Objekt-Orientierung im Compilerbau

Bernhard Kühl
Universität Osnabrück, Fachbereich Mathematik/Informatik
bernd@informatik.uni-osnabrueck.de
http://www.inf.uos.de/bernd

1 Inhaltverzeichnis

2 Objekt-Orientierung im Compilerbau — Motivation

- Am Anfang galten Compiler als nur sehr schwer zu implementierende Programme. Seitdem wurden aber systematische Techniken und Werkzeuge zur automatischen Erzeugung von (Teilen von) Compilern entwickelt.
- Nur wenige Personen kommen allerdings in die Situation, für eine höhere Programmiersprache einen Compiler zu entwickeln. Aber auch andere “Dinge” sind wie Programme einer Sprache zu analysieren und zu verarbeiten:
  - *grep*, *sed* oder *awk.*
  - Konfigurationsdateien.
  - Pretty-Printer.
  - Präprozessoren.
- Die Entwicklung derartiger Software ist noch immer nicht ohne Probleme:
  - Die Entwicklung von Übersetzer-Software ist von Hand sehr mühsam und fehleranfällig, und nur schwer zu erweitern.
  - Werkzeuge zur Generierung von Software sind für Anfänger nicht leicht zu erlernen und fest auf ihre Funktionalität begrenzt. Eine Erweiterung der Werkzeuge ist kaum möglich.
- Objekt-orientierte Programmierung ist das Programmierkonzept der vergangenen Jahre.
- Es ist daher naheliegend, die Objekt-Orientierung auf das Problem der Verarbeitung von Sprachen anzuwenden, um zu zeigen, daß ...
  - ... durch den Einsatz der objekt-orientierten Programmierung die Entwicklung von (Teilen von) Compilern vereinfacht werden kann.
  - ... die so entwickelten Werkzeuge und deren Resultate an Mächtigkeit gegenüber den Werkzeugen und Resultaten der gängigen Techniken gewinnen.
  - ... auch objekt-orientiert konzipierte Software des Compilerbaus der Vorteil der leichten Erweiterbarkeit zugute kommt.
  - ... die Wiederverwendung von Klassen und Objekten die Entwicklung und Verwendung von Scannern oder Compilern erleichtert.
  - ... die objekt-orientierte Programmierung ein Automatismus für *divide & conquer* ist und auch hier gewinnbringend genutzt werden kann. Selbst komplexe Algorithmen des Compilerbaus zerfallen so in kleine, einfache Teilprobleme und sind dadurch in der Lehre für Studierende leicht zu verstehen bzw. sogar selbständig zu entdecken.
  - ... der Einsatz verschiedener typischer Entwurfsmuster der Objekt-Orientierung die Flexibilität und Erweiterbarkeit zum Beispiel eines Scanners oder eines Compilers wesentlich erhöht.

3 Vor- und Nachteile der Objekt-Orientierung

Vorteile

- Kapselung (*information hiding*).
- Die objekt-orientierte Programmierung ist ein Automatismus für *divide & conquer*.
- Vererbung.
- Erweiterbarkeit.
- Wiederverwendung.
- Strukturen sind durch Graphen von Objekten leicht zu modellieren.
- Persistene Objekte.
- Entwurfsmuster.

Nachteile

- Objekt-orientierte Systeme bestehen aus sehr vielen Klassen und Objekten.
- Performance.

4 Thesen

- Inhalt meiner Arbeit ist es, die Anwendung der objekt-orientierten Programmierung auf den Compilerbau zu untersuchen.
- Die Arbeit soll zeigen, daß unter anderem folgende Vorteile der Objekt-Orientierung den Compilerbau zu neuen Techniken und Lösungen vorantreiben:
  - Wiederverwendung.
  - Entwurfsmuster.
  - Erweiterbarkeit/Vererbung.
  - Die objekt-orientierte Programmierung ist ein Automatismus für *divide & conquer.*

5 Lexikalische Analyse

Gängige Techniken

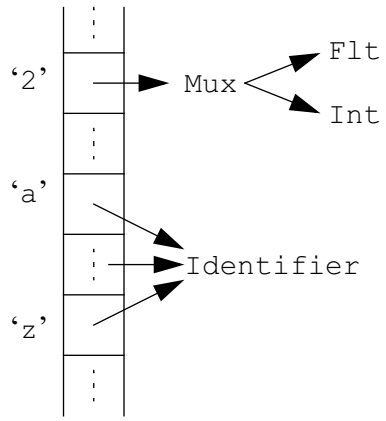
- handgeschriebene Scanner.
- fertige Scanner[klassen], Beispiel `java.io.StreamTokenizer`.
- Scannergeneratoren, Beispiel *lex/flex/JLex*.
- Nachteile handgeschriebener Scanner:
  - mühsam zu implementieren (Beispiel: Gleitkommazahlen).
  - fehleranfällig.
  - schwer erweiterbar.
  - Performance.
- Nachteile fertiger Scanner[klassen]:
  - Nicht allgemein verwendbar.
  - `StreamTokenizer`: Zwischenraum wird immer ignoriert, Zwischenraum ist als Symbol nicht greifbar, Integer- und Gleitkommazahlen können nicht gleichzeitig erkannt werden und Exponentialdarstellung für Gleitkommazahlen kann man gar nicht erkennen, ...
- Nachteile Scannergeneratoren:
  - Der Entwicklungszyklus hat zwei Schritte.
  - Soll der Scanner erweitert oder geändert werden, müssen beide Schritte erneut ausgeführt werden.
  - Reguläre Ausdrücke sind für Anfänger schwer zu erlernen und vor allem kryptisch.
  - Um einen fremden oder älteren komplexeren regulären Ausdruck zu verstehen, bedarf es einiger Erfahrung:

    ```
    "/*"([^*]|"*"+[^/*])*"*"+"/"
    ```

  - Ohne einen ausführlichen Test ist ein komplexer regulärer Ausdruck außerdem fehleranfällig.
  - Es gibt reale Probleme, die mit regulären Ausdrücken nur schwer oder auch gar nicht lösbar sind: z.B. einen fehlerfreien regulären Ausdruck für einen C-Kommentar oder gar einen Ausdruck für geschachtelte C-Kommentare.

lolo (language-oriented lexer objects)

- Das Design von lolo ist strikt objekt-orientiert.
- Ein Scanner wird als Wettbewerb von Objekten in einem Raum modelliert.
- Jedes der Objekte ist in der Lage, genau ein Symbol zu erkennen.
- Ein Raum dieser Objekte verwaltet die Eingabe und liefert den Objekten Zeichen für Zeichen aus der Eingabe.
- Die Erkenner-Objekte teilen dem Raum (möglicherweise mehrmals) mit, wenn sie mit dem aktuellen Zeichen ein Symbol erkannt haben.
- Objekte, die mit dem Zeichen nichts anfangen, verlassen den Raum.
- Die Erkennungs-Runde endet, wenn alle Erkenner keine weiteren Zeichen akzeptieren, also alle den Raum verlassen haben.
- Gewinner der Runde und damit Repräsentant des gescannten Symbols ist das Objekt, welches als letztes Erkennung signalisiert hat, wenn alle den Raum verlassen haben (Mehrdeutigkeit ist auflösbar).
- Typische Symbol-Erkenner sind daher als Klassen in einer Bibliothek vorbereitet: `CComment`, `QuotedText`, `Char`, `Flt`, `Set`, `HashComment`, `SetMN`, `Int`, `SimpleIdentifier`, `JavaIdentifier`, `SimpleWhitespace`, `JavaWhitespace`, `SlashSlashComment`, `JavadocComment`, `Word`, `QuotedChar`, ...
- Gewinnt ein Objekt eine Erkennungsrunde, kennt das Objekt optional ein anderes Objekt, welches die zugehörige Aktion implementiert.
- Alle Symbol-Erkenner- und Aktions-Klassen sind serialisierbar.
- Aus Performancegründen treibt eine Tabelle die Erkennung über das erste Zeichen eines Symbols. Nachteil: die Tabelle muß einmalig berechnet werden.



Fazit

- Der neue Ansatz ist objekt-orientiert und damit für objekt-orientierte Sprachen wie Java, Objective-C oder C++ intuitiver.
- lolo kann sehr einfach an durch Parsergeneratoren (wie zum Beispiel *oops* oder *jay*) erzeugte Parser angeschlossen werden.
- Eine Firma, die eine andere Bibliothek von Erkennern entwickelt und verkauft, kann die Klassen in binärer Form verkaufen.
- Spezialisierung existenter Klassen durch Vererbung bzw. Ableitung, Beispiel `SimpleWhitespace` und `JavaWhitespace`.
- Die Entwicklungszeit für lolo-Scanner ist sehr gering. Das System ist daher ideal für rapid prototyping.
- Die Klassen der Bibliothek sind für verschiedene Scanner zu verwenden (reuse).
- Serialisierung: Scanner können von verschiedenen Projekten in binärer Form geteilt werden (reuse).
- Umkonfiguration (Scanner, Aktionen) im laufenden Betrieb.
- Natürliche Unicode-Unterstützung.
- Da Objekte Zustände besitzen, können die Symbol-Erkenner mächtiger als endliche Automaten sein.
- Nachteil: Performance.

Thesen

- Wiederverwendung.
- Erweiterbarkeit/Vererbung.
- Die objekt-orientierte Programmierung ist ein Automatismus für *divide & conquer.*

Ein Beispiel

- ScannerExample.java:

```
public class ScannerExample {
    public static void main(String args[]) throws Exception {
        lolo.Scan.Action action = new lolo.Scan.Action () {
            public void action(lolo.Scan sender, char [] buffer, int off,
                               int len) {
                System.out.println("\tfound -"+new String(buffer, off, len)+
                                   "-\tsender is "+sender);
            }
        };

        Scanner scanner = new Scanner(
            new lolo.Scan[] {
                new Word("<=").setAction(action),
                new Word(">=").setAction(action),
                new Word("==").setAction(action)
        });
        scanner.add(new Int().setAction(action));
        scanner.add(new Flt(true).setAction(action));
        scanner.add(new SimpleIdentifier().setAction(action));
        scanner.add(new SimpleWhitespace());
        scanner.add(new QuotedText('"').setAction(action));
        scanner.add(new CComment(true).setAction(action).setIgnore(false));
        scanner.add(new Set("+-=/*<>;(){}").setAction(action));
        scanner.pack();

        Input input = new Input(new InputStreamReader(System.in), 2);
        lolo.Scan winner;
        while ((winner = scanner.scan(input)) != null)
            ;
    }
}
```

