УДК 004.4'413 DOI 10.17223/20710410/55/8

ПОСТРОЕНИЕ СИНТАКСИЧЕСКИХ АНАЛИЗАТОРОВ НА ОСНОВЕ СИНТАКСИЧЕСКИХ ДИАГРАММ С МНОГОВХОДОВЫМИ КОМПОНЕНТАМИ

Ю. Д. Рязанов, С. В. Назина

Белгородский государственный технологический университет им. В. Г. Шухова, г. Белгород, Россия

E-mail: razanov.yd@bstu.ru, lanalana9808@gmail.com

Рассматривается задача построения синтаксических анализаторов по синтаксическим диаграммам с многовходовыми компонентами (СД). Предлагается основанный на алгоритме GLL алгоритм построения синтаксического анализатора, результатом работы которого является компактное преставление леса разбора входной цепочки. Предложенный алгоритм позволяет строить синтаксические анализаторы по СД произвольной структуры и не требует предварительных преобразований СД. Построенные синтаксические анализаторы могут применяться для анализа любых контекстно-свободных языков, включая недетерминированные и неоднозначные. Вводятся понятия «дерево вывода» и «лес разбора» для СД, описываются структуры данных, используемые анализатором, такие, как стек с графовой структурой, дескриптор синтаксического анализатора, компактное представление леса разбора. Описывается алгоритм построения синтаксических анализаторов по СД и приводится пример построения такого анализатора.

Ключевые слова: *синтаксический анализ, синтаксические диаграммы с многовходовыми компонентами, лес разбора.*

BUILDING PARSERS BASED ON SYNTAX DIAGRAMS WITH MULTIPORT COMPONENTS

Yu. D. Ryazanov, S. V. Nazina

Belgorod State Technological University named after V. G. Shukhov, Belgorod, Russia

The problem of constructing parsers from syntax diagrams with multiport components (SD) is solved. An algorithm for constructing a parser based on the GLL algorithm is proposed, which results in the compact representation of the input chain parse forest. The proposed algorithm makes it possible to build parsers based on the SD of an arbitrary structure and does not require preliminary SD transformations. We introduce the concepts of "inference tree" and "parsing forest" for SD and describe the data structures used by the parser, such as a graph-structured stack, a parser descriptor, and a compact representation of the parsing forest. The algorithm for constructing parsers based on SD is described and an example of parser constructing is given.

Keywords: *parsing, syntax diagrams with multiport components, parse forest.*

Введение

Среди алгоритмов синтаксического анализа наиболее простым и наглядным является метод рекурсивного спуска [1], позволяющий строить анализаторы, структура которых соответствует структуре грамматики. Недостатком таких анализаторов является уязвимость к левой рекурсии, что приводит к необходимости предварительной обработки грамматики. Кроме того, использование возвратов в случае недетерминированного разбора может привести к экспоненциальному росту времени работы анализатора. Эти проблемы решает Generalised LL (GLL-анализ) [2, 3] — алгоритм, расширяющий метод рекурсивного спуска до построения синтаксических анализаторов без ограничений на класс контекстно-свободных грамматик. Результатом работы анализатора является лес разбора, который может представлять бесконечное число различных деревьев вывода заданной цепочки. Построенные анализаторы имеют в худшем случае кубическую сложность (временную и по памяти) и линейную сложность для LL-грамматик [3].

Описанный в [2, 3] алгоритм применим для построения синтаксических анализаторов по формальным грамматикам. Синтаксические диаграммы с многовходовыми компонентами (СД) [4] являются наглядным и более компактным представлением языка, чем формальные грамматики или диаграммы Вирта. GLL-анализ может быть расширен до построения синтаксических анализаторов по синтаксическим диаграммам с многовходовыми компонентами, что позволит строить компактные и эффективные по памяти и времени работы анализаторы, структура которых соответствует структуре исходной диаграммы.

В работе приводится основанный на GLL-анализе алгоритм построения синтаксических анализаторов по СД. Сначала поясняются понятия, связанные с синтаксическими диаграммами с многовходовыми компонентами, необходимые для дальнейшего изложения. Далее приводится описание структур данных, используемых анализатором в процессе разбора: леса разбора, стека с графовой структурой и дескриптора анализатора. После этого описывается построение синтаксических анализаторов по СД и приводится пример построения и работы такого анализатора.

1. Синтаксические диаграммы с многовходовыми компонентами

Синтаксическая диаграмма с многовходовыми компонентами представлена на рис. 1.

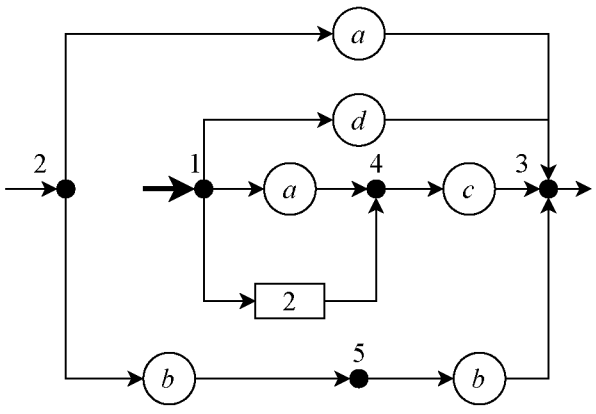


Рис. 1. Синтаксическая диаграмма с многовходовыми компонентами

Синтаксическая диаграмма с многовходовыми компонентами [4] задается восьмёркой $R = (T, u, u', u'', u_0, G, F_T, F_U)$, где

- T — конечное множество терминалов;
- u — конечное множество узлов;
- u' — конечное множество начальных узлов, $u' \subseteq u$;
- u'' — конечное множество заключительных узлов, $u'' \subseteq u$;
- u_0 – стартовый узел, $u_0 \in u'$;
- $G = (V, E)$ — ориентированный граф, $V = V_T \cup V_N \cup u$ — множество вершин, V_T — множество терминальных вершин, V_N — множество нетерминальных вершин, внутри которых записан начальный узел; $E = E_1 \cup E_2$, E_1 — множество дуг, выходящих из узлов и входящих в терминальные и нетерминальные вершины, E_2 — множество дуг, выходящих из терминальных и нетерминальных вершин и входящих в узлы;
- $F_T : V_T \to T$ — отображение множества терминальных вершин в множество терминалов;
- $F_U : V_N \to u'$ — отображение множества нетерминальных вершин в множество начальных узлов.

