МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

Вл. Пономарев

Практикум 1

по теории языков программирования и методам трансляции

Учебно-методическое пособие

Часть 1. Преобразования грамматик

Озерск, 2018

УДК 681.3.06
П 56

Вл. Пономарев. Практикум 1 по теории языков программирования и методам трансляции. Учебно-методическое пособие. Часть 1. Преобразования грамматик. Озерск: ОТИ НИЯУ МИФИ, 2018. — 61 с.

В пособии подробно излагается, как выполнять практические работы по дисциплине «Теория языков программирования и методы трансляции». Работы первого семестра изучения дисциплины включают в себя алгоритмы преобразования грамматик и конечных автоматов.

В качестве основного материала при выполнении практических работ пособие предназначено для студентов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника» и специальности 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения».

Рецензенты:
1.
2.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Общие цели занятий

В ходе практических работ данной части предлагается изучить и реализовать следующие алгоритмы преобразования грамматик.

1) Введение в программирование алгоритмов с грамматиками.
2) Группирование правил грамматики.
3) Удаление бесплодных символов.
4) Удаление недостижимых символов.
5)** Устранение пустых правил.
6)* Устранение цепных правил.
7)** Устранение левой рекурсии.
8⁻)* Левая факторизация.
9⁻) Приведение к нормальной форме Хомского.

На выполнение одной работы предварительно отводится 2 академических часа, однако некоторые, наиболее сложные алгоритмы могут потребовать большего времени. Поэтому, в зависимости от количества учебных часов, выделенных на проведение работ, сложности работ и навыков обучающегося, преподаватель может выбирать индивидуальные траектории.

Работы, которые можно без ущерба для процесса обучения изъять из приведенной общей траектории, отмечены знаком «минус». Сложные работы отмечены звездочками. Каждая звездочка — это 2 часа.

Для выполнения работ используется библиотека классов grammar.lib, разработанная автором специально для учебных целей. Программирование алгоритмов ведется в среде Microsoft Visual C++ на языке программирования C++. Рабочее пространство для проведения работ представляет собой консольное приложение grom (grammar object modeling).

Для защиты знаний и навыков, полученных в ходе выполнения практических работ студент должен иметь тетрадь (12-18 листов). Для каждой выполненной работы в тетрадь записывается отчет.

Отчет по работе начинается с заголовка:

1. Фамилия Имя Отчество
2. Группа
3. Дата начала выполнения работы
4. Код работы
5. Название работы
6. Цели работы
7. Задачи работы

При необходимости при выполнении работы в отчет записываются контрольные значения. Во время защиты тетрадь используется для вопросов преподавателя и ответов студента.

1. Работа PL-101. Программирование алгоритмов с грамматиками

Цели:

- изучение библиотеки grammar.lib.

Задачи:

- изучение двоичного представления объектов, необходимых для программирования алгоритмов преобразования грамматик;
- изучение приемов работы с двоичными объектами;

Перед началом выполнения работы нужно установить на компьютер рабочий проект. Проект предоставляется преподавателем. Каталог grom архива копируется на диск C: в корневой каталог. Далее следует убедиться в том, что целевой платформой является x86 или win32, и в том, что проект компилируется и запускается.

Все работы выполняются в одном и том же рабочем проекте.

1.1. Формат SYNAX

При выполнении работ используется описание грамматик, называемое здесь «формат SYNAX» по названию программы автора, в которой этот формат впервые применен. Как известно из теоретического курса, грамматика представляет собой систему из четырех компонент: множество терминалов, множество нетерминалов, множество правил и целевой символ.

Для задания грамматики на самом деле достаточно описать только ее правила. Символы грамматики могут быть «собраны» во время чтения описания из правил, при этом нужен механизм, отличающий нетерминальные символы от терминальных. Что касается целевого символа, то его легко определить, если придерживаться, например, принципа: первое правило должно описывать любое правило для целевого символа.

Практически значимыми являются грамматики класса не ниже 2 по классификации Хомского, а именно контекстно-свободные и регулярные. Они отличаются тем, что левая часть любого правила представляет собой один-единственный нетерминальный символ.

В формате SYNAX нетерминалы записываются в угловых скобках, а терминалы — в квадратных скобках. Первая строка записи грамматики должна содержать целевой символ. Следующие строки содержат правила, по одному правилу в строке. Правая часть правила отделяется от левой части знаком равно. Пустая строка обозначается точкой или никак.

В качестве примера рассмотрим следующую грамматику.

$S \rightarrow ABC$
$A \rightarrow a \mid B$
$B \rightarrow b \mid \lambda$
$C \rightarrow c \mid \lambda$

Целевым символом является нетерминал *S*. В грамматике 7 правил.

В формате SYNAX эта грамматика записывается следующим образом:

```
<S>
<S>=<A><B><C>
<A>=[a]
<A>=<B>
<B>=[b]
<B>=.
<C>=[c]
<C>=.
```

Здесь точкой обозначена пустая цепочка λ.

1.2. Входные файлы

Описание грамматики (или конечного автомата) представляет из себя текстовый файл в кодировке Windows-1251 с расширением .sxg. Библиотека grammar.lib распознает описание по первой строке. Если первая строка начинается со знака треугольной скобки, библиотека считает этот файл описанием грамматики. Если первая строка начинается со знака фигурной скобки, библиотека считает этот файл описанием конечного автомата любого вида. В любом другом случае библиотека считает, что файл содержит регулярное выражение. В первых двух случаях библиотека разбирает входной файл, и формирует внутреннее представление либо грамматики, либо конечного автомата.

Сейчас создадим текстовый файл для описания грамматики G1.

1. Откроем FAR и перейдем в каталог C:\grom\Debug.
2. Введем Shift+F4 для начала работы с текстовым файлом.
3. Введем название файла "001.sxg" и Enter.
4. Введем Shift+F8 и выберем кодировку 1251.
5. Наберем текст грамматики (в конце последней строки Enter):

```
<S>
<S>=<S>[+]<T>
<S>=<T>
<T>=<T>[*]<P>
<T>=<P>
<P>=[a]
<P>=[(]<S>[)]
•
```

6. Нажмем Escape и Enter. Файл готов. Чтобы редактировать этот файл повторно, нужно установить на него указатель и нажать F4.

1.3. Рабочее пространство

Получим архив проекта grom. Извлечем из архива каталог grom в корневой каталог диска C:. Откроем проект. Убедимся, что целевой платформой является x86 или Win32, в зависимости от того, что есть.

В приложении выполняемое действие задается ключами командной строки в следующем формате:

```
grom grammar -key -key . . .
```

Здесь grom — название приложения, grammar — спецификация файла грамматики (автомата, выражения), key — один из возможных ключей. Примерно в середине модуля main.cpp есть функция get_key, в которой перечислены все возможные ключи программы. Например, следующая строка этой функции определяет ключ "-gr", который определяет действие «группирование правил грамматики»:

```
    if (_stricmp("-gr", buf) == 0) return ACT_GR;
```

Одновременно можно использовать несколько ключей, если они совместимы. Если ключи не указаны, откроется область для общих действий с грамматикой при условии, что спецификация файла грамматики указана.

Откроем модуль 01-gen.cpp.

В этом модуле выполняется первая практическая работа.

В модуле определены следующие функции:

```
// упражнение 1
void test1 (grammar & g1, grammar & g2) {}
. . .
// алгоритмы общих действий с грамматикой
void grammar_general(grammar & g1, grammar & g2) {
    test1(g1, g2);
}
// точка входа в алгоритм
// общие действия с грамматикой
int algorithm_general(grammar & g1, FILE * target) {
    // результирующая грамматика
    grammar g2;
    // алгоритм
    grammar_general(g1, g2);
    // выводим грамматику в консоль
    g2.print(stdout);
    // выводим грамматику в файл
    g2.print(target);
    // выходная грамматика
    g1 = g2;
    return 0;
}
```

Функция, начинающаяся с "algorithm_", является точкой входа в реализацию алгоритма, однако сама реализация должна выполняться в функции, расположенной в модуле выше, в данном случае в grammar_general. Точка входа требуется для управления объектами, например, для объявления вспомогательных грамматик или конечных автоматов, их анализа, вывода на консоль и в файл.

Функция точки входа обязана сообщать результат. Возвращаемое нулевое значение означает, что результат алгоритма неуспешен, при этом выполнение последующих действий, если они заданы, прерывается и приложение завершает работу.

Поставим точку остановки на операторе return.

Запустим приложение. Управление не приходит в точку остановки. В консоль при этом выводится сообщение "Source file path required".

Откроем свойства проекта, установим параметр Working Directory равным ..\Debug, параметр Command Arguments равным "001".

Запустим приложение. Управление должно прийти в точку остановки.

Запустим проект Ctrl+F5, чтобы увидеть вывод программы.

Анализатор входного файла преобразует текст в двоичное представление заданного типа. Распознанный текст выводится в консоль:

```
-- parse grammar
<S>
<S>=<S>[+]<T>
<S>=<T>
<T>=<T>[*]<P>
<T>=<P>
<P>=[a]
<P>=[(]<S>[)]
.Consistent

-- grammar_general
Grammar is empty.
```

Каждое действие предваряется комментарием "-- действие".

После разбора грамматики производится ее анализ и пишется одно из слов, Consistent или Inconsistent. Первое из слов означает, что все нетерминалы имеют определение и порождают терминальные цепочки.

1.4. Главные правила работы с библиотекой

При работе с библиотекой нужно помнить следующие правила.

1. Для обозначения символов грамматики используются целые числа типа short, который обозначается меткой типа SYMB.

2. Терминалы обозначаются положительными числами.

3. Нетерминалы обозначаются отрицательными числами.

4. Нумерация любых объектов начинается с единицы, а не с нуля, поэтому цикл перебора объектов должен иметь следующий вид:

```
for (int iterator = 1; iterator <= count; iterator++) { }
```

Здесь предполагается, что count хранит количество элементов.

5. Исходная грамматика или автомат никогда не изменяются, за исключением преобразований «устранение левой рекурсии» и «левая факторизация». Вместо этого формируется новый объект.

6. Для того, чтобы можно было использовать число, заменяющее символ (или состояние), сначала нужно зарегистрировать идентификатор этого символа. Регистрация определяет возможность добавления символа и выдает число, которое этот символ обозначает.

1.5. Упражнение 1. Формирование грамматики из ничего

Покажем, как можно сформировать грамматику, не имея никакой исходной грамматики. Предположим, мы хотим сформировать грамматику:

$S \rightarrow^{(1)} AB$
$A \rightarrow^{(2)} a \mid^{(3)} BB$
$B \rightarrow^{(4)} b \mid^{(5)} \lambda$

Чтобы сформировать правило, нужно иметь зарегистрированные терминалы и нетерминалы. Начнем с их регистрации (этот код нужно писать):

```
void test1(grammar & g1, grammar & g2) {
    SYMB S = g2.reg_nont("S");
    SYMB A = g2.reg_nont("A");
    SYMB B = g2.reg_nont("B");
    SYMB a = g2.reg_term("a");
    SYMB b = g2.reg_term("b");
}
```

Исполняя код шаг за шагом, убеждаемся, что переменные S, A, B, a, b получают значения –1, –2, –3, 1 и 2 соответственно. Кроме того, нужно ввести в окно Watch переменную g2 и посмотреть, как в массиве ident появляются идентификаторы символов (в середине массива).

Далее нужно сформировать правила. Для этого нужен класс RULE.

Пример формирования первого правила: $S \rightarrow AB$:

```
RULE rule;
rule[0] = S;
rule.append(A);
rule.append(B);
```

Этот код нужно писать в продолжение предыдущего. Исполняя его шаг за шагом, нужно наблюдать, как в массив symbols объекта rule добавляются числа, обозначающие символы. При этом мы увидим, что правило 1 на самом деле массив, содержащий числа –1, –2, –3.

Обратим внимание, как в правило записывается левый нетерминал и символы тела правила. Символы тела обычно записывает метод append.

Далее добавляем правило в грамматику, это строчка кода:

```
g2.rule_add(rule);
```

Теперь запускаем программу на выполнение при помощи Ctrl+F5 и убеждаемся, что первое правило выводится, вместо целевого символа выводится знак вопроса, и грамматика Inconsistent.

Целевой символ нужно указать следующим образом (пишем далее):

```
g2.set_start(S);
```

Пробуем Ctrl+F5 еще раз.

Чтобы использовать переменную rule повторно, ее нужно очистить:

```
rule.clear();
```

Формируем правило 2: $A \rightarrow a$:

```
rule.clear();
rule[0] = A;
rule.append(a);
g2.rule_add(rule);
```

Пробуем Ctrl+F5.

Формируем правило 3: $A \rightarrow BB$. Поскольку нетерминал правила 3 тот же, что и в правиле 2, используем метод clean, очищающий только тело:

```
rule.clean();
rule.append(B);
rule.append(B);
g2.rule_add(rule);
```

Пробуем Ctrl+F5.

Формируем правило 4: $B \rightarrow b$:

```
rule.clear();
rule[0] = B;
rule.append(b);
g2.rule_add(rule);
```

Формируем правило 5: $B \rightarrow \lambda$. Оно отличается от правила 4 тем, что в нем нет тела, поэтому нужно просто очистить тело переменной rule:

```
rule.clean();
g2.rule_add(rule);
```

Убедимся, что создается правильная грамматика и она Consistent.

