STUDIENARBEIT

Berufsakademie Stuttgart − Außenstelle Horb

Staatliche Studienakademie

Fachrichtung Informationstechnik

Erweiterung der semantischen Analyse des Java−Compilers

Mai 2003

Eingereicht von:

Felix Reichenbach

Ortlerstr. 12

71069 Sindelfingen

Firma:

DaimlerChrysler AG

71059 Sindelfingen

Betreuer:

Prof. Dr. Martin Plümicke

Abstract

Der im Rahmen der Vorlesungen Informatik 4 und Sortware Eingineering entstandene Java−Compiler wurde erweitert, so dass eine Typisierung von Attributen und Methoden bei abstrakten Datentypen (Klassen) möglich ist. Dies beinhaltet die Anpassung der lexikalischen Regeln, der Grammatik sowie der semantischen Analyse. Die Motivation hierfür ist die Vermeidung von Laufzeitfehlern durch Fehlerhafte Zuweisungen von Variablen und Methodenaufrufen, unter Verwendung von Listen, Hashtabellen und Vectoren.

Ehrenwörtliche Erklärung

Ich habe die Studienarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Sindelfingen, den 27.05.03 Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Erklärung von Abkürzungen

ADT	Abstrakter Datentyp
Lex / Jlex	Tool zur lexikalischen Analyse
jacc	jet another compiler compiler
Jay	Compiler−Generator
UML	Unified Modeling Language
ws	white space

1 Einleitung

Die Programmiersprache Java soll so erweitert werden, dass Typisierung von Attributen bei abstrakten Datentypen (Klassen) möglich ist. Dies soll vor allem bei der Implementierung von Listen, Hashtabellen und ähnlichen Containern zur Speicherung von Objekten zur Anwendung kommen.

Die Motivation für diese Erweiterung begründet sich in der Tatsache, dass die bisherigen Java−Compiler beim Umgang (Zuweisungen oder Variablenübergaben) mit Elementen aus solchen abstrakten Datentypen (ADT) keine Typfehler feststellen können. Derartige Fehler können somit erst zur Laufzeit erkannt und durch Exception abgefangen werden.

Ziel dieser Arbeit ist es, das Projekt 'Java−Compiler' aus dem 4. Semester so zu erweitern, dass es der oben beschriebenen Aufgabenstellung gerecht wird.

Dies umfasst folgende Arbeitsschritte:

- Untersuchung und gegebenenfalls Korrektur des vorhandenen Quellcodes
- Erweiterung der lexikalischen Regeln
- Erweiterung der syntaktischen Regeln (Grammatik)
- Ausbau der semantischen Analyse

Zum compilieren der Projektdateien liegt ein Makefile vor, welches durch den einfachen Befehl 'make' ausgeführt wird.

Der Compiler selbst wird als Java−Programm mit folgendem Befehl ausgeführt:

```
java MyCompiler < MyFile.java
```

2 Analyse des Problems

2.1 Klassendeklaration

Die Parameterdeklaration erfolgt direkt mit der Deklaration der Klasse. Parameter werden direkt nach dem Klassennamen in spitzen Klammern ’<...>’ angegeben. Diese Parameterliste wird vom Parser eingelesen und die Typen in einem Vector in der Klasse ’ParaList’ gespeichert.

Beispiel 2.1.1 Klassendeklaration:

```
class List<A>{
        ...
}
```

Wird eine solche parametrisierbare Klasse vererbt, so ist bei der Angabe der Basisklasse die Parameterliste mit anzugeben.

Beispiel 2.1.2 Klassendeklaration bei Vererbung:

```
class newList extends List<A>{
        ...
}
```

Soll die Child−Klasse wiederum Parameter enthalten, so werden diese genauso nach dem Klassennamen angegeben. Ebenfalls können bei Vererbung die Parameter der Basisklassen festgelegt werden, so dass diese bei der weiteren Verwendung einen festen Typ besitzen. Dieser Typ muss eine bereits bekannte Klasse sein. Sollte dieser Typ ebenfalls parametrisierbar sein, so müssen alle seine Parameter festgelegt sein.

Beispiel 2.1.3:

```
class newList<B, C> extends List<List<Sting>>{
        ...
}
```

2.2 Variablendeklaration

Die Syntax der Deklaration für die zu parametriserenden Attribute innerhalb einer Klasse erfolgt durch Voranstellung des entsprechenden Platzhalters. Wird eine Klasse instantiiert, so werden die dort angegebenen Parametertypen dem entsprechenden Attribut zugewiesen.

Beispiel 2.2.1:

```
class Container<A,B,C>{
        A object1;
        B object2;
        C getObject1(){ ... }
}
```

2.3 Instantiierung

Bei der Instantiierung der Klasse erfolgt die Zuweisung der Typen. Der neuen Instanz wird eine Liste von Typen mitgeteilt, welche anstelle den − bei der Deklaration angegebenen − Platzhaltern der Klasse zugewiesen werden.

Beispiel 2.3.1:

```
class main(){
        public void main(){
                List <String> l =new List();
                l.object = 'hello world';
                System.out.println(l.getObject());
        }
}
```

2.4 Methodendeklaration

Der Rückgabewert von Methoden soll ebenfalls parametriserbar sein. Hierbei soll sich die Syntax analog zu der bei Variablendeklarationen verhalten (vgl. Beispiel 2.2.1)

Die Angabe von parametrisierbaren Objekten als Übergabeparameter, wie in Beispiel 2.3.1 dargestellt, ist nicht implementiert.

Beispiel 2.3.1

```
int insert(A element){
        ...
}
```

3. Lösungsschritte

3.1 Erweiterung des Parsers

Die Grammatik des Parsers ist dahingehend zu erweitern, dass die in Kapitel 2 vorgestellte Erweiterung der Syntax erkannt und akzeptiert wird. Begonnen wird in Abschnitt 3.1.1 mit der Deklaration des neuen Datentypes ’ParaList’ im 2. Sektor der Spezifikationsdatei ’JavaParser.jay’ (Vgl. Kapitel 4.2.2).

3.1.1 Die Parameterliste

Alle Parameterlisten, die im Zusammenhang mit der Parametrisierung von Klassen vorkommen, werden in Instanzen der Klasse ’ParaList’ repräsentiert.

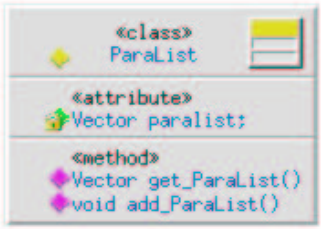

Abbildung 1: Klasse: ParaList

Das Attribut ’paralist’ der Klasse ’ParaList’ referenziert auf Objekte vom Typ ’Type’. Auch die Klasse ’Type’ wurde um das Attribut ’Vector paralist’ erweitert, so dass diese wiederum Referenzen auf Paramterlisten enthalten kann. Die Regeln für die Auswertung von rekursiven Parameterlisten (Bsp.: list<list<int>>) wurden wie folgt realisiert:

```
paralist : IDENTIFIER                                  { ... }
         | IDENTIFIER ’<’ paralist ’>’                 { ... }
         | paralist ’,’ IDENTIFIER                     { ... }
         | paralist ’,’ IDENTIFIER ’<’ paralist ’>’    { ... }
```

3.1.2 Klassendeklaration

Bei der Klassendeklaration wird die Parameterliste an die Klasse ’Class’ übergeben und im Attribut ’paralist’ gespeichert. Abbildung 2 zeigt die neue Klassenstruktur um die Klasse ’Class’.

