Министерство образования и науки РФ

Белгородский государственный технологический университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

Курсовой проект

по дисциплине «Теория автоматов и формальных языков»  
  
тема: «Синтаксически управляемая трансляция»

**Автор работы:**  
студент группы ПВ-31  
Адаменко И. И.

**Руководитель проекта:**  
доцент  
Рязанов Ю. Д.

Белгород  
2015

Оглавление

[Цель выполнения курсового проекта 3](#_Toc410808071)

[Постановка задачи 4](#_Toc410808072)

[Лексемы исходного языка 4](#_Toc410808073)

[Грамматика исходного языка 5](#_Toc410808074)

[Теоретические сведения 7](#_Toc410808075)

[Лексический анализ 9](#_Toc410808076)

[Синтаксический анализ 9](#_Toc410808077)

[Решение поставленной задачи 10](#_Toc410808078)

[Лексический анализатор 10](#_Toc410808079)

[Синтаксический анализатор 11](#_Toc410808080)

[Тестирование 21](#_Toc410808081)

[Заключение 23](#_Toc410808082)

[Список литературы 24](#_Toc410808083)

[Приложение 25](#_Toc410808084)

# Цель выполнения курсового проекта

Целью выполнения данного курсового проекта является изучение принципов синтаксически управляемой трансляции, а также создание транслятора, который может преобразовать программу на исходном языке в семантически эквивалентную ей программу на языке Си. Транслятор должен обрабатывать программу, не содержащую лексический, синтаксических и семантических ошибок. Если транслятор обрабатывает программу, содержащую ошибки, он должен выводить о них соответствующие сообщения.

# Постановка задачи

1. Получить транслирующую грамматику, задающую перевод текста программы на исходном языке, грамматика которого представлена заданием, в семантически эквивалентную программу на языке Си. В программе на языке Си не должно быть выражений (арифметических и логических), содержащих более одной операции, и структурированных операторов управления. Управление в программе на языке Си задаётся только условными и безусловными переходами на метки.
2. Преобразовать транслирующую грамматику так, чтобы грамматика входного языка принадлежала классу LL(1)-грамматик. Если устранением левой рекурсии и выполнением факторизации не удаётся получить грамматику указанного класса, то допускается внесение изменений в грамматику, не изменяющих или незначительно изменяющих исходный язык.
3. Определить множества первых и следующих для каждого нетерминала грамматики.
4. Определить множества первых для правой части каждого правила грамматики.
5. Определить множества выбора для каждого правила грамматики.
6. Построить нисходящий МП-транслятор.
7. Выбрать метод реализации транслятора.
8. Выбрать способ взаимодействия лексического и синтаксического анализаторов.
9. Написать программу, реализующую заданный транслятор.
10. Транслировать программу на исходном языке, не содержащую лексических, синтаксических и семантических ошибок, в семантически эквивалентную программу на языке Си. Используя компилятор языка Си выполнить результат трансляции.
11. Обработать транслятором программы на исходном языке, содержащие ошибки.

## Лексемы исходного языка

1. Идентификаторы целочисленных переменных представляют собой букву, за которой следуют буквы или цифры. Все идентификаторы имеют один и тот же тип лексемы.
2. Целочисленные константы представляют собой последовательности десятичных цифр. Все целочисленные константы имеют один и тот же тип лексемы.
3. Строковая константа состоит из одной или нескольких частей, соединённых символом «+». Каждая часть представляет собой последовательности символов, не содержащую апострофы, заключённую в апострофы, или последовательность символов, не содержащих кавычки, заключённую в кавычки. Все строковые константы имеют один и тот же тип лексемы, значение лексемы — последовательность символов, образованная конкатенацией всех частей лексемы без ограничивающих символов, заключённая в апострофы. Апострофы внутри части лексемы заменяются парой апострофов.
4. Ключевые слова: var, begin, end, read, write, if, then, else. Каждое слово имеет свой тип лексемы.
5. Арифметические операции: + и −. Каждая операция имеет свой тип лексемы.
6. Логические операции: not, and, or. Каждая операция имеет свой тип лексемы.
7. Операции отношения: =, <>, <, >, <=, >=. Все операции имеют один тип лексемы.
8. Знак оператора присваивания: :=.
9. Специальные знаки: (, ), ,, ;. Каждый знак имеет свой тип лексемы.
10. Комментарии заключаются в фигурные скобки { и }. Могут располагаться после любой лексемы, но не могут находиться внутри лексемы.

## Грамматика исходного языка

1. <программа> → var <переменные> begin <операторы> end
2. <переменные> → идентификатор, <переменные>

| идентификатор

1. <операторы> → <операторы> ; <оператор>

| <оператор>

1. <оператор> → read (строка, идентификатор)

| write (строка, идентификатор)

1. <оператор> → идентификатор := <выражение>
2. <оператор> → if <условие> then <оператор> else <оператор>
3. <выражение> → идентификатор

| число

| <выражение> + <выражение>

| <выражение> − <выражение>

1. <условие> → <условие> or <терм>

| <терм>

1. <терм> → <терм> and <множитель>

| <множитель>

1. <множитель> → not (<условие>)

| (<условие>)

| <отношение>

1. <отношение> → <выражение> операция\_отношения <выражение>

Комментарии:

— идентификатор — имя переменной;

— все переменные — целочисленные;

— число — целочисленная константа;

— строка — строковая константа.

# Теоретические сведения

**Формальный язык** — это множество *цепочек,* образованных по определённым правилам. Множество цепочек может быть конечным или бесконечным.

**Цепочка** — это конечная последовательность символов, которая может заканчиваться *концевым маркером* (┤). *Пустая цепочка* (обозначается Ɛ) не содержит ни одного символа.

