Technische Universität München
Fakultät für Informatik
R. Höllerer

SS 2004
26. Mai 2004

Übersetzung von Programmiersprachen

Merkblatt 3

Verwendung von classgen

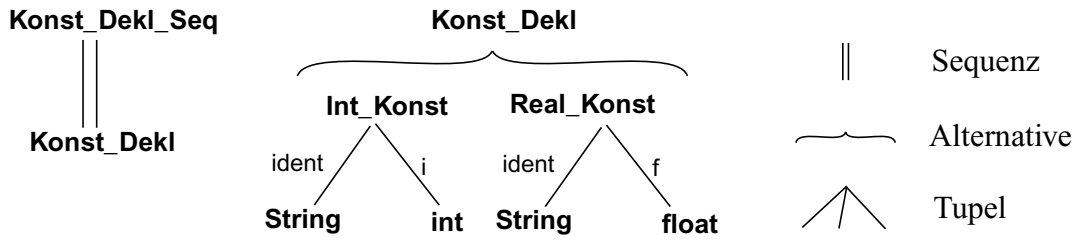
1 Vorbereitungen

Das Werkzeug classgen wurde von Gerwin Klein[1] während seines Studiums an der TUM implementiert. classgen spart dem Anwender viel Schreibarbeit, in dem es Beschreibungen von rekursiven Datentypen automatisch in passende Java-Klassen umwandelt. Als Beschreibungsmittel werden Grammatiken verwendet, so daß classgen besonders gut im Compilerbau, z. B. zur Definition des abstrakten Syntaxbaumes, eingesetzt werden kann. classgen unterstüzt darüber hinaus das sogenannte „visitor pattern“ [GH96] und die Verwendung von Attributen.

- classgen findet sich in der Informatik-Halle unter dem Pfad
  /usr/proj/compiler/compilerbau/bin/classgen
- Aufruf: classgen [-f|-overwrite] [-public] [-visitor] <inputfile>

2 classgen-Eingabe

Betrachten wir folgenden Teil der abstrakten Syntax, wie es im Arbeitsblatt 8 in graphischer Notation definiert wurde:



Diese abstrakte Syntax gibt die Struktur für eine Liste von Konstantendefinitionen an. Eine Konst_Dekl_Seq besteht aus einer Sequenz von Konst_Dekl, die wiederum eine int-Konstante (Int_Konst) oder eine real-Konstante (Real_Konst) sein können.

---

1. Nun Mitarbeiter der Fakultät s. http://www.informatik.tu-muenchen.de/~kleing

Konst_Dekl_Seq wird als *Sequenztyp* bezeichnet, Konst_Dekl als *Variantentyp* und Int_Konst und Real_Konst als *Tupeltypen.* Mit den sogenannten *Selektoren* ident, i und f werden die Tupelelemente referenziert.

Um diesen Datentyp in Java zu implementieren müssen 4 Klassen erstellt werden. Dies kann von classgen automatisch erledigt werden. Die Beschreibung des Datentyps in einer Datei mimaAST.cl könnte wie folgt aussehen:

```
package node;

Konst_Dekl_Seq ::= Konst_Dekl*

Konst_Dekl ::= {Int_Konst}     String:ident   "int": i
    | {Real_Konst} String:ident   "float": f
```

Durch Starten des Werkzeugs mit dem Kommando classgen mimaAST.cl werden automatisch die Klassen Konst_Dekl_Seq, Konst_Dekl, Int_Konst und Real_Konst in entsprechenden Dateien im Verzeichnis node erzeugt.

