|  |  |
| --- | --- |
| 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 №3**

**по курсу «Проектирование компиляторов»**

**Тема «Синтаксический разбор с использованием метода рекурсивного спуска»**

**Вариант 1**

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

(группа) (И.О.Фамилия)

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

(И.О.Фамилия)

*2021 г.*

**Задание**

Дополнить грамматику в стиле языка Си или Алгол-Паскаль блоком, состоящим из последовательности операторов присваивания. Объединить получившуюся грамматику с грамматикой из индивидуального варианта.

Для модифицированной грамматики написать программу нисходящего синтаксического анализа с использованием метода рекурсивного спуска.

**Подготовка грамматики**

Листинг 1. Грамматика из индивидуального варианта.

|  |
| --- |
| <выражение> ->      <простое выражение> |      <простое выражение> <операция отношения> <простое выражение>  <простое выражение> ->      <терм> |      <знак> <терм> |      <простое выражение> <операция типа сложения> <терм>  <терм> ->      <фактор> |      <терм> <операция типа умножения> <фактор>  <фактор> ->      <идентификатор> |      <константа> |      ( <простое выражение> ) |      not <фактор>  <операция отношения> ->      = | <> | < | <= | > | >=  <знак> ->      + | -  <операция типа сложения> ->      + | - | or  <операция типа умножения> ->      \* | / | div | mod | and |

Листинг 2. Грамматика, полученная в результате объединения грамматики в стиле языка Си и грамматики из индивидуального варианта, без левой рекурсии и с выполненной левой факторизацией.

|  |
| --- |
| {      "terms": [          "(", ")", "not", "==", "<>", "<", "<=", ">", ">=", "+", "-", "or", "\*",          "/", "div", "mod", "and", "{", "}", ";", "=", "константа", "идентификатор"      ],      "startSymbol": "программа",      "prods": {          "программа": [              ["блок"]          ],          "блок": [              ["{", "список операторов", "}"]          ],          "список операторов": [              ["оператор", "список операторов 1"]          ],          "список операторов 1": [              ["хвост"],              ["eps"]          ],          "хвост": [              [";", "оператор", "хвост 1"]          ],          "хвост 1": [              ["хвост"],              ["eps"]          ],          "оператор": [              ["идентификатор", "=", "выражение"],              ["{", "список операторов", "}"]          ],          "выражение": [              ["фактор", "выражение 1"],              ["знак", "тттерм", "выражение 2"]          ],          "выражение 1": [              ["простое выражение 2", "выражение 3"],              ["тттерм 2", "выражение 4"],              ["оп отношения", "простое выражение"],              ["eps"]          ],          "выражение 2": [              ["простое выражение 2", "выражение 5"],              ["оп отношения", "простое выражение"],              ["eps"]          ],          "выражение 3": [              ["оп отношения", "простое выражение"],              ["eps"]          ],          "выражение 4": [              ["простое выражение 2", "выражение 6"],              ["оп отношения", "простое выражение"],              ["eps"]          ],          "выражение 5": [              ["оп отношения", "простое выражение"],              ["eps"]          ],          "выражение 6": [              ["оп отношения", "простое выражение"],              ["eps"]          ],          "простое выражение": [              ["тттерм", "простое выражение 1"],              ["знак", "тттерм", "простое выражение 4"]          ],          "простое выражение 1": [              ["простое выражение 2"],              ["eps"]          ],          "простое выражение 2": [              ["оп типа сложения", "тттерм", "простое выражение 3"]          ],          "простое выражение 3": [              ["простое выражение 2"],              ["eps"]          ],          "простое выражение 4": [              ["простое выражение 2"],              ["eps"]          ],          "тттерм": [              ["фактор", "тттерм 1"]          ],          "тттерм 1": [              ["тттерм 2"],              ["eps"]          ],          "тттерм 2": [              ["оп типа умножения", "фактор", "тттерм 3"]          ],          "тттерм 3": [              ["тттерм 2"],              ["eps"]          ],          "фактор": [              ["идентификатор"],              ["константа"],              ["(", "простое выражение", ")"],              ["not", "фактор"]          ],          "знак": [["+"], ["-"]],          "оп отношения": [["=="], ["<>"], ["<"], ["<="], [">"], [">="]],          "оп типа сложения": [["+"], ["-"], ["or"]],          "оп типа умножения": [["\*"], ["/"], ["div"], ["mod"], ["and"]]      }  } |

