Prof. Dr. M. Jäger FB MNI

Praktikumshinweise Compilerbau SS 2018

Abstrakter Syntaxbaum und Visitor-Entwurfsmuster

Abstrakter Syntaxbaum: Rolle im Compiler und Aufbau

Der Abstrakte Syntaxbaum (AST – *Abstract Syntax Tree*) ist die Schnittstelle zwischen dem Frontend und dem Backend des SPL-Compilers. Ein AST repräsentiert ein SPL-Programm in einer für die Weiterverarbeitung durch das Backend geeigneten Form. Die Wurzel des AST verkörpert das SPL-Programm als Ganzes. Die inneren Knoten stellen komplexe Sprachkonstrukte wie Prozedurdefinitionen, While-Schleifen oder Wertzuweisungen dar. Dafür werden unterschiedliche Knotentypen definiert. Abhängig vom Knotentyp kann ein AST-Knoten einfache Attribute und komplexe Unterstrukturen haben, die dann als Nachfolgeknoten im AST erscheinen. Die vom Scanner gelieferten Tokens werden als Blattknoten in den AST integriert.

Im Gegensatz zum Ableitungsbaum („konkreter“ Syntaxbaum) sind syntaktische Elemente aus dem Quelltext, etwa Schlüsselwörter, Klammern und Semikolons, im AST nicht mehr vorhanden, wenn sie nur für die Syntaxerkennung selbst, nicht aber für die Weiterverarbeitung gebraucht werden.

Während der Syntaxanalyse eines SPL-Quelltextes muss der Parser den AST aufbauen. Die semantische Analyse benutzt den AST als Ausgangsbasis u.a. für die Konstruktion von Symboltabellen. Der Codegenerator benutzt sowohl den AST als auch die Symboltabellen für die Erzeugung von Assemblerbefehlen. Dabei wird jeweils der AST von der Wurzel ausgehend traversiert, wobei die Verarbeitung jedes Baumknotens von dessen Knotentyp abhängig ist.

Ein Beispiel

Der SPL-Compiler kann zu einem SPL-Programm eine textuelle Darstellung des AST erzeugen.

Beispiel: Der AST zum nachfolgenden SPL-Programm `test13.spl` kann mit dem Kommando

```
spl --absyn test13.spl test13.asm
```

ausgegeben werden.

`test13.spl:`

```
proc aux(ref i: int) {
  var j: int;
  j := i + 1;
}
proc main() {
  var i: int;
  aux(i);
}
```

Die textuelle Ausgabe des AST:

```
DecList(
  ProcDec(
    aux,
    DecList(
      ParDec(
        i,
        NameTy(int),
        true)),
    DecList(
      VarDec(
        j,
        NameTy(int))),
    StmList(
      AssignStm(
        SimpleVar(j),
        OpExp(
          ADD,
          VarExp(
            SimpleVar(i)),
          IntExp(1))))),
  ProcDec(
    main,
    DecList(),
    DecList(
      VarDec(
        i,
        NameTy(int))),
    StmList(
      CallStm(
        aux,
        ExpList(
          VarExp(
            SimpleVar(i)))))))
```

Erläuterung einiger Knotentypen:

- `DecList` repräsentiert eine Liste von Deklarationen. Für die Listenelemente gibt es verschiedene Varianten, z.B. Typdeklarationen (`TypeDec`), Prozedurdeklarationen (`ProcDec`) oder Variablendeklarationen (`VarDec`).
- `ProcDec` repräsentiert eine Prozedurdeklaration. Der Baumknoten hat ein Attribut vom Typ *Sym*, das den Prodedurbezeichner „aux“ enthält, und drei komplexe Unterstrukturen, die durch Kindknoten repräsentiert werden:
  - Die Liste der Parameterdeklarationen: Ein `DecList`-Knoten
  - Die Liste der Deklarationen lokaler Variablen: Ebenfalls ein `DecList`-Knoten
  - Der Prozedur-Rumpf: Ein `StmList`-Knoten, der die Sequenz von Anweisungen („Statement List“) repräsentiert, aus denen der Rumpf besteht.

  Was es mit der Klasse *Sym* auf sich hat, die nicht nur für Prodedurbezeichner, sondern für Bezeichner aller Art wird, ist weiter unten erläutert.
- `SimpleVar` und `ArrayVar` sind Referenzen, also Speicherplatzrepräsentanten, die an bestimmten Stellen implizit dereferenziert werden. Im AST wird eine Dereferenzierung durch einen `VarExp`-Knoten dargestellt.

