Compilerkonstruktion

Wintersemester 2015/16

Prof. Dr. R. Parchmann

24. November 2015

Übersetzung Boolescher Ausdrücke

Boolesche Ausdrücke seien gemäß der folgenden Grammatik festgelegt:

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

- `relOp` ist einer der üblichen arithmetischen Vergleichsoperatoren
- die Operationen `or` und `and` sind linksassoziativ
- `not` hat eine höhere Priorität als `and` und `and` wiederum eine höhere als `or`

Möglichkeiten der Darstellung des Boolescher Werte

1. Durch numerische Kodierung, etwa $1 \mathrel{\hat{=}} \texttt{true}$ und $0 \mathrel{\hat{=}} \texttt{false}$ oder aber auch jeder Wert ungleich $0 \mathrel{\hat{=}} \texttt{true}$ und $0 \mathrel{\hat{=}} \texttt{false}$.
2. Durch die Steuerung des Programmablaufs, d.h. der Wert eines Booleschen Ausdrucks wird durch eine im Programm erreichte Position dargestellt.

Die zweite Methode hat signifikante Vorteile bei der Implementation von Ablaufstruktutren. 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.

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.

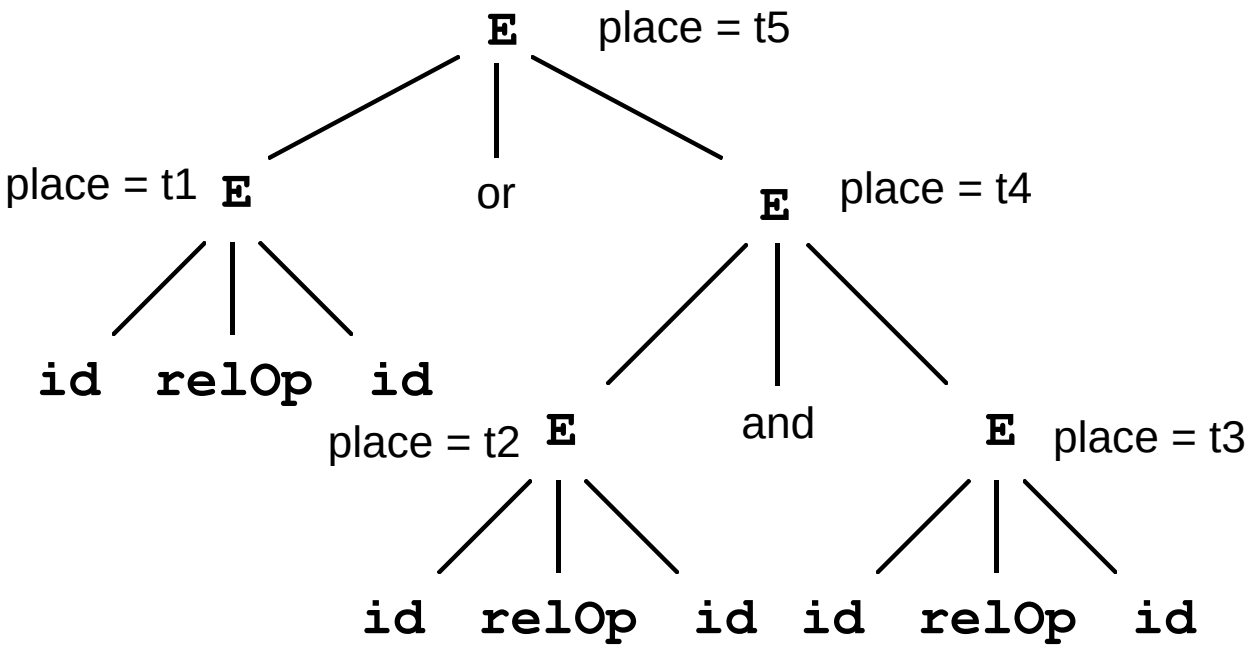
Produktion	*Semantische Regel*
$E \rightarrow E_1$ or E_2	E.place := newtemp(); emit (E.place ':=' E_1.place 'or' E_2.place);
$E \rightarrow E_1$ and E_2	E.place := newtemp(); emit (E.place ':=' E_1.place 'and' E_2.place);
$E \rightarrow$ not E_1	E.place := newtemp(); emit (E.place ':=' 'not' E_1.place);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

Attributierter Ableitungsbaum zu `a<b or c<d and e>f` und die Übersetzung in Drei-Adress-Befehle.



