Hüllenalgorithmen für Scanner

- Hüllenalgorithmen
  - behandeln Informationsgewinnung in rekursiven Strukturen
  - an mehreren Stellen in der Vorlesung verwendet, z.B. in der Scannergenerierung
- Scannergenerierung
  - reguläre Ausdrücke und EBNF
  - automatische Scannergenerierung
    - dotted items (reg. Ausdrücke mit Positionsmarke)
    - Konstruktion eines erkennenden Automaten

Hüllenalgorithmen

Informationsgewinnung: lokale Inf. $\rightarrow$ globale Inf.

- Allgemeine Form
  - Datendefinitionen: Semantik der Informationsteile
  - Initialisierungen: Regeln zur Generierung der Anfangswerte
  - Inferenzregeln: Regeln zur Gewinnung neuer Information
- Beispiel: Erkennung rekursiver Funktionen
  - Datendefinitionen: Kanten im Aufrufgraph
  - Initialisierungen: A enthält Aufruf von B $\Rightarrow$ A$\rightarrow$B
  - Inferenzregeln: A$\rightarrow$B $\wedge$ B$\rightarrow$C $\Rightarrow$ A$\rightarrow$C
- Merke:
  - keine zusätzliche Information: *kleinster* Fixpunkt!
  - Terminationsproblem! Lösung: *bottom-up*-Berechnung

Data definitions:

1. Let G be a directed graph with one node for each routine. The information items are arrows in G.
2. An arrow from a node A to a node B means that routine A calls routine B directly or indirectly.

Initializations:

If the body of a routine A contains a call to routine B, an arrow from A to B must be present.

Inference rules:

If there is an arrow from node A to node B and one from B to C, an arrow from A to C must be present.

Figure 1.31 Recursion detection as a closure algorithm.

```
void P() { ... Q(); .... S(); ... }
void Q() { ... R(); .... T(); ... }
void R() { ... P(); }
void S() { ... }
void T() { ... }
```

Figure 1.28 Sample C program used in the construction of a calling graph.

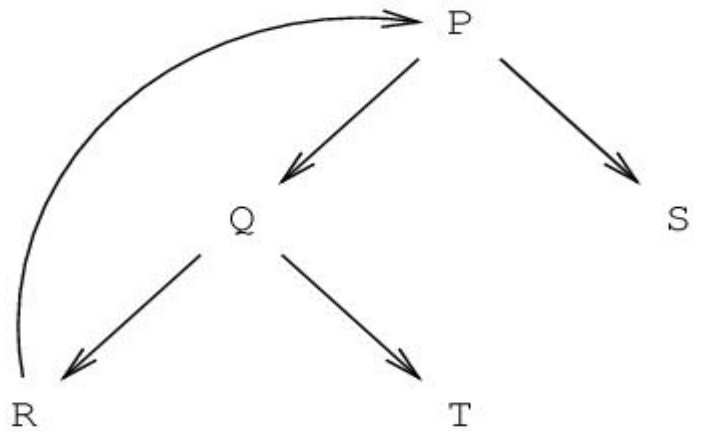

Figure 1.29 Initial (direct) calling graph of the code in Figure 1.28.

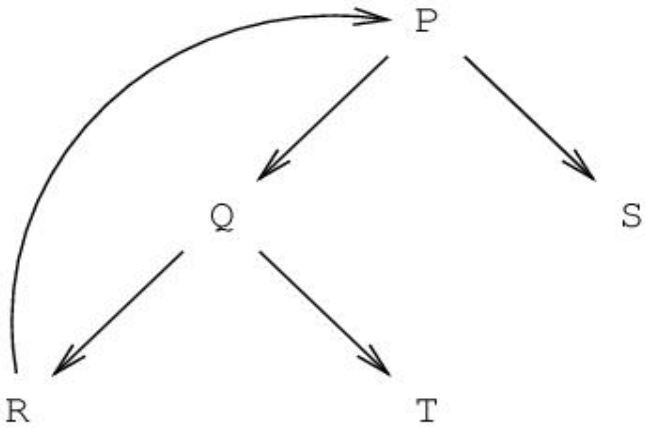

Figure 1.29 Initial (direct) calling graph of the code in Figure 1.28.

