Konfigurationen für ein Wort

Mehrdeutige Grammatiken beim Bottom-Up Parsing

Wird in einen Parser-Generator eine Grammatik eingegeben, die kein deterministisches Bottom-Up Parsing ermöglicht, so erhält man Fehlermeldungen.

Dabei werden Situationen des Parsers beschrieben,

- in der sowohl eine shift- als auch eine reduce-Aktion durchgeführt werden könnte (shift/reduce-Konflikt).
- in der eine reduce-Aktion für zwei verschiedene Produktionen ausgeführt werden könnte (reduce/reduce-Konflikt).

Mögliche Ursachen für mehrdeutige Grammatiken

Einen shift/reduce-Konflikt erhält man, wenn die rechte Seite einer Produktion auch *Präfix* einer anderen Produktion mit gleichen Symbol auf der linken Seite ist:

$S \rightarrow \textbf{if } E \textbf{ then } S \textbf{ else } S \mid \textbf{if } E \textbf{ then } S$

Einen reduce/reduce-Konflikt erhält man, wenn die rechte Seite einer Produktion auch *Suffix* einer anderen Produktion mit gleichem Symbol auf der linken Seite ist:

$E \rightarrow E + T \mid T$

Mehrdeutige Grammatiken beim Bottom-Up Parsing

Beispiel

Z.B. tritt bei folgenden Produktionen ein shift/reduce-Konflikt auf:

```
<stmt> → if <expr> then <stmt>
       | if <expr> then <stmt> else <stmt>
       | ...
```

Mit der shift-Operation erhält man die „übliche“ Zusammenfassung von if-then-else-Schachtelungen:

```
if <expr> then  if <expr> then <stmt> else <stmt>
```

Besser ist natürlich die Vermeidung der Mehrdeutigkeit:

```
<stmt> → if <expr> then <stmt> fi
       | if <expr> then <stmt> else <stmt> fi
```

Auflösung von Mehrdeutigkeiten

Viele Parser-Generatoren lösen diese Mehrdeutigkeiten mit zwei zusätzlichen Regeln, die man auch bei der Konstruktion einer Grammatik ausnutzen kann:

- bei shift/reduce-Konflikten wählt man die shift-Operation.
- bei reduce/reduce-Konflikten reduziert man mit der zuerst in der Grammatik auftretenden Produktion.

`Yacc` und `Bison` können mehrdeutige Grammatiken (z.B. zur Beschreibung arithmetischer Ausdrücke) sogar noch cleverer verarbeiten.

Dazu teilt man dem Parser-Generator die **Priorität** und die **Assoziativität** der verwendeten Operatoren mit.

Priorität und Assoziativität von Operatoren

Dann löst der Generator die auftretenden shift/reduce-Konflikte so:

Ist die Priorität der Operation auf dem Keller

- größer als die Priorität der Operation in der Vorschau, so soll zuerst die Operation auf dem Keller ausgeführt werden, dazu wird reduziert;
- kleiner als die Priorität der Operation in der Vorschau, so soll zuerst die Operation der Vorschau ausgeführt werden, dazu muss der zweite Operand eingelesen werden, also ein shift durchgeführt werden.

Bei gleichen Prioritäten entscheidet die Assoziativität:
Sind beide Operationen

- linksassoziativ, so wird zuerst die Operation auf dem Keller ausgeführt, also reduziert;
- rechtsassoziativ, so wird zuerst die Operation in der Vorschau ausgeführt, also geshiftet.

Beispiel für die Auflösung von Mehrdeutigkeiten

Beispiel

Betrachte die Grammatik $G_5 = (\{E\}, \{a, +, *, (,)\}, P, E)$ mit den Produktionen

1. $E \to E + E$
2. $E \to E * E$
3. $E \to (E)$
4. $E \to a$

Diese Grammatik ist mehrdeutig!

