6. Die semantische Analyse

wichtigstes Hilfsmittel: Symboltabelle

- Datenstruktur zur Verwaltung von (Typ-)Informationen über Konstrukte der Quellsprache
- wird von anschließenden Synthesephasen benutzt
- lexikalische Analyse: Eintragen jedes Bezeichners in Symboltabelle
- nachfolgende Phasen: ergänzen Typangabe, Verwendungsangabe, Speicheradresse,…

6. Die semantische Analyse

Operationen auf Symboltabelle:

- insert (s , t)
  s Name des Bezeichners
  t Tokentyp (id für Bezeichner, vordef. für Schlüsselwörter)
  Rückgabe: Eintragsindex in S-Tabelle
- lookup (s)
  s Zeichenkette
  Rückgabe: Index von s, falls s in S-Tabelle, 0 sonst

Schlüsselwörter der Sprache: Bezeichner

⇨ sollten initial in S-Tabelle enthalten sein

⇨ alternativ durch Scanner herausfiltern

6. Die semantische Analyse

Implementierungen für Symboltabellen:

1. als ein Array

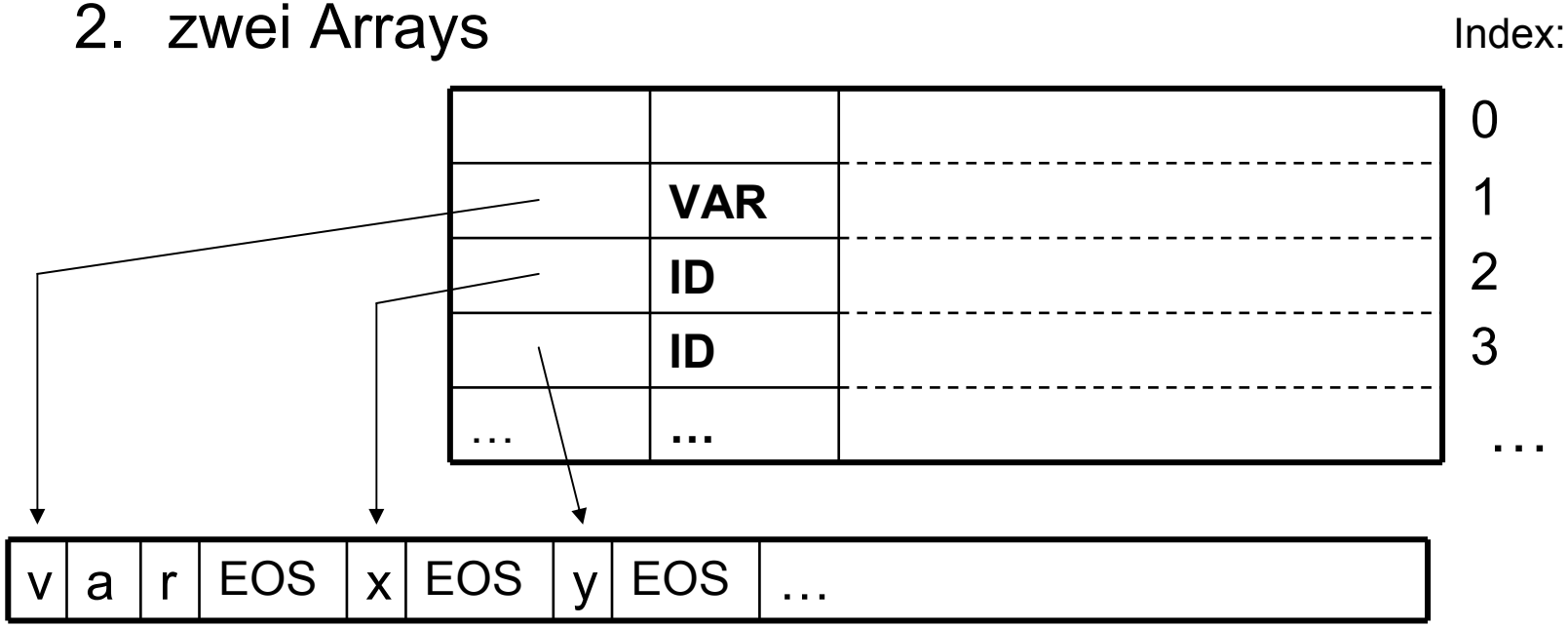
			Index:
			0
„var“	**VAR**		1
„x“	**ID**		2
„y“	**ID**		3
…	**…**		…