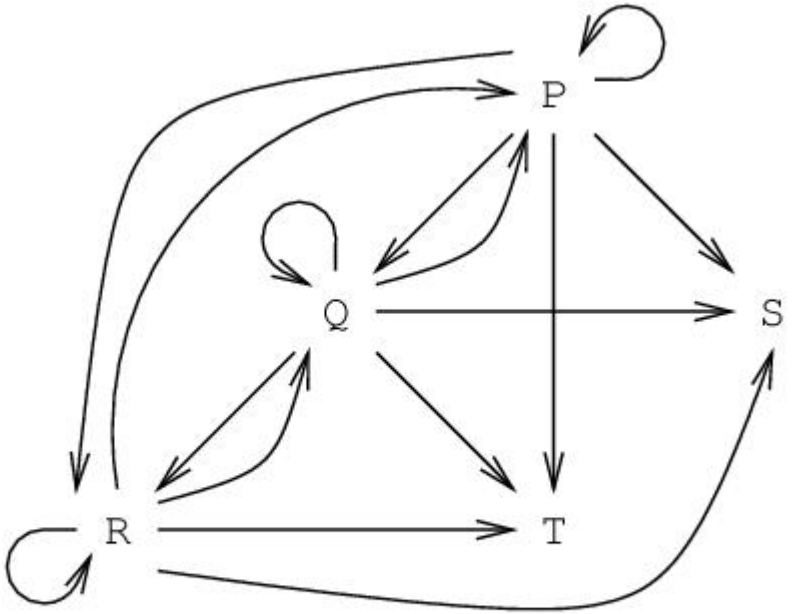

Figure 1.30 Calling graph of the code in Figure 1.28.

Naive Implementierung

folgendes Prolog-Programm zur Berechnung der transitiven Hülle gerät in eine unendliche Rekursion:

```
calls(A, C) :- calls(A, B), calls(B, C).
calls(a, b).
calls(b, a).
:-? calls(a, a).
```

Figure 1.32 A Prolog program corresponding to the closure algorithm of Figure 1.31.

Sicheres Bottom-Up-Verfahren

```
SET the flag Something changed TO True;

WHILE Something changed:
    SET Something changed TO False;

    FOR EACH Node 1 IN Graph:
        FOR EACH Node 2 IN Descendants of Node 1:
            FOR EACH Node 3 IN Descendants of Node 2:
                IF there is no arrow from Node 1 to Node 3:
                    Add an arrow from Node 1 to Node 3;
                    SET Something changed TO True;
```

Figure 1.33 Outline of a bottom-up algorithm for transitive closure.

Lexikalische Analyse (Scanning)

- Token
  - kann mit Blanks separiert werden
String	3+9	:=	“Ich und du”	(* Kommentar *)
Tokens	3, +, 9	:=	“Ich und du”	
- Tokensprache:
  - regulärer Ausdruck (Abb. 2.4)
- Reguläre Beschreibung:
  - EBNF-Grammatik ohne Rekursion

```
letter          → [a-zA-Z]
digit           → [0-9]
underscore      → _
letter_or_digit → letter | digit
underscore_tail → underscore letter_or_digit
identifier      → letter letter_or_digit underscore_tail
```

Induktive Definition regulärer Ausdrücke

Basic patterns:	*Matching:*
x	The character x
.	Any character, usually except a newline
$[xyz...]$	Any of the characters x, y, z, ...
Repetition operators:	
$R^?$	An R or nothing (= optionally an R)
R^*	Zero or more occurrences of R
R^+	One or more occurrences of R
Composition operators:	
$R_1 R_2$	An R_1 followed by an R_2
$R_1 \mid R_2$	Either an R_1 or an R_2
Grouping:	
(R)	R itself

Figure 2.4 Components of regular expressions.

Naiver (ineffizienter) Scanner

```
SET the global token (Token .class, Token .length) TO (0, 0);

// Try to match token description T1→R1:
FOR EACH Length SUCH THAT the input matches T1→R1 over Length:
    IF Length > Token .length:
        SET (Token .class, Token .length) TO (T1, Length);

// Try to match token description T2→R2:
FOR EACH Length SUCH THAT the input matches T2→R2 over Length:
    IF Length > Token .length:
        SET (Token .class, Token .length) TO (T2, Length);

...

FOR EACH Length SUCH THAT the input matches Tn→Rn over Length:
    IF Length > Token .length:
        SET (Token .class, Token .length) TO (Tn, Length);

IF Token .length = 0:
    Handle non-matching character;
```

Figure 2.16 Outline of a naive generated lexical analyzer.

