Topologische Sortierung

- Input: gerichteter, kreisfreier Graph
- Output: Knotenliste, deren Ordnung mit der Richtung der Kanten übereinstimmt
- Algorithmus (Funktionsargumente unterstrichen):

```
FUNCTION Topological sort of (a set Set) RETURNING a list:
    SET List TO Empty list;
    WHILE there is a Node in Set but not in List:
        Append Node and its predecessors to List;
    RETURN List;

PROCEDURE Append Node and its predecessors to List:
    // First append the predecessors of Node:
    FOR EACH Node_1 IN the Set of nodes that Node is dependent on:
        IF Node_1 is not in List:
            Append Node_1 and its predecessors to List;
    Append Node to List;
```

Figure 3.16 Outline code for a simple implementation of topological sort.

Zyklenerkennung (1)

- Dynamisch:
  - Nachweis durch Überwachung der Attributauswertung
  - erkennt Zyklus nur für die *jeweils durchgeführte* aktuelle Herleitung
  - mehr Durchläufe als Attribute ⇒ es existiert ein Zyklus
- Statisch:
  - analysiert die attributierte Grammatik
  - erkennt Abwesenheit jeglicher Zyklen *in allen möglichen* Herleitungen
- Quellen von Zyklen: (Abb. 3.17)
  - eine einzelne Produktion kann keinen Zyklus enthalten
    (die gelesenen und geschriebenen Attributinstanzen sind disjunkt)
  - zwei Arten von Abhängigkeiten (Abb. 3.18)
    - IS (inh.→syn.) von allen Unterbäumen des Knotens
    - SI (syn.→inh.) von allen Bäumen, zu denen der Knoten gehört

Zyklenerkennung (2)

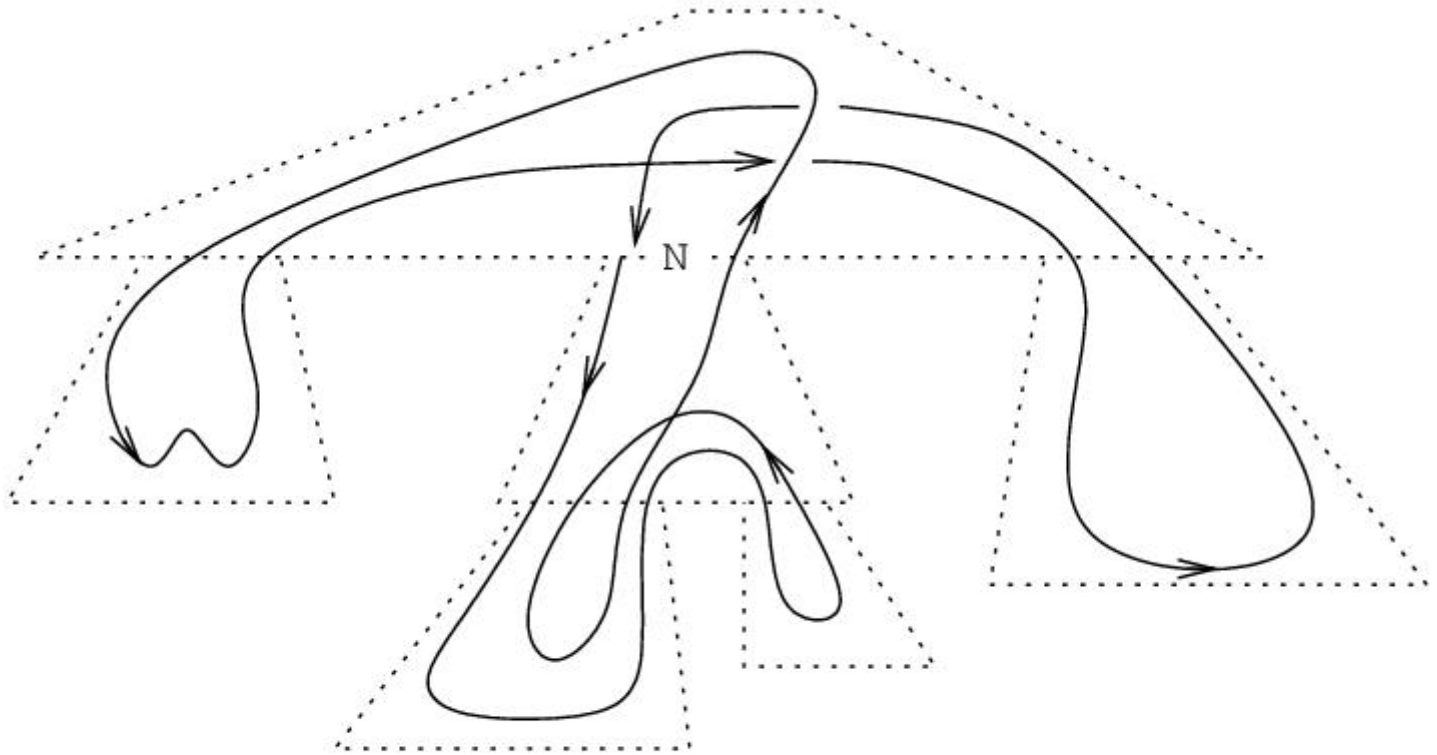


Figure 3.17 A fairly long, possibly circular, data-flow path.

