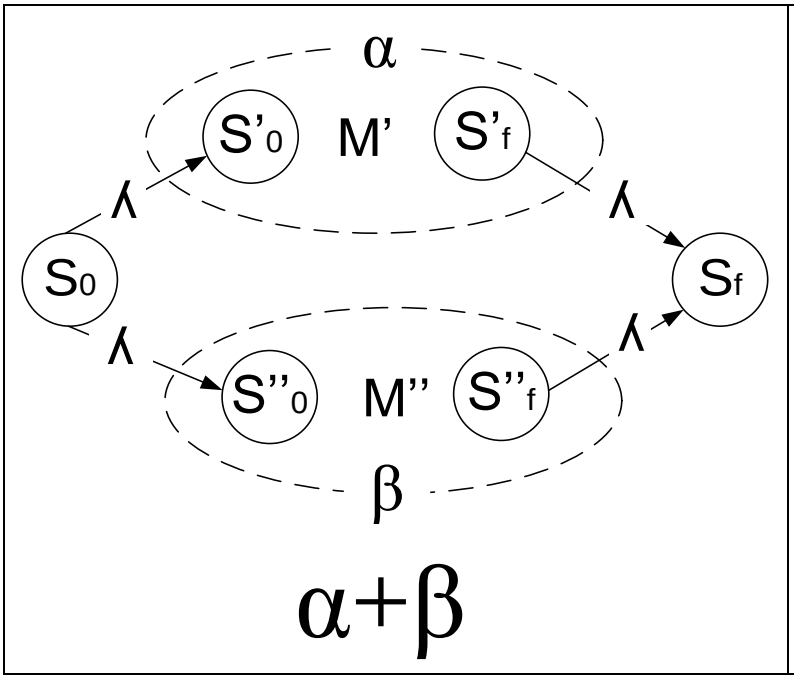


Автомат для выражения λ.



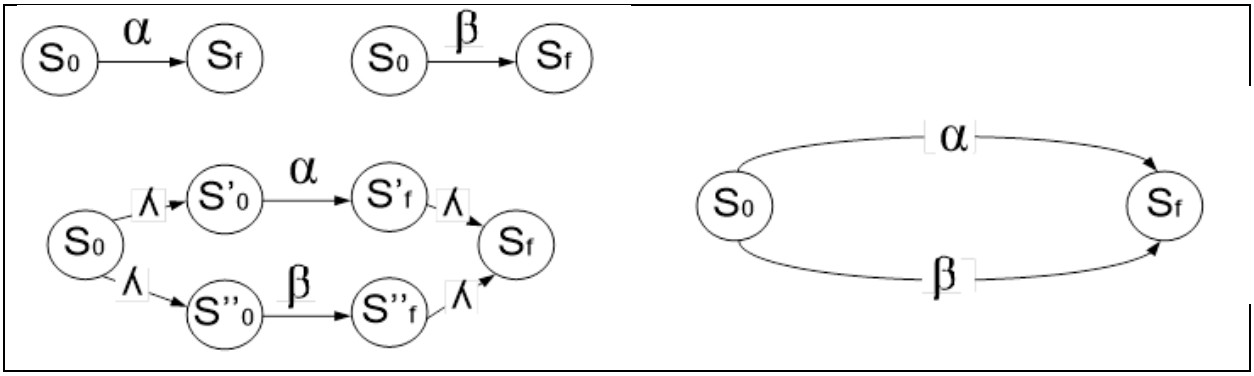
Автомат для выражения a.

Строим автомат для выражения α+β.

