Министерство науки и высшего образования Российской Федерации

Муромский институт (филиал)

Федерального государственного бюджетного образовательного учреждения высшего образования

«Владимирский государственный университет   
имени Александра Григорьевича и Николая Григорьевича Столетовых»

Факультет ИТР

Кафедра ПИн

*КУРСОВАЯ РАБОТА*

По Теория автоматов и формальных языков

Тема Транслятор с подмножества языка VB

Руководитель

Кульков Я. Ю.

(фамилия, инициалы)

(подпись) (дата)

Студент ПИН - 121

(группа)

Кокурин Я. Д.

(фамилия, инициалы)

(подпись) (дата)

Муром 2023

В данной курсовой работе необходимо было спроектировать транслятор подмножества языка Visual Basic. В качестве средств разработки приложения была использована среда Microsoft Visual Studio 2022. Язык разработки: С#.

In this course work, it was necessary to design a translator for a subset of the Visual Basic language. The Microsoft Visual Studio 2022 environment was used as the application development tools. Development language: C#.

Содержание

[Введение 6](#_Toc135344337)

[1. Анализ технического задания 7](#_Toc135344338)

[2. Описание грамматики языка. 8](#_Toc135344339)

[3. Разработка архитектуры системы и алгоритмов 11](#_Toc135344340)

[4. Тестирование 19](#_Toc135344341)

[5. Руководство пользователя 24](#_Toc135344342)

[6. Руководство программиста 25](#_Toc135344343)

[Заключение 37](#_Toc135344344)

[Список литературы 38](#_Toc135344345)

[Приложение 1 40](#_Toc135344346)

[Приложение 2 44](#_Toc135344347)

Введение

В настоящее время искусственные языки широко применяются в программировании, а также в других областях. Для того, чтобы корректно составлять структуры всевозможных документов, графических интерфейсов и так далее, используются трансляторы – особые системы, служащие для анализа и интерпретации текстов.

Несмотря на то, что в наше время разработано огромное количество различных языков и их трансляторов, процесс создания новых приложений не останавливается. Это связано с тем, что технологии производства вычислительных систем крайне быстро развиваются, и трансляторы быстро устаревают.

Исходными данными для транслятора служит текст входной программы, то есть некая последовательность символов входного языка программирования, удовлетворяющая синтаксическим требованиям.

Идея транслятора легла в основу создания многих языков программирования, например Visual Basic - процедурного языка с элементами компонентной и структурной парадигмой программирования.

1. Анализ технического задания

В Данной курсовой работе необходимо создать транслятор с подмножества языка Visual Basic.

В проектируемой программе необходимо:

- Обеспечить развернутую диагностику ошибок;

- Реализовать класс транслятора;

- Синтаксический разбор - на основе LR(k)-грамматик;

- Разбор логических выражений выполнять методом Бауэра - Замельзона;

- Поддержка в программе следующих параметров:

1) У идентификатора 8 символов значащие;

2) Не менее трёх директив описания переменных;

3) Простой арифметический оператор;

4) Сложное логическое выражение;

5) Условный оператор if … then … else.

Реализация курсовой работы заключается в написании приложения Windows Forms. В приложении должны быть реализованы:

1. Возможность загрузки анализируемого текста из файла, а также написание этого фрагмента вручную с клавиатуры;
2. Вывод на экран результатов работы лексического анализа, классификатора лексем, восходящего анализатора, а также метода разбора сложных логических выражений.

1. Описание грамматики языка.

Алфавит языка Visual Basic включает следующий набор символов:

- прописные и строчные буквы латинского алфавита;

- цифры от 0 до 9;

- знаки арифметических операций: «+, -, \*, /, |, ^»;

- знаки операций отношения: «=, <,>»;

- знаки препинания и разделители: «, . : ; ( )»;

- символ подчеркивания «\_»;

- не изображаемые символы (пробел, табуляция, переход на новую строку);

В алфавит языка входят также зарезервированные слова, которые не могут быть использованы в качестве имен переменных или процедур. Примеры зарезервированных слов: Dim, sub, integer и т.д.

В Visual Basic существуют 4 категории операций:

Арифметические операции (^(возведение в степень), + (сложение), -(вычитание), \*(умножение), /(деление));

Операции отношений (=(равенство), <>(неравно), >(больше), <(меньше), >=(больше или равно), <=(меньше или равно));

Логические операции (NOT отрицание ("НЕ"), AND логическое умножение ("И"), OR логическое сложение ("ИЛИ"));

Функциональные операции (ABS(x) (абсолютная величина), SQR(x) (квадратный корень), x MOD y (остаток целочисленного деления)).

С помощью перечисленных символов происходит образование функций, выражений и операторов языка. При построении элементов языка используются элементарные конструкции, представляющие собой неделимые символы, образованные из нескольких первичных символов (букв, цифр, специальных знаков) – идентификаторы, числа и служебные слова (терминалы).

Идентификаторы состоят из букв и цифр, причем на первом месте должна стоять буква. Длина ограничена восемью символами.

Служебными словами в данном подмножестве являются: Dim, as, integer, string, double, or, and, if, then, else, end.

Сложное логическое выражение состоит из элементарных конструкций – идентификаторов, знаков операций. Исходя из правил написания операторов языка, можно составить грамматику данного подмножества языка Visual Basic.

Оператор If...Then...Else используется для выполнения определенного оператора или блока операторов в зависимости от значения условия. Операторы If...Then...Else могут быть вложены в любое необходимое число слоев.

Чтобы выполнить только один оператор, когда условие равно True, используется однострочный синтаксис оператора If...Then...Else.

Служебное слово then обозначает ветвь операторов, выполняющихся в случае, если результатом логического выражения является True.

Служебное слово else обозначает ветвь операторов, выполняющихся в случае, если результатом логического выражения является False.

Чтобы выполнить несколько строк кода, необходимо использовать многострочный синтаксис. Этот синтаксис включает оператор End If.

Учитывая вышеперечисленные правила, в ходе работы создана грамматика языка.

G={T,N,P,<программа>}

T={Dim, as, if, then, else, end, integer, double, string, or, and, +, -, \*, /, =, >, <, <,>, (, ), \n,id,lit,expr }

N={<программа>,<спис\_опер>,<опер>,<иниц>,<услов>,<присв>,<спис\_перем>,<тип>,<услов>,<операнд>,<знак>}

P={

<программа>::=<спис\_опер>

<спис\_опер>::=<опер>|<спис\_опер>\n <опер>

<опер>::=<иниц>|<услов>|<присв>

<иниц>::=Dim <спис\_перем> as <тип>

<спис\_перем>::=id|<спис\_перем>, id

<тип>::=integer|double|string

<услов>::= if expr then\n <спис\_опер>\n end if|

if expr then\n <спис\_опер>\n else\n <спис\_опер>\n end if

<присв>::=id=<операнд><знак><операнд>|id=<операнд>

<знак>::=+|-|\*|/

<операнд>::=id|lit

}

1. Разработка архитектуры системы и алгоритмов

В процессе разработки курсового проекта был создан лексический и синтаксический анализаторы, а также реализован метод Бауэра-Замельзона для разбора сложных выражений.

Лексема (лексическая единица языка) – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своем составе других структурных единиц языка. Состав возможных лексем определяется синтаксисом языка программирования. Обычно принято выделять следующие типы лексем: идентификаторы, константы, ключевые слова, ограничители (разделители).

