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

Кафедра прикладной математики

Вл. Пономарев

ПРАКТИКУМ

по теории языков программирования и методам трансляции

Учебно-методическое пособие

Часть 1. Преобразование грамматик и конечных автоматов

2017 г.

УДК 681.3.06
П 56

Вл. Пономарев. Практикум по теории языков программирования и методам трансляции. Учебно-методическое пособие. Часть 1. Преобразование грамматик и конечных автоматов. Озерск: ОТИ НИЯУ МИФИ, 2017. — 106 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. Зав. кафедрой прикладной математики ОТИ НИЯУ МИФИ,
к.ф.-м.н. Акопян Р.Р.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Общие цели занятий

В ходе практических работ предлагается изучить и реализовать следующие алгоритмы преобразования грамматик и конечных автоматов:

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;
10⁻) преобразование регулярной грамматики в автоматную;
11) программирование алгоритмов с конечными автоматами;
12) удаление недостижимых состояний конечного автомата;
13) преобразование автоматной грамматики в конечный автомат;
14) преобразование конечного автомата в автоматную грамматику;
15*) построение детерминированного конечного автомата;
16*⁻) минимизация конечного автомата.

Описание алгоритмов приводится в учебно-методическом пособии автора «Конспективное изложение теории языков программирования и методов трансляции». Наиболее актуальная версия этого документа доступна в Интернет по адресу http://revol.ponocom.ru.

На выполнение каждой работы предположительно отводится 2 академических часа, однако некоторые, наиболее сложные алгоритмы могут потребовать большего времени, например, устранение левой рекурсии.

Поэтому, в зависимости от количества учебных часов, выделенных на проведение практических работ, сложности работ и навыков обучающегося, преподаватель может выбирать индивидуальные траектории работ.

Темы работ, которые можно без ущерба для процесса обучения изъять из приведенной общей траектории, отмечены знаком «минус». Работы повышенной сложности отмечены звездочками.

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 из себя консольное приложение grom, предоставляемое преподавателем.

Предполагается, что перед выполнением работы обучающийся самостоятельно изучает описание работы и собственно изучаемый алгоритм. Это позволит сэкономить время, отведенное на выполнение работы.

Каждая выполненная работа должна быть защищена обучающимся.

1. Программирование алгоритмов с грамматиками

Цели:

- изучение классов библиотеки grammar.lib;
- изучение вспомогательных функций 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 в виде архива. Архив распаковывается на диск C:, каталог проекта C:\grom.

Следует убедиться в том, что проект компилируется и запускается.

Последующие работы выполняются в этом же рабочем проекте.

1.1. Форматы SYNAX

Прежде всего нужно знать, как описывается грамматика в виде текста.

Существуют общеизвестные форматы записи грамматик, такие, как форма Бэкуса-Наура (БНФ) или расширенная форма Бэкуса-Наура (РБНФ).

При выполнении работ используется похожий способ, называемый здесь «формат SYNAX» по названию программы автора, в которой этот формат впервые применен.

Есть две разновидности этого формата. Строгий формат сложился исторически в рамках разработки программы SYNAX, которая поддерживает только этот формат. При разработке настоящих практических заданий автором выработан менее строгий формат, называемый здесь «свободным».

Как известно из теоретического курса, грамматика представляет собой математическую систему из четырех компонент:

- множество терминальных символов (терминалов, токенов) V_T;
- множество нетерминальных символов (нетерминалов) V_N;
- множество правил P;
- целевой символ S.

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 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 (context-free grammar — CFG) и регулярные.

Такие грамматики отличаются тем, что левая часть любого правила представляет собой один-единственный нетерминальный символ.

1.1.1. Строгий формат SYNAX

Пример записи грамматики в строгом формате SYNAX:

```
#0 <S>
#1 <S>=<A><B><C>
#2 <A>=[a]
#3 <A>=<B>
#4 <B>=[b]
#5 <B>=.
#6 <C>=[c]
#7 <C>=.
```

В первой строке записи находится единственный нетерминальный символ, задающий целевой символ грамматики. В последующих строчках располагаются правила, по одному правилу в строке.

Для обозначения нетерминального символа используются угловые скобки, например <S>. Между скобками размещается идентификатор.

Для обозначения терминального символа используются квадратные скобки, например [+]. Между скобками размещается идентификатор, но в отличие от идентификатора нетерминального символа, это, как правило, собственно нетерминал грамматики.

Пустая цепочка в формате SYNAX обозначается символом «точка».

В БНФ между левой и правой частями правила записывается символ из трех знаков "::=". В формате SYNAX используется знак равенства "=".

Формат SYNAX предусматривает возможность нумерации правил для облегчения ориентации и возможности ссылаться на них в сообщениях.

1.1.2. Свободный формат SYNAX

Свободный формат используется в данных практических работах. Его отличия от строгого формата следующие:

- указание целевого символа может находиться в произвольной строке описания, и может встречаться несколько раз, хотя в этом и нет смысла.

- знак «точка» используется не как признак пустой цепочки, а как символ начала комментария.

В свободном формате грамматика может иметь, например, такой вид:

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

При этом нумерация правил грамматики становится затруднительной и она в данном формате не применяется, хотя и может присутствовать.

Заметим, что во входном файле можно записывать какие угодно комментарии, но анализатор входного файла их отбрасывает и в результирующей грамматике они отсутствуют, тем более что грамматика, как правило, видоизменяется и комментарии теряют смысл.

Заметим также, что в свободном формате допускаются пробелы в любом месте, — анализатор входного файла сначала удаляет все пробелы.

1.2. Входные файлы

Описание грамматики (и конечного автомата) представляет из себя текстовый файл в кодировке Windows-1251, имеющий расширение:

.sxg для грамматики,

.sxa для конечного автомата,

.sxr для регулярного выражения.

Поскольку используется обычный текстовый файл, и не предусмотрено никаких указаний в самом файле на вид описания, расширение является критичным, — только оно указывает анализатору, что описывается в данном файле. Это правило нужно обязательно соблюдать.

Сейчас нам нужно создать несколько текстовых файлов для описания грамматик g23a, g26a, g28a, g32a (номера грамматик приведены в соответствии с учебным пособием по теории). Используйте для этого файловый менеджер FAR. Каждую отдельную грамматику запишите в отдельный текстовый файл с названием грамматики и расширением .sxg, например, для первой из указанных грамматик название файла g23a.sxg.

Порядок работы в FAR следующий.

1. Откройте FAR Manager и перейдите в каталог C:\grom\Debug.

2. Нажмите сочетание клавиш «Shift+F4» для создания и начала работы с текстовым файлом.

3. Введите название первого файла "g23a.sxg" и нажмите Enter.

4. Нажмите сочетание клавиш «Shift+F8» и выберите кодировку 1251.

5. Наберите текст грамматики:

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

6. Нажмите Escape и Enter. Файл готов. Чтобы редактировать его повторно, установите указатель на файл и нажмите F4.

Повторите указанные действия для следующих грамматик.

Грамматика g26a.sxg:

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

Грамматика g28a.sxg:

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

Грамматика g32a.sxg:

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

1.3. Рабочее пространство

Откроем проект консольного приложения grom. Откроем модуль umain.cpp, и найдем функцию get_key(). Эта функция определяет действие, выполняемое приложением. Поскольку приложение консольное, действие задается ключами командной строки. Командная строка должна иметь следующий вид:

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

Здесь grom — название приложения, grammar — спецификация файла грамматики (конечного автомата), key — один из возможных ключей.

Ключи видны в функции get_key, например:

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

В этой строке кода проверяется ключ "**-gr**", определяющий действие «группирование правил грамматики».

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.

Перейдем в начало модуля umain.cpp. Здесь видны описания функций, являющихся точками входа в реализацию алгоритмов.

```
// 1. ознакомительные действия
int algorithm_general(grammar & G1, FILE * target);
// 2. группирование правил
int algorithm_group_rules(grammar & G1, FILE * target);
// 3-4. удаление бесплодных и недостижимых символов
int algorithm_remove_useless(grammar & G1, FILE * target);
// 5. удаление пустых правил
int algorithm_remove_empty(grammar & G1, FILE * target);
// 6. удаление цепных правил
int algorithm_remove_cycles(grammar & G1, FILE * target);
// 7. устранение левой рекурсии
int algorithm_eliminate_leftr(grammar & G1, FILE * target);
// 8. левая факторизация
int algorithm_left_factoring(grammar & G1, FILE * target);
// 9. приведение к нормальной форме Хомского
int algorithm_chomsky_normal_form(grammar & G1, FILE * target);
// 10. преобразование регулярной грамматики в автоматную
int algorithm_regular_to_auto(grammar & G1, FILE * target);
// 11. ознакомительные действия с конечным автоматом
int algorithm_fa_general(UFA & FA, FILE * target);
// 12. удаление недостижимых состояний конечного автомата
int algorithm_fa_remove_unreachable(UFA & FA, FILE * target);
// 13. преобразование автоматной грамматики в конечный автомат
int algorithm_grau_to_fa(grammar G1, UFA & FA, FILE * target);
// 14. преобразование конечного автомата в автоматную грамматику
int algorithm_fa_to_grau(UFA FA, grammar & G1, FILE * target);
// 15. преобразование НКА в ДКА
int algorithm_dfa(UFA & FA, FILE * target);
// 16. минимизация ДКА
int algorithm_min_dfa(UFA & FA, FILE * target);
```

Откроем модуль gugen.cpp. В этом модуле выполняется первая практическая работа. В модуле gugen.cpp две функции.

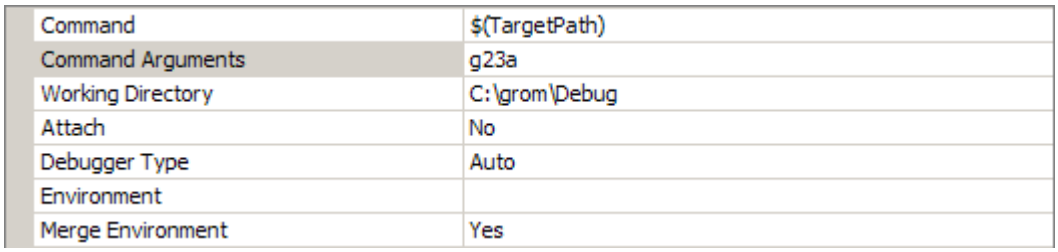
```
// алгоритмы общих действий с грамматикой
int grammar_general(grammar & G1) {
    return 1;
}

// точка входа в алгоритм
// общие действия с грамматикой
int algorithm_general(grammar & G1, FILE * target) {
    // результирующая грамматика
    grammar G2;
    // алгоритм
    int result = grammar_general(G1, G2);
    // анализ грамматики для последующих алгоритмов
    G2.analyse();
    // выводим грамматику на терминал
    G2.print_out(stdout);
    // выводим свойства грамматики на терминал
    G2.print_props(stdout);
    // выводим грамматику в файл
    G2.print_out(target);
    // выходная грамматика
    G1 = G2;
    // результат
    return result;
}
```

Запомним, что функция, начинающаяся с algorithm_, является точкой входа в реализацию алгоритма, однако сама реализация должна выполняться в функции, расположенной в модуле выше, — grammar_general().

Точка входа требуется для управления объектами, например, для объявления вспомогательных грамматик (конечных автоматов), их анализа, вывода на терминал или в файл.

Функция точки входа обязана сообщать результат. Возвращаемое нулевое значение из этой функции 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. На терминал при этом выводится сообщение "Source file path required.". Откроем свойства проекта, выберем Configuration Properties, выберем Debugging (рисунок 1).



Command	$(TargetPath)
Command Arguments	g23a
Working Directory	C:\grom\Debug
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes

Рисунок 1

Зададим рабочий каталог "C:\grom\Debug", зададим название файла грамматики g23a.

Запустим приложение. Управление должно прийти в точку остановки.

Если управление не приходит в точку установки, скорее всего, неправильно задан входной файл, либо его не существует в рабочем каталоге.

Запустим приложение Ctrl+F5, чтобы увидеть вывод приложения.

Поскольку входной объект задан, то срабатывает анализатор входного файла, который преобразует входной текст в двоичное представление заданного типа. По мере разбора распознанный текст выводится на терминал, объект анализируется и его свойства также выводятся на терминал:

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

-- grammar_general
.Grammar is empty.
```

Каждое действие предваряется комментарием "`-- действие`".

1.4. Внутреннее представление грамматики

Для данной грамматики при анализе входного текстового файла будет сформировано внутреннее представление. Это представление нужно внимательно изучить, так как именно оно определяет все, что можно сделать.

Входная грамматика сейчас следующая

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

Во время анализа терминальные и нетерминальные символы обнаруживаются по наличию соответствующих скобок. Идентификатор символа запоминается в массиве symbol, а *индекс* в массиве *является двоичным представлением* данного символа и используется *вместо символа*. Процесс называется *регистрацией*, и это самое важное, что нужно понимать при работе с двоичным представлением грамматики.

Терминалы имеют индексы от 0 до MAX_TOK – 1.

Под индексом 0 в массиве symbol записан терминал TOK_EOT.

Нетерминалы имеют индекс от MAX_TOK до MAX_TOK + MAX_SYM – 1.

Константа MAX_TOK имеет значение 61. Этому числовому коду соответствует символ знака равенства "=". Поскольку знак равенства является частью правила, часто происходит путаница между внешним и внутренним представлением.

Для данной грамматики сначала будет зарегистрирован целевой символ, идентификатор которого "S" состоит из одной буквы, имеющей код 83, а внутреннее представление целевого символа будет равно 61, то есть "=".

Нужно всегда смотреть на код, а не на букву, которой он представлен.

Далее читаются правила, одно за другим.

Каждый символ правила регистрируется.

Если идентификатора символа в массиве symbol нет, формируется новая запись, и ее индекс — индекс данного символа. Если идентификатор существует, возвращается индекс существующего символа.

Первое правило имеет вид <S>=<S>[+]<T>.

При регистрации левого нетерминала возвращается 61, так как идентификатор нетерминала "S" уже зарегистрирован.

Знак «равно» пропускается.

Первый символ правой части правила — нетерминал "S", поэтому возвращается также код 61.

Следующий символ правила — терминал "+". Он регистрируется, при этом идентификатор "+" записывается в первый элемент массива symbol, и возвращается код этого терминала, равный 1.

Последний символ правила "T" регистрируется и получает код 62.

Правило записывается как массив символов, при этом нулевой элемент массива соответствует левому нетерминалу, первый элемент — первому символу правой части правила, второй элемент — второму символу правой части правила и т.д. Таким образом, для первого правила получим последовательность кодов 61-61-1-62. Если эту последовательность рассматривать как символы, увидим "==┌> ", где код 1 отображается как "┌".

Чтобы убедиться в этом, установите точку остановки на операторе return функции grammar_general в модуле gugen.cpp, и введите для просмотра в окно Watch переменную G1. Раскройте эту переменную и посмотрите массив symbol, и далее массив rule. В первом массиве найдите символы с индексами 1, 61 и 62, во втором массиве раскройте правило с индексом 1. Заметим, что правило с индексом 0 является служебным, имеет вид "~=".

Второе правило <S>=<T> записывается как последовательность кодов 61-62 и имеет строковый вид "=>".

Третье правило <T>=<T>[*]<P> регистрирует два новых символа: терминал "*" и нетерминал "P", которые получают коды 2 и 63 соответственно, а правило имеет строковый вид ">>┌?".

Другие правила анализируются аналогичным образом.

1.5. Классы библиотеки grammar.lib

Для работы с грамматиками используются классы:

- grammar (грамматика);
- RULE (правило);
- GSET (множество).

Запомним, что элементы классов нумеруются от *единицы*. Поэтому все циклы, перебирающие элементы, должны иметь вид:

```
for (char element = 1; element <= count; element++);
```

1.5.1. Грамматики и правила

В качестве примера посмотрим, как получить копию входной грамматики G1 в новую грамматику G2, не выполняя при этом преобразований.

Чтобы получить копии правил грамматики G1 в грамматике G2, нужно создать новое правило для грамматики G2, установить его символы равным символам правила грамматики G1, предварительно зарегистрировав их.

Правило состоит из левой и правой частей. Символ левой части читается и устанавливается методом rule::symbol(). Символ правой части читается методом rule::symbol(index), где параметром является индекс символа, а устанавливается при помощи метода rule::append(symbol), который приписывает символ в конец правила. Обработка левого символа правила может выполняться до или после обработки символов правой части правила.

Последовательность действий следующая:

1) зарегистрировать целевой символ;
2) для каждого правила:
- зарегистрировать левый символ;
- записать левый символ в новое правило;
- для каждого символа правой части правила:
    - зарегистрировать символ;
    - добавить символ к правилу;
3) добавить новое правило в грамматику G2.

Получим следующий общий вид функции grammar_general():

```
// алгоритмы общих действий с грамматикой
int grammar_general(grammar & G1, grammar & G2) {
    // символ G1
    char X1 = 0;
    // символ G2
    char X2 = 0;
    // регистрируем целевой символ
    //
    for (int r = 1; r <= 1;) {//rule_count; r++) {
        // регистрируем левый символ
        //
        // символы правой части правила
        for (int s = 1; s <= 1) {//rule_old.count(); s++) {
            // символ правила
        }
        // добавляем правило в G2
        //
    }
    return 1;
}
```

Записываем этот код и начинаем выполнять заданные действия.

Первое, что нужно сделать, — это зарегистрировать целевой символ.

Регистрацию выполняет метод grammar::symbol_reg():

```
    // регистрируем целевой символ
    X2 = G2.symbol_reg(G1.symbol_id(G1.start()), S_SYM);
    // проверяем успешность
    if (X2 == 0) return 0;
```

Первым параметром метода является идентификатор символа, который можно получить из массива symbol методом symbol_id(). Параметром метода symbol_id() является индекс символа, который возвращает метод start().

Метод symbol_reg() возвращает либо индекс символа, который будет больше единицы, либо ноль в случае ошибки. После регистрации полученный индекс проверяется на правильность.

Далее в первом цикле разбираем правило за правилом.

Обозначим rule_old — это правило G1, а rule_new — правило G2.

Сначала объявляем эти переменные и получаем правило грамматики G1 в переменную rule_old:

```
    for (int r = 1; r <= rule_count; r++) {
        // текущее правило G2
        RULE rule_new;
        // текущее правило G1
        RULE rule_old = G1.rule(r);
        // регистрируем левый символ
    }
```

Далее регистрируем левый символ и записываем его в новое правило:

```
        // регистрируем левый символ
        X2 = G2.symbol_reg(G1.symbol_id(rule_old.symbol()), S_SYM);
        // проверяем успешность
        if (X2 < 0) return 0;
        // запишем левый символ
        rule_new.symbol() = X2;
```

Во втором, внутреннем цикле символ за символом регистрируем и добавляем в rule_new:

```
        // символы правой части правила
        for (int s = 1; s <= rule_old.count(); s++) {
            // символ правила
            X1 = rule_old.symbol(s);
            // регистрируем символ
            X2 = G2.symbol_reg(G1.symbol_id(X1), G1.symbol_typ(X1));
            // проверяем успешность
            if (X2 < 0) return 0;
            // приписываем символ
            if (rule_new.append(X2) < 0) return 0;
        }
```

Последнее, — добавляем новое правило и проверяем результат:

```
// алгоритмы общих действий с грамматикой
int grammar_general(grammar & G1, grammar & G2) {
    . . .
    for (int r = 1; r <= rule_count; r++) {
        . . .
        for (int s = 1; s <= rule_old.count(); s++) {
            . . .
        }
        // добавляем правило в G2
        if (G2.rule_add(rule_new) < 0) return 0;
    }
    return 1;
}
```

Таким образом, мы изучили основные принципы работы с классами грамматики и правила. Эти принципы включают в себя:

1. Циклы строятся от единицы до счетчика *включительно*.
2. Символы требуют регистрации.
3. Результат регистрации должен проверяться.
4. Результат добавления объекта также должен проверяться.
5. Методы возвращают 0 в случае ошибки.

1.5.2. Вспомогательные функции

Есть несколько вспомогательных функций, помогающих писать код.

grammar_set_start(grammar & G2, char A, grammar G1) — устанавливает целевой символ грамматики G2 равным символу A грамматики G1.

grammar_reg_symbol(grammar & G2, char X, grammar G1) — регистрирует в грамматике G2 символ X грамматики G1.

grammar_reg_tok(grammar & G2, char * id) — регистрирует терминал с идентификатором id в грамматике G2.

grammar_reg_sym(grammar & G2, char * id) — регистрирует нетерминал с идентификатором id в грамматике G2.

grammar_reg_symbol_marked(grammar & G2, char A, char tail_char) — формирует нетерминал с маркером (штрихом) из символа A и регистрирует его в грамматике G2. Символ маркера — последний параметр.

grammar_add_rule(grammar & G2, RULE rule) — добавляет в грамматику G2 правило rule. Символы правила должны быть зарегистрированы в грамматике.

grammar_add_rule_from(grammar & G2, RULE rule_old, grammar G1) — добавляет в грамматику G2 правило rule грамматики G1.

rule_append_symbol(grammar & G1, RULE & rule, char X) — добавляет в правило rule грамматики G1 символ X. Символ должен существовать в G1.

Все функции, как и методы классов, возвращают 0 в случае ошибки. В отличие от методов, функции выводят на терминал сообщения об ошибках.

Закомментируем код функции grammar_general.

Новый код с применением вспомогательных функций:

```
// алгоритмы общих действий с грамматикой
int grammar_general(grammar & G1, grammar & G2) {
    // регистрируем целевой символ
    char X2 = grammar_reg_symbol(G2, G1.start(), G1);
    // проверяем успешность
    if (X2 == 0) return 0;
    // количество правил G1
    int rule_count = G1.rule_count();
    // копирование правил
    for (int r = 1; r <= rule_count; r++) {
        // добавляем правило в G2
        X2 = grammar_add_rule_from(G2, G1.rule(r), G1);
        // проверяем успешность
        if (X2 == 0) return 0;
    }
    // успешное завершение
    return 1;
}
```

Разница не только в объеме. Как было сказано, вспомогательные функции выводят сообщения об ошибках, которые в случае применения методов программист должен формировать самостоятельно. При этом код ошибки возвращает метод op_error() соответствующего класса.

1.5.3. Обработка ошибок

Как было сказано, методы и вспомогательные функции возвращают 0 при обнаружении какой-либо ошибки. Если алгоритм обработки данных не содержит логических ошибок, то и ошибки функций не должны возникать. Однако во время разработки достигнуть этого сложно. Поэтому ошибки нужно контролировать.

Методы, возвращая нулевое значение, код возникшей ошибки записывают в свойства объекта. Этот код возвращает метод op_error(). В следующем примере показано, как контролируется ошибка метода.