Zyklenerkennung (3)

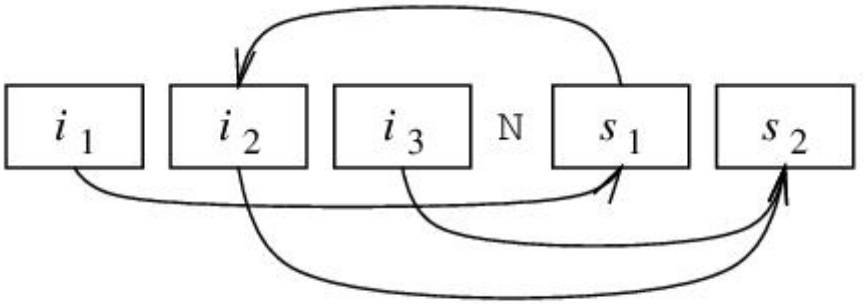


Figure 3.18 An example of an IS-SI graph.

Zyklenerkennung (4)

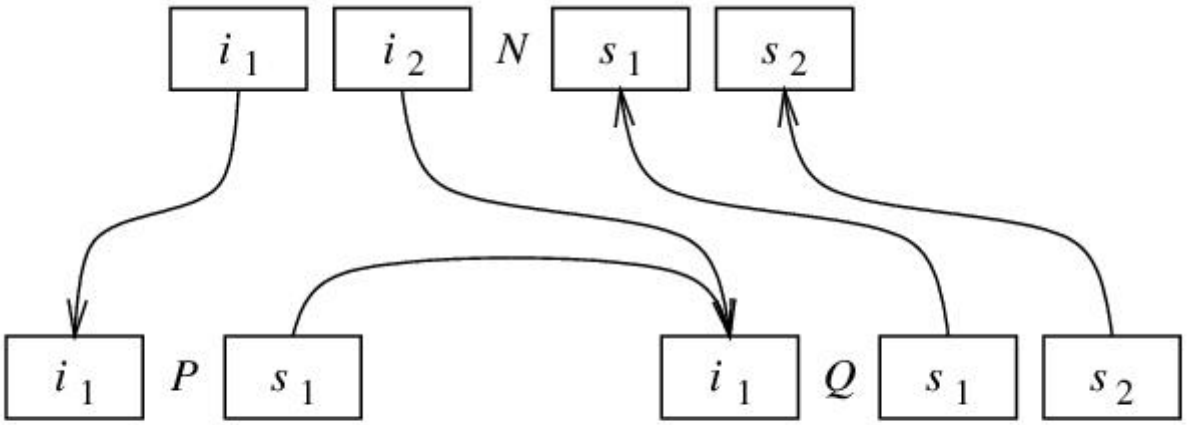


Figure 3.19 The dependency graph for the production rule $N \rightarrow PQ$.

IS-SI-Graphenkonstruktion

- IS und SI-Abhängigkeiten für einzelne Produktion: N→PQ
  - gegeben:
    - Abhängigkeiten für die Produktion N→PQ
    - IS-SI-Graphen für P und Q / für N
  - bilde transitive Hülle (deckt möglichen Zyklus auf)
- IS-SI-Graph für die gesamte Grammatik: (Abb. 3.20)
  - Initialisierung: alle Nichtterminale haben leere IS-SI-Graphen
  - bis ein Fixpunkt erreicht ist
    1. kopiere den Abhängigkeitsgraphen einer Produktion
    2. ergänze die Abhängigkeiten der IS-SI-Graphen der beteiligten Grammatiksymbole
    3. bilde die transitive Hülle, falls Zyklus: Fehler
    4. ergänze neue Abhängigkeiten in den IS-SI-Graphen der beteiligten Grammatiksymbole

```
SET the flag Something was changed TO True;

// Initialization step:
FOR EACH terminal T IN Attribute grammar:
    SET the IS-SI graph of T TO T's dependency graph;

FOR EACH non-terminal N IN Attribute grammar:
    SET the IS-SI graph of N TO the empty set;

WHILE Something was changed:
    SET Something was changed TO False;

    FOR EACH production rule P = M_0→M_1...M_n IN Attribute grammar:

        // Construct the dependency graph copy D:
        SET the dependency graph D TO a copy of the dependency graph of P;
        // Add the dependencies already found for M_{i=0...n}:
        FOR EACH M IN M_0...M_n:
            FOR EACH dependency d IN the IS-SI graph of M:
                Insert d in D;

        // Use the dependency graph D:
        Compute all induced dependencies in D by transitive closure;
        IF D contains a cycle: ERROR "Cycle found in production", P
        // Propagate the newly discovered dependencies:
        FOR EACH M IN M_0...M_n:
            FOR EACH dependency d IN D
                SUCH THAT the attributes in d are attributes of M:
                IF there is no dependency d in the IS-SI graph of M:
                    Enter a dependency d into the IS-SI graph of M;
                    SET Something was changed TO True;
```

Figure 3.20 Outline of the strong-cyclicity test for an attribute grammar.

