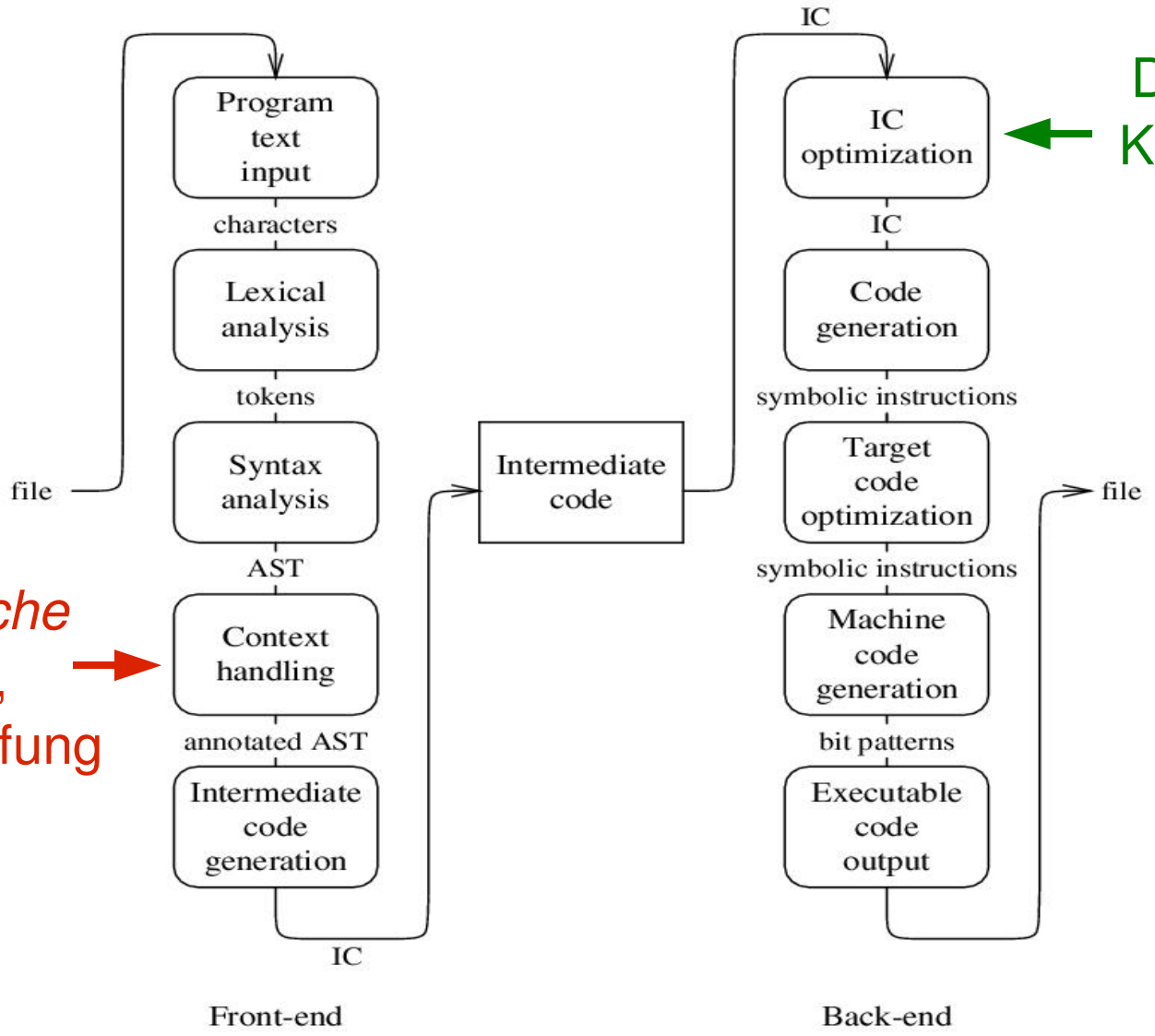
Context-Handling

- befasst sich mit nicht-lokalen Eigenschaften von Programmen
  - Gültigkeitsbereiche von Definitionen
  - Typprüfung/Typinferenz
  - Zusammenhang gegeben durch Bezeichner (für Variablen, Methoden, ...)
- Paradigma-unabhängige Spracherkennung (SIPS I):
  - diverse Arten von Attributberechnungen auf dem abstrakten Syntaxbaum (AST)
- Besonderheiten imperativer Sprachen (SIPS II)
  - Identifikation von Bezeichnern, finde zu jeder Anwendungsstelle die entsprechende Definitionsstelle
  - Typprüfung

Stellen nicht-lokaler Programm-Analyse



Semantische Analyse, z.B. Typprüfung

Datenfluss-/ Kontrollfluss-Analyse

Figure 1.21 Structure of a compiler.