Терминальная вершина изображается на диаграмме кружком, в который вписан терминал. Нетерминальная вершина изображается прямоугольником, в который вписан начальный узел одной из компонент. Узлы пронумерованы и обозначаются жирными точками. Терминальные и нетерминальные вершины назовём символьными. Дуги соединяют между собой узел и символьную вершину. Начальные узлы обозначаются входящей стрелкой, заключительные — выходящей, стартовый — жирной входящей стрелкой. В символьную вершину может входить только одна дуга, и из символьной вершины может выходить только одна дуга. Ограничений на количество дуг, входящих и выходящих из узлов, нет.

С помощью СД можно получить любую цепочку языка, заданного СД. Процесс получения цепочки языка назовём выводом. Для определения вывода в работе [4] вводятся следующие понятия:

- цепочка, связывающая узел u с заключительными узлами — это цепочка, состоящая из терминалов и/или начальных узлов, которую можно получить, «двигаясь» в СД от узла u к заключительному узлу и выписывая из вершин по пути символы (терминалы или начальные узлы) в изначально пустую цепочку;
- $L(u)$ — множество всех цепочек, связывающих узел u с заключительными узлами.

Вывод цепочки языка, заданного СД со стартовым узлом u_0, заключается в следующем. Возьмём цепочку, принадлежащую $L(u_0)$. Если она содержит начальный узел u_i, заменим его цепочкой из множества $L(u_i)$. Если новая цепочка содержит начальный узел, то аналогичные действия повторяем. Вывод заканчивается, когда будет получена цепочка, не содержащая начальных узлов. Такая цепочка, дополненная концевым маркером, принадлежит языку, заданному СД.

Любая цепочка языка, заданного СД, имеет хотя бы один вывод. Если любая цепочка языка имеет только один вывод в заданной СД, то такую СД назовём однозначной; если хотя бы одна цепочка языка имеет более одного вывода, то — неоднозначной. Например, цепочка ac принадлежит языку, заданному СД (рис. 1), и имеет два вывода, которые можно представить следующим образом:

1) $1 \Rightarrow ac$;
2) $1 \Rightarrow 2c \Rightarrow ac$.

Следовательно, СД на рис. 1 является неоднозначной.

Вывод в СД можно представить и по-другому: как движение по дугам от стартового узла начальной компоненты к точке выхода. При этом если дуга идёт в терминальную вершину, то вписанный в неё терминальный символ добавляется в терминальную цепочку; если в нетерминальную, то переходим во вписанный в неё начальный узел и движемся далее аналогичным образом до точки выхода, после чего возвращаемся в предыдущую компоненту и продолжаем движение. После прохождения выходной дуги начальной компоненты в терминальную цепочку добавляем концевой маркер и вывод заканчивается.

Символ x, который может быть добавлен в терминальную цепочку непосредственно после прохождения дуги e, принадлежит множеству выбора дуги e ($x \in$ ВЫБОР(e)). Алгоритм вычисления множества выбора дуги описан в [5–7]. Объединение множеств выбора всех выходящих из узла дуг образует множество выбора узла. Узел u называется детерминированным, если множества выбора любых двух дуг, выходящих из узла u, не пересекаются. Назовём вывод цепочки в СД детерминированным, если на каждом шаге вывода текущий символ цепочки принадлежит множеству выбора только одной выходящей из узла дуги. СД является детерминированной, если в ней все узлы детерминированные, иначе — недетерминированной [8, 9].

2. Дерево вывода

Любой вывод цепочки, принадлежащей языку, заданному СД, можно представить деревом вывода. Ориентированное упорядоченное дерево D назовём деревом вывода в синтаксической диаграмме с многовходовыми компонентами $R = (T, u, u', u'', u_0, G, F_T, F_U)$, если выполняются следующие условия:

- корень дерева отмечен узлом u_0;
- листья дерева отмечены терминалами из множества T или символом ϵ, обозначающим пустую цепочку;
- внутренние узлы дерева отмечены узлами из множества u';
- если узел w_0 дерева вывода отмечен начальным узлом z_0 и имеет сыновей $w_1, \ldots, w_n$, $n \geqslant 1$, отмеченных соответственно терминалами или начальными узлами $z_1, \ldots, z_n$, то в R существует путь из узла z_0 до заключительного узла компоненты, последовательно проходящий только через вершины, отмеченные $z_1, \ldots, z_n$;
- если узел w, отмеченный начальным узлом z_0, имеет единственного сына, отмеченного ϵ, то $z_0 \in u''$.

Два вывода цепочки ac в СД на рис. 1, приведённые в п. 1, представлены в виде двух деревьев на рис. 2.

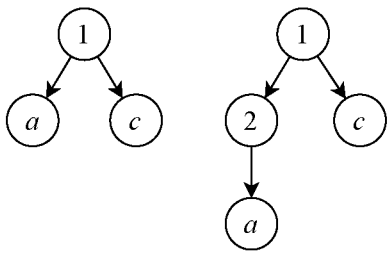


Рис. 2. Деревья вывода цепочки ac

3. Лес разбора

Множество всех деревьев вывода заданной цепочки в неоднозначной СД образует лес разбора. Если СД содержит путь ненулевой длины из узла u в узел u, при движении по которому выводится пустая терминальная цепочка, то возможно бесконечное множество различных выводов одной цепочки, проходящих через узел u. Лес разбора в этом случае содержит бесконечное множество деревьев. В этой работе будем предполагать, что любая выводимая в СД цепочка имеет конечное множество деревьев вывода.

Для компактного представления леса разбора (КПЛР) используется структура, основанная на SPPF (Shared Packed Parse Forest) [10], которая имеет три типа узлов: символьные, промежуточные и упаковывающие. Деревья на рис. 2 образуют КПЛР цепочки ac на рис. 3.