IS-SI-Graphenkonstruktion

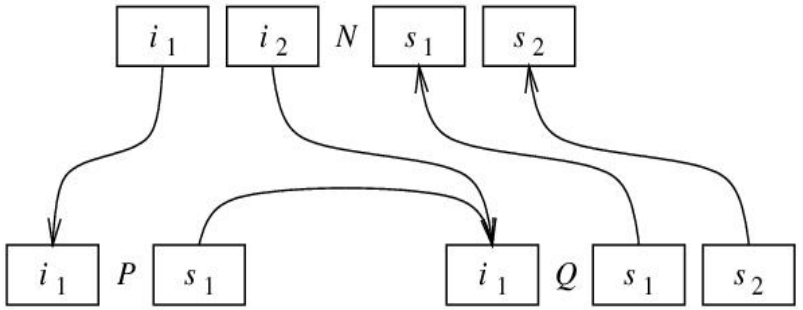


Figure 3.19 The dependency graph for the production rule $N \rightarrow PQ$.

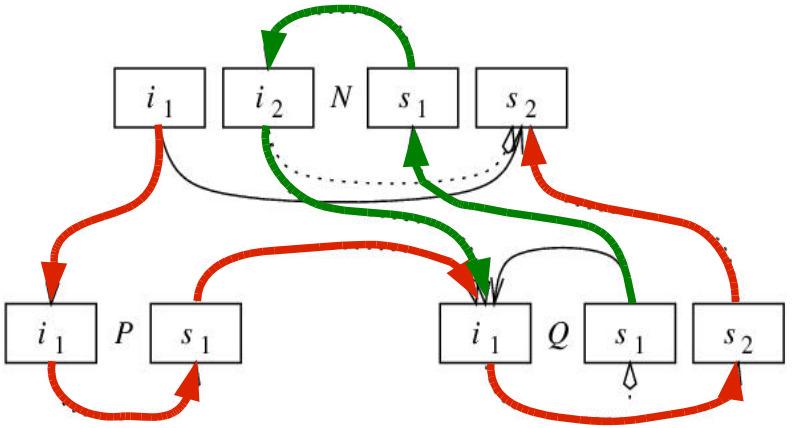
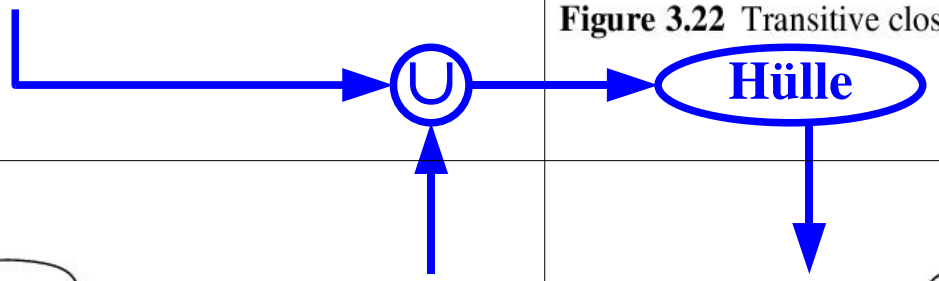


Figure 3.22 Transitive closure over the dependencies of N, P, Q and D.



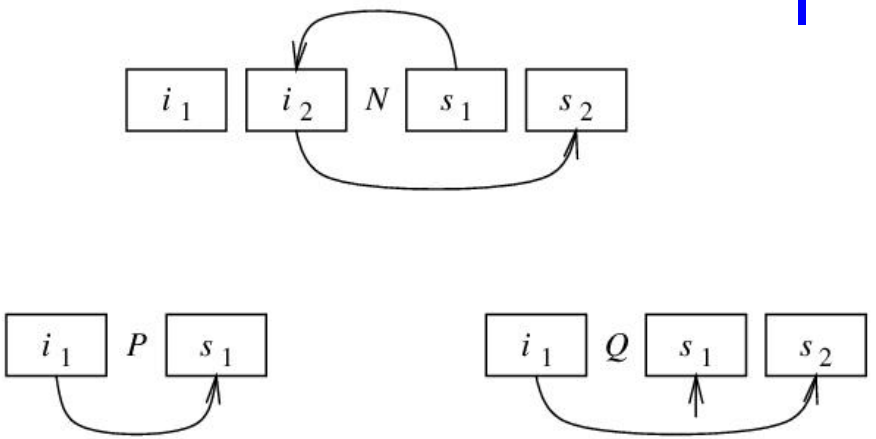


Figure 3.21 The IS-SI graphs of N, P, and Q collected so far.

