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

Кафедра прикладной математики

Вл. Пономарев

Практикум 1

по теории языков программирования и методам трансляции

Учебно-методическое пособие

Часть 2. Преобразования конечных автоматов

Озерск, 2018

УДК 681.3.06
П 56

Вл. Пономарев. Практикум 1 по теории языков программирования и методам трансляции. Учебно-методическое пособие. Часть 2. Преобразования конечных автоматов. Озерск: ОТИ НИЯУ МИФИ, 2018. — 35 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 преобразований грамматик и конечных автоматов.

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

Рецензенты:
1.
2.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Общие цели занятий

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

1) Удаление недостижимых состояний конечного автомата.

2) Преобразование конечного автомата в автоматную грамматику.

3) Преобразование автоматной грамматики в конечный автомат.

4**) Построение детерминированного конечного автомата.

5) Минимизация конечного автомата методом Бржозовского.

6**⁻) Минимизация конечного автомата.

7**) Построение конечного автомата по регулярному выражению.

8**⁻) Построение регулярного выражения по конечному автомату.

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

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

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

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

Отчет по работе начинается с заголовка:

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

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

1. Работа PL-210. Устранение недостижимых состояний автомата

Цели:
- изучение двоичного представления конечного автомата;
- изучение алгоритма устранения недостижимых состояний.
Задачи:
- конструирование алгоритма устранения недостижимых состояний.

Опорные документы:
[2, с.140]

1.1. Описание конечного автомата в формате SYNAX

Конечный автомат описывается в текстовом файле, имеющем кодировку Windows-1251. В качестве примера рассмотрим описание:

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

В первой строке указывается множество идентификаторов начальных состояний в фигурных скобках.

Далее идут строки, описывающие переходы. В круглых скобках описываются параметры перехода, затем следует равенства, затем множество состояний, которое возвращает переход. Параметрами перехода являются идентификатор состояний, за которым следует запятая и символ перехода. Символ перехода может отсутствовать, в этом случае переход является пустым.

В последней строке записывается множество финальных (допускающих) состояний в фигурных скобках.

Множество состояний, возвращаемое переходом, может содержать несколько идентификаторов состояний или один. Для недетерминированного автомата отдельные состояния могут быть записаны как в одном множестве, в одной строке, так и в разных строчках.

Например, тот же автомат может быть записан следующим образом:

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

Здесь переход из состояния B по символу *a* записан в двух строках, а в предыдущем варианте была использована одна строка. Оба варианта записи НКА являются равнозначными. При выводе описания автомата библиотека использует вторую форму.

1.2. Рабочее пространство

Установим на диск C: проект grom, полученный при выполнении предыдущих работ. Работа выполняется в модуле 10-faus.cpp.

Откроем его. Модуль содержит следующие функции:

```
// общие действия с автоматом
void fa_general(NFA & fa) {
}
// удаление недостижимых состояний конечного автомата
void fa_remove_unreachable(NFA & fa1, NFA & fa2) {
}
// точка входа в алгоритм
// устранение недостижимых состояний КА
int algorithm_fa_remove_unreachable(NFA & fa1, FILE * target) {
    // выходной КА
    NFA fa2;
    // алгоритм
    fa_remove_unreachable(fa1, fa2);
    // выводим КА в консоль
    fa2.printi(stdout);
    // выводим КА в файл
    fa2.printi(target);
    // результирующий КА
    fa1 = fa2;
    // результат
    return 1;
}
```

Откроем FAR. Перейдем в каталог C:\grom\Debug. Нажмем Shift+F4, введем название файла a13b.sxg, введем текст автомата, приведенный выше в разделе 1.1. Сохраним и закроем файл.

Установим параметры командной строки "a13b -faus".

Установим точку остановки на строке return 1.

Запустим программу F5. Если управление не приходит в точку остановки, проверим параметры командной строки и наличие файла a13b.sxg.

Запустим программу Ctrl+F5. Вывод программы следующий:

```
-- parse automation
{A}
(A,a)={A}
(A,b)={B}
(B,a)={A,S}
{S}
.NFA
-- fa remove unreachable
{}
{}
.DFA
```

Вывод показывает, что описание автомата успешно разобрано, автомат является недетерминированным. Выходной автомат не сформирован.

1.3. Практика формирования автомата

Сначала рассмотрим двоичное представление НКА.

Автомат описывается классом NFA, переход — классом TRAN. Автомат в двоичном представлении состоит из массива переходов delta, массива идентификаторов состояний ident, множества начальных состояний initials, множества финальных состояний finals.

Переход состоит из четырех элементов:

SYMB a — состояние, из которого выполняется переход,

char c — символ, по которому выполняется переход,

SYMB b — состояние, в которое выполняется переход,

char f — атрибут, имеет вспомогательное назначение.

Опишем вызов функции fa_general в функции fa_remove_unreachable, передавая в качестве параметра автомат fa2. В функции fa_general изучаем основы работы с двоичным представлением автомата.

Перейдем в функцию fa_general.

Будем описывать автомат, требующийся для реализации алгоритма устранения недостижимых состояний:

```
{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}
(G,1)={F}
{D,E}
```

Регистрируем идентификаторы состояний от A до G:

```
void fa_general(NFA & fa) {
    // регистрируем состояния
    SYMB A = fa.reg_state_id("A");
    . . .
    SYMB A = fa.reg_state_id("G");
}
```

Затем добавляем переходы, для примера переход (A,0)={B}:

```
TRAN tran;
tran.a = A;
tran.c = '0';
tran.b = B;
fa.tran_add(tran);
```

Другие переходы, всего 11, добавим самостоятельно.

После добавления всех переходов формируем множества начальных и финальных состояний:

```
fa.initials.insert(A);
fa.finals.insert(D);
fa.finals.insert(E);
```

Запускаем программу, убеждаемся, что автомат сформирован.

Откроем FAR. Перейдем в каталог C:\grom\Debug. Установим курсор на файл a13b-faus.sxg, переименуем его в a10.sxg клавишами Shift+F6.

На этом считаем знакомство с двоичным представлением автомата законченным, другие методы изучим в ходе разработки алгоритмов.

Вызов функции fa_general заключим в комментарий.

Установим параметры командной строки "a10 -faus".

1.4. Конструирование алгоритма

Состояние автомата является недостижимым, если оно не встречается в правой части ни одного перехода.

Алгоритм в целом состоит из следующих шагов.

Шаг 1. Построение множества достижимых состояний R

Пусть R — множество.

Базис. Если A — начальное состояние, то A∈R.

Индукция. Если есть переход (A, a)={B}, и A∈R, то B∈R.

Шаг 2. Поиск действительных переходов.

Если есть переход (A, a)={B}, для которого A∈R и B∈R, то этот переход включаем в результирующий автомат.

Шаг 3. Формирование начальных состояний.

Если A — начальное состояние исходного автомата и A∈R, то включаем A во множество начальных состояний результирующего автомата.

Шаг 4. Формирование финальных состояний.

Если A — финальное состояние исходного автомата и A∈R, то включаем A во множество финальных состояний результирующего автомата.

1.5. Конструирование функции

Сначала присваиваем множеству R множество fa1.initials. Затем регистрируем каждый символ множества R как начальный символ автомата fa2 при помощи метода reg_initial_from.