Лексический анализатор – это часть компилятора, которая читает литеры программы и строит из них слова (лексемы) исходного языка. Лексический анализ включает в себя сканирование исходного текста программы, распознавание лексем и их классификацию. Кроме того, на этапе лексического анализа отбрасываются комментарии. Программу, которая выполняет лексический анализ, называют лексическим анализатором или сканером.

Лексический анализатор выделяет из текста лексемы различных типов: идентификаторы, литералы (числовые и символьные константы), разделители. Выделение (сборка) лексемы сопровождается проверкой её правильности. Обнаруженные лексические ошибки фиксируются.

Язык описания лексических единиц в большинстве случаев является регулярным, то есть может быть описан с помощью регулярных грамматик. Распознавателями регулярных языков являются конечные автоматы. Одним из способов описания конечного автомата является графическое его представление в виде маркированного однонаправленного графа, в котором узлы соответствуют состояниям конечного автомата, дуги отображают переходы из одного состояния в другое, а символы маркировки дуг соответствуют функции перехода конечного автомата.

Таблица 1 – Пример работы сканера

|  |  |
| --- | --- |
| Лексема | Предварительный тип |
| Dim | идентификатор |
| a | идентификатор |
| as | идентификатор |
| = | разделитель |
| \n | разделитель |
| 8 | литерал |

Лексический анализ — процесс аналитического разбора входной последовательности символов на распознанные группы — лексемы, с целью получения на выходе идентифицированных последовательностей, называемых «токенами». В простых случаях понятия «лексема» и «токен» идентичны, но более сложные анализаторы дополнительно выполняют классификацию лексем по различным типам («идентификатор», «оператор», «разделитель» и т.п.). Лексический анализ используется в компиляторах и интерпретаторах как предварительный этап перед синтаксическим или семантическим анализом.

Во время лексического анализа исходный текст разбивается на лексемы. При этом идентификаторы и константы, имеющие произвольную длину, заменяются символами фиксированной длины (токенами). Это позволяет облегчить работу последующих фаз компилятора (синтаксического и семантического анализа, оптимизации и др.).

Сканер работает с тремя группами: служебные слова и разделители языка, символические имена (идентификаторы) и литералы (числовые или символьные константы). Таблица служебных слов – это предопределенный фиксированный список, который может содержать и служебные слова и разделители.

На этапе классификации лексем самый долгий процесс – поиск в таблицах. Лучших результатов можно достичь, если применить методы, связанные с использованием хэш-функций и хэш-адресации.

Результатом работы сканера является последовательность кодов лексем. Эту последовательность обычно называют таблицей стандартных символов, так как в ней хранятся стандартизованные представления лексем. Информация в этой таблице расположена в том же порядке, что и в исходной программе.

При описании лексического анализа, важно понимать и разделять три связанных понятия.

Токен - структура, состоящая из имени и набора необязательных произвольных атрибутов. Имя токена – абстрактный символ, представляющий тип лексической единицы, например, <ключевое слово>, <название переменной>, и т.п.

Лексема - представляет собой последовательность символов исходной программы, которая идентифицируется лексическим анализатором как экземпляр токена.

Шаблон - описание вида, который может принимать лексема токена. Лексический анализатор принимает решение о принадлежности лексемы токену на основе шаблона.

Лексический токен (или просто токен) – это строка с присвоенным и, таким образом, идентифицированным значением. Он структурирован как пара, состоящая из имени токена и необязательного значения токена. Имя токена — это категория лексической единицы.

На вход лексического анализатора поступает текст исходной программы, а выходная информация передаётся для дальнейшей обработки синтаксическому анализатору. Каждой выделенной из текста лексеме сканер ставит в соответствие токен вида: ‹имя\_токена, значение\_атрибута›.

<имя\_токена>, представляет собой абстрактный символ, использующийся во время синтаксического анализа, а второй компонент, значение атрибута, указывает на запись в таблице идентификаторов, соответствующую данному токену.

Токенизация — это процесс классификации разделов строки входных символов. Полученные токены затем передаются на следующий этап компиляции, то есть в синтаксический анализатор.

Итогом работы лексического анализатора должен быть список лексем, в котором каждой сопоставлен ее конкретный тип. Далее эта информация будет использована синтаксическим анализатором для проверки принадлежности грамматике.

Таблица 2 – Пример работы лексического анализатора

|  |  |  |
| --- | --- | --- |
| Лексема | Токен | Значение |
| Dim | DIM | - |
| a | IDENTIFIER | a |
| as | AS | - |
| = | EQUAL | - |
| \n | LINEBREAK | - |
| 8 | LITERAL | 8 |

Синтаксическим анализом (разбором) называется процесс, определяющий порождается ли данная строка лексем данной грамматикой.

Синтаксический анализатор может быть построен для любой грамматики. Для любой контекстно свободной грамматики существует анализатор, требующий самое большее O(n3) времени для разбора строки из n лексем. Но для языков, которые встречаются на практике, хватает линейных алгоритмов. Анализаторы языков программирования почти всегда проходят один раз слева направо входной текст, считывая за раз одну лексему.

Большинство методов синтаксического разбора относятся к «восходящим» или «нисходящим» методам. В первом случае построение дерева начинается с корня и продолжается в направлении листьев, в то время как во втором – построение начинается с листьев и развивается по направлению к корню. Популярность анализаторов «сверху-вниз» объясняется тем, что эффективный анализатор можно легко построить «вручную». Анализаторы «снизу-вверх» могут работать с большим классом грамматик и схем перевода, поэтому в программах-генераторах анализаторов всё больше используются восходящие алгоритмы разбора.

Восходящий анализ в классе LR(k)-грамматик.

Восходящий синтаксический анализ для грамматики G = (T, N, P, S) начинает разбор с конкретных слов (лексем) из множества Т, связывая сначала пары слов, затем подсоединяет к этим парам новые слова или другие связанные пары образуя нетерминалы из множества N. Постепенно процесс связывания доходит до начального нетерминала S - то есть все лексемы оказываются связаны в единую структуру.

Восходящие методы синтаксического анализа состоят в том, что в цепочке (промежуточной или терминальной) ищется правая часть очередного правила, которое должно быть заменено своим нетерминалом. Т.е. синтаксическое дерево строится снизу-вверх: в текущем множества «незакрытых» вершин ищется подмножество потомков и над ними «надстраивается» вершина-предок. При этом обход вершин и, аналогичный, просмотр цепочки символов происходит слева- направо.

Задача восходящего анализа и состоит в поиске "редукции" исходной терминальной цепочки в аксиому (нетерминал), где на каждом шаге производится поиск основы в текущую цепочку с последующей заменой ее соответствующим нетерминалом, т.е. применяется "инверсия" некоторого правила данной КС - грамматики: правая часть правила заменяется левой. Нетрудно понять, что если мы заменяем на каждом шаге такой "редукции" самую левую основу, то восстанавливаем с конца к началу правый вывод исходной цепочки, а если самую правую основу, то "реконструируем" левый вывод.

Во - первых, по аналогии с нисходящим разбором можно предположить, что для обнаружения основы достаточно пары символов – последнего символа основы и следующего символа строки. Т.е. для каждой пары символов грамматики однозначно можно сформулировать утверждение, является ли эта пара концом основы или нет. Опять - таки это связано с глубиной просмотра вперед входной строки – она равна 1.