Теперь перейдем в FAR и откроем каталог C:\grom\Debug. Мы обнаружим там новый файл 001-gen.sxg, который является результатом наших действий. Название результирующего файла формируется из названия исходного файла приписыванием всех примененных ключей. Мы ключ не указывали, но -gen — это ключ по умолчанию.

Эта грамматика потребуется для реализации алгоритма устранения пустых правил, поэтому переименуем файл 001-gen.sxg в 05.sxg (для пятой работы). Чтобы переименовать, установите указатель на файл, нажмите сочетание Shift+F6, введите новое имя и Enter.

Что еще интересного можно почерпнуть в этом упражнении.

Например, что дает повторная регистрация того же символа. Дописываем в конец нашего кода повторную регистрацию символа A:

```
SYMB X = g2.reg_nont("A");
```

Тестируем эту строчку кода и убеждаемся, что X = –2. Таким образом, можно сколь угодно раз регистрировать один и тот же идентификатор, получая при этом один и тот же код этого идентификатора.

Можно совмещать некоторые инструкции. Например, закомментируем строчку кода, регистрирующую символ *a*:

```
//SYMB a = g2.reg_term("a");
```

Тогда при формировании правила 2, $A \to a$, вместо строки

```
rule.append(a);
```

можно записать следующую строку:

```
rule.append(g2.reg_term("a"));
```

Можно упростить формирование целевого символа, так как есть метод set_start_id, который регистрирует идентификатор. Найдем и закомментируем строчку, которая устанавливает целевой символ:

```
//g2.set_start(S);
```

Теперь перейдем в начало функции и изменим первую строчку, которая регистрирует символ S. Заменим метод reg_nont на метод set_start_id:

```
SYMB S = g2.set_start_id("S");
```

Грамматика будет правильно сформирована.

Для выполнения следующего упражнения нужно удалить написанный код из функции grammar_general. Лучше всего вырезать и разместить код в комментарии в конце или начале модуля.

1.6. Упражнение 2. Перебор правил грамматики (копия грамматики)

Копирование правил грамматики встречается в алгоритмах довольно часто. Это простая операция, она реализуется библиотекой. Главная цель этого упражнения — научиться формировать перебор правил грамматики.

Первое, что нужно сделать, это получить количество правил:

```
void test2(grammar & g1, grammar & g2) {
    int rule_count = g1.count();
}
```

Затем формируем параметрический цикл по переменной r, обозначающей номер правила, получаем правило в переменную rule, и добавляем его в целевую грамматику методом rule_add_from:

```
void test2(grammar & g1, grammar & g2) {
    int rule_count = g1.count();
    for (int r = 1; r <= rule_count; r++) {
        RULE rule = g1[r];
        g2.rule_add_from(g1, rule);
    }
}
```

Запускаем проект Ctrl+F5.

Убеждаемся, что в грамматике не задан целевой символ. Зададим его при помощи метода set_start_from:

```
g2.set_start_from(g1);
```

Этот метод сам находит целевой символ другой грамматики, регистрирует его идентификатор, и записывает символ как целевой.

Запускаем проект Ctrl+F5.

Мы договариваемся о том, что любой цикл, перебирающий правила грамматики, имеет только такой вид. Это позволит легче ориентироваться в вашем коде другим. К зачету не будут приниматься программы, отступающие от этого соглашения без веского обоснования.

Может возникнуть вопрос — зачем нужна переменная rule? Причина проста. Если бы целью преобразования являлось копирование правил, то в чем бы заключалось преобразование? Обычно копируются только правила, удовлетворяющие некоторому условию, например, не пустые. Для проверки условия нужно иметь само правило, поэтому создается его копия, и она тестируется.

1.7. Упражнение 3. Перебор символов правила

Мы используем код, написанный в предыдущем упражнении. Главная цель этого упражнения — формирование цикла, перебирающего символы правила. Размещаем этот цикл внутри цикла, перебирающего правила:

```
void test2(grammar & g1, grammar & g2) {
    int rule_count = g1.count();
    for (int r = 1; r <= rule_count; r++) {
        RULE rule = g1[r];
        int symbol_count = rule.count();
        for (int s = 1; s <= symbol_count; s++) {
        }
        g2.rule_add_from(g1, rule);
    }
    g2.set_start_from(g1);
}
```

Здесь переменная цикла s обозначает номер символа тела правила.

Мы договариваемся о том, что любой цикл, перебирающий символы правила, имеет только такой вид, какой здесь был применен. К зачету не будут приниматься программы, отступающие от этого соглашения без веского обоснования.

Следующая часть этого упражнения показывает, как определить тип символа: терминал или нетерминал. Вспоминаем, что нетерминалы обозначаются отрицательными числами. Получить символ можно при помощи операции индексирования переменной правила:

```
for (int s = 1; s <= symbol_count; s++) {
    if (rule[s] < 0) {
        printf("nont ");
    } else {
        printf("term ");
    }
}
printf("\n");
```

Запускаем проект Ctrl+F5.

1.8. Упражнение 4. Замена символов правила

Мы используем код, написанный в предыдущем упражнении. Главная цель этого упражнения — практика замены символов правила.

Символ правила можно заменить, если код заменяющего символа является действительным кодом грамматики, и номер заменяемого символа не превышает количество символов в правиле.

Заменим все символы всех правил терминалом *a*. Для этого сначала нужно получить код терминала *a*. Делаем это в начале функции:

```
void test2(grammar & g1, grammar & g2) {
    SYMB a = g1.find_term("a");
    . . .
}
```

В цикле по символам запишем полученный код вместо всех символов:

```
void test2(grammar & g1, grammar & g2) {
    . . .
        for (int s = 1; s <= symbol_count; s++) {
            if (rule[s] < 0) {
                printf("nont ");
            } else {
                printf("term ");
            }
            rule[s] = a;
        }
    . . .
}
```

Запускаем проект Ctrl+F5.

1.9. Упражнение 5. Множество символов SSET

Практически все алгоритмы используют множества символов. Для этой цели в грамматике есть класс SSET. Главная цель этого упражнения — научиться работать с множеством символов.

Задача будет заключаться в том, чтобы найти все достижимые нетерминалы исходной грамматики, сформировав из них множество достижимых символов. Прежде нам нужна грамматика, в которой есть недостижимые символы:

$S \rightarrow A$
$D \rightarrow d$
$C \rightarrow c$
$B \rightarrow b$
$A \rightarrow a \mid B$

Очевидно, достижимыми являются нетерминалы *S*, *A*, *B* (–1, –2, и –5).

Перейдем в FAR Manager. Каталог C:\grom\Debug. Создадим новый файл 101.sxg (Shift+F4). Запишем в файл грамматику в формате SYNAX:

```
<S>
<S>=<A>
<D>=[d]
<C>=[c]
<B>=[b]
<A>=[a]
<A>=<B>
•
```

Сохраним F2 и закроем файл Escape. Открываем свойства проекта и заменяем аргумент командной строки 001 на 101.

Метод поиска всех достижимых символов следующий. Первым достижимым символом является целевой символ. Включаем его в множество достижимых символов *R*. Строим бесконечный цикл, в котором просматриваем все правила грамматики (в своем, внутреннем цикле). Если левый нетерминал правила входит в множество достижимых символов, то и само правило достижимо. Тогда все нетерминалы, входящие в правило, также являются достижимыми, и мы включаем их в множество. Процесс поиска завершается, когда нельзя будет найти ни одного нового нетерминала. Этому соответствует неизменность размера множества в двух итерациях внешнего, бесконечного цикла.

Начинам программирование с того, что определяем множество *R* типа SSET, и включаем в него целевой символ грамматики, точнее, его код:

```
void test3(grammar & g1, grammar & g2) {
    SSET R;
    R.insert(g1.start());
}
```

Ставим точку остановки на закрывающей скобке функции и запускаем программу F5. В окно Watch вводим переменную *R*. Наблюдаем, что в множество включен один символ, код которого –1. Нулевой элемент множества содержит количество символов.

Останавливаем программу Shift+F5.

Теперь нужно определить количество правил входной грамматики. Оно не будет меняться, поэтому делаем это до цикла:

```
void test3(grammar & g1, grammar & g2) {
    SSET R;
    R.insert(g1.start());
    int rule_count = g1.count();
}
```

Теперь формируем внешний бесконечный цикл:

```
void test3(grammar & g1, grammar & g2) {
    SSET R;
    R.insert(g1.start());
    int rule_count = g1.count();
    while (1) {
    }
}
```

Не надо запускать эту программу, она зависнет. Если все же запустили, остановите ее, введя Ctrl+C или Ctrl+Break.

В бесконечном цикле нужно сформировать условие его завершения, которое, как было сказано, заключается в неизменности размера множества. В начале цикла фиксируем количество символов в множестве, а в конце цикла сравниваем зафиксированное значение с текущим количеством:

```
SSET R;
R.insert(g1.start());
int rule_count = g1.count();
while (1) {
    int fix = R.count();
    if (fix == R.count()) break;
}
```

Эту программу, очевидно, можно запустить, поскольку в цикле множество не изменяется. Далее между двумя новыми строчками разместим стандартный цикл по всем правилам входной грамматики:

```
    while (1) {
        int fix = R.count();
        for (int r = 1; r <= rule_count; r++) {
            RULE rule = g1[r];
        }
        if (fix == R.count()) break;
    }
```

Теперь нужно убедиться в том, что левый нетерминал входит в множество *R*, и если входит, то просматриваем все символы этого правила:

```
        for (int r = 1; r <= rule_count; r++) {
            RULE rule = g1[r];
            if (R.contains(rule[0])) {
                int symbol_count = rule.count();
                for (int s = 1; s <= symbol_count; s++) {
                }
            }
        }
```

Проверяем символы правила, и нетерминалы включаем в множество:

```
for (int r = 1; r <= rule_count; r++) {
    RULE rule = g1[r];
    if (R.contains(rule[0])) {
        int symbol_count = rule.count();
        for (int s = 1; s <= symbol_count; s++) {
            SYMB A = rule[s];
            if (A < 0) R.insert(A);
        }
    }
}
```

Выполняя программу шаг за шагом, убеждаемся в том, что в множестве постепенно появляются символы, коды которых –2 и –5.

Наконец, по завершении бесконечного цикла все символы множества выводим в консоль:

```
while (1) {
    . . .
}
int symbol_count = R.count();
for (int s = 1; s <= symbol_count; s++) {
    printf("%s\n", g1.getid(R[s]));
}
```

Для получения идентификатора символа используется метод getid.

Будем считать, что знакомство с библиотекой прошло успешно, и мы готовы программировать алгоритмы преобразований.

1.10. Контрольные вопросы и упражнения

1. Опишите состав формальной грамматики.
2. Опишите формат SYNAX.
3. Перечислите 7 главных правил работы с библиотекой grammar.
4. Как устроен класс грамматики grammar?
5. Как устроен класс правила RULE?
6. Как устроен класс множества символов SSET?

2. Работа PL-102. Группирование правил грамматики

Цели:

- закрепление навыков использования библиотеки grammar.

Задачи:

- разработка алгоритма группирования правил грамматики.

Данный алгоритм не приводится в учебниках по теории языков программирования, однако он необходим на практике для упорядочивания порядка правил грамматики после ее преобразования.

Цель алгоритма заключается в том, чтобы все правила с одинаковым левым нетерминалом были сосредоточены в описании грамматики в одном месте друг за другом. При этом правила должны начинаться с правил для целевого символа, и далее должны описываться правила для нетерминалов в том порядке, в котором они встречаются в предыдущих правилах.

Результат работы алгоритма может получаться разным для одной и той же грамматики из-за произвольного порядка описания правил. Рассмотрим конкретный пример. Пусть в результате преобразований мы получили грамматику, показанную далее слева под меткой «До 1»:

До 1	После 1
$A \rightarrow SA$	$S \rightarrow B$
$A \rightarrow B$	$S \rightarrow AB$
$B \rightarrow b$	$B \rightarrow b$
$A \rightarrow a$	$A \rightarrow SA$
$S \rightarrow B$	$A \rightarrow B$
$S \rightarrow AB$	$A \rightarrow a$

После группирования правил получим порядок правил, приведенный выше справа, под меткой «После 1». Если же два последних правила поменять местами, порядок правил после группирования изменится:

До 2	После 2
$A \rightarrow SA$	$S \rightarrow AB$
$A \rightarrow B$	$S \rightarrow B$
$B \rightarrow b$	$A \rightarrow SA$
$A \rightarrow a$	$A \rightarrow B$
$S \rightarrow AB$	$A \rightarrow a$
$S \rightarrow B$	$B \rightarrow b$

Для тестирования нам понадобится исходная грамматика, показанная выше под меткой «До 1».

Устанавливаем на диск C: проект, полученный в результате предыдущей работы. Переходим в FAR, выбираем каталог C:\grom\Debug. Вводим Shift+F4, вводим название файла 02.sxg, нажимаем Enter и далее вводим текст грамматики в формате SYNAX:

```
<S>
<A>=<S><A>
<A>=<B>
<B>=[b]
<A>=[a]
<S>=<B>
<S>=<A><B>
•
```

Не забываем нажимать Enter в последней строке. Точка • обозначает, где находится конец файла. Сохраняем F2, закрываем Escape.