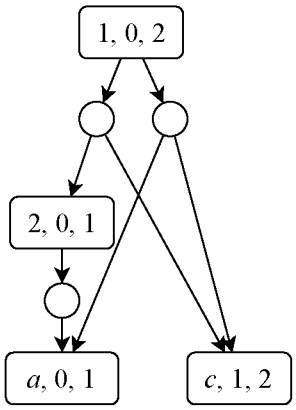


Рис. 3. КПЛР цепочки ac

Символьные узлы имеют вид (x, i, j), где x — терминал или начальный узел компоненты; i и j — левая и правая границы подцепочки, выводимой из x. Если x — терминал, то назовём такой узел терминальным, иначе — нетерминальным. Символьные узлы изображаются в виде прямоугольников с закругленными углами.

Промежуточные узлы имеют метку $(s \to v, i, j)$, где s — начальный узел компоненты; v — узел компоненты; i и j — левая и правая границы подцепочки, выводимой на промежутке движения от узла s к узлу v. Промежуточные узлы изображаются в виде прямоугольников.

Дочерними узлами символьных и промежуточных узлов могут быть только упаковывающие узлы, ограничение на количество дочерних узлов не накладывается. Упаковывающие узлы являются уникальными в множестве детей своего родителя и имеют не более двух дочерних узлов. Правый дочерний узел упаковывающего узла является символьным узлом, а левый может быть символьным или промежуточным. Метка упаковывающего узла позволяет идентифицировать конкретный вывод подцепочки. Необходимо определить такую метку упаковывающего узла, которая бы однозначно определяла ветвь вывода. Положим, левый дочерний узел имеет метку (t, i, k), а правый — (x, k, j). При этом если левый дочерний узел — символьный, то t — терминал или начальный узел, иначе t имеет вид $s \to w$. Тогда определим метку упаковывающего узла как (t, x, k). Если к родительскому узлу необходимо присоединить дочерние узлы через упаковывающий узел (t, x, k), то сначала проверяется, не существует ли узла (t, x, k) в множестве детей родительского узла. Если такой узел найден, то соответ-

ствующий ему вывод уже добавлен в лес разбора. Чтобы не загромождать рисунок, упаковывающие узлы будем изображать окружностью без меток.

Если лес разбора представляет собой конечное множество деревьев вывода, то КПЛР не содержит циклов.

Рассмотрим получение всех деревьев вывода по КПЛР, не содержащему циклов (алгоритм 1). Будем использовать следующие обозначения: $[a]$ — список, состоящий из элемента a; \$ — неопределённое значение переменной.

Алгоритм 1. Получение деревьев вывода по КПЛР

Вход: v — узел леса разбора.
Выход: деревья вывода.

```
1: ПРОЦЕДУРА getTrees(v)
2: H ← ∅.
3: Если v — терминальный узел с меткой (x, i, j), то
4:   T ← создать дерево с корнем, отмеченным x; добавить [T] в H;
5: иначе если v — нетерминальный узел с меткой (x, i, j), то
6:   Для всех дочерних узлов p узла v
7:     пусть z и r — левый и правый дочерние узлы p соответственно;
8:     H_1 ← getTrees(z).
9:     Если r ≠ $, то
10:      H_2 ← getTrees(r).
11:      Для всех (h_1, h_2) ∈ H_1 × H_2
12:        T ← создать дерево с корнем w, отмеченным x;
13:        последовательно сделать все корневые узлы деревьев из списков h_1 и h_2
           дочерними узлами w;
14:        добавить [T] в H;
15:    иначе
16:      Для всех h ∈ H_1
17:        T ← создать дерево с корнем w, отмеченным x;
18:        последовательно сделать все корневые узлы деревьев из списка h_1 дочер-
           ними узлами w;
19:        добавить [T] в H;
20: иначе
21:   Для всех дочерних узлов p узла v
22:     пусть z и r — левый и правый дочерние узлы p соответственно;
23:     H_1 ← getTrees(z).
24:     Если r ≠ $, то
25:       H_2 ← getTrees(r).
26:       Для всех (h_1, h_2) ∈ H_1 × H_2
27:         L ← конкатенация списков h_1 и h_2;
28:         добавить L в H;
29:     иначе
30:       добавить все элементы H_1 в H.
31: Вернуть H.
```

Будем обрабатывать узлы леса разбора, начиная с корневого узла, отмеченного $(s, 0, m)$, где s — стартовый узел СД; m — длина входной цепочки. В результате при-

менения процедуры $getTrees$ к корневому узлу леса разбора получим множество H, состоящее из одноэлементных списков. Элементы списков будут представлять собой деревья вывода исходной цепочки.

4. Процедуры для работы с лесом разбора

Опишем используемые синтаксическим анализатором процедуры для работы с лесом разбора.

Обозначим $getNodeI(s, v, w, z)$ процедуру, которая находит или создаёт промежуточный узел со значением $s \to v$, при этом левым дочерним узлом упаковывающего узла становится узел w, а правым — z (алгоритм 2).

Алгоритм 2. Получение промежуточного узла леса разбора

Вход: s — начальный узел СД; v — текущий узел СД; w — левый дочерний узел; z — правый дочерний узел.
Выход: промежуточный узел леса разбора.

ПРОЦЕДУРА $getNodeI(s, v, w, z)$
Если $(w = \$)$, **то**
  Вернуть z,
иначе
  пусть w отмечен (t, i, k), z отмечен (q, k, j);
  $y \leftarrow$ найти или создать узел с меткой $(s \to v, i, j)$.
  Если y не имеет упаковывающего ребёнка с меткой (t, q, k), **то**
    создать такого ребёнка с левым потомком w и правым z.
Вернуть y.

Обозначим $getNodeN(s, w, z)$ процедуру, которая находит или создаёт нетерминальный узел со значением s, при этом левым дочерним узлом упаковывающего узла становится узел w, а правым — z (алгоритм 3).

Алгоритм 3. Получение нетерминального узла леса разбора

Вход: s — начальный узел СД; w — левый дочерний узел; z — правый дочерний узел.
Выход: нетерминальный узел леса разбора.