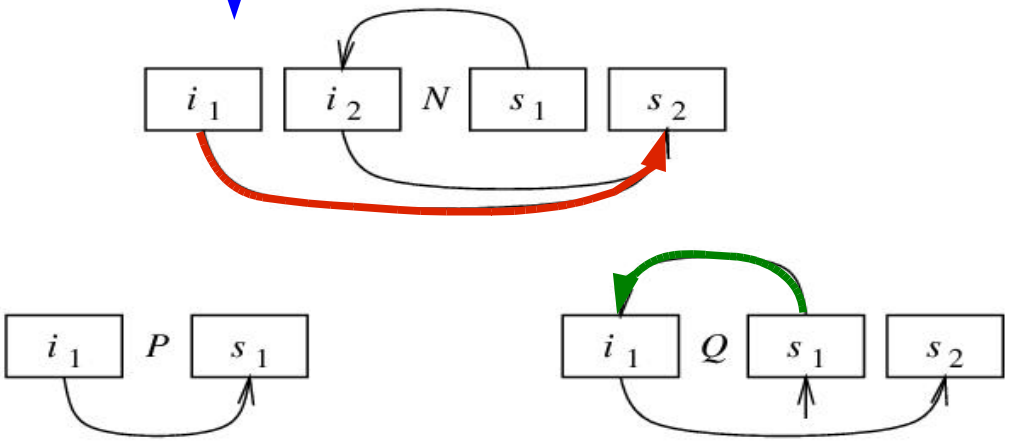


Figure 3.23 The new IS-SI graphs of N, P, and Q.

IS-SI-Graphenkonstruktion

- Garantie:
  - Jeder existierende Abhängigkeitszyklus wird gefunden (Beweis durch Widerspruch)
- Nachteil: pessimistische Analyse
  - es kann ein Zyklus konstruiert werden, der in keiner konkreten Situation auftreten kann
  - Grund: "globale" Konstruktion, an jedem Knoten Vereinigung der Abh. aus allen Produktionen eines NT, obwohl nur eine relevant
  - führt zu Flusskombinationen, die nicht existieren müssen (Drachenbuch, Bsp. 5.32)

  ⇒ konstruierte Graphen sind *streng zyklenfrei*
- Verfeinerung des Verfahrens
  - Beschreibe jedes Nichtterminal durch eine Mengen von IS-SI-Graphen (Drachenbuch, Abb. 5.62)
  - Jeder IS-SI-Graph beschreibt einen Fluss, der existiert
  - Analogie: LR(1)-Lookaheads statt FOLLOW-Mengen

  ⇒ konstruierte Graphen sind *zyklenfrei*

Multi-Visit Grammatiken

- Zwei Arten von Knotenbesuchen für Regel $N \rightarrow M_1,...,M_n$:
  - Besuch eines Kindknotens M_i von N:
    - übergib eine Menge ererbter Attributwerte
    - besuche M_i
    - sammle eine Menge synthetisierter Attributwerte
  - Besuch des Elternknotens der Regel:
    - übergib eine Menge synthetisierter Attributwerte
    - besuche Elternknoten
    - sammle eine Menge ererbter Attributwerte
- Partitionierung der Attribute der Regel $N \rightarrow M_1,...,M_n$:
  - Mengenpaar (IN_i, SN_i): Attribute für i-ten Besuch des Regelknotens (IN_1 u.U. leer)
  - Vorgabe einer akzeptablen Partitionierung ermöglicht eine automatische Erstellung des Evaluierers

Multi-Visit Grammatiken

- Schema (Abb. 3.24)
- Beispiel: (Drachenbuch, Abb. 5.40-41)
- Pro Produktionsregel $N \rightarrow M_1,...,M_n$ (statisch):
  - für den j-ten Besuch von N eine separate Routine
- Pro Nichtterminal (dynamische Fallunterscheidung):
  - Auswahl der für den aktuellen Knoten relevanten Produktion (Abb. 3.26)

```
// Visit 1 from the parent: flow of control from parent enters here.
// The parent has set some inherited attributes, the set IN_1.

    // Visit some children M_k, M_l, ...:

    Compute some inherited attributes of M_k, the set (IM_k)_1;
    Visit M_k for the first time;
    // M_k returns with some of its synthesized attributes evaluated.

    Compute some inherited attributes of M_l, the set (IM_l)_1;
    Visit M_l for the first time;
    // M_l returns with some of its synthesized attributes evaluated.

    ... // Perhaps visit some more children, including possibly M_k or
        // M_l again, while supplying the proper inherited attributes
        // and obtaining synthesized attributes in return.

    // End of the visits to children.

    Compute some of N's synthesized attributes, the set SN_1;
    Leave to the parent;
// End of visit 1 from the parent.

// Visit 2 from the parent: flow of control re-enters here.
// The parent has set some inherited attributes, the set IN_2.

    ... // Again visit some children while supplying inherited
        // attributes and obtaining synthesized attributes in return.

    Compute some of N's synthesized attributes, the set SN_2;
    Leave to the parent;
// End of visit 2 from the parent.

... // Perhaps code for some more visits 3..n from the parent,
    // supplying sets IN_3 to IN_n and yielding
    // sets SN_3 to SN_n.
```

Figure 3.24 Outline code for multi-visit attribute evaluation.

