oolex
Lexikalische Analyse mit Objekten

Bernd Kühl, bekuehl@uos.de
Fachbereich Mathematik-Informatik
Universität Osnabrück
IFC Sommersemester 2000

Inhalt

1 Einführung

1.1 Was ist lexikalische Analyse?

• Lexikalische Analyse im Compilerbau: "Der Teil des Compilers, der Eingabesymbole aus der unstrukturierten Folge von Zeichen konstruiert, die der Compiler als eigentliche erhält."

• Ein Beispiel:

```
$ java Scanner
var1 = 2.3;
        identifier: var1
        character: =
        floating point: 2.3
        character: ;
while(23>0)
        keyword: while
        character: (
        integer: 23
        character: >
        integer: 0
        character: )
# hello world!
        #-comment:  hello world!
```

• Programme oder Funktionen zur lexikalischen Analyse werden *Scanner* genannt.

• Scanner zerlegen einen Strom von Eingabezeichen in sogenannte Symbole oder auch Tokens.

• Scanner verbrauchen einen großen Teil des Rechenaufwands eines Compilers.

• Typischerweise finden Scanner im Compilerbau Anwendung, können aber auch als Filter eingesetzt werden.

• Beispiele von typischen Symbolen im Compilerbau:

+ Oktal-, Hexadezimal, Integer- und Gleitkommazahlen

+ Identifier

+ Zitierter Text mit Ersatzdarstellung

+ Operatoren

+ Kommentare verschiedenster Art

+ Zwischenraum

+ einzelne Zeichen

1.2 Was ist oolex?

- *oolex* ist ...

  + ... ein System zur lexikalischen Analyse.

  + ... strikt objekt-orientiert entworfen.

- Der Vortrag soll zeigen:

  + Die herkömmlichen Techniken haben Schwächen.

  + *oolex* hat durch das objekt-orientierte Design diese Schwächen nicht und bietet weiterhin viele Vorteile.

2 Herkömmliche Techniken zur lexikalischen Analyse

• Die herkömmlichen Techniken zur lexikalischen Analyse sollen kurz am gleichen Beispiel vorgestellt werden.

• Die Beispiel-Scanner sollen vier verschiedene Symbole erkennen:

+ Identifier: Folgen von mindestens einem Buchstaben.

+ Integer-Zahl: Folgen von mindestens einer Zahl.

+ Zwischenraum: Folgen von mindestens einem Zeichen aus der Menge `' '`, `'\t'`, `'\r'` und `'\n'`

+ Alle andere Zeichen sollen als einzelne Zeichen erkannt werden.

2.1 Handgeschriebene Scanner

• Einfache Scanner können als endliche Automaten (noch) von Hand geschrieben werden

• Vorteile:

+ Performance

• Nachteile:

+ mühsam zu implementieren (Beispiel: Gleitkommazahlen)

+ fehleranfällig

+ schwer erweiterbar

+ Performance

+ Backtracking

• Das Beispiel: `simple_example.byhand.Scanner`

• Der Scanner in Ausführung:

```
$ make test
CLASSPATH=../.. java simple_example.byhand.Scanner
while(variable<23)
        identifier: while
        character: (
        identifier: variable
        character: <
        integer: 23
        character: )
```

2.2 Fertige Scanner[klassen]

• Es gibt fertige Scanner[klassen]. Diese sind aber für ein festes Einsatzgebiet konzipiert.

• Instanzen der Klasse `java.io.StreamTokenizer` erkennen als Symbole:

+ Identifier

+ Zahlen

+ Quoted Strings

+ (verschiedene Kommentare)

+ Zeilenende

+ einzelne Zeichen

+ (Zwischenraum)

• Das genaue Verhalten kann nach der Konstruktion eines Objekts durch (viele) Methoden eingestellt werden.

• Vorteile:

+ Für Aufgaben auf dem konzipierten Einsatzgebiet sehr gut und schnell zu gebrauchen.

+ Getestet und damit (hoffentlich) fehlerfrei

• Nachteile:

+ Nicht allgemein verwendbar

+ `StreamTokenizer`:

- Die Dokumentation ist eher schlecht. Aus einer Diskussion mit Herr Schreiner: "Man muß die Dokumentation kreativ lesen....".