```
char grammar_add_rule(grammar & G2, RULE rule) {
    char rules_count = G2.rule_count();
    char result = G2.rule_add(rule);
    if (result) return result;
    // проверяем ошибку
    result = G2.op_error();
    if (result == GERR_OVERFLOW_RULE) {
        printf("Error add rule: Rules overflow.\n\n");
    } else if (result == GERR_INVALID_SYMBOL_INDEX) {
        printf("Error add rule: Invalid symbol index.\n\n");
    } else if (result == GERR_RULE_USELESS) {
        return 127;
    } else if (result <= rules_count) {
        return -1;
    }
    return 0;
}
```

Здесь ошибка может возникнуть при добавлении правила rule в грамматику G2. Метод возвращает результат в переменную result. Далее проверяется, является ли result ненулевым значением. Если является, то значение result возвращается и функция завершается. В противном случае в result записывается код ошибки методом op_error(), и значение анализируется.

При использовании вспомогательной функции, такой, как в приведенном выше примере, контроль ошибок обычно отсутствует, поскольку это делает сама функция. Поэтому возможны следующие варианты вызова вспомогательных функций. Вариант 1:

```
// добавляем правило
X2 = grammar_add_rule(G1, rule);
// проверяем успешность
if (X2 == 0) return 0;
```

Вариант 2:

```
// добавляем правило и проверяем успешность
if ( ! grammar_add_rule(G1, rule) ) {
    return 0;
}
```

Во втором случае код получается немного короче и яснее.

1.5.4. Класс RULE (правило)

Класс RULE определяет также методы, которые возвращают характеристики правил, необходимые при разработке алгоритмов.

Метод is_leftr() возвращает 1, если правило леворекурсивное.

Метод is_chain() возвращает 1, если правило цепное.

Метод is_empty() возвращает 1, если правило пустое.

Другие методы класса RULE:

Метод remove(index) удаляет символ из правой части правила по его порядковому номеру index.

Метод replace(index, symbol) заменяет символ правой части правила, имеющий порядковый номер index, символом symbol.

1.5.5. Класс grammar (грамматика)

Для собственно грамматики определены следующие признаки.

Метод is_leftr_symbol() возвращает признак леворекурсивного символа, причем имеется ввиду рекурсия по правилу (явная). Этот метод вычисляет признак непосредственно во время вызова.

Важным является метод is_consistent(), который проверяет, все ли нетерминальные символы определены. Этот метод нужно вызывать перед реализацией алгоритма, чтобы убедиться в правильности грамматики.

После формирования грамматики следует всегда выполнять анализ грамматики методом analyse(), чтобы установить свойства.

Количество терминалов, нетерминалов и правил возвращают методы tok_count(), sym_count(), rule_count() соответственно.

Следует обратить внимание на методы, возвращающие первый и последний индексы нетерминалов: sym_first() и sym_last(). Если алгоритм предполагает перебор нетерминалов, нужно использовать именно эти методы, так как они предотвращают неправильное функционирование алгоритма в случае, если в грамматике не зарегистрированы нетерминалы.

Пример программирования перечисления нетерминалов:

```
// первый нетерминал
char sym_first = G1.sym_first();
// последний нетерминал
char sym_last = G1.sym_last();
for (char A = sym_first; A <= sym_last; A++) {
    printf("%s\n", G1.symbol_id(A));
}
```

Здесь переменная A является правильным индексом нетерминала.

Из других методов отметим grammar::symbol_typ(), который определяет тип символа или отсутствие символа (возвращая в этом случае S_NONE).

Дополнительно нужно изучать файл grammar.h, содержащий описание классов, а также констант перечислений, ограничений и ошибок.

1.5.6. Класс GSET (множество)

Другим важным классом является класс множества GSET, так как практически все алгоритмы используют множества. При работе со множеством нужно учитывать, что количество элементов множества ограничено.

При объявлении множества можно сразу задать его равным какому-либо одному элементу, либо равным другому множеству:

```
GSET Y = TOK_EOT;
GSET Z = Y;
```

Элемент добавляется во множество методом insert(). При этом элемент не добавляется, если он уже содержится во множестве. Лучше использовать вспомогательную функцию set_insert().

Во многих случаях основной цикл алгоритма завершается, если искомое множество не изменилось в итерации. В качестве примера можно привести следующий алгоритм:

Алгоритм — Удаление бесплодных символов

На входе: КС-грамматика $G(V_T, V_N, \mathbf{P}, S)$.

На выходе: эквивалентная КС-грамматика $G'(V_T', V_N', \mathbf{P}', S')$.

❶ $Y_0 = \varnothing$, $i = 1$.

❷ $Y_i = Y_{i-1} \cup \{ A \mid A \to \alpha, \alpha \in (Y_{i-1} \cup V_T)^* \}$.

❸ Если $Y_i = Y_{i-1}$, то п.4, иначе $i = i + 1$ и п.2.

❹ $V_N' = Y_i$, $V_T' = V_T$, $S' = S$, $\mathbf{P}' = \{ (A \to \alpha) \in \mathbf{P} \mid A \in Y_i$ и $\alpha \in (Y_i \cup V_T) \}$.

●

Цикл описывается в пункте 2, а пункт 3 обеспечивает завершение.

Для программирования алгоритма следует использовать методы fix() и changed(). Первый метод запоминает текущее количество элементов, второй метод возвращает 1, если текущее количество элементов изменилось.

Цикл программируется оператором while (1) {} так:

```
GSET Y;
while (1) {
    // фиксируем множество
    Y.fix();
    // операции, изменяющие или не изменяющие множество
    // проверяем, изменилось ли множество
    if ( ! Y.changed() ) break;
}
```

В этом примере сначала задается пустое множество оператором объявления GSET Y, соответствующее пункту 1 алгоритма. Далее формируется бесконечный цикл, в рамках которого выполняются пункты 2 и 3.

Пункт 2 описан как комментарий «операции, изменяющие или не изменяющие множество». Для разных алгоритмов это разные действия, ведущие или не ведущие к добавлению во множество новых символов.

Пункт 3 описывается вызовом метода changed() с отрицанием.

Заметим, что данный пример успешно завершается, так как множество в цикле не изменяется.

В реальном случае может произойти зацикливание, если алгоритм реализован неверно. Если установлен контроль переполнения множества, то обнаружение зацикливания алгоритма в этом случае также возможно:

```
GSET Y;
char X = 1;
while (1) {
    // фиксируем множество
    Y.fix();
    // операции, изменяющие или не изменяющие множество
    if (set_insert(Y, X++) == 0) {
        // переполнение множества
        return 0;
    }
    // проверяем, изменилось ли множество
    if ( ! Y.changed() ) break;
}
```

Здесь во множество непрерывно добавляются символы, поскольку переменная X непрерывно увеличивается, формируя новые коды. Однако по достижении значения 63 происходит переполнение, функция set_insert() возвращает 0, и выполнение цикла завершается.

Другой часто встречающейся задачей является перебор элементов множества. Это выполняется следующим образом:

```
for (char j = 1; j <= Y.count(); j++) {
    char a = Y[j];
}
```

В цикле переменная a последовательно принимает значения элементов множества Y. Здесь также цикл пробегает значения от единицы до количества элементов включительно.

Проверка наличия или отсутствия символа во множестве выполняется методами contains() и misses(), например:

```
if (Y.contains(1)) printf("Contains 1\n");
if (Y.misses(63)) printf("Misses 63\n");
```

Здесь используется множество Y, полученное ранее.

Метод contains() возвращает 1, если символ есть во множестве.

Метод misses() возвращает 1, если символа во множестве нет.

Для работы со множеством определены также следующие методы.

Метод clear() очищает множество для последующего использования.

Метод count() возвращает количество элементов.

Метод remove_by_value() удаляет элемент, используя сам элемент, метод remove_by_order() удаляет элемент по порядковому номеру.

Количество элементов в пересечении двух множеств вычисляется при помощи метода intersection_count().

Пересечение двух множеств вычисляет метод intersect().

Метод unite() вычисляет объединение двух множеств.

1.6. Формирование выходных файлов

Любое преобразование формирует выходной файл, название которого состоит из названия исходного файла и ключей, которые были использованы программой.

Так, если входной файл называется g00.sxg и задано преобразование grau, выходной файл получит название g00-grau.sxg.

Если задано несколько преобразований в командной строке, то выходной файл получит название, отражающее все преобразования. Для того же входного файла можно задать преобразования grau, aufa, dfa, faus, и выходной файл при этом будет иметь название g00-grau-aufa-dfa-faus.sxa.

Заметим, что тип выходного файла по мере преобразований может меняться, что нашло отражение в приведенном примере.

1.7. Соглашения о стандартном программировании

Некоторые действия встречаются практически во всех алгоритмах.

Это перебор правил грамматики, перебор символов правила, перебор нетерминальных символов, перебор символов множества.

Будем называть такие переборы стандартными циклами и всегда программировать их одинаковым образом. Это способствует быстрому программированию за счет того, что не нужно задумываться об именах переменных, типах данных, условиях цикла. Кроме того, появляется возможность копирования частей алгоритма из одних функций в другие.

1.7.1. Стандартный цикл по правилам

Перебирает правила грамматики. Стандартная форма:

```
// количество правил G1
char rules_count = G1.rule_count();
// просматриваем правила G1
for (char r = 1; r <= rules_count; r++) {
    // копия текущего правила
    RULE rule_old = G1.rule(r);
}
```

1.7.2. Стандартный цикл по символам правила

Перебирает символы правила. Стандартная форма:

```
// длина правила
char ru_len = rule.count();
// просматриваем символы правила
for (char s = 1; s <= ru_len; s++) {
    // текущий символ правила
    char X = rule.symbol(s);
}
```

1.7.3. Стандартный цикл по нетерминальным символам

Перебирает нетерминальные символы грамматики.
Стандартная форма:

```
// первый нетерминал
char sym_first = G1.sym_first();
// последний нетерминал
char sym_last = G1.sym_last();
// просматриваем нетерминалы G1
for (char A = sym_first; A <= sym_last; A++) {
    // действия с нетерминалом A
}
```

1.7.4. Стандартный цикл по символам множества

Перебирает символы множества. Стандартная форма:

```
// размер множества
char Y_count = Y.count();
// просматриваем символы множества
for (char j = 1; j <= Y_count; j++) {
    // текущий символ множества
    char a = Y[j];
}
```

Если в приведенных циклах количество элементов перебора не требуется, метод count() можно записывать непосредственно в условие цикла.

1.7.5. Имена переменных

При программировании следует придерживать постоянных обозначений для одних и тех же объектов. В следующей таблице приведены рекомендуемые имена переменных.

Переменная	Назначение
X1, X	символ исходной грамматики
X2, X	символ результирующей грамматики
S	целевой символ
S1	целевой символ со штрихом
L, X1, X2	левый символ правила
r	номер правила в цикле
s	номер символа правила в цикле
j	номер элемента множества в цикле
rules_count	количество правил
ru_len	количество символов правила
Y_count, count, M	количество элементов множества
rule, rule_old	правило исходной грамматики
rule, rule_new	правило результирующей грамматики

1.8. Описание алгоритмов

Любая работа в пособии сопровождается тем или иным описанием алгоритма. Это описание дается в свободной текстовой форме, с целью перечисления тех действий, которые должны быть выполнены при программировании. Задача обучаемого состоит в том, чтобы сопоставить действия с подходящими алгоритмическими конструкциями и вызовами частных или вспомогательных функций.

В качестве примера рассмотрим следующее описание.

1. Пусть есть пустое множество Y.
2. Фиксируем количество элементов Y.
3. Просматриваем правила G1.
4. Пусть rule — текущее правило.
5. Если левый символ текущего правила входит в Y, то: *добавляем в Y все нетерминальные символы правой части текущего правила*.
6. Если после просмотра всех правил Y не изменилось, переходим к 8.
7. Переходим к 2.
8. *Находим все достижимые правила*.

Описание, имеющее вид «Пусть [есть] …», предполагает объявление локальной переменной (или параметра) функции. В данном случае для правильной работы алгоритма требуется пустое множество.

Описание, имеющее вид «Просматриваем …», предполагает использование того или иного стандартного цикла. В данном случае описывается стандартный цикл по правилам грамматики.

Особую сложность представляет выявление в алгоритме бесконечного цикла, который составляет основу большинства алгоритмов.

В данном примере на цикл указывает пункт 7, который возвращает алгоритм к пункту 2. Следовательно, пункты 2–7 составляют цикл. Завершение цикла происходит в пункте 6, где проверяется условие завершения.

Программируется данный цикл примерно так.

```
while (1) {
    // фиксируем множество ...
    Y.fix();
    // просматриваем правила G1
    for (char r = 1; r <= G1.rule_count(); r++) {
        // текущее правило
        RULE rule = G1.rule(r);
        // если левый символ ...
        if (Y.contains(rule.symbol())) {
            // какие-то действия
        }
    }
    if (!Y.changed()) {
        // множество не изменилось - завершаем цикл
        break;
    }
}
```

1.9. Оформление кода

Оформление кода должно соответствовать требованиям стандарта кафедры [2]. К каждой строке кода должен быть предоставлен понятный комментарий на русском языке.

1.10. Вопросы и упражнения

1. Назовите составляющие объекта типа grammar, RULE, GSET.

2. В чем разница между символом и его двоичным представлением в двоичной грамматике?

3. Пусть исходная грамматика имеет вид:

```
S
S⟶AB
A⟶a
A⟶aA
B⟶b
```

а) запишите грамматику в строгом формате SYNAX.

б) вычислите, какие индексы получат все символы данной грамматики после ее преобразования в двоичный формат?

4. Измените грамматику G2 следующий образом:

```
<T*>
<T*>=
<T*>=<T>
<T>=<T>[*]<P>
<T>=<P>
<P>=[a]
```

Вам необходимо зарегистрировать целевой символ с апострофом '*', и добавить в грамматику два новых правила.

5. Добавьте в грамматику G2 правило 3 грамматики G1, поменяв символы правила задом наперед.

6. Добавьте в грамматику G2 правило 3 грамматики G1, заменив нетерминальные символы правой частью правила 5.

2. 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 нетерминальным символом были сосредоточены в описании грамматики в одном месте друг за другом. При этом нетерминальные символы должны начинаться с целевого символа грамматики, и далее должны описываться нетерминальные символы в том порядке, в котором они встречаются в 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, приведенную в левом столбце следующей таблицы:

До 1	После 1	До 2	После 2
A⟶SA	S⟶B	A⟶SA	S⟶B
A⟶B	S⟶AB	A⟶B	S⟶AB
B⟶b	B⟶b	B⟶b	A⟶SA
A⟶a	A⟶SA	A⟶a	A⟶B
S⟶B	A⟶B	S⟶AB	A⟶a
S⟶AB	A⟶a	S⟶B	B⟶b

После преобразования этой грамматики получим порядок правил, приведенный в первом слева столбце, обозначенном «После 1».

Если же два последних правила поменять местами, порядок правил после группирования изменится, что показано во второй группе столбцов. Группы правил для нетерминалов A и B поменялись местами.

2.1. Подготовка проекта

Для выполнения работы нам требуется грамматика.

В каталоге C:\grom\Debug сформируем файл ggr.sxg для тестовой грамматики, правила которой имеют следующий вид:

```
A⟶SA
A⟶B
B⟶b
A⟶a
S⟶B
S⟶AB
```

2.2. Конструирование алгоритма

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

Алгоритм ищет в G1 правила для некоторого нетерминала N, и записывает их в G2. Затем просматривает правила G2, ищет в правых частях некоторый нетерминал M, который еще не был исследован, принимает N равным M, и повторяет описанные действия.

Алгоритм начинает работу, принимая N равным целевому символу.

Алгоритм завершает работу, когда не может обнаружить ни одного нетерминала, который бы не был исследован.

Должно быть очевидно, что для работы алгоритма требуется множество, содержащее исследованные символы.

Недостаток алгоритма в том, что он многократно просматривает правила G2, в том числе уже просмотренные и исследованные.

Приведенный алгоритм неточен в смысле описания практических действий, ведущих к получению правильной двоичной грамматики. В нем не учитывается тот факт, что символы требуют регистрации. Если для копирования правил из одной грамматики в другую есть подходящая вспомогательная функция grammar_add_rule_from(), которая берет регистрацию на себя, то регистрация целевого символа должна быть частью алгоритма.

В конечном итоге точность алгоритма определяется тем, насколько описание соответствует имеющимся инструментам. Инструментами являются методы классов, вспомогательные функции, и структуры данных.

Мы можем создать новые инструменты. Например, для заданного нетерминала N можно сконструировать *частную функцию*, которая скопирует правила для N из G1 в G2. Название для этой функции примем равным add_rules_for_N(), и это наш новый инструмент.

Для заданного множества исследованных нетерминалов можно сконструировать *частную функцию*, которая просмотрит правила G2, и найдет или не найдет неисследованный нетерминал. Название этой функции примем равным find_unknown_N(). Это еще один инструмент.

Тогда алгоритм можно записать так (функции выделены курсивом).

1. Пусть N — целевой символ G1.
2. Зарегистрировать N в G2, в случае неудачи завершить функцию, при этом вернуть значение 0.
3. Пусть есть пустое множество P.
4. *Добавить правила для* N, в случае неудачи завершить функцию, при этом вернуть значение 0. В случае удачи N включен в P.
5. *Найти неисследованный нетерминал* в G2, записать его в N.
6. Если нетерминал не найден, перейти к п.8.
7. Перейти к п.4.
8. Завершить функцию и вернуть 1.

Этот алгоритм кажется корректным, но нужно точно определить, что делают *частные функции* add_rules_for_N() и find_unknown_N().

Функция add_rules_for_N() просматривает правила G1, и обнаружив правило, левый символ которого совпадает с N, и записывает его в G2. Если какое-то правило добавить не удалось, функция возвращает 0, если все правила добавлены, возвращает 1. Кроме того, включает N во множество P.

Функция find_unknown_N() просматривает правила G2, и в этой части она совпадает с предыдущей функцией. Но далее для каждого правила функция проверяет каждый символ правой части и определяет, является ли он нетерминальным. Если является, и если он отсутствует во множестве P, то функция возвращает найденный нетерминал в качестве результата. Если же после просмотра всех правил не найдется ни одного неисследованного нетерминала, в качестве результата функция возвращает 0.

2.3. Рабочее пространство

Программирование выполняется в модуле gugr.cpp.

В этом модуле определены две функции:

```
// группирует правила в порядке появления нетерминалов
int group_rules(grammar G1, grammar & G2) {
    return 1;
}

// точка входа в алгоритм
// группирование правил
int algorithm_group_rules(grammar & G1, FILE * target) {
    // результирующая грамматика
    grammar G2;
    // алгоритм
    int result = grammar_group_rules(G1, G2);
    // анализ грамматики для последующих алгоритмов
    G2.analyse();
    // выводим грамматику на терминал
    G2.print_out(stdout);
    // выводим свойства грамматики на терминал
    G2.print_props(stdout);
    // выводим грамматику в файл
    G2.print_out(target);
    // выходная грамматика
    G1 = G2;
    return result;
}
```

В свойствах проекта установим параметры командной строки:

```
ggr -gr
```

Следует убедиться, что управление приходит в модуль gagr.cpp.

Программирование алгоритма выполняется в функции group_rules().

Частные функции должны располагаем перед функцией group_rules() в порядке: сначала find_unknown_N(), затем add_rules_for_N().

2.4. Конструирование функций

Сначала сконструируем функцию add_rules_for_N().

Нужно решить следующие вопросы:

- какой тип функция будет возвращать?
- какие параметры функция будет иметь?

Эта функция возвращает логический результат, поэтому тип функции принимается равным int.

Функция посматривает правила грамматики G1 и записывает правила в грамматику G2, поэтому грамматики являются параметрами этой функции. Грамматика G2 будет изменяться, поэтому она передается по ссылке.

Поскольку функция ищет правила, в которых левый символ совпадает с нетерминалом N, N является параметром функции.

Поскольку функция добавляет N во множество P, P также является параметром. Множество P будет изменяться, поэтому P передается по ссылке.

Порядок параметров принимаем принятый в библиотеке: сначала целевая грамматика G2, затем искомый нетерминал N, затем исходная грамматика G1, затем множество P.

Над функцией group_rules() объявляем функцию add_rules_for_N():

```
// добавляет в G2 правила G1 для нетерминала N
int add_rules_for_N(grammar & G2, char N, grammar G1, GSET & P) {
    return 1;
}
```

Реализуем в функции стандартный цикл перебора правил G1, текущее правило записываем в переменную rule.

Сравниваем левый символ правила с N.

Если символы совпадают, добавляем правило rule в G2 при помощи вспомогательной функции grammar_add_rule_from(). Результат проверяем. Если результат равен нулю, завершаем функцию и возвращаем 0.

После цикла просмотра правил должен остаться оператор return 1.

Остается добавить нетерминал N во множество P.

Переходим к функции find_unknown_N().

Эта функция возвращает нетерминал или 0, поэтому тип возвращаемого значения должен быть char.

Параметрами функции являются грамматика G2 и множество P. Описываем функцию перед add_rules_for_N():

```
// ищет неисследованный нетерминал в G2
char find_unknown_N(grammar G2, GSET P) {
    return 0;
}
```

Функция возвращает 0 по завершении просмотра всех правил.

Реализуем в функции стандартный цикл перебора правил G2, текущее правило записываем в переменную rule.

Далее реализуем стандартный цикл перебора символов правила по переменной s. В переменную M типа char записываем символ правила номер s. Проверяем тип символа M. Если тип равен S_SYM, то M — нетерминал. В этом случае проверяем, отсутствует ли M во множестве P. Если отсутствует, то функция завершается и возвращает M.

Остается сконструировать функцию group_rules(). Именно она реализует описанный выше алгоритм.

В алгоритме может показаться неясным один момент. Пункт 7 гласит, что нужно перейти к пункту 4. А это означает, что алгоритм образует в пунктах 4–7 бесконечный цикл, выход из которого обеспечивает пункт 6, а пункт 8 уже реализован, — это имеющийся оператор return 1.

Пункт 6, в свою очередь, гласит, что выход из цикла осуществляется, если не найден неисследованный нетерминал. Это определяется следующим образом. Функция find_unknown_N() возвращает либо неисследованный нетерминал, либо 0. Возвращаемый результат функции запоминается в переменной N. Поэтому достаточно проверить значение N, чтобы выполнить пункт 6.

Реализуем эту функцию и проверяем, как работает программа.

Если все сделано правильно, при входных данных «До 1» получаются выходные данные «После 2». Хотя упорядочивание правил происходит, программа работает неверно, а это значит, что алгоритм некорректен.

Прежде, чем мы разберемся, что не так, попробуйте сами проанализировать работу программы, и найти причину неправильной работы.

2.5. Проверка корректности программы

Некорректность данного алгоритма весьма показательна.

Изучение этой некорректности поможет в дальнейшем предвидеть подобные ошибки и лучше понять двоичное представление грамматики.

Поэтому внимательно прослеживаем, как работает программа, и соответствует ли она алгоритму.

Сначала нужно записать или запомнить соответствие между символами нетерминалов и их индексами в грамматике G1. Оно следующее:

S = 61
A = 62
B = 63

В этом легко убедиться, если поставить точку остановки в начале функции group_rules() и посмотреть массив symbols грамматики G1.

Алгоритм начинается с того, что запоминает в качестве N целевой символ. При вызове функции add_rules_for_N() с N = 61 два правила для целевого символа S = 61 записываются в грамматику G2:

S→B
S→AB

При этом в грамматике G2 нетерминалы регистрируются так:

S = 61
B = 62
A = 63

Очевидно, что индексы нетерминалов A и B поменялись местами.

Последующий вызов функции find_unknown_N() обнаруживает первый неисследованный нетерминал с индексом 62, соответствующий символу B в грамматике G2. Обратим внимание, что символ обнаружен правильный, B, но его индекс 62 в грамматике G1 соответствует символу A. Поэтому последующий вызов функции add_rules_for_N() ищет в грамматике G1 не правила для B, а правила для A.

Спорным является вопрос, нужно или нет вносить коррективы в описанный алгоритм. Но вносить изменения в программу, очевидно, нужно. Вопрос только в том, куда. На самом деле имеет значение, что возвращается из функции find_unknown_N(), а также сравнение найденного нетерминала со множеством. Поэтому перед сравнением переменную M нужно заменить значением, которое возвращает функция sym_exists(), примененная к грамматике G1. Эта функция возвращает индекс символа по его идентификатору. Идентификатор символа возвращает функция symbol_id(), примененная к грамматике G2, в целом код выглядит примерно так:

```
M = G1.sym_exists(G2.symbol_id(M));
```

Теперь в первой итерации будет получен идентификатор именно символа B, и группирование правил получится правильным.

2.6. Вопросы и упражнения

1. Предложите вариант данного алгоритма, который не просматривает несколько раз одни и те же правила грамматики G2.
2. Предложите другой вариант алгоритма.

3. Удаление бесплодных символов

Цели:
- изучение алгоритма удаления бесплодных символов;
Задачи:
- конструирование функции для удаления бесплодных символов;
Алгоритм приведен в учебно-методическом пособии [1].

3.1. Конструирование алгоритма

Бесплодным называется символ, не порождающий ни одной терминальной цепочки, включая пустую. Рассмотрим грамматику G26A:

S⟶a
S⟶A
A⟶AB
B⟶C
C⟶b

В этой грамматике нетерминал A не порождает никаких терминальных цепочек. Действительно, выводы из A всегда начинаются с A:

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

Формальное описание алгоритма имеет следующий вид:

Алгоритм — Удаление бесплодных символов

На входе: КС-грамматика $G(V_T, V_N, \mathbf{P}, S)$.

На выходе: эквивалентная КС-грамматика $G'(V_T', V_N', \mathbf{P}', S')$.

❶ $Y_0 = \varnothing$, $i = 1$.

❷ $Y_i = Y_{i-1} \cup \{ A \mid A \to \alpha, \alpha \in (Y_{i-1} \cup V_T)^* \}$.

❸ Если $Y_i = Y_{i-1}$, то п.4, иначе $i = i + 1$ и п.2.

❹ $V_N' = Y_i$, $V_T' = V_T$, $S' = S$, $\mathbf{P}' = \{ (A \to \alpha) \in \mathbf{P} \mid A \in Y_i$ и $\alpha \in (Y_i \cup V_T) \}$.

●

Алгоритм строит множество Y таких нетерминалов, из которых могут быть выведены терминальные цепочки. В первой итерации ищутся нетерминалы, из которых терминальные цепочки выводятся непосредственно.

В данной грамматике это символы S и C.

В последующих итерациях ищутся нетерминалы, из которых выводятся цепочки, состоящие только из терминальных символов и символов из Y.

В данной грамматике из символа B выводится символ C, который был включен во множество при первоначальном поиске.

Построение множества Y завершается, когда очередная итерация не изменяет множество Y. После этого в результирующую грамматику записываются только те правила исходной грамматики, все нетерминалы которых, включая левый символ, входят во множество Y.

Определим *частные функции*, которые упрощают построение как алгоритма, так и программы.

Для первой итерации алгоритма нужна функция, которая определяет, что правая часть правила не содержит нетерминалов (состоит только из терминальных символов).

Для последующих итераций нужна функция, которая определяет, что правая часть правила не содержит нетерминалов, не включенных во множество Y.

Поскольку в первой итерации множество Y пусто, функция, которая применима к последующим итерациям, применима и к первой итерации.

Название *частной функции* примем равным syms_are_known().

Если задано множество Y и известны правила G1, то можно сконструировать функцию, которая сформирует грамматику G2.

Для этой *частной функции* подойдет название find_usefull_rules().

С учетом этих инструментов основной алгоритм можно описать следующим образом (*частные функции* выделены курсивом):

1. Пусть есть пустое множество небесплодных символов Y.
2. Зафиксируем количество элементов Y.
3. Просматриваем правила грамматики G1 одно за другим:
4. Если для текущего правила rule выполняется условие: *все нетерминальные символы правой части правила принадлежат* Y, то: левый символ правила rule добавляем в Y.
5. Если после просмотра всех правил Y не изменилось, перейдем к 7.
6. Перейдем к 2.
7. Вернем результат *поиска небесплодных правил* в G2.

Алгоритм функции syms_are_known() работает следующим образом.

Просматриваем все символы правой части правила, и если обнаруживаем нетерминал, не включенный в Y, возвращает 0, иначе возвращаем 1:

1. Просматриваем символы правила rule.
2. Пусть X — текущий символ правила rule.
3. Если: X — нетерминал, то: если X не содержится в Y, то: функция завершается и возвращает 0.
4. После просмотра всех символов функция возвращает 1.

Алгоритм функции find_usefull_rules() работает следующим образом.

Просматриваем правила грамматики G1. Если левый символ правила включен в Y, и все нетерминалы правила включены в Y, то добавляем правило в грамматику G2:

1. Просматриваем правила G1.
2. Пусть rule — текущее правило G1.
3. Если: Y содержит левый символ текущего правила rule, то: если выполняется условие: *все нетерминальные символы правой части правила принадлежат* Y, то: добавляем правило в G2.
4. Регистрируем целевой символ G1 в G2.

Здесь неявно предполагается, что функция возвращает либо 0 либо 1.

3.2. Рабочее пространство

Работа выполняется в модуле gaus.cpp.

Для отладки используется файл грамматики g26a.sxg. Необходимо открыть свойства проекта и установить параметры командной строки:

g26a -us

После этого нужно убедиться, что управление программой приходит в функцию algorithm_remove_useless() модуля gaus.cpp.

В модуле три функции:

```
// удаляет бесплодные символы
int remove_useless(grammar & G1, grammar & G2) {
    return 1;
}

// удаляет недостижимые символы
int remove_unreachable(grammar & G1, grammar & G2) {
    return 1;
}

// точка входа в алгоритм
// удаление бесплодных и недостижимых символов
int algorithm_remove_useless(grammar & G1, FILE * target) {
    // результирующая грамматика
    grammar G2;
    // удаление бесплодных символов
    int result = remove_useless(G1, G2);
    // анализ грамматики для последующих алгоритмов
    G2.analyse();
    // выводим грамматику на терминал
    G2.print_out(stdout);
    // выводим грамматику в файл
    G2.print_out(target);
    // входная грамматика для следующего алгоритма
    G1 = G2;
    // результат удаления бесплодных символов
    if (!result) return result;
    // результирующая грамматика
    G2.clear();
    // удаление недостижимых символов
    result = remove_unreachable(G1, G2);
    // анализ грамматики для последующих алгоритмов
    G2.analyse();
    // выводим грамматику на терминал
    G2.print_out(stdout);
    // выводим грамматику в файл
    G2.print_out(target);
    // выходная грамматика
    G1 = G2;
    return result;
}
```

Функция точки входа полностью готова к работе. Вторая функция remove_unreachable() разрабатывается в следующей работе в этом же модуле.

3.3. Конструирование функций

Функция syms_are_known().

Параметрами функции являются некоторое правило rule, грамматика G1 и множество Y. Грамматика нужна для того, чтобы определить, является ли символ нетерминальным. Описываем ее в начале модуля:

```
// возвращает 1, если все нетерминалы правила rule грамматики G1
// находятся в множестве Y
int syms_are_known(RULE rule, grammar G1, GSET Y) {
    return 1;
}
```

Основу функции составляет стандартный цикл по символам правила. Каких-либо особенностей функция не имеет.

Функция find_usefull_rules().

Параметрами функции являются грамматика G2, грамматика G1 и множество Y. Грамматика G2 должна передаваться по ссылке, так как она изменяется. Описываем функцию после функции syms_are_known():

```
// записывает в G2 правила, нетерминалы которых принадлежат Y
int find_usefull_rules(grammar & G2, grammar G1, GSET Y) {
    return 1;
}
```

Основу функции составляет стандартный цикл по правилам грамматики. У этой функции есть следующие особенности.

1. При добавлении правила следует проверять результат и в случае ошибки завершать функцию с возвратом значения 0.

2. Функция регистрирует целевой символ после добавления всех правил. Поскольку регистрация при помощи вспомогательной функции возвращает либо 0, либо 1, в завершении этой функции можно просто вернуть результат вызова *функции регистрации*.

Основная функция remove_useless().

В соответствии с алгоритмом эта функция начинается с объявления множества Y. Основу функции составляет бесконечный цикл (соответствующий пунктам 2–6 алгоритма), завершение которого определяется по неизменности множества Y в очередной итерации.

Основу бесконечного цикла составляет цикл по правилам G1.

В этой функции так же, как и в предыдущей, в завершении просто возвращается вызов функции *поиска небесплодных правил*.

3.4. Вопросы и упражнения

1. Почему в алгоритме не проверяются терминальные символы?

2. Будет ли включено в грамматику G2 пустое правило?

3. Что произойдет, если целевой символ не генерирует терминальных цепочек?

4. Удаление недостижимых символов

Цели:
- изучение алгоритма удаления недостижимых символов;
Задачи:
- конструирование функции для удаления недостижимых символов;
Алгоритм приведен в учебно-методическом пособии [1].

4.1. Конструирование алгоритма

Недостижимым называется символ, который не встречается ни в одной сентенциальной форме грамматики.

Формальное описание алгоритма имеет следующий вид:

Алгоритм — Удаление недостижимых символов

На входе: КС-грамматика $G(V_T, V_N, \mathbf{P}, S)$.

На выходе: эквивалентная КС-грамматика $G'(V_T', V_N', \mathbf{P}', S')$.

❶ $Y_0 = \{S\}$, $i = 1$.

❷ $Y_i = Y_{i-1} \cup \{ x \mid x \in V$ и $(A \to \alpha x \beta) \in \mathbf{P}, A \in Y_{i-1}, \alpha, \beta \in V^* \}$.

❸ Если $Y_i = Y_{i-1}$, то п.4, иначе $i = i + 1$ и п.2.

❹ $V_N' = V_N \cap Y_i$, $V_T' = V_T \cap Y_i$, $S' = S$, $\mathbf{P}' = \{ (A \to \alpha) \in \mathbf{P} \mid A \in Y_i \}$.

●

Алгоритм удаления недостижимых символов строит множество достижимых символов Y. Символ достижим, если он встречается в выводах из целевого символа. Удалению недостижимых символов должно предшествовать удаление бесплодных символов.

Рассмотрим грамматику G26A:

S⟶a
S⟶A
A⟶AB
B⟶C
C⟶b

В этой грамматике все символы достижимы, потому что встречаются в сентенциальных формах. Действительно, рассмотрим выводы:

S ⇒ A ⇒ AB ⇒ AC ⇒ Ab
S ⇒ a

В этих выводах все символы грамматики, нетерминальные и терминальные, встречаются хотя бы раз. Применение алгоритма удаления недостижимых символов к данной грамматике не изменяет ее.

После удаления бесплодных символов при помощи алгоритма из предыдущей работы грамматика содержит недостижимые символы B, C и b:

S⟶a
B⟶C
C⟶b

Оригинальный алгоритм во множество достижимых символов включает как нетерминальные, так и терминальные символы. Это позволяет сформировать алфавиты результирующей грамматики. На практике нет необходимости включать во множество терминальные символы. Почему?

Следующий три соображения убеждают нас в этом:

1. Терминальный символ не порождает цепочек, поэтому его включение в множество не расширяет множество впоследствии.

2. Если левый символ правила достижим, то достижимы и все терминальные символы правила.

3. Множество терминальных символов двоичной грамматики формируется автоматически по мере добавления в нее правил.

Для реализации алгоритма можно сконструировать *частную функцию*, которая в данном правиле находит нетерминалы в правой части и включает их во множество Y. Название функции find_rule_syms().

Для формирования грамматики G2 на основе грамматики G1 и множества Y можно сконструировать *частную функцию*, название которой может быть find_reachable_rules().

Получаем следующий алгоритм удаления недостижимых символов:

1. Пусть есть пустое множество достижимых символов Y.

2. Целевой символ является достижимым, включаем его в Y.

3. Фиксируем количество элементов Y.

4. Просматриваем правила G1.

5. Если левый символ текущего правила входит в Y, то: *добавляем в Y все нетерминальные символы правой части текущего правила*.

6. Если после просмотра всех правил количество элементов множества Y не изменилось, переходим к 8.

7. Переходим к 3.

8. *Находим все достижимые правила*.

Алгоритм *частной функции* find_rule_syms() простой:

1. Просматриваем символы правой части правила rule.

2. Пусть X — текущий символ правила rule.

3. Если X — нетерминал, то: включаем его в Y.

Здесь неявно предполагается, что функция возвращает 1 в случае, если все символы правой части правила были просмотрены, или 0, если символ не удалось добавить во множество Y.

Алгоритм *частной функции* find_reachable_rules():

1. Просматриваем правила G1.

2. Пусть rule — текущее правило G1.

3. Если: Y содержит левый символ текущего правила rule, то: если выполняется условие: *все нетерминальные символы правой части правила принадлежат* Y, то: добавляем правило в G2.

4. Регистрируем целевой символ G1 в G2.

Здесь неявно предполагается, что функция возвращает либо 0 либо 1.

4.2. Рабочее пространство

Рабочим модулем является gaus.cpp.

Сейчас в этом модуле незавершенной является функция:

```
// удаляет недостижимые символы
int remove_unreachable(grammar & G1, grammar & G2) {
    return 1;
}
```

Реализация алгоритма выполняется в рамках этой функции.

Необходимо убедиться, что в свойствах проекта установлены правильные параметры командной строки:

```
g26a -us
```

Необходимо также убедиться, что управление программой приходит в модуль gaus.cpp, и в функцию remove_unreachable().

Если управление приходит в модуль, но не приходит в функцию, алгоритм удаления бесплодных символов работает некорректно и возвращает ноль.

4.3. Конструирование функций

Функции find_rule_syms() и find_reachable_rules() очень похожи на соответствующие функции предыдущей работы syms_are_known() и find_usefull_rules().

Поэтому предлагается скопировать функции из предыдущей работы, вставить их перед функцией remove_unreachable(), переименовать, исправить комментарий, и немного изменить логику работы. Каких-либо особенностей в функциях нет.

Основная функция алгоритма remove_unreachable() также не имеет никаких особенных черт, и в значительной мере похожа на функцию из предыдущей работы remove_useless().

Программируем, отлаживаем, убеждаемся в правильной работе.

4.4. Вопросы и упражнения

1. В результате выполнения алгоритма упорядочиваются ли правила грамматики G2, или нет?

2. Удаляет ли алгоритм группирования правил грамматики недостижимые символы?

3. Можно ли заменить функцию поиска find_reachable_rules() функцией поиска find_usefull_rules()?

5. Устранение пустых правил

Цели:
- изучение алгоритма устранения пустых правил.
Задачи:
- конструирование алгоритма устранения пустых правил;
- разработка алгоритма формирования сочетаний символов.
Алгоритм приведен в учебно-методическом пособии [1].

5.1. Рабочее пространство

Рабочим модулем является gempty.cpp.
В модуле сейчас две функции:

```
// устраняет пустые правила
int remove_empty(grammar G1, grammar & G2) {
    return 1;
}

// точка входа в алгоритм
// устранение пустых правил
int algorithm_remove_empty(grammar & G1, FILE * target) {
    // результирующая грамматика
    grammar G2;
    // алгоритм
    int result = remove_empty(G1, G2);
    // анализ грамматики для последующих алгоритмов
    G2.analyse();
    // выводим грамматику на терминал
    G2.print_out(stdout);
    // выводим свойства грамматики на терминал
    G2.print_props(stdout);
    // выводим грамматику в файл
    G2.print_out(target);
    // выходная грамматика
    G1 = G2;
    return result;
}
```

Реализация алгоритма выполняется в функции remove_empty().
Необходимо убедиться, что в свойствах проекта установлены правильные параметры командной строки:

```
g28a -re -gr
```

Необходимо также убедиться, что управление программой приходит в модуль gempty.cpp, и в функцию remove_empty().

5.2. Предварительное конструирование алгоритма

Пустым или λ-правилом называется правило вида A→λ.
Алгоритм устранения пустых правил является очень сложным в реализации из-за необходимости формировать множество новых правил.

Рассмотрим оригинальный алгоритм.

Алгоритм — Устранение пустых правил

На входе: КС-грамматика $G(V_T, V_N, \mathbf{P}, S)$.

На выходе: эквивалентная КС-грамматика $G'(V_T', V_N', \mathbf{P}', S')$.

❶ $Y_0 = \{ A \mid (A \to \lambda) \in \mathbf{P}, A \in V_N \}$, $i = 1$.

❷ $Y_i = Y_{i-1} \cup \{ A \mid (A \to \alpha) \in \mathbf{P}, A \in V_N, \alpha \in Y_{i-1}^{+} \}$.

❸ Если $Y_i = Y_{i-1}$, то п.4, иначе $i = i + 1$ и п.2.

❹ $V_N' = V_N$, $V_T' = V_T$, $\mathbf{P}' = \{ (A \to \alpha) \in \mathbf{P} \mid \alpha \neq \lambda \}$.

❺ Если $(A \to \alpha) \in \mathbf{P}'$ и в цепочку α входят символы из множества Y_i, то на основе цепочки α строится множество цепочек $\{\alpha'\}$ исключением из α всех комбинаций символов из Y_i, и все правила вида $A \to \alpha'$ добавляются в $\mathbf{P}'$, за исключением правил вида $A \to \lambda$.

❻ Если $S \in Y_i$, то $\mathbf{P}' = \mathbf{P}' \cup \{S' \to S \mid \lambda\}$, $V_N' = V_N \cup \{S'\}$, иначе $S' = S$.

●

----- Часть I -----

1. Пусть есть пустое множество Y.

2. Просматриваем правила G1 и включаем в множество Y левые нетерминальные символы пустых правил.

Результат — Y содержит левые символы пустых правил.

----- Часть II -----

3. Фиксируем число элементов Y.

4. Просматриваем правила G1 и включаем в Y левые символы правил, в правую часть которых входят только символы из Y.

5. Если число элементов множества Y не изменилось, переходим к 7.

6. Переходим к 3.

Результат — в Y левые символы правил, вырождающихся в пусто.

----- Часть III -----

7. Просматриваем правила G1 и непустые правила включаем в G2.

Результат — все непустые правила включены в G2.

----- Часть IV -----

8. Просматриваем правила G2.

Если в правой части текущего правила есть символы из Y, то из правой части правила последовательно исключаем все возможные комбинации нетерминальных символов, входящих и в Y и в правило, и из получающихся новых правых частей формируем новые правила с левым символом, равным A. Если новое правило не пустое, добавляем его в G2.

Результат — для каждого вырождающегося в пусто символа в G2 формируется множество правил, не вырождающихся в пусто.

----- Часть V -----

10. Если целевой символ S не входит в Y, целевой символ G1 становится целевым символом G2, иначе добавляем в G2 новый целевой символ S', и два новых правила S'→λ, S'→S.

Результат — определен целевой символ G2 и порождаемый язык.

В целом алгоритм довольно простой. Сложным является пункт 8.

Во-первых, исходный алгоритм можно модифицировать, объединив пункты 2 и 7, после чего третья часть исчезает. Очевидно, что это изменение не нарушает корректность алгоритма.

Кроме того, если в результате выполнения первой части модифицированного алгоритма множество Y пусто, алгоритм завершается, так как исходная грамматика не содержит пустых правил.

Чтобы при этом был установлен целевой символ, нужно разделить пятую часть алгоритма на два условия. Условие, когда целевой символ не входит в множество Y, можно перенести в начало алгоритма.

Будем реализовывать алгоритм часть за частью.

5.3. Поиск пустых и непустых правил

Сначала реализуем часть алгоритма, которая ищет нетерминалы пустых правил, а непустые правила записывает в G2. Определим для этой цели *частную функцию* look_for_empty().

Алгоритм функции.

1. Просматриваем правила G1.
2. Пусть rule — текущее правило.
3. Если правило пустое, то: левый нетерминал правила добавляем в Y.
4. Если правило непустое, то: добавляем правило в G2.

По умолчанию функция возвращает 1. Нулевое значение возвращается в случае, когда вспомогательные функции возвращают 0.

Описываем эту функцию в начале модуля. Поскольку функция изменяет грамматику G2 и множество Y, эти параметры передаем по ссылке:

```
// непустые правила копирует в G2
// символы пустых правил добавляет в Y
int look_for_empty(grammar & G2, grammar G1, GSET & Y) {
    return 1;
}
```

Реализуем алгоритм функции.

Для определения пустого правила используется метод RULE::count() или метод RULE::is_empty(), первый метод предпочтительнее.

Переходим к функции grammar_remove_empty(). Начальная часть основного алгоритма устранения пустых правил:

1. Пусть S — целевой символ G1.
2. Регистрируем S как целевой символ G2 по умолчанию.
3. Пусть Y — пустое множество.
4. *Ищем пустые и непустые правила*.
5. Если множество Y пусто, функция завершается и возвращает 1.

Здесь курсивом выделена *частная функция* look_for_empty().

Реализуем эту часть алгоритма. Не забываем, что если look_for_empty() возвращает 0, то и grammar_remove_empty() возвращает 0.

Убеждаемся, что формируется множество Y и грамматика G2.

Для данной грамматики G1:

Y = {B, C}, B = 63, C = 63,

а грамматика G2 содержит правила

S⟶ABC
A⟶a
A⟶B
B⟶b
C⟶c .

Переходим к следующей части только после того, как данная часть будет полностью отлажена.

5.4. Поиск вырождающихся правил

В этой части нужно найти все нетерминальные символы, вырождающиеся в пустые цепочки (для простоты — *вырождающиеся*).

Часть этих символов уже включена во множество Y. Другие символы появляются в случае, если в правой части некоторого правила все символы являются вырождающимися.

В данной грамматике символы B и C вырождаются в пусто по определению. Поскольку в грамматике есть правило A⟶B, символ A является вырождающимся, а из правила S⟶ABC следует, что и символ S является вырождающимся. Именно эти символы должны быть включены в этой части во множество Y.

Для реализации данной части нужна *частная функция*, вычисляющая вырождающееся правило, назовем ее is_nullable_rule().

Алгоритм функции is_nullable_rule() простой:

1. Просматриваем символы правила rule.
2. Пусть X — текущий символ правила rule.
3. Если X не входит в Y, функция завершается и возвращает 0.
4. По завершении просмотра всех символов возвращается 1.

Таким образом, если функция is_nullable_rule() возвращает 1, то правило вырождающееся, и левый символ нужно включить во множество Y.

Реализуем данный алгоритм.

Описываем функцию is_nullable_rule() после функции look_for_empty().

Параметрами функции являются правило rule и множество Y:

```
// возвращает 0, если хотя бы один символ правила не входит в Y
// возвращает 1, если правило вырождающееся
int is_nullable_rule(RULE rule, GSET Y) {
    return 1;
}
```

Переходим к собственно алгоритму данной части. Реализуем его в еще одной *частной функции*, которую назовем look_for_nullable_rules().

Алгоритм функции look_for_nullable_rules():

1. Фиксируем множество Y.
2. Просматриваем правила G1.
3. Пусть rule — текущее правило.
4. Если *правило вырождающееся*, то: включаем в Y левый символ rule.
5. Если Y не изменилось, переходим к 7.
6. Переходим к 1.
7. Возвращаем 1.

Здесь курсивом выделена частная функция is_nullable_rule().

Реализуем данный алгоритм в функции look_for_nullable_rules().

Параметрами функции являются грамматика G1 и множество Y. Множество Y изменяется функцией, поэтому передается по ссылке.

Размещаем эту функцию после функции is_nullable_rule():

```
// формирует множество вырождающихся нетерминалов
int look_for_nullable_rules(grammar G1, GSET & Y) {
    return 1;
}
```

Описываем функцию в соответствии с алгоритмом.

Основу функции составляет бесконечный цикл (пункты 1–6) в виде, например, оператора while (1) {}. Завершение цикла происходит в случае, если в текущей итерации множество Y не изменилось. Каких-либо особенностей в функции нет.

Переходим к основному алгоритму устранения пустых правил, то есть к функции grammar_remove_empty(). Данная часть алгоритма реализуется в основном алгоритме как вызов частной функции look_for_nullable_rules().

