

Ein Parser aus Objekten

(Objektorientierung im Compilerbau)

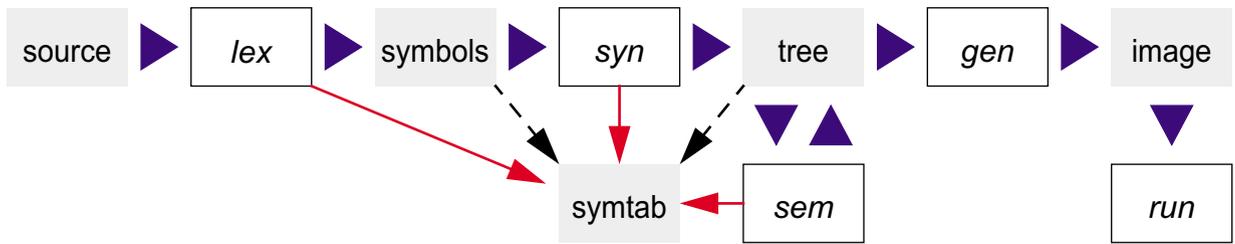
Bernd Kühl

University of Osnabrück, Germany

`bernd@informatik.uni-osnabrueck.de`

`http://www.inf.uos.de/bernd`

1 Compilerphasen



- Inhalt meiner Promotion:
Was bringt der Einsatz von Objektorientierung in den einzelnen Phasen.
- Inhalt des Vortrags:
Syntax-Analyse und Aktionen.

2 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.

oolex (object orientated lexer)

- Das Design von oolex ist strikt objekt-orientiert.
- Als Scanner stehen viele Objekte als Symbol-Erkennen im Wettbewerb.
- Ein Raum dieser Objekte verwaltet die Eingabe und liefert den Objekten Zeichen für Zeichen aus der Eingabe.
- Objekte, die mit dem Zeichen nichts anfangen, verlassen den Raum.
- Das letzte Objekt im Raum stellt als längstes das erkannte Symbol dar.
- Typische Symbol-Erkennen sind daher als Klassen in einer Bibliothek vorbereitet.
- Gewinnt ein Objekt eine Erkennungsrunde, kennt das Objekt optional ein anderes Objekt, welches die zugehörige Aktion implementiert.
- Alle Symbol-Erkennen- und Aktions-Klassen sind serialisierbar.

Fazit

- Der neue Ansatz ist objekt-orientiert und damit für objekt-orientierte Sprachen wie Java, Objective-C oder C++ intuitiver.
- 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.
- Die Entwicklungszeit für oolex-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.
- Da Objekte Zustände besitzen, können die Symbol-Erkennen mächtiger als endliche Automaten sein.
- Nachteil: Performance.

3 oops — Ein Parser aus Objekten

Objekte für Grammatiken

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

```
identifizier : letter | identifizier letter | identifizier number
```

- Beispiel EBNF

```
identifizier : 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 identifizier
    Seq
      Id letter
      Many
        Alt
          Id letter
          Id number
```

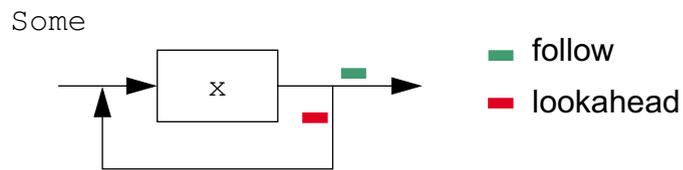
- `oops`: object orientated parser system.
- Fähigkeiten des Baums?
 - Baum repräsentiert Grammatik.
 - Parsierung: rekursiver Abstieg, `parse()`-Methode.
 - Prüfung der Grammatik.
- `oops`-Parser sind serialisierbar und damit wiederverwendbar.

