5.5 Übersetzung Boolescher Ausdrücke

Es sollen im folgenden Boolesche Ausdrücke betrachtet werden, die gemäß der folgenden Grammatik festgelegt seien:

$$\texttt{E} \rightarrow \texttt{E or E} \mid \texttt{E and E} \mid \texttt{not E} \mid \texttt{(E)} \mid \texttt{id relOp id} \mid \texttt{true} \mid \texttt{false}$$

Dabei steht `relOp` für einen der üblichen arithmetischen Vergleichsoperatoren und es wird angenommen, dass die Operationen `or` und `and` linksassoziativ sind und dass `not` eine höhere Priorität als `and`, und `and` wiederum eine höhere als `or` hat.

Es gibt nun zwei prinzipielle Möglichkeiten, den Wert eines Booleschen Ausdrucks darzustellen:

1) Durch numerische Kodierung, etwa $1 \mathrel{\hat{=}}$ `true` und $0 \mathrel{\hat{=}}$ `false` oder aber auch jeder Wert ungleich $0 \mathrel{\hat{=}}$ `true` und $0 \mathrel{\hat{=}}$ `false`.

   Die Kodierung sollte so gewählt werden, dass für die Booleschen Operationen möglichst die in der Hardware der Zielmaschine vorhandenen Befehle verwendet werden können.
   Diese Methode ist besonders geeignet, wenn Boolesche Werte etwa für Boolesche Variablen gespeichert werden müssen.

2) Durch die Steuerung des Programmablaufs, d.h. der Wert eines Booleschen Ausdrucks wird durch eine im Programm erreichte Position dargestellt. Diese Methode hat Vorteile bei der Implementation von Ablaufstrukturen.

   Wichtig ist speziell bei dieser Methode die Frage, ob z.B. bei der Abarbeitung eines Ausdrucks $\texttt{E}_1$ `or` $\texttt{E}_2$ immer beide Ausdrücke abgearbeitet werden müssen, oder ob auf die Abarbeitung von $\texttt{E}_2$ verzichtet werden kann, falls $\texttt{E}_1$ bereits den Wert `true` hat. Diese „abgekürzte Auswertung“ wird in einigen Programmiersprachen gefordert (etwa in C), in anderen verboten (in Standard-Pascal von N. Wirth) und in einigen hat der Programmierer es in der Hand, welche Methode er wählt. (etwa in Java).

5.5.1 Numerische Kodierung

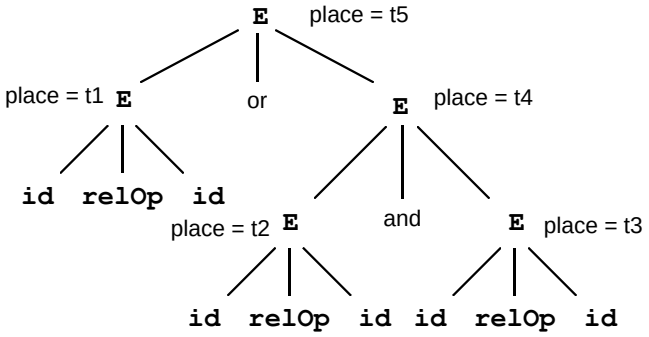
In diesem Fall werden Boolesche Ausdrücke wie arithmetische Ausdrücke behandelt. `E` bekommt ein synthetisches Attribut `place`, das den Namen der Variablen angibt, in der später der Wert des korrespondierenden Ausdrücks steht. Die Größe `nextstat` bezeichne immer den nächsten freien Platz im Feld der durch `emit` ausgegebenen Drei-Adress-Befehle. Sprungziele werden also in diesem Fall durch Indizes angegeben.