- Zwischenraum wird immer ignoriert und ist als Symbol nicht greifbar.

- Kein Unicode.

- Integer- und Gleitkommazahlen können nicht gleichzeitig und Exponentialdarstellung für Gleikommazahlen kann nicht erkannt werden.

- Sobald nach Wörtern gescannt wird, werden Gleitkommazahlen als Zahlen erkannt und gehören Zahlen zu den legalen Zeichen von Wörtern...

• Das Beispiel `simple_example.streamtokenizer.Scanner` benutzt eine Instanz der Klasse `StreamTokenizer` als Scanner:

• Die Ausführung:

```
$ make test
CLASSPATH=../.. java simple_example.streamtokenizer.Scanner
abc12de
        identifier: abc12de
23.4
        integer: 23.4
```

2.3 Scannergeneratoren lex/flex/JLex

lex und flex

• *lex* und *flex* akzeptieren jeweils eine Tabelle von egrep-artigen Mustern und C-Anweisungen und konstruieren eine C-Funktion `yylex()` zur Texterkennung oder -verarbeitung.

• Wenn ein Stück der Eingabe einem Muster genügt, wird die zugehörige C-Anweisung ausgeführt.

• Nicht erkannte Teile der Eingabe werden kopiert.

• Das Beispiel als *flex*-Scanner:

```
%{
#include <stdio.h>
%}

%%

[ \n\t\r]    {}
[a-zA-Z]+    { printf("identifier: %s\n", yytext); }
[0-9]+       { printf("integer: %s\n", yytext); }
.            { printf("character: %c\n", *yytext); }

%%
```

• Übersetzung

```
$ lex -l -t num.l > num.c
$ cc -c num.c -o num.o
$ cc -o num num.o -lfl
```

• *[f]lex* ist kein Scanner, sondern ein Scanner-Generator. Eine Übersetzung besteht daher immer aus zwei Schritten.

JLex

• *JLex* ist eine Neuimplementierung von *[f]lex* in Java und folglich auf vielen Plattformen sofort verfügbar.

• Die Syntax der Tabelle unterscheidet sich wenig, aber im Vorspann sind sehr viele(!) Java-ismen zu beachten.

• *JLex*-Scanner kommen ohne Laufzeit-Support aus.

• Das Beispiel als JLex-Scanner: Scanner.lex

Fazit

• Vorteile:

+ Das Verfahren ist durch die Verwendung von regulären Ausdrücken sehr mächtig, nicht nur für Scanner, sondern auch für Filter, die zeilenübergreifend arbeiten sollen.

+ Im ersten Schritt der Übersetzung wird aus den Mustern ein deterministischer Automat

berechnet. Der erzeugte Scanner ist daher schnell.

+ Die Technik ist sehr verbreitet.

• Nachteile:

+ Der Entwicklungszyklus hat zwei Schritte: Tabellen erzeugen und übersetzen.

+ Soll der Scanner erweitert oder geändert werden, müssen beide Schritte erneut ausgeführt werden.

+ Reguläre Ausdrücke sind für Anfänger schwer zu erlernen und sind 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.

2.4 Fazit der herkömmlichen Techniken

• Alle drei vorgestellten gängigen Techniken bieten ernsthafte Nachteile.

• Für große Scanner, wie zum Beispiel im Compilerbau, kommen bislang nur Scanner-Generatoren in Betracht.

• Diese Nachteile der gängigen Techniken waren Motivation für einen neuen und einfacher zu verwendenen Ansatz: eben *oolex*.

3
oolex

3.1 Die Idee von oolex

• Das Design von *oolex* ist strikt objekt-orientiert.

• Als Scanner stehen viele Objekte als Symbol-Erkenner im Wettbewerb.

• Ein Raum dieser Objekte verwaltet die Eingabe und wirft den Objekten Zeichen für Zeichen aus der Eingabe vor.

• Objekte, die mit dem Zeichen nichts anfangen, verlassen den Raum.

• Das letzte Objekt im Raum stellt als längstes das erkannte Symbol dar.