2.1. Конструирование алгоритма

Формального описания алгоритма нет, его нужно сейчас сконструировать. Пусть G1 — исходная грамматика, G2 — преобразованная. Неформально алгоритм можно описать следующим образом.

Пусть есть два множества. Множество Q содержит неисследованные нетерминалы, а множество P — исследованные. Пусть целевой нетерминал грамматики G1 — это (переменная) S. Тогда первый неисследованный нетерминал множества Q — это S. Это начало алгоритма.

Далее повторяем следующие действия до тех пор, пока есть неисследованные нетерминалы, или, иначе говоря, пока множество Q не пусто.

Пусть A — первый неисследованный нетерминал множества Q. Тогда добавим A в множество исследованных нетерминалов P, и исключим A из множества неисследованных нетерминалов Q. Далее ищем правила, в левой части которых нетерминал A, и если такое правило найдется, то включим его в результирующую грамматику G2.

В первой итерации нетерминалом A окажется S, поскольку только он содержался в Q. Тогда в первой итерации в G2 будут включены все правила для S, то есть для целевого символа.

После того, как правило для A будет добавлено, нужно исследовать тело правила, и найти все нетерминалы B, которые в нем есть. Если найденный нетерминал B отсутствует в множестве исследованных нетерминалов P, то включим его в множество неисследованных нетерминалов Q.

Посмотрим, что произойдет в первой итерации после добавления первого правила для нетерминала S: $S \rightarrow B$. Исследуя символы тела, алгоритм найдет символ B, являющийся нетерминалом, и, поскольку B отсутствует в P, включит его в Q: Q = {B}.

Далее алгоритм обнаружит второе правило для S: $S \rightarrow AB$. Исследуя символы тела этого правила, алгоритм находит нетерминал A, который отсутствует в P, поэтому он включает A в Q: Q = {B, A}. Затем алгоритм находит нетерминал B, отсутствующий в P, поэтому он повторно включает B в множество Q. По определению множества, дважды один символ не включается, поэтому Q = {B, A}.

На этом первая итерация завершается, и, поскольку Q не пусто, начинается вторая итерация. Множество Q содержит два нетерминала, и первый из них B. Действия повторяются. B включается в P, исключается из Q и ищутся правила для B. Правила для B включаются в G2, а в Q ничего не добавляется, поскольку в правилах для B нет других нетерминалов.

Начинается третья итерация, так как в Q есть еще нетерминал A. В результате правила для A включаются в G2, а в Q ничего не добавляется, так как в правилах для A нет неисследованных нетерминалов. Множество Q становится пустым, и цикл исследования завершается.

Теперь можно составить более формальное определение алгоритма.

1. Пусть P — множество.
2. Пусть Q — множество.
3. Пусть S — целевой символ G1.
4. Добавим S в Q.
5. Пусть rule_count — количество правил G1.
6. Пока Q не пусто выполнять 7-18.
7. Пусть A — это первый элемент Q.
8. Добавим A в P.
9. Исключим из Q первый элемент.
10. Просматриваем правила G1 от r =1 до r = rule_count.
11. Пусть rule — правило r грамматики G1.
12. Если левый нетерминал rule совпадает с A то выполнять 13-18.
13. Добавим rule в G2.
14. Пусть symbol_count — количество символов правила rule.
15. Просматриваем символы rule от s = 1 до s = symbol_count.
16. Пусть B — символ s правила rule.
17. Если B — нетерминал, то выполнять 18.
18. Если B отсутствует в P, то включим его в Q.
19. Установить целевой символ G2 равным целевому символу G1.

Этот алгоритм не является единственно возможным, можно найти и другие алгоритмы. Во-первых, заметим, что данный алгоритм находит все достижимые нетерминалы грамматики, постепенно перемещая их из одного множества в другое. В упражнении 5 предыдущей работы мы конструировали алгоритм, который находит достижимые нетерминалы, и порядок нахождения нетерминалов совпадает с порядком нахождения неисследованных нетерминалов. По завершении алгоритма нахождения достижимых нетерминалов множество *R* содержит нетерминалы в порядке их появления в правилах, и остается включить правила для этих нетерминалов в G2.

Сравнивая эти два алгоритма, заметим, что в предлагаемом алгоритме перебор правил грамматики выполняется один раз, а в алгоритме с поиском достижимых нетерминалов перебор выполняется два раза. Из этого можно заключить, что алгоритм с поиском достижимых нетерминалов работает медленнее.

2.2. Рабочее пространство

Откроем свойства проекта и зададим аргументы командной строки равными строке "02 -gr" (файл 02.sxg, алгоритм группирования). Между 02 и ключом должен быть как минимум один пробел.

Работа выполняется в модуле 02-gr.cpp. Откроем его.

В модуле две функции:

```
// группирует правила в порядке появления нетерминалов
void group_rules(grammar & g1, grammar & g2) {
}

// точка входа в алгоритм
// группирование правил
int algorithm_group_rules(grammar & g1, FILE * target) {
    // результирующая грамматика
    grammar g2;
    // алгоритм
    group_rules(g1, g2);
    // выводим грамматику в консоль
    g2.print(stdout);
    // выводим грамматику в файл
    g2.print(target);
    // выходная грамматика
    g1 = g2;
    // результат
    return 1;
}
```

Установим точку остановки на операторе return. Запустим проект F5. Если управление не приходит в точку остановки, что-то задано не так.

2.3. Конструирование функции

Далее конструируем функцию group_rules в соответствии с одним из выбранных вами алгоритмов. Если используется алгоритм, приведенный в этой работе, следует иметь в виду, что фраза вида «Пусть …» или вида «Пусть есть …» обозначает объявление переменной. Рекомендуется использовать имена переменных, приведенные в описании алгоритма, так легче сравнивать функцию с алгоритмом.

Всегда анализируем вывод в консоли.

2.4. Контрольные вопросы и упражнения

1. Докажите, что данный алгоритм находит достижимые нетерминалы грамматики G1.

3. Работа PL-103. Устранение бесплодных символов

Цели:
- изучить алгоритм устранения бесплодных символов грамматики.

Задачи:
- разработать алгоритм проверки тела правила;
- разработать алгоритм нахождения полезных нетерминалов;
- разработать алгоритм нахождения полезных правил.

Опорные документы:
[1, «Устранение бесплодных символов»].
[4, с.269-273, раздел 7.1.1].

3.1. Бесплодные символы

Нетерминал *A* называется бесплодным, если из него не выводится ни одной цепочки терминальных символов.

Рассмотрим грамматику, приведенную в [1, с.26]:

$S \rightarrow a$
$S \rightarrow A$
$A \rightarrow AB$
$B \rightarrow C$
$C \rightarrow b$

Нетерминал *A* этой грамматики не порождает терминальных цепочек. Действительно, выводы из *A* всегда начинаются с *A*:

$$A \Rightarrow AB \Rightarrow AC \Rightarrow Ab \Rightarrow ABb \Rightarrow^* Abb\ldots b$$

Следовательно, нетерминал *A* является бесплодным, и его необходимо удалить из грамматики. При этом должны быть удалены правила 2 и 3.

3.2. Рабочее пространство

Установим на диск C: проект grom, полученный при выполнении предыдущей работы. Создадим файл грамматики. Для этого открываем FAR, переходим в каталог C:\grom\Debug, нажимаем Shift+F4, вводим название файла 03.sxg, вводим текст грамматики в формате SYNAX:

```
<S>
<S>=[a]
<S>=<A>
<A>=<A><B>
<B>=<C>
<C>=[b]
.
```

Сохраним файл F2, закроем редактор Escape.

Откроем свойства проекта и зададим аргументы командной строки: "03 -us".

Работа выполняется в модуле 03-us.cpp. Откроем его.

В модуле несколько функций:

```
// возвращает 1, если все нетерминалы правила в R
int are_known_syms(RULE rule, SSET R) {
    return 1;
}
// формирует множество полезных нетерминалов
void find_usefull_syms(grammar & g1, SSET & R) {
}
// записывает в g2 правила, нетерминалы которых принадлежат R
void find_usefull_rules(grammar & g2, grammar & g1, SSET R) {
}
// удаляет бесплодные символы
void remove_useless(grammar & g1, grammar & g2) {
}
// точка входа в алгоритм
// удаление бесплодных символов
int algorithm_remove_useless(grammar & g1, FILE * target) {
    // результирующая грамматика
    grammar g2;
    // удаление бесплодных символов
    remove_useless(g1, g2);
    // выводим грамматику в консоль
    g2.print(stdout);
    // выводим грамматику в файл
    g2.print(target);
    // выходная грамматика
    g1 = g2;
    // результат
    return 1;
}
```

Установим точку остановки на строке grammar g2 последней функции. Запустим проект F5. Если управление не приходит в точку остановки, что-то задано не так, требуется проверка настроек.

3.3. Конструирование алгоритма

Алгоритм [1, с.26] формирует множество R нетерминалов, из которых выводятся терминальные цепочки. Пусть есть некоторое правило

$$A \rightarrow X_1 X_2 \ldots X_k$$

Здесь X_i — некоторый символ, терминал, или нетерминал. Если все X_i являются терминалами, то A порождает терминальную цепочку (включая пустую), и его можно включить в R. Это базис алгоритма.

Пусть некоторый X_i является нетерминалом. Если $X_i \in R$, значит, X_i порождает терминальную цепочку. Если все нетерминалы X_i входят в R, то A порождает терминальную цепочку. Это индукция алгоритма.

Записывается это следующим образом:

Базис. Если есть правило $A \to \alpha$, и $\alpha \in \Sigma^*$, то $A \in R$.

Индукция. Если есть правило $A \to \alpha$, и $\alpha \in (R \cup \Sigma)^*$, то $A \in R$.

По определению базиса, нужно просматривать правила грамматики и искать правила, в которых нет нетерминалов. По определению индукции, нужно просматривать правила грамматики и искать правила, в которых нет нетерминалов, которых нет в R. Соединяя оба определения, получим, что нужно искать правила, в которых нет нетерминалов, которых нет в R. Если такое правило найдется, то включим левый нетерминал правила в R.

Пусть множество R определено, то есть, просматривая правила, мы не можем больше найти ни одного нового нетерминала. Тогда просматриваем правила исходной грамматики и ищем такие, левый нетерминал которых есть в R, а тело не содержит нетерминалов, которых нет в R. Если такие правила найдутся, то включим их в результирующую грамматику.

Это неформальное описание алгоритма.

3.3.1. Общий алгоритм

Введем понятие частного алгоритма и частной функции. Будем называть алгоритм *частным*, если он описывает часть общего алгоритма. Будем называть функцию частной, если она реализует частный алгоритм.

Рассматривая алгоритм устранения бесплодных символов, можно выделить следующие два частных алгоритма:

1. Вычисление множества R на основе правил исходной грамматики.
2. Поиск правил исходной грамматики, не содержащих нетерминалов, которых нет в R, и включение этих правил в целевую грамматику.

Пусть задана исходная грамматика G1 и целевая грамматика G2.

Частный алгоритм 1.

Вход: грамматика G1, множество R.
Выход: множество R полезных нетерминальных символов.

Частный алгоритм 2.

Вход: грамматика G2, грамматика G1, множество R.
Выход: грамматика G2, не содержащая бесплодных символов.

Тогда общий алгоритм описывается так.

Общий алгоритм 03.

Вход: грамматика G1, грамматика G2.
Выход: грамматика G2, не содержащая бесплодных символов.
1. Установить целевой символ G2 равным целевому символу G1.
2. Пусть R — множество.
3. Применить алгоритм 1 к (G1, R).
4. Применить алгоритм 2 к (G2, G1, R).

3.3.2. Частный алгоритм 1

Частный алгоритм 1.

Вход: грамматика G1, множество *R*.
Выход: множество *R* полезных нетерминальных символов.
1. Пусть rule_count — количество правил G1.
2. Бесконечно выполнять 3-8.
3. Пусть fix равно количество символов *R*.
4. Просматриваем правила G1 от r =1 до r = rule_count.
5. Пусть rule — правило r грамматики G1.
6. Если rule не содержит нетерминалов, которых нет в *R*, то 7.
7. Включить левый нетерминал rule в *R*.
8. Если fix равно количеству символов *R*, то выполнить 9.
9. Конец.

Обратим внимание, что пункт 6 алгоритма 1 описан некоторым общим образом. Выполнение этого пункта предполагает просмотр символов правила, и чтобы не усложнять алгоритм 1, выделим пункт 6 как еще один частный алгоритм 3, решающий свою простую задачу.

3.3.3. Частный алгоритм 3

Цель частного алгоритма 3 — определить, содержит или нет тело некоторого правила только такие нетерминалы, которые есть в *R*. Возвращаемое значение этого алгоритма логическое — правило либо удовлетворяет условиям алгоритма, либо нет. Необходимость этого алгоритма не так очевидна, она определяется в ходе конструирования алгоритма 1, когда алгоритм становится запутанным. Действительно, чтобы выполнить пункт 7 алгоритма 1, нужно выяснить, содержит ли правило rule нетерминалы, входящие в *R*. Для этого нужно конструировать цикл по символам, и результатом этого цикла является признак, равный истине или лжи. Появление в алгоритме некоторого признака, который нужно вычислить, чтобы принять какое-то решение, является явным указанием на то, что требуется вспомогательный алгоритм, который упростит основной алгоритм.