Во-вторых, распознавателю необходим стек, и для него требуется определить функциональное назначение. Поскольку просмотр строки в поисках основы требует сохранения пройденных символов, резонно это делать в стеке. Тогда замена правой части правила (основы) на левую будет также производиться в вершине стека. Сам стек будет хранить «недосвернутую» просмотренную часть цепочки, для которой еще не накоплена основа.

Теперь можно сформулировать основные принципы восходящего разбора c использованием магазинного автомата, именуемого также методом «свёртка-сдвиг»:

1) Первоначально в стек помещается первый символ входной строки, а второй становится текущим;

2) Автомат выполняет два основных действия:

– сдвиг очередного символа из входной строки в стек (с переходом к следующему);

– поиск правила, правая часть которого хранится в стеке. В таком случае производится замена ее на левую – свёртка;

Решение, какое из действий – перенос или свёртка должно быть выполнено на данном шаге, принимается на основе анализа пары символов – символа в вершине стека и очередного символа входной строки. Свёртка соответствует наличию в стеке основы, при ее отсутствии выполняется перенос. Управляющими данными автомата является решающая таблица, содержащая для каждой пары символов грамматики указание на выполняемое действие (свёртка, сдвиг или переход в другое состояние) и сами правила грамматики.

Конфликт относится к типу сдвиг-свёртка, если для одной цепочки допустимы и сдвиг и свёртка. Конфликт относится к типу свёртка-свёртка, если допустимы свёртки по различным правилам.

Если в процессе LR-разбора принять детерминированное решение о сдвиге/свертке удаётся, рассматривая только цепочку α и первые k символов не просмотренной части входной цепочки, говорят, что грамматика обладает LR(k) - свойством.

В LR(k) разборе учитывается k-первых символов не просмотренной части входной цепочки.

Положительным результатом работы автомата будет наличие начального нетерминала грамматики в стеке при пустой входной строке.

Построим граф состояний автомата (Приложение 1), решающую таблицу детерминированного автомата (Приложение 2). На основе анализа решающей таблицы и конфликтов сделаем вывод о принадлежности грамматики к классу LR(k) и определим k.

Для однозначного определения основы по левому набору вхождения правой части некоторого правила вывода необходим 1-буквенный префикс правого набора символов, следовательно грамматика принадлежит к классу LR(1).

Разбор сложных логических выражений осуществляется методом Бауэра-Замельзона.

В методе используются два стека и таблица переходов. Один стек, обозначим его E, используется для хранения операндов, другой стек – Т, для хранения знаков операций.

Над стеком Е выполняются две операции:

К(id) – выбрать элемент с именем id из входного потока, положить на

вершину стека Е, перейти к следующему элементу входного потока;

К(ОР) – извлечь два верхних операнда из стека Е, записать тройку: (ОР,

операнд, операнд) в матрицу арифметического оператора; записать результат на вершину стека Е.

Над стеком Т выполняются операции согласно таблице переходов.

В таблице переходов задаются действия, которые должен выполнить транслятор при разборе выражения

Список действий для логических и арифметических операторов приведён в таблице 3.

Таблица 3 – Список действий для логических и арифметических операторов

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Входной символ | | | | | |
|  |  | $ | ( | < > = | + - or | \* / and | ) |
| Символ на вершине стека | ε | D6 | D1 | D1 | D1 | D1 | D5 |
| ( | D5 | D1 | D1 | D1 | D1 | D3 |
| < > = | D4 | D1 | D2 | D1 | D1 | D4 |
| + - or | D4 | D1 | D4 | D2 | D1 | D4 |
| \* / and | D4 | D1 | D4 | D4 | D2 | D4 |

В таблице символ «$» означает признак конца выражения. В зависимости от размещения сложного выражения в грамматике языка, это может быть символ «;» или лексема «else»

Алгоритм метода Бауэра-Замельзона состоит из следующих этапов:

1) Просматриваем входную строку слева направо;

2) Если текущий элемент – операнд, то выполняем операцию K(операнд).

3) Если текущий элемент – операция, то читаем элемент с вершины стека T, из таблицы переходов выбираем действие, соответствующее паре (элемент c вершины стека, символ входного потока). Выполняем выбранное действие. Возможны шесть действий при прочтении операции ОР из входной строки и операции ОР1 на вершине стека Т:

D1. Записать ОР в стек Т и читать следующий символ строки.

D2. Удалить ОР1 из стека Т и генерировать команду К(ОР1);

Записать ОР в стек Т и читать следующий символ строки.

D3. Удалить ОР1 из стека Т и читать следующий символ строки.

D4. Удалить ОР1 из стека Т и генерировать команду К(ОР1);

D5. Ошибка в выражении. Конец разбора.

D6. Успешное завершение разбора

1. Тестирование

Целью проведения тестирования является подтверждение реализации требуемой функциональной системы. Случаем, когда тестирование прошло успешно является совпадение с ожидаемым результатом.

При прохождении всех элементов лексическим анализатором и при отсутствии ошибок, на экран выводится список лексем.

При обнаружении ошибки анализатором, анализ завершает свою работу, и выводит сообщение на экран, содержащее текст ошибки.

В результате проверки правильности работы программы были получены следующие результаты:

1. Проверка лексического анализатора:

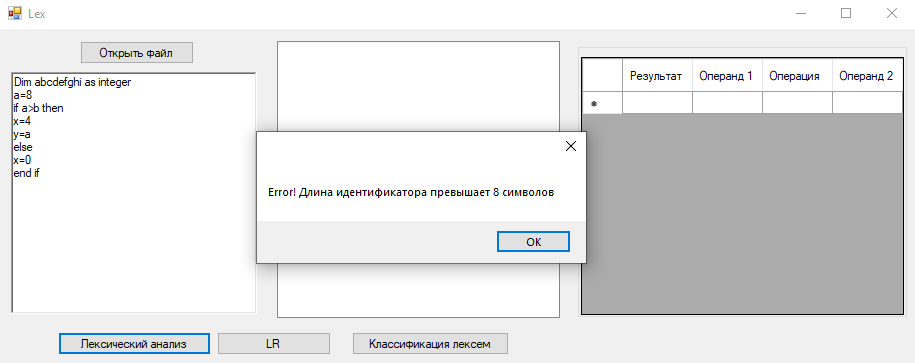


Рисунок 1 – Обнаружение ошибки ввода идентификатора лексическим анализатором

Данная ошибка обусловлена тем, что пользователь ввёл имя идентификатора длиной более 8 символов.

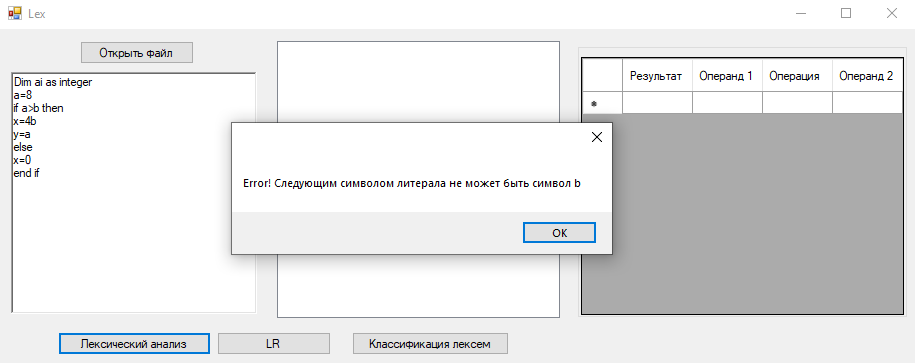


Рисунок 2 – Обнаружение ошибки ввода литерала лексическим анализатором

Данная ошибка обусловлена тем, что пользователь ввёл литерал, вторым символом которого не являлась цифра.