Effizientes Verfahren

Idee: Aufspaltung des regulären Ausdrucks

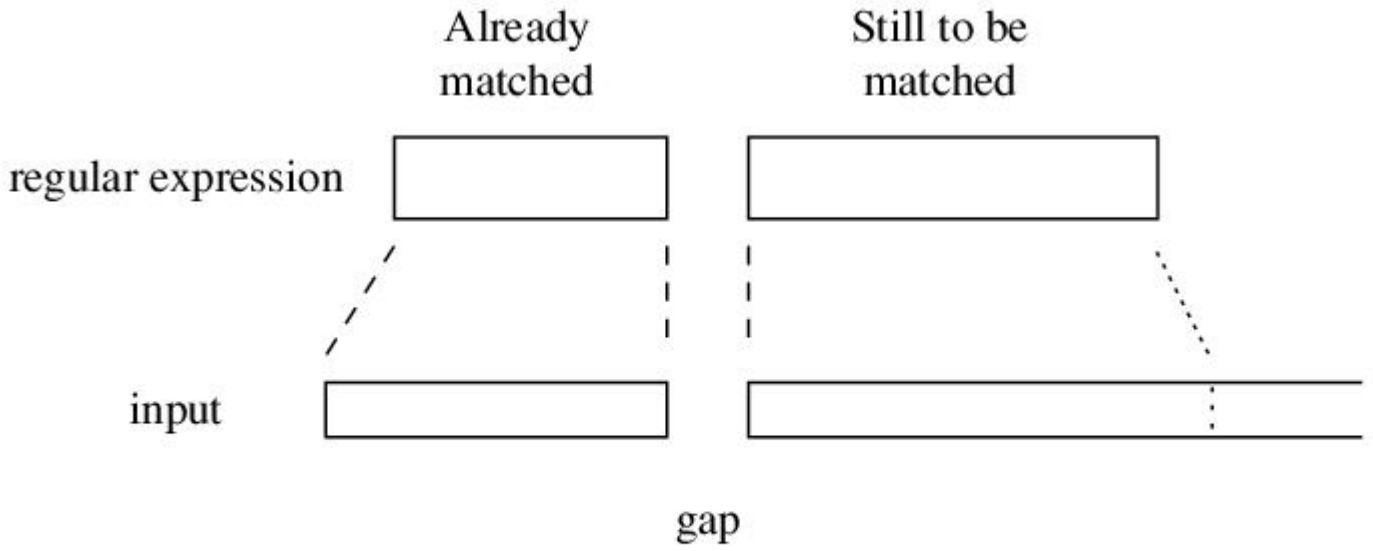

Figure 2.17 Components of a token description and components of the input.

Dotted Items (1)

reguläre Ausdrücke mit Positionsangabe

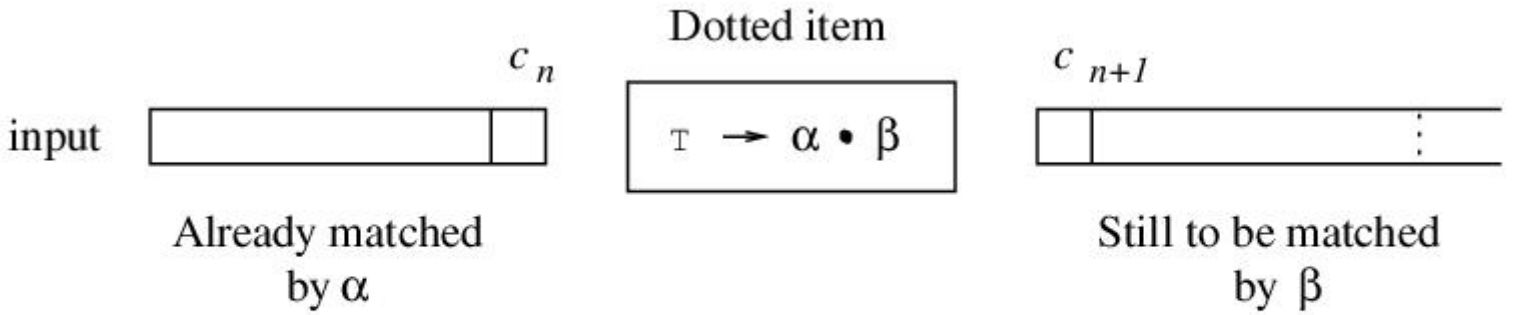

Figure 2.18 The relation between a dotted item and the input.

