Namensanalyse (Bezeichneridentifikation)

Bezeichner benennen (statische) Programmobjekte.

Definition für Bezeichner b führt neues Programmobjekt ein und bindet es an den Bezeichner.

Die Bindung gilt in einem bestimmten Bereich des Programms: **Gültigkeitsbereich** der Definition.

Aufgabe: jedem Auftreten eines Bezeichners den Schlüssel des Programmobjektes zuordnen. *(konsistente Umbenennung)*

Grundlage: Gültigkeitsregeln der Sprache *(scope rules)*

Verdeckungsregeln für Sprachen mit geschachtelten Strukturen:

- **Algol-Regel:** Die Definition eines Bezeichners b gilt im **ganzen** kleinsten sie umfassenden Abschnitt, aber nicht in darin enthaltenen Abschnitten mit einer Definition von b.

- **C-Regel:** Die Definition eines Bezeichners b gilt im kleinsten sie umfassenden Abschnitt **von der Definitionsstelle an**, aber nicht in darin enthaltenen Abschnitten mit einer Definition von b von der Definitionsstelle an**.**

Abschnitt je nach Sprache: Block. Prozedur, Modul, ...

Implementierung: Operationen des **Umgebungsmoduls** in Baumkontexten aufrufen.

Umgebungsmodul

implementiert den abstrakten Datentyp **Environment**:

hierarchisch geschachtelte Mengen von Bindungen
(Bezeichner, Schlüssel)

Funktionen:

NewEnv () erzeugt ein neues Environment e als Wurzel.

NewScope (e1) erzeugt neues Environment e2.
Alle Bindungen aus e1 sind auch in e2, falls sie nicht durch Bindungen in e2 verdeckt werden.

DefineIdn (e, b) fügt Bindung (b, k) in e ein, falls sie dort noch nicht exisitiert;
k ist dann ein neuer Objektschlüssel.
Liefert in jedem Fall den an b gebundenen Schlüssel k.

KeyInEnv (e, b) liefert den Schlüssel k eines Paares (b, k) aus e (einschließlich aller e_i, aus denen e erzeugt wurde).
Liefert NoKey, falls kein solches Paar existiert.

KeyInScope (e, b) liefert den Schlüssel eines Paares (b, k), falls unmittelbar in e enthalten, NoKey sonst.

Umgebungsoperationen im Baum

Operationen in Baumkontexten und Reihenfolge ihrer Ausführung modelliert Gültigkeitsregeln.

Wurzelkontext `Root`:

```
Root.Env = NewEnv ();
```

Abschnitte `Range`, die Definitionen enthalten können:

```
Range.Env =
  NewScope(INCLUDING (Range.Env, Root.Env));
            nächst umgebende Range oder Root
```

definierendes Auftreten eines Bezeichners `IdDefScope`:

```
IdDefScope.Key =
  DefineIdn(INCLUDING Range.Env,
             IdDefScope.Symb);
```

angewandtes Auftreten eines Bezeichners `IdUseEnv`:

```
IdUseEnv.Key =
  KeyInEnv (INCLUDING Range.Env,
             IdUseEnv.Symb);
```

Vorbedingungen je nach Gültigkeitsregeln:
Algol-Regel: Vorbedingung für `KeyInEnv`:
alle `DefineIdn` aller umgebenden Ranges sind ausgeführt.

C-Regel: `KeyInEnv` und `DefineIdn` werden in der Reihenfolge des Programmtextes ausgeführt (links-abwärts).

weitere Prüfungen an dem Ergebnis der Bez.identifikation:

- keine Anwendung ohne Definition
- keine 2 Definitionen für einen Bezeichner in einem Range
- keine Anwendung vor der Definition (Pascal)
- Vorwärtsdefinitionen
- ...

Module zur Namensanalyse

Symbolrollen:

Wurzel der Grammatik:

```
SYMBOL Program INHERITS RootScope END;
```

Abschnitte mit Definitionen:

```
SYMBOL Block INHERITS RangeScope END;
```

Definierter Bezeichner:

```
SYMBOL DefIdent INHERITS IdDefScope END;
```

Angewandte Bezeichner:

```
SYMBOL UseIdent INHERITS IdUseEnv END;
SYMBOL TypeUseIdent INHERITS IdUseEnv END;
```

Benutzbare Attribute:

```
DefIdent.Key, UseIdent.Key
Program.Env, Block.Env
```

Instantiierung für Algol-Scope-Rules:

```
$/Name/AlgScope.gnrc : inst
```

Instantiierung für C-Scope-Rules:

```
$/Name/CScope.gnrc : inst
```

Instantiierung für neuen Namensraum

```
$/Name/AlgScope.gnrc +instance=Label :inst
```

Symbolrollen: **LabelRootScope, LabelRangeScope**, ...

Instantiierung für zusätzliches Key-Attribut:

```
$/Name/AlgScope.gnrc +instance=Label +referto=Lab :inst
```

Attributnamen: `DefIdent.LabKey, UseIdent.LabKey`

PDL: Generator für Definitionsmodule

zentrale Datenstruktur ordnet **Programmobjekten** z*. B. Typen, Variablen, ...* **Eigenschaften** zu
z. B. T*yp einer Variablen, Elementtyp eines Array-Typs.*

Objekte werden durch Schlüssel *(key)* identifiziert.

