5 Zwischencode-Erzeugung

Warum verwendet man einen Zwischencode und erzeugt nicht direkt ein Programm in der Zielsprache? Da gibt es mehrere Gründe:

1) Leichteres Ändern des Compilers auf eine neue Zielsprache – im Idealfall müsste nur der Maschinencode-Erzeuger geändert werden (das sogenannte **Back-End** des Compilers).

2) Die Code-Optimierung ist weitestgehend unabhängig von der Zielsprache und kann somit für mehrere Compiler benutzt werden.

3) Es ist einfacher, eine Übersetzung in zwei (oder mehr) kleineren Schritten als in einem großen Schritt durchzuführen.

5.1 Form des Zwischencodes

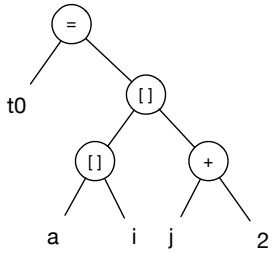
Man unterscheidet die folgenden Hauptkategorien von Formen des Zwischencodes:

- HIL - High-level Intermediate Languages werden nur in den ersten Stufen des Übersetzungsprozesses benutzt.
- MIL - Medium-level Intermediate Languages spiegeln die Möglichkeiten der Quellsprache in sehr vereinfachter Form.
- LIL - Low-level Intermediate Languages sind nahe an der Zielsprache.

Beispiel 5.1:
Ein Feld `a` sei deklariert als `float a[20][10]`, wobei die untere Feldgrenze jeweils 0 sei.
Der Ausdruck `a[i][j+2]` würde etwa übersetzt werden

HIL: in einen Ausdruck wie etwa

`t0 = a[i,j+2]` oder in einen Syntaxbaum wie



oder auch in einen gerichteten, azyklischen Graphen (DAG), wenn man gemeinsame Teilausdrücke berücksichtigen will.

MIL: in ein sehr einfach strukturiertes Programm, in dem jeder Befehl nur eine elementare Operation ausführt:

```
t1 = j + 2
t2 = i * 10
t3 = t1 + t2
t4 = 4 * t3          /* Größe einer Float-Zahl sei 4 Bytes */
t0 = a[t4]            /* a[t4] entspricht Anfangsadresse von a + t4 */
```

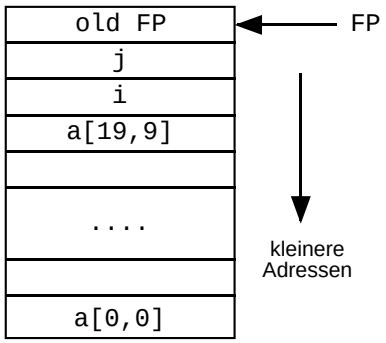
wobei die `ti` temporäre Namen sind.

Es wird also der Ausdruck (`addr a`) $+ 4 * (i * 10 + j + 2)$ berechnet. Jeder Befehl der Zwischensprache enthält nur eine Operation. Über die Speicherung des Feldes `a` muss nur bekannt sein, dass sie zeilenweise erfolgt und dass jedes Feldelement 4 Bytes belegt.

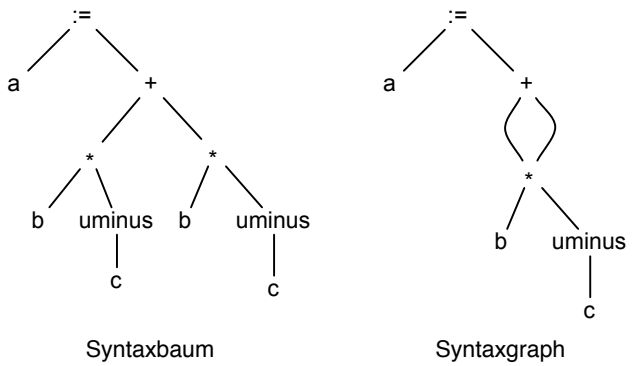
LIL: in eine einfache Art von Maschinensprachen eines hypothetischen Rechners unter Berücksichtigung der Speicherzuordnung.