Частный алгоритм 3.

Вход: правило rule, множество *R*.
Выход: возвращает 1, если тело rule не содержит нетерминалов, которых нет в *R*, иначе возвращает 0.
1. Пусть symbol_count — количество символов тела правила rule.
2. Просматриваем символы rule от s = 1 до s = symbol_count.
3. Пусть *A* — символ s правила rule.
4. Если *A* — нетерминал и *A* не входит в *R*, то вернуть 0, конец.
5. Если все символы просмотрены, вернуть 1.

3.3.4. Частный алгоритм 2

Частный алгоритм 2.

Вход: грамматика G2, грамматика G1, множество *R*.
Выход: грамматика G2, не содержащая бесплодных символов.
1. Пусть rule_count — количество правил G1.
2. Просматриваем правила G1 от r =1 до r = rule_count.
3. Пусть rule — правило r грамматики G1.
4. Если левый нетерминал rule входит в *R*, то 5.
5. Если rule не содержит нетерминалов, которых нет в *R*, то 6.
6. Добавить правило rule в G2.

Заметим, что пункт 5 этого алгоритма равен пункту 6 алгоритма 1.

3.4. Конструирование функций

Алгоритму 3 соответствует функция are_known_syms.
Алгоритму 1 соответствует функция find_usefull_syms.
Алгоритму 2 соответствует функция find_usefull_rules.
Общему алгоритму соответствует функция remove_useless.
Порядок определения функций значения не имеет, но предпочтительным представляется следующий:
- общий алгоритм.
- алгоритм 3.
- алгоритм 1.
- убеждаемся, что вычисляется правильное множество *R*.
- алгоритм 2.

3.5. Контрольные вопросы и упражнения

1. Дайте определение бесплодного символа.
2. Сформулируйте определение базиса данного алгоритма.
3. Сформулируйте определение индукции данного алгоритма.
4. Докажите, что определение индукции перекрывает (включает в себя) определение базиса.
5. Вычислите алгоритмическую сложность данного алгоритма в зависимости от количества правил грамматики.
6. Вычислите алгоритмическую сложность данного алгоритма в зависимости от количества нетерминальных символов.
7. Предложите усовершенствование реализации данного алгоритма.

4. Работа PL-104. Устранение недостижимых символов

Цели:
- изучить алгоритм устранения недостижимых символов грамматики.
Задачи:
- разработать алгоритм поиска нетерминалов тела правила;
- разработать алгоритм нахождения достижимых нетерминалов;
- разработать алгоритм нахождения достижимых правил.
Опорные документы:
[1, , «Устранение недостижимых символов»].
[4, с.269-273, раздел 7.1.1].

4.1. Недостижимые символы

Пусть задана грамматика G = (Σ, N, P, *S*), $V = \Sigma \cup N$.

Символ $X \in V$ называется недостижимым, если он не встречается ни в одной сентенциальной форме грамматики. Очевидно, что недостижимые символы являются бесполезными, и их нужно удалить из грамматики.

Рассмотрим грамматику, которая получается после удаления бесплодных символов в предыдущей работе. Она приведена также в [1, с.28]:

$S \rightarrow a$
$B \rightarrow C$
$C \rightarrow b$

Очевидно, что достижимыми символами в этой грамматике являются символы *S* и *a*, так как единственно возможный вывод содержит только их:

$S \Rightarrow a$

4.2. Рабочее пространство

Установим на диск C: проект grom, полученный при выполнении предыдущей работы. Для выполнения этой работы используется грамматика, формируемая в предыдущей работе.

Откроем свойства проекта и зададим аргументы командной строки:
"03 -us -ur".

В командной строке два ключа, это не ошибка.

Работа выполняется в модуле 04-ur.cpp. Откроем его.

В модуле несколько функций:

```
// добавляет нетерминалы правила rule в множество R
void find_rule_syms(RULE rule, SSET & R) {
}
// формирует множество достижимых нетерминалов
void find_reachable_syms(grammar & g1, SSET & R) {
}
```

```
// записывает в g2 правила, левые символы которых принадлежат R
void find_reachable_rules(grammar & g2, grammar & g1, SSET R) {
}
// удаляет недостижимые символы
void remove_unreachable(grammar & g1, grammar & g2) {
}
// точка входа в алгоритм
// удаление недостижимых символов
int algorithm_remove_unreachable(grammar & g1, FILE * target) {
    // результирующая грамматика
    grammar g2;
    // удаление недостижимых символов
    remove_unreachable(g1, g2);
    // выводим грамматику в консоль
    g2.print(stdout);
    // выводим грамматику в файл
    g2.print(target);
    // выходная грамматика
    g1 = g2;
    // результат
    return 1;
}
```

Установим точку остановки на строке grammar g2 последней функции. Запустим проект F5. Если управление не приходит в точку остановки, что-то задано не так, требуется проверка настроек.

4.3. Конструирование алгоритма

Алгоритм формирует множество R достижимых символов.

Целевой символ грамматики достижим по определению, включим его в R. Это базис алгоритма. Пусть есть некоторое правило

$$A \rightarrow X_1 X_2 \ldots X_k$$

Здесь X_i — некоторый символ, терминал, или нетерминал. Пусть левый нетерминал A правила входит в множество R. Тогда любой из символов X_i является достижимым. Это индукция алгоритма.

Записывается это следующим образом:

Базис. $S \in R$ (по определению).

Индукция. Если есть правило $A \rightarrow \alpha X \beta$, и $A \in R$, то $X \in R$, $\alpha, \beta \in V^*$.

По определению базиса целевой символ грамматики является достижимым. По определению индукции, достижимым является любой символ тела правила, левый нетерминал которого входит в R.

Пусть множество R определено. Тогда просматриваем правила исходной грамматики, и ищем такие, левый нетерминал которых есть в R. Если такие правила найдутся, то добавим их в результирующую грамматику.

Данный алгоритм подразумевает, что множество R содержит как нетерминальные, так и терминальные символы. Можно доказать, что алгоритм будет корректным, если в R включать только нетерминалы.

Доказательство предлагается в качестве упражнения.

4.3.1. Общий алгоритм

Алгоритм устранения недостижимых символов состоит из следующих двух частей, соответствующих частным алгоритмам:

1. Вычисление множества достижимых нетерминальных символов *R*.
2. Поиск правил исходной грамматики, левые нетерминалы которых есть в *R*, и включение этих правил в целевую грамматику.

Пусть задана исходная грамматика G1 и целевая грамматика G2.

Частный алгоритм 1.

Вход: грамматика G1, множество *R*.
Выход: множество *R* достижимых нетерминальных символов.

Частный алгоритм 2.

Вход: грамматика G2, грамматика G1, множество *R*.
Выход: грамматика G2, не содержащая недостижимых символов.

Тогда общий алгоритм описывается так.

Общий алгоритм 04.

Вход: грамматика G1, грамматика G2.
Выход: грамматика G2, не содержащая недостижимых символов.
1. Пусть *S* — целевой символ G2, равный целевому символу G1
2. Пусть *R* — множество, состоящее из *S*.
3. Применить алгоритм 1 к (G1, *R*).
4. Применить алгоритм 2 к (G2, G1, *R*).

4.3.2. Частный алгоритм 1

Частный алгоритм 1.

Вход: грамматика G1, множество *R*.
Выход: множество *R* достижимых нетерминальных символов.
1. Пусть rule_count — количество правил G1.
2. Бесконечно выполнять 3-8.
3. Пусть fix равно количество символов *R*.
4. Просматриваем правила G1 от r =1 до r = rule_count.
5. Пусть rule — правило r грамматики G1.
6. Если левый нетерминал правила rule есть в *R*, то выполнить 7.
7. Включить нетерминалы тела правила rule в *R*.
8. Если fix равно количеству символов *R*, то выполнить 9.
9. Конец.

Выполнению пункта 7 соответствует частный алгоритм 3.

4.3.3. Частный алгоритм 3

Частный алгоритм 3.

Вход: правило rule, множество *R*.
Выход: множество *R*.
1. Пусть symbol_count — количество символов тела правила rule.
2. Просматриваем символы rule от s = 1 до s = symbol_count.
3. Пусть *A* — символ s правила rule.
4. Если *A* — нетерминал, то включить его в *R*.

4.3.4. Частный алгоритм 2

Частный алгоритм 2.

Вход: грамматика G2, грамматика G1, множество *R*.
Выход: грамматика G2, не содержащая недостижимых символов.
1. Пусть rule_count — количество правил G1.
2. Просматриваем правила G1 от r =1 до r = rule_count.
3. Пусть rule — правило r грамматики G1.
4. Если левый нетерминал rule входит в *R*, то добавить rule в G2.

4.4. Конструирование функций

Алгоритму 3 соответствует функция find_rule_syms.
Алгоритму 1 соответствует функция find_reachable_syms.
Алгоритму 2 соответствует функция find_reachable_rules.
Общему алгоритму соответствует функция remove_unreachable.

4.5. Контрольные вопросы и упражнения

1. Дайте определение недостижимого символа.
2. Сформулируйте определение базиса данного алгоритма.
3. Сформулируйте определение индукции данного алгоритма.
4. Вычислите алгоритмическую сложность данного алгоритма в зависимости от количества правил.
5. Вычислите алгоритмическую сложность данного алгоритма в зависимости от количества нетерминальных символов.
6. Поменяйте ключи командной строки местами, сначала ключ -ur, затем -us. Объясните результат выполнения программой алгоритмов 04 и 03.
7. Докажите, что включение в множество *R* только нетерминальных символов не нарушает корректность алгоритма.

5. Работа PL-105. Устранение пустых правил

Цели:
- изучение алгоритма устранения пустых правил грамматики.

Задачи:
разработать алгоритмы для следующих частных целей:
- определение вырождающегося правила;
- определение вырождающихся символов грамматики;
- поиск непустых правил;
- удаление вырождающихся символов из правила;
- формирование правил, компенсирующих пустые правила;
- формирование стартового символа целевой грамматики.

Опорные документы:
[1, «Устранение пустых правил»].
[4, с.273-276, раздел 7.1.3].

5.1. Изучение алгоритма

Пусть задана грамматика $G = (\Sigma, N, P, S)$, $V = \Sigma \cup N$.

Правило называется пустым (λ-правилом), если тело правила пусто (правило генерирует пустую цепочку). В одних классах грамматик пустые правила являются необходимыми, в других классах грамматик пустые правила являются недопустимыми, поэтому нужен алгоритм их устранения.

Алгоритм строит множество R вырождающихся символов. Некоторый символ $A \in N$ называют вырождающимся символом, если какое-либо правило для этого символа может породить пустую цепочку. Такое правило также называют вырождающимся.

Пустое правило является вырождающимся, поэтому левый нетерминал пустого правила включаем в R. Это определение базиса алгоритма.

Пусть есть правило

$$A \to X_1 X_1 \ldots X_k$$

Если каждый из символов X_i является вырождающимся, то и правило является вырождающимся, поэтому A включаем в множество R. Это определение индукции алгоритма.

Записывается это следующим образом:

Базис. Если есть правило $A \to \lambda$, то $A \in R$.

Индукция. Если есть правило $A \to \alpha$, и $\alpha \in R^+$, то $A \in R$.

Пусть множество R определено. Включим в целевую грамматику все непустые правила исходной грамматики и будем их рассматривать.

Пусть правило имеет следующий вид:

$$A \to XXYZ$$

Пусть символы X и Z являются вырождающимися. Тогда это правило должно дополнительно порождать цепочки XYZ, YZ, XY, XXY и Y. Эти цепочки получаются последовательным исключением из цепочки $XXYZ$ всех возможных комбинаций символов X и Z. Это обеспечит эквивалентность цепочек, порождаемых целевой грамматикой. Заметим, что после того, как в целевую грамматику были добавлены непустые правила, никакой символ целевой грамматики, за исключением целевого, не может вырождаться.

После формирования дополнительных правил получим грамматику без пустых правил, эквивалентную исходной минус пустая цепочка. Чтобы обеспечить полную эквивалентность, нужно определить целевой символ. Если целевой символ исходной грамматики не является вырождающимся, тогда он является целевым символом результирующей грамматики. В противном случае создается новый нетерминал S' и правила $S' \to S$ и $S' \to \lambda$.

5.2. Рабочее пространство

Установим на диск C: проект grom, полученный при выполнении предыдущей работы. Если в каталоге C:\grom\Debug нет файла 05.sxg, тогда его нужно создать. Для этого открываем FAR, переходим в каталог C:\grom\Debug, нажимаем Shift+F4, вводим название файла 05.sxg, и вводим текст грамматики в формате SYNAX:

```
<S>
<S>=<A><B>
<A>=[a]
<A>=<B><B>
<B>=[b]
<B>=.
.
```

Точка показывает конец файла.

Сохраним F2, закроем редактор Escape.

Откроем свойства проекта и зададим аргументы командной строки:

"05 -re -gr"

Работа выполняется в модуле 05-re.cpp. Откроем его.

