|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Информатика и системы управления\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

КАФЕДРА \_\_\_\_\_\_\_\_Программное обеспечение ЭВМ и информационные технологии\_\_\_\_\_\_\_\_\_

**ОТЧЕТ**

**по лабораторной работе №1**

**по курсу «Проектирование компиляторов»**

**Тема «Распознавание цепочек регулярного языка»**

**Вариант 4**

Студент \_\_\_ИУ7-23М\_\_\_\_ \_\_\_\_\_Н.И. Иксарица\_\_\_\_

(группа) (И.О.Фамилия)

Преподаватель \_\_\_\_А.А. Ступников\_\_\_\_

(И.О.Фамилия)

*2021 г.*

**Цель работы**: приобретение практических навыков реализации важнейших элементов лексических анализаторов на примере распознавания цепочек регулярного языка.

**Задачи**

Задачи работы:

1. Ознакомиться с основными понятиями и определениями, лежащими в основе построения лексических анализаторов.
2. Прояснить связь между регулярным множеством, регулярным выражением, праволинейным языком, конечно-автоматным языком и недетерминированным конечно-автоматным языком.
3. Разработать, тестировать и отладить программу распознавания цепочек регулярного или праволинейного языка в соответствии с предложенным вариантом грамматики.

**Вариант 4**

Необходимо разработать программу, которая в качестве входа принимает произвольное регулярное выражение, и выполняет следующие преобразования:

1. По регулярному выражению строит недетерминированный конечный автомат (НКА).
2. По полученному НКА строит эквивалентный ему детерминированный конечный автомат (ДКА).
3. По ДКА строит эквивалентный ему конечный автомат, имеющий наименьшее возможное количество состояний (с использованием алгоритма Бржозовского).
4. Моделирует минимальный конечный автомат для входной цепочки из терминалов исходной грамматики.

**Листинги кода**

Листинг 1. Построение недетерминированного конечного автомата путём разбора строки регулярного выражения.

|  |
| --- |
| (ns cc.lab01.regex-parser    (:require [cc.lab01.expressions :as ex]))  (declare parse           parse-alt           parse-succ           parse-unary           parse-brackets           parse-sym           parse-null)  (def debug? true)  (def unary-ops (set "+\*?"))  (def binary-ops (set "|"))  (defn- balanced?    "Если круглые скобки рассталены правильно, возвращает true, иначе false."    [input]    (empty? (reduce (fn [stack c] (case c                                    \( (conj stack c)                                    \) (if (seq stack)                                         (pop stack)                                         (reduced [:UNDERFLOW]))                                    stack))                    '()                    input)))  (defn- get-depth-vec    "Возвращает вектор, в котором каждому символу из входной строки s соответствует число,     характеризующее текущую глубину вложенности круглых скобок.     Пример: \"(ab()))(\" -> [1 1 1 2 1 0 -1 0]"    [s]    (->> s         (map #(case % \( 1 \) -1 0))         (reductions +)         (vec)))  (defn- indexes-of    "Возвращает индексы символа value (символ или строка из одного символа) в s. Если value не     найдено, будет возвращена пустая коллекция."    ([s value]     (->> s          (map-indexed (fn [index c]                         (when (= (str c) (str value))                           index)))          (filter some?)))    ([s v1 & values]     (->> (conj values v1)          (set)          (map #(indexes-of s %))          (apply concat)          (sort))))  (defn parse [input]    (when (not (balanced? input))      (throw (Exception. "Не удалось построить КА по введённой строке.")))    (if-let [graph (parse-alt input)]      graph      (if-let [graph (parse-succ input)]        graph        (if-let [graph (parse-unary input)]          graph          (if-let [graph (parse-brackets input)]            graph            (if-let [graph (parse-sym input)]              graph              (if-let [graph (parse-null input)]                graph                (throw (Exception. "Не удалось построить КА по введённой строке.")))))))))  (defn- parse-alt [input]    (when (> (count input) 2)      (let [depth-vec (get-depth-vec input)            indexes (indexes-of input "|")]        (->> indexes             (filter #(zero? (depth-vec %)))             (first)             ((fn [index]                (when (some? index)                  (ex/alt (parse (subs input 0 index))                          (parse (subs input (inc index)))))))))))  (defn- parse-succ [input]    (when (> (count input) 1)      (let [depth-vec (get-depth-vec input)]        (->> input             (partition 2 1)             (map-indexed #(vector % %2))             (filter (fn [[i [a b]]] (and (not (binary-ops a))                                          (not (binary-ops b))                                          (not (unary-ops b))                                          (zero? (depth-vec i)))))             (first)             (first)             ((fn [index]                (when (some? index)                  (ex/succ (parse (subs input 0 (inc index)))                           (parse (subs input (inc index)))))))))))  (defn- parse-unary [input]    (when (> (count input) 1)      (when-let [sym (last input)]        (when-let [\_ (unary-ops sym)]          (when-let [\_ (zero? (last (get-depth-vec input)))]            (let [substr (apply str (butlast input))]              (case sym                \\* (ex/star (parse substr))                \+ (ex/plus (parse substr))                \? (ex/opt (parse substr))                nil)))))))  (defn- parse-brackets [input]    (when (> (count input) 1)      (when-let [\_ (balanced? input)]        (when-let [\_ (and (= (first input) \()                          (= (last input) \)))]          (parse (apply str (butlast (rest input))))))))  (defn- parse-sym [input]    (when (= (count input) 1)      (when-let [\_ (not (unary-ops (first input)))]        (when-let [\_ (not (binary-ops (first input)))]          (ex/sym input)))))  (defn- parse-null [input]    (when (empty? input)      (ex/null))) |