Bezeichner-Erkennung (Identification)

- Problem: welches Objekt verbirgt sich hinter einem Namen?
- Komplikation: mit mehreren Definitionen belegte Namen
  - in verschiedenen Scopes (Disambiguierung oder Verschattung)
  - für verschiedene Umgebungen (Overloading)
- Namensdefinition: (*defining occurrence*)
  - i.d.R. eindeutig (in festem Scope/Kontext)
  - muss im Quelltext nicht existieren (interner Name)
- Namensverwendung: (*applied occurrence*)
  - folgt i.d.R. der Namensdefinition (sonst *Vorwärtsreferenz*)
- Namensbereiche: Beispiel C (*name spaces*)
  - Namen für `enum`-, `struct`- und `union`-Typen
  - Labels
  - Bezeichner für Variablen, Routinen, Typen, Aufzählungstypen

Symboltabelle (SymT)

- Datenstruktur zur Verwaltung von Informationen über Bezeichner
- Modifikation der Programmrepräsentation
  - Eingabe: abstrakter Syntaxbaum (AST), oft auch schon SymT
  - Ausgabe: annotierter AST, zusätzliche Einträge in SymT
- Aufbau der SymT:
  - Schlüssel: Name von Variablen, Routinen (Methoden), ...
  - eingetragener Wert: Typ, Liste von Unterstrukturen, ...
- Implementierung:
  - schneller Zugriff: Hashtabelle, Name: Zahlcode statt String
  - Scope-Stack/Tabelle zur Verwaltung geschachtelter Blöcke

Quellprogramm / Ausschnitt Symboltabelle

```
void rotate(double angle) {
    ...
}
void paint(int left, int right) {
    Shade matt, signal;
    ...
    {   Counter right, wrong;
        ...
    }
}
```

Figure 6.3 C code leading to the symbol table in Figure 6.2.

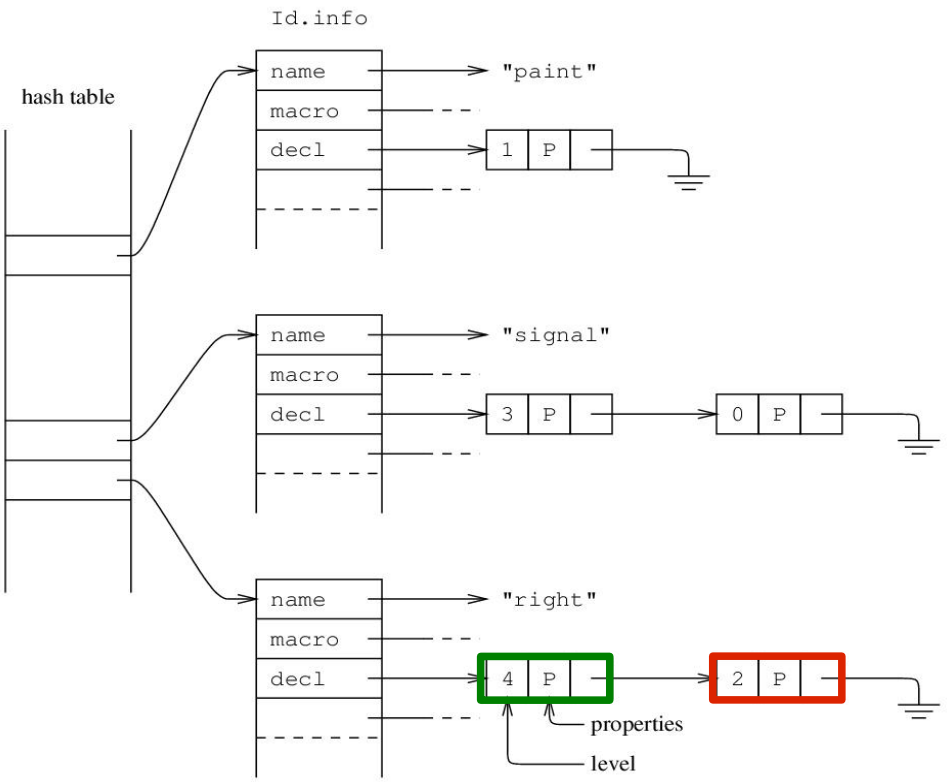


Figure 6.4 A hash-table based symbol table.

Scope (1)

- Idee: Organisation der Namen mit Scopestruktur
- naive Implementierung:
  - Codebeispiel (Abb. 6.3)
  - Symboltabelle nach Scope organisiert (Abb. 6.2)
  - ein globaler Namensbereich (mit Verschattung)
  - so kein effizienter Zugriff auf Namen **(!)**

```
void rotate(double angle) {
    ...
}

void paint(int left, int right) {
    Shade matt, signal;

    ...
    {   Counter right, wrong;
        ...
    }
}
```

Figure 6.3 C code leading to the symbol table in Figure 6.2.