Beispiel: `j := i;`

Auf der linken Seite der Wertzuweisung steht der Variablenbezeichner *j* für den Speicherplatz. Auf der rechten Seite steht ebenfalls ein Variablenbezeichner (*i*), der aber nicht den Speicherplatz, sondern den dort abgespeicherten Wert repräsentiert. Die Dereferenzierung, die zum Speicherplatz den Wert liefert, erscheint im Baum als *VarExp*-Operation.

```
AST:        AssignStm(
             SimpleVar(j),
             VarExp(
               SimpleVar(i)))
```

Man beachte dabei, dass auf der Maschinenebene eine Referenz durch eine Speicheradresse repräsentiert wird. Die Dereferenzierung, also die Ermittlung des Werts erfolgt durch eine Maschinenoperation, nämlich einen lesenden Hauptspeicherzugriff, z.B. *ldw* („Load Word"-Instruktion)

Ein `ArrayVar`-Knoten steht für eine Feldkomponenten-Selektion, z.B. `meinfeld[i+1][10]` , ebenfalls eine Referenz, die z.B. auf der rechten Seite von Wertzuweisungen implizit dereferenziert wird.

Spezifikation des AST

Der AST ist im Praktikum vorgegeben, weil das AST-Design den Erfordernissen des Compiler-Backends Rechnung tragen muss und daher ohne die Erfahrung einer Backend-Implementierung nicht vernünftig definiert werden kann.

Das vorgegebene Compiler-Skelett enthält dazu das Paket *absyn*.

Die Bestimmung geeigneter Knotentypen für der AST hängt von den Anforderungen des Backends ab. Betrachtet man beispielsweise binäre Operationen (`+, -, *, /, =, <, >, ...`), so könnte man

a) für jede Operation einen eigenen Knotentyp definieren. Man bekommt dann viele Knotentypen, alle mit den gleichen Attributen: linker Operand und rechter Operand, beide vom Typ AST, oder

b) für alle Operationen denselben Knotentyp *BinärOperation* verwenden. Zu den beiden Operanden kommt dann ein weiteres einfaches Attribut *Operator* hinzu.

Für die Entwurfsentscheidung ist ausschlaggebend, ob es bei der Verarbeitung der Operationen mehr Gemeinsamkeiten oder mehr Unterschiede gibt. Sie werden am Ende des Praktikums feststellen, dass die Gemeinsamkeiten bei weitem überwiegen und dass daher die Entscheidung, mit nur einem Knotentyp *OpExp* zu arbeiten, viel Redundanz vermeidet:

```
package absyn;
public class OpExp extends Exp {
  public final static int EQU = 0;  // Codierungen der Operatoren
  public final static int NEQ = 1;
  ...
  public int op;    // Operator
  public Exp left;  // linker Operand
  public Exp right; // rechter Operand
  ...
}
```

Übersicht über die AST-Klassen

Im Paket *absyn* sind eine ganze Reihe von Knotentypen für den AST in Form einer Klassenhierarchie definiert. Die Konstruktoren sind für den SPL-Parser wichtig, der den AST aufbaut.

Klassenbezeichner	**Oberklasse**	**abstrakt**	**Verwendung**
Absyn	–	ja	Abstrakter Syntaxbaum
ListNode	*Absyn*	ja	definiert Iterator für alle Listen
DecList	*ListNode*	nein	Liste von Deklarationen
ExpList	*ListNode*	nein	Liste von Ausdrücken
StmList	*ListNode*	nein	Liste von Anweisungen
Ty	*Absyn*	ja	Typausdruck
ArrayTy	*Ty*	nein	Array-Typausdruck
NameTy	*Ty*	nein	Typbezeichner
Dec	*Absyn*	ja	Deklaration
TypeDec	*Dec*	nein	Typdeklaration
VarDec	*Dec*	nein	Variablendeklaration
ParDec	*Dec*	nein	Parameterdeklaration
ProcDec	*Dec*	nein	Prozedurdeklaration
Stm	*Absyn*	ja	Anweisung (Statement)
EmptyStm	*Stm*	nein	leere Anweisung
AssignStm	*Stm*	nein	Wertzuweisung
CallStm	*Stm*	nein	Prozeduraufruf
CompStm	*Stm*	nein	Verbundanweisung
WhileStm	*Stm*	nein	While-Anweisung
IfStm	*Stm*	nein	If-Anweisung
Exp	*Absyn*	ja	Ausdruck (Expression)
IntExp	*Exp*	nein	Ganzzahl
OpExp	*Exp*	nein	binäre Operation
VarExp	*Exp*	nein	Dereferenzierung einer Referenz
Var	*Absyn*	ja	Referenz
SimpleVar	*Var*	nein	Variablenbezeichner
ArrayVar	*Var*	nein	Array-Komponentenselektion

