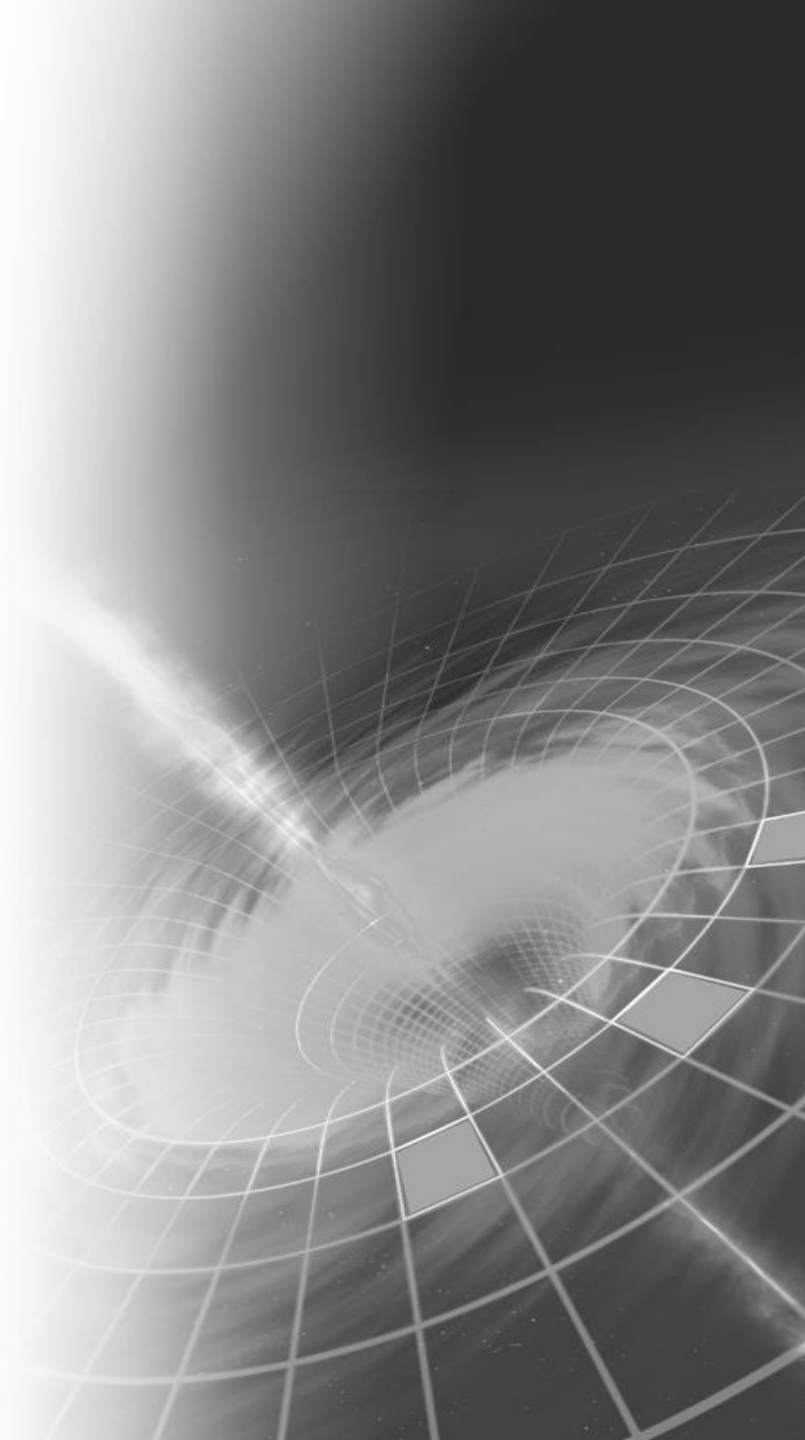
Bachelor-Programm

Compilerbau

im SoSe 2014

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage

fischer@informatik.hu-berlin.de



Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbesch
- Kapitel 3
  Lexikalische Analyse: der Sca
- Kapitel 4
  Syntaktische Analyse: der Pa
- Kapitel 5
  Parser-Generatoren: Yacc, Bis
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerierung

6.1
Überblick: Grammatik-basierte Übersetzung

6.2
Attribut-Grammatiken

6.3
S-attributierte Syntaxdefinitionen

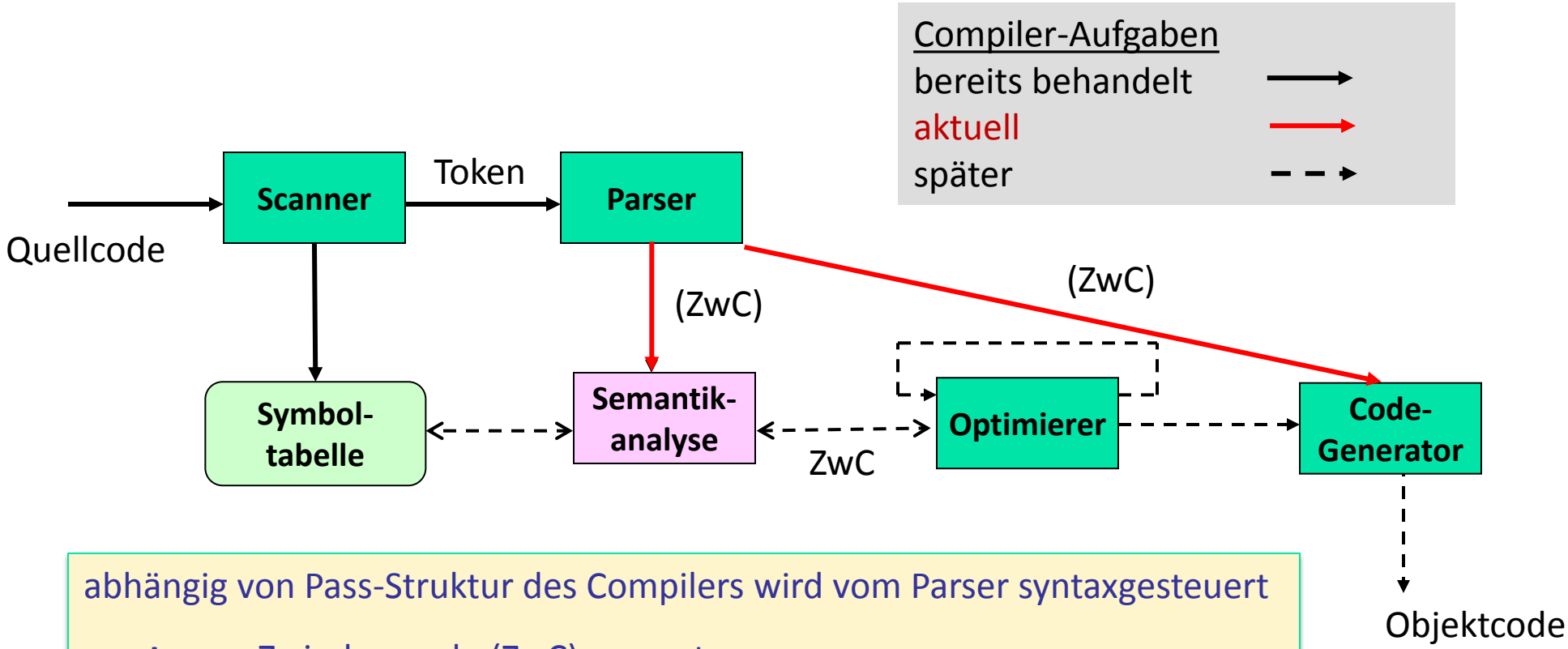
6.4
L-attributierte Syntaxdefinitionen

6.5
Syntaxgesteuerte Übersetzungen im Überblick

6.6
Entwurf syntaxgesteuerter Übersetzungen

6.7
Symboltabelle

Wo stehen wir?



abhängig von Pass-Struktur des Compilers wird vom Parser syntaxgesteuert

- **externer** Zwischencode (ZwC) erzeugt

oder

- **on-the-fly** die Compilationsaufgabe erledigt

Statische Semantik

einige Eigenschaften eines Quell-Programms

- können bereits zur Compile-Zeit berechnet werden
- gelten für jede Ausführung des Programms unabhängig von Programmumgebung (E/A)

Beispiel

```
int a;
float c= getRuntimeValue();
a= c + 99;  /* je nach Sprache ist Zuweisung zulässig oder nicht */
```

Unser Ziel: Bereitstellung allgemeiner Übersetzungstechniken
für kontextfreie Sprachen
(möglichst unabhängig vom Parsertyp)

zur Realisierung von

- statischer Semantikanalyse und
- Zwischencode-Erzeugung

Zur Erinnerung: 1. Beispiel

unser einfacher
Einpass-Compiler (Front- und Back-End)

als **Transformation**:

Folge von Semikolon-separarierten ***Infix***-*Ausdrücken*

➔

Folge von Semikolon-separarierten ***Postfix***-*Ausdrücke*

bei **Einsatz**

(1) eines handgeschriebenen Scanners

(2) der prädiktiven Syntaxanalyse-Technik:

rekursiv absteigender Parser

(3) der Übersetzungstechnik:

Übersetzungsschema

als kfG mit
eingestreuten semantischen Aktionen {...}
in den rechten Seiten der Produktionen

eindeutig, linksrekursive Regeln

1.	*goal* → *list*
2.	*list* → *expr*; *list*
3.	| ε
4.	*expr* → *expr* **+** *term* {print('+')}
5.	| *expr* **–** *term* {print('-')}
6.	| *term*
7.	*term* → *term* ***** *factor* {print('*')}
8.	| *term* **/** *factor* {print('/')}
9.	| *term* **DIV** *factor* {print('DIV')}
10.	| *term* **MOD** *factor* {print('MOD')}
11.	| *factor*
12.	*factor* → **num** {print(**num**.value)}
	| **id** {print(**id**.lexeme)}

Erste Erkenntnis als Hoffnung für Verallgemeinerung

Analyse-/Übersetzungsprozess

wird durch die syntaktische Struktur des Programms getrieben,
so wie sie vom Parser konstruiert wird

Semantische Aktionen

- werden mit einzelnen Produktionsregeln der kontextfreien Grammatik oder mit einem (Teil-)Baum des Syntaxbaumes assoziiert
- interpretieren die Bedeutung des Programms auf der syntaktischen Struktur
- haben zwei Ziele:
  1. Vervollständigung der Analyse durch Berücksichtigung zusätzlicher kontextsensitiver Informationen
  2. Vorbereitung der Synthese durch die Erzeugung der Zwischenrepräsentation bzw. des Zielcodes

Charakteristik Semantischer Aktionen

- **Parser-Aufgaben**
  - accept/reject
  - initiiert u.a. auch Analyse/Übersetzung durch Ausführung **semantischer Aktionen**

Attribut-Grammatiken
(Donald Knuth, 1968)
- Knoten im Syntaxbaum haben Attribute
- Attribute dienen als Zwischenspeicher für den Datenfluss zwischen den Knoten

- **semantische Aktionen**
  sind Routinen,
  die durch den Parser bei der Erkennung eines syntaktischen Symbols ausgeführt werden
  - jedes Grammatiksymbol hat einen assoziierten semantischen Wert
  - jetzt Verallgemeinerung: **Attribute** von Knoten des Syntaxbaums
    mit unterschiedlicher Handhabung für **LL**- und **LR**-Parser

- **Ausführung (Auswertung) semantischer Aktionen** kann
  - temporär im Hauptspeicher den Syntaxbaum erzeugen
  - Informationen in einer Symboltabelle speichern
  - (Zwischen-)Code generieren
  - Fehlermeldungen erzeugen

FAZIT: Realisierung der statischen Semantik-Analyse/Übersetzung des Token-Eingabestroms erfolgt durch Attributberechnungen

LL-Parser und Aktionen (1)

Attributberechnung als Teil der statischen Analyse/Übersetzung erfolgt mittels „**Attribut-Auswerter**“

- on-the-fly oder
- in einem separaten Pass

bei Durchmusterung des Syntaxbaums (als ZwC)

syntaktische Struktur treibt Analyse/Übersetzung an

Wann und in welcher Reihenfolge werden Aktionen von **LL-Parsern** ausgeführt?

