Тема 6. Генерация промежуточного кода

6.8. *Метод обратных поправок для оператора* case (switch)

(2-е издание Ахо раздел 6.8 или 1-е издание раздел 8.5)

Операторы выбора (switch, case) доступны во многих языках, например, один из возможных синтаксисов оператора выбора:

switch (*E***) {**
    case V_1 : S_1
    case V_2 : S_2
    …
    case V_{n-1} : S_{n-1}
    default : S_n
}

Имеется вычисляемое выражение-селектор E, за которым следуют n константных значений V_1, V_2, ..., V_n, которые может принимать выражение, а также, возможно, значение по умолчанию (**default**), которое считается всегда соответствующим значению E в том случае, когда оно не равно ни одному другому значению.

Трансляция оператора выбора должна состоять в следующем.

1. Вычисление выражения E.

2. Поиск в списке вариантов значения V_j, которое равно значению выражения. Напомним, что значение по умолчанию соответствует выражению, если ему не соответствует ни одно явно указанное значение V_j.

3. Выполнение инструкции S_j, связанной с найденным значением.

Шаг 2 представляет собой n-путевое ветвление, которое может быть реализовано одним из нескольких способов. Если количество вариантов невелико, скажем, не более 10, то имеет смысл воспользоваться последовательностью условных переходов, каждый из которых выполняет проверку на равенство одному из значений и передает управление соответствующему коду при совпадении.

Компактный способ реализации такой последовательности условных переходов состоит в создании таблицы пар, каждая из которых состоит из значения и метки кода соответствующей инструкции. Вычисленное значение самого выражения в паре с меткой для инструкции по умолчанию помещается в конце таблицы во время выполнения программы. Затем компилятором генерируется простой цикл, который сравнивает значение выражения с каждым значением из таблицы, которая гарантирует, что если не найдется другого соответствующего значения, то будет выполнена инструкция по умолчанию.

Если количество значений превышает 10 или около того, более эффективным способом является построение хеш-таблицы значений с метками для разных значений в качестве записей. Если в таблице не находится запись для вычисленного значения, генерируется переход к инструкции по умолчанию.

Существует распространенный частный случай, который может быть реализован еще более эффективно, чем *n*-путевое ветвление. Если все возможные значения лежат в некотором небольшом диапазоне, скажем, от *min* до *max*, и количество различных значений представляет собой существенную долю от *max* – *min* + 1, то можно построить массив из *max* – *min* + 1 блоков, где блок *j* – *min* (если индексация массива начинается с нуля) содержит метку инструкции для значения *j*; все блоки, которые остаются незаполненными при этом процессе, заполняются метками инструкции по умолчанию.

Для выполнения выбора вычисляется значение выражения *j*. Затем убеждаемся в том, что это значение находится в диапазоне от *min* до *max*, и выполняем косвенный переход с использованием записи таблицы со смещением *j* – *min*. Например, если выражение имеет символьный тип, может быть создана таблица из, скажем, 128 записей (зависит от используемого множества символов), и переход может выполняться даже без проверки вычисленного значения на принадлежность диапазону.

Рассмотрим способы трансляции.

Прямой способ – формирование трехадресного кода по аналогии с вложенными операторами **if then else**. Создается достаточно простой промежуточный код, показанный на рис. 1, однако компилятор должен выполнить обширный анализ для поиска наиболее эффективной реализации.

```
        Код вычисления E в t
        if t ≠ V1 goto L1
        Код для S1
        goto next
L1:     if t ≠ V2 goto L2
        Код для S2
        goto next
L2:
        …
Ln–2:   if t ≠ Vn–1 goto Ln–1
        Код для Sn–1
        goto next
Ln–1:   Код для Sn
next:
```

```
switch (E) {
    case V1 : S1
    case V2 : S2
    …
    case Vn–1 : Sn–1
    default : Sn
}
```

Рис. 1

Более удобно транслировать эту конструкцию в промежуточный код, представленный на рис. 2.

switch (*E***) {**
    case V_1 : S_1
    case V_2 : S_2
    …
    case V_{n-1} : S_{n-1}
    default : S_n
}