**Алфавит языка** — это конечно множество символов, из которых могут быть образованы цепочки языка.

**Терминал**— символ, принадлежащий алфавиту языка.

**Формальная грамматика** G = (N, A, P, S), где:

* N — конечное множество нетерминальных символов (нетерминалов),
* A — конечное множество терминальных символов (терминалов),
* P — конечное множество правил (продукций) вида α → β, где α и β — последовательности терминальных и нетерминальных символов,
* S — начальный нетерминал.

Вывод, на каждом шаге которого правило вывода применяется к самому левому нетерминалу промежуточной цепочки, называется **левым** (или **левосторонним**). Этим выводом мы и будем пользоваться в курсовом проекте.

Грамматики G1 и G2 называются **эквивалентными**, если равны порождаемые ими языки.

Среди символов КС-грамматики можно выделить две группы лишних символов:

1. **Бесплодные нетерминалы** — это нетерминалы, которые не порождают ни одной терминальной цепочки. Такие нетерминалы не могут участвовать в выводе терминальных цепочек.
2. **Недостижимые символы** — это символы (терминалы и нетерминалы), которые не встречаются ни в одной цепочке (терминальной или промежуточной), выводимой из начального нетерминала. Правила, в которые входят лишние символы (в левую или правую их часть), можно исключить из множества правил грамматики.

Правило вида А → Ɛ называется **Ɛ-правилом**. Грамматику, порождающую язык, не содержащий пустую цепочку, можно преобразовать в эквивалентную ей грамматику без Ɛ-правил, а грамматику, порождающую язык, содержащий пустую цепочку, можно преобразовать в эквивалентную ей грамматику с единственным Ɛ-правилом S → Ɛ, где S — начальный нетерминал.

**Циклом (циклическим выводом)** называется вывод вида A => \*A, где A — нетерминал грамматики. Циклический вывод бесполезен. Циклы возможны только в том случае, если в грамматике есть **цепные правила** вида A → B, где A и B — нетерминалы грамматики.

Если грамматика содержит n правил вида A → αi, где и других правил с левой частью A нет, и в грамматике есть правило вида B → βAγ, то его можно заменить на n правила вида B → βαiγ, где . Такое преобразование грамматики называется **заменой**, а нетерминал A в правиле B → βAγ — **заменяемым**.

В правиле грамматики назовём самое левое вхождение символа в его правую часть **краем** правила. Если заменяемый нетерминал является краем правила, то такая замена называется **заменой края**. Преобразование, обратное замене, назовём **выделением нетерминала**. Особым случаем выделения нетерминала является левая факторизация. Если правил грамматики имеют одинаковые левые части, допустим нетерминал А, и правые части начинаются одним или несколькими одинаковыми символами (имеют общий префикс α), т.е. i-ое правило имеет вид А → αβi, где , то можно общий префикс α вынести в отдельное правило А → αВ, где В — новый нетерминал, и добавить n правил вида В → βi, где . Такое преобразование называется **левой факторизацией***.* Результат применения левой факторизации неоднозначный, зависит от выбора префикса, выносимого в отдельное правило (например, в качестве префикса можно взять один символ или общий префикс наибольшей длины) и количества правил, участвующих в факторизации.

Правило А → γ называется **рекурсивным**, если существует вывод γ => \*αАβ. Если α = Ɛ и β, то правило А → γ называется **леворекурсивным**. Правило называется **самолеворекурсивным**, если его край совпадает с левой частью. Самолеворекурсивное правило также является и леворекурсивным.

Определить принадлежность заданной (входной) цепочки заданному регулярному языку можно с помощью конечного распознавателя. **Конечный** **распознаватель** *—* это конечный автомат Рабина-Скотта, который определяется пятёркой: A = <S, X, s0, δ, F>, где:

* S — конечное непустое множество состояний,
* X — конечное непустое множество входных символов,
* s0 — начальное состояние,
* δ: SxX → S — функция переходов, которая паре (состояние, входной символ) ставит в соответствие новое состояние (состояние перехода),
* F — множество допускающих состояний.

Конечный распознаватель допускает входную цепочку, если она переводит его из начального состояния в одно из допускающих. Конечный распознаватель, который не может одновременно находиться более чем в одном состоянии, называется **детерминированным**. Конечный распознаватель, который может одновременно находиться более чем в одном состоянии, называется **недетерминированным***.* В данном курсовом проекте используется детерминированный конечный распознаватель.

## Лексический анализ

Программы для обработки языков программирования можно разделить на трансляторы и интерпретаторы.

**Транслятор**— это программа, которая переводит программу на исходном (входном) языке в семантически эквивалентную ей программу на выходном языке. Транслятор, который переводит программу на исходном языке в эквивалентную ей программу на языке машинных команд или на языке ассемблера, называют **компилятором**.

**Интерпретатор**– это программа, которая воспринимает программу на исходном языке и выполняет её.

**Лексема**— это структурная единица языка, которая состоит из символов алфавита и не содержит в себе других структурных единиц языка. **Таблица лексем**представляет собой последовательность элементов, каждый из которых содержит тип лексемы и её значение.

Лексический анализатор представляет собой конечный распознаватель, способный обнаруживать границы лексем и выполнять действия, связанные с лексемами.

## Синтаксический анализ

**Синтаксический анализ**— процесс сопоставления линейной последовательности лексем естественного или формального языка с его формальной грамматикой

**Синтаксический анализатор** — это программа или часть программы, выполняющая синтаксический анализ.

# Решение поставленной задачи

## Лексический анализатор

Для программной реализации лексического анализатора был выбран компиляционный способ. На основе исходного языка была построена программа, в которой за состояния отвечают подпрограммы. Так же созданы функции для состояний ошибки. Определение границы лексемы осуществляется по методу выбора лексемы наибольшей длины, то есть, очередной символ исходного текста добавляется в лексему всегда, когда он может быть туда добавлен.