Regeln werden **vor** der Analyse der rechten Seite (also vor Auswertung der semantischen Aktionen) einer Regel expandiert

daher

- werden Aktionen wie alle anderen Grammatiksymbole zunächst auf dem Keller gespeichert (push)
- erscheinen diese als oberstes Kellerzeichen (tos), werden sie ausgeführt.

LL-Parser und Aktionen (2)

Erweiterung
eines tabellen-gesteuerten LL-Parsers

```
push EOF;
push Start_Symbol;
token := next_token();
repeat
    pop X;
    if X is (a_terminal or EOF) then
        if X is token then
            token := next_token();
        else error()
    else if X is an_action then
           perform X
         else  /* X is a non-terminal symbol */
         if M[X, token] == X → Y1Y2…Yn then
               push(an, Yn, an-1, Yn-1…, a1, Y1)
         else error();
until X == EOF;
```

Syntaxanalyse-Tabelle

Aktionen a_i

LR-Parser und Aktionen (3)

Wie werden Aktionen vom **LR-Parser** ausgeführt?

Analyse der gesamten rechten Seite (RS) einer Regel ist notwendig,
bevor die Regel
zur Reduktion angewandt wird

deshalb kann eine Aktion

- **erst nach** dem Scannen der **kompletten** RS ausgeführt werden

- und nur als **letztes** »Symbol« einer Produktion erscheinen
(→ Restriktion gegenüber LL-Verfahren)

impliziert u.U. neue (zusätzliche) Nicht-Terminalsymbole (als Marker) und Zusatzregeln, um diese Restriktion zu umgehen

- aus: $A \rightarrow \mathbf{w}\ \mathit{action}\ \beta$
- wird: $A \rightarrow M\beta$ und $M \rightarrow \mathbf{w}\ \mathit{action}$

zusätzliche Symbole und Regeln werden von Yacc/Bison automatisch erzeugt

aber u.U. neue Konflikte, die schwerer zu finden sind

Aktionsgesteuerter Keller von LR-Parsern

Ansatz zur Semantik-Aktions-Ausführung

- Semantikaktionen arbeiten auf (separatem) Wertekeller (Attributwerte)
- Aktionen erhalten Argumente explizit vom Keller
- Aktionen platzieren Ergebnis wiederum auf dem Keller

Vorteil

- Aktionen haben Zugriff auf den gesamten Keller
  (ohne pop-Operationen ausführen zu müssen)

Nachteil

- Implementierung der Aktionen ist direkt an Keller orientiert
- Aktionen enthalten demnach expliziten Code zur Handhabung des Kellers

Bison erleichtert Zugang mittels $-Variablen

Aufbau einer Bison-(Eingabe-)Datei: parser.y

Struktur: drei Bereiche

```
/* Bison-Deklarationen und Prolog */

%%

/* Grammatikregeln */

%%

/* benutzerdefinierte Routinen */
```

- Definition von Token
- Einbinden von C- bzw. C++ -Code
- Festlegung von Prioritäts- u. Assoziationsbedingungen
- Definition von beliebigen Rückgabetypen

- implizite Festlegung von Nichtterminalsymbolen (Kleinschreibung)
- jedes Symbol einer RS kann mit Aktionen (C/C++ Anweisungen) verbunden werden

- Bereitstellung von int yylex()
- Überschreiben von yyerrror() und int main()
- Funktionen, die in Aktionen benutzt werden

Der Regelteil in Bison

1	program	→	program expr **\n**
3	expr	→	**INTEGER**
4	|	→	expr **+** expr
5	|	→	expr **-** expr

```
regel ::= NAME ':' alternativen ';' ;

alternativen ::=
      [alternative] ('|' alternative)* ;

alternative ::=
      element+                          ;

element ::=
      NAME
    | CHARACTER
    | '{' semAktion '}'
                                        ;
```

Bison-Eingabe: parser.y

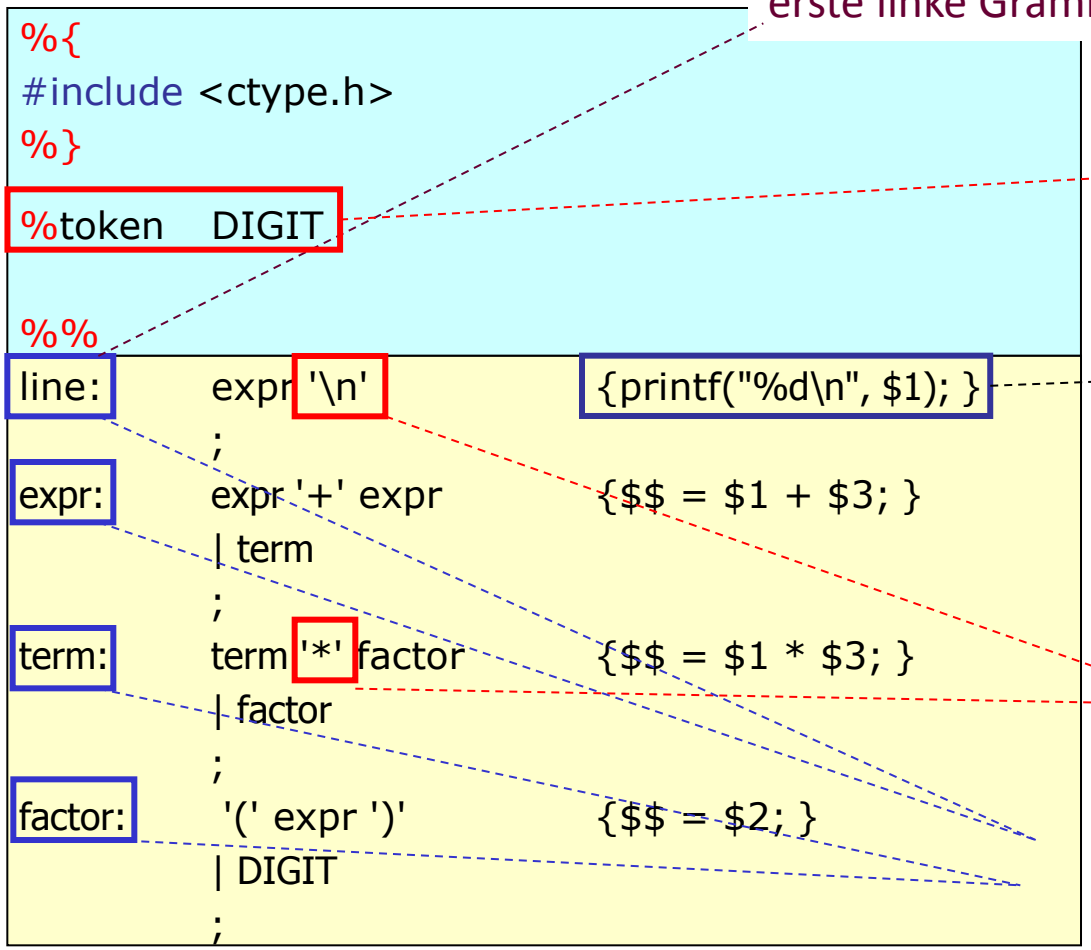
```
...
program:
                  program expr '\n'
                  ;

expr :
                  INTEGER
                  | expr '+' expr
                  | expr '-' expr
                  ;
...
```

noch ohne semantische Aktionen

Bison erlaubt nur BNF, nicht EBNF

Notationskonvention *Terminal- und Nichtterminalsymbol* in Bison



erste linke Grammatik-Seite: **Startsymbol**

Tokendefinition

semantische Aktion

gequotete einzelne Zeichen:
Terminalsymbole (Token)

Namen, die nicht als Token vereinbart worden sind:
Nichtterminalsymbole

Semantische Werte *(Attribute von Grammatiksymbolen)* in Bison

- Jedes Token verfügt über semantischen Wert (Attribut)
  - INTEGER → {4, 1, 3042}
  - IDENTIFIER → Zeiger in Symboltabelle
  - , [einzelne Zeichen] → ASCII-Wert, aber wird nicht benötigt
- Jedes Nichtterminal und jede Gruppierung von Terminal-und Nichtterminalsymbolen besitzt einen Wert (Attribut),

  z.B. Ausdruck (Baumstruktur als Bedeutung des Ausdrucks)

universeller Typ YYSTYPE

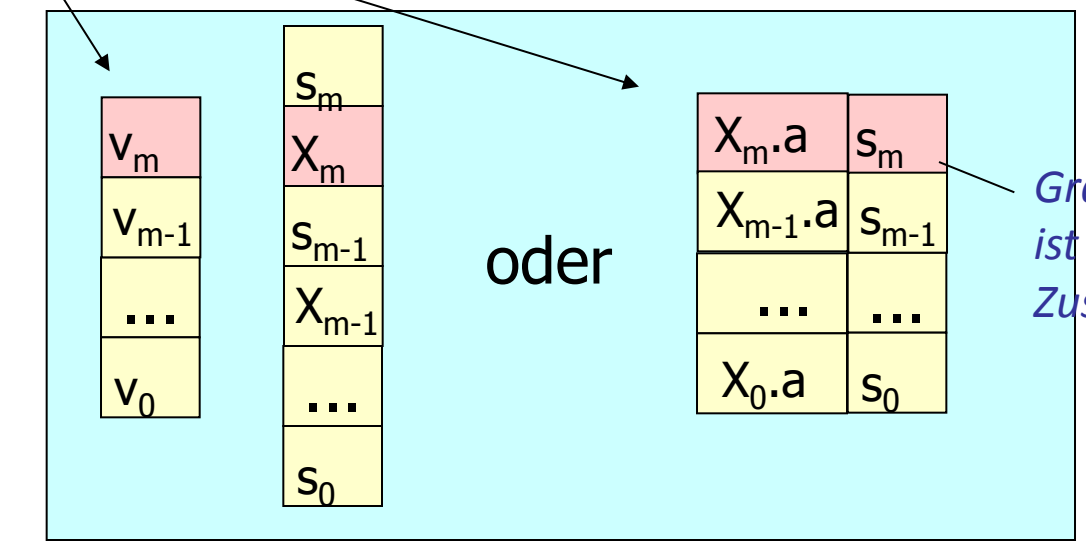
Synchrone Kellerverwaltung

Bison verwaltet als LR-Parser intern **zwei** synchrone Kellerspeicher:

