Тема 6. Генерация промежуточного кода

6.7. Метод обратных поправок для оператора **for**

Один из возможных синтаксисов оператора цикла с параметром (счетчиком):

for id := *Ebeg* **to** *Eend* **by** *Estep* **do**
    S_1;
    S_2;
    …
    S_n
end

Переменная **id** является параметром (счетчиком) цикла, арифметические выражения *Ebeg*, *Eend* и *Estep* определяют соответственно начальное, конечное значения и шаг изменения параметра цикла **id**. Для простоты будем считать, что параметр **id** и выражения *Ebeg*, *Eend* и *Estep* должны быть целого типа.

Рассмотрим трансляцию.

Формирование трехадресного кода можно реализовать по аналогии с оператором цикла с предусловием **while**. Создается достаточно простой промежуточный код (для удобства указаны обозначения выражений и **id**, очевидно, что в процессе выполнения программы рассматриваются их значения):

```
        id := Ebeg
beg:    if id > Eend goto next
        Код для S1
        Код для S2
        …
        Код для Sn
        id := id + Estep
        goto beg
next:
```

for id := *Ebeg* **to** *Eend* **by** *Estep* **do**
    S_1;
    S_2;
    …
    S_n
end

Таким образом, перед телом цикла следует сгенерировать команды установки начального значения параметра цикла и проверки условия выхода из цикла

id := *Ebeg*
beg: **if id** > *Eend* **goto** *next*

В конец тела цикла надо добавить команды изменения значения параметра и безусловного перехода к команде проверки условия выхода из цикла (метка *beg*).

id := **id** + *Estep*
goto *beg*

Для построения СУО рассмотрим следующую грамматику оператора **for**:

OpFor → **for id ass** *Expr* **to** *Expr* **by** *Expr* **do** *LstStmt* **end**

или в сокращенном виде:

S → **for id ass** *E* **to** *E* **by** *E* **do** *L* **end**

Тогда СУО для трансляции можно представить следующим образом:

СУО для трансляции оператора цикла **for** методом обратных поправок

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*
$S \to$ **for id ass** E_1 **to** E_2 **by** E_3 **do** P L **end**	*Gen*(**id.***pnt* ′:=′ **id.***pnt* ′+′ *E*$_3$**.***addr*) *Gen*(′**goto**′ *P***.***instr*) *BackPatch*(*P***.***nextlist*, *nextinstr*) *S***.***nextlist* := **null**
$P \to \varepsilon$	*Gen* (**id.***pnt* ′:=′ *E*$_1$**.***addr*) *P***.***nextlist* := *MakeList* (*nextinstr*) *P***.***instr* := *nextinstr* *Gen* (′**if**′ **id.***pnt* ′>′ *E*$_2$**.***addr* ′**goto ?**′)

Необходимость маркера *P* объясняется тем, что до формирования трехадресных команд тела цикла *L* перед его первой командой необходимо вставить команду установки начального значения параметра **id** и команду условного перехода (обозначим ее через *beg*). Команда *beg* реализует (при выполнении условия выхода из цикла) передачу управления к первой команде, непосредственно следующей за *S* (ее адрес *next* еще неизвестен). Поэтому создается список ***P.****nextlist* с адресом команды *beg* для последующего выполнения обратной поправки, а в *P.instr* запоминается ее адрес для передачи управления на эту команду в конце тела цикла.

После генерации команд из *L* добавляются команда приращения параметра цикла и команда безусловного перехода на команду *beg* (ее адрес содержится в ***P.****instr*). Поскольку на этот момент сформирована последняя команда цикла, становится известен адрес следующей за *S* команды, т. е. *next* = *nextinstr*. Поэтому выполняется обратная поправка для перехода из списка ***P.****nextlist* (а это не что иное, как команда *beg*) установкой целевой метки *nextinstr*.

Семантические правила настолько очевидны, что не вижу смысла рассматривать пример аннотированного дерева разбора для этого оператора. Попробуйте это сделать самостоятельно.

Очень важное замечание!

Рассмотренное выше СУО корректно для положительного шага приращения параметра цикла (предполагается, что значение выражения E_3 больше нуля). Поэтому при практической реализации вместо одной команды

beg: **if id** > *Eend* **goto** *next*

следует формировать последовательность команд вида

beg: **if** *Estep* < 0 **goto** L_1 //отрицательны шаг
if *Estep* > 0 **goto** L_2 //положительный шаг
if *Estep* = 0 **exception** *runtime_error* //ошибка времени выполнения

L_1: **if id** < *Eend* **goto** *next*
goto L_3

L_2: **if id** > *Eend* **goto** *next*

L_3: Код для S_1

Очевидно, что необходимо добавить семантические правила для объединения в один список с помощью функции *Merge* адресов команд с метками L_1 и L_2 для последующего выполнения обратной поправки. Для остальных команд перехода соответствующие значения целевых меток L_1, L_2 и L_3 (адресов передачи управления) легко можно вычислить исходя из последовательности генерируемых команд.

Рассмотренные нюансы, в общем-то, не оказывают существенного влияния на общие принципы разработки СУО, более того, в рамках РГР мы не рассматриваем формирование трехадресных команд для обнаружения ошибок времени выполнения (*runtime_error*). Поэтому в РГР для оператора **for** можно использовать более простой синтаксис, жестко зафиксировав направление (возрастание или убывание) и/или величину шага изменения значения параметра, как например, в языке Паскаль, с помощью ключевых слов **to** (шаг равен +1) и **downto** (шаг равен –1).