Лекція 10. Розробка синтаксичних діаграм для роспізнавача DPL

Преобразование пользовательского синтаксиса к КС(1) грамматике

Разработка распознавателя базируется на создании синтаксиса языка программирования, обеспечивающего успешную программную реализацию. Исходный пользовательский синтаксис не подходит для написания программы по ряду причин, среди которых следует отметить следующие:

- обозначения, используемые для нетерминальных символов, написаны фразами на русском языке и в языках программирования не могут быть напрямую использованы (необходима формализация обозначений);

- синтаксические правила не соответствуют КС(1) грамматике, что не позволяет реализовать распознаватель изученными методами (необходимо преобразование к КС(1) грамматике с проведением соответствующих доказательств);

- для построения распознавателя в виде набора динамически порождаемых конечных автоматов необходимо осуществить разметку состояний в окончательных правилах.

После устрания описанных причин и решения указнанных задач переход к программной реализации может быть проведен с использованием формализованных подходов.

Замена обозначений нетерминалов в исходных диаграммах Вирта

В целом выполнение этого действия не вызывает затруднений. Для модификации используются диаграммы, не вошедщие в лексический анализатор. Обозначния лексем выбирается исходня из их обозначений в соответствующем перечислимом типе сканера. Помимо этого в формируемых диаграммах лексемы обозначаются в виде терминальных символо (овалами) вместо ранее используемого их обозначения в виде нетерминалов (прямоугольников). Такая трансформация обуславливается структурой разрабатываемого транслятора в котором лексический анализатор выдает уже неделимые базовые элементы.

Исключение могут составлять обозначения ключевых слов и разделителей, так как использование в данной ситуации «ручной» разработки транслятора позоляет легко подставить нужное обозначение при написании программы вместо название ключевого слова, используемого в диаграмме Вирта (нужно только добавить префикс ***kw***). Вместе с тем, использование истинных названий ключевых слов и разделителей на наш взгляд облегчает восприятие полученных диаграмм.

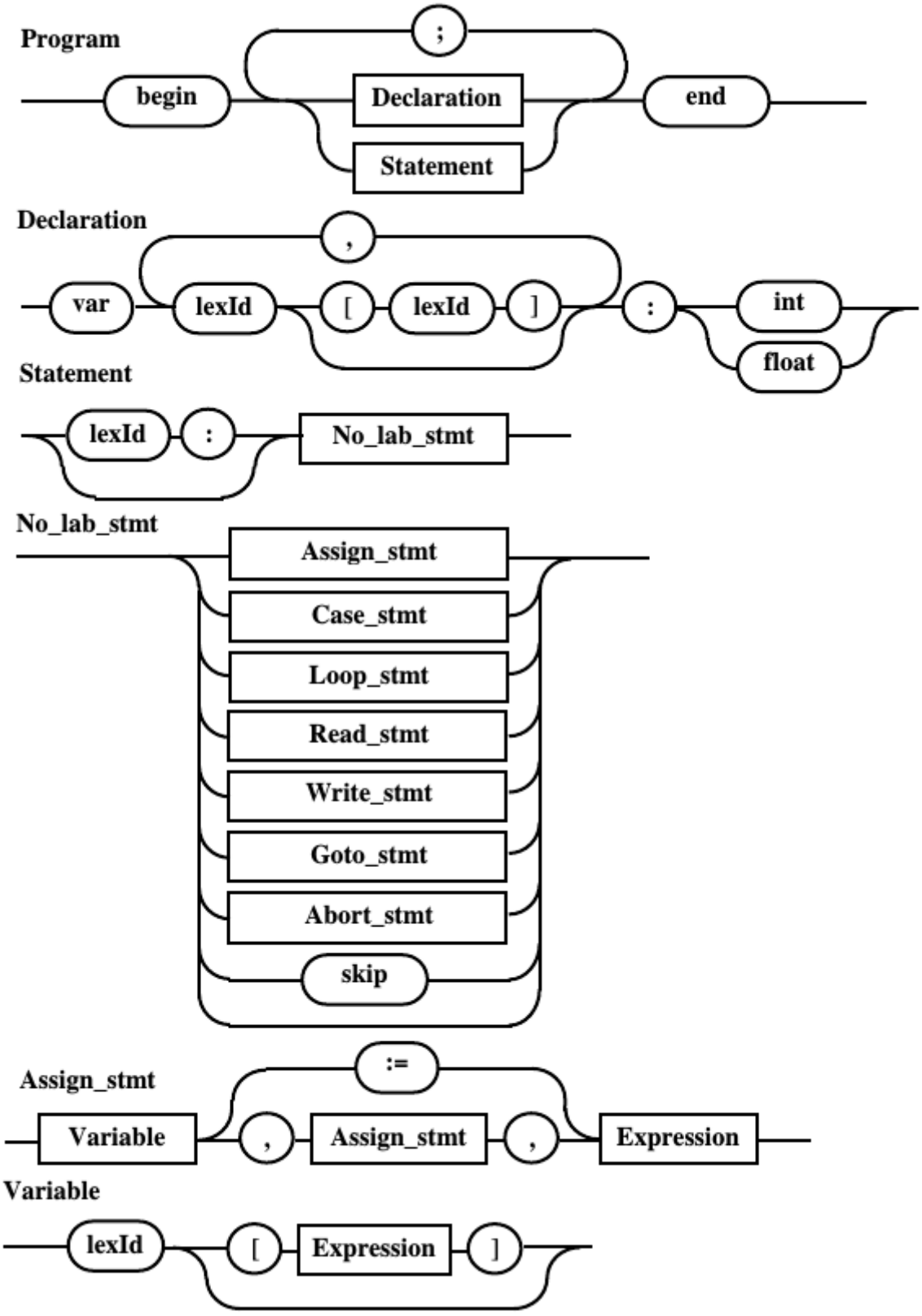
И, наконец, осуществляется замена неформальных названий нетерминалов и, следовательно, обозначений правил на идентификаторы, которые будут использоваться в программе. Эти

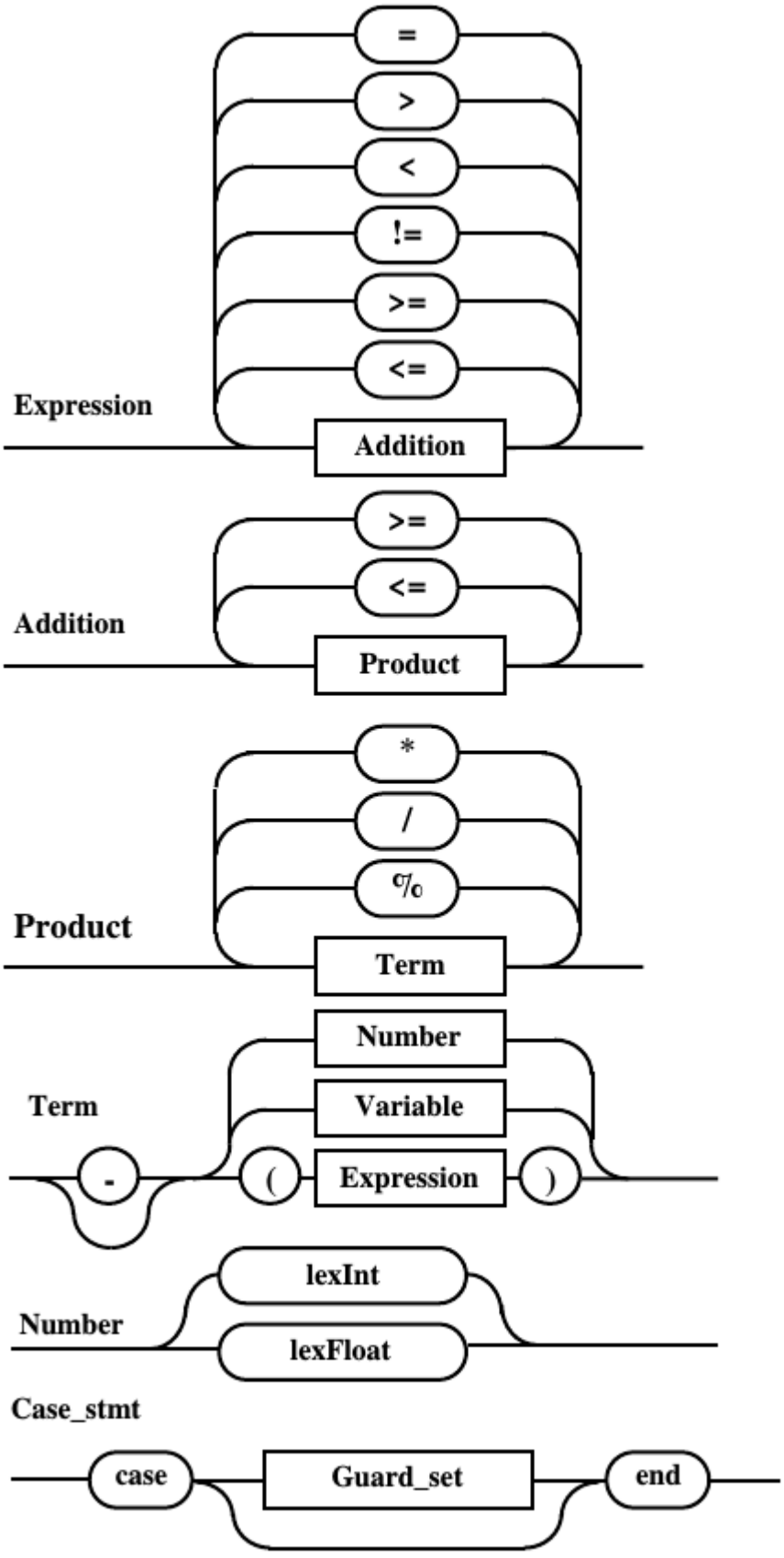
идентификаторы выбираются в соответствии с именами процедур или функций, допустимых в языке, используемом для реализации транслятора.