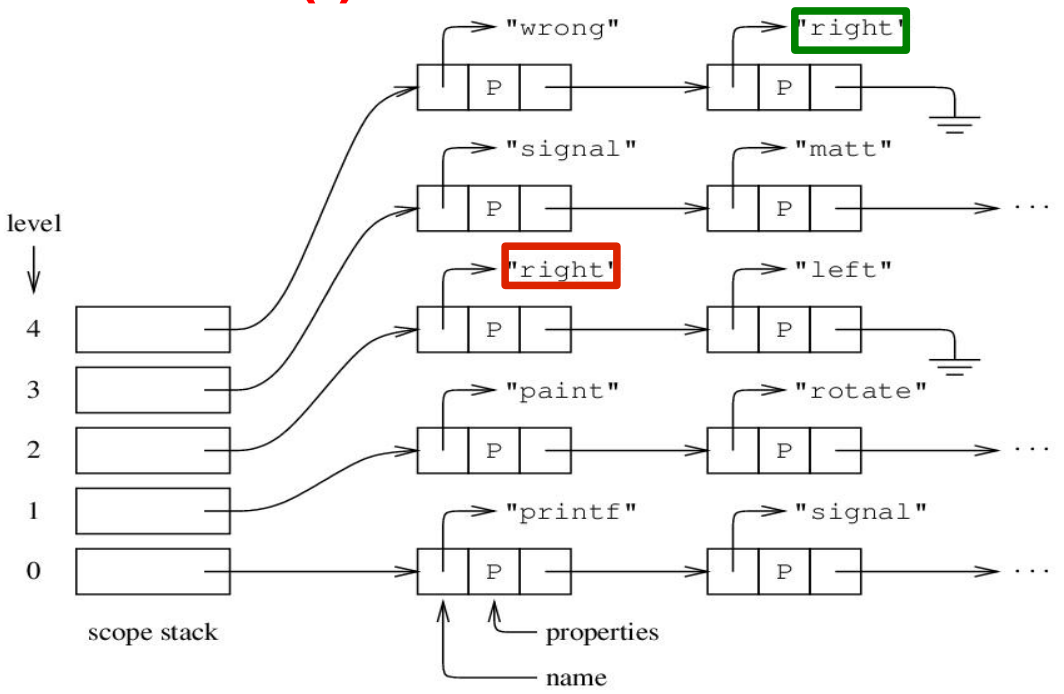


Figure 6.2 A naive scope-structured symbol table.

Scope (2)

- Verbesserung: für C-ähnliche Sprachen
  - Symboltabelle mit Hash-Zugriff und pro Name
    1. Tabelleneintrag (`id.info`)
    2. Stack von separaten Namensbereichen, in denen er auftritt
  - Scopestack (Abb. 6.5)
    - zur Einführung und Entfernung von Scopes in der Symboltabelle
    - ähnlich Abb. 6.2, verweist jetzt aber auf die Einträge in der Symboltabelle

Scope (3)

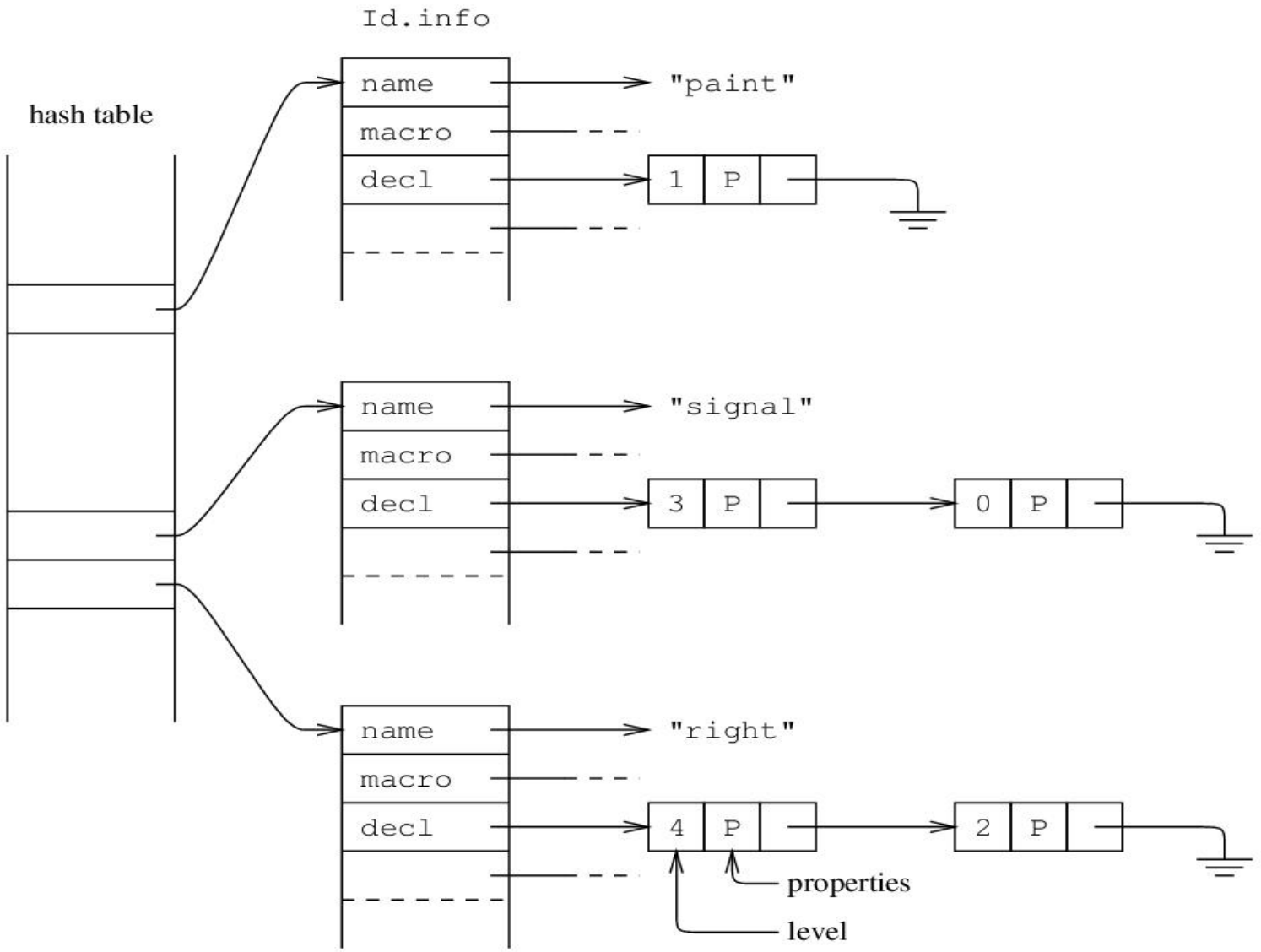


Figure 6.4 A hash-table based symbol table.