Bezeichner, die Klasse *Sym* und das „String-Interning“

Bei Betrachtung der Java-Klassen zum AST fällt auf, dass Bezeichner aus dem Quelltext im Baum nicht als *String*-Objekte auftauchen, sondern immer logisch eingebettet werden in Container-Objekte der Klasse *Sym* aus dem Paket *sym*. Beispielsweise ist der Typbezeichner, der in einer Typdeklaration neu definiert wird im Baum als Attribut „name“ der Klasse „Sym“ definiert:

```
public class TypeDec extends Dec {

  public Sym name;
  public Ty ty;
  ...
}
```

Um aus einem *Sym*-Objekt das eingebettete *String*-Objekt wieder zu extrahieren, wird die *Sym*-Methode `toString()` verwendet.

Dahinter steckt eine optimierte Speicherung von Zeichenketten, die man „String-Interning“ nennt. In einem String-Pool wird in der „intern“-Methode beim Einfügen zunächst geprüft, ob der String schon vorhanden ist. Nur für noch nicht vorhandene String-Literale wird neuer Speicherplatz im Pool reserviert. Nach dem logischen Einfügen in den Pool wird mit einem Verweis auf den Speicherort gearbeitet. Ein Vergleich zweier Zeichenketten auf Gleichheit kann in diesem Fall durch einen Adressvergleich für deren Speichorte implementiert werden, was effizienter ist. Die Implementierung der Klasse *Sym* in Java basiert auf einer Hashmap als String-Pool und der in Java schon vorhandenen *intern*-Methode für die Klasse *String*.

```
package sym;

public class Sym {

  private static java.util.Map<String,Sym> dict =
                          new java.util.HashMap<String,Sym>();
  private String name;
  private Sym(String n) { name = n; }
  public String toString() { return name;  }

  public static Sym newSym(String n) {
    String u = n.intern();
    Sym s = dict.get(u);
    if (s == null) {
      s = new Sym(u);
      dict.put(u, s);
    }
    return s;
  }
}
```

Anmerkung für die C-Implementierung:

In C wird jeder Knotentyp durch einen `struct`-Typ repräsentiert, der die zugehörigen Attribute als Komponenten enthält. Eine binäre Operation (Java: Klasse *OpExp*) sieht wie folgt aus (siehe `absyn.h`):

```
struct {  int op;                // Operator
          struct absyn *left;    // linker Operand
          struct absyn *right;   // rechter Operand
} opExp;
```

Der AST mit seinen verschiedenen Knotentypen wird wie folgt modelliert:

```
typedef struct absyn {
  int type;     // Knotentyp-ID
  int line;     // Zeilennumer für Fehlermeldungen
  union {
     ...        // hier stehen die Definitionen aller Knotentypen
       } u;
} Absyn;
```

Für den Knotentyp `type` sind in `absyn.h` symbolische Konstanten definiert, z.B. `ABSYN_PROCDEC` für die Prozedurdeklaration. Das `union`-Attribut `u` enthält die einzelnen Knotentypen in Form von `union`-Komponenten, z.B. die oben angegebene `struct opExp`. Für das String-Interning sind der Konstruktor *newSym* und die Funktion *symToString* verfügbar (vgl. `sym.h`).

Compiler Frontend: Konstruktion des AST

Beispiel für die Verwendung der Konstruktoren zum Baumaufbau

Der Parser soll nach Analyse eines korrekten SPL-Programms den zugehörigen AST als Resultat liefern. Für den Aufbau nutzt er die Konstruktoren der einzelnen Knotentypen. Ein Beispiel zur Baumaufbau „von Hand“:

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

Den AST dazu könnte man wie folgt aufbauen:

```
TypeDec td = new TypeDec(
                 row1,
                 col1,
                 new Sym("vector"),
                 new ArrayTy(
                     row2,
                     col2,
                     10,
                     new NameTy(
                         row3,
                         col3,
                         new Sym("int"))));
```

- Die Verschachtelung der Konstruktoren spiegelt die hierarchische Struktur der Typdeklaration wieder. Das ganze ist ein Typeklaration (*TypeDec*-Objekt). Die rechte Seite ist ein Typausdruck der Variante Array-Typausdruck (Objekt der Klasse *ArrayTy*). Zu einem Array-Typausdruck gehört immer als Untertyp der Komponententyp, im Beispiel ein Typname (Objekt der Klasse *NameTy*).
- Zu jedem Knoten gehören eine Zeilen- und eine Spaltennummer (Attribute: row, col). Diese Attribute sind in der Klasse *Absyn* definiert und werden an alle Subklassen vererbt. Sie dienen im Fall eines Syntaxfehlers dazu, die Stelle des Fehlers in der Fehlermeldung ausgeben zu können, z.B. „Fehler in Typdeklaration in Zeile 10, Spalte 25.“

