СОЗДАНИЕ СИСТЕМЫ АВТОМАТИЧЕСКОГО ЗАВЕРШЕНИЯ ВВОДА С ИСПОЛЬЗОВАНИЕМ ПАКЕТА *UNIMOD*

В.С. Гуров, М.А. Мазин
Научный руководитель – д.т.н., проф. А.А. Шалыто

В работе описана технология разработки системы автоматического завершения ввода, предназначенная для удобного редактирования программных текстов. Используя грамматику, задающую язык, предлагается строить конечный автомат типа Мили. Для разработки и отладки конечных автоматов используется программный пакет с открытым кодом *UniMod* (http://unimod.sf.net).

Введение

В работе [1] предложен метод проектирования событийных объектно-ориентированных программ с явным выделением состояний, названный «*SWITCH*-технологией». Особенность этого подхода состоит в том, что поведение в таких программах описывается с помощью графов переходов структурных конечных автоматов с нотацией, предложенной в работе [2]. *SWITCH*-технология для описания каждого автомата определяет два типа диаграмм (схемы связей и графы переходов). При наличии нескольких автоматов, кроме того, строится схема их взаимодействия. *SWITCH*-технология задает нотацию и операционную семантику используемых диаграмм.

Программный пакет с открытым кодом *UniMod* (http://unimod.sf.net**),** созданный авторами статьи, обеспечивает разработку и выполнение автоматно-ориентированных программ. Пакет, сохраняя автоматный подход, позволяет использовать *UML*-нотацию при построении диаграмм в рамках *SWITCH*-технологии. При этом схемы связей, определяющие интерфейс автоматов, строятся в нотации диаграмм классов языка *UML*, а графы переходов — в *UML*-нотации диаграмм состояний. В состав пакета *UniMod* входит встраиваемый модуль (*plug-in*) для платформы *Eclipse* (http://www.eclipse.org**),** позволяющий создавать и редактировать *UML*-диаграммы классов и состояний, которые соответствуют схеме связей и графу переходов.

На сегодняшний день интегрированные системы для разработки программ предоставляют удобные средства для работы с кодом, такие как, например:

- подсветка семантических и синтаксических ошибок;
- автоматическое завершение ввода и автоматическое исправление ошибок;
- форматирование и рефакторинг [3] кода;
- запуск и отладка программы внутри среды разработки.

В английском языке эти средства получили название "*code assist*".

В рамках создания очередной версии пакета *UniMod* перед авторами встала задача реализации системы автоматического завершения ввода при редактировании условий на переходах на *UML*-диаграмме состояний. В статье описана технология создания такой системы и ее автоматно-ориентированная реализация, выполненная с помощью пакета *UniMod*.

Постановка задачи

Автоматическим завершением ввода, применительно к редактированию программных текстов, традиционно называют технологию, позволяющую пользователю получить список строк, при добавлении которых в текст после позиции курсора программа будет синтаксически верна. Например, на рис. 1 показано, как среда разработки *Eclipse* предлагает варианты автоматического завершения ввода для текущей позиции курсора.

Сформулируем требования к проектируемой системе автоматического завершения ввода. Пусть задан язык L и на вход системы подана строка α.

1. Если поданная на вход строка α является префиксом предложения языка L ($\exists\omega : \alpha\omega \in L$), то система должна возвращать множество строк $C(\alpha) = \{\beta_i\}_{i=1..n}$, любая из которых может являться продолжением данной α ($\forall i \in [1..n] \exists\gamma : \alpha\beta_i\gamma \in L$);
2. Если поданная на вход строка α не является префиксом предложения на заданном языке ($\neg\exists\omega : \alpha\omega \in L$), то система должна с помощью дополнения строки недостающими символами или с помощью удаления лишних символов трансформировать строку в правильный префикс предложения языка. Количество дополнений и удалений должно быть как можно меньше.

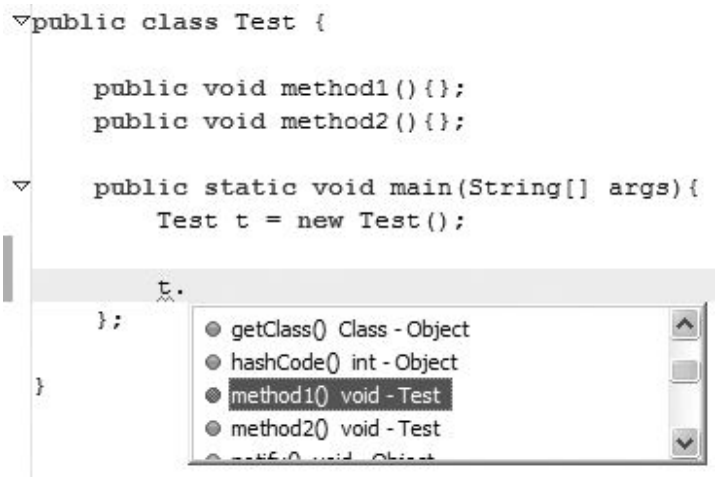

Рис. 1. Пример автоматического завершения ввода

Элементы теории построения компиляторов

Если исходный язык L задан формальной порождающей грамматикой, то очевидно, что для построения такой системы необходимо использовать методы проектирования компиляторов [5]. Существует множество инструментов для автоматического создания компиляторов по заданной грамматике. Достаточно большой список приведен, например, на сайте **http://www.kulichki.net/kit/tools/java.html**.

На рис. 2 приведена обобщенная структура компилятора. Лексический анализатор осуществляет чтение входной цепочки символов и их группировку в элементарные конструкции, называемые лексемами. Синтаксический анализатор осуществляет разбор исходной программы, используя поступающие лексемы. семантический анализ программы и построение промежуточного представления программы. Генератор кода преобразует промежуточное представление программы в объектный код. Генератор кода может быть заменен интерпретатором, при этом вместо генерации объектного кода будет выполняться интерпретация промежуточного представления программы.

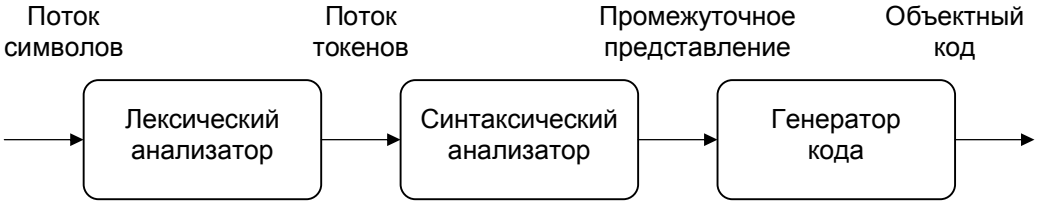

Рис. 2. Обобщенная структура компилятора

В проекте *UniMod* трансляция выражений на переходах выполняется с помощью, так называемого, «компилятора компиляторов» *ANTRL* [4]. Он по заданной *LL(k)* грам-

матике строит код на языке *Java*, реализующий лексический анализатор и рекурсивный нисходящий синтаксический анализатор. Построенный синтаксический анализатор может быть использован и как распознаватель принадлежности выражения заданному грамматикой языку, и как транслятор выражений в абстрактное синтаксическое дерево. Данный анализатор не может быть использован для построения системы автоматического завершения ввода, так как в случае подачи ему на вход префикса для выражения на заданном языке вместо законченного выражения, он выдает ошибку.