Erzeugter Code

Für einen *Tupeltyp* werden private Instanzvariablen für die Komponenten sowie zugehörige Zugriffsmethoden (zum Setzen und Holen der Komponente) und ein Konstruktor erzeugt, z.B. für Int_Konst:

```
private String ident;
  private int i;

  public Int_Konst (String ident, int i)

  public String getIdent()
  public void setIdent(String ident)

  public int getI()
  public void setI(int i)
```

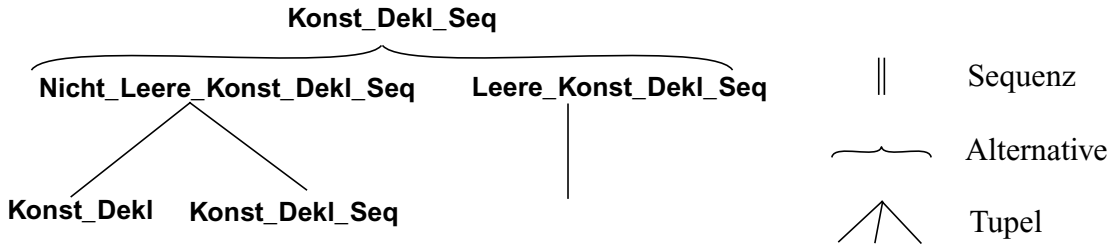
Ist man sicher, daß andere Programmteile direkt auf die Komponenten zugreifen dürfen (wie z.B. innerhalb eines Compilers), kann man beim classgen-Aufruf die Option -public wählen. Dadurch werden die Instanzvariablen public definiert und die zugehörigen get- und set-Methoden fallen weg.

Für einen *Variantentyp* wird eine abstrakte Klasse erzeugt und die Alternativen werden als Unterklassen davon definiert; d.h. im obigen Beispiel wird Konst_Dekl als abstrakte Klasse und die Alternativen Int_Konst und Real_Konst werden als Unterklassen davon definiert.

Für einen *Sequenztyp* wird ein privater Vector items definiert, in der die Elemente der Sequenz gespeichert werden. Dazu werden auch die zugehörigen Zugriffsmethoden erzeugt und es werden zwei Konstruktoren definiert: Konstruktor für eine leere Sequenz und ein Konstruktor für eine Sequenz bestehend aus einem Element. Für obiges Beispiel wären es u.a.:

```
private Vector items;

public Konst_Dekl_Seq()
public Konst_Dekl_Seq(Konst_Dekl anItem)

public Konst_Dekl_Seq append(Konst_Dekl anItem)

public Konst_Dekl elementAt(int index)
public void setElementAt(Konst_Dekl item, int index)

public int size()
```

Will man keine Listen verwenden, könnte man Konst_Dekl_Seq links- oder rechtsrekursiv definieren:



Die classgen-Eingabe wäre

```
Konst_Dekl_Seq ::= {Nicht_Leere_Konst_Dekl_Seq}     Konst_Dekl    Konst_Dekl_Seq
    | {Leere_Konst_Dekl_Seq}   /* leere Sequenz */
```

Für Konst_Dekl_Seq wird nun eine abstrakte Klasse erzeugt und die Alternativen Nicht_Leere_Konst_Dekl_Seq und Leere_Konst_Dekl_Seq werden als Unterklassen davon definiert.

Für die Komponenten von Nicht_Leere_Konst_Dekl_Seq werden private Instanzvariablen sowie zugehörige Zugriffsmethoden und ein Konstruktor erzeugt; bzw. mit Angabe der Option -public werden die Instanzvariablen public definiert und die zugehörigen get- und set-Methoden fallen weg.

```
private Konst_Dekl   konst_Dekl;
private Konst_Dekl_Seq    konst_Dekl_Seq;

public Konst_Dekl getKonst_Dekl()
public void setKonst_Dekl(Konst_Dekl konst_Dekl)

public Konst_Dekl_Seq getKonst_Dekl_Seq()
public void setKonst_Dekl_Seq(Konst_Dekl_Seq konst_Dekl_Seq)

public Nicht_Leere_Konst_Dekl_Seq (Konst_Dekl  konst_Dekl,
            Konst_Dekl_Seq  konst_Dekl_Seq)
```

Für Leere_Konst_Dekl_Seq wird nur der Konstruktor public Leere_Konst_Dekl_Seq () erzeugt.