Scope (4)

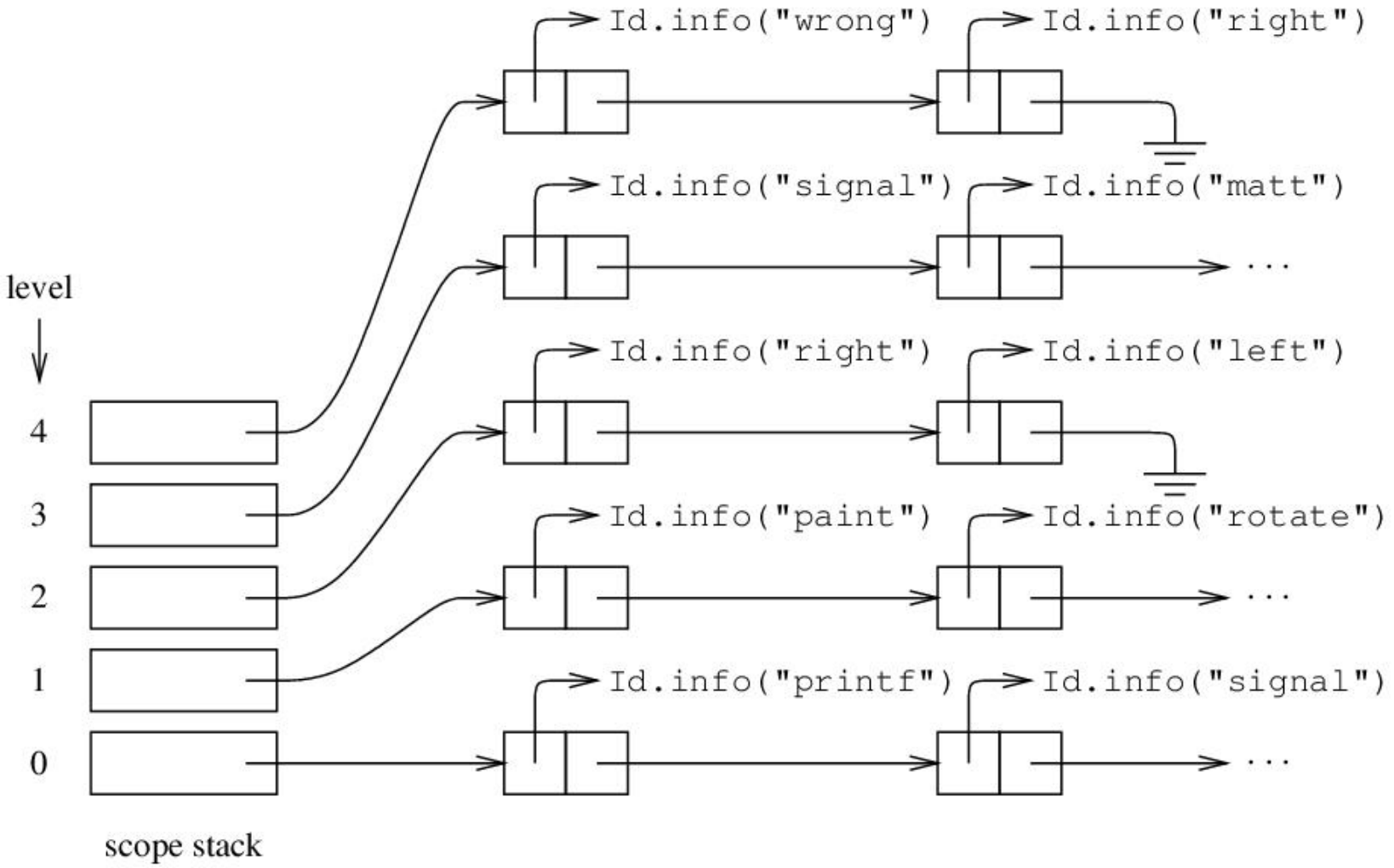


Figure 6.5 Scope table for the hash-table based symbol table.

Scope (5)

- Entfernung des obersten Namensbereichs:

```
PROCEDURE Remove topmost scope ():
    SET Link pointer TO Scope stack [Top level];
    WHILE Link pointer /= No link:
        // Get the next Identification info record
        SET Idf pointer TO Link pointer .idf_info;
        SET Link pointer TO Link pointer .next;
        // Get its first Declaration info record
        SET Declaration pointer TO Idf pointer .decl;
        // Now Declaration pointer .level = Top level
        // Detach the first Declaration info record
        SET Idf pointer .decl TO Declaration pointer .next;
        Free the record pointed at by Declaration pointer;
    Free Scope stack [Top level];
    SET Top level TO Top level - 1;
```

Figure 6.6 Outline code for removing declarations at scope exit.