Листинг 2. Функции для создания НКА при помощи конструктора Томпсона.

|  |
| --- |
| (ns cc.lab01.expressions    (:require [clojure.set]))  (declare null)  (declare sym)  (declare alt)  (declare succ)  (declare star)  (declare plus)  (declare opt)  (declare rng)  (defn null    "Возвращает НКА, который принимает только пустую последовательность."    []    (let [start (gensym "s")          finish (gensym "s")]      {:graph-type :NFA       :start #{start}       :finish #{finish}       :alphabet #{}       :states #{start finish}       :transitions {start {:null #{finish}}}}))  (defn sym    "Возвращает НКА, который принимает только s, где s может быть строкой или одним символом."    [s]    (let [start (gensym "s")          finish (gensym "s")]      {:graph-type :NFA       :start #{start}       :finish #{finish}       :alphabet #{(str s)}       :states #{start finish}       :transitions {start {(str s) #{finish}}}}))  (defn simplify [graph]    (when (not= (count (:start graph)) 1)      (throw (Exception.              (str "Число начальных состояний в конструкторе Томпсона должно быть равно единице."))))    (when (not= (count (:finish graph)) 1)      (throw (Exception.              (str "Число конечных состояний в конструкторе Томпсона должно быть равно единице."))))    (let [graph (update graph :start first)          graph (update graph :finish first)]      graph))  (defn alt    "Возвращает НКА, который принимает всё, что принимается любым из графов graphs. Ведет себя как     (null) если подграфы не заданы."    ([] (null))    ([& graphs]     (let [graphs (map simplify graphs)           start (gensym "s")           finish (gensym "s")           alphabet (apply clojure.set/union (map :alphabet graphs))           states (apply clojure.set/union (map :states graphs))           states (into #{start finish} states)           transitions (apply merge (map :transitions graphs))           transitions (reduce (fn [transitions graph]                                 (update-in transitions                                            [start :null]                                            (comp set conj) (:start graph)))                               transitions                               graphs)           transitions (reduce (fn [transitions graph]                                 (update-in transitions                                            [(:finish graph) :null]                                            (comp set conj) finish))                               transitions                               graphs)]       {:graph-type :NFA        :start #{start}        :finish #{finish}        :alphabet alphabet        :states states        :transitions transitions})))  (defn succ    "Возвращает НКА, который принимает всё, что принимается графами graphs выстроенными     последовательно. Ведет себя как (null), если подграфы не заданы."    ([] (null))    ([& graphs]     (let [graphs (map simplify graphs)           start (gensym "s")           finish (gensym "s")           alphabet (apply clojure.set/union (map :alphabet graphs))           states (apply clojure.set/union (map :states graphs))           states (into #{start finish} states)           transitions (apply merge (map :transitions graphs))           pairs (partition 2 1 graphs)           transitions (reduce (fn [transitions pair]                                 (let [f (:finish (first pair))                                       s (:start (second pair))]                                   (update-in transitions                                              [f :null]                                              (comp set conj) s)))                               transitions                               pairs)           transitions (update-in transitions                                  [start :null]                                  (comp set conj) (:start (first graphs)))           transitions (update-in transitions                                  [(:finish (last graphs)) :null]                                  (comp set conj) finish)]       {:graph-type :NFA        :start #{start}        :finish #{finish}        :alphabet alphabet        :states states        :transitions transitions})))  (defn star    "Возвращает НКА, который принимает все, что принимается графом НКА ноль или более раз."    [graph]    (let [graph (simplify graph)          start (gensym "s")          finish (gensym "s")          alphabet (:alphabet graph)          states (into #{start finish} (:states graph))          transitions (:transitions graph)          transitions (update-in transitions                                 [start :null]                                 (comp set conj) (:start graph))          transitions (update-in transitions                                 [(:finish graph) :null]                                 (comp set conj) finish)          transitions (update-in transitions                                 [start :null]                                 (comp set conj) finish)          transitions (update-in transitions                                 [(:finish graph) :null]                                 (comp set conj) (:start graph))]      {:graph-type :NFA       :start #{start}       :finish #{finish}       :alphabet alphabet       :states states       :transitions transitions}))  (defmacro plus    "Возвращает НКА, который принимает все, что принимается графом НКА один или несколько раз."    [graph]    `(succ ~graph           (star ~graph)))  (defmacro opt    "Возвращает НКА, который принимает все, что принимается графом НКА или пустой     последовательностью."    [graph]    `(alt (null)          ~graph))  (defmacro rng    "Возвращает НКА, который принимает все, что принимается графом НКА, последовательно     реплицированным от lower до upper (включительно) раз.     Если верхняя граница не зададна, то граф будет реплецирован ровно lower раз.     Если верхняя граница равна :inf, то НКА принимает все, что принимается графом, реплицированным по     крайней мере lower раз подряд."    ([graph n]     `(succ ~@(repeat n graph)))    ([graph lower upper]     (if (= upper :inf)       `(succ ~@(repeat lower graph) (star ~graph))       `(alt ~@(map #(apply list `succ (repeat % graph))                    (range lower (+ upper 1))))))) |

