

Тема 6. Генерация промежуточного кода

6.3. Трансляция логических выражений

Логические выражения строятся с помощью логических операций (**or**, **and**, **not**), операций отношения типа $a > b$ и логических констант **true** и **false**. Важной их особенностью является возможность применения методов сокращенного вычисления. Если в выражении a **and** b (a и b логические выражения) установлено, что $a = \text{false}$, то без вычисления b можно сказать, что все выражение будет иметь значение **false**. Аналогично для a **or** b , если $a = \text{true}$, то и значение всего выражения будет **true**. Исключение составляют выражения с побочными действиями, например функция, изменяющая некоторую глобальную переменную. В этом случае может понадобиться полное вычисление выражения (это не лучший стиль программирования).

Трансляция логических выражений зависит от места их использования (синтаксического контекста).

Если логическое выражение используется в качестве условия в управляющих операторах, то обычно используются команды условных и безусловных переходов, которые в зависимости от логического условия передают управление в ту или иную позицию кода (само значение не вычисляется).

Если же логическое выражение используется в правой части оператора присваивания, то может формироваться трехадресный код для его вычисления, подобный коду для арифметических выражений. Можно также вместо вычисления логического значения использовать команды условных и безусловных переходов, когда значение выражения устанавливается в зависимости от достигнутой программой позиции.

Для определения синтаксического контекста использования логических выражений для обозначения арифметических и логических выражений будем использовать нетерминалы E и B соответственно.

СУО в табл. 11 иллюстрирует формирование трехадресного кода для логического выражения (используется инкрементная трансляция и предполагается, что выражение вычисляется полностью) по аналогии с арифметическим выражением.

Таблица 11

СУО для формирования трехадресного кода
для логического выражения (инкрементная трансляция)

| Продукция | Семантические правила |
|--------------------------------------|--|
| $B \rightarrow B_1 \text{ or } B_2$ | $B.addr := NewTemp()$ $Gen(B.addr := B_1.addr \text{ or } B_2.addr)$ |
| $B \rightarrow B_1 \text{ and } B_2$ | $B.addr := NewTemp()$ $Gen(B.addr := B_1.addr \text{ and } B_2.addr)$ |
| $B \rightarrow \text{not } B_1$ | $B.addr := NewTemp()$ $Gen(B.addr := \text{'not' } B_1.addr)$ |
| $B \rightarrow E_1 \text{ rel } E_2$ | $B.addr := NewTemp()$ $Gen(\text{'if' } E_1.addr \text{ rel.op } E_2.addr \text{ goto next+3})$ $Gen(B.addr := \text{'false'})$ $Gen(\text{'goto' next+2})$ $Gen(B.addr := \text{'true'})$ |
| $B \rightarrow (B_1)$ | $B.addr := B_1.addr$ |
| $B \rightarrow \text{true}$ | $B.addr := NewTemp()$ $Gen(B.addr := \text{'true'})$ |
| $B \rightarrow \text{false}$ | $B.addr := NewTemp()$ $Gen(B.addr := \text{'false'})$ |

В синтезируемом атрибуте *addr* устанавливается логическое значение (**true** или **false**) соответствующего выражения. Переменная *next* содержит индекс очередной трехадресной команды в формируемом потоке. Процедура *Gen* после построения каждой команды инкрементирует значение *next*.

Для выражения $b < c$ **and not** ($d > e$ **or** $f < g$) СЮО сгенерирует следующий код (условно начиная с позиции 50):

```

50: if b < c goto 53
51: t1:=false
52: goto 54
53: t1:=true
54: if d > e goto 57
55: t2:=false
56: goto 58
57: t2:=true
58: if f < g goto 61
59: t3:=false
60: goto 62
61: t3:=true
62: t4:=t2 or t3
63: t5:=not t4
64: t6:=t1 and t5.