Одним из возможных вариантов реализации требуемой системы может быть использование нерекурсивного нисходящего синтаксического анализатора явно использующего стек и управляемого таблицей разбора. Таблица разбора представляет собой двумерный массив $M[A,a]$, где A – нетерминал, а a – терминал (лексема) или символ конца потока *$*. В ячейках таблицы записываются продукции грамматики, с помощью которых заменяются нетерминалы на вершине стека, пустые ячейки таблицы означают ошибки. Подробно работа такого анализатора описана в работе [5].

При подаче на вход описанному выше анализатору незавершенной строки α без символа конца потока, анализатор остановиться, имея какой-то нетерминал на вершине стека. В этом случае множество терминалов $C(\alpha)$, ожидаемых вслед за обработанной строкой, может быть определено как $\{\beta : M[T,\beta] \neq \varnothing\}$, где T - нетерминал на вершине стека после остановки анализатора.

Для реализации восстановления после ошибок в «режиме паники», таблица разбора может быть дополнена синхронизирующими символами, которые вписываются в некоторые пустые ячейки. При получении неожидаемого терминала, анализатор пропускает символы входного потока до тех пор, пока не будет обнаружен терминал, соответствующий синхронизирующему символу. Для восстановления на уровне фразы в некоторые пустые ячейки вписываются указатели на подпрограммы обработки ошибок, которые могут изменять, вставлять или удалять терминалы входного потока или элементы стека.

Описание предлагаемой технологии

В [7] показано как создать программу нерекурсивного нисходящего синтаксического анализатора, используя автоматно-ориентированный подход, при этом таблица разбора в работе [7] оставлена и выступает в роли объекта управления.

В настоящей работе предлагается технология создания системы автоматического завершения ввода, позволяющая исключить таблицу разбора нисходящего нерекурсивного синтаксического анализатора и использующая гибкий алгоритм восстановления после ошибок на уровне фразы.

Технология основывается на том, чтобы для заданной *LL(1)* грамматики построить конечный автомат типа Мили, который будет являться синтаксическим анализатором. Автомат должен реагировать на события, которые поставляет ему лексический анализатор. Каждому событию соответствует терминал. В работе [5, 12] приведено описание подхода для создания нисходящего синтаксического анализатора на основе диаграмм переходов. При этом предлагается записывать по одной диаграмме для каждого правила вывода грамматики. Описываемая в настоящей статье технология предлагает сворачивать все диаграммы в одну, при необходимости удаляя рекурсию с помощью метода, описанного в работе [6]. Такой подход позволяет избавиться от упоминания нетерминалов на диаграммах переходов и, следовательно, разорвать семантическую связь с исходной грамматикой. Такой разрыв позволит описывать язык только с помощью диаграммы переходов и автоматически получать реализацию распознавателя для данного языка.

При подаче на вход системе, построенной описанным выше образом, незавершенной строки, автомат, реализующий синтаксический анализатор, останавливается в каком-то состоянии. События, заданные на переходах из состояния, в котором остановился автомат, определяют множество терминалов, которые могут следовать за последним терминалом, извлеченным из входной строки. После построения такого множества терминалов, каждый терминал обратно преобразуется в строку символов.

Построение диаграммы переходов синтаксического анализатора

Пусть *LL(1)* грамматика для нашего примера задана следующим множеством правил вывода:

1) `S  -> else | T S'`
2) `S' -> or T S' | ε`
3) `T  -> L T'`
4) `T' -> and L T' | ε`
5) `L  -> not L | P`
6) `P  -> '(' S ')' | int rel N | bool | N P'`
7) `P' -> rel int | ε`
8) `N  -> id dot id`

Терминал **id** соответствует идентификатору, терминал **int** — целочисленной константе, терминал **bool** — булевской константе, а терминал **rel** — бинарному отношению (‘>’, ‘<’, ‘>=’, ‘<=’, ‘=’, ‘≠’). Опишем формальный процесс построения автомата типа Мили для данной грамматики.

На рис. 3 приведены диаграммы переходов для каждого нетерминала заданной грамматики, построенные с помощью метода, описанного в работе [5]. Единственным отличием указанных диаграмм, от диаграмм предлагаемых указанной работе, является наличие выделенных начального и конечного состояний, отображаемых закрашенным кругом и закрашенным кругом внутри окружности соответственно. Из начального состояния существует всегда только один переход, в конечное состояние также ведет только один переход.

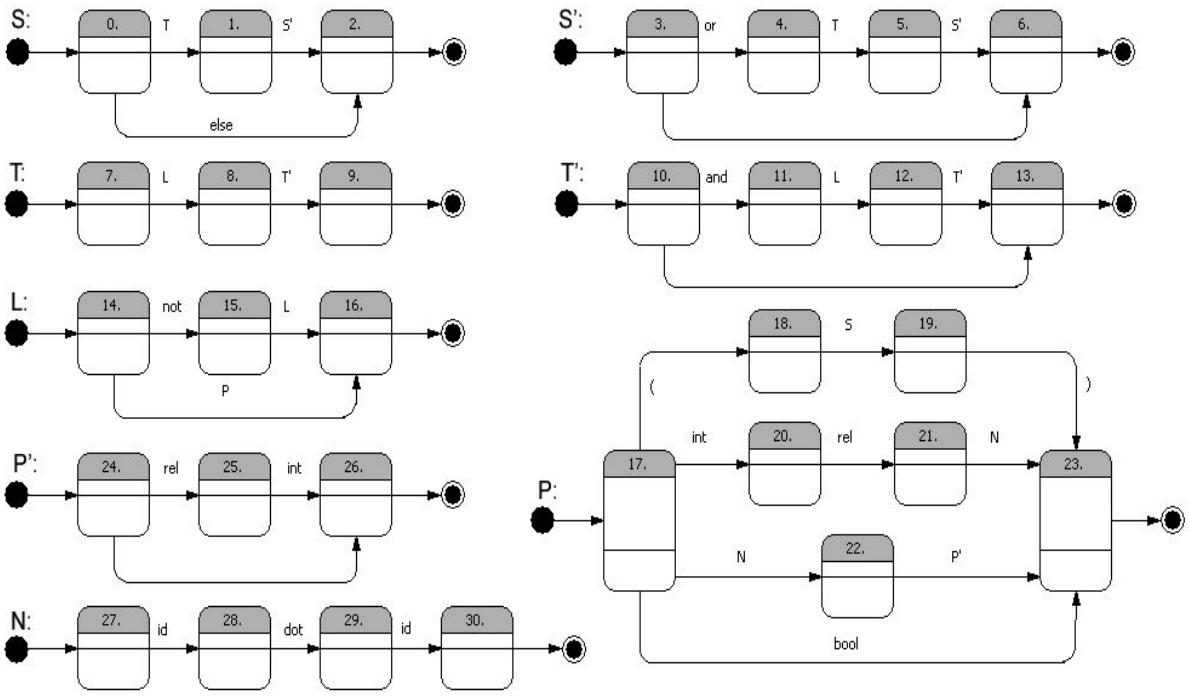

Рис. 3. Диаграммы переходов для каждого нетерминала грамматики

Состояния соответствуют позициям [8] в правилах вывода, метки на переходах — терминалам и нетерминалам, отделяющим позиции друг от друга. Если нетерминал выводит ε-правило, то из состояния, соответствующего начальной позиции, существует непомеченный переход в состояние, соответствующее конечной позиции. Непомеченные переходы также называются немотивированными.

Далее, множество диаграмм, представленных на рис. 3, можно преобразовать в одну диаграмму состояний, на которой все переходы будут помечены только терминалами. Процесс такого преобразования предполагает выполнение следующих шагов:

1. удаление правой рекурсии;
2. удаление немотивированных переходов;
3. подстановка диаграмм переходов друг в друга;
4. удаление срединной рекурсии;

Опишем каждый шаг подробно.

