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* = **false**, то без вычисления *b* можно сказать, что все выражение будет иметь значение **false**. Аналогично для *a* **or** *b*, если *a* = **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 нетеминалы *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
$B \to B_1$ **or** B_2	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ B_1.*addr* ′or′ B_2.*addr*)
$B \to B_1$ **and** B_2	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ B_1.*addr* ′and′ B_2.*addr*)
$B \to$ **not** B_1	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ ′not′ B_1.*addr*)
$B \to E_1$ **rel** E_2	*B.addr* := *NewTemp* () *Gen* (′if′ E_1.*addr* **rel**.*op* E_2.*addr* ′goto′ *next*+3) *Gen* (*B.addr* ′:=′ ′false′) *Gen* (′goto′ *next*+2) *Gen* (*B.addr* ′:=′ ′true′)
$B \to (B_1)$	*B.addr* := B_1.*addr*
$B \to$ **true**	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ ′true′)
$B \to$ **false**	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ ′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** $f < g)$ СУО 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
$B \to B_1$ **or** B_2	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ B_1.*addr* ′or′ B_2.*addr*)
$B \to B_1$ **and** B_2	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ B_1.*addr* ′and′ B_2.*addr*)
$B \to$ **not** B_1	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ ′not′ B_1.*addr*)
$B \to E_1$ **rel** E_2	*B.addr* := *NewTemp* () *Gen* (′if′ E_1.*addr* **rel**.*op* E_2.*addr* ′goto′ *next*+3) *Gen* (*B.addr* ′:=′ ′false′) *Gen* (′goto′ *next*+2) *Gen* (*B.addr* ′:=′ ′true′)
$B \to (B_1)$	*B.addr* := B_1.*addr*
$B \to$ **true**	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ ′true′)
$B \to$ **false**	*B.addr* := *NewTemp* () *Gen* (*B.addr* ′:=′ ′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

СУО для трансляции логических выражений

Продукция	Семантические правила
$S \to$ **id** $:= B$	*B.true* := *NewLabel* () *B.false* := *NewLabel* () *S.code* := *B.code* || *Label* (*B.true*) || *Gen* (**id**.*ns* ′:=′ ′true′) || *Gen* (′goto′ *S.next*) || *Label* (*B.false*) || *Gen* (**id**.*ns* ′:=′ ′false′)
$B \to B_1$ **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* || *Label* (B_1.*false*) || B_2.*code*
$B \to B_1$ **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* || *Label* (B_1.*true*) || B_2.*code*
$B \to$ **not** B_1	B_1.*true* := *B.false* B_1.*false* := *B.true* *B.code* := B_1.*code*
$B \to E_1$ **rel** E_2	*B.code* := E_1.*code* || E_2.*code* || *Gen* (′if′ E_1.*addr* **rel**.*op* E_2.*addr* ′goto′ *B.true*) || *Gen* (′goto′ *B.false*)
$B \to (B_1)$	B_1.*true* := *B.true* B_1.*false* := *B.false* *B.code* := B_1.*code*
$B \to$ **true**	*B.code* := *Gen* (′goto′ *B.true*)
$B \to$ **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* 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 $B \to B_1$ **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 $B \to B_1$ **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 $B \to$ **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.