```

| Продукция | Семантические правила |
|--------------------------------------|--|
| $B \rightarrow B_1 \text{ or } B_2$ | $B.addr := NewTemp()$ $Gen(B.addr := ' B_1.addr \text{ or } B_2.addr)$ |
| $B \rightarrow B_1 \text{ and } B_2$ | $B.addr := NewTemp()$ $Gen(B.addr := ' B_1.addr \text{ and } B_2.addr)$ |
| $B \rightarrow \text{not } B_1$ | $B.addr := NewTemp()$ $Gen(B.addr := ' \text{not } B_1.addr)$ |
| $B \rightarrow E_1 \text{ rel } E_2$ | $B.addr := NewTemp()$ $Gen(' \text{if } E_1.addr \text{ rel.op } E_2.addr \text{ goto } next+3)$ $Gen(B.addr := ' \text{'false'})$ $Gen(' \text{goto } next+2)$ $Gen(B.addr := ' \text{'true'})$ |
| $B \rightarrow (B_1)$ | $B.addr := B_1.addr$ |
| $B \rightarrow \text{true}$ | $B.addr := NewTemp()$ $Gen(B.addr := ' \text{'true'})$ |
| $B \rightarrow \text{false}$ | $B.addr := NewTemp()$ $Gen(B.addr := ' \text{'false'})$ |

Другой, более распространенный подход к вычислению логических выражений основан на использовании команд условных и безусловных переходов, которые в зависимости от логического условия передают управление в ту или иную позицию кода (само значение не вычисляется, в коде отсутствуют логические операции). Такой подход позволяет достаточно легко реализовать сокращенное вычисление выражений.

СУО для трансляции логических выражений с помощью команд условного и безусловного переходов показано в табл. 12.

Таблица 12 соответственно. Оператор присваивания S имеет наследуемый атрибут $S.next$, значением которого является метка, указывающая на команду, непосредственно следующую за кодом S (подробнее об этом атрибуте в подразд. 6.4).

Первая продукция специально добавлена в СУО для иллюстрации использования логического выражения в правой части оператора присваивания. Значение выражения определяется позицией в последовательности команд. Метки $B.true$ и $B.false$ означают позиции перехода для выражения в случае его истинности или ложности соответственно. Тогда логическое выражение истинно, если управление достигает команды с меткой $B.true$, и ложно, если достигает команды с меткой $B.false$.

Рассмотрим продукцию $B \rightarrow B_1 \text{ or } B_2$. Если B_1 истинно, то и все выражение B истинно, поэтому $B_1.true := B.true$. Если B_1 ложно, то следует перейти к вычислению B_2 , поэтому $B_1.false$ должна быть меткой первой команды кода для B_2 . Метки $B_2.true$ и $B_2.false$ совпадают с соответствующими метками для B .

В продукции $B \rightarrow B_1 \text{ and } B_2$ аналогичные правила. Отличие в том, что, если B_1 истинно, реализуется переход к вычислению B_2 , в противном случае B ложно.

В продукции $B \rightarrow \text{not } B_1$ не формируется никакой новый код, а просто реализуется перенаправление управления с истины на ложь и наоборот.

Семантические правила для остальных продукций очевидны и не требуют дополнительных пояснений.