1. Проверка классификатора лексем:

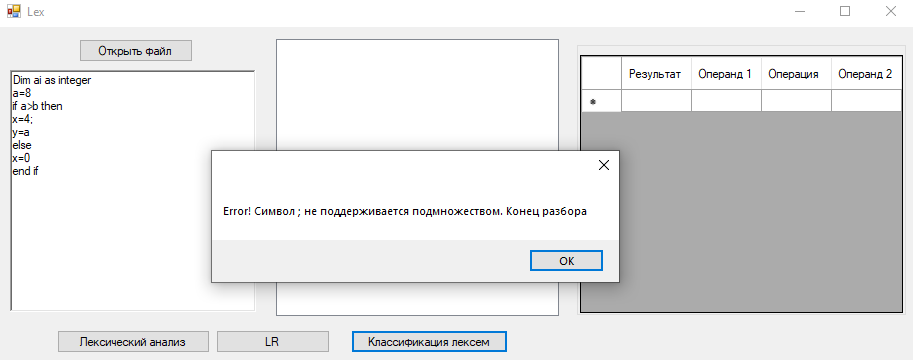


Рисунок 3 – Обнаружение ошибки ввода разделителя, неподдерживаемого подмножеством, классификатором лексем

Данная ошибка обусловлена тем, что пользователь ввёл разделитель, который не поддерживается подмножеством.

1. Проверка синтаксического анализатора:

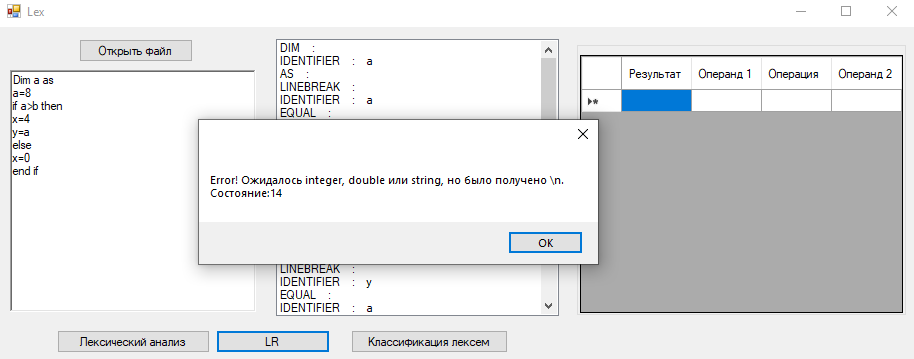


Рисунок 3 – Обнаружение ошибки в правиле <иниц> восходящим анализатором

Данная ошибка возникает в следующем правиле грамматики:

<иниц>::=Dim <спис\_перем> as <тип>

После лексемы «as» должна быть лексема «integer», «double» или «string», соответствующая правилу <тип>::=integer|double|string

Данная ошибка генерируется вызовом метода Error следующим методом:

private void State14()

{

switch (lexemStack.Peek().Type)

{

case TokenType.NETERM:

if (lexemStack.Peek().Value == "<тип>")

GoToState(17);

else

Error("<тип>");

break;

case TokenType.AS:

Shift();

break;

case TokenType.INTEGER:

GoToState(34);

break;

case TokenType.DOUBLE:

GoToState(35);

break;

case TokenType.STRING:

GoToState(36);

break;

default:

Error("integer, double или string");

break;

}

}

1. Проверка подпрограммы разбора сложных логических выражений

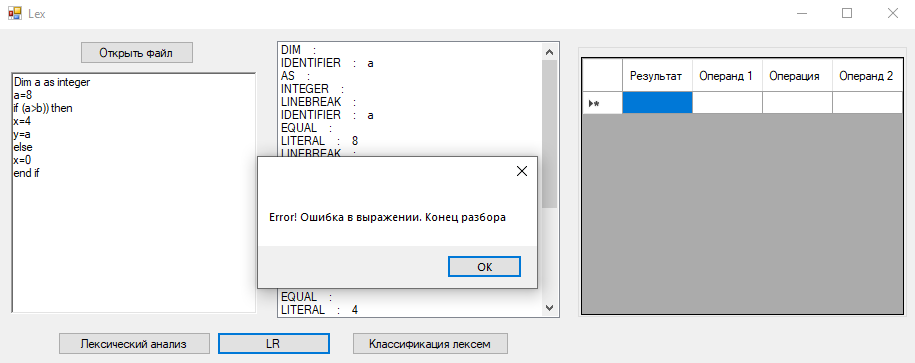


Рисунок 4 – Обнаружение ошибки в сложном логическом выражении подпрограммой разбора сложных логических выражений

Данная ошибка обусловлена тем, что пользователь ввёл ошибочное логическое выражение.

Данная ошибка генерируется вызовом метода D5 следующим методом:

private void Rpar()

{

{

if (T.Count == 0)

D5();

else

switch (T.Peek().Type)

{

case TokenType.LPAR:

D3();

break;

case TokenType.MORE:

D4();

break;

case TokenType.LESS:

D4();

break;

case TokenType.EQUAL:

D4();

break;

case TokenType.PLUS:

D4();

break;

case TokenType.MINUS:

D4();

break;

case TokenType.OR:

D4();

break;

case TokenType.MULTIPLY:

D4();

break;

case TokenType.DIVIDE:

D4();

break;

case TokenType.AND:

D4();

break;

default:

Error("+, -, \*, /, >, <, =, or, and, ( или )");

break;

}

}

}

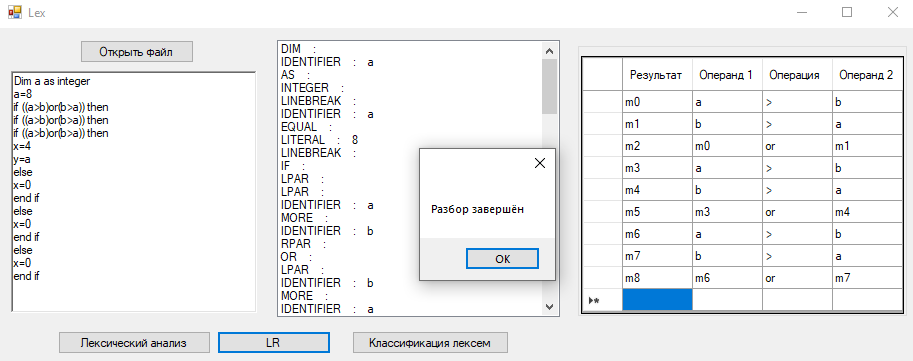


Рисунок 5 – Результат успешного выполнение программы

Результаты, полученные в ходе тестирования разработанного программного продукта, позволяют сделать вывод о том, что разработанная программа соответствует требованиям технического задания.