Dotted Items (2)

- Idee
  - Cursor • trennt erkannter von noch zu erkennender Eingabe
  - ”Hypothese(n)” über die Präsenz eines Tokens in der Eingabe
  - Cursor Move über ein Eingabesymbol kann berechnet werden
- Items
  - *Basic Items*
    - *Shift Item*: Cursor am Anfang eines Basismusters
    - *Reduce Item*: Cursor am Ende eines Basismusters
  - *Non-Basic Item:* Cursor vor Klammer oder Repetition
- Moves:
  - *Basic Items*
    - *Shift Item*: lese Zeichen c, *Regel:* $T \rightarrow \alpha \bullet c\beta \Rightarrow T \rightarrow \alpha c \bullet \beta$
    - *Reduce Item*: Match eines Tokens (mglw. nicht das längste Wort!)
  - *Non-Basic Item:* ε-Moves (Abb. 2.19)

$$
\begin{array}{lcl}
T \to \alpha \bullet (R)^* \beta & \Rightarrow & T \to \alpha (R)^* \bullet \beta \\
 & & T \to \alpha (\bullet R)^* \beta \\
T \to \alpha (R \bullet)^* \beta & \Rightarrow & T \to \alpha (R)^* \bullet \beta \\
 & & T \to \alpha (\bullet R)^* \beta \\
T \to \alpha \bullet (R)^+ \beta & \Rightarrow & T \to \alpha (\bullet R)^+ \beta \\
T \to \alpha (R \bullet)^+ \beta & \Rightarrow & T \to \alpha (R)^+ \bullet \beta \\
 & & T \to \alpha (\bullet R)^+ \beta \\
T \to \alpha \bullet (R)^? \beta & \Rightarrow & T \to \alpha (R)^? \bullet \beta \\
 & & T \to \alpha (\bullet R)^? \beta \\
T \to \alpha (R \bullet)^? \beta & \Rightarrow & T \to \alpha (R)^? \bullet \beta \\
T \to \alpha \bullet (R_1 \mid R_2 \mid \ldots) \beta & \Rightarrow & T \to \alpha (\bullet R_1 \mid R_2 \mid \ldots) \beta \\
 & & T \to \alpha (R_1 \mid \bullet R_2 \mid \ldots) \beta \\
 & & \ldots \\
T \to \alpha (R_1 \bullet \mid R_2 \mid \ldots) \beta & \Rightarrow & T \to \alpha (R_1 \mid R_2 \mid \ldots) \bullet \beta \\
T \to \alpha (R_1 \mid R_2 \bullet \mid \ldots) \beta & \Rightarrow & T \to \alpha (R_1 \mid R_2 \mid \ldots) \bullet \beta \\
\ldots & \ldots & \ldots
\end{array}
$$

Figure 2.19 ε move rules for the regular operators.

Nebenläufige Tokenerkennung

Beispieleingabe: **3.1**

Startmenge:

```
integral_number     →•([0-9])+
fixed_point_number →•([0-9])* ´.´ ([0-9])+
```

ε-Moves:

```
integral_number     →(•[0-9])+
fixed_point_number →(•[0-9])* ´.´ ([0-9])+
fixed_point_number →([0-9])* •´.´ ([0-9])+
```

Match **3**

```
integral_number     →([0-9]•)+
fixed_point_number →([0-9]•)* ´.´ ([0-9])+
```

ε-Moves:

```
integral_number     →(•[0-9])+
integral_number     →([0-9])+ •
fixed_point_number →(•[0-9])* ´.´ ([0-9])+
fixed_point_number →([0-9])* •´.´ ([0-9])+
```

vorige Folie	integral_number → (•[0-9])+
	integral_number → ([0-9])+ •
	fixed_point_number → (•[0-9])* ´.´ ([0-9])+
	fixed_point_number → ([0-9])* •´.´ ([0-9])+
Match .	fixed_point_number → ([0-9])* ´.´ •([0-9])+
ε-Moves	fixed_point_number → ([0-9])* ´.´(•[0-9])+
Match 1	fixed_point_number → ([0-9])* ´.´([0-9]•)+
ε-Moves	fixed_point_number → ([0-9])* ´.´(•[0-9])+
	fixed_point_number → ([0-9])* ´.´([0-9])+ •

erkannte Token:

- Länge 3, String: **3.1** Klasse: **fixed_point_number**
- Länge 1, String: **3** Klasse: **integral_number**

gib längstes Token zurück

vorige Folie

```
integral_number      →(•[0-9])+
integral_number      →([0-9])+ •
fixed_point_number   →(•[0-9])* ´.´ ([0-9])+
fixed_point_number   →([0-9])* •´.´ ([0-9])+
```

Match `.`

```
fixed_point_number   →([0-9])* ´.´ •([0-9])+
```

ε-Moves

```
fixed_point_number   →([0-9])* ´.´(•[0-9])+
```

Match `1`

```
fixed_point_number   →([0-9])* ´.´([0-9]•)+
```

ε-Moves

```
fixed_point_number   →([0-9])* ´.´(•[0-9])+
fixed_point_number   →([0-9])* ´.´([0-9])+ •
```

andere Situation: Buchstabe im Eingabestring:

- Menge der Items leer, aber kein Fehler!
- liefere längstes verbleibendes Token: `integral_number →([0-9])+ •`
- setze Lesezeiger zurück auf: • ´.´
- Beispiel in FORTRAN: `3.ge.1`, es folgt Operator `.ge.`

Scanning-Algorithmus

`Get next token` verwendet drei Funktionen:

- **`Initial item set`**
  - von der Tokenbeschreibung abgeleitete Shift Items
- **`Next item set(Item set, ch)`**: Items nach Move über **ch**
  - **`ε-closure(New Item Set)`**
    - bilde transitive Hülle der ε-Moves
    - entferne alle Non-Basic items
- **`Class of token recognized in (Item set)`**
  - suche nach Reduce Items, Rückgabe der entsprechenden Tokenklassen
  - falls mehrere Items gefunden, Auswahl nach folgender Ordnung

    1. *s* := längstes Segment im Eingabestring, Bsp.: 1, 1.2, 1.2e-7
    2. Token passend für *s*, welches zuerst in der Definition vorkommt

```
keyword    →while
identifier →[a-z]+
```

```
IMPORT Input char [1..];          // as from the previous module
SET Read index TO 1;              // the read index into Input char []

PROCEDURE Get next token:
    SET Start of token TO Read index;
    SET End of last token TO Uninitialized;
    SET Class of last token TO Uninitialized;

    SET Item set TO Initial item set ();
    WHILE Item set /= Empty:
        SET Ch TO Input char [Read index];
        SET Item set TO Next item set (Item set, Ch);
        SET Class TO Class of token recognized in (Item set);
        IF Class /= No class:
            SET Class of last token TO Class;
            SET End of last token TO Read index;
        SET Read index TO Read index + 1;

    SET Token .class TO Class of last token;
    SET Token .repr TO Input char [Start of token .. End of last token];
    SET Read index TO End of last token + 1;
```

Figure 2.21 Outline of a linear-time lexical analyzer.

```
FUNCTION Initial item set RETURNING an item set:
    SET New item set TO Empty;

    // Initial contents - obtain from the language specification:
    FOR EACH token description T→R IN the language specification:
        SET New item set TO New item set + item T→•R;

    RETURN ε closure (New item set);
```

Figure 2.22 The function `Initial item set` for a lexical analyzer.

```
FUNCTION Next item set (Item set, Ch) RETURNING an item set:
    SET New item set TO Empty;

    // Initial contents - obtain from character moves:
    FOR EACH item T→α•Bβ IN Item set:
        IF B is a basic pattern AND B matches Ch:
            SET New item set TO New item set + item T→αB•β;

    RETURN ε closure (New item set);
```

Figure 2.23 The function `Next item set()` for a lexical analyzer.

```
FUNCTION ε closure (Item set) RETURNING an item set:
    SET Closure set TO the Closure set produced by the
        closure algorithm 2.25, passing the Item set to it;

    // Filter out the interesting items:
    SET New item set TO Empty set;
    FOR EACH item I IN Closure set:
        IF I is a basic item:
            Insert I in New item set;

    RETURN New item set;
```

Figure 2.24 The function ε `closure()` for a lexical analyzer.

Data definitions:

Let `Closure set` be a set of dotted items.

Initializations:

Put each item in `Item set` in `Closure set`.

Inference rules:

If an item in `Closure set` matches the left-hand side of one of the ε moves in Figure 2.19, the corresponding right-hand side must be present in `Closure set`.

Figure 2.25 Closure algorithm for dotted items.

Vorberechnen der Itemmengen

... durch einen endlichen Automaten ...