Scope (6)

- Namensbereiche für Komponentennamen: z.B. `idf.sel`
  - suche im Namensbereich für Variablen nach `idf`
  - in der `Id.info` für `idf`: Typname *T*
    (*T* kann in einem älteren Scope definiert sein!)
  - in der `Id.info` für *T*: existiert eine Komponente `sel` ?
    falls ja, greife auf sie zu, sonst Fehlermeldung

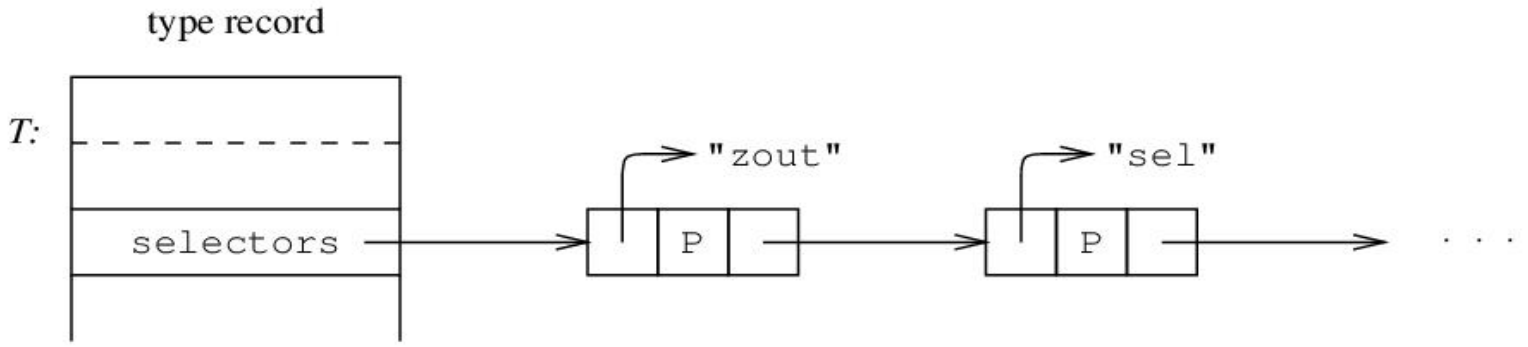


Figure 6.7 Finding the type of a selector.

Overloading (1)

- Problem: unterschiedliche Definition eines Namens in unterschiedlichen Kontexten
  - Unterprogrammdefinition in Abh. vom Parametertyp (Ada, Java)
  - Def. arithmetischer Operationen in Abh. vom Operandentyp (in den meisten Sprachen)
- Aufgaben:
  1. Sammlung aller möglichen Definitionen
  2. Auswahl der im Aufrufkontext passenden Definition(en)
- Komplikationen:
  - bleibt genau eine Definition übrig?
  - der aktuelle Parameter kann wieder einen überladenen Namen enthalten

Overloading (2)

- Unterprogrammaufruf: `PUT("Hello")` (in Ada)
- Scopestack: (P*n* = Properties-Eintrag in Symboltabelle)
  - Level 1:

```
procedure PUT(STRING) = P64
procedure PUT(MATRIX) = P57
```

  - Level 0:

```
procedure PUT(INTEGER)= P48
procedure PUT(STRING) = P33
procedure PUT(FILE_TYPE, INTEGER)= P27
procedure PUT(FILE_TYPE, STRING) = P14
```

- mögliche Definitionen: (`P33` verschattet!)

```
procedure PUT(STRING) = P64
procedure PUT(MATRIX) = P57
procedure PUT(INTEGER)= P48
procedure PUT(FILE_TYPE, INTEGER)= P27
procedure PUT(FILE_TYPE, STRING) = P14
```

- mögliche Definitionen: (Pattern Match)

```
procedure PUT(STRING) = P64
```

Overloading (3a)

