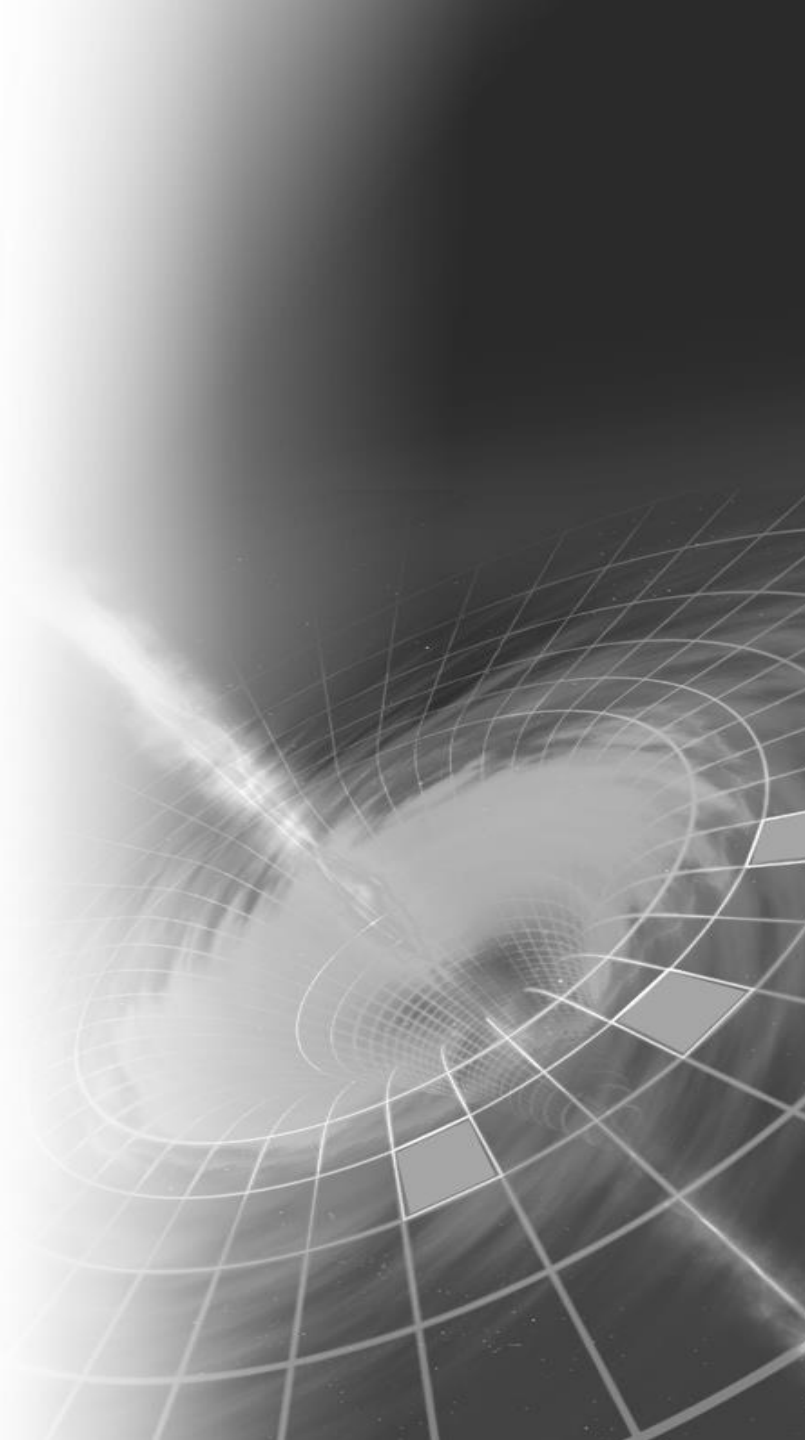
Bachelor-Programm

Compilerbau

im SoSe 2014

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage

fischer@informatik.hu-berlin.de



Typausdrücke, Typprüfung und Typsystem (Wdh.)

Basis

Compiler (**Typ-Checker)**

- weist jeder Komponente im Quellprogramm einen Typausdruck zu
- stellt fest, ob die Typausdrücke konform zu einer Menge von Regeln (Typsystem der Sprache) sind
- hat Potential für Fehlerbestimmung

Typ-Checker ist allgemein ein Übersetzungsschema,

das den Typ jedes Ausdrucks aus den Typen seiner Teilausdrücke bestimmt

und über semantische Regeln die Typtests realisiert

Typausdrücke

- **array** [256] **of char**

  führt zum Typausdruck: *array* ((1..256), *char*)

Attribute für Typprüfung (Wdh.)

- ererbtes Attribut: **type**

- … und deren Simulation:
  mittels
  a) globale Größe
  oder
  b) Zugriff auf Kellerinhalt ~ vorheriger Regel mit bekannter Position

Typkonstruktoren für Beispiel-Sprache

einfache Sprache

als Folge von Deklarationen, gefolgt von einem einzigen Ausdruck

1	P	→	D **;** E
2, 3	D	→	D **;** D | **id:** T
4, 5, 6, 7	T	→	**char** | **integer** | **array [num] of** T | ↑ T
8, 9, 10, 11	E	→	**literal** | **num** | **id** | E **mod** E | E **[** E **]**

Typausdrücke

Basistypen
- char
- integer
- type_error

Konstruktoren
- array
- pointer

Beispiel:

```
x: integer;
y: array [256] of char;
z: ↑char;
x mod 2014
```

zusätzlicher Typ zur Anzeige von Fehlern

Typausdrücke aus Deklarationen und Ausdrücken

P → D ; E
D → D ; D | **id:** T
T → **char** | **integer** | **array [num] of** T | ↑ T
E → **literal** | **num** | **id** | E **mod** E | E **[** E **]**

(1)	P → D; E	
(2)	D → D; D	
(3)	D → id : T	{ addentry (id.name, T.type) }
(4)	T → char	{ T.type := character }
(5)	T → int	{ T.type := integer }
(6)	T → array [num] of T_1	{ T.type := array (1 .. num.val, T_1.type) }
(7)	T → ↑ T_1	{ T.type := pointer (T_1.type) }
(8)	E → literal	{ E.type := character}
(9)	E → num	{ E.type := integer }
(10)	E → id	{E.Type:= lookup (id.name);}
(11)	E → E_1 mod E_2	{ **if** (E_1.type = integer) **and** (E_2.type = integer) **then** E.type := integer **else** E.type := type_error }
(12)	E → E_1 [E_2]	{ **if** (E_2.type = integer) **and** (E_1.type = array (s, t)) **then** E.type := t **else** E.type := type_error }

Symbole mit ihren Typ-Werten (Symboltabelle)

(1)	P → D; E	
(2)	D → D; D	
(3)	D → id : T	{ addtype (id.entry, T.type) }
(4)	T → char	{ T.type := character }
(5)	T → int	{ T.type := integer }
(6)	T → array [num] of T_1	{ T.type := array (num.val, T_1.type) }
(7)	T → ↑ T_1	{ T.type := pointer (T_1.type) }
(8)	E → literal	{ E.type := character}
(9)	E → num	{ E.type := integer }
(10)	E → id	{ E.type := lookup (id.name) }
(11)	E → E_1 mod E_2	{ **if** (E_1.type = integer) **and** (E_2.type = integer) **then** E.type := integer **else** E.type := type_error }
(12)	E → E_1 [E_2]	{ **if** (E_2.type = integer) **and** (E_1.type = array (s, t)) **then** E.type := t **else** E.type := type_error }

in RS von (1) erscheint D vor E
➔ Typen aller **Bezeichner** sind bereits gespeichert bevor Ausdruck E überprüft wird

Symbole mit ihren Typ-Werten (Symboltabelle)

(1)	P → D; E	
(2)	D → D; D	
(3)	D → id : T	{ addentry (id.name, T.type) }
(4)	T → char	{ T.type := character }
(5)	T → int	{ T.type := integer }
(6)	T → array [num] of	{ T.type := array (1 .. num.val, T_1.type) }
(7)	T → ↑ T_1	{ T.type := pointer (T_1.type) }
(8)	E → literal	{ E.type := character}
(9)	E → num	{ E.type := integer }
(10)		
(11)		ype = integer) **then** E.type := integer **else** E.type := type_error }
(12)	E → E_1 [E_2]	{ **if** (E_2.type = integer) **and** (E_1.type = array (s, t)) **then** E.type := t **else** E.type := type_error }