В модуле много функций:

```
// возвращает 1, если правило вырождающееся
int is_nullable_rule(RULE rule, SSET R) {
    return 1;
}

// формирует множество вырождающихся нетерминалов R
void nullable_syms(grammar & g1, SSET & R) {
}

// непустые правила копирует в g2
void useful_rules(grammar & g2, grammar g1) {
}
```

```
// удаляет символ N списка P из правила rule
// если полученное правило не пусто, добавляет его в g2
// рекурсивно вызывает себя для полученного остатка правила
// последовательно для символов N-1, N-2...
void remove_nullable_from(grammar & g2, RULE rule, SSET P, int N) {
}

// формирует список вырождающихся символов правила
void rule_nullable_syms(RULE rule, SSET R, SSET & P) {
}

// формирует правила, компенсирующие удаленные пустые правила
void complement_rules(grammar & g2, SSET R) {
}

// устраняет пустые правила
void remove_empty(grammar & g1, grammar & g2) {
}

// точка входа в алгоритм
// устранение пустых правил
int algorithm_remove_empty(grammar & g1, FILE * target) {
    // результирующая грамматика
    grammar g2;
    // алгоритм
    remove_empty(g1, g2);
    // выводим грамматику в консоль
    g2.print(stdout);
    // выводим грамматику в файл
    g2.print(target);
    // выходная грамматика
    g1 = g2;
    // результат
    return 1;
}
```

Установим точку остановки на строке grammar g2 последней функции. Запустим проект F5. Если управление не приходит в точку остановки, что-то задано не так, требуется проверка настроек.

5.3. Формирование множества вырождающихся символов

Переходим в функцию основного алгоритма remove_empty. Устанавливаем целевой символ, описываем множество и первый шаг:

```
void remove_empty(grammar & g1, grammar & g2) {
    // целевой символ G2
    g2.set_start_from(g1);
    // множество вырождающихся нетерминалов
    SSET R;
    // шаг 1 - формируем R
    nullable_syms(g1, R);
}
```

Частная функция nullable_syms вычисляет множество вырождающихся нетерминалов. Для этого она использует вспомогательную частную

функцию is_nullable_rule, которая проверяет, является ли конкретное правило вырождающимся. Видимо, с этой функции и следует начать.

Функция проверяет символы правила rule, которое ей передается.

Кроме правила, функции передается множество *R*, которое содержит уже обнаруженные вырождающиеся нетерминалы. Если правило содержит символ, которого нет в *R*, функция is_nullable_rule немедленно завершается, возвращая 0. Если все символы правила просмотрены, и при этом функция не завершилась, то функция возвращает 1.

Алгоритм функции простой:

1. Пусть symbol_count — количество символов правила.
2. Если symbol_count равно нулю, завершить алгоритм, вернуть 1.
3. Просматриваем символы rule от s = 1 до s = symbol_count.
4. Пусть *X* — символ номер s правила rule.
5. Если *X* не входит в *R*, завершить алгоритм, вернуть 0.
6. Если все символы просмотрены, вернуть 1.

Реализуем этот алгоритм и переходим к функции nullable_syms, которая находит все вырождающиеся нетерминалы.

Алгоритм функции nullable_syms:

1. Пусть rule_count — количество правил G1.
2. Бесконечно выполнять 3-8.
3. Пусть fix равно количество символов *R*.
4. Просматриваем правила G1 от r =1 до r = rule_count.
5. Пусть rule — правило номер r грамматики G1.
6. Если правило rule вырождающееся, то выполнить 7.
7. Включить левый нетерминал правила в *R*.
8. Если fix равно количеству символов *R*, то выполнить 9.
9. Конец.

Описываем этот алгоритм и отлаживаем программу.

Убеждаемся, что формируется множество, в которое входят все нетерминалы G1.

5.4. Копирование непустых правил

Функция remove_empty.

Описываем следующий шаг алгоритма:

```
void remove_empty(grammar & g1, grammar & g2) {
    . . .
    // шаг 1 - формируем R
    nullable_syms(g1, R);
    // шаг 2 - ищем непустые правила
    useful_rules(g2, g1);
}
```

Следующий шаг заключается в том, чтобы скопировать непустые правила в целевую грамматику.

Для этого просматриваем все правила исходной грамматики, и если будет найдено непустое правило, включим его в целевую грамматику.

Эти действия выполняются в функции useful_rules.

Алгоритм функции useful_rules:

1. Пусть rule_count — количество правил G1.
2. Просматриваем правила G1 от r =1 до r = rule_count.
3. Пусть rule — правило номер r грамматики G1.
4. Если правило rule непустое, то выполнить 5.
5. Включить правило в G2.
6. Следующее правило.

Реализуем этот алгоритм.

В результате выполнения этого шага должна получиться грамматика:

$S \rightarrow AB$
$A \rightarrow a \mid BB$
$B \rightarrow b$

Эта грамматика не содержит пустых правил, но не является эквивалентной исходной, так как порождает не все цепочки из тех, что порождала исходная грамматика. Поэтому нужен еще один шаг, формирующий новые правила, которые компенсируют удаленные пустые правила.

5.5. Формирование компенсирующих правил

В функции remove_empty описываем шаг 3:

```
void remove_empty(grammar & g1, grammar & g2) {
    . . .
    // шаг 1 - формируем R
    nullable_syms(g1, R);
    // шаг 2 - ищем непустые правила
    useful_rules(g2, g1, R);
    // проверяем множество R
    if (R.count() == 0) {
        // конец - нет вырождающихся символов
        return;
    }
    // шаг 3 - формируем новые правила
    complement_rules(g2, R);
}
```

Формирование новых компенсирующих правил выполняет функция complement_rules. Однако сначала нужно определить две вспомогательные функции. Для этого рассмотрим, как можно исключить из правила все возможные комбинации вырождающихся нетерминалов.

Пусть есть множество P и некоторое правило rule. Тогда просмотрим все символы правила rule, и если какой-то символ есть в R, то включим в P порядковый номер этого нетерминала. Для примера рассмотрим правило

$A \rightarrow xXyXZ$. Пусть вырождающимися являются символы X и Z. Тогда мы должны получить множество P, содержащее номера {2, 4, 5}.

Пусть есть рекурсивная функция, которая удаляет из тела переданного ей правила символ, номер которого задан порядковым номером N числа, записанного в P, после чего рекурсивно вызывает себя, передавая в качестве правила полученный остаток правила и последовательно номера N, начиная от номера N–1 до 1. Все получающиеся остатки правил функция записывает в целевую грамматику при условии, что остаток не пустой.

Сначала функция получает правило с телом $xXyXZ$, и $N = 3$. Функция получает также множество P.

Функция удаляет символ 5, равный $P[3]$, получая остаток $xXyX$.

Остаток $xXyX$ передается этой же функции с $N = 2$.

Функция удаляет символ 4, равный $P[2]$, получая остаток xXy.

Остаток xXy передается этой же функции с $N = 1$.

Функция удаляет символ 2, равный $P[1]$, получая остаток xy.

Остаток $xXyX$ передается этой же функции с $N = 1$.

Функция удаляет символ 2, равный $P[1]$, получая остаток xyX.

Теперь начнем с начального правила, но $N = 2$.

Функция удаляет символ 4, равный $P[2]$, получая остаток $xXyZ$.

Остаток $xXyZ$ передается этой же функции с $N = 1$.

Функция удаляет символ 2, равный $P[1]$, получая остаток xyZ.

Наконец, начнем с начального правила, но $N = 1$.

Функция удаляет символ 2, равный $P[1]$, получая остаток $xyXZ$.

Перейдем к реализации функций.

Сначала рассмотрим функцию rule_nullable_syms, которая формирует множество P. Эта функция проста. Она просматривает символы правила и обнаружив символ, который есть в R, включает номер символа в P.

Алгоритм функции rule_nullable_syms:

1. Пусть symbol_count — количество символов правила rule.
2. Просматриваем символы rule от s = 1 до s = symbol_count.
3. Пусть X — символ номер s правила rule.
4. Если X присутствует в R, то выполнить 5.
5. Включить s в P.
6. Если просмотрены все символы, то конец.

Реализуем этот алгоритм. Проверит его следующая функция.

Алгоритм функции complement_rules:

1. Пусть rule_count — количество правил G2.
2. Просматриваем правила G2 от r =1 до r = rule_count.
3. Пусть rule — правило номер r грамматики G2.
4. Пусть P — множество.
5. Вычислим множество P при помощи предыдущего алгоритма.
6. Пусть p_count — количество символов в P.
7. Пока конец алгоритма.

Реализуем этот алгоритм.

Отлаживаем программу. Убеждаемся, что формируются множества {1, 2}, {}, {1, 2} и {} для соответственно правил 1, 2, 3 и 4.

Остается алгоритм удаления вырождающихся нетерминалов.

Алгоритм рекурсивной функции remove_nullable_from.

На входе: грамматика G2, правило rule, множество *P*, номер *N*.

1. Удалить из rule символ номер *P*[*N*].

2. Если правило rule не пусто, то выполнить 3.

3. Добавить правило rule в G2.

4. Для каждого *M* от *N*–1 до 1 вызвать рекурсивно алгоритм, передавая G2, rule, *P* и *M*.

Реализуем этот алгоритм.

Возвращаемся к алгоритму функции complement_rules:

6. Пусть p_count — количество символов в *P*.

7. Для каждого *N* от p_count до 1 вызвать remove_nullable_from, передавая G2, rule, *P* и *N*.

Отлаживаем эту часть программы.

В результате должна формироваться грамматика:

$S \rightarrow AB \mid B \mid A$
$A \rightarrow a \mid BB \mid B$
$B \rightarrow b$

Теперь почти все выполнено, за исключением определения целевого символа G2, если целевой символ G1 является вырождающимся.

Если в грамматике G2 только одно правило, завершаем функцию, так как целевой символ G2 установлен и неважно, какое правило осталось. Если целевой символ G1 не входит в *R*, то функцию также можно завершить.

Если две указанные проверки не завершили функцию, добавляем новый нетерминал, два правила для него, и делаем новый символ целевым:

```
// шаг 4 - новый целевой символ
// правило для целевого символа
RULE rule;
// новый целевой символ
// SYMB S1 = g2.get_new_nont(); // другой вариант нового символа
SYMB S1 = g2.get_star_nont(g1.start());
// правило <S'>=.
rule[0] = S1;
g2.rule_add(rule);
// правило <S'>=<S>
rule.append(g2.start());
g2.rule_add(rule);
// целевой символ грамматики g2
g2.set_start(S1);
```

Тестируем эту часть кода.

Убеждаемся, что формируется новый целевой символ.

Окончательно получаем грамматику:

$S' \to S \mid \lambda$
$S \to AB \mid B \mid A$
$A \to a \mid BB \mid B$
$B \to b$

5.6. Контрольные вопросы и упражнения

1. Какое правило называется пустым?
2. Какая грамматика называется грамматикой без пустых правил?
3. Дайте определение вырождающемуся символу.
4. Поясните алгоритм поиска недостижимых символов.
5. Поясните принцип устранения пустых правил.
6. Поясните алгоритм удаления вырождающихся символов из правил.
7. Докажите, что алгоритм определения вырождающихся символов находит все вырождающиеся символы и только их.
8. Докажите, что $L(\text{G2}) = L(\text{G1})$.

6. Работа PL-106. Устранение цепных правил

Цели:
- изучение алгоритма устранения цепных правил грамматики.
Задачи:
разработать алгоритмы для следующих частных целей:
- поиск не цепных правил и включение их в целевую грамматику;
- поиск цепных символов для некоторого нетерминала;
- формирование новых правил.
Опорные документы:
[1, «Устранение цепных правил»].
[3, с.180].

6.1. Рабочее пространство

Установим на диск C: проект grom, полученный при выполнении предыдущей работы. Зададим аргументы командной строки:

"05 -re -gr -rc -gr"

Работа выполняется в модуле 06-rc.cpp. Откроем его.

В модуле несколько функций:

```
// ищет не цепные правила g1 и добавляет их в g2
void nonunit_rules(grammar & g1, grammar & g2) {
}

// ищет цепочку нетерминала A
void find_chain_for(grammar & g1, SSET & R) {
}

// формирует новые не цепные правила
void nonunit_rules_for(grammar & g2, grammar & g1, SSET R) {
}

// устраняет цепные правила
void remove_units(grammar & g1, grammar & g2) {
}

// точка входа в алгоритм
// устранение цепных правил
int algorithm_remove_units(grammar & g1, FILE * target) {
    // результирующая грамматика
    grammar g2;
    // алгоритм
    remove_units(g1, g2);
    // выводим грамматику в консоль
    g2.print(stdout);
    // выводим грамматику в файл
    g2.print(target);
    // выходная грамматика
    g1 = g2;
    // результат
    return 1;
}
```

Установим точку остановки на строке grammar g2 последней функции. Запустим проект F5. Если управление не приходит в точку остановки, что-то задано не так, требуется проверка настроек.

6.2. Исследование алгоритма

Цепным называется правило вида $A \to B$, $AB, \in N$. Цепные правила могут вести к образованию циклов. Циклом, или циклическим выводом называется вывод $A \Rightarrow^* A$. Очевидно, что такой вывод бесполезен, он приводит к зацикливанию алгоритмов разбора.

В качестве примера будем рассматривать грамматику, полученную в результате работы предыдущего алгоритма:

$S' \to S \mid \lambda$
$S \to AB \mid B \mid A$
$A \to a \mid BB \mid B$
$B \to b$

Цепными в ней являются правила $S' \to S$, $S \to B$, $S \to A$, $A \to B$.