ПРОЦЕДУРА $getNodeN(s, w, z)$
Пусть z отмечен (q, k, j).
Если $w \neq \$$, **то**
  пусть w отмечен (t, i, k),
иначе
  $i \leftarrow k$, $t \leftarrow \$$;
$y \leftarrow$ найти или создать узел с меткой (s, i, j).
Если y не имеет дочернего упаковывающего узла с меткой (t, q, k), **то**
  Если $w \neq \$$, **то**
    создать такой дочерний узел с левым потомком w и правым z,
  иначе
    создать такой дочерний узел с потомком z.
Вернуть y.

Обозначим:

— $convert(z)$ — процедура, которая преобразует промежуточный узел z к символьному узлу (алгоритм 4);
— $getNodeT(x, i)$ — процедура, которая находит или создаёт терминальный узел с меткой $(x, i, i + 1)$, где x — терминал, соответствующий i-й позиции входного буфера;
— $getNodeE(i)$ — процедура, которая находит или создаёт ϵ-узел с меткой (ϵ, i), где i — позиция входного буфера.

Эти процедуры возвращают найденный или созданный узел леса разбора.

Алгоритм 4. Преобразование промежуточного узла леса разбора в символьный узел

Вход: z — промежуточный узел леса разбора.
Выход: символьный узел леса разбора.

```
1: ПРОЦЕДУРА convert(z)
2: пусть (n → v, i, j) — метка z;
3: y ← найти или создать узел леса разбора с меткой (n, i, j).
4: Для всех дочерних узлов x узла z:
5:     Если не существует дуги из y в x, то
6:         добавить дугу.
7: Вернуть y.
```

5. Дескриптор и стек синтаксического анализатора

Определим конфигурацию синтаксического анализатора пятёркой (s, L, u, i, y), где s — начальный узел компоненты; L — метка программы; u — узел стека; i — позиция входного буфера; y — узел леса разбора. Назовём пятёрку (s, L, u, i, y) дескриптором синтаксического анализатора. Дескрипторы помещаются в множество дескрипторов D. Во внешнем цикле алгоритма работы синтаксического анализатора из D извлекается дескриптор и анализатор возобновляет свою работу, применяя сохранённую конфигурацию. Чтобы избежать зацикливания, дополнительно вводится множество U, в котором сохраняются все созданные дескрипторы.

Перед созданием дескриптора проверяется, существует ли уже такой дескриптор в множестве U. В этом случае повторного добавления дескриптора в множество D не происходит. Обозначим $add(s, L, u, i, y)$ процедуру, включающую дескриптор (s, L, u, i, y) в множества U и D, если множество U ещё не содержит (s, L, u, i, y).

Представим разбор как движение по СД, начинающееся со стартового узла. Если при обработке узла возможно несколько путей разбора (текущий символ принадлежит множеству выбора нескольких выходящих из узла дуг), то для каждого из них создаётся соответствующий дескриптор и помещается в множество D, а затем происходит переход к внешнему циклу алгоритма. При этом метка, сохраняемая в дескрипторе, — это метка программы, соответствующая фрагменту, описывающему разбор по одному из возможных путей.

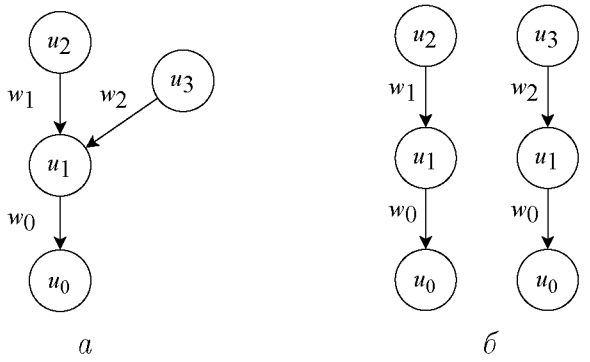
При движении по компоненте будем строить часть леса разбора, соответствующую этой компоненте. Текущий узел на каждом шаге будем рассматривать как корень леса разбора. При входе в компоненту текущий узел леса разбора не определён. При выходе из компоненты корень должен представлять собой символьный узел (s, i, j), где s — начальный узел, через который был произведён вход в компоненту; i и j — границы подцепочки, выводимой в ходе движения по компоненте.

При переходе через терминальную вершину будем создавать в лесу разбора соответствующий терминалу узел, а затем объединять его с текущим корнем, создавая для них родительский промежуточный узел. Текущий узел леса разбора при этом станет левым ребёнком, а терминальный узел — правым. Созданный родительский узел станет новым корнем леса разбора.

При переходе через нетерминальную вершину будем запоминать позицию возврата и текущий корень леса разбора в стеке (см. ниже), а затем переходить к обработке компоненты, начальный узел которой записан в нетерминальной вершине.

При выходе из компоненты необходимо «извлечь» текущую вершину стека. При извлечении узла стека в множество D добавляются дескрипторы для каждой выходящей из извлекаемого узла дуги. Текущий корень леса разбора при этом нужно преобразовать в символьный узел, соответствующий начальному узлу обработанной компоненты, а затем объединить с узлом, которым отмечена выходящая дуга. Результат объединения добавляется в дескриптор в качестве узла леса разбора. После обработки компоненты происходит переход к внешнему циклу алгоритма работы синтаксического анализатора. В конце успешного разбора в лесу разбора должен быть создан корневой узел $(s, 0, m)$, где s — стартовый узел; m — длина входной цепочки.

Синтаксический анализатор использует стек с графовой структурой (Graph-Structured Stack, GSS) [11] для сохранения позиции возврата после обработки компоненты. Стек с графовой структурой представляет собой направленный граф, где каждый путь является стеком. Например, стек на рис. 4, *а* задаёт множество стеков на рис. 4, *б*. Если в процессе анализа происходит переход через нетерминальную вершину, то необходимо сохранить информацию об узле, следующем за нетерминальной вершиной, и текущем входе компоненты. Дуги стека отмечаются узлами леса разбора. В алгоритме GLL в стеке с графовой структурой разрешаются циклы, необходимые для обработки левой рекурсии.