Код вычисления E в t
    goto *test*
L_1: Код для S_1
    goto *next*
L_2: Код для S_2
    goto *next*
    …
L_{n-1}: Код для S_{n-1}
    goto *next*
L_n: Код для S_n
    goto *next*
test: **if** $t = V_1$ **goto** L_1
    if $t = V_2$ **goto** L_2
    …
    if $t = V_{n-1}$ **goto** L_{n-1}
    goto L_n
next:

Рис. 2

Все проверки оказываются в конце, так что генератор кода может распознать многопутевое разветвление программы и сгенерировать для него эффективный код, основанный на одной из описанных технологий. Заметим, что размещение кода ветвления в начале неудобно, поскольку тогда компилятор не может строить код для каждого S_i в момент его появления.

Для трансляции ключевого слова **switch** создаем две новые метки *test* и *next* и новую переменную *t*. Затем, по мере разбора выражения *E*, генерируется код для вычисления *E* в *t*. После обработки *E* генерируем переход **goto** *test*.

Затем, когда появляется ключевое слово **case**, создаем новую метку L_i. В очередь, используемую исключительно для хранения информации о значениях **case**, мы помещаем метку и значение V_i константы при **case**.

Обрабатываем каждую инструкцию **case** V_i: S_i путем генерации новой метки L_i, за которой следует код для S_i с последующим переходом **goto** *next*. После того как появится завершающее конструкцию ключевое слово **end**, мы готовы сгенерировать код ветвления. Считывая пары указатель-значение из очереди, мы можем генерировать последовательность трехадресных инструкций вида

case V_1 L_1
case V_2 L_2
…
case V_{n-1} L_{n-1}
case t L_n
next **:**

где t – имя, хранящее значение селектора E, а L_n – метка инструкции по умолчанию. Трехадресная инструкция **case** V_i L_i является синонимом **if** $t = V_i$ **goto** L_i (рис. 2), однако использование **case** облегчает генератору целевого кода задачу нахождения потенциальных кандидатов для специальной обработки. На стадии генерации кода последовательность инструкций **case** может быть транслирована в n-путевое ветвление наиболее эффективного вида, в зависимости от количества значений и их размещения в небольшом диапазоне.

Для построения СУО рассмотрим следующий ситаксис оператор варианта.

РБНФ оператора:

ОператорВарианта = "**case**" Выражение "**of**" ЭлемСпВар { "**;**" ЭлемСпВар }
[**default** Оператор] **end**
ЭлемСпВар = СписМеток "**:**" Оператор
СписМеток = МеткаВар { "**,**" МеткаВар }
МеткаВар = Константа

Упростим синтаксис (запретим список вариантов):

ОператорВарианта = "**case**" Выражение "**of**" ЭлемСпВар { "**;**" ЭлемСпВар }
[**default** Оператор] **end**
ЭлемСпВар = МеткаВар "**:**" Оператор
МеткаВар = Константа (**num**, если ограничим целым типом)

Формальная грамматика:

OpCase → **case** *Expr* **of** *LstCase Def* **end**
LstCase → *ElLst* | *LstCase* **;** *ElLst*
ElLst → **num :** *Stmt*
Def → **default** *Stmt* | ε

или в сокращенном виде:

S → **case** *E* **of** *V D* **end**
V → *C* | *V* **;** *C*
C → **num :** *S*
D → **def** *S* | ε

или после замены вхождений:

S → **case** *E* **of** *V D* **end**
V → **num :** *S* | *V* **; num :** *S*
D → **def** *S* | ε

Построим СУО для генерации промежуточного кода в соответствии с рис. 2, т. е

Код вычисления E в t
goto *test*
L_1: Код для S_1
goto *next*
L_2: Код для S_2
goto *next*
…
L_{n-1}: Код для S_{n-1}
goto *next*
L_n: Код для S_n
goto *next*
test: **if** $t = V_1$ **goto** L_1
if $t = V_2$ **goto** L_2
…
if $t = V_{n-1}$ **goto** L_{n-1}
goto L_n
next:

СУО для трансляции оператора варианта методом обратных поправок (рис. 2)

