Санкт-Петербургский государственный университет
информационных технологий, механики и оптики
Факультет информационных технологий и программирования
Кафедра «Компьютерные технологии»

И. И. Чернявский

Генерация автоматных лексических анализаторов по регулярным выражениям

Санкт-Петербург
2009

Оглавление

Введение

В данной курсовой работе показано применение конечных автоматов для лексического анализа последовательности символов. Результатом работы является программа, написанная на языке программирования *Eiffel* [1], которая принимает на вход регулярные выражения, строит по ним конечный автомат и выдает в качестве результата исходный код лексического анализатора, использующего построенный автомат.

1. Постановка задачи

Цель данной работы – создание генератора лексических анализаторов. Для точной формулировки понятия лексического анализатора приведем ряд определений [2].

1.1. Основные определения

Будем называть *алфавитом* любое конечное множество символов, *словом* – последовательность символов из алфавита, *языком* – множество слов. Слово нулевой длины будем обозначать как ε.

Введем следующие операции над языками:

- *конкатенация* языков *L* и *M* – язык, слова которого можно образовать путем дописывания любого слова из *M* к слову из *L*;
- *объединение* языков *L* и *M* – язык, любое слово которого принадлежит либо L, либо *M*, либо пересечению множеств слов языков *L* и *M*;
- *замыкание Клини* (*итерация*) языка *L* – язык, состоящий из всех слов, которые можно образовать путем конкатенации любого числа слов из *L*.

Назовем *регулярными* следующие языки:

- пустой язык (не содержащий в себе слов), язык, содержащий единственное слово – ε, а также языки, содержащие единственное слово, являющееся символом алфавита;
- языки, являющиеся конкатенациями регулярных языков;
- языки, являющиеся объединениями регулярных языков;
- языки, являющиеся замыканиями Клини регулярных языков.

Для задания и описания регулярных языков используются *регулярные выражения*:

- для пустого языка регулярным выражением является Ø;
- для языка, содержащего единственное слово – ε, регулярным выражением является ε;
- для языка, содержащего единственное слово – символ алфавита, регулярным выражением является данный символ алфавита;
- для конкатенации регулярных языков, имеющих регулярные выражения α и β – (α)(β);
- для объединения регулярных языков, имеющих регулярные выражения α и β – (α)|(β);
- для замыкания Клини регулярного языка, имеющего регулярное выражение α – (α)*.

Для того, чтобы уменьшить число скобок в регулярных выражениях, введем приоритеты операций – наивысший приоритет присвоим операции замыкания Клини, меньший приоритет – операции конкатенации и наименьший приоритет – операции объединения.

Для удобства записи регулярных выражений введем дополнительные обозначения:

- (α)? – эквивалентно записи ((α)|ε) и означает «ноль или один раз»;
- (α)+ – эквивалентно записи ((α)(α)*) и означает «один и более раз»;
- [a-c] – эквивалентно записи (a|b|c) для заданного диапазона символов.

Приведем примеры регулярных выражений:

1. `(0|1)*0` – регулярное выражение, описывающее язык двоичных записей четных натуральных чисел.
2. `[a-z]*abba[a-z]*` – регулярное выражение, описывающее язык слов, состоящих из строчных латинских букв и содержащих в себе подстроку `abba`.

1.2. Лексический анализатор

Лексический анализатор – программа, принимающая на вход текст (последовательность символов из алфавита) и разбивающая его на подстроки (*лексемы*) в соответствии с некоторым набором регулярных выражений. Последовательность лексем такова, что каждая лексема принадлежит хотя бы одному из языков, задаваемых регулярными выражениями.

1.3. Конечный автомат

Детерминированный конечный автомат (ДКА) – пятерка `{Q, Ξ, δ, q, F}`, где

- `Q` – конечное множество состояний;
- `Ξ` – конечное множество входных символов;
- `δ` – функция переходов, аргументами которой являются текущее состояние и входной символ, а значением – новое состояние;
- `q` – начальное состояние;
- `F` – множество допускающих состояний.

Недетерминированный конечный автомат (НКА) отличается от ДКА значением функции переходов, которое является множеством состояний, а не единичным состоянием.

Конечный автомат можно представить в виде *диаграммы переходов*, пример которой изображен на рис. 1.

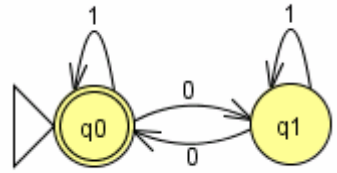

Рис. 1

Диаграмма переходов – граф, вершины которого соответствуют состояниям автомата, а дуги – переходам из одних состояний в другие, определяемым функцией переходов `δ`. Дуга подписывается символом, по которому осуществляется переход. Также на диаграмме отмечаются допускающие состояния и начальное состояние.

Начиная работу в начальном состоянии, ДКА читает входную строку символ за символом и, на основе функции переходов, определяет состояние, в которое необходимо перейти. Автомат допускает строку, если после прочтения всех ее символов находится в допускающем состоянии.

В случае НКА, автомат на каждом шаге определяет множество состояний, в которое он может перейти, и выбирает любое из них для перехода. Автомат допускает строку, если существует последовательность выборов состояний, приводящая к допускающему состоянию.

Введем следующее расширение понятия НКА – ε-НКА. Это расширение отличается от НКА функцией переходов, которая может принимать в качестве аргументов не только пару, состоящую из состояния и символа алфавита, но и пару, состоящую из состояния и специального символа ε. В этом случае говорят, что автомат может совершать спонтанные ε-переходы из одного состояния в другое, не читая при этом следующий символ строки.

ε-НКА полезны при рассмотрении вопроса о построении автоматов по регулярным выражениям (разд. 2.1).

1.4. Генератор лексических анализаторов

Задачей генератора лексических анализаторов является создание лексического анализатора по заданному набору регулярных выражений.

В данной работе используется автоматный подход к реализации лексических анализаторов – вся логика анализа строки на наличие в ней искомых лексем сосредоточена в построенном генератором конечном автомате. Программа, которая является результатом работы программы-генератора, эмулирует работу построенного автомата для поиска лексем, соответствующих регулярным выражениям.

2. Принцип работы

Работа программы-генератора состоит из следующих этапов.

1. Преобразование регулярного выражения в ε-НКА.
2. Преобразование ε-НКА в ДКА.
3. Минимизация ДКА.

Рассмотрим каждый из них.

2.1. Преобразование регулярного выражения в недетерминированный автомат

Перед началом построения ε-НКА программа-генератор преобразовывает регулярное выражение в постфиксную форму. Постфиксная форма позволяет избежать использования скобок в выражении и упрощает процесс построения автомата.

Программа читает регулярное выражение слева направо посимвольно и выполняет следующие действия:

- если прочитан символ алфавита – программа создает и размещает в стеке автомат, допускающий строку длины один, содержащую данный символ. Этот автомат показан на рис. 2;

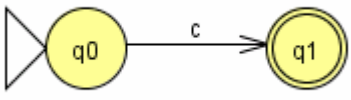

Рис. 2

- если прочитан оператор конкатенации – программа вынимает из стека два автомата, конкатенирует их, как показано на рис. 3, и размещает в стеке новый автомат;

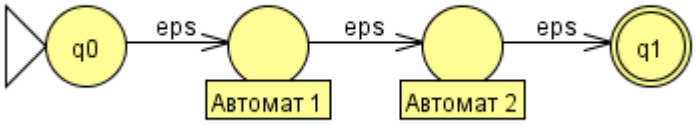

Рис. 3

- если прочитан оператор объединения – программа вынимает из стека два автомата, объединяет их, как показано на рис. 4, и размещает в стеке новый автомат;

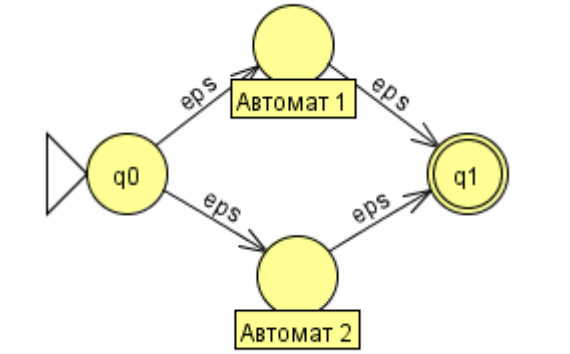

Рис. 4

- если прочитан оператор итерации – программа вынимает из стека автомат, изменяет его, как показано на рис. 5, и размещает в стеке новый автомат;

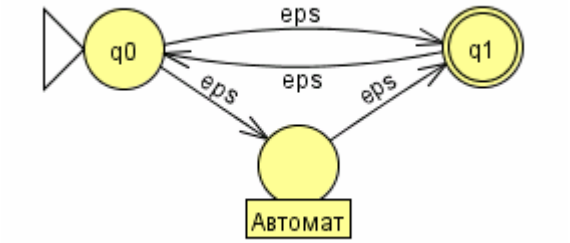

Рис. 5

- если прочитан оператор ? – программа вынимает из стека два автомата, строит новый, как показано на рис. 6, и размещает его в стеке;

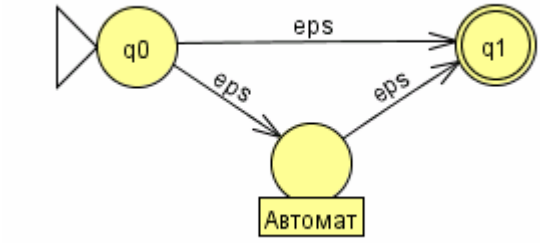

Рис. 6

- если прочитан оператор + - программа вынимает из стека два автомата, строит новый, как показано на рис. 7, и размещает его в стеке.