(1) Ableitungskeller Zustand des Parsers

(2) Wertekeller als Array von YYSTYPE-Elementen
yylval-Werte

sind Werte der Attribute der Grammatiksymbole



Grammatiksymbol ist implizit im Zustand enthalten

hat ein Symbol **kein** Attribut, ist der entspr. Werteeintrag undefiniert

Semantische Aktionen in Bison

Semantische Aktionen können über Attributwerte von Grammatiksymbolen operieren

```
%%
line:     expr '\n'   {printf("%d\n", $1); }
          ;
expr:     expr '+' expr          {$$ = $1 + $3; }
          | term
          ;
term:     term '*' factor        {$$ = $1 * $3; }
          | factor
          ;
factor:    '(' expr ')'          {$$ = $2; }
          | DIGIT
          ;
```

$$ - Attributwert vom Nichtterminalsymbol der **LS** einer Regel

$i – Attributwert des i-ten Grammatiksymbols der **RS** einer Regel (Terminal- oder Nichtterminalsymbol)

In der Situation,
wo eine **Regel** in Bison zur Anwendung kommt,
kann als semantische Aktion
- der Wert eines Symbols **gelesen** werden, das auf der RS einer Regel vorkommt
- ein neuer Wert dem Symbol auf der LS **zugewiesen** werden

Beispiel: Ausdrucksberechnung

```
/* Reverse polish notation calculator */
%{
#define YYSTYPE double
#include <math.h>
#include <ctype.h>
%}
%token    NUM
%%
prog:     /* empty */
          | prog line
          ;
line:     '\n'
          | expr '\n'              {printf("\t%.10g\n", $1); }
          ;
expr:     NUM                      {$$ = $1; }
          | expr expr '+'          {$$ = $1 + $2; }
          | expr expr '-'          {$$ = $1 - $2; }
          | expr expr '*'          {$$ = $1 * $2; }
          | expr expr '/'          {$$ = $1 / $2; }
          | expr expr '^'          {$$ = pow($1, $2); }
          | expr 'n'               {$$ = -$1; }
;
%%
```

Linksrekursion, bei LR möglich

```
/* returns a double floating point number on
the stack and the token NUM, or the the ASCII
character read if not a number.
skip all blanks and tabs, returns 0 for EOF */

int yylex () {
    int c;
    /* skip white spaces */
    while ((c = getchar() == '  ' || c== '\t' )
            ;
    /* process numbers */
    if (c == '.' || isdigit(c)) {
            unget (c, stdin);
            scanf("%lf", &yylval);
            return NUM;
    }
    /* return end-of-file */
    if (c == EOF)
            return 0;
    /* return single chars */
    return c;
}

int main (void) {
    yyparse();
}
```

```
#include <stdio.h>

/* called by yyparse on error */
void yyerror (char *s) {
    fprintf (stderr, "%s\n", s);
}
```

Beispiel: Programmierbarer Taschenrechner

```
...
program:
                function                                { exit(0); }
                ;

function:
                function stmt                           { ex($2); freeNode($2); }
                | /* NULL */
                ;

stmt:
                ';'                                     { $$= opr(';', 2, NULL, NULL); }
                | expr                                  { $$ = $1; }
                | PRINT expr ';'                        { $$= opr(PRINT, 1, $2 ); }
                | VARIABLE '=' expr ';'                 { $$= opr('=', 2, id($1), $3); }
                | WHILE '(' expr ')' stmt               { $$= opr(WHILE, 2, $3, $5); }
                | IF'(' expr ')' stmt %prec IFX         { $$= opr(IF, 2, $3, $5); }
                | IF'(' expr ')' stmt ELSE stmt         { $$= opr(IF, 3, $3, $5, $7); }
                | '{' stmt_list '}'                     { $$= $2; }
                ;
...
```

konkrete
Yacc/Bison-Technologie

modellmäßige
Verallgemeinerung
unabhängig
der Technologie ?

Parser erledigte zusätzlich
- sukzessiven Aufbau des Syntaxbaums (Teilbäume für function) bei Belegung der Symboltabelle
- Durchmusterung des Baums (Tiefe-zu-erst): Interpretation
- Speicherfreigabe des Teilbaums

Verbindung semantischer Aktionen mit Grammatikregeln

zwei mögliche Notationsformen grammatikbasierter Übersetzungen

beide basieren auf dem Modell

attributierter Grammatiken (Knuth)

1. Syntaxgesteuerte Definitionen abstraktere Spezifikation von Übersetzungen (Verallgemeinerung attributierter Grammatiken)
   - verbergen Implementationsdetails
   - befreien den Nutzer von expliziter Angabe von Reihenfolgen der Übersetzung (Abhängigkeiten und evtl. Konflikte müssen automatisch erkannt werden)

2. Übersetzungsschemata (wie in Bison)
   - Programmfragmente werden in der RS einer Regel {…} platziert, legen Reihenfolgen der Regelauswertung fest,
   - erlauben deshalb auch Definition von Implementationsdetails

Syntaxgesteuerte Definition – Übersetzungschema

- syntaxgesteuerte Definitionen (technologie-unabhängig)

keine Seiteneffekte
(wird später aber aufgeweicht !)

Produktion	semantische Regel
...	...
$A \rightarrow X\ Y\ Z$	A.a:= f (X.x, Y.y, Z.z)
...	...

- Übersetzungsschemata (technologie-abhängig, z.B. Bison)

mit Seiteneffekten

Produktionen mit semantischen Regeln
$E \rightarrow T\ R$
$R \rightarrow$ **op** T { print(**op**.lexem) } R1 | ε
$T \rightarrow$ **num** {print(**num**.val)}

Beispiel einer grammatik-basierten Übersetzung

… eines Infix-Postfix-Operators als …

syntaxgesteuerte Definition (technologie-unabhängig)

Produktion	Semantische Regel
E → E1 + T	E.code= E1.code || T.code || '+'

strukturell überzeugend,
aber String-Verarbeitung ist für die reale Übersetzung nicht effizient genug

unabhängig ob LL- oder LR-Technik zum Einsatz kommt

Übersetzungsschema (technologie-abhängig)

integrierte Notation

E → E1 + T {print '+'}

i.allg. können Aktionen nach jedem Symbol der RS vorkommen
(ACHTUNG: LR-Parser müssen hier aber „tricksen")

Syntaxgesteuerte Definition, Attributgrammatik

Eine **syntaxgesteuerte Definition** ist eine kfG mit

- Attributen (verbunden mit den Grammatiksymbolen)
  Beispiele:
  Zahlen, Typen, Tabellenverweise, Strings (Codesequenzen einer Zwischensprache)

- Regeln (Verbunden mit Produktionen)
  Operationen über die Attribute der Grammatiksymbole eines generierten Syntaxbaumes
  - mit und ohne Seiteneffekte

Bezugnahme: X.a (Zugriff auf Attribut a von X)

- Spezialfall: syntaxgerichtete Definition ohne Seiteneffekte = **Attributgrammatik**
  als Spezifikation von Zuweisungen zur Definition von (eindeutigen) Werten,
  wobei Attribute des Eltern- und der Kinderknoten benutzt werden können

Beispiel: Attribut-Grammatiken

Beispiel:

- Aufnahme von Attributen *type* und *class* in Knoten, die **Ausdrücke** darstellen

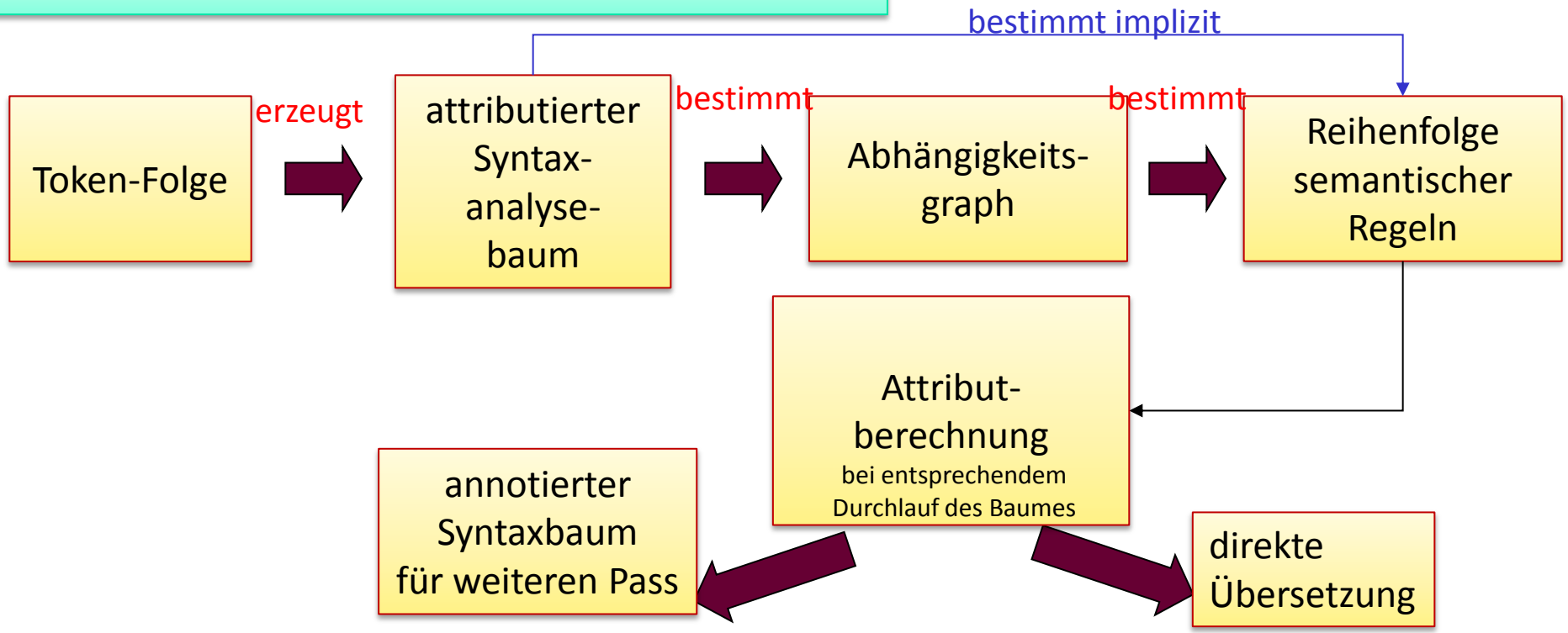
type ist der abgeleitete oder definierte Typ
- eines Wertes
- einer Variablen
- eines Ausdrucks

class ist die Knotenklasse z.B.
- Variable
- Konstante
- Ausdruck

- Einführung und Anwendung von Überprüfungsregeln (Gleichungen) am Beispiel des Zuweisungsknotens ":="
  - ist LS.class eine Variable ?
  - ist LS.class eine Konstante ?
  - sind LS.type und RS.type konsistent ?