Продукция	Семантические правила
1) $L \to L_1$; M S	*BackPatch* (*L*$_1$**.***nextlist*, *M***.***instr*); *L***.***nextlist* := *S***.***nextlist*
2) $L \to S$	*L***.***nextlist* := *S***.***nextlist*
3) $S \to$ **case** E **of** N V D **end**	*BackPatch* (*N***.***nextlist*, *nextinstr*) //индекс метки *test* **while** $Q \neq \varnothing$ **do** (c, a) $\Leftarrow Q$ **if** $Q \neq \varnothing$ **then** *Gen* ('if' *E***.***addr* '=' c 'goto' a) **else** *Gen* ('goto ?') **end** *BackPatch* (*Merge* (*V***.***nextlist*, *D***.***nextlist*), *nextinstr*) //индекс метки *next* *S***.***nextlist* := **null**
4) $V \to$ **num :** M S	$Q \Leftarrow$ (**num.***val*, *M***.***instr*) *V***.***nextlist* := *MakeList* (*nextinstr*) *V***.***nextlist* := *Merge* (*V***.***nextlist*, *S***.***nextlist*) *Gen* ('goto ?') //переход на метку *next*
5) $V \to V_1$ **; num :** M S	$Q \Leftarrow$ (**num.***val*, *M***.***instr*) *V***.***nextlist* := *MakeList* (*nextinstr*) *tmp* := *Merge* (*V*$_1$**.***nextlist*, *V***.***nextlist*) *V***.***nextlist* := *Merge* (*tmp*, *S***.***nextlist*) *Gen* ('goto ?') //переход на метку *next*

6) $D \to$ **def** $M\ S$	$Q \Leftarrow (\#, M.instr)$ $D.nextlist := MakeList(nextinstr)$ $D.nextlist := Merge(D.nextlist, S.nextlist)$ $Gen('\text{goto ?}')$ //переход на метку *next*
7) $D \to \varepsilon$	$D.nextlist :=$ **null**
8) $M \to \varepsilon$	$M.instr := nextinstr$
9) $N \to \varepsilon$	$N.nextlist := MakeList(nextinstr)$ $Gen('\text{goto ?}')$ //переход на метку *test* $Q := \varnothing$ //пустая очередь

В продукции 6 в очередь заносится пара (#, *M.instr*), где символ # означает произвольное значение, поскольку значение селектора здесь не нужно, оператор *S* выполняется при любых других значениях селектора.

Очередь *Q* – глобальная. Для вложенных **case** надо подумать о хранении в очереди каких-либо маркеров, чтобы отличать, с каким **case** идет работа.

Другой вариант – нетерминалу *N* добавить синтезируемый атрибут (указатель на очередь), например, ***N.queue***, должна быть функция создания новой пустой очереди, например, ***N.queue*** := *NewQueue*(). Для добавления в очередь соответствующих пар значений (*значение*, *адрес*), чтобы была доступна очередь в соответствующих продукциях, нетерминалу *V* добавить наследуемый атрибут (указатель на очередь), например, ***V.que*** (здесь название атрибута лучше дать другое, а не *queue*, чтобы визуально их отличать, поскольку *queue* – синтезируемый атрибут, а *que* – наследуемый).

В СУО в кодах для ветвления формируются команды вида **if** $t = V_i$ **goto** L_i. Можно легко изменить соответствующие семантические правила для генерации трехадресных команд вида **case** V_i L_i.

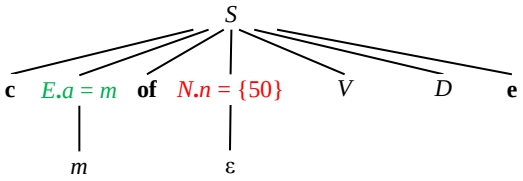
Рассмотренное СУО лично проверил, реализовав программно, все работает.