Produktion	Semantische Regel
$E \rightarrow E_1$ or E_2	E.place := newtemp(); emit (E.place ':=' E_1.place 'or' E_2.place);
$E \rightarrow E_1$ and E_2	E.place := newtemp(); emit (E.place ':=' E_1.place 'and' E_2.place);
$E \rightarrow$ not E_1	E.place := newtemp(); emit (E.place ':=' 'not' E_1.place);
$E \rightarrow$ (E_1)	E.place := E_1.place;

$E \rightarrow id_1$ relOp id_2	E.place := newtemp();
	emit ('if' id_1.place relOp.op id_2.place;
	'goto' nextstat+3);
	emit (E.place ':=' 0);
	emit ('goto' nextstat+2);
	emit (E.place ':=' '1');

$E \rightarrow$ true	E.place := newtemp(); emit (E.place ':=' '1');
$E \rightarrow$ false	E.place := newtemp(); emit (E.place ':=' '0');

Beispiel 5.9:
Im folgenden sieht man den attributierten Ableitungsbaum zu `a<b or c<d and e>f` und die Übersetzung in Drei-Adress-Befehle.




```
 1) if a<b goto 4
 2) t1 := 0
 3) goto 5
 4) t1 := 1
 5) if c<d goto 8
 6) t2 := 0
 7) goto 9
 8) t2 := 1
 9) if e>f goto 12
10) t3 := 0
11) goto 13
12) t3 = 1
13) t4 := t2 and t3
14) t5 := t1 or t4
```

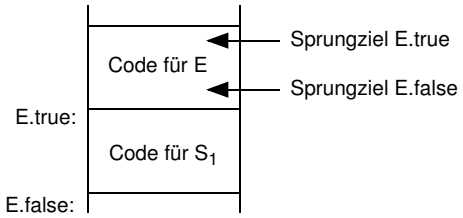
5.5.2 Steuerung des Programmablaufs

Betrachten wir zunächst Produktionen, die einige der Steuerbefehle in üblichen Programmiersprachen modellieren. Die Variable `S` steht dabei für eine Anweisung (**statement**) und die Variable `E` für einen Booleschen Ausdruck (**expression**).

```
S →  if E then S1 |
     if E then S1 else S2 |
     while E do S1
```

Betrachtet man die erste Produktion, so ist klar, dass zunächst der Drei-Adress-Code für `E` und danach der für S_1 erzeugt wird.

Die Sprungziele der bedingten und unbedingten Drei-Adress-Sprungbefehle im Code für `E` müssen also *vor* Abarbeitung von S_1 bekannt sein. Das ist das bekannte Problem der „forward references“, wie es auch z.B. in Assemblern auftritt. Die erste Lösungsmöglichkeit, die hier zunächst beschritten werden soll, besteht darin, Marken für die Sprungziele als Werte inheriter Attribute von `E` vorzugeben und diese Marken erst später als Sprungziel vor einem Drei-Adressbefehl zu schreiben.



`E` erhält also drei Attribute:

1) Das inherite Attribut `true` gibt das Sprungziel an, das in dem Fall angesprungen werden soll, wenn der Ausdruck den Wert `true` ergibt,

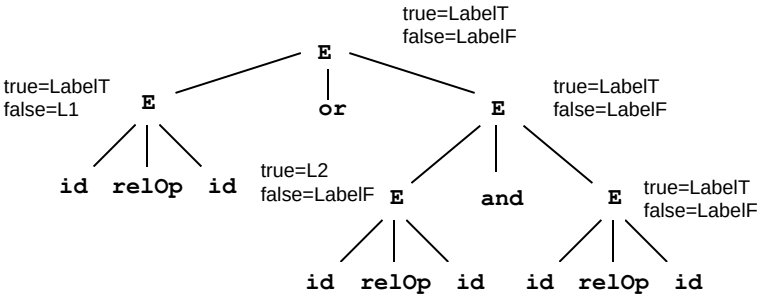
2) das inherite Attribut `false` gibt das Sprungziel an, das in dem Fall angesprungen werden soll, wenn der Ausdruck den Wert `false` ergibt und

3) das synthetische Attribut `code` enthält die Folge der Drei-Adress-Befehle zur Auswertung von `E`.