Typinformationen werden mit Namen des Bezeichners in der Symboltabelle verknüpft

Bildung von Typausdrücken *mit Werten aus der Symboltabelle*

(1)	P → D; E	
(2)	D → D; D	
(3)	D → id : T	{ addentry (id.name, T.type) }
(4)	T → char	
(5)	T → int	
(6)	T → array [num] of T_1	
(7)	T → ↑ T_1	{ T := pointer (T_1.type) }
(8)	E → literal	{ E.type := character}
(9)	E → num	{ E.type := integer }
(10)	E → id	{E.type:= lookup (id.name);}
(11)	E → E_1 mod E_2	
(12)	E → E_1 [E_2]	**then** E.type := t **else** E.type := type_error }

Konstanten (repräsentiert durch die Token literal und num) erhalten den Typ aus Wertdarstellung (z.B. Codierung für character bzw. integer)

Der Typ eines **Bezeichners** in einem Ausdruck ergibt sich aus seinem Symboltabelleneintrag (zugeordnetes Attribut)

Bildung gültiger und ungültiger Typausdrücke (1)

(1)	$P \rightarrow D; E$	
(2)	$D \rightarrow D; D$	
(3)	$D \rightarrow id : T$	{ addentry (id.name, T.type) }
(4)	$T \rightarrow char$	{ T.type := character }
(5)	$T \rightarrow int$	{ T.type := integer }
(6)	$T \rightarrow array\ [num]\ of\ T_1$	
(7)	$T \rightarrow \uparrow T_1$	
(8)	$E \rightarrow literal$	
(9)	$E \rightarrow num$	
(10)	$E \rightarrow id$	
(11)	$E \rightarrow E_1\ mod\ E_2$	{ **if** (E_1.type = integer) **and** (E_2.type = integer) **then** E.type := integer **else** E.type := type_error }
(12)	$E \rightarrow E_1\ [E_2]$	{ **if** (E_2.type = integer) **and** (E_1.type = array (s, t)) **then** E.type := t **else** E.type := type_error }

Typ eines **mod**-Ausdrucks ist integer, wenn beide Teilausdrücke integer sind,

sonst ein künstlich eingeführter Fehlertyp type_error, der in weiteren Typberechnungen verwandt werden kann

Bildung gültiger und ungültiger Typausdrücke (2)

(1)	$P \rightarrow D; E$	
(2)	$D \rightarrow D; D$	
(3)	$D \rightarrow id : T$	{ addentry (id.name, T.type) }
(4)	$T \rightarrow char$	{ T.type := character }
(5)	$T \rightarrow int$	{ T.type := integer }
(6)	$T \rightarrow array\ [num]\ of\ T_1$	{ T.type := array (1..num.val, T.type) }
(7)	$T \rightarrow \uparrow T_1$	
(8)	$E \rightarrow literal$	
(9)	$E \rightarrow num$	
(10)	$E \rightarrow id$	
(11)	$E \rightarrow E_1\ mod\ E_2$	**if** E.type := integer **else** E.type := type_error }
(12)	$E \rightarrow E_1\ [E_2]$	{ **if** (E_2.type = integer) **and** (E_1.type = array (i, t)) **then** E.type := t **else** E.type := type_error }

Achtung:
die Indexmenge des Arrays wird **hier** für Typ-berechnung nicht eingesetzt)

- Index beim Aufruf muss integer sein
- Feld-Ausdruck wurde aus i und t konstruiert

➔ t wird Ergebnistyp, sonst Fehlertyp

Erweiterung der Beispielsprache

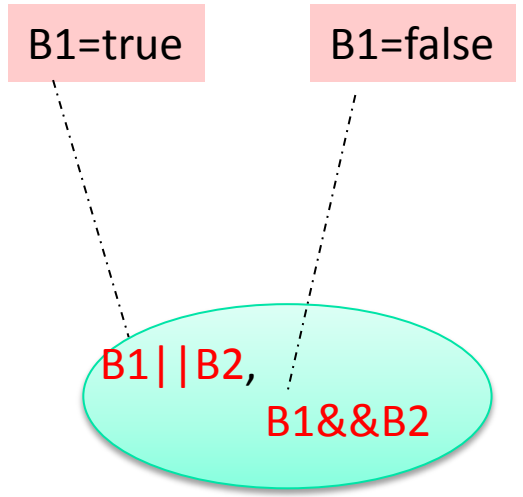
(1) weitere Typen und Ausdrücke

z.B.

real: Arithmetische Ausdrücke

boolean: Vergleichsausdrücke

statische Semantik einer Sprache legt fest,
inwieweit boolesche Ausdrücke in Kontrollflussanweisungen
komplett oder nur partiell ausgewertet werden



prinzipiell erfolgt die Ausdrucksberechnung nach dem gleichen Schema

s. letzte Vorlesung

(2) Anweisungen (Zuweisungen, Verzweigungen, Schleifen)

- werden hier durch Semikolons getrennt
- haben (typischer Weise) keine Werte,
  dennoch Verwendung von void im Normalfall und im Fehlerfall type_error

Erweiterung der Sprache

Programm: Sequenz von Deklarationen, gefolgt von Anweisungen
neue Regeln: (1), (13)-(16)

1	P →	D **;** S
2, 3	D →	D **;** D | **id:** T
4, 5, 6, 7	T →	**boolean** | **char** | **int** | **array [num] of** T | ↑ T
8, 9, 10, 11, 12	E →	**literal** | **num** | **id** | E **mod** E | E **[** E **]**
13	S →	**id :=** E
14	S →	**if** E **then** S_1
15	S →	**while** E **do** S_1
16	S →	S_1 **;** S_2

Semantische Regeln für Anweisungen

(12)	…	
(13)	$S \rightarrow$ id := E	{ S.type := **if** (id.type **=** E.type) **then** void **else** type_error }
(14)	$S \rightarrow$ if E then S_1	{ S.type := **if** (E.type = boolean) **then** S_1.type **else** type_error }
(15)	$S \rightarrow$ while E do S_1	{ S.type := **if** (E.type = boolean) **then** S_1.type **else** type_error}
(16)	$S \rightarrow S_1; S_2$	{ S.type := **if** (S_1.type = void) **and** (S_2.type = void) **then** void **else** type_error }

Weiterverbreitung von Typunverträglichkeiten

Behandlung von Funktionsaufrufen

- zur Aufnahme von Funktionstypen in **Deklarationen**

$T \rightarrow T_1 \Rightarrow T_2$	{ T.type:= function (T_1.type, T_2.type) }

- **Anwendung** einer Funktion auf ein Argument (Call)

$E \rightarrow E_1 (E_2)$	{ E.type:= **if** E_2 .type = s **and** E_1 .type = function(s, t) **then** t **else** type_error }

bedeutet: Typ E_1 muss eine Funktion $s \Rightarrow t$ sein,
dessen Definitionsbereich s vom Typ E_2 sein muss
und dessen Resultattyp t ein Teilbereichstyp von E ist

- Aufnahme weiterer Argumente
  möglich bei Verwendung eines Produkttypkonstruktors

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprac
- Kapitel 3
  Lexikalische Analyse: d
- Kapitel 4
  Syntaktische Analyse:
- Kapitel 5
  Parser-Generatoren: Ya
- Kapitel 6
  Statische Semantikana
- Kapitel 7
  Laufzeitsysteme

- 6.1
  Überblic
- 6.2
  Attribut
- 6.3
  S-attribu
- 6.4
  Attributi
  und erer
- 6.5
  L-attribu
- 6.6
  Verfahre
  Überblic
- 6.7
  Entwurf
- 6.8
  Drei-Adress-Code-Generierung (einige Aspekte)
- 6.9
  Symboltabelle
- 6.10
  Typprüfung

- 6.10.1
  Typsysteme, Typchecker
- 6.10.2
  Typchecker für eine einfache Sprache
- 6.10.3
  Wandlung kompatibler Typen
- 6.10.4
  Überladung von Operatoren und Funktionen
- 6.10.5
  Typcodierung

Typumwandlungen (Typanpassung)

Ausdruck x + i

- es gibt unterschiedliche Maschinenbefehle
  für Operationen in Abhängigkeit der Operandentypen
- Problem
  falls Operanden nicht vom gleichen Typ (z.B. x **real** und i **integer**)
- Sicherstellung,
  dass beide Operanden vom gleichen Typ sind
  (übliche Abfolge: Wandlung des int-Operanden nach real, real-Addition, real-Ergebnis)
- Typumwandlung kann vom Typüberprüfer als semantische Aktion realisiert werden

Postfixnotation im Zwischencode: x i inttoreal realadd

- x auf den Stack,
- i auf den Stack,
- int-nach-real-Umwandlung (tos→ 1.Operand),
- real-Addition (tos→ 2.Operand, tos→ 1.Operand)