а *б*

Рис. 4. Стек с графовой структурой (*а*) и задаваемые им стеки (*б*)

Определим метку узла стека как (s, v, i), где s — начальный узел СД, соответствующий текущему входу компоненты; v — узел СД, следующий за нетерминальной вершиной. Корневой узел стека обозначим u_0. Введём множество P, в которое будем сохранять узел стека u и узел леса разбора, которым отмечена выходящая из u дуга, при извлечении u из стека.

Определим процедуру $create(s, v, u, i, w)$, создающую в стеке родительский для узла u узел с меткой (s, v, i) (алгоритм 5). Дуга при этом помечается узлом w леса разбора.

Алгоритм 5. Создание узла стека

Вход: s — начальный узел компоненты; v — узел компоненты; u — узел стека; i — позиция входного буфера; w — узел леса разбора.
Выход: узел стека.

ПРОЦЕДУРА $create(s, v, u, i, w)$
$y \leftarrow$ найти или создать узел стека с меткой (s, v, i).
Если не существует дуги из y в u с меткой w, **то**
    создать дугу из y в u с меткой w.
Для всех $(y, z) \in P$
    $x \leftarrow getNodeI(s, v, w, z)$;
    пусть h — правая граница z;
    $add(s, L_v, u, h, x)$.
Вернуть y.

Определим процедуру $pop(u, i, z)$, извлекающую узел u из стека (алгоритм 6). Для каждой выходной дуги создаётся соответствующий ей дескриптор. В качестве узла леса разбора в дескриптор помещается родительский узел для z и узла, которым отмечена дуга.

Алгоритм 6. Извлечение узла стека

Вход: u — узел стека; u_0 — корневой узел стека; i — позиция входного буфера; z — узел леса разбора.

ПРОЦЕДУРА $pop(u, i, z)$
Если $u \neq u_0$, **то**
    пусть (s, v, k) — метка u;
    добавить (u, z) в P.
    Для всех дуг (u, w, y)
        $x \leftarrow getNodeI(s, v, w, z)$;
        $add(s, L_v, y, i, x)$.

6. Построение синтаксического анализатора

Синтаксический анализатор состоит из фрагментов, соответствующих узлам СД, фрагментов, соответствующих выходящим из узлов СД дугам, и фрагмента, соответствующего внешнему циклу алгоритма. Каждому узлу v СД поставим в соответствие фрагмент, отмеченный меткой L_v; i-й дуге, выходящей из узла v, — фрагмент, отмеченный меткой $L_{(v,i)}$. Фрагмент, соответствующий внешнему циклу алгоритма, отметим L_0.

Введём следующие обозначения:

— I — входной поток токенов;
— c_I — текущая позиция во входном буфере;
— c_U — текущий узел стека;
— c_N — текущий узел леса разбора;
— c_S — текущий начальный узел компоненты СД.

Вместо вызова функций и возврата из функций, осуществляемых анализатором рекурсивного спуска, синтаксический анализатор будет выполнять переход на соответствующую метку.

6.1. Определение фрагментов, соответствующих узлам

Пусть из узла v выходят n дуг $e_1, \ldots, e_n$. Множество выбора узла v равно объединению множеств выбора выходящих из него дуг: ВЫБОР(v) = ВЫБОР$(e_1) \cup \ldots \cup$ $\cup$ВЫБОР(e_n). Для каждого терминала $t \in$ ВЫБОР(v) определим множество дуг $E(t)$, таких, что t принадлежит множеству выбора этих дуг: $E(t) = \{e : t \in \text{ВЫБОР}(e)\}$.

Определим отношение R на множестве ВЫБОР(v): $R = \{(t_i, t_j) : E(t_i) = E(t_j)\}$. Оно является отношением эквивалентности и разбивает множество ВЫБОР(v) на классы эквивалентности $Q_1, \ldots, Q_n$. Если $(t_i, t_j) \in R$, то терминалы t_i и t_j назовём эквивалентными. Класс эквивалентности Q_i представляет собой множество попарно эквивалентных терминалов. Множество дуг $E(Q_i)$ равно множеству $E(t)$, если $t \in Q_i$.

Фрагмент для узла v опишем следующим образом: L_v : $code(v)$. Определение $code(v)$ приведено в алгоритме 7.

Алгоритм 7. Фрагмент $code(v)$, соответствующий узлу v

1: **Если** $I[c_I] \in Q_1$, **то**
2: $\quad code(Q_1)$.
3: ...
4: **Если** $I[c_I] \in Q_n$, **то**
5: $\quad code(Q_n)$.
6: Переход L_0.

Положим $E(Q_i) = \{e_1, \ldots, e_n\}$. Определение $code(Q_i)$ приведено в алгоритме 8.

Алгоритм 8. Фрагмент $code(Q_i)$, соответствующий множеству Q_i

1: $add(c_S, L_{(v,1)}, c_U, c_I, c_N)$;
2: ...
3: $add(c_S, L_{(v,n)}, c_U, c_I, c_N)$;
4: переход L_0.

Если $E(Q_i) = \{e_i\}$ (одноэлементное множество), то создание дескриптора можно опустить. В этом случае $code(Q_i)$ = переход $L_{(v,i)}$.

6.2. Определение фрагментов, соответствующих дугам

Определим фрагмент, описывающий i-ю дугу e, выходящую из узла v, следующим образом: $L_{(v,i)} : code(e)$.

Для дуги $e = (v, x, w)$, с которой начинается переход из узла v в узел w через вершину x, возможны следующие ситуации:

1) Если x — терминал, то в лесу разбора необходимо создать терминальный узел, а затем объединить его с текущим корнем леса разбора (алгоритм 9).
2) Если x — начальный узел, то в стек помещается позиция возврата и происходит переход на метку, соответствующую этому узлу (алгоритм 10).

При переходе по выходной дуге $e = (v, \epsilon)$ необходимо преобразовать текущий корень леса разбора в символьный узел, соответствующий входу обрабатываемой компоненты, а затем извлечь узел из стека. Текущий корень c_N может быть как промежу-

Алгоритм 9. Фрагмент $code(v, x, w)$, соответствующий дуге $e = (v, x, w)$

1: $c_R \leftarrow getNodeT(x, c_I)$;
2: $c_N \leftarrow getNodeI(c_S, w, c_N, c_R)$;
3: $c_I \leftarrow c_I + 1$;
4: переход L_w.