Allgemeines Prinzip einer grammatik-basierten Übersetzung

… realisierbar durch beide Notationsformen



wissen aber bereits: u.U. auch in einem Schritt möglich,

d.h. ohne Erstellung von Syntaxbaum und Abhängigkeitsgraph
(Effizienzsteigerung der Übersetzung)

➔ Compilerpraxis kennt eine Vielzahl von Spezialfällen

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbesch
- Kapitel 3
  Lexikalische Analyse: der Sca
- Kapitel 4
  Syntaktische Analyse: der Pa
- Kapitel 5
  Parser-Generatoren: Yacc, Bis
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerierung

6.1
Überblick: Grammatik-basierte Übersetzung

6.2
Attribut-Grammatiken

6.3
S-attributierte Syntaxdefinitionen

6.4
L-attributierte Syntaxdefinitionen

6.5
Syntaxgesteuerte Übersetzungen im Überblick

6.6
Entwurf syntaxgesteuerter Übersetzungen

6.7
Symboltabelle

Attributierte Grammatiken (Zusammenfassung)

Idee (D. Knuth)

der Implementationsunabhängigkeit kontextsensitiver Aktionen

wird durch das Konzept **Attribut-Grammatiken** formalisiert

Vorgehensweise

- Ausgangsbasis: Grammatik-basierte Spezifikation der Knotenattribute
  - jedes Grammatiksymbol (Terminal oder Nichtterminal) besitzt eine ihm zugeordnete Attributmenge
  - Attribut kann sein:
    - String, Zahl,
    - Typ, Symboltabelleneintrag,
    - **aber auch**: Code-Sequenz, …
- Wertezuweisungen für Knotenattribute werden mit den Produktionsregeln festgelegt/kombiniert
- jedes Attribut erhält einen eindeutig festgelegten Wert

Synthetisierte und ererbte Attribute

… nach Unterscheidung der Art der Attributberechnung

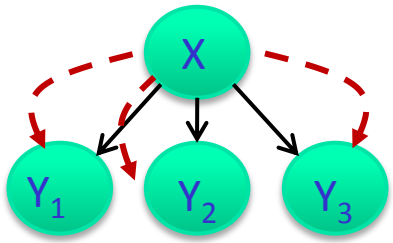
$X \rightarrow Y_1 Y_2 Y_3$

X. a

Y_1.a
Y_2.a
Y_3.a

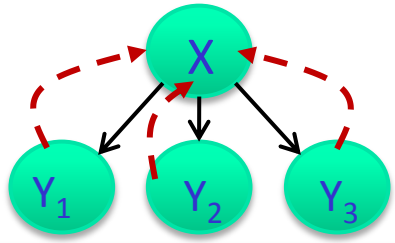
ererbte Attribute

$Y_1.a = f(X.a)$
$Y_2.a = f(X.a)$
$Y_3.a = f(X.a)$



Werte ererbter Attribute
werden von oberen Knoten
an untere zugewiesen (top-down)

synthetisierte Attribute



$X.a = f(Y_1.a, Y_2.a, Y_3.a)$

Werte synthetisierter Attribute
werden von unten nach oben „gereicht“
(bottom-up)

- **Token** (Terminalsymbole)
  - haben nur synthetisierte Attribute
  - ihre Werte werden i.allg. vom Scanner geliefert

- **Startsymbol**
  - hat keine ererbten Attribute

Berechnung von Attributwerten: Prinzip

- Wann wird berechnet?
  - während des Parsens der Eingabe oder
  - nach kompletter Syntaxbaumkonstruktion?

- Reihenfolge der Berechnung von Attributwerten ist zu beachten
  - falls Reihenfolge unbekannt, muss diese vorab bestimmt werden
  - entsprechende Topologie ergibt sich aus Attributabhängigkeiten
    (Spezialfall: synthetisierte Attribute, geerbte Attribute)

    ➔ Abhängigkeitsgraph bestimmt die Reihenfolge der Berechnung

- Berechnungsvorschrift für Attribute an einem Knoten (Semantikregel)
  - kann Seiteneffekte haben:
    Ausgabe eines Wertes, Update globaler Variablen
  - jeder Grammatikproduktion wird eine Menge von Semantikregeln zugeordnet
  - Funktionen meist nur als Ausdrücke

Problem mit beliebigem Mix geerbter und synthetisierter Attribute

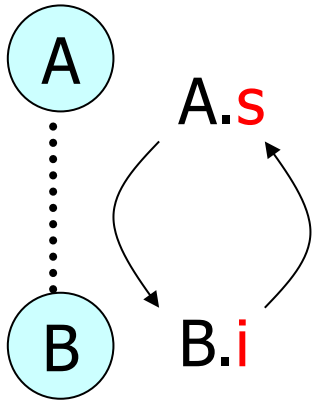
- keine Garantie für Eindeutigkeit der Reihenfolge der Attributberechnung

Beispiel:

A → B

A.s:= B.i

B.i:= A.s + 1



Regeln können i.allg. zirkulär sein

Ziel: Formulierung von Einschränkungen
bzgl. der Abhängigkeiten von Attributberechnungen,
die Zirkelfreiheit garantieren

Attributauswerter als Implementationen
- L-attributierter und
- S-attributierte Syntaxgesteuerter Definitionen (Spezialfall L-attributierte Definitionen)

garantieren Zirkelfreiheit und lassen sich zudem in einem Durchlauf realisieren

L-attributierte syntaxgesteuerte Definitionen

- … gestatten die Auswertung der semantischen Regeln in einem einzigen Durchlauf (wichtig für Laufzeiteffizienz, **L**: left-to-right)
- damit erlauben sie einem Top-Down-Parser die Realisierung der kompletten Übersetzung
- ein Syntaxbaum muss dabei aber nicht zwingend explizit konstruiert werden
- ermöglichen die Konstruktion von Ein-Pass-Compilern

Frage: lassen sich syntaxgesteuerte Übersetzungen sowohl für Top-Down- als auch Bottom-Up-Parser konstruieren ?

Antwort: **Ja**,

aber nur eingeschränkt in Abhängigkeit des Typs der zugeordneten Attributgrammatik/syntaxgesteuerten Definition

Berechnung von Attributwerten: Scheinattribute

- gelegentliches Ziel einer semantischen Aktion:
  Erzielung von Seiteneffekten (z.B. Ausgabe, Symboltabellenmodifikation)

  - Notation der Aktion als Prozeduraufruf:
    implizit aufgefasst als Semantikregel über ein imaginäres Attribut:
    liefert Wert eines synthetisierten Scheinattributes eines Nichtterminals der linken Seite

  Beispiel:

  - Regel: $L \rightarrow E$ **/n**
  - assoziierte semantische Aktion: print(E.val)

  L.Scheinattribut:= print(E.val)

Begriffe

- bewerteter (annotated) Syntaxbaum:
  ein Syntaxbaum, der die Werte seiner Attribute an jedem Knoten zeigt
- Bewertung:
  Prozess der Berechnung der Attributwerte (Ausführung des Attributauswerters)

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbesch
- Kapitel 3
  Lexikalische Analyse: der Sca
- Kapitel 4
  Syntaktische Analyse: der Pa
- Kapitel 5
  Parser-Generatoren: Yacc, Bis
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerierung

6.1
Überblick: Grammatik-basierte Übersetzung

6.2
Attribut-Grammatiken

6.3
S-attributierte Syntaxdefinitionen

6.4
L-attributierte Syntaxdefinitionen

6.5
Syntaxgesteuerte Übersetzungen im Überblick

6.6
Entwurf syntaxgesteuerter Übersetzungen

6.7
Symboltabelle

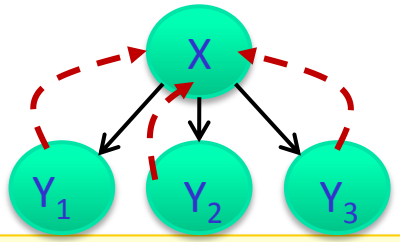
S-Attribut-Grammatik

Sonderfall: **S-Attribut-Grammatik**
verwendet nur synthetisierte Attribute

$X \rightarrow Y_1 Y_2 Y_3$

synthetisierte Attribute

$X.a = f(Y_1.b, Y_2.c, Y_3.d)$



aus Werten der Attribute der Nachfolgerknoten

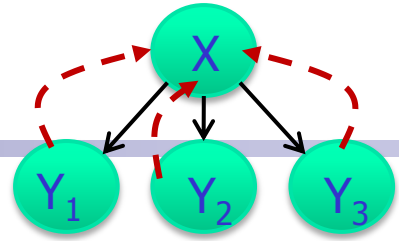
gewisse Einschränkung
(selten praktisch nutzbar)

effiziente Attribut-Auswerter

Ausdrucksberechnung basiert auf einer S-Attribut-Grammatik

synthetisierte Attribute

Beispiel für S-Attribut-Grammatik



$X.a = f(Y_1.b, Y_2.c, Y_3.d)$

Taschenrechner: Eingabezeile (enthält Ausdruck) wird mit **/n** abgeschlossen, berechneter Wert wird ausgegeben

Produktion	Semantische Regel	Bison
$L \rightarrow E$ **/n**	L.val:= print (E.val)	print(E.val)
$E \rightarrow E_1$ **+** T	E.val:= E_1.val + T.val	$$= $1 + $3
$E \rightarrow T$	E.val:= T.val	$$= $1
$T \rightarrow T_1$ * F	T.val:= T_1.val * F.val	$$= $1 * $3
$T \rightarrow F$	T.val:= F.val	$$= $1
F → **(** E **)**	F.val:= E.val	$$= $2
F → **digit**	F.val:= **digit**.lexval	$$= …