• Erkennen mehrere Symbol-Erkenner eine gleich lange Zeichenfolge, gewinnt das Objekt, welches als erstes den Raum betreten hat.

• Der Ansatz soll für den Anwender einfach zu handhaben sein. Typische Symbol-Erkenner sind daher als Klassen in einer Bibliothek vorbereitet.

• In der Bibliothek gibt es u.a. Klassen für:

  + verschiedene Arten von Zahlen

  + Zeichenfolgen

  + Zeichenklassen

  + verschiedene Kommentare

  + Zwischenraum

  + einzelne Zeichen

  + zitierten Text mit Ersatzdarstellungen

• Der Anwender kann für den Wettbewerb Instanzen der Klassen im Raum versammeln und muß sich keine Gedanken über die Funktionsweise der Erkennung der einzelnen Objekte machen.

• Im Extremfall kann eine Klasse als Symbol-Erkenner auch einen regulären Ausdruck repräsentieren.

• Da Objekte Zustände besitzen, können die Symbol-Erkenner mächtiger als endliche Automaten sein. Die Erkennung von Symbolen wie geschachtelte C-Kommentare stellt damit kein Problem dar.

• Durch dem Raum wird aus den vielen Objekten ein großer nichtdeterministischer Automat.

• Gewinnt ein Objekt eine Erkennungsrunde, kennt das Objekt optional ein anderes Objekt, welches die zugehörige Aktion implementiert.

3.2 Die Implementierung von oolex

Die Klasse Input

```
package oolex;

public class Input {
        public Input (Reader in, int front, int rear) { ... }
        public Input (Reader in [], int front, int rear) { ... }
        ...
        protected char buffer [];
        ...
        public static class EmptyTokenException extends Exception { ... }
        public static class IllegalCharacterException extends Exception { ...}
        ...
        public boolean token (Scan entry) throws IOException,
                        EmptyTokenException, IllegalCharacterException { ... }
        public String toString() { ... }
        public int line() { ... }
}
```

• Die Klasse `Input` repräsentiert die Eingabe-Seite des Systems und stellt damit einen Teil der Idee des Raums von Objekten als Scanner dar.

• Ein `Input`-Objekt liest viele Zeichen aus der Eingabe in einen Buffer und kennt die Position des nächsten Zeichens im Buffer.

• Überschreitet die Position im Buffer `front`, wird der Buffer auf der linken Seite kompaktiert. Sind alle Zeichen im Buffer verarbeitet, wird versucht von `in` mindestens `rear` Zeichen in den Buffer einzulesen.

• Ein Aufruf von `token()` startet eine neue Runde zum Erkennen des durch `entry` repräsentierten Symbols.

• Innerhalb von `token()` wird `entry` Zeichen für Zeichen aus der Eingabe vorgeworfen. Das Objekt signalisiert irgendwann, daß es das Symbol erkannt hat oder auch nicht. `token()` liefert `true`, wenn ein Symbol erkannt wurde, und `false` am Dateiende.

• Hatte die Erkennung keinen Erfolg, paßte die Zeichenfolge nicht zum Symbol-Erkenner. Es liegt eine illegale Eingabe vor und `token()` wirft eine `IllegalCharacterException`.

• `Input`-Objekte arbeiten ausschließlich auf `Reader`-Basis. Damit unterstützt das System ganz natürlich Unicode als Eingabe.

3.3 Die Klasse Scan

```
public abstract class Scan implements Serializable {
        protected transient int length;
        public int getLength() { return length; }

        protected transient boolean accepted;
        public boolean getAccepted() { return accepted; }

        public abstract Scan init (Scan scan);
        public abstract boolean next (int ch);

        public static Scan newRe (String pattern) { ... }
        ...
}
```

• Alle Klassen zur Erkennung eines Symbols müssen direkt oder indirekt von `Scan` abstammen.

• Die Eingabe schickt aus `token()` heraus einem `Scan`-Objekt durch viele `next()`-Aufrufe jeweils ein Zeichen. Es antwortet mit `true`, wenn es noch weitere Zeichen bekommen möchte, sonst mit `false`.