Алгоритм 10. Фрагмент $code(v, X, w)$, соответствующий дуге $e = (v, X, w)$

1: $c_U \leftarrow create(c_S, w, c_U, c_I, c_N)$;
2: $c_N \leftarrow \$$;
3: $c_S \leftarrow X$;
4: переход L_x.

точным узлом, если в ходе движения по компоненте было пройдено несколько терминальных или нетерминальных вершин, так и символьным узлом, если в ходе движения по компоненте была пройдена только одна вершина. Если $c_N = \$$, то такая ситуация соответствует переходу по ϵ-пути в компоненте и требует создания ϵ-узла в дереве разбора. Определение $code(v, \epsilon)$ приведено в алгоритме 11. При этом если в компоненте нет ϵ-пути, то ветвь по условию $c_N = \$$ можно опустить.

Алгоритм 11. Фрагмент $code(v, \epsilon)$, соответствующий выходной дуге

1: **Если** $c_N = \$$, **то**
2: $c_R \leftarrow getNodeE(c_I)$;
3: $c_N \leftarrow getNodeN(c_S, c_N, c_R)$,
4: **иначе**
5: **Если** c_N — символьный узел, **то**
6: $c_N \leftarrow getNodeN(c_S, \$, c_N)$,
7: **иначе**
8: $c_N \leftarrow convert(c_N)$.
9: $pop(c_U, c_I, c_N)$;
10: переход L_0.

6.3. Уменьшение количества операторов перехода

В предложенном подходе переход на метку будет выполняться после обработки каждой выходящей из узла дуги. Для уменьшения количества операторов перехода определим цепочку как последовательность узлов и дуг, которые могут быть обработаны без передачи управления другому фрагменту. Вместо фрагментов, описывающих выходящие из узлов дуги, будем использовать фрагменты, описывающие цепочки.

Цепочка начинается дугой, выходящей из узла, и заканчивается дугой, входящей в узел. Промежуточные узлы и дуги принадлежат цепочке. На рис. 5 из узла 1 выходят четыре цепочки: первая цепочка начинается дугой e_1 и заканчивается дугой e_2, вторая — начинается дугой e_3 и заканчивается дугой e_4, третья — начинается дугой e_5 и заканчивается дугой e_6, четвертая — начинается дугой e_7 и заканчивается дугой e_8. Анализатор будет содержать фрагмент, описывающий узел, только в том случае, если узел не принадлежит ни одной цепочке. Такими узлами являются начальные узлы компонент и узлы, в которые входят последние дуги цепочек (на рис. 5 это узлы 1, 4, 6 и 8). Определим множество выбора цепочки как множество выбора дуги, с которой

начинается цепочка. Во фрагменте, описывающем узел, будем проверять принадлежность текущего символа множествам выбора выходящих из узла цепочек.

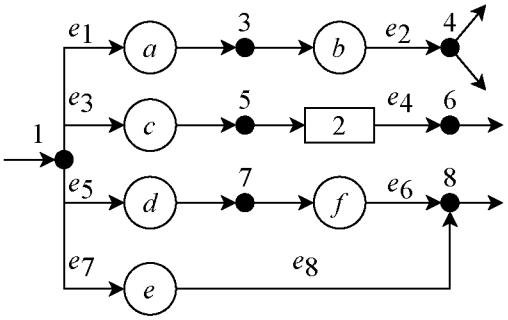


Рис. 5. Выделение цепочек

Пусть $e_1, \ldots, e_n$ — принадлежащие одной цепочке дуги, выходящие из узлов, и первая дуга цепочки — это i-я дуга, выходящая из узла v. Определим фрагмент, описывающий эту цепочку, следующим образом: $L_{(v,i)}$: $code(v, i)$. Определение $code(v, i)$ приведено в алгоритме 12.

Алгоритм 12. Фрагмент $code(v, i)$, соответствующий цепочке

```
code(e_1)
Если I[c_I] ∈ ВЫБОР(e_2), то
    code(e_2),
иначе
    переход L_0.
...
Если I[c_I] ∈ ВЫБОР(e_n), то
    code(e_n),
иначе
    переход L_0.
```

При этом из всех фрагментов $code(e_i)$ (см. п. 6.2), кроме последнего, необходимо исключить операторы перехода.

Определим ситуации, в которых цепочка прерывается:

1) дуга входит в узел, из которого выходит несколько дуг (цепочка $e_1 - e_2$);
2) дуга выходит из нетерминальной вершины (цепочка $e_3 - e_4$);
3) дуга входит в узел, в который входит несколько дуг (цепочки $e_5 - e_6$, $e_7 - e_8$);
4) дуга входит в начальный узел компоненты.

6.4. Общий вид синтаксического анализатора

Общий вид синтаксического анализатора, построенного по синтаксической диаграмме с многовходовыми компонентами, приведён в алгоритме 13. Здесь $A, \ldots, X$ — узлы СД, из которых выходят цепочки; S — стартовый узел; n — количество цепочек, выходящих из узла A; k — количество цепочек, выходящих из узла X.

Алгоритм 13. Общий вид синтаксического анализатора

Вход: m — длина входного буфера без учёта символа конца цепочки; I — входной поток токенов.

Выход: корневой узел КПЛР входной цепочки.

ПРОЦЕДУРА $parse(I, m)$
  $c_U \leftarrow$ создать в стеке узел u_0 с меткой $(s, 0, 0)$;
  $c_N \leftarrow \$$, $c_I \leftarrow 0$, $c_S \leftarrow S$;
  $U \leftarrow \varnothing$, $R \leftarrow \varnothing$, $P \leftarrow \varnothing$;
  переход L_S.
L_0:
  Если $D \neq \varnothing$, **то**
    извлечь (v, L_k, u, i, w) из D;
    $c_U \leftarrow u$, $c_N \leftarrow w$, $c_I \leftarrow I$, $c_S \leftarrow v$;
    переход L_k,
  иначе
    Если существует узел леса разбора с меткой $(S, 0, m)$, **то**
      удалить из леса разбора все узлы, недостижимые из узла $(S, 0, m)$.
      Вернуть узел $(S, 0, m)$,
    иначе
      Вернуть ошибку.