Сначала удалим из грамматики все цепные правила:

$S' \to \lambda$
$S \to AB$
$A \to a \mid BB$
$B \to b$

Если в исходной грамматике есть цепное правило $A \to B$, и в результирующей грамматике есть не цепное правило $B \to \alpha$, то добавим в результирующую грамматику правило $A \to \alpha$.

Тогда для правила $A \to B$ следует добавить правило $A \to b$, для правила $S \to B$ следует добавить правило $S \to b$, для правила $S \to A$ следует добавить правила $S \to a$ и $S \to BB$, для правила $S' \to S$ следует добавить правила $S' \to AB$, $S' \to a$, $S' \to BB$, $S' \to b$.

В результате этих преобразований получим грамматику:

$S' \to \lambda \mid AB \mid a \mid BB \mid b$
$S \to AB \mid a \mid BB \mid b$
$A \to a \mid BB \mid b$
$B \to b$

В этой грамматике символ S является недостижимым, поэтому после данного алгоритма нужно устранять недостижимые символы.

Для формирования новых правил сначала нужно найти все цепочки вида $A \to B \to C \ldots$. Эти цепочки формируют множество цепных символов R нетерминала A. Множество R для A формируется следующим образом.

Базис. $A \in R$.

Индукция. Если есть цепное правило $B \to C$, и $B \in R$, тогда $C \in R$.

После того, как множество R нетерминала A будет построено, то просматриваем не цепные правила, и если есть правило $B \to \alpha$, и $A \neq B$, и $B \in R$, то добавим в грамматику правило $A \to \alpha$.

6.3. Поиск не цепных правил

Сначала включим в результирующую грамматику все не цепные правила исходной грамматики. В главной функции remove_units формируем начальные действия:

```
void remove_units(grammar & g1, grammar & g2) {
    // установим целевой символ
    g2.set_start_from(g1);
    // ищем не цепные правила
    nonunit_rules(g1, g2);
}
```

Переходим в функцию nonunit_rules и формируем в ней стандартный алгоритм перебора правил грамматики G1. Каждое правило грамматики G1 проверяем, и если оно не цепное, то включаем его в грамматику G2.

Если некоторое правило цепное, то метод RULE::is_unit возвращает истину. Таким образом можно определить, является ли правило цепным или не цепным.

Алгоритм перебора правил неоднократно был использован ранее и нет необходимости описывать его здесь подробно. Программируем этот алгоритм и убеждаемся, что результирующая грамматика имеет вид:

$S' \to \lambda$
$S \to AB$
$A \to a \mid BB$
$B \to b$

Поскольку в командной строке за преобразованием указан алгоритм группирования, который устраняет недостижимые нетерминалы, то после группирования мы увидим грамматику из одного пустого правила. Это нормально на данном этапе.

6.4. Формирование множества цепных символов

Следующая задача — сформировать функцию find_chain_for, которая вычислит множество цепных нетерминалов R для нетерминала A. Нетерминал A будет записан в множество R, которое передается функции.

Формирование подобного множества также неоднократно рассматривалось ранее, поэтому алгоритм функции только в общих чертах. Это бесконечный цикл, в котором просматриваются правила грамматики G1. Если будет найдено цепное правило, в котором левый нетерминал входит в R, то правый нетерминал также включаем в R. Бесконечный цикл завершается, когда в двух итерациях R не изменится.

6.5. Формирование новых правил

Следующая часть алгоритма формирует новые не цепные правила для некоторого нетерминала A, который является первым символом множества R. Этому алгоритму соответствует функция nonunit_rules_for.

Пусть r_count — число символов R.

Просматриваем символы s множества R от 2-го до r_count.

Просматриваем правила грамматики G2.

Пусть rule — текущее правило G2.

Если левый нетерминал совпадает с символом R[s], заменяем левый нетерминал символом R[1], и включаем rule в грамматику G2.

Программируем функцию nonunit_rules_for.

6.6. Перебор нетерминалов исходной грамматики

Следующая часть алгоритма перебирает все нетерминалы грамматики, ищет для каждого нетерминала цепочку цепных символов (множество R), и формирует новые правила для данного нетерминала. Эту часть программируем в функции remove_units. Если last — количество нетерминалов, то перебирая числа от 1 до last включительно, получим отрицательные значения индексов нетерминалов:

```
void remove_units(grammar & g1, grammar & g2) {
    // установим целевой символ
    g2.set_start_from(g1);
    // ищем не цепные правила
    nonunit_rules(g1, g2);
    // отрицательный индекс последнего нетерминала
    SYMB last = g1.nont_count();
    // цикл по всем нетерминалам
    for (SYMB A = 1; A <= last; A++) {
        // рабочее множество = сам нетерминал
        SSET R = -A;
        // ищем цепочку нетерминала A
        find_chain_for(g1, R);
        // формируем новые правила для A
        nonunit_rules_for(g2, g1, R);
    }
}
```

Программируем эту часть алгоритма и тестируем алгоритм в целом.

Результирующая грамматика:

$S' \to \lambda \mid AB \mid a \mid BB \mid b$

$S \to AB \mid a \mid BB \mid b$

$A \to a \mid BB \mid b$

$B \to b$

В ней символ S является недостижимым. Поэтому после удаления цепных правил следует устранять недостижимые символы.

Удаление недостижимых нетерминалов выполняет также группирование правил грамматики. После этого в грамматике останутся правила:

$S' \to \lambda \mid AB \mid a \mid BB \mid b$
$A \to a \mid BB \mid b$
$B \to b$

6.7. Контрольные вопросы и упражнения

1. Какое правило называется цепным?
2. Что называется циклом?
3. Поясните принцип устранения цепных правил.
4. Поясните алгоритм построения множества цепных символов R.
5. Докажите, что алгоритм построения множества R находит все цепные символы нетерминала A и только их.
6. Докажите, что $L(\text{G2}) = L(\text{G1})$.

7. Работа PL-107. Устранение левой рекурсии

Цели:
- изучение алгоритма устранения левой рекурсии
Задачи:
- устранение явной рекурсии;
- замена нетерминала его правилами;
- преобразование неявной рекурсии в явную;
- устранение рекурсии в целом;

Опорные документы:
[1, «Устранение левой рекурсии»]
[3, с.180]

7.1. Изучение алгоритма

Алгоритм устранения левой рекурсии использует два важных шага.

1. Если нетерминал A леворекурсивный по правилу, то явная рекурсия устраняется формированием двух групп правил таких, в которых левая рекурсия заменена правой рекурсией.

2. Если для нетерминала A_i существует правило $A_i \rightarrow A_j\alpha$, где $j < i$, тогда нетерминал A_j заменяется телами своих правил. При этом неявная рекурсия, если она есть, преобразуется в явную, и устраняется на шаге 1 следующей итерации алгоритма.

Шаги применяются в последовательности 1–2 с использованием всех возможных комбинаций i и j.

Особенность алгоритма в том, что очень сложно формировать новую грамматику, преобразуя правила исходной. Вместо этого будем преобразовывать непосредственно исходную грамматику.

В качестве примера будем рассматривать грамматику 07:

$A \rightarrow Bb \mid a$
$B \rightarrow Aa \mid Bb \mid b$

Выполняя шаг 1 для нетерминала A, обнаруживаем, что он нелеворекурсивный, поэтому ничего не делаем на этом шаге.

Выполняя шаг 2 для нетерминала B, обнаруживаем, что есть правило $B \rightarrow Aa$, где индекс нетерминала A меньше индекса нетерминала B (по абсолютной величине). Тогда заменяем правило $B \rightarrow Aa$ на правила $B \rightarrow Bba$ и $B \rightarrow aa$, и получаем грамматику:

$A \rightarrow Bb \mid a$
$B \rightarrow Bba \mid aa \mid Bb \mid b$

В результате для нетерминала B проявилась явная рекурсия.

Выполняя шаг 1 для нетерминала B, устраняем явную рекурсию.

Сначала определим цепочки α и β. Цепочки α следуют за нетерминалом *B*, а цепочки β не начинаются с нетерминала *B*. Тогда:

$\alpha_1 = ba$, $\alpha 2 = b$, $\beta_1 = aa$, $\beta_1 = b$.

Первая группа правил для нетерминала *B*:

$B \rightarrow \beta_1 \mid \beta_2 \mid \beta_1 B' \mid \beta_2 B'$

Вторая группа правил для нетерминала *B′*:

$B' \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_1 B' \mid \alpha_2 B'$

Выполняя это преобразование, получим грамматику:

$A \rightarrow Bb \mid a$

$B \rightarrow aa \mid b \mid aaB' \mid bB'$

$B' \rightarrow ba \mid b \mid baB' \mid bB'$

Эта грамматика окончательная.

Пусть правила для *B* проще:

$B \rightarrow B\alpha \mid \beta$

Тогда преобразование шага 1 дает следующие правила:

$B \rightarrow \beta \mid \beta B'$

$B' \rightarrow \alpha \mid \alpha B'$

Рассмотрим вывод из *B* по исходным правилам:

$B \Rightarrow B\alpha \Rightarrow \beta\alpha$

Вывод из *B* по преобразованным правилам:

$B \Rightarrow \beta B' \Rightarrow \beta\alpha$

Можно доказать, что любая цепочка, выводимая из *B* по исходным правилам, выводится из *B* по преобразованным правилам. Из *B* выводятся цепочки, начинающиеся с β, за которыми следует произвольное количество цепочек α, в том числе нулевое. Из *B′* выводится произвольное количество цепочек α, не меньшее одной, а из *B* выводится нуль цепочек α.

На этом можно строить доказательство. Если таким образом доказать корректность шага 1, то корректность шага 2 доказывается просто.

Поскольку других преобразований в алгоритме нет, то доказательство корректности двух шагов доказывает корректность алгоритма в целом.

7.2. Рабочее пространство

Установим на диск C: проект grom, полученный при выполнении предыдущей работы. Зададим аргументы командной строки:

"07 -lr -gr"

Ключ *lr* образован от слов *left recursion*, в начале ключа не единица.

Работа выполняется в модуле 07-lr.cpp. Откроем его.

В модуле несколько функций:

```
// устраняет непосредственную рекурсию правил для A
void eliminate_direct(SYMB Ai, grammar & g1) {
}

// формирует новое правило из двух
void compile_rule(RULE rule_k, RULE rule_j, RULE & rule_new) {
}

// заменяет правило Ak->Ajx на правила Ak->wx, w - правые части Aj
// k - номер правила Ak->Ajx
void replace_Aj(int k, grammar & g1) {
}

// преобразует неявную рекурсию в явную
// для нетерминалов от первого до Ak
// заменяет правила вида Ak->Ajx на Ak->wx, w - правые части Aj
void indirect_to_direct(SYMB Ak, grammar & g1) {
}

// устраняет левую рекурсию
void eliminate_leftr(grammar & g1) {
}

// точка входа в алгоритм
// устранение левой рекурсии
int algorithm_eliminate_leftr(grammar & g1, FILE * target) {
    // алгоритм
    eliminate_leftr(g1);
    // выводим грамматику в консоль
    g1.print(stdout);
    // выводим грамматику в файл
    g1.print(target);
    // результат
    return 1;
}
```

Откроем FAR. Перейдем в каталог C:\grom\Debug. Нажмем Shift+F4, введем название файла 07.sxg, Enter, и введем текст грамматики:

```
<A>
<A>=<B>[b]
<A>=[a]
<B>=<A>[a]
<B>=<B>[b]
<B>=[b]
•
```

Точка показывает конец файла.

Сохраним F2, закроем редактор Escape (или Escape и Enter).

7.3. Устранение левой рекурсии в целом

Поскольку все необходимые функции определены, мы можем описать алгоритм в целом.

Пусть last — индекс последнего нетерминала. Его можно получить при помощи метода grammar::nont_count.

Тогда сформируем цикл по переменной A типа SYMB от единицы до last включительно. В цикле выполняются шаги 1 и 2.

Сначала выполняется шаг 1, при условии, что символ является леворекурсивным по правилу (явно леворекурсивным). Символ –A является леворекурсивным по правилу, если метод grammar::is_symbol_leftr возвращает истину. Если это так, то вызываем частную функцию eliminate_direct с параметрами –A и g1. Эта функция устранит явную рекурсию.

Далее проверяем условие завершения. Если символ A равен значению last, то завершаем цикл (выходим из него, и алгоритм завершается).

В противном случае выполняем шаг 2, который реализует частная функция indirect_to_direct, с параметрами A+1 и g1.

Это весь основной алгоритм.

7.4. Преобразование неявной рекурсии в явную

Это преобразование для данной грамматики выполняется первым, оно выполняется в частной функции indirect_to_direct.

Параметрами функции являются символ A_k и грамматика g1. Следует учитывать, что A_k является положительным числом, а символ представляет отрицание этого числа.

Нам нужно найти правила грамматики, для которых левый нетерминал совпадает с A_k, а первый нетерминал правила, если такой есть, имеет индекс, меньший индекса символа A_k.