Gibt man keine Namen für die Alternativen an, werden diese automatisch erzeugt, d.h. aus der Eigabe

```
Konst_Dekl_Seq ::= Konst_Dekl    Konst_Dekl_Seq
    | /* leere Sequenz */
```

wird die abstrakte Klasse Konst_Dekl_Seq und die Unterklassen Konst_Dekl_Seq0 und Konst_Dekl_Seq1 erzeugt.

Neben der Methoden zur Konstruktion und Zugriff auf Komponenten, werden für jede Klasse Methoden zur Ausgabe des Baumes erzeugt, die den Baum schön eingerückt als String aufbauen, mit den Signaturen

```
public String toString()
public String toString(String tab)
```

wobei der Parameter tab die Einrücktiefe angibt.

Weitere Hinweise

- Einfache Tupelproduktionen (ohne Alternativen) können auch verwendet werden, z.B.:
  `T ::= A:a   B:b`

- Das Werkzeug versucht die Java-Konvention einzuhalten, daß Klassennamen mit Großbuchstaben anfangen, Methoden und Felder dagegen mit Kleinbuchstaben. Falls notwendig, wird dazu auch die Schreibweise aus der Eingabe korrigiert. In einigen Fällen möchte man jedoch eine bestimmte Schreibweise erzwingen - dies ist durch den Gebrauch von Anführungszeichen möglich (wie in unserem Beispiel bei "int" und ”float“ geschehen).

- Werden keine Selektoren angegeben, erzeugt classgen automatisch aus den Komponentenbezeichnern Selektornamen, die mit Kleinbuchstaben anfangen.

3 Visitoren

Nun wollen wir auch die Semantikbehandlung implementieren, d.h. u.a. Identifikation, Typbehandlung und Codeerzeugung realisieren. Dafür könnten wir die automatisch erzeugten Java-Klassen nachträglich um zusätzliche Funktionalität in Form von Methoden und zusäzlichen Feldern erweitern.

Diese Methodik hat Nachteile, die nicht übersehen werden sollten:

1. Da die von classgen erzeugten Klassen nachträglich von Hand erweitert werden, darf sich der jeweils verwendete Datentyp später nicht mehr ändern, sonst würde der zusätzliche Code bei Neuerzeugung wieder gelöscht.

2. Bei jeder Änderung der Klassen – z.B. Hinzufügen von Methoden – müssen die Klassen selbst und oft sehr viele weitere davon abhängige Klassen neu übersetzt werden (dieser Nachteil wird bei großen Projekten gravierend).

Diese Nachteile können umgangen werden, indem bei Datentypen (abstrakte) Syntax und Semantik getrennt werden, d.h. in unterschiedlichen Klassen implementiert werden. Am Beispiel Konst_Dekl_Seq bedeutet dies, daß die von classgen erzeugten Klassen unverändert bleiben und die für Identifikation, Typbehandlung und Codeerzeugung benötigten Berechnungen in einer (oder mehreren) eigenen zusätzlichen Klassen implementiert werden.

Erster Versuch

Wir werden zuerst einen ersten naiven Ansatz untersuchen, der die erwähnten Nachteile behebt (dafür aber leider neue Nachteile bringt). Dazu betrachten wir folgenden einfachen Datentyp List

```
List ::= {Cons} ”int”: head    List:tail
 | {Empty}
```

und nehmen an, daß von classgen entsprechende Klassen bereits erzeugt wurden:

```
public abstract class List {
}

public class Empty extends List {
public Empty() {
}
}

public class Cons extends List {

  public int head;
  public List tail;

  public Cons (int head, List tail) {
    this.head = head;
    this.tail = tail;
  }
}
```

Nun soll als einfache semantische Analyse-Aufgabe realisiert werden, daß alle Elemente einer gegebenen Liste aufsummiert werden. Bei der „von Hand“ Erweiterung hätten wir dazu die obigen Klassen um eine Methode public int sum() ergänzt. Stattdessen erstellen wir jetzt versuchsweise folgende neue Klasse:

```
public class ListSum {
   public int sum = 0;
   public void process( List l ) {
      if ( l instanceof Empty )
        return;
      else if ( l instanceof Cons ) {
            sum += ((Cons) l).head;
            process(  ((Cons) l ).tail  );
    }
  }
}
```

