Technische Universität Braunschweig
Institut für Software Systems Engineering

Hilfswissenschaftliche Tätigkeit

Monticore zum Parsen von großen Datenmengen

Jan Oliver Ringert
Matrikel-Nr.: 2772985

Aufgabenstellung: Eshref Januzaj
Betreuer: Eshref Januzaj

Braunschweig, den 9. Februar 2007

Abstract

Diese Arbeit befasst sich damit, in wie weit MontiCore (GKR06) für das Parsen von großen Datenmengen eingesetzt werden kann. Als Beispiel soll das Protokoll eines Sensors geparst werden (siehe Listing 5). Es handelt sich um eine einfache Grammatik (siehe Listing 4), die jedoch bei üblichen Datenmengen eine sehr großen AST erzeugt. Im Weiteren wird untersucht, welche Alternativen sich für das Parsen von großen Datenmengen anbieten.

Inhaltsverzeichnis

1 Einführung 1

2 Speicherplatzproblem 2
- 2.1 Menge der AST-Objekte 4
- 2.2 Bestimmung des Speicherverbrauchs 4
- 2.3 Eigentlicher Speicherplatzbedarf 5

3 Mögliche Lösungen 6
- 3.1 ANTLR 6
- 3.2 Erweiterungen von ANTLR 7
- 3.3 Spezielle Java Programme 7

4 Zusammenfassung 8

5 Anhang: Listings 9

1 Einführung

Im Zuge einer hilfswissenschaftlichen Tätigkeit habe ich die Aufgabe erhalten, eine Grammatik zu definieren, mit der Sensordaten in bestimmten Formaten gelesen werden können. Diese Daten sollen mit Hilfe des Konzeptes PrettyPrinter transformiert werden, um sie aus verschiedenen Quellen einheitlich verwenden zu können. Es sollten später weitere Grammatiken definiert werden, mit denen die verschiedenen Ausgaben von Sensoren gelesen werden könnten.
Während der Ausarbeitung einer passenden Grammatik traten Probleme auf, die zeigen, dass MontiCore nicht besonders gut für diese Aufgabe geeignet ist. Im folgenden Abschnitt wird auf die Probleme eingegangen.
In Abschnitt 3 wird untersucht, welche Alternativen zu MontiCore möglich sind, um die gestellte Aufgabe zu lösen. Dabei werden hauptsächlich ANTLR und verwandte Arbeiten betrachtet.

2 Speicherplatzproblem

Die Grammatik zum erfassen der Messdaten sollte sich erst einmal auf einen Datenblock beschränken. Dieser Block besteht aus einer Titelzeile und beliebig vielen weiteren Zeilen mit Daten. Jede Zeile besteht aus einem Datum, einer Uhrzeit und einer gleich bleibenden Anzahl von Datenfeldern, welche jeweils eine rationale Zahl enthalten. Ein Beispiel für solche Daten findet sich in Listing 5. Die zugehörige Grammatik aus Listing 4 wird in Abbildung 1 grafisch dargestellt.

