Тема 6. Генерация промежуточного кода

6.5. Метод обратных поправок

При инкрементной трансляции (когда формируется единый поток генерируемых трехадресных команд в некотором глобальном массиве или файле) логических выражений и управляющих операторов возникает проблема, которая заключается в том, что к моменту создания команды перехода позиция команды, к которой должно перейти управление, еще неизвестна. Например, в продукции $S \rightarrow$ **if** B **then** S_1 при трансляции B для случая его ложного значения формируется переход к первой команде кода для оператора, следующего за S. Однако к этому моменту, возможно, еще не завершена трансляция всего выражения B и еще не выполнена трансляция S_1. Поэтому целевая метка для перехода еще не известна.

Простейшим способом решения проблемы является использование двух проходов. Сначала выполняется трансляция с использованием символьных меток (см. СУО в табл. 12 и 13), а затем производится замена этих меток целевыми метками (индексами глобального массива трехадресного кода).

Другой подход, ориентированный на однопроходную трансляцию, формирует команду перехода с временно неопределенными переходами, которые доопределяются целевыми метками в момент, когда становятся известными позиции команд, к которым передается управление. Такое последовательное заполнение команд метками называется *методом обратных поправок* [1; 2].

В этом методе после генерации команды с временно неопределенным переходом эта команда (а точнее, ее индекс в массиве команд) помещается в специальный список команд, метки которых будут указаны после того, как они будут определены. Все переходы группируются в списки так, что команды из одного списка будут иметь одну и ту же целевую метку.

Пусть для определенности трехадресный код генерируется в виде массива, тогда метки представляют собой индексы этого массива.

Для работы со списками переходов используются следующие функции и процедура:

функция *MakeList*(*i*) создает новый одноэлементный список, состоящий только из *i* (индекс в массиве команд), возвращает указатель на созданный список;

функция *Merge*(p_1, p_2) объединяет списки, на которые указывают p_1 и p_2, возвращает указатель на объединенный список;

процедура *BackPatch*(*p, i*) устанавливает *i* в качестве целевой метки в каждую команду из списка, на который указывает *p*, после этого время жизни списка завершается.

Вместо наследуемых атрибутов *B.true* и *B.false*, как это было в СУО в табл. 12 и 13, используются синтезируемые атрибуты *B.truelist* и *B.falselist*. Эти атрибуты представляют собой указатели на списки команд перехода, которые должны получить метку команды, которой передается управление при истинности или ложности выражения *B* соответственно. Аналогично вместо наследуемого атрибута *S.next* в СУО в табл. 13 используется синтезируемый атрибут *S.nextlist*, представляющий собой указатель на список команд переходов к команде, идущей непосредственно за кодом *S*.

СУО для инкрементной трансляции логических выражений методом обратных поправок показано в табл. 14.

Таблица 14

СУО для трансляции логических выражений методом обратных поправок

Продукция	Семантические правила
1) $S \to$ **id** $:= B$	*BackPatch* (*B.truelist*, *nextinstr*) *BackPatch* (*B.falselist*, *nextinstr* + 2) *Gen* (**id**.*ns* ′:=′ ′true′) *Gen* (′goto′ *nextinstr* + 2) *Gen* (**id**.*ns* ′:=′ ′false′) *S.nextlist* := **null**
2) $B \to B_1$ **or** $M\ B_2$	*BackPatch* (B_1.*falselist*, *M.instr*) *B.truelist* := *Merge* (B_1.*truelist*, B_2.*truelist*) *B.falselist* := B_2.*falselist*
3) $B \to B_1$ **and** $M\ B_2$	*BackPatch* (B_1.*truelist*, *M.instr*) *B.truelist* := B_2.*truelist* *B.falselist* := *Merge* (B_1.*falselist*, B_2.*falselist*)
4) $B \to$ **not** B_1	*B.truelist* := B_1.*falselist* *B.falselist* := B_1.*truelist*
5) $B \to E_1$ **rel** E_2	*B.truelist* := *MakeList* (*nextinstr*) *B.falselist* := *MakeList* (*nextinstr* + 1) *Gen* (′if′ E_1.*addr* **rel**.*op* E_2.*addr* ′goto ?′) *Gen* (′goto ?′)
6) $B \to (B_1)$	*B.truelist* := B_1.*truelist* *B.falselist* := B_1.*falselist*
7) $B \to$ **true**	*B.truelist* := *MakeList* (*nextinstr*) *Gen* (′goto ?′)
8) $B \to$ **false**	*B.falselist* := *MakeList* (*nextinstr*) *Gen* (′goto ?′)
9) $M \to \varepsilon$	*M.instr* := *nextinstr*