Multi-Visit Grammatiken

... i-ter Besuch von N (Alternative k) ...

- Bekannt:
  - $next\_visit\_number[M_x]$ f.a. M_x in der k-*ten* Produktion für N
  - Menge E der bereits evaluierten Attribute
  - IN_i
- Aktionen:
  - $j = next\_visit\_number[M_x]$
  - selektiere die M_x, für die Attribute evaluiert werden
  - Voraussetzung: $(IM_x)_j$ muss sich über E berechnen lassen
  - rekursive Codegenerierung für die selektierten Kindknoten
  - Codegenerierung für die Evaluierung der synthetisierten Attribute in SN_i

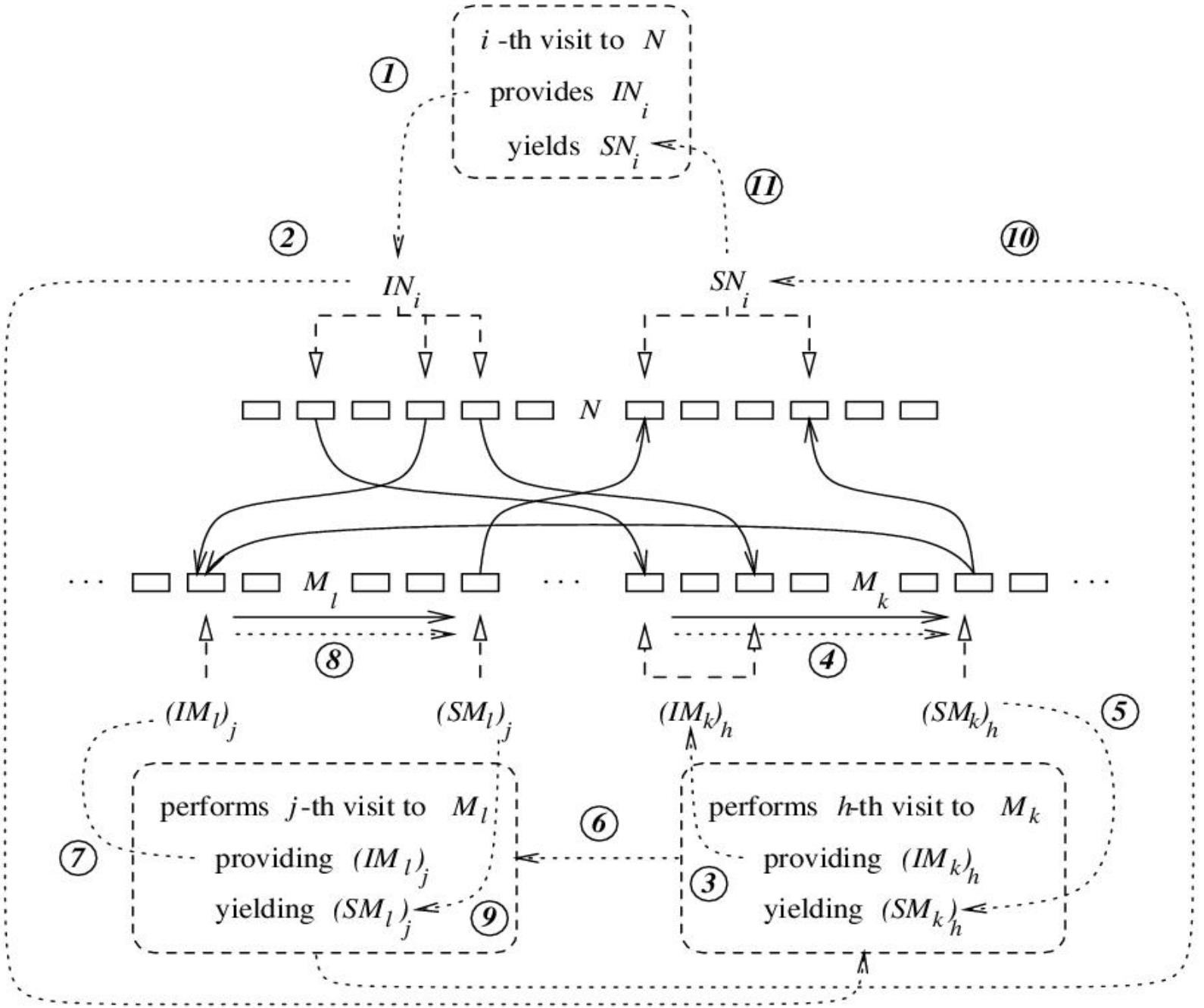


Figure 3.25 The i-th visit to a node N, visiting two children, M_k and M_l.

Code für den *i*-ten Besuch von *N*

```
PROCEDURE Visit_i to N (pointer to an N node Node):
    SELECT Node .type:
        CASE alternative_1:
            Visit_i to N alternative_1 (Node);
        ...
        CASE alternative_k:
            Visit_i to N alternative_k (Node);
        ...
```

Figure 3.26 Structure of an *i*-th visit routine for *N*.

