Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Факультет систем управления (ФСУ)

Кафедра автоматизированных систем управления (АСУ)

СИНТАКСИЧЕСКИЙ АНАЛИЗ С ИСПОЛЬЗОВАНИЕМ LL-ГРАММАТИК

Лабораторная работа №3 по дисциплине

«Теория языков программирования и методы трансляции»

Выполнил: студент гр. 430-2

_____А.А. Лузинсан

«____»_____2023 г.

Проверил: к.т.н., доц. каф. АСУ ТУСУР

_____ В.В. Романенко

«____»_____2023 г.

Томск 2023

Сокращения, обозначения, термины и определения

КС — контекстно-свободная грамматика;

$\perp$ - маркер конца цепочки, который должен присутствовать в конце каждой анализируемой цепочки;

Нетерминалы N — элементы грамматики, которые встречаются слева от знака вывода в порождающих правилах;

Терминалы Σ — все элементы грамматики, за исключением нетерминалов N и символа пустой цепочки e.

Оглавление

Введение

Цель: научиться применять на практике такие средства синтаксического анализа, как контекстно-свободные грамматики типа LL(1).

Задание: написать программу, которая должна читать входные данные и описание грамматики из текстовых файлов (например, имеющих имена «input.txt» и «grammar.txt» соответственно), и выдавать результат работы в текстовый файл (например, имеющий имя «output.txt»). Для ввода и вывода данных допускается использование в программе визуального интерфейса вместо файлового ввода/вывода.

Вариант 1. На вход программы подается описание функций/пространств имён на выбранном языке (Pascal, C++ и т. д.), а также описание LL(1)-грамматики. Программа должна проверить, является ли описание функций/пространств имён корректным с точки зрения заданной грамматики и не содержатся ли в нём конфликты имён.

Таким образом, задание разбивается на две части:

- Проверка синтаксиса.
- Проверка семантики.

Семантика зависит от выбранного языка, и поэтому ее проверка жестко привязана к анализатору. Грамматика же должна быть универсальной, т.е. должна позволять задавать любые правила для разбора структур/записей (и не только). Например, должны быть доступны изменения: ключевых слов, знаков пунктуации, правил разбора идентификаторов, а также добавление новых языковых конструкций и т. п.

1 КРАТКАЯ ТЕОРИЯ

В данном разделе приведена краткая теория, которая использовалась в ходе выполнения программной части лабораторной работы.

1.1 Синтаксис и семантика описания функций и пространств имён

В качестве языка программирования был выбран C++, описание функций и пространств имён которого подавалось на вход программы.

Функции и процедуры в языке C++ могут быть описаны глобально, либо внутри пространства имен. Пространства имен также могут быть вложенными друг в друга. Описание пространства имен начинается с ключевого слова namespace и имени пространства, являющегося обычным идентификатором. Содержимое пространства имен заключается в фигурные скобки.

Описание функции начинается с типа возвращаемого значения. Поддерживаемый список типов данных следующий: int, double, long, short, char, float. Помимо этого поддерживаются модификаторы размера типа: long, short — и модификаторы знака: signed, unsigned.

Описание процедуры начинается со специального пустого типа void. Имя функции или процедуры является обычным идентификатором. Далее в скобках приводится список параметров, который может:

• Включать описание формальных аргументов, подобно описанию переменных, но разделяемых запятыми, а не точками с запятой. Также, в отличие от обычного описания переменных, после типа может либо вообще не быть идентификатора (прототип функции), либо лишь один идентификатор. Если есть хотя бы один аргумент подобного типа, то последним аргументом может быть «...», что означает переменное число аргументов в списке (эллипс).

• Содержать специальное ключевое слово void, означающее, что функция или процедура не имеет параметров.

• Быть пустым. Это также означает, что аргументов у функции или процедуры нет.

После скобок ставится точка с запятой, если это предварительное описание прототипа процедуры или функции, либо фигурные скобки с телом процедуры или функции, если это ее реализация. Для упрощения синтаксиса в лабораторной работе полагается, что тела процедур и функций являются пустыми. Проверка, что всем прототипам соответствует реализация, не требуется.

Две глобальные реализации функции или процедуры, либо реализации функции и процедуры, находящиеся на одном уровне вложенности, могут иметь одинаковое имя, если в списках их формальных аргументов типы аргументов отличаются. Для прототипов такого ограничения нет – можно описать любое количество совпадающих прототипов процедур или функций.

Конфликтом имен будет ситуация, когда:

• совпадают имена аргументов в списке у одной и той же процедуры или функции (не важно, прототип это или реализация);

• глобально или на одном и том же уровне вложенности описано пространство имен и процедура или функция с одинаковыми именами (не важно, прототип это или реализация);

• глобально или на одном и том же уровне вложенности имеются реализации процедуры или функции с одинаковыми именами и одинаковыми типами аргументов в списках формальных аргументов.

1.2 Правила грамматики

Языки, определяемые детерминированными МП-автоматами, т. е. такими, которые в каждой конфигурации могут сделать не более одного очередного такта, называются детерминированными КС-языками, а их грамматики – КС-грамматиками.

Формальное описание КС-грамматики представлено в формуле 1.1. Её

можно задавать лишь множеством правил при условии, что правила со стартовым символом в левой части записаны первыми. Тогда:

• Нетерминалами N будут те элементы грамматики, которые встречаются слева от знака вывода в порождающих правилах;

• Терминалами Σ будут все остальные элементы грамматики, за исключением символа пустой цепочки e;

• Стартовым символом грамматики S будет нетерминал из левой части первого порождающего правила.

$$\mathrm{M} = (\mathrm{Q},\ \Sigma, P, S), \tag{1.1}$$

1.3 Формирование таблицы разбора LL(1)-грамматики

LL(1)-грамматика — это КС-грамматика, в которой для каждого нетерминала, появляющегося в левой части более одного порождающего правила, множество направляющих символов, соответствующих правым частям альтернативных порождающих правил, – непересекающиеся.

1.3.1 Алгоритм поиска символов-предшественников

Множество терминальных символов-предшественников (от англ. Start) определяется по формуле 1.2:

$$\mathrm{a} \in S(\alpha) \iff \alpha \Rightarrow^* a\beta \tag{1.2}$$

где:

• a – терминал или пустая цепочка, $a \in \Sigma \cup \{e\}$;

• α и β – произвольные цепочки терминалов и/или нетерминалов, α, $\beta \in (N \cup \Sigma)^*$;

• – S(α) – множество символов-предшественников цепочки α.

То есть во множество символов-предшественников входят такие терминалы, которые могут появиться в начале цепочки, выводимой из α. Пустая цепочка также может являться элементом множества S(α), если она выводится из α.

Пусть α = $X_1X_2...X_n$, где $X_i \in N \cup \Sigma\{e\}$. Тогда формальное выражение для нахождения элементов множества предшествующих символов выглядит так, как показано в формуле 1.3:

$$S(\alpha) = \bigcup_{i=1}^{k}(S(X_i) - \{e\}) \cup \Delta, \tag{1.3}$$

где переменные k и Δ раскрываются в формулах 1.4 и 1.5.

$$k = \{ \begin{array}{c} j|e \notin S(X_j) \wedge e \in S(X_i), i < j, \\ n|e \in S(X_i), i = 1,2,...,n \end{array} \tag{1.4}$$

$$\Delta = \{ \begin{array}{c} \{e\}|e \in S(X_i), i = 1,2,...,n, \\ \varnothing|e \notin S(X_j) \wedge e \in S(X_i), i \neq j \end{array} \tag{1.5}$$

То есть k – это номер первого символа цепочки, для которого $e \notin S(X_k)$. Если же пустая цепочка присутствует во всех $S(X_i)$, i = 1,2,...,n, то k = n. Пустая цепочка войдет во множество S(α) только в том случае, если для любого X_i, i = 1,2,...,n, выполняется условие $e \in S(X_i)$. В этом случае получаем Δ = {e}, в противном случае Δ = ∅

Для нахождения множества символов-предшественников для левых частей всех правил грамматики G = (N, Σ, P, S) существует следующий нерекурсивный алгоритм:

1. Для всех правил грамматики $(A \to \alpha) \in P$ положить $S(A) = \varnothing$.