Просматриваем переходы автомата. Если символ *a* перехода входит в R, включаем в R символ *b* перехода. Цикл просмотра переходов повторяем до тех пор, пока множество R пополняется.

Просматриваем множество финальных состояний автомата fa1. Если финальное состояние входит в R, регистрируем его как финальное состояние автомата fa2 при помощи метода reg_final_from.

Просматриваем переходы автомата fa1. Если символы *a* и *b* перехода входят в R, добавляем переход в автомат fa2.

Переходим к тестированию.

Сначала R = {A}. В первой итерации в R добавляются символы B и C. Во второй итерации в R добавляются символы D и E. В третьей итерации множество не изменяется.

В автомат fa2 добавляются первые восемь переходов, все финальные состояния fa1 включаются в fa2.

1.6. Контрольные вопросы и упражнения

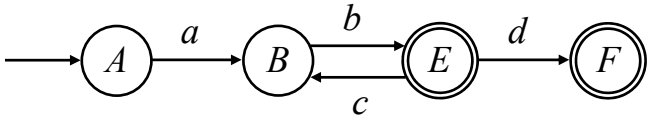
1. Приведите формальное описание конечного автомата.
2. Опишите синтаксис описания конечного автомата.
3. Опишите двоичное представление конечного автомата.
4. Нарисуйте в отчете граф переходов автомата a10-faus.sxg.
5. Какое состояние автомата является недостижимым?
6. Опишите алгоритм вычисления множества R.
7. Докажите корректность алгоритма построения множества R.

2. Работа PL-211. Преобразование автомата в грамматику

Цели:
- изучение алгоритма преобразования автомата в грамматику.
Задачи:
- конструирование алгоритма преобразования автомата в грамматику.
Опорные документы:
[1, «Конечные автоматы и регулярные грамматики»]

2.1. Изучение алгоритма

Пусть есть следующий автомат:
M(Q={*A*, *B*, *E*, *F*}, Σ={*a*, *b*, *c*, *d*}, δ, I={*A*}, F={*E*, *F*}), δ = {
δ(*A*, *a*) = {*B*}
δ(*B*, *b*) = {*E*}
δ(*E*, *d*) = {*F*}
δ(*E*, *c*) = {*B*}
}
Граф переходов этого автомата имеет следующий вид:



Просматриваем переходы автомата.
- если есть переход (*A*, *a*)={*B*}, и *B*∉F, формируем правило *A*→*aB*.
- если есть переход (*A*, *a*)={*B*}, и *B*∈F, формируем правило *A*→*a*. Если при этом есть переход (*B*, *b*)={*C*}, то формируем также правило *A*→*aB*.

Если у автомата одно начальное состояние, то оно преобразуется в целевой символ. Если у автомата несколько начальных состояний, то для каждого правила *A*→α, *A*∈I, формируем правило *S′*→α, после чего удаляем недостижимые состояния, а целевым символом становится *S′*.

В целях упрощения алгоритма будем считать, что у исходного автомата есть только одно начальное состояние.

Рассматривая переходы, получаем следующие правила грамматики:
- переход δ(*A*, *a*) = {*B*} приводит к правилу *A*→*aB*.
- переход δ(*B*, *b*) = {*E*} приводит к правилам *B*→*b*, *B*→*bE*.
- переход δ(*E*, *d*) = {*F*} приводит к правилу *E*→*d*.
- переход δ(*E*, *c*) = {*B*} приводит к правилу *E*→*cB*.

Результирующая грамматика:

A→ *aB*
B→ *b* | *bE*
E→ *d* | *cB*

Целевым символом является *A*.

2.2. Рабочее пространство

Установим на диск C: проект grom, полученный при выполнении предыдущих работ. Откроем FAR. Перейдем в каталог C:\grom\Debug. Нажмем Shift+F4, введем название файла a11.sxg, введем текст автомата:

```
{A}
(A,a)={B}
(B,b)={E}
(E,d)={F}
(E,c)={B}
{E,F}
•
```

Точка • показывает конец файла. Нажмем Escape, Enter.

Зададим аргументы командной строки: a11 -faau

Работа выполняется в модуле 11-faau.cpp. Откроем его.

В модуле определены функции:

```
// определяет наличие перехода из финального состояния
int has_tran_from_final(SYMB A, NFA & fa) {
    return 0;
}

// преобразует конечный автомат в грамматику
void fa_to_grau(NFA & fa, grammar & gr) {
}

// точка входа в алгоритм
// преобразование КА в автоматную грамматику
int algorithm_fa_to_grau(NFA & fa, grammar & gr, FILE * target) {
    // алгоритм
    fa_to_grau(fa, gr);
    // выводим грамматику в консоль
    gr.print(stdout);
    // выводим грамматику в файл
    gr.print(target);
    // результат
    return 1;
}
```

Установим точку остановки на операторе return 1.

Запустим программу F5. Если управление не приходит в точку остановки, проверим параметры командной строки и наличие файла a11.sxg.

2.3. Конструирование алгоритма частной функции

В проекте одна частная функция has_tran_from_final.

Ее назначение — определить, что из состояния A есть переход. Если переход есть, функция возвращает 1. Если все переходы просмотрены, и переход не найден, функция возвращает 0.

Мы принимаем, что цикл перебора переходов автомата всегда имеет следующий стандартный вид:

```
int has_tran_from_final(SYMB A, NFA & fa) {
    // количество переходов
    int delta_count = fa.delta_count();
    // просматриваем переходы
    for (int t = 1; t <= delta_count; t++) {
        // текущий переход
        transit tran = fa[t];
        . . .
    }
    return 0;
}
```

В этой функции остается написать одну строчку.

2.4. Конструирование общего алгоритма

Функция fa_to_grau. Просматриваем все переходы автомата.

Пусть tran — текущий переход, rule — новое правило.

Если состояние tran.b входит во множество финальных состояний, то формируем правило вида $A \to a$, где A — это символ tran.a, a — tran.c.

Для регистрации символа tran.a используем метод grammar::reg_from, для регистрации символа tran.c используем метод grammar::reg_char:

```
// правило A->a
rule[0] = gr.reg_from(fa, tran.a);
rule.append(gr.reg_char(tran.c));
```

После этого правило добавляем в грамматику методом rule_add.

Далее проверяем, есть ли переход из состояния tran.b при помощи частной функции has_tran_from_final. Если есть, то добавляем в правило символ tran.b, и полученное правило вида $A \to aB$ добавляем в грамматику.

На этом заканчивается ветка обработки перехода, в котором tran.b является финальным состоянием. Если tran.b не является финальным состоянием, то формируем правило вида $A \to aB$, и добавляем его в грамматику.

В конце функции формируем стартовый символ грамматики, регистрируя первый символ множества начальных символов автомата.

2.5. Контрольные вопросы и упражнения

1. Опишите правостороннюю автоматную грамматику.
2. Опишите алгоритм преобразования автомата в грамматику.
3. Докажите корректность изученного алгоритма.
4. Замените командную строку проекта следующей: a10 -faus -faau.

Запишите в отчет грамматику a10-faus-faau.sxg. Сравните ее с графом переходов автомата a10-faus.sxg.