Помимо этого, в ряде случаев можно провести модификацию исходных диаграмм Вирта, обеспечивающую впоследсвтии более эффективную реализацию. В частности диаграмма пустого оператора можно полностью перенести в непомеченный оператор, так как это избавляет от необходимости вызывать дополнительные процедуры при распознавании из-за анализа на одно ключевое слова. Аналогичные преобразования можно проделать и с рядом других диаграмм.

Диаграммы Вирта, полученные после подстановки лексем и замены названий на идентификаторы

В результате преобразования исходных правил получены диаграммы, представленные ниже:


Program
;
begin
Declaration
Statement
end
Declaration
,
var
lexId
[
lexId
]
:
int
float
Statement
lexId
:
No_lab_stmt
No_lab_stmt
Assign_stmt
Case_stmt
Loop_stmt
Read_stmt
Write_stmt
Goto_stmt
Abort_stmt
skip
Assign_stmt
:=
Variable
,
Assign_stmt
,
Expression
Variable
lexId
[
Expression
]


=
>
<
!=
>=
<=
Expression
Addition
>=
<=
Addition
Product
*
/
%
Product
Term
Number
Variable
Term
-
(
Expression
)
lexInt
Number
lexFloat
Case_stmt
case
Guard_set
end

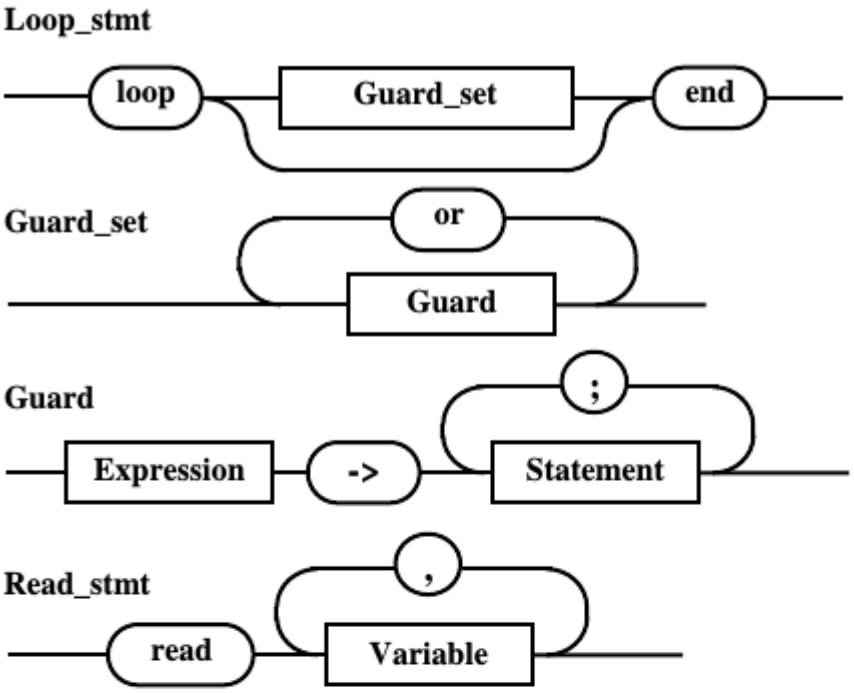
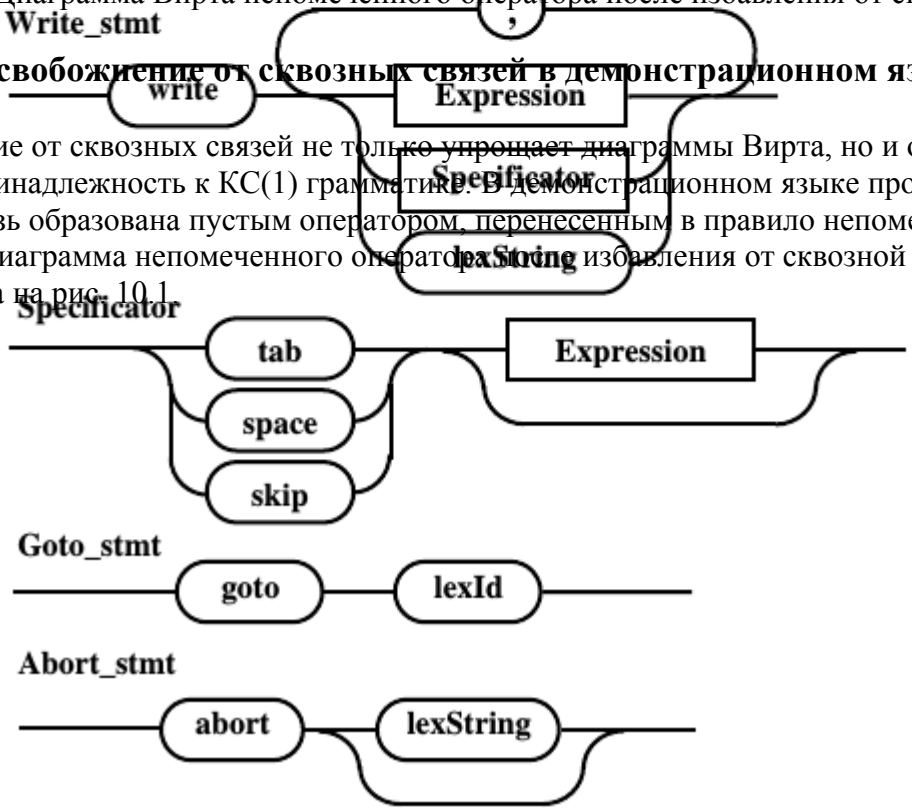


Рис. 10.g. Диаграмма Вирта непомеченного оператора после избавления от сквозной связи



Освобождение от сквозных связей в демонстрационном языке

Освобождение от сквозных связей не только упрощает диаграммы Вирта, но и облегчает их анализ на принадлежность к КС(1) грамматике. В демонстрационном языке программирования сквозная связь образована пустым оператором, перенесенным в правило непомеченного оператора. Диаграмма непомеченного оператора после избавления от сквозной связи представлена на рис. 10.1.

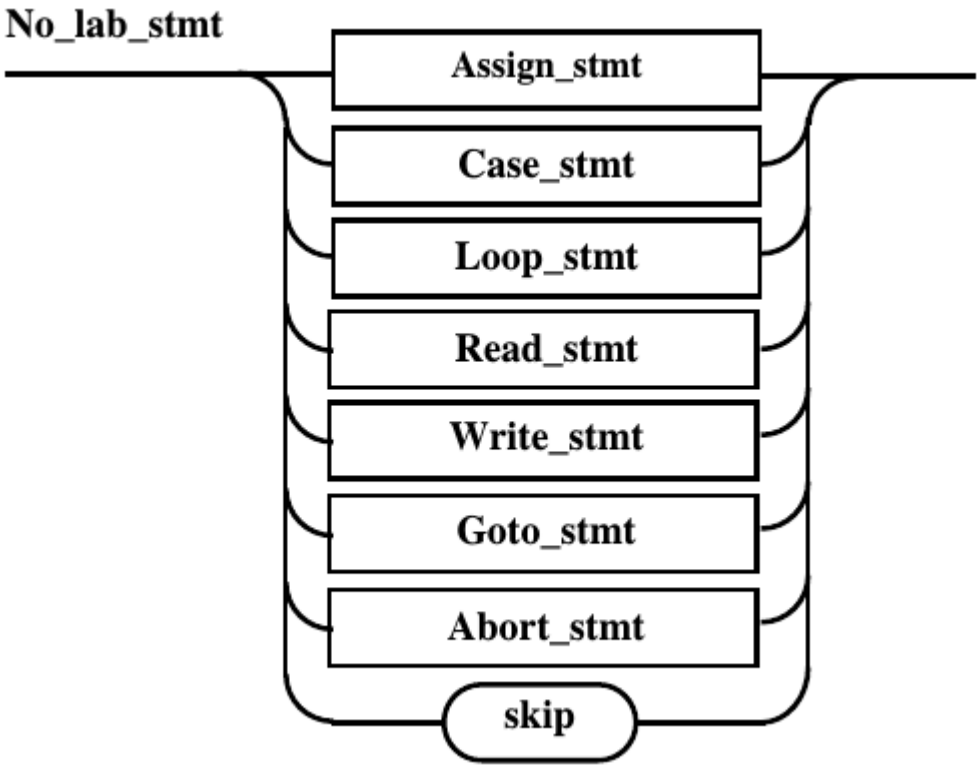


Рис. 10.1. Диаграмма Вирта непомеченного оператора после избавления от сквозной связи

Однако, для сохранения корректности оставшихся правил необходимо учесть вынос сквозной связи из данной диаграммы в другие, которые включают соответствующий нетерминал. В ряде случаев это ведет к появлению новых сквозных связей, от которых тоже необходимо избавиться. В демонстрационном языке подобной метаморфозе подвергается диаграмма, задающая оператор (с меткой). Преобразования этого оператора приведены на рис. 10.2.

