Praktikumshinweise Compilerbau SS 2018

Typen, Symboltabellen und Semantische Analyse für SPL

Typen in SPL

SPL hat zwei primitive Typen: *Boolean* und *Integer*. Es gibt nur einen Typkonstruktor, mit dem komplexe Typen gebaut werden können: *Array*. Der zu einem Array-Typ gehörende Komponententyp kann selbst wieder ein Array-Typ sein, so dass beliebig verschachtelte Typhierarchien möglich sind. Als interne Repräsentation dieser Hierarchien werden Typ-Bäume verwendet.

In Java ist ein Typ-Baum ein Objekt der abstrakten Klasse *Type*. Diese Klasse hat zwei Subklassen: *PrimitiveType* und *ArrayType*. Ein primitiver Typ wird intern als Objekt der Klasse *PrimitiveType* dargestellt und ein Array-Typ als Objekt der Klasse *ArrayType*:

```
public class ArrayType extends Type {
  public int size;         // Anzahl der Komponenten
  public Type baseType;    // Typ der Komponenten
  public ArrayType(int s, Type t) {
    size = s;
    baseType = t;
  }

  public void show() { ... }
}
```

In der Klasse *PrimitiveType* gibt es nur zwei Objekte, je eines für *Boolean* und *Integer*.

```
static final Type intType =  new PrimitiveType("int", ...);
static final Type boolType = new PrimitiveType("boolean", ...);
```

Gleichheit von Typen

Durch eine Typdefinition

```
type <IDENT> = <TYPAUSDRUCK> ;
```

wird kein neuer Typ erzeugt. Der neue Typbezeichner hat denselben Typ, wie der Typausdruck auf der rechten Seite. Beispiel:

```
type zahl = int ;
```

Der Typbezeichner „zahl“ kann nun genau wie „int“ für den vordefinierten Integer-Typ verwendet werden. Eine Variable vom Typ „zahl“ hat denselben Typ, wie eine Variable vom Typ „int“. Ein Typbezeichner kann insofern als ein Verweis auf einen vorhandenen Typ angesehen werden. Für die Prüfung der Typgleichheit müssen solche Verweise bzw. Verweisketten aufgelöst werden.

Jeder Array-Typausdruck definiert einen neuen Array-Typ. Auch wenn zwei Array-Typausdrücke genauso aussehen, sind die Typen, die sie repräsentieren, verschieden. Beispiel:

```
type vector  = array [10] of int;
type vector2 = vector;
```

```
var feld1 : array [10] of int;
var feld2 : array [10] of int;
var feld3 : vector;
var feld4 : vector;
var feld5 : vector2;
```

Die beiden Typausdrücke für *feld1* und *feld2* gleichen sich zwar, die Typen werden aber als verschieden betrachtet. Die Variablen „feld3“, „feld4“ und „feld5“ haben dagegen alle denselben Typ, da die Typdefinitionen von „vector“ und „vector2“ keine neuen Typen einführen und somit alle drei Variablen auf denselben Array-Typ zurückführbar sind.

In der interne Typrepräsentation wird daher für jeden Array-Typausdruck ein neuer Typ-Baum aufgebaut. Im Beispiel oben gibt es zu den drei im Quelltext vorkommenden Array-Typausdrücken also drei verschiedene Typ-Bäume, die allerdings alle das gleiche Aussehen haben:

```
new ArrayType(10, intType);
```

Für die semantische Analyse gilt also im Hinblick auf die Typen:

- Zwei Typen sind genau dann gleich, wenn sie durch dasselbe *Type*-Objekt repräsentiert werden.
- Zu jedem *ArrayTy*-Objekt im AST muss ein neues *ArrayType*-Objekt als Typrepräsentation gebaut werden. Da dazu die Typrepräsentation des Basistyps (Typ der Array-Komponenten) benötigt wird, ist der Algorithmus rekursiv.
- Zu jedem *NameTy*-Objekt im AST muss das zugehörige *Type*-Objekt in der Symboltabelle stehen. Es kann sich dabei um ein *PrimitiveType*-Objekt handeln (bei „int“, „zahl“) oder um ein *ArrayType*-Objekt (bei „vector“ und „vector2“).