L_A:
  $code(A)$
...
L_X:
  $code(X)$
$L_{(A,1)}$:
  $code(A, 1)$
...
$L_{(A,n)}$:
  $code(A, n)$
...
$L_{(X,1)}$:
  $code(X, 1)$
...
$L_{(X,k)}$:
  $code(X, k)$

7. Пример построения синтаксического анализатора на основе синтаксической диаграммы с многовходовыми компонентами

Построим синтаксический анализатор по синтаксической диаграмме с многовходовыми компонентами на рис. 1. Общий вид анализатора приведён в алгоритме 14.

Обозначим i-ю дугу, выходящую из узла v, как v_i (нумерация сверху вниз). Определим множества выбора всех дуг СД ($\perp$ — символ конца цепочки):

ВЫБОР$(1_1) = \{d\}$;
ВЫБОР$(1_2) = \{a\}$;
ВЫБОР$(1_3) = \{a, b\}$;
ВЫБОР$(2_1) = \{a\}$;
ВЫБОР$(2_2) = \{b\}$;
ВЫБОР$(3_1) = \{c, \perp\}$;
ВЫБОР$(4_1) = \{c\}$;
ВЫБОР$(5_1) = \{b\}$.

Определим узлы, из которых выходят цепочки: 1, 2, 3, 4. Узлы 1 и 2 являются начальными, в узлы 3 и 4 входит несколько дуг, кроме того, в узел 4 входит дуга,

Алгоритм 14. Общий вид анализатора, построенного по СД на рис. 1

```
1: ПРОЦЕДУРА parse(I, m)
2:   c_U ← создать в стеке узел u_0 с меткой (1, 0, 0);
3:   c_N ← $, c_I ← 0, c_S ← 1, U ← ∅, D ← l, P ← ∅;
4:   переход L_1.
5: L_0:
6:   Если D ≠ ∅, то
7:     извлечь (v, L_k, u, i, w) из D;
8:     c_U ← u, c_N ← w, c_I ← I, c_S ← v;
9:     переход L_k,
10:  иначе
11:    Если существует узел леса разбора с меткой (1, 0, m), то
12:      удалить из леса разбора все недостижимые из узла (1, 0, m) узлы.
13:      Вернуть узел (1, 0, m),
14:    иначе
15:      Вернуть ошибку.
16: <фрагменты для узлов СД>
17: <фрагменты для цепочек СД>
```

выходящая из нетерминальной вершины. Фрагменты для цепочек приведены в алгоритме 15.

Разобьём терминалы из множества выбора каждого узла на непересекающиеся подмножества $Q_1, \ldots, Q_n$ и определим для каждого подмножества множество $E(Q_i)$. Множество выбора узла 1 содержит терминалы a, b, d. Терминал d принадлежит множеству выбора первой дуги, терминал a — множествам выбора второй и третьей дуги, терминал b — множеству выбора третьей дуги. Получаем $Q_1 = \{d\}$ и $E(Q_1) = \{1\}$, $Q_2 = \{a\}$ и $E(Q_2) = \{2, 3\}$, $Q_3 = \{b\}$ и $E(Q_3) = \{3\}$. Фрагменты для узлов приведены в алгоритме 16.

Алгоритм 15. Фрагменты для цепочек СД на рис. 1

$L_{(1,1)}$:
  $c_R \leftarrow getNodeT(d, c_I)$;
  $c_N \leftarrow getNodeI(c_S, 3, c_N, c_R)$;
  $c_I \leftarrow c_I + 1$;
  переход L_3.
$L_{(1,2)}$:
  $c_R \leftarrow getNodeT(a, c_I)$;
  $c_N \leftarrow getNodeI(c_S, 4, c_N, c_R)$;
  $c_I \leftarrow c_I + 1$;
  переход L_4.
$L_{(1,3)}$:
  $c_U \leftarrow create(c_S, 4, c_U, c_I, c_N)$;
  $c_N \leftarrow \$$;
  $c_S \leftarrow 2$;
  переход L_2.
$L_{(2,1)}$:
  $c_R \leftarrow getNodeT(a, c_I)$;
  $c_N \leftarrow getNodeI(c_S, 3, c_N, c_R)$;
  $c_I \leftarrow c_I + 1$;
  переход L_3.
$L_{(2,2)}$:
  $c_R \leftarrow getNodeT(b, c_I)$;
  $c_N \leftarrow getNodeI(c_S, 5, c_N, c_R)$;
  $c_I \leftarrow c_I + 1$;
  Если $I[c_I] \in \{b\}$, **то**
    $c_R \leftarrow getNodeT(b, c_I)$;
    $_N \leftarrow getNodeI(c_S, 3, c_N, c_R)$;
    $c_I \leftarrow c_I + 1$;
    переход L_3,
  иначе
    переход L_0.
$L_{(3,1)}$:
  Если c_N — символьный узел, **то**
    $c_N \leftarrow getNodeN(c_S, \$, c_N)$,
  иначе
    $c_N \leftarrow convert(c_N)$.
  $pop(c_U, c_I, c_N)$;
  переход L_0.
$L_{(4,1)}$:
  $c_R \leftarrow getNodeT(c, c_I)$;
  $c_N \leftarrow getNodeI(c_S, 3, c_N, c_R)$;
  $c_I \leftarrow c_I + 1$;
  переход L_3.

Алгоритм 16. Фрагменты для узлов СД на рис. 1

L_1:
  Если $I[c_I] \in \{d\}$, **то**
    переход $L_{(1,1)}$.
  Если $I[c_I] \in \{a\}$, **то**
    $add(c_S, L_{(1,2)}, c_U, c_I, c_N)$;
    $add(c_S, L_{(1,3)}, c_U, c_I, c_N)$;
    переход L_0.
  Если $I[c_I] \in \{b\}$, **то**
    переход $L_{(1,3)}$;
  переход L_0.
L_2:
  Если $I[c_I] \in \{a\}$, **то**
    переход $L_{(2,1)}$.
  Если $I[c_I] \in \{b\}$, **то**
    переход $L_{(2,2)}$;
  переход L_0.
L_3:
  Если $I[c_I] \in \{c, \perp\}$, **то**
    переход $L_{(3,1)}$;
  переход L_0.
L_4:
  Если $I[c_I] \in \{c\}$, **то**
    переход $L_{(4,1)}$;
  переход L_0.