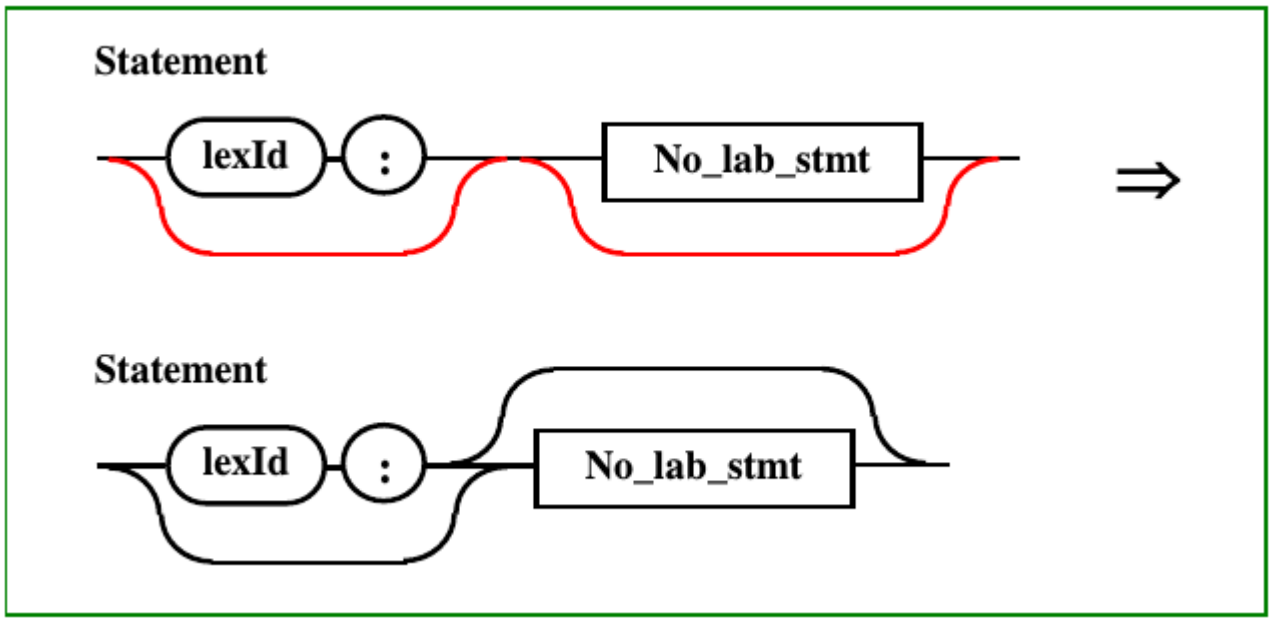


Рис. 10.2. Избавление от сквозной связи в операторе

Возникающая в нем сквозная связь, на следующем шаге преобразования переносится в вышестащие диаграммы (рис. 10.3), задающие программу (***Program***) и защиту (***Guard***). При этом в последнем случае появляется пустая связь, которая ведет к концентрации альтернатив на конце правила, что необходимо учесть в дальнейшем при доказательстве принадлежности к КС(1) грамматике.

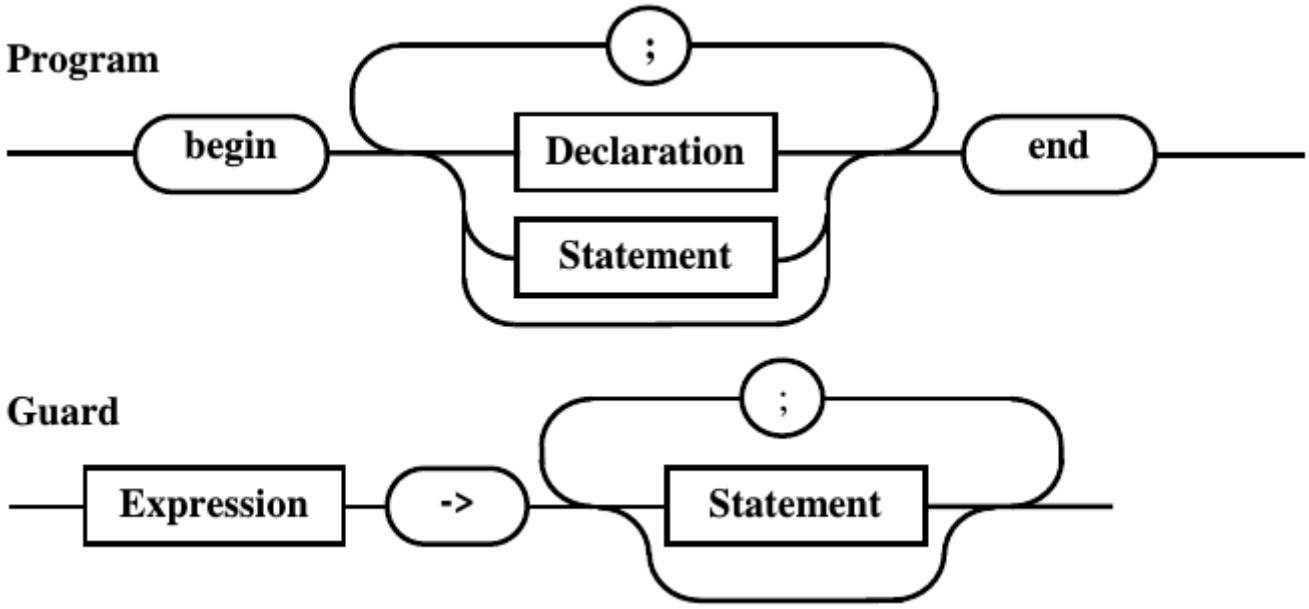


Рис. 10.3. Преобразованные диаграммы **Program** и **Guard**

Модификация правил с одинаковыми начальными лексемами

Наличие у нескольких альтернатвно используемых правил одинаковых начальных лексем говорит о том, перед нами не КС(1) грамматика. Поэтому необходимы преобразования к КС(1) грамматике или использование других приемов обеспечивающих избавление от возникающей неоднозначности. К этим приемам можно, например, отнести использование для данных альтернатив просмотра на несколько символов вперед, анализ с возвратами или применение семантического анализа.

В демонстрационном языке программирования альтернативными правилами, начинающимися с идентификатора, являются: оператор (***Statement***), начинающийся с метки, и непоименченный оператор (***No_lab_stmt***), в котором с идентификатора начинается оператор присваивания (***Assign_stmt***), содержаащий, в свою очередь, переменную (***Variable***). На рис. 10.4. правила, для удобства анализа, собраны в одну группу.

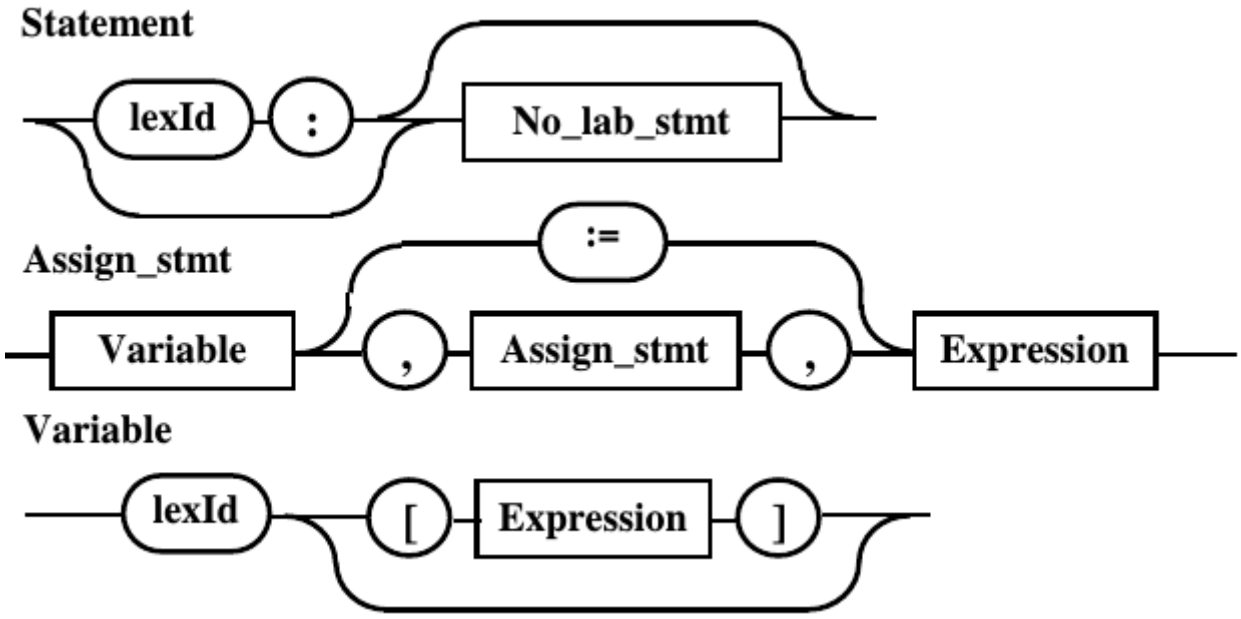


Рис. 10.4. Правила, конфликтующие по лексеме «идентификатор»