В продукции для операций **or** и **and** добавлен специальный нетерминал-маркер M, с которым связана продукция $M \to \varepsilon$. Этот маркер фиксирует момент, когда необходимо получить индекс очередной команды непосредственно перед ее генерацией. При преобразовании СУО в СУТ это действие должно выполняться непосредственно перед нетерминалом, для которого будет генерироваться код. В соответствии с правилами из подразд. 3.2 для восходящей трансляции все действия должны быть в конце правой части продукции, что и достигается добавлением нетерминалов-маркеров.

Глобальная переменная *nextinstr* выполняет функции счетчика трехадресных команд и хранит индекс очередной команды. Константа **null** служит для инициализации пустых списков.

Как и в СУО в табл. 12, первая продукция добавлена для иллюстрации использования логического выражения в правой части оператора присваивания. К моменту выполнения связанных с продукцией семантических правил код для B уже сформирован. Логическое выражение истинно, если управление достигает команды из списка *B.truelist*, и ложно, если достигает команды из списка *B.falselist*. Поэтому для команд перехода из этих списков устанавливаются целевые метки генерируемых команд присваивания идентификатору соответствующих значений логического выражения B. Поскольку из оператора присваивания нет никаких переходов, список *S.nextlist* делается пустым.

В продукции $B \to B_1$ **or** $M\ B_2$, если B_1 истинно, то и все выражение B истинно, поэтому список B_1.*truelist* объединяется со списком *B.truelist*. Если B_1 ложно, то следует перейти к вычислению B_2, поэтому целевой меткой переходов из B_1.*falselist* устанавливается метка первой команды кода для B_2. Эта метка получается с помощью синтезируемого атрибута *M.instr* маркера M. Списком *B.falselist* становится список B_2.*falselist*.

В продукции $B \to B_1$ **and** $M\ B_2$ 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 для реализации перенаправления управления меняются местами списки для ложного и истинного значений выражения.

Для простоты в продукции $B \to E_1$ **rel** E_2 генерируются две команды переходов (условный и безусловный) без применения методов сокращения избыточных переходов (см. подразд. 6.3).

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

Преобразование данного СУО в СУТ приведет к тому, что все действия будут расположены в конце правых частей продукций и легко могут быть выполнены при свертке в процессе восходящего разбора.

Рассмотрим присваивание a := *b* < *c* **and not** (*d* > *e* **or** *f* < *g*). Отсчет номеров позиций команд начинается с 50. На аннотированном дереве разбора (рис. 10) атрибуты для удобства обозначены: *truelist* – *t*, *falselist* – *f*, *instr* – *i*, значения атрибутов *truelist* и *falselist* показываются как содержимое списков.

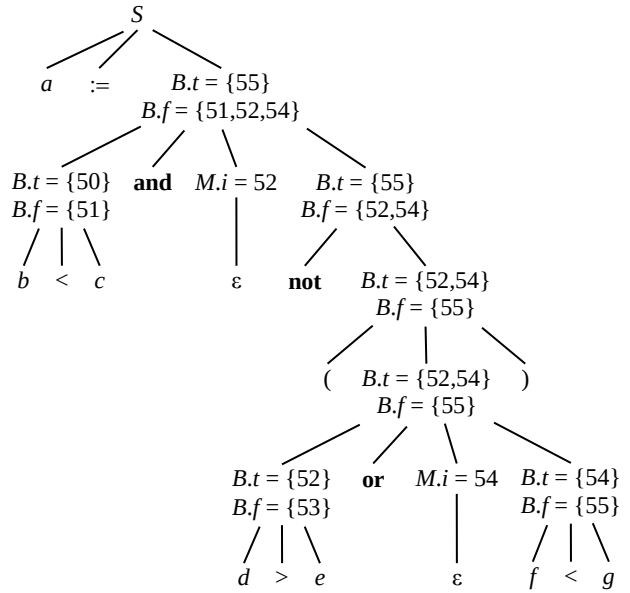

Рис. 10. Аннотированное дерево разбора для a := *b* < *c* **and not** (*d* > *e* **or** *f* < *g*)