- Erster Eintrag leer: Index 0 als Rückgabe, wenn Bezeichner nicht in Tabelle
- Speicherverschwendung in 1. Spalte durch bel. lange Variablennamen

6. Die semantische Analyse

Implementierungen für Symboltabellen:

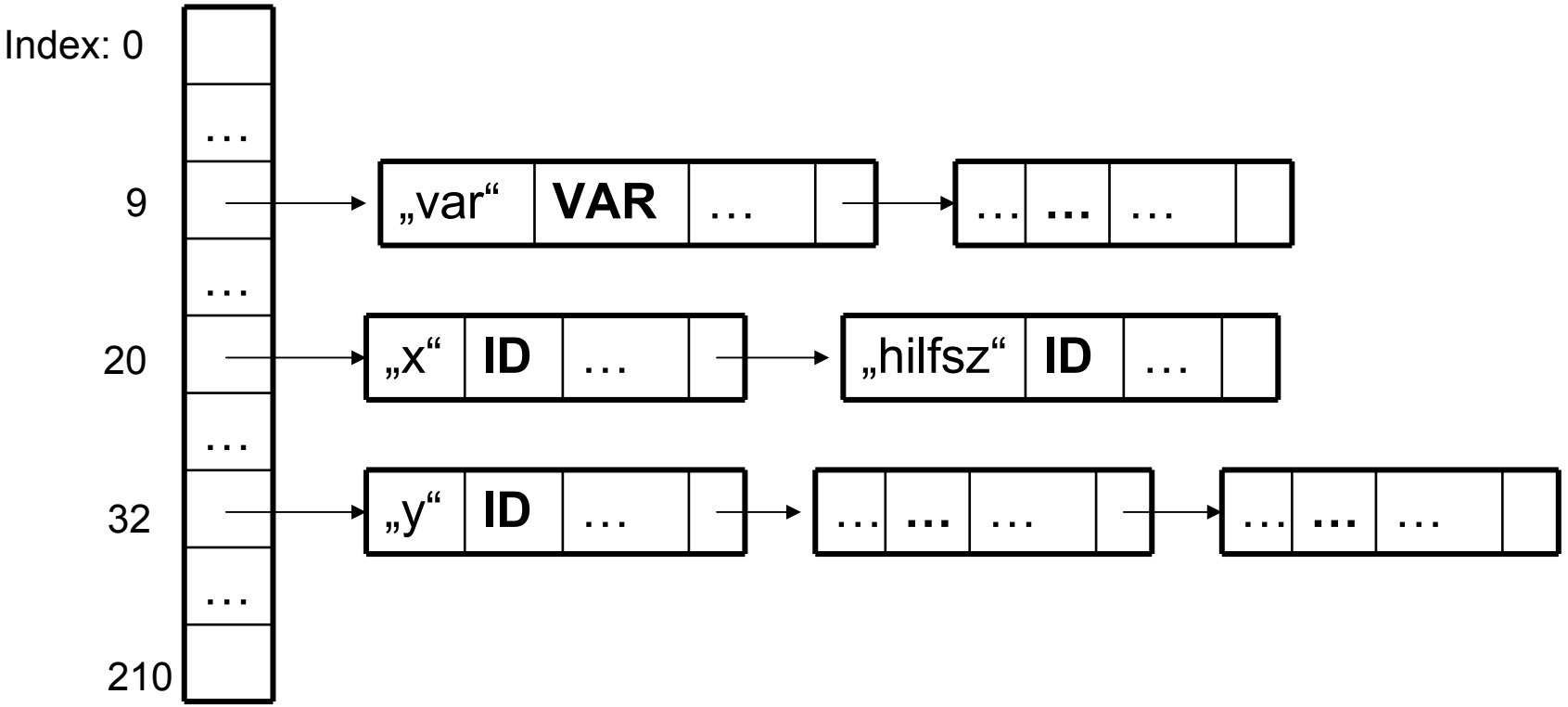
2. zwei Arrays



6. Die semantische Analyse

Implementierungen für Symboltabellen:

3. Hash-Tabelle



6. Die semantische Analyse

Hash-Tabelle:

- festes Array mit m Zeigern auf „Tabelleneinträge“
- Länge der Eintragslisten sollte nicht zu groß sein, m entspr. wählen
- Hashfunktion h berechnet zu s Index in Hashtabelle (0<= h(s) <= m-1)
- oftmals h(s) = (Summe (int (s_i))) mod m, wobei s_i die einzelnen Zeichen von s sind
- m sollte Primzahl sein (gleichmäßigere Verteilung)
- besser: gewichtete Summe: $h'_0 = 0$, $h'_i = \alpha * h'_{i-1} + int(s_i)$, $h' = h'_k$ mit k Länge von s, h(s) = h‘ mod m
- h sollte für oft auftretende Namen (i,j,x,y,…) unterschiedliche Werte liefern

6. Die semantische Analyse

„Erweiterung“ der Scanner-Aufgaben:

- liest Zeichenfolge (Token)
- prüft, ob Token in Symboltabelle (lookup):
- Rückgabewert 0: erzeugt neuen Eintrag in Symboltabelle (insert); liefert Index des Eintrags
- Rückgabewert > 0: erhält Index des Eintrags
- weiter mit Parser wie gehabt

6. Die semantische Analyse

1. Problem:

gleiche Bezeichner für verschiedene Objekte erlaubt

```
int x;
struct x {
   float y;
   float z;
}
```

Lösung:

– Parser bekommt von Scanner nur Tokennamen
– Eintrag in Symboltabelle erst, wenn syntaktische Funktion, die Name einnimmt, erkannt ist
– obiges Bsp: erzeugt 2 Sym.tabelleneinträge (1x Tokentyp id, 1x Tokentyp struct-id)

6. Die semantische Analyse

2. Problem:

gleiche Bezeichner in Gültigkeitsbereichen verschiedener Deklarationen (globales x, lokales x in Prozedur)

Lösung:

- pro Gültigkeitsbereich/Prozedur eigene Symboltabelle
- Suche lokaler Bezeichner in aktueller Sym.tabelle
- Suche globaler Bezeichner in Tabellen der umgebenden Prozeduren (gemäß Bindungsregel der Sprache)

6. Die semantische Analyse

Gültigkeits-Begrenzer :

- Module
- Pakete
- Blöcke
- Prozeduren
- Funktionen

Gültigkeitsbereich eines definierenden

Vorkommens (bzgl. Bezeichner x):

Teil des Programms, in dem sich angewandtes Vorkommen von x auf dieses definierende beziehen kann

6. Die semantische Analyse

Typsysteme, Typausdrücke:

- in Sprachdefinition Zuordnung von Typen zu Sprachkonstrukten
- Typen: *einfache* (atomare, Aufzählungstyp
  Unterbereichstyp)
  zusammengesetzte (Arrays, Records, Sets,
  Zeiger, evtl. Funktionen)
- jeder Ausdruck mit Typ assoziiert
- dieser durch *Typausdruck* gegeben
- Typsystem: Sammlung von Regeln zur Zuweisung von Typausdrücken an die Programmteile

6. Die semantische Analyse

Typausdrücke:

- jeder einfache Typ *boolean, char, integer, real, void*
- spezieller Typausdruck: *type_error*
- mittels Typkonstruktoren:
  - Arrays: `var A:array[1..50] of integer;`
    assoziiert A mit: *array(1..50,integer)*
  - Kartesisches Produkt: T1, T2 Typausdrücke, dann auch T1 x T2
  - Records:
```
type eintrag=record
                wert: integer;
                name: array[1..15] of char
        end;
```
    assoziiert `eintrag` mit dem Typausdruck
    *record ((*wert x *integer)* x *(*name x *array(1..15,char))*

6. Die semantische Analyse

Typausdrücke:

- es gibt benannte Typausdrücke -> Typnamen sind Typausdrücke (s. wert, name bei Records)
- weitere Typkonstruktoren:
  - Zeiger: `var P: ^eintrag;`

    assoziiert `P` mit: *pointer(eintrag)*
  - Funktionen: Abbildung Definitionsbereich in Bildbereich

    `function f (a,b:char):^integer;`

    assoziiert `f` mit *char* x *char* → *pointer(integer)*

Übung: Wozu gehört der folgende Typausdruck?