При использовании анализа с возвратами назад правила можно не модифицировать. Достаточно организовать разбор таким образом, чтобы при отказе можно было восстановить конфигурацию исполнительной среды в состояние, предшествующее разбору текущего правила, после чего перейти к следующей альтернативе. Ошибка в этом случае выдается после отказа всех альтернатив. Несмотря на замедление разбора анализ с возвратами является вполне разумным решением, применяемых в случае сложных грамматик как универсальный подход.

Для ускорения анализа с возвратами можно воспользоваться дополнительным анализом контекста. Например, можно обратиться к таблицам имен и проверить: является ли идентификатор именем переменной или метки. При отсутствии там идентификатора разумно предположить, что это имя метки и продолжить разбор. Если окажется, что это имя неописанной переменной, то будет выдана ошибка. Такое решение может эффективно применяться в языках со строгой типизации, для которых характерно предварительное описание переменной. Однако в языках с динамической типизацией оно менее удобно из-за наличия многозначности в контексте анализируемого объекта.

При просмотре на несколько символов, характерном для КС(n) грамматик, необходимо модифицировать правила, собрав их в единую конструкцию таким образом, чтобы одинаковые анализируемые альтернативы находились на одном уровне. На рис. 10.5 приведена модификация правил для рассматриваемого случая. Идентификаторы неоднозначно доступные на первом шаге, выделены прерывистыми линиями. Применение связано с анализом идентификатора и следующего за ним символа, который уже отличается для различных ситуаций.

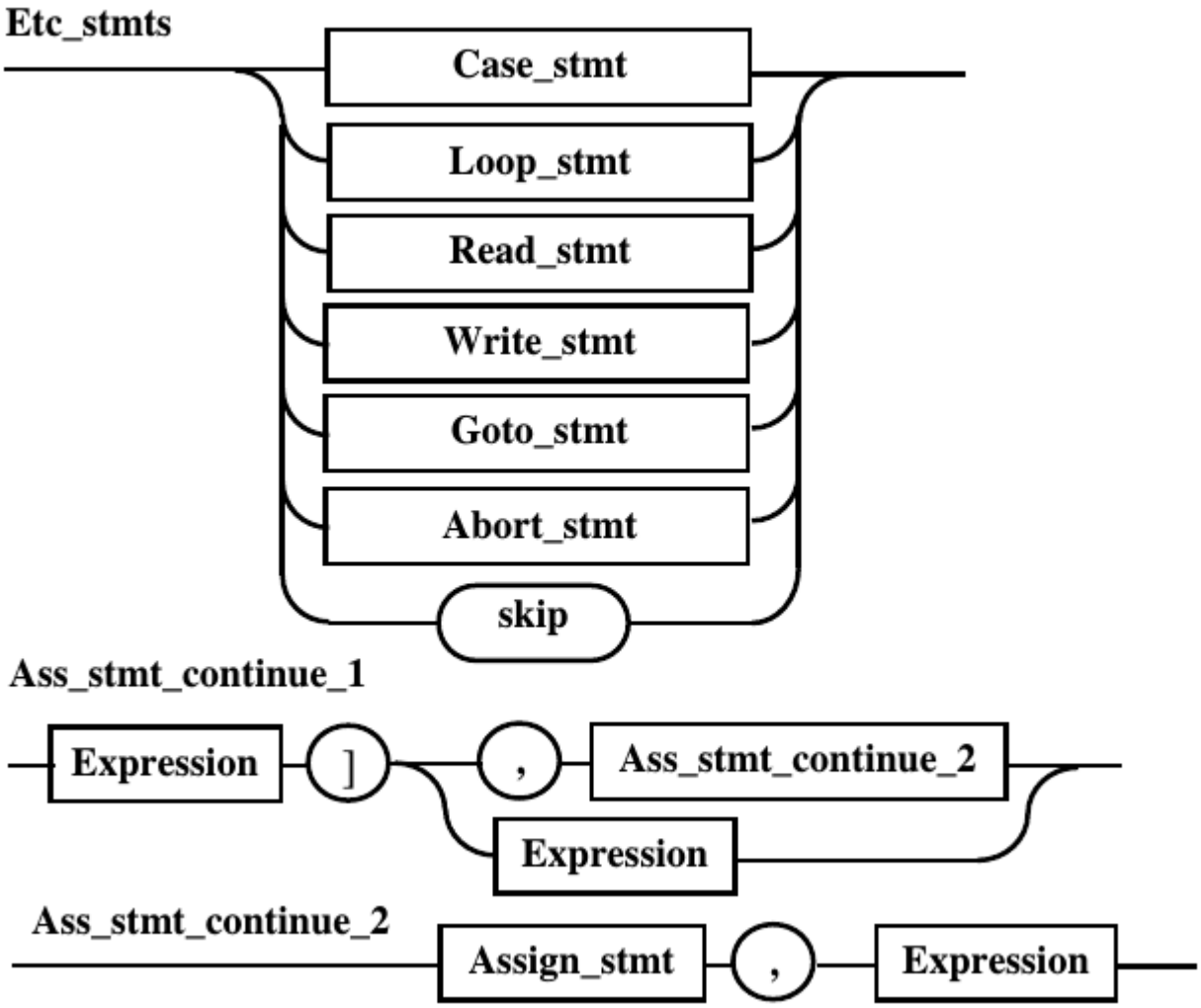


Рис. 10.5. Модификация требуемых правил для КС(2) разбора

Модификация правила ***Statement*** ведет к появлению дополнительных промежуточных правил, которые представлены на рис 10.6.

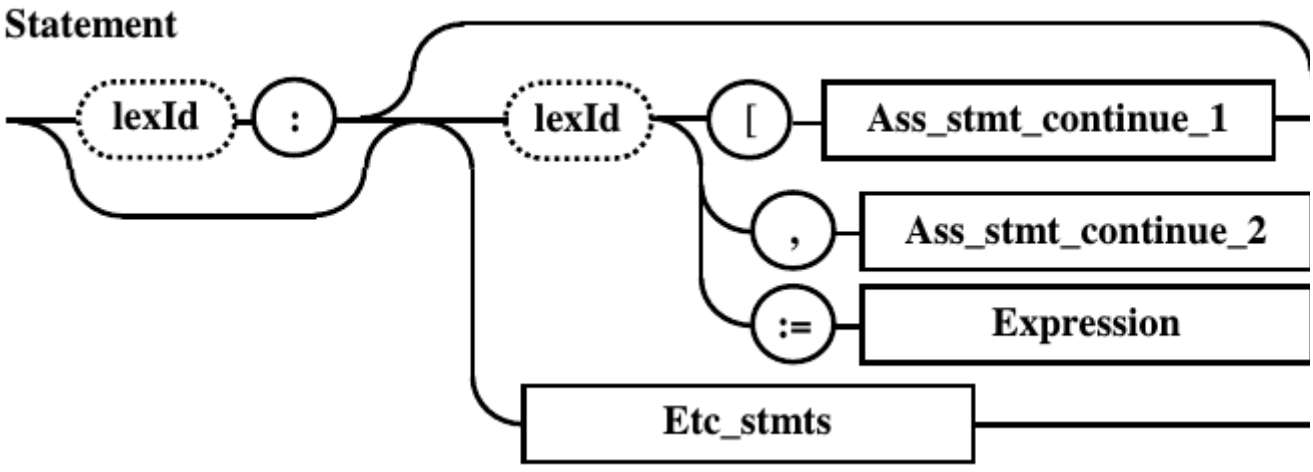


Рис.10.6. Дополнительные промежуточные правила, полученные после сборки воедино

конфликтующих правил

Подход обладает хорошей скоростью анализа удобен для применения в локальном контексте, затрагивющим небольшое число правил. Однако необходимость преобразования правил и неформальность выполняемых при этом действий затрудняют его использоание.

Вместе с тем приведенное на рис. 10.5. правило можно легко преобразовать к КС(1) виду, что сведет разбор к изученному материал. Для этого достаточно слить выделенные лексемы, задающие идентификаторы с сохранение всех прочих альтернативных связей. Полученное правило предствалено на рис. 10.7. Следует отметить появление непомеченного оператора после выявления метки, что позволяет сократить структуру правила и использовать ранее разработанные диаграммы.

