Compilerkonstruktion

Wintersemester 2015/16

Prof. Dr. R. Parchmann

17. November 2015

Zwischencode-Erzeugung

Warum verwendet man einen Zwischencode und erzeugt nicht direkt ein Programm in der Zielsprache?

Zwischencode-Erzeugung

Warum verwendet man einen Zwischencode und erzeugt nicht direkt ein Programm in der Zielsprache?

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

Zwischencode-Erzeugung

Warum verwendet man einen Zwischencode und erzeugt nicht direkt ein Programm in der Zielsprache?

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.

Zwischencode-Erzeugung

Warum verwendet man einen Zwischencode und erzeugt nicht direkt ein Programm in der Zielsprache?

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.

Form des Zwischencodes

Man unterscheidet die folgenden Hauptkategorien von Formen des Zwischencodes:

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

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

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.

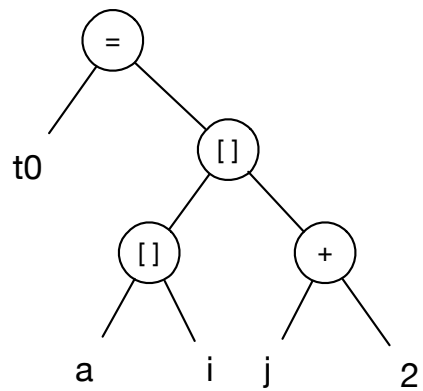
Beispiel

a sei deklariert als `float a[20][10]` mit unterer Feldgrenze 0
Der Ausdruck `a[i][j+2]` würde übersetzt

HIL: in einen Ausdruck wie etwa

```
t0 = a[i,j+2]
```

mit t0 als temporären Namen oder in einen Syntaxbaum wie



MIL:

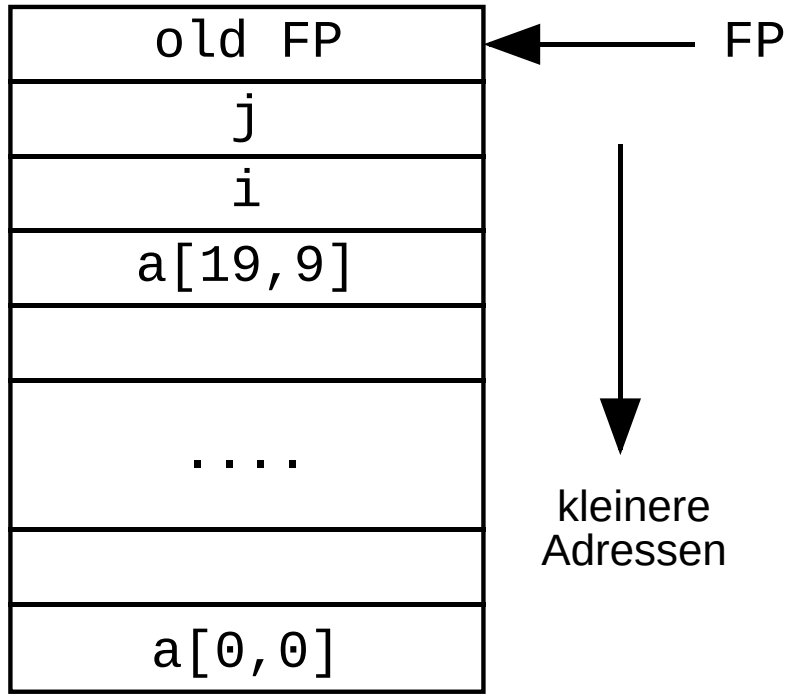
```
t1 = j + 2
t2 = i * 10
t3 = t1 + t2
t4 = 4 * t3      /* Float-Zahl = 4 Bytes */
t0 = a[t4]       /* a[t4] entspricht */
                 /*Anfangsadresse a + t4 */
```

wobei die `ti` temporäre Namen sind.
Es wird also der Ausdruck $(\texttt{addr a}) + 4 * (i * 10 + j + 2)$ berechnet. Über die Speicherung des Feldes a muss nur bekannt sein, dass sie zeilenweise erfolgt und dass jedes Feldelement 4 Bytes belegt.