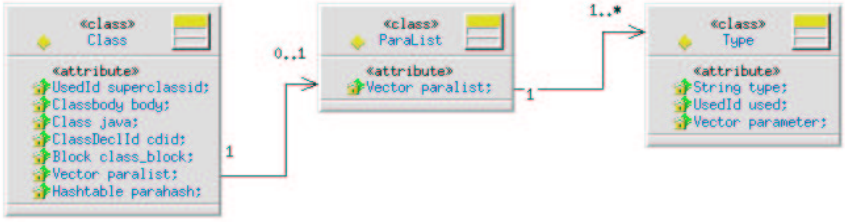

Abbildung 2: Class−>ParaList −> Type

Die Grammatik wurde um folgende Regel erweitert:

```
classtype       : classorinterfacetype                          { ... }
                | classorinterfacetype    ’<’ paralist ’>’  { ... }
```

3.1.3 Variablendeklaration

Platzhalter (siehe Beispiel 2.2.1) werden zunächst wie gewöhnliche Typangaben behandelt und als Objekt vom Typ ’Type’ abgelegt, wie das unveränderte Klassendiagramm in Abbildung 3 zeigt.

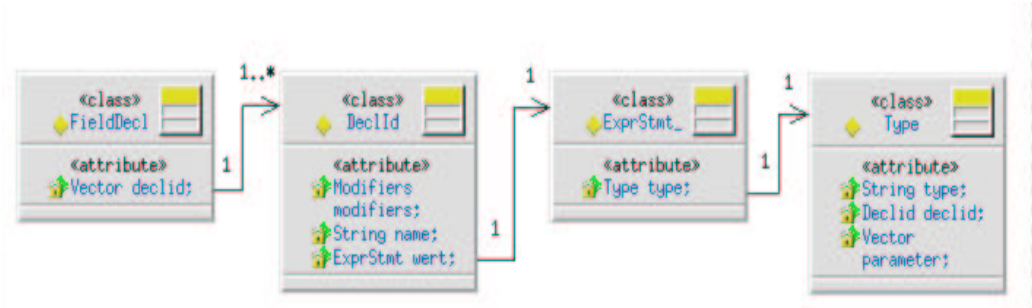

Abbildung 3: FieldDecl −> DeclId −> ExprStmt −> Type

3.1.4 Instantiierung

Wird eine Instanz einer parametrisierbaren Klasse erzeugt, so wird die Parameterliste im gleichnamigen Attribut ’paralist’ in der Klasse ’LocalVarDecl’ gespeichert (Abbildung 4).

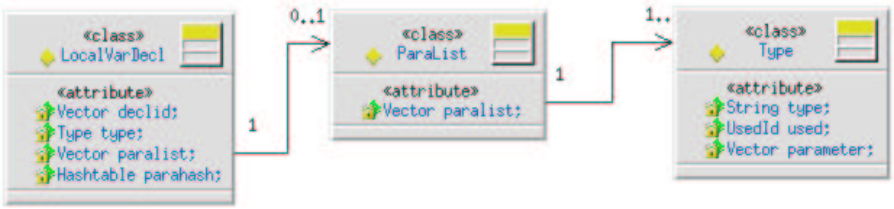

Abbildung 4: LovalVarDecl –> ParaList –> Type

Dies erforderte eine Erweiterung der Grammatik um folgende Regel:

variabledeclarator	: variabledeclaratorid	{ ... }
	| variabledeclaratorid ’=’ variableinitializer	{ ... }
	| ’<’ paralist’>’ variabledeclaratorid ’=’ variableinitializer	{ ... }

3.1.5 Methoden

Bei der Deklaration von Methoden werden die Platzhalter zur Parametrisierung der Rückgabewerte wie auch bei Variablendeklarationen als Objekt der Klasse ’Type’ gespeichert, da Methodendeklarationen wie auch die Variablendeklarationen als Objekt der Klasse ’FieldDecl’ gespeichert werden (vgl. Abblidung 3).

3.2 Erweiterung der semantischen Analyse

Aus der Klasse ’MyCompiler’ wird die Funktion ’sc_check(boolean ext)’ des Attributes ’Sourcefile SC_File’ aufgerufen, welche iterativ alle Elemente des Klassenvektors ’classes’ durchläuft und für alle Klassen die Funktion ’sc_check(boolean ext)’ aufruft. Diese Funktion steigt rekursiv durch den gesammten Syntaxbaum hindurch und überprüft ihn auf syntaktische Korrektheit.

Wird ein syntaktischer Fehler entdeckt, so wird ein Exceptionvektor erzeugt, welcher nach Beendigung der Funktion ’sc_check(boolean ext)’ ausgewertet wird und als Fehlermeldung im ausführenden Terminal erscheint.

3.2.1 Klassendefinition

Die Funktion ’void sc_check(Vector classlist, boolean ext)’ in der Klasse ’Class’ ruft die neue Member−Funktion ’void para_check(Vector classlist, boolean ext)’ auf. Diese Funktion organisiert sämtliche Aufgaben, welche zur syntaktischen Analyse der Parameterliste notwendig sind.

Zu Beginn wird − aus der bei der Klassendefinition angegebenen Parameterliste − eine Hashtabelle erzeugt, welche als Schlüssel die parametrisierbaren Attribute und Methoden und als Werte die Platzhalter aus der Parameterliste enthält. Somit ist eine Zuordnung eines jeden Variablen− bzw. Methodennamens zum entsprechenden Platzhalter geschaffen. Die Hashtabelle ’parahash’ sowie der Vector ’paralist’ sind neue Attribute der Klasse ’Class’ (vgl. Abbildung 2).

Die Erstellung der Hashtabelle selbst findet jedoch in der Funktion ’Hashtable init_parahashtable(Boolean ext)’ der Klasse ’ClassBody’ statt. Hier werden die im Vector ’fielddecl’ stehenden Objekte vom Typ ’InstVarDecl’ bzw. ’Method’ ausgelesen und bei Übereinstimmung von Typ und Platzhalter aus der Parameterliste in die Hashtabelle eingefügt.

Erbt die entsprechende Klasse von einer anderen, ebenfalls parametrisierbaren Klasse, so vervollständigt die Funktion ’Hashtable complete_parahashtable(Vector classlist,UsedId superclassid,Hashtable childhash,boolean ext)’ (diese Funktion befindet

sich ebenfalls in der Klasse ’ClassBody’) die Hashtabelle sowie die Funktion ’Vector complete_paralist(boolean ext)’ die Parameterliste der Child−Klasse (Klasse ’Class’).

Werden bei Vererbung den Basisklassen feste Parameter zugewiesen (vgl. Beispiel 2.1.3), so wird die Parameterliste gekürzt und das Attribut ist in der Child−Klasse nichtmehr parametriserbar. Diese Funktionalität wird ebenfalls von der Funktion ’complete_parahashtable’ abgedeckt.