- Die Ausführung:

```
$ java -classpath .:../jars/lolo.jar ScannerExample
/* /* */
*/ while ( 2 >= a ) { print "yep"; }
        found -/* /* */
*/-     sender is lolo.scans.CComment[nested]
        found -while-   sender is lolo.scans.SimpleIdentifier
        found -(-       sender is lolo.scans.Set[+-=/*<>;(){},true]
        found -2-       sender is lolo.scans.Int[maxDigits 2147483647]
        found ->=-      sender is lolo.scans.Word[>=]
        found -a-       sender is lolo.scans.SimpleIdentifier
        found -)-       sender is lolo.scans.Set[+-=/*<>;(){},true]
        found -{-       sender is lolo.scans.Set[+-=/*<>;(){},true]
        found -print-   sender is lolo.scans.SimpleIdentifier
        found -yep-     sender is lolo.scans.QuotedText["]
        found -;-       sender is lolo.scans.Set[+-=/*<>;(){},true]
        found -}-       sender is lolo.scans.Set[+-=/*<>;(){},true]
```

6 Ein Parser aus Objekten

Objekte für Grammatiken

- Kontextfreie Grammatik: Terminals, Non-Terminals, Regeln und eine Start-Regel.
- Beispiel BNF:

```
identifier : letter | identifier letter | identifier number
```

- Beispiel EBNF:

```
identifier : letter [{ letter | number }]
```

- Instanzen von Objekten des Pakets `oops.parser` repräsentieren eine EBNF-Grammatik:
  - Ein `Parser`-Objekt verweist auf viele `Rule`-Instanzen.
  - Jede `Rule` verbindet das Non-Terminal auf der linken Seite mit der rechten Seite.
  - Die rechte Seite einer `Rule` ist ein einzelnes Objekt, zum Beispiel eine Sequenz von Symbolen in einer `Seq`-Instanz.
  - Alternativen werden in einem `Alt`-Objekt gesammelt.
  - `Some` (1-n) und `Many` (0-n) kapseln Wiederholungen und `Opt` (0-1) repräsentiert einen optionalen Teil.
  - `Lit` stellt Literale (`"<="`), `Token` Symbol-Kategorien (Zahlen) und `Id` Verweise (auf Regeln) dar.
- Das Beispiel als Objekt-Baum:

```
Parser
   Rule identifier
      Seq
         Token letter
         Many
            Alt
               Token letter
               Token number
```

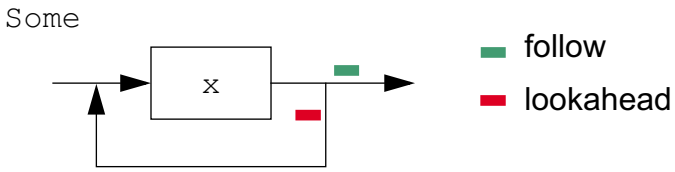
- oops: object orientated parser system
- Fähigkeiten des Baums?
  - Baum repräsentiert Grammatik.
  - Parsierung: rekursiver Abstieg, `parse()`-Methode.
  - Prüfung der Grammatik.
- oops-Parser sind serialisierbar und damit wiederverwendbar.

NSD anstelle von Syntaxgraphen

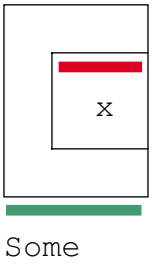
- EBNF:

```
some : { x };
```

- Syntaxgraph:

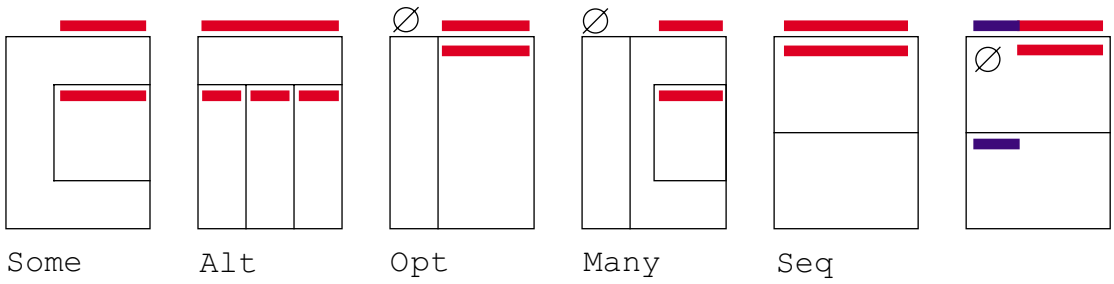


- NSD:



Lookahead-Mengen

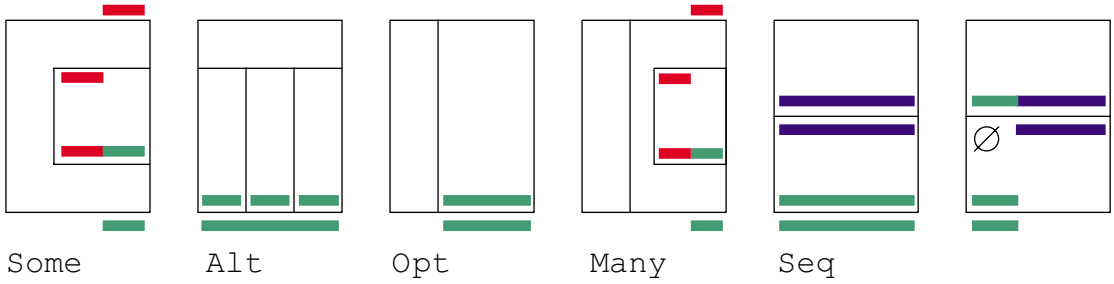
- Während der Parsierung treffen die Objekte Entscheidungen aufgrund der lookahead-Mengen.
- Lookahead: Die Menge aller Eingabesymbole, die an dieser Stelle als nächstes in der Eingabe auftreten können.
- Die Berechnung des lookahead ist lokal auf die Klassen verteilt:



  - `Some` übernimmt den lookahead von seinem Unterknoten.
  - `Alt` berechnet den lookahead durch Vereinigung der lookaheads der Alternativen.
  - `Opt` und `Many` übernehmen den lookahead von ihrem Unterknoten und fügen die leere Eingabe hinzu.
  - `Seq` nimmt den lookahead des ersten Sequenz-Elements und addiert den lookahead der folgenden Elemente, solange die leere Eingabe akzeptiert wird. Alle Elemente werden zur lookahead-Berechnung aufgefordert.
  - `Rule` übernimmt den lookahead von der rechten Seite und `Id` von der `Rule`.
  - `Lit` und `Token`: Terminal ist lookahead.
- Die Berechnung startet mit der Start-Regel.
- Die Verteilung der Berechnung zerlegt diese in kleine und einfache Teilprobleme. Der Algorithmus ist so leicht zu verstehen.

Follow-Mengen

- Follow: Die Menge aller Eingabesymbole, die nach diesem Knoten als nächstes in der Eingabe auftreten können.
- Auch die Berechnung der follow-Menge ist lokal auf die Klassen verteilt:



  - `Some`, `Alt`, `Opt` und `Many` geben ihre follow-Menge an den Unterknoten weiter. `Some` und `Many` fügen ihren lookahead zur follow-Menge des Unterknotens hinzu.
  - `Seq` reicht seine follow-Menge an das letzte Element weiter. Ist die leere Eingabe in dem lookahead des Elements, wird die Vereinigung von follow und lookahead an das nächstletzte Element überreicht
  - `Id` reicht follow an die zugehörige `Rule`, und diese reicht die Menge an das Objekt der rechten Seite weiter.
- Die Berechnung startet mit dem EOF-Symbol für die rechte Seite der Start-Regel.
- Wiederholung der Berechnung bis alle follow-Mengen sich nicht mehr ändern.
- Der Algorithmus muß terminieren, da es nur eine endliche Anzahl Terminals und Regeln gibt.
- Die Verteilung der Berechnung zerlegt diese in kleine und einfache Teilprobleme. Der Algorithmus ist so leicht zu verstehen.

Mehrdeutige Grammatik

- Parsierung: rekursiver Abstieg, LL(1), nicht alle Grammatiken sind LL(1):

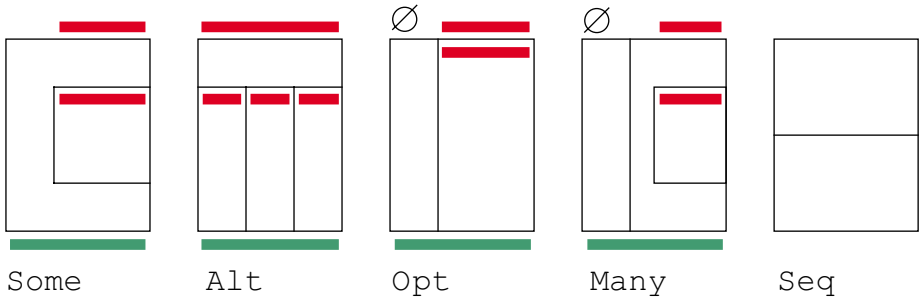
```
bits : { 0 | 1 } ( 0 | 1 )
```

- Der Baum und die lookahead - und follow-Mengen:

```
                lookahead   follow
seq             0 1         EOF
  some          0 1         0 1
    alt         0 1         0 1
      lit 0     0           0 1
      lit 1     1           0 1
  alt           0 1         EOF
    lit 0       0           EOF
    lit 1       1           EOF
```

Prüfen einer Grammatik

- Auch das Prüfen der Grammatik auf LL(1) verteilt sich lokal auf die Klassen:



  - `Some`, `Opt`, `Many`: Die lookahead- und follow-Menge müssen/sollten disjunkt sein (shift/reduce-Konflikt).
  - `Alt`: Die lookahead-Mengen der Alternativen müssen paarweise disjunkt sein. Enthält der lookahead die leere Eingabe, müssen follow und lookahead disjunkt sein.
  - `Seq` ruft lediglich bei allen Elementen die Prüfmethode auf.
- Weiterhin wird geprüft, daß jede Regel verbunden ist und daß keine unendliche Rekursion auftritt:

```
x: "x" x;
```

Beispiel arithmetische Ausdrücke

- Die Grammatik `arith.ebnf`:

```
expr    : [{ [  sum ]  ";" }];
sum     : product [{  ( "+" | "-") product }];
product : term [{  ( "*" | "/" ) term }];
term    : NUMBER | "(" sum ")";
```

- Von der Grammatik zum Parser:
  - Von Hand:

```
arithParser = new oops.parser.Parser(
       new oops.parser.Rule[] {
         new oops.parser.Rule( // expr : [{ [  sum ]  ";" }];
           new oops.parser.Id("expr"),                    // lhs
           new oops.parser.Many(                          // rhs
             new oops.parser.Seq(new oops.parser.Node[] {
               new oops.parser.Opt(new oops.parser.Id("sum")),
               new oops.parser.Lit(";")
             })
           )), ... };
```

  - Mit Hilfe eines Parsergenerators:

```
$ make arith.ser
java -classpath ..:../jars/lolo.jar:../jars/oops.jar \
> -Doops.oops.actionType=none oops.EBNF arith.ebnf > arith.ser
This is oops version 1.10
Copyright Axel-Tobias Schreiner (axel@informatik.uni-osnabrueck.de)
and Bernd Kuehl (bernd@informatik.uni-osnabrueck.de).
Using ebnf parser generator generated by oops on Wed Dec 12 10:05:19 CET 2001
$ ls -l arith.ser
-rw-r--r--  1 bernd  staff  2978 Jan 29 20:59 arith.ser
```