LIL: Annahmen:

- das Feld `a` ist lokal
- Speicherplatz für `a` und für `i` und `j` ist im Aktivierungs-Record
- der Zugriff auf lokale Größen erfolgt über einen Framepointer `FP`
- die Objekte sind in der Reihenfolge `j`, `i` und dann `a` im Aktivierungs-Record
- jede Integergröße ist 2 Bytes und jedes Feldelement 4 Bytes lang.
- der Laufzeitstack wächst zu kleineren Adressen hin
- Die `ri` sind (fiktive) Integer-Register, die `fi` (fiktive) spezielle Floating-Point Register.

Aufbau des Aktivierungs-Records

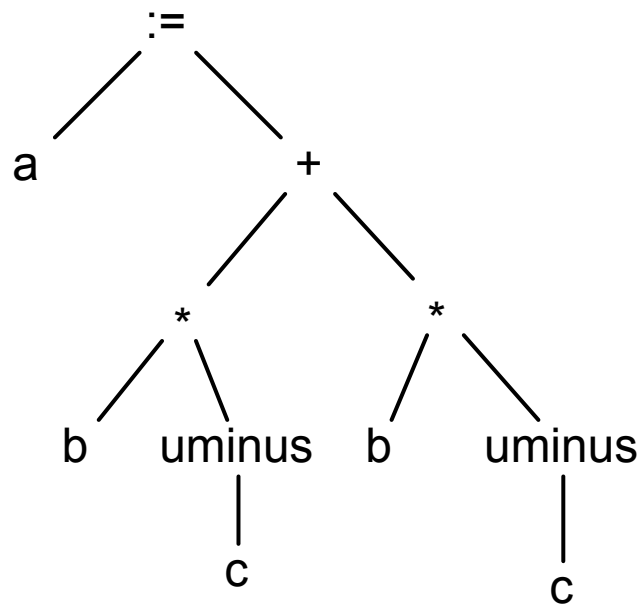


Äquivalentes Programm in einer LIL-Zwischensprache:

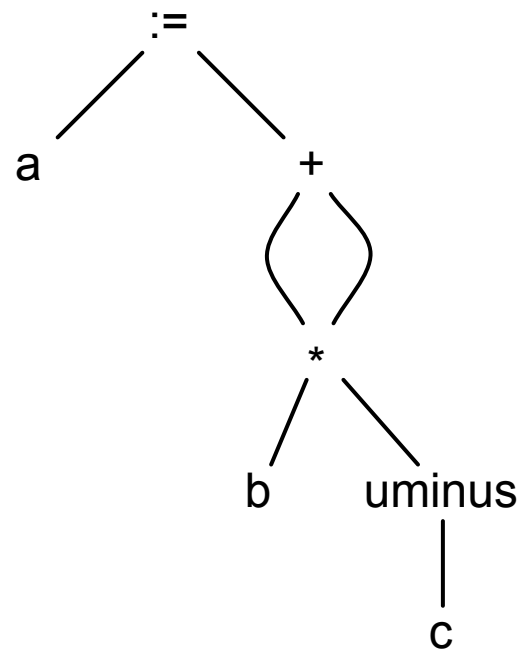
```
r1 <- [FP-2]      /* r1 enthält j */
r2 <- r1 + 2
r3 <- [FP-4]      /* r3 enthält i */
r4 <- r3 * 10
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- FP - 804    /* Startadresse */
                  /* des Feldes a */
f1 <- [r7+r6]
```

Übersetzung von Wertzuweisungen in Syntaxbäume bzw. Syntaxgraphen

Gegeben sei der Ausdruck `a := b*-c+b*-c`, dann wäre der zugehörige Syntaxbaum bzw Syntaxgraph:



Syntaxbaum



Syntaxgraph

Gegeben seien zwei Funktionen:

Gegeben seien zwei Funktionen:

- `mkleaf(id, id.place)` erzeugt ein Blatt mit Markierung `id` und einem Zeiger in die Symboltabelle auf den korrespondierenden Eintrag.