- Gedankengang
  - Ineffizienz: Itemmengen müssen immer wieder neu berechnet werden
  - Besser: konstruiere deterministischen endlichen Automaten für die Übergänge zwischen Itemmengen
- Komplexität des Automaten
  - Zeit: linear in der Länge der Eingabe (unabhängig von der Zahl der Items)
  - Speicher: exponenziell in der Anzahl der Items (Worst Case, in der Praxis meistens unproblematisch)

Automatenerstellung

- Hüllenalgorithmus: Teilmengenkonstruktion (Abb. 2.26)
- Beispiel: Fest”komma”zahl
  - Grammatik: **I** → **D+**
    F → **D* ´.´ D+**
  - Automat
    - Graph: Abb. 2.28, 2.30
    - Transitionstabelle: Abb. 2.27
    - Progamm: Abb. 2.29, vgl. Abb. 2.21
- Anwendungen
  - Scanner in Sprachprozessoren
  - Virenscanner
- Linearitätsprobleme
  - Grammatik: **single_a** → **´a´**
    as_plus_b → **´a´* ´b´**
  - Eingabe: **aaaaaa...aaa** (sehr viele **a**s, ohne folgendes **b**)

Data definitions:

1a. A 'state' is a set of items.

1b. Let `States` be a set of states.

2a. A 'state transition' is a triple (start state, character, end state).

2b. Let `Transitions` be a set of state transitions.

Initializations:

1. Set `States` to contain a single state, `Initial item set()`.

2. Set `Transitions` to the empty set.

Inference rules:

If `States` contains a state `S`, `States` must contain the state `E` and `Transitions` must contain the state transition (`S`, `Ch`, `E`) for each character `Ch` in the input character set, where `E = Next item set (S, Ch)`.

Figure 2.26 The subset algorithm for lexical analyzers.