Hier wird angenommen, dass das Feld `a` lokal definiert ist und der Speicherplatz für `a` und die beiden Variablen `i` und `j` im Aktivierungs-Record reserviert wurde. Ein Frame-Pointer zeigt auf den Beginn des Aktivierungs-Records, dort sei zunächst Platz für die Variablen `j` und `i` reserviert und das Feld `a` schließt sich direkt an. Weiterhin sei angenommen, dass jede Integergröße 2 Bytes lang ist. Der Laufzeitstack wachse außerdem zu kleineren Adressen hin und `FP` bezeichne den Framepointer. Ein Ausdruck der Form `[c]` bezeichnet den Inhalt der Speicherzelle mit Adresse `c`. Die `ri` sind (fiktive) Register, die `fi` (fiktive) spezielle Floating-Point Register.

```
r1 <- [FP-2]     /* r1 enthält j */
r2 <- r1 + 2
r3 <- [FP-4]     /* r3 enthält i */
r4 <- r3 * 10
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- FP - 804   /* Startadresse */
                 /* des Feldes a */
f1 <- [r7+r6]
```



Beispiel 5.2:
Es soll beispielhaft die Übersetzung von Wertzuweisungen in Syntaxbäume betrachtet werden. Gegeben sei der Ausdruck `a := b*-c+b*-c`, dann wäre der zugehörige Syntaxbaum bzw. Syntaxgraph etwa



Wie kann man derartige Übersetzungen nun mit Hilfe einer attributierten Grammatik erzeugen? Betrachten wir zunächst das Problem für Syntaxbäume.
Gegeben seien zwei Funktionen:

- `mkleaf(id, id.entry)` erzeugt ein Blatt mit Markierung `id` und einem Zeiger in die Symboltabelle auf den korrespondierenden Eintrag.
- `mknode(op, left, right)` erzeugt einen internen Knoten mit Markierung `op` und zwei Zeigern mit Werten `left` und `right`. Die Variante `mknode(op, link)` erzeugt einen internen Knoten mit nur einem Zeiger.

Beide Funktionen geben einen Zeiger auf den erzeugten Knoten zurück.

Mit Hilfe dieser Funktionen ist es nun leicht, eine attributierte Übersetzung von Wertzuweisungen festzulegen. `nptr` ist ein synthetisches Attribut für jedes nichtterminale Symbol der Grammatik, das auf den Wurzelknoten des zugehörigen Syntaxbaums verweist.

Ein SDTS für diese Übersetzung wäre etwa:

Produktion	Semantische Regel
S → id := E	S.nptr := mknode('assign', mkleaf(id, id.entry), E.nptr)
$E \rightarrow E_1$ + E_2	E.nptr := mknode('+', E_1.nptr, E_2.nptr)
$E \rightarrow E_1$ * E_2	E.nptr := mknode('*', E_1.nptr, E_2.nptr)
E → - E_1	E.nptr := mknode('uminus', E_1.nptr)
E → (E_1)	E.nptr := E_1.nptr
E → id	E.nptr := mkleaf(id, id.entry)

Will man Syntaxgraphen erzeugen, so müssen die Funktion `mknode` und `mkleaf` vor dem Erzeugen eines neuen Knotens prüfen, ob ein entsprechender Knoten mit entsprechenden Nachfolgern bereits existiert. Falls dies der Fall ist, erzeugt man keinen neuen Knoten, sondern gibt den gefundenen Knoten zurück.

5.2 Drei-Adress-Befehle

Wir werden im Folgenden als Zwischencode einen sogenannten Drei-Adress-Code benutzen. Der Drei-Adress-Code ist eine MIL-Zwischensprache.

Beispiele für Drei-Adress-Befehle:

- Wertzuweisungen der Form: `x := y op z`, `op` ∈ { `+, -, *, /, <, =, ...` }
- Wertzuweisungen der Form: `x := op y`, `op` ∈ { `-, not, intToReal, ...` }
- Wertzuweisungen der Form: `x := y`
- Wertzuweisungen der Form `x := y[i]` und `x[i] := y`,
  dabei bedeutet `y[i]`: Adresse von `y + i` Speichereinheiten
- Wertzuweisungen der Form: `x := &y` (Adresse von `y`),
  `x := *y` (Dereferenzierung) oder `*x := y`
- Unbedingte Sprünge der Form: `goto L`, dabei ist `L` eine Marke.
  (Marken können vor jedem Drei-Adress-Befehl auftreten.)
- Bedingte Sprünge der Form: `if x relOp y goto L`, `relOp` ∈ { `<, =, ...` }.