Gegeben seien zwei Funktionen:

- `mkleaf(id, id.place)` 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`.

Gegeben seien zwei Funktionen:

- `mkleaf(id, id.place)` 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.

Alle Funktionen geben einen Zeiger auf den erzeugten Knoten zurück.

Attributierte Grammatik für die Übersetzung

nptr sei ein synthetisches Attribut eines jeden nichtterminalen Symbols, das auf den Wurzelknoten des zugehörigen Syntaxbaums verweist.

Attributierte Grammatik für die Übersetzung

nptr sei ein synthetisches Attribut eines jeden nichtterminalen Symbols, das auf den Wurzelknoten des zugehörigen Syntaxbaums verweist.

Ein SDTS für die Übersetzung wäre:

Produktion	*Semantische Regel*
S → id := E	S.nptr := mknode('assign', mkleaf(id, id.entry), E.nptr);
E → E_1 + E_2	E.nptr := mknode('+', E_1.nptr, E_2.nptr);
E → 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);

Drei-Adress-Befehle

Der Drei-Adress-Code ist eine MIL-Zwischensprache.

Drei-Adress-Befehle:

- Wertzuweisungen der Form: `x := y op z`,
- Wertzuweisungen der Form: `x := op y`,
- 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`
- 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`,

Zwei Arten von Sprungzielen: explizite Marken oder Indizes

Beispiel

Für die Anweisung `do i = i+1; while (a[i] < v);` sind die folgenden Übersetzungen in Drei-Adress-Code möglich:

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

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

Übersetzung von Wertzuweisungen mit einfachen arithmetischen Ausdrücken in Drei-Adress-Code.

Der Variablen `E`, die für einen arithmetischen Ausdruck steht, sind zwei Attribute zugeordnet:

1. `E.place`: Namen einer 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
- `gen( ... )` erzeugt einen Drei-Adress-Befehl.

Produktion	*Semantische Regel*
S → id := E	S.code := E.code || gen (id.lexeme ':=' E.place);
E → 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 → 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 → - E_1	E.place := newtemp(); E.code := E_1.code || gen (E.place ':=' 'uminus' E_1.place);
E → (E_1)	E.place := E_1.place; E.code := E_1.code;
E → id	E.place := id.lexeme; E.code := '';

Bemerkung

Man nennt Regeln, in denen die Zeichenkettenattribute der einzelnen nichtterminalen Symbole in der Reihenfolge ihres Auftretens in der rechten Seite der Produktion mit eventuell weiteren konstanten Zeichenketten konkateniert werden einfach.
Man betrachte die Produktion $A \to BC$ mit zugeordneter Regel

`A.code := string`$_B$ `|| B.code || string`$_C$ `|| C.code || string`$_D$

Diese Regel kann ersetzt werden durch

`A` $\to$ {`emit( string`$_B$`)`} `B` {`emit(string`$_C$`)`} `C` {`emit(string`$_D$`)`}

wobei `emit( .... )` *die als Parameter übergebene Zeichenkette ausgibt.*
Die Funktion `emit` *ist also eine Funktion mit Seiteneffekten, d.h. man muss auf die Reihenfolge der Auswertung achten!*

Damit lässt sich das SDTS umschreiben zu:

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;

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:

Idee: 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 oder aber auf einen Drei-Adress-Befehl sind.

Beispiel

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

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)` ein Zeiger auf den Drei-Adress-Befehl darstellt, mit dem der entsprechende Operand berechnet wurde.

Syntax-gesteuerte Übersetzung von Deklarationen

SDTS zur Berechnung des Typs und des Offsets im Aktivierungs-Record für lokale Variablen.