Symboltabellenzugriff

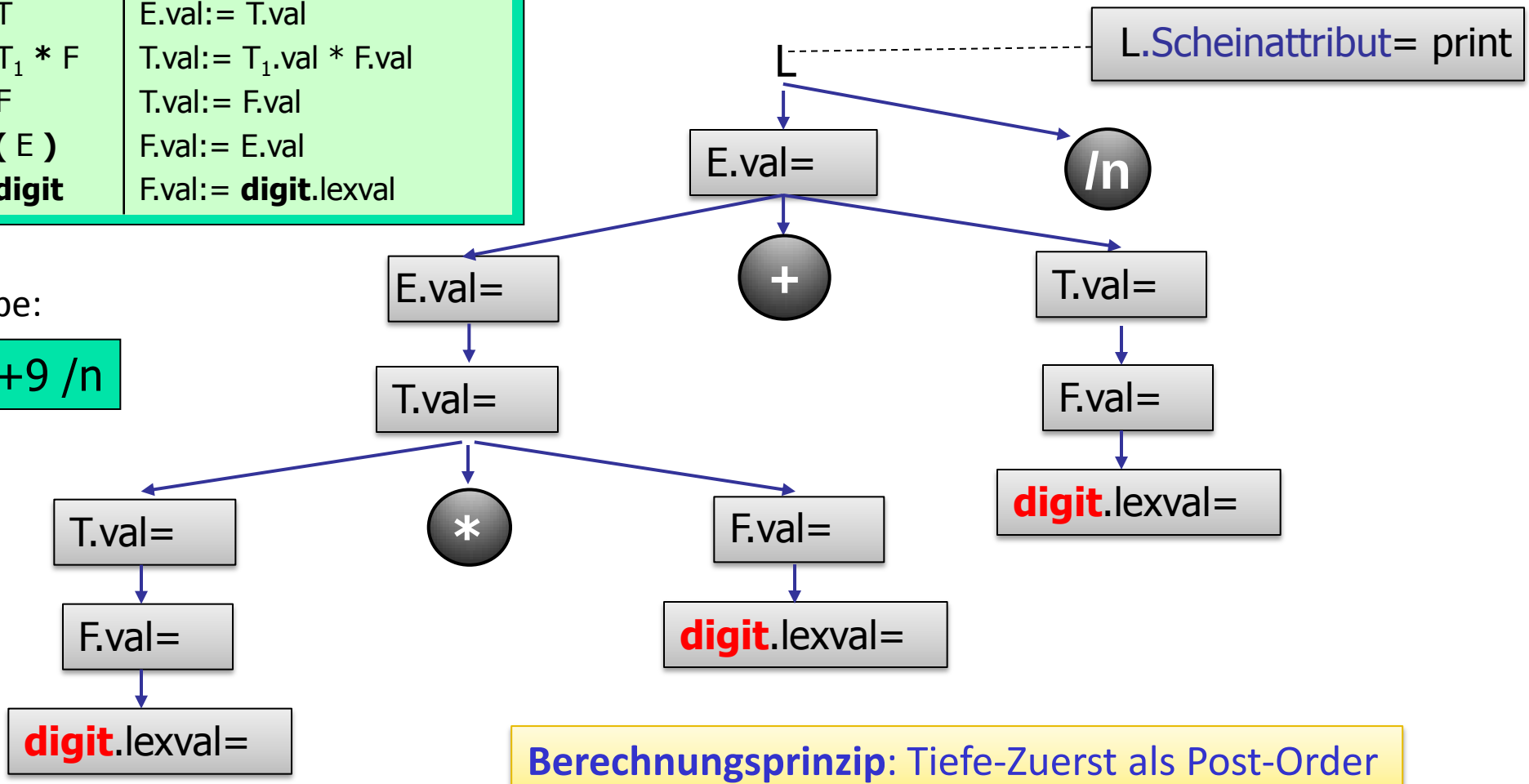
nur Werte von Nachfolgerknoten zur Berechnung: S-Attribut-Grammatik
unproblematisch für Bottom-Up (leicht von LR-Parsern zu implementieren)

Berechnung der Attributwerte

Produktion	Semantische Regel
L → E **/n**	L.val:= print(E.val)
E → E_1 **+** T	E.val:= E_1.val + T.val
E → T	E.val:= T.val
T → T_1 * F	T.val:= T_1.val * F.val
T → F	T.val:= F.val
F → **(** E **)**	F.val:= E.val
F → **digit**	F.val:= **digit**.lexval

Eingabe:

4*3+9 /n



Berechnungsprinzip: Tiefe-Zuerst als Post-Order

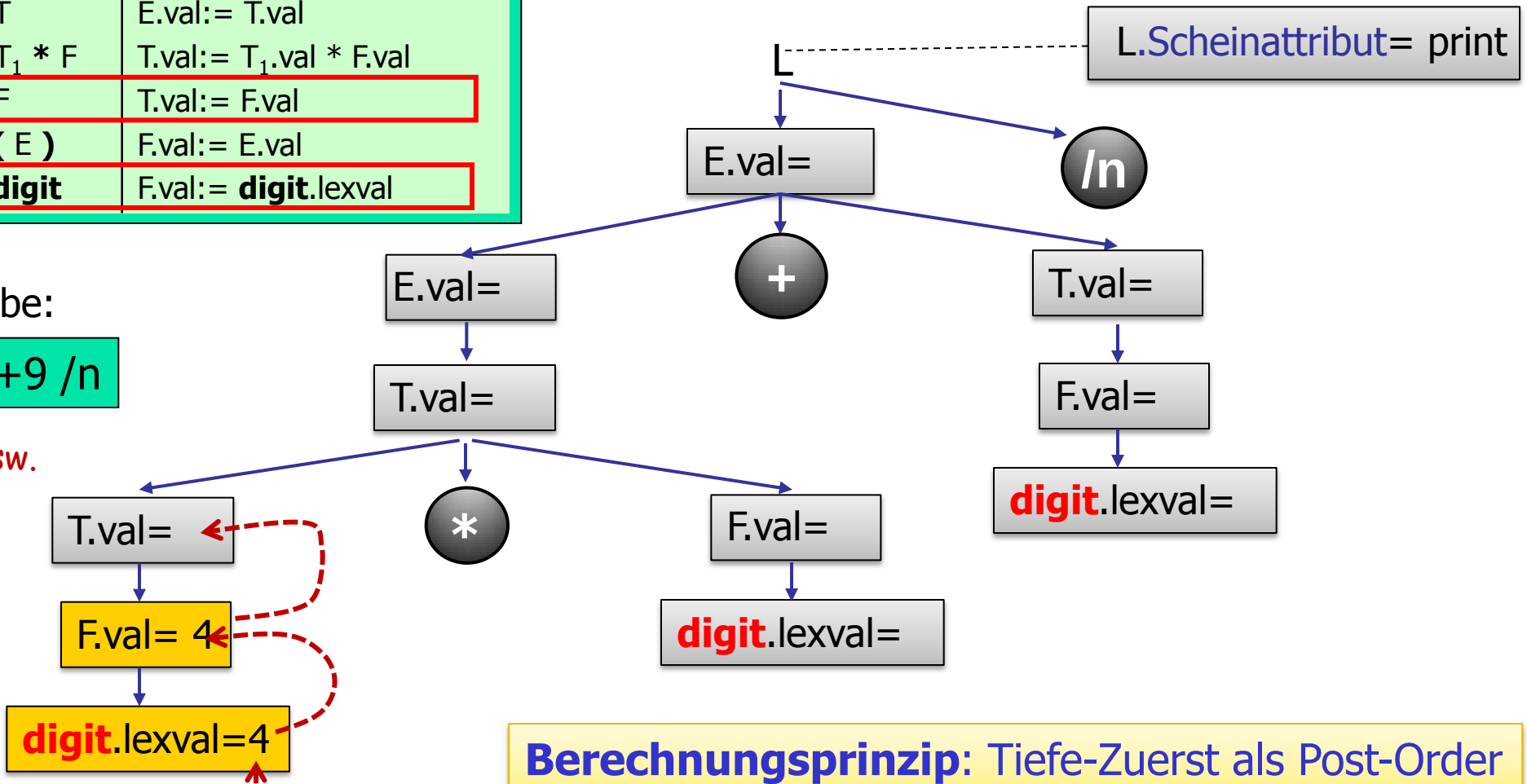
Berechnung der Attributwerte

Produktion	Semantische Regel
L → E **/n**	L.val:= print(E.val)
E → E_1 **+** T	E.val:= E_1.val + T.val
E → T	E.val:= T.val
T → T_1 * F	T.val:= T_1.val * F.val
T → F	T.val:= F.val
F → **(** E **)**	F.val:= E.val
F → **digit**	F.val:= **digit**.lexval

Eingabe:

4*3+9 /n

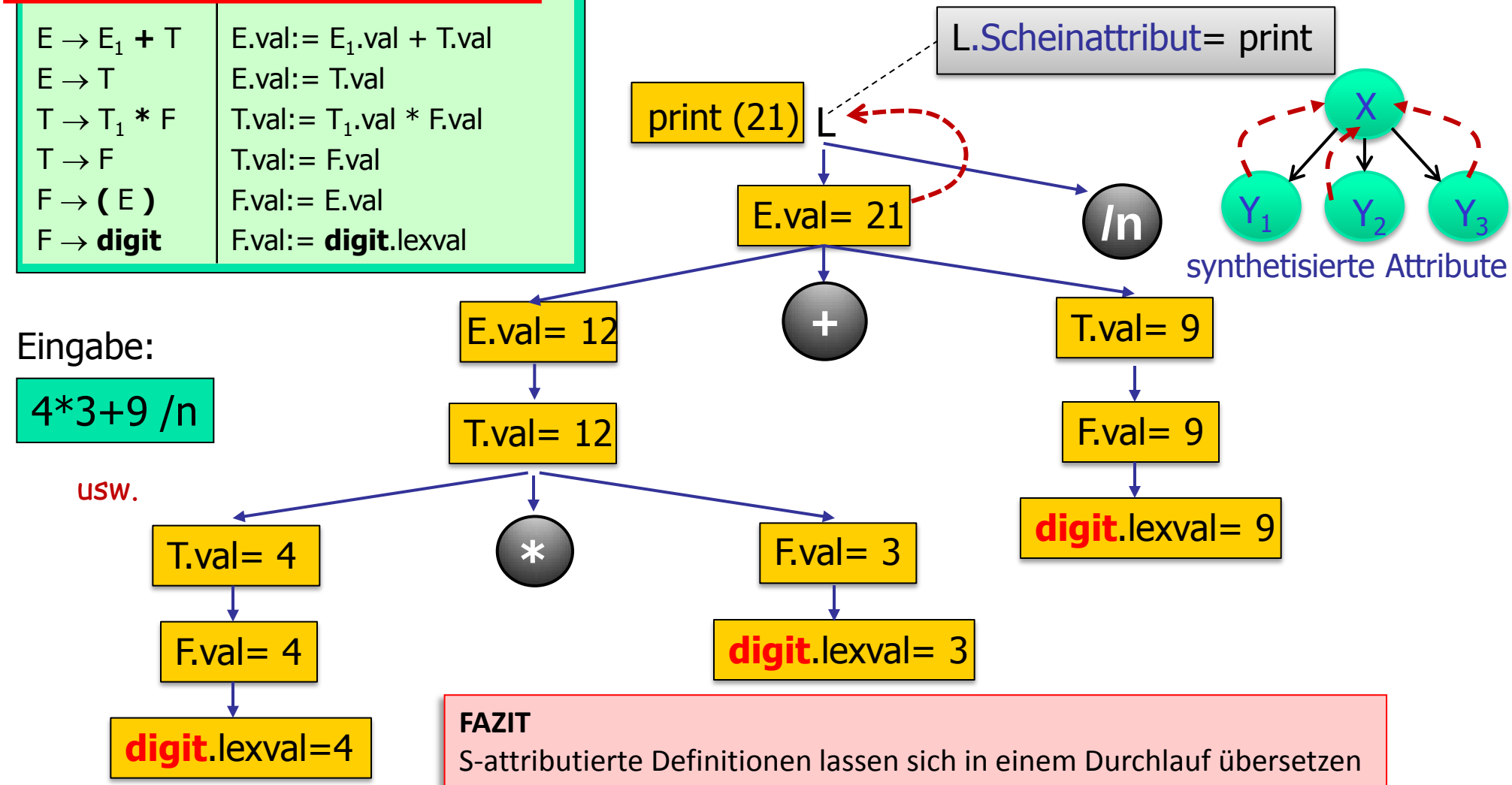
usw.