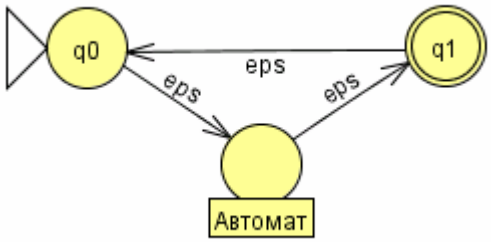

Рис. 7

После завершения чтения регулярного выражения, в стеке остается единственный элемент – построенный ε-НКА.

2.2. Преобразование недетерминированного автомата в детерминированный автомат

Определим *ε-замыкание* состояния q как множество состояний, доступных из q только по ε-переходам.

Детерминированный автомат, эквивалентный данному недетерминированному с ε-переходами, строится следующим образом:

- множество состояний – множество всех подмножеств состояний исходного автомата;
- множество входных символов такое же, как у исходного автомата;
- функция переходов принимает в качестве аргументов состояние (множество состояний исходного автомата) q и символ алфавита c, значение функции – состояние, соответствующее следующему множеству состояний исходного автомата, в которые можно перейти по символу c из ε-замыкания множества состояний q;
- начальное состояние – ε-замыкание начального состояния исходного автомата;
- допускающие состояния – все множества состояний исходного автомата, содержащие допускающие состояния.

2.3. Минимизация ДКА

В программе реализован алгоритм Хопкрофта [2] для минимизации ДКА.

Алгоритм итеративно строит разбиение множества состояний следующим образом.

1. Первоначальное разбиение множества состояний – допускающие состояния и не допускающие (два блока).
2. Алгоритм помещает в очередь все пары (B, c), где B – блок наименьшего размера из двух начальных, c – символ алфавита.
3. Далее, пока не пуста очередь, алгоритм выполняет п.4 – п.7.
4. Из очереди извлекается пара (B, c).
5. Все блоки текущего разбиения разбиваются на 2 блока (один из которых может быть пустым) – те состояния блока, которые по символу c переходят в состояние блока B и те, которые по символу c не переходят в состояние блока B.
6. Те блоки, которые разбились на два непустых подблока, заменяются подблоками в разбиении.
7. Для каждого блока B, разбитого на два непустых подблока, выполняется следующий цикл: для каждого символа c проверяется, есть ли в очереди пара (B, c). Если есть, то она

заменяется на две соответствующие пары для подблоков с символом c. Если нет, то в очередь добавляется пара, состоящая из подблока наименьшего размера и символа c.

Будем говорить, что строка различает два состояния, если, начав процесс допуска из одного состояния, автомат допускает строку, а из другого – не допускает. Назовем два состояния автомата *эквивалентными*, если не существует строки, которая может их различить.

Блоки полученного разбиения являются классами эквивалентности на множестве состояний автомата. Состояния, принадлежащие одному блоку можно объединить в одно, не изменяя при этом язык, допускаемый данным автоматом.

3. Реализация генератора лексических анализаторов

В данном разделе остановимся подробнее на программе-генераторе лексических анализаторов, созданной в данной работе.

3.1. Язык программирования

В качестве языка программирования для программы-генератора был выбран объектно-ориентированный язык *Eiffel*. Особенностью языка является встроенная поддержка *программирования по контракту* (*Design by Contract*) – подхода к разработке программного обеспечения, основанного на проектировании верифицируемых интерфейсов между программными компонентами. Использование данного подхода позволяет уменьшить количество ошибок в программах и увеличить надежность приложений.

3.2. Структура программы-генератора

Программа состоит из следующих классов:

- `APPLICATION` – класс, содержащий стартовую процедуру программы;
- `NFA` – класс, описывающий недетерминированный автомат;
- `DFA` – класс, описывающий детерминированный автомат;
- `STATE` – класс, описывающий состояние автомата;
- `STATES` – класс, описывающий хешируемое множество состояний автомата;
- `ID_COUNTER` – вспомогательный класс для нумерации состояний;
- `PARSER_EXCEPTION` – класс, описывающий информацию об ошибке в программе.

Результатом работы программы-генератора является программа лексического анализатора, которая состоит из следующих классов:

- `APPLICATION` – класс, содержащий стартовую процедуру программы;
- `LEX_PARSER` – класс, описывающий работу лексического анализатора;
- `LEXEME` – класс, описывающий лексему.

3.3. Интерфейс программы лексического анализатора

Программа лексического анализатора имеет текстовый интерфейс, позволяющий пользователю вводить строки текста и выводить на экран список лексем или информацию об ошибке.

Интерфейс программы и пример работы с ней приведен на рис. 8.

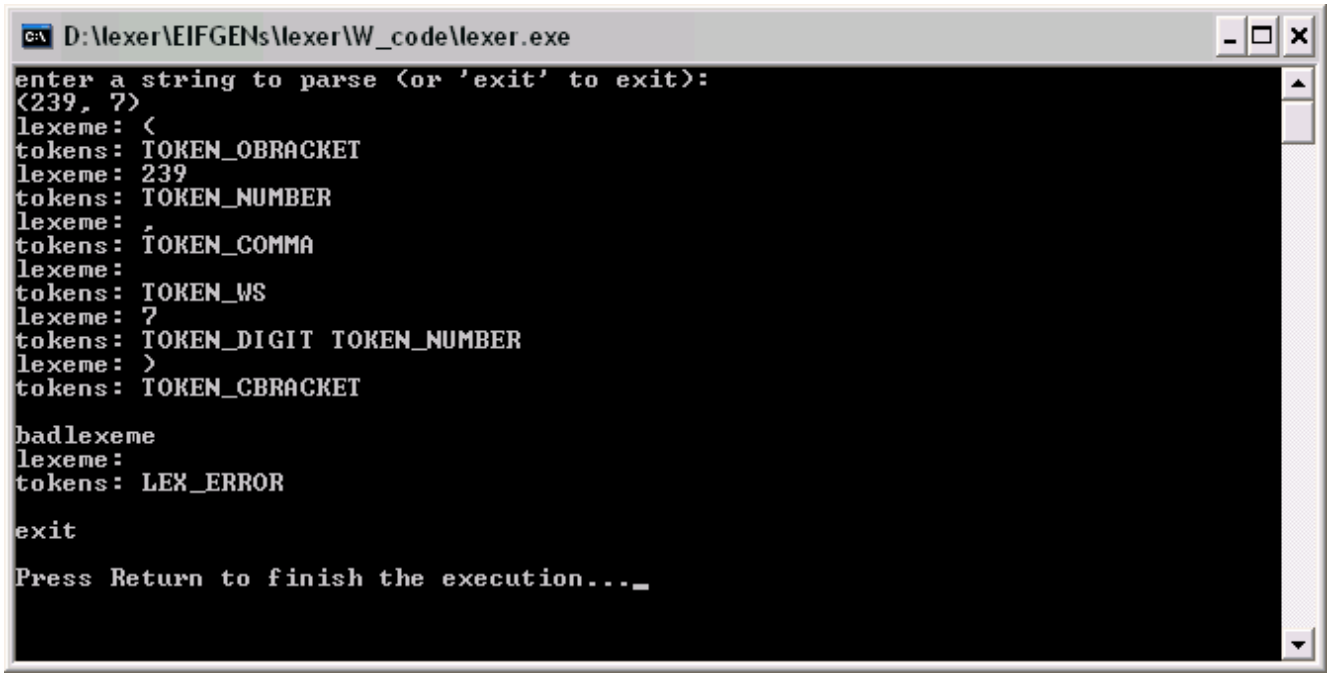

Рис. 8

4. Пример построения лексического анализатора

Рассмотрим построение лексического анализатора по следующим входным данным:

```
TOKEN_WS            [ ]*
TOKEN_COMMA         ,
TOKEN_CDE           [c-e]
TOKEN_AB            [ab]+
```

В начале своей работы программа-генератор строит НКА для каждого регулярного выражения. Рассмотрим построение НКА для регулярного выражения `[ab]+`.