Bei den Sprungbefehlen kann man zwei Arten von Sprungzielen verwenden. Eine erste Möglichkeit ist es, einzelnen Drei-Adress-Befehlen eine symbolische Marke zuzuordnen. Eine zweite Möglichkeit besteht darin, die Drei-Adress-Befehle in ein Feld zu speichern und den Index des Feldes als Sprungziel zu verwenden.

Beispiel 5.3:
Betrachtet man die Anweisung `do i = i+1; while (a[i] < v);` so wären die folgenden Übersetzungen in Drei-Adress-Code möglich:

```
L:   t1 = i + 1                  100:  t1 = i + 1
     i = t1                      101:  i = t1
     t2 = i * 8                  102:  t2 = i * 8
     t3 = a [ t2 ]               103:  t3 = a [ t2 ]
     if t3 < v goto L            104:  if t3 < v goto 100
```

Betrachten wir nun die Übersetzung für Wertzuweisungen mit einfachen arithmetischen Ausdrücken in Drei-Adress-Code. In der folgenden S-attributierten Grammatik sind der Variablen `E`, die für einen arithmetischen Ausdruck steht, zwei Attribute zugeordnet:

1) `E.place`: Namen der Variablen im Drei-Adress-Code, die den Wert des zu `E` korrespondierenden Ausdrucks enthält

2) `E.code`: Folge von Drei-Adress-Befehlen, die den zu `E` korrespondierenden Ausdruck auswertet.

`newtemp()` erzeugt bei jedem Aufruf einen neuen, temporären Variablennamen und
`gen( ... )` erzeugt einen entsprechenden Drei-Adress-Befehl.

Produktion	Semantische Regel
`S` $\rightarrow$ `id := E`	`S.code := E.code || gen (id.lexeme ':=' E.place)`
`E` $\rightarrow$ E_1 `+` E_2	`E.place := newtemp()`
	`E.code :=` E_1`.code ||` E_2`.code ||`
	`gen (E.place ':='` E_1`.place +` E_2`.place)`
`E` $\rightarrow$ E_1 `*` E_2	`E.place := newtemp()`
	`E.code :=` E_1`.code ||` E_2`.code ||`
	`gen (E.place ':='` E_1`.place *` E_2`.place)`
`E` $\rightarrow$ `-` E_1	`E.place := newtemp()`
	`E.code :=` E_1`.code || gen (E.place ':=' 'uminus'` E_1`.place)`
`E` $\rightarrow$ `(` E_1 `)`	`E.place :=` E_1`.place`
	`E.code :=` E_1`.code`
`E` $\rightarrow$ `id`	`E.place := id.lexeme`
	`E.code := ''`

Wenn man sich die Regeln zur Code-Erzeugung für die obigen Produktionen ansieht, stellt man fest, dass alle nach dem gleichen Schema aufgebaut sind:

Einer Produktion `A` $\rightarrow$ `BC` ist eine Regel der Form

$$\texttt{A.code := string}_B \texttt{ || B.code || string}_C \texttt{ || C.code || string}_D$$

zugeordnet.
Man nennt derartige Regeln, in denen Zeichenkettenattribute der einzelnen nichtterminalen Symbole in der Reihenfolge ihres Auftretens in der rechten Seite der Produktion mit eventuell weiteren konstanten Zeichenketten konkateniert werden **einfach**.
Derartige Regeln können ersetzt werden durch

$$\texttt{A} \rightarrow \{\texttt{emit(string}_B\texttt{)}\}\ \texttt{B}\ \{\texttt{emit(string}_C\texttt{)}\}\ \texttt{C}\ \{\texttt{emit(string}_D\texttt{)}\}$$

wobei `emit( .... )` die als Parameter übergebene Zeichenkette ausgibt.