1. Руководство пользователя

Данное приложение создано для выполнения лексического, синтаксического анализа, а также разбора сложных логических выражений подмножества языка Visual Basic.

В приложении реализованы:

- лексический разбор;

- классификация лексем, полученных в результате лексического разбора;

- синтаксический разбор на основе LR(k)-грамматик;

- разбор сложных логических выражений методом Бауэра – Замельзона;

При открытии приложения пользователь может видеть: кнопку «Открыть файл», с помощью которой может загрузить содержимое текстового файла на форму. Присутствует возможность ввести текст для анализа или редактировать загруженный текст вручную. Кнопка «Лексический анализ», проводит лексический анализ текста, находящегося на форме. Кнопка «Классификация лексем» выводит на экран типизированный список лексем, полученный в результате лексического анализа. В случае если лексема не поддерживается подмножеством, выводится сообщение об ошибке. При нажатии кнопки «LR», проводится восходящий анализ введенного кода, с использованием полученных при выполнении классификации лексем. В случае, если список лексем соответствует грамматике выводится сообщение «Разбор завершён», иначе выведется сообщение с ошибкой. Также, если в анализируемом фрагменте присутствует сложное логическое выражение, то результат его разбора будет выведен на форму в виде таблицы.

1. Руководство программиста

Входными данными для транслятора является текст программы, полученный от пользователя, с синтаксисом подмножества языка программирования VB. Выходными данными транслятора являются таблица лексического разбора программы, список ключевых слов, разделителей, идентификаторов и литералов, используемых в программе после выполнения лексического анализа. В случае обнаружения ошибки в одном из анализаторов программа может вывести сообщение об обнаружении ошибки синтаксическим анализатором. Ниже представлены основные методы программы и их назначения:

public struct Lex

Структура для хранения лексемы.

public string Lexema

Строка, содержащая текстовое значение лексемы.

public string Type

Строка, содержащая тип лексемы.

public Lex (string L, string Type)

Конструктор структуры Lex.

Параметры

L string

Строка, записываемая в поле Lexema структуры Lex.

Type string

Строка, записываемая в поле Type структуры Lex.

public class Analys

Класс для лексического анализа.

private string buff

Строка для накопления символов.

private int lenght

Целочисленная переменная, содержащая длину лексемы.

private char curS

Символьная переменная, содержащая текущий анализируемый символ.

private string stroka

Строка, содержащая анализируемый фрагмент текста.

private List<Lex> strokes

Список для хранения лексем.

public List<Lex> lexes { get { return strokes; } }

Свойство, возвращающее список лексем.

public Analys(string s)

Конструктор класса Analys.

Параметры

s string

Строка, содержащая анализируемый фрагмент текста.

string div = "+-\*/|^=><,.:;()\_\n!@\"'№#$%^&?{}[]"

Строка, содержащая разделители.

public void Razbor()

Метод разбора исходного фрагмента текста на категории идентификатор, литерал и разделитель.

Исключения

LenghtException

Длина идентификатора превышает 8 символов.

LiteralException

Следующим символом литерала не может быть символ.

public enum TokenType

Перечисление возможных типов лексем.

public class Token

Класс для токенизации.

public TokenType Type

Поле для хранения типа токена.

public string Value

Строка, содержащая значение токена.

public Token(TokenType type)

Конструктор класса Token.

Параметры

type TokenType

Поле для хранения типа токена.

private static TokenType[] Delimiters

Массив, содержащий типы разделителей подмножества.

public static bool IsDelimiter(Token token)

Параметры

token Token

Проверяемый токен.

Возвращаемое значение

bool

public static Dictionary<string, TokenType> SpecialWords

Словарь ключевых слов подмножества.

public static bool IsSpecialWord(string word)

Параметры

word string

Проверяемая лексема.

Возвращаемое значение

bool

public static Dictionary<char, TokenType> SpecialSymbols

Словарь разделителей подмножества.

public static bool IsSpecialSymbol(char ch)

Параметры

ch char

Проверяемый символ.

Возвращаемое значение

bool

public class LR

Класс для восходящего синтаксического анализа.

public List<Troyka> operatsii

Список для хранения всех логических операций.

private List<Token> tokens

Список токенов для анализа.

Stack<Token> lexemStack

Стек разбора.

Stack<int> stateStack

Стек для хранения состояний анализатора.

private int nextLex

Целочисленное значение, содержит номер анализируемого токена из списка токенов.

private int state

Целочисленное значение, содержит номер текущего состояния анализатора.

print int lastEXPRind

Целочисленное значение, содержит номер последней логической операции метода разбора сложных логических выражений.

private bool firstEXPR

Логическая переменная, содержит информацию о первом вызове метода разбора сложных логических выражений.

private bool isEnd

Логическая переменная, содержит информацию о том, что список токенов пуст

public LR(List<Token> vvodtoken)

Конструктор класса LR.

Параметры

vvodtoken List<Token>

Список токенов для анализа.

private Token GetLexeme(int nextLex)

Метод, возвращающий токен из списка токенов.

Параметры

nextLex int

Целочисленное значение, содержит номер анализируемого токена из списка токенов.

Возвращаемое значение

Token

private void Shift()

Метод сдвига очередного символа из входной строки в стек с переходом к следующему.

Исключения

ListIsEmptyException

Список лексем пуст, но анализ не был завершён.

private void GoToState(int state)

Метод перевода анализатора в другое состояние

Параметры

state int

Целочисленное значение, содержит номер текущего состояния анализатора.

private void Reduce(int num, string neterm)

Свёртка, метод замены, хранимой в стеке разбора правой части правила на левую.

Параметры

num int

Целочисленное значение, содержит число удаляемых из стеков символов.

neterm string

Строка, содержащая название нетерминала, отправляемого в стек разбора.

private void Expression()

Метод разбора сложных логических выражений.

public void Start()

Метод запуска восходящего анализатора.

private void Error(string ojid)

Метод обработки ошибок.

Параметры

ojid string

Строка, содержащая ожидаемые для разбора символы или правила.

Исключения

NetermException

$"Ожидалось {ojid}, но было получено {lexemStack.Peek().Value}. Состояние:{state}"

TermException

$"Ожидалось{ojid}, но было получено {ConvertLex(lexemStack.Peek().Type)}. Состояние:{state}"

public static string ConvertLex(TokenType type)

Метод преобразования типа токена в строку.

Параметры

type TokenType

Поле для хранения типа токена.

Возвращаемое значение

s string

private void State1() … private void State42()

Методы состояний, действия которых соответствует решающей таблице, приведённой в приложении 2.

В случае ошибки в данных методах вызывается метод обработки ошибок Error, в качестве параметра которому отдаётся строка с ожидаемыми для разбора символами или правила.

public struct Troyka

Структура для хранения логических операций.

public Token operand1;

Поле, содержащее токен, соответствующий первому операнду логической операции.

public Token operand2;

Поле, содержащее токен, соответствующий второму операнду логической операции.

public Token deystvie;

Поле, содержащее токен, соответствующий оператору логической операции.

public Troyka(Token dy, Token op2, Token op1)

Конструктор структуры Troyka.