Damit kann man jetzt eine attributierte Grammatik für die Übersetzung Boolescher Ausdrücke aufstellen:

Produktion	Semantische Regel
$E \rightarrow E_1$ or E_2	E_1.true := E.true; E_1.false := newlabel(); E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen (E_1.false ':') || E_2.code;
$E \rightarrow E_1$ and E_2	E_1.true := newlabel(); E_1.false := E.false; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen (E_1.true ':') || E_2.code;
$E \rightarrow$ not E_1	E_1.true := E.false; E_1.false := E.true; E.code := E_1.code;
$E \rightarrow$ (E_1)	E_1.true := E.true; E_1.false := E.false; E.code := E_1.code;
$E \rightarrow id_1$ relOp id_2	E.code := gen ('if' id_1.place relOp.op id_2.place 'goto' E.true) || gen ('goto' E.false);
$E \rightarrow$ true	E.code := gen ('goto' E.true);
$E \rightarrow$ false	E.code := gen ('goto' E.false);

Beispiel 5.10:
Der Ausdruck `a<b or c<d and e>f` würde zu dem folgenden attributierten Ableitungsbaum führen und den angegebenen Drei-Adress-Code liefern. Hier wird angenommen, dass die Werte der inheriten Attribute des Startsymbols `E` an der Wurzel des Ableitungsbaumes „von außen“ zu `labelT` bzw. `labelF` gesetzt werden.




```
        if a<b goto LabelT
        goto L1
L1:     if c<d goto L2
        goto LabelF
L2:     if e>f goto LabelT
        goto LabelF
```

Nun ist es nicht mehr schwer, für Steueranweisungen in der Programmiersprache Zwischencode zu erzeugen. Betrachtet man etwa die Produktion `S` $\rightarrow$ `if E then` S_1 `else` S_2, so ist es natürlich möglich, dass z.B. S_1 oder S_2 wiederum bedingte oder unbedingte Sprünge enthalten, die als Sprungziel den nach dieser Anweisung folgenden Befehl haben. Also muss man auch dem Symbol `S` ein inherites Attribut zuordnen, das dieses Sprungziel angibt.
S hat also die folgenden Attribute:

1) das inherite Attribut `next` gibt das Sprungziel an, das in dem Fall angesprungen werden soll, wenn zu der `S` folgenden Anweisung gesprungen werden soll und

2) das synthetische Attribut `code` enthält die Folge der Drei-Adress-Befehle zur Auswertung von `S`.

Damit ergeben sich die folgenden Erweiterungen der obigen attributierten Grammatik:

Produktion	Semantische Regel

`S` $\rightarrow$ `if E then` S_1	`E.true := newlabel();`
	`E.false := S.next;`
	S_1`.next := S.next;`
	`S.code := E.code || gen (E.true ':') ||` S_1`.code;`

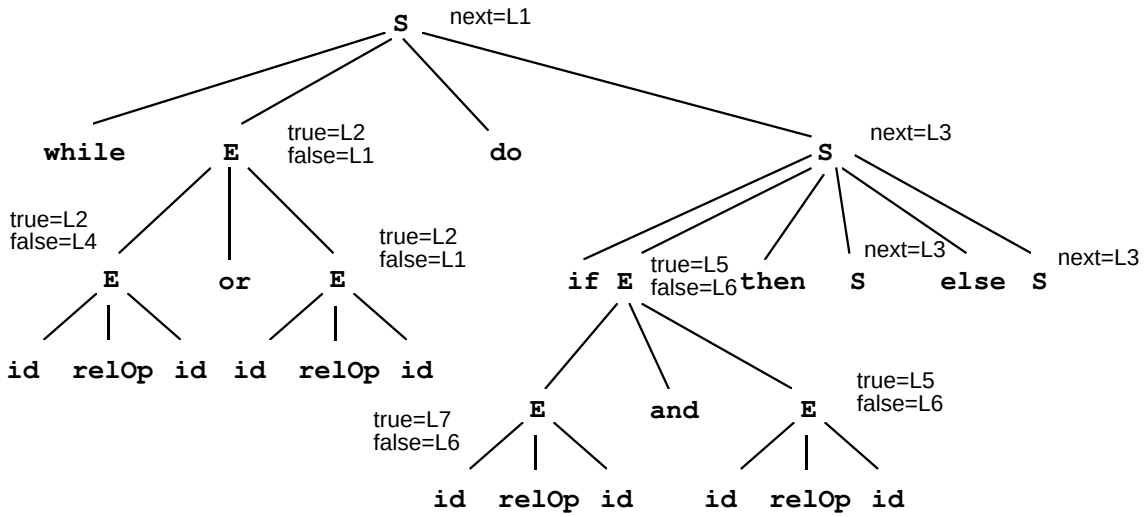
`S` $\rightarrow$ `if E then` S_1 `else` S_2	`E.true := newlabel();`
	`E.false := newlabel();`
	S_1`.next := S.next;`
	S_2`.next := S.next;`
	`S.code := E.code || gen (E.true ':') ||` S_1`.code ||`
	`gen ('goto' S.next) || gen (E.false ':') ||`
	S_2`.code;`