  Bei der Erzeugung neuer Knoten muss man diese Nummern angeben (oben z.B. `row1` und `col1`). Sie lassen sich aus den Tokens bestimmen, die der Scanner liefert. Wenn der Scanner das Token zum Schlüsselwort „type“ erzeugt, enthält dieses eine Zeilen und Spaltenangabe, die der Parser für die Typdeklaration als Ganzes übernehmen kann.

Anmerkung für die C-Implementierung:

Für den Aufbau der Bäume sind in `absyn.c` bzw. `absyn.c` eine Reihe von Konstruktoren definiert, die in der gleichen Weise verwendet werden, wie bei der Java-Implementierung, z.B.

```
Absyn *newTypeDec(int line, Sym *name, Absyn *ty);
Absyn *newNameTy(int line, Sym *name);
Absyn *newArrayTy(int line, int size, Absyn *ty);
Absyn *newTypeDec(int line, Sym *name, Absyn *ty);
```

Konstruktion des AST mit dem Parsergenerator

Ein Parsergenerator wir *cup* oder *bison* basiert auf dem Prinzip der *Syntaxorientierten Übersetzung*, das über die reine Syntaxprüfung hinaus gehende Berechnungen an die Ableitungsschritte gemäß der Grammatik koppelt. Dazu werden kontextfreie Grammatiken in zwei Aspekten erweitert:

1. Attributierte Grammatiken

Eine „Attributierte Grammatik“ ist eine Erweiterung, die es erlaubt, jedem Grammatiksymbol ein Attribut zuzuordnen. Dabei muss man den Typ des Attributs angeben. Das Attribut kann eine beliebige komplexe Datenstruktur sein. Intern arbeitet der Generator mit dem Attributtyp *Object*.

Beispiel 1: Für die Tokenkategorie INTLIT wird der Zahlenwert benötigt. Es wird ein Attribut vom Typ Integer definiert. In der cup-Eingabedatei wird dies wie folgt spezifiziert:

```
terminal Integer INTLIT;
```

Beispiel 2: Für die Tokenkategorie IDENT wird der Name des Bezeichners benötigt. Es wird ein Attribut vom Typ String definiert. In der cup-Eingabedatei wird dies wie folgt spezifiziert:

```
terminal String IDENT;
```

Beispiel 3: Für While-Schleifen wird ein abstrakter Syntaxbaum benötigt. Es wird ein Attribut vom Typ *Stm* definiert. In der cup-Eingabedatei wird dies wie folgt spezifiziert:

```
non terminal Stm while_statement;
```

Falls für ein Symbol kein Attribut benötigt wird, lässt man den Typ einfach weg:

```
terminal ARRAY, ELSE, IF, OF, PROC;
```

Anmerkung für die C-Implementierung mit *bison*:

Für den Compilergenerator bison gilt: Der Attributtyp ist `YYSTYPE`. Die obigen Beispiele sehen für *bison* wie folgt aus:

```
scanner.h:
    typedef struct { int line; }            NoVal;     // sonstige Tokens
    typedef struct { int line; int val; }   IntVal;    // fuer INTLIT
    typedef struct { int line; char *val; } StringVal; // fuer IDENT

parser.y:
    %union {
      NoVal noVal;
      IntVal intVal;
      StringVal stringVal;
      Absyn *node;
    }
    /* ACHTUNG: Daraus erzeugt bison YYSTYPE:  typedef union { ... } YYSTYPE; */
```

```
%token      <noVal>         ARRAY ELSE IF OF PROC
%token      <stringVal>     IDENT
%token      <intVal>        INTLIT

%type       <node>          while_statement
```

2. Semantische Aktionen

Die Ableitungsregeln der Grammatik können durch Code-Fragmente erweitert werden, die wir als semantische Aktionen bezeichnen. Wir benötigen nur Aktionen, die am Ende der rechten Regelseite (vor dem Semikolon) eingefügt werden: X `::=` $X_1 \ldots X_n$ `{:` $Aktion$ `:}` `;`

Die Aktionen bestehen aus Java-Code mit speziellen Erweiterungen zum Zugriff auf die Attribute. Als Beispiel nehmen wir die Ableitungsregel für eine Multiplikation:

```
term ::= term STAR factor;
```

Die Multiplikation ist linksassoziativ und wird daher (siehe Skript) mit einer linksrekursiven Regel definiert. Ein AST für die Multiplikation ist ein Objekt der Klasse *OpExp*. Der Konstruktor benötigt eine Zeilen- und Spaltennummer, die Codierung des Operators (hier `OpExp.MUL`), und die Bäume für die Operanden.