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 → ↑T_1	T.type := pointer (T_1.type); T.width := 4;

Bemerkung

`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 immer 0 ist.
- `offset` ist eine globale Größe.

Übersetzung von Feldzugriffen

Annahme: die Feldelemente werden fortlaufend abgespeichert und jedes Element hat eine feste Größe von `w` Bytes.

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`.

Bemerkung

Durch Umformung erhält man

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

wobei der zweite Summand $\texttt{base} - \texttt{low} * \texttt{w}$ *eventuell bereits zur Übersetzungszeit bestimmt werden kann.*

Zweidimensionale Felder

Die Deklaration des Feldes sei

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

Es gibt zwei prinzipielle Möglichkeiten:

- zeilenweise Speicherung (row-major) oder
- spaltenweise Speicherung (column-major)

Annahme: zeilenweise Speicherung.

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

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

wobei $\mathtt{n}_2$ `=` $\mathtt{high}_2$ `-` $\mathtt{low}_2$ `+ 1` (Anzahl der Spalten) ist.

Bemerkung

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

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

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

Mehrdimensionale Felder

Die Deklaration des Feldes sei

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

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.
Die Adresse des Feldelements $\texttt{a[i}_1\texttt{,i}_2\texttt{,...,i}_k\texttt{]}$ ist dann

$$\texttt{base} + ((\ldots(i_1 - \texttt{low}_1) * n_2 + i_2 - \texttt{low}_2) * n_3 + \ldots + i_k - \texttt{low}_k) * w$$

wobei $\texttt{n}_\texttt{i} = \texttt{high}_\texttt{i} - \texttt{low}_\texttt{i} + 1$ für $1 < \texttt{i} \leq \texttt{k}$ gilt.

Bemerkung

Durch Umformung erhält man einen Summanden, der zur Laufzeit berechnet werden muss und einen, der eventuell bereits zur Übersetzungszeit bekannt ist:

$$\begin{aligned}
&\mathtt{base} + ((\ldots(i_1 - \mathtt{low}_1) * n_2 + i_2 - \mathtt{low}_2) * n_3 \\
&\qquad + \ldots + i_k - \mathtt{low}_k) * w = \\
&((\ldots(\mathtt{i}_1 * \mathtt{n}_2) + \mathtt{i}_2) * \mathtt{n}_3 + \ldots) * \mathtt{n}_\mathtt{k} + \mathtt{i}_\mathtt{k}) * \mathtt{w} \\
&\qquad + \mathtt{base} - ((\ldots(\mathtt{low}_1 * \mathtt{n}_2) + \mathtt{low}_2) * \mathtt{n}_3 + \ldots) * \mathtt{n}_\mathtt{k} + \mathtt{low}_\mathtt{k}) * \mathtt{w}.
\end{aligned}$$

Bemerkung

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

$$\begin{aligned} \mathtt{e_1} &:= \mathtt{i_1}\ \mathit{und} \\ \mathtt{e_j} &:= \mathtt{e_{j-1}} * \mathtt{n_j} + \mathtt{i_j}\ \mathit{für}\ \mathtt{1} < \mathtt{j} \leq \mathtt{k} \end{aligned}$$

$\mathtt{e_k} * \mathtt{w}$ *ergibt dann den Wert des ersten Summanden.*

Übersetzung von Feldzugriffen

Als erster Ansatz könnte folgende Grammatik dienen:

$$\begin{aligned} \texttt{F} &\rightarrow \texttt{id [L]} \\ \texttt{L} &\rightarrow \texttt{L, E | E} \end{aligned}$$