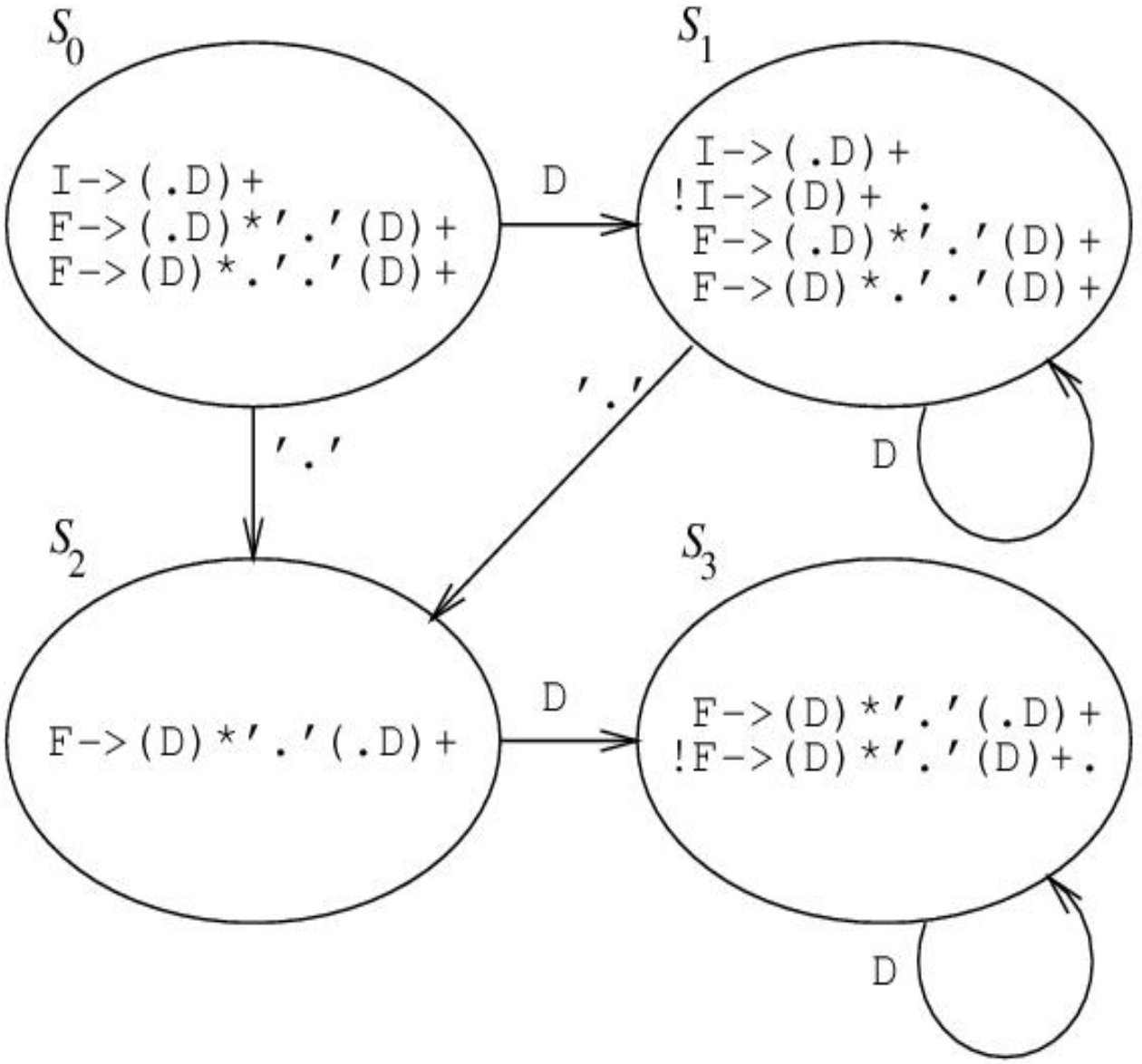

Figure 2.28 Transition diagram of the states and transitions for Figure 2.20.

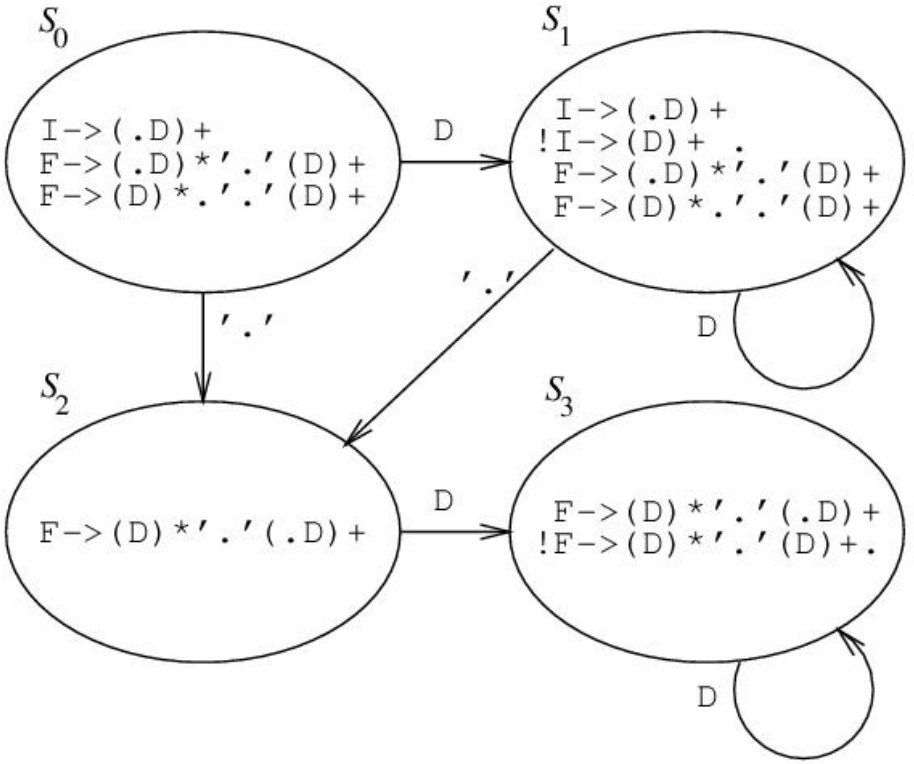

Figure 2.28 Transition diagram of the states and transitions for Figure 2.20.

State	Next state [] Ch			Class of token recognized in[]
	digit	point	other	
S_0	S_1	S_2	-	-
S_1	S_1	S_2	-	integral_number
S_2	S_3	-	-	-
S_3	S_3	-	-	fixed_point_number

Figure 2.27 Transition table and recognition table for the regular expressions from Figure 2.20.