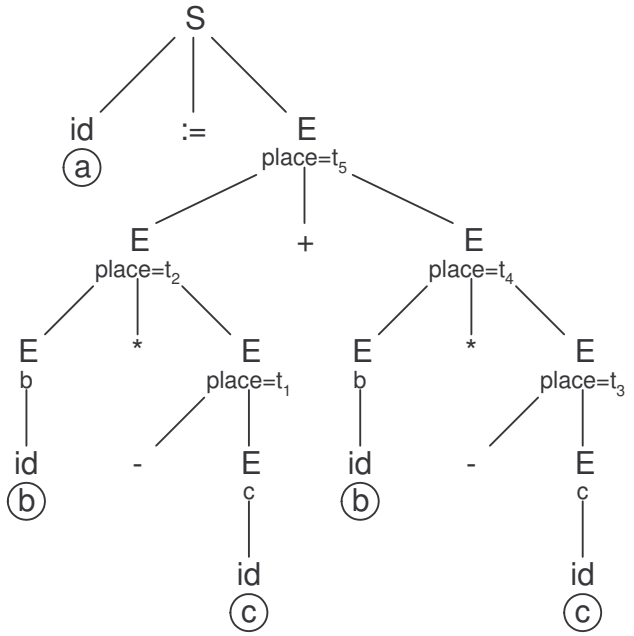
Die Funktion `emit` ist eine Funktion mit Seiteneffekten, d.h. man muss auf die Reihenfolge der Auswertung achten!

Mit diesen Änderungen erhalten wir das folgende SDTS zur Übersetzung in Drei-Adress-Code:

Produktion	Semantische Regel
S → id := E	emit(id.lexeme ':=' E.place)
E → E_1 + E_2	E.place := newtemp(); emit(E.place ':=' E_1.place + E_2.place)
E → E_1 * E_2	E.place := newtemp(); emit(E.place ':=' E_1.place * E_2.place)
E → - E_1	E.place := newtemp(); emit(E.place ':=' 'uminus' E_1.place)
E → (E_1)	E.place := E_1.place
E → id	E.place := id.lexeme

Man kann sich vorstellen, dass die Funktion `emit` die so erzeugten Drei-Adress-Befehle etwa in ein größeres Feld oder in eine Datei schreibt, damit die nachfolgenden Stufen des Compilers das gesamte Drei-Adress-Programm weiterverarbeiten können.