Удаление правой рекурсии

Наличие праворекурсивного правила вывода, означает, что на диаграмме, соответствующей некоторому терминалу *N*, есть переход, помеченный тем же нетерминалом *N*, ведущий в состояние, соответствующее конечной позиции.

Например, наличие праворекурсивного правила (2) влечет наличие перехода из состояния *5* в состояние *6* на рис. 3. Для устранения правой рекурсии этот переход должен быть заменен немотивированным переходом в состояние, соответствующее начальной позиции – в состояние *3* (рис. 4).

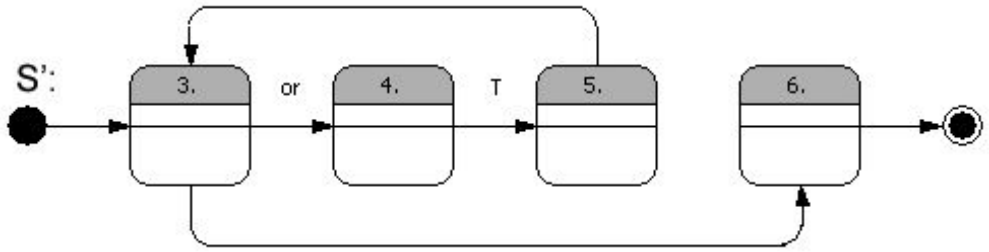

Рис. 4. Удаление правой рекурсии на диаграмме состояний для нетерминала S'

Удаление немотивированных переходов

Наличие немотивированного перехода из состояния S_1 в состояние S_2, означает, что за позицией, соответствующей состоянию S_1, могут следовать те же терминалы и нетерминалы, что и за позицией, соответствующей состоянию S_2.

Для устранения немотивированного перехода выполняются следующие операции:

1. создать сложное состояние $S_{1,2}$ [9];
2. поместить состояния S_1 и S_2 внутрь состояния $S_{1,2}$;
3. все переходы из состояния S_2 заменить аналогичными переходами из состояния $S_{1,2}$.

На рис. 5 показано удаление немотивированного перехода из состояния *5* в состояние *3*, присутствующего на рис. 4.

Будем говорить, что исходящий переход из состояния S_1 совпадает с исходящим переходом из состояния S_2, если он помечен тем же символом и ведет в тоже состояние.

После выделения группового состояния $S_{1,2}$ множества переходов, исходящих из состояний S_1 и S_2 могут совпасть. В этом случае из каждой пары совпадающих переходов следует оставить только один и его начало переместить в состояние $S_{1,2}$. Все переходы, входящие в состояния S_1 и S_2, перенаправить в состояние $S_{1,2}$. Состояния S_1 и S_2 ликвидировать.

Отметим, что данный алгоритм аналогичен вычеркиванию одинаковых записей из таблицы, задающей функцию переходов автомата [10].

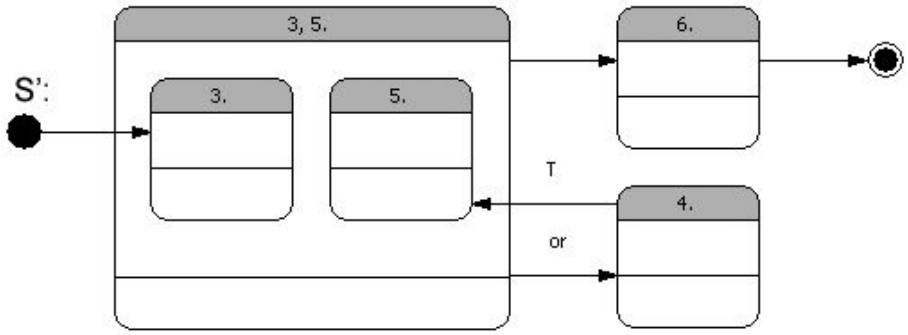

Рис. 5. Удаление немотивированного перехода из состояния 5 в состояние 3

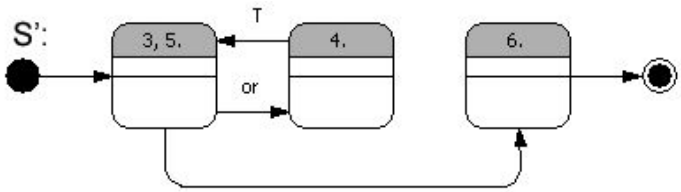

Рис. 6. Упрощение диаграммы состояний, показанной на рис. 5

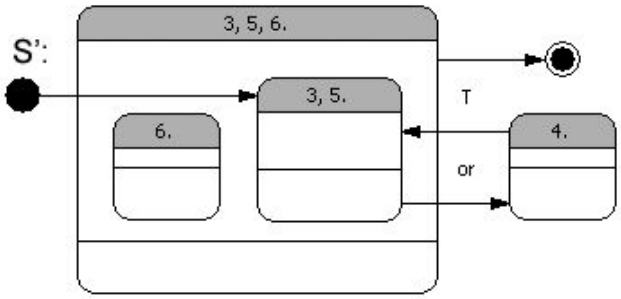

Рис. 7. Удаление немотивированного перехода из состояния 3,5 в состояние 6

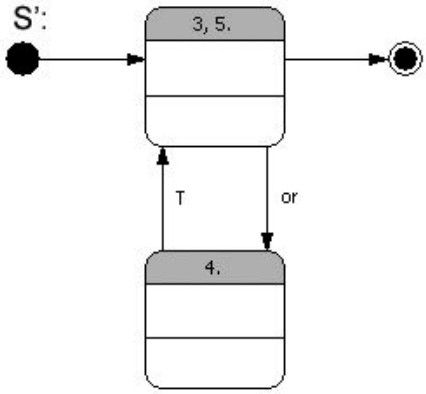

Рис. 8. Преобразованная диаграмма состояний для нетерминала S'

На рис. 5 описанный алгоритм можно применить для состояний *3* и *5*. Результирующая диаграмма показана на рис. 6. На рис. 7 показано удаление немотивированного переход из состояния *3,5* в состояние *6*. Состояние *6* на рис. 7 не имеет входящих переходов и, следовательно, не достижимо, поэтому оно может быть удалено. После удаления состояния *6* в состояние *3,5,6* будет вложено единственное состояние *3,5*. Начала переходов, исходящих из состояния *3,5,6* следует перенести в состояние *3,5*, а само состояние *3,5,6* удалить. На рис. 8 приведена диаграмма состояний для нетерминала `S'` после всех описанных модификаций.

Подстановка диаграмм переходов друг в друга

Диаграммы переходов могут быть упрощены подстановкой одних диаграмм в другие. Предположим, что на диаграмме для нетерминала N_1 существует переход из состояния S_1 в состояние S_2, помеченный нетерминалом N_2. Заменим такой переход на немотивированный из состояния S_1 в состояние, следующее за начальным на диаграмме переходов для нетерминала N_2. Добавим переход из состояния, предшествующего конечному, на диаграмме переходов для нетерминала N_2 в состояние S_2. Отметим, что указанную подстановку необходимо выполнять, только если $N_1 \neq N_2$, так как в противном случае имеет место срединная рекурсия, удаление которой будет описано ниже.

После выполнения такой подстановки, возникшие немотивированные переходы следует устранить описанным ранее способом. При этом сначала устраняется немотивированный переход, из состояния S_1, а затем переход в состояние S_2.