**Листинги кода**

Листинг 3. Вспомогательные функции.

|  |
| --- |
| (ns cc.lab03.helpers    (:require [clojure.data.json :as json]              [camel-snake-kebab.core :as csk]              [clojure.string]              [clojure.walk]              [clojure.set]))  (defn format-top-level-keys [hmap format-fn]    (->> hmap         (keys)         (mapcat #(vector % (format-fn %)))         (apply hash-map)         (clojure.set/rename-keys hmap)))  (defn json->grammar [path]    (let [json-string (slurp path)          grammar (json/read-str json-string                                 :key-fn str)          grammar (format-top-level-keys grammar                                         csk/->kebab-case-keyword)          grammar (-> grammar                      (update :terms set)                      (update :nonterms set))          prods (->> (:prods grammar)                     (mapcat (fn [[k v]]                               [k (set v)]))                     (apply hash-map))]      (assoc grammar :prods prods)))  (defn grammar->json [path grammar]    (let [json-string (-> grammar                          (format-top-level-keys csk/->camelCaseString)                          (json/write-str :key-fn name :escape-unicode false))]      (spit path json-string)))  #\_(grammar->json "resources/grammar-res.json" (json->grammar "resources/grammar.json"))  (defn upper-case? [input]    (when (not= (clojure.string/upper-case input)                (clojure.string/lower-case input))      (= (str input) (clojure.string/upper-case input))))  (def ru->en-map {"а" "a", "б" "b", "в" "v", "г" "g", "д" "d", "е" "e", "ё" "e", "ж" "j", "з" "z",                   "и" "i", "й" "i", "к" "k", "л" "l", "м" "m", "н" "n", "о" "o", "п" "p", "р" "r",                   "с" "s", "т" "t", "у" "u", "ф" "f", "х" "h", "ц" "c", "ч" "ch", "ш" "sh", "щ" "sc",                   "ъ" "", "ы" "y", "ь" "", "э" "e", "ю" "iu", "я" "iu"})  (defn ru->en    "Транслитерация в формате международной телеграммы."    [input]    (apply str (map #(if (upper-case? %)                       (->> %                            (str)                            (clojure.string/lower-case)                            (get ru->en-map)                            ((fn [sym] (if (nil? sym) % (clojure.string/upper-case sym)))))                       (->> %                            (str)                            (get ru->en-map)                            ((fn [sym] (if (nil? sym) % sym)))))                    input)))  (defmacro when-let\*    ([bindings & body]     (if (seq bindings)       `(when-let [~(first bindings) ~(second bindings)]          (when-let\* ~(drop 2 bindings) ~@body))       `(do ~@body))))  (defn expand-selected-macros    [form namespaced-syms]    (clojure.walk/postwalk (fn [el]                             (if (and (or (seq? el)                                          (list? el))                                      (some #{(first el)} namespaced-syms))                               (expand-selected-macros (macroexpand-1 el) namespaced-syms)                               el))                           form)) |

Листинг 4. Функции для генерации синтаксических анализаторов.