2. Для каждого правила грамматики $(A \to \alpha) \in P$ добавить к множеству S(A) элементы множества S(α) по формуле 1.6 учитывая формулу 1.7.

$$S(A) = S(A) \cup S(\alpha) \tag{1.6}$$

$$S(X_i) = \begin{pmatrix} \{X_i\}|X_i \in \Sigma\{e\}, \\ \bigcup_j S(X_j)|X_i \in N \wedge (X_j \to \beta) \in P \wedge X_j = X_i. \end{pmatrix} \tag{1.7}$$

То есть для терминала или пустой цепочки во множестве S(Xi) будет лишь один элемент – данный терминал или пустая цепочка. Для нетерминала множество S(Xi) получается путем объединения всех множеств символов-предшественников тех правил грамматики, у которых данный нетерминал стоит слева от знака вывода.

3. Если при во время шага 2 хотя бы в одном S(A) появился новый элемент, возвращаемся на шаг 2. Иначе алгоритм заканчивает свою работу.

1.3.2 Алгоритм поиска последующих символов

Множество терминальных последующих символов (от англ. follow) определяется следующим образом:

$$\mathrm{a} \in F(A) \iff \alpha A \beta \Rightarrow^* \alpha A a \gamma \tag{1.8}$$

где:

- a – терминал или признак конца цепочки, $\mathrm{a} \in \Sigma \cup \{\perp\}$;
- α, β и γ – произвольные цепочки терминалов и/или нетерминалов, α, β, $\gamma \in (N \cup \Sigma)^*$;
- A – нетерминал, $\mathrm{A} \in N$;
- F(A) – множество последующих символов для нетерминала A.

Множество последующих символов включает в себя символы, которые могут следовать в выводимых цепочках за нетерминалом A. При этом данное множество не может содержать пустую цепочку. Но если при выводе после нетерминала A входная цепочка может заканчиваться, то множество F(A) будет содержать специальный символ – маркер конца цепочки. Это может быть любой символ, не входящий в алфавит языка Σ, а также не совпадающий по написанию с нетерминалами грамматики N и символом пустой цепочки e.

По сути, для правила грамматики вида $\mathrm{B} \to \alpha A \beta$ во множество F(A) входят элементы множества S(β), кроме e, а если после нетерминала A цепочка β может заканчиваться ($\beta \Rightarrow^* e$), то также и элементы множества F(B), как показано в формуле 1.9.

$$\mathrm{F(A)} = (\mathrm{S}(\beta) - \{e\}) \cup \begin{pmatrix} F(B) | \beta = e \vee e \in S(\beta) \\ \varnothing | \beta \neq e \wedge e \neq S(\beta) \end{pmatrix} \tag{1.9}$$

Для нахождения множества последующих символов для нетерминалов грамматики G = (N, Σ, P, S) существует следующий нерекурсивный алгоритм:

1. Для всех нетерминалов грамматики $\mathrm{A} \in N$ положить $\mathrm{F(A)} = \varnothing$. Для стартового нетерминала положить $\mathrm{F(S)} = \{\perp\}$.

2. Для каждого вхождения нетерминала A в правую часть

порождающих правил грамматики вида $(\mathrm{B} \to \alpha A \beta) \in P$ добавить к множеству F(A) новые элементы по формуле 1.9.

3. Если при выполнении шага 2 хотя бы в одном множестве F(A) появился хотя бы один новый элемент, вернуться на шаг 2. Иначе алгоритм заканчивает свою работу.

1.3.3 Алгоритм поиска множества направляющих символов

Если A – нетерминал в левой части правила, то его направляющими символами T(A) будут символы-предшественники A и все символы, следующие за A, если A может генерировать пустую строку:

$$\mathrm{T(A)} = (\mathrm{S(A)} \text{ - } \{\mathrm{e}\}) \cup \begin{pmatrix} F(A) | e \in S(A) \\ \varnothing | e \neq S(A) \end{pmatrix} \tag{1.10}$$

Для нетерминала A в правой части правила $\mathrm{B} \to \alpha A \beta$ его направляющие символы определяются так, как показано в формуле 1.11.

$$\mathrm{T(A)} = \begin{pmatrix} (S(A\beta) - \{e\}) \cup F(B) | e \in S(A\beta) \\ S(A\beta) | e \neq S(A\beta) \end{pmatrix} \tag{1.11}$$

1.3.4 Алгоритм построения таблицы разбора

Алгоритм построения таблицы разбора следующий:

1. Разметить грамматику. При этом порядковые номера $\mathrm{i} \in M$ присваиваются всем элементам грамматики, от первого правила к последнему, от левого символа к правому.

2. Построить таблицу, состоящую из столбцов terminals, jump, accept, stack, return, error.

3. Заполнить ячейки таблицы.

3.1. Для нетерминала множество terminals совпадает с множеством направляющих символов. Для терминала множество terminals включает лишь сам терминал. Для пустой цепочки множество terminals совпадает с множеством направляющих символов соответствующего правила, у которого данная пустая цепочка стоит в правой части. Формально, нахождение

терминалов представлено в формуле 1.12.

$$\text{terminals}_i = \begin{pmatrix} T(X_i)|X_i \in N, \\ \{X_i\}|X_i \in \Sigma, \\ T(X_j)|X_i = e \wedge (X_j \to X_i) \in P \end{pmatrix} \quad (1.12)$$

3.2. Переход jump от нетерминала в левой части правила осуществляется к первому символу правой части этого правила. Переход от нетерминала в правой части правила осуществляется на такой же нетерминал в левой части. Причем если имеется несколько альтернатив соответствующего порождающего правила, переход осуществляется к первой из альтернатив. От терминала, не последнего в цепочке правой части правила, переход осуществляется к следующему символу цепочки. Если терминал завершает цепочку либо это символ e, то значение jump равно нулю. Формально, нахождение множества jump представлено в формуле 1.13.

$$\text{jump}_i = \begin{pmatrix} k|i \in M_L \wedge (X_i \to X_k\alpha) \in P, \\ k|X_i \in N \wedge i \notin M_L \wedge X_k = X_i \wedge j \in M_L \wedge j-1 \notin M_L, \\ i+1|i \in \Sigma \wedge i \notin M_R, \\ 0|i \in \Sigma \cup \{e\} \wedge i \in M_R \end{pmatrix} \quad (1.13)$$

3.3. Символ принимается (accept), если это терминал, как гласит формула 1.14.

$$\text{accept}_i = \begin{pmatrix} true|X_i \in \Sigma, \\ false|X_i \notin \Sigma \end{pmatrix} \quad (1.14)$$

3.4. Номер строки таблицы разбора помещается в стек (stack), если соответствующий символ грамматики – нетерминал в правой части порождающего правила, но не в конце цепочки правой части:

$$\text{stack}_i = \begin{pmatrix} true|X_i \in N \wedge i \notin M_L \wedge i \notin M_R, \\ false|X_i \notin N \vee i \in M_L \vee i \in M_R \end{pmatrix} \quad (1.15)$$

3.5. Возврат (return) по стеку при разборе осуществляется, если символ грамматики является терминалом, расположенным в конце цепочки правой части порождающего правила, или символом e (т.е. когда $\text{jump}_i = 0$):

$$\text{return}_i = \begin{pmatrix} true|i \in \Sigma \cup \{e\} \wedge i \in M_R, \\ fasle|i \notin \Sigma\{e\} \vee i \notin M_R \end{pmatrix} \quad (1.16)$$

3.6. Ошибка (error) при разборе не генерируется, если следующий

символ грамматики находится в левой части альтернативного порождающего правила:

$$\mathrm{error}_i = \begin{pmatrix} false | i \in M_L \wedge i+1 \in M_L \\ true | i \notin M_L \vee i+1 \notin M_L \end{pmatrix} \quad (1.17)$$

1.4 Алгоритмы разбора входной строки

Для разбора цепочки α = $a_1a_2...a_n\perp$ нам потребуется магазин (стек) M. Номер текущей строки таблицы разбора обозначим как i, номер текущего символа во входной строке – k.

