

Grundlagen des Übersetzerbaus (3)

Prof. Dr. Michael Philippsen



Modulare Struktur von Übersetzern

Analyse

Token

Syntax

Semantik

Abbildung

Transf.

Optim.

Codierung

Code-Erz.

Ass./Bind.

Quelltext

Token-
folge

*Struktur-
baum*

attrib.
Struktur-
baum

Zwischen-
Code

Ziel-
programm

Namens-
tabelle

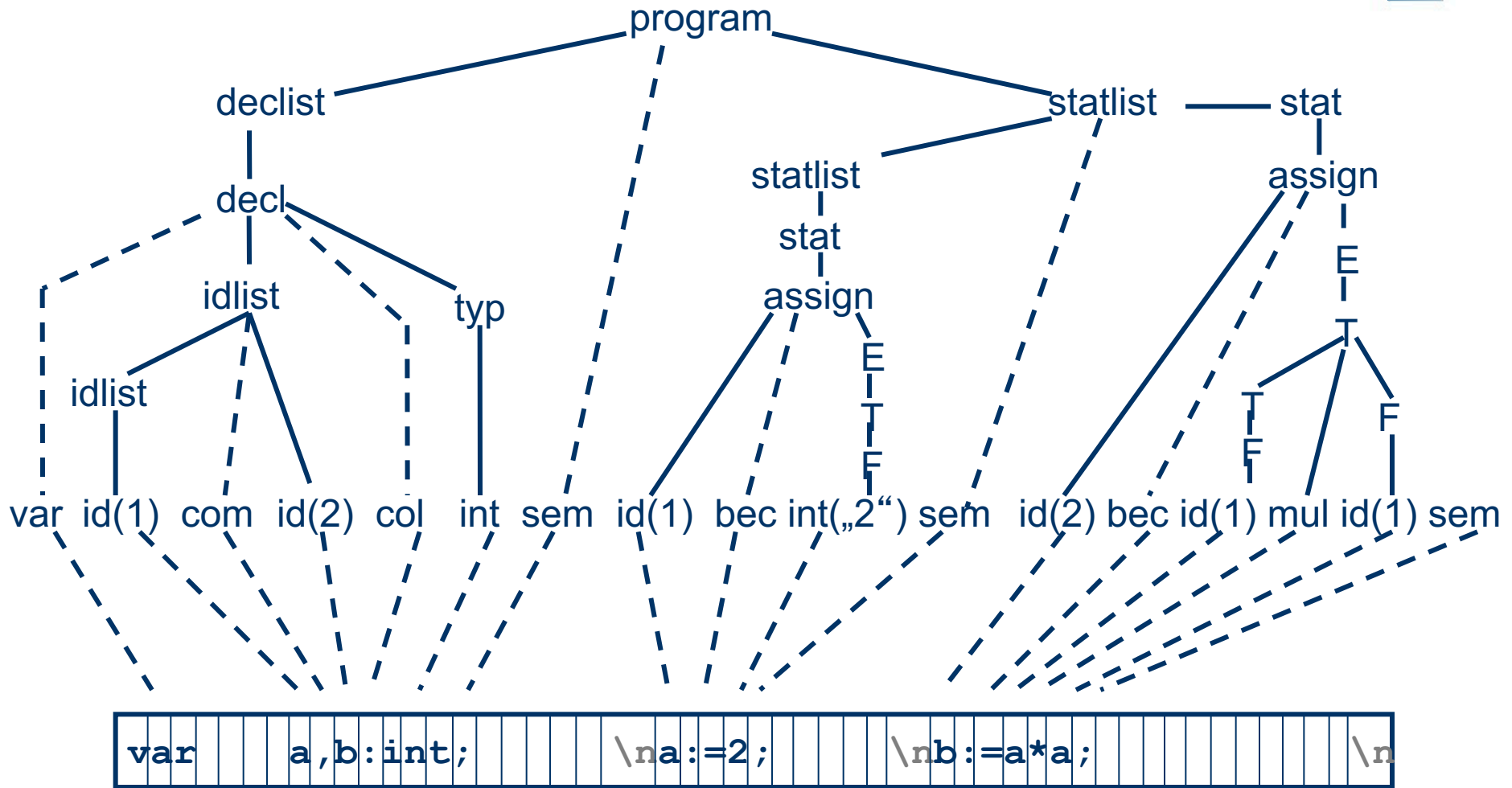
Definitions-
tabelle

- Aufgaben:
 - Feststellung der bedeutungstragenden Elemente
 - Zuordnung statischer Bedeutung
 - Konsistenzprüfung
- Schritte:
 - Lexikalische Analyse/Abtastung (Scanner)
→ Syntaktische Analyse/Zerteilung (Parser)
 - Semantische Analyse
- Noch keine Übersetzung, nur Analyse

n-Durchgangsübersetzer („n pass compiler“)

- 1-Durchgangsübersetzer:
 - Schnell und speicherplatzeffizient: Frühere Übersetzer arbeiteten auf langsamen Maschinen und verfügten über wenig Speicher.
 - Als Parser-Nebenprodukt wurde sofort Maschinen-Code erzeugt.
 - Kaum globale Optimierungen möglich.
 - Kein modularer Entwurf. Parser, semantische Analyse, Code-Selektion, ... alles in einem Programmteil zusammengemischt.
 - Keine Vorausschau möglich. FORWARD-Deklarationen.
- n-Durchgangsübersetzer ($n > 1$, 5-15 durchaus üblich):
 - Datenstruktur hebt Zwischenergebnisse der einzelnen (modularen) Übersetzerphasen auf.
 - Umfasst mehrfaches Traversieren von diversen Zwischenprodukten der Übersetzung.
 - Vorausschau möglich: Einfluss auf Programmiersprachen.

Zerteilungsbeispiel: konkrete Syntax

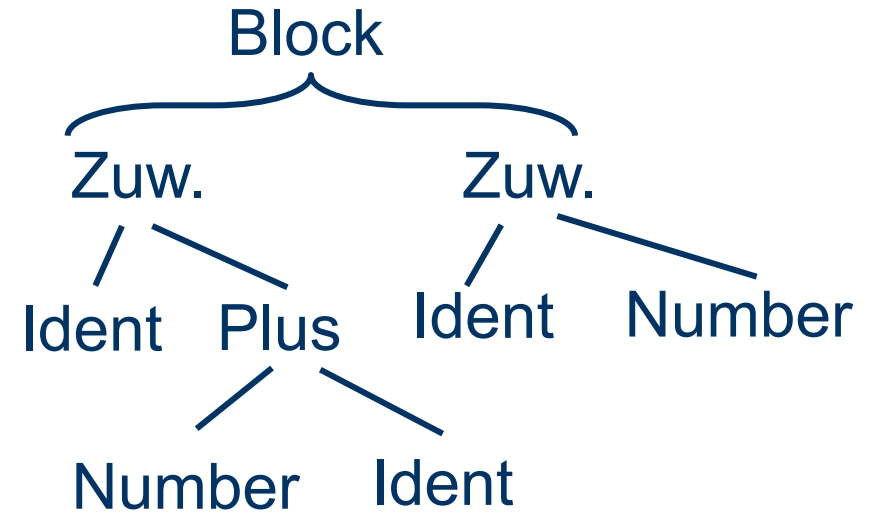
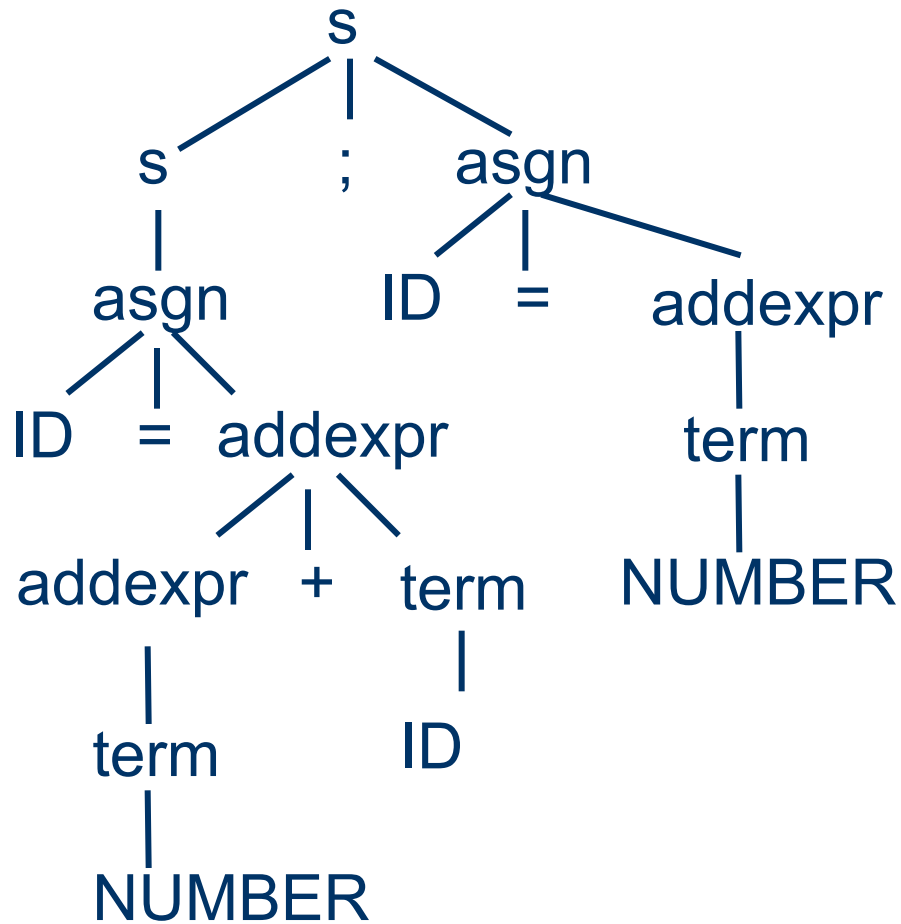


Nachteile des konkreten Syntaxbaums

- „Satzzeichen“ wie ; , ()
 - benötigt zum Parsen des Eingabetextes,
 - tragen keine Bedeutung.
- Die Information, die die Satzzeichen tragen, enthält nach dem Parsen die Struktur des Syntaxbaums.
- Struktur des Syntaxbaums
 - Bestimmt durch die konkret zum Parsen verwendete Grammatik (sog. konkrete Syntax).
 - Grammatik-Transformationen (z.B. Entfernung von Links-Rekursionen, Entfernung von Mehrdeutigkeit, ...) fügen oft zusätzliche Nicht-Terminalsymbole aus technischen Gründen ein und ändern dadurch den resultierenden konkr. Syntaxbaum.
- Abstrakte Syntax als Lösung.



Konkreter Syntaxbaum versus Abstrakter Syntaxbaum



- Prinzip der abstrakten Syntax: nur die bedeutungstragende Struktur wird bewahrt.
- Endprodukt des Parsens ist **Abstrakter Syntaxbaum (AST)**.
- Abstrakter Syntaxbaum heißt oft nur **Strukturbaum**.
- Spätere Schritte des Übersetzers traversieren den Strukturbaum und berechnen Informationen (sog. Attribute) für die Knoten des Strukturbaums.

Man spricht dann vom **attributierten Strukturbaum**.

Mögliche Attribute für AST-Knoten:

- (Literal-Werte, Positionsangabe; bereits im Lexer gesetzt.)
- Zeiger auf Definitionstabelle: Bedeutung von Namen.
- Zeiger auf Typstruktur für Typüberprüfung (siehe später).

Erzeugung des Abstrakten Syntaxbaums

- Zwei denkbare Vorgehensweisen:
 1. Beim Anwenden einer Produktion werden als Nebenprodukt die entsprechenden AST-Knoten erzeugt und verkettet.
 - Übliches Vorgehen im kommerziellen Übersetzerbau
 - Vorteil: Speicher- und Laufzeiteffizienz.
 - Nachteil: Eingabe des Parser-Generators ist Mischung von Grammatik und Spezifikation der Aktionen zur AST-Erstellung.
→ Ansatz im Übungsbetrieb
 2. Der Parser erzeugt erst den konkreten Syntaxbaum; dieser wird dann in den abstrakten Syntaxbaum transformiert.
 - Vorteil:
 - Grammatik ist frei von Aktionsspezifikationen und daher auch für andere Zwecke/Werkzeuge nutzbar.
 - Aktionen nicht über die Grammatik verstreut.
 - Nachteil: zusätzlicher Durchlauf

AST von javac (1)

```
public abstract class JCTree {  
    public int pos;   
    public Type type;  
    public abstract Tag getTag();  
    ...  
}
```

Jeder Knoten geht auf eine Programmstelle zurück.

Jeder Knoten hat einen Typ.

// z.B. **IMPORT**,
CLASSDEF,
METHODDEF, ...

Jede Blattklasse erhält eindeutigen Tag-Wert.
switch/if über **Tag**-Wert ist möglich.