Листинг 3. Минимизация конечного автомата.

|  |
| --- |
| (ns cc.lab01.morph    (:require [clojure.set]))  (defn graph-reverse [graph]    (let [graph (assoc graph                       :start (:finish graph)                       :finish (:start graph))          triads (for [i (-> graph :transitions keys)                       j (-> graph :transitions (get i) keys)                       k (-> graph :transitions (get i) (get j))]                   [k j i])          r-transitions (reduce (fn [acc [k j i]]                                  (update-in acc [k j] (comp set conj) i))                                {}                                triads)]      (assoc graph :transitions r-transitions)))  (defn minimize    "Минимизация КА с использованием алгоритма Бржозовского."    [graph]    (-> graph        (graph-reverse)        (nfa->dfa)        (graph-reverse)        (nfa->dfa))) |

Листинг 4. Детерминизация НКА.

|  |
| --- |
| (ns cc.lab01.morph    (:require [clojure.set]))    (declare null-closure)  (declare alpha-null-closure)  (declare nfa->dfa)  (defn null-closure [states nfa]    (loop [transitions (:transitions nfa)           closure states           checkpoints states]      (let [next-checkpoints (apply concat                                    (map #(get-in transitions [% :null])                                         checkpoints))            next-closure (into closure next-checkpoints)]        (if (= closure next-closure)          closure          (recur transitions                 next-closure                 (set next-checkpoints))))))  (defn alpha-null-closure [states nfa c]    (let [transitions (:transitions nfa)          alpha-states (set (apply concat                                   (map #(get-in transitions [% c])                                        states)))]      (set (apply concat                  (map #(null-closure #{%} nfa)                       alpha-states)))))  (defn nfa->dfa [nfa]    (loop [queue (list (null-closure (:start nfa) nfa))           alphabet (:alphabet nfa)           dfa-states #{}           dfa-transitions {}]      (if (empty? queue)        {:graph-type :DFA         :start #{(null-closure (:start nfa) nfa)}         :finish (->> dfa-states                      (filter #(seq (clojure.set/intersection (:finish nfa) %)))                      (set))         :alphabet alphabet         :states dfa-states         :transitions dfa-transitions}        (let [state (peek queue)              queue (pop queue)]          (if (contains? dfa-states state)            (recur queue alphabet dfa-states dfa-transitions)            (let [entries (for [c alphabet]                            (list c (alpha-null-closure state nfa c)))                  entries (filter #(not (empty? (second %))) entries)                  queue (into queue (map second entries))                  dfa-states (conj dfa-states state)                  dfa-transitions (reduce #(assoc-in %                                                    [state (first %2)]                                                    #{(second %2)})                                         dfa-transitions                                         entries)]              (recur queue alphabet dfa-states dfa-transitions)))))))  (defn graph-reverse [graph]    (let [graph (assoc graph                       :start (:finish graph)                       :finish (:start graph))          triads (for [i (-> graph :transitions keys)                       j (-> graph :transitions (get i) keys)                       k (-> graph :transitions (get i) (get j))]                   [k j i])          r-transitions (reduce (fn [acc [k j i]]                                  (update-in acc [k j] (comp set conj) i))                                {}                                triads)]      (assoc graph :transitions r-transitions))) |

Листинг 5. Функция для проверки соответствия строки регулярному выражению, моделируемому построенным ДКА.

|  |
| --- |
| (ns cc.lab01.bypass)  (defn dfa-accepts? [dfa stream]    {:pre [(= (count (:start dfa)) 1)]}    (doseq [i (-> dfa :transitions keys)            j (-> dfa :transitions (get i) keys)]      (when (-> dfa :transitions (get i) (get j) (count) (not= 1))        (throw (Exception. "ДКА должен содержать только одно начальное состояние."))))    (loop [cur-state (first (:start dfa))           cur-stream stream]      (if (empty? cur-stream)        (-> cur-state ((:finish dfa)) (some?))        (let [token (str (first cur-stream))              next-state (first (get-in dfa [:transitions cur-state token]))              next-stream (rest cur-stream)]          (if (some? next-state)            (recur next-state next-stream)            false))))) |