Конечный распознаватель заданного языка имеет следующий вид:

Алгоритм работы следующий:

1. Из входной последовательности считывается символ x.
2. Запускается подпрограмма s0, которая символизирует начальное состояние распознавателя.
3. Символ анализируется и запускается следующая подпрограмма, символизирующая то состояние, в которое может перейти анализатор из s0, если входной символ равен x.
4. Если в вызванной подпрограмме не получается распознать следующий символ последовательности, то значит распознаватель переходит в подпрограмму состояния ошибки, иначе — в подпрограмму следующего состояния.
5. Если распознаватель остановился в подпрограмме допускающего состояния и цепочка символов распознана, то полученная лексема добавляется в таблицу лексем, куда заносится её тип и значение.
6. Если же распознаватель остановится в подпрограмме состояния ошибки, то пользователю выводится сообщение о типе ошибки и её расположении (номер строки) в исходном тексте.

## Синтаксический анализатор

Синтаксический анализатор был реализован с помощью метода рекурсивного спуска.

Преобразуем исходную грамматику. Для удобства работы обозначим:

* A — <программа>;
* B — <переменная>;
* C — <операторы>;
* D — <оператор>;
* E — <выражение>;
* F — <условие>;
* G — <терм>;
* H — <множитель>;
* I — <отношение>;
* a — var;
* b — begin;
* c — end
* d — идентификатор;
* e — read;
* f — write;
* g — :=;
* h — if;
* i — then;
* j — else;
* k — число;
* l — or;
* m — and;
* n — not;
* o — операция\_отношения;
* p — строка.

Исходная грамматика в таком случае имеет вид:

1. A → aBbCc
2. B → d,B
3. B → d
4. C → C;D
5. C → D
6. D → e(p,d)
7. D → f(p,d)
8. D → dgE
9. D → hFiDjD
10. E → d
11. E → k
12. E → E+E
13. E → E–E
14. F → FlG
15. F → G
16. G → GmH
17. G → H
18. H → n(F)
19. H → (F)
20. H → I
21. I → EoE

Для того, чтобы грамматику можно было использовать в синтаксическом анализаторе она должна принадлежать к классу LL(1)-грамматик. Выполним преобразования. Сперва применим левую факторизацию для правил 2 и 3 (т. к. их правая часть начинается с **d**). Тем самым вводим нетерминал **J**:

1. A → aBbCc
2. **B → dJ**
3. **J → ,B**
4. **J → Ɛ**
5. C → C;D
6. C → D
7. D → e(p,d)
8. D → f(p,d)
9. D → dgE
10. D → hFiDjD
11. E → d
12. E → k
13. E → E+E
14. E → E–E
15. F → FlG
16. F → G
17. G → GmH
18. G → H
19. H → n(F)
20. H → (F)
21. H → I
22. I → EoE

Применим левую факторизацию к правилам 13 и 14, тем самым вводим нетерминал **K**:

1. A → aBbCc
2. B → dJ
3. J → ,B
4. J → Ɛ
5. C → C;D
6. C → D
7. D → e(p,d)
8. D → f(p,d)
9. D → dgE
10. D → hFiDjD
11. E → d
12. E → k
13. **E → EК**
14. **К → +E**
15. **К → –E**
16. F → FlG
17. F → G
18. G → GmH
19. G → H
20. H → n(F)
21. H → (F)
22. H → I
23. I → EoE

Правило 5 самолеворекурсивно. Заменим правила 5 и 6, введя новый нетерминал **L**:

1. A → aBbCc
2. B → dJ
3. J → ,B
4. J → Ɛ
5. **C → DL**
6. **L → ;DL**
7. **L → Ɛ**
8. D → e(p,d)
9. D → f(p,d)
10. D → dgE
11. D → hFiDjD
12. E → d
13. E → k
14. E → EК
15. К → +E
16. К → –E
17. F → FlG
18. F → G
19. G → GmH
20. G → H
21. H → n(F)
22. H → (F)
23. H → I
24. I → EoE

Правило 14 самолеворекурсивно. Заменим правила 12–14, введя новый нетерминал **M**:

1. A → aBbCc
2. B → dJ
3. J → ,B
4. J → Ɛ
5. C → DL
6. L → ;DL
7. L → Ɛ
8. D → e(p,d)
9. D → f(p,d)
10. D → dgE
11. D → hFiDjD
12. **E → dM**
13. **E → kM**
14. **M → КM**
15. **M → Ɛ**
16. К → +E
17. К → –E
18. F → FlG
19. F → G
20. G → GmH
21. G → H
22. H → n(F)
23. H → (F)
24. H → I
25. I → EoE

Правило 18 самолеворекурсивно. Заменим правила 18 и 19, введя новый нетерминал **N**:

1. A → aBbCc
2. B → dJ
3. J → ,B
4. J → Ɛ
5. C → DL
6. L → ;DL
7. L → Ɛ
8. D → e(p,d)
9. D → f(p,d)
10. D → dgE
11. D → hFiDjD
12. E → dM
13. E → kM
14. M → КM
15. M → Ɛ
16. К → +E
17. К → –E
18. **F → GN**
19. **N → lGN**
20. **N → Ɛ**
21. G → GmH
22. G → H
23. H → n(F)
24. H → (F)
25. H → I
26. I → EoE

Правило 21 самолеворекурсивно. Заменим правила 21 и 22, введя новый нетерминал **O**:

1. A → aBbCc
2. B → dJ
3. J → ,B
4. J → Ɛ
5. C → DL
6. L → ;DL
7. L → Ɛ
8. D → e(p,d)
9. D → f(p,d)
10. D → dgE
11. D → hFiDjD
12. E → dM
13. E → kM
14. M → КM
15. M → Ɛ
16. К → +E
17. К → –E
18. F → GN
19. N → lGN
20. N → Ɛ
21. **G → HO**
22. **O → mHO**
23. **O → Ɛ**
24. H → n(F)
25. H → (F)
26. H → I
27. I → EoE

Составим таблицу с множеством первых (FIRST) и следующих (FOLLOW). Множества находим по следующим правилам:

* Первые (FIRST):
  + Если в грамматике существуют правила A → Ɛ, то в множество первых для А заносим Ɛ.
  + Если есть правило A → αβ, то α добавляем в множество первых для А.
  + Если есть правило A → X1X2…Xn, где Xi — это терминалы или нетерминалы, то в множество первых для А добавляем множество первых для X1, кроме Ɛ (если есть правило X1 → Ɛ или неявное такое правило), если в множестве первых для X1 есть Ɛ, то в множество первых для А добавляем множество первых для X2 и т. д. Если в множестве первых Xn будет Ɛ, то мы добавляем Ɛ в множество первых для А.
* Следующие (FOLLOW):
  + Если S — начальный нетерминал, то в множество следующих для S добавляем маркер конца ┤.
  + Если есть правило A -> αBβ, то в множество следующих для B добавить множество первых для β, кроме Ɛ. Если же множество первых β содержит Ɛ, то множество следующих для B добавить множество следующих для А.

|  |  |  |
| --- | --- | --- |
| **<A>** | **FIRST** | **FOLLOW** |
| A | a | ┤ |
| B | d | b |
| C | e f d h | c |
| D | e f d h | j ; c |
| E | d k | m l i o j ; c + − ) |
| F | n ( d k | i ) |
| G | n ( d k | l i ) |
| H | n ( d k | m l i ) |
| I | d k | m l i ) |
| J | , Ɛ | b |
| K | + − | ; ) + − o m l i j c |
| L | ; Ɛ | c |
| M | + − Ɛ | m l i o j ; c + − ) |
| N | l Ɛ | i ) |
| O | m Ɛ | l i ) |

Найдём множество выбора для каждого правила грамматики по следующему алгоритму:

1. Если правило грамматики имеет вид: A → bα, где b - терминал, а α – цепочка терминалов и нетерминалов, то определим множество выбора для правила A → bα равным b. Обозначим данную запись следующим образом: ВЫБОР(A → bα) = b
2. Если правило имеет вид: A → Ɛ, то определим: ВЫБОР(A → Ɛ) = СЛЕД(A) (СЛЕД — это FOLLOW).

|  |  |
| --- | --- |
| **Правило** | **Мн-во** |
| A → aBbCc | a |
| B → dJ | d |
| J → ,B | , |
| J → Ɛ | b |
| C → DL | e f d h |
| L → ;DL | ; |
| L → Ɛ | c |
| D → e(p,d) | e |
| D → f(p,d) | f |
| D → dgE | d |
| D → hFiDjD | h |
| E → dM | d |
| E → kM | k |
| M → КM | + − |
| M → Ɛ | m l i o j ; c + − ) |
| К → +E | + |
| К → −E | − |
| F → GN | n ( d k |
| N → lGN | l |
| N → Ɛ | i ) |
| G → HO | n ( d k |
| O → mHO | m |
| O → Ɛ | l i ) |
| H → n(F) | n |
| H → (F) | ( |
| H → I | d k |
| I → EoE | d k |

Из-за пересечения множеств выбора правил 14 и 15 (ячейки таблицы выделены) грамматика не подходит под определение LL(1).

Поскольку грамматика не преобразовывается в LL(1), то внесём в неё небольшие изменения, добавив при левой факторизации (во время ввода нетерминала K) эпсилон-правило. Получим:

1. A → aBbCc
2. B → dJ
3. J → ,B
4. J → Ɛ
5. C → C;D
6. C → D
7. D → e(p,d)
8. D → f(p,d)
9. D → dgE
10. D → hFiDjD
11. E → d
12. E → k
13. E → EК
14. К → +E
15. К → –E
16. **K → Ɛ**
17. F → FlG
18. F → G
19. G → GmH
20. G → H
21. H → n(F)
22. H → (F)
23. H → I
24. I → EoE

Правило 5 самолеворекурсивно. Заменим правила 5 и 6, введя новый нетерминал **L**:

1. A → aBbCc
2. B → dJ
3. J → ,B
4. J → Ɛ
5. **C → DL**
6. **L → ;DL**
7. **L → Ɛ**
8. D → e(p,d)
9. D → f(p,d)
10. D → dgE
11. D → hFiDjD
12. E → d
13. E → k
14. E → EК
15. К → +E
16. К → –E
17. K → Ɛ
18. F → FlG
19. F → G
20. G → GmH
21. G → H
22. H → n(F)
23. H → (F)
24. H → I
25. I → EoE

Правило 14 самолеворекурсивно. Заменим правила 12–14, используя нетерминал **K**:

1. A → aBbCc
2. B → dJ
3. J → ,B
4. J → Ɛ
5. C → DL
6. L → ;DL
7. L → Ɛ
8. D → e(p,d)
9. D → f(p,d)
10. D → dgE
11. D → hFiDjD
12. **E → dK**
13. **E → kK**
14. **К → +E**
15. **К → –E**
16. **K → Ɛ**
17. F → FlG
18. F → G
19. G → GmH
20. G → H
21. H → n(F)
22. H → (F)
23. H → I
24. I → EoE

