Berechnungen zu Symbolen

Berechnungen können Symbolen zugeordnet werden, wenn sie für jeden RULE-Kontext gleich sind, z. B.

```
SYMBOL Expr COMPUTE
  printf("expression value %d in line %d\n", THIS.value, LINE);
END;
```

Symbolberechnungen können auch `INCLUDING-`, `CONSTITUENTS-`, oder `CHAIN-`Konstrukte enthalten:

```
SYMBOL Block COMPUTE
   printf ("%d uses occurred\n",
           CONSTITUENTS Usage.Count SHIELD Block
           WITH (int, ADD, IDENTICAL, ZERO));
END;

SYMBOL Usage COMPUTE
  SYNT.Count = 1;
END;
```

`SYNT.a` bzw. `INH.a` gibt an, daß die Berechnung zum unteren bzw. oberen Kontext des Symbols gehört.

Berechnungen im `RULE`-Kontext überschreiben Berechnungen im SYMBOL-Kontext für dasselbe Attribut.

Verwendung für Rekursionsanfänge oder für Ausnahmen von der Symbolberechnung:

```
SYMBOL Block COMPUTE
  INH.depth = ADD (INCLUDING Block.depth, 1);
END;

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

Berechnungsmodule mit CLASS-Symbolen

Wiederverwendung von Berechnungsmustern

Berechnungen werden neuen CLASS-Symbolen zugeordnet und durch **`INHERITS`** an Baum-Symbole gebunden.

```
ATTR Sym: int;
CLASS SYMBOL IdOcc COMPUTE SYNT.Sym = TERM; END;

SYMBOL DefIdent INHERITS IdOcc END;
SYMBOL UseIdent INHERITS IdOcc END;
```

Zusammenwirkende CLASS-Symbole:

z. B. Auftreten eines syntaktischen Konstruktes zählen:

```
CHAIN Occurrence: int;
ATTR OccNo, TotalOccs: int;

CLASS SYMBOL OccRoot COMPUTE
  CHAINSTART HEAD.Occurrence = 0;
  THIS.TotalOccs = TAIL.Occurrence;
END;

CLASS SYMBOL OccElem COMPUTE
  SYNT.OccNo = THIS.Occurrence;
  THIS.Occurrence = ADD (SYNT.OccNo, 1);
END;
```

Benutzung:

```
SYMBOL Block INHERITS OccRoot END;
SYMBOL Definition INHERITS OccElem END;

SYMBOL Statement INHERITS OccRoot END;
SYMBOL Usage INHERITS OccElem END;
```

Auf diesem Prinzip beruht die Benutzung Bibliotheksmodulen, z. B. zur Namensanalyse.

Bibliotheksmodule allgemein

Bibliotheksmodule machen vorgefertigte Lösungen für häufig auftretende Aufgaben verfügbar.

1. Modul in der Dokumentation identifizieren, Beschreibung lesen

2. Modul instantiieren:

   $/Prop/Unique.gnrc : inst

3. Symbolrollen Grammatiksymbolen zuordnen und Attribute benutzen:

```
SYMBOL Program  INHERITS RangeUnique END;
SYMBOL DefIdent INHERITS Unique COMPUTE
  IF (NOT (THIS.Unique),
  message (ERROR,
           CatStrInd
           ("identifier is multiply defined: ",
             THIS.Sym),
           0, COORDREF));
END;
```

4. Wenn nötig Modul-Implementierung beschaffen und lesen:

   -> $elipkg**/Prop/Unique.gnrc : inst :** viewlist

ADT Listen

Generischer Modul zur Listenimplementierung

```
$/Adt/List.gnrc+instance=DefTableKey +referto=deftbl :inst
```

Hier: Elementtyp `DefTableKey` definiert in Datei `deftbl.h`

Es werden der Typ `DefTableKeyList` und Funktionen wie

`ConsDefTableKeyList, HeadDefTableKeyList, TailDefTableKeyList,` ...und viele andere

als C-Modul verfügbar.

Lido-Unterstützung zur Listenbenutzung:

```
$/Adt/LidoList.gnrc+instance=DefTableKey+referto=deftbl :inst
```

Listen in Baumkontexten aufbauen:

```
RULE: FunctionHead ::= '(' Parameters ')' TypeDenoter
COMPUTE
   ...ResetParamTypes (FunctionHead.Type,
                       Parameters.DefTableKeyList) ...
END;
SYMBOL Parameters INHERITS DefTableKeyListRoot END;
SYMBOL Parameter  INHERITS DefTableKeyListElem END;
RULE: Parameter ::= TypeDenoter DefIdent COMPUTE
  Parameter.DefTableKeyElem = TypeDenoter.Type;
END;
```

Listen in Baumkontexten abbauen entsprechend.

Weitere ADT-Module: BitSet, Stack