Zur Übersicht ist der Ablauf sowie die Funktionalität der im letzten Abschnitt beschriebenen Funktionen noch einmal als Sequenzdiagramm (Abbildung 5) sowie als Tabelle (Tabelle 1) dargestellt.

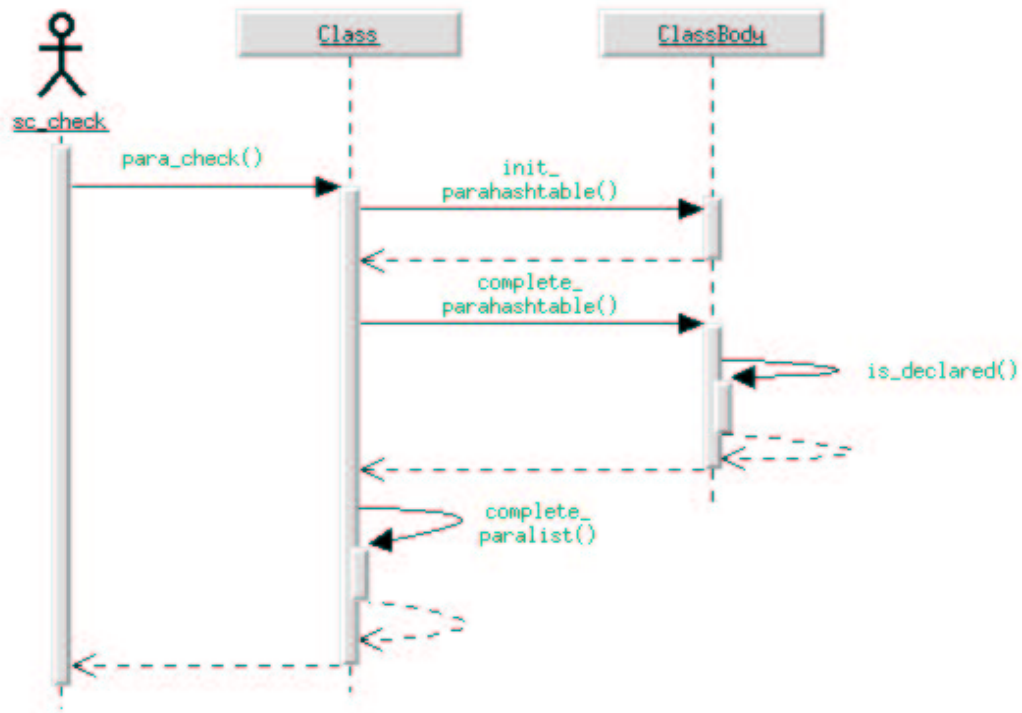

Abbildung 5: Sequenzdiagramm −> Class.para_check()

Name	***Klasse***	***Parameter***	***Funktion***
para_check()	Class.java	Vector classlist boolean ext	ruft 'init_parahashtable()' auf prüft, ob alle Platzhalter der Parameterliste auch in der Klasse deklariert sind prüft bei Vererbung, ob die Parameterlisten von Basis und Child übereinstimmen ruft 'complete_parahashtable()' auf ruft 'complete_paralist()' auf
complete_paralist()	Class.java	boolean ext	vervollständigt bei Vererbunng die Child−Parameterliste
init_parahashtable()	ClassBody.java	boolean ext	erstellt aus der Parameterliste und den Attributen der Klasse die Hashtabelle(Attribut−>Platzhalter)
complete_parahashtable()	ClassBody.java	Vector classlist UsedId superclassid Hashtable childhash boolean ext	vervollständigt bei Vererbung die Hashtabelle 'parahash' der Child−Klasse entfernt bei Vererbung vordefinierte Typen aus Parameterliste und Hashtabelle
void is_declared()	ClassBody.java	Type t Vector classlist	Prüft, ob ein angegebener Typ im Vector 'classlist' enthalten ist

Tabelle 1: Fuktionsübersicht Class.sc_check()

3.2.2 Instantiierung

Bei der Instantiierung eines parametrisierbaren Objektes (vgl. Beispiel 2.3.1) wird geprüft, ob die in der Parameterliste angegebenen Typen im Klassenvektor vorhanden und somit bekannt sind (Funktion ’void is_declared(Type t, Vector classlist)’).

Als nächster Schritt wird geprüft, ob die Anzahl der in der Parameterliste angegebenen Typen der Deklaration entspricht. (Funktion ’void check_anz(Type type, Vector paralist, Vector classlist)’).

Sind diese beiden Kriterien erfüllt, wird anhand der Parameterliste und der Hashtabelle aus Klasse ’Class’ eine instanzspezifische Hashtabelle erstellt, welche die Zuweisungen ’Variablenname−−>Typ’ der Instanz enthält.

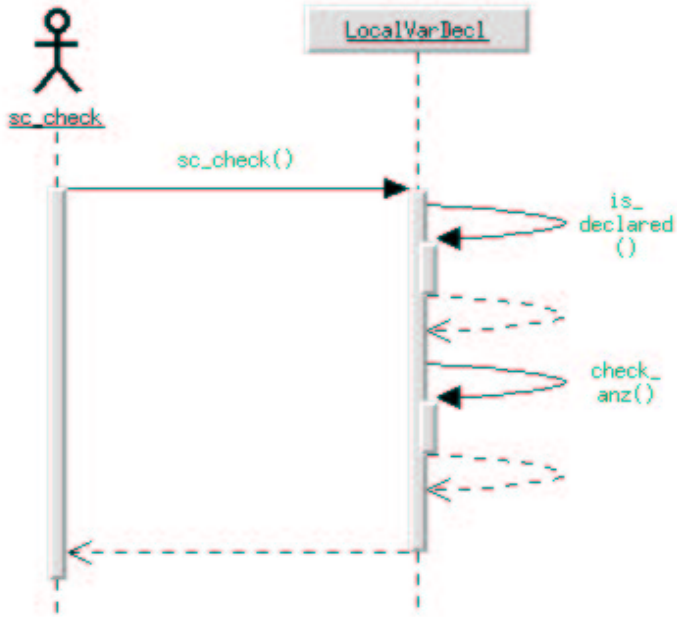

Abbildung 6: Sequenzdiagramm −> LocalVarDecl.para_check()

Name	*Klasse*	*Parameter*	*Funktion*
void is_declared()	LocalVarDecl.java	Type t Vector classlist	Prüft, ob ein angegebener Typ im Vector 'classlist' enthalten ist
void checke_anz()	LocalVarDecl.java	Type type Vector paralist Vector classlist	Prüft, ob die Anzahl der Parameter in der Parameterliste der der Deklaration entspricht

Tabelle 2: Funktionsübersicht LocalVarDecl.sc_check()

3.2.3 Methoden

Bei parametriserbaren Methoden wird der Typ des Rückgabewertes ähnlich wie bei Variablen nach Instantiierung eines Objektes als Typ des Rückgabewertes geschrieben.

Da die vorhandene Compilerstruktur jedoch keine Angabe eines 'Return Statements' erzwingt, ist diese Funkiton nur teilweise implementiert und bedingt funktionsfähig. Es kann der Rückgabewert der Funktion als Parameter angegeben werden, jedoch überprüft die syntaktische Analyse nicht, ob dieser Parameter mit dem tatsächlichen Rückgabewert der Funktion übereinstimmt.