Правило 17 самолеворекурсивно. Заменим правила 17 и 18, введя новый нетерминал **M**:

1. A → aBbCc
2. B → dJ
3. J → ,B
4. J → Ɛ
5. C → DL
6. L → ;DL
7. L → Ɛ
8. D → e(p,d)
9. D → f(p,d)
10. D → dgE
11. D → hFiDjD
12. E → dK
13. E → kK
14. К → +E
15. К → –E
16. K → Ɛ
17. **F → GM**
18. **M → lGM**
19. **M → Ɛ**
20. G → GmH
21. G → H
22. H → n(F)
23. H → (F)
24. H → I
25. I → EoE

Правило 20 самолеворекурсивно. Заменим правила 20 и 21, введя новый нетерминал **N**:

1. A → aBbCc
2. B → dJ
3. J → ,B
4. J → Ɛ
5. C → DL
6. L → ;DL
7. L → Ɛ
8. D → e(p,d)
9. D → f(p,d)
10. D → dgE
11. D → hFiDjD
12. E → dK
13. E → kK
14. К → +E
15. К → –E
16. K → Ɛ
17. F → GM
18. M → lGM
19. M → Ɛ
20. **G → HN**
21. **N → mHN**
22. **N → Ɛ**
23. H → n(F)
24. H → (F)
25. H → I
26. I → EoE

Составим таблицу с множеством первых (FIRST) и следующих (FOLLOW):

|  |  |  |
| --- | --- | --- |
| **<A>** | **FIRST** | **FOLLOW** |
| A | a | ┤ |
| B | d | b |
| C | e f d h | c |
| D | e f d h | j ; c |
| E | d k | o j ; c m l i ) |
| F | n ( d k | i ) |
| G | n ( d k | l i ) |
| H | n ( d k | m l i ) |
| I | d k | m l i ) |
| J | , Ɛ | b |
| K | + − Ɛ | o j ; c m l i ) |
| L | ; Ɛ | c |
| M | l Ɛ | i ) |
| N | m Ɛ | l i ) |

Найдём множество выбора для каждого правила грамматики:

|  |  |
| --- | --- |
| **Правило** | **Мн-во** |
| A → aBbCc | a |
| B → dJ | d |
| J → ,B | , |
| J → Ɛ | b |
| C → DL | e f d h |
| L → ;DL | ; |
| L → Ɛ | c |
| D → e(p,d) | e |
| D → f(p,d) | f |
| D → dgE | d |
| D → hFiDjD | h |
| E → dK | d |
| E → kK | k |
| К → +E | + |
| К → −E | − |
| K → Ɛ | o j ; c m l i ) |
| F → GM | n ( d k |
| M → lGM | l |
| M → Ɛ | i ) |
| G → HN | n ( d k |
| N → mHN | m |
| N → Ɛ | l i ) |
| H → n(F) | n |
| H → (F) | ( |
| H → I | d k |
| I → EoE | d k |

Так как множества выбора для правил с одинаковой левой частью не пересекаются, полученная грамматика — LL(1).

Введём следующие символы действий:

|  |  |
| --- | --- |
| **Символ** | **Действие** |
| {1} | Вывод в поток:  #include <stdio.h>  int main() { |
| {2} | Вывод в поток: ;\n |
| {3} | Вывод в поток: \n\treturn 0;\n} |
| {4} | Вывод в поток имени идентификатора |
| {5} | Вывод символа «,» в поток |
| {6} | Вывод в поток: \n\tscanf |
| {7} | Вывод символа «(» в поток |
| {8} | Вывод считанной строки в поток в двойных кавычках |
| {9} | Вывод символа «&» в поток |
| {10} | Вывод символа «)» в поток |
| {11} | Вывод символа «;» в поток |
| {12} | Вывод в поток: \n\tprintf |
| {13} | Вывод в поток строк, высчитывающих выражение, и присваивание значения этого выражения считанному ранее идентификатору. Занесение в стек свободных переменных временных идентификаторов |
| {14} | Вывод в поток строк, высчитывающих логическое выражение, и строки вида if (<условие>), где <условие> — имя переменной, хранящей значение логического выражение. Занесение в стек свободных переменных временных идентификаторов |
| {15} | Создание новой метки и занесение в поток строки goto <имя метки>; |
| {16} | Сохранение строк с действиями оператора |
| {17} | Вывод последних сохранённых строк с действиями оператора |
| {18} | Вывод в поток строки с названием предпоследней созданной метки и символа «:» |
| {19} | Вывод в поток строки с названием последней созданной метки и символа «:» |
| {20} | Вывод в поток числа |
| {21} | Вывод в поток символа «+» |
| {22} | Вывод в поток символа «−» |
| {23} | Сохранение текущей использующейся временной переменной |
| {24} | Создание новой временной переменной |
| {25} | Вывод в поток строки, в которой текущей временной переменной присваивается дизъюнкция двух ранее сохранённых временных переменных |
| {26} | Вывод в поток строки, в которой текущей временной переменной присваивается конъюнкция двух ранее сохранённых временных переменных |
| {27} | Вывод символа «!» в поток |
| {28} | Вывод в поток символа отношения |

Расставим символы действий:

1. A → a{1}Bb{2}Cc{3}
2. B → d{4}J
3. J → ,{5}B
4. J → Ɛ
5. C → DL
6. L → ;DL
7. L → Ɛ
8. D → e{6}({7}p{8},{5}d{9}{4}){10}{11}
9. D → f{12}({7}p{8},{5}d{4}){10}{11}
10. D → dgE{13}
11. D → hF{14}iD{16}jD{16}{15}{17}{15}{18}{17}{19}
12. E → d{4}K
13. E → k{20}K
14. К → +{21}E
15. К → –{22}E
16. K → Ɛ
17. F → G{23}M{23}{24}{25}{2}
18. M → lG{23}M{23}{24}{25}{2}
19. M → Ɛ
20. G → H{23}N{23}{24}{26}{2}
21. N → mH{23}N{23}{24}{26}{2}
22. N → Ɛ
23. H → n{27}(F{23}){2}
24. H → (F{23}){2}
25. H → I
26. I → E{13}o{28}E{17}

Синтаксический и лексический анализатор взаимодействуют последовательно, на вход синтаксическому анализатору приходит таблица лексем, составленных лексическим анализатором.