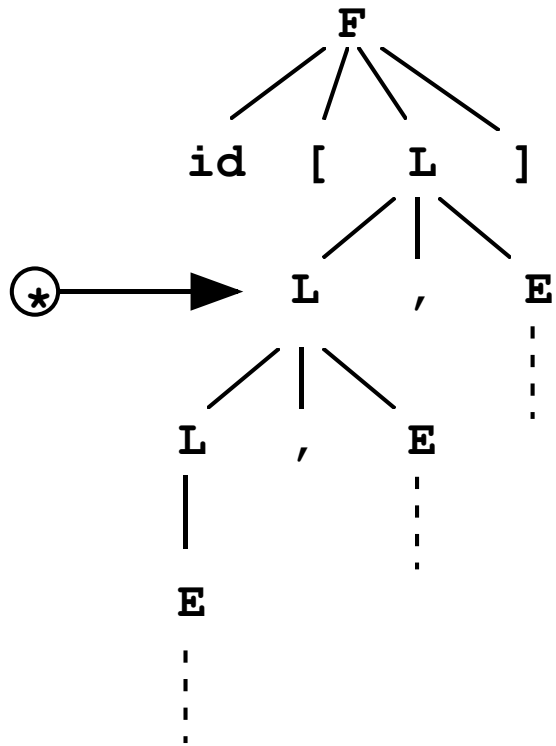
`F` steht dabei für ein Feldelement, `L` steht für eine Liste von Ausdrücken und E für einen arithmetischen Ausdruck..

`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 e_j enthält, wobei `j = L.ndim` ist.
- `limit(array,j)` gibt für das Feld `array` den Wert $n_j = high_j - low_j + 1$ zurück.

Beispiel

Der Ableitungsbaum für den Zugriff auf `a[1,2,3]`. Probleme ergeben sich beim Auswerten der Attribute an der markierten Stelle.



Es gibt zwei Möglichkeiten, dieses Problem zu umgehen:

Es gibt zwei Möglichkeiten, dieses Problem zu umgehen:

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

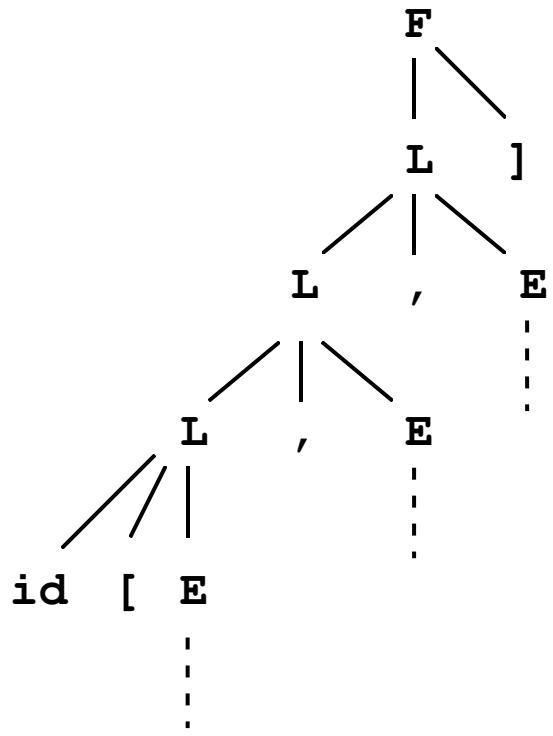
Es gibt zwei Möglichkeiten, dieses Problem zu umgehen:

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:

$$\texttt{F} \rightarrow \texttt{L]}$$

$$\texttt{L} \rightarrow \texttt{L, E} \mid \texttt{id [E}$$

Damit erhielte man folgenden Ableitungsbaum für das Beispiel:



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.