Prozedur-Signaturen und Parametertypen

Zu jedem Parameter einer Prozedur gibt es zwei für die semantische Analyse relevante Attribute, der Typ und das Parameterübergabeverfahren:

- Ein Referenzparameter (Schlüsselwort „ref“ im Quelltext) repräsentiert einen Speicherplatz der aufrufenden Prozedur, dessen Adresse an die aufgerufene Prozedur übergeben wird.
- Ein Wertparameter (kein „ref“ im Quelltext) ist eine lokale Variable der aufgerufenen Prozedur, die beim Aufruf mit einem vom Aufrufer gelieferten Wert initialisiert wird.

Im Paket *types* wird ein Parametertyp daher als ein Objekt der Klasse *ParamType* definiert, wobei das Attribut *isRef* angibt, ob es sich um einen Referenzparameter handelt:

```
public class ParamType {
  public Type type;
  public boolean isRef;
  ...
  public ParamType(Type t, boolean r) {
    type = t;
    isRef = r;
  }
```

Die semantische Analyse muss bei Prozeduraufrufen prüfen, ob die an der Aufrufstelle (im AST *ProcCall*-Objekt) übergebenen Argumente passen:

- Ist die Anzahl der Argumente korrekt?
- Für jedes Argument:
  - Stimmt der Typ?
  - Falls der Typ ein Arraytyp ist: Ist der Parameter ein Referenzparameter?
  - Falls der Parameter ein Referenzparameter ist: Ist das Argument eine Variable?

Für diese Prüfungen wird die Schnittstelleninformation oder „Signatur“ der aufgerufenen Prozedur benötigt, die die *ParamType*-Informationen aller Parameter enthält:

```
public class ParamTypeList extends ArrayList<ParamType> {
        ...
        public void add(Type type, boolean isRef) {
                this.add(new ParamType(type, isRef));
        }
```

Symboltabellen in SPL

In SPL gibt es für die Bezeichner eine flache Hierarchie von Gültigkeitsbereichen:

- Global definierte Bezeichner: Typen und Prozeduren
- Lokal innerhalb einer Prozedur definierte Bezeichner: Parameter und lokale Variablen

Eine Symboltabelle repräsentiert einen Gültigkeitsbereich und enthält Einträge für alle Bezeichner dieses Bereichs. Der SPL-Compiler benötigt also

- Eine globale Symboltabelle und
- für jede Prozedur eine lokale Symboltabelle.

In der globalen Tabelle stehen neben den im SPL-Programm definierten Typ- und Prozedurbezeichnern auch die vordefinierten Bezeicher („int“, „readc“, „printi“ usw.).

Im Paket *tables* werden Symboltabellen so definiert, dass eine mehrstufige Hierarchie aus ineinander verschachtelten Gültigkeitsbereichen unterstützt wird. Eine Tabelle ist ein *Table*-Objekt und kann mit einer übergeordneten Tabelle durch einen *upperLevel*-Verweis verknüpft werden. Für die Prozedurspezifischen lokalen Tabellen enthält also *upperLevel* einen Verweis auf die globale Tabelle.

```
package table;
  ...
public class Table {

    private Table upperLevel;
    private Map<Sym, Entry> dict;

    public Table() {
            upperLevel = null;
            dict = new HashMap<Sym, Entry>();
    }

    public Table(Table u) {
            upperLevel = u;
            dict = new HashMap<Sym, Entry>();
```

```
}

public Entry enter(...) { ... }
public Entry lookup(Sym sym) { ... }
```

Die neben den Konstruktoren wichtigen Methoden sind *enter* zum Eintragen einer neuen Definition (*Entry*-Objekt) und *lookup* zum Suchen eines Eintrags zu einem Symbol.

Symboltabelleneinträge

Ein Symboltabelleneintrag assoziiert einen Bezeicher mit einer Menge von Attributen. Dazu gehören Informationen, die unmittelbar aus der Definitionsstelle im Quelltext entnommen werden können, wie z.B. die Art des Bezeichners (Typ, Prozedur, . . .) oder das Parameter-Attribut *isRef*. Andere Attribute werden im Rahmen der semantischen Analyse bestimmt, z.B. zu einem Variablenbezeichner der Typ. Weitere Compiler-Komponenten nutzen die Symboltabelle ebenfalls, um dort Bezeichner-spezifische Informationen abzuspeichern, z.B. Adressen von Variablen.