- Der Parser aus Objekten:

```
$ make dump
java -classpath ..:../jars/lolo.jar:../jars/oops.jar oops.tools.Dump arith.ser
oops.parser.Parser
   oops.parser.Rule expr : [{  (  [  sum ]  ";" ) }]  .
      oops.parser.Many
         oops.parser.Seq
            oops.parser.Opt
               oops.parser.Id sum
            oops.parser.Lit ";"
   ...
```

- Parsierung:

```
$  make test
java -classpath ..:../jars/lolo.jar:../jars/oops.jar \
> oops.RunParser arith.ser parseArith.ArithScanner
2-3+4;
2;3-4;
^D
$
```

- Die Scanner-Schnittstelle

Fazit

- Beliebige EBNF-Grammatik sind als Baum von Objekten repräsentierbar.
- Der Baum als Repräsentation einer Grammatik ist serialisierbar.
- *divide & conquer* für verschiedenste Algorithmen, lokal einfache Teilalgorithmen.
- Einige Klasse sind aufgrund von Vererbung einfach zu implementieren.
- Mächtiger Ansatz: Auffinden unendlicher Rekursionen in einer Grammatik.
- Idealer Ansatz für die Lehre.

Thesen

- Wiederverwendung.
- Erweiterbarkeit/Vererbung.
- Die objekt-orientierte Programmierung ist ein Automatismus für *divide & conquer.*

7 Aktionen

- Parser als *black box* bzw. als Kontrollstruktur.
- Unterklassen erweitern Parser-Klassen um Aktionen zu erkannten Teilen der Grammatik (Template-Entwurfsmuster).
- Verschiedenste Arten von Aktionsmustern sind denkbar.

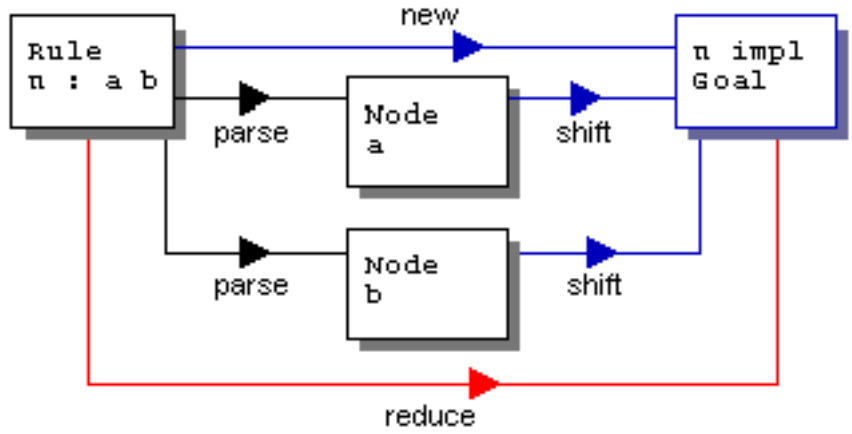
Ablaufverfolgung

- Erster Test der Idee.
- `Parser`, `Rule`, `Lit` und `Token` als Unterklassen der gleichnamigen Klassen im Paket `oops.parser.trace`.
- Beispiel arithmetische Ausdrücke.

Goal

- Das Goal-Interface:

```
public interface Goal {
    public void shift(Goal sender, Object value);
    public void shift(Token sender, Object value);
    public void shift(Lit sender, Object value);
    public Object reduce();
    public void error();
}
```



- `Parser`, `Rule`, `Lit` und `Token` als Unterklassen der gleichnamigen Klassen im Paket `oops.parser.goal`.
- Factories (Factory-Entwurfsmuster) erzeugen zur Laufzeit pro aktiver Regeln eine `Goal`-Instanz. oops verwendet ein doppeltes Factory-Muster:

```
public interface GoalMakerFactory {
    public GoalMaker goalMaker (String ruleName);
}

public interface GoalMaker  {
    public Goal goal ();
}
```

- Adapter-Klassen: `GoalAdapter`, `GoalDebugger` und `NopGoal`.
- Beispiel arithmetische Ausdrücke.

Reducer

- Analog zu `Goal`, aber einfacher:

```
public interface Reducer {
    public void error();
    public Object reduce(Object[] values);
}

public interface ReducerLit extends Reducer {}
```

- `Parser`, `Rule`, `Lit` und `Token` als Unterklassen der gleichnamigen Klassen im Paket `oops.parser.reducer`.
- Factories erzeugen zur Laufzeit pro aktiver Regeln eine `Reducer`- oder `ReducerLit`-Instanz. oops verwendet wieder ein doppeltes Factory-Muster:
- Adapter-Klassen.

Fazit

- Der objekt-orientierte Ansatz läßt Parser erweiterbar.
- Die `Goal`- oder die `Reducer`-Aktionsschnittstellen binden Aktionen auf eine für die Objekt-Orientierung natürliche Art und Weise.
- Trennung von Grammatik und Aktionen.
- Factories machen Parser für verschiedenartige Aktionen verwendbar bzw. sogar im Betrieb umschaltbar.
- Beliebige Arten von Aktionsmuster sind möglich.