3. Работа PL-212. Преобразование грамматики в автомат

Цели:
- изучение алгоритма преобразования грамматики в автомат.

Задачи:
- конструирование алгоритма преобразования грамматики в автомат.

Опорные документы:
[1, «Конечные автоматы и регулярные грамматики»]

3.1. Изучение алгоритма

Пусть есть следующая грамматика:

$A \rightarrow aB$
$B \rightarrow b \mid bE$
$E \rightarrow d \mid cB$

Эта праволинейная автоматная грамматика была получена в ходе выполнения предыдущей работы. Она преобразуется в конечный автомат следующим образом.

Просматриваем правила грамматики. Правило может иметь один из двух видов, $A \rightarrow a$, или $A \rightarrow aB$, и иметь длину один или два символа.

Для правила любого вида в грамматике может оказаться пара.

Если правило вида $A \rightarrow a$, парой является правило вида $A \rightarrow aB$.

Если правило вида $A \rightarrow aB$, парой является правило вида $A \rightarrow a$.

Если правило имеет пару, то символ B (правила или пары) соответствует финальному состоянию, и формируется переход $(A, a)=\{B\}$.

Пусть правило не имеет пары. Тогда:

- если длина правила равна единице, нужен дополнительный символ грамматики, предположим, f, который соответствует финальному состоянию автомата, и формируется переход $(A, a)=\{f\}$;
- если длина правила равна двум, то формируется переход $(A, a)=\{B\}$.

Выполняя преобразование, получим:

1) Для правила $A \rightarrow aB$ формируется переход $(A, a)=\{B\}$.

2) Для пары правил $B \rightarrow b \mid bT$ формируется переход $(B, b)=\{E\}$.

3) Для правила $E \rightarrow d$ формируется переход $(E, d)=\{C\}$.

При этом создается новый нетерминальный символ C.

4) Для пары правил $E \rightarrow cB$ формируется переход $(E, c)=\{B\}$.

При этом на шагах 2 и 3 формируются финальные состояния соответственно E и C.

Начальному состоянию автомата соответствует целевой символ грамматики, который является единственным. Данный алгоритм может формировать множество лишних финальных состояний, которые могут быть устранены минимизацией автомата.

3.2. Рабочее пространство

Установим на диск C: проект grom, полученный при выполнении предыдущих работ. Зададим командную строку проекта:

a11 -faau -aufa

Работа выполняется в модуле 12-aufa.cpp. Откроем его.

В модуле определены функции:

```
// находит пару A->a или A->aB, возвращает номер правила
int has_pair_rule(RULE rule, int length, grammar gr) {
    return 0;
}

// преобразует грамматику в конечный автомат
int grau_to_fa(grammar & gr, NFA & fa) {
    return 1;
}

// точка входа в алгоритм
// преобразование автоматной грамматики в конечный автомат
int algorithm_grau_to_fa(grammar & gr, NFA & fa, FILE * target) {
    // алгоритм
    int result = grau_to_fa(gr, fa);
    // выводим КА в консоль
    fa.print(stdout);
    // выводим КА в файл
    fa.print(target);
    // результат
    return result;
}
```

Установим точку остановки на операторе return 1.

Запустим программу F5. Если управление не приходит в точку остановки, проверим параметры командной строки и наличие файла a11.sxg.

3.3. Конструирование алгоритма частной функции

В проекте одна частная функция has_pair_rule. Ее назначение — определить, что для некоторого правила есть пара. Если пара есть, функция возвращает номер парного правила (на случай, если символ *B* отсутствует в исходном правиле). Если все правила просмотрены, и пара не найдена, функция возвращает 0.

Функция представляет собой стандартный цикл просмотра правил.

Пусть test — правило номер r. Тогда выполняем ряд проверок. Если нулевой символ test не совпадает с нулевым символом rule, то переходим к следующей итерации. Если первый символ test не совпадает с первым символом rule, то следующая итерация. Если длина правила test не совпадает с заданной длиной length, то следующая итерация. Если все проверки пройдены, возвращаем номер правила r.

3.4. Конструирование общего алгоритма

Функция grau_to_fa.

Ркгистрируем целевой символ грамматики как начальное состояние автомата. Затем просматриваем правила грамматики.

Пусть rule — текущее правило, tran — новый переход.

Пусть length — длина правила rule.

Если длина правила не равна 1 или 2, выводим в консоль сообщение «Grammar is wrong», завершаем функцию, возвращая нулевое значение.

Пусть pair — результат частной функции has_pair_rule. В эту частную функцию передаем правило rule, выражение `1 + (length % 2)`, грамматику.

Если длина правила равна единице и есть пара, формируем переход $(A, a)=\{B\}$. Регистрируем состояние tran.a методом NFA::reg_state_from, получаем символ tran.c методом grammar::get_char, регистрируем состояние tran.b методом NFA::reg_final_from, используя в качестве символа *B* второй символ правила pair.

Если длина правила равна единице и пары нет, формируем новый символ грамматики методом grammar::get_new_sym, и регистрируем его как финальный символ автомата методом NFA::reg_final_from. Пусть этот символ обозначен как F. Добавляем его во множество финальных состояний. Формируем переход, присваивая tran.b значение F.

Если длина правила равна двум и есть пара, формируем переход как в случае, когда длина равна единице, но состояние tran.b берем не из правила pair, а из правила rule.

Если длина правила равна двум и пары нет, формируем переход, в котором состояние tran.b регистрируется методом NFA::reg_state_from.

Если все манипуляции запрограммированы правильно, то программа сформирует автомат, который был исходным в преобразовании автомата в грамматику.

3.5. Контрольные вопросы и упражнения

1. Опишите алгоритм преобразования грамматики в автомат.
2. Докажите корректность изученного алгоритма.

4. Работа PL-213. Детерминизация конечного автомата

Цели:
- изучение алгоритма детерминизации конечного автомата.
Задачи:
- конструирование алгоритма именования состояний ДКА;
- конструирование алгоритма детерминизации КА.

Опорные документы:
[1, «Построение ДКА по НКА»], [2, с.138], [3, с.77, с.89]

4.1. Изучение алгоритма

Пусть есть недетерминированный конечный автомат:

```
{A}
(A,a)={A}
(A,a)={B}
(A,b)={A}
(B,b)={C}
```

Состояния ДКА равны множествам состояний НКА, в крайнем случае, состояние ДКА равно одному состоянию НКА.

Принимаем начальное состояние ДКА равным множеству начальных состояний НКА, и считаем это множество состояний достижимым. Далее будем искать другие достижимые множества состояний.

Пусть A — достижимое множество состояний НКА. Построим множество переходов T, определенных для состояний из A, а также множество символов C, по которым эти переходы могут быть сделаны. Для каждого символа $c \in C$ строим множество состояний B, в которые НКА переходит по переходам из T, и записываем переход ДКА (A, c) = B. Состояние A помечаем как исследованное.

Алгоритм в целом представляет собой рекурсивную процедуру, главным параметром которой является состояние ДКА, обозначенное как A. В процедуре вычисляются множества T и C, после чего исследуется каждый символ из множества C, который ведет к формированию множества B и перехода (A, c) = B. Если полученное состояние ДКА B является неисследованным, оно становится состоянием ДКА A, и процедура повторяется до тех пор, пока не будет исследован каждый символ C, и каждое неисследованное состояние B.