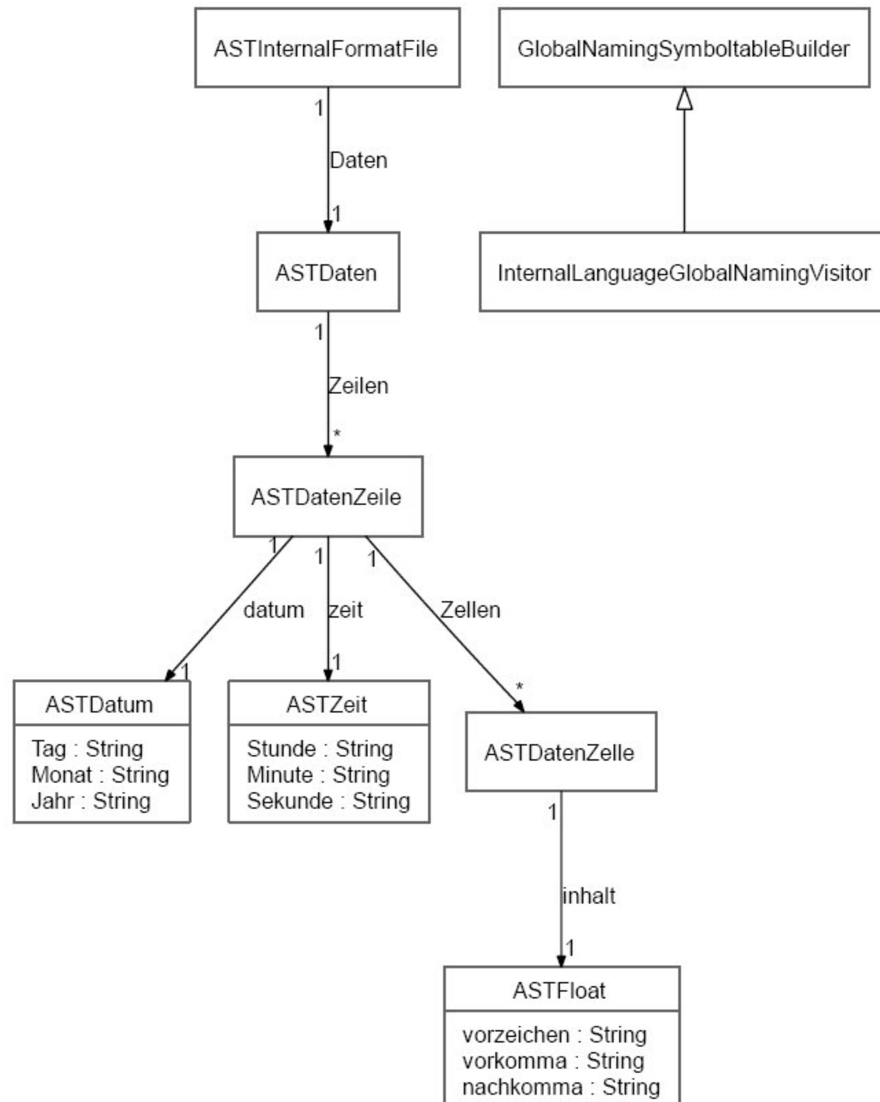

Abbildung 1: Klassendiagramm, das aus Listing 4 erzeugt wurde.

Die Klasse ASTInternalFormatFile repräsentiert die Datei, aus der die Daten gelesen werden sollen. Vereinfacht dargestellt hat sie hier nur eine Beziehung zu der Klasse ASTDaten, welche den eigentlichen Datenblock darstellt. Dieser Datenblock ist aufgeteilt in mehrere Zeilen vom Typ ASTDatenZeile. Jede Zeile besteht aus einem ASTDatum, einer ASTZeit und mehreren ASTDatenZellen. Die ASTDatenZelle ist der eigentlich interessante Wert, der hier vom Typ ASTFloat ist. Es fällt auf, dass eine DatenZeile mit mehreren DatenZellen assoziiert ist. Diese Datenzellen können optional einen Float erhalten. Dieser "Umweg" sorgt dafür, dass eine DatenZelle erstellt wird, selbst wenn kein Float vorhanden ist. Das Fehlen eines Float in der Daten Datei führt so nicht zu einem fehlerhaften AST. Die Po-