Erzeugter Drei-Adress-Code:

```
 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
```

Steuerung des Programmablaufs

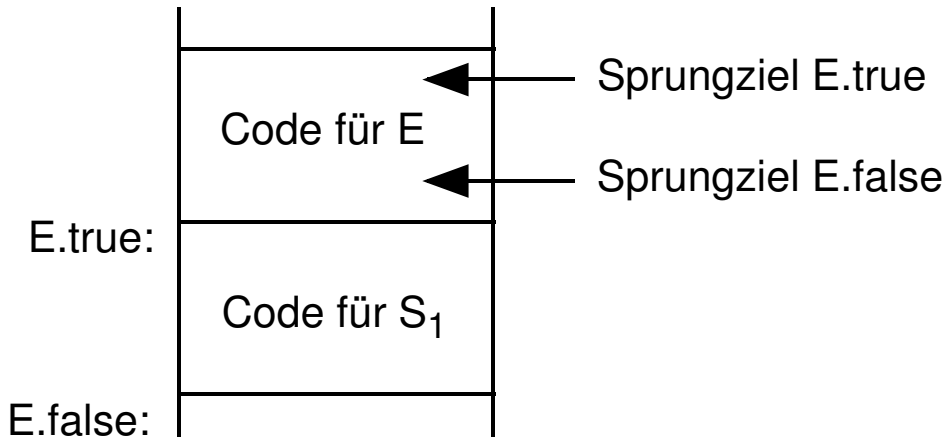
Betrachten wir zunächst Produktionen, die einige der Steuerbefehle in üblichen Programmiersprachen modellieren. Die Variable `S` steht dabei für eine Anweisung (*statement*) und die Variable `E` für einen Booleschen Ausdruck (*expression*).

$$
\begin{array}{lll}
\texttt{S} & \rightarrow & \texttt{if E then } \texttt{S}_1 \mid \\
 & & \texttt{if E then } \texttt{S}_1 \texttt{ else } \texttt{S}_2 \mid \\
 & & \texttt{while E do } \texttt{S}_1
\end{array}
$$

Betrachtet man die erste Produktion, so ist klar, dass zunächst der Drei-Adress-Code für `E` und danach der für $\texttt{S}_1$ erzeugt wird.

Die Sprungziele der bedingten und unbedingten Drei-Adress-Sprungbefehle im Code für E müssen also vor Abarbeitung von S_1 bekannt sein.

Eine Lösungsmöglichkeit: Sprungziele als Werte inheriter Attribute von E vorgeben



`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`.