Tabellenzugriffe beim Scannen statt Funktionsaufrufe

```
PROCEDURE Get next token:
    SET Start of token TO Read index;
    SET End of last token TO Uninitialized;
    SET Class of last token TO Uninitialized;

    SET Item set TO Initial item set;
    WHILE Item set /= Empty:
        SET Ch TO Input char [Read index];
        SET Item set TO Next item set [Item set, Ch];
        SET Class TO Class of token recognized in [Item set];
        IF Class /= No class:
            SET Class of last token TO Class;
            SET End of last token TO Read index;
        SET Read index TO Read index + 1;

    SET Token .class TO Class of last token;
    SET Token .repr TO Input char [Start of token .. End of last token];
    SET Read index TO End of last token + 1;
```

Tabellenzugriffe

Figure 2.29 Outline of an efficient linear-time routine `Get next token()`.

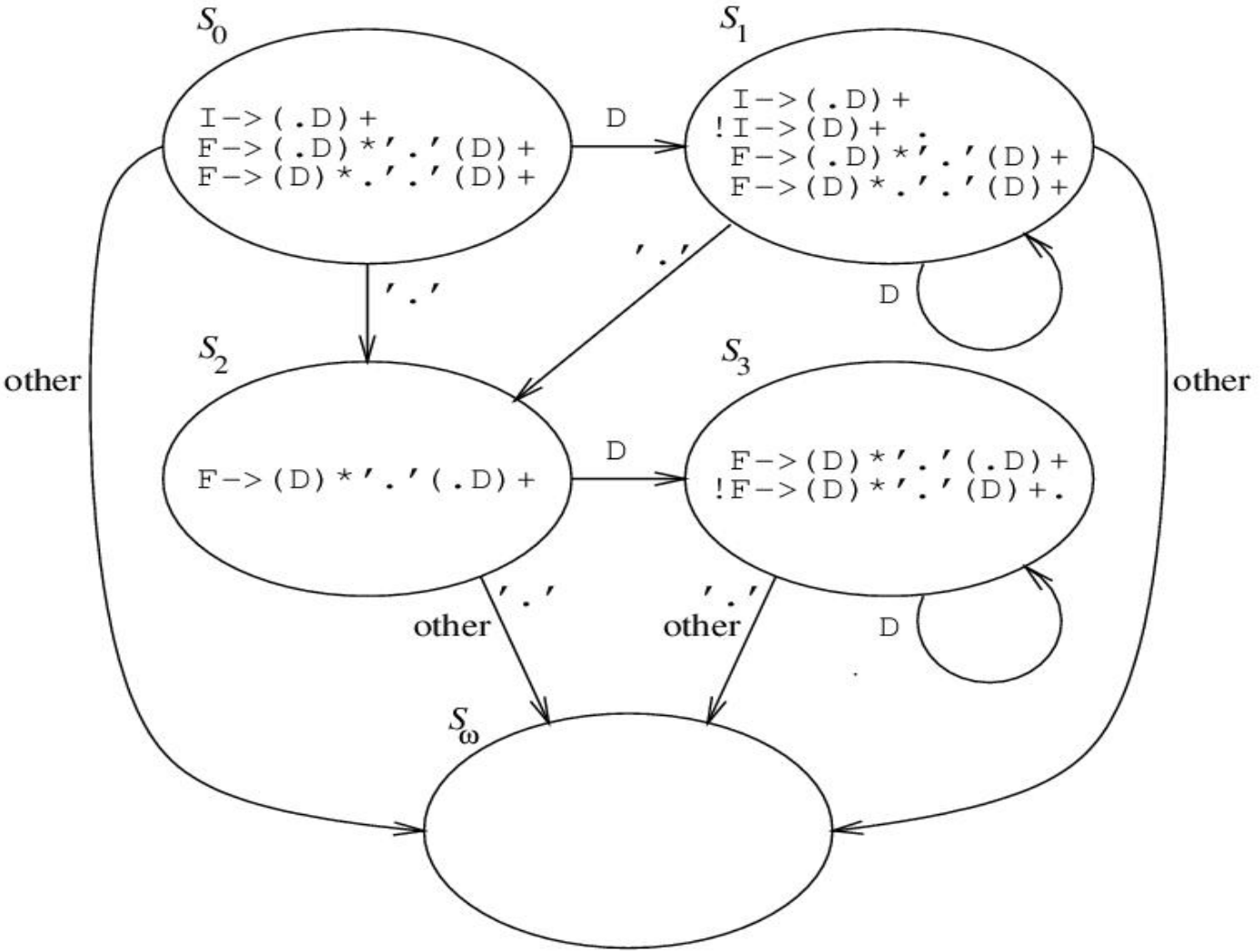

Figure 2.30 Transition diagram of *all* states and transitions for Figure 2.20.