Параметры

dy Token

Поле, содержащее токен, соответствующий оператору логической операции.

op2 Token

Поле, содержащее токен, соответствующий второму операнду логической операции.

op1 Token

Поле, содержащее токен, соответствующий первому операнду логической операции.

public class Bower

Класс для разбора сложных логических выражений.

private int index

Целочисленное значение, содержит номер текущей операции сложного логического выражения.

public List<Troyka> troyka

Список операций сложного логического выражения.

private List<Token> tokens

Список токенов для разбора.

private Stack<Token> E

Стек для хранения операндов

private Stack<Token> T

Стек для хранения операторов логических операций

private int nextlex

Целочисленное значение, содержит номер анализируемого токена из списка токенов.

public Bower(List<Token> inmet)

Конструктор класса Bower.

Параметры

inmet List<Token>

Список токенов для разбора.

public Bower(List<Token> inmet, int index)

Параметры

inmet List<Token>

Список токенов для разбора.

index int

Целочисленное значение, содержит номер текущей операции сложного логического выражения.

public int Lastindex { get { return index; } }

Свойство, возвращающее номер текущей операции сложного логического выражения.

private Token GetLexeme(int nextLex)

Метод, возвращающий токен из списка токенов.

Параметры

nextLex int

Целочисленное значение, содержит номер анализируемого токена из списка токенов.

Возвращаемое значение

Token

private void Operand()

Метод, реализующий выбор элемента из списка токенов для разбора, после чего добавление его в стек операндов, затем переход к следующему элементу списка.

private void Deystv()

Метод, реализующий извлечение двух элементов из стека операндов и элемента из стека операторов логических операций, запись этих элементов в структуру, которая в последствии отправлена в список операций сложного логического выражения.

Исключения

OperandException

Невозможно выполнить операцию, т. к. количество операндов не удовлетворяет условию.

public void Start()

Метод запуска разбора сложного логического выражения.

Исключения

NotExpectedException

Вызов метода Error, в качестве параметра которому отдаётся строка с ожидаемыми для разбора символами.

private void D1()

Метод, реализующий запись анализируемого токена на вершину стека для хранения операторов логических операций, затем переход к следующему токену для дальнейшего анализа.

private void D2()

Метод, реализующий вызов метода Deystv, затем запись анализируемого токена на вершину стека для хранения операторов логических операций, затем переход к следующему токену для дальнейшего анализа.

private void D3()

Метод, реализующий извлечение элемента из стека для хранения операторов логических операций, затем переход к следующему токену для дальнейшего анализа.

private void D4()

Метод, реализующий вызов метода Deystv.

private void D5()

Метод, реализующий исключение ErrorException: Ошибка в выражении. Конец разбора

private void PlusMinusOr()

Метод, реализующий вызов метода в соответствии со столбцом + - or таблицы 3.

Исключения

NotExpectedException

Вызов метода Error, в качестве параметра которому отдаётся строка с ожидаемыми для разбора символами.

private void MultiplyDivideAnd()

Метод, реализующий вызов метода в соответствии со столбцом \* / and таблицы 3.

Исключения

NotExpectedException

Вызов метода Error, в качестве параметра которому отдаётся строка с ожидаемыми для разбора символами.

private void MoreLessEqual()

Метод, реализующий вызов метода в соответствии со столбцом < > = таблицы 3.

Исключения

NotExpectedException

Вызов метода Error, в качестве параметра которому отдаётся строка с ожидаемыми для разбора символами.

private void Lpar()

Метод, реализующий вызов метода в соответствии со столбцом ( таблицы 3.

Исключения

NotExpectedException

Вызов метода Error, в качестве параметра которому отдаётся строка с ожидаемыми для разбора символами.

private void Rpar()

Метод, реализующий вызов метода в соответствии со столбцом ) таблицы 3.

Исключения

NotExpectedException

Вызов метода Error, в качестве параметра которому отдаётся строка с ожидаемыми для разбора символами.

private void EndList()

Метод, реализующий вызов метода в соответствии со столбцом $ таблицы 3. Логика метода D6 реализована в методе Start, в связи с чем отдельный метод D6 отсутствует.

Исключения

NotExpectedException

Вызов метода Error, в качестве параметра которому отдаётся строка с ожидаемыми для разбора символами.

private void Error(string ojid)

Метод обработки ошибок.

Параметры

ojid string

Строка, содержащая ожидаемые для разбора символы или правила.

Исключения

NetermException

$"Ожидалось {ojid}, но было получено {lexemStack.Peek().Value}. Состояние:{state}"

TermException

$"Ожидалось{ojid}, но было получено {LR.ConvertLex(lexemStack.Peek().Type)}. Состояние:{state}"

Заключение

В результате работы создан транслятор программы на подмножестве языка Visual Basic. Данная программа может выполнять лексический, синтаксический анализы, а также разбор сложного логического выражения. Для синтаксического разбора был использован метод LR(k)-грамматик. Для разбора сложного арифметического выражения использован метод Бауэра-Замельзона.

В ходе работы была составлена грамматика подмножества языка Visual Basic, реализованы методы лексического и синтаксического разбора полученного кода, а также выполнено тестирование программы.

Подводя итоги, можно считать, что разработанный транслятор соответствует требованиям технического задания.

Список литературы

1) Шульга, Т. Э. Теория автоматов и формальных языков: учебное пособие Т. Э. Шульга. — Саратов: Саратовский государственный технический университет имени Ю.А. Гагарина, ЭБС АСВ, 2015. — 104 c. — ISBN 987-5-7433-2968-7. — Текст: электронный // Электронно-библиотечная система IPR BOOKS: [сайт]. — URL: https://www.iprbookshop.ru/76519.html (дата обращения: 15.04.2021). — Режим доступа: для авторизир. пользователей. - DOI: https://doi.org/10.23682/76519 - https://www.iprbookshop.ru/76519.html

2) Алымова, Е. В. Конечные автоматы и формальные языки: учебник / Е. В. Алымова, В. М. Деундяк, А. М. Пеленицын. — Ростов-на-Дону, Таганрог: Издательство Южного федерального университета, 2018. — 292 c. — ISBN 978-5-9275-2397-9. — Текст: электронный // Электронно-библиотечная система IPR BOOKS: [сайт]. — URL: https://www.iprbookshop.ru/87427.html (дата обращения: 15.04.2021). — Режим доступа: для авторизир. пользователей -https://www.iprbookshop.ru/87427.html

3) Пентус, А. Е. Математическая теория формальных языков: учебное пособие / А. Е. Пентус, М. Р. Пентус. — 3-е изд. — Москва: Интернет Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2020. — 218 c. — ISBN 978-5-4497-0662-1. — Текст: электронный // Электроннобиблиотечная система IPR BOOKS: [сайт]. — URL: https://www.iprbookshop.ru/97548.html (дата обращения: 15.04.2021). — Режим доступа: для авторизир. пользователей - https://www.iprbookshop.ru/97548.html

4) Миронов, С. В. Формальные языки и грамматики : учебное пособие для студентов факультета компьютерных наук и информационных технологий / С. В. Миронов. — Саратов: Издательство Саратовского университета, 2019. — 80 c. — ISBN 978-5-292-04613-4. — Текст: электронный // Электронно-библиотечная

система IPR BOOKS: [сайт]. — URL: https://www.iprbookshop.ru/99047.html (дата обращения: 15.04.2021). — Режим доступа: для авторизир. пользователей - https://www.iprbookshop.ru/99047.html

5) Малявко, А. А. Формальные языки и компиляторы: учебник / А. А. Малявко. — Новосибирск: Новосибирский государственный технический университет, 2014. — 431 c. — ISBN 978-5-7782-2318-9. — Текст: электронный // Электронно-библиотечная система IPR BOOKS: [сайт]. — URL: https://www.iprbookshop.ru/47725.html (дата обращения: 15.04.2021). — Режим доступа: для авторизированных пользователей - https://www.iprbookshop.ru/47725.html.