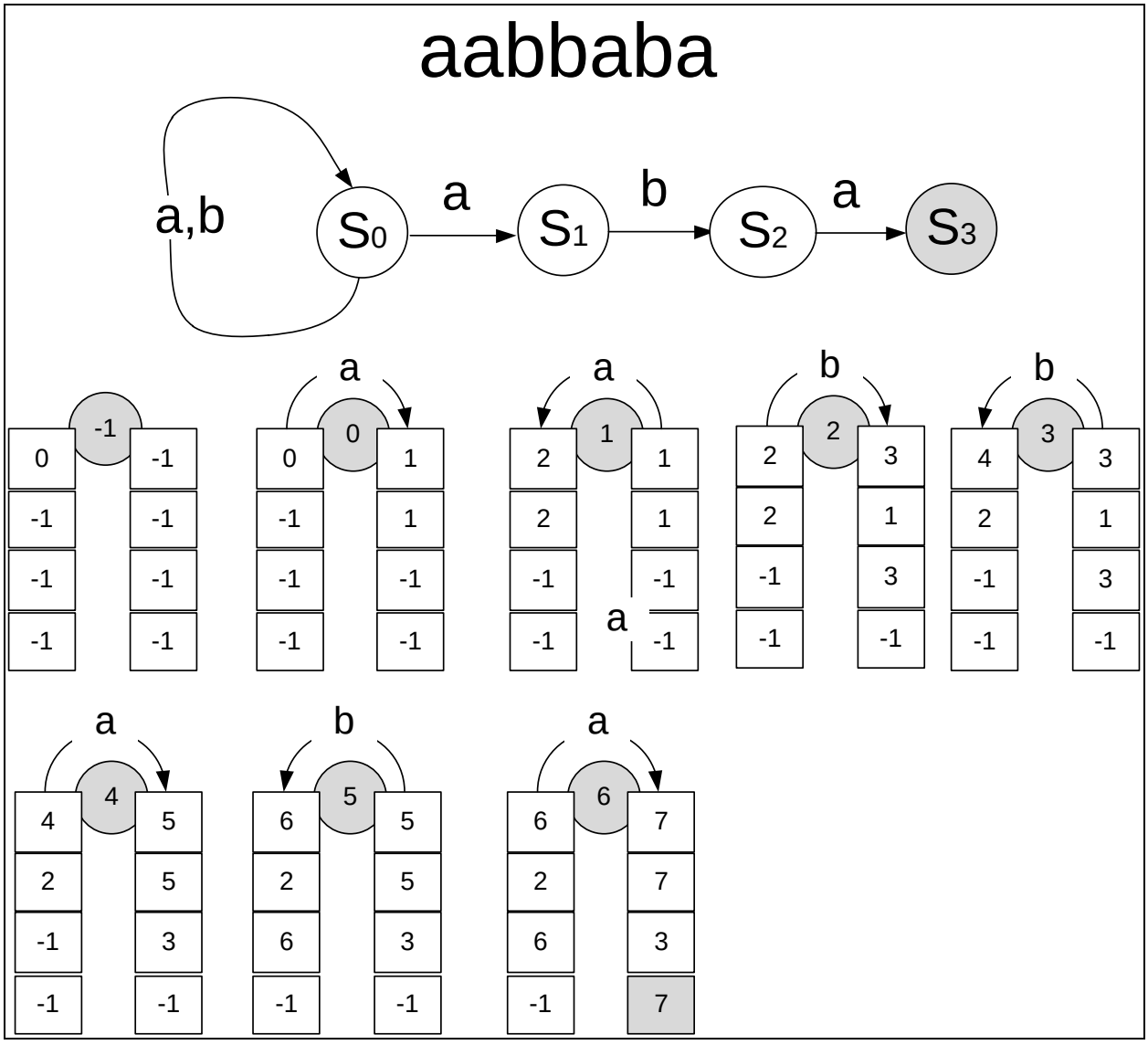


Здесь M’ и M’’ – автоматы для выражений α и β, S_0’, S_0’’ - новые начальные состояния и S_f’ и S_f’’ - новые заключительные состояния. Имеет место переход по λ из S_0 в начальные состояния автоматов M’ и M’’ и переход по λ из заключительных состояний M’ и M’’ в S_f.

Автомат для выражения α+β:



7. Алгоритм для разбора с двумя массивами.



Два массива, размерность которых равна количеству состояний автомата.
Инициализация:

позиция = -1,

значение первого элемента массива (элемент массива с индексом 0) = 0.

После каждой итерации моделирования такта работы автомата номер позиции увеличивается на 1. Значение элементов массива, индекс которых равен новому состоянию перехода по соответствующему символу перехода, увеличивается на 1.

Если разбор цепочки выполнен успешно (автомат разобрал цепочку), то возвращается **true**, иначе – **false**.

Признаком успешного разбора: значение последнего элемента результирующего массива равное количеству символов входной цепочки.

III Программирование лексического анализатора.

1. Определение границ лексем.

Определение границ лексем – это выделение тех строк в общем потоке входных символов, для которых надо выполнять распознавание.

Для простейших входных языков границы лексем распознаются по заданным терминальным символам (пробелы, скобки, знаки операций, символы комментариев, а также разделители (запятые, точки с запятой и др.)). Набор таких терминальных символов может варьироваться в зависимости от входного языка.

Важно: сами знаки операций также являются лексемами, и необходимо не пропустить их при распознавании текста.

2. Алгоритм работы простейшего сканера:

- проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для вычисления номера строки для каждой лексемы;
- для выделенной части входного потока выполняется функция распознавания лексемы;
- при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;
- формирует протокол работы;
- при неуспешном распознавании выдается сообщение об ошибке, а дальнейшие действия зависят от реализации сканера – либо его выполнение прекращается, либо делается попытка распознать следующую лексему (идет возврат к первому этапу алгоритма).