• `next()` liefert eigentlich drei Resultate:

  + ob bei diesem Aufruf etwas erkannt wurde.

  + die Anzahl bisher als korrekt erkannter Zeichen oder -1.

  + sowie ob durch weitere Zeichen eine längere Folge erkannt werden kann.

• Die beiden ersten Resultate werden mit den Variablen `length` und `accepted` beschrieben.

• Vor einer neuen Runde wird jedem beteiligtem Objekt `init()` geschickt. `init()` muß die Instanz seiner Klasse liefern, die eine Erkennung vornehmen soll. Durch Aufruf von `init()` bei der Oberklasse wird das komplette Resultatobjekt initialisiert.

3.4 Das Action Interface

```
public abstract class Scan implements Serializable {
        ...
        private Action action;

        protected void action (char buf[], int off, int len) { ... }
        public Scan setAction (Action action) { ... }
        public Action getAction () { ... }

        public interface Action extends Serializable {
                public void action(Scan sender, char buf[], int off, int len);
        }

        public Object genValue(char buf [], int off, int len) { ... }
}
```

• Die `Action`-Schnittstelle dient der Programmierung von Benutzeraktionen zu einem erfolgreich erkannten Symbol.

• Einer `Scan`-Instanz wird mit `setAction()` ein `Action`-Objekt gesetzt, welches mit `getAction()` auch erfragt werden kann.

• Aus der Methode `token()` der Klasse `Input` wird nach einer erfolgreichen Erkennungsrunde bei dem `Scan`-Objekt die Methode `action()` mit dem erkannten Text aufgerufen.

• Die Klasse `Scan` implementiert `action()` für Unterklassen vor: Wurde mit `setAction()` ein `Action`-Objekt gesetzt, wird bei dem Objekt `action()` mit dem erkannten Text aufgerufen. Ist keine `Action`-Instanz gesetzt, passiert nichts.

• *oolex*-Scanner eignen sich sowohl zum Anschluß an Parser als auch für Filter. Lediglich die Implementierung der `Action`-Instanzen ist entscheidend.

• In `action()` kann durch Aufruf von `genValue()` beim Absender ein Objekt für die erkannten Zeichen angefordert werden.

3.5 Die Klasse Alt

```
public static class Alt extends Scan {
        public Alt (Scan[] entry) { ... }
        protected final Scan[] entry;    // match one (or first) of these
        protected transient int winner; // index of first of max length

        public Scan init (Scan scan) { ... }
        public boolean next(int ch) { ... }
        public void action (char buf[], int off, int len) { ... }
}
```

• Die Klasse `Alt` managt einen Wettbewerb von alternativen Symbol-Erkennern um das längste (davon erstes) Symbol.

• Sie stellt damit den noch fehlenden Teil der Idee des Raums von Objekten dar.

• `Alt` stammt von `Scan` ab und kann damit als Parameter zu `token()` einer `Input`-Instanz benutzt werden.

• Nach einer erfolgreichen Runde zeigt `winner` als Index in `entry` auf den Gewinner des Wettbewerbs.

• Hat ein `Alt` eine `Action` gesetzt, wird dessen `action()`-Methode aufgerufen; sonst wird beim Gewinner `action()` aufgerufen.

3.6 Beispiele von Scan-Klassen

Im folgenden werden exemplarisch einige `Scan`-Klassen vorgestellt.

Set

```
public static class Set extends Scan {
        public Set (String set, boolean inside) {
                if (set == null)
                        throw new IllegalArgumentException("null set");
                this.set = set; this.inside = inside;
        }
        protected final String set;
        protected final boolean inside;

        public Scan init (Scan scan) {
                Scan result = scan == null ? new Set(set, inside) : scan;
                result.setAction(this.getAction());
                result.length = -1; return result;
        }

        public boolean next (int ch) {
                length = (set.indexOf((char)ch) >= 0) == inside ? 1 : -1;
                accepted = length == 1; return false;
        }
}
```

• Objekte der Klasse `Set` dienen zum Erkennen eines Zeichens aus oder gerade nicht aus einer Zeichenmenge.