Для этого введем понятие символа A_j. Это символ, индекс которого меньше, чем индекс символа A_k. Формируем цикл по символу A_j, который изменяется от единицы до A_k исключительно. В цикле формируем цикл по всем правилам грамматики, используя переменную цикла k, а не r, как обычно. Пусть rule — правило номер k. Тогда, если левый нетерминал правила rule совпадает с $–A_k$, и первый символ правила совпадает с $–A_j$, то вызываем частную функцию replace_Aj, которая заменит символ A_j в правиле телами правил для A_j. Передаем функции номер правила k и g1.

7.5. Замена нетерминала его правилами

Теперь нужно описать функцию replace_Aj. Параметрами функции является номер правила k и грамматика. Мы входим в нее, когда $k = 3$, и правило 3 — это $B \rightarrow Aa$.

Теперь нужно найти правила для нетерминала A, и заменить этот нетерминал во всех правилах вида $B \rightarrow A\alpha$ на цепочки правил для A.

Поскольку мы не можем удалять правила грамматики, то поступаем так: первое найденное правило изменяем, а другие правила формируем и добавляем в грамматику.

Пусть rule_k — правило g1 номер k.

Пусть A_j — первый символ rule_k.

Пусть есть новое правило rule_new.

Просматриваем правила грамматики по переменной j и ищем первое правило rule_j, для которого левый нетерминал совпадает с A_j. Нашли — передаем rule_k, rule_j и rule_new функции compile_rule. Функция сформирует rule_new, и мы заменим им правило k в грамматике: g1[k] = rule_new.

Далее ищем следующее правило для A, это правило $A \to a$, и формируем новое правило $B \to aa$, которое добавляем в грамматику.

Формирование новых правил из правил k и j выполнит частная функция compile_rule, которая к правилу j добавляет символы правила k, начиная со второго (пропуская нетерминал). Для этого сначала присвоим правилу rule_new правило rule_j, а затем добавим к нему символы правила rule_k.

7.6. Устранение явной рекурсии

Устранение явной рекурсии выполняем в функции eliminate_direct, которой передается леворекуривный символ A_i и грамматика.

Сначала формируем новый маркированный символ A_i':

```
// добавляем в g1 новый нетерминал Ai'
SYMB A1 = g1.get_star_nont(Ai);
```

Затем просматриваем правил грамматики и ищем правила для A_i.

Пусть найдено леворекурсивное правило $A_i \to A_i\beta$. Тогда удалим в нем первый символ и заменим левый нетерминал на A_i': $A_i' \to \beta$. При этом нужно изменить именно правило грамматики, поэтому действия выполняем не с копией правила, а непосредственно с правилом g1[r]. После этого копируем в rule_new полученное правило g1[r], добавляем к нему новый нетерминал A_i', и добавляем правило в грамматику. Для определения леворекурсивности правила используем метод RULE::is_leftr.

Если же найдено нелеворекурсивное правило вида $A_i \to \alpha$, то копируем в rule_new правило g1[r], добавляем к нему новый нетерминал A_i', и добавляем правило в грамматику.

7.7. Контрольные вопросы и упражнения

1. Какой символ называется рекурсивным, леворекурсивным, праворекурсивным?
2. Поясните метод устранения явной рекурсии.
3. Поясните метод выявления неявной рекурсии.
4. Докажите корректность метода выявления неявной рекурсии.
5. Докажите корректность метода выявления явной рекурсии.

8. Работа PL-108. Левая факторизация

Цели:
- изучение преобразования левой факторизации
Задачи:
- поиск префикса;
- устранение префикса.
Опорные документы:
[1, «Левая факторизация»]

8.1. Изучение преобразования

Левая факторизация — простое преобразование, устраняющее одинаковые префиксы правил для некоторого нетерминала.

Пусть есть правила для нетерминала A:

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

Здесь цепочка α является префиксом двух правил.
Заменяем цепочки β нетерминалом, из которого выводятся все β:

$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1 \mid \beta_2$

В результате правила для A не должны иметь одинаковых префиксов.
Но это только в теории. Рассмотрим следующие правила:

$A \rightarrow abcd \mid abab \mid abce \mid abcf$

Вопрос заключается в том, что здесь является префиксом, *ab* или *abc*.
Префикс *ab* есть во всех цепочках. Если его устранить, получим:

$A \rightarrow abA'$
$A' \rightarrow cd \mid ab \mid ce \mid cf$

Теперь появились префиксы для A'. Устраняя их, получим:

$A \rightarrow abA'$
$A' \rightarrow cA'' \mid ab$
$A'' \rightarrow d \mid e \mid f$

Если же устранить префикс *abc*, получим:

$A \rightarrow abcA' \mid abab$
$A' \rightarrow d \mid e \mid f$

Теперь остались префиксы для A. Устраняя их, получим:

$A \rightarrow abA''$
$A' \rightarrow d \mid e \mid f$
$A'' \rightarrow cA' \mid ab$

Очевидно, что оба преобразования дают одинаковый результат.

Есть еще один вариант, более простой. Будем устранять префиксы длиной только один символ. Тогда, устраняя префикс *a*, получим:

A→ aA′
A′→ bcd | bab | bce | bcf

Теперь устраняем префикс *b*, и получаем:

A→ aA′
A′→ bA″
A″→ cd | ab | ce | cf

Теперь устраняем префикс *c*, и получаем:

A→ aA′
A′→ bA″
A″→ cA‴ | ab
A‴→ d | e | f

Этот метод реализовать проще всего, но он ведет к порождению множества дополнительных нетерминалов.

На практике левая факторизация необходима для грамматик, используемых в предиктивном анализе. Это простой и понятный метод разбора синтаксических грамматик, не допускающий одинаковых префиксов.

Например, мы хотим описать условный оператор:

statement = if E then statement
statement = if E then statement else statement

Очевидно, что первое правило является префиксом второго.

Устраняя префикс, получим:

statement = if E then statement statement′
statement′ = λ | else statement

Новые правила подходят для предиктивного анализа.

Остается решить, каким способом выполнять левую факторизацию — выделяя префикс из одного символа, префикс максимальной длины, или префикс минимальной длины.

8.2. Рабочее пространство

Установим на диск C: проект grom, полученный при выполнении предыдущей работы. Зададим аргументы командной строки:

"08 -lf -gr"

Ключ *lf* образован от слов *left factor*, в начале ключа не единица.

Работа выполняется в модуле 08-lf.cpp. Откроем его.

В модуле несколько функций:

```
// является pref префиксом правила rule ?
int is_prefix_for(RULE rule, RULE pref) {
    return 1;
}

// максимальный префикс для правила j
// возвращает длину префикса правила j
int max_prefix_for_rule_number(int j, grammar g1) {
    return 0;
}

// префикс максимальной длины для A
// возвращает длину префикса
int find_prefix_for(SYMB A, grammar g1, RULE & pref) {
    return 0;
}

// устраняет префикс pref нетерминала pref[0]
void eliminate_prefix(grammar & g1, RULE pref) {
}

// алгоритм левой факторизации
void left_factoring(grammar & g1) {
}

// точка входа в алгоритм
// левая факторизация
int algorithm_left_factoring(grammar & g1, FILE * target) {
    // алгоритм
    left_factoring(g1);
    // выводим грамматику в консоль
    g1.print(stdout);
    // выводим грамматику в файл
    g1.print(target);
    // результат
    return 1;
}
```

Для выполнения работы требуется грамматика.

Откроем FAR. Перейдем в каталог C:\grom\Debug. Нажмем Shift+F4, введем название файла 08.sxg, Enter, и введем текст грамматики:

```
<A>
<A>=[a][b][a][b]
<A>=[a][b][c][d]
<A>=[a][b][c][e]
<A>=[a][b][c][f]
.
```

Точка показывает конец файла.

Сохраним F2, закроем редактор Escape (или Escape и Enter).

8.3. Конструирование общего алгоритма

Будем искать префикс максимальной длины, этот вариант кажется более прагматичным. Для этого нужно решить несколько задач.

Первая задача — как представлять префикс.

Учитывая, что префикс правила — это правило, возможно, меньшей длины, будем представлять префикс как объект типа RULE. Тогда левый нетерминал этого объекта может представлять нетерминал, для которого выполняется левая факторизация.

Результат будет трудно сформировать в новой грамматике, поэтому будем изменять исходную.

Как решить задачу в целом?

В результате преобразований могут появляться новые префиксы, поэтому алгоритм в целом — это бесконечный цикл, в котором мы пытаемся обнаружить какой-нибудь префикс. В начале этого цикла установим признак обнаружения префикса found в нулевое значение. Тогда, если после цикла поиска префикса этот признак останется нулевым, бесконечный цикл можно будет завершить.

Для обнаружения префикса сформируем цикл по всем нетерминалам A от первого до последнего, включая те нетерминалы, которые будут сформированы преобразованием, и пытаемся найти префикс для текущего нетерминала A. Если префикс найден, то:

1) запоминаем, что префикс найден, в признаке found;

2) устраняем префикс:

3) выводим полученную грамматику в консоль для контроля.

Этот алгоритм нужно сформировать в функции left_factoring. Поиск префикса для нетерминала A выполнит частная функция find_prefix_for, а устранение префикса — частная функция eliminate_prefix. Эти частные функции имеют параметром префикс pref, который нужно объявить внутри цикла по нетерминалам грамматики, тогда он всегда будет пустой при передаче в первую функцию find_prefix_for. Эта функция по предположению возвращает максимальную длину префикса, или 0, если никакой префикс не был найден. Собственно максимальная длина префикса в нашем общем алгоритме значения не имеет, имеет значение нулевая длина префикса.

Программируем общий алгоритм, тестирование откладываем до того момента, когда будет реализована функция find_prefix_for.

8.4. Алгоритм поиска максимального префикса

Как найти максимальный префикс?

Любое правило грамматики может являться префиксом любого другого правила для этого же нетерминала. Пусть pref — некоторое правило, и pref[0] — нетерминал A. Пусть prefix_count — счетчик. Просматриваем правила грамматики и ищем правила для A. Если правило для A найдено и pref является префиксом этого правила, подсчитываем количество найденных префиксов prefix_count. Если в результате поисков счетчик превысит значение 1 (то есть имеется как минимум два правила с одинаковым префиксом), то найден некоторый префикс.

При этом нужно учитывать и само правило, которое было взято в качестве префикса.

Пусть поиски префикса для правила целиком неуспешны и счетчик prefix_count остался равным нулю. Тогда уменьшаем длину префикса pref на один последний символ и повторяем поиск по всем правилам. Если длина префикса станет равной нулю в результате его уменьшения, поиск завершаем, префикс не найден.

Точно так же нужно поступить со всеми другими правилами грамматики. Поиск по каждому правилу даст или не даст какой-то префикс. Если префикс вообще был найден, тогда выбираем из найденных префиксов наибольший по длине и возвращаем его как результат поисков.

Этот алгоритм в целом сложный, поэтому поделим его на более простые части. Пусть частная функция max_prefix_for_rule_number найдет максимальный префикс для правила с заданным номером. Тогда несложно организовать алгоритм функции find_prefix_for для поиска максимального префикса.

Пусть max_len — переменная, которая хранит длину максимального префикса, изначально равная нулю. Пусть max_rule — переменная, которая хранит номер правила, для которого обнаружен префикс максимальной длины max_len; изначально эта переменная также равна нулю. Пусть len — длина текущего найденного префикса, а переменная r — номер правила, для которого найден какой-то префикс.

Тогда поиск максимального префикса, — это цикл по правилам грамматики, в котором текущее правило rule — это правило с номером r.

В цикле ищем максимальный префикс для правила r при помощи частной функции max_prefix_for_rule_number. Функция возвращает длину найденного префикса, которую мы принимаем в переменную len. Тогда, если значение переменной max_len меньше значения переменной len, то принимаем значение переменной max_len равной значению переменной len, а значение max_rule — равной значению r.

Когда цикл по правилам завершится, то оцениваем значение max_len. Если оно больше нуля, префикс найден. В этом случае принимаем pref[0] равным левому нетерминалу правила номер max_rule, и копируем из правила номер max_rule первые max_len символов в pref. Таким образом найженный префикс будет сформирован, а функция find_prefix_for вернет значение max_len для функции left_factoring.

Эти действия можно программировать в функции find_prefix_for.

8.5. Поиск префикса для правила

Остается определить алгоритмы функций max_prefix_for_rule_number и is_prefix_for. Проще определить вторую функцию. Она определяет, является префикс pref префиксом правила rule.

Это просто. Пусть symbol_count — длина префикса pref. Если эта длина больше длины правила rule, то возвращаем ноль. Если же длина префикса равна или меньше длины правила, то сравниваем символы правила с символами префикса, максимальное количество сравнений равно длине префикса. Если какой-то символ не совпадает, возвращаем ноль. Если цикл сравнений завершился, значит все символы правила совпали с символами префикса, возвращаем единицу.

Теперь можно сформировать алгоритм поиска префикса для правила с номером r в функции max_prefix_for_rule_number. В эту функцию r передается как параметр *j*. Пусть pref — это правило грамматики номер *j*. Пусть символ A — это левый нетерминал pref.

Процесс поиска — это бесконечный цикл, смысл которого — искать последовательно префиксы для правила целиком, потом для правила минус один символ, потом для правила минус два символа и т.д.

В начале цикла принимаем счетчик prefix_count равным нулю. В конце цикла проверяем условия завершения. Первое условие завершения — счетчик количества префиксов prefix_count больше единицы. Второе условие завершения проверяется после уменьшения длины префикса pref на один символ. Сначала уменьшаем длину префикса pref, затем проверяем получившуюся длину префикса pref.count(). Если она стала равной нулю, завершаем бесконечный цикл.