На рис. 9 продемонстрирована подстановка диаграммы переходов для нетерминала L в диаграмму переходов для нетерминала T'. На рис. 10 приведена диаграмма переходов после устранения немотивированных переходов.

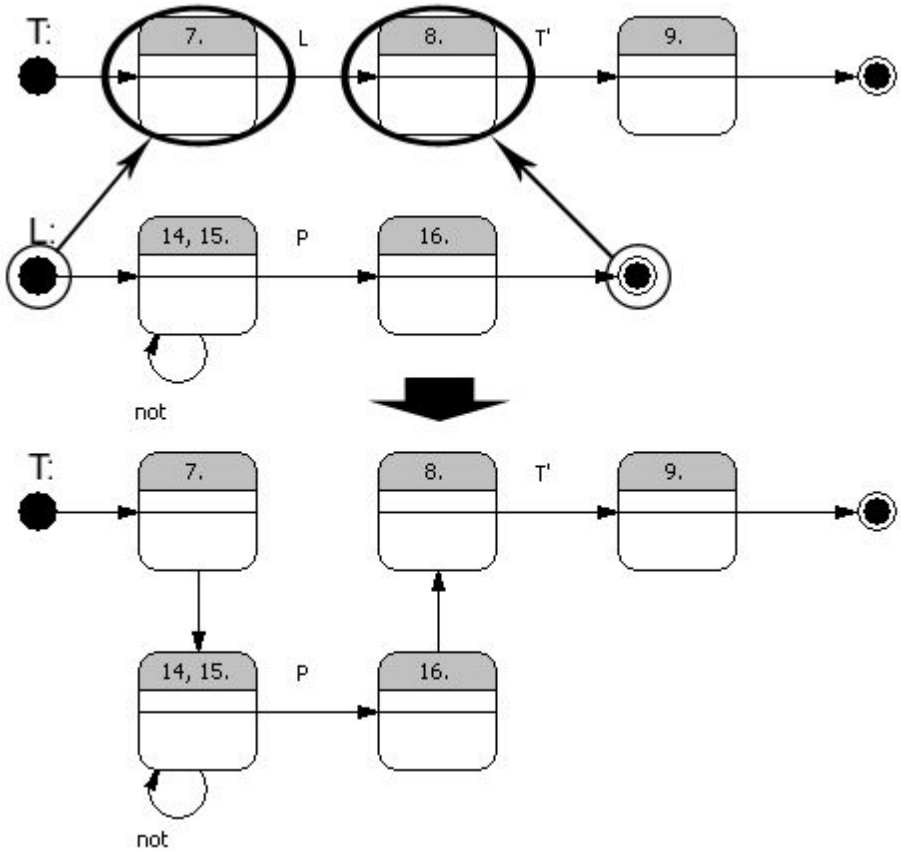

Рис. 9. Подстановка диаграмм переходов друг в друга

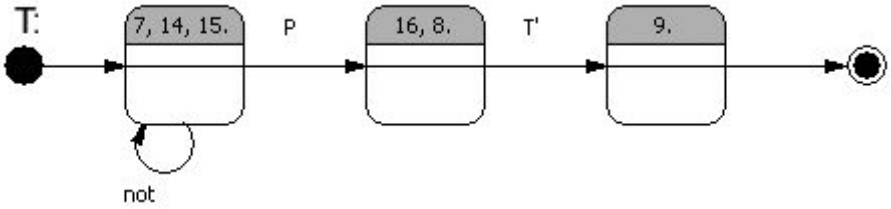

Рис. 10. Устранение немотивированных переходов после подстановки

Если некоторый нетерминал N_1 неоднократно присутствует на диаграмме для нетерминала N_2, то каждый переход, помеченный N_1, необходимо заменить соответст-

вующей диаграммой переходов. В результате количество однотипных подграфов на диаграмме N_2 чрезмерно возрастает.

Предлагается преобразовать диаграмму N_2 таким образом, чтобы нетерминал N_1 встречался на ней минимальное число раз. Для этого используется следующий метод: если несколько переходов помеченных нетерминалом N_1 ведут в одно и то же состояние S, то можно выделить составное состояние, и заменить все эти переходы единственным переходом, помеченным нетеминалом N_1, исходящим из группового состояния и входящим в состояние S.

На рис. 11 данный метод применен для диаграммы переходов нетерминала S.

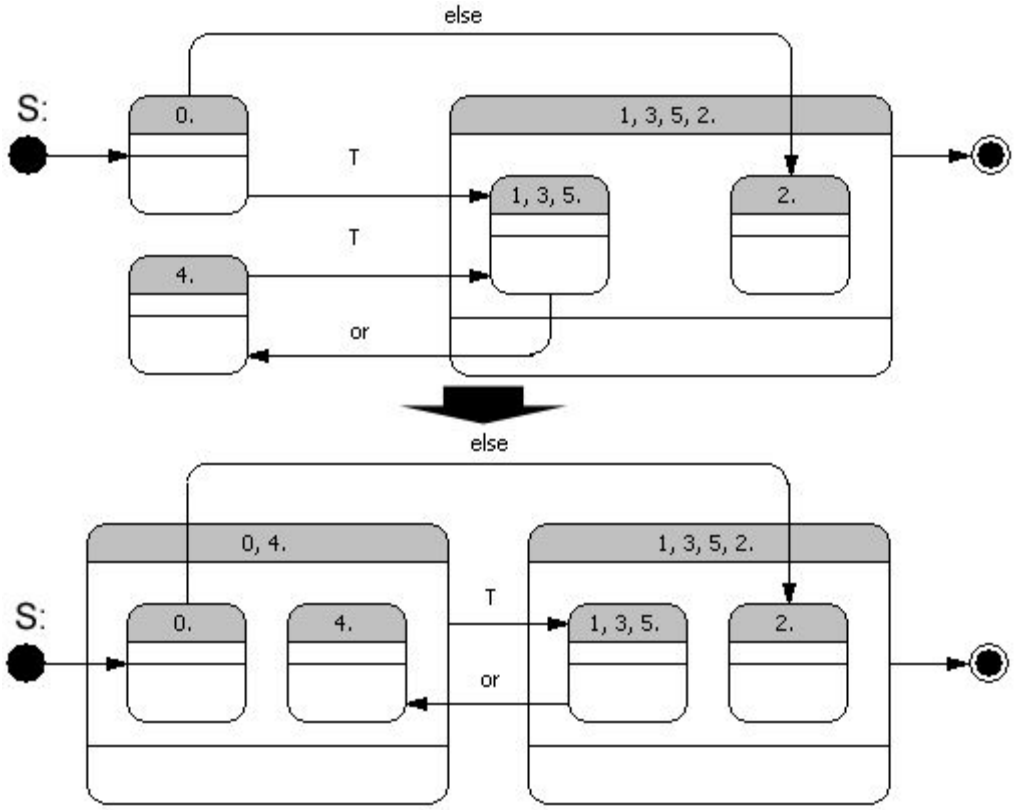

Рис. 11. Удаление двух переходов помеченных нетерминалом T на диаграмме переходов для нетерминала S

Удаление срединной рекурсии

В результате выполнения предыдущих шагов исходное множество диаграмм переходов преобразуется во множество диаграмм, возможно, имеющих срединную рекурсию. Для описанной выше грамматики исходное множество диаграмм, показанное на рис. 3, преобразуется в одну диаграмму, приведенную на рис. 12. На этой диаграмме присутствует срединная рекурсия – переход из состояния *18* в состояние *19*.