sitionen der folgenden Floats sind sehr wichtig für die richtige Zuordnung, die durch den eingefügten Platzhalter weiterhin möglich ist.
Wenn eine Datei gelesen werden soll, baut der von MontiCore erzeugte Code einen vollständigen AST auf, in dem alle Daten enthalten sind. Die Daten finden sich in konkreten Instanzen der oben erläuterten Klassen. In dem Beispiel, das dieser Arbeit zu Grunde liegt (Listing 5), ließen sich nur bis zu 5000 Zeilen parsen. Beim Überschreiten dieser Anzahl entstand eine der folgenden Fehlermeldungen:

```
1  Error: Give up parsing, too many errors!
2  oder
3  Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

Listing 1: Fehlermeldungen beim Einlesen des Datenblocks

Es stellte sich heraus, dass dieses Verhalten auf einen zu kleinen Heap Speicher der virtuellen Maschine von Java zurückzuführen war. Es werden standardmäßig 64MB verwendet. Nach einem Vergrößern des Heap Space mit der Option "-Xmx256m" auf 256MB (vergleiche (WWWa)) ließen sich größere Dateien einlesen.
Die Instanz eines ASTs wird in Abbildung 2 dargestellt.

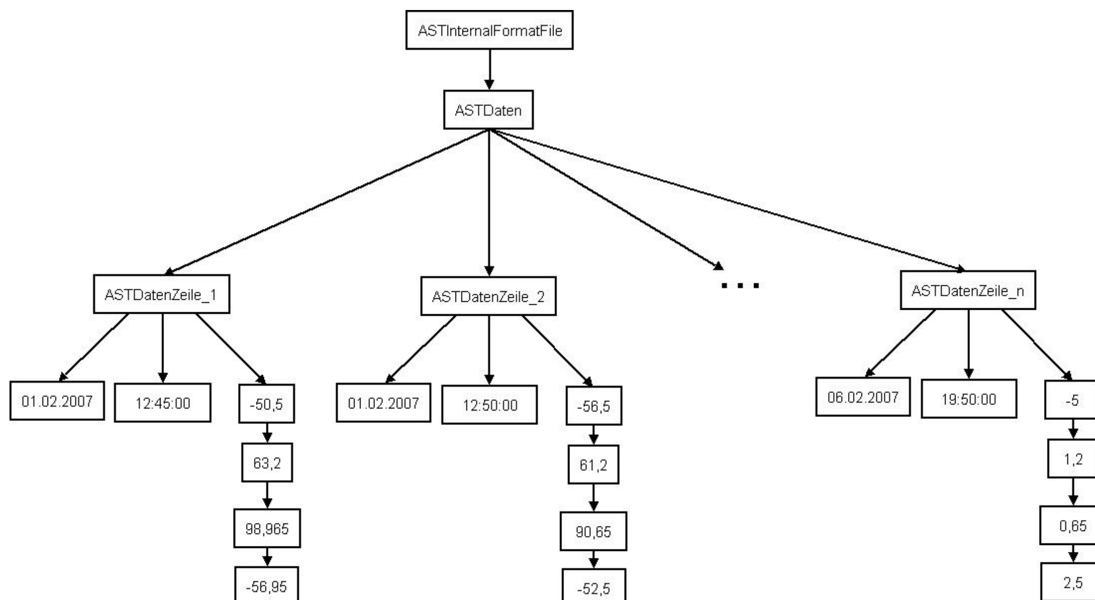

Abbildung 2: Instanz eines ASTs.

Diese Abbildung ist etwas vereinfacht dargestellt, da die Instanzen von ASTZeit, ASTDatum sowie ASTDatenZelle einfach durch ihre Entsprechungen in der Datei dargestellt werden. Weiterhin ist aus Platzgründen die Assoziation zwischen ASTDatenZeile und ASTDatenZelle als verkette Liste dargestellt.
Die genauere Betrachtung der Struktur findet sich in dem Objektdiagram in Abbildung 3. Hier wird gezeigt, welche Instanzen in dem durch MontiCore erstellten AST eine Zeile in der Daten Datei repräsentieren.

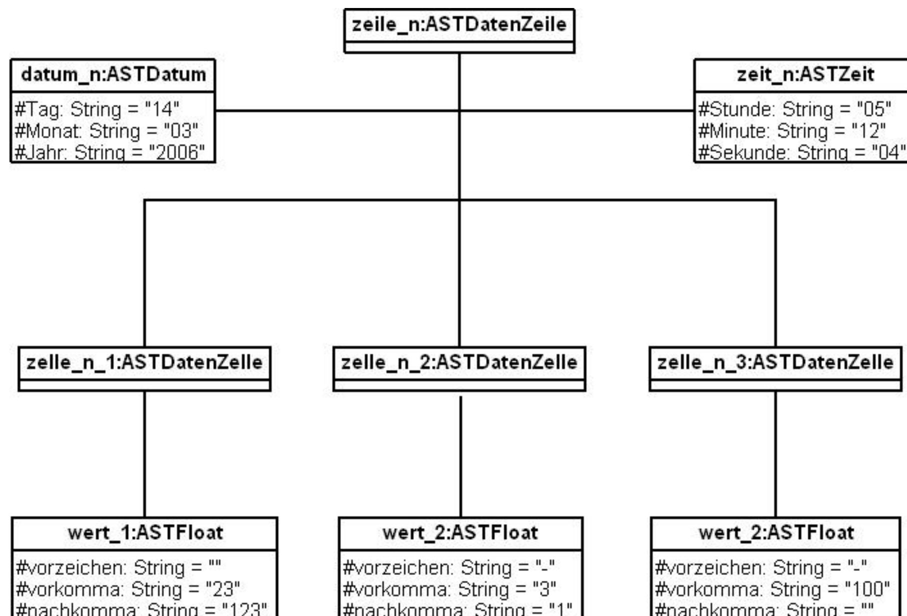

Abbildung 3: Detailansicht einer Zeile.

2.1 Menge der AST-Objekte

Die Menge der AST-Objekte, die beim Lesen einer Datendatei erzeugt werden müssen, lässt sich anhand der Struktur des Baumes bestimmen. Dabei sei im Folgenden n_{Zeile} die Anzahl der Zeilen und m_{Spalte} die Anzahl der Daten Zellen pro Spalte. Eine Zeile enthält demnach $m_{Spalte} + 2$ Zellen, da die ersten beiden Zellen das Datum und die Uhrzeit angeben.