Prinzip der Typumwandlung

E.type ϵ {integer, float}

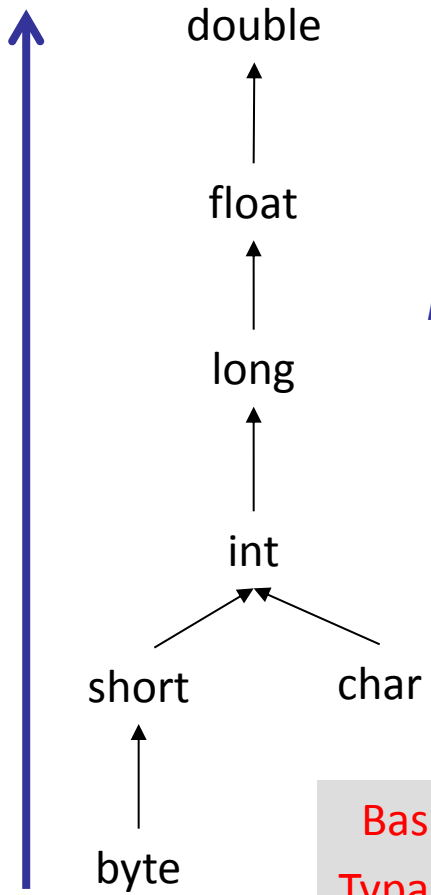
Typkonvertierungsregeln sind sprachabhängig

E → E1 + E2	{ **if** (E1.type = integer **and** E2.type = integer) **then** E.type = integer **else if** (E1.type = float **and** E2.type = integer) **then** E.type= … }

Beispiel Java: Die Sprache unterscheidet zwischen:

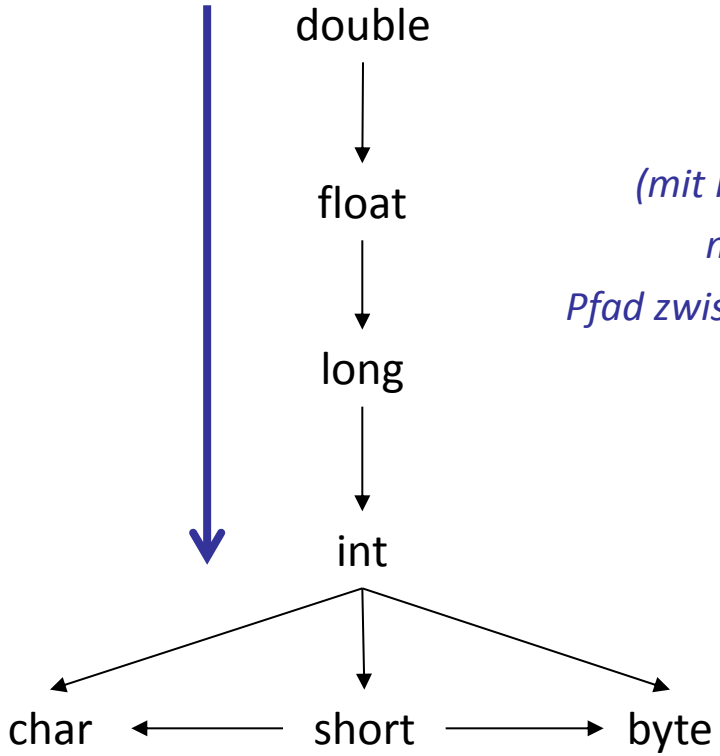
- erweiternder Konvertierung (widening): informationserhaltend
- einengender Konvertierung (narrowing): mit Informationsverlust

Konvertierungen primitiver Java-Typen



widening

jeder Typ einer Ebene kann durch Typ höherer Ebenen ohne Informationsverlust dargestellt werden



narrowing

Überführung (mit Informationsverlust) möglich, falls ein Pfad zwischen den Typen besteht

Basis für implizite Typanpassung durch den Compiler

Basis für erlaubte explizite Typanpassung per Cast im Quellprogramm

Implizite und explizite Typumwandlungen i.Allg.

implizite Umwandlungen

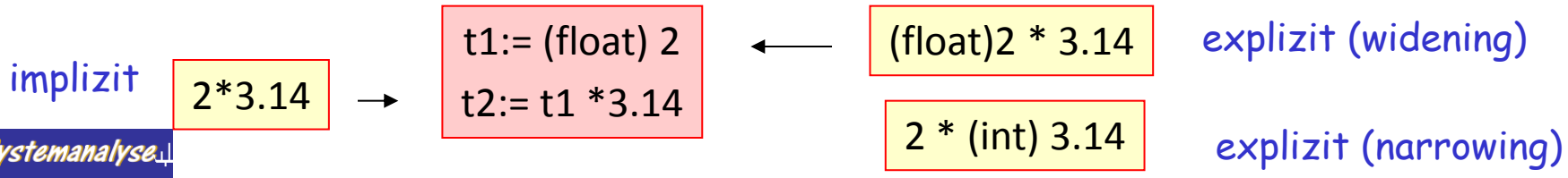
- automatisch durch den Compiler
- in den meisten Sprachen: nur widening-Konvertierungen erlaubt

Beispiel: real → int sollte **keine** implizite Umwandlung sein,
weil Nach-Komma-Stellen verloren gehen können

Sprachen haben u.U. sehr viele Typen
um Typanpassungsfälle in Grenzen zu halten,
bedarf es eines sorgfältigen Entwurfs semantischer Regeln

explizite Typumwandlung

- vom Programmierer erzwungen
(durch Aufruf von Funktionen, verursachen also keine neuen Probleme)
- narrowing-Konvertierungen (nicht-typsicheres »Casten«)



Semantische Aktionen für Typanpassung

Semantische Aktionen von E → E1 + E2 verwendet zwei Operationen

max und widen bilden Basis für **implizite** Typanpassung einer Sprache

type max (type t1, type t2)

kleinste gemeinsame Ausprägung
von t1 und t2 in der Typhierarchie oder Fehler (falls leer)

```
addr widen (addr a, type t, type w)
    if (t == w) return a;
    else if (t== integer and w==float) {
            temp= new Temp();
            gen (temp '=' '(float)' *a);
            return temp;
    }
    else error;
}
```

Konvertierung:
Inhalt von Adresse a vom Typ t zum Typ w
am Beispiel von integer und float

erzeugt Drei-Adressbefehle

Typanpassung in Gestalt semantischer Regeln

$E \rightarrow E_1 + E_2$	{ E.type:= max (E_1.type, E_2.type);
	a1:= widen (E_1.addr, E_1.type, E.type);
	a2:= widen (E_2.addr, E_2.type, E.type);
	
	E.addr= new Temp();
	gen (E.addr '=' a1 '+' a2); }

im Allg. werden Werte zweier verschiedener Typen
- in dritten Typ überführt (max)
- und dann verknüpft

temporäre Variablen: a1, a2

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprac
- Kapitel 3
  Lexikalische Analyse: d
- Kapitel 4
  Syntaktische Analyse:
- Kapitel 5
  Parser-Generatoren: Ya
- Kapitel 6
  Statische Semantikana
- Kapitel 7
  Laufzeitsysteme

- 6.1
  Überblic
- 6.2
  Attribut
- 6.3
  S-attribu
- 6.4
  Attributi
  und erer
- 6.5
  L-attribu
- 6.6
  Verfahre
  Überblic
- 6.7
  Entwurf
- 6.8
  Drei-Adress-Code-Generierung (einige Aspekte)
- 6.9
  Symboltabelle
- 6.10
  Typprüfung

- 6.10.1
  Typsysteme, Typchecker
- 6.10.2
  Typchecker für eine einfache Sprache
- 6.10.3
  Wandlung kompatibler Typen
- 6.10.4
  Überladung von Operatoren und Funktionen
- 6.10.5
  Typcodierung

Prinzip der Überladung

- überladene Symbole

  haben in Abhängigkeit vom Kontext verschiedene Bedeutungen

  Auflösung: durch Kontextfeststellung bei Identifikation der jeweiligen Symbolbedeutung

- Java-Funktion: Kontextbestimmung reduziert sich auf Auswertung der Typen der Parameter

  Beispiel:

  (1) vordefinierter Operator: **+** (Semantik: Stringverkettung oder Addition)

  (2)
  ```
  void err () { …}
  void err (String s) {…}
  ```

- Ada-Operator (vordefiniert): *
  - entspricht einer Funktion:
  - kann überladen werden mit:

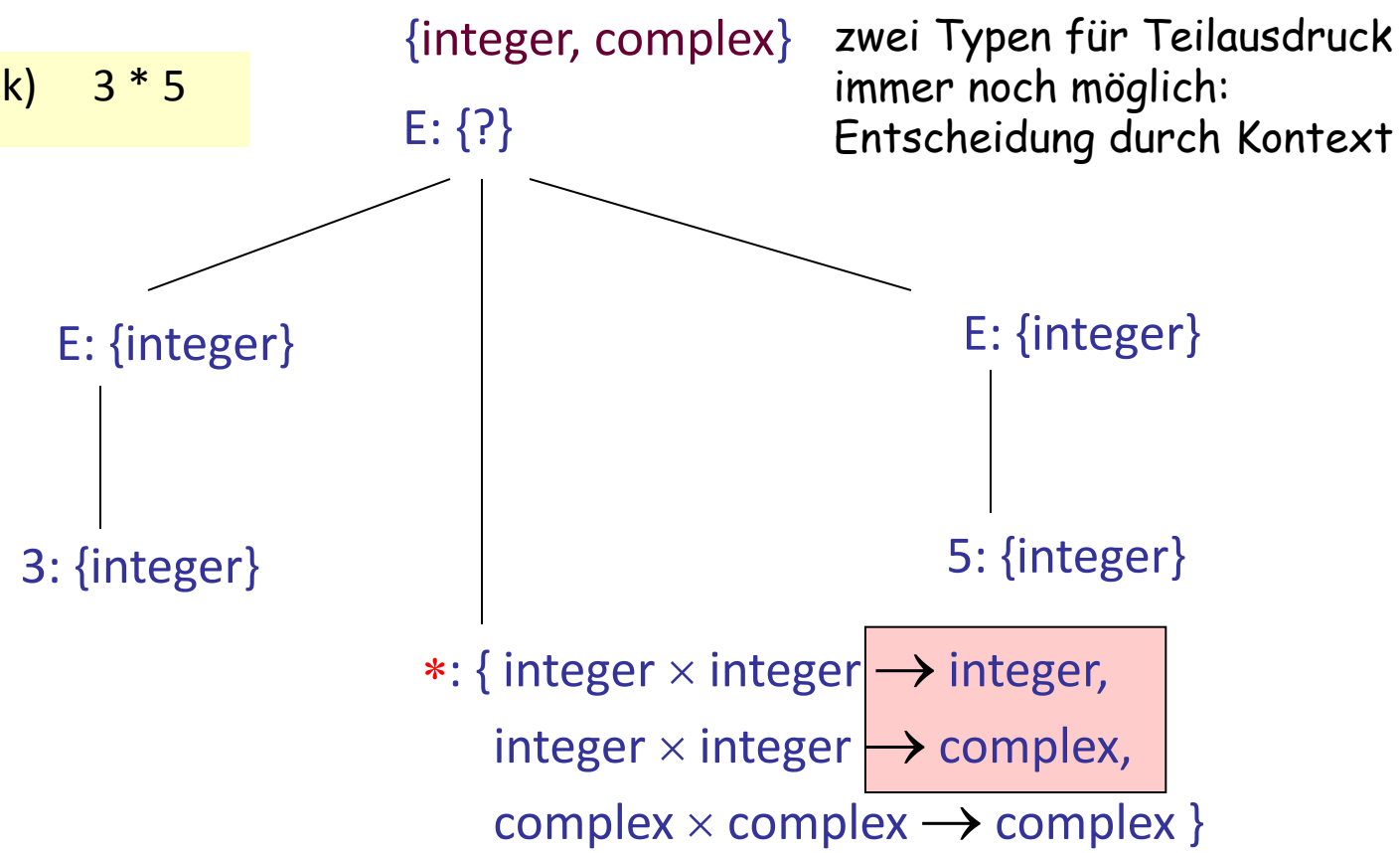
  ```
  function "*"(i, j: integer) return integer;
  function "*"(i, j: integer) return complex;
  function "*"(i, j: complex) return complex;
  ```

  - **Kontextbestimmung:** neben den Operanden muss auch das Ziel der Zuweisung ausgewertet werden

Überladung von Funktionen und Operatoren (3)

Beispiel

(Ada-Ausdruck) 3 * 5



Ada verlangt eine **Typfehlererkennung**,
falls für einen konkreten Kontext **mehr als ein Typ** für einen Ausdruck möglich ist

Berechnung von Typmengen

bislang: jeder Ausdruck ist nur von einem einzigen Typ, so auch ein Funktionsaufruf

(1)	$E \rightarrow E_1 (E_2)$	E.type := **if** (E_2.type = s) **and** (E_1.type = s ⇨ t **then** t **else** type_error

jetzt: Verallgemeinerung der Regel auf Mengen von Typen

(1)	$E' \rightarrow E$	E'.types := E.types
(2)	$E \rightarrow$ **id**	E.types := lookup (id.name)
(3)	$E \rightarrow E_1 (E_2)$	E.types := { t | es gibt ein s in E_2.types, so dass s ⇨ t in E_1.types } nur auf Basis der Operanden-Angabe !!!

liefert Menge möglicher Typen für einen Eintrag

Lesart: wenn s einer der Typen von E_2 ist und
einer der Typen von E_1 dieses s auf t abbilden kann,
dann ist t einer der Typen von E1(E2), also E

Ziel: E'.types darf nur einen einzigen Typ t enthalten, sonst Fehler
E: errerbtes Attribut unique

Reduktion der Typmenge eines Ausdrucks

Der vollständige Ausdruck wird durch E‘ generiert
E‘.types soll deshalb nur aus einem Element bestehen

der einzelne Typ t wird (als unique) ererbt

(1)	E' → E	E'.types := E.types E.unique:= **if** E'.types = {t} **then** t **else** type_error
(2)	E → **id**	E.types := lookup (id.name)
(3)	E → E_1 (E_2)	E.types := { s' | *es existiert ein* s *in* E_2.types, *so dass* s → s' *in* E_1.types } t:= E.unique S:= { s ∈ E_2.types **and** s → t ∈ E_1.types } E_2.unique:= **if** S = {s} **then** s **else** type_error E_1.unique:= **if** S = {s} **then** s → t **else** type_error

Wenn eine Funktion $E_1(E_2)$ den Typ t zurückliefert, dann können wir den Typ s finden, der für das Argument E_2 machbar ist

Gleichzeitig ist s→t für die Funktion machbar.

Menge S wird benutzt, um zu prüfen, dass es einen einzigen Typ s mit dieser Eigenschaft gibt

Erster Durchlauf: Berechnung von types (unten → oben)

(1)	$E' \rightarrow E$	E'.types := E.types E.unique:= **if** E'.types = {t} **then** t **else** type_error
(2)	$E \rightarrow$ **id**	E.types := lookup (id.name)
(3)	$E \rightarrow E_1 (E_2)$	E.types := { s' | es existiert ein s in E_2.types, so dass s → s' in E_1.types } t:= E.unique S:= { s ∈ E_2.types **and** s → t ∈ E_1.types } E_2.unique:= **if** S = {s} **then** s **else** type_error E_1.unique:= **if** S = {s} **then** s → t **else** type_error

3

1

2

Zweiter Durchlauf: Verbreitung von unique (oben→unten)

(1)	$E' \rightarrow E$	E'.types := E.types E.unique:= **if** E'.types = {t} **then** t **else** type_error E'.code:= E.code	1
(2)	$E \rightarrow$ **id**	E.types := lookup (id.name) E.code:= gen(id.lexeme ':' E.unique)	
(3)	$E \rightarrow E_1 (E_2)$	E.types := { s' | es existiert ein s in E_2.types, so dass s $\rightarrow$ s' in E_1.types } t:= E.unique S:= { s $\in$ E_2.types **and** s $\rightarrow$ t $\in$ E_1.types } E_2.unique:= **if** S = {s} **then** s **else** type_error E_1.unique:= **if** S = {s} **then** s $\rightarrow$ t **else** type_error E.code:= E_1.code || E_2.code || gen('apply' ':' E.unique)	2 3

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprac
- Kapitel 3
  Lexikalische Analyse: d
- Kapitel 4
  Syntaktische Analyse:
- Kapitel 5
  Parser-Generatoren: Ya
- Kapitel 6
  Statische Semantikana
- Kapitel 7
  Laufzeitsysteme

- 6.1
  Überblic
- 6.2
  Attribut
- 6.3
  S-attribu
- 6.4
  Attributi
  und erer
- 6.5
  L-attribu
- 6.6
  Verfahre
  Überblic
- 6.7
  Entwurf
- 6.8
  Drei-Adress-Code-Generierung (einige Aspekte)
- 6.9
  Symboltabelle
- 6.10
  Typprüfung

- 6.10.1
  Typsysteme, Typchecker
- 6.10.2
  Typchecker für eine einfache Sprache
- 6.10.3
  Wandlung kompatibler Typen
- 6.10.4
  Überladung von Operatoren und Funktionen
- 6.10.5
  Typcodierung

Strukturelle Typäquivalenz (bereits bekannt)

- natürlicher Ansatz,
  wenn Typausdrücke über einfache Typen und Konstruktoren gebildet werden