Для рассматриваемого автомата A_1 = {A}.
Помечаем A_1 как исследованное состояние ДКА.
Пусть переходы пронумерованы числами от 1 до 4.
T_1 = {1, 2, 3}, C_1 = {a_1, b_1}.
Рассматривая символ $\boldsymbol{a_1}$, получим B_1 = {A, B}.
Записываем переход ДКА (A_1, a) = B_1.

Повторяем процедуру для $A_2 = B_1 = \{A, B\}$.
Помечаем A_2 как исследованное состояние ДКА.
$T_2 = \{1, 2, 3, 4\}$, $C_2 = \{a_2, b_2\}$.
Рассматривая символ $\boldsymbol{a_2}$, получим $B_2 = \{A, B\} = B_1$.
Записываем переход ДКА $(A_2, a) = B_1$.
Рассматривая символ $\boldsymbol{b_2}$, получим $B_3 = \{A, C\}$.
Записываем переход ДКА $(A_2, b) = B_3$.
Повторяем процедуру для $A_3 = B_3 = \{A, C\}$.
Помечаем A_3 как исследованное состояние ДКА.
$T_3 = \{1, 2, 3\}$, $C_3 = \{a_3, b_3\}$.
Рассматривая символ $\boldsymbol{a_3}$, получим $B_4 = \{A, B\} = B_1$.
Записываем переход ДКА $(A_3, a) = B_1$.
Рассматривая символ $\boldsymbol{b_3}$, получим $B_5 = \{A\} = A_1$.
Записываем переход ДКА $(A_3, b) = A_1$.
Возвращаемся к A_1.
Рассматривая символ $\boldsymbol{b_1}$, получим $B_6 = \{A\} = A_1$.
Записываем переход ДКА $(A_1, b) = A_1$.
Все символы исследованы, процедура завершается. Каждому символу множества C_i соответствует ровно один переход ДКА.
Мы получили следующие переходы:
$(A_1, a) = B_1$
$(A_2, a) = B_1$
$(A_2, b) = B_3$
$(A_3, a) = B_1$
$(A_3, b) = A_1$
$(A_1, b) = A_1$
Учитывая, что множества B — это множества A следующей итерации алгоритма, эти переходы можно записать и так:
$(A_1, a) = A_2$
$(A_2, a) = A_2$
$(A_2, b) = A_3$
$(A_3, a) = A_2$
$(A_3, b) = A_1$
$(A_1, b) = A_1$
Здесь $A_1 = \{A\}$, $A_2 = \{A, B\}$, $A_3 = \{A, C\}$.

4.2. Рабочее пространство

Установим на диск C: проект grom.
Установим параметры командной строки "13a -dfa".
Откроем FAR. Перейдем в каталог C:\grom\Debug. Нажмем Shif+F4, введем название файла 13a.sxg, введем следующий текст, описывающий недетерминированный конечный автомат без λ-переходов:

```
{A}
(A,a)={A}
(A,a)={B}
(A,b)={A}
(B,b)={C}
{C}
•
```

Откроем проект.

Работа выполняется в модуле 13-dfa.cpp. Откроем его.

В модуле определены следующие функции и переменные:

```
// исследованные состояния ДКА
SSET inspected;

// вычисляет замыкание A состояния НКА state
void closure(SYMB state, SSET & A, NFA & fa1) {
}

// вычисляет замыкание A состояния ДКА A
void closureA(SSET & A, NFA & fa1) {
}

// вычисляет 2^(n-1)
int power2n(int n) {
}

// вычисляет идентификатор состояния ДКА
int reg_dfa_state(SSET & A, NFA & fa1, NFA & fa2) {
    return 0;
}

// ищет другие состояния
int follow_dfa_state(SSET A, NFA fa1, NFA & fa2, int initial = 0) {
    return 1;
}

// преобразует НКА в ДКА
int nfa_to_dfa(NFA & fa1, NFA & fa2) {
    inspected.clear();
    return follow_dfa_state(fa1.initials, fa1, fa2, 1);
}

// точка входа в алгоритм
// преобразование НКА в ДКА
int algorithm_dfa(NFA & fa1, FILE * target) {
    // выходной КА
    NFA fa2;
    // алгоритм
    int result = nfa_to_dfa(fa1, fa2);
    // выводим КА в консоль
    fa2.printn(stdout);
    // выводим КА в файл
    fa2.printn(target);
    // выходной КА
    fa1 = fa2;
    // результат
    return result;
}
```

Переменная inspected — множество исследованных состояний ДКА.

В начале алгоритма это множество очищается для многократного использования алгоритма в одном запуске программы.

Основные действия разворачиваются в функции follow_dfa_state.

На вход этой функции подается множество *A* состояний НКА, формирующих состояние ДКА. В первой итерации множество *A* соответствует множеству начальных состояний НКА:

```
int nfa_to_dfa(NFA & fa1, NFA & fa2) {
    inspected.clear();
    return follow_dfa_state(fa1.initials, fa1, fa2, 1);
}
```

Последний параметр этой функции указывает, что состояние является начальным, это нужно для регистрации состояния ДКА как начального.

Эта функция рекурсивная. После определения множества состояний *B*, в которые есть переходы из состояния *A*, функция рекурсивно вызывается, и множество *B* становится множеством *A* следующей итерации.

4.3. Идентификатор состояния ДКА

Состояние ДКА является множеством состояний НКА. Множества *A* и *B* представляют эти состояния. Нам нужен механизм, который формировал бы один и тот же идентификатор состояния ДКА при одном и том же множестве состояний НКА. Такой механизм является главной проблемой детерминизации. Я предлагаю следующее простое решение.

Каждое состояние НКА имеет свой порядковый номер. Тогда, если число состояний НКА не превышает 31, можно использовать номер состояния как номер бита в числе, обозначающем состояние. Например, если множество состояний ДКА равно {1, 2}, то состояние 1 устанавливает бит 0, а состояние 2 устанавливает бит 1, в результате получаем число 3. Если в следующий раз обнаружится состояние ДКА, равное множеству {2, 1}, получим то же самое число 3, то есть, то же самое состояние.

Для вычисления веса бита, соответствующего номеру состояния НКА, предназначена функция power2n, которую определим в первую очередь:

```
int power2n(int n) {
    if (n > 31) throw "power2n: too many states found";
    int m = 1;
    for (int i = 1; i < n; i++) m <<= 1;
    return m;
}
```

Если мы сформируем число, равное сумме весов битов состояний НКА, то можно использовать это число как идентификатор состояния. В этом случае класс NFA сможет определить, что состояние существует, и вернуть номер этого состояния, или, в противном случае, записать новое состояние при регистрации идентификатора.

Для этой цели в модуле определена функция reg_dfa_state. На вход функции поступает множество состояний НКА, функция формирует биты числа этого состояния, регистрирует его, а если во множестве *A* есть финальное состояние НКА, то регистрирует это состояние ДКА как финальное.

Сначала в функции нужно описать буфер buf типа char размером 16 для преобразования вычисленного числа состояния в строку.

Далее нам потребуется переменная number типа int для числа состояния, признак финального состояния, переменная final типа int, а также переменная regstate типа SYMB или int для номера зарегистрированного состояния. Начальные значения всех переменных равны нулю.