Ein Baum für einen Ausdruck ist ein Objekt der Klasse *Exp*. Wir definieren also:

```
non terminal Exp term;
non terminal Exp factor;
```

Der Parser baut den Baum Bottom-Up auf, also zuerst die Teilbäume und danach den Wurzelknoten. Die semantische Aktion beschreibt also, wie man den Baum für die gesamte Multiplikation aufbaut, wobei man die Attribute der Unterbestandteile nutzen kann. In der Aktion steht das Schlüsselwort `RESULT` für das Attribut des Nonterminals auf der linken Regelseite. In unserem Beispiel ist `RESULT` der AST für die Multiplikation. Auf die Attribute der Symbole auf der rechten Regelseite kann man zugreifen, indem man innerhalb der Regel eindeutige Namen (Tags) einführt und die Schreibweise *Symbol*:*Tag* benutzt.

Beispiel:

Statt

```
term ::= term STAR factor;
```

schreibt man

```
term ::= term:lop STAR:opr factor:rop;
```

Die Tags sind: lop (linker Operand), opr (Operator) und rop (rechter Operand). In der Aktion repräsentieren die Tags jetzt die Attribute der entsprechenden Symbole. Also ist `lop` der AST zum linken Operanden, `opr` das Attribut des Multiplikationsoperators und `rop` der AST zum rechten Operanden.

Zu jedem Tag *xxx* ist außerdem das Tag *xxx*`left` und das Tag *xxx*`right` in der Aktion verwendbar, die Zeilennummer und die Spaltennummer, die in den Knoten stehen.

Der Baumaufbau erfolgt in der semantischen Aktion:

```
term ::= term:lop STAR:opr factor:rop {:
              RESULT = new OpExp(oprleft, oprright, OpExp.MUL, lop, rop);
              :} ;
```

Ein anderes Beispiel: Variablendeklaration

```
non terminal DecList variable_declarations;
non terminal Dec     variable_declaration;

variable_declarations   ::= /* empty */
                            {:
                                RESULT = new DecList();
                            :}
                        |   variable_declaration:s1 variable_declarations:s2
                            {:
                                RESULT = new DecList(s1, s2);
                            :}
                        ;
```

Verarbeitung von Attributen und semantischen Aktion durch den Parsergenerator

Der vom Generator erzeugte Parser wird die Attribute in der Regel unabhängig vom Attributtyp handhaben. Dazu wird Parser-intern für alle Attribute derselbe Attributtyp *Object* verwendet. Ein SHIFT/REDUCE-Parser berechnet eine Ableitung rückwärts und bringt dabei zuerst die rechte Seite einer Ableitungsregel auf seinen Parser-Stack, um sie dann in einer REDUCE-Aktion durch die linke Regelseite zu ersetzen.

Als Beispiel betrachten wir noch einmal die Multiplikation:

```
term ::= term STAR factor;
```

Die REDUCE-Aktion findet statt, wenn alle Bestandteile der rechten Regelseite ganz oben auf dem Parser-Stack stehen, d.h. wenn der Parser den linken Operanden, den Operator und den rechten Operanden komplett gelesen hat.

```
Stack:      vor REDUCE                          nach REDUCE
                .                                   .
                .                                   .
                .                                   .
              term                  TOP ---->     term
              STAR
TOP ---->     factor
```

Um Attribute zu unterstützen, wird der Parser auf seinem Stack zu jedem Grammatik-Symbol X den dazu gehörenden Attributwert $attr_X$ mit auf dem Stack abspeichern. Auf dem Stack stehen also Paare der Form $(X, attr_X)$, im Beispiel:

```
Stack:      vor REDUCE                          nach REDUCE
                .                                   .
                .                                   .
                .                                   .
            (term,attr1)            TOP ---->   (term, attr4)
            (STAR,attr2)
TOP ---->   (factor,attr3)
```

Die semantischen Aktionen werden immer bei der REDUCE-Aktion ausgeführt. Da vor der Reduktion die rechte Regelseite mitsamt den Attributen oben auf dem Stack steht, können in der Aktion diese

Attribute (attr1,attr2,attr3) genutzt werden, um das neue Attribut für die linke Regelseite (attr4) zu berechnen. Dieses wird zwischengespeichert, bevor die Symbole der rechten Regelseite vom Stack entfernt werden. Danach wird das Symbol auf der linke Regelseite mit seinem zwischengespeicherten Attribut auf dem Stack abgelegt.