(integer → integer) → (real → integer)

6. Die semantische Analyse

Typüberprüfer:

- Typsystem gibt Typregeln der Sprache vor
- Typüberprüfer ist Implementierung des Typsystems zur Kontrolle der Regeleinhaltung
- Grammatik eines Programms:

  $P \rightarrow D;E$

  $D \rightarrow D;D$ | **id** : T

  $T \rightarrow$ **char** | **integer** | ***array[n1 .. n2] of*** T | **^**T

  $E \rightarrow$ **literal**| **num**| **id** | E **mod** E| E**[**E**]** | E**^**

6. Die semantische Analyse

Typüberprüfer:

- In Übersetzungsschema: Produktion mit Aktion verknüpft, die Typ „behandelt“:

  $P \rightarrow D;E$

  $D \rightarrow D;D$

  $D \rightarrow$ **id** : T *{insert(id,T.type)}*

  $T \rightarrow$ **char** *{ T.type := char}*

  $T \rightarrow$ **integer** *{ T.type := integer}*

  $T \rightarrow$ **^**T *{ T.type := pointer(T.type)}*

  $T \rightarrow$ **array[n1..n2] of** T *{ T.type := array(n1..n2,T.type)}*

6. Die semantische Analyse

Typüberprüfer:

- bei Ausdrücken:

  $E \rightarrow$ **literal** { *E.type:= char*}

  $E \rightarrow$ **num** { *E.type := integer*}

  $E \rightarrow$ **id** { *E.type := lookup(***id***)*}

  $E \rightarrow$ *E1* **mod** *E2* { *E.type :=* **if** *E1.type = integer* **and** *E2.type=integer* **then** *integer* **else** *type_error*}

  $E \rightarrow$ *E1[E2]* { *E.type :=* **if** *E2.type = integer* **and** *E1.type = array(s,t)* **then** *t* **else** *type_error*}

  $E \rightarrow$ *E1^* { *E.type :=* **if** *E1.type = pointer(t)* **then** *t* **else** *type_error*}

Aufgabe: Wie lautet die Aktion für folgende Statements?

1. S → id := E

2. S → *if E then S1*

6. Die semantische Analyse

Typüberprüfer:

- bei Anweisungen:

$S \rightarrow$ **id** *:=* E { *S.type:=* **if id**.*type* = *E.type* **then** *void* **else** *type_error*}

$S \rightarrow$ **if** E **then** $S1$ { *S.type :=* **if** *E.type* = *boolean* **then** *S1.type* **else** *type_error*}

$S \rightarrow$ **while** E **do** $S1$ {*S.type :=* **if** *E.type* = *boolean* **then** *S1.type* **else** *type_error*}

$S \rightarrow S1$ **;** $S2$ { *S.type :=* **if** *S1.type* = *void* **and** *S2.type=void* **then** *void* **else** *type_error*}

6. Die semantische Analyse

Typüberprüfer:

- bei Funktionen:

  $E \rightarrow E1(E2)$ { *E.type:*= **if** *E2.type* = *s* **and** *E1.type* = *s*→*t* **then** *t* **else** *type_error*}

- Funktionen mit mehr als einem Parameter:

  E2 als Produkttyp *T1 x ….x Tn*

6. Die semantische Analyse

Aufgabe:

1. Schreiben Sie für folgende Typen Typausdrücke:
    1. Ein Array mit Elementen vom Typ Zeiger auf real-Werte, wobei der Array-Index von 1 bis 100 läuft.
    2. Ein zweidimensionales Array mit integer-Werten, dessen Zeilen von 0 bis 9 und dessen Spalten von -10 bis 10 indiziert sind.
    3. Funktionen, deren Wertebereich Funktionen sind, die integer-Werte auf Zeiger auf integer-Werte abbilden, und deren Definitionsbereich Records sind, die aus einem integer-Wert und einem Zeichen bestehen.

6. Die semantische Analyse

Aufgabe:

2. Es sei folgende Grammatik gegeben, die u.a. Listen definiert:

   - $P \rightarrow D;E$
   - $D \rightarrow D;D$ | **id** : T
   - $T \rightarrow$ **char** | **integer** | **list of** T
   - $E \rightarrow$ **literal**| **num**| **id** | (L)
   - $L \rightarrow E,L \mid E$

   Schreiben Sie (nur) die Übersetzungsregeln, um die Typen der Ausdrücke (also E-Regeln) und der Listen (also L-Regeln) und des list of-Typs zu bestimmen.