Программа преобразует выражение, раскрывая класс символов `[ab]` в выражение `(a|b)` и переводя запись в постфиксную форму:

```
ab|+
```

Программа читает `ab|*` слева направо и выполняет следующие действия:

1. Встретив `a`, программа строит автомат для разбора регулярного выражения `a` (рис. 9).

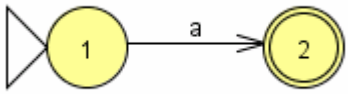

Рис. 9

2. Прочитав `b`, программа строит автомат для разбора регулярного выражения `b` (рис. 10).

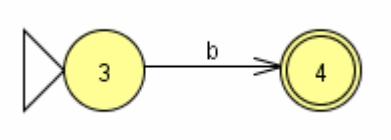

Рис. 10

3. Прочитав `|`, программа объединяет автоматы и получает автомат для выражения `a|b` (рис. 11).

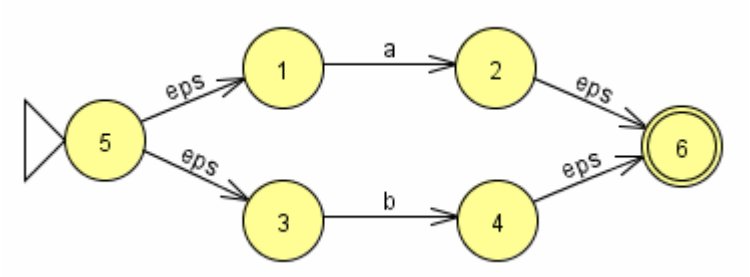

Рис. 11

4. Прочитав +, программа замыкает автомат и получает автомат для исходного выражения `[ab]+` (рис. 12).

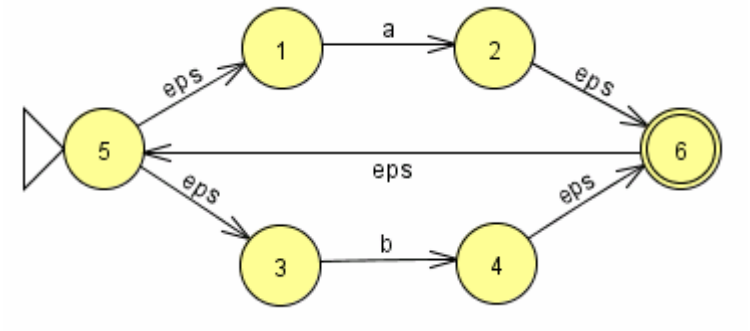

Рис. 12

Приведем автоматы для остальных регулярных выражений (рис. 13).

- `[ ]*` (рис. 13).

Рис. 13

- `,` (рис. 14).

Рис. 14

- `[c-e]` (рис. 15).

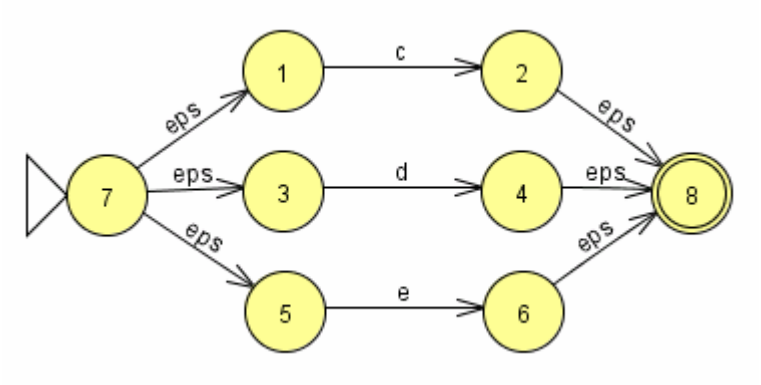

Рис. 15

После построения автоматов для регулярных выражений программа строит автомат для лексического анализа, объединяя все автоматы и подписывая заключительные состояния (рис. 16).

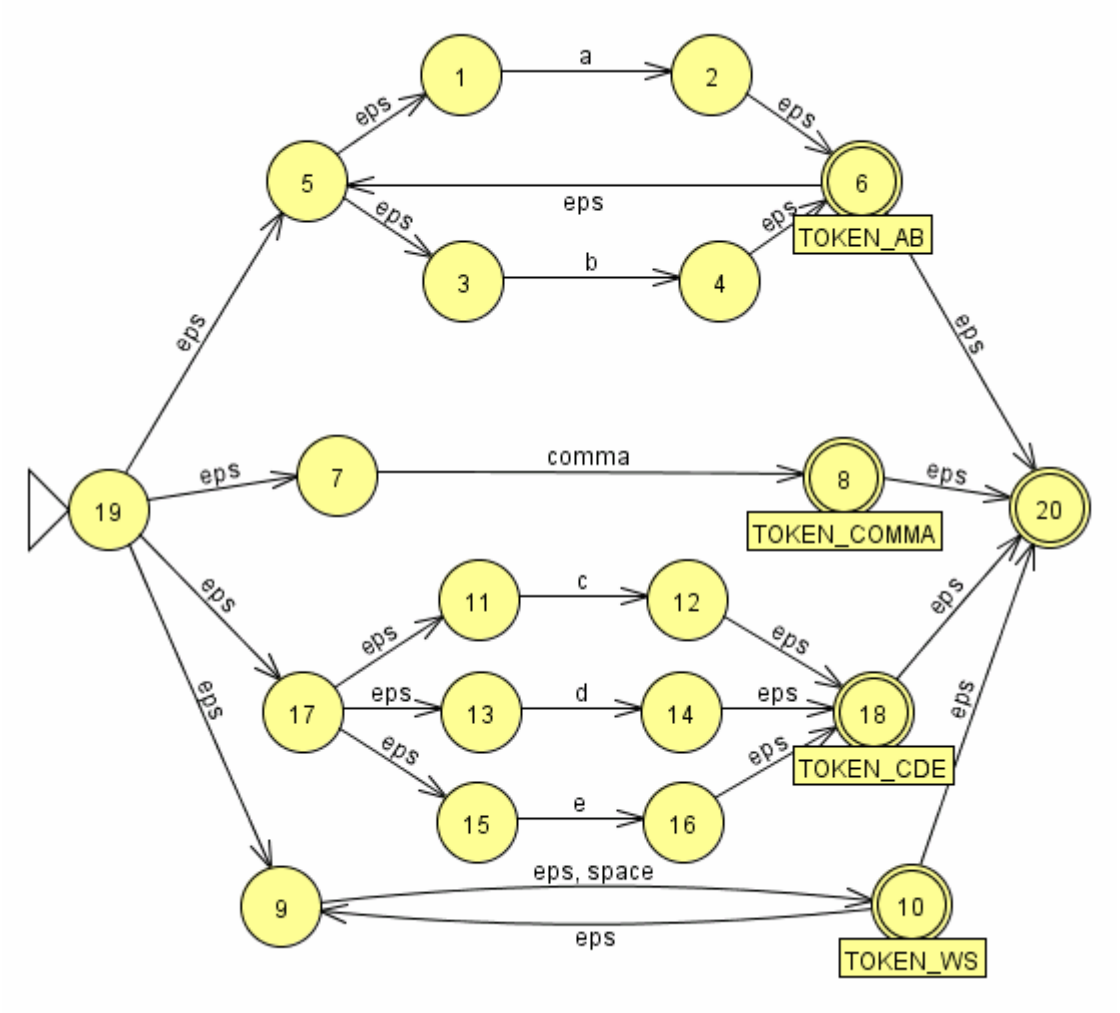

Рис. 16

Программа преобразует полученный НКА в ДКА и получает автомат (рис. 17).

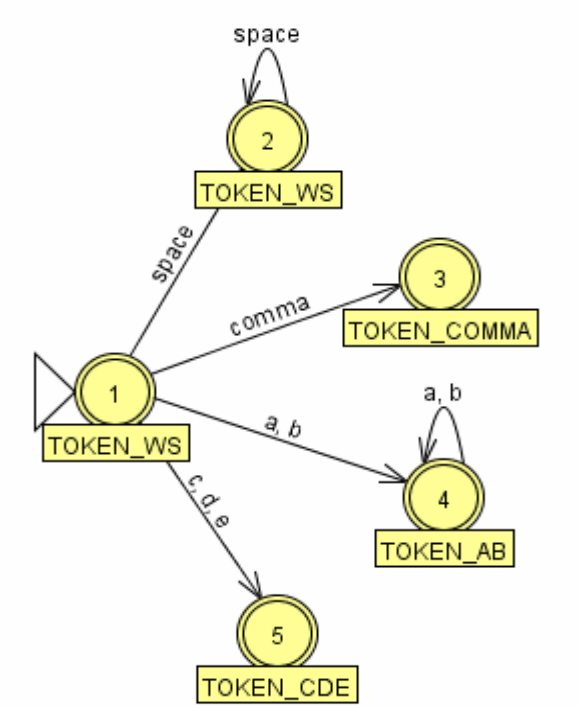

Рис. 17

К данному ДКА применяется алгоритм Хопкрофта для минимизации числа состояний. В результате получается эквивалентный автомат (рис. 18).