Заключение

Предложенный алгоритм позволяет строить компактные и эффективные по времени работы синтаксические анализаторы по синтаксическим диаграммам с многовходовыми компонентами. Для детерминированной СД временная сложность полученного анализатора — $O(m)$, где m — длина входной цепочки, при условии, что поиск узлов в лесу разбора и стеке организован за независимое от m время. Если вывод заданной цепочки в произвольной СД детерминирован, то время работы анализатора также линейно относительно длины цепочки. Для организации быстрого поиска узлы могут храниться в многомерных массивах, размер которых зависит от длины входной цепочки и количества узлов диаграммы. Недостатком этого подхода является большой объём памяти, занимаемой таким массивом.

Предложенный алгоритм может быть применён к СД произвольной структуры, что исключает необходимость её предварительной обработки.

Алгоритм построения синтаксических анализаторов может быть использован как в системах автоматизированного построения трансляторов, так и при «ручном» проектировании. Построенные по предложенному алгоритму синтаксические анализаторы могут применяться для анализа любых контекстно-свободных языков, включая недетерминированные и неоднозначные.

ЛИТЕРАТУРА

1. *Grau E. A.* Recursive processes and ALGOL translation // Comm. ACM. 1961. V. 4. P. 10–15.
2. *Scott E. and Johnstone A.* GLL parsing // Electr. Notes Theor. Comput. Sci. 2010. V. 253. P. 177–189.
3. *Scott E. and Johnstone A.* GLL parse-tree generation // Sci. Comput. Programming. 2013. V. 78. P. 1828–1844.
4. *Рязанов Ю. Д.* Минимизация синтаксических диаграмм с многовходовыми компонентами // Прикладная дискретная математика. 2018. № 41. С. 85–97.
5. *Рязанов Ю. Д., Севальнева М. Н.* Анализ синтаксических диаграмм и синтез программ-распознавателей линейной сложности // Научные ведомости БелГУ. Сер. История. Политология. Экономика. Информатика. 2013. № 8. С. 128–136.
6. *Рязанов Ю. Д., Крамаренко П. В.* Графовый способ анализа синтаксических диаграмм // Научный электронный архив. `http://econf.rae.ru/article/8214`. 2014.
7. *Поляков В. М., Рязанов Ю. Д.* Алгоритм построения нерекурсивных программ-распознавателей линейной сложности по детерминированным синтаксическим диаграммам // Вестник БГТУ им. В. Г. Шухова. 2013. № 6. С. 194–199.
8. *Рязанов Ю. Д.* Преобразование недетерминированных синтаксических диаграмм в детерминированные // Вестник Воронежского госуниверситета. Сер. Системный анализ и информационные технологии. 2015. № 1. С. 139–147.
9. *Рязанов Ю. Д.* Способ устранения конфликтов типа «переход — выход» в синтаксических диаграммах // Вестник Воронежского госуниверситета. Сер. Системный анализ и информационные технологии. 2015. № 4. С. 130–137.
10. *Tomita M.* Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publ., 1985.
11. *Tomita M.* Graph-structured stack and natural language parsing // 26th Ann. Meeting of the Association of Computational Linguistics, 7–10 June 1988, Buffalo, New York, USA. P. 249–257.

REFERENCES

1. *Grau E. A.* Recursive processes and ALGOL translation. Comm. ACM, 1961, vol. 4, pp. 10–15.
2. *Scott E. and Johnstone A.* GLL parsing. Electr. Notes Theor. Comput. Sci., 2010, vol. 253, pp. 177–189.
3. *Scott E. and Johnstone A.* GLL parse-tree generation. Sci. Comput. Programming, 2013, vol. 78, pp. 1828–1844.
4. *Ryazanov Yu. D.* Minimizaciya sintaksicheskih diagramm s mnogovhodovymi komponentami [Minimization syntax diagrams with multiport components]. Prikladnaya Diskretnaya Matematika, 2018, no. 41. pp. 85–97. (in Russian)
5. *Ryazanov Yu. D. and Seval'neva M. N.* Analiz sintaksicheskih diagramm i sintez programm-raspoznavatelej linejnoj slozhnosti [The analysis of syntax diagrams and automatic generation of linear-time programs-recognizer]. Belgorod State University Scientific Bull. Ser. History. Political Science. Economics. Inform. Technologies, 2013, no. 8, pp. 128–136. (in Russian)
6. *Ryazanov Yu. D. and Kramarenko P. V.* Grafovyj sposob analiza sintaksicheskih diagramm [Graph method for parsing syntax diagrams]. `http://econf.rae.ru/article/8214`, 2014. (in Russian)
7. *Polyakov V. M. and Ryazanov Yu. D.* Algoritm postroeniya nerekursivnyh programm-raspoznavatelej linejnoj slozhnosti po determinirovannym sintaksicheskim diagrammam [Algorithm for not recursive linear-time programs-recognizer design from deterministic syntax diagrams]. Bull. BSTU named after V. G. Shukhov, 2013, no. 6, pp. 194–199. (in Russian)
8. *Ryazanov Yu. D.* Preobrazovanie nedeterminirovannyh sintaksicheskih diagramm v determinirovannye [Converting nondeterministic syntax diagrams to deterministic ones]. Bull. Voronezh State University. Ser. Systems Analysis and Inform. Technology, 2015, no. 1, pp. 139–147. (in Russian)
9. *Ryazanov Yu. D.* Sposob ustraneniya konfliktov tipa "perekhod — vyhod" v sintaksicheskih diagrammah [Method for resolving conflicts of type "shift — reduce" in syntax diagrams]. Bull. Voronezh State University. Ser. Systems Analysis and Inform. Technology, 2015, no. 4, pp. 130–137. (in Russian)
10. *Tomita M.* Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publ., 1985.
11. *Tomita M.* Graph-structured stack and natural language parsing. 26th Ann. Meeting of the Association of Computational Linguistics, 7–10 June 1988, Buffalo, New York, USA, pp. 249–257.