Die zugehörige Syntaxanalyse-Tabelle enthält Konflikte:

	aktion						sprung
Zustand	*a*	+	∗	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		s4/r1	s5/r1		r1	r1	
8		s4/r2	s5/r2		r2	r2	
9		r3	r3		r3	r3	

Gibt man $'*'$ eine höhere Priorität als $'+'$ und legt fest, dass beide Operationen linksassoziativ sind, so werden die shift/reduce-Konflikte
im Zustand 7 durch r1 und s5, die
im Zustand 8 durch r2 und r2 gelöst.

Alternativ kann man versuchen, eine Grammatik mit Hilfe der Informationen über Prioritäten und Assoziativitäten eindeutig zu machen.

Folgende Prioritäten werden vergeben, um die Grammatik G_5 eindeutig zu machen.
Außerdem seien die Operatoren $*$ und $+$ *linksassoziativ*.

Prioritäten:

1. $E + E$
2. $E * E$
3. $a \mid (E)$

Jede Prioritätsstufe kennzeichnen wir durch ein eigenes Nichtterminal und reichen die Nichtterminale durch die Prioritätsstufen (von der niedrigsten zur höchsten) durch.

Die *Links*assoziativität bilden wir ab, indem der *linke* Operand einer binären Operation auf der gleichen Prioritätsstufe bleibt.
Bei einer rechtsassoziativen binären Operation bleibt der rechte Operand auf der gleichen Prioritätsstufe.

Beispiel

mehrdeutige Grammatik:

1. $E \to E + E$
2. $E \to E * E$
3. $E \to a \mid (E)$

eindeutige Grammatik:

1. $E_1 \to E_2 \mid E_1 + E_2$
2. $E_2 \to E_3 \mid E_2 * E_3$
3. $E_3 \to a \mid (E_1)$

Vermeidung von Sackgassen

Natürlich soll ein Bottom-up-Parser **deterministisch** arbeiten, also auch Sackgassen vermeiden.
Um Sackgassen zu vermeiden, nützt leider weder eine verlängerte Vorschau noch ein tieferer, aber (auf k Zeichen) begrenzter Blick in den Keller:

Beispiel

Die eindeutige kontextfreie Grammatik G_{ab} mit den Produktionen

$$S \to aA \mid bB, \qquad A \to d \mid cAc, \qquad B \to d \mid cBc.$$

erzeugt die Sprache $L(G_{ab}) = \{a, b\}\{c^n d c^n \mid n \geq 0\}$.

Für den oberen Kellerinhalt $c^k d$ und die Vorschau c^k (k beliebig!) sind beide Produktionen $A \to d$ und $B \to d$ anwendbar, aber eine Produktion führt in eine Sackgasse!

Wir brauchen Information über den ganzen Kellerinhalt!

erfolgreiche Kellerinhalte

1. Idee: Wir bestimmen alle **erfolgreichen Kellerinhalte** des Parsers, also die, die während einer akzeptierenden Berechnung entstehen können.

Beispiel (G_{ab}: $S \to aA \mid bB, \quad A \to d \mid cAc, \quad B \to d \mid cBc$)

Erfolgreiche Kellerinhalte (ohne \$) des Parsers der Grammatik G_{ab} sind:

$$\{\varepsilon\} \cup \{ac^n\} \cup \{ac^nd\} \cup \{ac^nA\} \cup \{ac^{n+1}Ac\}$$
$$\cup \{bc^n\} \cup \{bc^nd\} \cup \{bc^nB\} \cup \{bc^{n+1}Bc\} \cup \{S\}, \qquad n \geq 0$$

Auf jeden erfolgreichen Kellerinhalt sind Parseroperationen anwendbar. Nur einige davon liefern einen erfolgreichen Kellerinhalt als Ergebnis!

Beispiel

Die erfolgreichen Kellerinhalte ac^nd und bc^nd können mit beiden Produktionen $A \to d$ bzw. $B \to d$ reduziert werden, aber von den Ergebnissen ac^nA, bc^nB, ac^nB, bc^nA sind nur die ersten beiden erfolgreiche Kellerinhalte.

gültige Parseroperationen

2. Idee: Wir erlauben nur noch **gültige Parseroperationen**, also solche, die wieder zu einem erfolgreichen Kellerinhalt führen.