Berechnungsprinzip: Tiefe-Zuerst als Post-Order

Berechnung der Attributwerte

Produktion	Semantische Regel
L → E **/n**	L.val:= print(E.val)
E → E_1 **+** T	E.val:= E_1.val + T.val
E → T	E.val:= T.val
T → T_1 * F	T.val:= T_1.val * F.val
T → F	T.val:= F.val
F → **(** E **)**	F.val:= E.val
F → **digit**	F.val:= **digit**.lexval



Eingabe:

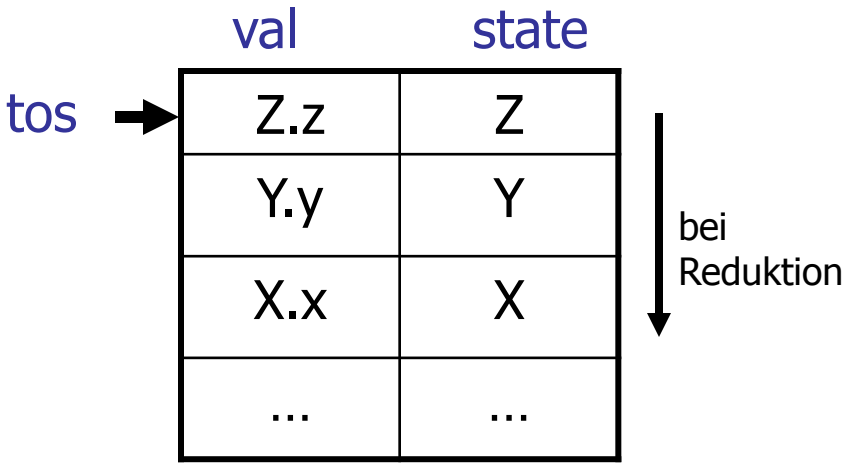
4*3+9 /n

usw.

FAZIT
S-attributierte Definitionen lassen sich in einem Durchlauf übersetzen (besitzen die L-Eigenschaft)

Synthetisierte Attribute auf dem Parser-Kellerspeicher

Parser-Keller mit zusätzlichen Feldern für Attribute (hier: 1)



Produktion	Semantische Regel
...	...
A → X Y Z	A.a:= f(X.x, Y.y, Z.z)
...	...

syntaxgesteuerte Definition (S-Attributgrammatik)

falls mehrere Attribute pro Symbol:

$i.a1 YYSTYPE *muss in der Variante*
$i.a2 *des Symbols* a1, a2 *und* a3
$i.a3 *als Komponenten aufweisen*

- **bevor** XYZ auf A reduziert wird, ist

  val[tos] = $1= Z.z

  val[tos-1]= $2= Y.y

  val[tos-2]= $3= X.x

- **nach** Reduktion XYZ → A:

  tos:= tos-2; state[tos]:= A; val[tos]:=A.a

Zwischenfazit

S-Attributgrammatik
zwar effizient berechenbar,
unterliegt aber gewissen Einschränkungen
(selten praktisch nutzbar)

- Suche nach Möglichkeit zur **einfachen Implementation**
  von solchen syntaxgesteuerten Definitionen,
  die neben synthetisierten auch gewisse ererbte Attribute zulassen

sinnvolle Beispiele kommen noch!

einfache Implementation
natürliche Navigation durch Baumstrukturen:
Tiefe-Zuerst (angewendet auf die Baumwurzel)

FRAGE: Welche Art von Attributabhängigkeit wird dabei vorausgesetzt?

ANTWORT: führt zur Definition von L-attributierten syntaxgesteuerten Definitionen

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbesch
- Kapitel 3
  Lexikalische Analyse: der Sca
- Kapitel 4
  Syntaktische Analyse: der Pa
- Kapitel 5
  Parser-Generatoren: Yacc, Bis
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerierung

6.1
Überblick: Grammatik-basierte Übersetzung

6.2
Attribut-Grammatiken

6.3
S-attributierte Syntaxdefinitionen

6.4
L-attributierte Syntaxdefinitionen

6.5
Syntaxgesteuerte Übersetzungen im Überblick

6.6
Entwurf syntaxgesteuerter Übersetzungen

6.7
Symboltabelle

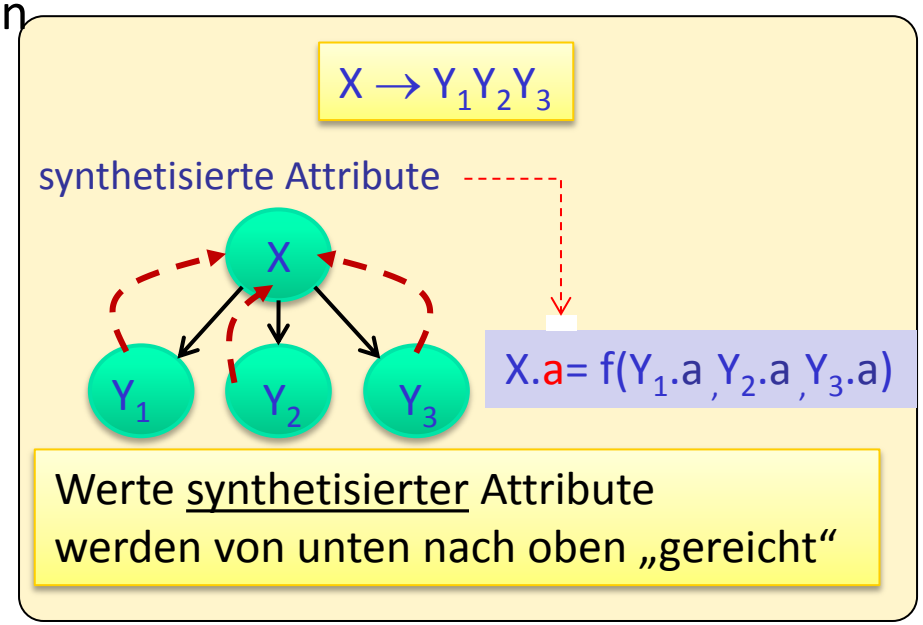
Implementierung syntaxgesteuerter Definitionen

- **schwierig**
  Implementierung eines Compilers, der einer beliebigen syntaxgesteuerten Definition folgt

- **Klassifizierung** syntaxgesteuerter Definitionen

S-attributierte Definitionen
(enthalten nur synthetisierte Attribute)

können durch **LL**- und **LR-Parser**
on-the-fly
ausgewertet werden



- Parser kann Attribute der Metasymbole (RS)
  zusätzlich zu den Metasymbolen auf dem Stack festhalten
- bei einer Reduktion stehen dann diese Werte zur Verfügung, um die Attribute (LS) zu bestimmen

Beispiel: inh-syn-Attribut-Mix

Produktion	Semantische Regel
$T \rightarrow F\ T'$	
$T' \rightarrow *\ F\ T'$	
$T' \rightarrow \varepsilon$	
$F \rightarrow$ **digit**	

Ausschnitt einer nicht-linksrekursiven Grammatik (prinzipiell geeignet für TopDown-Parsing)

Aufgabe der Regeln:
Berechnung des Wertes
für eine gültige Eingabe der Ausdrucksgrammatik

- jedes der **Nichtterminalsymbole** (T, F):
  synthetisiertes Attribut val
- **Terminalsymbol**:
  synthetisiertes Attribut lexval
- **Nichtterminalsymbol** T´: zwei Attribute
  synthetisiertes Attribut syn
  geerbtes Attribut inh

Beispiel: inh-syn-Attribut-Mix

Produktion	Semantische Regel
$T \rightarrow F\ T'$	$T'.inh := F.val$ $T.val := T'.syn$
$T' \rightarrow *\ F\ T'_1$	$T'_1.inh := T'.inh * F.val$ $T'.syn := T'_1.syn$
$T' \rightarrow \varepsilon$	$T'.syn := T'.inh$
$F \rightarrow$ **digit**	$F.val :=$ **digit**.lexval

Aufgabe der Regeln:
Berechnung des Wertes
für eine gültige Eingabe der Ausdrucksgrammatik

- jedes der **Nichtterminalsymbole** (T, F):
  synthetisiertes Attribut val
- **Terminalsymbol**:
  synthetisiertes Attribut lexval
- **Nichtterminalsymbol** T´: zwei Attribute
  synthetisiertes Attribut syn
  geerbtes Attribut inh

Idee für Semantikregeln
linker Operand von * wird vererbt

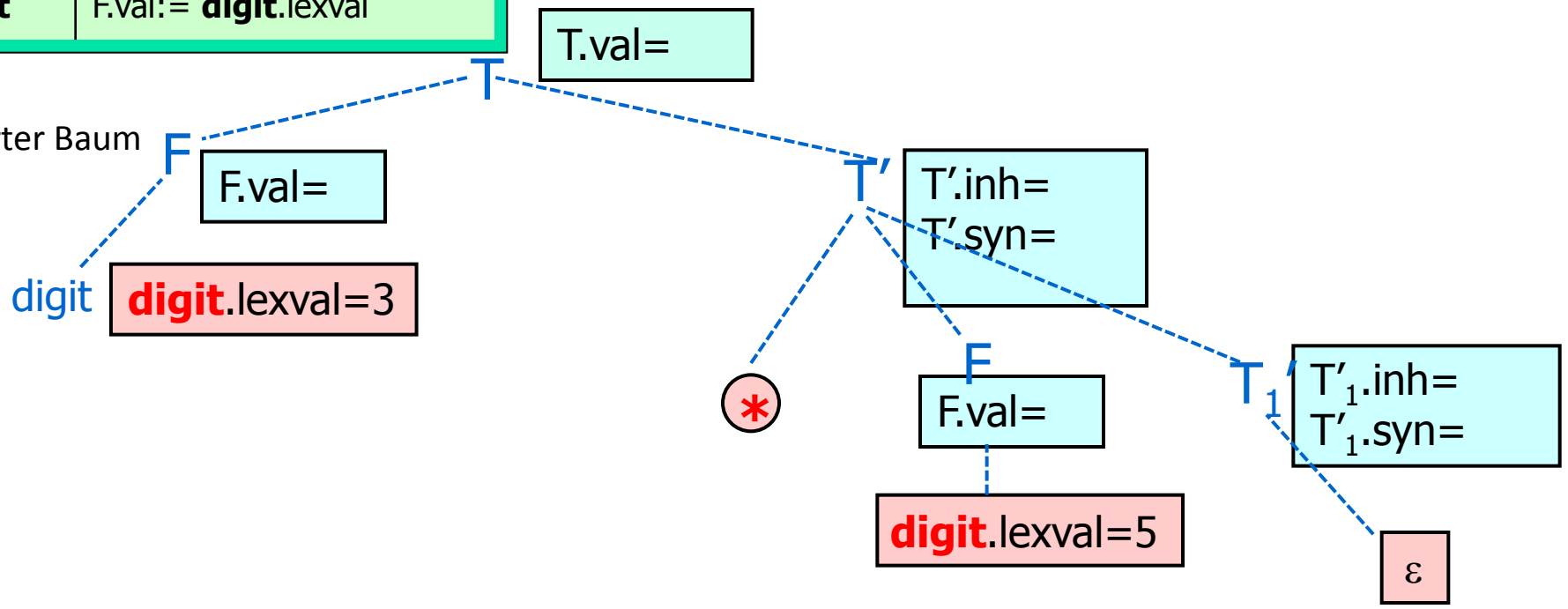
Term x*y*z
- Wurzel des Teilbaums *y*z erbt von x
- dann erbt Wurzel des Teilbaums *z von x*y usw.