Akzeptable Partitionierungen

- Partitionierung induziert Datenabhängigkeiten:
  - Attribute in IN_i vor Attributen in SN_i
  - Attribute in SN_i vor IN_{i+1}
- Akzeptabilitätstest:
  - füge die durch die Partitionierung induzierten Abhängigkeiten zu den IS-SI-Graphen aller Produktionen hinzu
  - teste auf Zyklen
- Generierung einer akzeptablen Partitionierung:
  - Heuristik für *geordnete* attributierte Grammatiken
  - lineare Zeitkomplexität

Geordnete attributierte Grammatiken

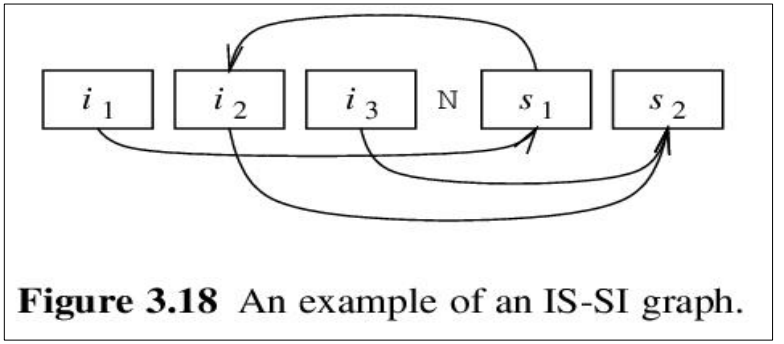


Figure 3.18 An example of an IS-SI graph.

- Konstruktionsidee:
  - akzeptable Partitionierung = IS-SI-Graph+Abhängigkeiten
  - Bsp.: (Abb. 3.18)
    - Partitionierung 1 = $(\{i_1,i_3\},\{s_1\}),(\{i_2\},\{s_2\})$
    - Partitionierung 2 = $(\{i_1\},\{s_1\}),(\{i_2,i_3\},\{s_2\})$
  - Heuristik: spätere Evaluierung ⇒ geringere Zyklengefahr
- Konstruktion der Partitionierung: von der IS-SI-Senke zur Quelle
  - zuletzt auszuwertende Attribute: in SN_{last} ($last$ zunächst unbekannt)
  - entferne Attribute in SN_{last} aus dem IS-SI-Graphen
  - jetzt zuletzt auszuwertende Attribute: in IN_{last}
  - entferner Attribute in IN_{last} aus dem IS-SI-Graphen
  - weiter mit ($IN_{last\text{-}1}$, $SN_{last\text{-}1}$), etc.

In der Praxis

- Zahl der Besuche eines Knotens:
  - in den meisten Fällen: 1
  - gelegentlich: 2
  - sehr selten: 3
- Bsp.: wann werden überhaupt z.B. 10 Besuche notwendig?
  - ≥ 19 disjunkte Mengen $IN_{i,i=2..10}$, $SN_{i,i=1..10}$

  ⇒ ≥ 9 ererbte Attribute

  ⇒ ≥ 10 synthetisierte Attribute

Bsp.: Oktal/Dezimal-Konstanten (1)

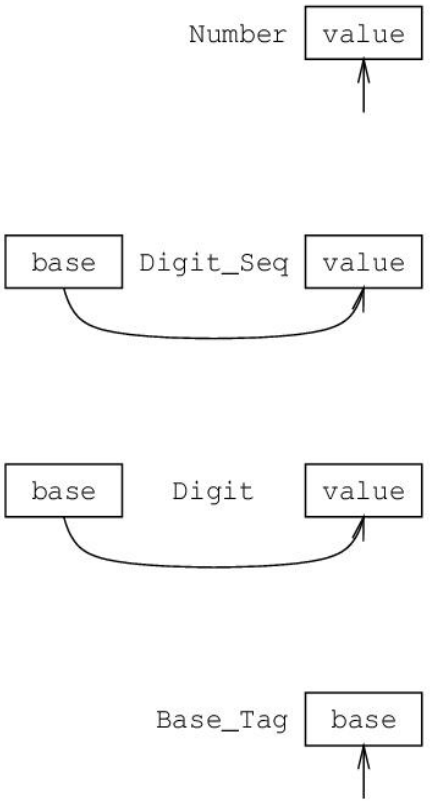
```
Number(SYN value) →
    Digit_Seq  Base_Tag
    ATTRIBUTE RULES
        SET Digit_Seq .base TO Base_Tag .base;
        SET Number .value TO Digit_Seq .value;

Digit_Seq(INH base, SYN value) →
    Digit_Seq[1]  Digit
    ATTRIBUTE RULES
        SET Digit_Seq[1] .base TO Digit_Seq .base;
        SET Digit .base TO Digit_Seq .base;
        SET Digit_Seq .value TO
            Digit_Seq[1] .value * Digit_Seq .base + Digit .value;
|
    Digit
    ATTRIBUTE RULES
        SET Digit .base TO Digit_Seq .base;
        SET Digit_Seq .value TO Digit .value;

Digit(INH base, SYN value) →
    Digit_Token
    ATTRIBUTE RULES
        SET Digit .value TO Checked digit value (
            Value_of (Digit_Token .repr [0]) - Value_of ('0'),
            base
        );

Base_Tag(SYN base) →
    'B'
    ATTRIBUTE RULES
        SET Base_Tag .base TO 8;
|
    'D'
    ATTRIBUTE RULES
        SET Base_Tag .base TO 10;
```

Figure 3.8 An attribute grammar for octal and decimal numbers.