3. Описание языка:

Компонента	Описание
Символы	Windows-1251
Символы-сепараторы	пробел – допускается везде кроме идентификаторов и ключевых слов; **;**(точка с запятой) – разделитель инструкций; **{}** – программный блок; () – параметры; () – приоритетность операций.
Идентификаторы	только малые буквы, от 1 до 5 букв идентификатор не может совпадать с ключевыми словами максимальное количество идентификаторов 2^{16}
Типы данных	**integer** – целочисленные данные (четыре байта, от -2^{31} до 2^{31}-1),автоматическая инициализация 0, LE; **string** – строка, любые символы, (макс. 255 символов, первый байт длина строки), автоматическая инициализация строкой длины 0
и т.д.	**…**

4. Пример правильной программы:

```
integer function fi(integer x, integer y)
{
 declare  integer z;
 z= x*(x+y);
 return z;
}
string function fs (string a, string b)
{
 declare  string c;
 declare  string function substr(string a, integer p,
                          integer n);
 c = substr(a, 1,3)+ b;
 return c;
};
main
{
 declare integer x;
 declare integer y;
 declare integer z;
 declare string sa;
 declare string sb;
 declare string sc;
 declare integer function  strlen(string p);
 x = 1;
 y = 5;
 sa =  '1234567890 ';
 sb =  '1234567890 ';
 z = fi(x,y);
 sc = fs(sa,sb);
 print  'контрольный пример ';
 print z;
 print sc;
 print strlen(sc);
 return 0;
};
```

5. **Убрать все лишние пробелы:**

- подстроки, состоящие из более, чем одного пробела заменить на один пробел;
- пробельные префиксы и суффиксы для символов ;,} {()=+-/*;

Для подсчета номера строки ввести специальный символ |.

6. **Построить регулярные выражения для лексем:**

- ***типы данных***: integer, string;
- ***идентификатор***: $(a+b+c+d+\ldots+z)^{+}$;
- ***ключевые слова***:
  - function;
  - declare;
  - main;
  - print;
  - return;
- операции: =+'+'+-+/+*;
- string-литерал: $+(a+b+c+\ldots+z+-+a+б+..+я+1+2+3\ldots+0)^{*}+$ ';
- integer-литерал: $(1+2+3+4+5+6+7+8+9+0)^{+}(1+2+3+4+5+6+7+8+9+0)^{*}$
- открытие блока: {
- закрытие блока: }
- левая круглая скобка: (
- правая круглая скобка:)

Для каждой лексемы разработать регулярное выражение, например, для лексемы **t** регулярное выражение **integer** (шаблон) соответствует непосредственно строке integer.

Далее по заданию строим распознаватель – граф переходов конечного автомата для этой и аналогично для всех остальных лексем.

7. **Лексемы:**

конструкция	лексема	примечание
`integer` `string`	`t`	ТИ: integer или string, значение по умолчанию: для integer – нуль, для string – пустая строка
<идентификатор>	`i`	ТИ: строка идентификатора, усеченная до 5 символов. Префикс: имя конструкции
…		**…**

Приложение вводит текст программы на языке SVV-2015 из входного файла, проверяет входные символы на допустимость, удаляет из текста лишние пробелы и т.п. Выделенные лексические единицы анализируются построенными конечными автоматами последовательно, и соответствующие этим токенам лексемы заносятся в таблицу лексем. При необходимости дополнительная информация для лексем заносится в таблицу идентификаторов.

8. Фрагмент исходного кода, в виде лексем:

```
01 tfi(ti,ti)
02 {
03 dti;
04 i=i*(i+i);
05 ri;
06 };
```

9. Представление таблиц (ТЛ и ТИ)

Выбирая способ представления таблиц (ТЛ и ТИ), следует руководствоваться следующими требованиями к ним:

- структура таблиц должна обеспечивать эффективность поиска и вставки в таблицах;
- структура таблиц должна обеспечивать возможность динамического роста объемов таблиц.

10.Ошибки

Если в процессе работы лексический анализатор не смог правильно определить тип лексемы, считается, что программа содержит ошибку. Информация об ошибке с указанием строки и позиции выдается пользователю (в консоль и сохраняется в протоколе).
На уровне лексического анализатора определяются только некоторые ошибки.

11.Результат лексического разбора (таблица лексем)

Вход лексического анализатора	Выход (таблица лексем)	Дополнительная информация (таблица идентификаторов)
integer	t	
fi	i	fi – идентификатор функции, integer
function	f	
(	(	
integer	t	
x	i	fix – имя, параметр, integer
,	,	
integer	t	
y	i	fiy– имя,параметр integer
)	)	
{	{	
declare	d	
integer	t	
z	i	fiz - имя, integer, значение: 0
…	…	…