Thesen

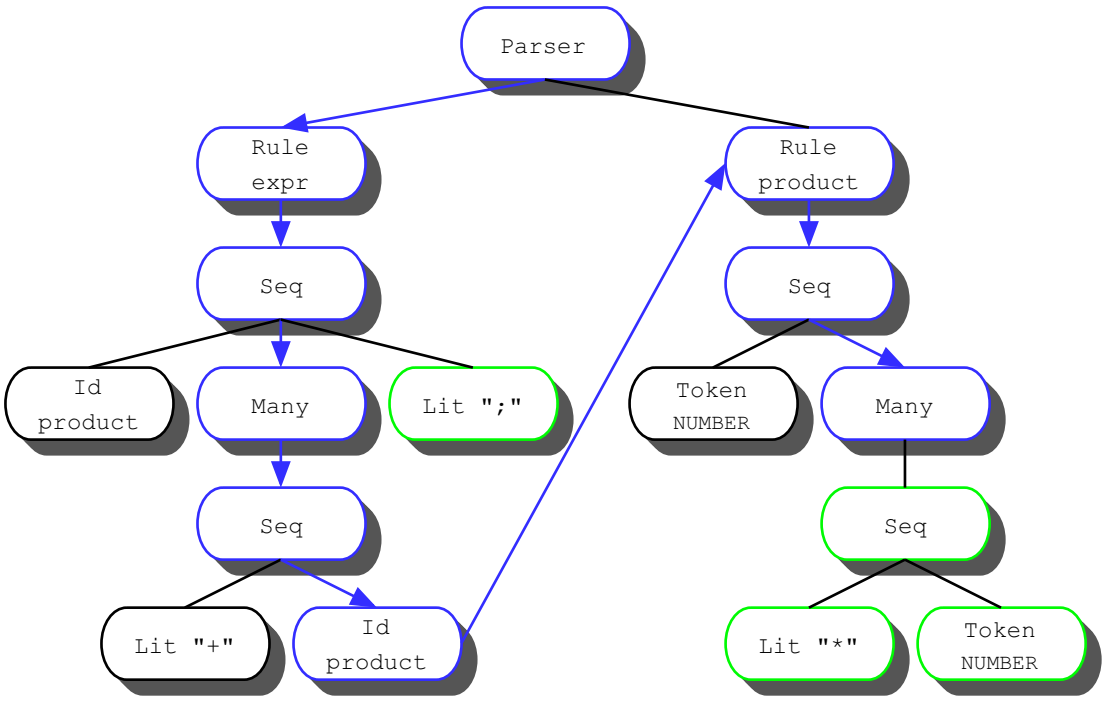
- Wiederverwendung.
- Entwurfsmuster.
- Erweiterbarkeit/Vererbung.

8 Fehlerbehandlung

- Fehlerbehandlung muß Parser möglichst gut erholen und sinnvolle Fehlermeldung liefern.
- Beispiel:

```
expr    : product {[  "+" product ]}  ";" ;
product : NUMBER {[  "*" NUMBER ]}  ;
```

- Situation nach dem erfolgreichen Erkennen von `2+3`:

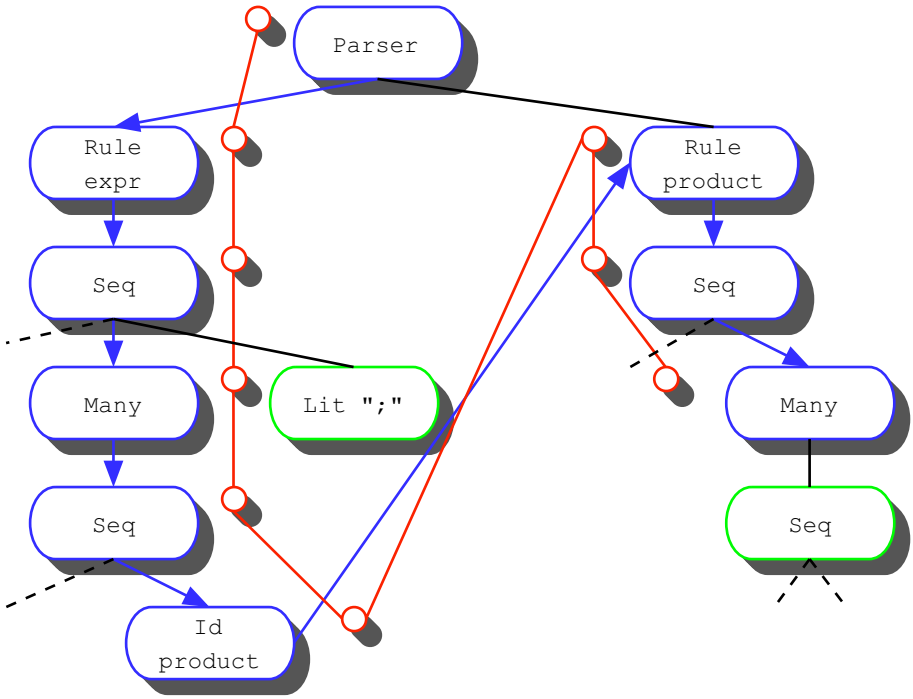


- Drei mögliche Vorgehensweisen zur Fehlerbehandlung:
  - Die Parrsierungs-Methode des momentanen Parser-Knotens beginnt mit der Erkennung der kompletten, repräsentierten Phrase von vorne (*retry*), ...
  - ... sie endet und delegiert die Verantwortung weiter nach außen in der Aufrufverschachtelung (*abort*), ...
  - ... oder sie versucht, an der aktuellen Stelle mit der Parsierung fortzufahren (*resume*), was das Verwerfen des illegalen Symbols beinhalten muß.
- `2+3*x9` `2+3*;` `2+3**9;`
- Fazit: lokale Entscheidungen in einem Knoten reichen nicht.