`S` $\rightarrow$ `while E do` S_1	`E.true := newlabel();`
	`E.false := S.next;`
	`beginlabel := newlabel();`
	S_1`.next := beginlabel;`
	`S.code := gen (beginlabel ':') || E.code ||`
	`gen (E.true ':') ||` S_1`.code ||`
	`gen ('goto' beginlabel);`

Beispiel 5.11:
Es soll folgendes Programmfragment mit der obigen attributierten Grammatik übersetzt werden:

```
while a<b or e>f do
    if c<d and g<h then
        x :=  y+z
    else
        x := y-z
```

Zunächst wird der zugehörige attributierte Ableitungsbaum betrachtet:



Man erhält durch die Auswertung den folgenden Drei-Adress-Code:

```
L3:  if a<b goto L2
     goto L4
L4:  if e>f goto L2
     goto L1
L2:  if c<d goto L7
     goto L6
L7:  if g<h goto L5
     goto L6
L5:  t1 := y+z
     x := t1
     goto L3
L6:  t2 := y-z
     x:= t2
     goto L3
```

5.6 Übersetzung Boolescher Ausdrücke mit Backpatching

Die Übersetzung Boolescher Ausdrücke mit der bisherigen Attributierung lässt sich nicht mit einer einfachen Top-Down Analyse koppeln, falls man als Sprungziele Indizes im Feld der abgespeicherten Drei-Adress-Befehle benutzen möchte. Das ist natürlich das übliche Problem bei Vorwärts-Referenzen, d.h. bei Sprüngen, deren Ziel ein nachfolgender Befehl ist. Es sind zwei Standardmethoden bekannt, wie man ein solches Problem lösen kann. Zum einen gibt es die übliche 2 Pass-Methode, bei der der Programmcode zweimal gelesen wird und es gibt die **Backpatch**-Methode, bei der unvollständige Sprungbefehle mit leerem Sprungziel erzeugt werden. Kommt man im Verlauf der weiteren Übersetzung an das eigentliche Sprungziel, so werden alle korrespondierenden unvollständigen Sprungbefehle vervollständigt, indem das jetzt bekannte Sprungziel nachgetragen wird.
Wir wollen uns vorstellen, dass die generierten Drei-Adress-Befehle in einem Feld abgelegt werden und wir wollen den Index des Feldes zur Lokalisierung von Befehlen benutzen. Das bedeutet, dass Marken jetzt Feldindizes sind. Bei der Übersetzung müssen also Listen mit Indizes von unvollständigen Sprungbefehlen manipuliert werden.
Zunächst werden drei Prozeduren zur Verwaltung derartiger Listen definiert:

1) `makelist(i)` erzeugt eine neue Liste mit einem Eintrag i und gibt einen Zeiger auf diese Liste zurück.

2) `merge(`p_1`, `p_2`)` erzeugt eine Liste, die aus den Elementen der beiden Listen besteht, auf die p_1 und p_2 verweisen. Es wird ein Zeiger auf die Ergebnisliste zurückgegeben.