Для исключения переходов, образующих срединную рекурсию, предлагается использовать метод, предложенный в работе [6]:

- пусть на диаграмме переходов для нетерминала N существует рекурсивный переход из состояний S_1 в состояние S_2, помеченный нетерминалом N;
- заменим такой переход двумя немотивированными переходами. При этом первый из них должен выходить из состояния S_1, а входить в состояние, следующее за начальным состоянием на диаграмме. Второй переход должен выходить из состояния, предшествующего конечному, а входить в состояние S_2;
- на первом переходе должно выполняться действие по добавлению в стек метки M_{S2}, соответствующей исходному целевому состоянию S_2, а на втором переходе — дей-

ствие по извлечению метки M_{S2} из стека, при условии, что метка M_{S2} находиться на вершине стека.

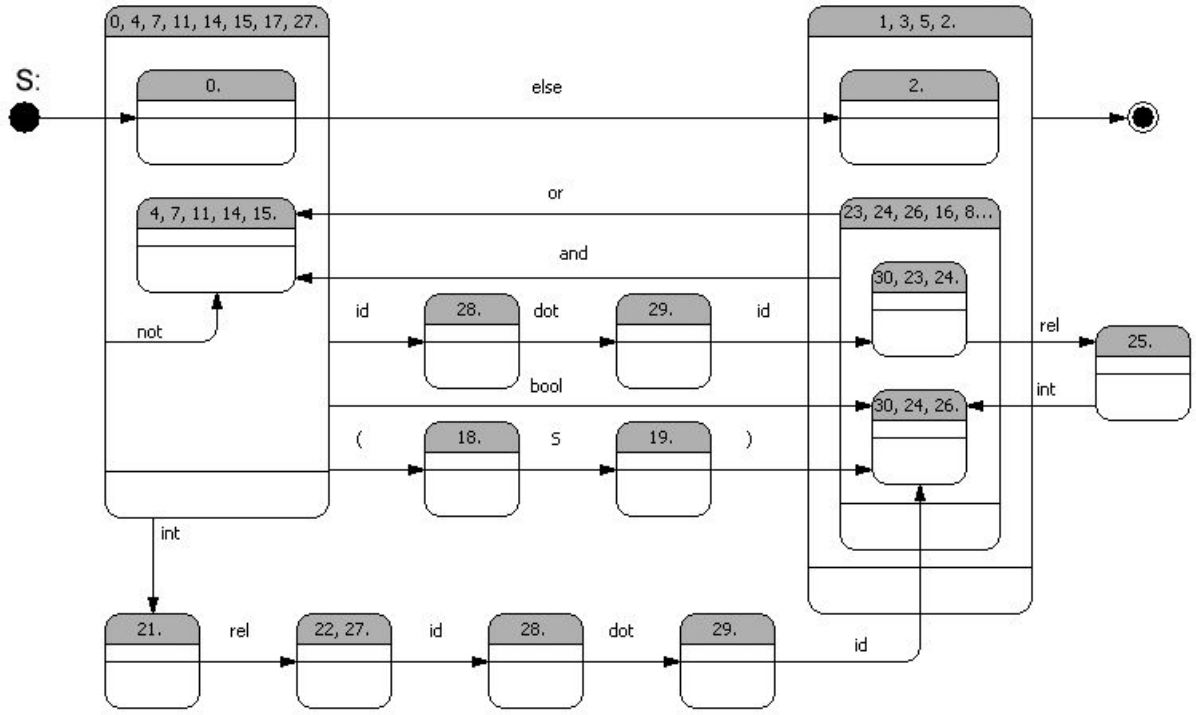

Рис. 12. Диаграмма состояний со срединной рекурсией

На рис. 13 показана диаграмма, приведенная на рис. 12, с удаленной срединной рекурсией.

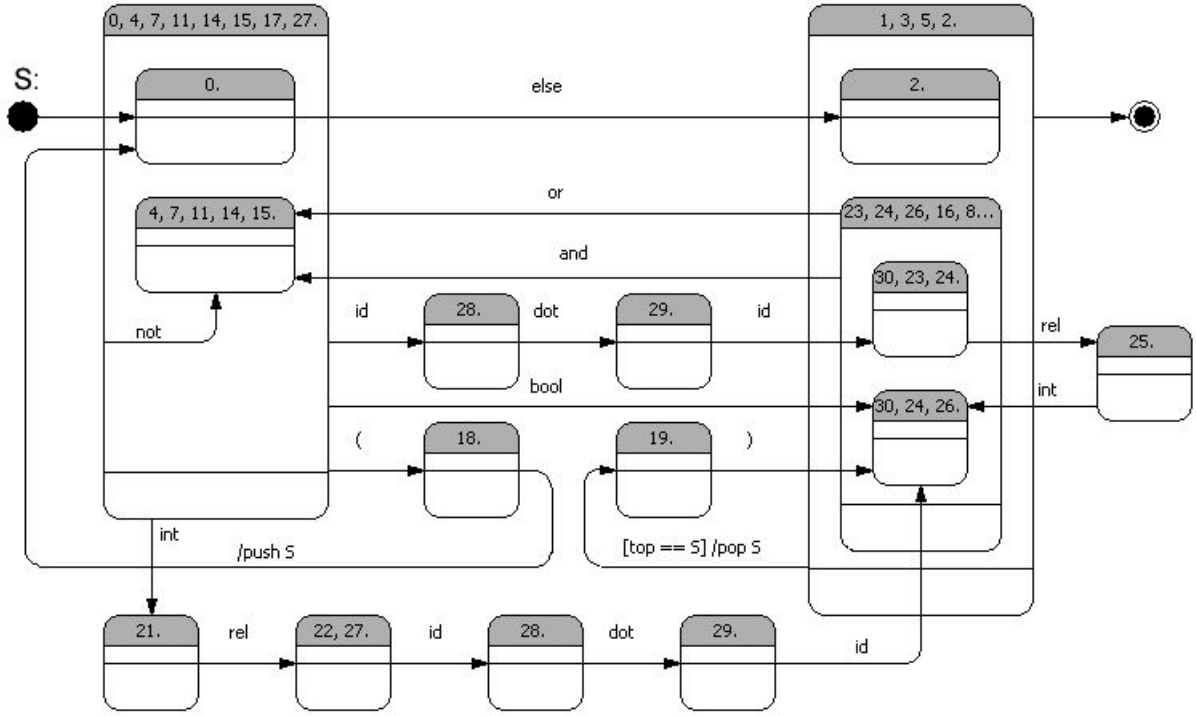

Рис. 13. Диаграмма состояний с удаленной срединной рекурсией

Отметим, что если преобразованное множество диаграмм состоит более чем из одной диаграммы, то после удаления срединной рекурсии в каждой из них, необходимо продолжить процесс преобразования с шага 3 – подстановка диаграмм переходов друг в друга.

Модель разрабатываемой системы

Описанные выше шаги приводят к построению диаграммы переходов для автомата типа Мили. Поставим в соответствие каждому терминалу событие, поставляемое лексическим анализатором, и создадим схему связей автомата [1] в виде *UML*-диаграммы классов [9].

На рис. 14 приведена такая схема. При этом на ней слева показан объект *p1*, соответствующий лексическому анализатору, в центре – объект *A*, соответствующий автомату Мили, а справа – объект управления *o1*, соответствующий стеку.

На рис. 15 показана *UML*-диаграмма состояний автомата, построенная на основе диаграммы, приведенной на рис. 13, с помощью замены терминалов событиями и замены действий на переходах ссылками на методы объекта управления *o1*. Состояния 18 и 19 на рис. 15 отсутствуют из-за удаления немотивированных переходов.