Betrachten wir nun die semantischen Aktionen an unserem Beispiel:

```
term ::= term:lop STAR:opr factor:rop {:
              RESULT = new OpExp(oprleft, oprright, OpExp.MUL, lop, rop);
              :} ;
```

Das Attribut des Multiplikations-Terms `RESULT` entspricht in der obigen Stack-Darstellung dem Attribut `attr4`. Zur Berechnung werden `attr1 (lop)`, `attr2 (opr)` und `attr3 (rop)` verwendet. Der Generator kann aus der Länge der rechten Regelseite den Speicherort der Attribute zu den rechts auftretenden Symbolen relativ zum TOP-Zeiger leicht bestimmen und im generierten Code entsprechende Stack-Zugriffe generieren. Da die Attributwerte vom Typ *Object* sind, muss im generierten Java-Code noch ein Cast auf den korrekten Attributtyp stehen. Dieser kann anhand der Typangabe für das Symbol erzeugt werden.

Beispiel: Der Attributwert für den rechten Operanden (Symbol: factor, Wert: rop) steht ganz oben auf dem Stack, weil *factor* das letzte Symbol der rechten Regelseite ist. Aus der Deklaration

```
non terminal Exp factor;
```

kann der Generator für den Attributwert einen Cast von *Object* nach *Exp* erzeugen. Durch die einmalige Typangabe für das Attribut erspart man sich also an allen Verwendungsstellen des Symbols das Casten des Attributwerts.

Die Zeilen- und Spaltennummern sind nichts anderes als zusätzliche, automatisch verwaltete Komponenten der Attribute.

Fazit: Der Parsergenerator kopiert im Prinzip die semantischen Aktionen aus seiner Input-Datei in den generierten Parser-Quelltext. Dabei muss er aber alle Zugriffe auf die Attribute im Hinblick auf Speicherort-Bestimmung und Downcast auf die richtige Absyn-Subklasse verarbeiten, um korrekten Java-Code zu erzeugen. Die Ausführung einer semantischen Aktion zu einer Regel R erfolgt immer bei der Reduktion der Regel.

Anmerkung für die C-Implementierung mit *bison*:

Für den Compilergenerator bison gilt im Prinzip das Gleiche, nur die Notation ist etwas anders. Unser Multiplikations-Beispiel für bison:

```
term : term STAR factor { $$ = newOpExp($2.line, ABSYN_OP_MUL, $1, $3); };
```

Statt `RESULT` verwendet man `$$` und das Attribut für das i-te Symbol auf der rechten Regelseite ist `$`i.

Für das Backend: Verarbeitung des AST gemäß Visitor-Entwurfsmuster

Im Compiler-Backend werden mehrere Algorithmen benötigt, die alle den AST von der Wurzel ausgehend rekursiv bearbeiten, z.B. für die Ausgabe des Baums zu Testzwecken, die semantische Analyse oder die Assemblercode-Erzeugung. Gemeinsam ist diesen Algorithmen, dass die Verarbeitung eines Knotens von dessen Typ abhängt. Da der Knotentyp in einer OO-Sprache wie Java durch eine AST-Subklasse implementiert wird, liegt es nahe, jede beim Durchlaufen des Baums benötigte Knotentyp-spezifischen Aktion als Methode der entsprechenden AST-Subklasse zu definieren. Wenn beispielsweise die Erzeugung von Assemblercode für einen AST-Knoten durch eine Methode *codegen* umgesetzt werden soll, steht die Code-Erzeugung für While-Schleifen als Methode *codegen* in der Klasse *WhileStm* und die Code-Erzeugung für Prozeduraufrufe als Methode *codegen* in der Klasse *CallStm*.