|  |
| --- |
| (ns cc.lab03.generator    (:require [cc.lab03.helpers :refer [ru->en when-let\* expand-selected-macros]]              [clojure.string]              [clojure.pprint]))  (defn transliterate-nonterm [nonterm]    (-> nonterm        (clojure.string/trim)        (clojure.string/replace #"\s+" "-")        (ru->en)        (#(if (#{"debug" "term" "epsilon"} %)            (throw (Exception. (str "Недопустимое имя '" nonterm "' для нетерминала.")))            %))))  #\_(transliterate-nonterm "список операторов")  (defmacro for-alts    "nt-alternatives - последовательность, каждый элемент которой - результат разбора (возможно nil)                       одного из альтернативных правил для нетерминала nt.     alt-sym - символ, с которым будет связан текущий рассматриваемый элемент последовательности               nt-alternatives.     form - какая-то форма, в которой доступен символ alt-sym со связанным значением.     Если form возвращает nil, то будет рассмотрен следующий символ из nt-alternatives, иначе обход     прекращается и возвращается результат form."    [[alt-sym nt-alternatives] form]    `(let [#\_#\_rollback-count# (atom 0)]       (-> (fn [\_# ~alt-sym]  ; Чтобы отладочная печать заработала необходимо раскрыть макрос for-alts (см. gen-analyzer).             #\_(when (some? ~alt-sym)               (swap! rollback-count# inc)               (when (and debug? (> @rollback-count# 1))                 (println "rollback №" @rollback-count#)                 (println "rollback tokens" (first ~alt-sym))                 (println "rollback outputs" (second ~alt-sym))))             (when-let [result# ~form]               (reduced result#)))         (reduce nil ~nt-alternatives))))  (defmacro succ-lazy    [chain {:keys [terms nonterms epsilon] :as grammar} tokens outputs]    (if (seq chain)      (let [sym (first chain)            curr-alt (gensym "curr-alt-")            succ-body `(when-let\* [[~tokens res#] ~(cond (get terms sym) `(term? ~sym ~tokens)                                                         (= epsilon sym) `(epsilon? ~tokens)                                                         :else curr-alt)                                   ~outputs (conj ~outputs res#)]                                  (succ-lazy ~(rest chain) ~grammar ~tokens ~outputs))]        (if (get nonterms sym)          `(for-alts [~curr-alt (~(-> (transliterate-nonterm sym)                                      (str "?")                                      (symbol)) ~tokens)]                     ~succ-body)          succ-body))      [tokens outputs]))  (defmacro succ    [chain {:keys [terms epsilon] :as grammar} tokens outputs]    (if (seq chain)      `(when-let\* [[~tokens res#] ~(let [sym (first chain)]                                     (cond (get terms sym) `(term? ~sym ~tokens)                                           (= epsilon sym) `(epsilon? ~tokens)                                           :else `(~(-> (transliterate-nonterm sym)                                                        (str "?")                                                        (symbol)) ~tokens)))                   ~outputs (conj ~outputs res#)]                  (succ ~(rest chain) ~grammar ~tokens ~outputs))      [tokens outputs]))  (defmacro alt-lazy [nt alt-number nt-prods grammar tokens outputs]    (when (seq nt-prods)      `(lazy-seq        (cons (when-let [branch# (do (when debug? (println ~nt ~alt-number))                                     (succ-lazy ~(first nt-prods) ~grammar ~tokens ~outputs))]                [(first branch#) (into [~(keyword nt)] (second branch#))])              (alt-lazy ~nt ~(inc alt-number) ~(rest nt-prods) ~grammar ~tokens ~outputs)))))  (defmacro alt [nt alt-number nt-prods grammar tokens outputs]    (when (seq nt-prods)      `(if-let [branch# (do (when debug? (println ~nt ~alt-number))                            (succ ~(first nt-prods) ~grammar ~tokens ~outputs))]         [(first branch#) (into [~(keyword nt)] (second branch#))]         (alt ~nt ~(inc alt-number) ~(rest nt-prods) ~grammar ~tokens ~outputs))))  (defmacro nt-analyzer [nt grammar lazy?]    (let [translit (transliterate-nonterm nt)]      `(defn ~(symbol (str translit "?")) [tokens]         (when debug? (println {:name ~translit :tokens tokens}))         (let [outputs []]           (~(if lazy? `alt-lazy `alt)            ~translit 1 ~(-> grammar :prods (get nt)) ~grammar tokens outputs)))))  (defn simplify [program]    (-> program        (clojure.string/replace #"(branch|res)\_\_\d+\_\_auto\_\_" "$1")        (clojure.string/replace #"curr-alt-\d+" "curr-alt")        (clojure.string/replace #"(outputs|\[tokens res\])\r\n\s+" "$1 ")))  (defn gen-analyzer [grammar & {:keys [ns-name lazy? debug?]                                 :or {ns-name "cc.lab03.autogen"                                      lazy? true debug? true}}]    (let [res (-> `((ns ~(symbol (name ns-name))                      (:require ~@(when lazy? `([cc.lab03.generator :refer [for-alts]]))                                [cc.lab03.helpers :refer [when-let\*]]))                    (def debug? ~debug?)                    ~@(for [nt (:nonterms grammar)]                        `(declare ~(-> (transliterate-nonterm nt) (str "?") (symbol))))                    (defn term? [term tokens]                      (when debug? (println {:name (format "term '%s'" term) :tokens tokens}))                      (when (= term (first tokens))                        [(rest tokens) [:term (first tokens)]]))                    (defn epsilon? [tokens]                      (when debug? (println {:name "epsilon" :tokens tokens}))                      [tokens [:epsilon]])                    ~@(for [nt (:nonterms grammar)]                        `(cc.lab03.generator/nt-analyzer ~nt ~grammar ~lazy?)))                  (expand-selected-macros ['cc.lab03.generator/nt-analyzer                                           'cc.lab03.generator/alt                                           'cc.lab03.generator/succ                                           'cc.lab03.generator/alt-lazy                                           'cc.lab03.generator/succ-lazy                                           #\_'cc.lab03.generator/for-alts])                  (#(with-out-str                      (apply clojure.pprint/write % [:dispatch clojure.pprint/code-dispatch                                                     :right-margin 100                                                     :suppress-namespaces true]))))          res (clojure.string/replace res #"\s+\((def|declare|defn) " "\r\n\r\n($1 ")          res (apply str (butlast (rest res)))          res (simplify res)]      res)) |