Далее формируем цикл, в котором просматриваем состояния множества *A*. Пусть s типа int — текущее состояние из множества *A* в некоторой итерации. Тогда:

1) проверяем, является ли s финальным состоянием НКА, и если является, то устанавливаем признак final;

2) прибавляем к значению числа number значение, которое вычислит функция power2n от значения s.

По завершении цикла формируем строковое значение числа number при помощи функции sprintf, и регистрируем полученный идентификатор в результирующем автомате при помощи метода NFA::reg_state_id, результат регистрации получаем в переменную regstate.

Если признак final установлен, дополнительно включаем состояние во множество финальных состояний.

Функция возвращает индекс состояния ДКА regstate.

4.4. Основной алгоритм

Переходим к формированию функции follow_dfa_state.

Нам потребуются множества *T* и *C*, описываем их. Для зарегистрированных номеров состояний ДКА *A* и *B* описываем две переменные, state_a и state_b соответственно, типа int. Начальные значения переменных равны нулю. Переменная state_a — это состояние будущего перехода ДКА TRAN::a, state_b — это состояние будущего перехода ДКА TRAN::b.

Регистрируем состояние ДКА *A* при помощи функции reg_dfa_state, значение принимаем в переменную state_a.

Проверяем признак initial. Если он установлен, состояние state_a регистрируем как начальное состояние ДКА.

Включаем состояние state_a в исследованные состояния inspected.

Следующий этап — формирование множеств *T* и *C*. Напоминаю, что множество *T* представляет номера переходов, достижимых из *A*, а множество *C* представляет символы, по которым есть переходы из *A*, соответствующие множеству *T*.

Формируем цикл по состояниям множества *A*.

Пусть state типа SYMB — текущее состояние из *A*.

Формируем стандартный цикл по переходам исходного автомата, переменная цикла *t*.

Пусть переменная tran типа TRAN — это текущий переход fa1[*t*].

Если символ перехода tran.a равен состоянию state, то:

1) включаем номер перехода *t* во множество *T*;

2) включаем символ перехода tran.c во множество *C*.

Этот фрагмент алгоритма следует протестировать. Установим точку остановки на операторе return функции follow_dfa_state.

Запустим программу F5.

Убедимся, что *A* = {1}, *T* = {1, 2, 3}, *C* = {97, 98}.

Остановим программу Shift+F5.

Анализируем результат.

Во множество *T* включены первые три перехода:

```
(A,a)={A}
(A,a)={B}
(A,b)={A}
```

Во множество *C* включены символы *a* (код 97) и *b* (код 98).

Остановим выполнение Shift+F5.

Следующий этап — формирование переходов из состояния ДКА *A* в состояния ДКА *B* по всем возможным переходам из *T* и символам из *C*.

Формируем цикл по символам из *C*, переменная цикла *c*.

Этот цикл последний в функции, за ним следует return 1.

Пусть symbol, переменная типа char — текущий символ (char)*C*[*c*].

Пусть *B* — множество.

Просматриваем все переходы множества *T*, переменная цикла *t*.

Пусть tran типа TRAN — текущий переход fa1[*T*[*t*]].

Если текущий символ symbol совпадает с символом перехода tran.c, то включаем во множество *B* состояние перехода tran.b.

По завершении цикла просмотра переходов *T* проверяем размер множества *B*, и, если оно равно нулю, переходим к следующей итерации цикла просмотра символов *C* при помощи оператора continue. Эта проверка является лишней, так как если есть символ перехода symbol, то есть и сам переход tran.symbol. Но это позволит нам установить здесь точку остановки.

Этот фрагмент алгоритма следует протестировать. Установим точку остановки на проверке множества *B*, запускаем программу F5.

Убедимся, что в первой итерации формируется множество *B* = {1, 2}, соответствующее символу *a*, во второй итерации формируется множество *B* = {1}, соответствующее символу *b*. Если это так, переходим к формированию переходов ДКА.

В первую очередь нужно получить номер state_b состояния ДКА *B* при помощи функции reg_dfa_state.

Пусть tran типа TRAN — формируемый переход. Тогда:

- tran.a присваиваем значение state_a,
- tran.b присваиваем значение state_b,
- tran.c присваиваем значение symbol,
- добавляем переход tran в автомат fa2 методом tran_add.

Тестируем этот фрагмент алгоритма.

Запускаем программу Ctrl+F5, убеждаемся, что в консоль выводится автомат из двух переходов:

```
{1}
(1,a)={2}
(1,b)={1}
{}
.DFA
.1=>1
.2=>3
```

Заметим, что для вывода автомата в библиотеке grammar предусмотрено две функции, printi и printn. Функция printi выводит идентификаторы состояний, а функция printn — номера состояний вместо идентификаторов. В данном случае используется функция printn. Эта функция в конце описания автомата дополнительно показывает идентификаторы.

Так, состояние номер 1 имеет идентификатор "1" (без кавычек), а состояние 2 имеет идентификатор "3", что соответствует состояниям 1 и 2 исходного автомата. Таким образом, {2} в выводе соответствует состоянию ДКА B_1, а {1} соответствует состоянию ДКА B_2.

Остается только выполнить рекурсивный вызов.

Перед этим проверяем, входит ли состояние state_b в исследованные состояния. Если входит, то переходим к следующей итерации цикла по символам множества C при помощи оператора continue.

Последняя строчка цикла — рекурсивный вызов, первый параметр B.

Если все действия описаны верно, получим следующий результат:

```
{1}
(1,a)={2}
(2,a)={2}
(2,b)={3}
(3,a)={2}
(3,b)={1}
(1,b)={1}
{3}
.DFA
.1=>1
.2=>3
.3=>5
```

Нарисуем в отчете граф переходов НКА 13a.sxg и ДКА 13a-dfa.sxg.

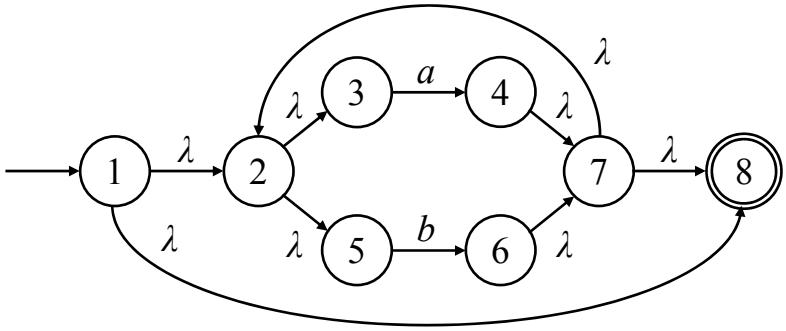
Сравним ДКА 13a-dfa.sxg с тем, который описан в пособии [1].

4.5. Детерминизация λ-НКА

Рассмотрим λ-НКА, полученный в результате реконструкции регулярного выражения (a+b)*:

```
{1}
(1, )={2,8}
(2, )={3,5}
(3,a)={4}
(4, )={7}
(7, )={2}
(5,b)={6}
(6, )={7}
(7, )={8}
{8}
```

Этому автомату соответствует следующий граф переходов:



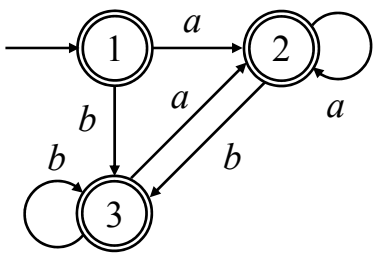
Этот автомат является симметричным. Если его инвертировать (поменять направления всех стрелок на противоположное, и поменять местами начальное и финальное состояния), получим точно такой же автомат.