Natürlich ist auch die Art des Eintrags selbst ein Attribut des Bezeichners. Dies wird in Java und C allerdings unterschiedlich gehandhabt:

- In C ist die Eintragsart ein normales Attribut *kind*.
- In Java ist ein Eintrag ein Objekt des abstrakten Klasse *Entry*. Die Art des Eintrags ergibt sich aus der *Entry*-Subklasse *VarEntry*, *TypeEntry* oder *ProcEntry*.

Die Attribute eines Bezeichners hängen von der Art des Bezeichers ab:

1. Zu einem Typbezeichner gehört ein Typ.
2. Zu einem Variablenbezeichner gehört ein Typ. (Später wird noch eine Speicheradresse dazu kommen.)
3. Zu einem Parameterbezeichner gehört ein Typ und ein *isRef*-Attribut. (Später wird noch eine Speicheradresse *offset* dazu kommen.)
4. Zu einem Prozedurbezeichner gehören eine Parametertypliste und eine lokale Symboltabelle. Für die Code-Erzeugung werden dann noch weitere Attribute benötigt, z.B. die Anzahl der Bytes, die für die Speicherung der lokalen Variablen nötig sind (int *localvarArea*).

Warum gibt es für die Parameter keine Klasse *ParEntry*? Da Symboltabelleneinträge für Variablen und Parameter weitgehend gleich genutzt werden, gibt es für beide Bezeichnerarten nur eine Art von Symboltabelleneinträgen, repräsentiert durch ein *VarEntry*-Objekt. Eine lokale Variable wird dort genau so eingetragen, wie ein Wertparameter (*isRef* = false).

Variableneinträge:

```
package table;
import types.*;

public class VarEntry extends Entry {
  public Type type;
  public boolean isRef;
  public int offset;               /* filled in by variable allocator */

  public VarEntry(Type t, boolean r) {
    type = t;
```

```
    isRef = r;
  }
  ...
```

Typeinträge:

```
package table;
import types.*;

public class TypeEntry extends Entry {

public Type type;

public TypeEntry(Type t) {
  type = t;
}
  ...
```

Prozedureinträge:

```
package table;
import types.*;

public class ProcEntry extends Entry {

  public ParamTypeList paramTypes;
  public Table localTable;
  public int argumentAreaSize;      /* filled in by variable allocator */
  public int localvarAreaSize;      /* filled in by variable allocator */
  public int outgoingAreaSize;      /* filled in by variable allocator */

  public ProcEntry(ParamTypeList p, Table t) { ... }
  ...
}
```

Semantische Analyse

Die Semantische Analyse muss alle Korrektheitsbedingungen prüfen, die nicht in der SPL-Grammatik stecken. Einige Beispiele:

- Sind beide Operanden einer Addition vom Typ *int*?
- Ist jeder Bezeichner auch passend zu seiner Verwendung definiert?
- Gibt es eine Definition der Prozedur „main“?
- Ist der Ausdruck auf der rechten Seite einer Wertzuweisung vom Typ *int*?
- Repräsentiert die linke Seite einer Wertzuweisung einen Speicherplatz?

Ein strukturiertes Vorgehen orientiert sich an der Syntax: Für jeden Knotentyp des AST werden die notwendigen Prüfungen separat analysiert und implementiert.

Bezeichner und das „Declare before Use“-Prinzip

Wenn im AST Bezeichner auftreten, handelt es sich entweder

- um die Definitionsstelle des Bezeichners, z.B.

  ```
  var i: int;
  ```

  (im AST ein *VarDec*-Objekt) oder
- um eine Verwendungsstelle, z.B. die linke Seite folgender Wertzuweisung

  ```
  i := 1;
  ```

  (im AST ein *SimpleVar*-Objekt)

Semantische Analyse einer Definitionsstelle heißt

- Entry-Objekt bauen und
- in die richtige Symboltabelle mit der *enter*-Methode eintragen.