# Тестирование

Результат работы программы без ошибок во входных данных:

|  |  |
| --- | --- |
| **Исходный код** | **Результат** |
| var aa, bb  begin  aa := 1 + 2 + 3;  bb := 5 + aa;  if (aa > bb and bb > 10 or aa < 5) then  aa := 1  else  aa := 10;  write("%d", aa);  ac := ab + 2 + 3  end | #include <stdio.h>  int main() {  int aa, bb, exp0, exp1, exp2;  exp0 = 1;  exp0 = exp0 + 2;  exp0 = exp0 + 3;  aa = exp0;  exp0 = 5;  exp0 = exp0 + aa;  bb = exp0;  exp0 = aa > bb;  exp1 = bb > 10;  exp2 = exp0 && exp1;  exp1 = aa < 5;  exp0 = exp2 || exp1;  if (exp0) goto label1;  aa = 10;  goto label2;  label1:  aa = 1;  label2:  printf("%d", aa);  exp0 = ab;  exp0 = exp0 + 2;  exp0 = exp0 + 3;  ac = exp0;  return 0;  } |

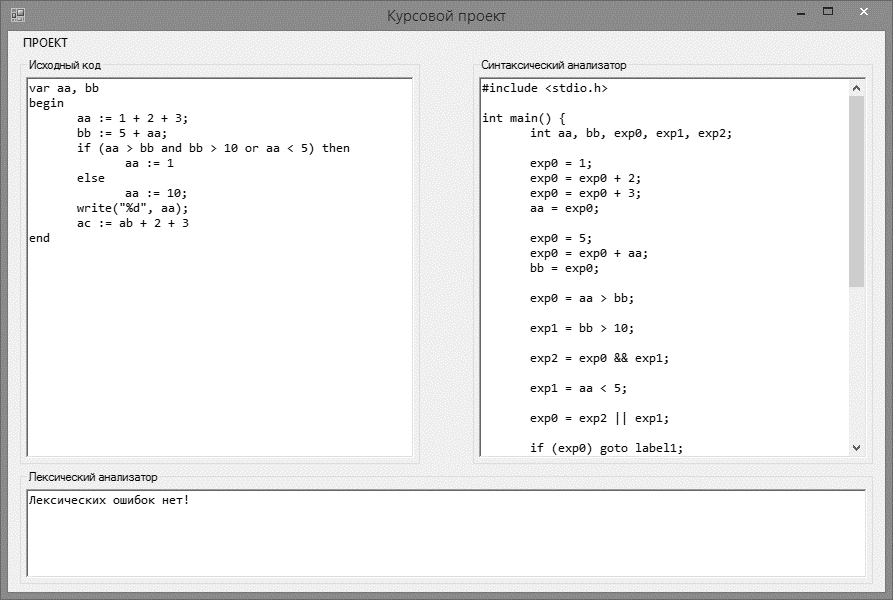
Обработка программы с лексической ошибкой:

|  |  |
| --- | --- |
| **Исходный код** | **Результат (в окне лексического анализатора)** |
| var a  begin  a := 1 + 2;  end | Ошибка: a; Строка 1. |

Обработка программы с синтаксической ошибкой:

|  |  |
| --- | --- |
| **Исходный код** | **Результат (в окне синтаксического анализатора)** |
| var aa  begin  aa := 1 + ;  end | Ожидался идентификатор или число после лексемы «+» (plus). |

Скриншот программы:



Код всей программы находится в Приложении.

# Заключение

В ходе выполнения этого курсового проекта были закреплены знания в области основных эквивалентных преобразований КС-грамматик. Эти преобразования были применены для приведения исходной грамматики к классу LL(1)-грамматик.

Были более подробно изучены основные способы построения, алгоритмы преобразования, анализа и реализации конечных распознавателей. Они применялись для создания конечных распознавателей лексем. Так же были изучены более подробно методы разработки и реализации лексических анализаторов и принципы синтаксически управляемой трансляции.

Все это в совокупности помогло написать транслятор, который может преобразовать программу на исходном языке, в семантически эквивалентную программу на языке Си. Полученный транслятор может обработать программу, не содержащую лексических, синтаксических и семантических ошибок, а также, программу, содержащую ошибки, в этом случае будет выдано сообщение о них.

# Список литературы

1. Рязанов Ю. Д. Пособие «Теория языков программирования» БГТУ им В. Г. Шухова — 116 с.
2. Льюне Ф. «Теоретические основы проектирования компиляторов». «МИР», Москва, 1979 — 655 с.
3. Каталог API (Microsoft) и справочных материалов [электронный ресурс]. — msdn.microsoft.com/ru-ru/library/.
4. Википедия — Свободная энциклопедия на русском языке [электронный ресурс] — ru.wikipedia.org.
5. Stack Overflow [электронный ресурс] — stackoverflow.com.
6. CodeProject — For those who code [электронный ресурс] — codeproject.com.