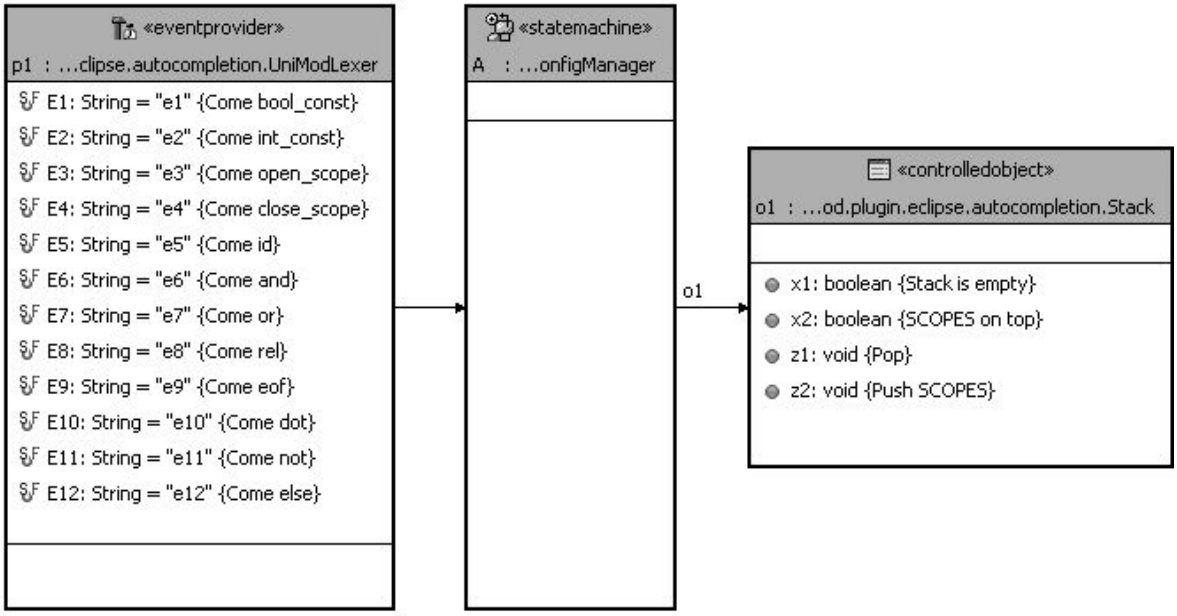

Рис.. 14. Схема связей автомата

Полученная модель системы состоит из двух *UML*-диаграмм (рис. 14, 15) и описывает распознаватель для языка, заданного приведенной выше грамматикой. Отметим, что информация о приоритете операций была потеряна в ходе преобразований, и, поэтому, модель может быть использована для распознавания принадлежности выражений языку, но не для трансляции выражений.

Выражение, принадлежащее языку, поданное на вход распознавателю, приводит автомат *A* в финальное состояние. При подаче на вход выражения являющегося префиксом какого-либо выражения принадлежащего языку, автомат остановиться в каком-то состоянии, множество исходящих переходов из которого определяет множество возможных следующих терминалов.

Если выражение языку не принадлежит и не является префиксом какого-либо выражения принадлежащего языку, то автомат *A* остановится в состоянии, в котором было получено событие, для которого не существовало исходящих переходов при текущих значениях входных переменных. В этом случае множество возможных следующих терминалов можно определить только для последнего правильно обработанного терминала.

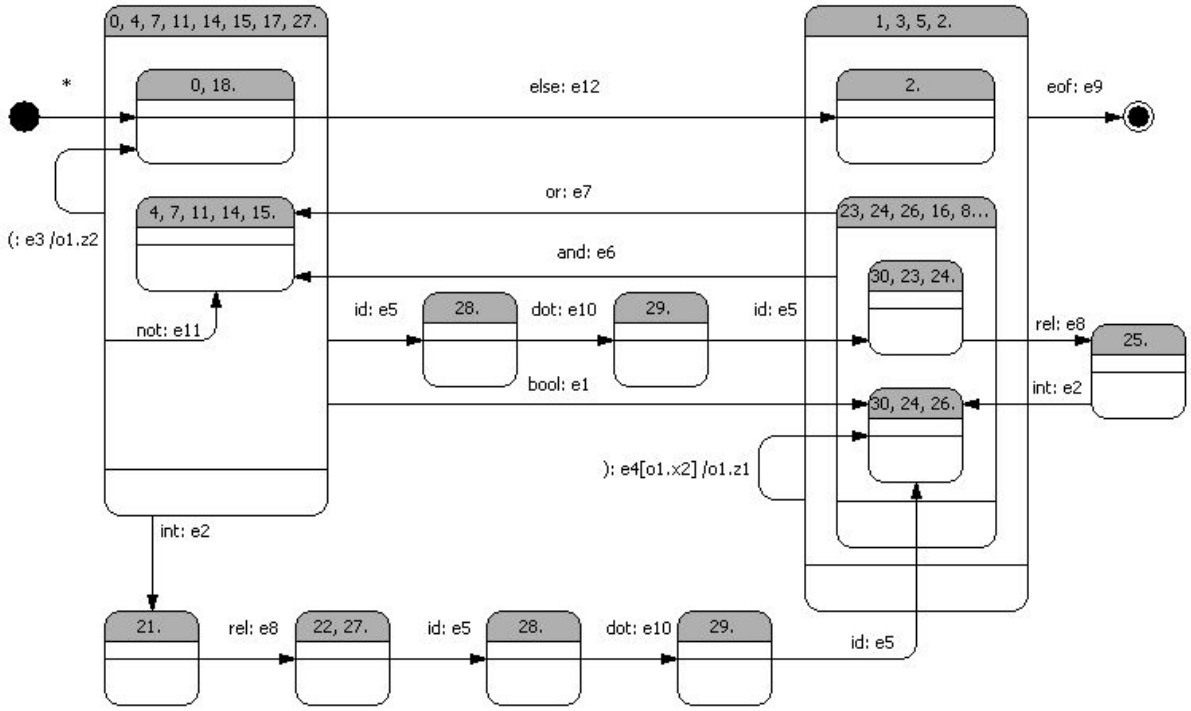

Рис. 15. Диаграмма состояний автомата

Восстановление после ошибок

Перейдем к реализации второго, предъявляемого к системе, требования – обработка ошибочных строк. Для его реализации автомат *A* необходимо модифицировать таким образом, чтобы он корректно восстанавливался в случае подачи на вход выражения, не являющегося префиксом какого-либо выражения принадлежащего языку. При этом автомат должен всегда останавливаться в состоянии, в которое существует переход по событию, соответствующему последнему терминалу, извлеченному из поданного на вход выражения.

Существует несколько возможных вариантов реализации восстановления автомата после ошибки [5]. Например, можно для каждого состояния добавить такой исходящий переход, ведущий в конечное «ошибочное» состояние, что он будет срабатывать в случае отсутствия какого-либо другого исходящего перехода для пришедшего события и текущих значений входных переменных. Это приведет к тому, что при появлении в процессе распознавания первого же ошибочного терминала, автомата завершит работу в «ошибочном» состоянии. Однако из «ошибочного» состояния нет исходящих переходов, и, следовательно, множество возможных последующих терминалов будет пустым.

Авторы предлагают использовать альтернативный алгоритм обработки ошибочных ситуаций, основанный на локально оптимальной коррекции входного потока терминалов от лексического анализатора. Такой подход также называют восстановлением на уровне фразы.