Schritt 0

Produktion	Semantische Regel
$T \rightarrow F\ T'$	T'.inh:= F.val T.val:= T'.syn
$T' \rightarrow *\ F\ T'_1$	T'_1.inh:= T'.inh * F.val T'.syn:= T'_1.syn
$T' \rightarrow \varepsilon$	T'.syn:= T'.inh
F → **digit**	F.val:= **digit**.lexval

Aufbau des Parsebaumes mit Knotenattributen

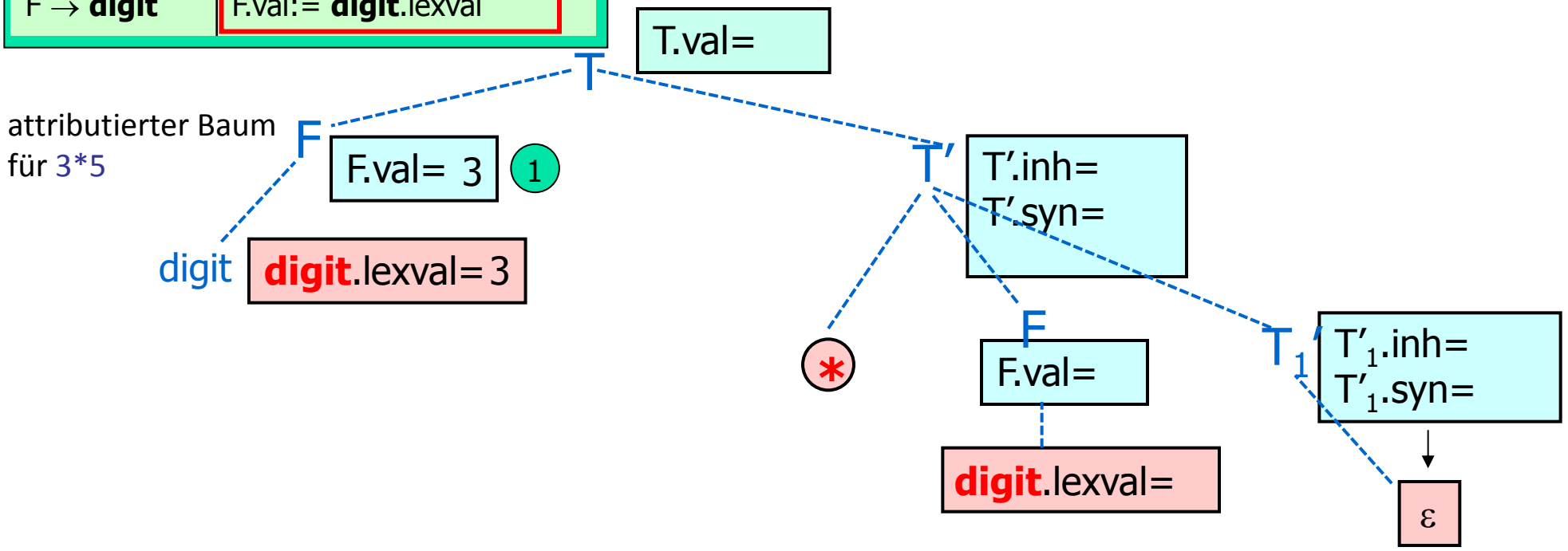
attributierter Baum für 3*5



Schritt 1

Produktion	Semantische Regel
$T \rightarrow F\ T'$	T'.inh:= F.val T.val:= T'.syn
$T' \rightarrow * F\ T'_1$	T'_1.inh:= T'.inh * F.val T'.syn:= T'_1.syn
$T' \rightarrow \varepsilon$	T'.syn:= T'.inh
$F \rightarrow$ **digit**	F.val:= **digit**.lexval

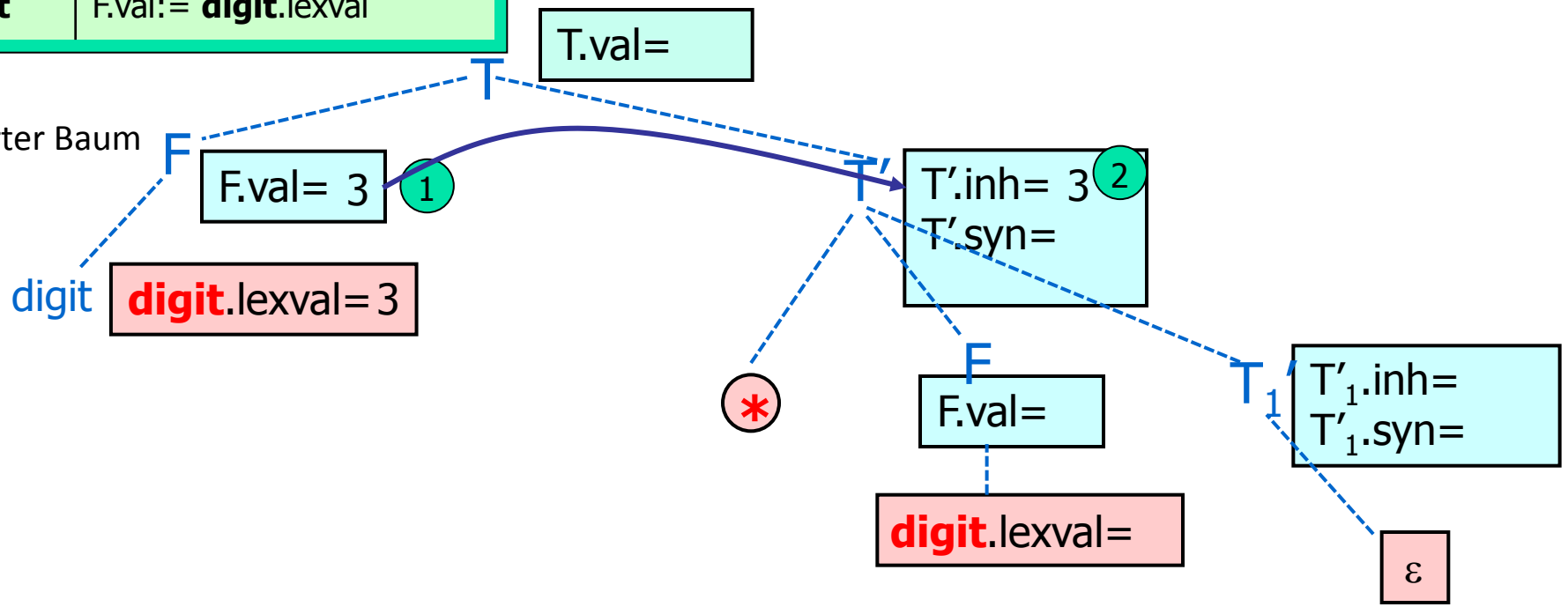
attributierter Baum für 3*5



Schritt 2

Produktion	Semantische Regel
$T \rightarrow F\ T'$	T'.inh:= F.val
	T.val:= T'.syn
$T' \rightarrow * F\ T'_1$	T'_1.inh:= T'.inh * F.val
	T'.syn:= T'_1.syn
$T' \rightarrow \varepsilon$	T'.syn:= T'.inh
$F \rightarrow$ **digit**	F.val:= **digit**.lexval

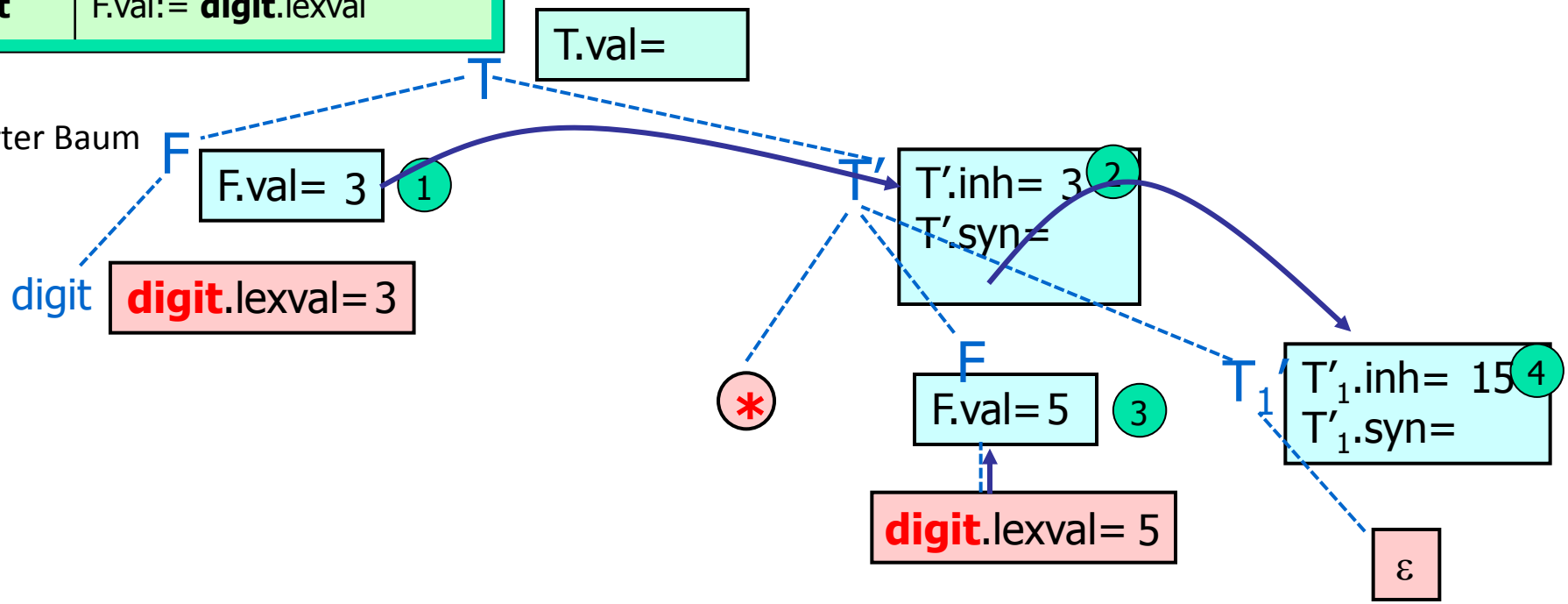
attributierter Baum
für 3*5



Schritte 3 und 4

Produktion	Semantische Regel
$T \rightarrow F\ T'$	T'.inh:= F.val T.val:= T'.syn
$T' \rightarrow * F\ T'_1$	T'_1.inh:= T'.inh * F.val T'.syn:= T'_1.syn
$T' \rightarrow \varepsilon$	T'.syn:= T'.inh
$F \rightarrow$ **digit**	F.val:= **digit**.lexval