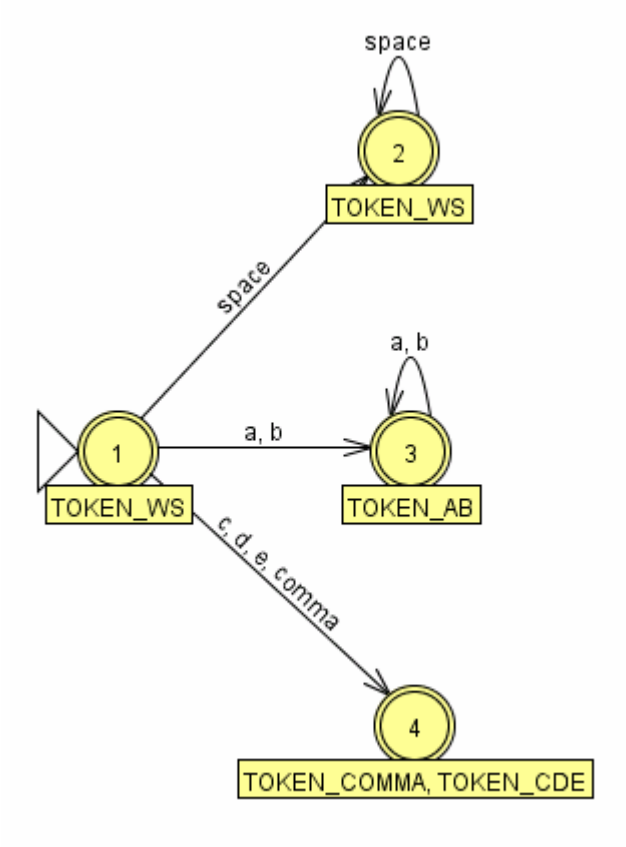

Рис. 18

Заметим, что, несмотря на то, что после минимизации стало на одно состояние меньше, минимизированный автомат не подходит для лексического анализа, так как соответствующий лексический анализатор не сможет различить лексемы `TOKEN_COMMA` и `TOKEN_CDE`.

Поэтому программа-генератор реализует модифицированный алгоритм Хопкрофта – на последнем шаге блоки состояний сжимаются в состояния так, чтобы заключительные состояния разных лексем не сливались в одно. В этом случае автомат на рис. 17 является оптимальным – число его состояний минимально.

Заключение

Автоматный подход к реализации лексических анализаторов, примененный в данной работе, завоевал большую популярность вследствие наглядности и относительной простоты [2]. Поэтому существует множество генераторов лексических анализаторов, использующих автоматы, например, генераторы *Flex* (http://flex.sourceforge.net), *ANTLR* (http://www.antlr.org).

Эта область, наряду с построением протоколов, была до последнего времени основной областью, в которой в программировании использовались автоматы. Применение автоматного программирования [3] резко увеличило сферу применений, в которых использовать автоматы при программировании целесообразно.

Источники

1. *Язык* программирования *Eiffel*. www.eiffel.com;
2. *Хопкрофт Д., Мотвани Р., Ульман Д.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002.
3. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб.: Питер, 2009.

Приложение

Исходные коды:

APPLICATION.e

```
class
      APPLICATION

inherit
      ARGUMENTS

create
      make

feature
      make
            local
                  fin: PLAIN_TEXT_FILE
                  fcl, fout: PLAIN_TEXT_FILE
            do
                  if  argument_count = 1 then
                        create fin.make (argument(1))
                        if fin.exists and fin.is_readable then
```

```
                              create res.make (64)

                              fin.open_read
                              fin.start
                              parse (fin)
                              fin.close

                              if res.count > 0 then
                                    compute_tables
                                    create fcl.make ("lex_parser.code")
                                    if fcl.exists and fcl.is_readable then
                                          create        fout.make_create_read_write
("lex_parser.e")
                                          if fout.is_writable then
                                                fout.start
                                                fcl.open_read
                                                fcl.start
                                                fcl.copy_to (fout)
                                                fcl.close
                                                write_tables(fout)
                                                fout.close
                                          else
                                                io.put_string ("error: can't write
to lex_parser.e%N")
                                          end
                                    else
                                          io.put_string ("error: can't read from
lex_parser.code%N")
                                    end
                              else
                                    io.put_string ("error: correct 'token regexp'
entries are not found%N")
                              end
                        else
                              io.put_string ("error: can't open/read input file%N")
                        end
                  else
                        io.put_string ("error: invalid number of arguments%N")
                  end
            end

feature {NONE}
      res: HASH_TABLE [NFA, STRING]

      transitions: ARRAY [ARRAY [INTEGER]]
      tokens: ARRAY [ARRAY [STRING]]

      raise_exc (msg: STRING)
            do
                  (create {PARSER_EXCEPTION}.make (msg)).raise
            end

      get_exc: EXCEPTION
            do
                  Result := (create {EXCEPTION_MANAGER}).last_exception
            end

      handle_exc
            local
                  e: EXCEPTION
            do
                  e := get_exc
                  if e.message /= Void and then not e.message.is_empty then
                        io.put_string ("error: " + e.message + "%N");
                  end
            end
```

```
	parse (fin: PLAIN_TEXT_FILE)
		local
			token, regexp: STRING
			fa: NFA
		do
			from
			until fin.off
			loop
				fin.read_word
				token := fin.last_string.twin
				if not fin.off then
					fin.read_line
					regexp := fin.last_string.twin
					trim_whitespace (regexp)
					if regexp.count /= 0 then
						io.put_string ("token: " + token + " regexp: "
+ regexp + "%N")

						fa := compute_fa (regexp);
						if res.has (token) then
							io.put_string	("warning:	token
redefinition%N")
						end

						res.force (fa, token)
					else
						raise_exc ("invalid file entry (" + token +
")%N")
					end
				else
					if token.count /= 0 then -- check if it's not last
empty line
						raise_exc ("invalid file entry (" + token +
")%N")
					end
				end
			end
		rescue
			handle_exc
			retry
		end


	trim_whitespace (str: STRING)
		do
			from
			until
				str.count = 0 or else str.at (1) /= ' ' and str.at (1) /=
'%T'
			loop
				str.remove (1)
			end
		end

	write_tables(fout: PLAIN_TEXT_FILE)
		local
			i, j: INTEGER
		do
			from
				i := 1
			until
				i > transitions.count
			loop
				fout.put_string ("
	<<")
```

```
                  from
                        j := 1
                  until
                        j > transitions[i].count
                  loop
                        fout.put_integer (transitions[i].at (j))
                        if j /= transitions[i].count then
                              fout.put_string (", ")
                        end
                        j := j + 1
                  end
                  fout.put_string (">>")
                  if i /= transitions.count then
                        fout.put_string (",")
                  end
                  fout.put_new_line
                  i := i + 1
            end

            fout.put_string ("                                                >>%N
            tokens := <<%N")
            from
                  i := 1
            until
                  i > tokens.count
            loop
                  if tokens[i] /= Void then
                        fout.put_string ("
<<")
                        from
                              j := 1
                        until
                              j > tokens[i].count
                        loop
                              fout.put_character ('"')
                              fout.put_string (tokens[i].at (j))
                              fout.put_character ('"')
                              if j /= tokens[i].count then
                                    fout.put_string(", ")
                              end
                              j := j + 1
                        end
                        fout.put_string (">>")
                  else
                        fout.put_string ("
Void")
                  end
                  if i /= tokens.count then
                        fout.put_string (",")
                  end
                  fout.put_new_line
                  i := i + 1
            end
            fout.put_string ("                                          >>%N
end%Nend%N")
      end

compute_tables
      local
            ffa: NFA
            df: DFA
      do
            create ffa.make_final (res)
            create df.make (ffa)
            df.minimize
```

```
            transitions := df.get_transitions_table
            tokens := df.get_tokens_table
      end

compute_fa (regexp: STRING): NFA
      local
            st: DS_LINKED_STACK[NFA]
            rxpand, rxp: STRING
            c: CHARACTER
            i: INTEGER
            fa, fb: NFA
            fc: NFA

      do
            rxpand := prepare (regexp)
            rxp := convert_to_postfix (rxpand)

            create st.make
            -- \*+?.|
            from
                  i := 1
            until
                  i > rxp.count
            loop
                  c := rxp.at (i)

                  inspect c
                  when '*' then
                        if st.count > 0 then
                              fa := st.item
                              st.remove
                              create fc.make_star (fa)
                              st.force (fc)
                        else
                              raise_exc ("malformed regexp")
                        end

                  when '+' then
                        if st.count > 0 then
                              fa := st.item
                              st.remove
                              create fc.make_plus (fa)
                              st.force (fc)
                        else
                              raise_exc ("malformed regexp")
                        end

                  when '?' then
                        if st.count > 0 then
                              fa := st.item
                              st.remove
                              create fc.make_quest (fa)
                              st.force (fc)
                        else
                              raise_exc ("malformed regexp")
                        end

                  when '.' then
                        if st.count > 1 then
                              fb := st.item
                              st.remove
                              fa := st.item
                              st.remove
                              create fc.make_concat (fa, fb)
                              st.force (fc)
                        else
```

```
                              raise_exc ("malformed regexp")
                        end

                  when '|' then
                        if st.count > 1 then
                              fb := st.item
                              st.remove
                              fa := st.item
                              st.remove
                              create fc.make_or (fa, fb)
                              st.force (fc)
                        else
                              raise_exc ("malformed regexp")
                        end
                  when '\' then
                        c := rxp.at (i + 1)
                        i := i + 1
                        create fc.make_character (c)
                        st.force (fc)

                  else
                        create fc.make_character (c)
                        st.force (fc)
                  end
                  i := i + 1
            end

            if st.count = 1 then
                  Result := st.item
            else
                  raise_exc ("malformed regexp")
            end

      rescue
            handle_exc
      end

convert_to_postfix (regexp: STRING): STRING
      local
            st: DS_LINKED_STACK[CHARACTER]
            c: CHARACTER
            i: INTEGER
      do
            create st.make
            create Result.make_empty

            from
                  i := 1
            until
                  i > regexp.count
            loop
                  c := regexp.at (i)
                  inspect c
                  when   ' '..'%'', ',', '-', '/'..'>', '@'..'[', ']'..'{',
'}', '~' then     -- readable except ()*+.?\|
                        Result.append_character (c)
                  when '(', '*', '+', '?' then
                        st.force (c)
                  when ')' then
                        from
                        until
                              st.count = 0 or else st.item = '('
                        loop
                              Result.append_character (st.item)
                              st.remove
                        end
```

```
                              if (st.count /= 0) then
                                    st.remove
                              else
                                    raise_exc ("matching ( is not found")
                              end
                        when '|' then
                              from
                              until
                                    st.count = 0 or else (st.item = '(' or st.item
= '|')
                              loop
                                    Result.append_character (st.item)
                                    st.remove
                              end
                              st.force (c)
                        when '\' then
                              Result.append_character (c)
                              Result.append_character (regexp.at (i + 1))
                              i := i + 1
                        when '.' then
                              from
                              until
                                    st.count = 0 or else (st.item = '(' or st.item
= '|' or st.item = '.')
                              loop
                                    Result.append_character (st.item)
                                    st.remove
                              end
                              st.force (c)
                        end
                        i := i + 1
                  end

                  from
                  until
                        st.count = 0
                  loop
                        if  st.item /= '(' then
                              Result.append_character (st.item)
                              st.remove
                        else
                              raise_exc ("matching ) is not found")
                        end
                  end

            rescue
                  handle_exc
            end

      prepare (regexp: STRING): STRING
            local
                  i, j: INTEGER
                  c, last_char: CHARACTER
            do
                  create Result.make_empty

                  from
                        i := 1
                  until
                        i > regexp.count
                  loop
                        c := regexp.at (i)
                        inspect c
                              when    '  '..'%'',  ',',  '-',  '/'..'>',  '@'..'Z',
'^'..'{', '}', '~' then -- readable except ()+*.?[\]|
                                    Result.append_character(c)
```

```
                                    if i < regexp.count and then conc_character
(regexp.at (i + 1)) then
                                          Result.append_character ('.')
                                    end

                              when '.' then
                                    Result.append_character ('\')
                                    Result.append_character (c)

                                    if i < regexp.count and then conc_character
(regexp.at (i + 1)) then
                                          Result.append_character ('.')
                                    end

                              when '\' then     -- \ and ()|*+[]\?
                                    if i < regexp.count and then escape_character
(regexp.at (i + 1)) then
                                          Result.append_character(c)
                                          Result.append_character (regexp.at (i +
1))
                                          i := i + 1

                                          if i < regexp.count and then
conc_character (regexp.at (i + 1)) then
                                                Result.append_character ('.')
                                          end
                                    else
                                          raise_exc ("invalid escape sequence")
                                    end

                              when '(', '|' then
                                    Result.append_character(c)

                              when ')', '+', '*', '?' then
                                    Result.append_character(c)

                                    if i < regexp.count and then conc_character
(regexp.at (i + 1)) then
                                          Result.append_character ('.')
                                    end

                              when ']' then
                                    raise_exc ("matching [ is not found")

                              when '[' then

                                    last_char := '%/0/'
                                    Result.append_character ('(')

                                    from
                                          i := i + 1
                                    until
                                          i > regexp.count or else regexp.at (i) =
']'
                                    loop
                                          c := regexp.at (i)

                                          inspect c
                                                when '\' then
                                                      if i < regexp.count and then
class_escape_character (regexp.at (i + 1)) then
                                                            if regexp.at (i + 1) =
'\' then

      Result.append_character ('\')
```

```
                                                        end
                                                        Result.append_character
(regexp.at (i + 1))
                                                        Result.append_character
('|')
                                                        last_char  :=  regexp.at
(i + 1)
                                                        i := i + 1
                                                  else
                                                        raise_exc     ("invalid
escape sequence inside symbol class")
                                                  end

                                              when '-' then
                                                        if  last_char  /=  '%/0/'  and
then (i + 1 < regexp.count and then (regexp.at (i + 1) /= ']' and regexp.at (i +
1) /= '-')) then
                                                        if regexp.at (i + 1) =
'\' then
                                                              if
class_escape_character (regexp.at (i + 2)) then
                                                                    i := i + 2
                                                              else
                                                                    raise_exc
("unknown escape sequence inside symbol class")
                                                              end
                                                        else
                                                              if regexp.at (i +
1) >= ' ' and regexp.at (i + 1) <= '~' then
                                                                    i := i + 1
                                                              else
                                                                    raise_exc
("invalid symbol inside symbol class")
                                                              end
                                                        end

                                                        if   last_char.code   <=
regexp.at (i).code then
                                                              from
                                                                    j         :=
last_char.code + 1
                                                              until
                                                                    j          >
regexp.at (i).code
                                                              loop
                                                                    c         :=
j.to_character_8
                                                                    if   c = '|'
or c = '(' or c = ')' or c = '*' or c = '+' or c = '.' or c = '?' or c = '\' then
      Result.append_character ('\')
                                                                    end

      Result.append_character (c)

      Result.append_character ('|')
                                                                    j := j + 1
                                                              end
                                                        else
                                                              raise_exc
("malformed symbol range")
                                                        end
                                                        last_char := '%/0/'
                                                  else
                                                        raise_exc    ("malformed
symbol range")
```

```
                                                end

                                          else
                                                if c >= ' ' and c <= '~' then
                                                      --   check   if   escape
sequence is needed
                                                      if   c = '|' or c = '('
or c = ')' or c = '*' or c = '+' or c = '.' or c = '?' or c = '\' then

      Result.append_character ('\')
                                                      end

                                                      Result.append_character
(c)
                                                      Result.append_character
('|')
                                                      last_char := c
                                                else
                                                      raise_exc      ("invalid
symbol inside symbol class")
                                                end
                                          end

                                    i := i + 1
                              end

                              if i <= regexp.count then
                                    if Result.at (Result.count) = '|' then
                                          Result.remove (Result.count)
                                    end

                                    Result.append_character (')')

                                    if    i    <    regexp.count    and    then
conc_character (regexp.at (i + 1)) then
                                          Result.append_character ('.')
                                    end
                              else
                                    raise_exc ("matching ] is not found")
                              end

                        else
                              raise_exc ("invalid symbol")
                  end
                  i := i + 1
            end

      rescue
            handle_exc
      end

class_escape_character (c: CHARACTER): BOOLEAN
      do
            inspect c
            when  '\', '-', ']' then
                  Result := TRUE
            else
                  Result := FALSE
            end
      end

escape_character (c: CHARACTER): BOOLEAN
      do
            inspect c
            when '(', ')', '|', '*', '+', '[', ']', '\', '?' then
                  Result := TRUE
```

```
                else
                        Result := FALSE
                end
        end

    conc_character (c: CHARACTER): BOOLEAN
        do
                inspect c
                when  ' '..'(', ','..'>', '@'..'\', '^'..'{', '}', '~' then
                        Result := TRUE
                else
                        Result := FALSE
                end
        end
end
```