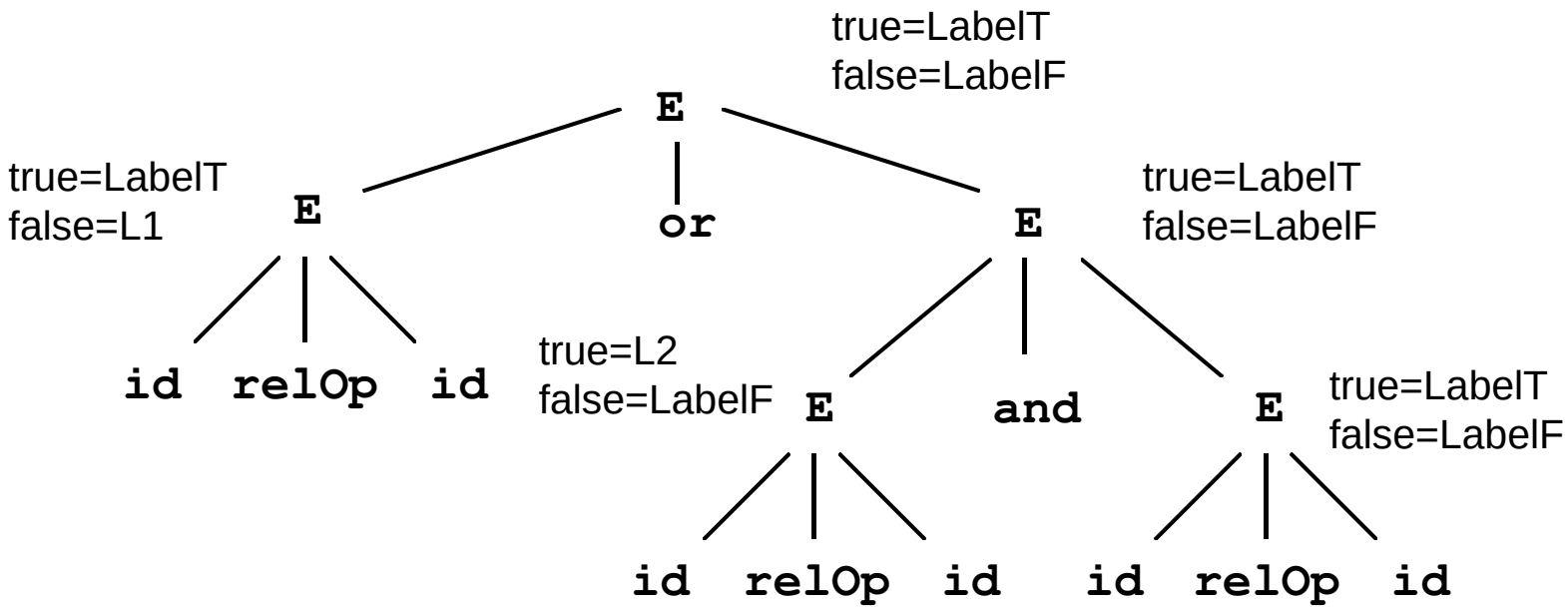
Damit kann man jetzt eine attributierte Grammatik für die Übersetzung Boolesche Ausdrücke aufstellen:

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;

Produktion	*Semantische Regel*
E → not E_1	E_1.true := E.false; E_1.false := E.true; E.code := E_1.code;
E → (E_1)	E_1.true := E.true; E_1.false := E.false; E.code := E_1.code;
E → 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 → true	E.code := gen ('goto' E.true);
E → false	E.code := gen ('goto' E.false);

Beispiel

Der Ausdruck `a<b or c<d and e>f` würde zu folgendem attributierten Ableitungsbaum führen:



Übersetzung von Steueranweisungen

Betrachtet man die Produktion $\mathtt{S} \rightarrow \mathtt{if\ E\ then\ S_1\ else\ S_2}$, dann ist es möglich, dass $\mathtt{S_1}$ oder $\mathtt{S_2}$ wiederum bedingte oder unbedingte Sprünge enthalten, die als Sprungziel den nach dier Anweisung folgenden Befehl haben.
Folglich muss man dem Symbol `S` ein inherites Attribut zuordnen, das dieses Sprungziel angibt.
Damit hat `S` 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;`