Рассмотрим пример аннотированного дерева разбора для оператора

case *m* **of**
    1 : *b* := *c* **;**
    2 : *b* := *d*
    def *b* := *h*
end

Отсчет номеров позиций команд начнем с 50. На аннотированном дереве разбора для компактности ключевые слова обозначены: **case** – **c**, **def** – **d**, **end** – **e**, атрибуты: *addr* – *a*, *instr* – *i*, *nextlist* – *n*, значения атрибутов *nextlist* показываются как содержимое списков, вместо **id.***pnt* указан сам идентификатор, а вместо **num.***pnt* – значение числовой константы.

case *m* **of** 1 : *b* := *c* **;** 2 : *b* := *d* **def** *b* := *h* **end**



50: **goto ?**
51:

$Q = \varnothing$

Продукция	Семантические правила
3) $S \to$ **case** E **of** $N\ V\ D$ **end**	
9) $N \to \varepsilon$	*N.nextlist* := *MakeList* (*nextinstr*) *Gen* ('goto ?') //переход на метку *test* $Q := \varnothing$ //пустая очередь

Из СУО для арифметических выражений

Продукция	Семантические правила
$E \to T$	*E.addr* := *T.addr*
$T \to F$	*T.addr* := *F.addr*
$F \to$ **id**	*F.addr* := **id**.*pnt*

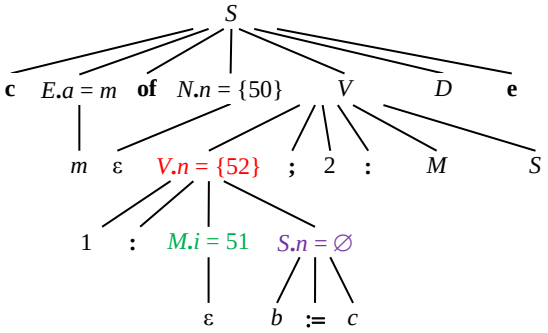
В соответствии с СУО для арифметических выражений значением *E.addr* становится **id**.*pnt* идентификатора *m* (промежуточные замены типа $E \Rightarrow T \Rightarrow F \Rightarrow$ **id** в дереве не показаны).

С помощью маркера *N* (продукция 9) в атрибуте *N.nextlist* сохраняется текущее значение *nextinstr*, равное 50, формируется команда

```
50: goto ?
```

и очередь *Q* делается пустой.

case *m* **of** 1 : *b* := *c* **;** 2 : *b* := *d* **def** *b* := *h* **end**



50: **goto ?**
51: b := c
52: **goto ?**
53:

Q = (1,51)

Продукция	Семантические правила
4) $V \to$ **num :** *M S*	*Q* ⇐ (**num.***val*, *M.instr*) *V.nextlist* := *MakeList* (*nextinstr*) *V.nextlist* := *Merge* (*V.nextlist*, *S.nextlist*) *Gen* ('goto ?') //переход на метку *next*
5) $V \to V_1$ **;** **num :** *M S*	*Q* ⇐ (**num.***val*, *M.instr*) *V.nextlist* := *MakeList* (*nextinstr*) *tmp* := *Merge* (V_1*.nextlist*, *V.nextlist*) *V.nextlist* := *Merge* (*tmp*, *S.nextlist*) *Gen* ('goto ?') //переход на метку *next*
8) $M \to \varepsilon$	*M.instr* := *nextinstr*

Из СУО для арифметических выражений

Продукция	Семантические правила
$S \to$ **id** := *E*	*Gen* (**id.***pnt* ':=' *E.addr*) *S.nextlist* := **null**

С помощью маркера *M* (продукция 8) в атрибуте *M.instr* сохраняется текущее значение *nextinstr*, равное 51.