DFA.e

```
class
    DFA

create
    make

feature
    make (nf: NFA)
        local
                rt: HASH_TABLE [STATE, STATES]
                tokens: LINKED_SET [STRING]
                qs: DS_LINKED_QUEUE [STATES]
                finls: LINKED_SET [STATE]
                ls: LINKED_SET [STATE]

                newinit, newstate, st: STATE
                newsts, sts: STATES
                c: CHARACTER

        do
                create ls.make
                ls.force (nf.get_init)

                create sts.make_set (nf.eps_closure (ls))
                create newinit.make (counter)
                create finls.make

                tokens := nf.get_tokens(sts)
                if not tokens.is_empty then
                        newinit.set_accept
                        newinit.add_tokens (tokens)
                        finls.force (newinit)
                end

                create rt.make (16)
                rt.force (newinit, sts)

                create qs.make
                qs.force (sts)

                from
                until
                        qs.is_empty
                loop
                        sts := qs.item
                        qs.remove
```

```
                  st := rt @ sts
                  from
                        c := ' '
                  until
                        c > '~'
                  loop
                        ls.wipe_out

                        from
                              sts.start
                        until
                              sts.off
                        loop
                              if sts.item.get_transition (c) /= Void then
                                    ls.append (sts.item.get_transition (c))
                              end
                              sts.forth
                        end

                        if not ls.is_empty then
                              create newsts.make_set (nf.eps_closure(ls))
                              if rt.has (newsts) then
                                    st.set_transition (rt @ newsts, c)
                              else
                                    create newstate.make (counter)
                                    tokens := nf.get_tokens (newsts)
                                    if not tokens.is_empty then
                                          newstate.set_accept
                                          newstate.add_tokens (tokens)
                                          finls.force (newstate)
                                    end
                                    rt.force (newstate, newsts)
                                    st.set_transition (newstate, c)
                                    qs.force (newsts)
                              end
                        end

                        c := c + 1
                  end
            end

            create mystates.make
            from
                  rt.start
            until
                  rt.off
            loop
                  mystates.force (rt.item_for_iteration)
                  rt.forth
            end

            init := newinit
            fins := finls
      end

get_init: STATE
      do
            Result := init
      end

get_fins: LINKED_SET [STATE]
      do
            Result := fins
      end

get_transitions_table: ARRAY [ARRAY [INTEGER]]
```

```
            local
                  c: CHARACTER
                  to: INTEGER
                  row: ARRAY [INTEGER]
            do
                  create Result.make (1, mystates.count)

                  from
                        mystates.start
                  until
                        mystates.off
                  loop
                        create row.make (1, 95)
                        Result.put (row, mystates.item_for_iteration.get_id)

                        from
                              c := ' '
                        until
                              c > '~'
                        loop
                              if mystates.item_for_iteration.get_transition (c) /=
Void then
                                    mystates.item_for_iteration.get_transition
(c).start
                                    to                                          :=
mystates.item_for_iteration.get_transition (c).item_for_iteration.get_id

                                    row.put (to, c.code - 31)
                              else
                                    row.put (0, c.code - 31)
                              end
                              c := c + 1
                        end
                        mystates.forth
                  end
            end

      get_tokens_table: ARRAY [ARRAY [STRING]]
            local
                  tokens: ARRAY [STRING]
                  i: INTEGER
            do
                  create Result.make (1, mystates.count)
                  from
                        mystates.start
                  until
                        mystates.off
                  loop
                        if mystates.item_for_iteration.get_tokens.count = 0 then

                              Result.put(Void, mystates.item_for_iteration.get_id)
                        else
                              create                tokens.make                (1,
mystates.item_for_iteration.get_tokens.count)
                              i := 1
                              from
                                    mystates.item_for_iteration.get_tokens.start
                              until
                                    mystates.item_for_iteration.get_tokens.off
                              loop
                                    tokens.put
(mystates.item_for_iteration.get_tokens.item_for_iteration, i)
                                    i := i + 1
                                    mystates.item_for_iteration.get_tokens.forth
                              end
```

```
                        Result.put(tokens,
mystates.item_for_iteration.get_id)
                  end
                  mystates.forth
            end
      end

      minimize
            local
                  qt: HASH_TABLE [LINKED_SET [CHARACTER], STATES]
                  block, fblock, bblock, minblock: STATES
                  newblocks, blocks: LINKED_SET [STATES]
                  ls: LINKED_SET [CHARACTER]
                  lst: LINKED_SET [STATE]
                  c: CHARACTER

                  initblock: STATES
                  finblocks: LINKED_SET [STATES]
                  minht: HASH_TABLE [LINKED_SET [STATE], STATES]
                  st: LINKED_SET [STATE]
            do
                  create fblock.make
                  create bblock.make

                  from
                        mystates.start
                  until
                        mystates.off
                  loop
                        if mystates.item.accept then
                              fblock.put (mystates.item)
                        else
                              bblock.put (mystates.item)
                        end
                        mystates.forth
                  end

                  create blocks.make
                  if fblock.count /= mystates.count and fblock.count /= 0
then
                        blocks.put (fblock)
                        blocks.put (bblock)
                        if fblock.count < bblock.count then
                              minblock := fblock
                        else
                              minblock := bblock
                        end
                  else
                        fblock := create {STATES}.make_set (mystates)
                        blocks.put (fblock)
                        minblock := fblock
                  end


                  create qt.make (256)
                  create ls.make
                  qt.put (ls, minblock)
                  from
                        c := ' '
                  until
                        c > '~'
                  loop
                        ls.put (c)
                        c := c + 1
                  end
```

```
                        from
                        until
                              qt.is_empty
                        loop
                              qt.start
                              block := qt.key_for_iteration
                              ls := qt.item_for_iteration
                              ls.start
                              c := ls.item
                              ls.remove
                              if ls.count = 0 then
                                    qt.remove (block)
                              end

                              create newblocks.make
                              from
                                    blocks.start
                              until
                                    blocks.off
                              loop
                                    create bblock.make
                                    create fblock.make
                                    from
                                          blocks.item.start
                                    until
                                          blocks.item.off
                                    loop
                                          if blocks.item.item.get_transition (c) /=
Void then
                                                lst                                   :=
blocks.item.item.get_transition (c)
                                                lst.start
                                                if block.has (lst.item) then
                                                      fblock.put (blocks.item.item)
                                                else
                                                      bblock.put (blocks.item.item)
                                                end
                                          else
                                                bblock.put (blocks.item.item)
                                          end
                                          blocks.item.forth
                                    end

                                    if   bblock.count   /=   blocks.item.count   and
bblock.count /= 0 then
                                          newblocks.put (fblock)
                                          newblocks.put (bblock)

                                          if fblock.count < bblock.count then
                                                minblock := fblock
                                          else
                                                minblock := bblock
                                          end

                                          from
                                                c := ' '
                                          until
                                                c > '~'
                                          loop
                                                if   qt.has_key   (blocks.item)   and
then qt.at (blocks.item).has (c) then
                                                      qt.at (blocks.item).prune (c)
                                                      if  qt.at  (blocks.item).count
= 0 then
                                                            qt.remove (blocks.item)
                                                      end
```

```
                                                qt.put   (create   {LINKED_SET
[CHARACTER]}.make, fblock)
                                                qt.at (fblock).put (c)
                                                qt.put   (create   {LINKED_SET
[CHARACTER]}.make, bblock)
                                                qt.at (bblock).put (c)
                                          else
                                                if qt.has_key (minblock) then
                                                      qt.at     (minblock).put
(c)
                                                else
                                                      qt.put           (create
{LINKED_SET [CHARACTER]}.make, minblock)
                                                      qt.at     (minblock).put
(c)
                                                end
                                          end
                                          c := c + 1
                                    end
                              else
                                    newblocks.put (blocks.item)
                              end
                              blocks.forth
                        end
                        blocks := newblocks
                  end

                  ----------------------------

                  create finblocks.make
                  from
                        blocks.start
                  until
                        blocks.off
                  loop
                        if blocks.item.has (init) then
                              initblock := blocks.item
                        end
                        blocks.item.start
                        if fins.has (blocks.item.item) then
                              finblocks.put (blocks.item)
                        end
                        blocks.forth
                  end

                  create mystates.make
                  create minht.make (64)
                  st := merge(initblock)
                  minht.put (st, initblock)
                  mystates.append (st)
                  blocks.prune (initblock)

                  from
                        blocks.start
                  until
                        blocks.off
                  loop
                        st := merge(blocks.item)
                        minht.put (st, blocks.item)
                        mystates.append (st)
                        blocks.forth
                  end

                  minht.at (initblock).start
                  init := minht.at (initblock).item
```

```
                        create fins.make
                        from
                              finblocks.start
                        until
                              finblocks.off
                        loop
                              fins.append (minht.at (finblocks.item))
                              finblocks.forth
                        end
                  end

      merge (block: STATES): LINKED_SET [STATE]
            local
                  myblock: STATES
                  c: CHARACTER
                  ls: LINKED_SET [STATE]
                  newstate: STATE
                  skip, has: BOOLEAN
                  delstates: LINKED_SET [STATE]
            do
                  create Result.make
                  create myblock.make
                  create delstates.make

                  from
                        block.start
                  until
                        block.off
                  loop
                        myblock.put (block.item)
                        block.forth
                  end

                  from
                  until
                        myblock.is_empty
                  loop
                        delstates.wipe_out
                        create newstate.make (mincounter)
                        from
                              myblock.start
                        until
                              myblock.off
                        loop
                              skip := false
                              if myblock.item.accept then
                                    if newstate.accept then
                                          if    myblock.item.get_tokens.count    /=
newstate.get_tokens.count then
                                                skip := true
                                          else
                                                from
                                                      myblock.item.get_tokens.start
                                                until
                                                      myblock.item.get_tokens.off
                                                loop
                                                      has := false
                                                      from

      newstate.get_tokens.start
                                                until
                                                      newstate.get_tokens.off
                                                loop
                                                      if
newstate.get_tokens.item.is_equal (myblock.item.get_tokens.item) then
                                                            has := true
```

```
                                                      end

      newstate.get_tokens.forth
                                                end
                                                if not has then
                                                      skip := true
                                                end
                                                myblock.item.get_tokens.forth
                                          end
                                    end
                              else
                                    newstate.set_accept
                                    newstate.add_tokens
(myblock.item.get_tokens)
                              end
                        end

                        if not skip then
                              delstates.put (myblock.item)
                              from
                                    myblock.item.links.start
                              until
                                    myblock.item.links.off
                              loop
                                    c := myblock.item.links.key_for_iteration
                                    ls                                          :=
myblock.item.links.item_for_iteration

                                    from
                                          ls.start
                                    until
                                          ls.off
                                    loop
                                          ls.item.inlinks.at         (c).prune
(myblock.item)
                                          newstate.set_transition    (ls.item,
c)
                                          ls.forth
                                    end
                                    myblock.item.links.forth
                              end

                              from
                                    myblock.item.inlinks.start
                              until
                                    myblock.item.inlinks.off
                              loop
                                    c                                           :=
myblock.item.inlinks.key_for_iteration
                                    ls                                          :=
myblock.item.inlinks.item_for_iteration

                                    from
                                          ls.start
                                    until
                                          ls.off
                                    loop
                                          ls.item.links.at           (c).prune
(myblock.item)
                                          ls.item.set_transition     (newstate,
c)
                                          ls.forth
                                    end
                                    myblock.item.inlinks.forth
                              end
                        end
```

```
                        myblock.forth
                  end

                  from
                        delstates.start
                  until
                        delstates.off
                  loop
                        myblock.prune(delstates.item)
                        delstates.forth
                  end
                  Result.put (newstate)
            end
      end


      describe
            do
                  io.put_string ("init: ")
                  io.put_integer (init.get_id)
                  io.put_string (" fins: ")
                  from
                        fins.start
                  until
                        fins.off
                  loop
                        io.put_integer (fins.item.get_id)
                        io.put_string (" ")
                        fins.forth
                  end
                  io.put_new_line

                  create described_states.make
                  describe_states (init)
            end

feature {NONE}

      init: STATE
      fins: LINKED_SET [STATE]
      mystates: LINKED_SET [STATE]

      counter: ID_COUNTER
            once
                  create Result.make
            end

      mincounter: ID_COUNTER
            once
                  create Result.make
            end

      described_states: LINKED_SET [STATE]

      describe_states (st: STATE)
            local
                  s: LINKED_SET [STATE]
            do
                  describe_state (st)
                  described_states.put (st)
                  s := get_states (st)

                  from
                        s.start
                  until
                        s.off
```

```
            loop
                  if not described_states.has (s.item) then
                        describe_states (s.item)
                  end

                  s.forth
            end
      end

get_states (st: STATE): LINKED_SET [STATE]
      local
            c: CHARACTER
      do
            create Result.make

            from
                  c := ' '
            until
                  c > '~'
            loop
                  if st.get_transition (c) /= Void then
                        Result.append (st.get_transition (c))
                  end

                  c := c + 1
            end
      end

describe_state (st: STATE)
      local
            s: LINKED_SET [STATE]
            c: CHARACTER
      do
            io.put_string ("state: ")
            io.put_integer (st.get_id)
            io.put_string (" accept: ")
            io.put_boolean (st.is_accept)
            io.put_new_line

            if not st.get_tokens.is_empty then
                  io.put_string ("tokens: ")
                  from
                        st.get_tokens.start
                  until
                        st.get_tokens.off
                  loop
                        io.put_string (st.get_tokens.item)
                        io.put_string (" ")
                        st.get_tokens.forth
                  end
                  io.put_new_line
            end

            io.put_string ("  children: %N")

            from
                  c := ' '
            until
                  c > '~'
            loop
                  s := st.get_transition (c)

                  if s /= Void then
                        io.put_string ("        '")
                        io.put_character (c)
                        io.put_string ("': ")
```

```
                        from
                              s.start
                        until
                              s.off
                        loop
                              io.put_integer (s.item.get_id)
                              io.put_string (" ")
                              s.forth
                        end
                        io.put_new_line
                  end
                  c := c + 1
            end
            io.put_new_line
      end
end
```