Детерминизация λ-НКА требует вычисления λ-замыкания.

Рассмотрим замыкание состояния 1. Из этого состояния по пустым переходам доступны, кроме состояния 1, состояния 8, 2, 3, 5. Обозначая это состояние как 1, получим, что из состояния 1 ДКА по символу *a* перейдет в состояние НКА 4, а по символу *b* — в состояние НКА 6.

Вычисляя замыкание состояния НКА 4, получим {4, 7, 2, 8}. Это состояние обозначим как состояние ДКА 2. Замыкание состояния НКА равно {6, 7, 2, 8}, обозначим его как состояние ДКА 3.

Из состояния 2 ДКА переходит по символу *a* в состояние НКА 4, а по символу *b* — в состояние НКА 6. Из состояния 3 ДКА переходит по символу *a* в состояние НКА 4, а по символу *b* — в состояние НКА 6. Результирующий автомат имеет следующий вид:



Откроем FAR. Перейдем в каталог C:\grom\Debug. Нажмем Shift+F4, введем название файла a13e.sxg, введем текст автомата, приведенный в начале раздела. Сохраним F2 и закроем Escape файл. В настройках проекта установим параметры командной строки a13e -dfa.

4.6. Функции замыкания

В модуле определены функции closure и closureA. Первая функция вычисляет замыкание некоторого состояния state, вторая функция вычисляет замыкание всех состояний множества состояний *A*.

Рассмотрим, как вычисляется замыкание состояния state.

Просматриваем все переходы исходного автомата. Пусть tran — текущий переход. Если tran.a совпадает с состоянием state, а символ перехода tran.c равен нулю (пустой переход), то если символ tran.b отсутствует во множестве *A*, то добавляем состояние tran.b в множество *A*, и выполняем рекурсивный вызов closure для состояния tran.b.

Программируем этот порядок действий в функции closure.

Перейдем к функции closureA.

Она проще. Просматриваем все состояния множества *A* и для каждого состояния вызываем функцию closure для текущего состояния *A*.

Программируем этот порядок действий в функции closureA.

Остается только вызвать функцию замыкания closureA в функции reg_dfa_state, перед циклом вычисления числа состояния number.

Если все действия описаны верно, получим следующий результат:

```
{1}
(1,a)={2}
(2,a)={2}
(2,b)={3}
(3,a)={2}
(3,b)={3}
(1,b)={3}
{1,2,3}
.DFA
.1=>31
.2=>126
.3=>222
```

4.7. Контрольные вопросы и упражнения

1. Расшифруйте идентификаторы состояний ДКА 31, 126 и 222, запишите расшифровку в отчет, сопоставьте с состояниями НКА. Учтите, что в двоичном представлении НКА номера состояний другие.
2. Что называется конструкцией подмножеств?
3. Опишите изученный алгоритм детерминизации.
4. Чем изученный алгоритм отличается от конструкции подмножеств?
5. Опишите вычисление замыкания состояния.

5. Работа PL-214. Минимизация автомата по методу Бржозовского

Цели:
- изучение алгоритма Бржозовского.

Задачи:
- инвертирование автомата.

Опорные документы:
[1, «Минимизация конечных автоматов»], [2, с.147], [3, с.177], [4]

5.1. Алгоритм Бржозовского

Если есть алгоритм, выполняющий детерминизацию, то можно легко вычислить минимальный автомат, используя следующую последовательность действий: инверсия, детерминизация, инверсия, детерминизация.

Инверсия конечного автомата — это обращение его переходов на противоположные, и обмен начальных и финальных состояний.

Установим проект grom на диск C:. Установим аргументы командной строки a13e -fainv -dfa -fainv -dfa.

Откроем модуль 14-fainv.cpp.

Опишем алгоритм инвертирования автомата в функции fa_invert.

Это простой алгоритм, поэтому никаких пояснений не приводится.

После этого запускаем программу и получаем:

```
{1}
(1,a)={1}
(1,b)={1}
{1}
.DFA
.1=>7
```

5.2. Контрольные вопросы и упражнения

1. Выполните минимизацию автомата a13b.sxg.

2. Зарисуйте в отчет графы всех автоматов, исходных, промежуточных и результирующих, которые формируются в результате работы алгоритма при минимизации автоматов a13e.sxg и a13b.sxg.

3. Поясните алгоритм Бржозовского.

6. Работа PL-216. От регулярного выражения к конечному автомату

Цели:
- изучение алгоритма построения λ-НКА по регулярному выражению.

Задачи:
- грамматика для разбора регулярного выражения;
- синтаксический анализ выражения;
- построение примитивного автомата;
- операция «звездочка»;
- операция конкатенации автоматов;
- операция объединения автоматов.

Опорные документы:
[1, «От регулярного выражения к конечному автомату»], [3, с.120]

6.1. Синтаксический разбор выражения

Прежде, чем строить автомат, нужно разобрать регулярное выражение с тем, чтобы выделить операнды и операции. Для этой цели лучше всего подходит классическая грамматика для разбора арифметических выражений со скобками, приведенная, например, в [1, грамматика G1].

Заметим, что арифметическая операция сложения соответствует регулярной операции "+". Арифметическая операция умножения соответствует регулярной операции «конкатенация». Унарной регулярной операции "*" в грамматике G1 нет соответствия, но, вообще, она соответствует унарной операции "++" в языке Си. Тогда можно предложить следующую грамматику для разбора регулярного выражения:

$E \rightarrow E+T \mid T$
$T \rightarrow TU \mid U$
$U \rightarrow P \mid P*$
$P \rightarrow a \mid (E)$

Здесь терминал a обозначает букву регулярного выражения.

При помощи этой грамматики выделение первичных автоматов происходит в нетерминале P, выполнение операции «звездочка» происходит в нетерминале U (*unary*), конкатенация автоматов происходит в нетерминале T, а объединение автоматов происходит в нетерминале E.

Синтаксический анализатор этой грамматики построим методом рекурсивного спуска, который является наиболее простым. В этом методе каждому нетерминалу грамматики соответствует функция, разбирающая все правила нетерминала. Таким образом, нам нужны функции E, T, U и P, каждая из которых выполняет часть построения. Параметры функций — автоматы fa и part. Автомат fa требуется для добавления новых состояний, автомат part — это частично построенный результирующий автомат.

6.2. Рабочее пространство

Установим на диск C: проект grom. Установим аргументы командной строки r16 -rexfa -fainv -dfa -fainv -dfa.

Работа выполняется в модуле 16-rexfa.cpp. Откроем его.