• $s \equiv t$,	falls s und t vom gleichen Basistyp sind
• $array(s_1,s_2) \equiv array(t_1,t_2)$,	falls $s_1 \equiv t_1$ und $s_2 \equiv t_2$
• $s_1 \times s_2 \equiv t_1 \times t_2$,	falls $s_1 \equiv t_1$ und $s_2 \equiv t_2$
• $\uparrow s \equiv \uparrow t$,	falls $s \equiv t$
• $s_1 \Rightarrow s_2 \equiv t_1 \Rightarrow t_2$	falls $s_1 \equiv t_1$ und $s_2 \equiv t_2$

Achtung: (wissen bereits) Abweichungen in der Praxis möglich

Algorithmus: Feststellung von struktureller Typgleichheit

```
(1)   function structEquiv (s, t : type): boolean;
(2)   begin
(3)     if s und t vom selben Basistyp
(4)         then return true
(5)     else if s = array(s1,s2) and t= array(t1,t2)
(6)         then return structEquiv(s1,t1) and structEquiv(s2,t2)
(7)     else if s = product(s1,s2) and t= product(t1,t2)
(8)         then return structEquiv(s1,t1) and structEquiv(s2,t2)
(9)     else if s = pointer(s1) and t= pointer(t1)
(10)        then return structEquiv(s1,t1)
(11)    else if s = function(s1,s2) and t= function(t1,t2)
(12)        then return structEquiv(s1,t1) and structEquiv(s2,t2)
(13)    else return false
(14)  end
```

rekursiver Vergleich der Struktur ohne Zyklentest
➔anwendbar auf DAG-Repräsentation

➔kann erweitert werden (weitere Typen)

➔kann modifiziert werden (Gleichheitsmodifikation)

Repräsentation von Typausdrücken

- bislang: ~ Syntaxbaum, es gibt weitere Formen
- kompakte Darstellung stammt von D.M. Ritchie:

  Typausdruck wird als Bitfolge codiert und als einzelner int-Wert interpretiert:
  - ➔ unterschiedliche Werte repräsentieren strukturell unterschiedliche Typen
- Verbindung des letzten Algorithmus mit Codierungsmethode:
  - zuerst Vergleich der strukturellen Gleichheit mit Codierungsmethode
  - Fortsetzung mit Algorithmus im Fall von Gleichheit

Beispiel: Alternative Typausdrucksrepräsentation

- Typkonstruktoren (vereinfacht)
  - pointer(t) — Zeiger auf Typ t
  - fReturns(t) — Funktion mit einigen Parametern, die ein Objekt vom Typ t zurückgibt (Beachtung der Parametertypen an anderer Stelle)
  - array(t) — Feld unbestimmter Länge mit Elementen von Typ t (Beachtung der Länge des Feldes an anderer Stelle)

Beispiel: einfache Struktur (Konstruktoren mit einem einzigen Parameter)

char
fReturns(char)
pointer(fReturns(char))
array(pointer(fReturns(char)))

jeder Ausdruck kann durch Bitfolge codiert werden

Beispiel: Alternative Typausdrucksrepräsentation

drei Typkonstruktoren → zwei Bits

Typkonstruktor	Codierung
pointer	01
array	10
fReturns	11

vier Basistypen → vier Bits
(erweiterungsfähig)

Basistyp	Codierung
boolean	0000
char	0001
integer	0010
real	0011

Ausdruck	Codierung
char	000000 0001
fReturns(char)	000011 0001
pointer(fReturns(char))	000111 0001
array(pointer(fReturns(char)))	100111 0001

Codierung des
einfachen Typs
(rechts stehende vier Bits)

Ausweitung auf Record-Typen:
jeder Record wird als ein Basistyp betrachtet
separate Bitfolge codiert Typen aller Felder

Codierung: Typausdrücke

Vorzüge

- Speichereffizienz
- einfache Verfolgung der Anwendung von Typkonstruktoren

Eigenschaften

- zwei verschiedene Bitfolgen
  können nicht den gleichen Typ repräsentieren
- unterschiedliche Typen können aber gleiche Bitfolgen haben (Nichtbeachtung von Funktionsargumenten, Arraygrößen)

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprac
- Kapitel 3
  Lexikalische Analyse: d
- Kapitel 4
  Syntaktische Analyse:
- Kapitel 5
  Parser-Generatoren: Ya
- Kapitel 6
  Statische Semantikana
- Kapitel 7
  Laufzeitsysteme

- 6.1
  Überblic
- 6.2
  Attribut
- 6.3
  S-attribu
- 6.4
  Attributi
  und erer
- 6.5
  L-attribu
- 6.6
  Verfahre
  Überblic
- 6.7
  Entwurf
- 6.8
  Drei-Adress-Code-Generierung (einige Aspekte)
- 6.9
  Symboltabelle
- 6.10
  Typprüfung

- 6.10.1
  Typsysteme, Typchecker
- 6.10.2
  Typchecker für eine einfache Sprache
- 6.10.3
  Wandlung kompatibler Typen
- 6.10.4
  Überladung von Operatoren und Funktionen
- 6.10.5
  Typcodierung

Position

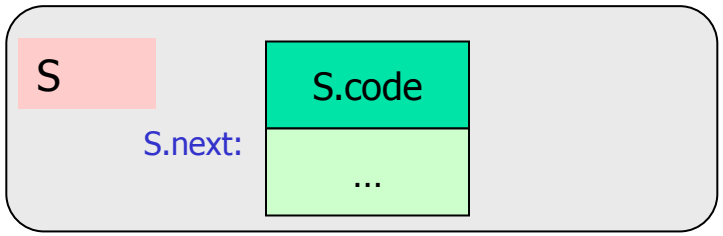
- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

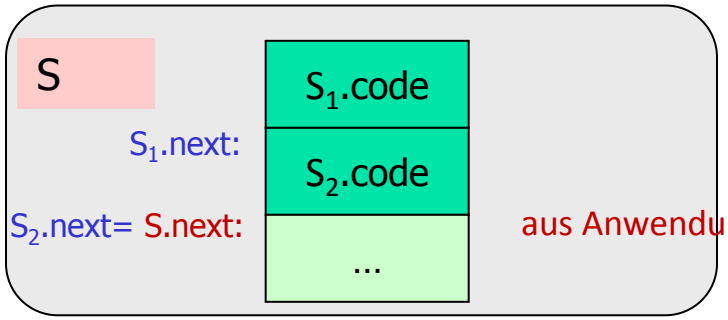
- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprac
- Kapitel 3
  Lexikalische Analyse: d
- Kapitel 4
  Syntaktische Analyse:
- Kapitel 5
  Parser-Generatoren: Ya
- Kapitel 6
  Statische Semantikana
- Kapitel 7
  Laufzeitsysteme
- ~~Kapitel 8~~
  ~~Ausblick: Codegenerier~~

- 6.1
  Überblick: Grammatik-basierte Übersetzung
- 6.2
  Attributgrammatiken
- 6.3
  S-attributierte Syntaxdefinitionen
- 6.4
  Attributierte Syntaxdefinitionen mit synthetisierten und ererbten Attributen
- 6.5
  L-attributierte Syntaxdefinitionen
- 6.6
  Verfahren syntaxgesteuerter Übersetzungen im Überblick
- 6.7
  Entwurf syntaxgesteuerter Übersetzungen
- 6.8
  Drei-Adress-Code-Generierung (einige Aspekte)
- 6.9
  Symboltabelle
- 6.10
  Typprüfung
- 6.11
  Drei-Adress-Code-Generierung (weitere Aspekte)

Übersetzung von Kontrollflussanweisungen (1)

$P \rightarrow S$;
$S \rightarrow S_1$; S_2
$S \rightarrow$ if (C) S_1
$S \rightarrow$ if (C) S_1 else S_2
$S \rightarrow$ while (C) S_1

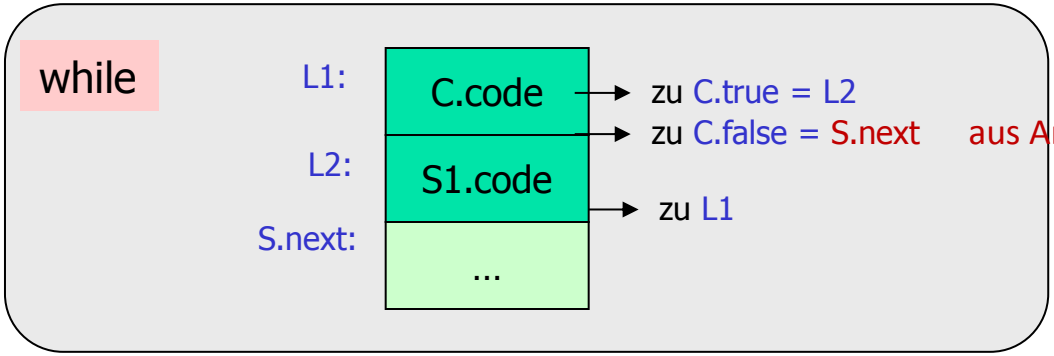




aus Anwendung der vorigen Regel

Produktion	semantische Regel
$P \rightarrow S$;	S.next:= newlabel() P.code:= S.code || label(S.next)
$S \rightarrow S_1$; S_2	S_2.next:= S.next S_1.next:= newlabel() S.code:= S_1.code || label(S_1.next) || S_2.code