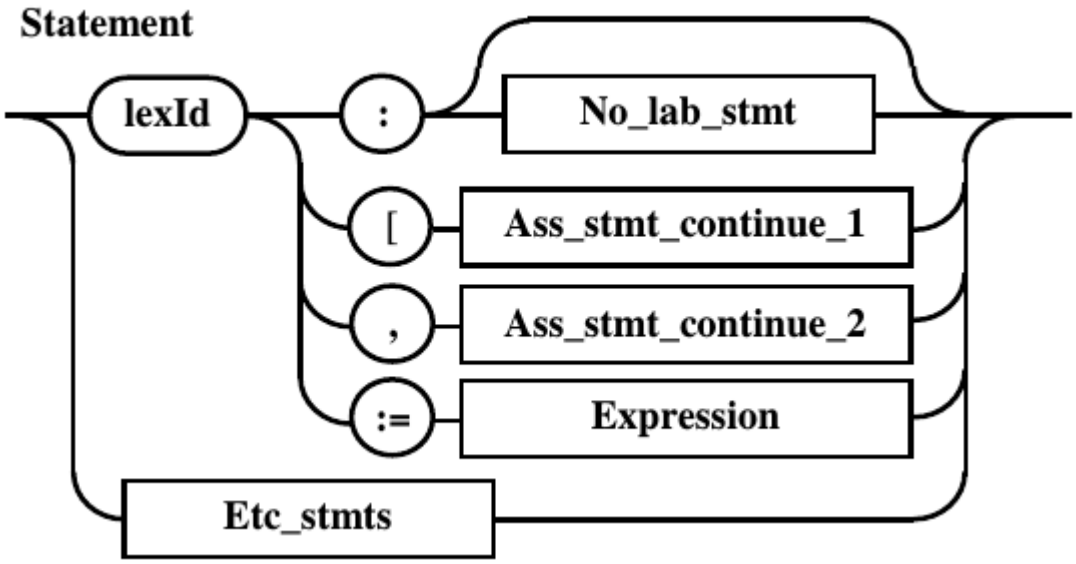


Рис. 10.7. Правило ***Statement***, преобразованное к КС(1) виду

Проверка принадлежности к КС(1) грамматике

После выявления неоднозначностей и сведению конфликтующих альтернативных диаграмм к КС(1) виду необходимо провести проверку принадлежности всей полученной грамматики к КС(1). После этого шага можно приступать к окончательному формированию диаграмм Вирта и программной реализации модуля распознавателя.

Проверка альтернатив в начале и середине правил осуществляется достаочно просто. Однако, избавление от сквозных связей привело к перетеканию альтернатив в конец правил. Поэтому необходимо проверить однозночность для этих ситуаций. Данной модификации подверглись правила, задающие оператор (***Statement***) и защиту (***Guard***). Последняя диаграмма находится на конце в правиле, задающем множество охраняемых (***Guard_set***). Поэтому проверка касается всех трех правил и правил в которые они входят.

Диаграмма оператора имеет связь, обеспечивающую перетекание альтернатив после лексемы, задающей двоеточие. В результате, на конце этого правила концентрируются все лексемы, определяющие альтернативы для непомеченного опиратора. Их список отображен на

диаграмми Вирта, приведенной на рис. 10.8. В качестве нетерминального символа оператор используется в правилах, определяющих программу (***Program***) и защиту (***Guard***). Поэтому представляет интерес рассмотреть, каким образом осуществляется это влияние

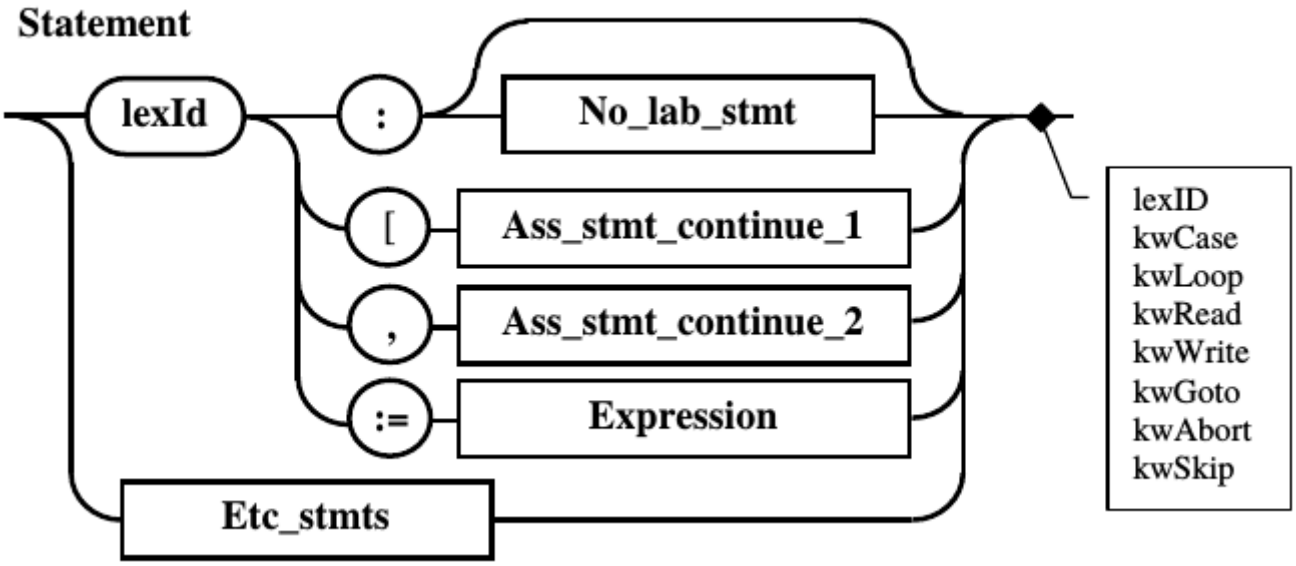


Рис. 10.8. Перетекание альтернатив в правиле ***Statement.***

На рис. 10.9. Приводятся диаграммы для правил ***Program*** и ***Guard***, на коорых показано, что в анализируемых точках к перетекющим альтернативам добавлены новые которые не пересекаются с уже существующими. Это показывает на то, что представленные правила полностью соответствуют КС(1) крамматике.

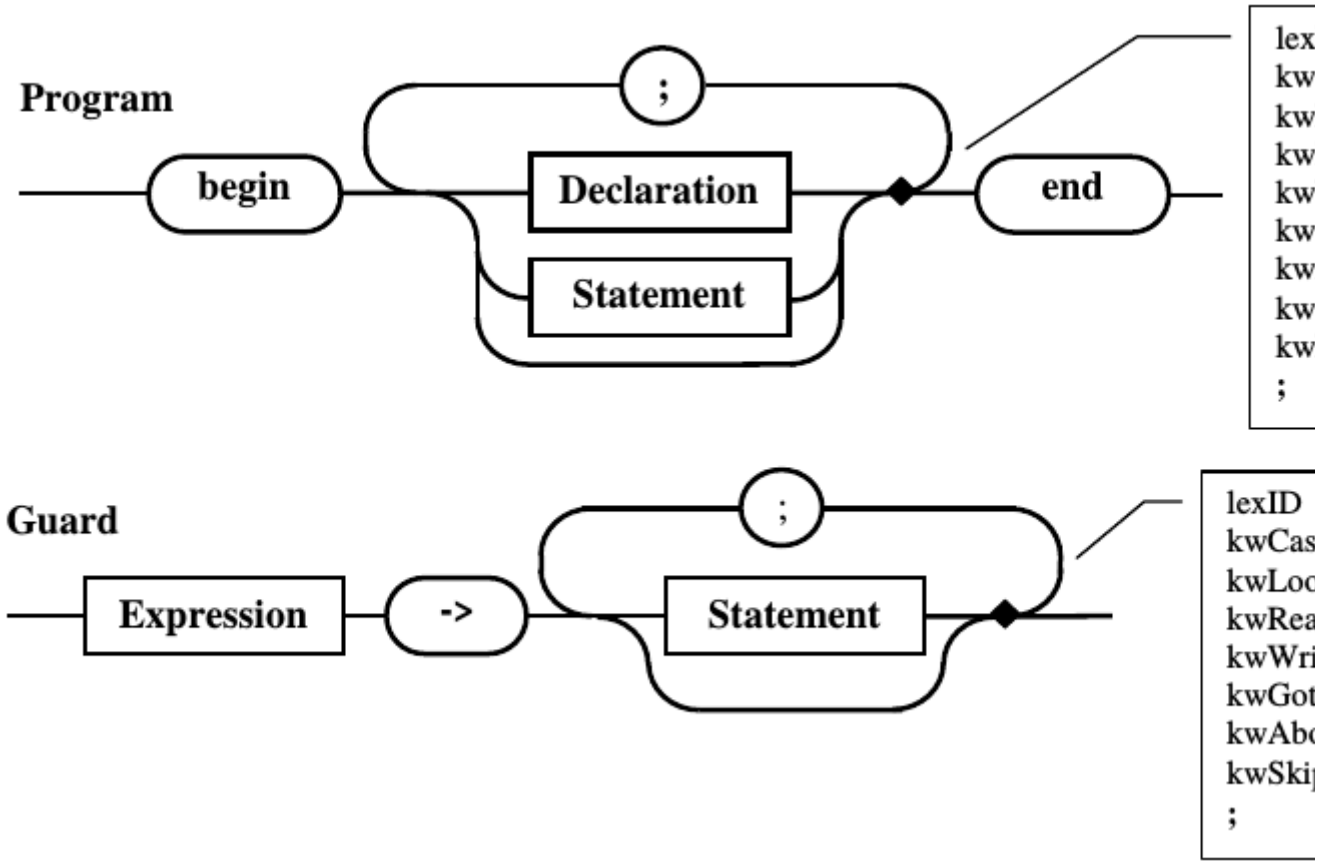


Рис. 10.9. Альтернативы правил ***Program*** и ***Guard***, полученные после анализа перетекания из правила ***Statement***