В модуле описаны следующие переменные и функции:

```
// предварительное описание
int E(NFA & fa, NFA & part);
// ссылка на выражение
char * rx = 0;
// текущая точка в выражении
int index = 0;
// текущий символ выражения
char current() {
    return 0;
}
// следующий символ выражения
char getnext() {
    return 0;
}
// первичный автомат
void primary(NFA & fa, NFA & part, char c) {
}
// итерация
void iteration(NFA & fa, NFA & part) {
}
// конкатенация
void concat(NFA & fa, NFA & part, NFA & tail) {
}
// объединение
void unite(NFA & fa, NFA & part, NFA & tail) {
}
// элемент выражения
int P(NFA & fa, NFA & part) {
    return 0;
}
// унарная операция
int U(NFA & fa, NFA & part) {
    return 1;
}
// конкатенации
int T(NFA & fa, NFA & part) {
    return 1;
}
// объединения
int E(NFA & fa, NFA & part) {
    return 1;
}

// преобразует регулярное выражение в конечный автомат
int rex_to_fa(char * rex, NFA & fa) {
    return 0;
}
```

```
// точка входа в алгоритм
// преобразование выражения в автомат
int algorithm_rex_to_fa(char * rex, NFA & fa, FILE * target) {
    // алгоритм
    int result = rex_to_fa(rex, fa);
    // выводим КА в консоль
    fa.printi(stdout);
    // выводим КА в файл
    fa.printi(target);
    // результат
    return result;
}
```

На вход алгоритма поступает регулярное выражение в виде строки символов, заканчивающейся символом с кодом 10 (перевод строки). Длина входной строки ограничена константой MAX_INREX. Проект анализирует входную строку, разрешая только английские буквы, цифру 1, и 4 знака:

( ) + *.

Прописные буквы переводятся в нижний регистр.

Эти ограничения упрощают формирование алгоритмов. Отсутствующие знаки могут быть закодированы буквами латинского алфавита.

Откроем FAR. Перейдем в каталог C:\grom\Debug. Нажмем Shift+F4, введем название файла r16.sxg, введем одну букву *a* (буква латинская), закроем Escape и сохраним Enter файл.

6.3. Первичные автоматы

Наша первая цель — построить первичный автомат, принимающий одну-единственную букву. Для этого нужно дать первоначальное описание всех функций рекурсивного спуска, чтобы дойти до нетерминала *P*, и описать создание автомата в функции primary. Кроме этого, нужно описать функции, которые возвращают символы строки.

Описываем сначала вспомогательные функции:

```
// текущий символ выражения
char current() {
    return rx[index];
}
// следующий символ выражения
char getnext() {
    char c = rx[index];
    index += (c != 10);
    return c;
}
```

Таким образом, если достигнут конец выражения, функция getnext всегда будет возвращать символ с кодом 10.

Функция rex_to_fa устанавливает ссылку на строку регулярного выражения, создает результирующий автомат part, вызывает функцию разбора выражения *E*, и присваивает результат автомату fa:

```
int rex_to_fa(char * rex, NFA & fa) {
    // ссылка на выражение
    rx = rex;
    // результирующий автомат
    NFA part;
    // разбираем выражение
    int result = E(fa, part);
    // результат
    fa = part;
    return result;
}
```

В функциях синтаксического анализатора описываем необходимые переменные. В функциях *T* и *E* нужны частичные автоматы tail, представляющие вторые операнды бинарных операций, тип переменных NFA. Во всех функциях нужна переменная *c* типа char, и переменная result типа int.

Начальные значения переменных нулевые.

Чтобы понять, что должны делать функции, рассмотрим вывод:

$E \Rightarrow T \Rightarrow U \Rightarrow P \Rightarrow a$

Очевидно, что на данном этапе функции вызывают функции, которые в грамматике находятся ниже по тексту. Начальный код функции *E*:

```
int E(NFA & fa, NFA & part) {
    NFA tail;
    char c = 0;
    int result = T(fa, part);
    if (result != 1) {
        return result;
    }
    return 1;
}
```

Функции *U* и *T* имеют точно такой же вид, только в функции *U* нет переменной tail, она не нужна, так как операция функции унарная.

Функция *P* получает очередной знак и разбирает его:

```
int P(NFA & fa, NFA & part) {
    int result = 0;
    // очередной символ
    char c = getnext();
    if (c == 10) {
        // конец выражения
    } else if (c == '(') {
        // выражение в скобках
    } else if (c == '1') {
        // константа
    } else if (c > 122) {
        // недопустимый знак
    } else if (c > 96) {
        // переменная
    }
    return 0;
}
```

Если текущий знак 10, функция возвращает 10.

Если текущим знаком является открывающая скобка, рекурсивно вызываем функцию *E*, принимая результат в result. Результат проверяем, если не единица, возвращаем результат. Этот рекурсивный вызов, кстати, дал название методу разбора «рекурсивный спуск». По завершении этого разбора проверяем наличие закрывающей скобки. Если скобка не обнаружена, возвращаем ноль, если обнаружена, ее нужно принять при помощи getnext, после чего вернуть единицу.

Если текущий знак цифра 1, формируем первичный автомат при помощи функции primary, передавая последним параметром нулевое значение, после чего возвращаем единицу.

Если код текущего знака больше 122, он недопустим, возвращаем 0.

Если код текущего знака больше 96 (буква), формируем первичный автомат при помощи функции primary, передавая ей последним параметром текущий знак, после чего возвращаем единицу.

Теперь нам остается только сформировать первичный автомат. Для этой цели в модуле определена функция primary, переходим к ней.

Прежде всего, нужно понять назначение автомата fa.

Он нужен для того, чтобы каждое новое состояние, которое мы будем формировать в автоматах part и tail, получило идентификатор, не совпадающий ни с каким другим идентификатором. Для этой цели вызывается метод get_new_state автомата fa. Заметим, что метод генерирует идентификаторы, в точности равные одной латинской букве, поэтому максимальное количество состояний, которое можно сгенерировать, равно 26, после чего произойдет выброс текстового исключения «overflow».

Символ, зарегистрированный в автомате fa, нужно также зарегистрировать в автомате part или tail, при помощи метода reg_state_from того автомата, в котором регистрируется состояние. Формирование автоматов в основном заключается в создании пустых переходов-связок. Исключение составляет первичный автомат, который только и формирует один переход по символу, да и то в том случае, когда символ не цифра "1".

Сначала в функции primary опишем переменную tran типа TRAN.

Затем формируем начальное и финальное состояния:

```
SYMB A = part.reg_state_from(fa, fa.get_new_state());
SYMB B = part.reg_state_from(fa, fa.get_new_state());
```

Затем формируем переход из состояния A в состояние B по символу *c*.

Полученный переход добавляем в автомат part.

Наконец, добавляем в начальные состояния автомата part состояние A, а в финальные состояния — состояние B. И это весь алгоритм.

Запускаем программу F5, убеждаемся, что формируется автомат:

```
{A}
(A,a)={B}
{B}
```

Теперь нужно убедиться, что выражение в скобках распознается.

Открываем FAR. Переходим в каталог C:\grom\Debug.

Открываем файл r16.sxg при помощи Shift+F4, заменяем текст файла на выражение в скобках, например, ((a)), после чего закрываем файл при помощи Escape, сохраняем при помощи Enter.

Запускаем программу F5, убеждаемся, что результат не изменился.

Если все правильно работает, переходим к операции «звездочка».

6.4. Операция «звездочка»

Итерация формирует два новых состояния и четыре пустых перехода.