Operationen:

`NewKey ( )`	liefert neuen Schlüssel
`ResetP (k, v)`	setzt zum Schlüssel `k` die Eigenschaft `P` mit Wert `v`
`SetP (k, v, d)`	setzt (ersetzt) zum Schlüssel `k` die Eigenschaft `P` mit Wert `v` (bzw. `d`)
`GetP (k, d)`	liefert Wert der Eigenschaft `P` des Schlüssels `k`; liefert `d`, falls `P` zu `k` nicht gesetzt ist

Aufrufe: abhängige Operationen im Baum

Implementierung: z. B. Liste von Eigenschaften zu jedem Schlüssel

Generierung des Definitionsmoduls:
Aus Spezifikationen

Eigenschaftsname: Eigenschaftstyp;

werden Funktionen `Reset, Set, Get` generiert.

Richtlinien zum Entwurf von Berechnungen im Baum

1. Zerlege die Aufgabe in **Teilaufgaben**, die klein genug sind, um durch wenige der u. g. Spezifikationsmuster gelöst zu werden. Entwickle für jede Teilaufgabe ein `.lido`-Fragment und erläutere es im umgebenden `.fw`-Text.

2. Arbeite den **zentralen Aspekt der Teilaufgabe** heraus und bilde ihn auf einen der folgenden Fälle ab:

   A. Der Aspekt wird in natürlicher Weise durch **Eigenschaften von einigen Programmkonstrukten** beschrieben, z. B. Typen vo Ausdrücken, Schachtelungstiefe von Blöken, Übersetzung der Anweisungen eines Blockes.

   B. Der Aspekt wird in natürlicher Weise durch **Eigenschaften von Programmobjekten** beschrieben, z. B. Relativadresse von Variablen, Benutzung von Variablen vor der Definition.

   Entwirf die Berechnung wie für A oder B beschrieben.

3. Im Schritt 2 können Anforderungen an **weitere Aspekte** der Teilaufgabe entstehen (benutzte Attribute, deren Berechnung noch nicht entworfen ist): Wiederhole Schritt 2 für diese.

A: Eigenschaften von Programmkonstrukten

Bestimme den **Typ der Werte**, die die Eigenschaft beschreiben. Führe **Attribute diese Typs zu allen Symbolen** ein, die die **Programmkonstrukte** repräsentieren. Prüfe, welcher der folgenden Fälle für die Bestimmung der Eigenschaft am besten zutrifft.

A1: Jeder **untere Kontext** bestimmt die Eigenschaft in unterschiedlicher Weise: Entwirf **RULE-Berechnungen**.

A2: Wie A1; aber **oberer Kontext**.

A3: Die Eigenschaft kann **unabhängig von den RULE-Kontexten** berechnet werden, wobei nur Attribute des Symbols oder über INCLUDING, CONSTITUENT(S), CHAIN erreichbare Attribute verwendet werden: Entwirf eine **untere (SYNT) SYMBOL-Berechnung**.

A4: Wie A3. Aber es gibt **wenige Ausnahmen**, wo entweder untere oder obere (nicht beide!) RULE-Kontexte die Eigenschaft abweichend bestimmen: Entwirf eine obere (INH) oder untere (SYNT) **SYMBOL-Berechnung** und **überschreibe sie in solchen RULE-Kontexten**.

A5: Wie A4; aber für **rekursive Symbole**: Die Fundierung der Rekursion ist die Ausnahme aus A4, z. B. Schachtelungstiefe von Blöcken.

Wenn keiner der Fälle paßt, muß die Modellierung der Eigenschaft überdacht werden. Sie könnte zu komplex sein und eine weitere Zerlegung erfordern.

B: Eigenschaften von Programmobjekten

Vorbereitung: Programmobjekte werden meist durch Bezeichner im Programm benannt. In diesem Fall verwendet man einen Modul zur **Namensanalyse**, um Programmobjekte (repräsentiert durch `DefTableKeys`) and die Auftreten von Bezeichnern zu binden. Damit hat jeder Symbolknoten der Bezeichner ein `Key`-Attribut, dessen Wert das Programmobjekt identifiziert.

Berechnungen, die die Eigenschaft des Programmobjektes **setzen oder lesen**, sollten vorzugsweise im unteren oder oberen **Kontext des Symbols** für den Bezeichner plaziert werden, statt die `Key`-Werte im Baum weiterzureichen. So wird die Zugehörigkeit der Berechnung zum Programmobjekt deutlicher. Wenn möglich sollten **SYMBOL-Berechnungen** verwendet werden (siehe A).

Im allgemeinen benötigt man folgende Entwurfsschritte:

1. Bestimme **Namen und Typ der Eigenschaft** und spezifiziere sie für PDL.
2. Bestimme die **Kontexte und die Operation für das Setzen** der Eigenschaft.
3. Bestimme die **Kontexte für das Lesen** der Eigenschaft.
4. Bestimme die **Abhängigkeiten zwischen (2) und (3)**. In einfachen Fällen ist dies: "alle Setzen vor jedem Lesen".
5. Spezifiziere (2), (3) und das Abhängigkeitsmuster (4).