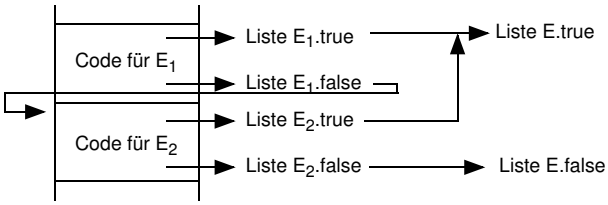
3) `backpatch(p, j)` ist die eigentliche Backpatch-Prozedur. `p` zeigt auf eine Liste mit Indizes unvollständiger Sprungbefehle. In jedem dieser Befehle wird durch diese Prozedur das Sprungziel `j` eingetragen.

Die Variable `E` erhält zwei synthetische Attribute, nämlich

1) `truelist` - ein Zeiger auf eine Liste der Indizes im Feld der Drei-Adress-Befehle. An diesen Indizes befinden sich die unvollständigen Sprungbefehle, über die aus dem Drei-Adress-Code von `E` gesprungen werden soll, falls der Ausdruck `E` wahr ist.

2) `falselist` - wie `truelist`, nur muss der Ausdruck `E` falsch sein.

Man kann nun im wesentlichen die Grammatik aus dem vorigen Abschnitt verwenden. Zur Verdeutlichung betrachten wir die Produktion $\texttt{E} \rightarrow \texttt{E}_1 \texttt{ or } \texttt{E}_2$.



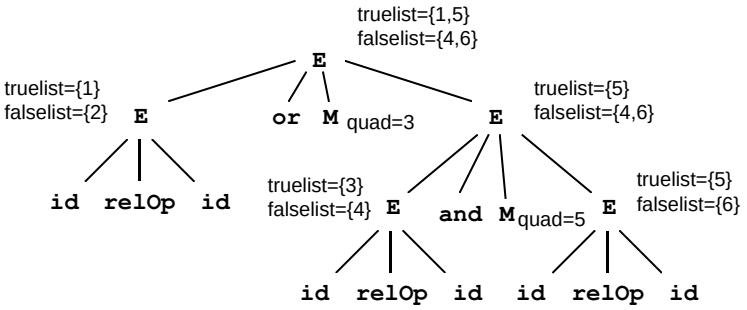
Wie man sieht, muss man die unvollständigen Sprünge in der Liste $\texttt{E}_1\texttt{.falselist}$ mit dem Index des ersten Drei-Adress-Befehls von $\texttt{E}_2$ als Sprungziel vervollständigen. Um diesen Wert festzuhalten, muss nach der Ableitung von $\texttt{E}_1$ und vor der Ableitung von $\texttt{E}_2$ der nächste freie Index im Feld der Drei-Adress-Befehle gespeichert werden. Zu diesem Zweck muss an einigen Stellen in den rechten Seiten der Produktionen eine neue, auf das leere Wort ableitbare Variable eingeführt werden, deren zugeordnete semantische Regel nur daraus besteht, die momentane Position im Feld der Drei-Adress-Befehle in einem synthetischen Attribut zu speichern.

Zu diesem Zweck führt man die Variable M mit der Produktion $M \rightarrow \varepsilon$ ein. M erhält ein synthetisches Attribut quad, dessen Wert den Index des nächsten freien Platzes im Feld der Drei-Adress-Befehle speichert.

Damit kann man dann die Übersetzung Boolescher Ausdrücke definieren:

Produktion	Semantische Regel
$E \rightarrow E_1$ or M E_2	backpatch (E_1.falselist, M.quad); E.truelist := merge (E_1.truelist, E_2.truelist); E.falselist := E_2.falselist;
$E \rightarrow E_1$ and M E_2	backpatch (E_1.truelist, M.quad); E.truelist := E_2.truelist; E.falselist := merge (E_1.falselist, E_2.falselist);
$E \rightarrow$ not E_1	E.truelist := E_1.falselist; E.falselist := E_1.truelist;
$E \rightarrow$ (E_1)	E.truelist := E_1.truelist; E.falselist := E_1.falselist;
$E \rightarrow id_1$ relOp id_2	E.truelist := makelist (nextquad()); E.falselist := makelist (nextquad()+1); emit ('if' id_1.place relOp.op id_2.place 'goto (_)'); emit ('goto (_)');
$E \rightarrow$ true	E.truelist := makelist (nextquad()); emit ('goto (_)');
$E \rightarrow$ false	E.falselist := makelist (nextquad()); emit ('goto (_)');
$M \rightarrow \varepsilon$	M.quad := nextquad();