Produktion	*Semantische Regel*

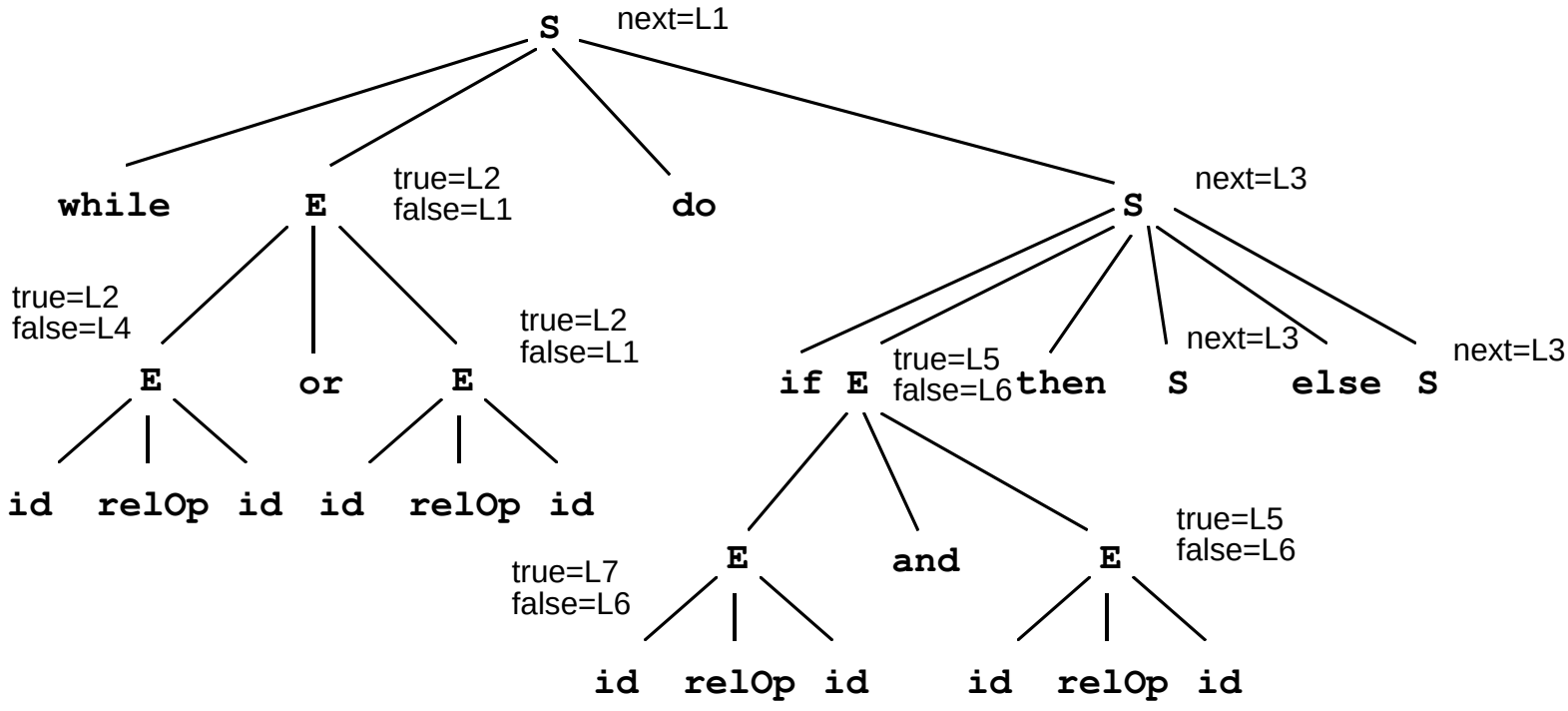
S $\rightarrow$ while E do S_1	E.true := newlabel();
	E.false := S.next;
	beginlabel := newlabel();
	S_1.next := beginlabel;
	S.code := gen (beginlabel ':')
	  || E.code || gen (E.true ':')
	  || S_1.code || gen ('goto' beginlabel);

Beispiel

Die Übersetzung des Programmfragments

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

liefert den attributierten Ableitungsbaum:



der zu folgendem Drei-Adress-Code führt:

```
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
```

Übersetzung Boolescher Ausdrücke mit Backpatching

Es sind zwei Standardmethoden bekannt, wie man das Problem der forward-references lösen kann.

1. die üblche 2 Pass-Methode, bei der der Programmcode zweimal gelesen wird und
2. die *Backpatch*-Methode, bei der unvollständige Sprungbefehle mit leerem Sprungziel erzeugt werden.

Kommt man bei der zweiten Methode im Verlauf der weiteren Übersetzung an das eigentliche Sprungziel, so werden alle korrespondierenden unvollständigen Sprungbefehle vervollständigt.