- Beispiel: in Ada
  - Standarddefinition in der Sprache

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

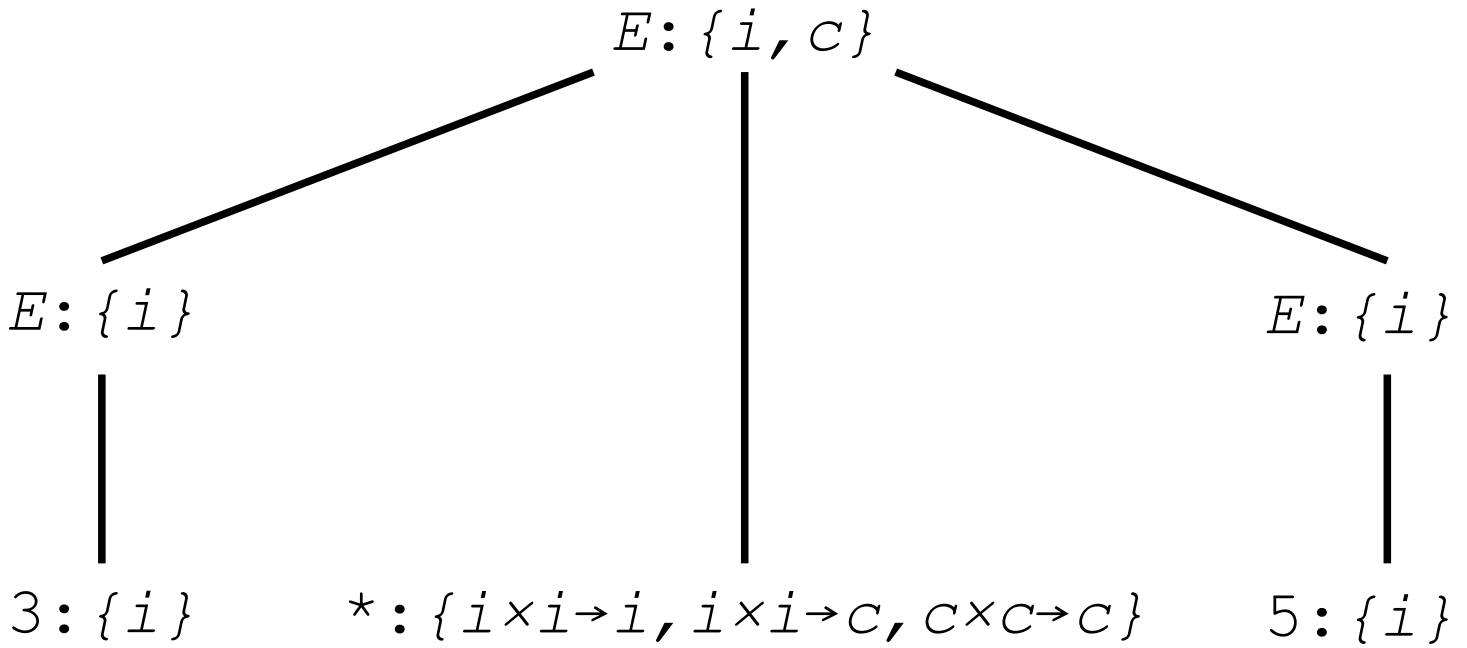
  - Annahme: weitere Definitionen vom Benutzer

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

  - sei `x:integer` und `z:complex`, was ist der Typ von `3*5` ?
    - in `x*(3*5)` `integer`
    - in `z*(3*5)` `complex`

Overloading (3b)

- Typgraph für `3*5`: (aus Drachenbuch, Abb. 6.11)



Importierter Scope

- Feature: benenne eine Komponente ohne Angabe des Strukturnamens
- Implementierung:
  - öffne eine neue Ebene auf dem Scopestack
    - C++: Scopeauflösungsoperator (`x::...`)
    - Pascal, Modula-2: `WITH`-Statement (`WITH x DO ...`)
  - füge die deklarierten namen zur obersten Ebene hinzu:

    (a) Importdeklaration in:
    - Modula-2: *`IMPORT`*-Liste (`FROM module IMPORT ...`)
    - Ada: `USE`-Klausel (`use` package)

    (b) Auflösung von Namenskonflikten:
    - Modula-2: explizit durch Auflistung der importierten Namen
    - Ada: automatisch durch Überladen oder Verschattung

Geschichte getypter Sprachen (1)

- Fortran: (1954-57) (John Backus)
  - gedacht zur Auswertung arithmetischer Formeln
  - `integer`, `real`, `complex`, `array`, `common block`
  - falls undeklariert:
    - `integer`, wenn der Name mit `I` bis `N` beginnt
    - `real` sonst
  - `DO 100 I = 1.10` (Tippfehler: Punkt statt Komma)
- LISP: (1956-58) (John McCarthy)
  - gedacht für symbolisches Rechnen (z.B. Differenzierung)
  - dynamisch getypt
  - erste Sprache mit Heap und automatischer Heapbereinigung

Geschichte getypter Sprachen (2)

- Algol: (1958-59) (internationales Komitee)
  - dynamische Arrays
  - explizite Deklaration
- Algol W: (Niklaus Wirth)
  - `record`, `pointer` (typgebunden)

```
record node (int data, ref(node) link);
ref node pointer;
```

  - keine Heapverwaltung

Geschichte getypter Sprachen (3)