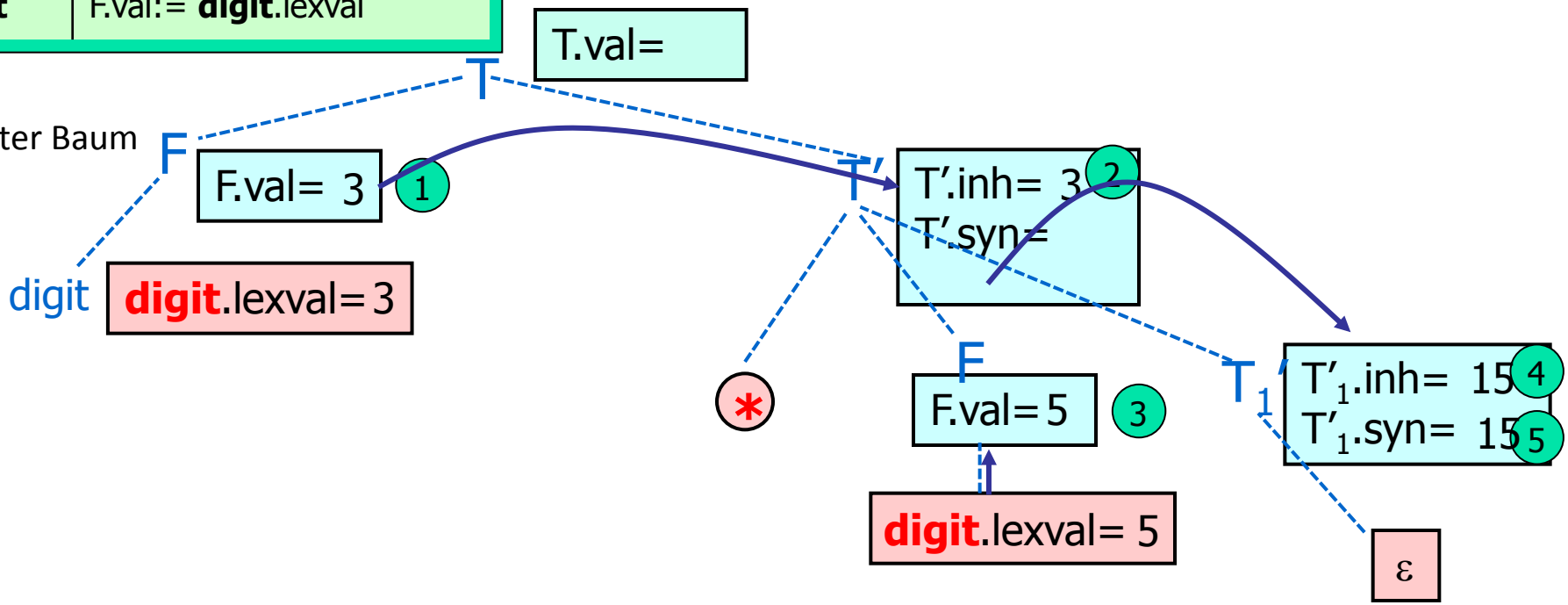
attributierter Baum für 3*5



Schritt 5

Produktion	Semantische Regel
$T \rightarrow F\ T'$	T'.inh:= F.val T.val:= T'.syn
$T' \rightarrow * F\ T'_1$	T'_1.inh:= T'.inh * F.val T'.syn:= T'_1.syn
$T' \rightarrow \varepsilon$	T'.syn:= T'.inh
F → **digit**	F.val:= **digit**.lexval

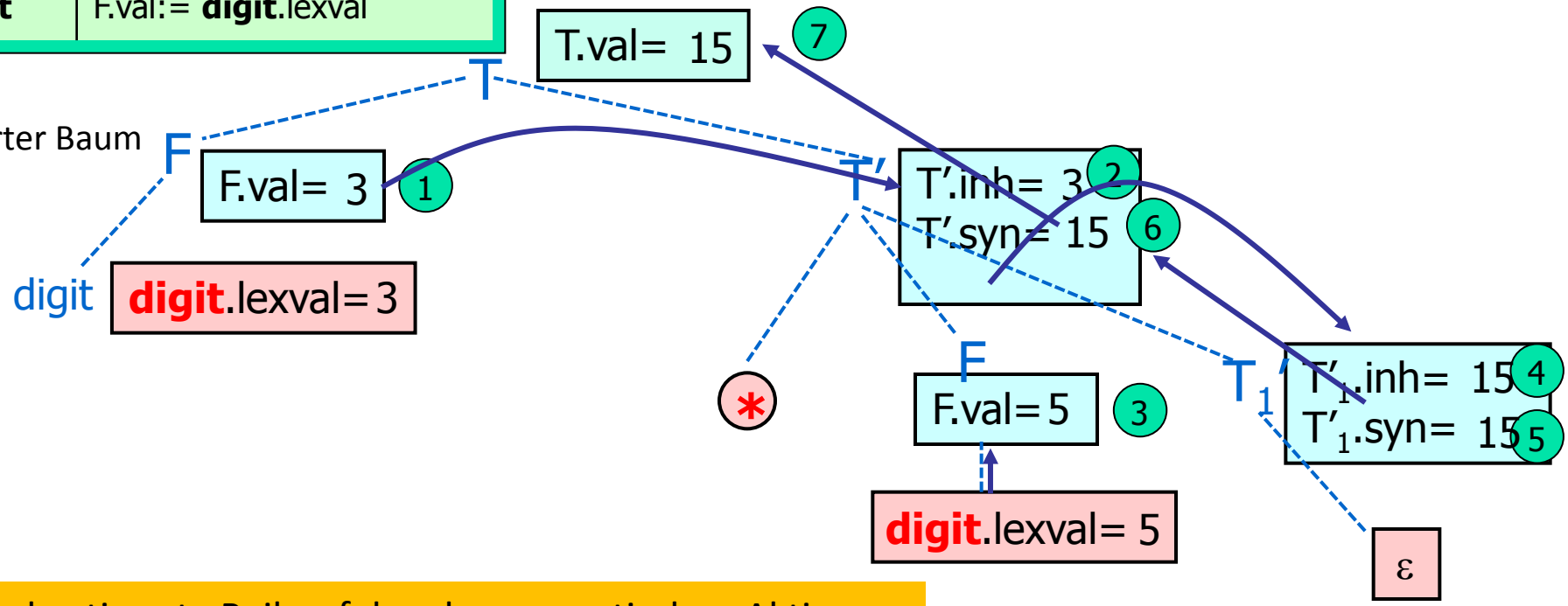
attributierter Baum für 3*5



Schritte 6 und 7

Produktion	Semantische Regel
$T \rightarrow F\ T'$	T'.inh:= F.val
	T.val:= T'.syn
$T' \rightarrow *\ F\ T'_1$	T'_1.inh:= T'.inh * F.val
	T'.syn:= T'_1.syn
$T' \rightarrow \varepsilon$	T'.syn:= T'.inh
$F \rightarrow$ **digit**	F.val:= **digit**.lexval

attributierter Baum für 3*5



Fazit: so bestimmte Reihenfolge der semantischen Aktionen garaniert die korrekte Wertermittlung

2. Beispiel: Typdeklaration von Bezeichern

Produktion	semantische Regel
$D \rightarrow T\ L$	
$T \rightarrow$ **int**	
$T \rightarrow$ **real**	
$L \rightarrow L_1$, **id**	
$L \rightarrow$ **id**	

Aufgabe der Regeln:
Übernahme der Typinformation eines Bezeichners aus einer Typdeklaration in die Symboltabelle

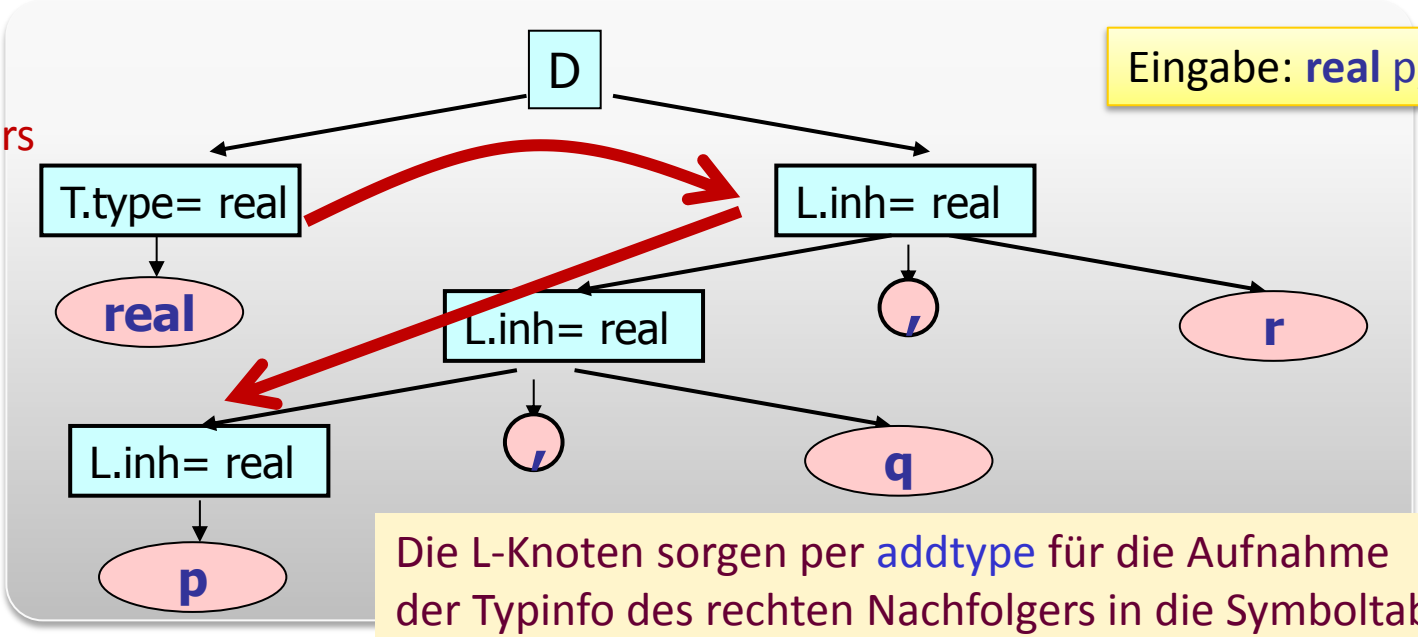
2. Beispiel: Typdeklaration von Bezeichern

Produktion	semantische Regel
D → T L	L.inh := T.type
T → **int**	T.type := integer
T → **real**	T.type := real
L → L_1, **id**	L_1.inh := L.inh
	addtype (**id**.entry, L.inh)
L → **id**	addtype (**id**.entry, L.inh)

als geerbtes Attribut inh wird die Typinformation top-down-artig im Baum verbreitet

Hinzufügen der Typinformation zum vorhandenen Bezeichner **id** in der Symboltabelle im Blattknoten

Bewegung des Parsers



Eingabe: **real** p, q, r

Die L-Knoten sorgen per addtype für die Aufnahme der Typinfo des rechten Nachfolgers in die Symboltabelle