Implementation des Bachpatchings

Bei der Übersetzung müssen Listen mit Indizes von unvollständigen Sprungbefehlen manipuliert werden. Dazu benötigt man drei Prozeduren:

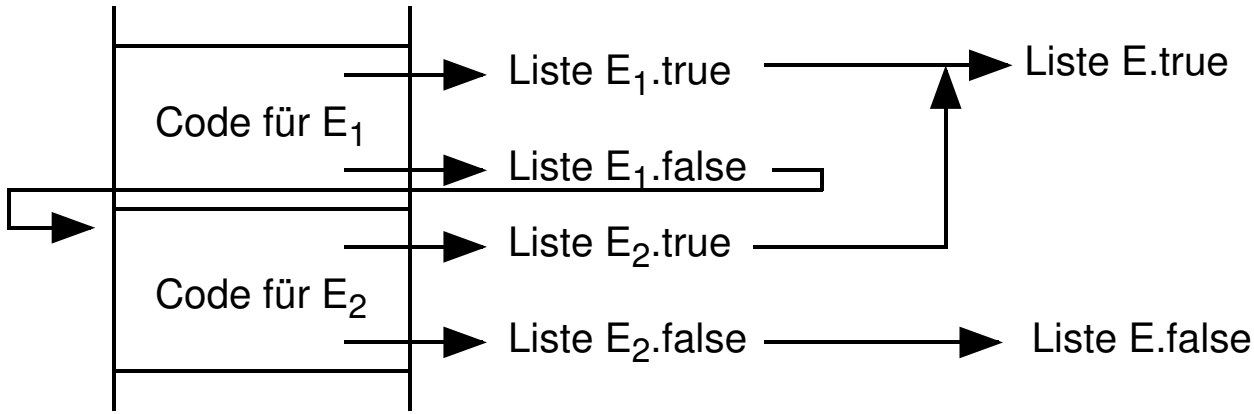
1. `makelist(i)` erzeugt eine neue Liste mit einem Eintrag i und gibt einen Zeiger auf diese Liste zurück.
2. `merge(`p_1`, `p_2`)` erzeugt eine Liste, die aus den Elementen der beiden Listen besteht, auf die p_1 und p_2 verweisen. Es wird ein Zeiger auf die Ergebnisliste zurückgegeben.
3. `backpatch(p, 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 von Indizes im Feld der Drei-Adress-Befehle, die unvollständige Sprungbefehle kennzeichnen, über die 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 $\mathtt{E} \rightarrow \mathtt{E}_1 \ \mathtt{or} \ \mathtt{E}_2$.

Man erhält folgendes Bild:



SDTS zur Übersetzung Boolescher Ausdrücke

Produktion	*Semantische Regel*
E → 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 → 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 → not E_1	E.truelist := E_1.falselist;
	E.falselist := E_1.truelist;
E → (E_1)	E.truelist := E_1.truelist;
	E.falselist := E_1.falselist;

Produktion	*Semantische Regel*
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 $\rightarrow$ ϵ	M.quad := nextquad();

Übersetzung von Programmsteuerbefehlen

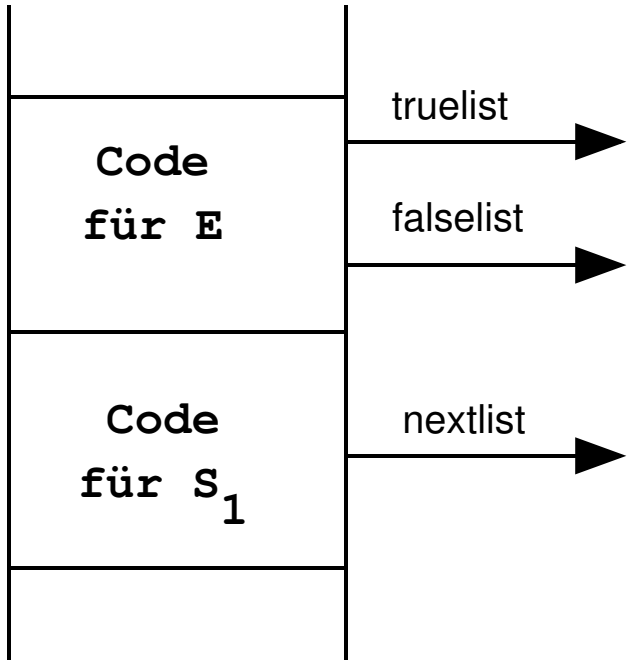
Im folgend soll gezeigt werden, wie man mit Hilfe der „Backpatch-Methode“ auch Steuerbefehle übersetzen kann.