```
    GSET Y;
    if (!look_for_empty(G2, G1, Y)) return 0;
    // проверяем множество Y
    if (Y.count() == 0) {
        // конец - нет пустых правил
        return 1;
    }
    // формируем множество вырождающихся нетерминалов
    if (!look_for_nullable_rules(G1, Y)) return 0;
```

Отлаживаем новые функции.

После первого прохода по правилам G1 в Y добавляется A.

После второго прохода по правилам G1 в Y добавляется S.

После третьего прохода множество Y не изменяется, и все нетерминальные символы G1 находятся в Y: Y = {B, C, A, S}.

5.5. Дополнение правил с вырождающимися символами

К этому моменту грамматика G2 содержит непустые правила G1, и грамматики G1 и G2 не эквивалентны. Следующая часть алгоритма должна восстановить их эквивалентность.

Рассмотрим выводы из грамматики G1.

Если A вырождается в пусто, получим вывод:

$S \Rightarrow ABC \Rightarrow^* BC \Rightarrow \ldots$

Поскольку в G2 нет пустых правил, и из символа A в ней выводится только a или b, получить данный вывод в грамматике G2 невозможно.

Поэтому для обеспечения эквивалентности в грамматику G2 должно быть добавлено правило, соответствующее этому выводу:

$S \rightarrow BC$.

Аналогичным образом, исключая из правой части правила для S различные комбинации символов A, B и C, дополнительно получим следующие выводы, которые невозможны в G2:

$S \Rightarrow ABC \Rightarrow^* AC \Rightarrow \ldots$
$S \Rightarrow ABC \Rightarrow^* AB \Rightarrow \ldots$
$S \Rightarrow ABC \Rightarrow^* C \Rightarrow \ldots$
$S \Rightarrow ABC \Rightarrow^* B \Rightarrow \ldots$
$S \Rightarrow ABC \Rightarrow^* A \Rightarrow \ldots$
$S \Rightarrow ABC \Rightarrow^* \lambda$.

Отсюда следует, что если в правой части какого-либо правила встречаются символы из множества Y, то из правой части правила формируются новые, дополнительные правые части путем исключения из начальной правой части всех комбинаций символов из Y, входящих в правую часть.

Если в результате исключения получается пустое правило, оно не добавляется в результирующую грамматику (это противоречит цели).

Таким образом, нужно разработать алгоритм исключения из правила комбинаций символов множества. Сначала разработаем механизм исключения, а затем рассмотрим то, что получится.

5.5.1. Алгоритм вычисления дополнительных правил

Пусть есть последовательность символов ABC. Будем последовательно исключать из нее все символы по одному, начиная с последнего.

Исключая C, получим остаток AB.

Исключая B, получим остаток AC.

Исключая A, получим остаток BC.

Применим этот же механизм к получившимся остаткам.

Исключая B из остатка AB получим остаток A.

Исключая A из остатка AB получим остаток B.

Исключая C из остатка AC получим остаток A.

Исключая A из остатка AC получим остаток C.

Исключая C из остатка BC получим остаток B.

Исключая B из остатка BC получим остаток C.

Если продолжать исключение дальше, получим пустые остатки. Недостаток этого алгоритма — дублирование получающихся остатков.

Алгоритм можно улучшить.

Будем реализовывать алгоритм в виде функции *f*(P, N), аргументами которой являются текущий остаток вырождающихся символов P, и некоторый символ этого остатка с порядковым номером N.

Пусть функция исключает символ N из остатка P, а затем несколько раз рекурсивно вызывает себя, передавая полученный новый остаток, и поочередно N, уменьшающиеся на единицу: N–1, N–2, …, 1.

Функция вызывается столько раз, сколько вырождающихся символов есть в первоначальном остатке P, при этом N последовательно получает значения от количества символов остатка до 1 включительно.

Если первоначальный остаток равен ABC, то функция *f* вызывается три раза со следующими параметрами:

1 — *f*(ABC, 3) ,

2 — *f*(ABC, 2) ,

3 — *f*(ABC, 1) .

Первый вызов *f*(ABC, 3) формирует остаток AB, после чего выполняется два рекурсивных вызова *f*(AB, 2) и *f*(AB, 1).

Первый рекурсивный вызов *f*(AB, 2) формирует остаток A, после чего выполняется рекурсивный вызов *f*(A, 1) с формированием пустого остатка.

Второй рекурсивный вызов *f*(AB, 1) формирует остаток B, а рекурсивных вызовов не выполняется, поскольку N равно 1.

Второй вызов *f*(ABC, 2) формирует остаток AC, после чего выполняется рекурсивный вызов *f*(AC, 1) с формированием остатка C.

Третий вызов *f*(ABC, 1) формирует остаток BC, и на этом вызовы завершаются.

При этом получается следующая последовательность вызовов и формирования остатков:

вызов *f*(ABC, 3) формирует остаток AB ,
вызов *f*(AB, 2) формирует остаток A ,
вызов *f*(A, 1) формирует остаток λ ,
вызов *f*(AB, 1) формирует остаток B ,
вызов *f*(ABC, 2) формирует остаток AC ,
вызов *f*(AC, 1) формирует остаток C ,
вызов *f*(ABC, 1) формирует остаток BC.

Таким образом, алгоритм выполняет ровно столько вызовов функции *f*, сколько остатков можно сформировать из данной последовательности.

5.5.2. Функция вычисления дополнительных правил

На основе рассмотренного алгоритма нужно сконструировать частную функцию *f* с названием remove_symbols_from(). Нам сейчас нужно определить, сколько и каких параметров эта функция будет принимать.

Очевидно, что функция формирует новые правила в грамматике G2, поэтому первый параметр — грамматика G2, передаваемая по ссылке.

Новые правила формируются исключением символов из начального и новых правил, которые содержат остатки. Поэтому второй параметр — правило rule, содержащее текущий остаток.

Поскольку правило не является чистым остатком, и может содержать символы, не являющиеся вырождающимися, нужна последовательность символов P, содержащая собственно текущий остаток. В качестве такой последовательности символов можно использовать множество GSET.

Наконец, последний параметр — номер символа N.

Частную функцию remove_symbols_from() располагаем перед функцией grammar_remove_empty():

```
// удаляет символ N множества Q 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...
char remove_symbols_from(grammar & G2, RULE rule, GSET P, char N) {
    return 1;
}
```

В основе алгоритма лежит удаление некоторого символа из правой части правила. Для этого нужно проверять каждый символ правила и сравнивать его с символом номер N остатка P. При удачном сравнении символ удаляется по его порядковому номеру методом RULE::remove().

Предлагается следующий способ реализации этой части.

Пусть нам известны не сами символы последовательности, а порядковые номера этих символов в правиле. Например, если правая часть правила равна aAbBcC, то последовательность P равна {2, 4, 6}. Это позволяет исключить из реализации поиск удаляемого символа и упростить ее.

Тогда реализация алгоритма включает в себя следующие шаги.

1. Удаляем из правила rule символ, порядковый номер которого задан в последовательности P под порядковым номером N.

2. Если получившееся новое правило rule не пусто, добавляем его в G2. Если добавить правило не удалось, завершаем функцию и возвращаем 0.

3. Удаляем из P символ с порядковым номером N.

4. Для всех N от N–1 до 1 включительно выполняем рекурсивный вызов, передавая новое правило rule, новый остаток P и новое значение N. Если вызов возвращает 0, завершаем функцию и возвращаем 0.

5. Возвращаем 1.

Реализуем алгоритм функции remove_symbols_from().

5.5.3. Формирование последовательности символов

Для формирования последовательности вырождающихся символов P некоторого правила rule сформируем частную функцию, назовем которую get_nullable_syms().

Параметрами функции являются правило rule, множество вырождающихся символов Y, и множество P, в которое запишется последовательность вырождающихся символов в правиле. Множество P передается по ссылке. Алгоритм этой функции очень простой.

1. Просматриваем символы правила rule.
2. Пусть X — текущий символ правила, имеющий номер s.
3. Если X входит в Y, то включаем s в P.

Описываем функцию перед функцией remove_symbols_from() и реализуем ее алгоритм:

```
// формирует множество P вырождающихся символов Y правила rule
int get_nullable_syms(RULE rule, GSET Y, GSET & P) {
    return 1;
}
```

5.5.4. Формирование дополнительных правил

Переходим к части 4 основного алгоритма, пункт 8. Основную работу в этой части выполняет частная функция remove_symbols_from(), последовательность P вычисляет частная функция get_nullable_syms().

С учетом этого алгоритм формирования всех дополнительных правил может быть следующим.

1. Просматриваем правила G2.
2. Пусть rule — текущее правило.
3. Пусть P — пустое множество.
4. *Определим последовательность* P (начальный остаток) правила rule.
5. Вызываем функцию remove_symbols_from() M раз, при этом N принимает значения M, M–1, …, 1 (M — количество символов в P). Если функция возвращает 0, завершаем алгоритм, возвращаем 0.
6. Возвращаем 1.

Для реализации алгоритма используем частную функцию, назовем ее get_new_rules(), и разместим ее перед функций grammar_remove_empty():

```
// формирует новые правила
int get_new_rules(grammar & G2, GSET Y) {
    return 1;
}
```

Параметрами функции являются грамматика G2, в которую будут записываться новые правила, и множество вырождающихся символов Y.

В функции grammar_remove_empty(), вызываем функцию get_new_rules():

```
    // формируем новые правила
    if (!get_new_rules(G2, Y)) return 0;
```

По завершении реализации всех функций нужно убедиться, что формируются 6 новых правил грамматики G2.

5.6. Целевой символ результирующей грамматики

В пятой части основного алгоритма нужно определить, входит ли целевой символ S грамматики G1 в множество Y. Если S входит в Y, значит он вырождается в пусто, а язык, порождаемый G1, допускает пустую цепочку.

В этом случае формируется новый целевой символ, и два новых правила для грамматики G2.

Алгоритм:

1. Если S не входит в Y, то успешное завершение основного алгоритма.
2. Пусть есть правило rule.
3. Формируем новый целевой символ S' (или, например, S*).
4. Формируем правило S'→λ и добавляем его в грамматику.
5. Формируем правило S'→S и добавляем его в грамматику.
6. Устанавливаем целевой символ S' грамматики G2 методом start().

При реализации пункта 1 лучше использовать метод GSET::misses().

Новый целевой символ S' формируется при помощи вспомогательной функции grammar_register_symbol_marked() в переменной S1 типа char. В качестве маркера используем звездочку, а не апостроф.

Для формирования двух новых правил достаточно одного объявления объекта rule. Сразу после объявления объект чист (пуст). Установка левого нетерминала правила S' формирует первое добавляемое правило. Добавление нетерминала S в правую часть этого же правила формирует второе правило.

Реализацию этой части алгоритма можно выполнить непосредственно в основной функции grammar_remove_empty().

5.7. Вопросы и упражнения

1. Корректен ли алгоритм, который вы реализовали? Чтобы убедиться в этом, замените правило S⟶ABC правилом S⟶ABCA. При этом в грамматику должно добавиться не 6, а 13 новых правил.

Если этого не происходит, проанализируйте, почему, исправьте алгоритм. Дополнительно можно заменить правило S⟶ABC правилом S⟶AaBaCa.

2. Почему исключение символов из последовательности производится задом наперед (от номера символа с бо́льшим номером к меньшему)?

6. Устранение цепных правил

Цели:
- изучение алгоритма устранения цепных правил
Задачи:
- конструирование функции для устранения цепных правил.
Алгоритм приведен в учебно-методическом пособии [1].

6.1. Конструирование алгоритма

Цепным называется правило вида A→B, где A и B — нетерминалы.

Алгоритм — Устранение цепных правил

На входе: КС-грамматика $G(V_T, V_N, \mathbf{P}, S)$.

На выходе: эквивалентная КС-грамматика $G'(V_T', V_N', \mathbf{P}', S')$.

❶ Для всех $X \in V_N$ выполнить п.2–4, затем перейти к п.5.

❷ $Y^X_0 = \{X\}$, $i = 1$.

❸ $Y^X_i = Y^X_{i-1} \cup \{ B \mid (A \to B) \in \mathbf{P}, A \in Y^X_{i-1}, B \in V_N \}$.

❹ Если $Y^X_i = Y^X_{i-1}$, то $Y^X = Y^X_i - \{X\}$ и п.2, иначе $i = i + 1$ и п.3.

❺ $V_N' = V_N$, $V_T' = V_T$, $S' = S$, $\mathbf{P}' = \mathbf{P}$ кроме правил вида $A \to B$.

❻ $\forall (A \to \alpha) \in \mathbf{P}'$: если $A \in Y^B$, $B \neq A$, то в $\mathbf{P}'$ добавляются правила $B \to \alpha$.

•

Цепные правила могут вести к образованию в грамматике циклов, поэтому в некоторых алгоритмах разбора грамматик цепные правила ведут к зацикливанию.

Алгоритм устранения цепных правил для каждого нетерминального символа строит множество нетерминальных символов, которые образуют цепные правила. При этом множество конкретного символа содержит сам символ и все символы, образующие всю цепочку цепных правил.

Затем правая часть каждого цепного правила заменяется каждой правой частью правил для правого нетерминального символа. Количество правил в грамматике при этом увеличивается.

Для примера рассмотрим грамматику G23A:

S→S+T
S→T
T→T*P
T→P
P→a
P→(S)

В ней цепными являются правила 2 и 4.

В оригинальном алгоритме сначала строятся множества цепных символов для всех нетерминальных символов, а затем формируются правила результирующей грамматики. При реализации можно обойтись одним множеством, и рассматривать символы один за другим.

Сначала все не цепные правила добавляем в G2.
Пусть текущий нетерминальный символ A есть символ S.
Принимаем Y = {S}.
Ищем цепные правила вида B→C, где B — символ, входящий в Y.
Если найдено, добавляем C в Y.
Первый просмотр правил добавляет в Y символ T (по правилу 2).
Получаем Y = {S, T}.
Второй просмотр правил добавляет в Y символ P (по правилу 4).
Получаем Y = {S, T, P}.
Третий просмотр не меняет Y, значит Y сформировано.
Удаляем из Y текущий нетерминальный символ.
Получаем Y = {T, P}.
Ищем не цепные правила вида B→α, где B входит в Y.
Если найдено, добавляем правило A→α, где A — текущий нетерминал.
Для символа T∈Y находим T→T*P, добавляем правило S→T*P.
Для символа P∈Y находим P→a, добавляем правило S→a.
Для символа P∈Y находим P→(S), добавляем правило S→(S).
Переходим к текущему символу T.
Принимаем Y = {T}.
После первого просмотра правил Y = {T, P} по правилу 4.
После второго просмотра Y не изменяется.
Удаляем T из Y. Получаем Y = {P}.
Для символа P∈Y находим P→a, добавляем правило T→a.
Для символа P∈Y находим P→(S), добавляем правило T→(S).
Переходим к текущему символу P.
Для символа P нет цепных правил, поэтому для него Y пусто.
В итоге добавлено 5 новых правил и получена грамматика:

S→S+T
S→T*P
S→a
S→(S)
T→T*P
T→a
T→(S)
P→a
P→(S)

Целевой символ остается прежним.
Алгоритм в целом состоит из трех действий.
1. Все не цепные правила добавляем в G2.
2. Для каждого нетерминала A:
2.1. Вычисляем в Y все нетерминалы, образующие цепочку от A;
2.2. Для каждого правила A→α добавляем правило B→α для каждого B, входящего в Y;

Очевидно, что пункт 2 (вместе с подпунктами) образует стандартный цикл по нетерминалам грамматики G1, A — текущий нетерминал.

В этом описании не указано, что нужно установить целевой символ G2 равным целевому символу G1, однако данное описание дает возможность понять составные части алгоритма.

Ими являются:

- поиск не цепных правил G1 и добавление из в G2;
- вычисление в Y цепочек от нетерминала A;
- формирование новых не цепных правил для нетерминалов в Y.

Эти три части алгоритма реализуем в виде частных функций:

- find_nochain_rules() ;
- find_chain_for() ;
- new_nochain_rules() .

6.2. Рабочее пространство

Рабочим модулем является gicyc.cpp. В модуле две функции:

```
// устраняет пустые правила
int remove_cycles(grammar G1, grammar & G2) {
    return 1;
}

// точка входа в алгоритм
// устранение пустых правил
int algorithm_remove_cycles(grammar & G1, FILE * target) {
    // результирующая грамматика
    grammar G2;
    // алгоритм
    int result = remove_cycles(G1, G2);
    // анализ грамматики для последующих алгоритмов
    G2.analyse();
    // выводим грамматику на терминал
    G2.print_out(stdout);
    // выводим свойства грамматики на терминал
    G2.print_props(stdout);
    // выводим грамматику в файл
    G2.print_out(target);
    // выходная грамматика
    G1 = G2;
    return result;
}
```

Реализация алгоритма выполняется в функции remove_empty().

Необходимо убедиться, что в свойствах проекта установлены правильные параметры командной строки:

```
g23a -rc -gr
```

Необходимо также убедиться, что управление программой приходит в модуль gicyc.cpp, и в функцию remove_cycles().

6.3. Поиск не цепных правил

Алгоритм функции find_nochain_rules() очень простой.

1. Пусть rules_count — количество правил G1.
2. Просматриваем правила G1.
3. Пусть rule — текущее правило.
4. Если правило не цепное, добавляем его в G2. Если добавить не удалось, завершаем алгоритм, возвращаем 0.
5. По окончании просмотра всех правил G1 возвращаем 1.

Реализуем этот алгоритм. Размещаем функцию find_nochain_rules() перед функцией remove_cycles(). Параметрами функции являются грамматики G1 и G2, причем грамматика G2 передается по ссылке:

```
// ищет не цепные правила G1 и добавляет их в G2
int find_nochain_rules(grammar G1, grammar & G2) {
    return 1;
}
```

Чтобы проверить, является правило цепным, или нет, можно использовать метод RULE::is_chain(), который возвращает 1, если правило цепное.

По завершении реализации функции find_nochain_rules() перейдем к основной функции remove_cycles().

Сначала установим целевой символ G2 равным целевому символу G1, так как это не входит ни в одну функцию. Далее вызываем частную функцию find_nochain_rules(). Если функция возвращает 0, то и основная функция возвращает 0.

Результат работы этой части алгоритма — грамматика из 4-х правил:

S⟶S+T
T⟶T*P
P⟶a
P⟶(S)

6.4. Поиск цепочек

Алгоритм формирования множества Y следующий.

1. Пусть rules_count — число правил грамматики G1.
2. Фиксируем множество Y.
3. Просматриваем правила G1.
4. Пусть rule — текущее правило G1.
5. Если правило rule цепное, и левый нетерминал правила входит в Y, то: включаем в Y правый (первый) нетерминал правила rule.
6. Если по окончании просмотра всех правил множество Y не изменилось, то перейдем к п. 8.
7. Перейдем к п.2.
8. Возвращаем 1.

Здесь неявно предполагается, что множество Y изначально содержит один из нетерминалов, обозначенных при описании как A.

Реализуем алгоритм в частной функции find_chain_for(). Параметрами функции являются грамматика G1 и множество Y, передаваемое по ссылке:

```
// ищет цепочку нетерминала A
int find_chain_for(grammar G1, GSET & Y) {
    return 1;
}
```

Размещаем функцию перед основной функцией remove_cycles(), и реализуем описанный алгоритм.

Далее в основной функции remove_cycles() формируем цикл по нетерминалам грамматики G1, так, как это описано в первой работе.

В цикле нужно объявить множество Y, и присвоить ему значение A, где A — обозначение текущего нетерминала G1. Затем вызываем частную функцию find_chain_for(), как обычно, с проверкой результата вызова.

При отладке этой части алгоритма должны получиться следующие множества Y для соответствующих нетерминалов:

Y = {S, T, P} для нетерминала S,

Y = {T, P} для нетерминала T,

Y = {P} для нетерминала P.

6.5. Формирование новых правил

Сначала вспомним, в чем заключается суть этой части.

Для символа S множество Y содержит символы {S, T, P}. Это означает, что из S выводятся цепочки из одного символа T или P. Так как из T выводится цепочка T*P, то она должна выводиться и из S. Так как из P выводятся цепочки a и (S), эти цепочки должны выводиться и из S. Поэтому алгоритм в общем виде заключается в следующем.

Берем второй символ множества Y, в данном случае T.

Просматриваем правила G1 и ищем правило вида T→α, которое не является цепным. Найдя его, заменяем левый символ на первый символ Y, и записываем получившееся правило S→α в G2. Переходим к следующему символу множества Y. Таким образом, мы перебираем символы Y, начиная со второго, и для каждого символа перебираем правила. Алгоритм.

1. Пусть rules_count — число правил G1.
2. Пусть M — число символов в Y.
3. Перебираем символы Y, начиная со второго.
4. Просматриваем правила G1.
5. Пусть rule — текущее правило.
6. Если правило rule не цепное, и левый символ совпадает с текущим символом Y, то: заменяем левый символ на первый символ Y и добавляем правило в G2. Если добавить не удалось, завершаем, возвращаем 0.
7. По завершении просмотра всех символов Y возвращаем 1.

Реализуем алгоритм в функции new_nochain_rules(). Параметрами являются грамматика G2, грамматика G1, и множество Y. Грамматика G2 передается по ссылке, так как она изменяется. Размещаем функцию перед функцией remove_cycles():