Beispiel 5.4:
Für den Ausdruck aus Beispiel 5.2 erhält man damit den ausgewerteten Ableitungsbaum und Drei-Adress-Befehle




```
t1 := -c
t2 := b * t2
t3 := -c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

5.2.1 Darstellung von Drei-Adress-Befehlen

Es gibt zwei wesentliche Möglichkeiten der Darstellung von Drei-Adress-Befehlen:

Quadrupel-Darstellung:
Ein Befehl wird durch einen Record mit 4 Feldern dargestellt, etwa

Operator	1. Operand	2. Operand	Ergebnis

also z.B. für den Befehl `x := y * z` den Record

*	y	z	x

,

wobei `x`, `y` und `z` Zeiger zum korrespondierenden Eintrag in der Symboltabelle sind. *Temporäre Namen müssen also bei dieser Methode in die Symboltabelle eingetragen werden.*

Tripel-Darstellung: (das Ergebnis wird durch den Befehl selbst repräsentiert).
Ein Befehl wird durch einen Record mit 3 Feldern dargestellt, etwa

Operator	1. Operand	2. Operand

also z.B. für den Befehl `x := y * z` den Record

*	y	z

,

wobei `y` und `z` Zeiger zum korrespondierenden Eintrag in der Symboltabelle sind oder aber auf einen Drei-Adress-Befehl verweisen.

Beispiel 5.5:
Betrachten wir wieder den Ausdruck `a := b*-c+b*-c`. Äquivalente Folgen von Drei-Adress-Befehlen wären:

Quadrupel-Darstellung

Nr.	Op	1. Op	2. Op	Erg.
0	uminus	c		t1
1	*	b	t1	t2
2	uminus	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	:=	t5		a

und

Tripel-Darstellung

Nr.	Op	1. Op	2. Op
0	uminus	c	
1	*	b	(0)
2	uminus	c	
3	*	b	(2)
4	+	(1)	(3)
5	:=	a	(4)

wobei `(i)` den Index des Drei-Adress-Befehls darstellt, mit dem der entsprechende Operand berechnet wurde.

Die Tripel-Darstellung hat den Vorteil, dass sie weniger Speicherplatz benötigt und die Symboltabelle nicht mit temporären Namen überflutet wird. Nachteilig ist, dass eine Umordnung der Befehle, etwa in der Optimierungsphase, nur schwierig zu realisieren ist und dass es bei einigen Drei-Adress-Befehlen (z.B. beim Befehl `x[i] := y` oder `x := y[i]`) keine vernünftige, einfache Darstellung in Tripelform gibt.
Es ist in der Quadrupel-Darstellung möglich, die Anzahl der temporären Namen durch Wiederbenutzung signifikant zu verringern, allerdings handelt man sich bei der späteren Code-Optimierung dadurch zusätzliche Probleme ein.

5.3 Syntax-gesteuerte Übersetzung von Deklarationen

Als erstes soll jetzt ein SDTS zur Berechnung des Typs und des Offsets im Aktivierungs-Record für lokale Variablen vorgestellt werden. An dieser Stelle muss bereits etwas über die Zielmaschine bekannt sein – für jeden elementaren Datentyp benötigt man die Anzahl Bytes zur Darstellung dieser Größen. Hinzu kommen noch Informationen über Wortgrenzen und Alignment, denn z.B. kann eine `float`-Zahl bei einigen Zielmaschinen nicht auf jeder Adresse abgelegt werden. Dies soll hier jedoch nicht berücksichtigt werden.

Produktion	Semantische Regel
`P →` `D`	`offset := 0;`
`D → D ; D`	
`D → id : T`	`enter (id.name, T.type, offset);` `offset := offset+T.width;`
`T → integer`	`T.type := integer;` `T.width := 4;`
`T → real`	`T.type := real;` `T.width := 8;`
`T → array [num] of` T_1	`T.type := array (num.val,` T_1`.type);` `T.width := num.val *` T_1`.width;`
`T →` $\uparrow T_1$	`T.type := pointer (`T_1`.type);` `T.width := 4;`

Bemerkungen zum obigen SDTS zur Übersetzung von Deklarationen:
`T` hat zwei synthetische Attribute:

1) `type` enthält den korrespondierenden Typausdruck
2) `width` enthält die zur Speicherung eines korrespondierenden Objektes benötigte Anzahl Bytes.

Es wird angenommen, dass der untere Index eines Feldes 0 ist. `offset` ist eine globale Größe, die zu Beginn auf 0 gesetzt wird. Damit dies geschieht, muss `offset` *vor* dem Ableiten von `D` in der ersten Produktion initialisiert werden. Dies erreicht man entweder mit einer Aktion, die innerhalb der rechten Seite der Produktion ausgeführt werden muss oder aber alternativ mit einer Umformung der Grammatik.
Die erste Produktion kann umgeschrieben werden zu

```
P → M D
M → ε        { offset := 0 }
```

`enter(name,type,offset)` erzeugt einen Eintrag in der Symboltabelle für `name` mit dem Typ `typ` und der relativen Adresse `offset` im Aktivierungs-Record.

Damit haben wir eine S-attributierte Grammatik erzielt, die die Deklaration abarbeitet und die notwendigen Einträge in die Symboltabelle vornimmt.

Bei einer Programmiersprache mit Blockstruktur können innere Blöcke natürlich so nicht abgearbeitet werden. Man benötigt eine anders strukturierte Symboltabelle, in der die Gültigkeitsbereiche der deklarierten Variablen sowie die Stufenzahl berücksichtigt werden muss. Der eigentliche Ablauf ist aber natürlich sehr ähnlich dem hier gezeigten und soll nicht weiter behandelt werden.

Beispiel 5.6:
Die Deklarationen