В соответствии с продукцией 5 формируются две команды:

```
50: if b < c goto ?
51: goto ?
```

Сгенерирован код для B_1 продукции $B \to B_1$ **and** $M\ B_2$. С помощью маркера M в атрибуте *M.instr* сохраняется текущее значение *nextinstr*, равное 52. Далее в соответствии с продукцией 5 генерируются команды:

```
52: if d > e goto ?
53: goto ?
```

Сгенерирован код для B_1 продукции $B \to B_1$ **or** $M\ B_2$. С помощью маркера M в атрибуте *M.instr* сохраняется текущее значение *nextinstr*, равное 54. Далее в соответствии с продукцией 5 генерируются команды:

```
54: if f < g goto ?
55: goto ?
```

Сгенерирован код для B_2 продукции $B \to B_1$ **or** $M\ B_2$. Поскольку в данный момент B_1.*falselist* = {53} и *M.instr* = 54, выполняется *BackPatch* ({53}, 54), в результате чего команда 53 получит целевую метку 54. Список *B.truelist* = {52, 54} образуется в результате объединения *Merge* (B_1.*truelist*, B_2.*truelist*). Списком *B.falselist* становится список B_2.*falselist* = {55}.

Продукция $B \to (B_1)$ не изменяет атрибуты. В соответствии с продукцией $B \to$ **not** B_1 списки B_1.*truelist* и B_1.*falselist* меняются местами, т.е. *B.truelist* = {55} и *B.falselist* = {52, 54}.

К данному моменту завершено формирование кода для B_2 продукции $B \to B_1$ **and** $M\ B_2$. Атрибуты имеют следующие значения: B_1.*truelist* = {50}, B_1.*falselist* = {51}, B_2.*truelist* = {55}, B_2.*falselist* = {52, 54}, *M.instr* = 52. В результате выполнения *BackPatch* ({50}, 52) команда 50 получит целевую метку 52. Списком *B.truelist* становится список B_2.*truelist* = {55}. Список *B.falselist* = {51, 52, 54} образуется в результате объединения *Merge* (B_1.*falselist*, B_2.*falselist*).

Завершена генерация кода для B продукции **id** := B. На данный момент *nextinstr* = 56. Поскольку *B.truelist* = {55}, в результате выполнения *BackPatch* ({55}, 56) команда 55 получит целевую метку 56. Так как *B.falselist* = {51, 52, 54}, в результате выполнения *BackPatch* ({51, 52, 54}, 58) команды 51, 52 и 54 получат целевую метку 58. Затем формируются команды для установки значений идентификатору **id**. В результате команды примут окончательный вид:

```
50: if b < c goto 52
51: goto 58
52: if d > e goto 58
53: goto 54
54: if f < g goto 58
55: goto 56
56: a:=true
57: goto 59
58: a:=false
59:
```

Метод обратных поправок можно использовать и для однопроходной инкрементной трансляции управляющих операторов. Соответствующее СУО представлено в табл. 15. Как и в СУО в табл. 14, логические выражения *B* имеют два списка переходов *B.truelist* и *B.falselist*, соответствующие истинным и ложным значениям *B*. Нетерминалы *L* (список операторов) и *S* (оператор) имеют атрибут *nextlist*, представляющий собой указатель на список команд переходов к команде, идущей непосредственно за кодом *L* или *S*. В эти списки группируются команды с временно неопределенными переходами.

Таблица 15

СУО для трансляции управляющих операторов 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*
$S \to$ **id** := B	см. табл. 14
$S \to$ **if** B **then** $M\ S_1$	*BackPatch* (*B.truelist*, *M.instr*) *S.nextlist* := *Merge* (*B.falselist*, *S*$_1$.*nextlist*)
$S \to$**if** B **then** $M_1\ S_1\ N$ **else** $M_2\ S_2$	*BackPatch* (*B.truelist*, *M*$_1$.*instr*) *BackPatch* (*B.falselist*, *M*$_2$.*instr*) *tmp* := *Merge* (*S*$_1$.*nextlist*, *N.nextlist*) *S.nextlist* := *Merge* (*tmp*, *S*$_2$.*nextlist*)
$S \to$ **while** $M_1\ B$ **do** $M_2\ S_1$	*BackPatch* (*S*$_1$.*nextlist*, *M*$_1$.*instr*) *BackPatch* (*B.truelist*, *M*$_2$.*instr*) *S.nextlist* := *B.falselist* *Gen* ('goto' *M*$_1$.*instr*)
$M \to \varepsilon$	*M.instr* := *nextinstr*
$N \to \varepsilon$	*N.nextlist* := *MakeList* (*nextinstr*) *Gen* ('goto ?')