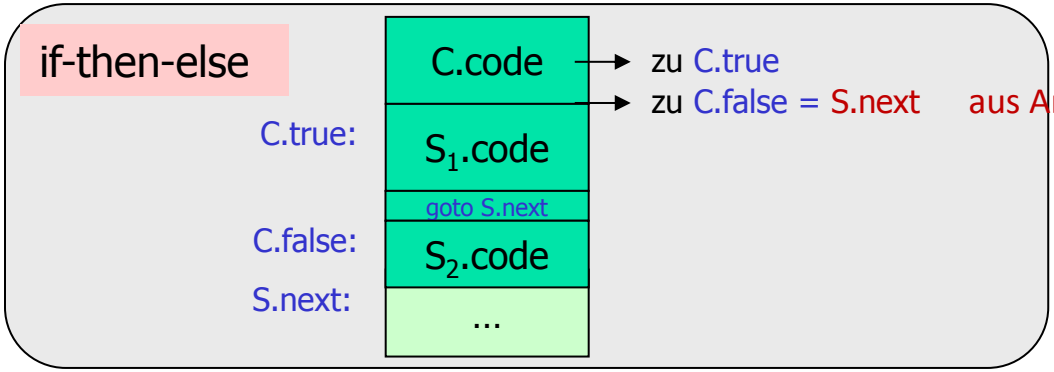
Übersetzung von Kontrollflussanweisungen (2)



Produktion	semantische Regel
S → **while** (C) **do** S_1	L1:= newlabel() L2:= newlabel() C.false:= S.next S_1.next:= L1 S.code || label(L1) || C.code || || label(L2) || S_1.code || **gen** (´goto´ L1)

gen
erzeugt Drei-Adressbefehle

Übersetzung von Kontrollflussanweisungen (3)



Produktion	semantische Regel
$S \rightarrow$ **if (**C**)** S_1 **else** S_2	C.true:= newlabel() C.false:= newlabel() S_1.next:= S_2.next:= S.next S.code := C.code || label(C.true) || S_1.code || gen(´goto´ S.next) || label(C.false) || S_2.code

Übersetzung von Zuweisungsanweisungen

Produktion	semantische Regel
S → **id:=** E	{ p:= lookup(id.name); **if** p=/ nil **then** gen(p ´:=´ E.entry) **else** error }
E → E1 **+** E2	{ E.entry:= newtemp(); gen(E1.entry ´+´ E2.entry) }
E → E1 * E2	{ E.entry:= newtemp(); gen(E1.entry ´*´ E2.entry) }
E → **-** E1	{ E.entry:= newtemp(); gen(E.entry ´:=´ ´uminus´ E1.entry) }
E → **(** E1 **)**	{ E.entry:= E1.entry }
E → **id**	{ p:= lookup(id.name); **if** p=/ nil **then** E.entry:= p **else** error }

lookup
prüft ob Eintrag vorhanden ist,
wenn ja, wird Zeiger - sonst nil zurückgegeben
(dabei werden die Deklarationsniveaus berücksichtigt)

gen
erzeugt Drei-Adressbefehle und schreibt sie in eine Ausgabedatei

Übersetzung von Deklarationen (Wdh.)

Produktionen mit semantischen Regeln

P → D	{ offset:= 0 }
D → D ; D	
D → **id** : T ;	{ top.put (id.lexeme, T.type, offset); offset:= offset + T.width }
T → **int**	{ T.type:= integer; T.width:=4}
T → **real**	{ T.type:= real; B.width:=8}
T → **array** **[num]** **of** T 1	{ T.type:= array(**num**.value, T1.type); T.width:= num.value * T1.width }
T → ↑ T1	{ T.type:= pointer(T1.type); T.width:= 4 }

P → MD

M → ε { offset:= 0 }

Übersetzung lokaler Prozedur-Deklarationen

P → D
D → D ; D
D → id: T

D → proc id ; D ; S

Vorgehen:

- jede Prozedur erhält eine **eigene** Symboltabelle, die wie andere Symboltabellen auf einen Stack geschoben werden
  (dabei Prinzip: wie bei verschachtelten Records, s. letzte Vorlesung)

- es stehen **zwei (synchrone) Stacks** zur Verfügung:
  - symTptr (Erfassung der hierarchischen Symboltabellen)
  - Offset (Erfassung der zugehörigen relativen Adressen der Anfänge der Symboltabellen)

```
program sort (input, output);
  var a: array [0..10] of integer;
      x: integer;

  procedure readarray;
      var i: integer;
  begin … a … end {readarray};

  procedure exchange (i, j: integer);
  begin
      x:= a[i]; a[i]:= a[j]; a[j]:= x
  end {exchange };

  procedure qicksort (m, n: integer);
      var k,v: integer;

      function partition (y, z: integer): integer;
          var i, j: integer;
      begin
          …a…
          …v…
          …exchange(i,j) …
      end {partition };

  begin
      …
  end {quicksort };
begin
…
end {sort }.
```

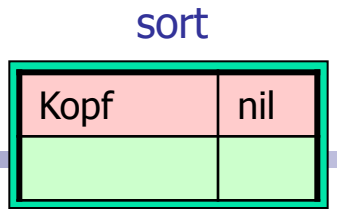
Verweis auf die vorherige Symboltabelle

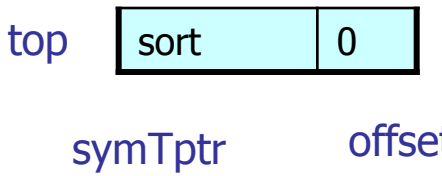
Produktionen mit semantischen Regeln

P → MD	{ addwidth (t, top(symTptr); top(offset)); pop(symTptr)); pop(offset) }
M → ε	{ t:= makeTable(nil); push(t, symTptr); push(0, offset) }
D → D **;** D	
D → **proc id;** N D1**;** S	{ t= top(symTptr)); addwidth (t, top(offset)); pop(symTptr); pop(offset); enterproc(top(symTptr), id.lexeme, t) }
D → **id:** T	{ enter(top(symTptr), id.lexeme, T.type, top(offset)); top(offset):= top(offset) + T.width}
N → ε	{ t:= makeTable(top(symTptr)); push(t, symTptr); push(0, offset)}

vor Analyse der globalen Deklarationen
- erzeugte t als neue Symboltabelle (für sort)
- schiebe t auf den Symboltabellen-Stack
- schiebe 0 als relative Adresse von t auf Offset-Stack

```
program sort (input, output);
  var a: array [0..10] of integer;
      x: integer;

  procedure readarray;
        var i: integer;
  begin … a … end {readarray};

  procedure exchange (i, j: integer);
  begin
        x:= a[i]; a[i]:= a[j]; a[j]:= x
  end {exchange };

  procedure qicksort (m, n: integer);
        var k,v: integer;

      function partition (y, z: integer): integer;
                var i, j: integer;
      begin
                …a…
                …v…
                …exchange(i,j) …
      end {partition };

  begin
        …
  end {quicksort };
begin
…
end {sort }.
```





```
program sort (input, output);
    var a: array [0..10] of integer;
        x: integer;

    procedure readarray;
            var i: integer;
    begin … a … end {readarray};

    procedure exchange (i, j: integer);
    begin
            x:= a[i]; a[i]:= a[j]; a[j]:= x
    end {exchange };

    procedure qicksort (m, n: integer);
            var k,v: integer;

        function partition (y, z: integer): integer;
                    var i, j: integer;
        begin
                    …a…
                    …v…
                    …exchange(i,j) …
        end {partition };

    begin
            …
    end {quicksort };
begin
…
end {sort }.
```

Produktionen mit semantischen Regeln

P → MD	{ addwidth (t, top(symTptr); top(offset)); pop(symTptr)); pop(offset) }
M → ε	{ t:= makeTable(nil); push(t, symTptr)); push(0, offset) }
D → D **;** D	
D → **proc id;** N D1**;** S	{ t= top(symTptr)); addwidth (t, top(offset)); pop(symTptr); pop(offset); enterproc(top(symTptr), id.lexeme, t) }
D → **id:** T	{ enter(top(symTptr), id.lexeme, T.type, top(offset)); top(offset):= top(offset) + T.width}
N → ε	{ t:= makeTable(top(symTptr)); push(t, symTptr); push(0, offset)}