• In `init()` wird `accepted` nicht extra der Wert `false` zugewiesen, da dies bereits der default Wert ist. Da ein `Set` nur ein Zeichen akzeptieren kann, endet `next()` sofort mit `false`.

Char

```
public class Char extends Set {
        public Char(char ch) { super(ch+"", true); } // match ch
        public Char() { super("", false); }          // match any
}
```

• Bereits bei dieser einfachen Klasse macht sich Vererbung und damit Spezializierung bezahlt. Ein `Char`-Instanz erkennt ein beliebiges oder ein bestimmtes Zeichen.

SetMN

```
public static class SetMN extends Scan {
        public SetMN (String set, boolean inside, int m, int n) { ... }

        protected final String set;
        protected final boolean inside;
        protected final int m, n;

        public Scan init (Scan scan) { ... }
        public boolean next (int ch) { ... }
        ...
}
```

• Die Klasse `SetMN` entspricht von der Idee her der Klasse `Set`. Allerdings werden zwischen `m` und `n` Zeichen aus (oder nicht aus) der Menge erkannt.

• Da `m` auch 0 sein darf, ist dies ein Beispiel für einen möglichen optionalen Erkenner, d.h. einem, der auch eine leere Eingabe akzeptiert.

• In `token()` einer `Input`-Instanz ist die Erkennung einer leeren Eingabe ein Probleme. Eine `EmptyTokenException` wird ausgelöst.

• Der Entwickler kann die `EmptyTokenException` abfangen und entsprechend reagieren.

Int

```
public class Int extends SetMN {
        public Int() { super("0123456789", true, 1, Integer.MAX_VALUE); }

        public Object genValue(char buf [], int off, int len) {
                return new Long(new String(buf, off, len));
        }

        public Scan init (Scan scan) {
                Int result = scan == null ? new Int() : (Int) scan;
                super.init(result); return result;
        }
}
```

• Die Klasse `Int` erkennt Integerzahlen ohne Vorzeichen. `Int` stammt von `SetMN` ab: Eine Instanz erkennt mindestens ein Zeichen aus der Menge der Zahlen.

• Hier wird der Grund für die `init()`-Kette klar: Würde `Int` die Methode nicht implementieren, nähme ein `SetMN`-Objekt an der Erkennung teil. Dieses liefert aber ein `String` als Resultat von `genValue()`.

3.7 Reguläre Ausdrücke

• Mit zwei bzw. drei weiteren `Scan`-Containerklassen im Stil von `Alt` kann man die üblichen regulären Ausdrücke auf `Scan`-Objekten abbilden.

• `Seq` kapselt als Sequenz im Gegensatz zu `Alt` keine Alternativen, sondern eine Folge von zwei `Scan`-Instanzen (`first` und `second`).

• Eine längere Folge kann durch Kaskadierung von `Seq`-Objekten erreicht werden.

• `Loop`-Objekte erkennen Wiederholungen einer `Scan`-Instanz. Die Anzahl der Wiederholungen wird durch zwei Zahlen angegeben.

• Ein optionaler Erkenner kann durch ein `Loop` mit 0 bis 1 Wiederholungen repräsentiert werden. Alternativ kann die einfachere Klasse `Opt` benutzt werden.

• Die Klassenmethode `newRe()` der Klassen `Scan` dient als eine Factory für Bäume von Objekten der Klassen `Loop`, `Alt`, `Opt` und `Seq` mit Objekten der Klassen `BOL`, `EOL`, `Set`, `SetMN` und `Word` als Blätter.

• Die Klassenmethode `newRe()` der Klasse Scan bekommt als `String` einen regulären Ausdruck, erzeugt mit einem rudimentären Parser einen Baum der Objekte und liefert die Wurzel des Baums als Resultat.

3.8 Serialisierung

• `Scan` und das `Action`-Interface adaptieren das `Serializable`-Interface.

• Durch die Serialisierung können die Objekte von Aufruf zu Aufruf eines Scanners wiederbenutzt oder von verschiedenen Scanner geteilt werden.

• In der Implementierung der `Scan`-Unterklassen wurde auf ein sinnvolles Verhalten bei der Serialisierung geachtet. Nur die notwendigen Instanzvariablen werden serialisiert.