Вместе с тем, следует отметить, что в правиле рассмариваемая альтернативная точка оказыается на конце правила и следовательно оказывет влияния на альтернативные точки правила, задающего множество охраняемых (***Guard_set***). Последнее, в свою очередь, добавляет к имеющим альтернативам лексему, задающую ключевое слово **or**, а рассматриваемая альтернативная точка вновь оказывается на конце правила. Поэтому, дополнительно анализируем правила задающие операторы выбора и цикла, которые уже не позволяют вновь оказаться альтернативной точке в конце правил. Результаты этого анализа представлены на рис. 10.10 и показывают на однозначность альтернатив. Это, в свою очередь подтверждает принадлежность грамматики демонстрационного языка программирования к КС(1) виду.

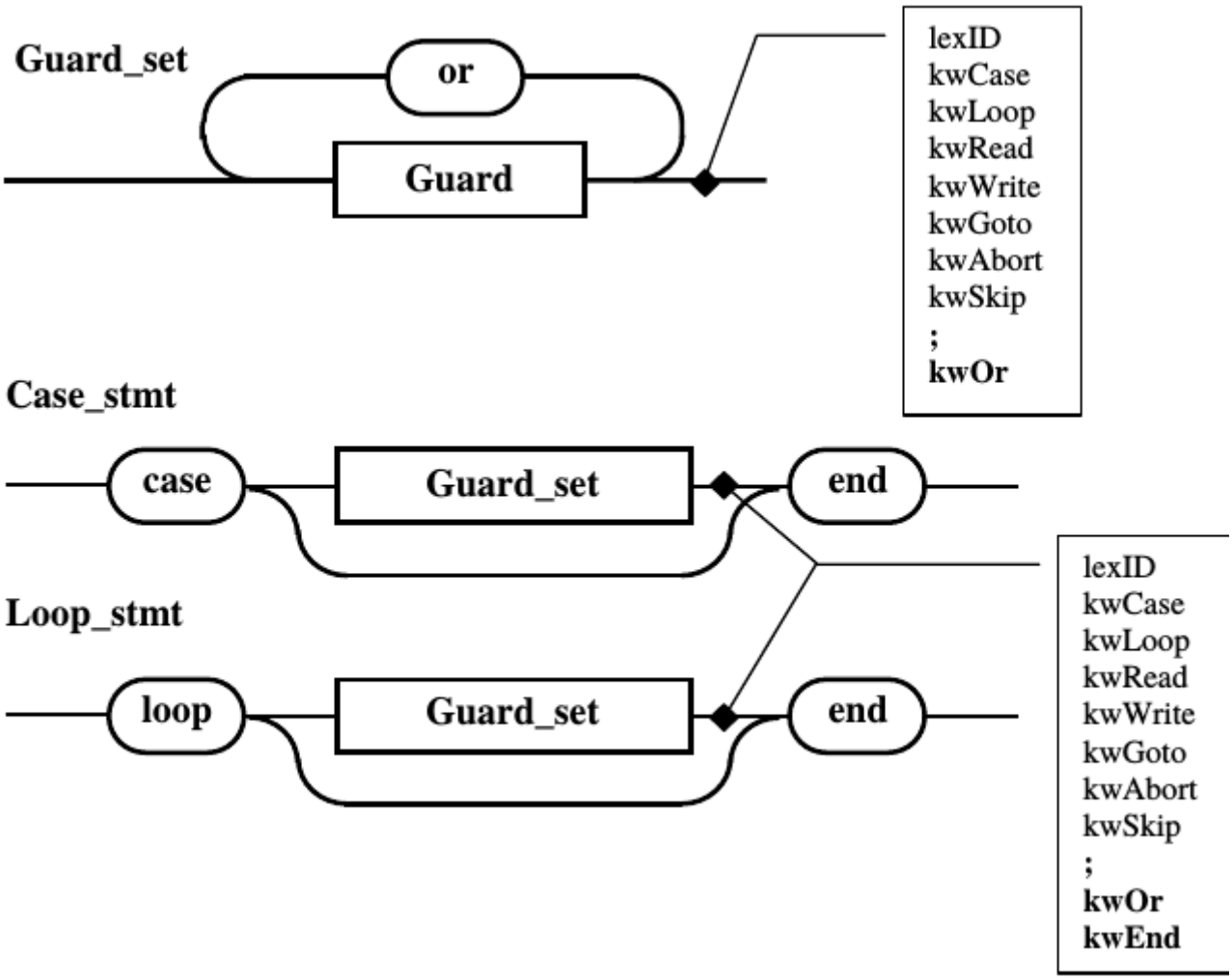


Рис. 10.10. Альтернативы правил ***Guard_set***, ***Case_stmt*** и ***Loop_stmt*** полученные после анализа перетекания из правила ***Guard***

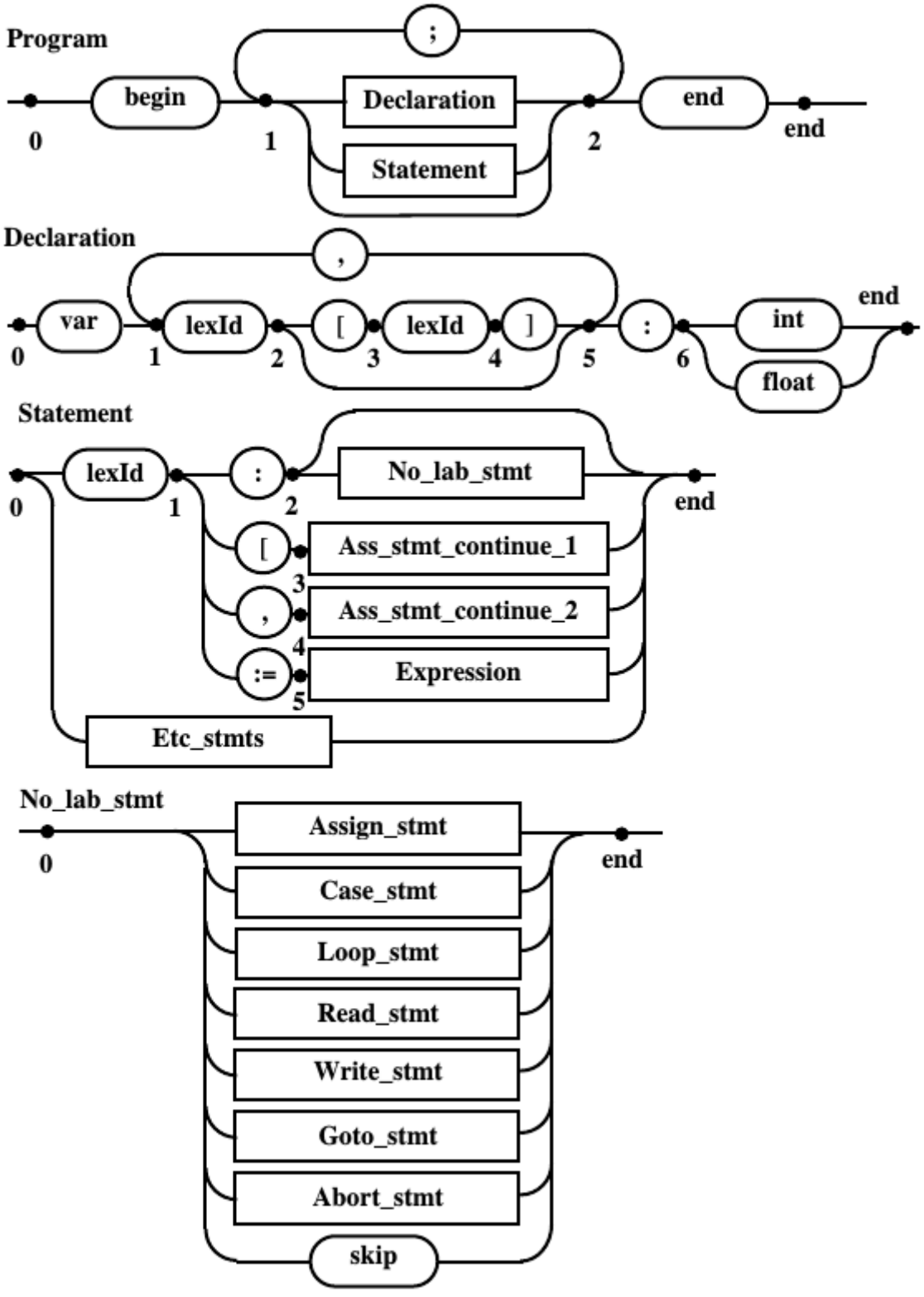
Получение окончательных диаграмм Вирта демонстрационного языка