Activation

- Pro aktiver Parsierungsmethode bildet eine Kette von `Activation`-Objekten die Aufrufverschachtelung ab:

```
public abstract class Activation extends Throwable {
    protected final Activation caller;
    protected Activation (Activation caller) {  this.caller = caller; }
    protected Object up (Object info) throws Activation {
        return caller.up(info);
    }
    protected final void handle (Object info) throws Activation {
        this.info = info; throw this;
    }
    private Object info;
    protected final Object info () {  return info; }
    protected void expect(Set expect) {
        if (caller != null) caller.expect(expect);
    }
    ...
}
```



- Pro Klasse entscheiden lokale Unterklassen von `Activation` über eine Fehlerbehandlung.
- Sehr mächtige und automatische Fehlerbehandlung.
- Technik dient auch zum Berechnen der an einer Stelle erlaubten Symbole.

Thesen

- Wiederverwendung.
- *divide & conquer.*

9 Extending EBNF

Grammatik-Notationen

- EBNF:

```
start : ( "a" "b" ) | [ "c" ] { "d" } {[ "e" ]} ;
```

- Alternative Notation:

```
start : ( "a" "b" ) | "c"? "d"+ "e"* ;
```

- Alternative Notation:

```
public static oops.parser.Parser grammar = new oops.parser.Parser (
    new oops.parser.Rule(
        new oops.parser.Id("start"),
        new oops.parser.Alt(new oops.parser.Node[] { ... }),
            new oops.parser.Seq (new oops.parser.Node[] {
                new oops.parser.Opt (new oops.parser.Lit("c")), ...})
        }) // end of Alt, rhs of start
    ) // end of Rule start
);
```

- Baum ist unabängige Darstellung.
- Probleme XML: Attribute.

Extending EBNF

- Die Klasse `And`:

```
and   : "a1" & "a2"+ & "a3";  // a1 und a3 genau einmal, a2 mindestens einmal
```

- Die Klasse `Or`:

```
or    : "o1" | "o2"* | "o3";  // o1 und o3 maximal einmal, o2 beliebig oft
```

- Die Klasse `Xor`:

```
xor   : "x1" ^ "x2" ^ "x3";  #  xor-Verknuepfung von Elementen
```

- Klassen stammen direkt oder indirekt von Alt ab.

XEBNF-Notation der Parsergeneratoren

- Angelehnt an bekannte Notationen:

```
parser  : { rule };
rule    : ID ":" or ";";
or      : xor [ "*" "|" xor [ "*" ]] [{ "|" xor [ "*" ] }];
xor     : and [{ "^" and }];
and     : seq [ "+" "&" seq [ "+" ]] [{ "&" seq [ "+" ] }];
seq     : { ID | LIT | TOKEN | some | opt | "(" or ")" };
some    : "{ " or "}";  // one or more
opt     : "[ " or "]";  // zero or one
```

Ein Beispiel

- DTD:

```
<?xml version='1.0' encoding="ISO-8859-1" ?>
<!ELEMENT personen             (person*)>
<!ELEMENT person               (vorname,nachname)>
    <!ATTLIST person   persnr      ID #REQUIRED>
    <!ATTLIST person   chef        IDREF #IMPLIED>
    <!ATTLIST person   geschlecht  (männlich|weiblich) #REQUIRED>
<!ELEMENT vorname              (#PCDATA)>
<!ELEMENT nachname             (#PCDATA)>
```

- XEBNF-Grammatik:

```
personen   : {[ person ]} ;
person     : "person" ( persnr & [ chef ] & geschlecht & vorname &
              nachname ) ";" ;
persnr     : "persnr" NUMBER;
chef       : "chef" NUMBER;
geschlecht : "maennlich" ^ "weiblich";
vorname    : "vorname" WORD ;
nachname   : "nachname" WORD ;
```

- Personenliste:

```
person
    persnr 2345
    weiblich
    vorname  Berta
    nachname Boss ;

person
    nachname Untertan
    weiblich
    chef 2345
    persnr 123
    vorname  Ulla ;

person
    vorname  Knut
    chef 123
    persnr 0815
    nachname Knetch
    maennlich ;
```

- Das Beispiel ist in BNF oder EBNF kaum darstellbar.

Fazit

- oops ist leicht um Klassen wie `And`, `Or` und `Xor` erweiterbar.
- Andere Idee, `Iteration`:

```
start :  "x" <2,6>
```

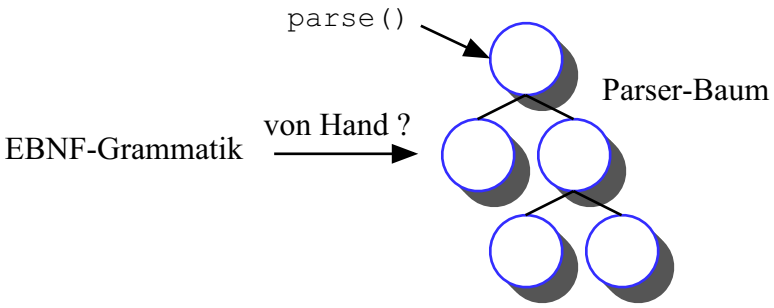
- Die neuen Klassen sind mit allen Aktionsmustern kompatibel.
- Die Grammatik-Notation ist austauschbar.