SDTS zur Übersetzung von Feldzugriffen

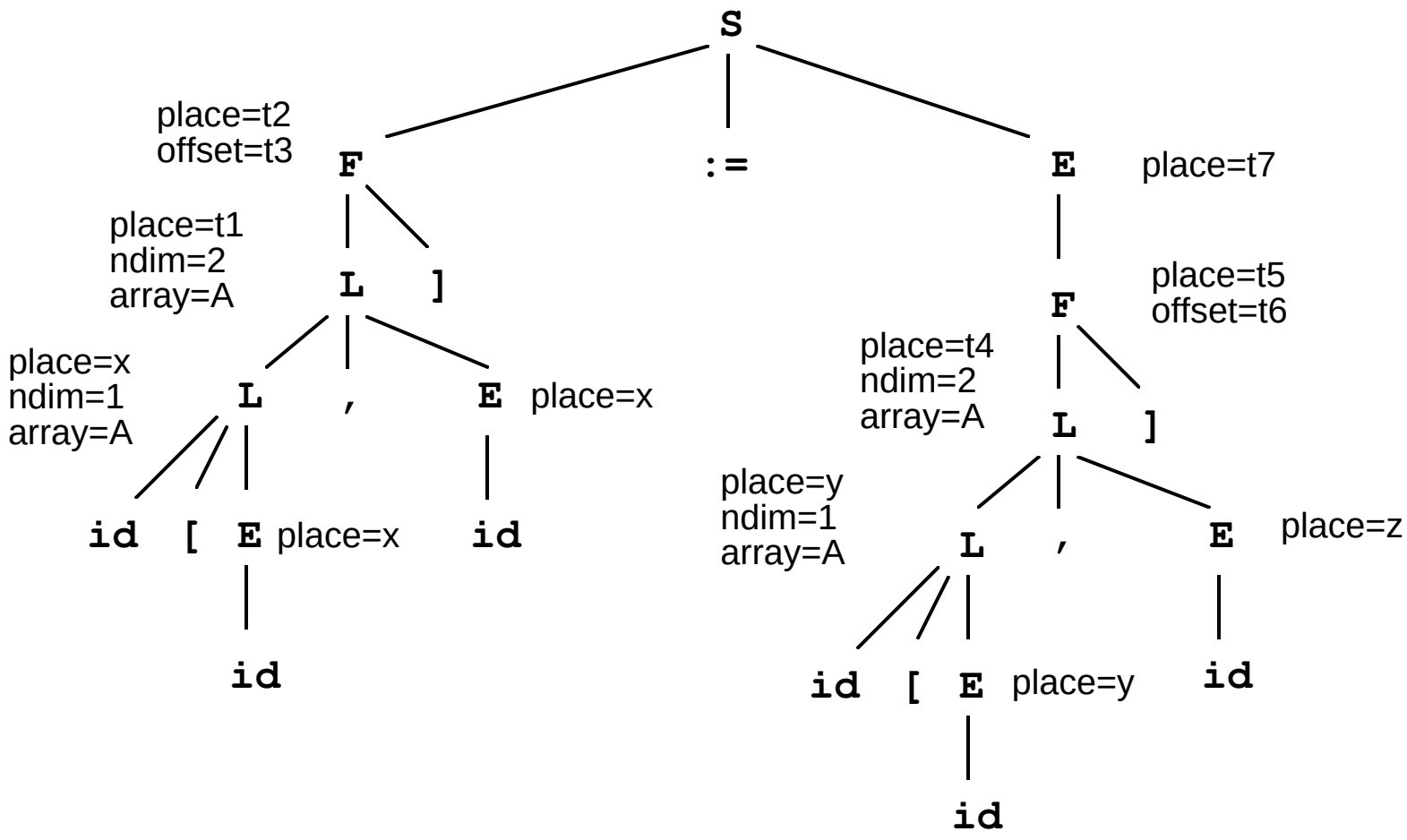
Produktion	*Semantische Regel*
S → id := E	emit (id.lexeme ':=' E.place);
S → F := E	emit (F.place '[' F.offset ']' ':=' E.place);
E → E_1 + E_2	E.place := newtemp();
	emit (E.place ':=' E_1.place + E_2.place);
E → (E_1)	E.place := E_1.place;
E → id	E.place := id.lexeme;
E → F	E.place := newtemp();
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

Produktion	*Semantische Regel*
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;

Beispiel

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:

Ableitungsbaum für das Beispiel:



Die erzeugten Drei-Adress-Befehle:

```
t1 := x * 20
t1 := t1 + x
t2 := A - 84
t3 := 4 * t1
t4 := y * 20
t4 := t4 + z
t5 := A - σ(A)
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.