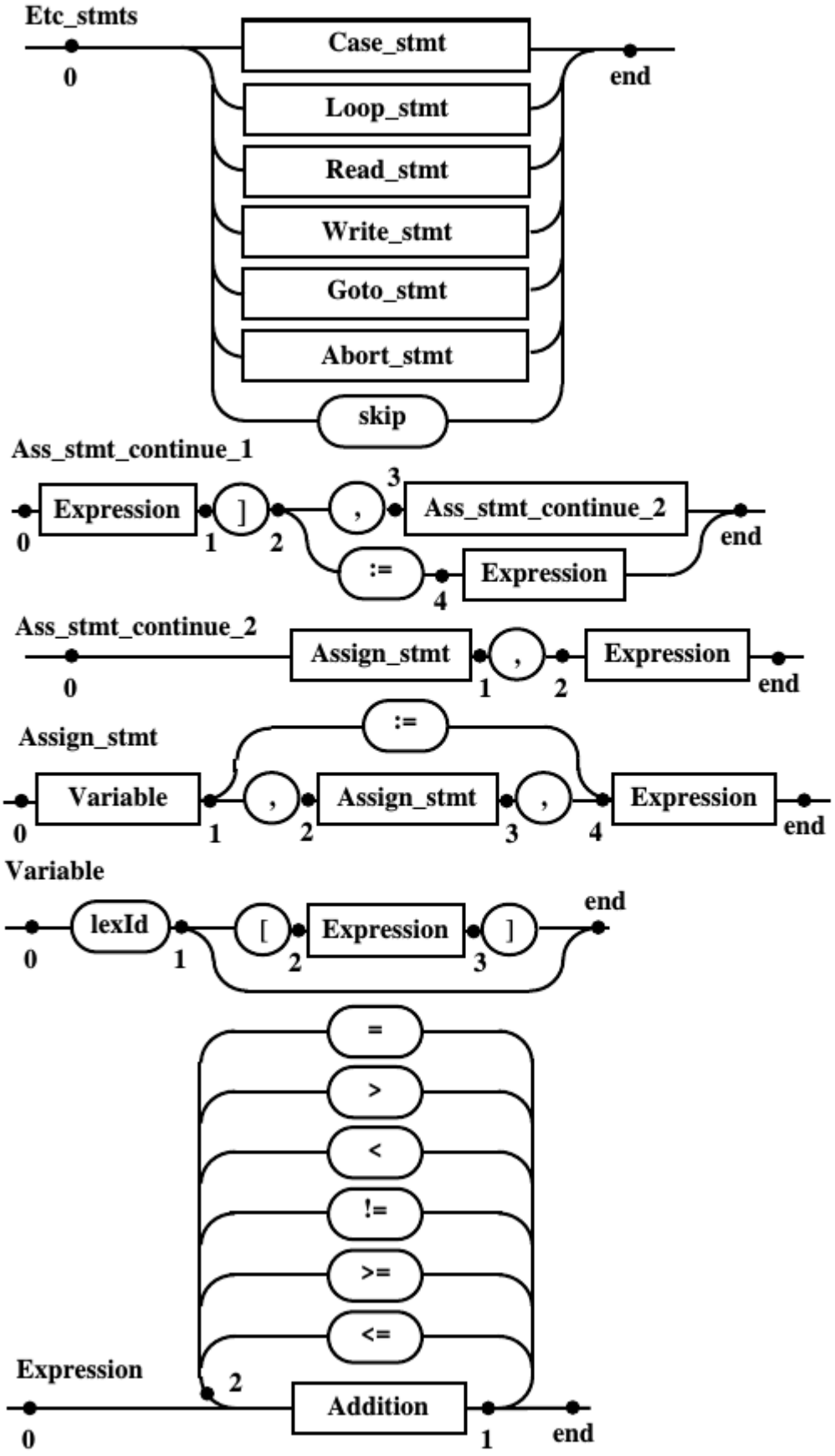
После проведенного анализа получаем диаграмми Вирта, которые можно исползовать для программной реализации модуля, выполняющего функции распознавателя. Дополнительным штрихом является явная расстановка на диаграммах альтернативных точек, полученных в ходе анализа на принадлежность к КС(1) грамматике. Это позоляет увидеть все состояния формирумой модели распознавателя на основе динамически порождаемых конечных автоматов.

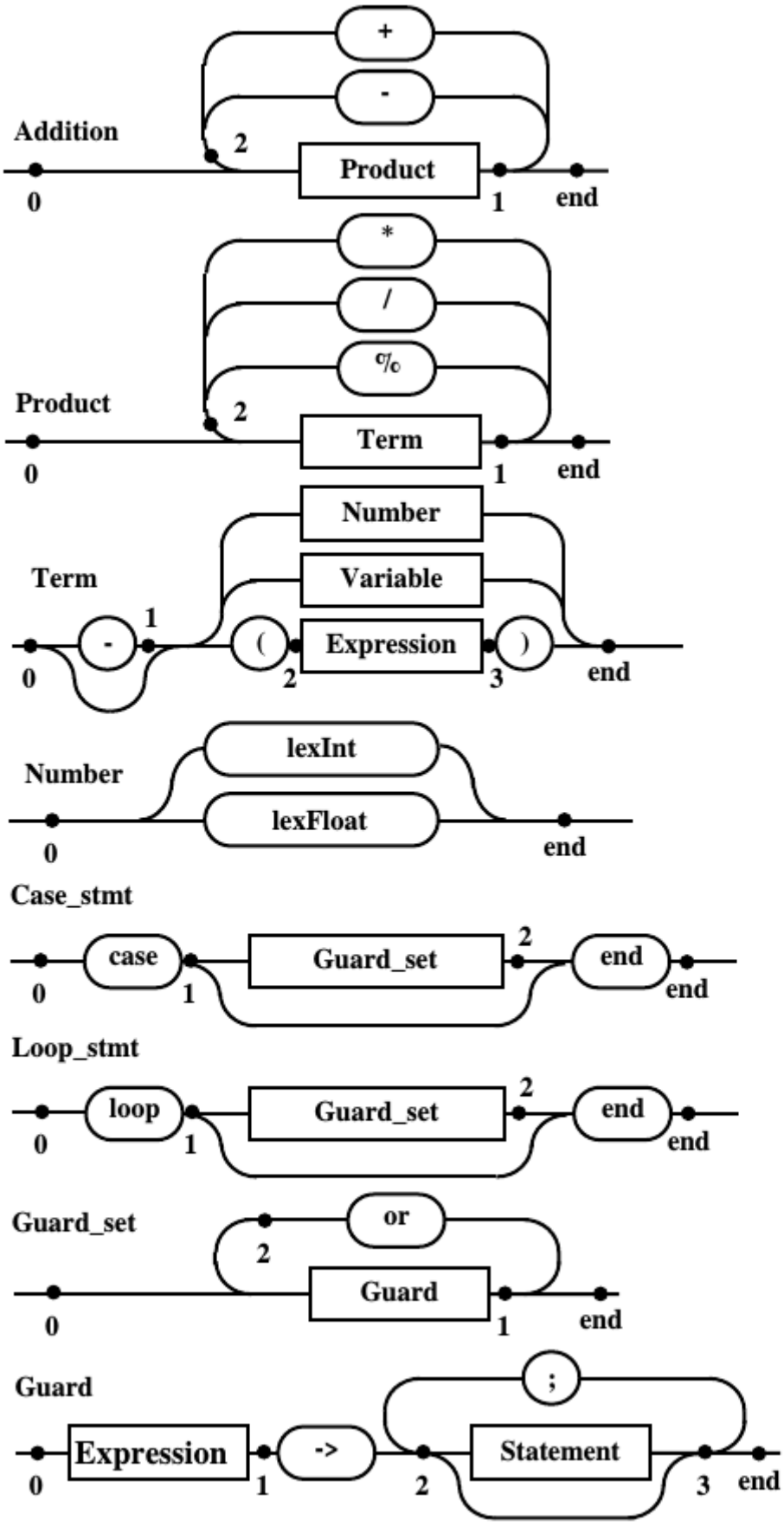
Метод разметки альтернативных точек не играет особого значения. Однако можно заметить, что в рамках предлагаемого подхода к реализации конечного автомата они играют в программе роль меток. Поэтому, желательно, чтобы обозначения этих точек лего преобразовывались в имена меток и, в тоже время, обеспечивали удобную идентификацию внутри диаграмм. В рамках учебного пособия все альтернативные точки нумеруются числами, начиная с нуля. При этом нулевое значение используется для начальной связи диаграммы

Вирта. Выходная связь диаграммы помечается именем **end**, что позволяет не привязываться к числу состояний порождаемого конечного автомата.