Auf der ersten und zweiten Ebene existieren jeweils ein Objekt. Auf der dritten Ebene werden so viele Objekte wie Zeilen erzeugt also n_{Zeile} Stück. Für jedes Objekt aus der dritten Ebene werden weitere $m_{Spalte} + 2$ AST-Objekte angelegt. Insgesamt ergeben sich damit

$$2 + n_{Zeile} + n_{Zeile} * (m_{Spalte} + 2)$$

AST-Objekte, die zum Lesen der Datendatei instanziiert werden müssen. Eine Beispieldatei, von der ein Auszug in Listing 5 gegeben ist, hat knapp 11.000 Zeilen mit je 8 Datenzellen. Es ergibt sich eine Größe des AST von

$$2 + 11.000 + 11.000 * 10 = 121.002$$

AST-Objekten.

2.2 Bestimmung des Speicherverbrauchs

Eine Bestimmung der Menge an Speicher, die ein Java Objekt zur Laufzeit verwendet ist nicht ohne weiteres möglich. In (Rou03) werden die Schwierigkeiten erläutert, die durch die verschiedenen Implementierungen und Versionen der Virtuellen Maschine von Java auftreten. Die Größe für bestimmte Objekte lässt sich nach (Rou03) nur experimentell bestimmen. Das dort vorgeschlagene Verfahren greift auf eine Messmethode zurück, die bereits in (Rou02) beschrieben wurde. Für einige Tests mit unterschiedlichen Daten Dateien wurde die genannte Methode implementiert. Als erstes wird mit Hilfe des Java Garbage Collectors so viel Speicher wie möglich freigegeben. Der zweite Schritt besteht darin, den Speicherverbrauch zu bestimmen und danach den AST aufzubauen. Nach dem Erstellen des ASTs wird der Speicherverbrauch erneut bestimmt. Die Differenz wird ausgegeben.
Diese Methode ist in ihrer Umsetzung im Zuge dieser Arbeit nicht ohne Einschränkungen

und Messungenauigkeiten zu verwenden. Die erste Messung findet vor dem Parsen durch Monticore statt. Die zweite führt der PrettyPrinter durch. Vor jeder Messung wird der Garbage Collector aufgerufen. Es kann jedoch sein, dass durch Monticore weitere Objekte erstellt werden, die sich zu dem Zeitpunkt der zweiten Messung noch nicht löschen lassen. Diese Objekte finden sich so im gemessenen Speicherverbrauch wieder. Aufgrund dieser möglichen Ungenauigkeit wurden verschiedene Messungen durchgeführt.