# Приложение

**Содержимое файла Parser.cs (синтаксический анализатор):**

using System.Collections.Generic;

using System.Data;

using System.Linq;

namespace Project

{

class Parser

{

int expsCount = 0;

int labelsCounter = 0;

int index = 0;

int errorCode = 0;

Stack<string> exps = new Stack<string>();

string curExp = "";

DataTable lexems;

public string Translate(DataTable lexem\_table)

{

lexems = lexem\_table;

if (lexems.RCount() == 0)

return "";

index = 0;

var outString = A();

if (outString != null && errorCode == 0 && index < lexems.RCount() && lexems.Type(index) == "-|")

return outString;

else

return GetLastError();

}

// A -> aBbCc

/\*

\* A — <программа>

\* B — <переменные>

\* C — <операторы>

\* a — var

\* b — begin

\* c — end

\*/

string A()

{

if (lexems.Type(index++) == "var")

{

string outString = B();

if (outString == null)

return null;

outString = "#include <stdio.h>\n\nint main() {\n\tint " + outString;

if (lexems.Type(index++) == "begin")

{

string opString = C();

if (opString == null)

return null;

if (lexems.Type(index++) == "end")

{

string variables = "";

expsCount--;

for (int i = 0; i <= expsCount; i++)

{

variables += ", exp" + i.ToString();

}

variables += ";\n";

return outString + variables + opString + "\n\treturn 0;\n}";

}

errorCode = 3;

return null;

}

errorCode = 2;

return null;

}

errorCode = 1;

return null;

}

// B —> dJ

/\*

\* B — <переменные>

\* J — <доп. нетерминал>

\* d — id

\*/

string B()

{

if (lexems.Type(index) == "id")

return lexems.Value(index++) + J();

errorCode = 4;

return null;

}

// J -> ,B

// J -> ε

/\*

\* J — <доп. нетерминал>

\* B — <переменные>

\*/

string J()

{

if (lexems.Type(index) == "comma")

{

var outString = lexems.Value(index++) + ' ';

var b = B();

if (b == null)

return null;

return outString + b;

}

return "";

}

// C -> DL

/\*

\* C — <операторы>

\* D — <оператор>

\* L — <доп. нетерминал>

\*/

string C()

{

var d = D();

if (d == null)

return null;

var l = L();

if (l == null)

return null;

return d + l;

}

// L -> ;DL

// L -> ε

/\*

\* D — <оператор>

\* L — <доп. нетерминал>

\*/

string L()

{

if (lexems.Type(index) == "s-colon")

{

index++;

var d = D();

if (d == null)

return null;

var l = L();

if (l == null)

return null;

return d + l;

}

return "";

}

// D -> e(p, d)

// D -> f(p, d)

// D -> dgE

// D -> hFiDjD

/\*

\* D — <оператор>

\* E — <выражение>

\* d — id

\* e — read

\* f — write

\* g — assign

\* h — if

\* i — then

\* j — else

\* p — string

\*/

string D()

{

var outString = "";

if (lexems.Type(index) == "read" || lexems.Type(index) == "write")

{

var type = lexems.Type(index);

var amp = "";

if (type == "read")

{

amp = "&";

outString = "\n\tscanf";

}

else

outString = "\n\tprintf";

if (lexems.Type(++index) == "bracket")

{

outString += lexems.Value(index++);

if (lexems.Type(index) == "string")

{

var tmp = lexems.Value(index++).Replace("''", "'").Remove(0, 1);

if (tmp != "")

outString += "\"" + tmp.Remove(tmp.Length - 1) + "\"";

else

outString += "\"\"";

if (lexems.Type(index) == "comma")

{

outString += lexems.Value(index++);

if (lexems.Type(index) == "id")

{

outString += " " + amp + lexems.Value(index++);

if (lexems.Type(index++) == "b-bracket")

{

return outString + ");\n";

}

errorCode = 9;

return null;

}

errorCode = 4;

return null;

}

errorCode = 8;

return null;

}

errorCode = 7;

return null;

}

errorCode = 6;

return null;

}

switch (lexems.Type(index))

{

case "id":

outString = lexems.Value(index++);

if (lexems.Type(index++) == "assign")

{

createCurExp();

var e = E();

if (e == null)

return null;

if (e.Contains('\n'))

{

var res = "\n\t" + curExp + " = " + e + ";\n\t" + outString + " = " + curExp + ";\n";

exps.Push(curExp);

curExp = "";

return res;

}

exps.Push(curExp);

curExp = "";

return "\n\t" + outString + " = " + e + ";\n";

}

errorCode = 10;

return null;

case "if":

outString = "\n\t" + lexems.Value(index++);

var condition = F();

if (condition == null)

return null;

outString = condition + outString + " (" + curExp + ")";

if (lexems.Type(index++) == "then")

{

var d1 = D();

if (d1 == null)

return null;

if (lexems.Type(index++) == "else")

{

var d2 = D();

if (d2 == null)

return null;

outString += " goto label" + ++labelsCounter + ";\n";

outString += d2 + "\n\tgoto label" + ++labelsCounter + ";\n\n";

outString += "label" + (labelsCounter - 1) + ":\n" + d1 + "\n";

return outString + "label" + labelsCounter + ":\n";

}

errorCode = 12;

return null;

}

errorCode = 11;

return null;

default:

errorCode = 5;

return null;

}

}

// E -> dK

// E -> kK

/\*

\* E — <выражение>

\* K — <доп. нетерминал>

\* d — id

\* k — num

\*/

string E()

{

var outString = "";

switch (lexems.Type(index))

{

case "id":

case "num":

outString = lexems.Value(index++);

break;

default:

errorCode = 13;

return null;

}

var k = K();

if (k == null)

return null;

if (k != "")

{

return outString + ";\n\t" + curExp + " = " + curExp + k;

}

return outString;