Die Menge der erfolgreichen Kellerinhalte ist i.Allg. unendlich, so dass es unmöglich scheint festzustellen, welche Operationen für welchen erfolgreichen Kellerinhalt gültig sind.

Glücklicherweise können wir die Menge der erfolgreichen Kellerinhalte stets in endlich viele Äquivalenzklassen aufteilen.

Äquivalenzklassen erfolgreicher Kellerinhalte

Beispiel (Äquivalenzklassen zu G_{ab}, $n \geq 0$)

Äquivalenzklasse	gültige Operationen
$\{\varepsilon\}$	shift a, shift b
$\{ac^n\} \cup \{bc^n\}$	shift d, shift c
$\{ac^n d\}$	reduce $A \to d$
$\{bc^n d\}$	reduce $B \to d$
$\{aA\}$	reduce $S \to aA$
$\{bB\}$	reduce $S \to bB$
$\{S\}$	accept
$\{ac^{n+1}A\} \cup \{bc^{n+1}B\}$	shift c
$\{ac^{n+1}Ac\}$	reduce $A \to cAc$
$\{bc^{n+1}Bc\}$	reduce $B \to cBc$

Zwei erfolgreiche Kellerinhalte gehören zur gleichen Äquivalenzklasse, wenn sie dieselbe Menge gültiger Operationen haben.

Anzahl der Äquivalenzklassen

Eine kontextfreie Grammatik $G = (N, T, P, S)$ hat

- $|T|$ shift-Operationen und
- $|P|$ reduce-Operationen,

also höchstens $2^{|T|+|P|}$ Äquivalenzklassen.

Vermeidung von Sackgassen

Etwas müssen wir die Äquivalenzrelation noch verfeinern:

- Die Äquivalenzrelation soll **rechtsinvariant** sein:
  Werden äquivalente, erfolgreiche Kellerinhalte δ_1 und δ_2 mit einem Symbol X zu erfolgreichen Kellerinhalten verlängert, so sollen sie äquivalent bleiben:
  $[\delta_1] = [\delta_2] \Rightarrow [\delta_1 X] = [\delta_2 X]$ (bzw. $[\delta_1 X] \neq [\delta_2 X] \Rightarrow [\delta_1] \neq [\delta_2]$).
- Bei einer Reduktion soll die anzuwendende Produktion eindeutig sein und nicht nur die zugehörigen Äquivalenzklassen.
  D.h. zu einer Produktion $A \to X_1 \dots X_n$ soll es keine Produktion $A' \to X'_1 \dots X'_n$ geben mit
  $[\delta A] = [\delta A']$, $[\delta X_1] = [\delta X'_1]$, $\dots$, $[\delta X_1 \dots X_n] = [\delta X'_1 \dots X'_n]$.
  Dies gilt sicher, wenn zwei äquivalente Kellerinhalte stets mit dem gleichen Symbol enden: $[\delta X] = [\delta X'] \Rightarrow X = X'$.

Vermeidung von Sackgassen

Beispiel (Grammatik G_{ab})

Um diese Anforderungen zu erfüllen, teilen wir Äquivalenzklassen auf:

- die Klasse $[ac^* \mid bc^*]$ in die Klassen $[a]$, $[ac^+]$, $[b]$ und $[bc^+]$.
- die Klasse $[ac^+A \mid bc^+B]$ in die Klassen $[ac^+A]$ und $[bc^+B]$.