NFA.e

```
class
      NFA

create
      make_character, make_star, make_plus, make_quest, make_concat, make_or,
make_final

feature

      make_character (c: CHARACTER)
            do
                  create init.make (counter)
                  create fin.make (counter)
                  fin.set_accept
                  init.set_transition (fin, c)
            end

      make_star (fa: NFA)
            local
                  lsi, lsf: LINKED_SET [STATE]
            do
                  create lsi.make
                  lsi.force (fa.get_init)
                  create lsf.make
                  lsf.force (fa.get_fin)

                  if eps_closure (lsi).has (fa.get_fin) and then eps_closure
(lsf).has (fa.get_init) then
                        init := fa.get_init
                        fin := fa.get_fin
                  elseif not fa.get_fin.has_outcome_links then
                        init := fa.get_fin
                        fin := fa.get_fin
                        init.set_transition (fa.get_init, epsilon)
                  else
                        create init.make (counter)
                        fin := init
                        fin.set_accept
                        fa.get_fin.unset_accept
                        fa.get_fin.set_transition (fin, epsilon)
                        init.set_transition (fa.get_init, epsilon)
                  end
            end

      make_plus (fa: NFA)
```

```
            local
                  ls: LINKED_SET [STATE]
            do
                  init := fa.get_init
                  fin := fa.get_fin

                  create ls.make
                  ls.force (fa.get_fin)
                  if not eps_closure (ls).has (fa.get_init) then
                        fin.set_transition (init, epsilon)
                  end
            end

      make_quest (fa: NFA)
            local
                  ls: LINKED_SET [STATE]
            do
                  create ls.make
                  ls.force (fa.get_init)
                  if not eps_closure (ls).has (fa.get_fin) then
                        if fa.get_init.has_income_links then
                              create init.make (counter)
                              init.set_transition (fa.get_init, epsilon)
                        else
                              init := fa.get_init
                        end

                        if fa.get_fin.has_outcome_links then
                              create fin.make (counter)
                              fa.get_fin.unset_accept
                              fin.set_accept
                              fa.get_fin.set_transition (fin, epsilon)
                        else
                              fin := fa.get_fin
                        end

                        init.set_transition (fin, epsilon)
                  else
                        init := fa.get_init
                        fin := fa.get_fin
                  end
            end

      make_concat (left: NFA; right: NFA)
            local
                  c: CHARACTER
                  ls: LINKED_SET [STATE]
                  st: STATE
            do
                  init := left.get_init
                  fin := right.get_fin
                  left.get_fin.unset_accept

                  if        not        (left.get_fin.has_outcome_links        and
right.get_init.has_income_links) then
                        from
                              left.get_fin.links.start
                        until
                              left.get_fin.links.off
                        loop
                              c := left.get_fin.links.key_for_iteration
                              ls := left.get_fin.links.item_for_iteration

                              from
                                    ls.start
                              until
```

```
                                    ls.off
                              loop
                                    (ls.item_for_iteration.inlinks    @    c).prune
(left.get_fin)
                                    ls.forth
                              end
                              right.get_init.set_transitions (ls, c)
                              left.get_fin.links.forth
                        end

                        from
                              left.get_fin.inlinks.start
                        until
                              left.get_fin.inlinks.off
                        loop
                              c := left.get_fin.inlinks.key_for_iteration
                              ls := left.get_fin.inlinks.item_for_iteration

                              from
                                    ls.start
                              until
                                    ls.off
                              loop
                                    st := ls.item_for_iteration ;

                                    (st.links @ c).prune (left.get_fin)
                                    st.set_transition (right.get_init, c)
                                    ls.forth
                              end
                              left.get_fin.inlinks.forth
                        end
                        if init = left.get_fin then
                              init := right.get_init
                        end
                  else
                        left.get_fin.set_transition(right.get_init, epsilon)
                  end
            end

      make_or (left: NFA; right: NFA)
            local
                  c: CHARACTER
                  ls: LINKED_SET [STATE]
                  st: STATE
            do
                  if     (not     left.get_init.has_income_links)     and     (not
right.get_init.has_income_links) then
                        from
                              left.get_init.links.start
                        until
                              left.get_init.links.off
                        loop
                              c := left.get_init.links.key_for_iteration
                              ls := left.get_init.links.item_for_iteration

                              from
                                    ls.start
                              until
                                    ls.off
                              loop
                                    (ls.item_for_iteration.inlinks    @    c).prune
(left.get_init)
                                    ls.forth
                              end
                              right.get_init.set_transitions (ls, c)
```

```
                        left.get_init.links.forth
                  end
                  init := right.get_init
            elseif not left.get_init.has_income_links then
                  init := left.get_init
                  init.set_transition (right.get_init, epsilon)
            elseif not right.get_init.has_income_links then
                  init := right.get_init
                  init.set_transition (left.get_init, epsilon)
            else
                  create init.make (counter)
                  init.set_transition (left.get_init, epsilon)
                  init.set_transition (right.get_init, epsilon)
            end

            if (not left.get_fin.has_outcome_links) and (not
right.get_fin.has_outcome_links) then
                  from
                        left.get_fin.inlinks.start
                  until
                        left.get_fin.inlinks.off
                  loop
                        c := left.get_fin.inlinks.key_for_iteration
                        ls := left.get_fin.inlinks.item_for_iteration

                        from
                              ls.start
                        until
                              ls.off
                        loop
                              st := ls.item_for_iteration ;

                              (st.links @ c).prune (left.get_fin)
                              st.set_transition (right.get_fin, c)
                              ls.forth
                        end
                        left.get_fin.inlinks.forth
                  end
                  fin := right.get_fin
            elseif not left.get_fin.has_outcome_links then
                  fin := left.get_fin
                  right.get_fin.unset_accept
                  right.get_fin.set_transition (fin, epsilon)
            elseif not right.get_fin.has_outcome_links then
                  fin := right.get_fin
                  left.get_fin.unset_accept
                  left.get_fin.set_transition (fin, epsilon)
            else
                  create fin.make (counter)
                  left.get_fin.unset_accept
                  left.get_fin.set_transition (fin, epsilon)
                  right.get_fin.unset_accept
                  right.get_fin.set_transition (fin, epsilon)
            end
      end

make_final (ht: HASH_TABLE [NFA, STRING])
      local
            cur_fa: NFA
      do
            create init.make (counter)
            create fin.make (counter)
            fin.set_accept

            from
                  ht.start
```

```
            until
                  ht.off
            loop
                  cur_fa := ht.item_for_iteration
                  cur_fa.get_fin.add_token (ht.key_for_iteration)
                  init.set_transition (cur_fa.get_init, epsilon)
                  cur_fa.get_fin.set_transition (fin, epsilon)

                  ht.forth
            end
      end

get_tokens (sts: STATES): LINKED_SET [STRING]
      do
            create Result.make

            from sts.start
            until sts.off
            loop
                  Result.append (sts.item.get_tokens)
                  sts.forth
            end
      end

get_init: STATE
      do
            Result := init
      end

get_fin: STATE
      do
            Result := fin
      end

eps_closure (sts: LINKED_SET [STATE]): LINKED_SET [STATE]
      local
            qs: LINKED_QUEUE [STATE]
            seen: LINKED_SET [STATE]
            cur: LINKED_SET [STATE]
            s: STATE
      do
            create qs.make
            create seen.make
            create Result.make

            qs.append (sts)
            seen.append (sts)

            from
            until
                  qs.is_empty
            loop
                  s := qs.item
                  qs.remove
                  Result.force (s)
                  seen.force (s)

                  if s.get_transition (epsilon) /= Void then
                        cur := s.get_transition (epsilon)
                        from cur.start
                        until cur.off
                        loop
                              if not seen.has (cur.item_for_iteration) then
                                    qs.force (cur.item_for_iteration)
                              end
                              cur.forth
```

```
                              end
                        end
                  end
            end

feature {NONE}
      counter: ID_COUNTER
            once
                  create Result.make
            end

      init: STATE
      fin: STATE

      epsilon: CHARACTER is '%/0/'
end
```