```
// формирует новые не цепные правила
int new_nochain_rules(grammar & G2, grammar G1, GSET Y) {
    return 1;
}
```

По завершении реализации убеждаемся, что формируется пять новых правил:

S⟶T*P
S⟶a
S⟶(S)
T⟶a
T⟶(S)

6.6. Вопросы и упражнения

1. Можно ли в функции new_nochain_rules() просматривать правила не грамматики G1, а правила грамматики G2? Дело в том, что в грамматике G2 нет цепных правил.

2.

7. Устранение левой рекурсии

Цели:

- изучение алгоритма устранения левой рекурсии.

Задачи:

- конструирование функции для устранения левой рекурсии.

Алгоритм приведен в учебно-методическом пособии [1].

7.1. Анализ алгоритма

Нетерминальный символ A называется леворекурсивным, если существует вывод $A \Rightarrow^* A\alpha$. Грамматика леворекурсивная, если в ней есть хотя бы один леворекурсивный символ.

Аналогичным образом определяется правая рекурсия.

Рекурсия явная, если существует правило $A \rightarrow A\alpha$.

Рекурсия неявная, если она обнаруживается во время вывода. Например, если есть правила $A \rightarrow B\beta$, $B \rightarrow A\alpha$, существует вывод $A \Rightarrow B\beta \Rightarrow A\alpha\beta$.

Некоторые алгоритмы распознавания не работают с леворекурсивными грамматиками, поэтому часто требуется устранить левую рекурсию.

Совсем избавиться от рекурсии невозможно, рекурсия — основа грамматик. Однако можно преобразовать левую рекурсию в правую.

От явной рекурсии избавиться легко.

Есть два варианта:

1) Переписать правила так, чтобы вместо левой рекурсии была правая.

Пусть есть правила с явной левой рекурсией:

$A \rightarrow A\alpha$

$A \rightarrow \beta$,

порождающие цепочки $\beta\alpha^*$.

Те же цепочки порождают следующие правила:

$A \rightarrow \beta A$

$A \rightarrow \alpha A$

$A \rightarrow \alpha$.

Здесь рекурсия правая.

2) Заменить правила в соответствии со следующей процедурой.

Пусть есть правила с явной левой рекурсией:

$A \rightarrow A\alpha$

$A \rightarrow \beta$.

Запишем вместо них две группы правил:

$A \rightarrow \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha$

$A' \rightarrow \alpha A'$.

Левая рекурсия преобразована в правую рекурсию.

Избавиться от неявной рекурсии сложнее.

Пусть есть правила, содержащие неявную левую рекурсию:

A → α
A → Bγ
B → Aβ

Неявную рекурсию нужно преобразовать в явную.

Для этого сначала нужно ввести порядок появления нетерминальных символов. Так, в приведенных правилах первым появляется символ A, этот символ в первом правиле вводит символ B. Следовательно, символ A появляется раньше символа B.

Тогда в правилах для нетерминального символа, который в правилах появляется позже, нужно найти правила, начинающиеся с нетерминального символа, который появляется раньше.

В найденных правилах правый нетерминальный символ заменяется правыми частями правил для этого символа.

В нашем случае нужно заменить A цепочками α и Bγ в правилах для B:

A → α
A → Bγ
B → αβ
B → Bγβ .

В результате неявная рекурсия стала явной.

Далее поступаем в соответствии с приведенной выше процедурой:

A → α
A → Bγ
B → αβ
B → αβB'
B' → γβ
B' → γβB' .

Алгоритм устранения левой рекурсии поочередно использует приведение неявной рекурсии к явному виду и устранение явной рекурсии.

Оригинальный алгоритм:

1. Обозначим нетерминальные символы в соответствии с моментом их появления A_1, A_2, …, A_n, n — количество нетерминалов, i=1.

2. Просматриваем правила. Если правила для A_i содержат явную левую рекурсию, запишем их в виде:

$A_i \to A_i\alpha_1 \mid A_i\alpha_2 \mid \ldots \mid A_i\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_p$.

Вместо этих правил запишем две группы других правил:

$A_i \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_p \mid \beta_1 A'_i \mid \beta_2 A'_i \mid \ldots \mid \beta_p A'_i$,

$A'_i \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A'_i \mid \alpha_2 A'_i \mid \ldots \mid \alpha_m A'_i$.

3. Если $i = n$, то перейдем к 6, иначе $i = i + 1$, $j = 1$.

4. Правила вида $A_i \to A_j\alpha$ заменяем правилами $A_i \to \gamma_1\alpha \mid \gamma_2\alpha \mid \ldots \mid \gamma_k\alpha$, где $A_j \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$ — все правила для A_j.

5. Если $j = i - 1$, то перейдем к 2, иначе $j = j + 1$, перейдем к 4.

6. Целевой символ — A_1.

Здесь в пункте 2 применяется процедура устранения явной рекурсии в правилах для A_i, в пункте 4 неявная рекурсия преобразуется в явную.

В пункте 2 опущено условие: цепочка β_p не должна начинаться с нетерминального символа A_l такого, что $l \le i$. Выполнение этого условия для целевого символа очевидно. Пункт 4 выполняет это условие для других символов.

В отсутствии этого условия появляется первое важное замечание: для правильной работы алгоритма необходимо, чтобы правила грамматики были сгруппированы по порядку появления нетерминальных символов.

Второй важное замечание: в алгоритме не формируется новая грамматика, а преобразуется исходная. Это усложняет алгоритм. Действительно, в одних случаях мы должны изменять существующее правило, а в других добавлять новое, так как в двоичной грамматике не предусмотрено удаление правил. Если бы удаление правил существовало, стандартные циклы просмотра правил перестали бы правильно работать.

Третье замечание: нет необходимости выполнять пункт 1 оригинального алгоритма, так как все нетерминальные символы в двоичной грамматике уже пронумерованы, только не от единицы, а от MAX_TOK, и все нетерминальные символы представляются в грамматике этими номерами.

Вследствие этого в алгоритме действия $i = 1$, $j = 1$ заменяются действиями i = MAX_TOK, j = MAX_TOK.

7.2. Рабочее пространство

Рабочим модулем является gleftr.cpp.

В нем определены две функции:

```
// устраняет левую рекурсию
int eliminate_leftr(grammar & G1) {
    return 1;
}

// точка входа в алгоритм
// устранение левой рекурсии
int algorithm_eliminate_leftr(grammar & G1, FILE * target) {
    // алгоритм
    int result = eliminate_leftr(G1);
    // анализ грамматики для последующих алгоритмов
    G1.analyse();
    // выводим грамматику на терминал
    G1.print_out(stdout);
    // выводим свойства грамматики на терминал
    G1.print_props(stdout);
    // выводим грамматику в файл
    G1.print_out(target);
    return result;
}
```

Реализация алгоритма выполняется в функции eliminate_leftr().

Необходимо убедиться, что в свойствах проекта установлены правильные параметры командной строки:

g32b -lr -gr

Необходимо также убедиться, что управление программой приходит в модуль gleftr.cpp, и в функцию eliminate_leftr().

Контрольной грамматикой является грамматика G32A по учебнику. Но поскольку она ведет к большому количеству правил и контролировать выполнение алгоритма на ней затруднительно, для отладки лучше использовать грамматику, приведенную выше.

Поэтому нужно создать файл грамматики g32b.sxg, и вписать следующие правила (условное название грамматики G32B):

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

Заметим, что эта грамматика используется также в следующей работе.

После отладки алгоритмов данной работы нужно убедиться, что левая рекурсия успешно устраняется в также и в грамматике G32A. Для этого нужно будет изменить параметры запуска программы.

7.3. Основной алгоритм

Поскольку конечный алгоритм очень сложный, будем использовать абстракции подчиненных алгоритмов.

Основной алгоритм состоит из двух чередующихся действий:

- устранение явной рекурсии;
- приведение неявной рекурсии к явной.

Устранение явной рекурсии выполняется для всех нетерминальных символов по порядку их регистрации в грамматике.

Пусть текущий символ A_i.

Преобразование неявной рекурсии в явную выполнятся для пар нетерминальных символов A_k и A_j таких, для которых существует правило вида $A_k \rightarrow A_j\alpha$, при этом $k = i + 1, j = 1 \ldots k - 1$.

Если в грамматике N нетерминальных символов, последовательность индексов i, j, k, а также правил, будет следующая:

Устранение явной	*Приведение к явной*
$i = 1$;	$k = 2; j = 1 \ldots 1$; $A_2 \rightarrow A_1$;
$i = 2$;	$k = 3; j = 1 \ldots 2$; $A_3 \rightarrow A_1$; $A_3 \rightarrow A_2$;
$i = 3$;	$k = 4; j = 1 \ldots 3$; $A_4 \rightarrow A_1$; $A_4 \rightarrow A_2$; $A_4 \rightarrow A_3$;
. . .	
$i = N - 1$;	$k = N; j = 1 \ldots N - 1$; $A_N \rightarrow A_1$; $A_N \rightarrow A_2$; ...; $A_N \rightarrow A_{N-1}$;
$i = N$.	

Учитывая, что индексам i, j, k соответствуют собственно индексы нетерминальных символов, вместо i, j, k будем использовать Ai и Ak и Aj.

С учетом всего основной алгоритм:

1. Пусть sym_first — индекс первого нетерминального символа.

2. Пусть sym_last — индекс последнего нетерминального символа.

3. Просматриваем нетерминальные символы от sym_first до sym_last включительно, текущий символ Ai.

4. Если символ Ai явно леворекурсивный, устраняем для правил этого символа явную рекурсию.

5. Если Ai = sym_last, завершаем просмотр нетерминалов.

6. Приводим неявную рекурсию к явной для Ak = Ai + 1.

Этот алгоритм реализуется при помощи двух вспомогательных алгоритмов:

1. Устранение явной рекурсии для Ai;

2. Приведение неявной рекурсии к явной для Ai + 1 (для Ak).

Оба алгоритма требуют индекс символа в качестве параметра.

Название частной функции для первого алгоритма — eliminate_direct(), название частной функции для второго алгоритма — indirect_to_direct().

7.4. Функция основного алгоритма

Для реализации функции основного алгоритма требуются две частных функции. Размещаем их перед функцией eliminate_leftr():

```
// устраняет непосредственную рекурсию правил для A
int eliminate_direct(char Ai, grammar & G1) {
    return 1;
}

// преобразует неявную рекурсию в явную
int indirect_to_direct(char Ak, grammar & G1) {
    return 1;
}
```

По алгоритму функциям требуется один параметр, однако реализация требует также передачи грамматики.

Переходим в функцию основного алгоритма eliminate_leftr().

В соответствии с алгоритмом объявляем и получаем значения переменных sym_first и sym_last.

Конструируем параметрический цикл по переменной A, принимающей значения от sym_first до sym_last включительно.

В цикле:

- проверяем, является ли символ A явно леворекурсивным, используя метод грамматики is_symbol_leftr(). Если является, вызываем частную функцию eliminate_direct() для символа A;

- если символ A совпадает с символом sym_last, то выходим из цикла;

- вызываем частную функцию indirect_to_direct() для A + 1;

Все вызовы частных функций, как обычно, должны выполняться в рамках условных операторов, проверяющих результат.

7.5. Алгоритм приведения неявной рекурсии к явной

Тестирование программы в ее текущем состоянии ничего не дает, так как частные функции ничего не делают, и грамматика не претерпевает никаких изменений.

По алгоритму первым является действие устранения явной рекурсии, однако для тестовой грамматики первым действием будет приведение неявной рекурсии к явной для нетерминала B. Поэтому сначала разработаем алгоритм этого действия.

Входным параметром алгоритма является символ Ak.

Нам нужно для каждого нетерминального символа Aj в диапазоне от первого включительно до Ak исключительно найти в грамматике правила вида $A_k \rightarrow A_j\alpha$, а затем для каждого найденного правила заменить символ Aj правыми частями правил для этого символа.

Будем использовать абстракцию подчиненного алгоритма, чтобы уменьшить сложность. Пусть функцию замены Aj правыми частями выполняет абстрактный алгоритм "заменяем Aj". Входным параметром алгоритма является номер правила $A_k \rightarrow A_j\alpha$.

Алгоритм приведения:

1. Есть символ Ak.
2. Есть грамматика G1.
3. Пусть sym_first — первый нетерминальный символ G1.
4. Просматриваем символы от sym_first включительно до Ak исключительно, текущий символ Aj.
5. Пусть rules_count — количество правил G1.
6. Просматриваем правила G1, номер текущего правила r.
7. Пусть rule — текущее правило.
8. Если левый символ правила совпадает с Ak, а правый (первый) символ совпадает с Aj, "заменяем Aj" для правила r.

7.6. Функция приведения неявной рекурсии к явной

Для реализации функции этого алгоритма требуется частная функция. Размещаем ее перед функцией indirect_to_direct():

```
// заменяет правило Ak->Ajx на правила Ak->wx, w - правые части Aj
// k - номер правила Ak->Ajx
int replace_Aj(int k, grammar & G1) {
    return 1;
}

// преобразует неявную рекурсию в явную
int indirect_to_direct(char Ak, grammar & G1) {
    return 1;
}
```

Переходим в функцию indirect_to_direct().

Сначала снова определяем первый нетерминальный символ sym_first.

Далее формируем параметрический цикл по символу Aj от sym_first включительно до Ak исключительно.

Внутри этого цикла:

- определяем текущее количество правил G1 в переменную rules_count;
- формируем стандартный цикл по правилам G1, переменная цикла r;

Внутри этого цикла:

- получаем текущее правило в переменную rule.
- проверяем условия:
    - левый символ правила rule совпадает с Ak;
    - правый (первый) символ правила rule совпадает с Aj.
    - если оба условия выполняются, вызываем функцию replace_Aj(), передаем ей номер текущего правила r и грамматику G1.

Программируем и проверяем, как выполняется алгоритм.

Нужно убедиться, что функция replace_Aj() вызывается с номером правила r, равным 3.

7.7. Алгоритм замены символа Aj

Теперь нам нужно разработать алгоритм замены символа Aj в правиле номер k вида $A_k \to A_j\alpha$ правыми частями правил для символа Aj.

Поскольку в grammar нет таких выразительных средств, как замена некоторого символа правила правой частью другого правила, мы оставим эту часть алгоритма абстракции вспомогательного алгоритма, который разработаем позже.

Сейчас нужно понять, что замена правила k не может выполняться одинаково для всех цепочек правил для Aj.

Предположим, есть правило $A_k \to A_j\alpha$ и два правила $Aj \to \omega_1$ и $Aj \to \omega_2$.

Одно правило грамматики нужно заменить двумя правилами.

Удалять правила мы не можем. Следовательно, имеющееся правило номер k нужно изменить на $A_k \to \omega_1\alpha$, а второе правило $A_k \to \omega_2\alpha$ добавить.

Поэтому предлагается поступить следующим образом.

Алгоритм должен искать правила для Aj при помощи цикла просмотра правил.

После того, как будет найдено первое правило для Aj номер j, мы формируем новое правило, заменяем им правило номер k, и завершаем цикл.

Далее формируем новый цикл, который просматривает правила, начиная с номера правила j + 1. В этом цикле, обнаружив правило для Aj с номером j, мы формируем новое правило и добавляем его в грамматику.

Алгоритм замены:

1. Пусть есть правило rule_k, равное правилу номер k грамматики G1.
2. Пусть есть символ Aj, равный первому символу правила rule_k.
3. Пусть есть новое правило rule_new.
4. Пусть есть количество правил rules_count.

5. Пусть есть номер текущего правила j, равный 0.
6. Просматриваем правила G1 от j + 1 до rules_count.
7. Получим копию текущего правила rule_j.
8. Если левый нетерминал текущего правила совпадает с Aj, то:
- *сформируем новое правило* из rule_k и rule_j,
- заменим им правило k,
- завершим просмотр правил.
9. Просматриваем правила G1 от j + 1 до rules_count.
10. Получим копию текущего правила rule_j.
11. Если левый нетерминал текущего правила совпадает с Aj, то:
- *сформируем новое правило* из rule_k и rule_j,
- добавим новое правило в G1.

7.8. Функция замены символа Aj

Нам требуется еще одна частная функция для формирования нового правила из двух. Описываем ее в начале модуля:

```
// формирует новое правило из двух
int compile_rule(RULE rule_k, RULE rule_j, RULE & rule, grammar G1) {
    return 1;
}
```

Функция принимает правило k, правило j, новое правило, грамматику.

Переходим в функцию replace_Aj().

В соответствии с алгоритмом, описываем и определяем значения всех требуемых переменных.

Формируем первый цикл по правилам G1.

В этом цикле:

- получим текущее правило в rule_j;
- сравним левый символ текущего правила с Aj.
- при успешном сравнении:
    - сформируем новое правило rule_new при помощи частной функции compile_rule; вызов функции нужно выполнить в рамках условного оператора, проверяющего результат;
    - заменим правило G1 номер k новым правилом;
    - завершим цикл.

Формируем еще один цикл по правилам G1, который почти ничем не отличается от предыдущего. Его можно просто скопировать и отредактировать в соответствии с алгоритмом.

Отличия второго цикла:

- вместо замены правила k добавляем новое правило, добавление выполняем в рамках условного оператора, проверяющего результат;

- после добавления нового правила цикл не завершается, а продолжается.

Поскольку частная функция не формирует никакого правила, результат тестирования сейчас бессмысленный.

7.9. Формирование нового правила

Сначала рассмотрим алгоритм. Алгоритм получает на входе правило для k, правило для j и новое правило, которое нужно сформировать.

Пусть новое правило пустое. Тогда нужно добавить в него сначала символы правила для j, а затем символы для правила k, начиная со второго.

Алгоритм:

1. Очистим новое правило.
2. Установим левый символ нового правила равным левому символу правила для k.
3. Пусть ru_len — число символов правила для j.
4. Просматриваем все символы правила для j, и добавляем каждый символ правила для j в новое правило.
5. Пусть ru_len — число символов правила для k.
4. Просматриваем символы правила для k, начиная со второго, и добавляем каждый символ правила для k в новое правило.

Переходим в функцию compile_rule().

В соответствии с алгоритмом описываем действия.

Функция состоит из двух почти одинаковых циклов добавления символов одного правила в другое.

Следует помнить, что добавление символа в правило должно выполняться в рамках условного оператора, который проверяет результат, и возвращает 0 в случае ошибки.

В конце функция возвращает 1.

По завершении конструирования этой частной функции нужно убедиться, что происходит формирование новых правил в грамматике для правил нетерминального символа B.

При этом правило

B → Ab

должно быть заменено правилом:

B → ab

и в грамматике должно появиться новое правило:

B → Bcb .

Если этого не происходит, проверяем все алгоритмы, все индексы нетерминальных символов, формирование нового правила.

7.10. Алгоритм устранения явной рекурсии

Нам остается сформировать только один алгоритм для завершения этой работы. При устранении явной рекурсии каждое правило для явно леворекурсивного символа A заменяется двумя правилами в соответствии со следующей процедурой.

Если правило имеет вид (нелеворекурсивное):

$A \rightarrow \omega$

то добавляется новое правило:

$A \rightarrow \omega A'$.

Если правило имеет вид (леворекурсивное):

$A \rightarrow A\omega$

то оно заменяется на правила:

$A' \rightarrow \omega$
$A' \rightarrow \omega A'$,

при этом первое правило заменяется, а второе правило добавляется.

Иначе говоря, в первом случае:

- добавляем это же правило, к которому приписан новый символ A',

а во втором случае:

- заменяем левый символ новым символом A';
- удаляем из этого правила первый символ;
- добавляем это же правило (без первого символа), к которому приписан новый символ A'.

Алгоритм устранения:

1. Добавляем новый символ A1, соответствующий A'.
2. Пусть есть новое правило rule_new.
3. Просматриваем правила G1. Пусть r — номер текущего правила.
4. Если левый символ текущего правила совпадает с A, то:

- если правило леворекурсивное, то:
    - заменяем левый символ символом A1;
    - удаляем первый символ правой части.

5. Копируем правило номер r в новое правило.
6. Добавляем к новому правилу символ A1.
7. Добавляем новое правило в G1.
8. Переходим к следующему правилу.

Алгоритм, как видим, оказывается простым.

7.11. Функция устранения явной рекурсии

Переходим в функцию eliminate_direct(). Сначала добавляем новый символ с помощью функции grammar_register_symbol_marked(), символ *.

Результат добавления проверяем.

Далее объявляем новое правило rule_new.

Получаем количество правил в переменную rules_count.

Формируем стандартный цикл по правилам G1 с переменной цикла r.

В этом цикле:

- проверяем, совпадает ли левый символ правила с A;
    - проверяем, является ли правило r леворекурсивным, с помощью метода RULE::is_leftr(), и если является, то:
        - заменяем левый символ правила r символом A1;
        - удаляем первый символ правила r;
    - копируем правило r в правило rule_new;
    - добавляем в rule_new символ A1;
    - добавляем rule_new в G1.

По окончании конструирования функции убеждаемся в формировании правил B→ab, B→cb, B*→abB*, B*→cbB*.

В завершение проверяем, как устраняется левая рекурсия в грамматике G32A, убеждаемся, что формируется следующая грамматика:

```
<A>
<A>=<B><C>
<A>=[a]
<B>=<C><A>
<B>=[a][b]
<B>=<C><A><B*>
<B>=[a][b]<B*>
<C>=[a]
<C>=[a]<B>
<C>=[a][b]<C><B>
<C>=[a][b]<B*><C><B>
<C>=[a]<C*>
<C>=[a]<B><C*>
<C>=[a][b]<C><B><C*>
<C>=[a][b]<B*><C><B><C*>
<B*>=<C>[b]
<B*>=<C>[b]<B*>
<C*>=<A><C><B>
<C*>=<C>
<C*>=<A><B*><C><B>
<C*>=<A><C><B><C*>
<C*>=<C><C*>
<C*>=<A><B*><C><B><C*>
```

7.12. Вопросы и упражнения

8. Левая факторизация

Цели:
- изучение левой факторизации.
Задачи:
- конструирование функции для левой факторизации
Левая факторизация описана в учебно-методическом пособии [1].

8.1. Конструирование алгоритма

Левой факторизацией называется устранение одинаковых префиксов в правилах для одного и того же нетерминального символа.

Пусть есть грамматика (полученная в предыдущей работе):

A⟶a
A⟶Bc
B⟶ab
B⟶abB*
B*⟶cb
B*⟶cbB*

Здесь есть одинаковые префиксы в правилах для B и B*.

Если устранять по одному символу префикса за один раз, то устранение префиксов для B приведет к следующим правилам для B:

B⟶aB**
B**⟶bB*
B**⟶bB*

В результате префиксы для B устранены, но появились в правилах для нового символа B**. Дальнейшее устранение префиксов в правилах для B** и B* приведет к грамматике:

A
A⟶a
A⟶Bc
B⟶aB**
B**⟶bB****
B****⟶λ
B****⟶B*
B*⟶cB***
B***⟶bB*****
B*****⟶λ
B*****⟶B*

Таким образом, каждый префикс ведет к добавлению нового символа.

Более эффективным является выявление максимально возможного префикса. В исходной грамматике таких префиксов два:

- ab для символа B,
- cb для символа B*.

Устранение этих префиксов приведет к грамматике:

A⟶a
A⟶Bc
B⟶abB**
B**⟶λ
B**⟶B*
B*⟶cb
B***⟶λ
B***⟶B*

Здесь новых символов и правил значительно меньше, однако разработка алгоритма поиска максимального префикса представляется очень сложной и выходящей за рамки данных практических работ.

8.2. Рабочее пространство

Рабочим модулем является glf.cpp. В модуле сейчас две функции:

```
// алгоритм левой факторизации
int left_factoring(grammar & G1) {
    return 1;
}

// точка входа в алгоритм
// левая факторизация
int algorithm_left_factoring(grammar & G1, FILE * target) {
    // алгоритм
    int result = left_factoring(G1);
    // анализ грамматики для последующих алгоритмов
    G1.analyse();
    // выводим грамматику на терминал
    G1.print_out(stdout);
    // выводим свойства грамматики на терминал
    G1.print_props(stdout);
    // выводим грамматику в файл
    G1.print_out(target);
    return result;
}
```

Необходимо убедиться, что в свойствах проекта установлены правильные параметры командной строки:

g32b -lr -gr -lf -gr

Необходимо также убедиться, что управление программой приходит в модуль glf.cpp, и в функцию left_factoring().

8.3. Конструирование основного алгоритма и функции

Алгоритм в целом может быть описан следующей фразой: пока в нескольких правилах для некоторого нетерминального символа A можно обнаружить одинаковый префикс x, устранить его.

В связи с этим основной алгоритм может иметь вид:

1. Пусть sym_first — первый нетерминальный символ.
2. Пусть sym_last — последний нетерминальный символ.
3. Просматриваем символы A от sym_first до sym_last.
4. Если префикс для A обнаружен, устранить его и перейти к 5.
5. Если после просмотра всех символов A префикс не обнаружен, завершить алгоритм.
6. Перейти к 2.

Этот алгоритм предполагает две абстракции:

1) обнаружение префикса prefix для символа A;
2) устранение префикса prefix для символа A.

Заметим, что пункты 2-6 представляют собой цикл с неопределенным числом итераций. В связи с тем, что при устранении префикса добавляются новые нетерминальные символы, правила для которых могут образовывать новые префиксы, вычисление индекса последнего символа sym_last должно каждый раз выполняться заново.

Для реализации алгоритма требуются две частные функции.

Размещаем их в начале модуля:

```
// устраняет префикс prefix нетерминала A
int eliminate_prefix(char prefix, char A, grammar & G1) {
    return 1;
}

// возвращает префикс для нетерминала A или 0
char find_prefix(char A, grammar G1) {
    return 0;
}
```

Теперь в функции left_factoring можно реализовать основной алгоритм.

Алгоритм достаточно простой.

Сначала определяем первый символ sym_first.

Далее конструируем бесконечный цикл в котором:

- определяем последний символ sym_last;
- описываем переменную prefix, равную нулю.
- конструируем стандартный цикл просмотра терминальных символов, параметр цикла — переменная A;
- проверяем значение переменной prefix, и если оно ложно, завершаем бесконечный цикл или алгоритм в целом.

Здесь мы вынуждены использовать переменную prefix в качестве флага обнаружения префикса.

Поскольку внутренний цикл сейчас ничего не содержит, внешний бесконечный цикл должен успешно завершаться после первой итерации.

Далее во внутреннем цикле:

- вызываем частную функцию find_prefix(), передаем ей текущий символ A и грамматику G1. Результат функции принимаем в переменную prefix. Поскольку функция по умолчанию возвращает 0, это изменение основной функции не ведет к зацикливанию;

- проверяем значение переменной prefix, и если оно не ложно, то взываем частную функцию eliminate_prefix(), передавая ей обнаруженный префикс prefix, текущий символ A, грамматику G1. Функция вызывается в рамках условного оператора, который проверяет результат и возвращает 0, если результат ложный, иначе выходим из цикла.

Таким образом, основной алгоритм реализован, но пока ничего не делает, поскольку функция find_prefix возвращает 0.

Если сначала реализовать функцию find_prefix, то алгоритм зациклится, чего хотелось бы избежать, поэтому далее разработаем функцию устранения префикса eliminate_prefix().

8.4. Конструирование алгоритма и функции устранения префикса

Рассмотрим, какие действия выполняются при устранении префикса в правилах для нетерминального символа B.

До устранения	*После устранения*
B⟶ab	B**⟶b
B⟶abB*	B**⟶bB*
	B⟶aB**

Таким образом, существующие правила изменяются так:

- левый символ заменяется новым нетерминальным символом;
- первый символ правой части удаляется.

Кроме этого, добавляется новое правило, которое формируется так:

- левый символ равен B;
- первый символ равен префиксу;
- второй символ равен новому нетерминальному символу.

Исходя из этого, алгоритм сначала должен добавлять новый нетерминальный символ. Чтобы обеспечить его уникальность, нужен частный алгоритм, который мы разработаем позднее.

Алгоритм:

1. Сформировать новый уникальный символ A1 из A.
2. Если это не удалось, завершить алгоритм, вернуть 0.
3. Пусть rules_count — количество правил G1.
4. Просматриваем правила G1, пусть r — номер текущего правила.
5. Если левый символ правила r совпадает с A и если длина правила более нуля и если первый символа совпадает с префиксом, то заменяем в правиле r левый символ на A1, удаляем первый символ правила.

6. По завершении просмотра всех правил формируем новое правило, в котором левый символ равен A, а в правой части символы: префикс и A1.

7. Добавляем новое правило в G1.

Для формирования нового уникального символа нам нужна частная функция. Размещаем ее в начале модуля:

```
// генерирует новый нетерминал
char create_unique_symbol(char A, grammar & G1) {
    return 0;
}
```

Теперь можно конструировать функцию eliminate_prefix().

Сначала формируем новый символ A1 с помощью частной функции create_unique_symbol(). Результат проверяем и в случае ошибки также возвращаем 0.

Далее в переменную rules_count получаем количество правил G1.

Формируем стандартный цикл по правилам G1, переменная цикла r.

Проверяем необходимые условия и изменяем правило r в случае выполнения всех условий.

После цикла объявляем переменную rule для нового правила.

Формируем новое правило в соответствии с алгоритмом.

Возвращаем из функции результат добавления нового правила в G1.

После завершения конструирования этой функции ничего не изменится в грамматике, поскольку новый символ пока не формируется.

8.5. Формирование нового символа

Переходим в функцию create_unique_symbol().

Мы не можем сформировать новый символ из заданного A просто путем добавления суффикса. В грамматике уже существуют такие символы.

Поэтому мы должны разработать алгоритм, который добавляет суффикс до тех пор, пока символ не станет уникальным. Определить уникальность символа можно методом grammar::symbol_exists().

Алгоритм:

1. Пусть есть идентификатор id в виде строки (буфер).

2. Скопируем в него идентификатор символа A.

3. Если длина идентификатора превышает максимально допустимую, равную константе MAX_ID, выводим на терминал сообщение, завершаем алгоритм и возвращаем 0.

4. Если в грамматике G1 существует символ с идентификатором id, то приписываем к id символ суффикса, иначе переходим к п.6.

5. Переходим к п.3

6. Регистрируем id в грамматике и возвращаем результат регистрации.

Заметим, что для регистрации символа нужно использовать вспомогательную функцию модуля util.h.

Конструируем функцию.

В начале функции объявляем буфер для идентификатора id в виде массива типа char размерностью 2 + MAX_ID.

Далее копируем в буфер идентификатор символа A.

Очевидно, что пункты 3-5 алгоритма представляют собой цикл с неопределенным числом итераций, поэтому формируем бесконечный цикл.

В этом цикле:

- проверяем текущую длину id, сравнивая ее с MAX_ID. Если текущая длина превышает MAX_ID, то выводим на терминал сообщение:
"Identifier overflow in create_unique_symbol.\n\n",
завершаем функцию и возвращаем 0.

- проверяем, существует ли в G1 символ с идентификатором id с помощью метода grammar::symbol_exists(). Если символ существует, то приписываем к буферу символ * с помощью функции strcat(). Если символ не существует, то выходим из бесконечного цикла.

В конце функция возвращает результат добавления нового символа.

Проверить правильную работу алгоритма можно, только написав вспомогательный код в основной функции, например, в ее начале:

```
char A1 = create_unique_symbol(MAX_TOK + 1, G1);
```

По завершении отладки функции этот код нужно удалить.

8.6. Конструирование алгоритма и функции поиска префикса

Для завершения работы остается разработать алгоритм для поиска префикса. В целом он выглядит так: просматриваем правила от первого до предпоследнего, и если левый символ текущего правила совпадает с A и длина правила более нуля, то запоминаем первый символ как префикс.

Затем просматриваем правила от текущего плюс один до последнего, и если левый символ текущего правила этого просмотра совпадает с A и длина правила более нуля и первый символ совпадает с префиксом, то возвращаем префикс.

Если после этих вложенных один в другой просмотров префикс не найден, возвращаем 0.

Фактически мы ищем два правила для A, которые начинаются с одного и того же символа.

Алгоритм:

1. Пусть rules_count — количество правил G1.

2. Просматриваем правила G1, от первого до rules_count исключительно, пусть r1 — номер текущего правила.

3. Если левый символ правила r1 совпадает с A, и длина правила более нуля, то запоминаем первый символ правой части правила r1 в переменной prefix.

4. Просматриваем правила G1, от r1 + 1 до rules_count включительно, пусть r2 — номер текущего правила.

5. Если левый символ правила r2 совпадает с A, и длина правила более нуля, и первый символ правила r2 совпадает с prefix, то завершаем алгоритм, возвращаем prefix.

6. Если после просмотра всех правил алгоритм не завершился, возвращаем 0.

Переходим в функцию find_prefix(), и конструируем этот алгоритм.

Сначала получаем количество правил G1 в переменную rules_count.

Далее конструируем цикл по правилам с переменной цикла r1, которая принимает значения от 1 до rules_count исключительно.

В цикле проверяем два условия и если они выполняются, то:

- запоминаем первый символ правила r1 в переменной prefix;

- конструируем второй, вложенный цикл по правилам с переменной цикла r2, которая принимает значения от r1 + 1 до rules_count включительно.

- во внутреннем цикле проверяем три условия и если они выполняются, завершаем функцию, возвращаем значение переменной prefix.

По завершении внешнего цикла возвращаем 0.

Убеждаемся, что окончательный результат работы алгоритма дает следующую грамматику:

```
<A>
<A>=[a]
<A>=<B>[c]
<B>=[a]<B**>
<B**>=[b]<B****>
<B****>=
<B****>=<B*>
<B*>=[c]<B***>
<B***>=[b]<B*****>
<B*****>=
<B*****>=<B*>
```

8.7. Вопросы и упражнения

1. Можно ли ускорить работу этого алгоритма?

2. Предложите вариант алгоритма, который определяет префикс максимальной длины.

9. Приведение к нормальной форме Хомского

Цели:

- изучение нормальной формы Хомского;
- изучение алгоритма приведения к нормальной форме Хомского.

Задачи:

- конструирование алгоритма приведения грамматики к нормальной форме Хомского.

Алгоритм приведения к нормальной форме Хомского описан в учебно-методическом пособии [1].

9.1. Конструирование алгоритма

Грамматикой в нормальной форме Хомского (CNF, Chomsky Normal Form) называется грамматика, правила которой могут иметь вид:

A→a
A→BC .

Кроме того, в грамматике может быть одно пустое правило S→λ, S — целевой символ, если язык вырождается в пусто.

Выводы в этой грамматике — бинарные деревья.

К нормальной форме Хомского может быть приведена только такая грамматика, в которой нет пустых и цепных правил.

Алгоритм ищет правила, которые не удовлетворяет условиям CNF.

Рассмотрим конкретный пример. Пусть есть приведенная грамматика:

<S>→<A><B><B> #1
<A>→<A>[a]b] #2
<A>→[a] #3
<B>→<B> [b] #4
<B>→[b][c] #5

Правило 3 сразу можно перенести в новую грамматику.

Рассмотрим правило 1.

Вместо него в новую грамматику записываются следующие правила:

<S>→<A><BB>
<BB>→<B><A>

Вместо правила 2 записываются правила:

<A>→<A><a*b*>
< a*b*>→<a*><b*>
<a*>→[a]
<b*>→[b]

Вместо правила 4 записываются правила:

<B>⟶<B><b*>
<b*>⟶[b]

Вместо правила 5 записываются правила:

<B>⟶<b*><c*>
<b*>⟶[b]
<c*>⟶[c]

Самым важным в алгоритме является формирование идентификаторов новых нетерминальных символов таким образом, чтобы избежать их коллизии. Для этого последовательности символов формируют идентификаторы новых нетерминальных символов конкатенацией идентификаторов, причем к идентификатору терминального символа приписывается суффикс в виде знака *, чтобы отличить его от идентификатора соответствующего нетерминального символа.

При этом нужно, чтобы в исходной грамматике не было идентификаторов со звездочкой, которые могут образоваться в результате работы других алгоритмов.

9.2. Рабочее пространство

Рабочим модулем является gcnf.cpp. Сейчас в нем две функции:

```
// 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) {
    // результирующая грамматика
    grammar G2;
    // алгоритм
    int result = chomsky_normal_form(G1, G2);
    // анализ грамматики для последующих алгоритмов
    G2.analyse();
    // выводим грамматику на терминал
    G2.print_out(stdout);
    // выводим свойства грамматики на терминал
    G2.print_props(stdout);
    // выводим грамматику в файл
    G2.print_out(target);
    // выходная грамматика
    G1 = G2;
    return result;
}
```

Алгоритм разрабатывается в функции chomsky_normal_form().

Перед выполнением работы в каталоге C:\grom\Debug нужно создать файл g03.sxg следующего содержания:

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

Необходимо убедиться, что в свойствах проекта установлены правильные параметры командной строки:

```
g03 -cnf -gr
```

Необходимо также убедиться, что управление программой приходит в модуль gcnf.cpp, и в функцию chomsky_normal_form().

9.3. Конструирование основного алгоритма и функции

Весь алгоритм целесообразно разбить на несколько частей и использовать частные алгоритмы.

Основной алгоритм просматривает правила G1 и в зависимости от длины правила выполняет разные действия.

Если длина правила равна нулю, грамматика не приведенная.

Если длина правила равна 1 и правый символ (первый символ правой части) нетерминальный, грамматика не приведенная.

В этих случаях алгоритм должен завершать работу с выдачей на терминал соответствующего сообщения.

Если длина правила равна 1, и правый символ терминальный, правило добавляется в G2.

Если длина правила более 1 символа, то требуется приведение этого правила к CNF, для чего будем использовать частный алгоритм.

Основной алгоритм:

1. Установить целевой символ G2 равным целевому символу G1.
2. Просматриваем правила G1.
3. Пусть rule — текущее правило.
4. Пусть ru_len — длина текущего правила.
5. Если длина правила равно нулю, выводим сообщение
"Grammar is not normalized.\n\n",
завершаем алгоритм, возвращаем 0.
6. Если длина правила равна единице, то:
- если правый символ правила терминал, добавляем правило в G2.
- иначе выводим сообщение
"Grammar is not normalized.\n\n",
завершаем алгоритм, возвращаем 0.
7. Если длина правила иная, то вызываем частный алгоритм.

Частный алгоритм будет разбирать входное правило и по мере формирования новых правил добавлять их в G2.

Сейчас нужно понять, какие параметры нужны для этого.

Пусть есть некое правило $A \rightarrow X_1X_2X_3X_4$.

Приведение этого правила порождает два правила:

$A \rightarrow A_1A_2$ (первое),

$A_2 \rightarrow X_2X_3X_4$ (второе).

Здесь A_1 — нетерминал, к которому будет преобразован первый символ исходного правила, A_2 — нетерминал, к которому будет преобразована оставшаяся часть исходного правила.

Второе правило можно разбирать этим же вспомогательным алгоритмом, используя рекурсию. Поэтому второе правило будет являться исходным при рекурсивном вызове вспомогательного алгоритма.

Проблема заключается в регистрации символов.

Когда алгоритм вызывается прямо, левый символ входного правила A зарегистрирован в G1, но не в G2.

При выполнении алгоритма в G2 регистрируются символы A_1 и A_2.

При рекурсивном вызове входным правилом является второе правило, левый символ A_2 которого зарегистрирован в G2, но не в G1.

Исходя из того, что во время выполнения алгоритма зарегистрировать левый символ A_2 в G1 представляется лишенным смысла, нужно поступить наоборот, и зарегистрировать в G2 левый символ входного правила при прямом вызове алгоритма.

Таким образом, входное правило частного алгоритма должно быть таким, чтобы левый его символ был зарегистрирован в G2.

В основном алгоритме в связи с этим нужно изменить в пункт 7: перед вызовом частного алгоритма нужно зарегистрировать в G2 левый символ правила, и установить его в качестве левого символа.

Теперь, когда понятно, какие параметры нужны для частного алгоритма, описываем его функцию в начале модуля:

```
// приводит правило длиной более 1 к нормальной форме Хомского
int rule_to_cnf(RULE rule, grammar G1, grammar & G2) {
    return 1;
}

// приводит грамматику к нормальной форме Хомского
int chomsky_normal_form(grammar G1, grammar & G2) {
    return 1;
}
```

Реализуем основной алгоритм.

Функция chomsky_normal_form().

Сначала регистрируем целевой символ при помощи вспомогательной функции grammar_set_start(). Вызов проверяем.

Далее получаем количество правил G1 в переменную rules_count. Формируем стандартный цикл просмотра правил по переменной r.

Получаем копию текущего правила в переменную rule.

Получаем длину текущего правила в переменную ru_len.

Далее каким либо образом (с помощью if или с помощью switch) выделяем значения ru_len, равные нулю, единице, и иному значению.

В случае, если ru_len равно нулю, выводим сообщение, возвращаем 0.

В случае, если ru_len равно единице, проверяем, является ли тип первого символа правила равным константе S_TOK.

Если является, то добавляем правило из G1 в G2 с помощью вспомогательной функции grammar_add_rule_from().

Если не является, выводим сообщение, возвращаем 0.

Наконец, в случае иной длины правила регистрируем левый символ в G2 и получаем индекс символа в переменную L. Результат регистрации проверяем. Если проверка успешна, устанавливаем L левым символом текущего правила, и вызываем частную функцию rule_to_cnf() в рамках условного оператора, проверяющего результат.

Тестирование должно приводить к грамматике

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

9.4. Конструирование алгоритма приведения к CNF

Пусть опять есть некое правило $A \rightarrow X_1X_2X_3X_4$. Приведение этого правила порождает два правила: первое $A \rightarrow A_1A_2$, второе $A_2 \rightarrow X_2X_3X_4$.

Заметим, что независимо от длины правила первый символ X_1 должен быть преобразован в нетерминальный символ A_1. При этом, если символ X_1 является терминальным, то дополнительно формируется еще одно правило $A_1 \rightarrow X_1$. Если же X_1 является нетерминальным, то он просто регистрируется.

Перейдем к оставшейся цепочке символов $X_2X_3X_4$.

Поскольку алгоритм рекурсивный, должно быть условие, когда рекурсия завершается. Этим условием является длина входного правила, равная двум. В этом случае, независимо от комбинации символов, функция должна завершаться. Поэтому сначала рассмотрим этот случай.

Второй символ X_2 в случае, когда символов два, так же, как и первый, преобразуется в нетерминальный символ A_2. При этом так же, как и в предыдущем случае, если символ X_2 является терминальным, то формируется правило $A_2 \rightarrow X_2$, иначе он просто регистрируется.

Затем формируется новое правило $A \rightarrow A_1A_2$, которое добавляется в G2, и функция завершается.

Остается только понять, что делать в случае, если длина входного правила больше двух символов и используется рекурсивный вызов.

В этом случае нужно сформировать правило $A_2 \rightarrow X_2X_3X_4$. Здесь A_2 — это нетерминальный символ, который получается в результате преобразования идентификаторов символов последовательности $X_2X_3X_4$.

Сама же последовательность $X_2X_3X_4$ получается в результате удаления первого символа входного правила. Заменяя во входном правиле левый символ символом A_2, который зарегистрирован в G2, получаем входное правило для рекурсивного вызова.

Заметим, что некоторые действия повторяются, а некоторые действия достаточно сложные. Поэтому имеет смысл выделить их в частные алгоритмы.

Первый частный алгоритм, получив символ X грамматики G1, формирует из него нетерминал грамматики G2 в соответствии с тем, как описано выше. Этот же алгоритм добавляет новое правило, если символ X является терминальным.

Второй частный алгоритм, получив на входе последовательность символов $X_2X_3X_4$, формирует из нее нетерминал грамматики G2.

Учитывая эти частные алгоритмы, алгоритм приведения правила к CNF может иметь вид:

1. Есть входное правило rule.
2. Пусть есть новое правило rule_new.
3. Установим левый символ нового правила rule_new равным левому символу входного правила rule.
4. Пусть ru_len — длина входного правила rule.
5. Пусть X1 — первый символ входного правила rule.
6. Сформируем нетерминал A1 из символа X1.
7. Добавим символ A1 в новое правило rule_new.
8. Если длина правила ru_len равна 2, то:
- пусть X2 — второй символ входного правила rule.
- сформируем нетерминал A2 из символа X2.
- добавим символ A2 в новое правило rule_new.
- добавим новое правило rule_new в G2, результат добавления — результат этого алгоритма.
9. Если длина правила ru_len имеет другое значение, то:
- удалим первый символ входного правила rule.
- сформируем нетерминал A2 из последовательности символов входного правила rule.
- добавим символ A2 в новое правило rule_new.
- добавим новое правило rule_new в G2.
- установим левый символ входного правила равным символу A2.
- рекурсивно вызываем себя с входным правилом rule, результат вызова — результат этого алгоритма.

9.5. Конструирование функции приведения к CNF

Прежде нужно описать две функции частных алгоритмов.

Первая функция реализует алгоритм формирования нового нетерминального символа G2 из некоторого символа G1:

```
// формирует нетерминал из символа
char create_sym_from_symbol(grammar & G2, char X, grammar G1) {
    return 0;
}
```

Вторая функция реализует алгоритм формирования нового нетерминального символа G2 из последовательности символов G1:

```
// создает нетерминал из последовательности
int create_sym_from_sequence(grammar & G2, RULE rule, grammar G1) {
    return 0;
}
```

Переходим в функцию rule_to_cnf().

В соответствии с алгоритмом, описываем новое правило rule_new, и устанавливаем его левый символ.

Далее нужно знать длину входного правила rule, вычисляем ее.

Получаем первый символ входного правила в переменную X1, и формируем из него новый нетерминал A1 при помощи частной функции create_sym_from_symbol(). После проверки результата функции добавляем символ A1 в новое правило.

Далее программируем часть алгоритма, соответствующую завершению рекурсии, то есть условию «длина входного правила равна двум».

В этой части получаем второй символ входного правила в переменную X2, и формируем из него новый нетерминал A2 при помощи частной функции create_sym_from_symbol(). После проверки результата функции добавляем символ A2 в новое правило. В завершение этой части возвращаем из функции результат добавления нового правила.

Наконец, программируем ту часть алгоритма, которая соответствует рекурсии, то есть условию «длина входного правила больше двух».

В этой части удаляем первый символ входного правила. Таким образом, символы правила составляют последовательность $X_2X_3X_4$, из которой необходимо получить новый нетерминальный символ A2 при помощи второй частной функции create_sym_from_sequence(). Этот символ добавляется в новое правило. Новое правило добавляется в G2 в рамках условного оператора, проверяющего результат.

Кроме того, что входное правило укорачивается на первый символ, левым символом правила должен быть только что полученный символ A2.

В конце этой части функция возвращает результат рекурсивного вызова, которому передается входное правило.

9.6. Формирование новых нетерминальных символов

Для завершения этой работы необходимо разработать частные функции для формирования новых нетерминальных символов.

Первая функция формирует новый символ следующим образом.

Если входной символ X является нетерминальным, то функция регистрирует его в G2 и возвращает результат регистрации.

Если входной символ X является терминальным, то функция формирует новый идентификатор из идентификатора символа X, добавляя к нему знак суффикса "*". Этот идентификатор регистрируется в G2 как символ A.

Этот символ нужно вернуть как результат выполнения функции, однако он нужен для того, чтобы сформировать новое правило. Кроме левого символа A, правило содержит собственно терминальный символ X.

Поэтому нужно зарегистрировать X в G2 с получением символа X1, который добавить в правило, а правило добавить в G2.

Пусть новый идентификатор для терминального символа формирует и регистрирует частная функция:

```
// формирует нетерминал из идентификатора терминала
int create_sym_from_tok(grammar & G2, char X, grammar G1) {
    return 0;
}
```

Тогда алгоритм первой функции:

1. Пусть st — тип входного символа X.

2. Если тип входного символа X совпадает с S_SYM, то функция возвращает результат регистрации X в G2.

3. Если тип входного символа X совпадает с S_TOK, то:

- сформируем нетерминальный символ A из входного символа X при помощи частной функции.
- зарегистрируем входной символ X в G2 как символ X1.
- пусть есть правило rule.
- установим левый символ правила равным A.
- добавим в правило символ X1.
- добавим правило в G2.
- функция возвращает A.

4. Функция возвращает 0 (если не S_SYM и не S_TOK).

Переходим в функцию create_sym_from_symbol(), и реализуем этот алгоритм. Заметим, что результат действия каждой функции нужно проверять и возвращать ноль в случае неудачи.

Вспомогательный алгоритм частной функции create_sym_from_tok():

1. Пусть есть буфер идентификатора id размером 2 + MAX_ID.

2. Скопируем в буфер идентификатор символа X.

3. Припишем к буферу знак "*" (при помощи функции strcat).

4. Функция возвращает результат регистрации идентификатора id в G2.

Переходим в функцию create_sym_from_tok() и реализуем данный алгоритм. Заметим, что буфер здесь — массив символов.

Остается реализовать вторую частную функцию.

Эта функция формирует идентификатор для нового нетерминального символа следующим образом:

1. Идентификатор текущего символа X записывается в буфер.

2. Если текущий символ X терминальный, то в буфер записывается "*".

Поскольку символов может оказаться достаточно много, нужно проверять длину идентификатора с тем, чтобы она не превысила максимально допустимую, равную константе MAX_ID.

Алгоритм:

1. Пусть есть буфер идентификатора id размером 2 + MAX_ID.

2. Пусть ru_len — длина входного правила rule.

3. Просматриваем символы правила.

4. Пусть X — текущий символ правила rule.

5. Пусть len — текущая длина идентификатора в буфере.

6. Пусть idx — указатель на идентификатор символа X в G1.

7. Прибавим длину idx к len.

8. Если len больше MAX_ID, выведем сообщение
"Identifier overflow in create_sym_from_sequence.\n\n",
завершаем функцию, возвращаем 0.

9. Приписываем идентификатор idx к буферу.

10. Если тип символа X совпадает с S_TOK, то:

- увеличим len на единицу.

- если len больше MAX_ID, выведем сообщение
"Identifier overflow in create_sym_from_sequence.\n\n",
завершаем функцию, возвращаем 0.

- приписываем суффикс "*" к буферу.

11. По завершении просмотра всех символов регистрируем идентификатор id в G2, результат регистрации — результат этой функции.

Переходим в функцию create_sym_from_sequence(), и реализуем этот алгоритм. Никаких особенностей он не имеет.

Окончательный результат работы — грамматика:

```
<S>
<S>=<A><BB>
<A>=<A><a*b*>
<A>=[a]
<BB>=<B><B>
<a*b*>=<a*><b*>
<B>=<B><b*>
<B>=<b*><c*>
<a*>=[a]
<b*>=[b]
<c*>=[c]
```

9.7. Вопросы и упражнения

Предложите другой вариант формирования новых нетерминальных символов.

10. Преобразование регулярной грамматики в автоматную

Цели:

- изучение алгоритма преобразования регулярной грамматики в автоматную.

Задачи:

- конструирование алгоритма преобразования регулярной грамматики в автоматную.

Алгоритм преобразования регулярной грамматики в автоматную приведен в учебно-методическом пособии [1].

10.1. Рабочее пространство

Рабочим модулем является grau.cpp. Сейчас в нем две функции:

```
// приводит грамматику к нормальной форме Хомского
int regular_to_auto(grammar G1, grammar & G2) {
    return 1;
}

// точка входа в алгоритм
// преобразование регулярной грамматики в автоматную
int algorithm_regular_to_auto(grammar & G1, FILE * target) {
    // результирующая грамматика
    grammar G2;
    // алгоритм
    int result = regular_to_auto(G1, G2);
    // анализ грамматики для последующих алгоритмов
    G2.analyse();
    // выводим грамматику на терминал
    G2.print_out(stdout);
    // выводим свойства грамматики на терминал
    G2.print_props(stdout);
    // выводим грамматику в файл
    G2.print_out(target);
    // выходная грамматика
    G1 = G2;
    return result;
}
```

Алгоритм разрабатывается в функции regular_to_auto().

Перед выполнением работы в каталоге C:\grom\Debug нужно создать файл грамматики g45a.sxg следующего содержания:

```
<S>
<S>=<A>[*][/]
<A>=[/][*]
<A>=<A>[x]
<A>=<A>[*]
<A>=<A>[/]
```

Необходимо убедиться, что в свойствах проекта установлены правильные параметры командной строки:

g45a -grau -gr

Необходимо также убедиться, что управление программой приходит в модуль grau.cpp, и в функцию regular_to_auto().

10.2. Основной алгоритм

Будем говорить только о приведенных леволинейных грамматиках, то есть таких, в которых все правила имеют одну из следующих форм:

$A \to Bx_1x_2x_3$
$A \to x_1x_2x_3$

Автоматная грамматика отличается от регулярной тем, что допускаются только правила вида:

$A \to Bx_1$
$A \to x_1$

Для приведения к автоматному виду:

- правило вида $A \to Bx_1x_2x_3$ заменим на два правила $A \to Cx_3$ и $C \to Bx_1x_2$.
- правило вида $A \to x_1x_2x_3$ заменим на два правила $A \to Bx_3$ и $B \to x_1x_2$.

Если при этом второе новое правило не удовлетворяет автоматному виду, применим к нему ту же процедуру замены.

Рассмотрим грамматику G45A:

$S \to A*/$
$A \to /*$
$A \to Ax$
$A \to A*$
$A \to A/$

Правило 1 не удовлетворяют условиям автоматной грамматики.

Заменим его на правила $S \to B/$ и $B \to A*$.

Правило 2 не удовлетворяют условиям автоматной грамматики.

Заменим его на правила $A \to C*$ и $C \to /$.

Новые правила удовлетворяют условиям автоматной грамматики.

Остается один вопрос: как сформировать уникальный идентификатор нового нетерминального символа. Есть следующие варианты:

1) поиск буквы, которая не используется;

2) поиск цифры, которая приписывается к определенной букве;

3) комбинация идентификаторов символов правила.

Во последнем случае требуется доказывать уникальность комбинированного идентификатора.

Учитывая, что исходные грамматики, как правило, не содержат много нетерминальных символов, первый вариант предпочтительнее.

Алгоритм похож на предыдущий (приведение к нормальной форме Хомского), поэтому будем конструировать его так же.

В общем алгоритме выделим частные подчиненные алгоритмы:

- формирование нового нетерминального символа;
- приведение правила к автоматному виду.

С учетом этого основной алгоритм:

1. Установить целевой символ G2 равным целевому символу G1.
2. Пусть rules_count — количество правил G1.
3. Просматриваем правила G1.
4. Пусть rule — текущее правило.
5. Пусть ru_len — длина текущего правила.
6. Если длина текущего правила равна нулю, вывести сообщение
"Grammar is not normalized.\n\n"
завершить алгоритм, вернуть 0.
7. Если длина текущего правила равна 1, то:
   - если первый символ терминальный, то добавить правило в G2.
   - если первый символ не терминальный, вывести сообщение
"Grammar is not normalized.\n\n"
завершить алгоритм, вернуть 0.
8. Если длина текущего правила более 1, то:
   - зарегистрировать левый символ текущего правила как L1.
   - установить левый символ текущего правила равным L1.
   - вызвать алгоритм приведения, передать ему текущее правило.

Основной алгоритм не имеет каких-либо особенностей.

Для реализации этого алгоритма нужна частная функция, которую нужно разместить в начале модуля:

```
// приводит правило к автоматному виду
int rule_to_auto(RULE rule, grammar G1, grammar & G2) {
    return 1;
}
```

Переходим в функцию regular_to_auto(), и реализуем основной алгоритм, используя частную функцию rule_to_auto().

При использовании вспомогательных функций для регистрации символов в новой грамматике вызовы этих функций, как обычно, следует выполнять в рамках условных операторов, проверяющих результат.

10.3. Алгоритм приведения правила к автоматному виду

Этот алгоритм рекурсивный и похож на алгоритм приведения правила к нормальной форме Хомского.

Условием завершения рекурсии является длина правила, равная двум. В этом случае формируется одно или два правила, в зависимости от того, какие символы в правиле.

Если длина правила больше двух, формируется рекурсивный вызов.

Для формирования нового символа нужен частный алгоритм. Для его реализации размещаем в начале модуля функцию:

```
// создает уникальный нетерминал
char create_unique_symbol(grammar & G2, grammar & G1) {
    return 0;
}
```

По умолчанию функция возвращает ложь.

Алгоритм приведения:

1. Есть входное правило rule.
2. Пусть есть новое правило rule_new.
3. Установим левый символ нового правила равным левому символу входного правила rule.
4. Пусть ru_len — длина входного правила rule.
5. Пусть X2 — последний символ входного правила.
6. Регистрируем X2 в G2, результат запишем в X2.
7. Если длина входного правила равна двум, то:
- пусть X1 — первый символ входного правила.
- пусть st — тип символа X1.
- если тип символа X1 — нетерминальный, то:
    -- регистрируем X1 в G2, результат запишем в A1.
    -- добавим в новое правило символ A1.
    -- добавим в новое правило символ X2.
    -- добавим новое правило в G2 (A→A1X2).
    -- завершаем алгоритм, возвращаем результат добавления.
- если тип символа X1 — терминальный, то:
    -- создадим новый уникальный нетерминальный символ A1.
    -- добавим в новое правило символ A1.
    -- добавим в новое правило символ X2.
    -- добавим новое правило в G2 (A→A1X2).
    -- очистим новое правило.
    -- установим левый символ нового правила равным A1.
    -- добавим в новое правило символ X1.
    -- добавим новое правило в G2 (A1→X1).
    -- завершаем алгоритм, возвращаем результат добавления.
8. Если длина входного правила не равна двум, то:
- создадим новый уникальный нетерминальный символ A1.
- добавим в новое правило символ A1.
- добавим в новое правило символ X2.
- добавим новое правило в G2 (A→A1X2).

- установим левый символ входного правила равным A1.

- удалим последний символ входного правила.

- рекурсивно вызываем себя с входным правилом rule, результат вызова — результат этого алгоритма.

Переходим в функцию regular_to_auto() и реализуем алгоритм.

Как обычно, все вызовы функций выполняются в рамках условных операторов, проверяющих результат и возвращающих 0 при лжи.

10.4. Алгоритм формирования уникального символа

Будем формировать новый символ поиском уникального идентификатора из одной буквы. Проверять уникальность идентификатора нужно как в грамматике G1, так и в грамматике G2. Действительно, если мы нашли новый уникальный идентификатор в грамматике G1, то далее он записывается в грамматику G2, и отсутствует в G1, поэтому нужно проверять уникальность идентификатора в G2.

Алгоритм:

1. Пусть есть буфер идентификатора id размером 2 + MAX_ID.

2. Пусть j = 65.

3. Запишем букву с кодом j в буфер id.

4. Если идентификатор id не существует в G1 и не существует в G2, то:

- регистрируем id в G2, завершаем алгоритм, возвращаем результат регистрации.

5. Увеличим j на единицу. Если j < 91, перейдем к 3.

6. Выводим на терминал сообщение

"Unable to create unique sym.\n\n"

завершаем алгоритм, возвращаем 0.

Переходим в функцию create_unique_symbol() и реализуем алгоритм.

Заметим, что пункты 3-5 формируют параметрический цикл, в котором переменная цикла j пробегает значения от 65 до 90 включительно, что соответствует прописным латинским буквам от A до Z.

Чтобы записать в буфер букву с кодом j, можно использовать функцию sprintf() и спецификацию %c.

10.5. Вопросы и упражнения

11. Программирование алгоритмов с конечными автоматами

Цели:

- изучение двоичного представления конечного автомата.

Задачи:

- формирование навыков разработки алгоритмов с двоичным представлением конечного автомата.

11.1. Синтаксис описания

Конечный автомат в математическом представлении есть пятерка:

- множество состояний Q;
- множество входных символов W;
- функция переходов δ;
- начальное состояние q_0;
- множество финальных состояний $F \in Q$.

Фактически конечный автомат можно задать функцией переходов, однако нужно обозначить начальное и финальные состояния.

Функция переходов состоит из переходов.

Переход описывает текущее состояние, входной символ, вызывающий переход, и состояние, в которое производится переход.

Так, если автомат находится в состоянии A, и есть переход из этого состояния в состояние B по символу x, то переход можно записать (A, x) = B.

Поскольку недетерминированные конечные автоматы могут из одного состояния переходить в несколько других по одному и тому же входному символу, то все эти переходы можно записать (A, x) = {B, C, D, …, Z}.

Пример описания конечного автомата:

```
<H>
(H,a)={A}
(A,b)={B}
(B,a)={A,S}
{S}
```

Основную часть описания составляют переходы.

Как было сказано, для задания конечного автомата необходимо также указать начальное состояние и финальные состояния.

Начальное состояние конечного автомата обозначается так же, как и целевой символ грамматики, то есть указанием идентификатора состояния в угловых скобках.

Финальное состояние обозначается указанием идентификатора состояния в фигурных скобках. Поскольку финальных состояний может быть несколько, записей о финальных состояниях также может быть несколько, однако синтаксис предполагает также возможность указания нескольких финальных состояний в одной записи, например {S1, S2, S2}.

Заметим, что следующая запись описывает тот же самый КА:

```
(H,a)={A}
(A,b)={B}
(B,a)={A}
(B,a)={S}
<H>
{S}
```

Здесь два перехода из состояния B записаны по отдельности и, кроме того, указание начального состояния перенесено из начала записи в конец.

Указание начального и финальных состояний может располагаться в записи в произвольном месте, однако условимся указывать начальное состояние в первой строке, а финальные состояния — в последних строчках.

11.2. Рабочее пространство

В каталоге C:\grom\Debug создадим новый текстовый файл f01.sxa и запишем в него описание конечного автомата M1:

```
<H>
(H,a)={A}
(A,b)={B}
(B,a)={A,S}
{S}
```

Рабочим модулем является fagen.cpp.

В модуле определены две функции:

```
// алгоритмы общих действий с конечным автоматом
int fa_general(UFA & FA1) {
    return 1;
}

// общие действия с конечным автоматом
int algorithm_fa_general(UFA & FA1, FILE * target) {
    // алгоритм
    int result = fa_general(FA1);
    // анализ КА для последующих алгоритмов
    FA1.analyse();
    // выводим КА на терминал
    FA1.print_out(stdout);
    // выводим свойства КА на терминал
    FA1.print_props(stdout);
    // выводим КА в файл
    FA1.print_out(target);
    return result;
}
```

Точкой входа в алгоритм является функция algorithm_fa_general().

Действия с конечным автоматом выполняются в функции fa_general().

В свойствах проекта нужно установить параметры командной строки:

```
f01.sxa -fa
```

Нужно убедиться, что управление программой приходит в модуль fagen.cpp и в функцию fa_general().

По мере анализа записи конечного автомата строки выводятся на терминал, после чего выводятся свойства конечного автомата.

В нашем случае на терминал будет выведено:

```
-- parse automation
<H>
(H,a)={A}
(A,b)={B}
(B,a)={A,S}
{S}
.NFA


-- fa_general
<H>
(H,a)={A}
(A,b)={B}
(B,a)={A}
(B,a)={S}
{S}

.NFA
```

Как видим, выводится всего одно свойство КА: строка .NFA обозначает недетерминированный КА;

Заметим, что методы, выводящие автомат на терминал или в файл, записывают по одному переходу в строке, поэтому записи входного конечного автомата до и после алгоритма немного отличаются.

11.3. Двоичное представление конечного автомата

Откроем модуль grammar.h и найдем описание класса UFA.

В двоичном представлении КА состоит из:

- зарегистрированных состояний states;
- зарегистрированных входных символов symbols;
- функции переходов delta.

Начальное и финальные состояния записываются в свободные байты этих структур. Заметим, что количество состояний и символов ограничено, равно как и количество финальных состояний.

Кроме того, длина идентификатора состояния или символа конечного автомата ограничена константой UFA_MAX_ID (а не MAX_ID).

Как и в случае программирования с двоичным представлением грамматик, состояния и символы должны быть зарегистрированы, при этом каждому состоянию или символу присваивается индекс, начиная от единицы.

Действия, которые выполняются с двоичным представлением, включают в себя регистрацию состояний и символов, добавление переходов, а также получение количеств состояний, символов, переходов.

Состояние регистрируется методом state_register(), а входной символ регистрируется методом symbol_register().

Для регистрации состояний и символов при программировании следует использовать вспомогательные функции, описанные в модуле gutil.h:

fa_register_state() — регистрирует состояние;

fa_register_symbol() — регистрирует входной символ.

Они отличаются тем, что в случае возникновения переполнения выводят на терминал соответствующие сообщения.

Количество зарегистрированных состояний и входных символов возвращают методы state_count() и symbol_count() соответственно.

Начальное состояние устанавливает и возвращает метод start(). Заметим, что при установке начального состояния следует использовать индекс зарегистрированного состояния.

Вспомогательная функция fa_set_start() может упростить процесс установки начального состояния.

Для задания финального состояния используется метод final_register().

Есть несколько вспомогательных функций fa_register_final(), с помощью которых можно задать финальное состояние, которые можно использовать при разных имеющихся данных.

Метод final_count() возвращает количество финальных состояний.

Метод final_get() возвращает финальное состояние по его номеру.

Метод finals() возвращает множество финальных состояний.

Метод is_final() возвращает истину, если указанное состояние является финальным.

Основу конечного автомата составляют переходы.

Количество переходов возвращают методы trans_count(). Можно узнать общее количество переходов, количество переходов из некоторого состояния и количество переходов из некоторого состояния по некоторому входному символу.

Метод to_states_of() возвращает множество состояний, в которые конечный автомат переходит из некоторого состояния по некоторому входному символу.

Новый переход добавляет метод transition_add(). На практике следует использовать одну из вспомогательных функций fa_add_transition(). Функция выбирается в зависимости от имеющихся данных.

Количество переходов из одного состояния по одному и тому же входному символу ограничено константой MAX_TRANSITION. На практике конечные автоматы очень редко имеют большое количество таких переходов. Количество состояний и символов также сильно ограничено константами MAX_UFA_STATE и MAX_UFA_SYMBOL, при это учитывалось, что все используемые конечные автоматы небольшие.

11.4. Практика формирования конечного автомата

Для выполнения следующей работы нам потребуется конечный автомат, имеющий недостижимые состояния. Вот его запись:

```
<A>
(A,0)={B}
(A,1)={C}
(B,1)={D}
(C,1)={E}
(D,0)={C}
(D,1)={E}
(E,0)={B}
(E,1)={D}
(F,0)={D}
(G,0)={F}
(G,1)={E}
{D,E}
```

Перейдем в функцию fa_general().

Прежде всего нужно очистить имеющийся конечный автомат FA1 методом clear(). Далее, используя вспомогательные функции, описанные в предыдущем разделе, нужно добавить заданные переходы, установить начальное состояние и финальные состояния.

По окончании формирования конечного автомата для каждого состояния определите количество переходов из этого состояния по каждому из входных символов.

По завершении формирования этого конечного автомата в каталоге C:\grom\Debug появится файл f01-fagen.sxa, который нужно переименовать в файл f02.sxa. Этот файл является исходным для следующей работы.

11.5. Вопросы и упражнения

12. Удаление недостижимых состояний КА

Цели:

- изучение алгоритма удаления недостижимых состояний КА.

Задачи:

- конструирование алгоритма удаления недостижимых состояний КА.

12.1. Конструирование алгоритма

Этот алгоритм достаточно простой.

В алгоритме формируется множество достижимых состояний Y.

Начальное состояние конечного автомата является достижимым, поэтому оно сразу включается в Y.

Если из этого состояния есть переходы, то все состояния, в которые переходит автомат, также являются достижимыми, а сами переходы допустимыми.

На следующей итерации рассматриваются переходы из новых состояний, которые появились в Y. Если же множество Y не изменилось, алгоритм завершает работу.

Алгоритм:

1. Пусть start — начальное состояние FA1.
2. Установим начальное состояние FA2 равным start.
3. Пусть symbol_count — количество символов FA1.
4. Пусть Y — множество, равное start.
5. Пусть начальное состояние поиска search = 1.
6. Зафиксируем в переменной count количество состояний в Y.
7. Просматриваем множество Y от search до count включительно.

7.1. Пусть state — текущее состояние Y.

7.2. Просматриваем символы FA1, текущий символ symbol.

7.2.1. Пусть S — множество состояний, в которые FA1 переходит из состояния state по символу symbol.

7.2.2. Просматриваем состояния перехода множества S, номер текущего состояния перехода k.

7.2.2.1. Пусть state_to — состояние перехода номер k из S.

7.2.2.2. Добавим state_to в Y, так как оно достижимо из state.

7.2.2.3. Добавим в FA2 переход из state в state_to по symbol.

7.3. Если state — финальное состояние FA1, зарегистрируем state как финальное состояние FA2.

8. Если количество состояний Y не изменилось, переходим к 11.
9. Новое начальное состояние поиска search = count + 1.
10. Переходим к 6.
11. Конец алгоритма, возвращаем 1.

В этом алгоритме по ходу формирования множества достижимых состояний Y формируются также и все переходы из состояний множества Y.

12.2. Рабочее пространство

Рабочим модулем является faus.cpp.

В модуле определены две функции:

```
// удаление недостижимых состояний конечного автомата
int fa_remove_unreachable(UFA FA1, UFA & FA2) {
    return 1;
}

// точка входа в алгоритм
// удаление недостижимых состояний КА
int algorithm_fa_remove_unreachable(UFA & FA1, FILE * target) {
    // выходной КА
    UFA FA2;
    // алгоритм
    int result = fa_remove_unreachable(FA1, FA2);
    // анализ КА для последующих алгоритмов
    FA2.analyse();
    // выводим КА на терминал
    FA2.print_out(stdout);
    // выводим свойства КА на терминал
    FA2.print_props(stdout);
    // выводим КА в файл
    FA2.print_out(target);
    // результирующий КА
    FA1 = FA2;
    return result;
}
```

Реализация алгоритма выполняется в функции fa_remove_unreachable().

Необходимо убедиться, что в свойствах проекта установлены правильные параметры командной строки:

```
f02.sxa -faus
```

Необходимо также убедиться, что управление программой приходит в модуль faus.cpp, и в функцию fa_remove_unreachable().

12.3. Конструирование функции

Перейдем в функцию fa_remove_unreachable().

Алгоритм состоит из начальной части, в которой задается начальное состояние FA2, и основного цикла, соответствующего пунктам 6-10.

В начальной части в переменную start нужно получить начальное состояние FA1. Затем с помощью вспомогательной функции fa_set_start() нужно установить начальное состояние FA2. Как обычно, вызов вспомогательной функции выполняется в рамках условного оператора.

Далее в переменную symbol_count нужно получить количество символов автомата FA1. Далее эта переменная используется в цикле по символам.

Далее создается множество Y и ему присваивается значение start.

Проверка этой части создает автомат с начальным состоянием.

За начальной частью располагается основной цикл. Перед этим задается начальное значение переменной search, равное единице. Эта переменная указывает то достижимое состояние Y, которое еще не использовалось для нахождения других достижимых состояний. Изначально множество Y содержит только начальное состояние автомата, и, соответственно, первый элемент множества достижимых состояний еще не использовался.

Конструируем цикл с неопределенным количеством итераций (бесконечный цикл). В начале цикла множество Y фиксируется, а в конце цикла проверяется, изменилось множество, или нет. Метод, фиксирующий множество, возвращает количество элементов множества, это значение нужно принять в переменную count, которая нужна для перемещения значения переменной search к следующему состоянию Y.

В рамках бесконечного цикла располагается цикл по достижимым состояниям множества Y, а внутри этого цикла — цикл по символам FA1.

В цикле по символам нужно получить множество символов перехода из достижимого состояния с помощью метода to_states_of(), и для каждого состояния этого множества включить состояние в множество Y, и добавить переход, соответствующий текущему достижимому состоянию, текущему символу и состоянию перехода.

Необходимо также проверить, является ли текущее достижимое состояние финальным, и если является, то зарегистрировать его в FA2.

В конце бесконечного цикла нужно передвинуть переменную search на следующее неисследованное достижимое состояние.

В первой итерации основного цикла в Y добавляются состояния B и C.

В конце итерации переменная search принимает значение 2.

Во второй итерации в Y добавляются состояния D и E.

В конце итерации переменная search принимает значение 4.

В третьей итерации добавляются только переходы из D и E, а новых достижимых состояний не добавляется, поэтому эта итерация последняя.

12.4. Вопросы и упражнения

1. Предложите другой алгоритм удаления недостижимых состояний.

13. Преобразование автоматной грамматики в конечный автомат

Цели:

- изучение алгоритма преобразования автоматной грамматики в конечный автомат.

Задачи:

- конструирование алгоритма преобразования автоматной грамматики в конечный автомат.

Алгоритм преобразования автоматной грамматики в конечный автомат описан в учебно-методическом пособии [1].

13.1. Конструирование основного алгоритма

Правила автоматной грамматики имеют один из двух видов:

A⟶Bx

A⟶x

Правило первого вида формирует переход A∈(B,x).

Правило второго вида формирует переход A∈(H,x), где H — начальное состояние конечного автомата.

Нетерминальные символы грамматики становятся состояниями, а терминальные символы — входными символами конечного автомата.

Начальное состояние конечного автомата добавляется к состояниям, полученным из нетерминальных символов. Единственная проблема при этом заключается в том, чтобы идентификатор начального состояния не конфликтовал с идентификаторами нетерминальных символов.

Целевой символ грамматики становится финальным состоянием.

Это достаточно простой алгоритм.

Для формирования идентификатора начального состояния предлагается использовать частный алгоритм. Он заключается в том, чтобы сначала попробовать сформировать идентификатор H, а если он используется в качестве идентификатора нетерминального символа грамматики, то приписывать к букве H целое число, начиная от нуля до тех пор, пока идентификатор не станет уникальным для грамматики. Название функции для этого частного алгоритма create_home_state().

Алгоритм:

1. Сформировать и зарегистрировать начальное состояние H конечного автомата. Если это сделать не удалось, завершить алгоритм, вернуть 0.

2. Установить начальное состояние FA1 равным H.

3. Пусть rules_count — количество правил G1.

4. Просматриваем правила G1.

4.1. Пусть rule — текущее правило.

4.2. Пусть L — левый символ текущего правила.

4.3. Пусть X1 — первый символ правой части текущего правила.

4.4. Пусть ru_len — длина текущего правила.

4.5. Если длина текущего правила равна единице, то:

4.5.1. Добавить переход L∈(H,X1).

4.6. Если длина текущего правила иная, то:

- пусть X2 — второй символ правой части текущего правила.

- добавить переход L∈(X1,X2).

5. Зарегистрировать целевой символ G1 как финальное состояние FA1.

6. Завершить алгоритм, вернуть результат регистрации.

13.2. Рабочее пространство

Рабочим модулем является grfa.cpp.

Сейчас в нем определены две функции:

```
// преобразует грамматику в конечный автомат
int grau_to_fa(grammar G1, UFA & FA1) {
    return 1;
}

// точка входа в алгоритм
// преобразование автоматной грамматики в конечный автомат
int algorithm_grau_to_fa(grammar G1, UFA & FA1, FILE * target) {
    // алгоритм
    int result = grau_to_fa(G1, FA1);
    // анализ КА для последующих алгоритмов
    FA1.analyse();
    // выводим КА на терминал
    FA1.print_out(stdout);
    // выводим свойства КА на терминал
    FA1.print_props(stdout);
    // выводим КА в файл
    FA1.print_out(target);
    return result;
}
```

Реализация алгоритма выполняется в функции grau_to_fa().

Необходимо убедиться, что в свойствах проекта установлены правильные параметры командной строки:

```
g45a -grau -aufa
```

Необходимо также убедиться, что управление программой приходит в модуль grfa.cpp, и в функцию grau_to_fa(). Заметим, что при этом используется входная регулярная грамматика G45A, которая сначала преобразуется в автоматную грамматику, а затем в конечный автомат.

13.3. Частная функция

Описываем частную функцию в начале модуля:

```
// формирует начальное состояние
char create_home_state(UFA & FA1, grammar G1) {
    return 0;
}
```

В начале функции задаем буфер id для идентификатора начального состояния размером 2 + UFA_MAX_ID. Начальное значение буфера равно "H".

Далее формируем параметрический цикл по переменной j, принимающей значения от 0 до 100. Внутри этого цикла:

- проверяем, существует ли идентификатор id в грамматике G1; если не существует, то регистрируем идентификатор как состояние конечного автомата FA1 и возвращаем результат регистрации;
- записываем в буфер символ H и число j, используя функцию sprintf().

По завершении цикла (что свидетельствует о том, что сформировать уникальный идентификатор не удалось), выводим на терминал
"Unable to create home state.\n\n"
завершаем функцию и возвращаем 0.

13.4. Основная функция

Переходим в функцию grau_to_fa().
Реализуем основной алгоритм.
Алгоритм не имеет каких-либо особенностей.

13.5. Вопросы и упражнения

14. Преобразование конечного автомата в автоматную грамматику

Цели:

- изучение алгоритма преобразования конечного автомата в автоматную грамматику.

Задачи:

- конструирование алгоритма преобразования конечного автомата в автоматную грамматику.

Алгоритм описан в учебно-методическом пособии [1].

14.1. Рабочее пространство

Рабочим модулем является faau.cpp.

Сейчас в нем определены две функции:

```
// преобразует конечный автомат в грамматику
int fa_to_grau(grammar G1, UFA & FA1) {
    return 1;
}

// точка входа в алгоритм
// преобразование конечного автомата в автоматную грамматику
int algorithm_fa_to_grau(UFA FA1, grammar & G1, FILE * target) {
    // алгоритм
    int result = fa_to_grau(FA1, G1);
    // анализ грамматики для последующих алгоритмов
    G1.analyse();
    // выводим грамматику на терминал
    G1.print_out(stdout);
    // выводим свойства грамматики на терминал
    G1.print_props(stdout);
    // выводим грамматику в файл
    G1.print_out(target);
    return result;
}
```

Реализация алгоритма выполняется в функции fa_to_grau().

Необходимо убедиться, что в свойствах проекта установлены правильные параметры командной строки:

```
g45a -grau -aufa -faau
```

Необходимо также убедиться, что управление программой приходит в модуль faau.cpp, и в функцию fa_to_grau(). Заметим, что изначальная входная грамматика преобразуется в автоматную, затем в конечный автомат, а затем снова в грамматику.

14.2. Изучение алгоритма

Переход вида $B \in (A,x)$ преобразуется в правило $B \to Ax$.

Переход вида $B \in (H,x)$ преобразуется в правило $B \to x$, H — начальное состояние конечного автомата.

Алгоритм обработки переходов первого вида заключается в том, чтобы перебирать состояния A, перебирать входные символы x, получать множества символов перехода B_i, и для каждой такой комбинации формировать правило. Обработка переходов второго вида отличается тем, что не нужно перебирать состояния.

Сложность представляет обработка финальных состояний. Если финальное состояние одно, оно становится целевым символом S.

Если же финальных состояний несколько, то формируется новый целевой символ S1, и столько новых правил, сколько финальных состояний, правила имеют вид $S1 \rightarrow F_i$, F_i — финальное состояние номер i.

При этом идентификатор символа S1 должен быть уникальным среди идентификаторов состояний. Для простоты положим, что в этом случае в конечном автомате нет идентификатора состояния S*.

Разделим алгоритм на три части.

В первой части обработаем финальные состояния. Во второй части обработаем переходы второго вида, а в третьей части обработаем переходы первого вида.

14.3. Формирования целевого символа

Алгоритм первой части:

1. Пусть final_count — количество финальных состояний.

2. Если финальное состояние одно, зарегистрируем его в G1 и установим как целевой символ G1.

3. Если финальных состояний более одного, то:

3.1. Зарегистрируем новый символ S*.

3.2. Просматриваем финальные состояния.

3.2.1. Зарегистрируем финальное состояние S в G1.

3.2.2. Сформируем правило $S^* \rightarrow S$.

3.2.3. Добавим правило в G1.

Эту часть общего алгоритма можно реализовать. Каких-либо особенностей здесь нет. При просмотре финальных состояний само финальное состояние можно получить с помощью метода final_get().

14.4. Переходы из начального состояния

Алгоритм:

1. Пусть symbol_count — количество символов FA1.

2. Просматриваем символы FA1, текущий символ symbol.

2.1. Пусть Q — множество символов перехода из первого состояния по текущему символу symbol.

2.2. Пусть count — количество символов перехода в Q.

2.3. Просматриваем символы перехода, номер текущего символа j.

2.3.1. Зарегистрируем символ перехода в G1 как терминал X.

2.3.2. Зарегистрируем состояние перехода в G1 как нетерминал B.

2.3.3. Пусть rule — правило.

2.3.4. Установим левый символ правила равным B.

2.3.5. Добавим в правило символ X.

2.3.6. Добавим правило в G1.

Эту часть общего алгоритма также можно реализовать. Каких-либо особенностей здесь так же нет.

14.5. Переходы из других состояний

Алгоритм этой части отличается от предыдущего тем, что предыдущий алгоритм «погружается» в цикл по не начальным состояниям.

Алгоритм:

1. Пусть state_count — количество состояний FA1.

2. Просматриваем состояния FA1, начиная со второго (полагая, что первое состояние соответствует начальному), текущее состояние state.

3. Регистрируем в G1 состояние state как A.

Далее повторяется алгоритм предыдущей части.

4. Просматриваем символы FA1, текущий символ symbol.

4.1. Пусть Q — множество символов перехода из первого состояния по текущему символу symbol.

4.2. Пусть count — количество символов перехода в Q.

4.3. Просматриваем символы перехода, номер текущего символа j.

4.3.1. Зарегистрируем символ перехода в G1 как терминал X.

4.3.2. Зарегистрируем состояние перехода в G1 как нетерминал B.

4.3.3. Пусть rule — правило.

4.3.4. Установим левый символ правила равным B.

4.3.5. Добавим в правило символ X.

4.3.6. Добавим правило в G1.

Реализуем эту часть общего алгоритма.

14.6. Вопросы и упражнения

1. В каких случаях данный алгоритм не будет работать?

2. Что нужно сделать для того, чтобы алгоритм стал корректным?

15. Построение детерминированного конечного автомата

Цели:
- изучение алгоритма построения ДКА.
Задачи:
- конструирование алгоритма построения ДКА.
Алгоритм приведен в учебно-методическом пособии [1].

15.1. Рабочее пространство

Рабочим модулем является dfa.cpp.
Сейчас в нем определены две функции:

```
// преобразует НКА в ДКА
int nfa_to_dfa(UFA & FA1) {
    return 1;
}

// точка входа в алгоритм
// преобразование конечного автомата в автоматную грамматику
int algorithm_dfa(UFA & FA1, FILE * target) {
    // алгоритм
    int result = nfa_to_dfa(FA1);
    // анализ КА для последующих алгоритмов
    FA1.analyse();
    // выводим КА на терминал
    FA1.print_out(stdout);
    // выводим свойства КА на терминал
    FA1.print_props(stdout);
    // выводим КА в файл
    FA1.print_out(target);
    return result;
}
```

Реализация алгоритма выполняется в функции nfa_to_dfa().
Необходимо убедиться, что в свойствах проекта установлены правильные параметры командной строки:

```
g45a -grau -aufa -faus -dfa -faus
```

Необходимо также убедиться, что управление программой приходит в модуль dfa.cpp, и в функцию nfa_to_dfa(). Заметим, что входная грамматика преобразуется в автоматную, затем в конечный автомат, который упорядочивается, затем приводится к детерминированному виду, после чего из него исключаются недостижимые состояния.

15.2. Конструирование алгоритма

Пусть есть несколько переходов из некоторого состояния state по символу symbol, и состояния переходов условно A, B, C: (state,symbol)={A,B,C}.
Добавим в конечный автомат к состояние ABC, идентификатор которого есть комбинация идентификаторов, назовем его комбинированным.

Заменим переходы в состояния комбинированного состояния одним переходом в комбинированное состояние: (state,symbol)={ABC}.

Добавим переходы из комбинированного состояния ABC по символу symbol_to, если есть переход из любого из состояний комбинированного состояния по этому символу. Если, например, есть переход (A,symbol_to)={D}, то добавим переход (ABC,symbol_to)={D}.

Таким образом все недетерминированные переходы будут заменены, однако при таком алгоритме в автомате могут оставаться недостижимые состояния, поэтому после приведения автомата к детерминированному виду нужно удалить недостижимые состояния.

Заметим, что можно сконструировать алгоритм, в котором оба действия совмещены (приведение к детерминированному виду и удаление недостижимых состояний).

Алгоритм:

1. Пусть есть признак изменения автомата changed.
2. Установим признак changed равным нулю.
3. Пусть state_count — количество состояний FA1.
4. Пусть symbol_count — количество символов FA1.
5. Просматриваем состояния FA1, текущее состояние state.
5.1. Просматриваем символы FA1, текущий символ symbol.
5.1.1. Пусть S — множество состояний перехода из state по symbol.
5.1.2. Пусть trans_count — количество состояний перехода S.
5.1.3. Если trans_count > 1, то:
5.1.3.1. Сформируем комбинированное состояние из идентификаторов символов перехода в S и зарегистрируем его в FA1 как complex.
5.1.3.2. Удалим переходы из state по symbol.
5.1.3.3. Добавим переход в complex из state по symbol.
5.1.3.4. Просматриваем состояния перехода в S.
5.1.3.4.1. Пусть state_to — текущее состояние S.
5.1.3.4.2. Просматриваем символы FA1, текущий символ symbol_to.
5.1.3.4.2.1. Пусть T — множество состояний перехода из состояния state_to по символу symbol_to.
5.1.3.4.2.2. Пусть count_to — количество состояний перехода T.
5.1.3.4.2.3. Просматриваем состояния перехода T.
5.1.3.4.2.3.1. Добавляем в FA1 переход из complex по symbol_to в T_i.
5.1.3.4.3. Если состояние state_to финальное, зарегистрируем состояние complex как финальное.
5.1.3.5. Установим признак changed равным единице.
5.1.3.6. Завершим просмотр символов symbol.
6. Если changed равно нулю, то завершение алгоритма, вернем 1.
7. Перейдем к 2.

Заметим, что алгоритм в целом есть бесконечный цикл, начинающийся с установки changed в значение 0, и заканчивающийся проверкой значения этой переменной, и завершения цикла, если changed равно нулю.

Внутри бесконечного цикла в цикл по состояниям вложен цикл по символам, в рамках которых ищутся комбинированные состояния. Если такое состояние обнаружено, оно полностью обрабатывается.

Рассмотрим, как все происходит на примере входного автомата:

```
<H>
(H,/)={C}
(C,*)={A}
(A,/)={A}
(A,*)={A}
(A,*)={B}
(A,x)={A}
(B,/)={S}
{S} .
```

При первом проходе по состоянию A по символу * обнаруживается два перехода: (A,*)={A,B}. Эти два перехода заменяются на один (A,*)={AB}, при этом в автомат добавляется новое состояние AB, и переходы из состояния AB: (AB,/)={A}, (AB,/)={S}, (AB,*)={AB}, (AB,x)={A}:

```
<H>
(H,/)={C}
(C,*)={A}
(A,/)={A}
(A,*)={AB}
(A,x)={A}
(B,/)={S}
(AB,/)={A}
(AB,/)={S}
(AB,*)={AB}
(AB,x)={A}
{S} .
```

При втором проходе по состоянию AB по символу / обнаруживается два перехода: (AB,/)={A,S}. Они заменяются на переход (AB,/)={AS}, и добавляются переходы из AS: (AS,/)={A}, (AS,*)={AB}, (AS,x)={A}:

```
<H>
(H,/)={C}
(C,*)={A}
(A,/)={A}
(A,*)={AB}
(A,x)={A}
(B,/)={S}
(AB,/)={AS}
(AB,*)={AB}
(AB,x)={A}
(AS,/)={A}
(AS,*)={AB}
(AS,x)={A}
{S,AS} .
```

Заметим, что добавляется новое финальное состояние AS.

После удаления недостижимых состояний состояния B и S удаляются, и финальное состояние S тоже:

```
<H>
(H,/)={C}
(C,*)={A}
(A,/)={A}
(A,*)={AB}
(A,x)={A}
(AB,/)={AS}
(AB,*)={AB}
(AB,x)={A}
(AS,/)={A}
(AS,*)={AB}
(AS,x)={A}
{AS} .
```

15.3. Конструирование функции

При конструировании функции важно разобраться с вложенностью циклов, которая достигает пяти (пять вложенных друг в друга циклов). Для каждого вложенного цикла нужно понять, что и зачем он перебирает.

Нужно также разобраться с множествами S и T, значения которых получают методом to_states_of().

Для формирования комбинированного состояния complex в библиотеке предусмотрена вспомогательная функция fa_complex_register().

Она формирует комбинированное состояние и регистрирует его, используя множество S, которое упорядочивается.

15.4. Вопросы и упражнения

1. Является ли алгоритм корректным?

Попробуйте привести к детерминированному виду автомат:

```
<H>
(H,a)={A}
(H,a)={B}
(A,b)={C}
(A,c)={B}
(B,b)={D}
(B,c)={A}
{C,D}
```

Объясните, почему этот автомат вызывает зацикливание.

Предложите решение проблемы.

2. Зачем упорядочивается множество S в функции fa_complex_register()?

16. Минимизация конечного автомата

Цели:
- изучение алгоритма минимизации конечного автомата.
Задачи:
- конструирование алгоритма минимизации конечного автомата.
Алгоритм приведен в учебно-методическом пособии [1].

16.1. Рабочее пространство

Рабочим модулем является minfa.cpp.
Сейчас в нем определены две функции:

```
// минимизирует конечный автомат
int minimize_dfa(UFA FA1, UFA & FA2) {
    return 1;
}

// точка входа в алгоритм
// минимизация КА
int algorithm_min_dfa(UFA & FA1, FILE * target) {
    // результирующий КА
    UFA FA2;
    // алгоритм
    int result = minimize_dfa(FA1, FA2);
    // анализ КА для последующих алгоритмов
    FA1.analyse();
    // выводим КА на терминал
    FA1.print_out(stdout);
    // выводим свойства КА на терминал
    FA1.print_props(stdout);
    // выводим КА в файл
    FA1.print_out(target);
    return result;
}
```

Реализация алгоритма выполняется в функции minimize_dfa().
Необходимо убедиться, что в свойствах проекта установлены правильные параметры командной строки:

f02.sxa -faus -minfa

Необходимо также убедиться, что управление программой приходит в модуль minfa.cpp, и в функцию minimize_dfa(). Заметим, что в исходном автомате сначала удаляются недостижимые символы.

16.2. Проектирование алгоритма

Минимизировать выделением классов эквивалентности можно только детерминированный конечный автомат. Изначально полагается два класса эквивалентности: все финальные и все не финальные состояния. Эти классы измельчается до тех пор, пока это возможно.

Литература

1. Пономарев В.В. Конспективное изложение теории языков программирования и методов трансляции. Учебно-методическое пособие. В двух книгах. Книга 1. Озерск: ОТИ НИЯУ МИФИ, 2016. — 202 с., ил.

2. Вл. Пономарев. ТПМ. Требования к программным модулям. Методические указания. Озерск: ОТИ МИФИ, 2006. — 40 с.