Алгоритм разбора цепочки по таблице:

1. Положить i := 1.
2. Положить k := 1.
3. M ← 0.
4. Если $a_k \in terminals_i$, то:
   - 4.1. Если $accept_i$ = true, то k := k + 1.
   - 4.2. Если stacki = true, то M ← i.
   - 4.3. Если $return_i$ = true, то:
     - 4.3.1. M → i;
     - 4.3.2. 4.3.2. Если i = 0, то перейти на шаг 6;
     - 4.3.3. 4.3.3. i := i + 1;
     - 4.3.4. Вернуться на шаг 4.
   - 4.4. Если $jump_i \neq 0$, то:
     - 4.4.1. i := $jump_i$;
     - 4.4.2. Вернуться на шаг 4.
   - 4.5. Иначе если $error_i$ = false, то переходим к следующей строке:
     - 4.5.1. i := i + 1;
     - 4.5.2. Вернуться на шаг 4.
   - 4.6. В противном случае разбор окончен. Если при этом стек M пуст, а $a_k=\perp$, то разбор завершен успешно. Иначе цепочка содержит синтаксическую ошибку и k – позиция этой ошибки.

2 РЕЗУЛЬТАТЫ РАБОТЫ

2.1 Описание правил LL(1)-грамматики

Для грамматики, представленной в приложении А.1, её описание выглядит следующим образом:

• N = {K0, S, REQ_S, INIT, BODY, MOD_TYPE, MOD, MOD_TYPE_NONVOID, TYPE, INT, DOUBLE, LONG, SHORT, MOD_TYPE_NONVOID_PARAMS, TYPE_PARAMS, TAIL_TYPES, COMMA_NON2, INT_PARAMS, INT_REQ_S, DOUBLE_PARAMS, DOUBLE_REQ_S, LONG_PARAMS, LONG_REQ_S, SHORT_PARAMS, SHORT_REQ_S, LENGTH, LONGTYPES, ID_FUNC*, TAIL_ID_FUNC*, ID*, INNERID*, ARRAY, OP*, OP1*, PARAMS0, PARAMS, PARAMS_NONVOID, ID_NON, COMMA_NON, TAIL}

• Σ = {«\s», «namespace», «{«, «}», «(«, «)», «;», «void», «unsigned», «signed», «int», «double», «float», «char», «long», «short», «,», [a-zA-Z_], «[», «]» , [1-9], «...»}

• S = K0

• actions = {<A1>, <A2>, <APPEND>, '<NAMESPASE>, <ADD>, '<\ APPEND>, <RW>, <RW_DOWN>, '<APPEND_DOWN>}

2.2 Результаты формирования таблицы разбора

В результате запуска программы, генерирующей таблицу разбора, было сформировано 3 листа, представляющие собой:

• Временный лист, предназначение которого заключается в итерационном поиске символов-предшественников, последующих символов и множества направляющих символов. В связи с большим объёмом грамматики, в приложениях Б.1-Б.3 приведены только соответствующие части этого листа.

• Лист с действиями, содержимое которого впоследствии копируется в лист с таблицей разбора.

• Лист непосредственно с данными таблицы разбора, являющийся целевым. Содержимое листа представлено в приложении Б.4.

2.3 Программная реализация

Программная реализация представлена в приложении В.1. В качестве языка программирования был выбран python версии 3.11. Вспомогательной библиотекой, реализующей управляющее устройство, явилась библиотека pandas. Графический интерфейс, поддерживающий считывание входного сообщения как вручную, так и из файла, был обеспечен библиотекой dearpygui.

Функционал программы поддерживает две функции: формирование таблицы разбора на основе файла с грамматикой и анализ входной строки на соответствие грамматике, описанной в указываемой таблицы разбора.

Таким образом во время формирования таблицы разбора алгоритм проходится во файлу с грамматикой и построчно считывает правила грамматики. Во время этого процесса данные заносятся в соответствующие структуры:

• если левосторонний символ имеет действие, то данное действие сохраняется во вспомогательном листе actions столько раз, сколько имеется правил у данного нетерминала;

• далее если правосторонние узлы имеют действия, то они также заносятся в вспомогательный лист;

• во время итерации по альтернативам одного левостороннего терминала правила сохраняются в словаре;

• в общий словарь dict_LL сохраняются альтернативные правила по ключу левостороннего терминала;

• инициализируется столбец символов-предшественников, если некоторое правило начинается с терминального узла.

Далее идёт итерационный поиск символов-предшественников в методе

find_start_nodes(). Данный метод проходится по значениям каждого ключа словаря dict_LL, а также по правилам этих значений. Если правило начинается с нетерминального узла, то обращаемся в словарю dict_LL по этому узлу и суммируем сведения о символах-предшественниках. В конце итерации вызывается метод проверки на соответствие предыдущего и текущего столбца. Если изменений не произошло, то символы-предшественники найдены.

Алгоритм поиска последующих символов также проходится по значениям каждого ключа словаря dict_LL. Однако далее алгоритм перебирает каждый узел текущего правила, чтобы назначить последующий символ текущему нетерминалу. Чтобы узнать последующий символ используется по сути поиск символов-предшественников следующего узла, однако, если среди символов-предшественников имеется символ пустой строки e, то алгоритм обращается к множеству стартовых символов последующего символа для текущего последующего символа. Если правило закончилось в некоторый момент, то алгоритм берет последующие символы порождающего правила.

Поиск направляющих символов реализован в точности также, как показано в формулах 1.10 и 1.11. Помимо этого, в процессе поиска направляющих символов для правосторонних узлов одновременно заполняется таблица разбора по приведённым выше формулам.

Помимо прочего, в программе был реализован семантический разбор входных данных, суть которого заключена в применении внедрённых действий. Идея разбора заключается в том, что по завершению считывания идентификатора функции, процедуры, имени пространства имён, типа данных, названия идентификатора и маркера, обозначающего принадлежность функции/пространства имён/прототипа, из содержимого буфера формируется узел дерева, который прикрепляется в текущему родительскому узлу.

После завершения синтаксического разбора идёт разбор данного дерева по следующей технологии:

• если узел дерева является пространством имён, то он рекурсивно спускается вглубь к дочерним узлам;

• если узел является реализацией функции, то собираются идентификаторы функции из списка параметров для проверки на уникальность, а также сравниваются списки [имя_функции, тип1, тип2] со всеми остальными, чтобы проверить на наличие конфликтов в возможных перегрузках функций;

• если узел является прототипом функции, то проверяется только уникальность идентификаторов, если они присутствуют в списке формальных параметров;

• просмотрев все дочерние узлы, и записав их идентификаторы в словарь, проверяется несовпадение пространств имён и имён функций (неважно, прототип это или реализация).

2.4 Тестирование программы

В ходе тестирования программы была проверена работоспособность программы в трёх аспектах: проверка грамматики во время генерации таблицы разбора, проверка входных данных на синтаксическую правильность и на семантическое соответствие заложенным правилам.

Соответственно, на рисунках 2.1 — 2.6 представлены все варианты работы программы с подробным описанием возникшей ошибки или комментарием правильного завершения работы алгоритма.