STATE.e

```
class
      STATE

inherit
      HASHABLE
            redefine
                  is_equal
            end

create
      make

feature
      make (counter: ID_COUNTER)
            do
                  id := counter.next_id
                  create links.make (64)
                  create inlinks.make (64)
                  create tokens.make
                  unset_accept
            end

      is_accept: BOOLEAN
            do
                  Result := accept
            end

      set_accept
            do
                  accept := true
            end

      unset_accept
            do
                  accept := false
            end

      set_transition (st: STATE; c: CHARACTER)
            local
                  ls: LINKED_SET [STATE]
            do
                  if links.has (c) then
                        ls := links @ c
                        ls.force (st)
                  else
                        create ls.make
                        ls.force (st)
                        links.force (ls, c)
```

```
            end

            if st.inlinks.has (c) then
                  ls := st.inlinks @ c
                  ls.force (current)
            else
                  create ls.make
                  ls.force (current)
                  st.inlinks.force (ls, c)
            end
      end

set_transitions (ls: LINKED_SET [STATE]; c: CHARACTER)
      do
            from
                  ls.start
            until
                  ls.off
            loop
                  set_transition (ls.item_for_iteration, c)
                  ls.forth
            end
      end

get_transition (c: CHARACTER): LINKED_SET[STATE]
      do
            if links.has (c) then
                  Result := links @ c
            else
                  Result := Void
            end
      end

get_income_links (c: CHARACTER): LINKED_SET[STATE]
      do
            if links.has (c) then
                  Result := inlinks @ c
            else
                  Result := Void
            end
      end

has_income_links: BOOLEAN
      do
            Result := not inlinks.is_empty
      end

has_outcome_links: BOOLEAN
      do
            Result := not links.is_empty
      end

get_id: INTEGER
      do
            Result := id
      end

add_token (token: STRING)
      do
            tokens.force (token)
      end

add_tokens (tks: LINKED_SET [STRING])
      do
            tokens.append (tks)
      end
```

```
	get_tokens: LINKED_SET [STRING]
		do
			Result := tokens
		end


	links: HASH_TABLE [LINKED_SET [STATE], CHARACTER]
	inlinks: HASH_TABLE [LINKED_SET [STATE], CHARACTER]

	tokens: LINKED_SET [STRING]

	accept: BOOLEAN
	id: INTEGER

feature
	hash_code: INTEGER
		do
			Result := get_id
		end

	is_equal (other: STATE): BOOLEAN
		do
			Result := get_id = other.get_id
		end
end
```

STATES.e

```
class
	STATES

inherit
	HASHABLE
		redefine
			is_equal
		end

create
	make, make_set

feature
	make
		do
			create sts.make
		end

	make_set (ls: LINKED_SET [STATE])
		do
			create sts.make
			sts.copy (ls)
		end

	force (st: STATE)
		do
			sts.force (st)
		end

	put (st: STATE)
		do
			sts.force (st)
		end

	count: INTEGER
		do
```

```
                Result := sts.count
        end

    wipe_out
        do
                sts.wipe_out
        end

    has (st: STATE): BOOLEAN
        do
                Result := sts.has (st)
        end

    is_empty: BOOLEAN
        do
                Result := sts.is_empty
        end

    prune (st: STATE)
        do
                sts.prune (st)
        end

feature
    hash_code: INTEGER
        do
                Result := 1
                from
                        sts.start
                until
                        sts.off
                loop
                        Result := Result * sts.item_for_iteration.get_id \\ 100000
                        sts.forth
                end
        end

    is_equal (other: STATES): BOOLEAN
        local
                ids: LINKED_SET [INTEGER]
        do
                if other.count = count then
                        Result := true

                        create ids.make
                        from
                                other.start
                        until
                                other.off
                        loop
                                ids.force (other.item.get_id)
                                other.forth
                        end

                        from
                                start
                        until
                                off
                        loop
                                if not ids.has (item.get_id) then
                                        Result := false
                                end
                                forth
                        end

                else
```

```
                        Result := false
                end
        end

feature
      start
            do
                  sts.start
            end

      off: BOOLEAN
            do
                  Result := sts.off
            end

      forth
            do
                  sts.forth
            end

      item: STATE
            do
                  Result := sts.item_for_iteration
            end

feature {NONE}
      sts: LINKED_SET [STATE]
end
```

PARSER_EXCEPTION.e

```
class PARSER_EXCEPTION

inherit
    DEVELOPER_EXCEPTION

create
      make

feature
      make (msg: STRING)
            do
                  set_message (msg)
            end
end
```

ID_COUNTER.e

```
class
      ID_COUNTER

create
      make

feature

      make
            do
                  count := 1
            end

      next_id: INTEGER
            do
                  Result := count
                  count := count + 1
```

```
            end

feature {NONE}

      count: INTEGER
end
```