Semantische Analyse einer Verwendungsstelle erfordert:

- in der Symboltabelle nach dem Bezeichner suchen
- bei Fehlschlag eine passende Fehlermeldung liefern
- bei Erfolg prüfen, ob die Art des Bezeichners zur Verwendung passt
- mit den Attributen aus dem Eintrag sonstige Korrektheitsbedingungen prüfen

Das „Declare before Use“-Prinzip fordert, dass jeder Bezeichner zuerst (weiter oben im Quelltext) deklariert werden muss, bevor er benutzt werden darf. Diese Spracheigenschaft ermöglicht eine semantische Analyse in einem Durchgang, weil bei der Prüfung jeder Verwendungsstelle der betreffende Bezeichner in der Symboltabelle stehen muss.

In SPL gilt diese Prinzip allerdings **nicht** für Prozedurbezeichner: Der Aufruf einer Prozedur ist schon vor der Prozedurdefinition möglich. Dies erlaubt gegenseitig rekursive Aufrufe.

Deswegen wird die semantische Analyse in zwei Durchgänge aufgeteilt:

1. Aufbau der Symboltabellen ohne Prüfung der Anweisungen in den Prozedur-Rümpfen
   - Erzeugen der globalen Symboltabelle
   - Eintragen der vordefinierten Bezeichner
   - Durchlauf durch den AST und bearbeiten aller Definitionen
2. Prüfung der Anweisungen in den Prozedur-Rümpfen

Nutzung des Visitor-Pattern

Die semantische Analyse wird in der Klasse *semant.SemanticChecker* implementiert. Jeder der beiden Durchgänge der Analyse wird in Form einer Visitor-Klasse implementiert, der erste Durchgang als *class TableBuilder*, der zweite als *class ProcedureBodyChecker*. Das Grundgerüst ist im Paket *semant* schon vorgegeben. Für die Initialisierung der globalen Tabelle mit den vordefinierten Bezeichnern steht die Klasse *TableInitializer* zur Verfügung.

Beide Durchgänge sollen das Entwurfsmuster „Visitor“ verwenden. Somit muss für jeden AST-Knotentyp eine passende visit-Methode bereitgestellt werden. Dabei stellen sich zwei Fragen:

1. Wie kann man bei rekursiven Aufrufen Parameter übergeben und Resultate erhalten, wenn die Signaturen von *visit* und *accept* dies nicht erlauben?

   Vorschlag: Definieren Sie für die Schnittstelle zwischen zwei Rekursionsebenen passende Attribute im Visitor-Objekt. Statt einer Parameterübergabe erfolgt eine Wertzuweisung an die entsprechenden Attribute vor dem rekursiven *visit*-Aufruf. Nach der Rückkehr aus dem rekursiven Aufruf findet die aufrufende Instanz eventuell vorhandene Resultatswerte ebenfalls in Instanz-Variablen des Visitor-Objekts. Im Beispiel unten wird für den von *visit* für *ArrayTy* bestimmten Typ dieses Prinzip genutzt: Die Instanz-Variable *resultType* enthält den Typ.

2. Welche Visitor-Objekte werden benötigt?

   Vorschlag: Entweder versuchen Sie, mit einem einzigen Visitor-Objekt auszukommen, oder Sie erzeugen für jede Rekursionsebene ein neues Visitor-Objekt. Im Beispiel unten wird nur ein Visitor-Objekt für alle AST-Knoten verwendet.

Beispiele:

`TableBuilder.java`:

```
    .
    .
    .
public void visit(DecList list) {
        for (Absyn elem: list)
                elem.accept(this);
}

public void visit(VarDec node) {
        node.ty.accept(this);
        enterDeclaration(node.name, node, new VarEntry(resultType, false));
}

public void visit(ArrayTy node) {
        node.baseTy.accept(this);
        resultType = new ArrayType(node.size, resultType);
}
    .
    .
    .
```

DoNothingVisitor

Manche Algorithmen, die den AST durchlaufen, bearbeiten nur einen Teil der Knotentypen. So ignoriert beispielsweise der *TableBuilder* die Anweisungen und Ausdrücke aller Art. Man kann sich in diesem Fall die Programmierung leerer *visit*-Methoden ersparen, indem man eine Visitor-Klasse mit leeren Methoden für alle Knotentypen implementiert und diese dann bei Bedarf überschreibt. Die Klasse *DoNothingVisitor* ist dafür vorgegeben.

Einen Nachteil hat diese Vorgehensweise: Wenn man für einen Knotentyp die Implementierung vergisst, wird man vom Java-Compiler keinen Hinweis bekommen.