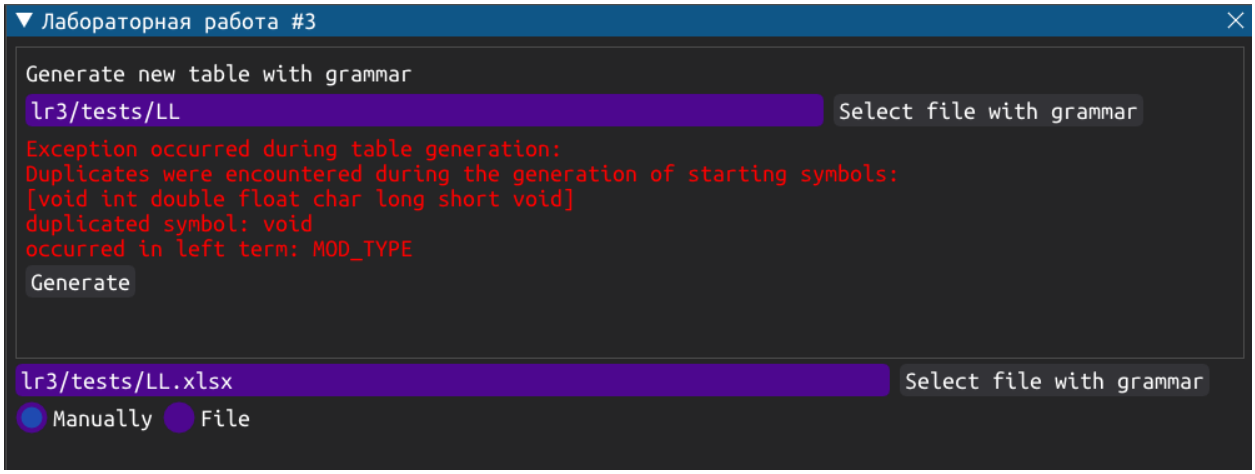

Рисунок 2.1 — Дублирование в списке символов-предшественников

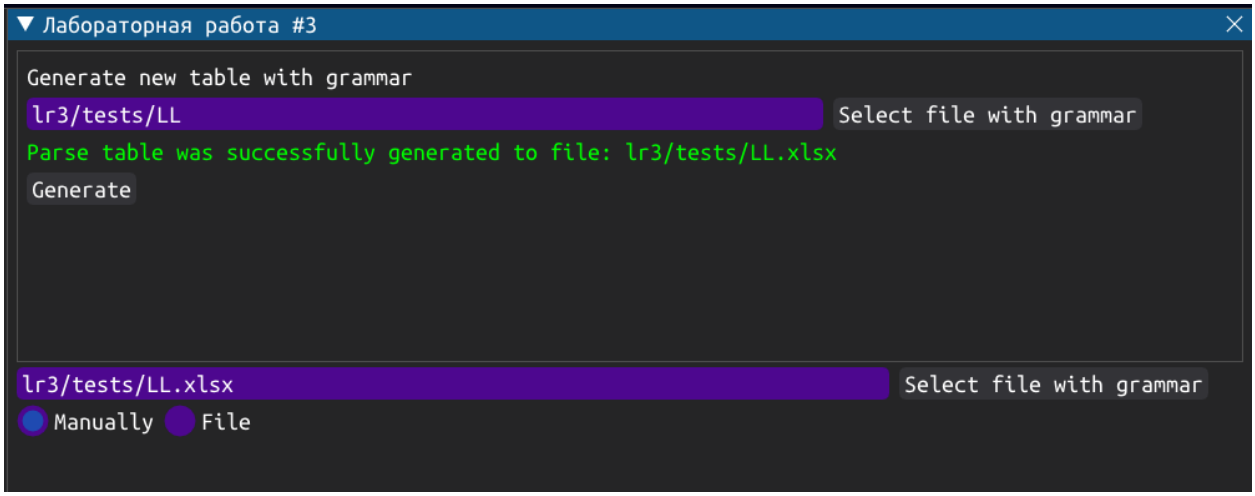

Рисунок 2.2 — Успешная генерация таблицы разбора

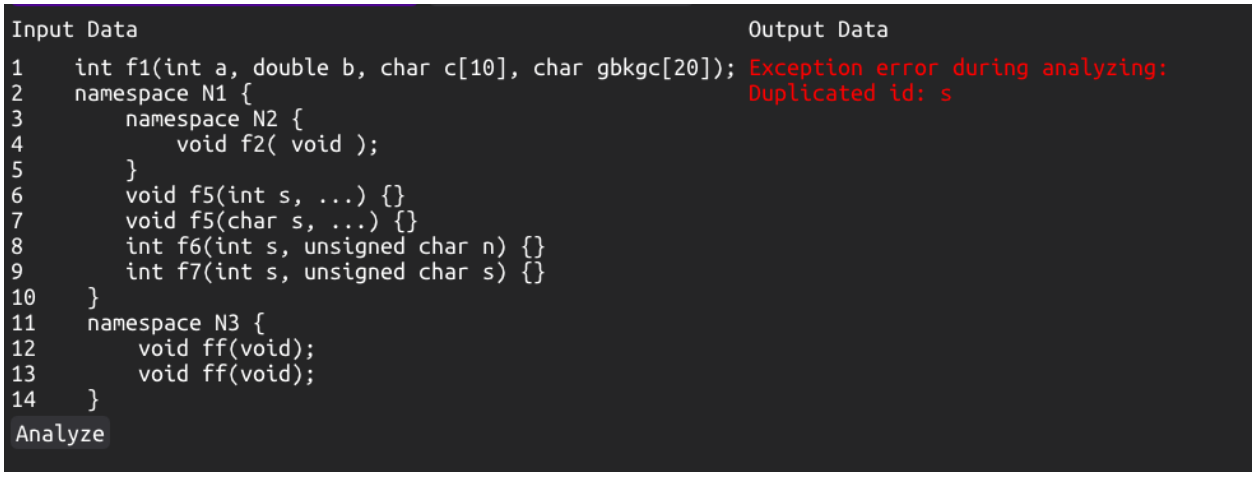

Рисунок 2.3 — Дублирование в списке идентификаторов параметров функции

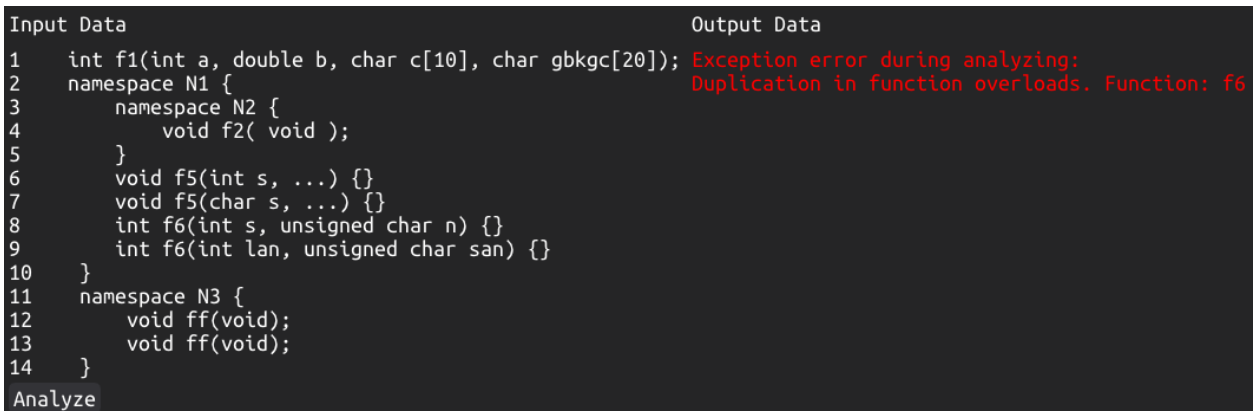

Рисунок 2.4 — Некорректное описание перегрузок функций

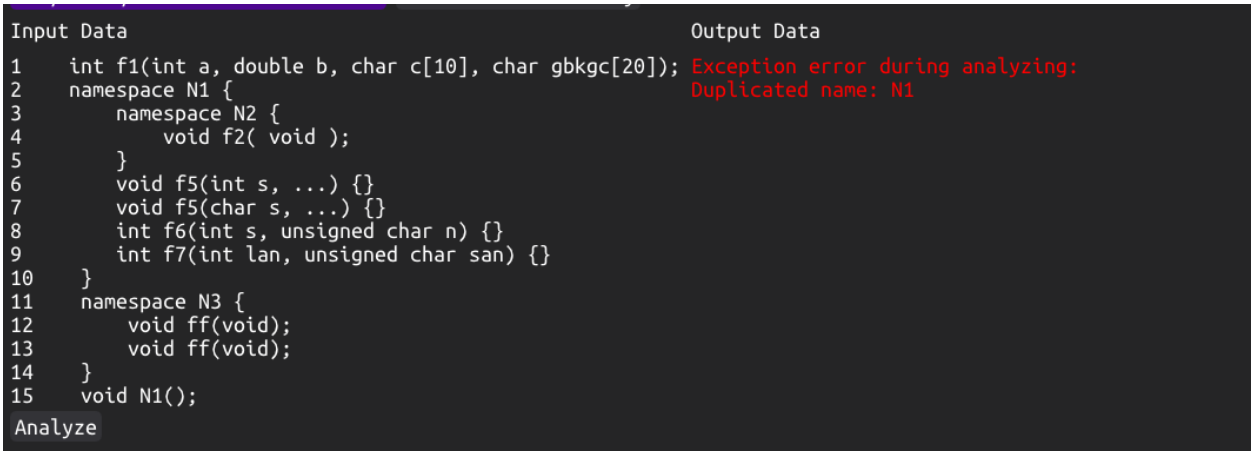

Рисунок 2.5 — Дублирование пространства имён и имени прототипа

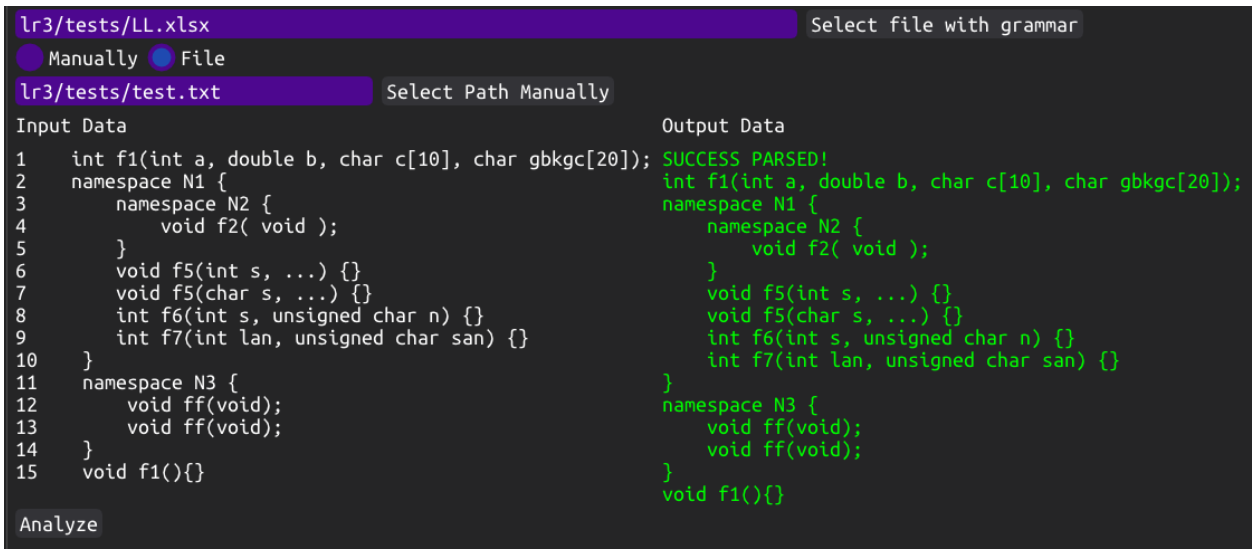

Рисунок 2.6 — Успешное завершение анализа входной строки на синтаксическую и семантическую корректность

Заключение

В результате выполнения лабораторной работы я научилась применять на практике такие средства синтаксического анализа, как контекстно-свободные грамматики типа LL(1).

Список использованных источников

1. Романенко, В. В. Теория языков программирования и методы трансляции // В. В. Романенко, В. Т. Калайда [Электронный ресурс]: научно-образовательный портал ТУСУР. URL: https://edu.tusur.ru/publications/9043 (дата обращения: 12.12.2023).
2. Романенко, В. В. Теория языков программирования и методы трансляции: Учебно-методическое пособие по выполнению лабораторных работ // В. В. Романенко, В. Т. Калайда [Электронный ресурс]: научно-образовательный портал. URL: https://edu.tusur.ru/publications/9044 (дата обращения: 12.12.2023).
3. Образовательный стандарт вуза ОС ТУСУР 01-2021. Работы студенческие по направлениям подготовки и специальностям технического профиля. Общие требования и правила оформления от 25.11.2021 [Электронный ресурс]: научно-образовательный портал. URL: https://regulations.tusur.ru/documents/70 (дата обращения: 12.12.2023).