AST von javac (2)

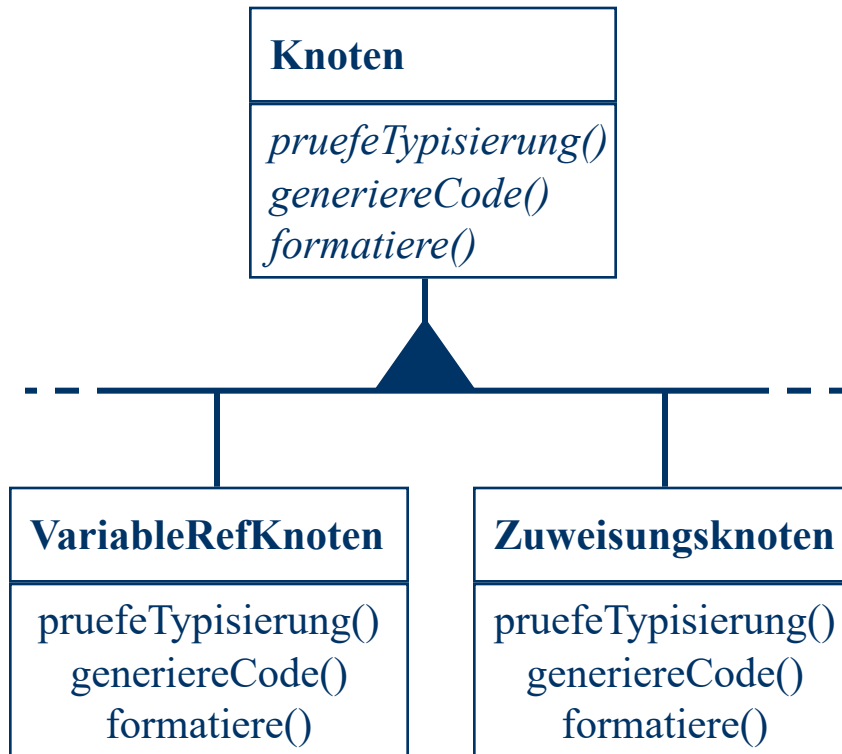
```
public static abstract class JCEXpression
    extends JCTree { ... }
```

```
public static class JCIIdent extends JCEXpression {
    public Name name; ←————— Kommt aus Namenstabelle.
    public Symbol sym; ←————— Kommt aus Definitionstabelle.
}
```

```
public static class JCAssignOp extends JCEXpression {
    public JCEXpression lhs;
    public JCEXpression rhs;
    public OperatorSymbol operator;
}
```

Exkurs: Entwurfsmuster „Besucher“ (1)

Beispiel: Übersetzer *ohne* Besucher



Operationen auf AST:

- voneinander unabhängig
- jeweils Traversierung
- Code über Klassen verstreut

Konsequenzen:

- schlecht für Team-Entwicklung
- Traversierung x-fach impl.
- jede neue Operation erzwingt Erweiterung aller Klassen.

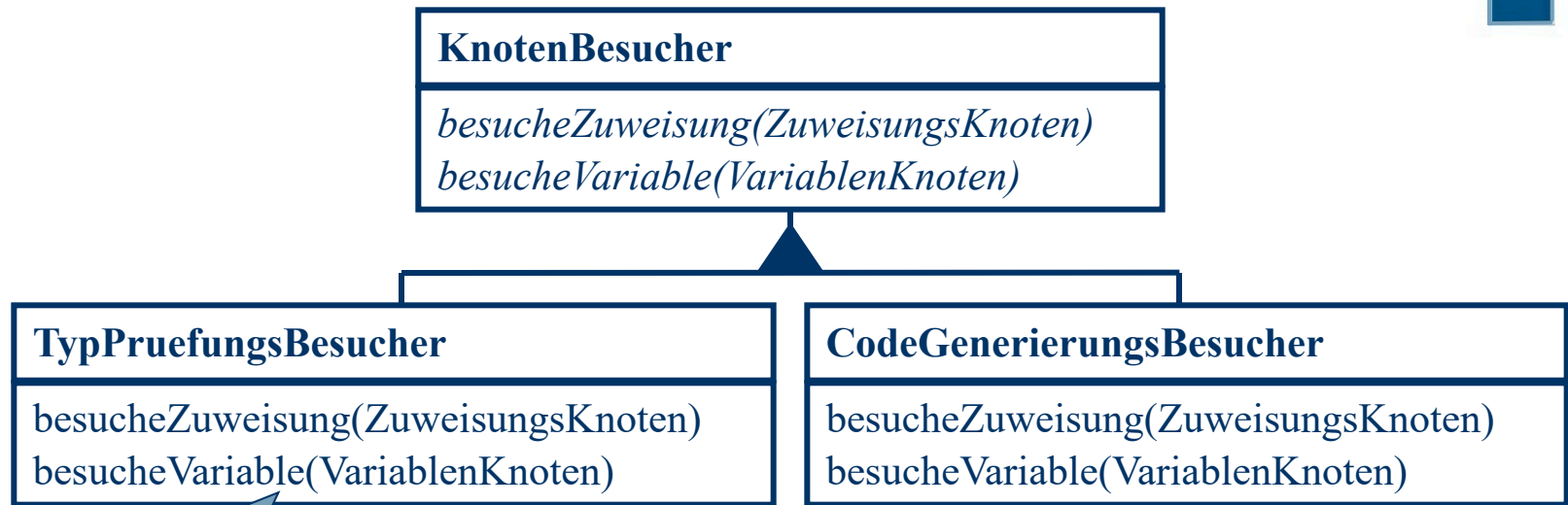
Zweck

Kapsle eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt.

Das Besuchermuster ermöglicht die Definition einer neuen Operation, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.

Exkurs: Entwurfsmuster „Besucher“ (3)

Beispiel: Übersetzer *mit* Besucher



Eine Besuchs-Methode pro
AST-Blatt-Klasse. Die
Methoden rufen sich
gegenseitig rekursiv auf.

Ein Besucher pro
Anwendungszweck.

Exkurs: Entwurfsmuster „Besucher“ (4)

- Die aufzurufende Besuchs-Methode hängt ab
 - vom Besucher (Laufzeit-Typ des Besucher-Objekts) *und*
 - vom zu besuchenden Knoten (Laufzeit-Typ des Knoten).
- Problem: Viele Programmiersprachen bieten nur Polymorphismus bzgl. *eines* Typs.

```
public static class JCAssignOp extends ... {  
    public JCEExpression lhs;  
    public JCEExpression rhs;  
    public OperatorSymbol operator;  
}  
//reicht nicht:  
lhs.visit(b);  
b.visit(lhs);
```

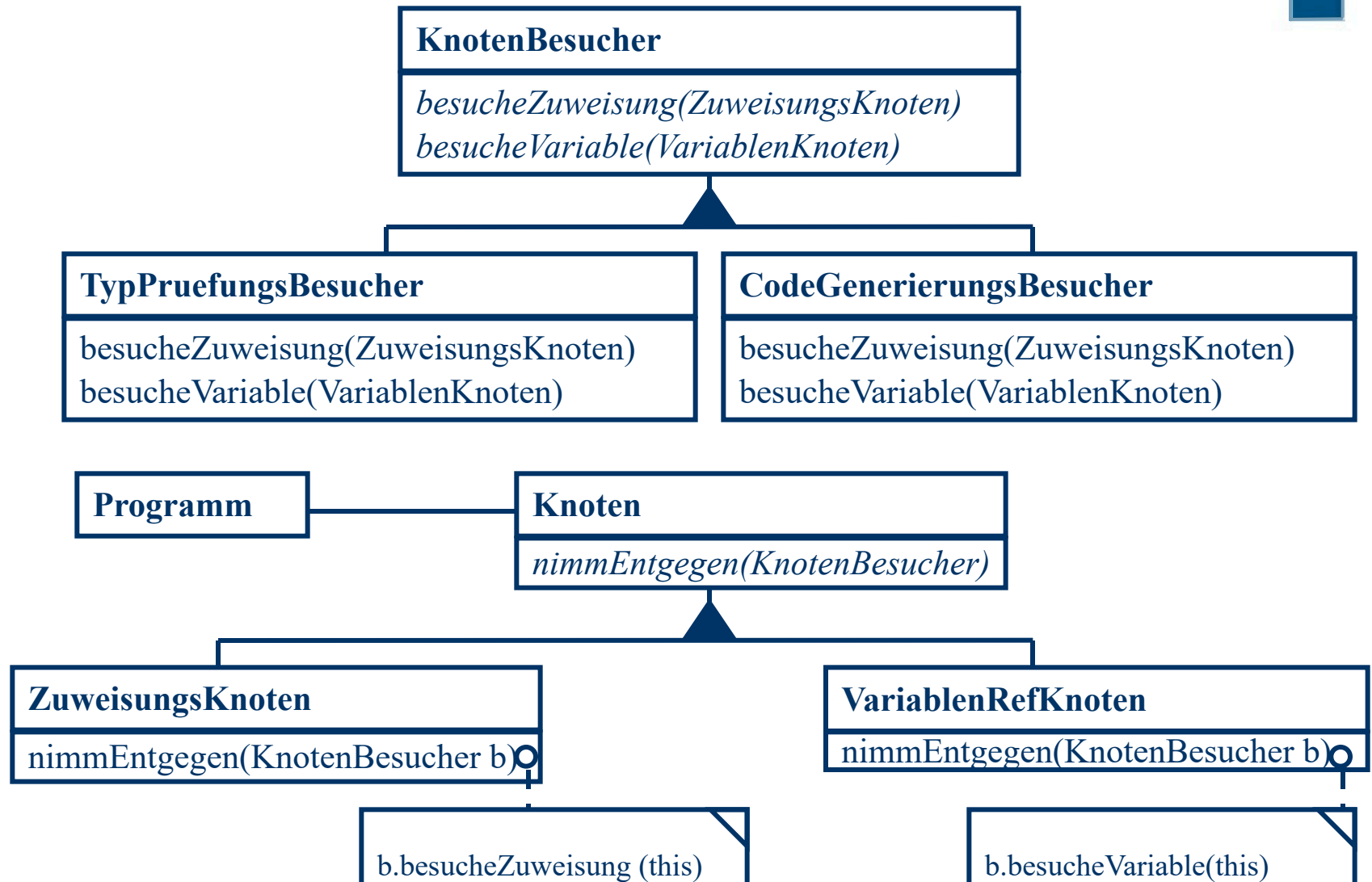
Variable? Addition?
Methodenaufruf?

Java: polymorphe Auswahl nur
bzgl. Laufzeit-Typ von **lhs**

Java: polymorphe Auswahl nur
bzgl. Laufzeit-Typ von **b**

Exkurs: Entwurfsmuster „Besucher“ (5)

Beispiel: Übersetzer *mit* Besucher

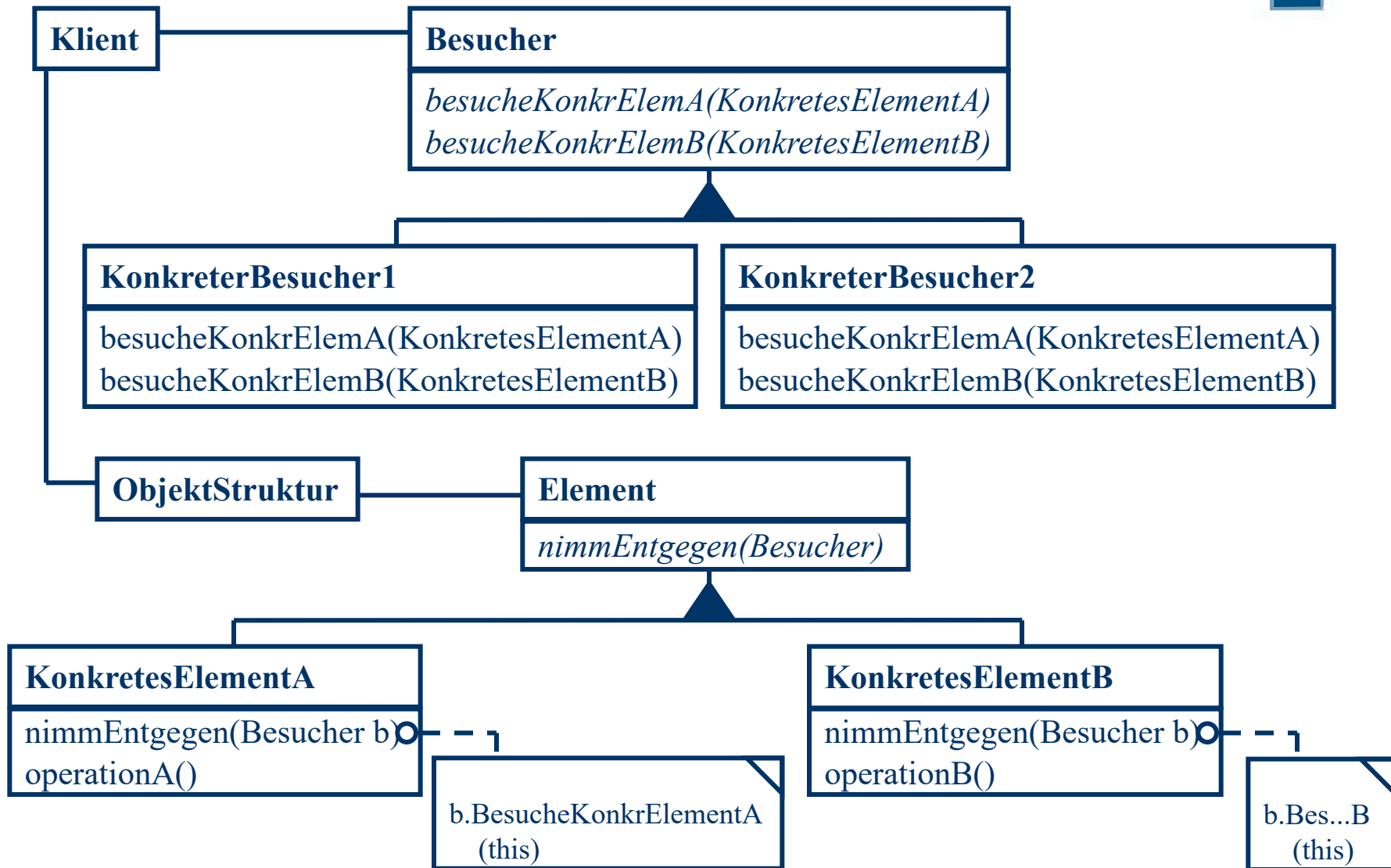


Exkurs: Entwurfsmuster „Besucher“ (6)

- Jeder AST-Blattknoten erhält eine Instanzmethode `nimmEntgegen` / `accept`.
 - Diese erhält als Argument eine Besucher-Instanz (plus ggf. weitere Argumente).
 - Dann ruft diese die passende `besuche*`- / `visit*`-Methode des übergebenen Besuchers auf.
 - Die Auswahl der passenden `nimmEntgegen`-Methode wird zur Laufzeit anhand des *dynamischen Typs* getroffen.
 - Innerhalb einer `nimmEntgegen`-Methode ist klar, welche Besuchermethode aufgerufen werden muss.
 - Java bietet die Möglichkeit, die `besuche`-Methoden zu überladen.
- Sogenannter **Double Dispatch**.