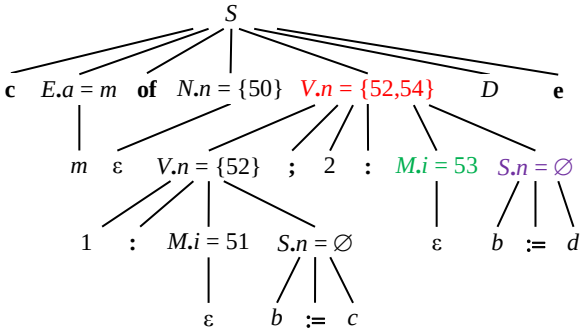
В соответствии с продукцией $S \to$ **id** := *E* из СУО для арифметических выражений формируется команда:

```
51: b := c
```

В соответствии с продукцией 4 в *Q* добавляется (1,51), устанавливается *V.nextlist* = *Merge* ({52}, ∅) = {52}, формируется команда

```
52: goto ?
```

case *m* **of** 1 : *b* := *c* **;** 2 : *b* := *d* **def** *b* := *h* **end**



50: **goto ?**
51: b := c
52: **goto ?**
53: b := d
54: **goto ?**
55:

Q = (1,51), (2,53)

Продукция	Семантические правила
5) $V \to V_1$ **;** **num** **:** *M S*	*Q* ⇐ (**num**.*val*, *M*.*instr*) *V*.*nextlist* := *MakeList* (*nextinstr*) *tmp* := *Merge* (V_1.*nextlist*, *V*.*nextlist*) *V*.*nextlist* := *Merge* (*tmp*, *S*.*nextlist*) *Gen* ('goto ?') //переход на метку *next*
8) $M \to \varepsilon$	*M*.*instr* := *nextinstr*

Из СУО для арифметических выражений

Продукция	Семантические правила
$S \to$ **id** := *E*	*Gen* (**id**.*pnt* ':=' *E*.*addr*) *S*.*nextlist* := **null**

С помощью маркера *M* (продукция 8) в атрибуте *M*.*instr* сохраняется текущее значение *nextinstr*, равное 53.

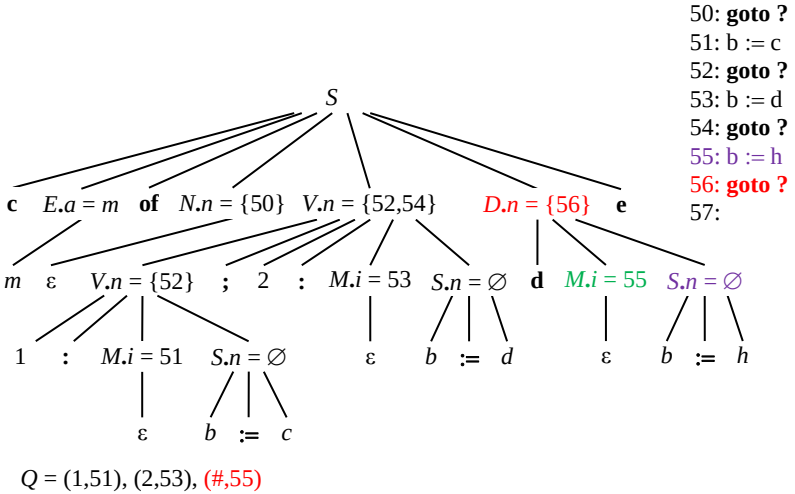
В соответствии с продукцией $S \to$ **id** := *E* из СУО для арифметических выражений формируется команда:

```
53: b := d
```

В соответствии с продукцией 5 в *Q* добавляется (2,53), устанавливается *V*.*nextlist* = *Merge* ({52},{54},∅) = {52, 54}, формируется команда

```
54: goto ?
```

case *m* **of** 1 : *b* := *c* **;** 2 : *b* := *d* **def** *b* := *h* **end**



Q = (1,51), (2,53), (#,55)

Продукция	Семантические правила
6) $D \to$ **def** *M S*	*Q* ⇐ (#, *M.instr*) *D.nextlist* := *MakeList* (*nextinstr*) *D.nextlist* := *Merge* (*D.nextlist*, *S.nextlist*) *Gen* ('goto ?') //переход на метку *next*
8) $M \to \varepsilon$	*M.instr* := *nextinstr*

Из СУО для арифметических выражений

Продукция	Семантические правила
$S \to$ **id** := *E*	*Gen* (**id**.*pnt* ':=' *E.addr*) *S.nextlist* := **null**

С помощью маркера *M* (продукция 8) в атрибуте *M.instr* сохраняется текущее значение *nextinstr*, равное 55.

