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;
```

Praktikum Sprachimplementierung mit Werkzeugen WS 1999/ 2000 / Folie 401

Ziele:

Berechnungen zu Symbolen statt identischer Berechnungen in vielen Kontexten.

im Vorlesungsteil:

Erläuterung der Beispiele

nachlesen:

Computation in Trees: 4.1

Verständnisfragen:

Welche Berechnungen von Folie 3.1 können nicht in Symbolberechnungen umgeschrieben werden? Warum?

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.

Praktikum Sprachimplementierung mit Werkzeugen WS 1999/ 2000 / Folie 402

Ziele:

Abstraktion durch zusammenwirkende CLASS-Symbole

im Vorlesungsteil:

- Erläuterung der Beispiele
- Benutzung von Bibliotheksmodulen

nachlesen:

Computation in Trees: 4.2

Übungsaufgaben:

Instantiieren Sie den Bibliotheksmodul Unique und erläutern Sie die Rollen seiner CLASS-Symbole.

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
```

Praktikum Sprachimplementierung mit Werkzeugen WS 1999/ 2000 / Folie 403

Ziele:

Einfache Modulbenutzung

im Vorlesungsteil:

- Modul Instantiierung
- Benutzung der Symbolrollen

nachlesen:

Modlib-Dokument: Instantiation and Use; Modul Unique

Übungsaufgaben:

- Mit welchem allgemeineren Modul der Bibliothek kann die Aufgabe des Unique-Moduls auch gelöst werden? geben Sie die Unterschiede an.

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

Praktikum Sprachimplementierung mit Werkzeugen WS 1999/2000 / Folie 404

Ziele:

ADT-Module kennenlernen

im Vorlesungsteil:

Erläuterungen dazu

nachlesen:

Modlib-Dokument: Abstract Data Types, Lists in LIDO Specifications, Linear Lists of Any Type

Übungsaufgaben:

Geben kurz Sie an, wozu der Modul PtrList nützlich ist.

Verständnisfragen:

Wie wie lauten die Instantiierungsparameter für float-Listen?