Приложение А
(обязательное)
LL(1)-грамматика

Листинг А.1 — Список правил грамматики, поддерживающий описание функций и пространств имён языка C++

K0 -> REQ_S INIT | INIT
S -> REQ_S | e
REQ_S -> \s S
INIT -> namespace REQ_S ID_FUNC* S \{ <APPEND> K0 \} <RW_DOWN> K0 | MOD_TYPE ID_FUNC* S \(<APPEND> PARAMS0 \) S BODY K0 | e
BODY -> ; <RW_DOWN> | \{ <A1> S \} <APPEND_DOWN>
MOD_TYPE -> MOD_TYPE_NONVOID | void REQ_S
MOD -> unsigned <A1> | signed <A1>
MOD_TYPE_NONVOID -> MOD REQ_S TYPE | TYPE
TYPE -> int REQ_S INT|double REQ_S DOUBLE|float REQ_S|char REQ_S|long REQ_S LONG|short REQ_S SHORT
INT -> LENGTH REQ_S | e
DOUBLE -> long REQ_S | e
LONG -> LONGTYPES REQ_S | e
SHORT -> int REQ_S | e
MOD_TYPE_NONVOID_PARAMS -> MOD REQ_S <A1> TYPE_PARAMS | TYPE_PARAMS
TYPE_PARAMS -> int <A1> INT_PARAMS | double <A1> DOUBLE_PARAMS |float <A1> TAIL_TYPES |char <A1> TAIL_TYPES |long <A1> LONG_PARAMS | short <A1> SHORT_PARAMS
TAIL_TYPES -> REQ_S ID_NON | COMMA_NON2
COMMA_NON2 <ADD> -> , S TAIL | e
INT_PARAMS -> REQ_S <A1> INT_REQ_S | COMMA_NON2
INT_REQ_S -> LENGTH TAIL_TYPES | ID_NON
DOUBLE_PARAMS -> REQ_S <A1> DOUBLE_REQ_S | COMMA_NON2
DOUBLE_REQ_S -> long <A1> TAIL_TYPES | ID_NON
LONG_PARAMS -> REQ_S <A1> LONG_REQ_S | COMMA_NON2
LONG_REQ_S -> LONGTYPES TAIL_TYPES | ID_NON
SHORT_PARAMS -> REQ_S <A1> SHORT_REQ_S | COMMA_NON2
SHORT_REQ_S -> int <A1> TAIL_TYPES | ID_NON
LENGTH -> long <A1> | short <A1>
LONGTYPES -> int <A1> | double <A1>
ID_FUNC* <A2> -> [a-zA-Z_] <A1> TAIL_ID_FUNC*
TAIL_ID_FUNC* -> [a-zA-Z0-9_] <A1> TAIL_ID_FUNC* | e
ID* -> [a-zA-Z_] <A1> INNERID*
INNERID* -> [a-zA-Z0-9_] <A1> INNERID* | REQ_S ARRAY | ARRAY
ARRAY -> \[S OP* S \] S ARRAY | e <ADD>