3.27 The IS-SI graphs of the non-terminals from grammar 3.8.

Bsp.: Oktal/Dezimal-Konstanten (2)

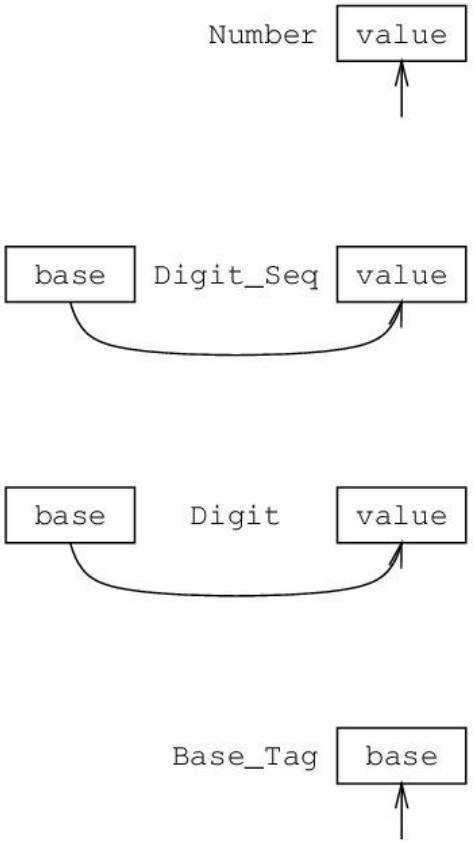


Figure 3.27 The IS-SI graphs of the non-terminals from grammar 3.8.

	IN_1	SN_1
Number		value
Digit_Seq	base	value
Digit	base	value
Base_Tag		base

Figure 3.28 Partitionings of the attributes of grammar 3.8.

Bsp.: Oktal/Dezimal-Konstanten (3)

- Grammatik: (Abb. 3.8)
- IS-SI-Graphen der Nichtterminale: (Abb. 3.27)
  - zyklenfrei
  - ohne SI-Abhängigkeiten
- Partitionen: (Abb. 3.28)
- Code für den (einzigen) Besuch von `Number`: (Abb. 3.29)
  - vorher evaluierte Attribute: $E = \{\ \}$
  - $next\_visit\_number_{\texttt{Digit\_Seq}} = next\_visit\_number_{\texttt{Base\_Tag}} = 0$
  - Code für den Besuch von `Digit_Seq` und `Base_Tag` (nächste Seite)

Bsp.: Oktal/Dezimal-Konstanten (4)

- Besuch von **Digit_Seq**:

```
// Requires Number_alt_1.Base_Tag.base to be set
// Compute the attributes in IN1 of Digit_Seq(), the set {base}
SET Number_alt_1.Digit_Seq.base TO Number_alt_1.Base_Tag.base;
// Visit Digit_Seq for the first time:
Visit_1_to_Digit_Seq (Number_alt_1.Digit_Seq);
// Digit_Seq returns with its SN1, the set {value}, evaluated;
// it supplies Number_alt_1.Digit_Seq.value.
```

- Besuch von **Base_Tag**:

```
// Requires nothing
// Compute the attributes in IN1 of Base_Tag(), the set {}:
// Visit Base_Tag for the first time:
Visit_1_to_Base_Tag (Number_alt_1.Base_Tag);
// Base_Tag returns with its SN1, the set {base}, evaluated;
// it supplies Number_alt_1.Base_Tag.base
```

- Reihenfolge:

```
  E = {}                                          ⇒ Besuch von Base_Tag
⇒ E = {Number_alt_1.Base_Tag.base}                ⇒ Besuch von Digit_Seq
⇒ E = {Number_alt_1.Base_Tag.base, Number_alt_1.Digit_Seq.value}
```

Bsp.: Oktal/Dezimal-Konstanten (5)