$$
\begin{array}{lll}
\texttt{S} \rightarrow & & \texttt{if E then } \texttt{S}_1 \\
& \mid & \texttt{if E then } \texttt{S}_1 \texttt{ else } \texttt{S}_2 \\
& \mid & \texttt{while E do } \texttt{S}_1 \\
& \mid & \texttt{begin L end} \\
& \mid & \texttt{A} \\
\texttt{L} \rightarrow & & \texttt{L ; S} \\
& \mid & \texttt{S}
\end{array}
$$

S und L bekommen jeweils ein synthetisches Attribut `nextlist` - ein Zeiger auf eine Liste von unvollständigen Sprungbefehle, über die an die auf S bzw. L folgende Anweisung gesprungen werden soll.

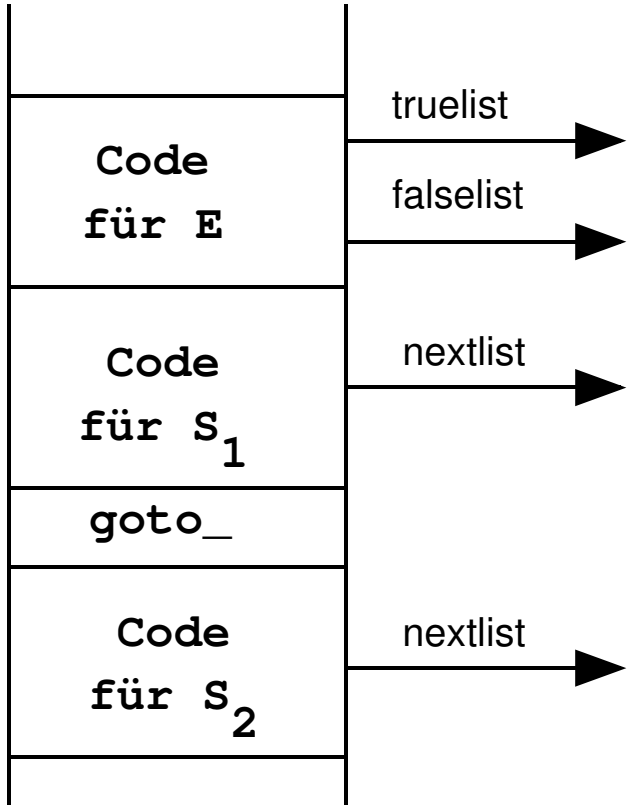
Beispiel

Übersetzung der `while`-Anweisung.
Es wird zunächst der Code für den Booleschen Ausdruck und danach der Code für den Rumpf erzeugt.



Beispiel

Übersetzung der `if-then-else` Anweisung.



SDTS für Steueranweisungen

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);