Приложение 1

Таблица 1 – Граф состояний автомата

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Состояние | Предыдущее состояние | Правила грамматики | | | Переход |
| 0 | - | <программа>::=●<спис\_опер>  <спис\_опер>::=●<опер>  <спис\_опер>::=●<спис\_опер>\n <опер>  <опер>::=●<иниц>  <опер>::=●<услов>  <опер>::=●<присв>  <иниц>::=●Dim <спис\_перем> as <тип>  <услов>::=● if expr then\n <спис\_опер>\n end if  <услов>::=● if expr then\n <спис\_опер>\n else\n <спис\_опер>\n end if  <присв>::=●id=<операнд><знак><операнд>  <присв>::=●id=<операнд> | | | 1  2  1  3  4  5  6  7  7  8  8 |
| 1 | 0 | <программа>::=<спис\_опер>●  <спис\_опер>::=<спис\_опер>●\n <опер> | | | x  9 |
| 2 | 0, 18,26 | <спис\_опер>::=<опер>● | | | x |
| 3 | 0,9,18,22,26,28 | <опер>::=<иниц>● | | | x |
| 4 | 0,9,18,22,26,28 | <опер>::=<услов>● | | | x |
| 5 | 0,9,18,22,26,28 | <опер>::=<присв>● | | | x |
| 6 | 0,9,18,22,26,28 | <иниц>::=Dim ●<спис\_перем> as <тип>  <спис\_перем>::=●id  <спис\_перем>::=●<спис\_перем>, id | | | 10  31  10 |
| 7 | 0,9,18,22,26,28 | <услов>::= if ●expr then\n <спис\_опер>\n end if  <услов>::= if ●expr then\n <спис\_опер>\n else\n <спис\_опер>\n end if | | | 11  11 |
| 8 | 0,9,18,22,26,28 | <присв>::=id●=<операнд><знак><операнд>  <присв>::=id●=<операнд> | | | 12  12 |
| 9 | 1 | | <спис\_опер>::=<спис\_опер>\n ●<опер>  <опер>::=●<иниц>  <опер>::=●<услов>  <опер>::=●<присв>  <иниц>::=●Dim <спис\_перем> as <тип>  <услов>::=● if expr then\n <спис\_опер>\n end if  <услов>::=● if expr then\n <спис\_опер>\n else\n <спис\_опер>\n end if  <присв>::=●id=<операнд><знак><операнд>  <присв>::=●id=<операнд> | 13  3  4  5  6  7  7  8  8 | |
| 10 | 6 | | <иниц>::=Dim <спис\_перем> ●as <тип>  <спис\_перем>::=<спис\_перем>●, id | 14  41 | |
| 11 | 7 | | <услов>::= if expr ●then\n <спис\_опер>\n end if  <услов>::= if expr ●then\n <спис\_опер>\n else\n <спис\_опер>\n end if | 15  15 | |

Продолжение таблицы 1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 12 | | 8 | | <присв>::=id=●<операнд><знак><операнд>  <присв>::=id=●<операнд>  <операнд>::=●id  <операнд>::=●lit | 16  16  32  33 | |
| 13 | | 9,22,28 | | <спис\_опер>::=<спис\_опер>\n <опер>● | x | |
| 14 | | 10 | | <иниц>::=Dim <спис\_перем> as ●<тип>  <тип>::=●integer  <тип>::=●double  <тип>::= ●string | 17  34  35  36 | |
| 15 | | 11 | | <услов>::= if expr then●\n <спис\_опер>\n end if  <услов>::= if expr then●\n <спис\_опер>\n else\n <спис\_опер>\n end if | 18  18 | |
| 16 | | 12 | | <присв>::=id=<операнд>●<знак><операнд>  <присв>::=id=<операнд>●  <знак>::=●+  <знак>::=●-  <знак>::=●\*  <знак>::=●/ | 19  x  37  38  39  40 | |
| 17 | | 14 | | <иниц>::=Dim <спис\_перем> as <тип>● | x | |
| 18 | 15 | | <услов>::= if expr then\n ●<спис\_опер>\n end if  <услов>::= if expr then\n ●<спис\_опер>\n else\n <спис\_опер>\n end if  <спис\_опер>::=●<опер>  <спис\_опер>::=●<спис\_опер>\n <опер>  <опер>::=●<иниц>  <опер>::=●<услов>  <опер>::=●<присв>  <иниц>::=●Dim <спис\_перем> as <тип>  <услов>::=● if expr then\n <спис\_опер>\n end if  <услов>::=● if expr then\n <спис\_опер>\n else\n <спис\_опер>\n end if  <присв>::=●id=<операнд><знак><операнд>  <присв>::=●id=<операнд> | | | 20  20  2  20  3  4  5  6  7  7  8  8 |
| 19 | 16 | | <присв>::=id=<операнд><знак>●<операнд>  <операнд>::=●id  <операнд>::=●lit | | | 21  32  33 |
| 20 | 18 | | <услов>::= if expr then\n <спис\_опер>●\n end if  <услов>::= if expr then\n <спис\_опер>●\n else\n <спис\_опер>\n end if  <спис\_опер>::=<спис\_опер>●\n <опер> | | | 22  22  22 |
| 21 | 19 | | <присв>::=id=<операнд><знак><операнд>● | | | x |

Продолжение таблицы 1.

|  |  |  |  |
| --- | --- | --- | --- |
| 22 | 20 | <услов>::= if expr then\n <спис\_опер>\n ●end if  <услов>::= if expr then\n <спис\_опер>\n ●else\n <спис\_опер>\n end if  <спис\_опер>::=<спис\_опер>\n ●<опер>  <опер>::=●<иниц>  <опер>::=●<услов>  <опер>::=●<присв>  <иниц>::=●Dim <спис\_перем> as <тип>  <услов>::=● if expr then\n <спис\_опер>\n end if  <услов>::=● if expr then\n <спис\_опер>\n else\n <спис\_опер>\n end if  <присв>::=●id=<операнд><знак><операнд>  <присв>::=●id=<операнд> | 23  24  13  3  4  5  6  7  7  8  8 |
| 23 | 22 | <услов>::= if expr then\n <спис\_опер>\n end ●if | 25 |
| 24 | 22 | <услов>::= if expr then\n <спис\_опер>\n else●\n <спис\_опер>\n end if | 26 |
| 25 | 23 | <услов>::= if expr then\n <спис\_опер>\n end if● | x |
| 26 | 24 | <услов>::= if expr then\n <спис\_опер>\n else\n ●<спис\_опер>\n end if  <спис\_опер>::=●<опер>  <спис\_опер>::=●<спис\_опер>\n <опер>  <опер>::=●<иниц>  <опер>::=●<услов>  <опер>::=●<присв>  <иниц>::=●Dim <спис\_перем> as <тип>  <услов>::=● if expr then\n <спис\_опер>\n end if  <услов>::=● if expr then\n <спис\_опер>\n else\n <спис\_опер>\n end if  <присв>::=●id=<операнд><знак><операнд>  <присв>::=●id=<операнд> | 27  2  27  3  4  5  6  7  7  8  8 |
| 27 | 26 | <услов>::= if expr then\n <спис\_опер>\n else\n <спис\_опер>●\n end if  <спис\_опер>::=<спис\_опер>●\n <опер> | 28  28 |