- Grammatik: (Abb. 3.8)
- IS-SI-Graphen der Nichtterminale: (Abb. 3.27)
  - zyklenfrei
  - ohne SI-Abhängigkeiten
- Partitionen: (Abb. 3.28)
- Code für den (einzigen) Besuch von `Number`: (Abb. 3.29)
  - vorher evaluierte Attribute: $E = \{\ \}$
  - $next\_visit\_number_{\texttt{Digit\_Seq}} = next\_visit\_number_{\texttt{Base\_Tag}} = 0$
  - Code für den Besuch von `Digit_Seq` und `Base_Tag`
    - Code für Synthetisierung von `Number.value`
      (über `Number_alt_1.Digit_Seq.value` aus E)
- Code für den Besuch der ersten Alternative von `Digit_Seq`:
  (Abb. 3.30; vgl. Abb. 3.14)

```
PROCEDURE Visit_1 to Number alternative_1 (
    pointer to number node Number,
    pointer to number alt_1 node Number alt_1
):
// Visit 1 from the parent: flow of control from the parent enters here.
// The parent has set the attributes in IN1 of Number, the set { }.

    // Visit some children:

    // Compute the attributes in IN1 of Base_Tag (), the set { }:
    // Visit Base_Tag for the first time:
    Visit_1 to Base_Tag (Number alt_1 .Base_Tag);
    // Base_Tag returns with its SN1, the set { base }, evaluated.

    // Compute the attributes in IN1 of Digit_Seq (), the set { base }:
    SET Number alt_1 .Digit_Seq .base TO Number alt_1 .Base_Tag .base;
    // Visit Digit_Seq for the first time:
    Visit_1 to Digit_Seq (Number alt_1 .Digit_Seq);
    // Digit_Seq returns with its SN1, the set { value }, evaluated.

    // End of the visits to children.

    // Compute the attributes in SN1 of Number, the set { value }:
    SET Number .value TO Number alt_1 .Digit_Seq .value;
```

Figure 3.29 Visiting code for `Number` nodes.

```
PROCEDURE Visit_1 to Digit_Seq alternative_1 (
    pointer to digit_seq node Digit_Seq,
    pointer to digit_seq alt_1 node Digit_Seq alt_1
):
// Visit 1 from the parent: flow of control from the parent enters here.
// The parent has set the attributes in IN1 of Digit_Seq, the set { base }

    // Visit some children:

    // Compute the attributes in IN1 of Digit_Seq (), the set { base }:
    SET Digit_Seq alt_1 .Digit_Seq .base TO Digit_Seq .base;
    // Visit Digit_Seq for the first time:
    Visit_1 to Digit_Seq (Digit_Seq alt_1 .Digit_Seq);
    // Digit_Seq returns with its SN1, the set { value }, evaluated.

    // Compute the attributes in IN1 of Digit (), the set { base }:
    SET Digit_Seq alt_1 .Digit .base TO Digit_Seq .base;
    // Visit Digit for the first time:
    Visit_1 to Digit (Digit_Seq alt_1 .Digit);
    // Digit returns with its SN1, the set { value }, evaluated.

    // End of the visits to children.

    // Compute the attributes in SN1 of Digit_Seq, the set { value }:
    SET Digit_Seq .value TO
        Digit_Seq alt_1 .Digit_Seq .value * Digit_Seq .base +
        Digit_Seq alt_1 .Digit .value;
```

Figure 3.30 Visiting code for `Digit_Seq alternative_1` nodes.

Konstruktion eines Auswerters

- Konstruktion:
  - Bilde die IS-SI-Graphen mit Tests auf Zyklenfreiheit
  - Partitioniere die IS-SI-Graphen mit as-late-as-possible Strategie
  - Generiere Code für die Besucherroutinen
    - beginne mit SN_{last} und gehe rückwärts vor (anforderungsgetrieben)
    - benutze die Datenabhängigkeiten und die IN- und SN-Mengen
- Korrektheit:
  - Wenn alle Codestücke erstellt werden können, ohne die Datenabhängigkeiten zu verletzen, ist die Grammatik geordnet

Hierarchie attributierter Grammatiken

... mit Worst-Case-Zeitkomplexitäten für Tests ...

Sukzessive Einschränkungen (echte Inklusionen: 1.⊃2.⊃3.⊃4.⊃5.)

1. wohlgeformt: pro Produktion Wertezuweisung an
   - alle synthetisierten Attribute
   - alle ererbten Attribute von Symbolen der rechten Seite
2. zyklenfrei:
   - Test exponenziell in der Attributzahl eines Nichtterminals (machbar)
3. streng zyklenfrei:
   - Test linear in der Attributzahl eines Nichtterminals
4. Multi-Visit:
   - Test exponenziell in der gesamten Attributzahl (nicht machbar)
5. geordnet (d.h., mit akzeptabler Partitionierung)
   - Test quadratisch in der Attributzahl eines Nichtterminals

Statische Attributauswertung

- Nach dem Parsevorgang:
  - Multi-Visit Grammatiken
  - mehrfache Baumdurchläufe möglich
- Mit dem Parsevorgang:
  - S-attributierte Grammatiken (nur synthetisierte Attribute)
  - L-attributierte Grammatiken (auch eingeschränkte ererbte Attribute)
  - ein Baumdurchlauf muss ausreichen, der des Parsevorgangs
- LL-Parsing:
  - Attributwerte in lokalen Variablen
  - Berechnung vor/zwischen/nach rekursiven Aufrufen
- LR-Parsing:
  - Attributwerte auf dem Stack (vgl. Drachenbuch, Bsp. 5.10)
  - Berechnung nur bei einer Reduktion möglich