}

// K -> +E

// K -> -E

// K -> ε

/\*

\* K — <доп. нетерминал>

\* E — <выражение>

\* + — plus

\* - — minus

\*/

string K()

{

switch (lexems.Type(index))

{

case "plus":

case "minus":

var res = " " + lexems.Value(index++) + " ";

var e = E();

if (e == null)

return null;

return res + e;

default:

return "";

}

}

// F -> GM

/\*

\* F — <условие>

\* G — <терм>

\* M — <доп. нетерминал>

\*/

string F()

{

return GM();

}

// M -> lGM

// M -> ε

/\*

\* G — <терм>

\* M — <доп. нетерминал>

\* l — or

\*/

string M()

{

if (lexems.Type(index) == "or")

{

index++;

return GM();

}

return "";

}

// Дополнительная функция для обработки пары нетерминалов GM,

// → которые встречаются в правилах F и M.

string GM()

{

var g = G();

if (g == null)

return null;

var saved = curExp;

var m = M();

if (m == null)

return null;

if (m != "")

{

var saved2 = curExp;

newCurExp();

var outString = g + m + "\n\t" + curExp + " = " + saved + " || " + saved2 + ";\n";

exps.Push(saved);

exps.Push(saved2);

return outString;

}

return g;

}

// G -> HN

/\*

\* G — <терм>

\* H — <множитель>

\* N — <доп. нетерминал>

\*/

string G()

{

return HN();

}

// N -> mHN

// N -> ε

/\*

\* H — <множитель>

\* N — <доп. нетерминал>

\* m — and

\*/

string N()

{

if (lexems.Type(index) == "and")

{

index++;

return HN();

}

return "";

}

// Дополнительная функция для обработки пары нетерминалов HN,

// → которые встречаются в правилах G и N.

string HN()

{

var h = H();

if (h == null)

return null;

var saved = curExp;

var n = N();

if (n == null)

return null;

if (n != "")

{

var saved2 = curExp;

newCurExp();

var outString = h + n + "\n\t" + curExp + " = " + saved + " && " + saved2 + ";\n";

exps.Push(saved);

exps.Push(saved2);

return outString;

}

return h;

}

// H -> n(F)

// H -> (F)

// H -> I

/\*

\* F — <условие>

\* H — <множитель>

\* I — <отношение>

\* n — not

\*/

string H()

{

var outString = "";

if (lexems.Type(index) == "not")

{

outString = "!";

if (lexems.Type(++index) == "bracket")

{

index++;

var f = F();

if (f == null)

return null;

if (lexems.Type(index++) == "b-bracket")

return f + "\n\t" + curExp + " = " + outString + curExp + ";\n";

errorCode = 9;

return null;

}

errorCode = 6;

return null;

}

else if (lexems.Type(index) == "bracket")

{

index++;

var f = F();

if (f == null)

return null;

outString += f;

if (lexems.Type(index++) == "b-bracket")

return outString;

errorCode = 9;

return null;

}

else

return I();

}

// I -> EoE

/\*

\* E — <выражение>

\* I — <отношение>

\* o — eq

\*/

string I()

{

var outString = "";

newCurExp();

var e = E();

if (e == null)

return null;

if (lexems.Type(index) == "eq")

{

var eq = " " + lexems.Value(index++) + " ";

if (eq == " = ") eq = " == ";

if (eq == " <> ") eq = " != ";

string saved = "";

if (e.Contains('\n'))

{

saved = curExp;

newCurExp();

}

var e2 = E();

if (e2 == null)

return null;

string saved2 = "";

if (e2.Contains('\n'))

{

saved2 = curExp;

newCurExp();

}

if (saved != "")

outString = "\n\t" + saved + " = " + e + ";\n";

if (saved2 != "")

outString += "\n\t" + saved2 + " = " + e2 + ";";

outString += "\n\t" + curExp + " = ";

if (saved != "")

{

outString += saved;

exps.Push(saved);

}

else

outString += e;

outString += eq;

if (saved2 != "")

{

outString += saved2 + ";\n";

exps.Push(saved2);

}

else

outString += e2 + ";\n";

return outString;

}

errorCode = 14;

return null;

}

void newCurExp()

{

if (exps.IsEmpty())

{

curExp = "exp" + expsCount++;

}

else

{

curExp = exps.Pop();

}

}

void createCurExp()

{

if (curExp == "")

{

newCurExp();

}

}

string GetLastError()

{

switch (errorCode)

{

case 0:

return "Неожиданный конец программы!";

case 1:

return "Программа должна начинаться с ключевого слова 'var'!";

case 2:

return "Ожидалось ключевое слово 'begin'" + beforeLexem(2);

case 3:

return "Ожидалось ключевое слово 'end' или ';'" + beforeLexem(2);

case 4:

return "Ожидался идентификатор" + beforeLexem();

case 5:

return "Ожидался оператор" + beforeLexem();

case 6:

return "Ожидался символ '('" + beforeLexem();

case 7:

return "Ожидалась строка" + beforeLexem();

case 8:

return "Ожидался символ ','" + beforeLexem();

case 9:

return "Ожидался символ ')'" + beforeLexem(2);

case 10:

return "Ожидался оператор ':='" + beforeLexem(2);

case 11:

return "Ожидалось ключевое слово 'then'" + beforeLexem(2);

case 12:

return "Ожидалось ключевое слово 'else'" + beforeLexem(2);

case 13:

return "Ожидался идентификатор или число" + beforeLexem(1);

case 14:

return "Ожидался операция отношения" + beforeLexem(1);

default:

return "Неизвестная ошибка.";

}

}

string beforeLexem(int shift = 1)

{

return " после лексемы «" + lexems.Value(index - shift) + "» (" + lexems.Type(index - shift) + ").";

}

}

}