СУО для трансляции логических выражений

| Продукция | Семантические правила |
|--------------------------------------|--|
| $S \rightarrow \text{id} := B$ | $B.true := NewLabel()$ $B.false := NewLabel()$ $S.code := B.code \parallel Label(B.true) \parallel Gen(\text{id.pnt} := 'true')$ $\parallel Gen('goto' S.next)$ $\parallel Label(B.false) \parallel Gen(\text{id.pnt} := 'false')$ |
| $B \rightarrow B_1 \text{ or } B_2$ | $B_1.true := B.true$ $B_1.false := NewLabel()$ $B_2.true := B.true$ $B_2.false := B.false$ $B.code := B_1.code \parallel Label(B_1.false) \parallel B_2.code$ |
| $B \rightarrow B_1 \text{ and } B_2$ | $B_1.true := NewLabel()$ $B_1.false := B.false$ $B_2.true := B.true$ $B_2.false := B.false$ $B.code := B_1.code \parallel Label(B_1.true) \parallel B_2.code$ |
| $B \rightarrow \text{not } B_1$ | $B_1.true := B.false$ $B_1.false := B.true$ $B.code := B_1.code$ |
| $B \rightarrow E_1 \text{ rel } E_2$ | $B.code := E_1.code \parallel E_2.code$ $\parallel Gen('if' E_1.addr \text{ rel } op E_2.addr 'goto' B.true)$ $\parallel Gen('goto' B.false)$ |
| $B \rightarrow (B_1)$ | $B_1.true := B.true$ $B_1.false := B.false$ $B.code := B_1.code$ |
| $B \rightarrow \text{true}$ | $B.code := Gen('goto' B.true)$ |
| $B \rightarrow \text{false}$ | $B.code := Gen('goto' B.false)$ |

В синтезируемом атрибуте *code* формируется трехадресный код для соответствующего нетерминала. Функция *NewLabel()* создает новую метку, а функция *Label()* назначает метку очередной трехадресной команде. Логическое выражение B имеет наследуемые атрибуты $B.true$ и $B.false$. Значениями этих атрибутов являются метки, которым передается управление в случае истинности или ложности выражения B

Для присваивания $a := b < c \text{ and not } (d > e \text{ or } f < g)$ данное СУО сгенерирует следующий код (*Snext* – метка команды, непосредственно следующей за данным оператором присваивания):

```
    if b < c goto L3
    goto L2
L3: if d > e goto L2
    goto L4
L4: if f < g goto L2
    goto L1
L1: a:=true
    goto Snext
L2: a:=false.
```

Инкрементная трансляция логического выражения

| Продукция | Семантические правила |
|---|--|
| 1) $S \rightarrow \text{id} := B$ | $B.true := \text{NewLabel}(); B.false := \text{NewLabel}()$ $\text{Gen}(\text{Label}(B.true): \text{id.pnt} := 'true');$ $\text{Gen}('goto' S.next)$ $\text{Gen}(\text{Label}(B.false): \text{id.pnt} := 'false')$ |
| 2) $B \rightarrow B_1 \text{ or } B_2$ | $B_1.true := B.true; B_1.false := \text{NewLabel}()$ $B_2.true := B.true; B_2.false := B.false$ $\text{Label}(B_1.false)$ |
| 3) $B \rightarrow B_1 \text{ and } B_2$ | $B_1.true := \text{NewLabel}(); B_1.false := B.false$ $B_2.true := B.true; B_2.false := B.false$ $\text{Label}(B_1.true)$ |
| 4) $B \rightarrow \text{not } B_1$ | $B_1.true := B.false; B_1.false := B.true$ |
| 5) $B \rightarrow E_1 \text{ rel } E_2$ | $\text{Gen}('if' E_1.addr \text{ rel.op } E_2.addr 'goto' B.true)$ $\text{Gen}('goto' B.false)$ |
| 6) $B \rightarrow (B_1)$ | $B_1.true := B.true; B_1.false := B.false$ |
| 7) $B \rightarrow \text{true}$ | $\text{Gen}('goto' B.true)$ |
| 8) $B \rightarrow \text{false}$ | $\text{Gen}('goto' B.false)$ |