Листинг 5. Вызов заранее сгенерированного синтаксического анализатора из функции -main.

|  |
| --- |
| (ns cc.lab03.core    (:require [cc.lab03.generator :refer [gen-analyzer]]              [cc.lab03.autogen :refer [programma?]]              [cc.lab03.helpers :refer [json->grammar]]              [clojure.string])    (:gen-class))  (comment    "Прежде чем использовать анализатор его необходимо сгенерировать, а затем скомпилировать."    (-> (json->grammar "resources/grammar.json")        (gen-analyzer :debug? false :lazy? true :ns-name "cc.lab03.autogen")        (->> (spit "src/cc/lab03/autogen.clj"))))  (defn -main    [path]    (let [program (slurp path)          tokens (-> program                     (clojure.string/trim)                     (clojure.string/replace #"\s+" " ")                     (clojure.string/split #" "))]      (second (programma? tokens))))  (-main "resources/program.lab") |

**Пример работы программы**

Листинг 6. Входные данные для разбора.

|  |
| --- |
| {  идент = конст ;    {  {  {  идент = - конст == конст  }  } ;  идент = идент  } ;    идент = ( идент \* конст ) < идент or not ( идент mod конст ) ;  идент = идент - конст  } |

Дерево, полученное в результате разбора методом рекурсивного спуска кода, приведённого в листинге 6, представлено в листинге 7.