Symboltabelleneintrag

- *aktuelle Symboltabelle (Zeiger vom Stack)*
- *Name mit Attributen und relative Adresse (vorher bestimmt aus Offset-Stack)*
- *Aktualisierung des Offsets der aktuellen Symboltabelle*

```
program sort (input, output);
  var a: array [0..10] of integer;
      x: integer;

  procedure readarray;
        var i: integer;
  begin … a … end {readarray};

  procedure exchange (i, j: integer);
  begin
        x:= a[i]; a[i]:= a[j]; a[j]:= x
  end {exchange };

  procedure qicksort (m, n: integer);
        var k,v: integer;

      function partition (y, z: integer): integer;
                var i, j: integer;
      begin
                …a…
                …v…
                …exchange(i,j) …
      end {partition };

  begin
        …
  end {quicksort };
begin
…
end {sort }.
```

sort

Kopf	nil
a	…
x	…

top

sort	0

symTptr offset

```
program sort (input, output);
  var a: array [0..10] of integer;
      x: integer;

  procedure readarray;
      var i: integer;
  begin … a … end {readarray};

  procedure exchange (i, j: integer);
  begin
      x:= a[i]; a[i]:= a[j]; a[j]:= x
  end {exchange };

  procedure qicksort (m, n: integer);
      var k,v: integer;

      function partition (y, z: integer): integer;
          var i, j: integer;
      begin
          …a…
          …v…
          …exchange(i,j) …
      end {partition };

  begin
      …
  end {quicksort };
begin
…
end {sort }.
```

Verweis auf die vorherige Symboltabelle

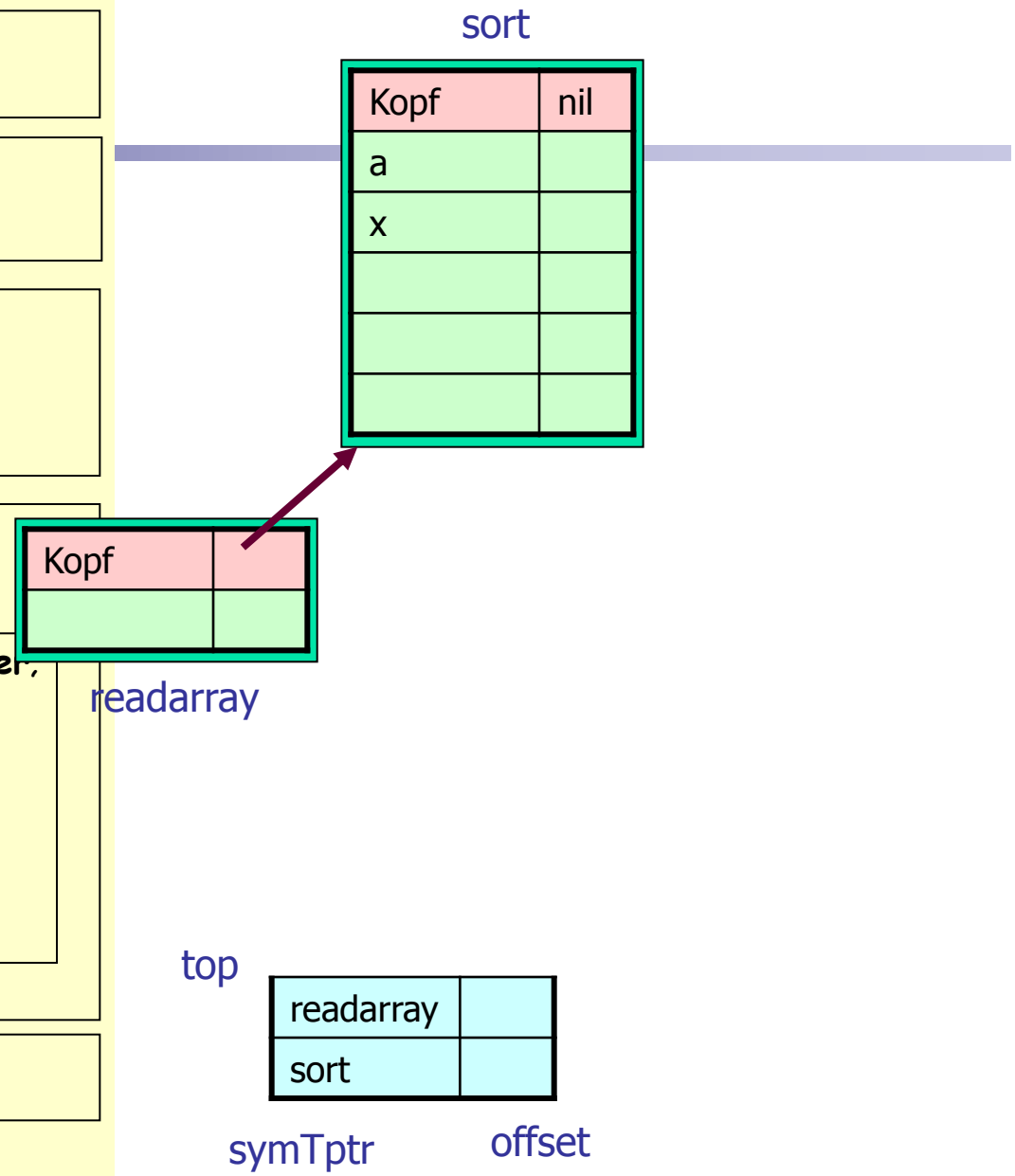
Produktionen mit semantischen Regeln

$P \rightarrow MD$	{ addwidth (t, top(symTptr); top(offset)); pop(symTptr)); pop(offset) }
$M \rightarrow \varepsilon$	{ t:= makeTable(nil); push(t, symTptr)); push(0, offset) }
$D \rightarrow D$ **;** D	
$D \rightarrow$ **proc id;** N D1**;** S	{ t= top(symTptr)); addwidth (t, top(offset)); pop(symTptr); pop(offset); enterproc(top(symTptr), id.lexeme, t) }
$D \rightarrow$ **id:** T	{ enter(top(symTptr), id.lexeme, T.type, top(offset)); top(offset):= top(offset) + T.width}
$N \rightarrow \varepsilon$	{ t:= makeTable(top(symTptr)); push(t, symTptr); push(0, offset)}

vor Analyse der lokalen Prozedur-Deklarationen

- *erzeugte t alsneue Symboltabelle (für readarray)*
- *schiebe t auf den Symboltabellen-Stack*
- *schiebe 0 als relative Adresse von t auf Offset-Stack*

```
program sort (input, output);
  var a: array [0..10] of integer;
      x: integer;

  procedure readarray;
        var i: integer;
  begin … a … end {readarray};

  procedure exchange (i, j: integer);
  begin
        x:= a[i]; a[i]:= a[j]; a[j]:= x
  end {exchange };

  procedure qicksort (m, n: integer);
        var k,v: integer;

      function partition (y, z: integer): integer;
                var i, j: integer;
      begin
                …a…
                …v…
                …exchange(i,j) …
      end {partition };

  begin
        …
  end {quicksort };
begin
…
end {sort }.
```



```
program sort (input, output);
  var a: array [0..10] of integer;
      x: integer;

  procedure readarray;
      var i: integer;
  begin … a … end {readarray};

  procedure exchange (i, j: integer);
  begin
      x:= a[i]; a[i]:= a[j]; a[j]:= x
  end {exchange };

  procedure qicksort (m, n: integer);
      var k,v: integer;

      function partition (y, z: integer): integer;
          var i, j: integer;
      begin
          …a…
          …v…
          …exchange(i,j) …
      end {partition };

  begin
      …
  end {quicksort };
begin
…
end {sort }.
```