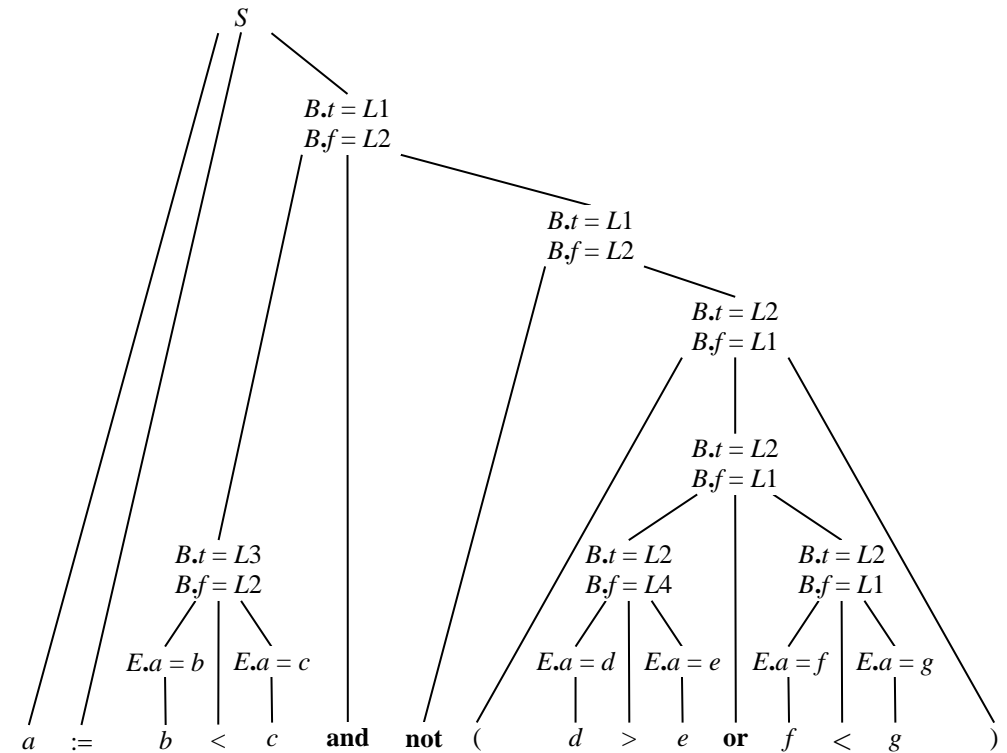
$a := b < c \text{ and not } (d > e \text{ or } f < g)$

На рис. вместо **id.pnt** указан сам идентификатор, атрибуты *true* и *false* обозначены как *t* и *f* соответственно.

```

if b < c goto L3
goto L2
L3: if d > e goto L2
    goto L4
L4: if f < g goto L2
    goto L1
L1: a:=true
    goto Snext
L2: a:=false.
Snext:

```



Рассмотрим подробнее процесс аннотирования дерева.

| Продукция | Семантические правила |
|---|---|
| 1) $S \rightarrow \text{id} := B$ | $B.true := NewLabel(); B.false := NewLabel();$ $Gen(Label(B.true); \text{id.pnt} := 'true');$ $Gen('goto' S.next)$ $Gen(Label(B.false); \text{id.pnt} := 'false');$ |
| 3) $B \rightarrow B_1 \text{ and } B_2$ | $B_1.true := NewLabel(); B_1.false := B.false$ $B_2.true := B.true; B_2.false := B.false$ $Label(B_1.true)$ |
| 5) $B \rightarrow E_1 \text{ rel } E_2$ | $Gen('if' E_1.addr \text{ rel.op } E_2.addr 'goto' B.true)$ $Gen('goto' B.false)$ |

В соответствии с продукцией 1 создаются новые метки и присваиваются наследуемым атрибутам B : $B.true = L1$ и $B.false = L2$.