Für den Entwickler des Algorithmus ist es allerdings äußert unpraktisch und unübersichtlich, wenn der Algorithmus auf 30 oder mehr Klassendefinitionen verteilt ist. Ein Beispiel mit 3 Algorithmen: *show* (Ausgabe des Baums), *codegen* (Code-Generator) und *typeCheck* (Typprüfung/semantische Analyse):

```
public abstract class Absyn {
  ...
  public abstract void show();
  public abstract Type typeCheck();
  public abstract void codegen();
  ...
}
```

Die Knotentyp-spezifischen Anteile der Algorithmen sind Methoden der AST-Subklassen:

```
class OpExp extends Exp {
  public void show ()        { ... }
  public Type typeCheck ()   { ... }
  public void codegen ()     { ... }
  ...
}

class VarExp extends Exp {
  public void show ()        { ... }
  public Type typeCheck ()   { ... }
  public void codegen ()     { ... }
  ...
}

class IntExp extends AST {
  public void prettyPrint () { ... }
  public Type typeCheck ()   { ... }
  public void codeGen ()     { ... }
  ...
}
  ...
```

Um das Dilemma der auf viele Klassen verteilten Algorithmen zu vermeiden, wird man nach einer Möglichkeit suchen, jeden Algorithmus zusammenhängend, d.h. innerhalb einer Klassendefinition, zu

implementieren. Man könnte jeden der Baumverarbeitungs-Algorithmen beispielsweise in eine einzige Methode packen, in der eine explizit programmierte Fallunterscheidung nach Knotentyp erfolgt, z.B.:

```
public class AbsynPrettyPrinter {
  public void showNode (){

    // fuer jeden Knotentyp spezifische Ausgabe:

    if      (node instanceof DecList) { ... }
    else if (node instanceof TypeDec)  { ... }
    else if (node instanceof ProcDec)  { ... }
    else if (node instanceof ParDec)  { ... }
    else if (node instanceof VarDec)  { ... }
    else if (node instanceof NameTy)  { ... }
    else if ...
      .
      .
      .
  }
}
```

Solche „Verteiler“ blähen nicht nur den Code auf, sie sind auch in der Ausführung sehr ineffizient. Daher wird stattdessen das Entwurfsmuster „Visitor“ verwendet. Dazu dient die abstrakte Klasse *Visitor*, die einen Baumverarbeitungs-Algorithmus repräsentiert. Ein konkreter Algorithmus, den den Baum durchläuft, wird als Subklasse von Visitor programmiert, z.B.

```
class AbsynPrettyPrinterVisitor extends Visitor { ... }
class CodeGeneratorVisitor extends Visitor { ... }
```

In dieser Visitor-Klasse stehen dann für alle Knotentypen die entsprechenden Methoden für die Code-Erzeugung. Konkret wird für jeden Knotentyp eine Methode „visit“ definiert, in der die spezifische Verarbeitung programmiert wird.

Baumausgabe:

```
class AbsynPrettyPrinterVisitor extends Visitor {
  void visit(DecList d) { ... }
  void visit(TypeDec t) { ... }
  void visit(ProcDec p) { ... }
  void visit(ParDec) p) { ... }
     ...
```

Codererzeugung:

```
class CodeGeneratorVisitor extends Visitor {
  void visit(DecList d) { ... }
  void visit(TypeDec t) { ... }
  void visit(ProcDec p) { ... }
  void visit(ParDec) p) { ... }
     ...
```

Überladungen, Polymorphismus und trickreiche Rekursion

Zur Erinnerung: Ein Methodenaufruf `obj.m(...)` in Java ist **überladen**, wenn es für *obj* mehrere Methoden mit dem gleichen Namen gibt, die sich anhand der Parameter-Signatur (Anzahl, Typen) unterscheiden. Die Unterscheidung erfolgt **zur Übersetzungszeit** durch den Java-Compiler!

Davon zu unterscheiden ist Polymorphismus. Am Beispiel:

```
class C { abstract void m(int i) {...} ... }
class C1 extends C { void m(int i) {...} ... } // C1 implementiert m
class C2 extends C { void m(int i) {...} ... } // C2 implementiert m auch

...
  C obj = null;
  if (...)
    obj = new C1(...);
  else
    obj = new C2(...);

  obj.m(0);   // Welches m? C1 oder C2? Polymorphismus
```

Java verlangt, dass zum Zeitpunkt des Methodenaufrufs *obj* inspiziert wird. Ist der aktuelle Wert ein Objekt der Subklasse *C1* muss auch die in *C1* definierte Methode *m* aktiviert werden. Dies gilt entsprechend auch für *C2*. Die Methodenzuordnung ist also eine Laufzeitaktion.

Eine „visit"-Methode für einen AST-Knoten ruft bei einer Rekursion diejenige „visit"-Methode für einen Kindknoten auf, die für dessen Knotentyp zuständig ist. Aus technischen Gründen erfolgt der rekursive Aufruf nicht direkt, sondern über den Umweg des Aufrufs einer „accept"-Methode, die zum Knotentyp gehört.