OP* -> [1-9] OP1*
OP1* -> [0-9] OP1* | e
PARAMS0 -> REQ_S PARAMS | PARAMS | e
PARAMS -> MOD_TYPE_NONVOID_PARAMS | void <RW> S
PARAMS_NONVOID -> MOD_TYPE_NONVOID_PARAMS
ID_NON <APPEND> -> ID* COMMA_NON | COMMA_NON
COMMA_NON <\APPEND> -> , S TAIL | e
TAIL -> PARAMS_NONVOID | \.\.\. <RW>

Приложение Б
(обязательное)
Таблица разбора

Таблица Б.1 — Часть содержимого листа с поиском символов-предшественников

ПРАВИЛО	START0	START1	START2	START3	START4	START5
K0 -> REQ_S INIT	∅	\s	\s	\s	\s	\s
K0 -> INIT	∅	namespace e	namespace void e	namespace void e	namespace unsigned signed int double float char long short void e	namespace unsigned signed int double float char long short void e
S -> REQ_S	∅	\s	\s	\s	\s	\s
S -> e	e	e	e	e	e	e
REQ_S -> \s S	\s	\s	\s	\s	\s	\s
INIT -> namespace REQ_S ID_FUNC* S \{ K0 \} K0	namespace	namespace	namespace	namespace	namespace	namespace
INIT -> MOD_TYPE ID_FUNC* S \(PARAMS0 \) S BODY K0	∅	void	void	unsigned signed int double float char long short void	unsigned signed int double float char long short void	unsigned signed int double float char long short void
INIT -> e	e	e	e	e	e	e
BODY -> ;	;	;	;	;	;	;
BODY -> \{ S \}	\{	\{	\{	\{	\{	\{
MOD_TYPE -> MOD_TYPE_NONVOID	∅	∅	unsigned signed int double float char long short	unsigned signed int double float char long short	unsigned signed int double float char long short	unsigned signed int double float char long short
MOD_TYPE -> void REQ_S	void	void	void	void	void	void
MOD -> unsigned	unsigned	unsigned	unsigned	unsigned	unsigned	unsigned

Таблица Б.2 — Часть содержимого листа с поиском последующих символов

ПРАВИЛО	FOLLOWS0	FOLLOWS1	FOLLOWS2	FOLLOWS3
K0 -> REQ_S INIT	\} ⊥	\} ⊥	\} ⊥	\} ⊥
K0 -> INIT				
S -> REQ_S	[a-zA-Z_] \[char double float int long namespace short signed unsigned void ⊥	, ; [1-9] [a-zA-Z_] \(\) \.\.\. \[\] \{ \} char double float int long namespace short signed unsigned void ⊥	, ; [1-9] [a-zA-Z_] \(\) \.\.\. \[\] \{ \} char double float int long namespace short signed unsigned void ⊥	, ; [1-9] [a-zA-Z_] \(\) \.\.\. \[\] \{ \} char double float int long namespace short signed unsigned void ⊥
S -> e	⊥	\} ⊥	\} ⊥	\} ⊥
REQ_S -> \s S				
INIT -> namespace REQ_S ID_FUNC* S \ { K0 \} K0				
INIT -> MOD_TYPE ID_FUNC* S \ (PARAMS0 \) S BODY K0	\s	\s	\s	\s
INIT -> e				
BODY -> ;	FOLLOWS0	FOLLOWS1	FOLLOWS2	FOLLOWS3
BODY -> \{ S \}	\} ⊥	\} ⊥	\} ⊥	\} ⊥
MOD_TYPE -> MOD_TYPE_NONVOID				

Таблица Б.3 — Часть содержимого листа с поиском направляющих символов порождающих правил

ПРАВИЛО	DIRECTIONS
K0 -> REQ_S INIT	\s
K0 -> INIT	namespace unsigned signed int double float char long short void \} ⊥
S -> REQ_S	\s
S -> e	, ; [1-9] [a-zA-Z_] \(\) \.\.\. \[\] \{ \} char double float int long namespace short signed unsigned void ⊥
REQ_S -> \s S	\s
INIT -> namespace REQ_S ID_FUNC* S \ { K0 \} K0	namespace
INIT -> MOD_TYPE ID_FUNC* S \ (PARAMS0 \) S BODY K0	unsigned signed int double float char long short void
INIT -> e	\} ⊥
BODY -> ;	;
BODY -> \{ S \}	\{
MOD_TYPE -> MOD_TYPE_NONVOID	unsigned signed int double float char long short
MOD_TYPE -> void REQ_S	void
MOD -> unsigned	unsigned
MOD -> signed	signed
MOD_TYPE_NONVOID -> MOD REQ_S TYPE	unsigned signed
MOD_TYPE_NONVOID -> TYPE	int double float char long short
TYPE -> int REQ_S INT	int
TYPE -> double REQ_S DOUBLE	double
TYPE -> float REQ_S	float
TYPE -> char REQ_S	char
TYPE -> long REQ_S LONG	long
TYPE -> short REQ_S SHORT	short
INT -> LENGTH REQ_S	long short
INT -> e	[a-zA-Z_]
DOUBLE -> long REQ_S	long
DOUBLE -> e	[a-zA-Z_]
LONG -> LONGTYPES REQ_S	int double

Таблица Б.4 — Часть содержимого таблицы разбора

nonterms	terminals	jump	accept	stack	return	error	action
left: K0	\s	4	False	False	False	False	
left: K0	namespace unsigned signed int double float char long short void \} ⊥	6	False	False	False	True	
right: REQ_S	\s	11	False	True	False	True	
right: INIT	namespace unsigned signed int double float char long short void \} ⊥	14	False	False	False	True	
right: INIT	namespace unsigned signed int double float char long short void \} ⊥	14	False	False	False	True	
left: S	\s	9	False	False	False	False	
left: S	, ; [1-9] [a-zA-Z_] \(\) \.\.\. \[\] \ { \} char double float int long namespace short signed unsigned void ⊥	10	False	False	False	True	
right: REQ_S	\s	11	False	False	False	True	
right: e	, ; [1-9] [a-zA-Z_] \(\) \.\.\. \[\] \ { \} char double float int long namespace short signed unsigned void ⊥	0	False	False	True	True	
left: REQ_S	\s	12	False	False	False	True	
right: \s	\s	13	True	False	False	True	
right: S	\s , ; [1-9] [a-zA-Z_] \(\) \.\.\. \ [\] \{ \} char double float int long namespace short signed unsigned void ⊥	7	False	False	False	True	
left: INIT	namespace	17	False	False	False	False	
left: INIT	unsigned signed int double float char long short void	25	False	False	False	False	
left: INIT	\} ⊥	34	False	False	False	True	
right: namespace	namespace	18	True	False	False	True	
right: REQ_S	\s	11	False	True	False	True	
right: ID_FUNC*	[a-zA-Z_]	181	False	True	False	True	