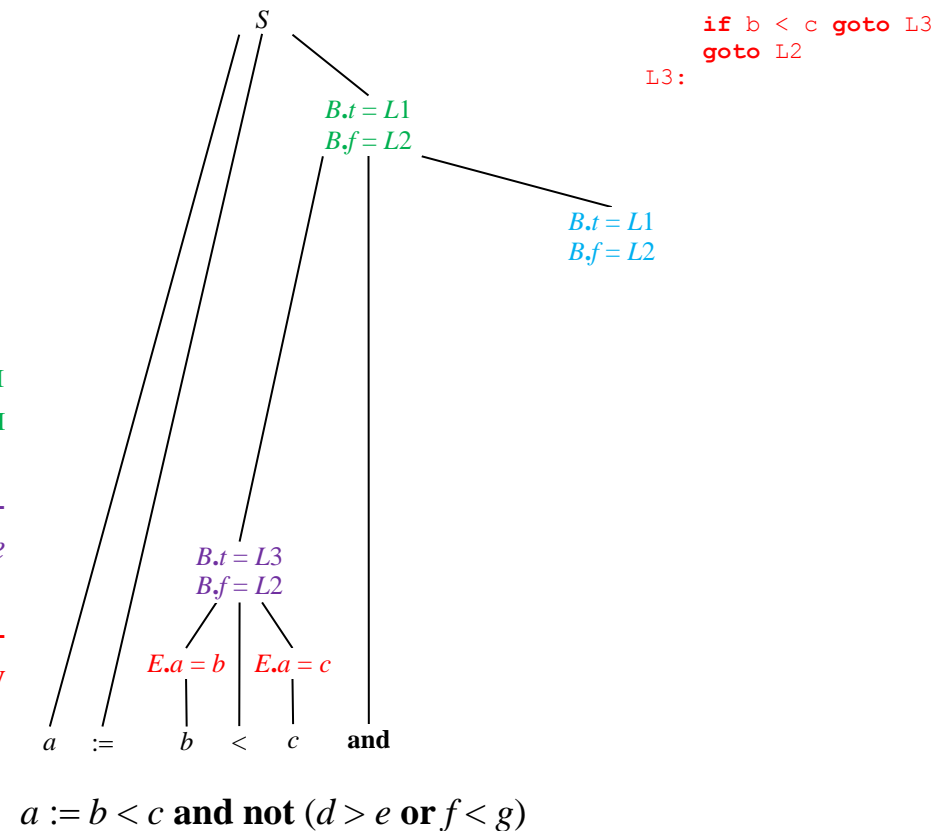
Затем в соответствии с продукцией 3 для наследуемого атрибута $B_1.true$ создается новая метка $L3$, а атрибуту $B_1.false$ присваивается значение $B.false = L2$.

В соответствии с продукцией 5 формируются две команды, а функция $Label(B_1.true)$ из продукции 3 назначает метку $L3$ очередной трехадресной команде:

```
if b < c goto L3
goto L2
```

$L3:$

В соответствии с продукцией 3 наследуемому атрибуту $B_2.true$ присваивается значение $B.true = L1$, а атрибуту $B_2.false$ присваивается значение $B.false = L2$.