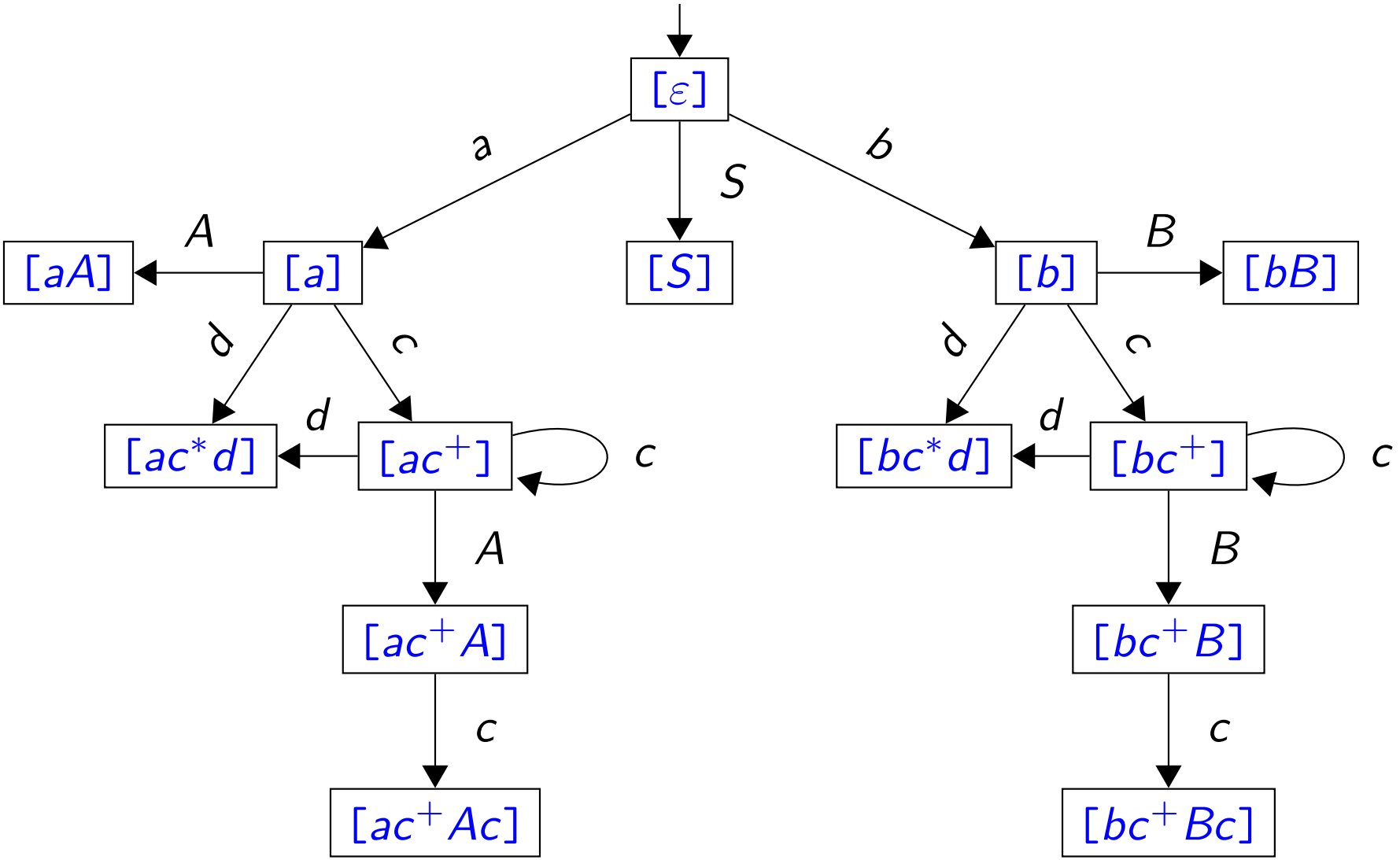
Vermeidung von Sackgassen

Diese Äquivalenzklassen fasst man als Zustände eines endlichen Automaten auf, die durch die Grammatiksymbole in andere Äquivalenzklassen übergehen.

Der Anfangszustand ist $[\varepsilon]$,
der einzige Endzustand eine (beliebige) Klasse $[\gamma]$.
Die akzeptierte (reguläre!) Sprache ist dann gerade $[\gamma]$.

Der (Fehler-)Zustand für die nicht-erfolgreichen Kellerinhalte und die Übergänge dorthin sind der Übersichtlichkeit halber weggelassen.

Vermeidung von Sackgassen



Vermeidung von Sackgassen

Wir wollen diese Idee, nämlich die Beschränkung auf *erfolgreiche* Kellerinhalte und *gültige* Parseroperationen sowie die Zerlegung der Menge der erfolgreichen Kellerinhalte in Äquivalenzklassen, von Kellerautomaten auf kontextfreie Grammatiken übertragen.