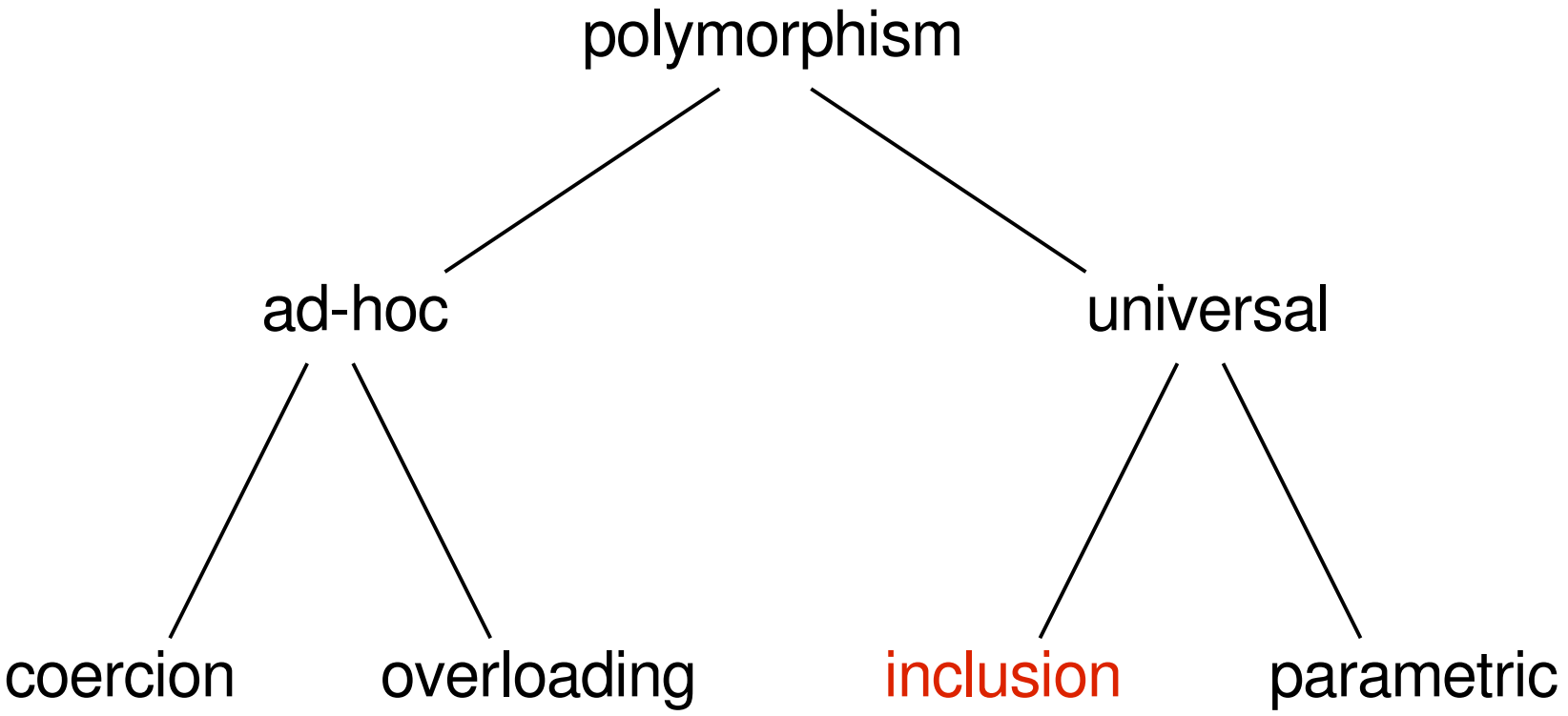
- Simula: (1967) (Ole-Johan Dahl, Kristen Nygaard)
  - Klassenkonzept (abstrakte Datentypen)
  - Objekt-orientierte Sprache zur Simulation
- Pascal: (1971) (Niklaus Wirth)
  - allgemeines Konzept von Typdefinitionen durch den Programmierer
- Concurrent Pascal: (1975) (Per Brinch-Hansen)
  - Monitor-Konzept als Abstraktion von Semaphoren
    - gemeinsame Datenstruktur für nebenläufige Prozesse
    - automatische Synchronisation, Warteschlange
  - Sprache zum strukturierten Betriebssystementwurf

Geschichte getypter Sprachen (4)

- ML: (1983) (Robin Milner)
  - funktionale Sprache (Vorgänger von Haskell und Ocaml)
  - zunächst nur gedacht zur Steuerung eines Theorembeweisers
  - Typsicherheit auch bei Exceptions
  - automatische Typinferenz (Hindley-Milner-Typsystem)
- Java: (1995) (James Gosling)
  - Objekt-orientierte Sprache ohne Mehrfachvererbung
  - Typsicherheit, keine Pointerarithmetik, `unions` ...
    (im Gegensatz zu C oder C++)
  - Typcasts nur entlang der Vererbungshierarchie
  - Verwendung von inclusion-polymorphism (Cardelli-Typsystem)

Polymorphie

Cardelli und Wegner [1985] verfeinerten den Polymorphiebegriff von Strachey für Vererbung



Typüberprüfung

- Namensattribute

  (a) Art: Konstante, Variable, Unterprogramm, Typ, Adresse, ...

  (b) Typ: repräsentiert Wertemenge und assoziierte Operationen

- Erscheinung: (im Quellprogramm)
  - benannt: in einer Typdefinition:

```
TYPE Int_Array = ARRAY [1..10] OF Integer;
```

  - unbenannt: Typausdruck in einer Deklaration:

```
VAR a: ARRAY [1..10] OF Integer;
```

    → wird mit internem Namen versehen:

```
TYPE T01_line35 = ARRAY [1..10] OF Integer;
VAR a: T01_line35;
```

Vorwärtsreferenzen

- kompliziert Wartung der Symboltabelle
- keine offenen Vorwärtsreferenzen am Ende eines Scopes!
- öffnet die Tür für zirkuläre Definitionen:

```
TYPE Ptr_List_Entry = POINTER TO List_Entry;
TYPE List_Entry =
   RECORD
      Element: Integer;
      Next: Ptr_List_Entry;
   END RECORD;
```

Typtabelle (1)

- Eintrag: (in Beispielen Vorkommende in rot)
  - Typkonstruktor: `basic`, `record`, `array`, `pointer`,... plus zusätzliche Informationen:
    - `basic`: tatsächlicher Typ (`integer`, `real`, `char`, `bool`, ...)
    - `record`: Komponentennamen und -typen
    - `array`: Dimensionalität, Indextyp und Elementtyp
    - `pointer`: Referenztyp
  - Größe und Alignmentbeschränkungen für Variablen des Typs
  - Typen der Komponenten (falls vorhanden)

Typtabelle (2)

- Indizierung: per Integer statt per Namen
  (wie bei optimierter Symboltabelle)
- Probleme:
  - Behandlung von Vorwärtsreferenzen:
    - repräsentiere sie beim Tabellenaufbau durch den Namen
    - ersetze sie nach Ende des Aufbaus durch die Typinformation
  - Mehrfachauftreten eines Typs: normalisiere die Einträge
    (so dass sie nicht wiederholt auf Typtabelleneinträge zeigen)
  - Finden von Zyklen (Abb. 6.8)

Finden zyklischer Typdefinitionen

Data definitions:

1. Let T be a type table that has entries containing either a type description or a reference (`TYPE`) to a type table entry.
2. Let *Cyclic* be a set of type table entries.