Окончательные диаграммы Вирта, описывающие синтаксис демонстрационного языка программирования DPL




Etc_stmts
0
Case_stmt
Loop_stmt
Read_stmt
Write_stmt
Goto_stmt
Abort_stmt
skip
end
Ass_stmt_continue_1
Expression
]
,
:=
Ass_stmt_continue_2
Expression
0
1
2
3
4
end
Ass_stmt_continue_2
Assign_stmt
,
Expression
0
1
2
end
Assign_stmt
:=
Variable
,
Assign_stmt
,
Expression
0
1
2
3
4
end
Variable
lexId
[
Expression
]
0
1
2
3
end
=
>
<
!=
>=
<=
Expression
2
Addition
0
1
end


Addition
+
-
2
Product
0
1
end
Product
*
/
%
2
Term
0
1
end
Term
Number
Variable
1
-
(
Expression
)
0
2
3
end
Number
lexInt
lexFloat
0
end
Case_stmt
case
Guard_set
2
end
0
1
end
Loop_stmt
loop
Guard_set
2
end
0
1
end
Guard_set
or
2
Guard
0
1
end
Guard
;
Expression
->
Statement
0
1
2
3
end

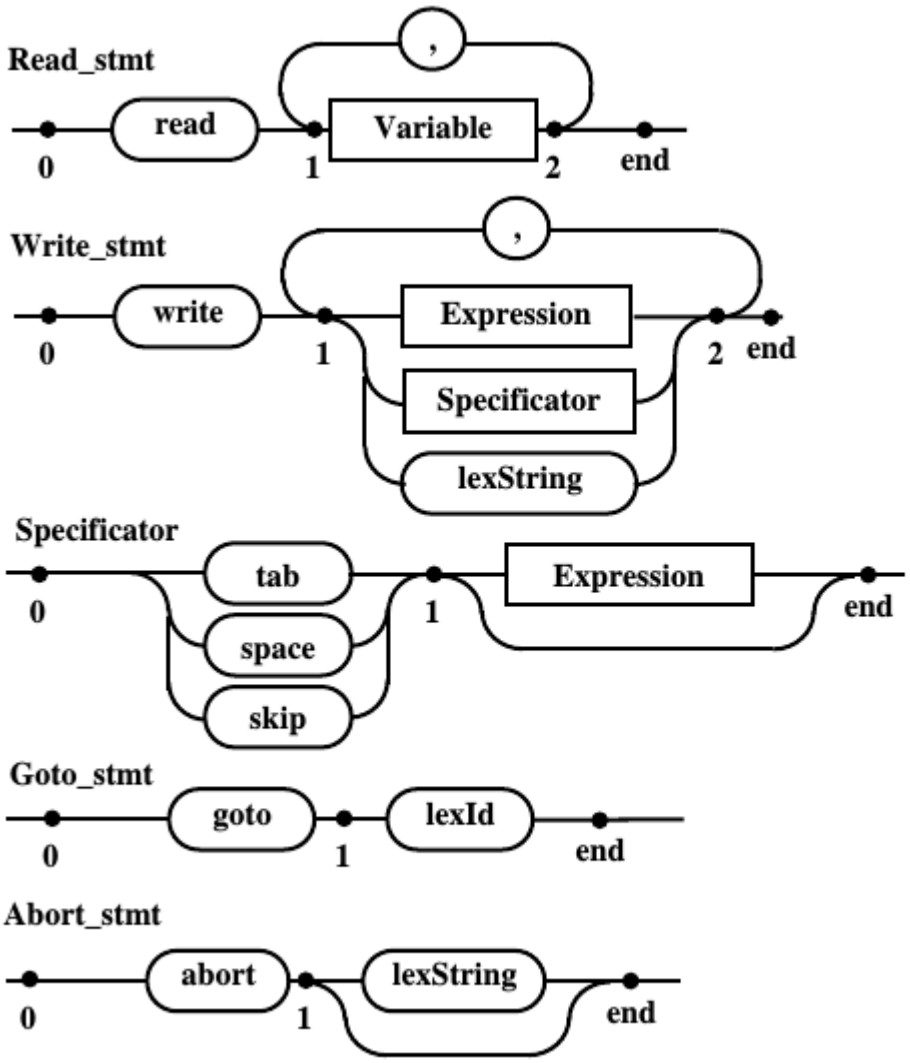


Рис. 10.d. Диаграммы Вирта предназначенные для написания программы распознавателя

Написание кода по разработанным диаграммам Вирта

Разработка программного кода, реализующего распознаватель осуществляется по простой методике, заключающейся практически в механическом отображении построенных диаграмм Вирта в функции используемого языка программирования. Нюансы такого преобразования исходной модели в реально работающий программный модуль определяются языком и избранной технологией программирования. Ниже рассматривается прямое отображение диаграмм Вирта в соответствующие конечные автоматы. В качестве примера приводятся отдельные диаграммы и функции написанные по ним.

На рис. 10.11 приведена диаграмма правила описывающего программу.

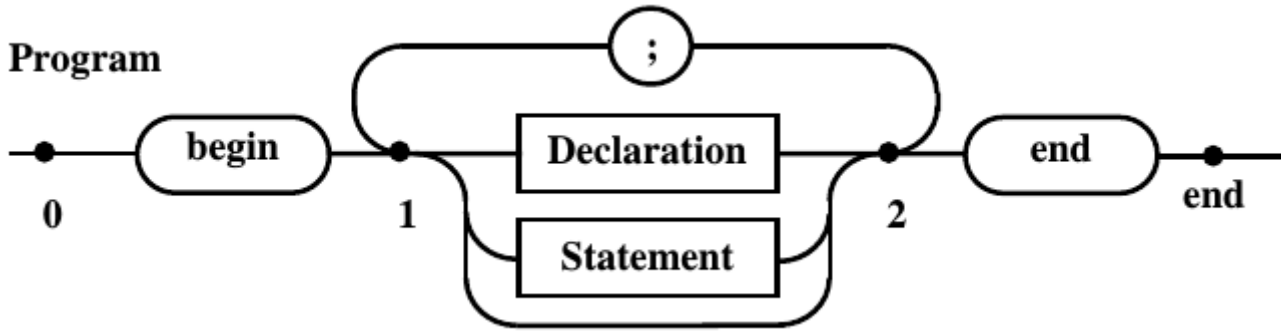


Рис. 10.11. Диаграмма Вирта, задающая правило ***Program***

Функция на языке C++, реализующая распознаватель для этого правила, выгляди следующим образом.

```
bool Program() {
//_0:
    if(lc==kwBegin) {nxl(); goto _1;}
    return false;
  _1:
    if(Declaration()) {goto _2;}
    if(Statement()) {goto _2;}
    if(lc==lexSemicolon) {nxl(); goto _1;}
    if(lc==kwEnd) {nxl(); goto _end;}
    er(7); return false;
  _2:
    if(lc==lexSemicolon) {nxl(); goto _1;}
    if(lc==kwEnd) {nxl(); goto _end;}
    er(7); return false;
  _end:
    er(7); return false;
  _end:
    return true;
}
```

Приведенный код показывает, что каждое правило преобразуется в отдельную функцию-автомат, возвращающую булевое значение, означающее допуск обрабатываемой подцепочки языка (***true***) или отказ (***false***). Сосотяния автоматов, задаваемые числами на диаграммах Вирта преобразуются в соответствующие метки. Конечное состояние преобразовано в метку ***end***.

После метки, задающей состояние, расположены условные операторы, каждый из которых задает один из переходов. Условие перехода определяется внутри условного оператора. Сам переход осуществляется оператором безусловного перехода размещаемым в его теле. Переход осуществляется по выполнению условия в котором проверяется совпадение с лексемой или нетерминалом.

Если переход из состояния должен происходить по лексеме, то перед ним осуществляется взятие следующей лексемы, так как предыдущая оказывается правильно обработанной. При переходе по нетерминалу, задаваемому вызовом функции, в которой реализована соответствующая диаграмма Вирта, взятие следующей лексемы не происходит. Это объясняется тем, что проверка лексемы, а следоваетльно и замена ее другой осуществляются внутри функций вложенных в вызывающую функцию только в том случае, когда происходит

непосредственная проверка значения лексемы.

Следует отметить, что в общепринятой практике программирование с использованием оператора ***goto*** считается дурным тоном. Однако в данном случае построение кода осуществляется механически по диаграммам Вирта, которые служать основным документам. Именно эта особенность позволяет на наш взгляд более эффективно работать с синтаксисом, особенно при разработке новых языков программирования.

При нежелании использовантъ оператор безусловного перехода можно струкутрировать имеющиеся диаграммы Вирта или использовать в качестве базового метаязыка расширенные формы Бэкуса-Наура. Однако в этом случае затрудняется исследование свойств грамматики разрабатываемого языка.