Продолжение таблицы 1.

|  |  |  |  |
| --- | --- | --- | --- |
| 28 | 27 | <услов>::= if expr then\n <спис\_опер>\n else\n <спис\_опер>\n ●end if  <спис\_опер>::=<спис\_опер>\n ●<опер>  <опер>::=●<иниц>  <опер>::=●<услов>  <опер>::=●<присв>  <иниц>::=●Dim <спис\_перем> as <тип>  <услов>::=● if expr then\n <спис\_опер>\n end if  <услов>::=● if expr then\n <спис\_опер>\n else\n <спис\_опер>\n end if  <присв>::=●id=<операнд><знак><операнд>  <присв>::=●id=<операнд> | 29  13  3  4  5  6  7  7  8  8 |
| 29 | 28 | <услов>::= if expr then\n <спис\_опер>\n else\n <спис\_опер>\n end ●if | 30 |
| 30 | 29 | <услов>::= if expr then\n <спис\_опер>\n else\n <спис\_опер>\n end if● | x |
| 31 | 6 | <спис\_перем>::=id● | x |
| 32 | 12,19 | <операнд>::=id● | x |
| 33 | 12,19 | <операнд>::=lit● | x |
| 34 | 14 | <тип>::=integer● | x |
| 35 | 14 | <тип>::=double● | x |
| 36 | 14 | <тип>::=string● | x |
| 37 | 16 | <знак>::=+● | x |
| 38 | 16 | <знак>::=-● | x |
| 39 | 16 | <знак>::=\*● | x |
| 40 | 16 | <знак>::=/● | x |
| 41 | 10 | <спис\_перем>::=<спис\_перем>, ●id | 42 |
| 42 | 41 | <спис\_перем>::=<спис\_перем>, id● | x |

Приложение 2

Таблица 1 – Решающая таблица детерминированного автомата

|  |  |  |  |
| --- | --- | --- | --- |
| Состояние | Стек разбора | Вход | Действие |
| 0 | <программа>  ε  <спис\_опер>  <опер>  <иниц>  <услов>  <присв>  Dim  if  id |  | Конец разбора  Сдвиг  Переход 1  Переход 2  Переход 3  Переход 4  Переход 5  Переход 6  Переход 7  Переход 8 |
| 1 | <спис\_опер>  <спис\_опер>  \n | $  \n | Свёртка (-1,<программа>)  Сдвиг  Переход 9 |
| 2 | <опер> |  | Свёртка(-1,<спис\_опер>) |
| 3 | <иниц> |  | Свёртка(-1,<опер>) |
| 4 | <услов> |  | Свёртка(-1,<опер>) |
| 5 | <присв> |  | Свёртка(-1,<опер>) |
| 6 | Dim  <спис\_перем>  id |  | Сдвиг  Переход 10  Переход 31 |
| 7 | If  expr |  | Сдвиг  Переход 11 |
| 8 | Id  = |  | Сдвиг  Переход 12 |
| 9 | \n  <опер>  <иниц>  <услов>  <присв>  Dim  If  id |  | Сдвиг  Переход 13  Переход 3  Переход 4  Переход 5  Переход 6  Переход 7  Переход 8 |
| 10 | <спис\_перем>  as  , |  | Сдвиг  Переход 14  Переход 41 |
| 11 | Expr  then |  | Сдвиг  Переход 15 |
| 12 | =  <операнд>  id  lit |  | Сдвиг  Переход 16  Переход 32  Переход 33 |
| 13 | <опер> |  | Свёртка(-3,< спис\_опер >) |
| 14 | As  <тип>  integer  double  string |  | Сдвиг  Переход 17  Переход 34  Переход 35  Переход 36 |

Продолжение таблицы 1.

|  |  |  |  |
| --- | --- | --- | --- |
| 15 | then  \n |  | Сдвиг  Переход 18 |
| 16 | <операнд>  <операнд>  <знак>  +  -  \*  / | \n,$  +,-,\*,/ | Свёртка(-3,<присв>)  Сдвиг  Переход 19  Переход 37  Переход 38  Переход 39  Переход 40 |
| 17 | <тип> |  | Свёртка(-4,<иниц>) |
| 18 | \n  <спис\_опер>  <опер>  <иниц>  <услов>  <присв>  Dim  If  id |  | Сдвиг  Переход 20  Переход 2  Переход 3  Переход 4  Переход 5  Переход 6  Переход 7  Переход 8 |
| 19 | <знак>  <операнд>  id  lit |  | Сдвиг  Переход 21  Переход 32  Переход 33 |
| 20 | <спис\_опер>  \n |  | Сдвиг  Переход 22 |
| 21 | <операнд> |  | Свёртка(-5,<присв>) |
| 22 | \n  end  else  <опер>  <иниц>  <услов>  <присв>  Dim  If  id |  | Сдвиг  Переход 23  Переход 24  Переход 13  Переход 3  Переход 4  Переход 5  Переход 6  Переход 7  Переход 8 |
| 23 | end  if |  | Сдвиг  Переход 25 |
| 24 | else  \n |  | Сдвиг  Переход 26 |
| 25 | if |  | Свёртка(-8,<услов>) |
| 26 | \n  <спис\_опер>  <опер>  <иниц>  <услов>  <присв>  Dim  If  id |  | Сдвиг  Переход 27  Переход 2  Переход 3  Переход 4  Переход 5  Переход 6  Переход 7  Переход 8 |

Продолжение таблицы 1.

|  |  |  |  |
| --- | --- | --- | --- |
| 27 | <спис\_опер>  \n |  | Сдвиг  Переход 28 |
| 28 | \n  end  <опер>  <иниц>  <услов>  <присв>  Dim  If  id |  | Сдвиг  Переход 29  Переход 13  Переход 3  Переход 4  Переход 5  Переход 6  Переход 7  Переход 8 |
| 29 | end  if |  | Сдвиг  Переход 30 |
| 30 | if |  | Свёртка(-12,<услов>) |
| 31 | id |  | Свёртка(-1,<спис\_перем>) |
| 32 | id |  | Свёртка(-1,<операнд>) |
| 33 | lit |  | Свёртка(-1,<операнд>) |
| 34 | integer |  | Свёртка(-1,<тип>) |
| 35 | double |  | Свёртка(-1,<тип>) |
| 36 | string |  | Свёртка(-1,<тип>) |
| 37 | + |  | Свёртка(-1,<знак>) |
| 38 | - |  | Свёртка(-,1<знак >) |
| 39 | \* |  | Свёртка(-1,<знак >) |
| 40 | / |  | Свёртка(-1,<знак >) |
| 41 | ,  id |  | Сдвиг  Переход 42 |
| 42 | id |  | Свёртка(-3,<спис\_перем>) |