3.2.4 Exceptionhandling

Werden von der syntaktischen Analyse Fehler entdeckt, so werden diese über Exceptions durch die Baumhierarchie hindurchgereicht und am Ende in der Klasse 'MyCompiler' ausgegeben. Die Fehlermeldungen erscheinen unter Angabe von Statement, Funktion und Klasse auf der Kommandozeile.

4 Tools

4.1 JLex

4.1.1 Einleitung

Um einen Datenstrom von Zeichen analysieren zu können, muss eine lexikalische Analyse durchgeführt werden. Diese Mechanismen werden in der Informatik auch als Automaten bezeichnet. Ein solcher Automat kann anhand seiner Eingabe entscheiden, ob der eingelesene Text der Definition einer bestimmten Sprache entspricht.

Da eine solche Analyse einen sehr aufwendigen Prozess darstellt, wurden in der Vergangenheit eine Vielzahl solcher Tools entwickelt.

Das wohl bekannteste Tool ist ’Lex’, welches auf den meisten UNIX−basierten Systemen bereits vorhanden ist. Lex ist ausgelegt für die Verwendung unter der Programmiersprache C. Anweisungen für die lexikalische Analyse sind in eine speziell formatierte Datei eingebettet (vorzüglich mit der Endung ’.lex’). Als Ausgabe erhält man eine C−Datei, welche für die weitere Verarbeitung noch zusätzlich verschiedene Methoden zur Verfügung stellt.

JLex basiert im Wesentlichen auf dem Modell des oben genannten Lex mit dem Unterschied, dass als Ausgabe Java Quellcode generiert wird, welcher wiederum entsprechend weiterverwendet werden kann.

JLex selbst ist ebenfalls in Java programmiert und wird auch als solches aufgerufen:

```
java JLex <Spezifikationsfile.lex>
```

4.1.2 Funktionsweise

Wie bereits erwähnt ist zur Verwendung von JLex eine Datei mit verschiedenen Spezifikationen zur Arbeitsweise von JLex zu erstellen. Diese Datei ist jeweils durch eine '%%'−Direktive am Zeilenanfang in 3 Sektoren untergliedert:

```
user code
%%
JLex directives
%%
regular expression rules
```

4.1.2.1 user code

Der user−code Sektor wird direkt an den Anfang der erzeugten Output Datei kopiert. Somit ist es möglich, an dieser Stelle direkt Package−Deklarationen oder Verweise auf externe Klassen sowie Implementierungen von Klassen vorzunehmen.

4.1.2.2 %%−Direktiven

Der zweite Abschnitt beginnt nach der ersten '%%'−Direktive und endet an der zweiten. Vorzugsweise werden hier Definitionen von Makros zur Deklaration von Zustandsnamen angegeben.

Hierbei werden zur Steuerung verschiedene Direktiven angewandt, wobei hier nur auf die Verwendeten eingegangen wird.

Die '%{ ... %}'−Direktive ermöglicht den Einschub von Quellcode in die erzeugte Analysator−Klasse.

```
%{
        <code>
%}
```

Die '%eofval{ ... %eofval}'−Direktive ermöglicht dem Benutzer den Rückgabewert der 'lexical analyzer tokenizing'−Funktion Yylex yylex() bei erreichen des Dateiendes

anzugeben. Die Syntax hierzu lautet wie folgt:

```
%eofval{
        <code>
%eofval}
```

Die ’%public’−Direktive deklariert die erzeugte Klasse als ’public’.

```
%public
```

Die ’%class’−Direktive ermöglicht eine Umbenennung der generierten Analysator−Klasse. Der hierfür vorgesehene Default−Name ist Yylex.

```
%class <name>
```

Durch die ’%type’−Direktive wird der Rückgabewert der Funktion Yylex yylex() explizit angegeben.

```
%type <type>
```

Ein Beispiel für die Definition von Makros ist die Ausblendung von sogenannten ’white spaces’ (ws), also Leerzeichen die durch verschiedene Steuerzeichen hervorgerufen werden (Space, Tabulator, Zeilenumbruch, etc...)

Syntax:

```
<name> = <definition>
```

Beispiel:

```
ws = [ \t\r\n\b\015]+
```

Der volle Funktionsumfang des Direktiven−Sektors ist in den Manual Pages von Jlex unter Abschnitt 2. Jlex Detectives zu finden.

4.1.2.3 lexical rules

Im dritten Abschnitt sind letztlich die Regeln für die lexikalische Analyse definiert. Diese Regeln bilden die komplette Sammlung der regulären Ausdrücke sowie die damit assoziierten Funktionen in Java.

Die Syntax dieser Regeln lässt sich wiederum in drei Teile unterteilen:

- Zustandsliste, optional (lexical states)
- reguläre Ausdrücke (regular expressions)
- Anweisungen (actions)

Syntax:

```
[<states>]<expression>{<action>}
```

- **longest match:**

  Falls bei einer lexikalischen Analyse mehr als eine Regel auf eine vorhandene Zeichenkette zutrifft, so wird − um Konflikte zwischen den Regeln zu vermeiden − diejenige ausgewählt, welche auf den längsten String zutrifft, auch als ’longest match’ bezeichnet (zu beachten: ein String gilt immer als abgeschlossen, sobald er durch ws−Zeichen unterbrochen wird).

Durch die angegebenen Regeln sollte die gesamte Bandbreite aller möglichen Zeichen abgedeckt werden, da unbekannte Zeichen eine Fehlerausgabe hervorrufen. Dies kann durch Angabe der folgenden Regel am Ende der JLex Spezifikation verhindert werden:

```
. { java.lang.System.out.println("Unmatched input: " + yytext()); }
```

Diese Regel wird wegen Angabe des Punktes (.) durch jedes Zeichen (außer ’newline’) zutreffen.

1. Abschnitt (Lexical States)

Optional kann eine Liste der lexikalischen Zustände angegeben werden. Diese Liste bestimmt dann, unter welchen Bedingungen die Regel angewandt wird. Dies bedeutet, dass bei einem Aufruf der Funktion yylex() während eines Zustands 'A' nur diejenigen Regeln beachtet werden, welche den entsprechenden Zustand auch in ihrer Zustandsliste aufgeführt haben.

Syntax:

<state[0], [state[1], state[2], ...]>

Falls auf die Angabe einer Zustandsliste verzichtet wurde, so ist die Regel in jedem lexikalischen Zustand gültig.

2. Abschnitt (Regular Expressions)

Die Angabe der regulären Ausdrücke sollte keine white−space−Zeichen enthalten, da diese als das Ende des aktuellen regulären Ausdrucks interpretiert werden (Ausnahme: in Anführungszeichen angeführte Leerzeichen).

JLex verwendet den ASCII Zeichensatz , wobei die in Tabelle 3 aufgeführten Metazeichen eine besondere Bedeutung haben.