Dazu müssen wir die shift-Operation des Kellerautomaten in den Produktionen sichtbar machen, indem wir verschiedene Stadien der Abarbeitung einer Produktion unterscheiden.

Wir führen dazu erfolgreiche Präfixe von Rechtsableitungen, gültige Stadien und eine Äquivalenzrelation auf der Menge der erfolgreichen Präfixe „hat gleiche Menge von gültigen Stadien“ ein.

Die Äquivalenzklassen der erfolgreichen Präfixe bilden dann die Zustände eines endlichen Automaten.

Es ist üblich, einen Zustand statt durch eine (reguläre) Menge von erfolgreichen Präfixen durch die Menge der für diese Präfixe gültigen Stadien zu beschreiben.

Teil VI

Semantische Analyse

Semantische Analyse

Ein Objekt einer Programmiersprache besitzt mehrere Eigenschaften:

1. Eine **Speicheradresse**, ab der dieses Objekt im Speicher abgelegt ist.
2. Eine **Codierung** des Objektes selbst, d.h. das **Bitmuster** der zugeordneten Speichereinheiten, das den **Wert** des Objektes darstellt.

Um mit dem Wert eines Objektes arbeiten zu können, muss man wissen, wie lang diese Codierung ist und wie man sie zu interpretieren hat.

Diese Information wird durch den **Typ des Objekts** festgelegt, der häufig, z.B. bei der Variablendeklaration, angegeben wird.

Typ-Prüfung

Ein Compiler kann oft prüfen, ob die durch die Sprachdefinition der Programmiersprache gegebenen Typ-Beschränkungen im vorgelegten Programm eingehalten werden.

- Prüfung während der Übersetzung (**statische Typ-Prüfung**).
- Prüfung zur Laufzeit (**dynamische Typ-Prüfung**)

Eine Programmiersprache mit **starker Typ-Prüfung** hat ein Typ-System, bei dem der Compiler im gewissen Umfang garantieren kann, dass beim übersetzten Programm kein Typ-Fehler zur Laufzeit auftritt.

- In vielen Programmiersprachen wird jeder Variablen durch die Deklaration ein Typ zugeordnet.
  Dann kann der Compiler auch Ausdrücken einen Typ zuordnen.
- Es gibt aber auch Programmiersprachen, in denen keine Typ-Deklaration vorgesehen ist. In diesen Fällen ist jedem Datenobjekt eine Kodierung des Typs (bzw. der Klasse) zugeordnet. Diese Kodierung muss gespeichert werden (**tag-System**) und wird immer überprüft, wenn das Objekt als Operand einer Operation auftritt.
- Den Prozess des Ableitens von Typen für Konstrukte der Programmiersprache aus verfügbaren Typ-Informationen nennt man **Typ-Inferenz**.

Typ-Ausdrücke zur Darstellung von Typen

Jedem Objekt und jedem Ausdruck im Programm wird vom Compiler während der semantischen Analyse ein Typ, dargestellt durch einen Typ-Ausdruck, zugeordnet.

Definition von Typ-Ausdrücken

1. Jeder Basistyp ist ein Typ-Ausdruck.
   Beispiele: *int*, *real*, *char*, *bool*, ...
2. Sind Namen als Abkürzungen für Typen erlaubt, dann sind diese Namen ebenfalls Typ-Ausdrücke.
   Beispiel: `type zeichen = char` liefert Typ-Ausdruck *zeichen*
3. Ist T ein Typ-Ausdruck, so ist *array(T)* ebenfalls ein Typ-Ausdruck, der einen Feld-Typ beschreibt.
   Beispiel: `A: array[1..10] of integer` liefert Typ-Ausdruck *array(int)* oder *array(10, int)*.