Da wir hier die automatisch erzeugten Klassen unverändert lassen, ist damit der Nachteil 1 behoben. Auch Nachteil 2 ist auf diese Weise behoben. Falls z.B. zusätzlich die Funk-

tionalität zum Ausgeben einer Liste am Bildschirm benötigt wird, könnte eine vom bisher existierenden Code unabhängige Klasse ListPrinter implementiert werden.
Allerdings hat diese Lösung den Nebeneffekt, daß unschöne instanceof's und Typecasts notwendig sind. Auch dies kann mit kleinen Tricks behoben werden, wie im folgenden gezeigt wird. Die resultierende Implementierungstechnik wird Entwurfsmuster „Visitor" genannt (Visitor Design Pattern [GH96]).

Entwurfsmuster „Visitor"

Nehmen wir an, daß die von classgen erzeugten Klassen eine Methode

```
public void accept( Visitor v ) { v.visit( this ); }
```

unterstützen, wobei Visitor als folgendes Interface definiert ist:

```
public interface Visitor {
  public void visit( Empty e );
  public void visit( Cons c );
}
```

Dann kann die Klasse ListSum wie folgt neu implementiert werden:

```
public class ListSum2 implements Visitor {
  public int sum = 0;
  public void visit( Empty e ) {}
  public void visit( Cons c ) {
    sum += c.head;
    c.tail.accept( this );   // (*)[2]
  }
}
```

und die Summe kann folgendermaßen berechnet werden:

```
List l = new Cons(11, new Cons(22, new Cons(33, new Empty())));
     ListSum2 ls = new ListSum2();
     l.accept(ls);
     System.out.println("Summe ist: " + ls.sum);
```

In dieser Version entfallen also auch noch alle Typecasts, wodurch eine recht saubere Lösung entsteht.

Durch diesen Lösungsweg können die verschiedenen Teilaufgaben der Semantikbehandlung (z.B. Identifikation, Typbehandlung und Codeerzeugung) nicht nur konzeptionell, sondern auch tatsächlich, in verschiedenen Klassen realisiert werden. Man könnte sich

---

2. **Frage:** Warum kann die in der Klasse ListSum2 mit (*) markierte Zeile nicht als
visit(c.getTail())
geschrieben werden?
Antwort: Die dynamische Bindung bezieht sich nur auf den *Empfänger* der Methode, nicht auf *Parameter.* Im Aufruf visit(c.getTail()) wird statisch gebunden, d.h. es wird, da Ergebnis vom getTail() vom Typ List definiert ist, visit(List list) aufgerufen. Wir wollen aber entweder visit(Empty e) oder visit(Cons c) aufrufen je nachdem, von welchem Typ c.getTail() *zur Laufzeit* ist.

vorstellen, daß z.B. die Identifikationsklasse einen top down-Durchlauf des abstrakten Syntaxbaumes realisiert und bei einer Deklaration die Deklarationsinformation in die Symboltabelle einträgt und bei der Anwendunsstelle die Information aus der Symboltabelle holt und an der Anwendungstelle (d.h. am Anwendungsknoten) aufhebt. Die Codeerzeugungsklasse würde diese Information dann bei der Erzeugung des Codes verwenden.

Vergleichen wir die Vorgehensweise mit Spezifizieren mithilfe von attributierten Grammatiken, so muß man hier neben den Berechnungen, die an den Knoten zu auszuführen sind, auch die Strategie angeben, in welcher Reihenfolge die Knoten zu besuchen sind. Bei attributierten Grammatiken muß man nur die Berechnungen spezifizieren und es wird entweder eine Standardstrategie (z.B. top down bei LL-Grammatiken) gewählt, oder Compilergenerator berechnet aus Attributabhängigkeiten die Strategie und erzeugt ein entsprechendes Attributierungsmodul (z.B. Kennedy/Warren-Attributauswertung).