Exkurs: Entwurfsmuster „Besucher“ (7)

Allgemeine Struktur



Anwendbarkeit

- wenn eine Objektstruktur viele Klassen von Objekten mit unterschiedlichen Schnittstellen enthält und Operationen auf diesen Objekten ausgeführt werden sollen, die von ihren konkreten Klassen abhängen,
- wenn viele unterschiedliche und nicht miteinander verwandte Operationen auf den Objekten einer Objektstruktur ausgeführt werden müssen und diese Klassen nicht mit diesen Operationen „verschmutzt“ werden sollen,
- wenn sich die Klassen, die eine Objektstruktur definieren, praktisch nie ändern, aber häufig neue Operationen für die Struktur definiert werden.

Tree-Besucher von javac

- Der Java-Übersetzer benutzt das Besucher-Muster.
Jeder Tree-Knoten muss folgende Methode implementieren:

Besucher-Objekt

```
public abstract <R,D> R accept(TreeVisitor<R,D> v, D arg);
```

AST-Klasse JCMethodeDecl:

```
public <R,D> R accept(TreeVisitor<R,D> v, D d) {  
    return v.visitMethod(this, d);  
}
```

AST-Klasse JCAssignOp:

```
public <R,D> R accept(TreeVisitor<R,D> v, D d) {  
    return v.visitCompoundAssignment(this, d);  
}
```

Aufruf der Methode des Besuchers, die für
den jeweiligen Knotentyp spezialisiert ist.

Pretty-Printer zum Test

- Ein „Pretty-Printer“ traversiert den AST eines Programms und gibt dabei ein Programm in der Quellsprache aus.
- Aus der Struktur des AST (abstrakte Syntax) werden die „Satzzeichen“ der konkreten Syntax wieder abgeleitet.
- Typischer Test:
Wenn `parse(P)` für ein Programm `P` einen AST `T` erzeugt und `pretty(T)` einen Programmtext rekonstruiert,
 - dann *sollte* gelten:
 $\text{pretty}(\text{parse}(P)) \sim P$ //bis auf Kommentare, Einrückungen,
//äquivalente Ablaufsteuerung,
//explizite Default-Initialisierungen, ...
 - dann *muss* gelten:
 $\text{pretty}(\text{parse}(\text{pretty}(\text{parse}(P)))) = \text{pretty}(\text{parse}(P))$

Pretty-Printer von javac (1)

```
public void visitIf(JCIf tree) {
    try {
        print("if ");
        if (tree.cond.hasTag(PARENS)) {
            printExpr(tree.cond, TreeInfo.noPrec);
        } else {
            print("(");
            printExpr(tree.cond, TreeInfo.noPrec);
            print(")");
        }
        print(" ");
        printStat(tree.thenpart);
        if (tree.elsepart != null) {
            print(" else ");
            printStat(tree.elsepart);
        }
    } catch (IOException e) {
        throw new UncheckedIOException(e);
    }
}
```

Pretty-Printer von javac (2)

```
public void printExpr(JCTree tree, int prec) throws IOException {  
    int prevPrec = this.prec;  
    try {  
        this.prec = prec;  
        if (tree == null) print("/*missing*/");  
        else {  
            tree.accept(this);  
        }  
    } catch (UncheckedIOException ex) {  
        ...  
    } finally {  
        this.prec = prevPrec;  
    }  
}
```

Typ des Ausdrucks
statisch unbekannt →
Double Dispatch

Behandlung des
Präzedenzlevels für
Operationen.

Modulare Struktur von Übersetzern

Analyse

Token

Syntax

Semantik

Abbildung

Transf.

Optim.

Codierung

Code-Erz.

Ass./Bind.

Quelltext

Token-
folge

*Struktur-
baum*

*attrib.
Struktur-
baum*

Zwischen-
Code

Ziel-
programm

Namens-
tabelle

*Definitions-
tabelle*

- Aufgaben:
 - Feststellung der bedeutungstragenden Elemente
 - Zuordnung statischer Bedeutung
 - Konsistenzprüfung
- Schritte:
 - Lexikalische Analyse/Abtastung (Scanner)
 - Syntaktische Analyse/Zerteilung (Parser)
→ Semantische Analyse
- Noch keine Übersetzung, nur Analyse

Semantische Analyse

- Es gibt Eigenschaften von Programmiersprachen, die nicht durch kontextfreie Grammatiken beschreibbar sind.
- Diese Eigenschaften (= statische Semantik) werden durch Kontextbedingungen festgelegt.
- Dazu gehören:
 1. *Deklarierttheitseigenschaft:*

Zu jedem angewandt auftretenden Bezeichner (Verwendung im Code) muss es genau eine explizite (verfügbare) Deklaration geben.
 2. *Typkonsistenz:*

Zur Laufzeit wird keine Operation auf Operanden angewendet, auf die sie vom Argumenttyp her nicht passt.
 3. *Programmiersprachliche Regeln.* Beispiele:
 - Wurde lokale Variable vor Verwendung initialisiert? (z.B. Java)
 - Darf Wert in aktuellem Kontext verändert werden? (z.B. Rust)
 - Um eindeutige Grammatik anzugeben, müssen ggf. Überprüfungen auf semant. Analyse verschoben werden (z.B. erlaubte Flags).

Semantische Analyse: Schwierigkeiten

- Prüfung der Deklariertheitseigenschaft:
 - Buchhaltung über gültige und sichtbare Bezeichner.
 - Namensanalyse (welcher Bezeichner ist gemeint bzw. wo ist die zugehörige Deklaration).
 - Benötigt komplexe Datenstruktur.
 - Umfangreiche Suchaufgabe: muss effizient gemacht sein.
- Oft sind die Prüfung der Deklariertheitseigenschaft und die Prüfung der Typkonsistenz (und weitere Konsistenzprüfungen) miteinander verschränkt.

Isolierte Betrachtung (und Implementierung) in der Praxis kaum möglich.

Gültigkeit von Bezeichnern

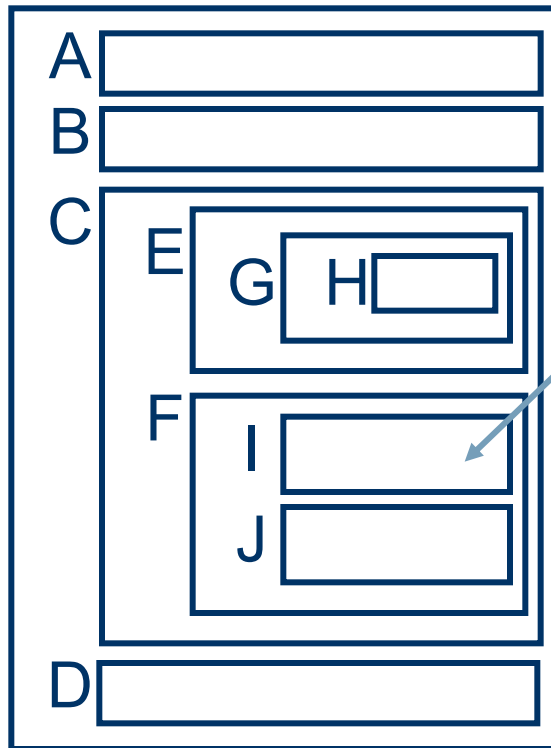
- Programmiersprachen definieren die Gültigkeit von Variablen-/Methoden/...-Bezeichnern.
- Üblich: Bezeichner ist **gültig**
 - in der Programmeinheit (Prozedur, Block, ...), die die Deklaration statisch umfasst.
 - in der Programmeinheit (Prozedur, Block, ...), die die Deklaration statisch umfasst, aber nur nach der Stelle der Deklaration (heute nur noch bei lokalen Variablen üblich).
- Gültigkeit bedeutet *nicht*, dass der Bezeichner an einer Verwendungsstelle auch **sichtbar** ist.

Sichtbarkeit von Bezeichnern

- Üblich in ALGOL-artigen Sprachen ist statische Bindung.
- Locker:
Variablen, die in inneren Blöcken deklariert werden, „verdecken“ weiter außen deklarierte Variablen gleichen Namens.
- Formal:
Ein definierendes Auftreten eines Namens ist sichtbar in einer Programmeinheit (Prozedur oder Block), in deren Deklarations- oder Spezifikationsteil die Definition steht, abzüglich aller von dieser Programmeinheit echt (statisch) umfassten Programmeinheiten, die eine neue Definition des Namens enthalten.
- Oft kann auf gültige, aber nicht sichtbare Bezeichner trotzdem zugegriffen werden (super.... oder ::-Operator von C++).

Statische, geschachtelte Sichtbarkeit (1)

Standard in modernen Sprachen:
Bezeichner sind gültig im deklarierenden Block.



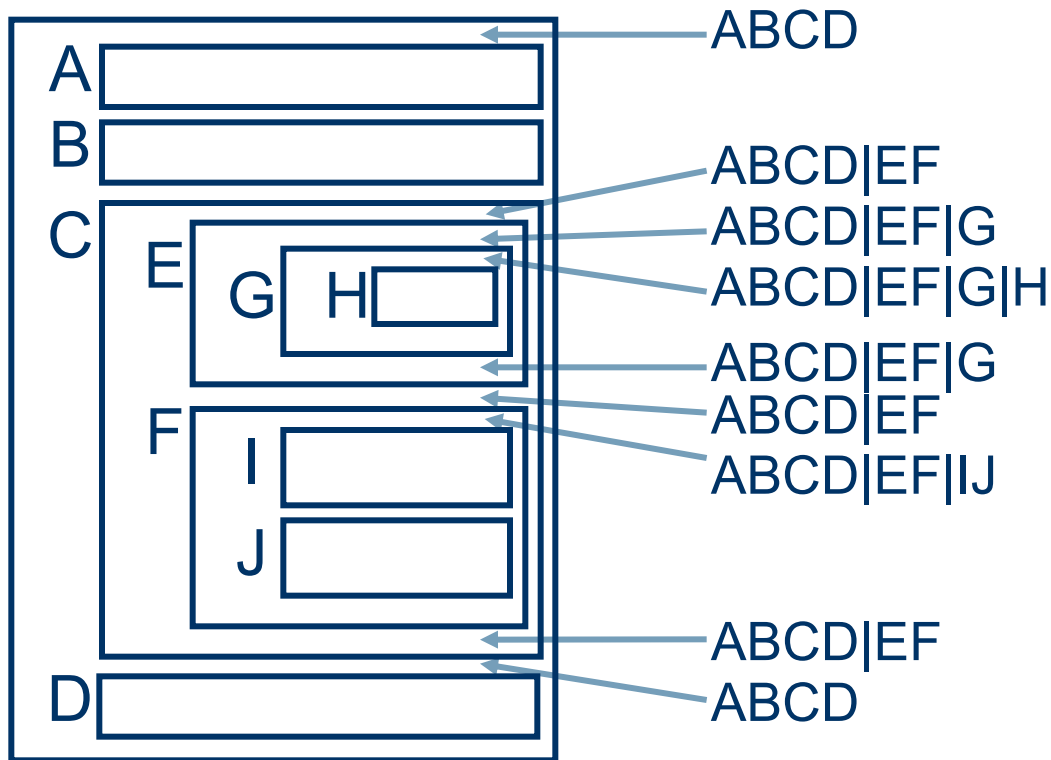
sichtbar:

A
B
C
D
E
F
I
J

nicht gültig:

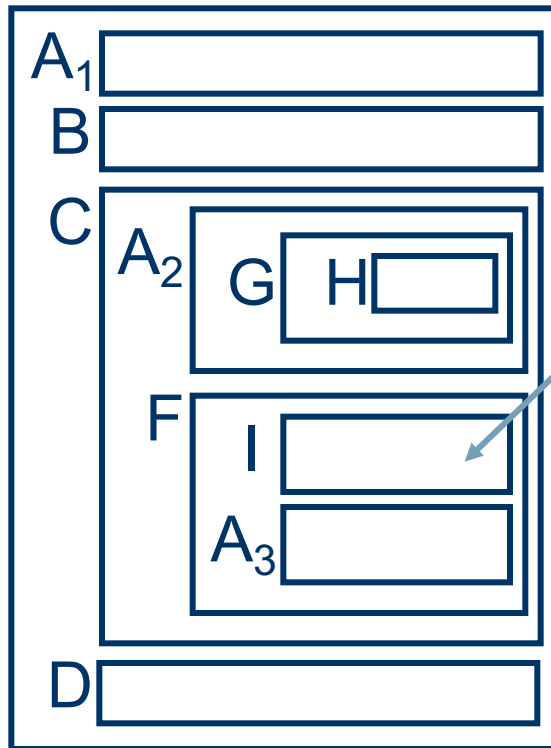
G
H

„Sichtbarkeitsschichten“ haben Stapel-Charakter



Statische, geschachtelte Sichtbarkeit (2)

- Sichtbare Bezeichner in einer Ebene müssen immer verschieden sein.



sichtbar:

A_3 → Äußere Deklaration mit
gleichem Namen (A_1, A_2)
sind zwar gültig, aber
nicht sichtbar.
 B
 C
 D
 F
 I

Pathologisches Beispiel (1)

```
class Bazz { ... }
```

```
class Foo {
```

```
    Bazz field;
```

```
    class Bar {
```

```
        void set(Bazz val) {
```

```
            field = val;
```

```
        }
```

```
        Bazz field;
```

```
        class Bazz { ... }
```

```
    }
```

```
}
```

Hier beginnt eine neue Sichtbarkeitsschachtel.