4. Ist T ein Typ-Ausdruck, so ist *list(T)* ebenfalls ein Typ-Ausdruck, der einen Listen-Typ beschreibt.
5. Sind $T_1, \ldots, T_r$ Typ-Ausdrücke mit $r \geq 1$, dann ist
$$T_1 \times \ldots \times T_r \text{ ein Typ-Ausdruck,}$$
der ein Tupel mit r Komponenten vom Typ $T_1, \ldots, T_r$ beschreibt.
6. Ist T ein Typ-Ausdruck, dann ist *pointer(T)* ein Typ-Ausdruck, der einen Zeigertyp beschreibt.
Beispiel: `var p: ↑integer` ordnet p den Typ-Ausdruck *pointer(int)* zu.

7. Sind T, T_1, ..., T_r Typ-Ausdrücke, dann ist
$$T_1 \times \ldots \times T_r \rightarrow T$$
ein Typ-Ausdruck, der eine Funktion mit $r \geq 1$ Parametern vom Typ T_1, ..., T_r beschreibt, die ein Objekt vom Typ T zurückgibt.
Beispiel: `function f(a,b: char): ↑integer`
ordnet dem Namen `f` den Typ-Ausdruck
$\mathit{char} \times \mathit{char} \rightarrow \mathit{pointer(int)}$ zu.

Bemerkung

Der Operator $\times$ ist linksassoziativ und hat eine größere Priorität als der rechtsassoziative Operator $\rightarrow$.

Es gibt leider viele Konstruktionen in höheren Programmiersprachen, die implementationsabhängig sind.

In Pascal betrachte man z.B.

```
var y: 1..20;
    z: 10..50;
    a: array[1..10] of integer;
    b: array[0..9] of integer;
```

Soll der Ausdruck `y+z` erlaubt werden?
Und welchen Typ hat dieser Ausdruck?

Ist eine Wertzuweisung `a := b` bei gleich großen Feldern erlaubt?

Äquivalenz von Typ-Ausdrücken

Die Typ-Prüfung testet, ob der Typ-Ausdruck *typ1* eines aktuellen Operanden mit dem Typ-Ausdruck *typ2* eines vorgesehenen Operanden einer Operation zusammenpasst, also die Typen *typ1* und *typ2* **äquivalent** sind.

Solange keine Namen als Abkürzungen für Typ-Ausdrücke auftreten, ist die Sache relativ einfach:

Zwei Typ-Ausdrücke sind äquivalent, wenn sie identisch sind.

Der Typ-Prüfer muss also nur beide Typ-Ausdrücke bzgl. ihres Aufbaus miteinander vergleichen. Dies geschieht am besten rekursiv.

Namen in Typ-Ausdrücken

Namen in Typ-Ausdrücken verursachen Probleme!

Sind die Pascal-Variablen `next` und `last` vom gleichen Typ?

```
type link = ↑cell;
var  next : link;
     last : ↑cell;
```

Namensäquivalenz: (inzwischen Standard)
Jeder Typ-Name legt einen neuen Typ fest.

Strukturäquivalenz: (früher sehr gebräuchlich)
Jeder Typ-Name ist nur eine Abkürzung für den definierten Typ-Ausdruck.

Typ-Umwandlungen

Speziell für Basistypen haben viele Programmiersprachen Umwandlungsregeln, die eine automatische Anpassung der Typen von Operanden an die zulässigen Typen eines Operators ermöglichen.

Beispiel

`y := x + i;` mit `x,y` *real* und `i` *int*

Da es für diese Additionsoperation i.Allg. keinen Operator gibt, muss der Compiler oder der Programmierer etwas tun:

- Der Compiler führt eine automatische Typ-Anpassung (*type coercion*) durch.
  Dies ist meist nur erlaubt, wenn damit keine Genauigkeitsverluste verbunden sind.
- Der Compiler signalisiert eine Fehlersituation.

  Dann kann der Programmierer eine explizite Typ-Umwandlung vornehmen:
  - durch Aufruf einer Bibliotheksfunktion, etwa `x + real(i)` oder
  - durch „casting“, z.B. `(char) 20`.
    Dadurch wird der Typ-Ausdruck geändert, aber nicht notwendig die interne Darstellung des Wertes.