6. Die semantische Analyse

Typüberprüfer:

- „Problem“:

  in Typprüfungsregeln ständig Typvergleiche
- wann sind 2 Typen gleich?
- ⇨ namentliche vs. strukturelle Gleichheit
- bei einfachen Typen (Grundtypen, Array, Pointer, kart. Produkt, Funktion) Gleichheit immer strukturell: mit gleichem Typkonstruktor gebildet
- ⇨ Test auf strukturelle Gleichheit mittels:

  (nächste Folie)

6. Die semantische Analyse

Typüberprüfer:

function *sgleich(s,t): boolean*;
begin
if s und t vom selben Basistyp **then return true**
else if *s = array(s1,s2)* **and** *t = array(t1,t2)* **then**
return *sgleich(s1,t1)* **and** *sgleich(s2,t2)*
else if *s = s1 x s2* **and** *t = t1 x t2* **then**
return *sgleich(s1,t1)* **and** *sgleich(s2,t2)*
else if *s = pointer(s1)* **and** *t = pointer(t1)*
return *sgleich(s1,t1)*
else if *s = s1→s2* **and** *t = t1→t2* **then**
return *sgleich(s1,t1)* **and** *sgleich(s2,t2)*
else return false
end

6. Die semantische Analyse

Typüberprüfer:

- 1. Problem: Typnamen
- ```
  type verweis = ^zelle;
  var   next: verweis;
        letztes: verweis;
        p: ^zelle;
        q,r: ^zelle;
  ```
- Sind next, letztes, p, q und r typgleich?
- Namensgleichheit: jeder Typname ist eigener Typ

  → Typgleichheit von Typausdrücken nur bei Identität
- Strukturelle Gleichheit: Name wird durch zugehörigen Typausdruck ersetzt

  → zwei Namen-Typausdrücke sind strukturell gleich, wenn sie zwei strukturell gleiche Typausdrücke repräsentieren
  ```

6. Die semantische Analyse

Typüberprüfer:

- Im Beispiel:
- Name: Typausdruck:

Name:	Typausdruck:
`next`	`verweis`
`letztes`	`verweis`
`p`	*pointer(*`zelle`*)*
`q`	*pointer(*`zelle`*)*
`r`	*pointer(*`zelle`*)*

- Bei Namensgleichheit: `next` und `letztes` gleich,
  `p`, `q`, `r` gleich, aber nicht z.B. `next` und `p`
- Bei struktureller Gleichheit: alle gleich, da `verweis` Name für Typausdruck *pointer(*`zelle`*)*

6. Die semantische Analyse

Typüberprüfer:

- 2. Problem: Records mit Zyklen
-
```
type verweis = ^zelle;
     zelle= record
                daten: integer;
                next: verweis
            end;
```

- Setze im Typausdruck des Records für verweis *pointer(*zelle*)*:
  zelle wird assoziiert mit
  *record ((*daten x *integer)* x *(*next x *pointer(zelle)))*

- C z.B. benutzt strukturelle Gleichheit für alle Typen außer für Strukturen (hier Namensgleichheit)

6. Die semantische Analyse

Typüberprüfer:

- weitere Aufgabe: Typumwandlung
- ```
  i: Integer;
  r: Real;
  i := 5; r := i + 2.0;
  ```
- Sprachdefinitionen spezifizieren, welche Umwandlungen zulässig, z.B.
  - bei Zuweisungen Integer in Real, Real in Integer, Char in Integer, Integer in Char, ..
  - bei Real-Operationen Integer in Real
  - Bei Integer-Operationen Char in Integer

- implizite Umwandlung: automatisch intern durch Compiler
- explizite Umwandlung: durch Programmierer angegeben (Casts), werden vom Typprüfer wie Funktionen behandelt
  ```

6. Die semantische Analyse

Typüberprüfer:

- Problem: Überladung von Funktionen/Operatoren
- Bedeutung je nach Kontext unterschiedlich
- Beispiel: Additionsoperator + , unterschiedliche Bedeutung für Integer, Real, komplexe Zahlen, Matrizen,…
- Überladung oft nicht an Ort und Stelle auflösbar
- Abhilfe: Ausdruck in Symboltabelle Menge möglicher Typen zuordnen
- Typfehler: Menge der möglichen Typen ist leer

6. Die semantische Analyse

Fragestellungen zu Kapitel 6:

- Welche Aufgaben hat semantische Analyse?
- Auf welcher Art von Syntaxbaum arbeitet diese Phase?
- Was sind statische semantische Eigenschaften, was dynamische? Beispiele?
- Mögliche Implementierungen einer Symboltabelle?
- Typische Probleme Typeintrag in Symboltabelle
- Typsystem, Typausdrücke, Typüberprüfer
- Typgleichheit: namentlich, strukturell
- Typumwandlung