NSD anstelle von Syntaxgraphen

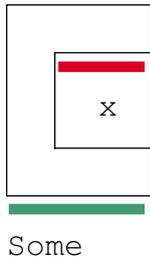
EBNF:

some : { x } ;

Syntaxgraph:

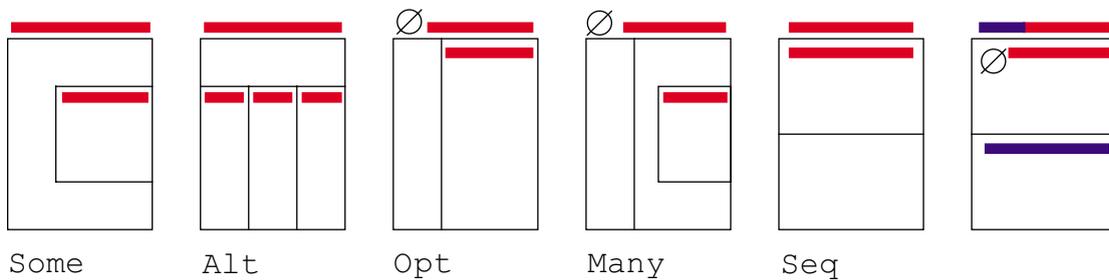


NSD:



Lookahead-Mengen

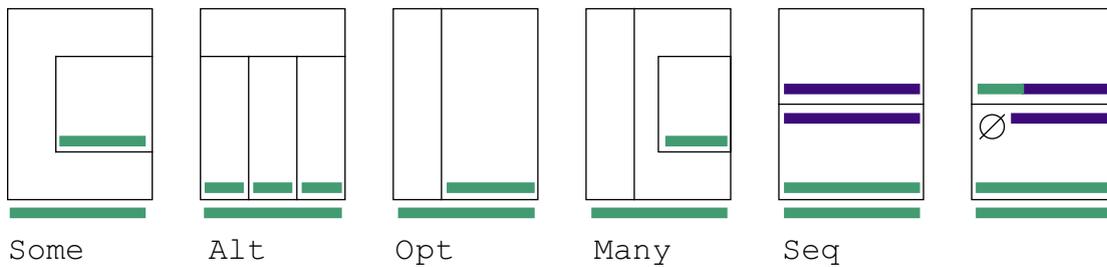
- Während der Parsierung treffen die Objekte Entscheidungen aufgrund von lookahead-Mengen.
- Lookahead: Die Menge aller Eingabesymbole, die an dieser Stelle als nächstes in der Eingabe auftreten können.
- Die Berechnung des lookahead ist lokal auf die Klassen verteilt:



- **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.
 - 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):

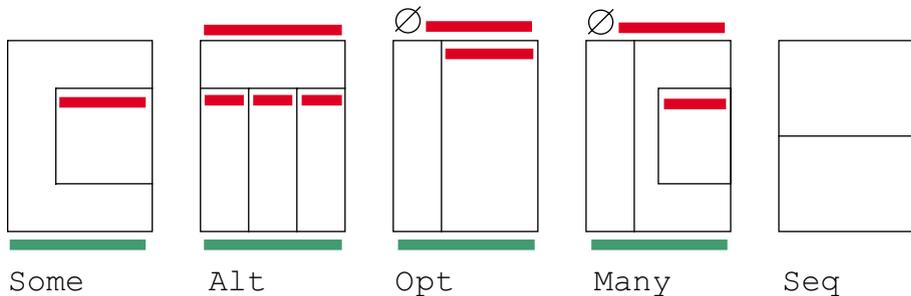
```
bits : { 0 | 1 } ( 0 | 1 )
```

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

	lookahead	follow
seq	0 1	[empty]
some	0 1	0 1
alt	0 1	0 1
lit 0	0	0 1
lit 1	1	0 1
alt	0 1	[empty]
lit 0	0	[empty]
lit 1	1	[empty]

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;
```

Scanner-Schnittstelle

- Ein oops-Parser benötigt eine Implementierung von `oops.scanner.Scanner` als Partner:

```
public interface Scanner {
    public void scan(SymTab symtab, Hashtable tokens);
    public boolean advance() throws IOException;
    public boolean atEnd();
    public Object symbol();
}
```

- Der Parser ruft einmal `scan()` beim `Scanner` zur Initialisierung auf.
- `advance()` rückt ein Symbol in der Eingabe vor. Die Methode hat `false` am Dateieende, sonst `true` zu liefern.
- `atEnd()` zeigt als `boolean` an, ob das Dateieende erreicht worden ist.
- Das Resultat von `symbol()` ist ein Objekt, welches das momentane Symbol repräsentiert.
- Bei Aufruf von `scan()` ist `symtab` mit den reservierten Wörtern vorgeladen und agiert damit als Screener:

```
public interface SymTab extends Serializable {
    public Handle get(String key);
    public Enumeration keys();

    public interface Handle {
        public boolean isEntered();
        public void enter(Object symbol, Object value);
        public void remove();
        public Object symbol();
        public Object value();
        public String key();
    }
}
```

- In `tokens` ist für alle Tokennamen als Schlüssel jeweils ein zugehöriges Objekt für `symbol()` zur Repräsentierung des Tokens abgelegt worden.

Beispiel: Ein Parser für arithmetische Ausdrücke

Die Grammatik

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

Der Code:

```
package parserByConstructors;

import ...

public class ArithParser {
    private static oops.parser.Parser arithParser;
    static {
        try {
            // build parser...
            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(";")
                            })
                        )
                    ),
                    new oops.parser.Rule( // sum : product [{ ( "+" | "-" ) product }];
                        new oops.parser.Id("sum"), // lhs
                        new oops.parser.Seq(new oops.parser.Node[] { // rhs
                            new oops.parser.Id("product"),
                            new oops.parser.Many(
                                new oops.parser.Seq(new oops.parser.Node[] {
                                    new oops.parser.Alt(new oops.parser.Node[] {
                                        new oops.parser.Lit("+"),
                                        new oops.parser.Lit("-")
                                    })
                                })
                            ),
                            new oops.parser.Id("product")
                        })
                    )
                }
            ),
            new oops.parser.Rule( // product : term [{ ( "*" | "/" ) term }];
                new oops.parser.Id("product"), // lhs
                new oops.parser.Seq(new oops.parser.Node[] { // rhs
                    new oops.parser.Id("term"),
                    new oops.parser.Many(
                        new oops.parser.Seq(new oops.parser.Node[] {
                            new oops.parser.Alt(new oops.parser.Node[] {
                                new oops.parser.Lit("*"),
                                new oops.parser.Lit("/")
                            })
                        })
                    ),
                    new oops.parser.Id("term")
                })
            )
        }
    }
}
```

```

        new oops.parser.Rule( // term : NUMBER | "(" sum ")";
            new oops.parser.Id("term"), // lhs
            new oops.parser.Alt(new oops.parser.Node[] { // rhs
                new oops.parser.Id("NUMBER"),
                new oops.parser.Seq(new oops.parser.Node[] {
                    new oops.parser.Lit("("),
                    new oops.parser.Id("sum"),
                    new oops.parser.Lit(")"),
                })
            })
        )
    }
};
// check parser...
arithParser.check();
} catch (oops.parser.ParserBuildException pe) {
    System.err.println(pe);
    System.exit(1);
} catch (oops.parser.CheckLL1Exception ce) {
    System.err.println(ce);
    System.exit(1);
}
}