classgen unterstützt den Benutzer auch bei der Realisierung der Baumdurchläufe. Für jede Klasse werden neben der accept-Methode auch Baumduchlauf-Methoden erzeugt:

```
public void accept(Visitor visitor)
public void childrenAccept(Visitor visitor)
public void traverseTopDown(Visitor visitor)
public void traverseBottomUp(Visitor visitor) }
```

Daneben wird die Schnittstelle für die Visitoren vordefiniert, d.h. für unser List-Beispiel:

```
public interface Visitor {

  public void visit(List list);
  public void visit(Cons cons);
  public void visit(Empty empty);

}
```

und zusätzlich wird eine abstrakte Klasse VisitorAdaptor erzeugt, als deren Unterklasse man die eigene Visitorklasse definieren kann.

```
public abstract class VisitorAdaptor implements Visitor {

  public void visit(List list) { visit(); }
  public void visit(Cons cons) { visit(); }
  public void visit(Empty empty) { visit(); }

  public void visit() { }

}
```

Daß heißt, daß man bei Standardstrategie – z.B. bei top down-Durchlauf für die Identifikation – in einer eigenen Identifikationsvisitor nur visit-Methoden für die Knoten definieren muß, bei denen etwas zu tun ist. Stößt man dann in der Main-Methode den top down-Durchlauf für den Wurzelknoten an, werden alle Knoten in top down-Reihenfolge besucht und die angegebenen Berechnungen durchgeführt. Da in unserem List-Beispiel die Strategie keine Rolle spielt, könnte man das gleiche Ergebnis erzielen, wenn man die in der Klasse ListSum2 mit (*) markierte Zeile streicht und einen top down- oder bottom up-Durchlauf anstößt.

Unsere Summenberechnung wäre nun:

```
    List l = new Cons(11, new Cons(22, new Cons(33, new Empty())));
    ListSum2 ls = new ListSum2();
 l.traverseTopDown(ls);
// l.traverseBottomUp(ls);
    System.out.println("Summe ist: "+ ls.sum);
```

4 Attribute

Informationen wie z.B. die Symboltabelle kann man global in dem Identifikationsmodul halten. Manche Informationen wie z.B. die Deklarationinformation an der Anwendunsstelle eines Identifiers wird vom Identifikationsmodul bereitgestellt und vom Codeerzeugungsmodul benötigt. Solche Informationen werden am besten am jeweiligen Knoten aufgehoben. Dafür bietet classgen Attribute an, d.h. man kann wie in attributierten Grammatiken zu Nichterminalen der Grammatik Attribute mit Namen und Typ definieren. Diese werden als zusätzliche Instanzvariablen der Knotenklasse realisiert.

In der classgen-Eingabe könnte man z.B. für das Nichterminal Var, das die Anwendung eines vorher deklarierten Identifiers in einem Ausdruck spezifiziert, ein Attribut mit den Namen deklInfo und vom Typ DeklInfo folgendermaßen definieren:

```
attr  DeklInfo deklInfo with Var;

Expr    ::= {BinExpr}  Expr:lhs  „int“:OP Expr:rhs
        | {Var}      String:ident
```

Der Identifikationsmodul könnte folgende visit-Methode für den Var-Knoten enthalten, in der die Deklarationsinformation aus der Symboltabelle symtab geholt und in das deklInfo-Attribut gespeichert wird (vorausgesetzt es wurde beim classgen-Aufruf die Option -public gewählt).

```
public void visit( Var var) {
var.deklInfo = (DeklInfo)symtab.get( var.ident );
}
```

Der Codeerzeugungsmodul könnte dann die im Attribut deklInfo gespeicherte Deklarationsinformation bei der Codeerzeugung verwenden (nämlich die Relativadresse reladr), etwa:

```
public void visit(Var var) {
   System.out.println("MOVE W " + bdisp(var.deklInfo.reladr, basreg) + push);
 }
```

5 Dokumentation

Unter /usr/proj/compiler/compilerbau/docs/classgen ist ausführliche Dokumentation über classgen zu finden. Über Entwurfsmuster kann man folgendes Buch empfehlen:

[GH96] E.Gamma, R.Helm, R.Johnson, J.Vlissides: Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley, 1996.