Zeichen	***Bedeutung***
?	Nachfolgender Ausdruck muss genau 0 oder 1 mal vorkommen
*	Nachfolgender Ausdruck kann 0..n mal vorkommen
|	Oder−Verknüpfung zweier Ausdrücke
(...)	Gruppierung von Ausdrücken
^	Negierung eines Ausdruckes
$	Das Dollar−Zeichen steht am Ende eines regulären Ausdrucks und stellt einen Zeilenumbruch dar
[...]	Gruppierung von Zeichen zu einer 'Klasse'
{name}	Macroaufruf
a − z	Range von Zeichen
"..."	Metazeichen zwischen Anführungszeichen verlieren ihre besondere Bedeutung
\	Metazeichen auf einen Backslash folgend verlieren ebenfalls ihre Bedeutung

Tabelle 3: JLex Metazeichen

Folgende Escape−Sequenzen werden ebenfalls beachtet:

Sequenz	*Bedeutung*
\b	Backspace
\n	Zeilenumbruch
\t	Tabulator
\f	Formfeed
\r	Carriage Return
\ddd	Angabe des Zeichens als oct−Zahl
\xdd	Angabe des zeichens als hex−Zahl
\udddd	Angabe des Zeichens als hex−Zahl (Unicode)
\^C	Steuerzeichen (Ctrl−C)

Tabelle 4: JLex Escape−Sequenzen

Beispiel:

[A−Za−z]	Groß− und Kleinbuchstaben
(*[0−9]hex)|(0x*[0−9])	hexadezimale Zahl in der Notation ’0x...’ oder ’...hex’
["A−B"]	eines der 3 Zeichen ’A’ ’−’ ’B’

3. Abschnitt (Associated Actions)

Der Dritte Abschnitt beschreibt die Funktion der Regel. Der auszuführende Java−Code wird hierbei in geschweiften Klammern angegeben. Auf eine Bildung weiterer Funktions− und Programmblöcken mit ’{}’ sollte hierbei verzichtet werden.

- **Anweisungen und Rekursion**

  Wird bei einer Regel kein Rückgabewert angegeben, so wird dies ignoriert und der die Funktion yylex() arbeitet gewöhnlich weiter. Die explizite Rückgabe der Funktion yylex() hat die gleiche Wirkung.

```
{ ...
        return yylex();
}
```

  Dieser Aufruf der Funktion yylex() verursacht jedoch nicht einen rekursiven Aufruf von yylex() sondern die Weiterverarbeitung des Input−Streams.

Ein erzwungener rekursiver Aufruf von yylex() darf deshalb nicht über das ’return’−Statement erfolgen sondern muss explizit angegeben werden.

```
{ ...
        next = yylex();
... }
```

Bei Verwendung von Variablen wie ’yyline’ oder ’yychar’ ist unter Anwendung solcher rekursiver Aufrufe zu beachten, dass diese dadurch verändert werden können.

- **Zustandsänderungen:**

  Falls eine implementierte Regel Zustandsänderungen erfordert, so können diese ebenfalls innerhalb des Funktionsblocks durch die Funktion:

  ```
  void yybegin(state)
  ```

  angegeben werden. Der Zustand muss jedoch als ’%state’−Direktive deklariert worden sein (vgl. Jlex−Manual 2.2.5 State Declarations).

- **Zählen von Zeichen und Zeilen:**

  Das Zählen von Zeichen und Zeilen ist defaultmäßig ausgeschaltet und kann durch die Direktiven

  %line und

  %char

  aktiviert werden. Die Werte werden in den Integer Variablen ’yyline’ bzw. ’yychar’ abgelegt.

Der vollständige Funktionsumfang der lexikalischen Regeln ist in den Manual Pages von Jlex unter Abschnitt 2.3 Regular Expression Rules zu finden.

4.1.3 Auszug der Jlex−Spezifikation des Java−Compilers:

```
/*****************************************      *
 * File: JavaLexer.lex                          *
 * enthält die JLex−Spezifikation für die       *
 * Generierung des lexical analyzers            *
 *****************************************      */
        // 1. Sektion
//user code...

%%     //2. Sektion
%{
   Token token;
%}
%public
%class JavaLexer
%type boolean
%eofval{
  return false;
%eofval}
ws = [ \t\r\n\b\015]+

%%     //3.Sektion
abstract {this.token = new Token(JavaParser.ABSTRACT, yytext());return true;}
catch {this.token = new Token(JavaParser.CATCH, yytext());return true;}
char {this.token = new Token(JavaParser.CHAR, yytext());return true;}
// ...

[()\{\}\[\];,.]     { this.token = new Token(JavaParser.BRACE,yytext().charAt(0));return true;}
[=><!~?:]           { this.token = new Token(JavaParser.RELOP,yytext().charAt(0));return true;}
// ...

"==" { this.token = new Token(JavaParser.EQUAL, yytext());return true;}
"<=" { this.token = new Token(JavaParser.LESSEQUAL, yytext());return true;}
">=" {this.token = new Token(JavaParser.GREATEREQUAL, yytext());return true;}
// ...
```

4.2 Jay

4.2.1 Einführung

Um nun die von der lexikalischen Analyse (Scanner) erzeugten Tokens in einen sinnvollen Kontext zu bringen, wird ein sogenannter Parser benötigt. Dieser Parser bildet mit Hilfe der entsprechenden Grammatik einen Ableitungsbaum (abstrakte Syntax). Dieser Baum stellt eine interne Repräsentation des Source−Files dar.

Auch hierfür sind bereits einige Tools vorhanden, als die Bekanntesten sind wohl yacc und bison zu erwähnen, welche wiederum beide auf die Programmiersprache C, bzw. C++ ausgelegt sind. Analog zu JLex ist Jay nun das Java−Pendant zu yacc .

Im Gegensatz zu JLex ist Jay selbst jedoch in C programmiert und kann somit direkt aufgerufen werden:

```
jay Spezifikationsfile.jay <skeleton> Javafile.java
```

Im Wesentlichen sind alle Grammatiken von Programmiersprachen kontextfrei und lassen sich nach der Sprachhierrachie von Noam Chomsky in die Kategorie Chomsky−1 einordnen.

Zu einer kontextfreien Grammatik gehören vier Komponenten

1. Eine Menge von Terminalen. Terminale sind die Symbole einer Sprache. Terminal heißen sie, weil sie am Ende der Auflösung stehen. Beispiele sind Stern, Klammern, aber auch Schlüsselworte (if, then, while) oder Konstanten.
2. Eine Menge von Nichtterminalen. Nichtterminale sind Zwischensymbole der Grammatik, die noch nicht vollständig aufgelöst sind.
3. Eine Menge von Produktionen. Eine Produktion besteht links aus einem Nichtterminal, gefolgt von einem Pfeil und einer rechten Seite, die aus Terminalen und Nichtterminalen bestehen kann. Eine Produktion ist eine Regel, die beschreibt, wie sich zulässige Worte (bzw. Sätze) der Grammatik bilden lassen.
4. Ein Startsymbol. Das Startsymbol ist ein Nichtterminal.

Beispiel:

statement −> if (expression) statement else statement

4.2.2 Funktionsweise

Durch den Aufruf des Programms wird eine Java−Klasse mit dem angegebenen Namen erzeugt.