Пусть из состояния *S* нет исходящего перехода для пришедшего события *e*, соответствующего некоторому терминалу. Тогда коррекция потока может осуществляться двумя способами: дополнением потока недостающими терминалами; пропуском лишних терминалов в потоке.

Для того, чтобы автомат *A*, находясь в состоянии *S*, пропустил в потоке терминал, соответствующий пришедшему событию *e*, необходимо добавить в автомат петлю в состоянии *S* по событию *e*. Тогда, находясь в состоянии *S* и получив событие *e*, автомат останется в состоянии *S*, то есть проигнорирует пришедшее событие, и, как, следствие, пропустит терминал в потоке.

Для того чтобы автомат *A*, находясь в состоянии *S*, дополнил поток недостающими терминалами, следует выполнить указанные ниже операции:

1. найти достижимое из *S* состояние S_h, такое, что в нем существует исходящий переход по событию *e*. Если из состояния *S* достижимы несколько таких состояний, то выберем ближайшее из них;
2. если из ближайшего найденного состояния S_h есть переход в некоторое состояние S_t по событию *e* при условии *c*, то необходимо добавить в автомат переход из состояния *S* в состояние S_t по событию *e* при условии *c*. Отметим, что отсутствие условия трактуется как тождественная истина.

Последовательность терминалов, соответствующих событиям, которыми помечен кратчайший путь из состояния *S* в состояние S_h, можно использовать для вставки в поток перед терминалом, соответствующим пришедшему событию *e*.

Если лексический анализатор позволяет заглядывать на произвольное количество терминалов вперед, то можно применять оба способа коррекции одновременно, выбирая оптимальный способ в процессе разбора.

Для выбора оптимального способа авторы предлагают использовать следующее правило:

1. при получении ошибочного терминала в текущем состоянии вычислим количество терминалов, которыми нужно дополнить поток;
2. вычислим количество терминалов, которое нужно пропустить в потоке, до следующего обрабатываемого в текущем состоянии терминала;
3. выполним коррекцию, требуемое количество терминалов для которой минимально.

Для реализации этого способа коррекции к автомату *A* распознавателя в качестве объекта управления добавим лексический анализатор (рис. 16). Лексический анализатор предоставляет автомату распознавателя целочисленную входную переменную *o2.x1*. Ее значение равно числу терминалов, которые необходимо пропустить в потоке, до следующего терминала, обрабатываемого в текущем состоянии. Если входной поток вообще не содержит терминалов, обрабатываемых в текущем состоянии, то значение переменной *o2.x1* больше любого наперед заданного целого числа.

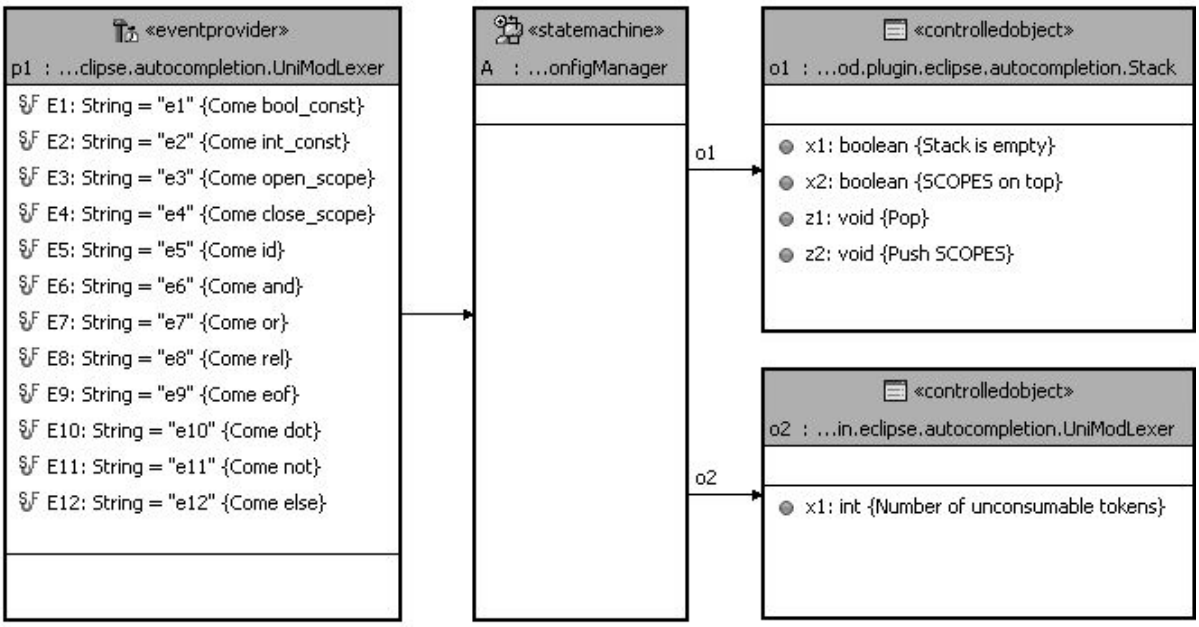

Рис. 16. Схема связей модели распознавателя с лексическим анализатором в качестве объекта управления

В автомат *A* добавляются переходы, реализующие и добавление и пропуск терминалов в потоке. Переходы, реализующие дополнение потока, помечаются следующим ус-

ловием: длина пути из состояния S в состояние S_h меньше или равна значению входной переменной *o2.x1*. Переходы, реализующие пропуск терминалов, помечаются отрицанием того же условия. Если для состояния S не существует состояния S_h, то переход, удаляющий лексему, выполняется безусловно.

Например, в состоянии *21* нет переходов по событию *e10* – в этом состоянии появление во входном потоке терминала **`dot`** (точка) не ожидается. Для того чтобы обработать ошибочное появление этого терминала, необходимо добавить два перехода, исходящих из состояния *21* (рис. 17). Ближайшее состояние, в котором обрабатывается событие *e10* — состояние *28*. Длина пути из состояния *21* в состояние *28* равна двум. Поэтому условие на петле в состоянии *21* по событию *e10* имеет вид *o1.x1 < 2*. Таким образом, в случае, если сразу за терминалом **`dot`** в потоке следует терминал **`rel`** (отношение), то терминал **`dot`** игнорируется. Если следует какой-нибудь другой терминал, то целесообразно сразу перейти в состояние *29*, то есть добавить в поток отсутствующие терминалы **`rel`** и **`id`** (идентификатор).