Zuordnung des richtigen Operators bei überlagerten Funktionen

Verschiedene Operatoren haben eine identische lexikale Darstellung, z.B. kann + eine Ganzzahl-Addition, eine Gleitpunkt-Addition oder eine Zeichenketten-Konkatenation repräsentieren.

\+ ist daher ein **überlagerter Operator**.

Aufgabe des Compilers ist es, aus dem Kontext den „richtigen“ Operator zu identifizieren.

Überlagerte Operatoren

Das Problem verschärft sich bei Programmiersprachen, die dem Programmierer mehrere Definitionen einer Funktion mit unterschiedlichen Parameter- oder Ergebnistypen erlauben.

In Ada sind z.B. folgende Definitionen gleichzeitig erlaubt:

```
function '*' (i, j: integer) return complex;
function '*' (x, y: complex) return complex;
```

Damit hat eine Funktion (genauer ein Funktionsname) eine Vielzahl von Typ-Ausdrücken und auch arithmetische Ausdrücke haben nicht notwendigerweise nur einen Typ!

Beispiel (Überlagerte Operatoren)

Nehmen wir an, dass der Operator $*$ die folgenden Typ-Ausdrücke hat:

- $int \times int \rightarrow int$
- $int \times int \rightarrow complex$
- $complex \times complex \rightarrow complex$

Haben dann die Literale 2, 3 und 5 den einzig möglichen Typ `integer`
und ist z eine Variable vom Typ `complex`,
dann kann $3 * 5$ den Typ `integer` oder `complex` haben.

Im Ausdruck $(3 * 5) * 2$ hat $3 * 5$ zwingend den Typ `integer`,
im Ausdruck $(3 * 5) * z$ hat $3 * 5$ zwingend den Typ `complex`.

Polymorphe Funktionen

Weitere Probleme treten auf, wenn die Programmiersprache „polymorphe Funktionen“ oder „polymorphe Operatoren“ zulässt. Dies sind Funktionen, bei denen der Typ der Parameter nicht eindeutig festgelegt werden muss.

Beispiel

Eine Funktion zum Bestimmen der Länge einer Liste (in Haskell):

```
length(ls) = if null(ls) then 0
             else length(tail(ls)) + 1
```

Typ-Variable

Um Typ-Ausdrücke für polymorphe Funktionen angeben zu können, braucht man **Typ-Variablen**. Eine Typ-Variable steht dabei für einen beliebigen Typ.

Im obigen Beispiel könnte man der Funktion `length` den Typ-Ausdruck $\mathit{list}(\alpha) \rightarrow \mathit{int}$ zuordnen, wobei α die Typ-Variable ist.

Typ-Prüfung mit polymorphen Funktionen wird signifikant schwieriger:

- Verschiedene Vorkommen einer polymorphen Funktion in einem Ausdruck können Argumente unterschiedlichen Typs haben.
- Typ-Äquivalenz für Typ-Ausdrücke mit Variablen ist neu zu definieren. Die auftretenden Typ-Variablen müssen durch Typ-Ausdrücke (mit evtl. anderen Variablen!) ersetzt werden.
  Gelingt es, beide Ausdrücke auf diese Weise identisch zu machen (zu unifizieren), sind beide Typen äquivalent.
- Das Problem ist unter dem Namen Unifikation bekannt.

Andere Möglichkeiten der Typ-Prüfung in Programmiersprachen, die polymorphe Funktionen erlauben:

1. Man verzichtet auf die Typisierung in der Programmiersprache und auf eine Überprüfung zur Übersetzungszeit und benutzt eine dynamische Typ-Überprüfung zur Laufzeit (z.B. in LISP, Scheme oder Smalltalk)
2. Man verzichtet auf die Typisierung in der Programmiersprache, prüft aber in der Übersetzungsphase jeden Gebrauch eines Namens oder eines anderen Sprachkonstrukts auf einen konsistenten Gebrauch (z.B. in ML oder Haskell)
3. Man erweitert die Syntax der Programmiersprache um Typ-Variable einführen zu können oder man „durchlöchert“ das übliche Typ-System durch die Verwendung von Zeigern oder durch beliebiges „casting“.