Verwendung textuell früher als zugehörige Definition.



Pathologisches Beispiel (2)

```
class Bazz { ... }
```

```
class Foo {
```

```
    Bazz field;
```

```
    class Bar {
```

```
        void set(Bazz val) {
```

```
            field = val;
```

```
        }
```

```
        Bazz field;
```

```
        class Bazz { ... }
```

```
    }
```

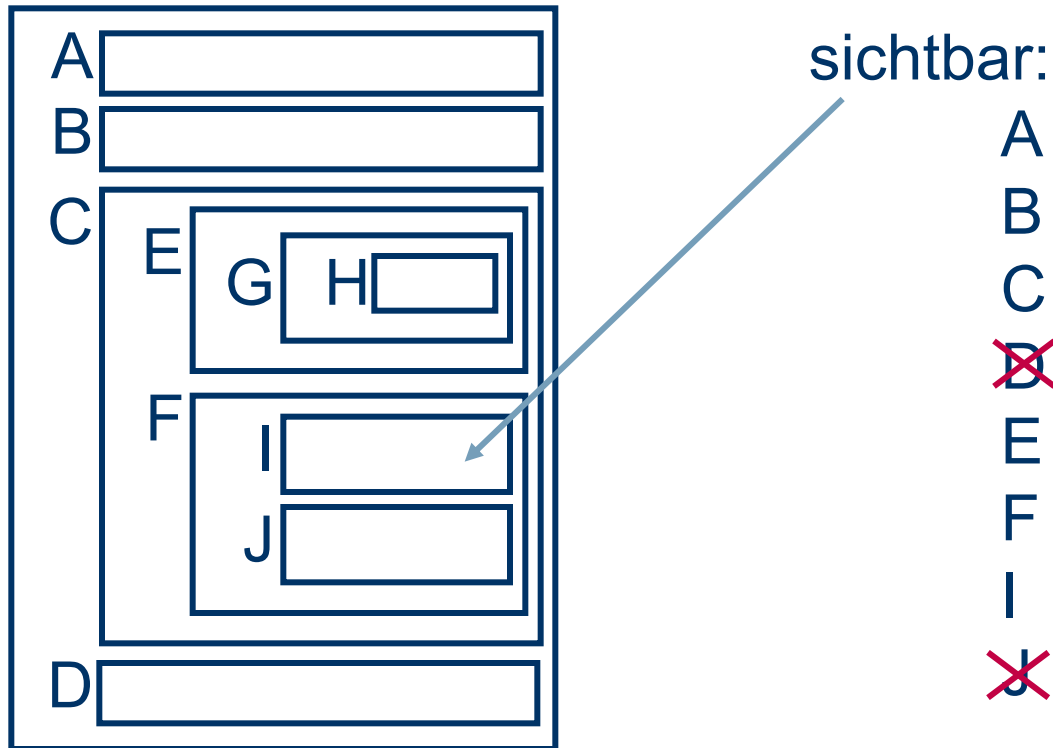
```
}
```

Hier beginnt eine neue Sichtbarkeitsschachtel.

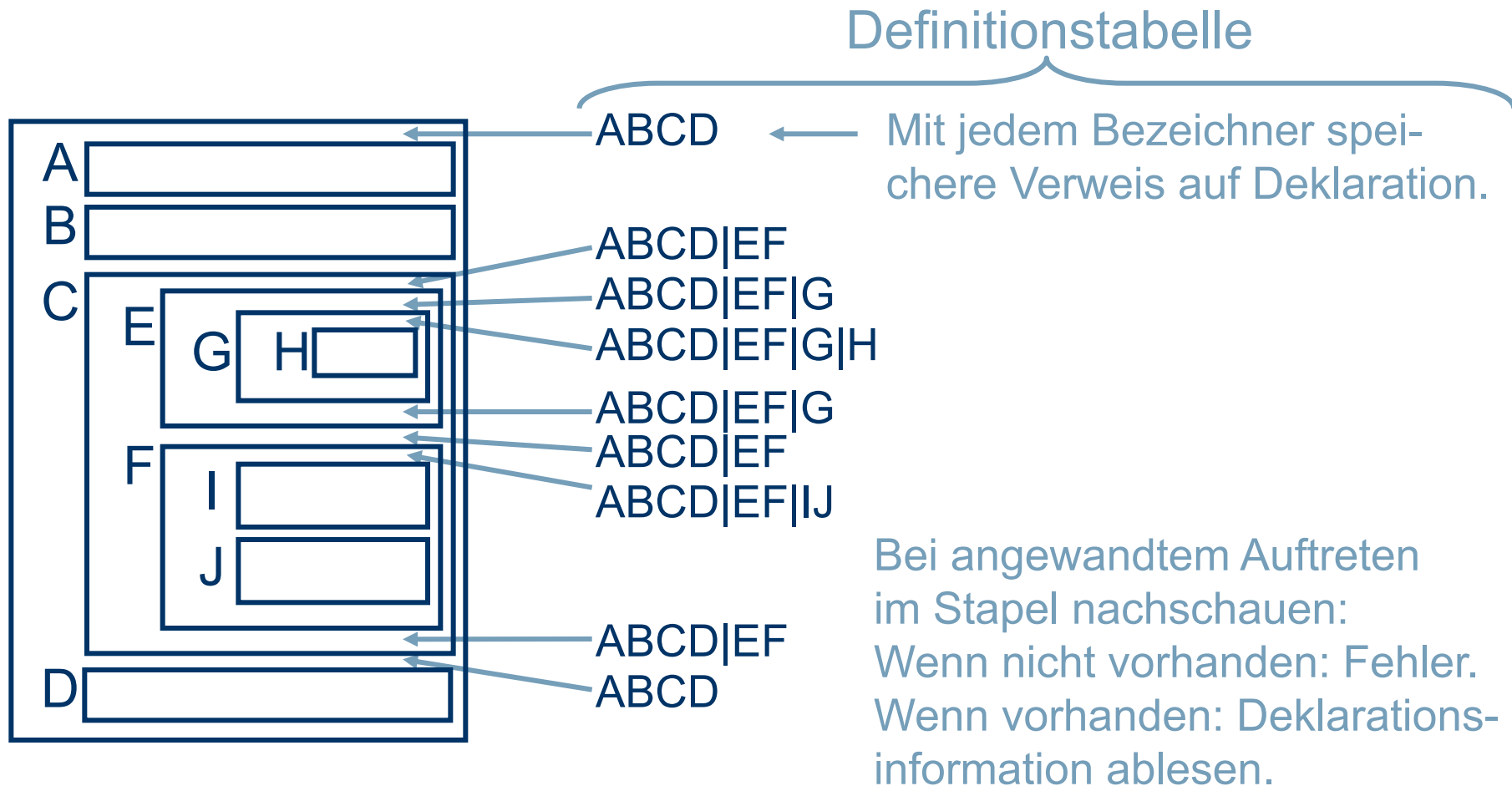
Verwendung textuell früher als zugehörige Definition.

Frühere „1-Durchgangssprachen“

- Gültigkeit ab Deklarationsstelle, FORWARD-Deklaration.



Stapel zur Deklariertheitsprüfung



Definitionstabellen

- Für jedes definierende Vorkommen eines Bezeichners wird die zugehörige deklarative Information gespeichert:
Bezeichner → Information (z.B. Typ, Adresse im Speicher, ...)
Eintrag in der Definitionstabelle bzw. Erzeugung eines Symbol-Objekts.
- Die Definitionstabelle erlaubt, bei einem gegebenen angewandten Vorkommen eines Bezeichners schnell die zugehörige (sichtbare) Definition (Symbol-Objekt) aufzufinden.
- In Sprachen, in denen unterschiedliche Sprachkonstrukte unterschiedliche Namensräume verwenden, bietet sich die Verwendung mehrerer Definitionstabellen an.
Z.B. für Klassennamen, Methodennamen, Variablennamen.

Prüfung der Deklariertheitseigenschaft

Besuche den AST rekursiv, top-down:

- Am Blockeintritt neue „Sichtbarkeitsschachtel“ öffnen.
 - Trage bei jedem Deklarationsknoten des Blocks die neu deklarierten Bezeichner mit Deklarationsinformation in die Definitionstabelle ein.
 - (Dabei wird auch der Typ (=Zeiger auf Typ-Objekt) eingetragen.)
 - Melde Fehler, wenn Duplikat.
 - Untersuche die Knoten des Blocks, die keine Deklarationen sind (z.B. Anweisungen, enthaltene Blöcke).
 - Prüfe bei jedem angewandt vorkommenden Bezeichner, *ob er in der Definitionstabelle vorkommt*. (Wird später noch vertieft!)
 - Melde Fehler, wenn Bezeichner nicht gefunden werden kann.
 - Bei überladenen Operationen: Fehler, wenn kein „passender“ Bezeichner gefunden werden kann (→ Typprüfung).
 - Speichere Deklarationsinformation an „Verwendungsstelle“.
- Am Blockende „Sichtbarkeitsschachtel“ schließen (zuletzt neu eingeführte Einträge aus der Definitionstabelle entfernen).

Namensanalyse erfordert oft Mehrfachtraversierung

- Obiges allgemeines Prüfungsschema traversiert einen Block zweimal:
 - Deklarationsknoten
 - Nicht-Deklarationsknoten
- Wenn die Programmiersprache vorschreibt, dass Deklarationen textuell vor einer Verwendung erscheinen müssen, wird es für die Namensanalyse einfacher: *ein* Besuch des AST reicht.
- Achtung: Manchmal sind Deklarationsknoten und Nicht-Deklarationsknoten gemischt:
 - `int a=b;`
 - `a + (let b=c in b*a)`

- Am Ende der Deklariertheitsprüfung ist der Analysestapel („Definitionstabelle“) wieder leer.
- Analyseergebnisse liegen nur temporär in Definitionstabelle vor und müssen daher anderweitig gespeichert werden.
Verschiedene Möglichkeiten:
 - AST-Knoten zeigen auf Symbol-/Typ-Objekte:
 - Alternativer nicht-oo-Ansatz: Symbol-/Typ-Informationen werden in Tabelle abgelegt. AST-Knoten zeigen auf Tabelleneinträge. Diese Tabelle wird ebenfalls „Definitionstabelle“ genannt. Stapel wird getrennt von dieser „Definitionstabelle“ geführt.
 - Angewandte Bezeichner im AST zeigen auf zugehörige Deklarationsknoten im AST.

Sonstige Sichtbarkeitsschachteln

- Sichtbarkeitsschachteln werden nicht nur durch Klassen, Methoden und Blöcke aufgespannt.
- with-Anweisung: **with x do begin ... a ... end**
 - Öffnet den Verbundtyp **t** von **x** als zusätzliche Sichtbarkeitsschachtel.
 - Suche nach **a** beginnt in den Namen des Verbundtyps; dann wird bei den Namen des umgebenden Blocks fortgesetzt.
- Qualifizierte Zugriffe: **x . a**
 - Nach dem Qualifikationspunkt ist ein anderer Namensraum zulässig.
 - **Problem:** Der Typ des Qualifikators **x** muss bereits bekannt sein, damit die typspezifischen, nach dem Qualifikationspunkt zur Verfügung stehenden Namen bekannt sind.