Pro

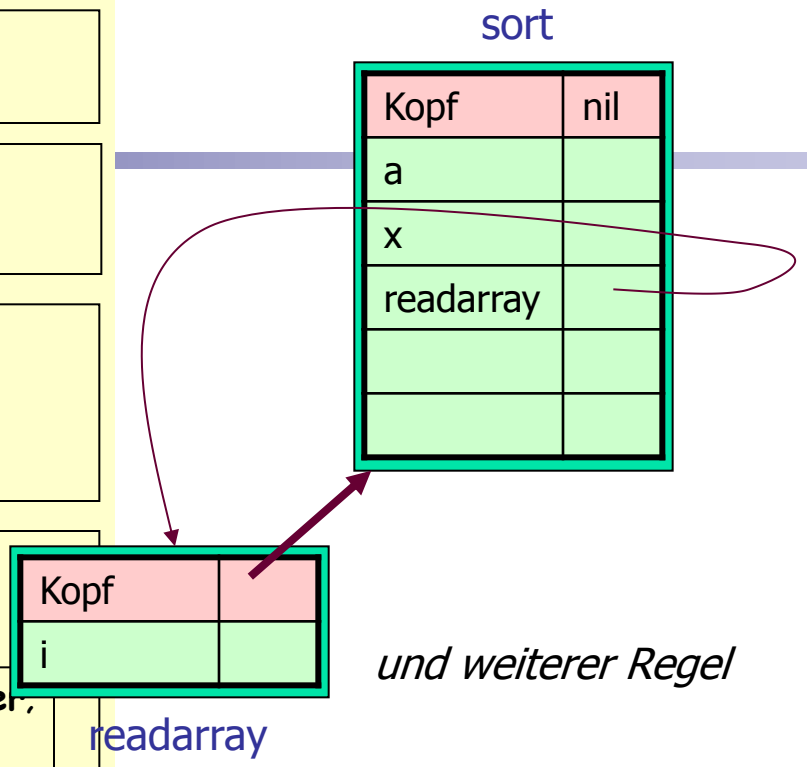
P → MD	
M → ε	{ t:= makeTable(nil); push(t, symTptr)); push(0, offset) }
D → D ; D	
D → **proc id;** N D1; S	{ t= top(symTptr)); addwidth (t, top(offset)); pop(symTptr); pop(offset); enterproc(top(symTptr), id.lexeme, t) }
D → **id:** T	{ enter(top(symTptr), id.lexeme, T.type, top(offset)); top(offset):= top(offset) + T.width}

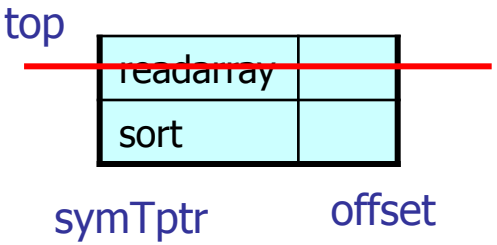
- erzeugt einen neuen Eintrag für die Prozedur mit Namen *id.lexeme* in der Tabelle *top(symTptr)* (ist die Symboltabelle der umschließenden Prozedur)

Symboltabelleneintrag

- *Aktualisierung der Länge der aktuellen Symboltabelle (Zeiger vom Stack) mit Wert aus Offset-Stack*
- *Name mit Attributen und relative Adresse (vorher bestimmt aus Offset-Stack)*
- *Aktualisierung des Offsets der aktuellen Symboltabelle*
- *oberste Elemente von beiden Stapeln werden entfernt Fortführung der Analyse der umschließenden Prozedur*

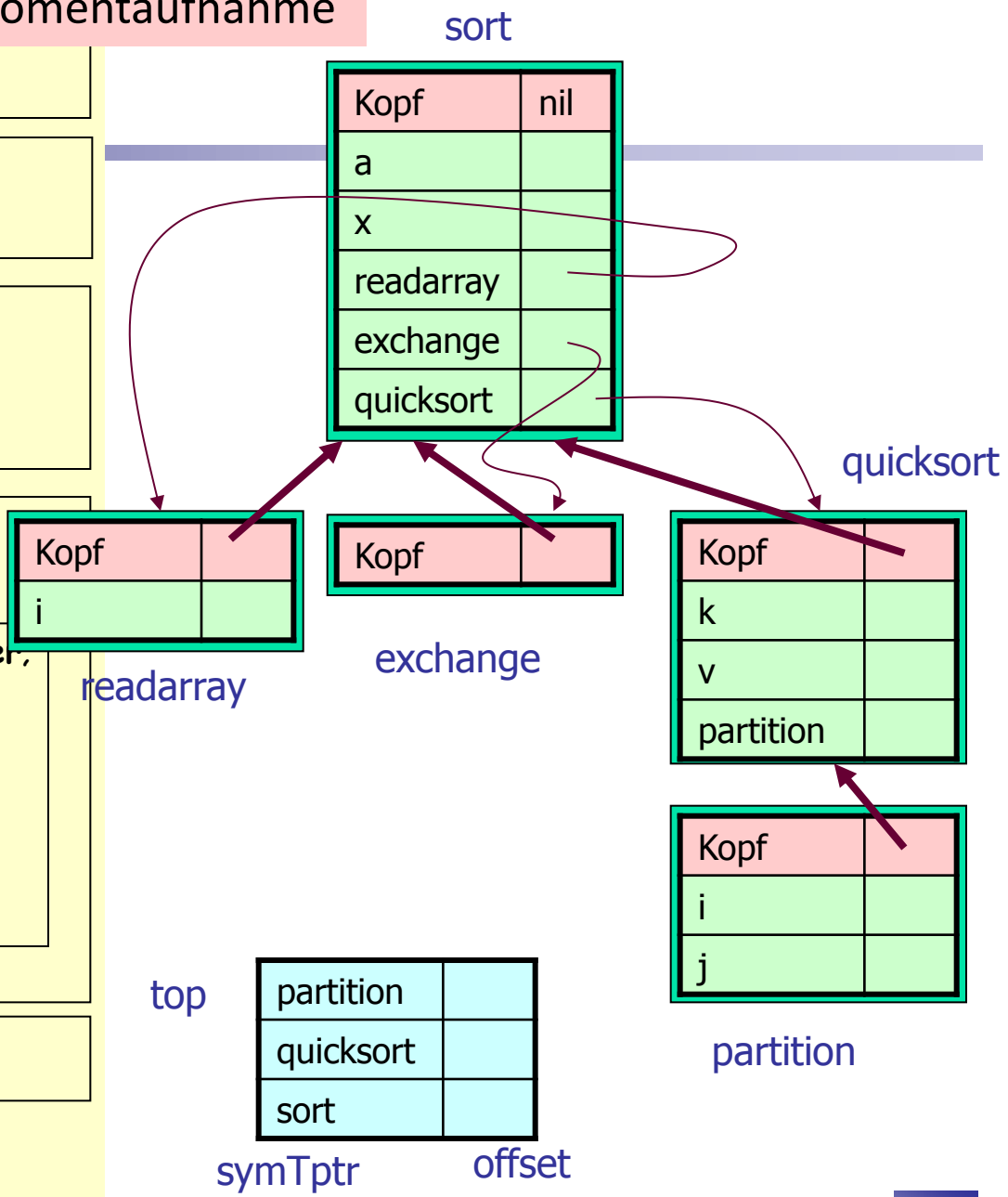
```
program sort (input, output);
  var a: array [0..10] of integer;
      x: integer;

  procedure readarray;
      var i: integer;
  begin … a … end {readarray};

  procedure exchange (i, j: integer);
  begin
      x:= a[i]; a[i]:= a[j]; a[j]:= x
  end {exchange };

  procedure qicksort (m, n: integer);
      var k,v: integer;

      function partition (y, z: integer): integer;
          var i, j: integer;
      begin
          …a…
          …v…
          …exchange(i,j) …
      end {partition };

  begin
      …
  end {quicksort };
begin
…
end {sort }.
```





```
program sort (input, output);
  var a: array [0..10] of integer;
      x: integer;

  procedure readarray;
      var i: integer;
  begin … a … end {readarray};

  procedure exchange (i, j: integer);
  begin
      x:= a[i]; a[i]:= a[j]; a[j]:= x
  end {exchange };

  procedure qicksort (m, n: integer);
      var k,v: integer;

      function partition (y, z: integer): integer;
          var i, j: integer;
      begin
          …a…
          …v…
          …exchange(i,j) …
      end {partition };

  begin
      …
  end {quicksort };
begin
…
end {sort }.
```

finale Momentaufnahme



Übersetzung von Prozeduraufrufen

Drei-Adressbefehle

typische Verwendung für
p(x1, ..., xn) im Quelltext
Entsteht durch Compilierung:

```
param x1
param x2
...
param xn
call p, n
```

dazu sind vom Compiler Laufzeitroutinen bereitzustellen:

zum Prozedurruf

- Aufbau von Speicherplatz für das Aktivierungselement
- Verfügbarkeit der Parameter
- Umgebungszeiger für globale Größen
- Retten des Zustandes der rufenden Prozedur
  (so dass auch Rückkehr vollzogen werden kann)

zum Return

- Ergebnisbereitstellung für die Umgebung
- Wiederherstellung des Aktivierungssegmentes
- Sprung zur Rückkehradresse

Übersetzung von Prozeduraufrufen (2)

Beispiel:

r= f (a[i*4])

```
1)   t1:= i * 4
2)   t2:= a [ t1 ]
3)   param t2
4)   t3:= call f, 1
5)   r:= t3
```

Drei-Adresscode