Wie auch bei der lexikalischen Analyse benötigt auch der Parser eine Spezifikation oder Grammatik. Diese Spezifikation wird in einem ’.jay“−File’ angegeben und kann in 3 Teile untergliedert werden:

Definitionen
%%
Regeln
%%
Java−Funktionen

Zudem wird noch eine Datei namens ’y.output’ erzeugt, in der alle erkannten Parserzustände sowie eventuell auftretende Konflikte aufgelistet werden.

4.2.1.1 Definitionen

Der erste Abschnitt des ’.jay’−Files dient der Angabe von Definitionen. Hierbei können wiederum durch verschiedene Direktiven Angaben und Codeergänzungen zum generierten Java−File gemacht werden. Ausserdem müssen in diesem Abschnitt alle Tokens sowie alle Klassen deklariert werden, die der Parser verarbeiten soll.

Direktive	***Erklärung***
%{ <java code> %}	Der Angegebene Java−Code wird in die erzeugte Klasse kopiert.
%token <type> name	Deklaration der Tokens des Eingabestrings, die Angabe des Types ist optional

Direktive	*Erklärung*
%type name	Typdeklaration der Nichtterminale
%start s	Deklaration des Startsymbols der Grammatik
%left	
%right	

Tabelle 5: Jay Direktiven

4.2.1.2 Regeln

Im 2. Abschnitt werden die Ableitungsregeln und die entsprechenden Aktionen festgelegt. Hierbei werden alles Symbole, die nicht als Tokens deklariert wurden, als Nichtterminale behandelt. Die Aktionen werden ausgeführt, sobald die jeweilige Regel reduziert wird.

Syntax:

```
A     :     B token1          {action1;}
      |     C token2          {action2;}
```

Die Aktionen werden als Java−Code angegeben. Mit Angaben wie ’$<num>’ kann auf Symbole zugegriffen werden ($1 greift auf das erste, $2 auf das zweite Symbol zu usw.). Mit ’$$’ wird ein Symbol bei Reduzierung der Regel an die nächsthöhere Hierarchie weitergegeben.

Da die Symbole jeweils aus Objekten bestehen ergibt sich der Nachteil, dass auch primitive Datentypen immer in Objekte gekapselt werden müssen.

4.2.1.3 Java−Funktionen

Im 3. Abschnitt können weitere Funktionen für die Klasse angegeben werden (z.B. eine ’main’−Funktion zur Steuerung des Parsers oder Routinen zur Fehlerbehandlung etc).

4.2.3 Auszug der Jay−Spezifikation des Java−Compilers:

```
/*****************************************  *
 * file: JavaParser.jay                      *
```

```
 *                                                *
 * enthält die Jay−Spezifikation                  *
 *                                                *
 ************************************************ */
%{
        import java.util.Hashtable;
        import java.util.Vector;
        class JavaParser{
        public Vector path = new Vector();
%}

%token <Token> ABSTRACT
%token <Token> BOOLEAN
%token BREAK
%token CASE
%token CATCH
%token <Token> CHAR
//...

%%
compilationunit      : typedeclarations                  { $$=$1; }

typedeclarations     : typedeclaration                   { Sourcefile Scfile = new Sourcefile();
                                                           Scfile.set_Class($1);
                                                           $$=Scfile; }
                     | typedeclarations typedeclaration  { $1.set_Class($2);
                                                           $$ = $1; }

name                 : qualifiedname                     { $$=$1; }
                     | simplename                        { $$=$1; }
//...
%%
```

4.3 Innovator

Als Modellierungswerkzeug wurde der Innovator benutzt. Dieses Tool ermöglicht eine schnelle Erstellung von Use−Case− und UML−Diagrammen.

Die angelegten Projektdaten werden in sogenannten Repositories gespeichert. Um auf ein solches Repository zugreifen zu können, muss zunächst der entsprechende Repository−Server gestartet werden. Der Repository−Server stellt die Verbindung zwischen der GUI und dem Repository her und kontrolliert den Zugriff auf die Projektdaten.

Benutzeroberfläche, Grafikeditoren usw		
Repository− Server 1	Repository− Server 1	Repository− Server n
Repository		

Tabelle 6: Client−Server−Architektur des Innovator

Um eine Übersicht über die Repositories zu erhalten, startet man das Unterprogramm 'inotree'. Hierbei werden alle aktiven Repositories in einem Baum angezeigt und können zur Bearbeitung angewählt werden. Hierzu erscheint dann der Modellbrowser, welcher die Arbeit innerhalb des Projektes ermöglicht.

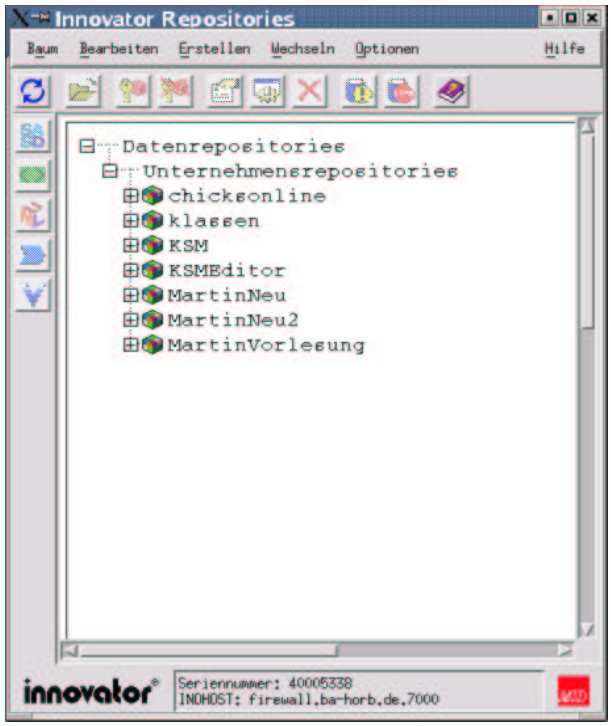

Abbildung 7: Innovator Modellbaum

Der Modellbrowser ist in 4 Teile untergliedert:

- Modellbaum
- Modellelemente
- Detailansicht
- Ergebnisbereich

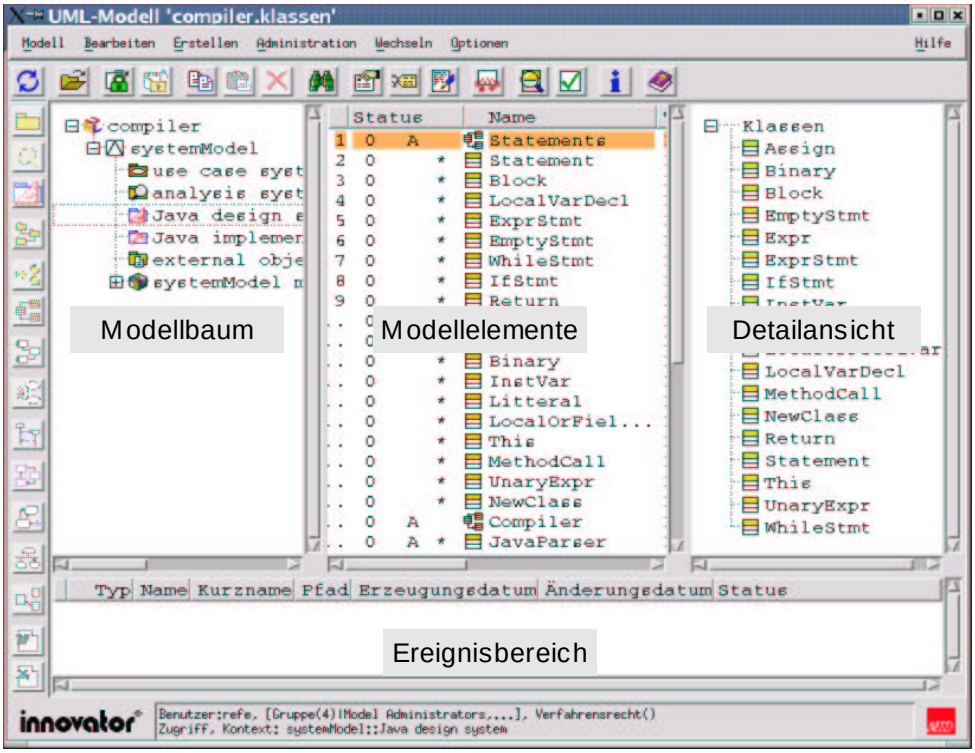

Abbildung 8: Innovator Modellbrowser

Im Modellbaum wird die Struktur des Modells dargestellt. Der Modellinhalt ist in Pakete zerlegt. Die Teilbäume können auf− und zugeklappt werden. Wird im Modellbaum ein Element selektiert, so werden im mittleren Bereich − der Liste der Modellelemente − alle Elemente angezeigt, die einem Paket zugeordnet sind. Wenn Sie eines dieser Elemente selektieren, erhalten Sie die Detailansicht angezeigt. Hier werden alle Referenzen des in der Liste der Modellelemente selektierten Objektes angezeigt. Man sieht alle Elemente auf die sich eine Änderung des selektierten Elementes auswirken wird. Im Ergebnisbereich werden Elemente dargestellt, auf die diagrammübergreifende Funktionen des Innovator angewendet werden sollen (z. B. Dokumentationsgenerierung).

Nach Auswahl eines Modellelements erscheint das dazugehörige Diagramm welches dann mit dem Diagramm−Editor modifiziert werden kann.

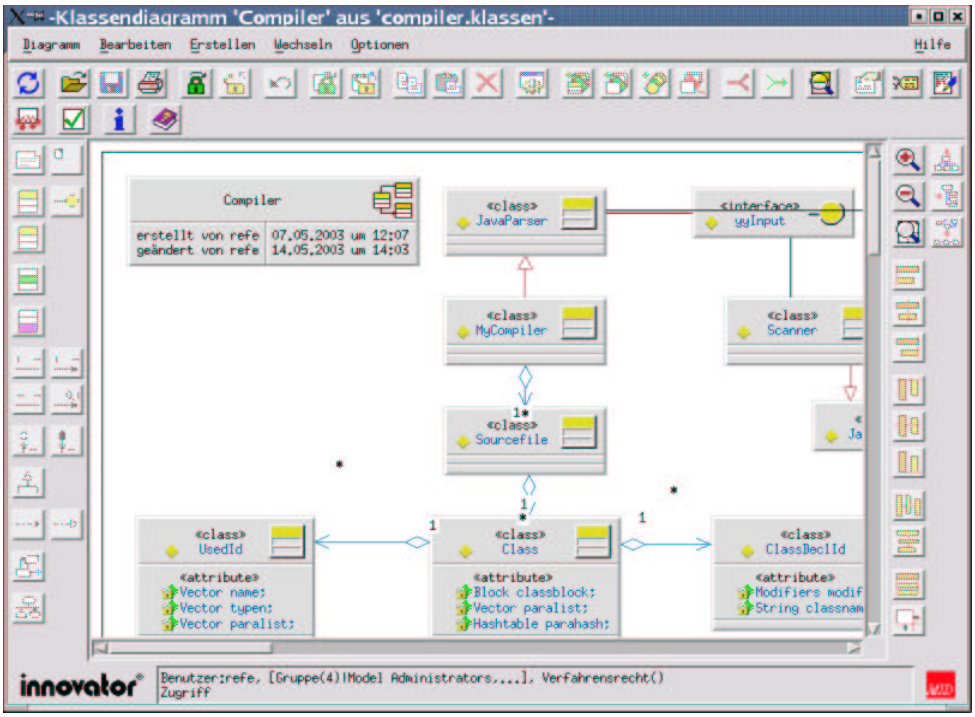

Abbildung 9: Innovator Diagramm−Editor

5 Zusammenfassung und Ausblick

Die Aufgabenstellung konnte nur bedingt gelöst werden, da der vorhandene Compiler für die Erfüllung der Aufgabenstellung des Sortwareengineering−Projektes ’Compilerbau’ optimiert ist und somit immer wieder systematische Fehler von Seiten der Grammatik und der Implementierung des Syntaxbaumes auftauchten, welche die eigentliche Arbeit dieser Studienarbeit erschwerten.

Für die korrekte Inplementierung einer solchen Erweiterung der Sprache wäre eine vollständige Überarbeitung oder Neugestaltung der Grammatik und des daraus erzeugten Syntaxbaumes − inklusive der Funktionen − notwendig, um ein durchgängiges Design zu erhalten, welches zu derartigen Erweiterungen fähig ist.

Die intensive Beschäftigung mit dieser Studienarbeit verdeutlichte sehr gut die Komplexität dieses Themengebietes der Informatik und zeigte einmal mehr die Notwendigkeit eines klaren Systemdesigns sowie einer vollständigen Dokumentation der Software.

Quellenverzeichnis

Bücher, Manuals

[1] Jay−Tutorial

Bernd Kühl, Axel−Tobias Schreiner

[2] Jlex−Manual

Elliot Berk

[3] Compilerbau

Aho, Sethi, Ullmann, Addison Wesley 1988

Internetadressen:

[1] http://pizzacompiler.sourceforge.net/index.html

[2] http://www.compilerbau.de/

[3] http://www.willemer.de/informatik/

[4] http://www.dbg.rt.bw.schule.de/lehrer/ritters/info/bagcb/compbau.htm

[5] http://www.fbi.fh−darmstadt.de/~case/innovator/

Anhang

Beispiel eines ADT's vom Typ Liste:

Deklaration:

```
class List<A>{
        List <A>next = new List();
        A object = null;
        A getObjectAt(int n){
                if(n==0){
                        return object;
                }
                else{
                return this.next.getObjectAt(n−1);
                }
        }
        void insert(A object){
                if(this. Object == null){
                        this.object = object;
                }
                else{
                        this.next.insert(object);
                }
        }
}
```

Instantiierung:

```
List <String> l = new List();
l.object = new String('hello');
l.next.l = new String('world');
```