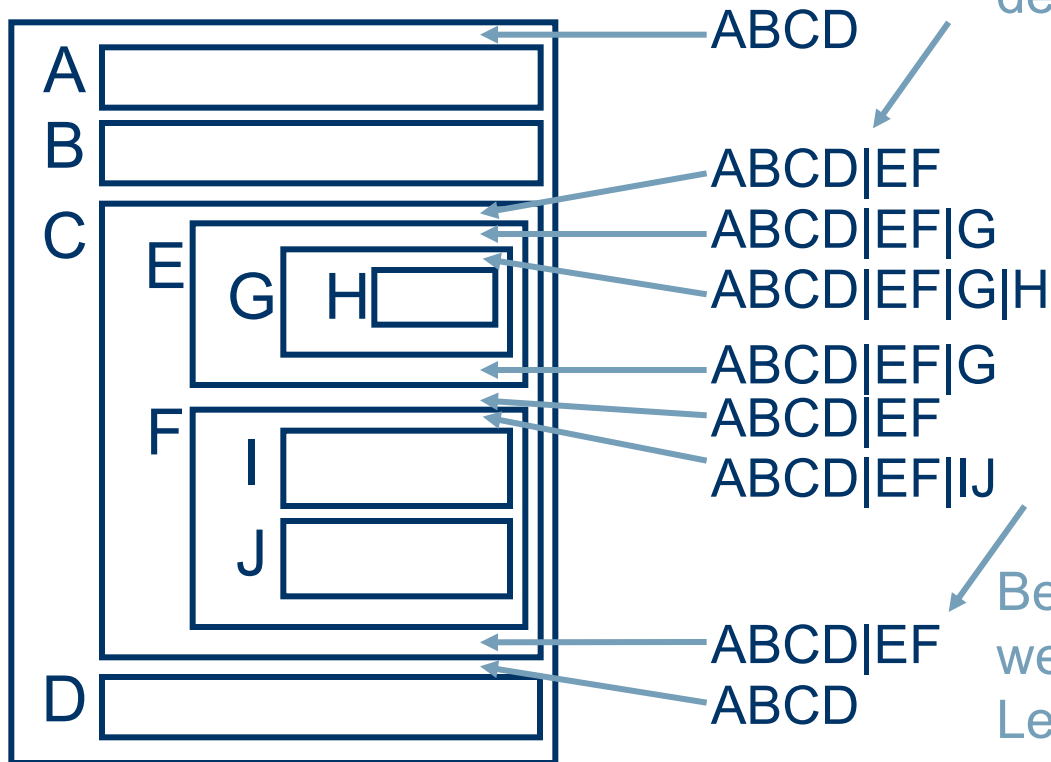
Weitere Abhängigkeiten zur Typanalyse

- Bei Vererbung in oo-Sprachen müssen die zur Verfügung stehenden Attribute/Funktionen der Oberklasse bekannt sein.
- Bei Identifikation von Funktionen abhängig von der Signatur (überladene Funktionen) müssen die Argumenttypen bekannt sein.

Implementierung der Definitionstabellen - erster Versuch -

- Die Definitionstabelle könnte als einfacher Stapel implementiert werden.
 - `pushSymbol(SymbolTable, name)`
 - `popSymbol(SymbolTable)`
 - `getSymbol(SymbolTable, name)`
- Frage: Wie kann man aber prüfen, ob innerhalb einer Ebene ein Bezeichner mehrfach deklariert wird?
- Antwort: „Lesezeichen“ (|) einführen.
 - `scopeSymbolTable(SymbolTable)`
 - `putSymbol(SymbolTable, name)`
 - `unscopeSymbolTable(SymbolTable)` //alles bis zum letzten |
 - `getSymbol(SymbolTable, name)`

Definitionstabelle benimmt sich wie Stapel



Beim Betreten eines Blocks wird zunächst ein Lesezeichen | auf den Stapel gelegt.

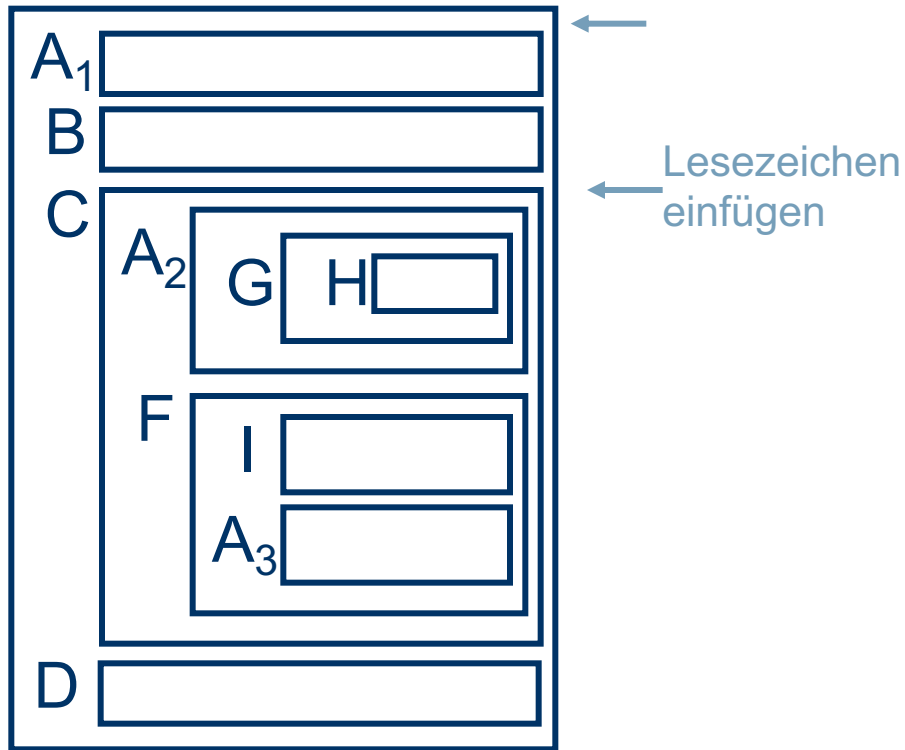
Problem:
Lineare Suche!

Beim Verlassen des Blocks werden alle Einträge bis zum Lesezeichen | vom Stapel entfernt.

Verbesserte Implementierung der Definitionstabelle

- Streuspeicherung (Hash-Tabelle) mit Verkettung:
 - Alle Bezeichner einer Ebene in die Streutabelle eintragen.
 - Ist ein Eintrag schon vorhanden: Fehler.
 - Konzeptuell: zusätzlicher Stapel
 - Jeder Bezeichner wird im Stapel eingetragen.
 - Beginnt eine neue Sichtbarkeitsebene, dann
 - auf dem Stapel ein Lesezeichen hinterlassen.
 - Bezeichner werden weiterhin in die gleiche Streutabelle eingetragen.
 - Verketteten, wenn ein Eintrag schon vorhanden ist: Liste wird erzeugt, neuester Eintrag wird unmittelbar aus der Streutabelle erreicht.
 - Endet eine Sichtbarkeitsebene, dann wird der Stapel bis zum (inkl.) Lesezeichen abgearbeitet:
 - Jeder Bezeichner wird aus der Streutabelle entfernt.
 - Ist er am Kopf einer Liste, rücken früher deklarierte Einträge auf.

Definitionstabelle mit Streuspeicherung am Beispiel (1)

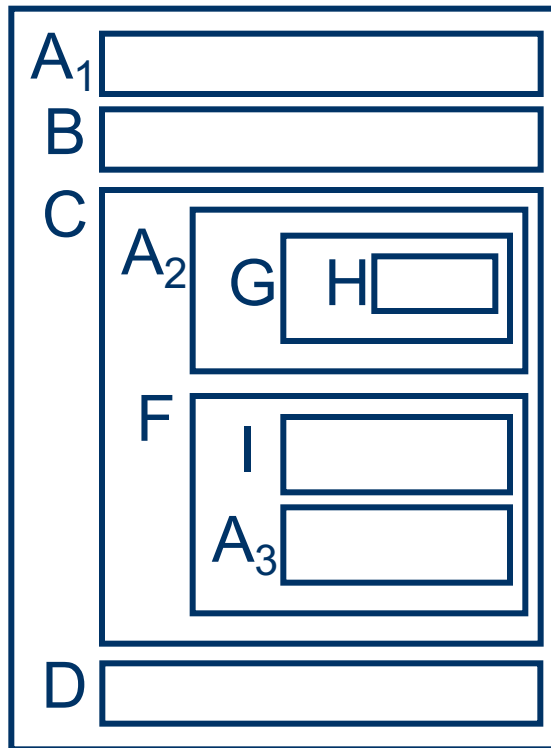


Stapel: A₁BCD|

Streutabelle:

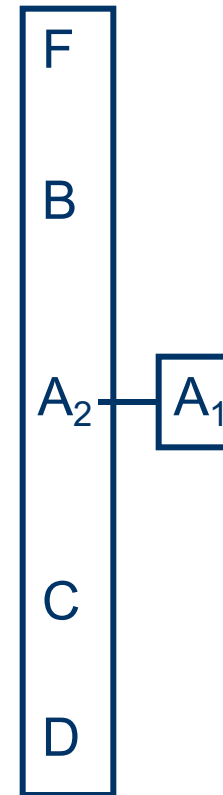


Definitionstabelle mit Streuspeicherung am Beispiel (2)

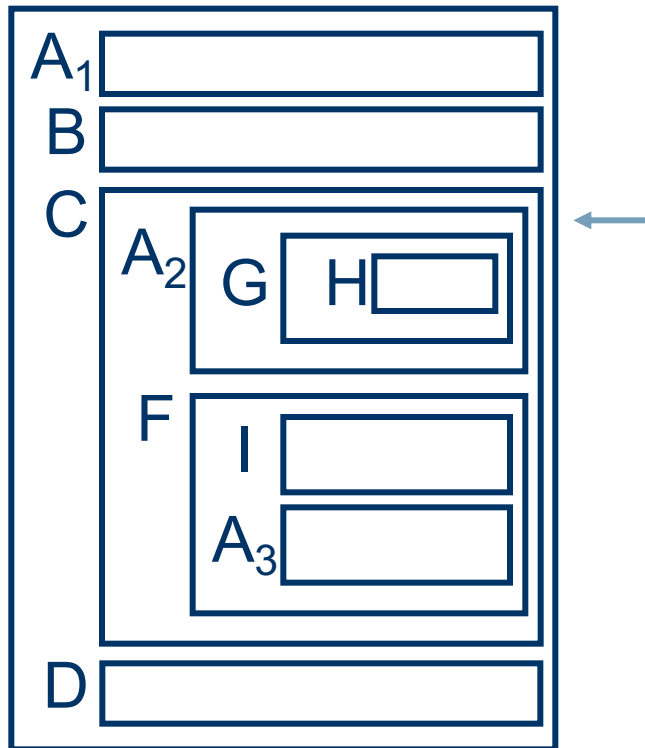


Stapel: $A_1BCD|FA_2$

Streutabelle:

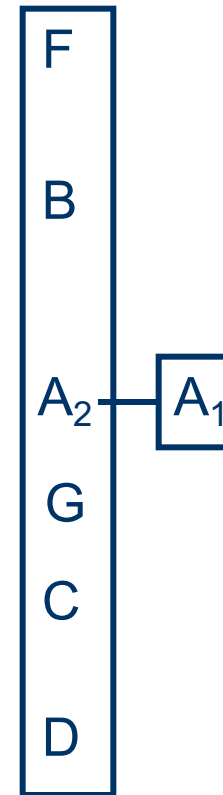


Definitionstabelle mit Streuspeicherung am Beispiel (3)



Stapel: $A_1BCDFA_2|G$

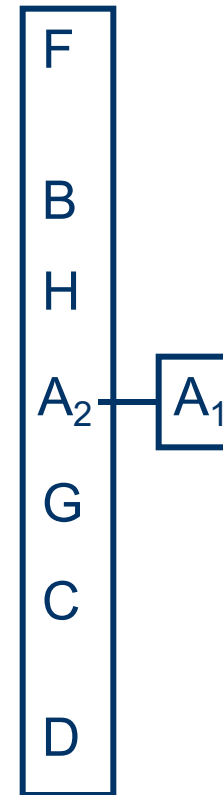
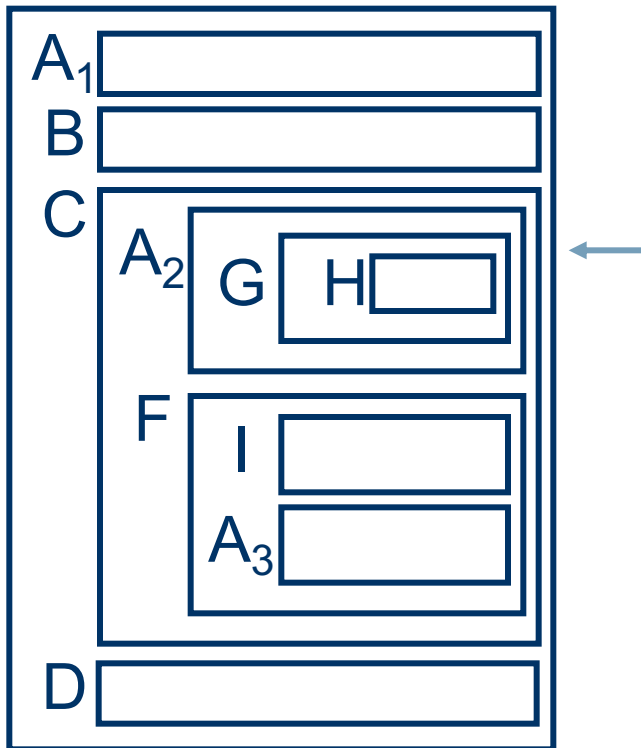
Streutabelle:



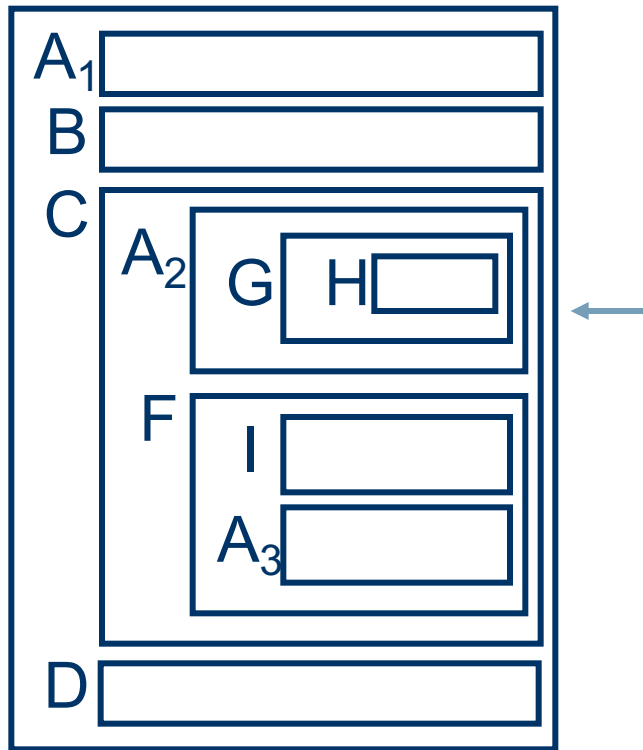
Definitionstabelle mit Streuspeicherung am Beispiel (4)

Stapel: $A_1BCD|FA_2|G|H$

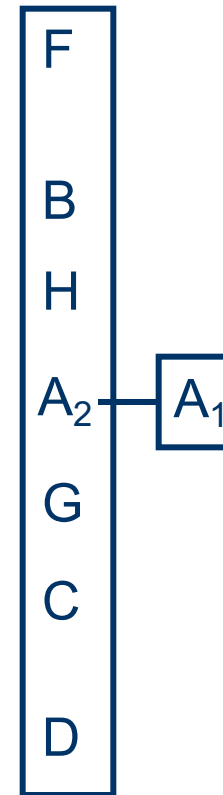
Streutabelle:



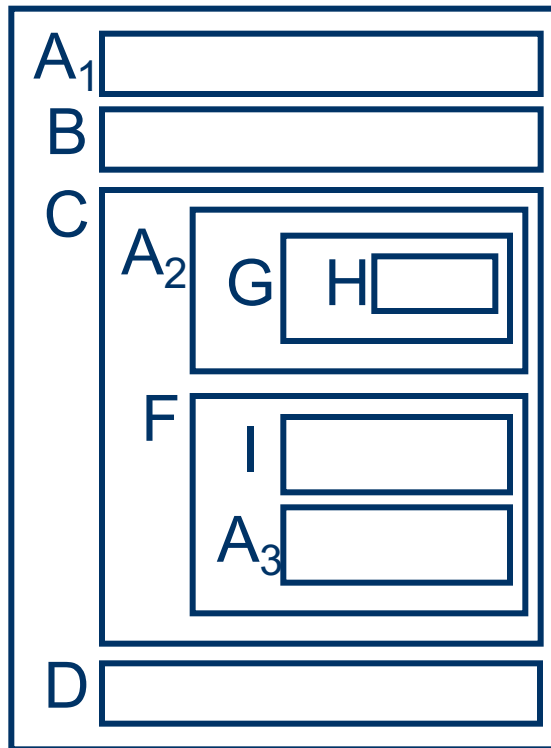
Definitionstabelle mit Streuspeicherung am Beispiel (5)



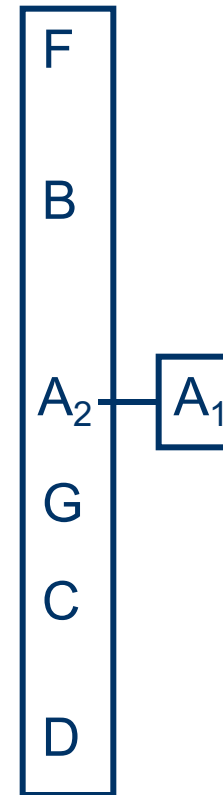
Stapel: $A_1BCD|FA_2|G|H$
Streutabelle:



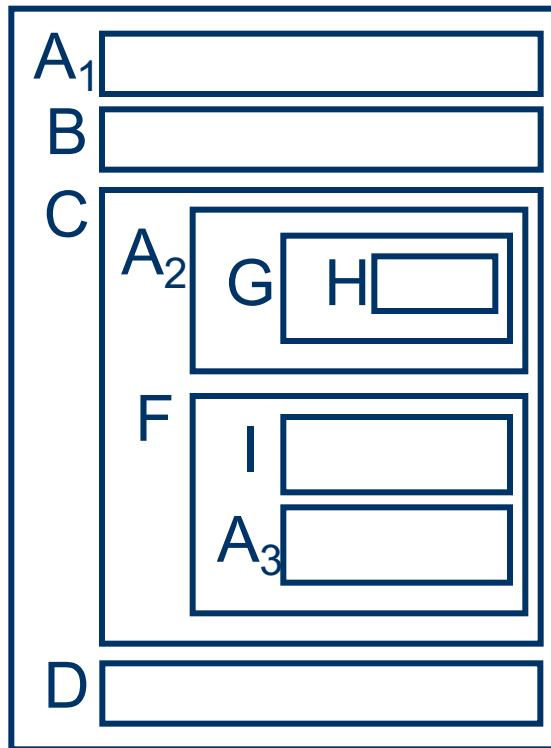
Definitionstabelle mit Streuspeicherung am Beispiel (6)



Stapel: $A_1BCD|FA_2|G$
Streutabelle: $\xleftarrow{\text{pop}} G$

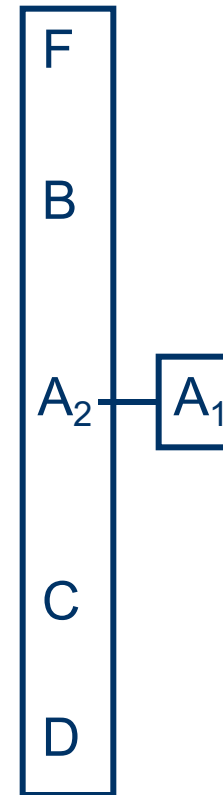


Definitionstabelle mit Streuspeicherung am Beispiel (7)

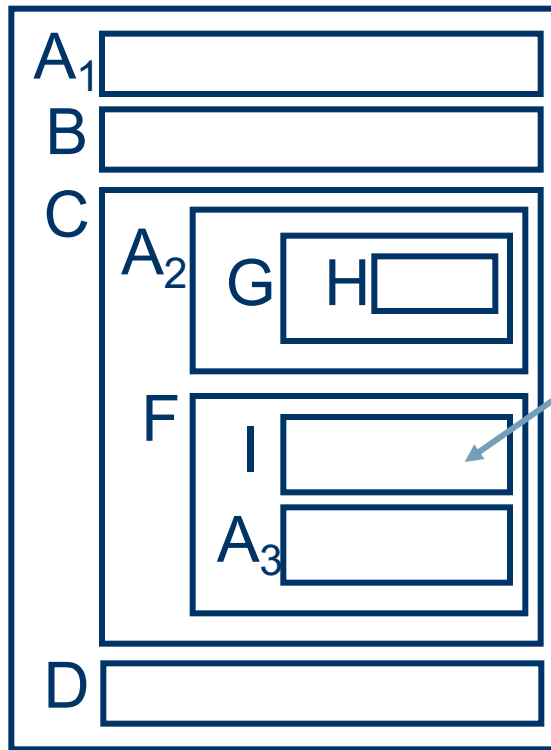


Stapel: $A_1BCD|FA_2$

Streutabelle:

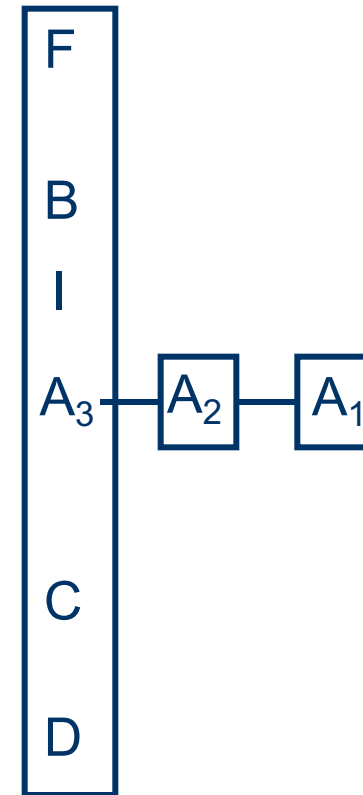


Definitionstabelle mit Streuspeicherung am Beispiel (8)



Stapel: $A_1BCD|FA_2|IA_3$

Streutabelle:



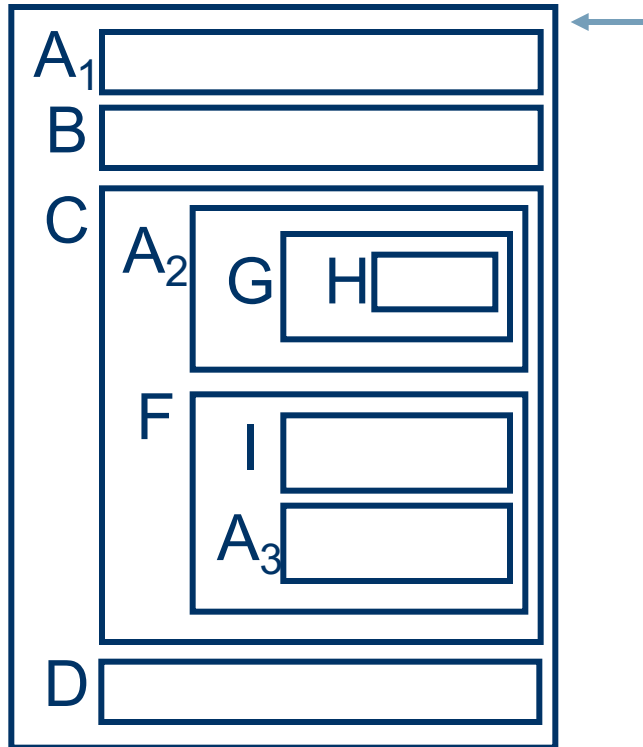
Streuspeicherung mit Verkettung

- Vorteil:
 - Bezeichner kann mit $O(1)$ nachgeschlagen werden (solange es keine Kollisionen in der Streutabelle gibt).
- Nachteile:
 - Streutabellen wachsen mit der Programmlänge. Daher ist ggf. teure Tabellenvergrößerung nötig.
 - Streutabellen brauchen recht viel Platz, um das Kollisionsrisiko klein zu halten.

Konzeptueller Stapel

- Statt zusätzlich zur Streutabelle einen echten Stapel zu verwalten, kann man den Stapel auch einfacher realisieren („Durchfädeln des konzeptuellen Kellers“):
 - Zeiger zeigt auf den zuletzt eingetragenen Bezeichner.
 - Jeder Tabellen-Eintrag hat einen Zeiger auf den letzten davor eingetragenen Bezeichner.
- Beginnt eine neue Sichtbarkeitsebene, wird der oberste Eintrag markiert.
- Beim Verlassen einer Sichtbarkeitsebene werden mithilfe des globalen Zeigers und der Verzeigerung alle Einträge bis zum nächsten markierten Eintrag gelöscht.