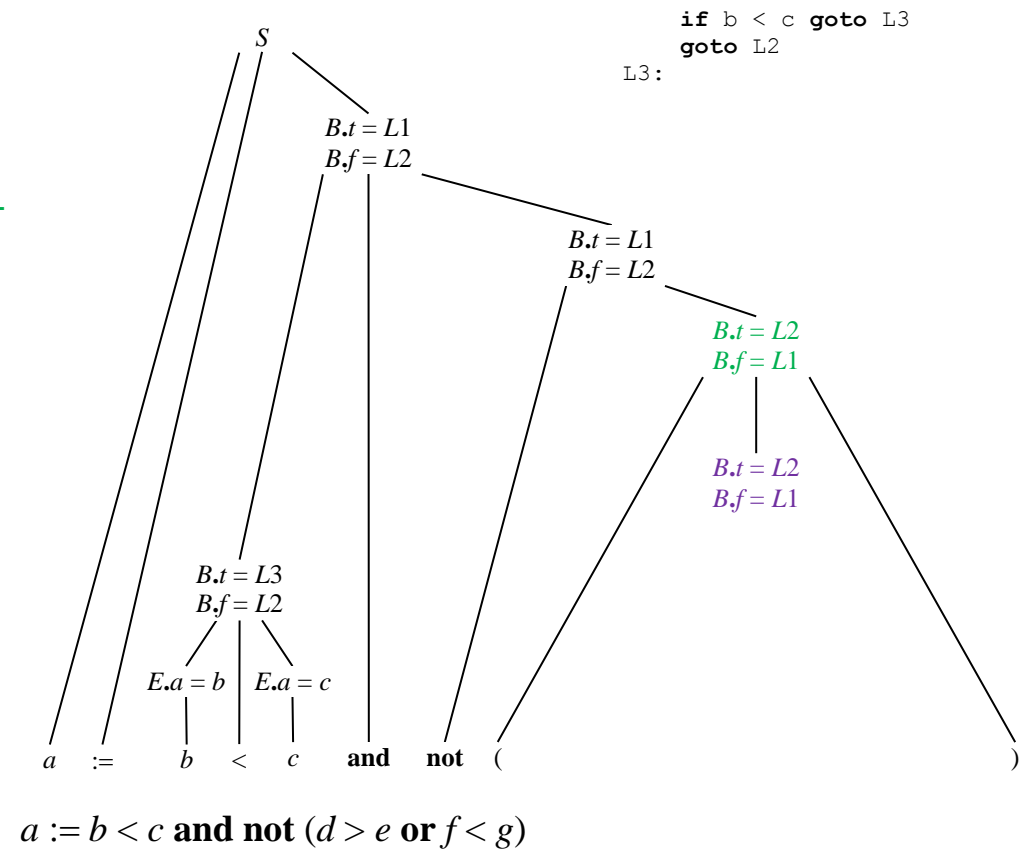
| Продукция | Семантические правила |
|------------------------------------|--|
| 4) $B \rightarrow \text{not } B_1$ | $B_1.true := B.false; B_1.false := B.true$ |
| 6) $B \rightarrow (B_1)$ | $B_1.true := B.true; B_1.false := B.false$ |

В соответствии с продукцией 4 реализуется перенаправление управления с истины на ложь и наоборот:

$B_1.true = B.false = L2$ и $B_1.false = B.true = L1$.

В соответствии с продукцией 6

$B_1.true = B.true = L2, B_1.false = B.false = L1$.




```

    if b < c goto L3
    goto L2
L3: if d > e goto L2
    goto L4
L4: if f < g goto L2
    goto L1
L1: a:=true
    goto Snext
L2: a:=false

```

```

    ifFalse b < c goto L2
    if d > e goto L2
    if f < g goto L2
L1: a:=true
    goto Snext
L2: a:=false.

```

Такое удаление избыточных команд перехода можно выполнить в процессе оптимизации кода.

Обычно генерируемый код не оптимален. Например, можно без всяких последствий удалить четвертую и шестую команды (безусловные переходы **goto L4** и **goto L1**), поскольку они реализуют переход на непосредственно следующие за ними команды. Можно убрать также вторую команду (**goto L2**), если в первой команде заменить **if** на **ifFalse**, т.е. поменяв условие на обратное. Тогда улучшенный трехадресный код будет иметь следующий вид:

Возможен и другой подход, связанный с соответствующим изменением семантических правил так, чтобы избыточные переходы не включались в генерируемый код. Один из таких подходов основан на использовании специальной метки, означающей отсутствие перехода. Рассмотрим детали этого метода [1].

Обозначим через *fall* специальную метку, означающую отсутствие перехода. Если логическое выражение *B* используется в управляющих операторах, то правила для соответствующих этим операторам продукций (например, СУО из подразд. 6.4) можно модифицировать так, чтобы установить *B.true* равным *fall*. Это позволит управлению проходить сквозь код *B*.

Рассмотрим правила для продукции $B \rightarrow E_1 \text{ rel } E_2$ (табл. 12). Для их модификации необходимо рассмотреть следующие ситуации. Если *B.true* и *B.false* являются явными метками (ни одна из них не равна *fall*), то должны генерироваться две команды переходов, как это было ранее. Если *B.true* представляет собой явную метку, а *B.false* – метку *fall*, то должна генерироваться одна команда **if**. Если же *B.true* = *fall*, а *B.false* ≠ *fall*, то должна генерироваться команда **ifFalse**. В случае, когда и *B.true*, и *B.false* равны *fall*, никакие переходы не генерируются. Тогда новые семантические правила для продукции будут следующими:

$test := E_1.addr \text{ rel.op } E_2.addr$
 $str := \text{if } B.true \neq fall \text{ and } B.false \neq fall \text{ then}$
 $Gen('if \text{ test } goto' B.true) \parallel Gen('goto' B.false)$
 else if $B.true \neq fall$ **then** $Gen('if \text{ test } goto' B.true)$
 else if $B.false \neq fall$ **then** $Gen('ifFalse' test 'goto' B.false)$
 else ' '
 $B.code := E_1.code \parallel E_2.code \parallel str.$

Новые правила для продукции $B \rightarrow B_1 \text{ or } B_2$:

if $B.true \neq fall$ **then** $B_1.true := B.true$ **else** $B_1.true :=NewLabel()$
 $B_1.false := fall$
 $B_2.true := B.true$
 $B_2.false := B.false$
if $B.true \neq fall$ **then** $B.code := B_1.code \parallel B_2.code$
else $B.code := B_1.code \parallel B_2.code \parallel Label(B_1.true).$

Следует заметить, что смысл метки *fall* для B отличается от смысла для B_1 . Пусть $B.true = fall$. Это значит, что управление проходит через B , если значение B истинно. Хотя значение B истинно, если это же значение принимает B_1 , метка $B_1.true$ должна обеспечивать переход управления через код B_2 к команде, непосредственно следующей за кодом B . Если же вычисленное значение B_1 ложно, то истинность значения B определяется значением B_2 . Поэтому приведенные правила гарантируют, что метка $B_1.false$ обеспечивает прохождение управления от B_1 к коду B_2 .

Аналогичным образом можно модифицировать семантические правила для других продукций СУО в табл. 12.

Некоторые рекомендации по разработке грамматик арифметических и логических выражений:

Рассмотренное выше СУО не подходит ни для восходящей, ни для нисходящей синтаксически управляемой трансляции. Цель – показать принципы разработки семантических правил для трансляции логических выражений. Поэтому при практической реализации рассмотренные семантические правила следует приспособить к соответствующим продукциям грамматики, определяющей в общем случае как логические, так и арифметические выражения, изменив надлежащим образом обозначения символов грамматики (нетерминалов и терминалов) в семантических правилах.

Ниже приведено соответствие продукций простейшей грамматики выражений и рассмотренной выше грамматики логического выражения.

Соответствие продукций грамматики выражения и логического выражения из СУО

| Грамматика выражения | Грамматика логического выражения из СУО |
|---|---|
| $S \rightarrow \mathbf{id} := Expr$ | $S \rightarrow \mathbf{id} := B$ |
| $Expr \rightarrow SmplExpr \mathbf{rel} SmplExpr$ | $B \rightarrow E_1 \mathbf{rel} E_2$ |
| $Expr \rightarrow SmplExpr$ | |
| $SmplExpr \rightarrow Term$ | |
| $SmplExpr \rightarrow + Term$ | |
| $SmplExpr \rightarrow SmplExpr + Term$ | |
| $SmplExpr \rightarrow SmplExpr \mathbf{or} Term$ | $B \rightarrow B_1 \mathbf{or} B_2$ |
| $Term \rightarrow Factor$ | |
| $Term \rightarrow Term * Factor$ | |
| $Term \rightarrow Term \mathbf{and} Factor$ | $B \rightarrow B_1 \mathbf{and} B_2$ |
| $Factor \rightarrow (Expr)$ | $B \rightarrow (B_1)$ |
| $Factor \rightarrow \mathbf{not} Factor$ | $B \rightarrow \mathbf{not} B_1$ |
| $Factor \rightarrow \mathbf{id}$ | |
| $Factor \rightarrow \mathbf{num}$ | |
| $Factor \rightarrow \mathbf{true}$ | $B \rightarrow \mathbf{true}$ |
| $Factor \rightarrow \mathbf{false}$ | $B \rightarrow \mathbf{false}$ |