```
a: array [10] of real;
b: ↑ array [10] of integer;
c: real;
```

würden in der Symboltabelle zu den folgenden Einträgen führen:

a	array(10,real)	0
b	pointer(array(10,integer))	80
c	real	84

5.4 Übersetzung von Feldzugriffen

Es wird angenommen, dass die Feldelemente fortlaufend abgespeichert werden und dass jedes Element des Feldes eine feste Größe von `w` Bytes hat.

5.4.1 Eindimensionale Felder

Die Deklaration des Feldes sei `a: array[low..high] of ....`

Dann ergibt sich die Adresse des Feldelements `a[i]` zu

$$\texttt{base} + (\texttt{i} - \texttt{low}) * \texttt{w},$$

wobei `base` die Anfangsadresse des Speicherbereichs für das Feld `a` und `low` der Index des ersten Elementes in `a` ist, d.h. `a[low]` beginnt auf Adresse `base`.

Durch Umformung erhält man die Summe

$$\texttt{i} * \texttt{w} + \texttt{base} - \texttt{low} * \texttt{w},$$

wobei der zweite Summand eventuell bereits zur Übersetzungszeit bestimmt werden kann.

5.4.2 Zweidimensionale Felder

Die Deklaration des Feldes sei

$$\texttt{a: array[low}_1\texttt{..high}_1\texttt{,low}_2\texttt{..high}_2\texttt{] of ....}$$

Es gibt zwei prinzipielle Möglichkeiten:

- zeilenweise Speicherung (row-major), wird z.B. in Pascal angewendet
- spaltenweise Speicherung (column-major), wird z.B. in Fortran angewendet.

Wir wollen im folgenden eine zeilenweise Speicherung annehmen.

Dann ergibt sich die Adresse des Feldelements $\texttt{a[i}_1\texttt{,i}_2\texttt{]}$ zu

$$\texttt{base} + ((\texttt{i}_1 - \texttt{low}_1) * \texttt{n}_2 + \texttt{i}_2 - \texttt{low}_2) * \texttt{w}$$

wobei $\texttt{n}_2$ `=` $\texttt{high}_2$ `-` $\texttt{low}_2$ `+1`, d.h. die Anzahl der Spalten angibt.

Auch hier kann man die Formel umformen in einen Summanden, der zur Laufzeit berechnet werden muss und einen zweiten, der eventuell bereits zur Übersetzungszeit bekannt ist.
Man erhält:

$$((\texttt{i}_1 * \texttt{n}_2) + \texttt{i}_2) * \texttt{w} + (\texttt{base} - ((\texttt{low}_1 * \texttt{n}_2) + \texttt{low}_2) * \texttt{w}).$$

5.4.3 Mehrdimensionale Felder

Die Deklaration des Feldes sei nun

$$\texttt{a: array[low}_1\texttt{..high}_1\texttt{,low}_2\texttt{..high}_2\texttt{,..., low}_k\texttt{..high}_k\texttt{] of ....}$$

Verallgemeinert man die zeilenweise Speicherung auf mehr als zwei Dimensionen, dann erhält man eine sequentielle Speicherung der Feldelemente in einer Form, dass der letzte Index am schnellsten variiert.
Dann ergibt sich die Adresse des Feldelements $\texttt{a[i}_1\texttt{,i}_2\texttt{,...,i}_k\texttt{]}$ zu

$$\texttt{base} + ((\ldots(\texttt{i}_1 - \texttt{low}_1) * \texttt{n}_2 + \texttt{i}_2 - \texttt{low}_2) * \texttt{n}_3 + \ldots + \texttt{i}_\texttt{k} - \texttt{low}_\texttt{k}) * \texttt{w}$$

wobei $\texttt{n}_\texttt{i} = \texttt{high}_\texttt{i} - \texttt{low}_\texttt{i} + 1$ für $1 < \texttt{i} \leq \texttt{k}$ gilt.
Durch Umformung erhält man wie oben einen Summanden, der zur Laufzeit berechnet werden muss und einen zweiten, der eventuell bereits zur Übersetzungszeit bekannt ist:

$$\begin{aligned}&((\ldots(\texttt{i}_1 * \texttt{n}_2) + \texttt{i}_2) * \texttt{n}_3 + \ldots) * \texttt{n}_\texttt{k} + \texttt{i}_\texttt{k}) * \texttt{w}\\ &\quad + \texttt{base} - ((\ldots(\texttt{low}_1 * \texttt{n}_2) + \texttt{low}_2) * \texttt{n}_3 + \ldots) * \texttt{n}_\texttt{k} + \texttt{low}_\texttt{k}) * \texttt{w}.\end{aligned}$$

Der zur Laufzeit zu berechnende erste Summand der Formel lässt sich wie folgt rekursiv bestimmen:

$$\begin{aligned}\texttt{e}_1 &:= \texttt{i}_1 \text{ und}\\ \texttt{e}_\texttt{j} &:= \texttt{e}_{\texttt{j}-1} * \texttt{n}_\texttt{j} + \texttt{i}_\texttt{j} \text{ für } 1 < \texttt{j} \leq \texttt{k}\end{aligned}$$

und $\texttt{e}_\texttt{k} * \texttt{w}$ ergibt dann den Wert des ersten Summanden.
Wie übersetzt man nun Zugriffe auf Feldelemente?
Als erster Ansatz könnte folgende Grammatik dienen:

```
F  →  id [L]
L  →  L,E | E
```

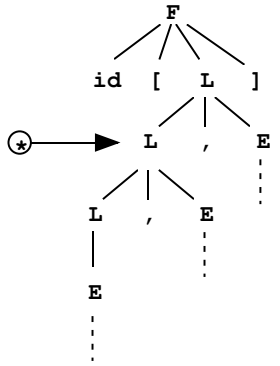
`F` steht dabei für ein Feldelement, `L` steht für eine Liste von Ausdrücken und `E` für einen Ausdruck. `F` und `L` haben je zwei synthetische Attribute:

- `F.place` enthält den Namen einer Variablen, die den Wert des zweiten, hier als zur Übersetzungszeit bekannt angenommenen Summanden der obigen Zugriffsformel enthält,
- `F.offset` enthält den Namen einer Variablen, die den Wert des ersten Summanden der obigen Zugriffsformel enthält.
- `L.ndim` enthält die Anzahl der arithmetischen Ausdrücke in `L`
- `L.place` gibt den Namen der Variablen an, die den Wert der Zugriffsformel für die Ausdrücke in `L`, also den Wert $\texttt{e}_\texttt{j}$ enthält.

Weiterhin soll es eine Funktion `limit(array,j)` geben, die für das Feld `array` den korrespondierenden Wert $\texttt{n}_\texttt{j} = \texttt{high}_\texttt{j} - \texttt{low}_\texttt{j} + 1$ zurückgibt.

Beispiel 5.7:
Zugriff auf `a[1,2,3]`. Wir betrachten den Ableitungsbaum und versuchen, ob eine Auswertung der Attribute in dieser Form möglich ist.

An der mit einem Stern markierten Stelle kann man keine korrekte Drei-Adress-Befehle für den Ausdruck $e_2 = e_1 * n_2 + i_2$ erzeugen. Der Wert von n_2 muss mit der Funktion `limit` aus der Symboltabelle extrahiert werden, aber der Name des Feldes ist an dieser Stelle nicht verfügbar!



Es gibt nun zwei Möglichkeiten, dieses Problem zu umgehen:

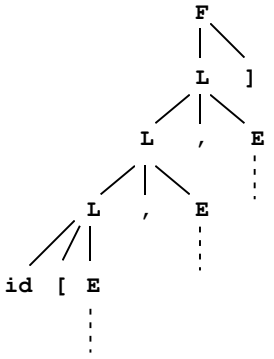
1) Einführung eines inheriten Attributs für `L` und Herunterreichen des Feldnamens im Ableitungsbaum

2) Umformung der Grammatik, um bei einer S-attributierten Grammatik bleiben zu können.
   Eine Möglichkeit wäre:

```
F  →  L ]
L  →  L, E | id [ E
```

Damit erhielte man folgenden Ableitungsbaum für den Ausdruck aus Beispiel 5.7:

`L` benötigt noch ein weiteres synthetische Attribut `L.array`, das den Namen des Feldes bzw. einen Zeiger auf den entsprechenden Eintrag in der Symboltabelle enthält. Damit ist eine syntax-gesteuerte Übersetzung von Feldzugriffen möglich.



Mit dieser Idee kann man das folgende SDTS zur Übersetzung von Feldzugriffen definieren:

Produktion	Semantische Regel
S → id := E	emit (id.lexeme ':=' E.place);
S → F := E	emit (F.place '[' F.offset ']' ':=' E.place);
E → E_1 + E_2	E.place := newtemp(); emit (E.place ':=' E_1.place + E_2.place);
E → (E_1)	E.place := E_1.place;
E → id	E.place := id.lexeme;
E → F	E.place := newtemp(); emit (E.place ':=' F.place '[' F.offset ']');

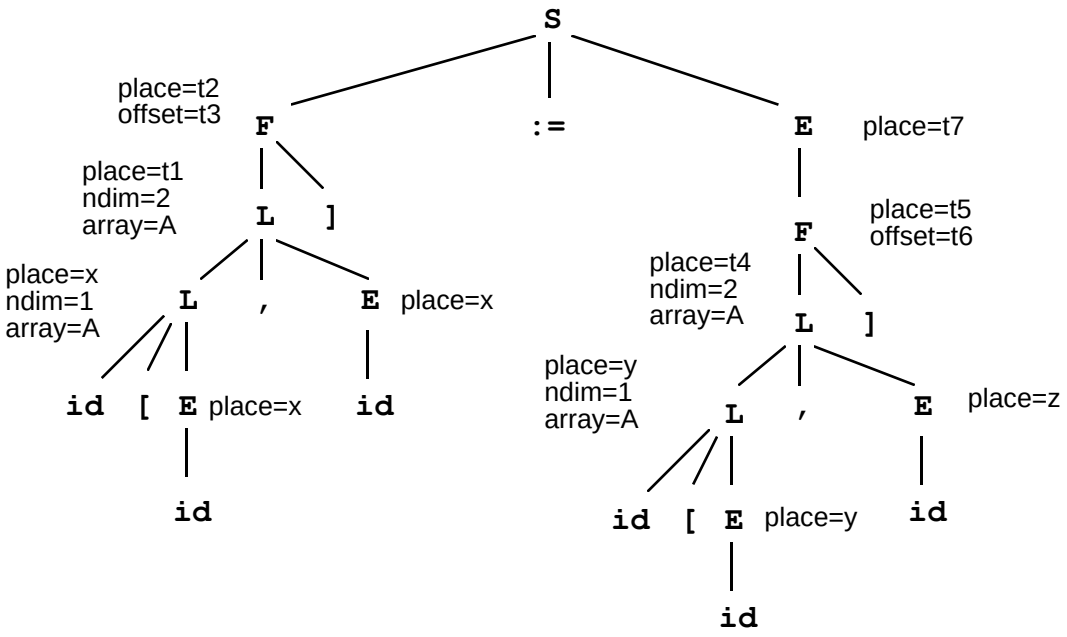
F → L]	F.place := newtemp();
	emit (F.place ':=' L.array '-' σ(L.array);
	F.offset := newtemp();
	emit (F.offset ':=' w '*' L.place);

L → L_1 , E	t := newtemp();
	j := L_1.ndim + 1;
	emit (t ':=' L_1.place '*' limit (L_1.array, j));
	emit (t ':=' t '+' E.place);
	L.array := L_1.array;
	L.place := t;
	L.ndim := j;

L → id [E	L.place := E.place; L.ndim := 1; L.array := id.lexeme;

Bemerkung: Der durch emit(F.place ':=' L.array '-' σ(L.array)) ausgegebene Drei-Adress-Befehl soll den festen Teil der Zugriffsformel darstellen. L.array entspricht dem Wert base und σ(L.array) dem geschachtelten Ausdruck.

Beispiel 5.8:
Sei `A` ein 10×20-Feld, d.h. `A : array[1..10,1..20] of ...`. Es ist also $n_1 = 10$ und $n_2 = 20$. und es gelte `w = 4`. Übersetzt werden soll die Wertzuweisung `A[x,x] := A[y,z]`. Es ergibt sich folgender attributierter Ableitungsbaum:



Die erzeugten Drei-Adress-Befehle wären dann:

```
t1 := x * 20
t1 := t1 + x
t2 := A - 84
t3 := 4 * t1
t4 := y * 20
t4 := t4 + z
t5 := A - 84
t6 := 4 * t4
t7 := t5[t6]
t2[t3] := t7
```

wobei $\sigma(\mathtt{A}) = ((\mathtt{low}_1 * \mathtt{n}_2) + \mathtt{low}_2) * \mathtt{w}$ den Wert 84 ergibt.