3.9 Alle Scan-Klassen

Basis-Klassen

• Typische Basis-Erkenner sind als innere Klassen zu `Scan` gekapselt:

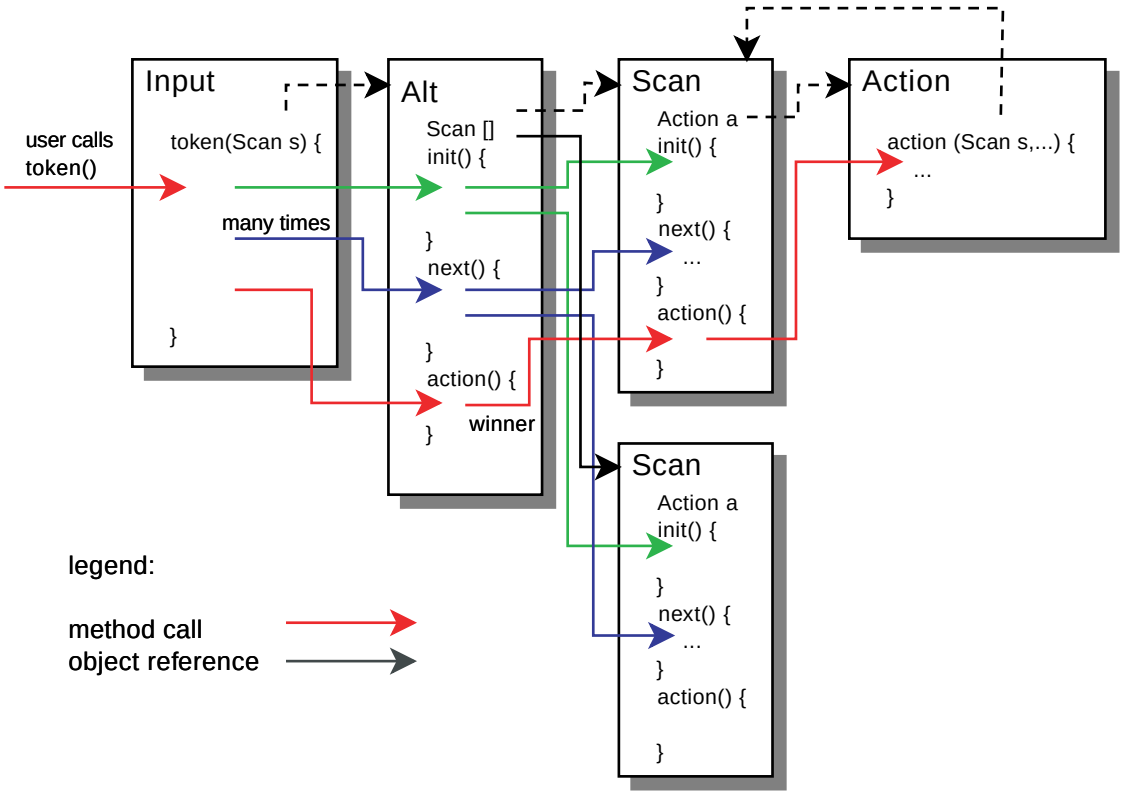
```
package oolex;

public abstract class Scan implements Serializable {
        ...
        public interface Action extends Serializable { ... }
        public static class Alt extends Scan { ... }
        public static class Seq extends Scan { ... }
        public static class Opt extends Scan { ... }
        public static class Loop extends Scan { ... }
        public static class Set extends Scan { ... }
        public static class SetMN extends Scan { ... }
        public static class Word extends Scan { ... }
        public static class EOL extends Word { ... }
        public static class BOL extends Scan { ... }
}
```

Die Bibliothek

• Die primäre Verwendung von Scannern liegt im Compilerbau. Die Bibliothek ist noch im Aufbau begriffen und enthält als Paket `oolex.scanobjects` Klassen zur Erkennung typischer Sprach-Symbole:

```
Balanced          HashComment       JavadocComment
CComment          HexNumber         OctalNumber
CIdentifier       Int               QuotedString
Char              JavaIdentifier    SlashSlashComment
Flt               JavaWhitespace
```

3.10 Kontrollfluß



• Das `Input`-Objekt bekommt als Argument eine `Alt`-Instanz mit zwei Erkennern. Der obere Scanner verweist auf ein `Action`-Objekt.