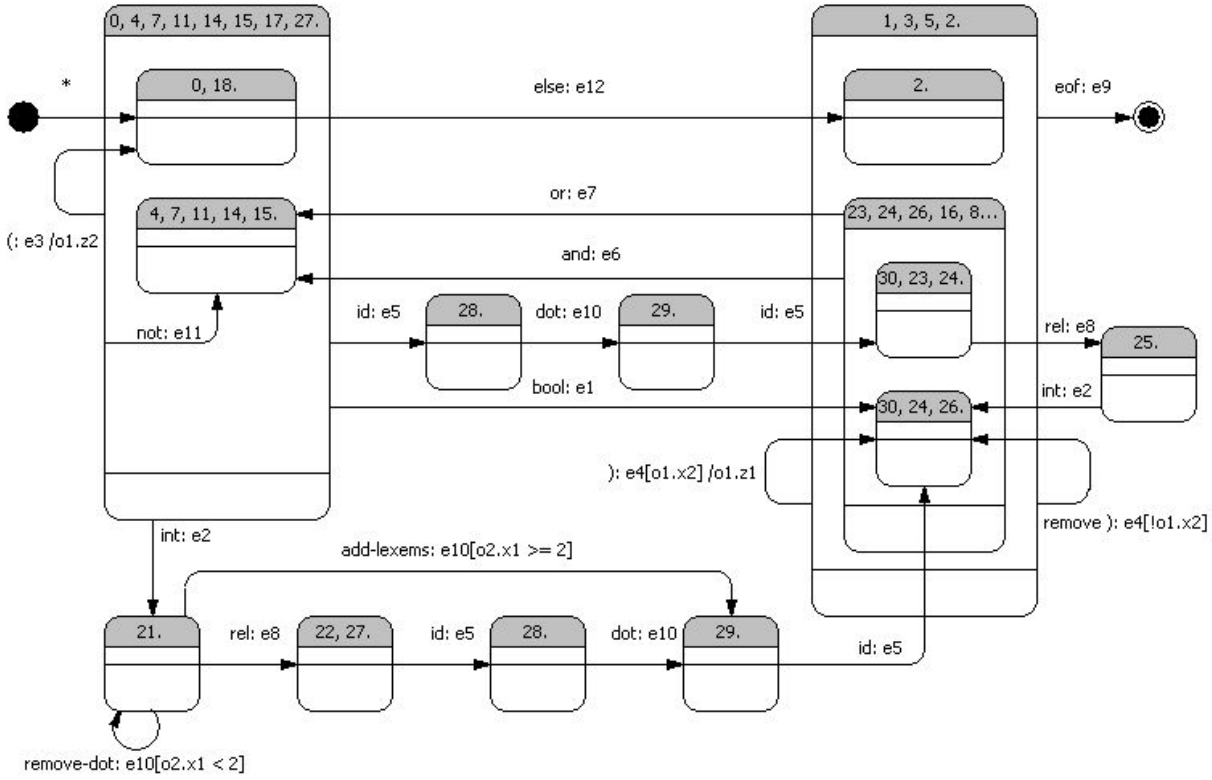

Рис. 17. Добавление переходов, корректирующих поток, в состоянии 21

Описанные выше преобразования могут быть выполнены автоматически для любой диаграммы переходов, так как ближайшее состояние, в котором обрабатывается неожидаемый терминал для данного состояния, может быть вычислено, используя, например, алгоритм Флойда-Уоршала [11].

Получение множества строк для автоматического завершения ввода

Предлагаемый алгоритм коррекции входного потока позволяет вычислить множество вариантов завершения как для выражений, являющихся префиксами принадлежащих языку выражений, так и для ошибочных выражений.

После того, как автомат распознавателя, дополненный корректирующими переходами, обработает все терминалы, извлеченные из поданного на вход выражения, он окажется в некотором состоянии *S*. Для построения множества вариантов завершения следует определить множество переходов, исходящих из *S*, условия на которых при те-

кущих значениях входных переменных истинны. Терминалы, соответствующие событиям, которыми помечены эти переходы, должны быть преобразованы обратно во множество строк. Например, терминал **id** должен быть преобразован во множество имен переменных, а терминал **and** – в строку «&&». Полученное множество строк и будет множеством вариантов завершения.

Пример работы системы

Приведем пример построения множества вариантов завершения.

Пусть на вход распознавателю подана строка:

```
! o1.x1 &&
```

Лексический анализатор преобразует ее в поток терминалов:

```
not id dot id and,
```

которому соответствует последовательность событий:

```
e11, e5, e10, e5, e6
```

В процессе обработки этих событий автомат изменяет состояния в следующем порядке:

```
(0,18)→(4,7,11,14,15)→(28)→(29)→(30,23,24)→(4,7,11,14,15)
```

Состояние `(4,7,11,14,15)`, в котором остановился автомат, содержит исходящие переходы для событий:

```
e1, e2, e3, e5, e11.
```

Этим событиям соответствуют терминалы:

```
bool, int, '(', id, not,
```

которые преобразуются в строки:

```
"true", "false", "(", "o1", "o2", "o22", "!".
```

Эти строки и формируют множество вариантов завершения для строки, поданной на вход распознавателю.

На рис. 18 показан фрагмент среды разработки с встроенной системой автоматического завершения ввода, описанной в настоящей статье.

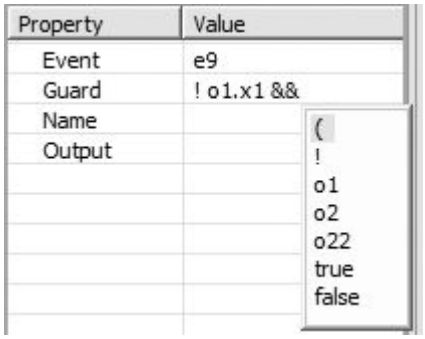

Рис. 18. Пример автоматического завершения ввода

Заключение

В работе [5] отмечено, что нерекурсивные нисходящие синтаксические анализаторы можно строить, используя диаграммы переходов, записанные для каждого нетерминала исходной грамматики. Настоящая работа предлагает подход для построения всего одной диаграммы переходов для исходной грамматики. На базе построенной диаграммы реализуется система автоматического завершения ввода. Также отметим, что, в известной авторам литературе, описание формального метода построения подобных систем отсутствует. Данная работа устраняет указанный пробел.

Реализация системы автоматического завершения ввода для следующей версии проекта *UniMod* выполнена с помощью предыдущей версии проекта, вследствие чего

часть проектной документация была получена «автоматически», так как диаграммы созданные с помощью *UniMod* редактора являются автоматными программами и могут быть включены в проектную документацию без изменений. Разработка последующих версий средств разработки с помощью предыдущих является общепринятой практикой и позволяет говорить о зрелости программного продукта.

Литература

1. Шалыто А.А., Туккель Н.И. Танки и автоматы //BYTE/Россия. 2003. № 2, с. 69-73. http://is.ifmo.ru/ (раздел «Статьи»).
2. Шалыто А.А., Туккель Н.И. SWITCH-технология — автоматный подход к созданию программного обеспечения "реактивных" систем //Программирование. 2001. № 5, с. 45-62. http://is.ifmo.ru/ (раздел «Статьи»).
3. Фаулер М. Рефакторинг. Улучшение существующего кода. М.: Символ-Плюс, 2003. — 623 с.
4. Parr T.J., Quong R.W. ANTRL: A Predicated-LL(k) Parser Generator // Software — Practice And Experience. 1995, №25(7). p. 789-810.
5. Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструменты. Вильямс. 2001. — 768 с.
6. Шалыто А.А., Туккель Н. И., Шамгунов Н.Н. Реализация рекурсивных алгоритмов на основе автоматного подхода. //Телекоммуникации и информатизация образования, 2002, № 5.
7. Шалыто А.А., Штучкин А.А., Совместное использование теории построения компиляторов и SWITCH-технологии (на примере построения калькулятора). **http://is.ifmo.ru** раздел «Статьи».
8. Хантер Р. Основные концепции компиляторов. Вильямс. 2002. — 256 с.
9. Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. М.: ДМК. 2000. — 320 с.
10. Акимов О.Е. Дискретная математика: логика, группы, графы. М.: Лаборатория Базовых Знаний. 2003. – 376 с.
11. Кормен Т., Лайзерсон Ч., Ривест Р. Алгоритмы. Построение и анализ. М.: МЦМНО. 2000. — 960 с.
12. Легалов А. Основы Разработки трансляторов. Использование диаграмм Вирта для представления динамически порождаемых конечных автоматов, распознающих КС(1) грамматику. **http://softcraft.ru/translat/lect/t08-04.shtml**