Функция iteration начинается примерно так же, как и primary. Отличие в том, что после создания перехода tran поле *c* принимается равным нулю, так как все переходы пустые. Затем регистрируются символы A и B.

Далее формируем переходы. Поскольку состояния, участвующие в переходах, повторяются, код можно сократить.

Первый переход из состояния A в состояние part.initials[1].

Второй переход из состояния A в состояние B. Поскольку состояние tran.a уже задано, повтрно его можно не задавать.

Третий переход из состояния part.finals[1] в состояние B.

Поскольку состояние tran.b уже задано, его можно не задавать.

Последний переход из состояния part.finals[1] в part.initials[1].

Поскольку состояние tran.a уже задано, его можно не задавать.

В конце функции обновляем начальное и финальное состояния.

Сначала эти множества автомата part нужно очистить. Затем в начальные состояния добавляем символ A, а в финальные — символ B.

Переходим к функции *U*, которая формирует итерацию.

После того, как функция получила частичный автомат part, она проверяет, какой символ следует дальше:

```
if (current() == STAR) {
    // пропускаем '*'
    c = getnext();
    iteration(fa, part);
}
```

Заметим, что здесь используется константа STAR, определенная в модуле grammar.h следующим образом:

```
// символ операции *
#define STAR '*'
```

Кроме того, функция пропускает все другие звездочки, которые могут следовать за первой, так как эти звездочки не изменяют выражение:

```
while (current() == STAR) {
    // пропускаем '*'
    c = getnext();
}
```

Для тестирования изменяем файл r16.sxg так, как описано выше, заменяя текст выражения на (a)*. Запускаем программу Ctrl+F5, убеждаемся, что формируется следующий автомат:

```
{C}
(A,a)={B}
(C, )={A}
(C, )={D}
(B, )={D}
(B, )={A}
{D}
```

Детерминизация и минимизация этого автомата приводит к ДКА:

```
{1}
(1,a)={1}
{1}
```

Если все получилось, переходим к конкатенации автоматов.

6.5. Конкатенация автоматов

Рассмотрим вывод цепочки *aa*:

$E \Rightarrow \boldsymbol{T} \Rightarrow TU \Rightarrow \boldsymbol{UU} \Rightarrow PU \Rightarrow aU \Rightarrow aP \Rightarrow aa$

Из него следует, что нетерминал *T* есть последовательность частичных автоматов, возможно, итерированных нетерминалом *U*.

Следовательно, если после получения первого частичного автомата в нетерминале *T* следующий символ является переменной, нужно получить следующий частичный автомат, и присоединить его к предыдущему, то есть выполнить конкатенацию автоматов part и tail. Последовательность конкатенаций может быть сколь угодно длинной.

Сначала опишем конкатенацию автоматов в функции concat.

Объявляем переменную tran и переменную *i* для цикла. Полю tran.c присваиваем нулевое значение, так как единственный переход пустой.

Формируем переход от финального состояния автомата part к начальному состоянию автомата tail. Заметим, что начальное состояние автомата tail нужно зарегистрировать в автомате part при помощи reg_state_from.

Добавляем этот переход в автомат part.

Затем нужно заменить финальное состояние. Для этого очищаем множество финальных состояний автомата part, и добавляем в него зарегистрированное в автомате part финальное состояние автомата tail.

В конце функции копируем переходы автомата tail в автомат part при помощи метода tran_add_from.

Переходим к функции *T*.

Формируем цикл по условию «текущий символ больше 96 и меньше 123 или равен "("», располагая его перед завершающим оператором return:

```
while (current() > 96 && current() < 123 || current() == '(') {
}
```

В цикле:

- очищаем автомат tail,

- вызываем функцию *U*, передавая автоматы fa и tail,

- проверяем результат вызова, в случае неудачи возвращаем результат вызова,

- соединяем автоматы функцией concat.

Для тестирования используем выражение (ab)*.

При этом должен генерироваться автомат:

```
{E}
(A,a)={B}
(B, )={C}
(C,b)={D}
(E, )={A}
(E, )={F}
(D, )={F}
(D, )={A}
{F}
```

Детерминизация и минимизация этого автомата дают ДКА:

```
{1}
(1,a)={2}
(2,b)={1}
{1}
```

Если все получилось, остается выполнить объединение автоматов.

6.6. Объединение автоматов

Рассмотрим вывод цепочки *a*+*a*:

$\boldsymbol{E} \Rightarrow E+T \Rightarrow \boldsymbol{T}+\boldsymbol{T} \Rightarrow P+T \Rightarrow a+T \Rightarrow a+P \Rightarrow a+a$

Из него следует, что нетерминал *E* есть последовательность автоматов, возвращаемых нетерминалом *T*, соединенных знаком операции объединения. Следовательно, если после получения первого автомата part от нетерминала *T* следующий знак равен "+", то нужно пропустить знак "+", получить следующий автомат tail и выполнить объединение part и tail.

Описываем объединение в функции unite.

Объявляем переменную tran, и полю tran.c присваиваем нулевое значение, так как все переходы пустые. Объявляем переменную *i* для цикла.

Формируем символы A и B, как и раньше.

Формируем переходы.

Первый переход из состояния A в состояние part.initials[1].

Второй переход из состояния A в состояние tail.initials[1], которое нужно зарегистрировать в part методом reg_state_from.

Третий переход из состояния part.finals[1] в состояние B.

Четвертый переход из состояния tail.finals[1], которое нужно зарегистрировать в part методом reg_state_from, в состояние B.

Остается изменить начальное и финальное состояния. Очищаем множества начальных и финальных состояний, включаем состояние A во множество начальных состояний, а состояние B — во множество финальных.

Переходим к функции *E*.

Формируем цикл, выполняющий объединения, располагая его перед завершающим оператором return:

```
while (current() == '+') {
}
```

В цикле:

- получаем символ *c* при помощи функции getnext,
- очищаем автомат tail,
- вызываем функцию *T*, передавая автоматы fa и tail, результат получаем в переменную result,
- проверяем значение result, если оно не равно единице, то завершаем функцию, возвращая result,
- выполняем объединение автоматов part и tail функцией unite.

Для тестирования используем выражение (a+b)*.

При этом должен генерироваться автомат:

```
{G}
(A,a)={B}
(E, )={A}
(E, )={C}
(B, )={F}
(D, )={F}
(C,b)={D}
(G, )={E}
(G, )={H}
(F, )={H}
(F, )={E}
{H}
```

Детерминизация и минимизация этого автомата дают ДКА:

```
{1}
(1,a)={1}
(1,b)={1}
{1}
```

При помощи этого алгоритма в сочетании с алгоритмами детерминизации и минимизации можно проверять тождества регулярных выражений, которые кажутся невероятными, например, (a*b)*a* = (a+b)*.

6.7. Контрольные вопросы и упражнения

1. Сформируйте автомат по выражению (a*+b*)*.
2. Опишите, как выполняется итерация частичного автомата.
3. Опишите, как выполняется конкатенация частичных автоматов.
4. Опишите, как выполняется объединение частичных автоматов.

Литература

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

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

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

4. J. M. Champarnaud, A. Khorsi, T. Paranthoën. Split and join for minimizing: Brzozowski's algorithm. The Prague Stringology Conference, 2002.