Приложение В
(обязательное)
Листинг программы

Листинг В.1 — Содержимое файла lr3.py

```
import dearpygui.dearpygui as dpg
import regex as re
from __init__ import initialize, select_path
from openpyxl import load_workbook
import openpyxl
from enum import IntEnum
from anytree import Node, RenderTree

class LL:
    table: openpyxl.worksheet.worksheet.Worksheet
    buffer: str
    root: Node
    current_node: Node
    def __init__(self):
        self.buffer = ""
        self.root = Node('root')
        self.current_node = self.root

    class Column(IntEnum):
        LEFT = 1
        TERMS = 2
        JUMP = 3
        ACCEPT = 4
        STACK = 5
        RETURN = 6
        ERROR = 7
        ACTION = 8
    def open_parse_table(self, file_parse_table: str):
        try:
            self.table = load_workbook(filename=file_parse_table, read_only=True)['parse_table']
        except BaseException as err:
            raise FileNotFoundError(err)

    # region Generate parse table
    class ParseTable:
        wb: openpyxl.Workbook
        ws: openpyxl.worksheet.worksheet.Worksheet
        parse_table: openpyxl.worksheet.worksheet.Worksheet
        actions: openpyxl.worksheet.worksheet.Worksheet
        dict_LL: dict
        dict_M: dict
        start_col: int
        follow_col: int
        num_rows: int
        def __init__(self):
```

```
        self.dict_LL = dict()
        self.dict_M = dict()
        self.wb = openpyxl.Workbook()
        self.parse_table = self.wb.create_sheet("parse_table", 0)
        self.parse_table.append(('НЕТЕРМИНАЛЫ', "terminals",
                                 "jump", "accept", "stack", "return", "error", "action"))
        self.ws = self.wb.create_sheet("temp_list", 1)
        self.ws.append(('ПРАВИЛО', "START0"))
        self.actions = self.wb.create_sheet("actions", 2)
        self.actions.append(('ПРАВИЛО', "action"))

    def generate_parse_table(self, file_grammar: str):
        self.parse_raw_rules(file_grammar)
        self.find_start_nodes()
        self.find_follow_nodes()
        self.find_direction_nodes()
        self.find_jumps()
        self.put_actions()
        self.wb.save(f"{file_grammar}.xlsx")

    @staticmethod
    def is_nonterm(s):
        return s[0].isupper()

    def parse_raw_rules(self, file_grammar):
        try:
            with open(file_grammar) as file:
                index = 0
                global_index = 1
                for row in file.readlines():
                    if len(list_splitted := row.split(" ->")) == 2:
                        left, right = list_splitted
                        action = ''
                        if len(left_list := left.split(" ")) == 2:
                            left, action = left_list
                        for rule in right.split('|'):
                            global_index += 1
                            self.actions.append((f"{left} -> ", action))
                        rules = {}
                        check_list = []
                        for rule in right.split('|'):
                            rule = rule.strip()
                            nodes = rule.split(" ")
                            rule = re.sub(r'\s<.*?>', '', rule)
                            for node in nodes:
                                global_index += 1
                                value = f"{node}"
                                if node[0] == '<':
                                    global_index -= 1
                                    action = node
                                    self.actions.cell(row=global_index, column=2, value=action)
```

```
                        else:
                            self.actions.append((value, ""))
                    rules.update({index: rule})
                    state0 = "∅" if LL.ParseTable.is_nonterm(rule) \
                        else rule.split(" ")[0]
                    check_list.append(state0)
                    self.ws.append((f"{left} -> {rule}", state0))
                    index += 1
                LL.ParseTable.check_duplicates(list(filter('∅'.__ne__, check_list)),
                                               left)
                self.dict_LL[left] = rules
        except BaseException as err:
            raise FileExistsError(err)

    def put_actions(self):
        for row in range(2, self.actions.max_row + 1):
            value = self.actions.cell(row=row, column=2).value
            self.parse_table.cell(row=row, column=LL.Column.ACTION, value=value)

    @staticmethod
    def is_equal_cols(rows: int, worksheet: openpyxl.worksheet.worksheet.Worksheet,
                      compare_col1: int, compare_col2: int) -> bool:
        for row in range(2, rows + 1):
            if worksheet.cell(row=row, column=compare_col1).value \
                    != worksheet.cell(row=row, column=compare_col2).value:
                return False
        return True

    @staticmethod
    def check_duplicates(check_list: list, key: str, stage: str = 'starting'):
        for item in check_list:
            if check_list.count(item) > 1:
                    check_list = " ".join([f"{i}\n" if (index+1) % 10 == 0 else i for index, i in
enumerate(check_list)])
                raise GeneratorExit(f"Duplicates were encountered during the generation of {stage}
symbols:"
                                    f"\n[{check_list}]"
                                    f"\nduplicated symbol: {item}"
                                    f"\noccurred in left term: {key}")

    def find_start_nodes(self):
        # print("find_start_nodes")
        self.start_col = 2
        self.num_rows = 0
        while True:
            self.ws.cell(row=1, column=self.start_col + 1, value=f"START{self.start_col - 1}")
            for key in self.dict_LL.keys():
                check_list = []
                for index, rule in self.dict_LL[key].items():
                    new_starts = []
                    if LL.ParseTable.is_nonterm(rule):
```

```
                key_start = rule.split(" ")[0]
                for start_index_row in self.dict_LL[key_start].keys():
                    new_starts.append(self.ws.cell(row=start_index_row + 2,
                                                   column=self.start_col).value)
                new_starts = list(filter('∅'.__ne__, new_starts))
                if len(new_starts) == 0:
                    new_starts.append('∅')
            else:
                new_starts = [self.ws.cell(row=index + 2, column=self.start_col).value]
            check_list += new_starts
            self.ws.cell(row=index + 2, column=self.start_col + 1, value=" ".join(new_starts))
            self.num_rows = index + 2
        check_list = list(filter('∅'.__ne__, check_list))
        LL.ParseTable.check_duplicates(check_list, key)
    if LL.ParseTable.is_equal_cols(self.num_rows, self.ws,
                                   self.start_col, self.start_col + 1):
        break
    self.start_col += 1

def get_row_follow_node(self, key, operation=min):
    return operation(self.dict_LL[key].keys()) + 2
def get_follow_nodes(self, key, col) -> list:
    raw_follows = self.ws.cell(row=self.get_row_follow_node(key, min), column=col).value
    return raw_follows.split(" ") if raw_follows else []
def set_follow_nodes(self, key, col, follows: list):
    self.ws.cell(row=self.get_row_follow_node(key, min),
                 column=col, value=" ".join(follows))

def review_next(self, key, nodes, index_node, follows, base_col):
    next_node = ''
    try:
        next_node = nodes[index_node + 1]
        for start_index_row in self.dict_LL[next_node].keys():
            start_terms: list = self.ws.cell(row=start_index_row + 2,
                                             column=self.start_col).value.split(" ")
            if 'e' in start_terms:
                start_terms.remove('e')
                follows += self.review_next(key, nodes, index_node + 1, follows, base_col)
            else:
                follows += start_terms
    except IndexError as _:
        follows += self.get_follow_nodes(key, base_col)
    except KeyError as _:
        follows.append(next_node)
    follows = list(set(follows))
    follows.sort()
    return follows
def put_follows(self, col: int):
    self.ws.cell(row=1, column=col, value=f"FOLLOWS{col - self.start_col - 1}")
    for key in self.dict_LL.keys():
        for index_rule, rule in self.dict_LL[key].items():
```

```
            nodes = rule.split(" ")
            for index_node, node in enumerate(nodes):
                if LL.ParseTable.is_nonterm(node):
                    follows = self.get_follow_nodes(node, col)
                    follows = self.review_next(key, nodes, index_node, follows, col)
                    self.set_follow_nodes(node, col, follows)

    def find_follow_nodes(self):
        self.start_col += 1
        self.follow_col = self.start_col + 1
        for key in self.dict_LL.keys():
            self.ws.merge_cells(start_row=self.get_row_follow_node(key, min),
                                start_column=self.follow_col,
                                end_row=self.get_row_follow_node(key, max),
                                end_column=self.follow_col)
        self.ws.cell(row=2, column=self.follow_col, value="⊥")
        self.put_follows(self.follow_col)
        self.follow_col += 1
        count_interations = 0
        while True:
            old_value = []
            for key in self.dict_LL.keys():
                self.ws.merge_cells(start_row=self.get_row_follow_node(key, min),
                                    start_column=self.follow_col,
                                    end_row=self.get_row_follow_node(key, max),
                                    end_column=self.follow_col)
                old_value = self.get_follow_nodes(key, self.follow_col - 1)
                self.set_follow_nodes(key, self.follow_col, old_value)
            self.put_follows(self.follow_col)