Initializations:

Initialize the *Cyclic* set to empty.

Inference rules:

If there exists a `TYPE` type table entry t_1 in T and t_1 is not a member of *Cyclic*, let t_2 be the type table entry referred to by t_1.

1. If t_2 is t_1 then add t_1 to *Cyclic*.
2. If t_2 is a member of *Cyclic* then add t_1 to *Cyclic*.
3. If t_2 is again a `TYPE` type table entry, replace, in t_1, the reference to t_2 with the reference referred to by t_2.

Figure 6.8 Closure algorithm for detecting cycles in type definitions.

Konstruktion der Typtabelle (1)

Typdeklarationen

```
TYPE a = b;
TYPE b = POINTER TO a;
TYPE c = d;
TYPE d = c;
```

Ausgangspunkt: vordefinierter Typ `INTEGER:`

Typtabelle

```
TYPE 0: INTEGER;
```

Symboltabelle

```
"integer": TYPE 0
```

Konstruktion der Typtabelle (2)

Typdeklarationen

```
TYPE a = b;
TYPE b = POINTER TO a;
TYPE c = d;
TYPE d = c;
```

1. Typdeklaration

Typtabelle

```
TYPE 0: INTEGER;
TYPE 1: ID_REF "b";
```

Symboltabelle

```
"integer": TYPE 0
"a"      : TYPE 1
"b"      : UNDEFINED
```

Konstruktion der Typtabelle (3)

Typdeklarationen

```
TYPE a = b;
TYPE b = POINTER TO a;
TYPE c = d;
TYPE d = c;
```

2.Typdeklaration

Typtabelle

```
TYPE 0: INTEGER;
TYPE 1: ID_REF "b";
TYPE 2: ID_REF "a";
TYPE 3: POINTER TO TYPE 2;
```

Symboltabelle

```
"integer": TYPE 0
"a"      : TYPE 1
"b"      : TYPE 3
```

Konstruktion der Typtabelle (4)

Typdeklarationen

```
TYPE a = b;
TYPE b = POINTER TO a;
TYPE c = d;
TYPE d = c;
```

3.+4. Typdeklaration

Typtabelle

```
TYPE 0: INTEGER;
TYPE 1: ID_REF "b";
TYPE 2: ID_REF "a";
TYPE 3: POINTER TO TYPE 2;
TYPE 4: ID_REF "d";
TYPE 5: ID_REF "c";
```

Symboltabelle

```
"integer": TYPE 0
"a"      : TYPE 1
"b"      : TYPE 3
"c"      : TYPE 4
"d"      : TYPE 5
```

Konstruktion der Typtabelle (5)

Typdeklarationen

```
TYPE a = b;
TYPE b = POINTER TO a;
TYPE c = d;
TYPE d = c;
```

ersetze in der Typtabelle Referenzen durch Nummern:

Typtabelle

```
TYPE 0: INTEGER;
TYPE 1: TYPE 3;
TYPE 2: TYPE 1;
TYPE 3: POINTER TO TYPE 2;
TYPE 4: TYPE 5;
TYPE 5: TYPE 4;
```

Symboltabelle

```
"integer": TYPE 0
"a"      : TYPE 1
"b"      : TYPE 3
"c"      : TYPE 4
"d"      : TYPE 5
```

Konstruktion der Typtabelle (6)

Typdeklarationen

```
TYPE a = b;
TYPE b = POINTER TO a;
TYPE c = d;
TYPE d = c;
```

Zyklenerkennung, Ergebnis: cyclic_set={TYPE 4, TYPE 5}

Typtabelle

```
TYPE 0: INTEGER;
TYPE 1: TYPE 3;
TYPE 2: TYPE 3;
TYPE 3: POINTER TO TYPE 2;
TYPE 4: TYPE 4;
TYPE 5: TYPE 4;
```

Symboltabelle

```
"integer": TYPE 0
"a"      : TYPE 1
"b"      : TYPE 3
"c"      : TYPE 4
"d"      : TYPE 5
```

Konstruktion der Typtabelle (7)

Typdeklarationen

```
TYPE a = b;
TYPE b = POINTER TO a;
TYPE c = d;
TYPE d = c;
```

Entfernung von Indirektionen, Fehlertyp bei Zyklen

Typtabelle

```
TYPE 0: INTEGER;
TYPE 1: TYPE 3;
TYPE 2: TYPE 3;
TYPE 3: POINTER TO TYPE 2;
TYPE 4: ERRONEOUS_TYPE;
TYPE 5: ERRONEOUS_TYPE;
```

Symboltabelle

```
"integer": TYPE 0
"a"      : TYPE 3
"b"      : TYPE 3
"c"      : TYPE 4
"d"      : TYPE 5
```