• Zu Beginn von `token()` wird beim Scanner `init()` aufgerufen. Das `Alt`-Objekt reicht den Aufruf an alle Instanzen im Raum weiter.

• Anschließend werden dem `Alt`-Objekt durch Aufruf von `next()` Zeichen aus der Eingabe vorgeworfen. Es reicht die Zeichen wiederum an noch aktive Instanzen im Raum weiter. Verläßt das letzte Objekt den Raum, werden `accepted` und `length` gesetzt und es wird `false` als Resultat von `next()` geliefert.

• Hat die Erkennung Erfolg, wird aus `token()` beim Scanner `action()` mit den erkannten Zeichen aufgerufen. Da das `Alt`-Objekt selber mit keiner `Action`-Instanz verbunden ist, ruft es beim Gewinner `action()`. Der Gewinner ruft die `action()` bei seiner `Action`-Instanz auf.

• Abschließend endet `token()` mit `true`.

3.11 Das Anfangsbeispiel mit oolex

```
Scan s = new Alt(new Scan[] {
    new Int().setAction(new Action () {
        public void action(Scan s, char buf[], int off, int len) {
            System.out.println("integer: "+s.genValue(buf, off, len));
        }
    }), new SetMN("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ",
              true, 1, Integer.MAX_VALUE).setAction(new Action () {
        public void action(Scan s, char buf[], int off, int len) {
            System.out.println("identifier:"+s.genValue(buf, off, len));
        }
    }), new JavaWhitespace().setAction(new Action () {
        public void action(Scan s, char buf[], int off, int len) {
            System.out.println("whitespace: "+s.genValue(buf, off, len));
        }
    }), new Char().setAction(new Action () {
        public void action(Scan s, char buf[], int off, int len) {
            System.out.println("character: "+s.genValue(buf, off, len));
        }
    })
});
```

4 Diskussion

4.1 Nachteile

• Findet sich für ein Problem in der Bibliothek keine geeignete Klasse, ist die Implementation einer neuen `Scan`-Klasse aufwendiger als das Hinzufügen einer weiteren Zeile mit einem regulären Ausdruck. Aber: Läßt sich das Problem durch einen regulären Ausdruck beschreiben, kann auch der automatisch erzeugte Baum von `Scan`-Objekten verwendet werden.

• Um einen *oolex*-Scanner verwenden zu können, sind als Runtime-Unterstützung die Klassendateien der verwendeten Klassen nötig.

• Der einzige echte Nachteil stellt die Performance dar. *[fJ]lex*-Scanner sind endliche deterministische Automaten mit einer berechneten Tabelle von Zustandsübergangs-Information. Ein *oolex*-Scanner dagegen betreibt in einer `Alt`-Instanz die Erkennung der Alternativen parallel. Weiter betreibt eine `Seq`-Instanz das erste `Scan`-Objekt und möglicherweise mehr als eine Instanz des zweiten `Scan`-Objekts parallel. Damit muß das System zwangsläufig langsamer sein.

• Ein erster schneller Vergleich von *oolex* mit *JLex* zeigt, daß für eine 370 K große C-Datei ein Scanner von typischen Symbolen circa um den Faktor vier langsamer ist.

4.2 Vorteile

• Der neue Ansatz ist objekt-orientiert und damit für objekt-orientierte Sprachen wie Java, Objective-C oder C++ intuitiver.

• Die Bibliothek soll einfach zu benutzene, aber mächtige Klassen für den Anwender enthalten. Komplexe und damit kryptische und potentiell fehleranfällige reguläre Ausdrücke sind das Gegenstück auf der *[fJ]lex*-Seite.

• Die Klassen der Bibliothek sind für verschiedene Scanner zu verwenden (Reuse). Analog müßte der Anwender auf der *[fJ]lex*-Seite reguläre Ausdrücke von Hand kopieren.