Definitionstabelle mit Streuspeicherung und integriertem Stapel am Beispiel (1)

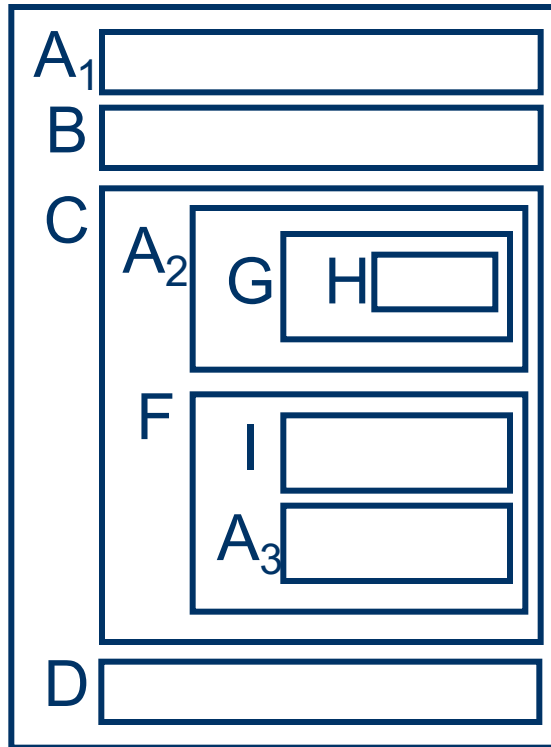


Letztes Element:

Streutabelle:



Definitionstabelle mit Streuspeicherung und integriertem Stapel am Beispiel (2)



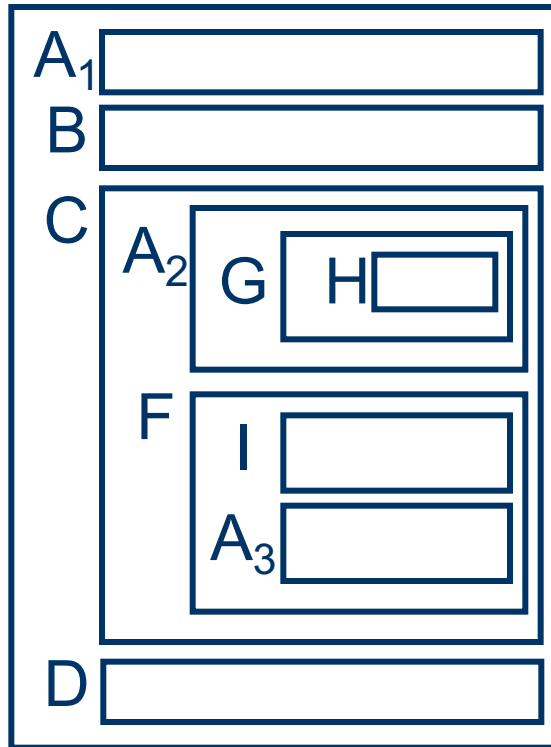
Marke * an
top-Element

Letztes Element:

Streutabelle:

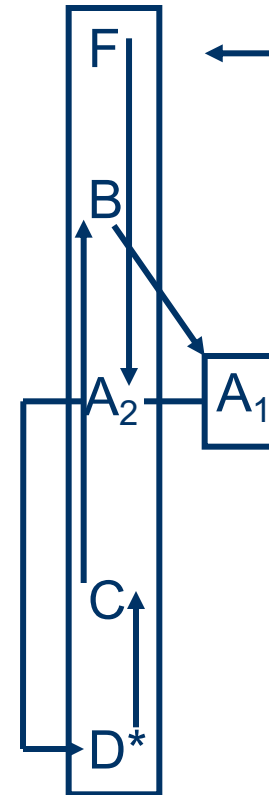


Definitionstabelle mit Streuspeicherung und integriertem Stapel am Beispiel (3)

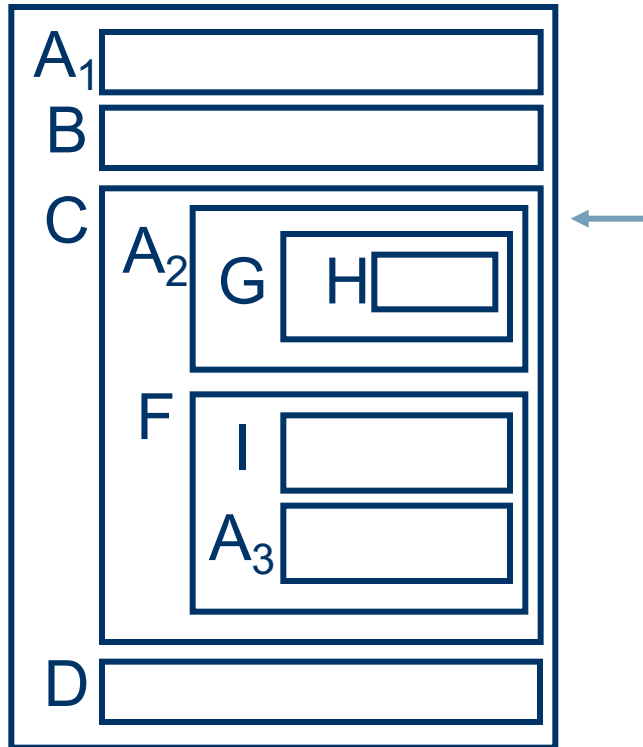


Letztes Element:

Streutabelle:

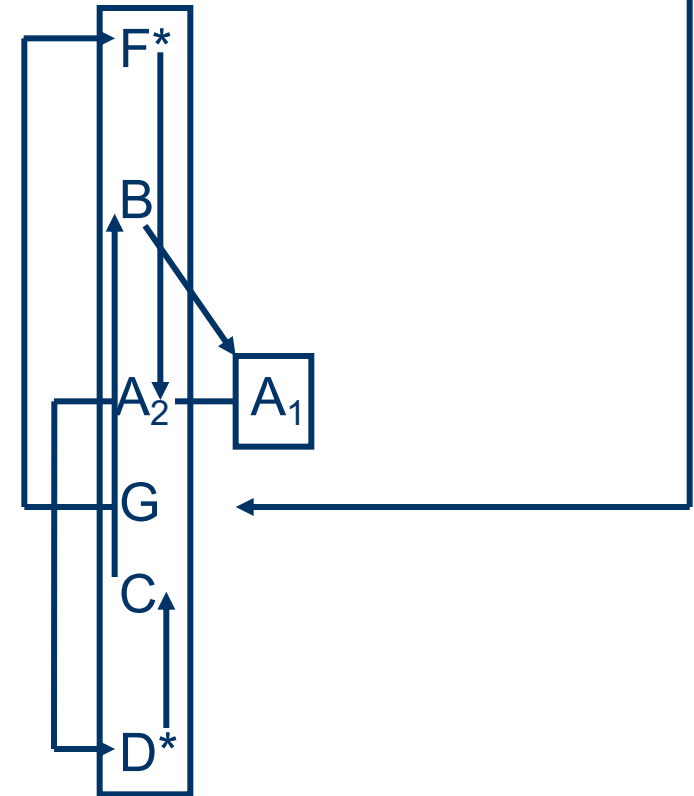


Definitionstabelle mit Streuspeicherung und integriertem Stapel am Beispiel (4)

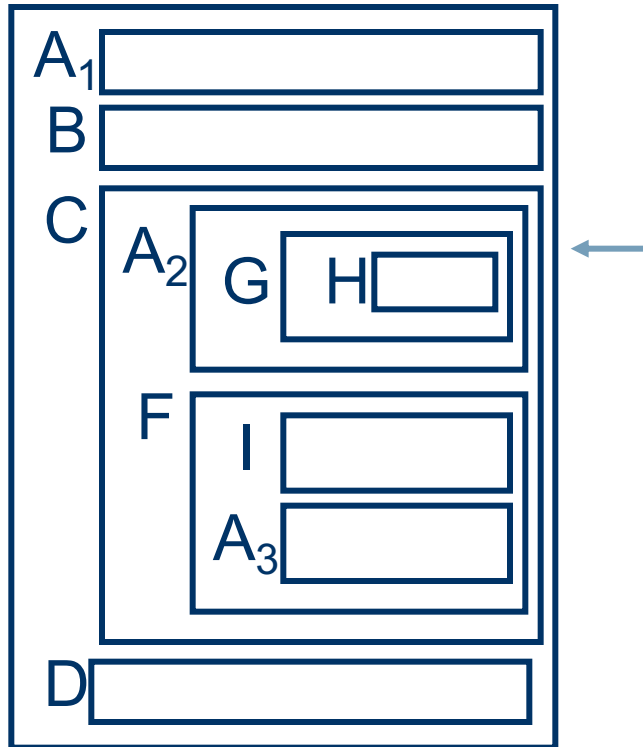


Letztes Element:

Streutabelle:

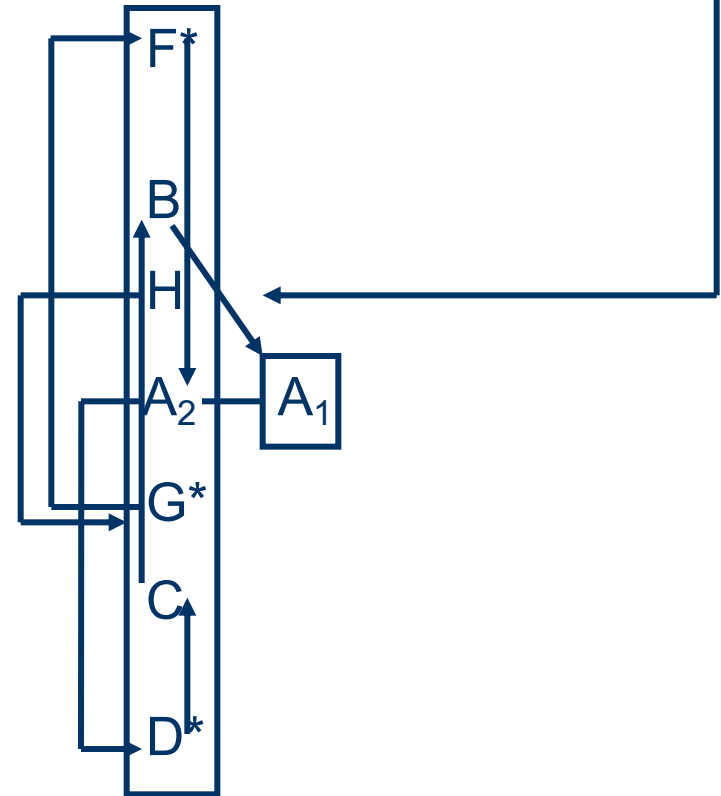


Definitionstabelle mit Streuspeicherung und integriertem Stapel am Beispiel (5)

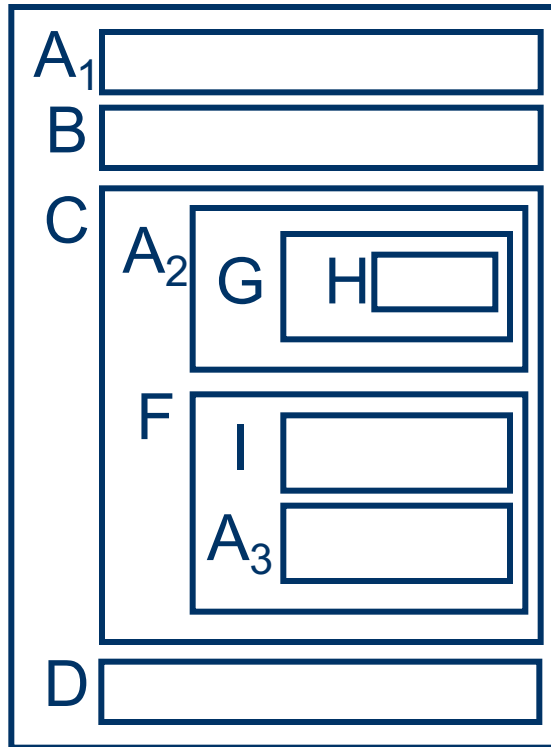


Letztes Element:

Streutabelle:



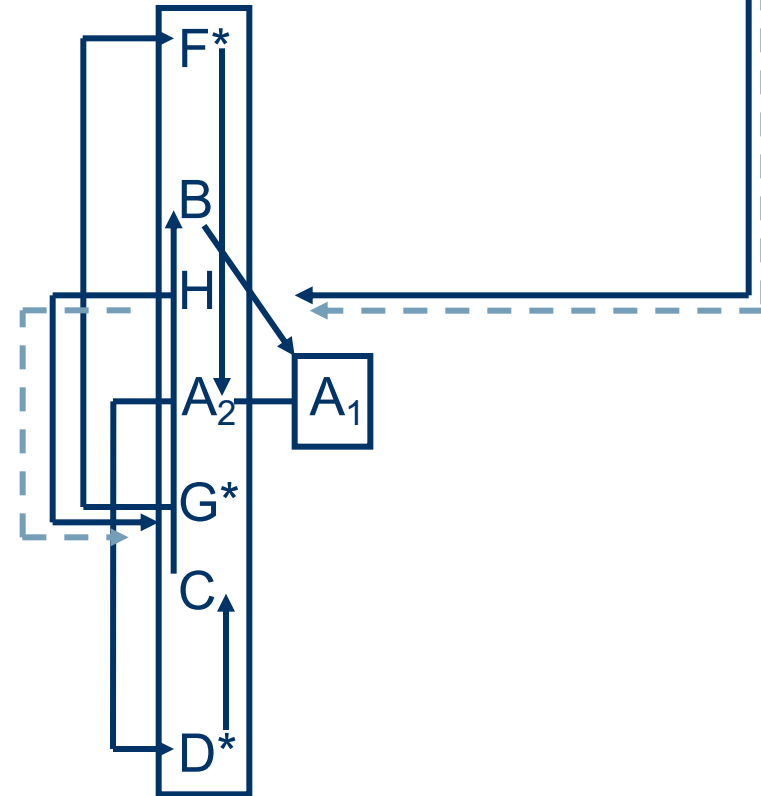
Definitionstabelle mit Streuspeicherung und integriertem Stapel am Beispiel (6)



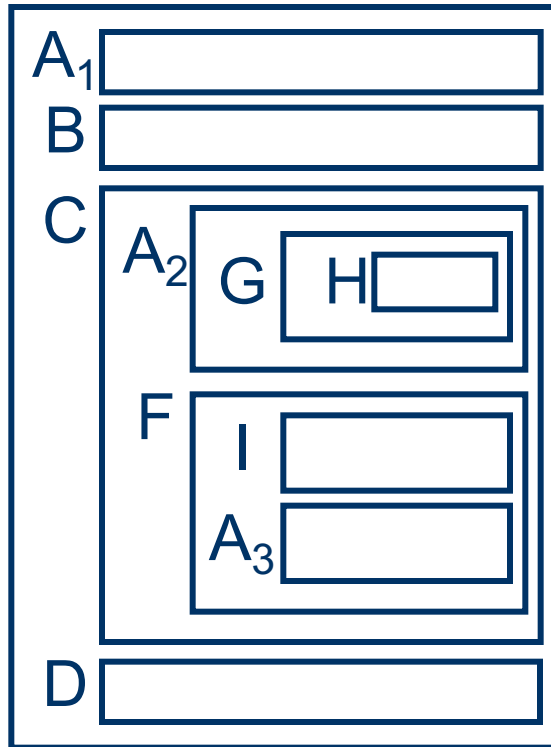
← pop bis zum *

Letztes Element:

Streutabelle:

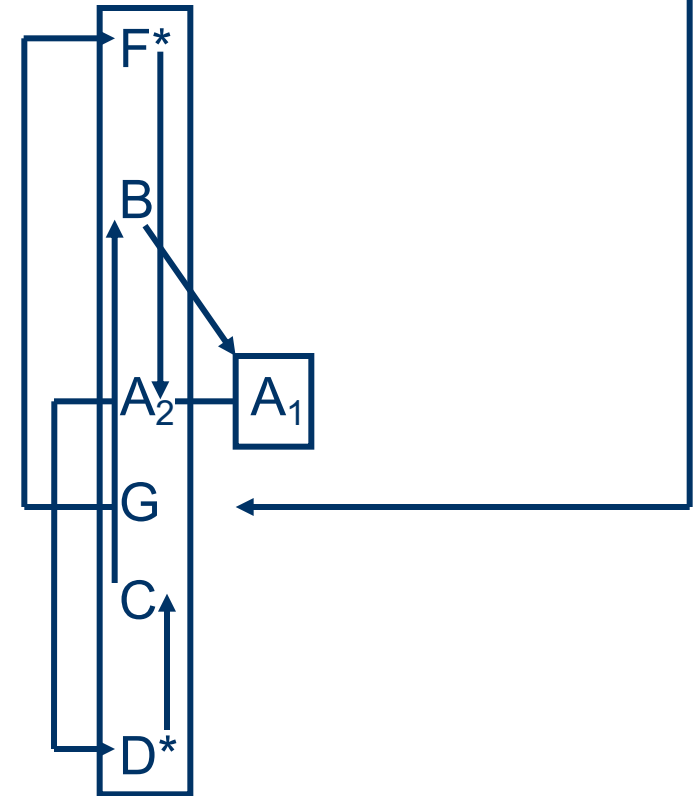


Definitionstabelle mit Streuspeicherung und integriertem Stapel am Beispiel (7)

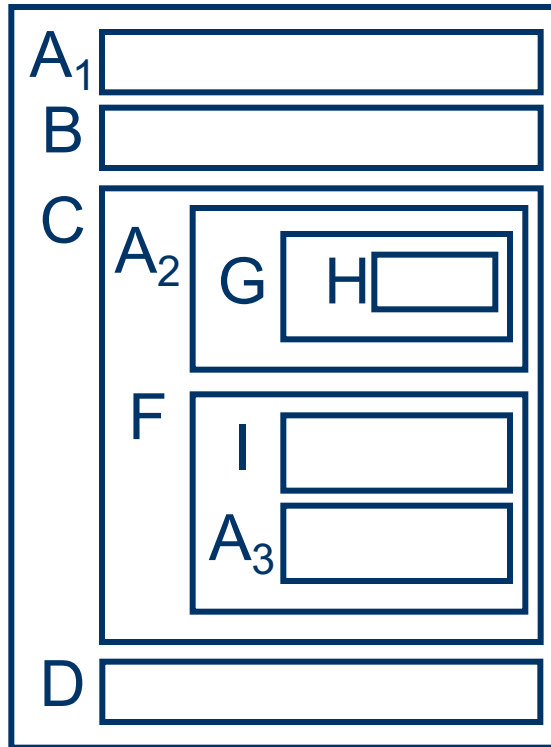


Letztes Element:

Streutabelle:

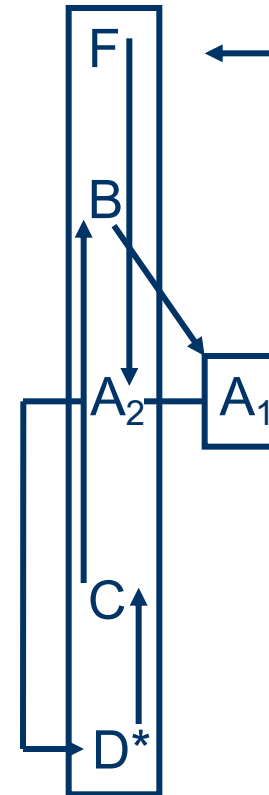


Definitionstabelle mit Streuspeicherung und integriertem Stapel am Beispiel (8)

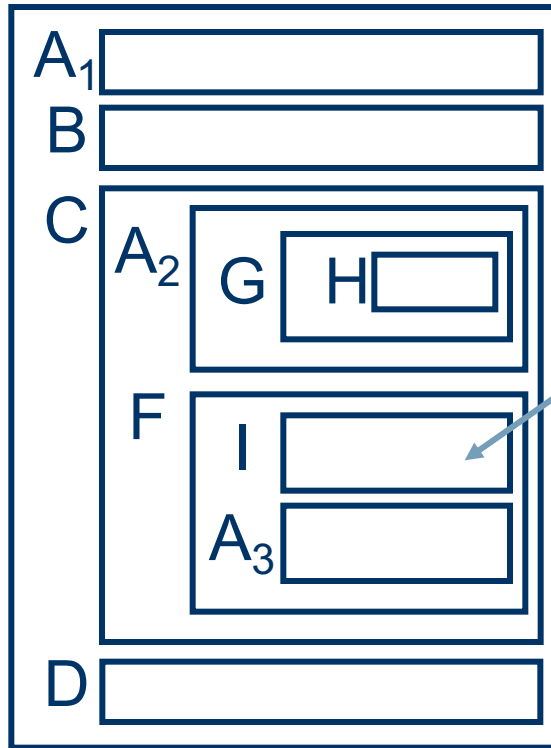


Letztes Element:

Streutabelle:



Definitionstabelle mit Streuspeicherung und integriertem Stapel am Beispiel (9)

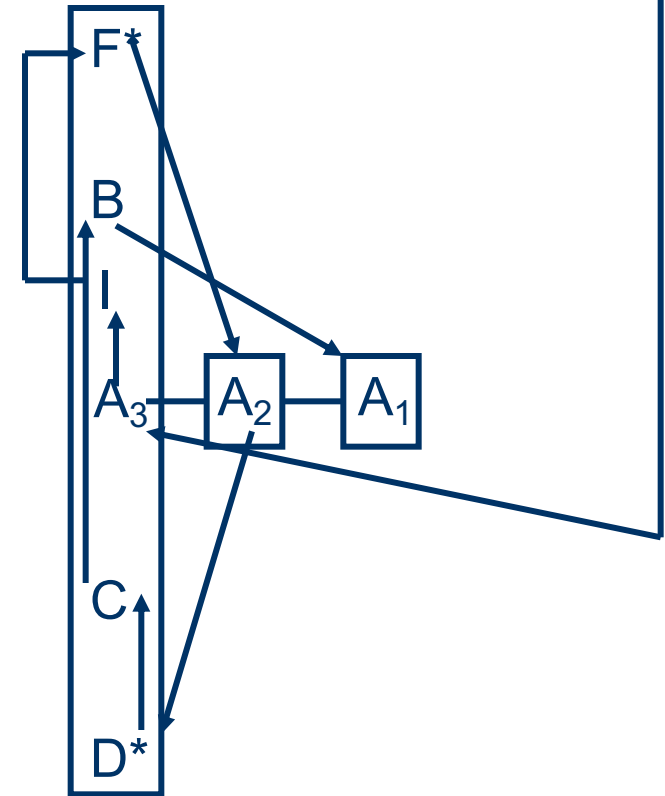


sichtbar:

A₃
B
C
D
F
I

Letztes Element:

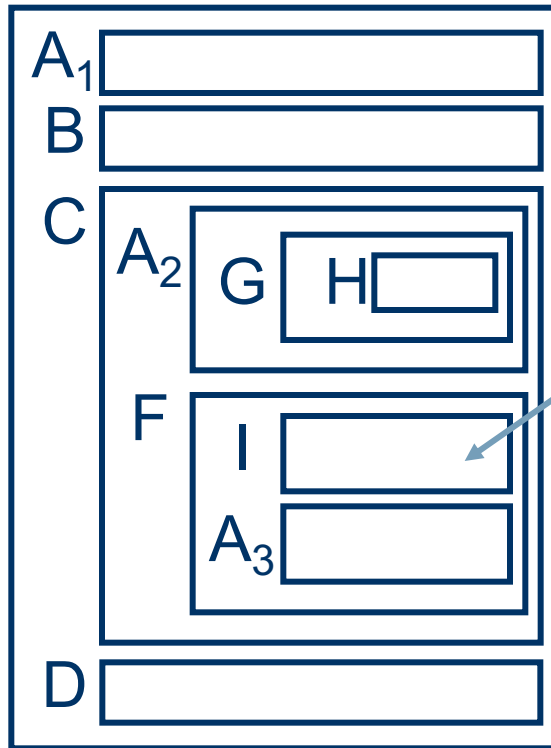
Streutabelle:



Übliche Implementierung der Definitionstabelle

- Geschichtete Streutabellen:
 - Alle Bezeichner einer Ebene in die Streutabelle eintragen.
 - Ist ein Eintrag schon vorhanden: Fehler.
 - Beginnt eine neue Sichtbarkeitsebene, dann:
 - Neue Streutabelle anlegen, die auf ihre Vorgängertabelle verweist.
 - Bezeichner in die neue Streutabelle eintragen.
 - Nachschlagen:
 - Ein Bezeichner wird immer in der jüngsten Streutabelle gesucht.
 - Ist er dort nicht zu finden, wird schrittweise in den Vorgängertabellen nachgeschlagen.
 - Kein Eintrag in erster Streutabelle: Fehler.
 - Endet eine Sichtbarkeitsebene, dann wird die oberste Streutabelle verworfen.

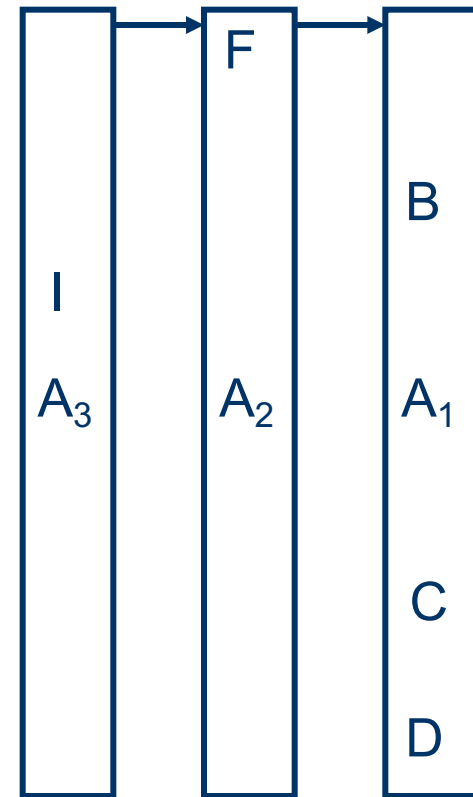
Definitionstabelle mit geschichteter Streuspeicherung am Beispiel



sichtbar:


A₃
B
C
D
F
I

Streutabellen:



- Vorteile:
 - Keine Verwaltung des konzeptuellen Stapels auf Eintrageebene (sondern gröber: Sichtbarkeitsebene).
 - Streutabellen haben im Allg. nur wenige Einträge, wachsen mit Sichtbarkeitsebene.
- Nachteil:
 - Sequentielles Nachschlagen in mehreren Streutabellen.

Definitionstabellen in javac (1)

- 
- `class Symbol (kind, flags, name, type, owner, ...)`
 - `class TypeSymbol`
 - `class PackageSymbol (memberfields, ...)`
 - `class ClassSymbol (memberfields, sourcefile, classfile, pool, ...)`
 - `class VarSymbol (adr, constantValue, ...)`
 - `class MethodSymbol (code, ...)`
 - `class OperatorSymbol (opcode, ...)`
 - Mehrere Definitionstabellen als Streutabellen mit Verketteten und durchgefädeltem konzeptuellen Keller:
 - für Klassen-Namen,
 - für statische und Instanz-Variablen von Klassen,
 - für Methodendefinitionen von Klassen.

Definitionstabellen in javac (2)

Mehrere Schritte:

- Vorab-Befüllung der Definitionstabellen mit vordefinierten Dingen (`java.lang.Object`, `java.lang.String`, `+-Operator`, primitive Typen, ...).
- Eintragen der Klassen (Besucher)
 - Für `ClassDef`-Knoten werden Symbol-Objekte angelegt.
 - Überprüfungen:
 - Klassenname muss mit Dateinamen übereinstimmen.
 - Klasse/Interface darf nur einmal definiert werden.
 - Keine zyklische Vererbung.
 - Erlaubte Flags.
 - ...
- Laden der Methoden und Variablen der bereits vorübersetzten Klassen.
- Eintragen der im zu übersetzenden Code deklarierten Methoden, Variablen (Besucher).
 - Zweischrittig: Deklarationen von Verwendungen getrennt.
 - Für `VarDef`- und `MethodDef`-Knoten werden Symbol-Objekte angelegt
 - Überprüfungen:
 - Erlaubte Flags
 - Doppel-Deklaration
 - ...