            if LL.ParseTable.is_equal_cols(self.num_rows, self.ws, self.follow_col — 1,
                                           self.follow_col):
                break
            self.follow_col += 1
            count_interations += 1
            if count_interations > 20:
                raise StopIteration("Oops, the algorithm entered an infinite loop"
                                    f"follows: {old_value}")

    def next_starts(self, key, starts, nodes, index_node):
        try:
            next_node = nodes[index_node + 1]
            for start_index_row in self.dict_LL[next_node].keys():
                starts += self.ws.cell(row=start_index_row + 2,
                                       column=self.start_col).value.split(" ")
                if 'e' in starts:
                    starts.remove("e")
                    starts = self.next_starts(key, starts, nodes, index_node + 1)
        except IndexError as _:
            starts += self.get_follow_nodes(key, self.follow_col)
        except KeyError as _:
            starts.append(next_node)
```

```
    return starts

@staticmethod
def is_last_node(current_index, rule_nodes):
    try:
        if rule_nodes[current_index + 1]:
            return False
    except IndexError as _:
        return True

def find_direction_nodes(self):
    self.ws.cell(row=1, column=self.follow_col + 1, value="DIRECTIONS")
    index_term = 1
    for key in self.dict_LL.keys():
        index_term = self.find_left_direction_nodes(key, index_term)
        index_term = self.find_right_direction_nodes(key, index_term, index_term - 1)

def find_left_direction_nodes(self, key, index_term):
    check_list = []
    for index in self.dict_LL[key].keys():
        index_term += 1
        direction_terms: list = self.ws.cell(row=index + 2,
                                             column=self.start_col).value.split(" ")
        if 'e' in direction_terms:
            direction_terms.remove("e")
            follow_e = self.get_follow_nodes(key, self.follow_col)
            direction_terms += follow_e
        check_list += direction_terms
        self.ws.cell(row=index + 2,
                     column=self.follow_col + 1, value=" ".join(direction_terms))
        self.dict_M.update({(key, index_term): {}})
        self.parse_table.append(
            ("left: " + key, " ".join(direction_terms), "", "False", "False", "False", "False"))
    check_list = list(filter(' ⊥'.__ne__, check_list))
    LL.ParseTable.check_duplicates(check_list, key, 'direction')
    self.parse_table.cell(row=index_term, column=LL.Column.ERROR, value="True")
    return index_term

def find_right_direction_nodes(self, key, index_term, last_left_term):
    for index_rule, rule in self.dict_LL[key].items():
        nodes = rule.split(" ")
        for index_node, node in enumerate(nodes):
            index_term += 1
            accept = "False"
            stack = "False"
            if LL.ParseTable.is_nonterm(node):
                stack = str(not LL.ParseTable.is_last_node(index_node, nodes))
                starts = []
                for start_index_row in self.dict_LL[node].keys():
                    starts += self.ws.cell(row=start_index_row + 2,
                                           column=self.start_col).value.split(" ")
```

```
                    if 'e' in starts:
                        starts.remove("e")
                        starts = self.next_starts(key, starts, nodes, index_node)
                elif node == 'e':
                    starts = self.get_follow_nodes(key, self.follow_col)
                else:
                    starts = [node]
                    accept = "True"
                global_index_rule = index_rule - min(self.dict_LL[key].keys())
                self.dict_M[(key,
                     last_left_term - len(self.dict_LL[key]) + 2 + global_index_rule)].update(
                                {index_term: node})
                self.parse_table.append(("right: " + node, " ".join(starts), "",
                                         accept, stack, "False", "True"))
        return index_term

    def find_jumps(self):
        for key, values in self.dict_M.items():
            self.parse_table.cell(row=key[1],
                                  column=LL.Column.JUMP, value=min(values.keys()))
            for index_node, node in values.items():
                if LL.ParseTable.is_nonterm(node):
                    for root_key in self.dict_M.keys():
                        if root_key[0] == node:
                            self.parse_table.cell(row=index_node,
                                                  column=LL.Column.JUMP, value=root_key[1])
                            break
                else:
                    try:
                        next_node = values[index_node + 1]
                        self.parse_table.cell(row=index_node,
                                              column=LL.Column.JUMP, value=index_node + 1)
                    except KeyError as _:
                        self.parse_table.cell(row=index_node, column=LL.Column.JUMP, value=0)
                        self.parse_table.cell(row=index_node,
                                              column=LL.Column.RETURN, value="True")
# endregion

def parse_value(self, row, name_col: Column):
    return self.table.cell(row=row, column=name_col).value
@staticmethod
def what_is_node(node: Node):
    is_node = 'node'
    for child in node.children:
        if child.name[0] == '{' and child.name[-1] == '}':
            is_node = 'implementation'
        elif child.name == ';':
            is_node = 'prototype'
        elif child.name == '}':
            is_node = 'namespace'
    return is_node
```

```
    @staticmethod
    def check_implementation(node: Node, implementation: list):
        ids = []
        func_args = [node.name]
        for type_id in node.children:
            if type_id.name[0] == '{' and type_id.name[-1] == '}':
                break
            func_args.append(type_id.name)
            if type_id.name == '...':
                break
            id_node = type_id.children
            if id_node:
                ids.append(type_id.children[0].name)
            else:
                    raise NameError(f"Missing identifiers in implementation argument list. Function:
{node.name}")
        for id_name in ids:
            if ids.count(id_name) > 1:
                raise NameError(f"Duplicated id: {id_name}")
        for item in implementation:
            if func_args == item:
                raise TypeError(f"Duplication in function overloads. Function: {node.name}")
        return [func_args]
    @staticmethod
    def check_prototype(node: Node):
        ids = []
        for type_id in node.children:
            if type_id == ';':
                break
            id_node = type_id.children
            if id_node:
                ids.append(type_id.children[0].name)
        for id_name in ids:
            if ids.count(id_name) > 1:
                raise NameError(f"Duplicated id: {id_name}")
    @staticmethod
    def check_semantics(node: Node):
        names = {'implementation': [],
                 'prototype': [],
                 'namespace': [],
                 'node': []}
        implementation = list()
        for child in node.children:
            names[LL.what_is_node(child)] += [child.name]
            match LL.what_is_node(child):
                case 'implementation':
                    implementation += LL.check_implementation(child, implementation)
                case 'namespace':
                    LL.check_semantics(child)
                case 'prototype':
```

```
            LL.check_prototype(child)
    for name in names['namespace']:
        if name in names['prototype'] + names['implementation']:
            raise NameError(f"Duplicated name: {name}")

def apply_func(self, token: str, action: str):
    match action:
        case '<A1>':
            self.buffer += token
        case '<A2>':
            self.buffer = ''
        case '<APPEND>':
            self.current_node = Node(self.buffer, parent=self.current_node)
            self.buffer = ''
        case '<ADD>':
            Node(self.buffer, parent=self.current_node)
            self.buffer = ''
        case '<\APPEND>':
            self.current_node = self.current_node.parent
        case '<RW>':
            Node(token, parent=self.current_node)
            self.buffer = ''
        case '<RW_DOWN>':
            Node(token, parent=self.current_node)
            self.current_node = self.current_node.parent
            self.buffer = ''
        case '<APPEND_DOWN>':
            self.buffer += token
            Node(self.buffer, parent=self.current_node)
            self.buffer = ''
            self.current_node = self.current_node.parent

def analyze(self, input_string: str):
    input_string += ' ⊥ '
    i = 2
    k = 0
    Stack = [0]
    while True:
        terms_str = "|".join(self.parse_value(i, LL.Column.TERMS).split(" "))
        terms = re.compile(terms_str)
        if match := terms.match(input_string, pos=k):
            match = match[0]
            len_shift = len(match)
            self.apply_func(match, self.parse_value(i, LL.Column.ACTION))
            if self.parse_value(i, LL.Column.ACCEPT) == 'True':
                k += len_shift
            if self.parse_value(i, LL.Column.STACK) == "True":
                Stack.append(i + 1)
            if self.parse_value(i, LL.Column.RETURN) == "True":
                i = Stack.pop()
                if i == 0:
```

```
                break
            else:
                continue
        else:  # LL.Column.JUMP
            i = self.parse_value(i, LL.Column.JUMP)
    elif self.parse_value(i, LL.Column.ERROR) == "False":
        i += 1
    else:
        break
if len(Stack) == 0 and match == '⊥':
    self.check_semantics(self.root)
    return f"SUCCESS PARSED!"
else:
    raise SyntaxError(f"FAILED PARSED! at: {k}\n->{input_string[k:]}")
```