Thesen

- Wiederverwendung.
- Erweiterbarkeit/Vererbung.
- *divide & conquer.*

10 Parsergenerator

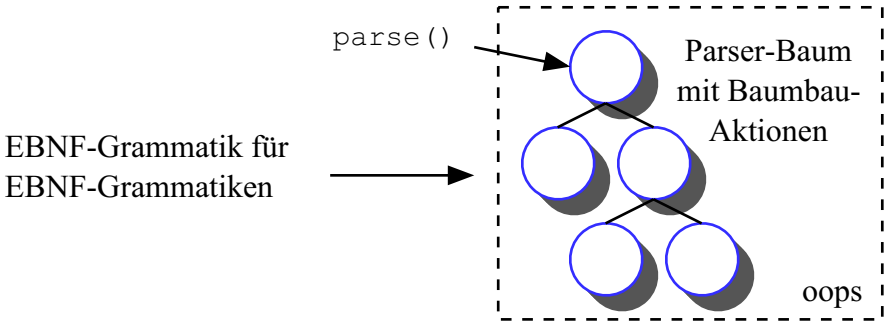
- oops ist neben einem Klassensystem auch ein Parsergenerator.
- Von der Grammatik zum Parser-Baum.



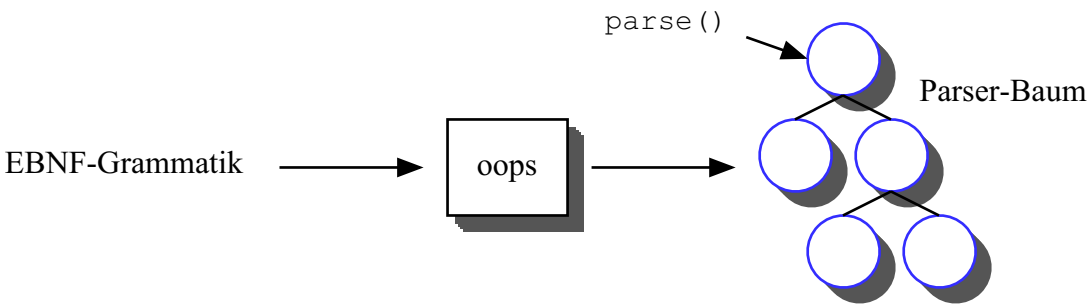
- Eine EBNF-Grammatik für EBNF:

```
parser  : { rule };             // a parser is one ore more rules.
rule    : ID ":" alt ";";       // rule is a name and a right hand side
alt     : seq [{ "|" seq }];    // alternatives
seq     : { ID | LIT | TOKEN | some | opt | "(" alt ")" }; // sequence
some    : "{ " alt "}";         // one or more
opt     : "[ " alt "]";         // zero or one
```

- Von einer Grammatik für EBNF zu oops.



- oops als Parsergenerator.



11 Fazit und Ausblick

Thesen

- Wiederverwendung von Klassen und Objekten.
  - Scanner-Framework, Erkenner-Klassen, Scanner.
  - Parser-Klassen, Parser.
- Entwurfsmuster.
  - Template-Entwurfsmuster.
  - Factory-Entwurfsmuster: oops-Parser und -Scanner.
- *divide & conquer.*
  - komplexe Algorithmen des Compilerbaus zerfallen in pro Klasse kleine Teilprobleme.
  - Erkenner-Klassen, Scanner in lolo.
  - lookahead und follow Berechnung.
  - Prüfen einer Grammatik.
  - Fehlerbehandlung im Parser.
- Erweiterbarkeit, Vererbung.
  - Erkenner-Klassen in lolo.
  - Parser-Klassen in oops.
  - Aktionsmuster in oops.
  - oops ist für weitere Klassen (Beispiel `And`, `Or`, `Xor`) offen.

Fazit

- Die Arbeit hat gezeigt, daß die typischen Vorteile der Objekt-Orientierung auch im Compilerbau wiederzuentdecken sind.
- Der Einsatz der Objekt-Orientierung bringt auch hier eine klarere Strukturierung und einen Zugewinn an Möglichkeiten, Flexibilität und Mächtigkeit.
- Alle hier vorgestellten Ideen und Ansätze sind allgemein verwendbar und sprachunabhängig.

Ausblick

- Thread-sichere lolo-Scanner und oops-Parser.
- Kombination von Erkenner-Instanzen des lolo-Systems.
- Erweiterung von oops-Parsern um neue Knoten-Klassen.
- Konfigurationsdateien für lolo-Scanner.

```
Skip
    lolo.scans.JavaWhitespace, lolo.scans.JavadocComment;

lolo.scans.Char,
lolo.scans.Char char 'c',
lolo.scans.Word String "hello world!",
lolo.scans.Set String "+-/*" boolean true;
```

- Automatischer Scanner-Anschluß an oops-Parser.

```
exprs   : [{ [  sum ]  ";" }];
sum     : product [{  ("+" | "-") product }];
product : term [{  ("*" | "/") term }];
term    : lolo.scans.Flt | "(" sum ")";
```

- Semantische Analyse, Interpreterbaum, Code-Generierung.
- LL(k)- und LR-Parser aus Objekten.

12 Links

[1] Homepage lolo: `http://www.inf.uos.de/bernd/lolo`

[2] Homepage oops: `http://www.inf.uos.de/bernd/oops`

[3] Text meiner Dissertation und Image der CD mit allen Beispielen und Quellen:
`http://www.inf.uos.de/bernd/thesis`

[4] Beispiele zu diesem Vortrag:
`http://www.inf.uos.de/bernd/talks/wuerzburg/codeWuerzburg.zip`

[5] Meine Homepage: `http://www.inf.uos.de/bernd`