Abhängige Berechnungen

Der Wert eines Ausdrucks wird berechnet und ausgegeben:

```
ATTR value: int;

RULE: Root ::=  Expr  COMPUTE
   printf ("value is %d\n", Expr.value);
END;

TERM Number: int;

RULE: Expr ::= Number COMPUTE
  Expr.value = Number;
END;

RULE: Expr ::= Expr Opr Expr COMPUTE
  Expr[1].value = Opr.value;
  Opr.left  = Expr[2].value;
  Opr.right = Expr[3].value;
END;

SYMBOL Opr: left, right: int;

RULE: Opr ::=  '+'  COMPUTE
  Opr.value  = ADD (Opr.left, Opr.right);
END;

RULE: Opr ::=  '*'  COMPUTE
  Opr.value = MUL (Opr.left, Opr.right);
END;
```

Vor- und Nachbedingungen von Berechnungen

Ausgabe von Ausdrücken in Postfix-Form:

```
RULE: Root ::= Expr COMPUTE
  Expr.print = "yes";
  printf ("\n") <- Expr.printed;
END;

RULE: Expr ::= Number COMPUTE
  Expr.printed =
    printf ("%d ", Number) <- Expr.print;
END;

RULE: Opr  ::= '+' COMPUTE
  Opr.printed = printf ("+ ") <- Opr.print;
END;

RULE: Opr  ::= '*' COMPUTE
  Opr.printed = printf ("* ") <- Opr.print;
END;

RULE: Expr  ::= Expr Opr Expr COMPUTE
  Expr[2].print = Expr[1].print;
  Expr[3].print = Expr[2].printed;
  Opr.print = Expr[3].printed;
  Expr[1].printed = Opr.printed;
END;
```

Die Attribute `print` und `printed` haben keinen Wert `(Typ VOID)`.

Sie beschreiben Zustände als Vor- und Nachbedingungen für Berechnungen:

`Expr.print`: Postfix-Ausgabe bis vor diesem Knoten ist erledigt.

`Expr.printed`: Postfix-Ausgabe bis einschließlich dieses Knotens ist erledigt.

Abhängigkeitsmuster: CHAIN

CHAIN spezifiziert **left-right-depth-first** Abhängigkeit.
Berechnungen werden in die CHAIN „eingehängt“.

Ausgabe von Ausdrücken in Postfix-Form:

```
CHAIN print: VOID;

RULE: Root ::= Expr COMPUTE
  CHAINSTART HEAD.print = "yes";
  printf ("\n ") <- TAIL.print;
END;

RULE: Expr ::= Number COMPUTE
  Expr.print =
    printf ("%d ", Number) <- Expr.print;
END;

RULE: Opr ::= '+' COMPUTE
  Opr.post = printf ("+") <- Opr.pre;
END;

RULE: Expr ::= Expr Opr Expr COMPUTE
  Opr.pre = Expr[3].print;
  Expr[1].print = Opr.post;
END;
```

Wertberechnung entlang einer CHAIN, z. B. Variablenadressen:

```
CHAIN RelAdr: int;

RULE: Block ::= '{' Sequence '}' COMPUTE
  CHAINSTART HEAD.RelAdr = 0;
END;

RULE: Definition ::= 'define' Ident COMPUTE
  Definition.RelAdr =
    ADD (Definition.RelAdr, VariableSize);
END;
```

Die Kontexte, in denen die CHAIN nicht benutzt wird, brauchen nicht erwähnt zu werden!

Abhängigkeitsmuster: INCLUDING

Ein Attribut wird tiefer im Unterbaum benutzt.

`INCLUDING Block.depth` bezieht sich auf das `depth`-Attribut des nächsten Oberknotens `Block`.

Berechnung der Schachtelungstiefe von Blöcken:

```
ATTR depth: int;

RULE: Root ::= Block COMPUTE
  Block.depth = 0;
END;

RULE: Statement ::= Block COMPUTE
  Block.depth = ADD (INCLUDING Block.depth, 1);
END;

TERM Ident: int;

RULE: Definition ::= 'define' Ident  COMPUTE
  printf ("%s defined on depth %d\n ",
           StringTable (Ident),
           INCLUDING Block.depth);
END;
```

Die Kontexte zwischen der Definition und der Benutzung der `INCLUDING`-Attribute brauchen nicht erwähnt zu werden!

Abhängigkeitsmuster: CONSTITUENTS für Zustände

Zusammenfassen von Attributen aus Unterbaum

`CONSTITUENTS Definition.DefDone` bezieht sich auf die `DefDone`-Attribute aller `Definition`-Knoten im Unterbaum.

Ausgabe aller Definitionen vor allen Anwendungen:

```
RULE: Block ::= '{' Sequence '}' COMPUTE
  Block.DefDone =
    CONSTITUENTS Definition.DefDone;
END;
RULE: Definition ::= 'Define' Ident COMPUTE
  Definition.DefDone =
    printf ("%s defined in line %d\n",
            StringTable(Ident), LINE);
END;
RULE: Usage ::= 'use' Ident COMPUTE
   printf ("%s used in line %d\n ",
           StringTable(Ident), LINE),
   <- INCLUDING BLOCK.DefDone;
END;
```

Die Kontexte zwischen der Definition und der Benutzung der `CONSTITUENTS`-Attribute brauchen nicht erwähnt zu werden!

Abhängigkeitsmuster: CONSTITUENTS für Werte

Zusammenfassung von Werten erfordert

```
CONSTITUENTS ... WITH (Type, BinFct, UnFct, ZeroFct)
```

`Type`	Typ des Ergebnis und der propagierten Werte
`BinFct`	verknüpft zwei Werte; sollte assoziativ sein
`UnFct`	erzeugt einen Wert aus dem Attributwert
`ZeroFct`	erzeugt einen "neutralen" Wert

Ausgabe der Anzahl der Anwendungen:

```
ATTR Count: int;

RULE: Block ::= '{' Sequence '}' COMPUTE
  printf  ("%d uses occurred\n",
           CONSTITUENTS Usage.Count
           WITH (int, ADD, IDENTICAL, ZERO));
END;

RULE: Usage ::= 'use' Ident COMPUTE
  Usage.Count = 1;
END;
```

Die Werte werden gemäß der Baumstruktur zusammengefaßt.