• Eine Firma, die eine andere Bibliothek von Erkennern entwickelt hat und verkaufen möchte, kann die Klassen in binärer Form verkaufen. Die Klassen sind ohne jede weitere Aktion vom Anwender verwendbar.

• *oolex*-Scanner arbeitet auf Unicode-Basis. Die tabellengetriebenen Scanner berechnen aus den regulären Ausdrücken eine Zustandsübergangsmatrix; für Unicode ist die Berechnung der Tabelle sehr aufwendig.

• Die Entwicklungszeit für *oolex*-Scanner ist sehr gering. Das System ist daher ideal für *rapid prototyping*.

• `Scan`- und `Action`-Instanzen sind serialisierbar. Damit sind nicht nur die Klassen, sondern auch Instanzen der Klassen wiederverwendbar. Scanner können so von verschiedenen Projekten in binärer Form geteilt werden.

• Einer `Input`-Instanz kann bei unterschiedlichen `token()`-Aufrufen verschiedene `Scan`-

Instanzen mitgeteilt werden. So kann durch verschiedene Räume z.B. Zustände modelliert werden.

• Darüber hinaus könnte einem `Alt`-Objekt durch eine Methode weitere Alternativen mitgeteilt werden. Bei *[fJ]lex* würde dies einer weiteren Zeile mit einem neuen Muster entsprechen. Dies kann bei *[fJ]lex*-Scannern im Gegensatz zu *oolex* nicht im laufenden Betrieb geschehen.

• Jedem Scan-Objekt kann im laufenden Betrieb ein anderes `Action`-Objekt mitgeteilt werden. Dies entspricht in *[fJ]lex* einer neuen Aktion zu einem Muster. Auch dies zieht eine Neuübersetzung des *[fJ]lex*-Scanners nach sich.

• Der objekt-orientierte Ansatz bietet natürlich durch Vererbung bzw. Ableitung die Möglichkeit, existente Klassen zu spezialisieren (Bsp. Klasse `Char`) oder abzuändern. Eine andere `Alt`-Klasse könnte sich bei Mehrdeutigkeit z.B. für den letzten oder den mittleren als Gewinner entscheiden.

• Dadurch, daß die Erkennung eines Symbols mit einer Klasse korrespondiert, können Symbole erkannt werden, die mit regulären Ausdrücken bzw. mit endlichen Automaten nur schwer oder gar nicht erkannt werden können. Verschachtelte C-Kommentare stellen als Unterklasse von `Balanced` kein Problem dar.

5 Fazit

• *oolex* eignet sich zum Anschluß an von Parser-Generatoren wie *jay* oder *oops* erzeugten Parsern.

• *oolex* bietet sehr viele Vorteile gegenüber der herkömmlichen Technik der tabellengetriebenen Scanner. Der Preis ist die langsamere Performance.

• Da der Anwender zur Not einen Baum von `Scan`-Objekten als Repräsentierung eines regulären Ausdrucks verwenden kann, verliert er mit dem neuen Ansatz nicht an Funktionalität.

• Dank der Bibliothek fertiger Klassen zum Erkennen von Symbolen ist die Benutzung des Systems um ein Vielfaches leichter.

• Dank der Verwendung von Objektorientierung ist das System um ein Vielfaches mächtiger.

• **Fazit: *oolex* ist eben lexikalische Analyse mit Klasse!**

6 Literatur/Links

• Die oolex-Dokumentation.

• V. Paxson, Flex - Fast lexical analyzer generator, Lawrence Berkeley Laboratory, ftp://ftp.ee.lbl.gov/flex-2.5.4.tar.gz, 1995.

• Homepage JLex: http://www.cs.princeton.edu/~appel/modern/java/JLex/

• Homepage jay: http://www.inf.uos.de/jay/

• Vorlesung Compilerbau, Wintersemester 1998/1999, http://www.vorlesungen.uni-osnabrueck.de/informatik/compilerbau98/

• Andrew W. Appel, Modern compiler implementation in Java, Cambridge University Press, 1997.

• Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers -- Principles, Techniques and Tools, Bell Laboratories, 1986/87.

• Dieser Vortrag als pdf-Datei.

• Die Beispiele des Vortrags (code.zip).