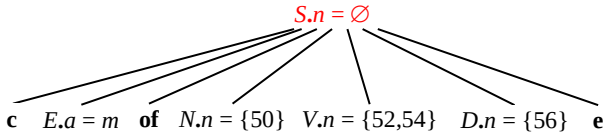
В соответствии с продукцией $S \to$ **id** := *E* из СУО для арифметических выражений формируется команда:

```
55: b := h
```

В соответствии с продукцией 6 в *Q* добавляется (#,55), устанавливается *D.nextlist* = *Merge* ({56},∅) = {56}, формируется команда

```
56: goto ?
```

case *m* **of** 1 : *b* := *c* **;** 2 : *b* := *d* **def** *b* := *h* **end**



Q = (1,51), (2,53), (#,55)

50: **goto 57**
51: b := c
52: **goto 60**
53: b := d
54: **goto 60**
55: b := h
56: **goto 60**
57:

Продукция	Семантические правила
3) $S \rightarrow$ **case** *E* **of** *N V D* **end**	*BackPatch* (*N.nextlist*, *nextinstr*) **while** $Q \neq \varnothing$ **do** (*c*, *a*) $\Leftarrow$ *Q* **if** $Q \neq \varnothing$ **then** *Gen* ('if' *E.addr* '=' *c* 'goto' *a*) **else** *Gen* ('goto' *a*) **end** *BackPatch* (*Merge* (*V.nextlist*, *D.nextlist*), *nextinstr*) *S.nextlist* := **null**

К данному моменту завершено формирование кода для *V* и *D* продукции 3. Атрибуты имеют следующие значения: *N.nextlist* = {50}, *V.nextlist* = {52,54}, *D.nextlist* = {56}, *E.addr* = *m*, *nextinstr* = 57, очередь *Q* содержит пары (1,51), (2,53), (#,55).

В результате выполнения процедуры *BackPatch* ({50}, 55) команда 50 получит целевую метку 57.

В цикле **while** формируются команды

```
57: if m = 1 goto 51
58: if m = 2 goto 53
59: goto 55
60:
```

В результате выполнения процедуры *BackPatch* ({52,54,56}, 60) команды 52, 54 и 56 получат целевую метку 60.

Списком *S.nextlist* становится пустой список.

Таким образом, для оператора

case *m* **of**
1 : *b* := *c* ;
2 : *b* := *d*
def *b* := *h*
end

будет сформирован следующий трехадресный код:

```
50: goto 57
51: b := c
52: goto 60
53: b := d
54: goto 60
55: b := h
56: goto 60
57: if m = 1 goto 51
58: if m = 2 goto 53
59: goto 55
60:
```

Код вычисления E в t
goto *test*
L_1: Код для S_1
goto *next*
L_2: Код для S_2
goto *next*
…
L_{n-1}: Код для S_{n-1}
goto *next*
L_n: Код для S_n
goto *next*
test: **if** $t = V_1$ **goto** L_1
if $t = V_2$ **goto** L_2
…
if $t = V_{n-1}$ **goto** L_{n-1}
Код для S_n **goto** L_n
next:

Рис. 3

Далее не из Ахо, а моя модификация.

Можно несколько изменить структуру генерируемого кода (рис. 3), а именно: для **default**-части вместо команды перехода **goto** L_n поместить сам код для оператора S_n (показано зеленым), а команды

L_n: Код для S_n
goto *next*

не формировать. Удаляемые команды выделены желтым. В результате будет сформировано на две команды безусловного перехода меньше (исключаются одна команда **goto** *next* и команда **goto** L_n).

Для реализации следует весь код от начала до команды

if $t = V_{n-1}$ **goto** L_{n-1}

включительно сформировать до начала разбора **default**-части, а не в конце продукции, как это делалось ранее.

Ниже приведено соответствующее СУО.