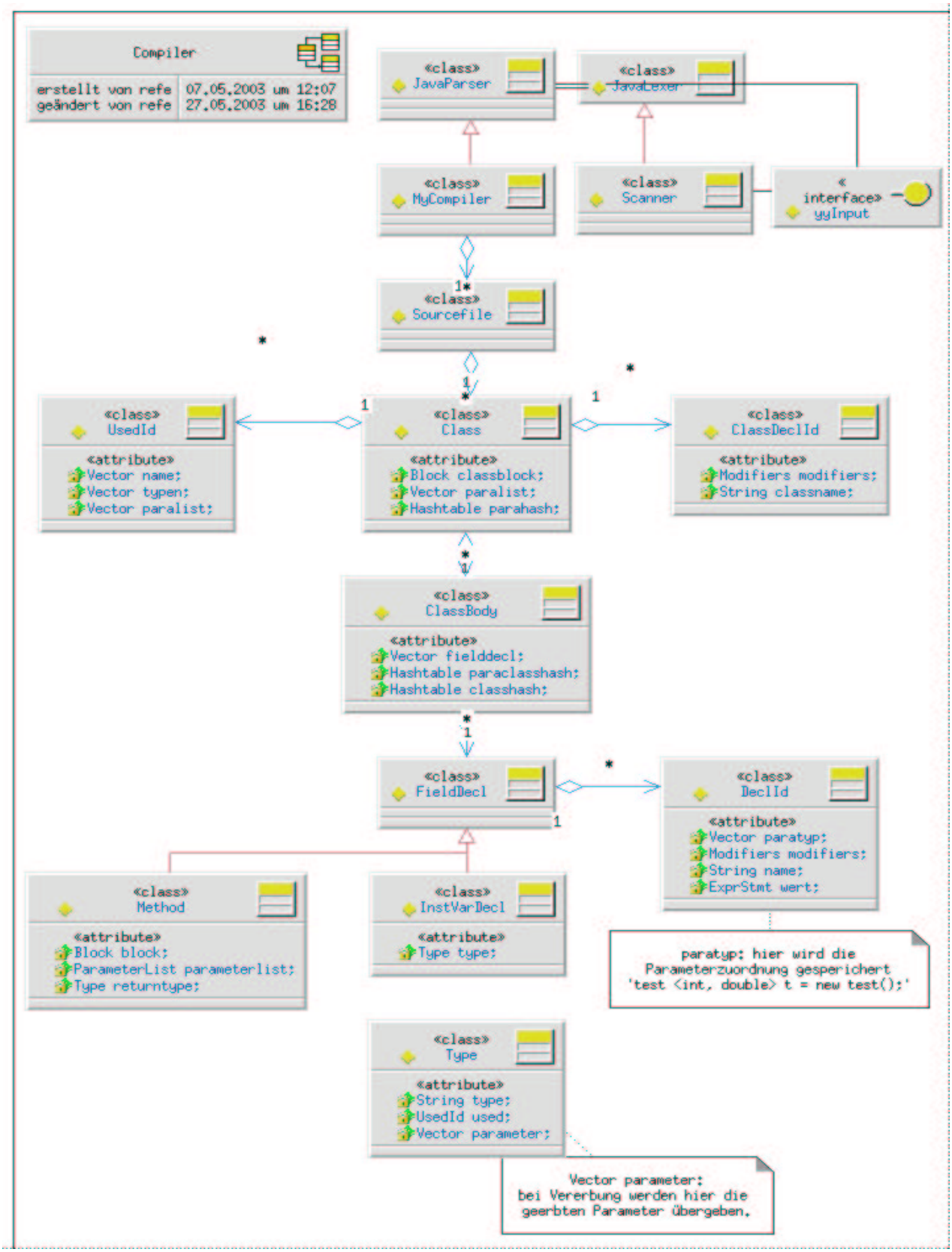

Abbildung 10: Compiler

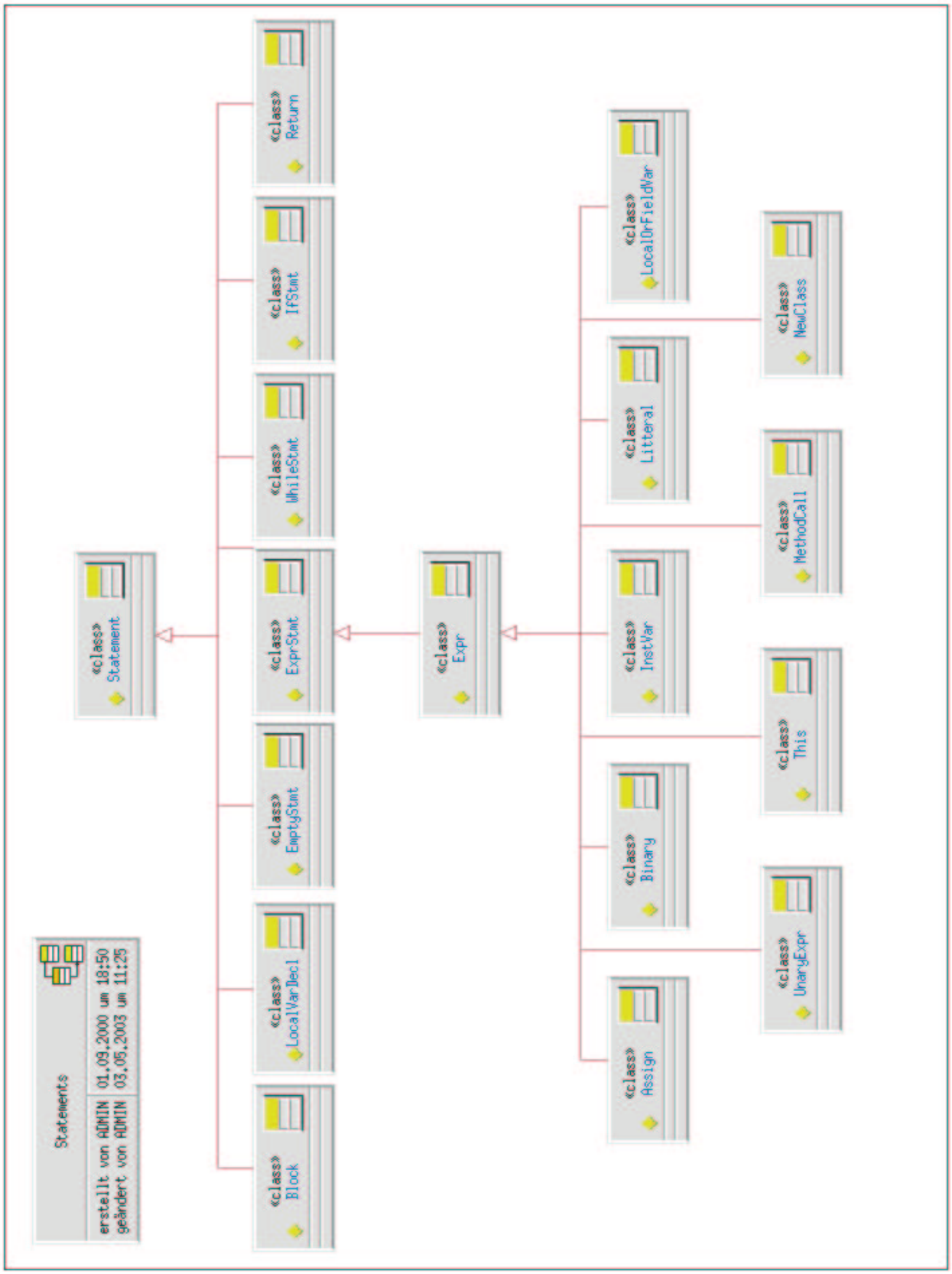

Abbildung 11: Statements