В продукции для оператора **if-then**, если выражение *B* истинно, то следует перейти к вычислению S_1, поэтому целевой меткой переходов из *B.truelist* устанавливается метка первой команды S_1. Эта метка получается с помощью синтезируемого атрибута *M.instr* маркера *M*. Если *B* ложно, необходимо обеспечить пропуск оператора S_1 и переход к первой команде, непосредственно следующей за *S* (эта команда является также непосредственно следующей за S_1). Поэтому выполняется объединение списков *B.falselist* и S_1.*nextlist* в список *S.nextlist*.

При трансляции оператора **if-then-else** выполняется обратная поправка для переходов из списка *B.truelist* установкой целевой метки M_1.*instr*, представляющей начало кода S_1 и из списка *B.falselist* установкой целевой метки M_2.*instr*, представляющей начало кода S_2. В конец кода S_1 необходимо добавить безусловный переход для пропуска кода S_2 и передачи управления команде, непосредственно следующей за *S*. Позиция этой команды в данный момент неизвестна. Поэтому с помощью маркера *N* с продукцией $N \to \varepsilon$ формируется команда безусловного перехода и создается одноэлементный список *N.nextlist* с индексом этой команды. Затем списки S_1.*nextlist*, *N.nextlist* и S_2.*nextlist* объединяются в список *S.nextlist* (для объединения используется локальная переменная *tmp*).

В продукции для оператора цикла **while** используются маркеры M_1 для первой команды кода *B* и M_2 для первой команды кода S_1. Необходимость маркера M_1 объясняется тем, что в конец кода S_1 добавляется безусловный переход для передачи управления на первую команду кода *B*, индекс которой должен быть известен. При трансляции выполняется обратная поправка для переходов из списка S_1.*nextlist* установкой целевой метки M_1.*instr*, представляющей начало кода *B* и из списка *B.truelist* установкой целевой метки M_2.*instr*, представляющей начало кода S_1. Если *B* ложно, необходимо обеспечить пропуск оператора S_1 и переход к первой команде, непосредственно следующей за *S*. Поэтому списком *S.nextlist* становится список *B.falselist*. Добавляется команда безусловного перехода для передачи управления первой команде кода *B*, индекс которой сохранен в M_1.*instr*.

Последовательность операторов представляется продукцией $L \to L_1$; $M\ S$. Выполняется обратная поправка для переходов из списка L_1.*nextlist* установкой метки *M.instr*, представляющей начало кода *S*. Списком *L.nextlist* становится список *S.nextlist*.

Для продукции $L \to S$ список *L.nextlist* совпадает *S.nextlist*.

В рассмотренном СУО трехадресная команда генерируется только в продукциях для операторов **if-then-else** (через маркер *N* и продукцию $N \to \varepsilon$) и **while**. Весь остальной код формируется в семантических правилах, связанных с операторами присваивания и выражениями.

В рассмотренных выше СУО, использующих метод обратных поправок, все семантические правила выполняются в конце правых частей продукций. Это означает, что это практически готовые схемы трансляции для восходящего разбора. Достаточно решить вопросы хранения атрибутов и детализировать действия соответствующими операциями со стеком.

В заключение следует отметить, что в лекциях основное внимание было уделено методам построения СУО для решения задач проверки типов и генерации промежуточного кода. Поэтому, чтобы упростить изложение этих методов, не рассматривались структурированные типы данных (массивы, записи и т.п.) и применение конструкторов типа. Варианты СУО и СУТ, обеспечивающих проверку типов, распределение памяти и генерацию промежуточного кода для различных структурированных типов данных, указателей, операторов языков программирования для самостоятельной проработки можно найти в работах [1; 2]. Рекомендации по практическому применению методов для реализации различных фаз компиляции можно посмотреть также в работе [9].