СУО для трансляции оператора варианта методом обратных поправок (рис. 3)

Продукция	Семантические правила
1) $L \to L_1$; M S	*BackPatch* (*L*$_1$.*nextlist*, *M*.*instr*); *L*.*nextlist* := *S*.*nextlist*
2) $L \to S$	*L*.*nextlist* := *S*.*nextlist*
3) $S \to$ **case** E **of** N V D **end**	//перед *D* *BackPatch* (*N*.*nextlist*, *nextinstr*) //индекс метки *test* **while** $Q \neq \varnothing$ **do** (*c*, *a*) ⇐ *Q* *Gen* ('if' *E*.*addr* '=' *c* 'goto' *a*) **end** //в конце продукции *BackPatch* (*Merge* (*V*.*nextlist*, *D*.*nextlist*), *nextinstr*) //индекс метки *next* *S*.*nextlist* := **null**
4) $V \to$ **num :** M S	*Q* ⇐ (**num**.*val*, *M*.*instr*) *V*.*nextlist* := *MakeList* (*nextinstr*) *V*.*nextlist* := *Merge* (*V*.*nextlist*, *S*.*nextlist*) *Gen* ('goto ?') //переход на метку *next*
5) $V \to V_1$ **; num :** M S	*Q* ⇐ (**num**.*val*, *M*.*instr*) *V*.*nextlist* := *MakeList* (*nextinstr*) *tmp* := *Merge* (*V*$_1$.*nextlist*, *V*.*nextlist*) *V*.*nextlist* := *Merge* (*tmp*, *S*.*nextlist*) *Gen* ('goto ?') //переход на метку *next*
6) $D \to$ **def** S	*D*.*nextlist* := *S*.*nextlist*

7) $D \rightarrow \varepsilon$	*D.nextlist* := **null**
8) $M \rightarrow \varepsilon$	*M.instr* := *nextinstr*
9) $N \rightarrow \varepsilon$	*N.nextlist* := *MakeList* (*nextinstr*) *Gen* ('goto ?') //переход на метку *test* *Q* := ∅ //пустая очередь

СУО вполне рабочее, проверено программной реализацией.

Для сравнения ниже приведено СУО (программно не проверял, можете попробовать) для генерации промежуточного кода в соответствии с рис. 1, т. е

Код вычисления E в t
if $t \neq V_1$ goto L_1
Код для S_1
goto *next*
L_1: **if** $t \neq V_2$ goto L_2
Код для S_2
goto *next*
L_2:
…
L_{n-2}: **if** $t \neq V_{n-1}$ goto L_{n-1}
Код для S_{n-1}
goto *next*
L_{n-1}: Код для S_n
next:

В приведенном ниже СУО (без части **default**) V**.***val* – наследуемый атрибут – указатель (*addr*) на переменную, где содержится значение выражения E.

СУО для трансляции оператора варианта методом обратных поправок (рис. 1)

Продукция	Семантические правила
$L \to L_1$; M S	*BackPatch* (*L_1.nextlist*, *M.instr*) *L.nextlist* := *S.nextlist*
$L \to S$	*L.nextlist* := *S.nextlist*
$S \to$ **case** E **of** V **end**	*V.val* := *E.addr* //перед *V* *BackPatch* (*V.nextlist*, *nextinstr*) *S.nextlist* := *MakeList* (*nextinstr*)
$V \to$ **num :** S	//перед *S* *tmp* := *MakeList* (*nextinstr*) *Gen* ('**if**' *V.val* '≠' **num.***val* '**goto ?**') //после *S* *V.nextlist* := *MakeList* (*nextinstr*) *Gen* ('**goto ?**') *BackPatch* (*tmp*, *nextinstr*) *V.nextlist* := *Merge* (*V.nextlist*, *S.nextlist*)

$V \to V_1$ **;** **num** **:** S	//перед S V_1**.***val* := V**.***val* *tmp* := *MakeList* (*nextinstr*) *Gen* (′**if**′ V**.***val* ′≠′ **num.***val* ′**goto ?**′) //после S V**.***nextlist* := *MakeList* (*nextinstr*) *Gen* (′**goto ?**′) *BackPatch* (*tmp*, *nextinstr*) *tmp* := *Merge* (V_1**.***nextlist*, V**.***nextlist*) V**.***nextlist* := *Merge* (*tmp*, S**.***nextlist*)
$M \to \varepsilon$	M**.***instr* := *nextinstr*