Для присваивания a := *b* < *c* **and not** (*d* > *e* **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.
```

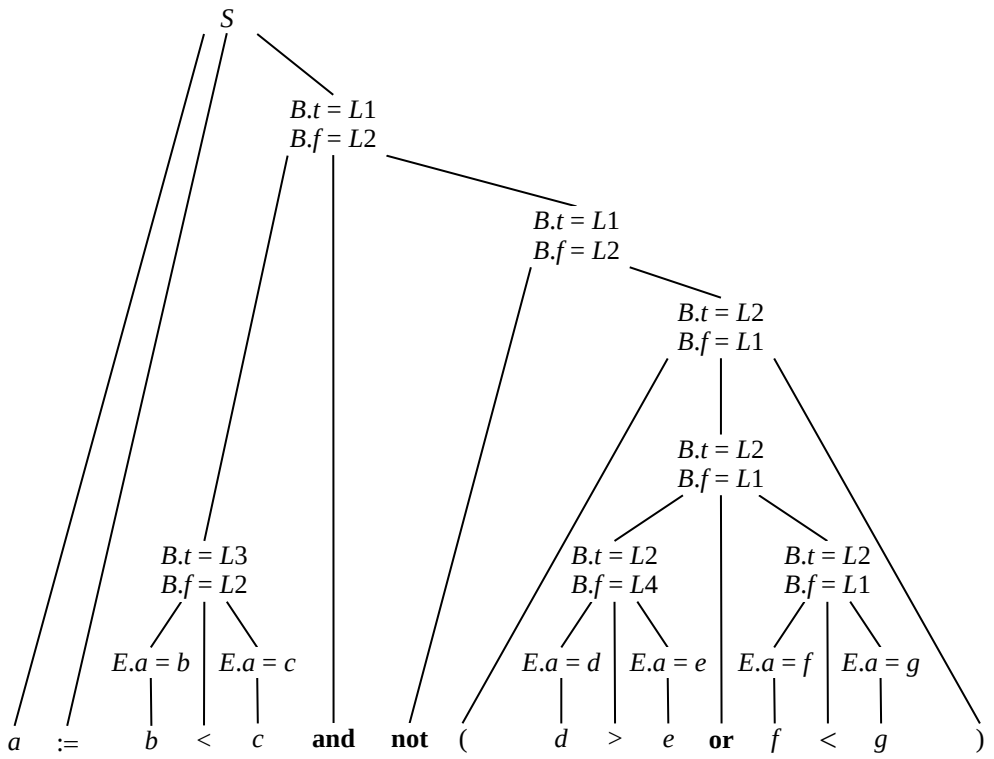
Инкрементная трансляция логического выражения

Продукция	Семантические правила
$S \to$ **id** := B	*B.true* := *NewLabel* (); *B.false* := *NewLabel* () *Gen* (*Label* (*B.true*): **id**.*ns* ':=' 'true'); *Gen* ('goto' *S.next*) *Gen* (*Label* (*B.false*): **id**.*ns* ':=' 'false')
$B \to B_1$ **or** B_2	B_1.*true* := *B.true*; B_1.*false* := *NewLabel* () B_2.*true* := *B.true*; B_2.*false* := *B.false* *Label* (B_1.*false*)
$B \to B_1$ **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*)
$B \to$ **not** B_1	B_1.*true* := *B.false*; B_1.*false* := *B.true*
$B \to E_1$ **rel** E_2	*Gen* ('if' E_1.*addr* **rel**.*op* E_2.*addr* 'goto' *B.true*) *Gen* ('goto' *B.false*)
$B \to (B_1)$	B_1.*true* := *B.true*; B_1.*false* := *B.false*
$B \to$ **true**	*Gen* ('goto' *B.true*)
$B \to$ **false**	*Gen* ('goto' *B.false*)

$a := b < c$ **and not** $(d > e$ **or** $f < g)$

На рис. вместо **id**.*ns* 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.
```



$S \to$ **id :=** B
$B \to B$ **or** T
$B \to T$
$B \to$ **not** T
$T \to T$ **and** F
$T \to F$
$F \to$ **(** B **)**
$F \to$ **id rel id**
$F \to$ **id**
$F \to$ **true**
$F \to$ **false**

Обычно генерируемый код не оптимален. Например, можно без всяких последствий удалить четвертую и шестую команды (безусловные переходы **goto** *L*4 и **goto** *L*1), поскольку они реализуют переход на непосредственно следующие за ними команды. Можно убрать также вторую команду (**goto** *L*2), если в первой команде заменить **if** на **ifFalse**, т.е. поменяв условие на обратное. Тогда улучшенный трехадресный код будет иметь следующий вид:

```
      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.

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 $B \rightarrow E_1$ **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* **rel**.*op* E_2.*addr*
str := **if** *B.true* ≠ *fall* **and** *B.false* ≠ *fall* **then**
Gen (′if′ *test* ′goto′ *B.true*) || *Gen* (′goto′ *B.false*)
else if *B.true* ≠ *fall* **then** *Gen* (′if′ *test* ′goto′ *B.true*)
else if *B.false* ≠ *fall* **then** *Gen* (′ifFalse′ *test* ′goto′ *B.false*)
else ′ ′
B.code := E_1.*code* || E_2.*code* || *str*.

Новые правила для продукции $B \rightarrow B_1$ **or** B_2:

if *B.true* ≠ *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* ≠ *fall* **then** *B.code* := B_1.*code* || B_2.*code*
else *B.code* := B_1.*code* || B_2.*code* || *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.