По завершении бесконечного цикла возвращаем длину префикса, которая передается функции find_prefix_for, и попадает в переменную len.

В середине бесконечного цикла мы просматриваем правила грамматики и ищем правила для нетерминала A. Если такое правило найдено, и текущий префикс pref является префиксом этого правила, о чем нам скажет функция is_prefix_for, то подсчитываем префикс, увеличивая prefix_count.

Программируем эти функции и приступаем к отладке.

Первое исследуемое правило — *A→abab*, [–1, 1, 2, 1, 2]. В квадратных скобках показаны значения первого исследуемого префикса. Поиск префикса в первой итерации бесконечного цикла max_prefix_for_rule_number должен дать значения счетчика prefix_count, равное единице (и это само правило). Во второй и третьей итерации prefix_count также равен единице. В четвертой итерации prefix_count должно быть равно четырем, при этом префикс сократился до *A→ab*, [–1, 1, 2], префикс *ab* есть во всех правилах.

При этом бесконечный цикл max_prefix_for_rule_number завершается и мы возвращаемся в find_prefix_for, переменная len принимает значение 2, max_len равно 2, max_rule равно 1.

Второе исследуемое правило — *A→abcd*, [–1, 1, 2, 3, 4]. Это префикс уменьшается до *A→abc*, [–1, 1, 2, 3], и в len приходит значение 3, соответственно, max_len увеличивается до трех, max_rule равно двум. Следующие два правила ситуацию не меняют. В результате в функцию left_factoring должен прийти префикс *A→abc*, [–1, 1, 2, 3].

8.6. Алгоритм устранения префикса

Если вычисления выполняются так, как описано, есть надежда, что все правильно запрограммировано, и можно переходить к устранению найденного префикса в функции eliminate_prefix.

Пусть обнаружен префикс $A \rightarrow abc$, [–1, 1, 2, 3]. Пусть symbol_count — длина префикса (равная трем). Пусть символ A — это pref[0], пусть символ A1 — это новый маркированный символ грамматики, производный от A.

Тогда просматриваем правила грамматики, и если обнаружено правило для A, и префикс pref является префиксом правила, то заменяем левый нетерминал этого правила (правила g1[r]) на A1, и удаляем из правила первые symbol_count символов. Первым будет обнаружено правило номер 2 $A \rightarrow abcd$, [–1, 1, 2, 3, 4]. Удаление из него первых трех символов приведет к изменению правила на правило $A^* \rightarrow d$, [–2, 4]. Должны измениться также и правила 3 и 4.

По завершении цикла нужно сформировать новое правило $A \rightarrow abcA^*$, [–1, 1, 2, 3, –2]. Для этого объявляем переменную rule, задаем левый нетерминал равным A, добавляем в правило все символы префикса и новый нетерминал A1.

Для справки: новый маркированный символ грамматики находит метод grammar::get_star_nont, просто новый символ грамматики находит метод grammar::get_new_nont.

8.7. Контрольные вопросы и упражнения

1. Что называется префиксом?
2. В чем заключается левая факторизация?
3. Докажите, что левая факторизация не изменяет язык, порождаемый грамматикой.

9. Работа PL-109. Приведение к нормальной форме Хомского

Цели:
- изучение грамматик в нормальной форме Хомского (CNF).

Задачи:
- выявление правил, соответствующих CHF;
- формирование правил для вывода терминалов;
- замена терминалов нетерминалами;
- преобразование правил, не соответствующих CHF.

Опорные документы:
[1, «Грамматики в нормальной форме Хомского»]
[3, с.176]
[4, с.280]
[5]

9.1. Грамматики в нормальной форме Хомского

Грамматика называется грамматикой в нормальной форме Хомского, если правила грамматики имеют одну из двух форм:

1) $A \to a$, $A \in \mathrm{N}$, $a \in \Sigma$,
2) $A \to BC$, $A, B, C \in \mathrm{N}$.

В грамматике может быть также правило $S \to \lambda$, S — целевой символ, в случае, если $\lambda \in L(\mathrm{G})$, при условии, что символ S не является правой частью никакого правила грамматики.

Грамматики в нормальной форме Хомского отличаются тем, что все выводы в них представляют бинарные деревья. Эти грамматики используются также для доказательства принадлежности языка, порождаемого грамматикой, к контекстно-свободным языкам.

9.2. Приведение к нормальной форме Хомского

Алгоритм приведения грамматики к нормальной форме Хомского применяется к грамматикам в канонической форме. Исходная грамматика не должна содержать пустых и цепных правил, бесплодных и недостижимых символов. Это достигается применением соответствующих алгоритмов в определенном порядке.

Пусть есть грамматика в каноническом виде G = (Σ, N, P, S).

Такой грамматикой является, например, грамматика G2:

$S \to S+S \mid S*S \mid a \mid (S)$

Будем использовать эту грамматику для приведения к CNF.

В грамматике в каноническом виде могут встречаться правила в одной из следующих форм:

1) $A \to a$, $A \in N$, $a \in \Sigma$,
2) $A \to BC$, $A,B,C \in N$.
3) $A \to \alpha$, $A \in N$, $\alpha \in (N \cup \Sigma)^+$, $|\alpha| \geq 2$.

Правила первых двух видов переносятся в целевую грамматику как есть, так как они удовлетворяют требованиям CNF.

Остаются только правила вида $A \to \alpha$, в которых цепочка α состоит из произвольных символов, и не удовлетворяет требованиям CNF.

При этом возможны два варианта.

Далее символом G1 будем обозначать исходную грамматику, а символом G2 будем обозначать целевую грамматику.

1. Если длина правила равна двум, то терминалы правила заменяются нетерминалами с добавлением в грамматику правил для вывода терминалов, после чего само правило также добавляется в грамматику.

Например, если есть правило $A \to aB$, то формируем два новых правила в грамматике G2: $A \to CB$, $C \to a$.

2. Если длина правила больше двух, то формируем два новых правила, одно в грамматике G2, другое временное.

Для примера, пусть есть правило $S \to S+S$.

Тогда формируем в грамматике G2 правило $S \to SA$, где A — новый символ грамматики G1, и временное правило $A \to +S$, которое считаем правилом вида $A \to \alpha$. Обрабатывая это временное правило по указанным выше вариантам, получаем новые правила грамматики G2: $A \to BS$, $B \to +$.

Приведенным правилам обработки правил вида $A \to \alpha$ соответствует следующее формальное описание:

Пусть есть правило $A \to \alpha$. Заменим каждое вхождение нетерминала a в цепочку α правилом $B \to a$. Тогда правило примет вид $A \to B_1 B_2 \ldots B_n$.

Базис. Если $|\alpha| = 2$, правило $A \to B_1 B_2$ является допустимым.

Индукция. Если $|\alpha| > 2$, заменим правило $A \to B_1 B_2 \ldots B_n$ на два правила $A \to B_1 C_1$, $C_1 \to B_2 \ldots B_n$. Тогда первое правило является допустимым, а второе правило является правилом вида $A \to B_1 B_2 \ldots B_n$, и к нему можно применить тот же алгоритм.

Для грамматики $S \to S+S \mid S*S \mid a \mid (S)$ данное преобразование приводит к грамматике

$S \to SB \mid SD \mid a \mid EG$
$B \to AS$
$D \to CS$
$E \to ($
$G \to SF$
$A \to +$
$C \to *$
$F \to)$

9.3. Рабочее пространство

Установим на диск C: проект grom, полученный при выполнении предыдущей работы. Зададим аргументы командной строки:

"002 -cnf -gr"

Работа выполняется в модуле 09-cnf.cpp. Откроем его:

```
// правило для вывода терминала
void term_rule(SYMB A, SYMB a, grammar g1, grammar & g2) {
}

// приводит правило к нормальной форме Хомского
void rule_to_cnf(RULE rule_old, grammar & g1, grammar & g2) {
}

// заменяет терминалы нетерминалами
void terms_to_nonts(RULE & rule, grammar & g1, grammar & g2) {
}

// приводит грамматику к нормальной форме Хомского
int chomsky_normal_form(grammar & g1, grammar & g2) {
    return 1;
}

// точка входа в алгоритм
// приведение к нормальной форме Хомского
int algorithm_chomsky_normal_form(grammar & g1, FILE * target) {
    . . .
    grammar g2;
    // алгоритм
    int result = chomsky_normal_form(g1, g2);
    // выводим грамматику в консоль
    g2.print(stdout);
    // выводим грамматику в файл
    g2.print(target);
    // выходная грамматика
    g1 = g2;
    // результат
    return result;
}
```

Для выполнения работы требуется грамматика G2.

Откроем FAR. Перейдем в каталог C:\grom\Debug. Нажмем Shift+F4, введем название файла 002.sxg, Enter, и введем текст грамматики:

```
<S>
<S>=<S>[+]<S>
<S>=<S>[*]<S>
<S>=[a]
<S>=[(]<S>[)]
.
```

Точка показывает конец файла.

Сохраним F2, закроем редактор Escape (или Escape и Enter).

9.4. Основной алгоритм

Основной алгоритм, функция chomsky_normal_form, простой.

Сначала устанавливаем целевой символ целевой грамматики равным целевому символу исходной грамматики.

Далее формируем цикл по правилам грамматики и анализируем каждое правило по длине.

Если длина правила равна нулю, то правило может быть только $S \to \lambda$. Если это так, то добавляем правило в G2. Иначе выводим в консоль сообщение «Unexpected empty rule» и завершаем функцию, возвращая 0.

Если длина правила равна единице, то если первым символом правила является терминал, добавляем правило в G2. Иначе выводим в консоль сообщение «Unexpected unit rule» и завершаем функцию, возвращая 0.

Если длина правила больше единицы, то:

- заменяем терминалы нетерминалами, вызывая terms_to_nonts;
- приводим правило к CNF, вызывая rule_to_cnf.

Дополнительно для контроля можно в этом месте выводить грамматику G2 в консоль.

9.5. Замена терминалов нетерминалами

Функция terms_to_nonts тоже простая.

Просматриваем каждый символ X правила rule.

Если символ является терминалом, то:

- формируем новый символ A в грамматике g1;
- заменяем символ X правила rule символом A;
- формируем правило для терминала, вызывая term_rule, и передавая параметры A, X, g1, g2.

9.6. Формирование правила для терминала

Функция term_rule также простая.

Пусть есть правило rule.

Зарегистрируем в G2 символы A и X.

Левый нетерминал правила равен A, а единственный символ правила равен X. Добавляем новое правило в целевую грамматику G2.

9.7. Приведение правила к CNF

Остается привести правило вида $A \to B_1 B_2 \ldots B_n$ к CNF.

Функция rule_to_cnf.

Преобразуемое правило названо rule_old.

Пусть есть правило rule_new. В нем нужно сформировать новое правило и добавить его в целевую грамматику G2.

При этом возможны два варианта.

Если длина правила rule_old равна двум, то rule_new = rule_old.

Если длина правила rule_old больше двух, его нужно разбить на новое правило rule_new и временное правило rule_old.

Независимо от длины правила правило rule_new начинается с двух символов: $A \rightarrow B_1$. Поэтому можно зарегистрировать в G2 левый нетерминал и первый символ правила rule_old и записать зарегистрированные символы в правило rule_new.

Если длина правила равна двум, это соответствует завершению рекурсии функции rule_to_cnf. При этом нужно зарегистрировать второй символ правила rule_old, добавить зарегистрированный символ к rule_new, и добавить правило rule_new в G2.

Если длина правила rule_old больше двух, то:

- формируем новый символ A в грамматике G1;
- заменяем левый нетерминал правила rule_old символом A;
- удаляем первый символ правила rule_old;
- добавляем символ A к правилу rule_new;
- добавляем правило rule_new в G2;
- рекурсивно вызываем rule_to_cnf, передавая правило rule_old.

9.8. Контрольные вопросы и упражнения

1. Опишите грамматику в нормальной форме Хомского.
2. Чем характерна грамматика в нормальной форме Хомского?
3. Для чего используется грамматика в нормальной форме Хомского?
4. Опишите алгоритм приведения к нормальной форме Хомского.
5. Докажите, что алгоритм приведения грамматики к нормальной форме Хомского не изменяет язык, порождаемый грамматикой.

Литература

1. Вл. Пономарев. Конспективное изложение теории языков и методов трансляции. Учебно-методическое пособие. В 4-х книгах. Книга 1. Формальные языки и грамматики. Озерск: ОТИ НИЯУ МИФИ, 2018.

2. Вл. Пономарев. Конспективное изложение теории языков и методов трансляции. Учебно-методическое пособие. В 4-х книгах. Книга 2. Лексический анализ. Озерск: ОТИ НИЯУ МИФИ, 2018.

3. А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции. Том 1. Синтаксический анализ. Пер. с англ. М.: Мир, 1978.

4. Хопкрофт, Джон, Э., Мотвани, Раджив, Ульман, Джеффри, Д. Введение в теорию автоматов, языков и вычислений, 2-е изд. : Пер. с англ. — М.: Издательский дом «Вильямс», 2002.

5. N. Chomsky. On certain formal properties of grammars. Information and Control 2:2 (1959), pp. 137-167.