Листинг 7. Дерево разбора, построенное для кода из листинга 6.

|  |
| --- |
| [:programma   [:blok    [:term "{"]    [:spisok-operatorov     [:operator      [:term "идент"]      [:term "="]      [:vyrajenie [:faktor [:term "конст"]] [:vyrajenie-1 [:epsilon]]]]     [:spisok-operatorov-1      [:hvost       [:term ";"]       [:operator        [:term "{"]        [:spisok-operatorov         [:operator          [:term "{"]          [:spisok-operatorov           [:operator            [:term "{"]            [:spisok-operatorov             [:operator              [:term "идент"]              [:term "="]              [:vyrajenie               [:znak [:term "-"]]               [:ttterm [:faktor [:term "конст"]] [:ttterm-1 [:epsilon]]]               [:vyrajenie-2                [:op-otnosheniiu [:term "=="]]                [:prostoe-vyrajenie                 [:ttterm [:faktor [:term "конст"]] [:ttterm-1 [:epsilon]]]                 [:prostoe-vyrajenie-1 [:epsilon]]]]]]             [:spisok-operatorov-1 [:epsilon]]]            [:term "}"]]           [:spisok-operatorov-1 [:epsilon]]]          [:term "}"]]         [:spisok-operatorov-1          [:hvost           [:term ";"]           [:operator            [:term "идент"]            [:term "="]            [:vyrajenie [:faktor [:term "идент"]] [:vyrajenie-1 [:epsilon]]]]           [:hvost-1 [:epsilon]]]]]        [:term "}"]]       [:hvost-1        [:hvost         [:term ";"]         [:operator          [:term "идент"]          [:term "="]          [:vyrajenie           [:faktor            [:term "("]            [:prostoe-vyrajenie             [:ttterm              [:faktor [:term "идент"]]              [:ttterm-1               [:ttterm-2                [:op-tipa-umnojeniiu [:term "\*"]]                [:faktor [:term "конст"]]                [:ttterm-3 [:epsilon]]]]]             [:prostoe-vyrajenie-1 [:epsilon]]]            [:term ")"]]           [:vyrajenie-1            [:op-otnosheniiu [:term "<"]]            [:prostoe-vyrajenie             [:ttterm [:faktor [:term "идент"]] [:ttterm-1 [:epsilon]]]             [:prostoe-vyrajenie-1              [:prostoe-vyrajenie-2               [:op-tipa-slojeniiu [:term "or"]]               [:ttterm                [:faktor                 [:term "not"]                 [:faktor                  [:term "("]                  [:prostoe-vyrajenie                   [:ttterm                    [:faktor [:term "идент"]]                    [:ttterm-1                     [:ttterm-2                      [:op-tipa-umnojeniiu [:term "mod"]]                      [:faktor [:term "конст"]]                      [:ttterm-3 [:epsilon]]]]]                   [:prostoe-vyrajenie-1 [:epsilon]]]                  [:term ")"]]]                [:ttterm-1 [:epsilon]]]               [:prostoe-vyrajenie-3 [:epsilon]]]]]]]]         [:hvost-1          [:hvost           [:term ";"]           [:operator            [:term "идент"]            [:term "="]            [:vyrajenie             [:faktor [:term "идент"]]             [:vyrajenie-1              [:prostoe-vyrajenie-2               [:op-tipa-slojeniiu [:term "-"]]               [:ttterm [:faktor [:term "конст"]] [:ttterm-1 [:epsilon]]]               [:prostoe-vyrajenie-3 [:epsilon]]]              [:vyrajenie-3 [:epsilon]]]]]           [:hvost-1 [:epsilon]]]]]]]]]    [:term "}"]]] |

**Заключение**

В ходе выполнения работы были изучены методы синтаксического анализа, в частности, метод рекурсивного спуска, применяемый в данной работе.

Для грамматики из индивидуального варианта была построена новая, дополняющая исходную конструкциями, описывающими блоки с последовательностями присваивания в стиле языка Си. Из полученной грамматики была удалена левая рекурсия и проведена левая факторизация.

На основе описания рассмотренного метода была реализована программа для синтаксического анализа методом рекурсивного спуска для модифицированной грамматики.