Beispiel 5.12:
Der Ausdruck a<b or c<d and e>f würde mit diesem SDTS zu folgendem attributierten Ableitungsbaum führen und den angegebenen Drei-Adress-Code liefern:



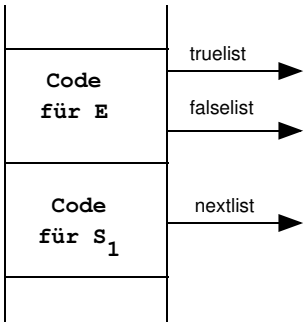

```
(1)    if a<b goto ( )
(2)    goto (3)
(3)    if c<d goto (5)
(4)    goto
(5)    if e>f goto ( )
(6)    goto ( )
```

5.7 Übersetzung von Programmsteuerbefehlen (mit Backpatching)

Im folgenden soll gezeigt werden, wie man mit Hilfe der „Backpatch-Methode“ auch Steuerbefehle elegant übersetzen kann. Wir erweitern die Grammatik für Boolesche Ausdrücke um die folgenden Produktionen, wobei `S` für eine Anweisung, `L` für eine Liste von Anweisungen und `A` für eine Wertzuweisung steht.

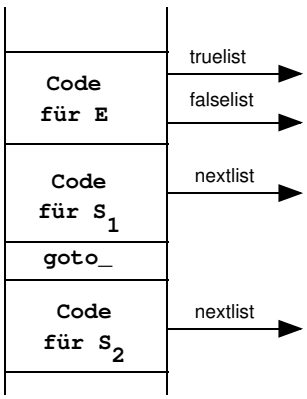
```
S →    if E then S1
     | if E then S1 else S2
     | while E do S1
     | begin L end
     | A
L →    L ; S
     | S
```

`S` und `L` bekommen jeweils ein synthetisches Attribut `nextlist` - ein Zeiger auf eine Liste der Indizes im Feld der Drei-Adress-Befehle, die die unvollständigen Sprungbefehle kennzeichnen, über die an die auf `S` bzw. `L` folgende Anweisung gesprungen werden soll.
Betrachten wir etwa die `while`-Anweisung. Es wird zunächst der Code für den Booleschen Ausdruck und danach der Code für den Rumpf erzeugt.



Alle Sprünge in `truelist` müssen auf den Anfang des Codes von S_1, alle Sprünge in `nextlist` müssen dagegen auf den Anfang des Codes von `E` verweisen. Man muss also, um die Sprungziele festzuhalten, an zwei Stellen in der rechten Seite der Produktion das zusätzliche nichtterminale Symbol `M` einführen. Mit den Werten des Attributs `quad` kann dann die `backpatch`-Prozedur für die beiden Listen aufgerufen werden. Das Attribut `nextlist` bekommt dann die Liste `falselist` übergeben. Am Ende muss noch ein unbedingter Sprung an den Anfang des Codes von `E` eingefügt werden, damit man nicht aus dem Rumpf der Schleifen „herausfallen“ kann.

Ähnliche Überlegungen muss man etwa bei der `if-then-else` Anweisung anstellen.



Alle Sprünge in `truelist` müssen auf den Anfang des Codes von S_1, alle Sprünge in `falselist` müssen dagegen auf den Anfang des Codes von S_2 verweisen. Beide Positionen müssen wieder durch Einfügen des zusätzlichen nichtterminalen Symbols `M` bereitgestellt werden. Damit nach Abarbeitung des Codes für S_1 nicht gleich in den Code für S_2 verzweigt wird, muss an dieser Stelle der unbedingte Sprung eingesetzt werden. Dies erreicht man wieder durch Einführung einer neuen Variablen `N` in die rechte Seite der Produktion und Hinzufügung einer Produktion $N \to \varepsilon$ mit der entsprechenden Regel. Da das Sprungziel nicht bekannt ist, erhält `N` ein Attribut `nextlist` mit der gleichen Bedeutung wie bei `S`. Das Attribut `nextlist` bekommt dann als Wert die Vereinigung der drei `nextlist` Listen von S_1, S_2 und `N` übergeben.