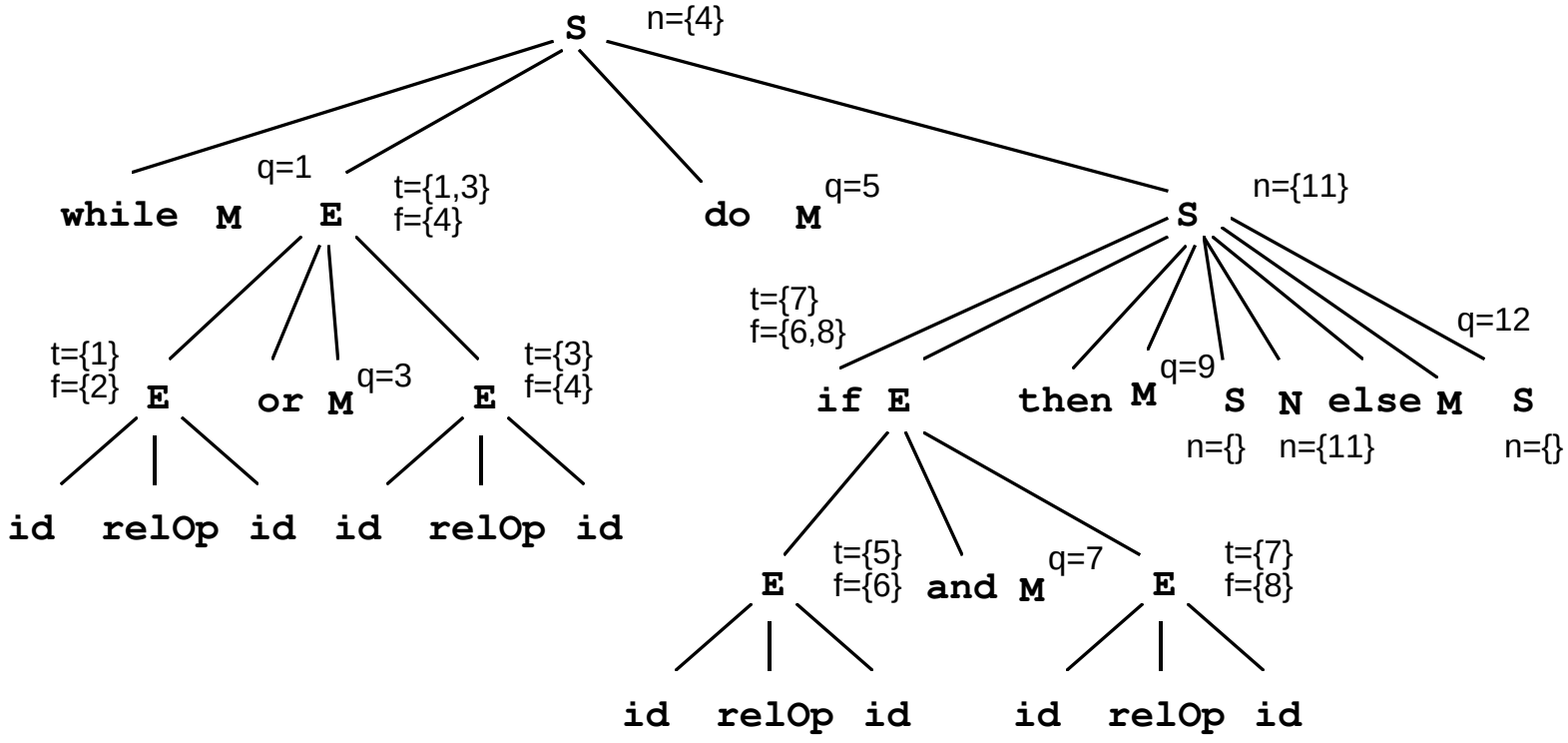
Produktion	*Semantische Regel*
S $\rightarrow$ begin L end	S.nextlist := L.nextlist;
S $\rightarrow$ A	S.nextlist = makelist( );
L $\rightarrow$ L_1 ; M S	backpatch (L_1.nextlist, M.quad);
	L.nextlist := S.nextlist;
L $\rightarrow$ S	L.nextlist := S.nextlist;
M $\rightarrow \epsilon$	M.quad := nextquad();
N $\rightarrow \epsilon$	N.nextlist := makelist (nextquad());
	emit ('goto' (_));

Beispiel

Die Übersetzung des Programmfragments

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

liefert den attributierten Ableitungsbaum:



dabei steht `n` für `nextlist`, `q` für quad, `t` für `truelist` und `f` für `falselist`.

Man erhält folgendem 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)
```