Folgendes Beispiel (Baumausgabe für eine Liste von Deklarationen - *DecList*) zeigt, dass ein direkt rekursiver Aufruf von *visit* nicht ohne weiteres möglich ist:

```
class AbsynPrettyPrinterVisitor extends Visitor {
  void visit(DecList decList) {
     // Ausgabe der Liste decList -> jedes Element von decList ausgeben
     for (Dec dec: decList)
        visit(dec);  // FEHLER: visit-Methode hängt vom Typ von dec ab!
  }
  ...
```

Das Problem ist, dass für jeden Algorithmus und jeden Knotentyp eine *visit*-Methode existiert. Bei n Algorithmen und m Knotentypen gibt es also $n * m$ verschiedene *visit*-Methoden. Um den Aufruf `visit(dec)` also eindeutig zuordnen zu können, ist die Visitor-Subklasse (Klasse von *this*) und die *Absyn*-Subklasse von *dec* nötig. Während im Beispiel-Aufruf die Visitor-Subklasse bekannt ist (*AbsynPrettyPrinterVisitor*), kann der Typ von *dec* vom Java-Compiler nicht näher bestimmt werden. Erst zur Laufzeit kann man feststellen, ob es sich um *TypeDec*, *ProcDec*, *ParDec* oder *VarDec* handelt, welche alle ihre eigenen Ausgabemethode haben. Java verlangt aber, dass die Auflösung von Überladungen, also die eindeutige Zuordnung der aufzurufenden Methode anhand der Parametertypen zur Übersetzungszeit möglich sein muss.

Daher behilft man sich mit einem indirekten rekursiven Aufruf:

```
class AbsynPrettyPrinterVisitor extends Visitor {
  void visit(DecList decList) {
     // Ausgabe der Liste decList -> jedes Element von decList ausgeben
     for (Dec dec: decList)
        dec.accept(this);
  }
  ...
```

Da *dec* jetzt nicht mehr Parameter ist, sondern das Objekt, dessen *accept*-Methode bestimmt werden muss, handelt es sich nicht um eine Überladung, sondern um Polymorphismus, dessen Auflösung zur Laufzeit von Java unterstützt wird. Das Objekt *dec* enthält seinen Knotentyp, z.B. *TypeDec*. Zum Zeitpunkt des Methodenaufrufs wird das Objekt von der Java VM inspiziert und anhand des Typs die *accept*-Methode der Klasse *TypeDec* aufgerufen.

```
public class TypeDec extends Dec {
    ...
    public void accept(Visitor v){
        v.visit(this);
    }
}
```

Dort erfolgt wiederum ein *visit*-Aufruf: `v.visit(this)` Wie wird hier die zuständige *visit*-Methode ermittelt? Der Typ des Arguments *this* ist zur Übersetzungszeit bekannt (*TypeDec*), der Typ von *v* kann zur Laufzeit ermittelt werden, da es sich hier um Polymorphismus bzgl. der Visitor-Klasse handelt.

Der Ablauf komplett:

1. Beim Aufruf `dec.accept(this)` ist der Typ des Arguments zur Übersetzungszeit bekannt: *AbsynPrettyPrinterVisitor*. Der Typ von *dec* wird durch Inspektion des Objekts zur Laufzeit ermittelt und die zuständige accept-Methode von *TypeDec* aufgerufen.
2. Beim Aufruf `v.visit(this)` ist der Typ des Arguments zur Übersetzungszeit bekannt: *TypeDec*. Der Typ von *v* wird durch Inspektion des Objekts zur Laufzeit ermittelt und die für *TypeDec* zuständige visit-Methode von *AbsynPrettyPrinterVisitor* aufgerufen.

Anmerkung für die C-Implementierung:

Die ganze Diskussion zum Visitor-Entwurfsmuster ist für nicht OO-Sprachen irrelevant. Die Auswahl der zur Knotenklasse passenden spezifischen Verarbeitung, die bei Java automatisch im Rahmen der Polymorphismus-Auflösung erfolgt, muss in C von Hand codiert werden. Dem Programmierer bleibt nichts anderes übrig, als bei **jeder** Knotenverarbeitung eine Fallunterscheidung über alle Knotentypen explizit zu programmieren. Beispiel:

```
static void showNode(Absyn *node, int indent) {
  if (node == NULL) { error("showNode got NULL node pointer");  }
  switch (node->type) {
    case ABSYN_NAMETY:
      showNameTy(node, indent);
      break;
    case ABSYN_ARRAYTY:
      showArrayTy(node, indent);
      break;
    case ABSYN_TYPEDEC:
      showTypeDec(node, indent);
      break;

     .   usw.
     .   usw.
     .   usw.
   default:
      error("unknown node type %d in showAbsyn", node->type);
  }
}
```