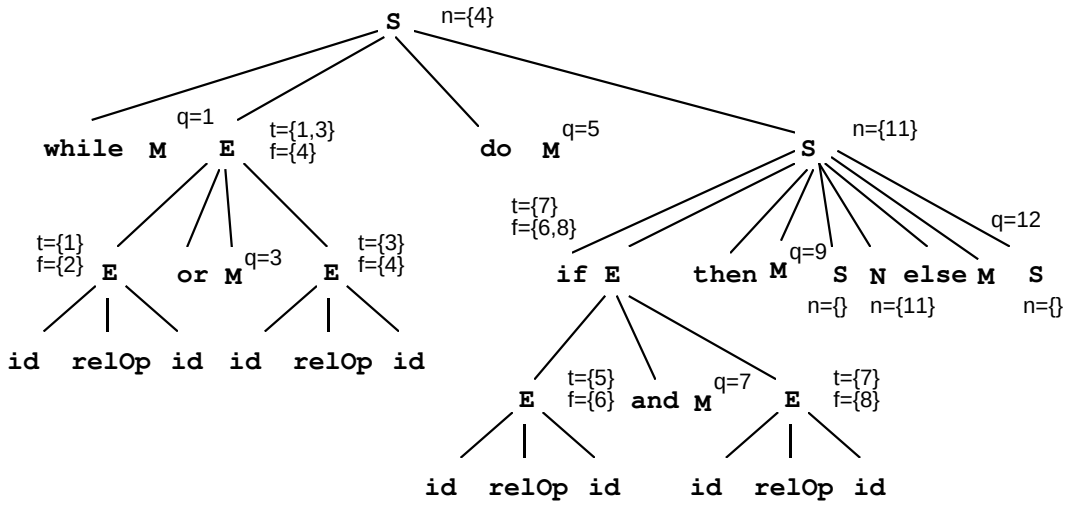
Mit diesen Überlegungen kann man nun ein SDTS für Steueranweisungen angeben:

Produktion	Semantische Regel
S → if E then M S_1	backpatch (E.truelist, M.quad); S.nextlist := merge (E.falselist, S_1.nextlist);
S → if E then M_1 S_1 N else M_2 S_2	backpatch (E.truelist, M_1.quad); backpatch (E.falselist, M_2.quad); S.nextlist := merge (S_1.nextlist, N.nextlist, S_2.nextlist);
S → while M_1 E do M_2 S_1	backpatch (S_1.nextlist, M_1.quad); backpatch (E.truelist, M_2.quad); S.nextlist := E.falselist; emit ('goto' M_1.quad);
S → begin L end	S.nextlist := L.nextlist;
S → A	S.nextlist = makelist();
L → L_1 ; M S	backpatch (L_1.nextlist, M.quad); L.nextlist := S.nextlist;
L → S	L.nextlist := S.nextlist;
M → ε	M.quad := nextquad();
N → ε	N.nextlist := makelist (nextquad()); emit ('goto' (_);

Beispiel 5.13:
Wie im Beispiel 5.11 soll jetzt das Programmfragment mit dem obigen SDTS übersetzt werden:

```
while a<b or e>f do
    if c<d and g<h then
        x :=  y+z
    else
        x := y-z
```

Zunächst wird der zugehörige attributierte Ableitungsbaum betrachtet:



Aus Platzgründen sind die Attributnamen abgekürzt. Es steht `n` für `nextlist`, `q` für `quad`, `t` für `truelist` und `f` für `falselist`.

Man erhält den folgenden Drei-Adress-Code:

```
( 1)    if a<b goto (5)
( 2)    goto (3)
( 3)    if e>f goto (5)
( 4)    goto (_)
( 5)    if c<d goto (7)
( 6)    goto (12)
( 7)    if g<h goto (9)
( 8)    goto (12)
( 9)    t1 := y+z
(10)    x := t1
(11)    goto (1)
(12)    t2 := y-z
(13)    x:= t2
(14)    goto (1)
```