public static void main(String args []) throws
oops.parser.ParseException, oops.opi.OpiException, IOException,
ClassNotFoundException {
    if (args.length == 0 ||
        (args[0].equals("-deserialize") && args.length < 3)) usage();
    int n = 0;
    if (args[0].equals("-deserialize")) {
        n += 2;
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(args[1]));
        arithParser = (oops.parser.Parser) in.readObject();
        in.close();
    }
    if (args[n].equals("parse")) {
        oops.scanner.Scanner scanner = new Scanner(
            new oops.opi.InputSource(System.in));
        arithParser.parse(scanner, (oops.scanner.TableFactory) null);
    } else if (args[n].equals("dump")) {
        oops.tools.Dump.dump(arithParser);
    } else if (args[n].equals("serialize")) {
        ObjectOutputStream out = new ObjectOutputStream(System.out);
        out.writeObject(arithParser);
        out.flush(); out.close();
    } else usage();
}
private static void usage() {
    System.err.println("usage: ArithParser [-deserialize file] "+
        "(parse | dump | serialize)");
    System.exit(1);
}
}
}

```

4 Vererbung — Aktionen

- Parser ist eine Art Kontrollstruktur.
- Parser als Kontrollstruktur löst Aktivität als Methodenaufruf aus.
- Trennung von Grammatik und Aktionen: Kapselung von Aktion in Objekten.
- Ideen:
 - Ein Objekt pro Parsierung (DefaultHandler).
 - Ein Objekt pro Regel (Listener).
 - Ein Objekt pro Aktivierung einer Regel (Goal, Reducer).
- Durch Spezialisierung/Vererbung können Aktionen eingebaut werden.

Scanner-Schnittstelle

- Erweiterung des Scanner-Interfaces:

```
public interface ValueScanner extends Scanner {  
    public Object value();  
}
```

- `value()` liefert ein Werte-Objekt oder `null` zum aktuellen Symbol.

Goal

oops.parser.goal-Klassen/Interface: u.a. Parser, Rule, Lit, Token und Goal.

Goal-Interface:

```
package oops.parser.goal;

import ...

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 ();
}
```

Die Goal-Instanzen werden pro Regel-Aktivierung über ein Factory-Muster erzeugt.

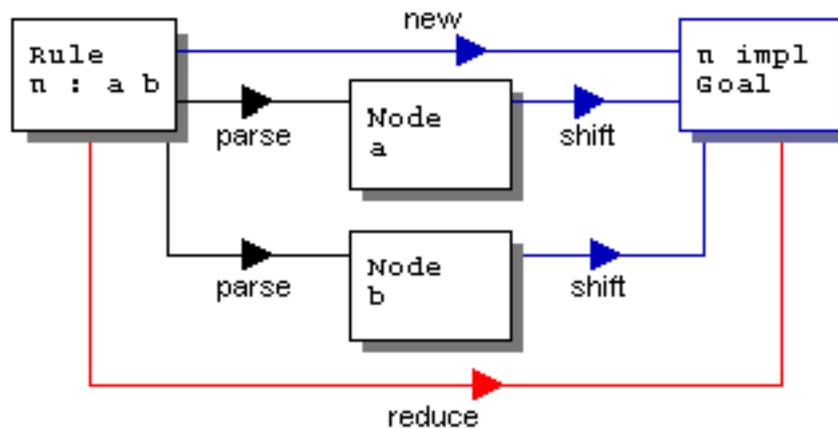
Es gibt Standard-Implementierungen für die Factories und Goal.

Die default Factory erzeugt Goal-Instanzen aufgrund von Regelnamen. Für

```
sum      : product [{ ( "+" | "-" ) product }];
```

wird zum Beispiel eine Instanz der Goal-Klasse `sum` erzeugt. Existiert keine Klasse mit dem Namen, wird ein `GoalAdapter` verwendet.

Für jede aktive Regel ist eine Goal-Instanz aktiv:



Arithmetische Ausdrücke per Goal

Die Grammatik:

```
expressions      : [{ [ sum ] ( ";" | "\n" ) }];
sum              : product [{ sum.add | sum.sub }];
sum.add         : "+" product;
sum.sub         : "-" product;
product         : term [{ product.mul | product.div }];
product.mul     : "*" term;
product.div     : "/" term;
term            : NUMBER | term.minus | "(" sum ")";
term.minus     : "-" term;
```

term.java:

```
public class term extends oops.parser.goal.GoalAdapter {
    public static class minus extends GoalAdapter {
        public Object reduce () {
            return new Double(- ((Number)result).doubleValue());
        }
    }
}
```

sum.java:

```
import oops.parser.Lit;
import oops.parser.goal.Goal;
import oops.parser.goal.GoalAdapter;

public class sum extends GoalAdapter {
    public void shift (Goal sender, Object value) {
        if (result == null) result = value;
        else result = ((eval) sender).eval((Number) result, (Number) value);
    }
    public abstract static class eval extends GoalAdapter {
        public abstract Object eval (Number left, Number right);
        public void shift (Lit sender, Object value) {}
    }
    public static class add extends eval {
        public Object eval (Number left, Number right) {
            return new Double(((Number) left).doubleValue() +
                ((Number) right).doubleValue());
        }
    }
    public static class sub extends eval {
        public Object eval (Number left, Number right) {
            return new Double(((Number) left).doubleValue() -
                ((Number) right).doubleValue());
        }
    }
}
```

- In der Regel bilden (und serialisieren) Goal-Instanzen einen Baum zur Repräsentierung der Eingabe.

Weitere Aktionsideen

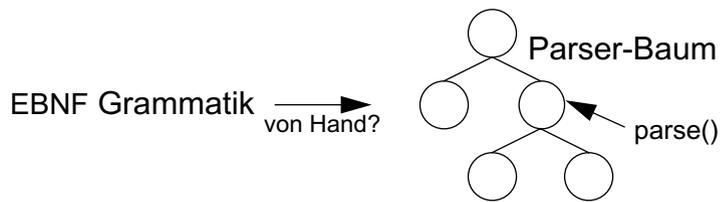
- `oops.parser.trace`: die Knoten des Parser-Baums schreiben während der Erkennung eines Programms eine Trace.
- `oops.parser.reducer`: analog zu `Goal`, aber wesentlich einfacher. Am Ende einer Regel wird durch Aufruf von `reduce()` in einem Argument-Array das Resultat von `reduce()` gegen Unterregeln, erkannte `Lit`- und erkannte `Token`-Instanzen übergeben.

```
package oops.parser.reducer;  
  
public interface Reducer {  
    public void error();  
    public Object reduce(Object[] values);  
}
```

- Listener- und Handler-Schnittstelle.
- Handler und Listener sind aufgrund der `Goal`-Factory per `Goal` sehr leicht zu realisieren.

5 oops als Parser-Generator

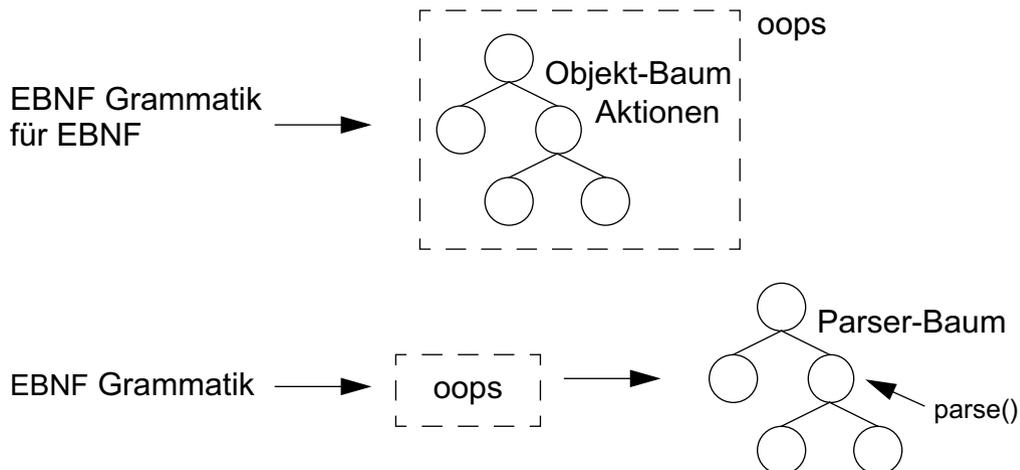
- oops als Parser:



- EBNF-Grammatik für EBNF:

```
parser : { rule }; // a parser is one or 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
```

- oops als Parser-Generator:



- Beispiel:

```
$ export CLASSPATH=../../jars/oops.jar:../../
$ java -Doops.oops.actionType=goal oops.Compile ... arith.ebnf > arith.ser
Using ebnf parser generator generated by oops on Mon Apr 16 12:49:34 CEST
2001
$
```

- Unterstützung für `oops.parse`, `oops.parse.trace`, `oops.parse.reduce` und `oops.parse.goal`.
- oops ist bootstrap-fähig.

6 Erweiterung — Extended EBNF

- Aktionen sind ein Beispiel für Spezialisierung durch Vererbung.
- Spezialisierung durch Erweiterung d.h. neuer Klassen des Objekt-Baums.
- HTML oder XML sind schon aufgrund der Attribute in [E]BNF kaum formulierbar.
- Mögliche neue Grammatik-Syntax:

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

|, ^ und & in Anlehnung an bekannte Operatoren aus Programmiersprachen

- inklusives oder, |: Alle Alternativen dürfen (müssen aber nicht) in beliebiger Reihenfolge maximal einmal vorkommen.
- exklusives oder, ^: Nur eine Alternative wird erkannt.
- and, &: Jede Alternative muß (in beliebiger Reihenfolge) vorkommen.
- HTML, XML: Attribute per | aufzählen.
- Weitere mögliche neue Grammatik-Syntax:

```
or     : xor [ { "|" xor [ "*" ] } ];
xor    : and [ { "^" and } ];
and    : seq [ { "&" seq [ "+" ] } ];
```

* und + erlauben wiederholtes Auftreten.

- Spezialisierung durch neue Klassen: `oops.parser.XOr`, `oops.parser.Or`, `oops.parser.And`.
- Neue Klassen erben Teile (Speicherung von Unterknoten, Prüfalgorithmen, Mengen-Berechnung, ...) von der Oberklasse `oops.parser.Alt`.
- Die jeweils neue Funktionalität ist mit wenig Code implementiert.

7 Fazit

- Grammatiken sind als Objektbaum repräsentierbar.
- oops besteht aus den Klassen zur Repräsentierung eines Parsers und aus dem Parsergenerator.
- Objekte kapseln Aktionen und sind im laufenden austauschbar.
- Die Serialisierung von Parsern macht diese wiederverwendbar.
- oops trennt Grammatik und Aktionen.
- Die Aufteilung der Mengen-Berechnung, der Prüfung einer Grammatik oder des Parsierens auf Instanz von Klassen lokalisiert die Probleme auf kleine Teile und macht die Algorithmen damit entdeckbar.
- Vererbung oder Erweiterung durch neue Klassen macht oops mächtiger: Aktionen bzw. Extended EBNF.
- Automatische error recovery machbar.

8 Literatur und Links

- [1] **Homepage oops:** <http://www.inf.uos.de/bernd/oops>
- [2] **Homepage oolex:** <http://www.inf.uos.de/bernd/oolex>
- [3] **Dieser Vortrag als PDF:**
<http://www.inf.uos.de/bernd/talks/berlin01/talk.pdf>
- [4] **Dieser Vortrag als HTML:**
<http://www.inf.uos.de/bernd/talks/berlin01/html/talk.html>
- [5] **Die Beispiele des Vortrags:**
<http://www.inf.uos.de/bernd/talks/berlin01/code.zip>
- [6] **Vorlesung Compilerbau mit Java, WS 1998/99, Prof. Dr. Axel-Tobias Schreiner:**
<http://www.vorlesungen.uni-osnabrueck.de/informatik/compilerbau98/>
- [7] **Bernd Kühl, Prof. Dr. Axel-Tobias Schreiner, An object-oriented LL(1) parser generator, SIGPLAN Notices, ACM December 2000**
- [8] **Bernd Kühl, oolex — Lexikalische Analyse mit Objekten, IFC Seminar, Sommersemester 2000, Universität Osnabrück.**
<http://www.inf.uos.de/bernd/talks/oolex-ifc00/html/talk.html>
- [9] **Prof. Dr. Axel-Tobias Schreiner, Bernd Kühl, Exceptions einmal anders, iX, Heft 11/99, Verlag Heinz Heise.**
<http://www.ix.de/ix/artikel/1999/11/194/>