```
 5 000 Zeilen: 46669208 Bytes = 44.507MB

 6 000 Zeilen: 56396384 Bytes = 53.784MB

 7 000 Zeilen: 66708816 Bytes = 63.618MB

 8 000 Zeilen: 75392912 Bytes = 71.900MB

 9 000 Zeilen: 84061192 Bytes = 80.167MB

10 000 Zeilen: 92729624 Bytes = 88.434MB
```

Listing 2: Speicherverbrauch des AST für verschiedene Zeilenanzahl der Daten Datei.

Der Zuwachs des Speichers pro 1000 Zeilen mit je acht Datenspalten liegt zwischen 8,5MB und 10MB. Es ist zu beachten, dass während des Einlesens und Parsens der Datei mehr Speicher verwendet wird. So kommt es, dass der Speicherverbrauch des AST mit 6000 Zeilen nur 54MB beträgt, dieser AST jedoch nicht mehr mit einem Heap von 64MB erstellt werden kann. MontiCore und die Komponenten wie das DSLTool benötigen für diese Arbeit anscheinend weitere (eigene) 10MB Speicherplatz.

2.3 Eigentlicher Speicherplatzbedarf

Der Bedarf an Speicherplatz unterscheidet sich bei diesem Problem deutlich von dem Verbrauch des Speicherplatzes. Der Verbrauch geht über die "normalen" Grenzen eines kleinen Programmes hinaus. Der Bedarf jedoch ist ziemlich gering. Neben dem Datenblock besteht eine typische zu verarbeitende Datei noch aus anderen Teilen (siehe Listing 6). Die Abschnitte "@sse" und "@Meta" enthalten Informationen über die Sensoren, deren Messergebnisse sich im Block "@Data" befinden. Teile dieser Informationen, die dazu dienen, die Datensätze zuordnen zu können, müssen während der gesamten Zeit der Verarbeitung einer Datei im Speicher gehalten werden. Die Daten aus dem Datenblock ("@Data") selbst sind unabhängig von anderen Zeilen in diesem Block. Es wäre also möglich, diesen letzten Bock zeilenweise zu lesen und zu verarbeiten. Der Speicherplatzverbrauch ist in diesem Fall sehr gering. Das Anlegen eines AST ist für die Verarbeitung der Daten nicht nötig.

3 Mögliche Lösungen

Es scheint, dass für die gestellte Aufgabe MontiCore nicht geignet ist. Es ist jedoch ein gute Idee, vorhandene Tools zu nutzen, um ähnliche Arbeiten schnell erledigen zu können. Ziel ist es verschiedene Grammatiken für die verschiedenen Ausgaben von Sensordaten zu erzeugen. So soll es möglich sein, mit der Zeit weitere Importmöglichkeiten bereitzustellen, die nicht an ein festes Schema gebunden sind. Zu diesem Zweck werden nun verwandte Arbeiten betrachtet, die alle (so wie MontiCore) auf ANTLR (WWWb) basieren.

3.1 ANTLR

ANTLR kann aus Grammatikdefinitionen die nötigen Java Programme zum Erkennen von Sprachen generieren. Die beiden hier interessanten Komponenten sind der Lexer und der Parser. Die Aufgabe des Lexers ist es, einen Eigabestrom (z.B. eine Datei) in kleine Abschnitte zu zerlegen. Diese Abschnitte sind die kleinsten Einheiten der Sprache, welche in der Grammatik definiert sind. Es handelt sich um die Terminalsymbole. Diese Symbole müssen mit eindeutigen Regeln charakterisiert werden. Beispiele hierfür sind Zahlen oder in Anführungszeichen eingefasste Zeichenketten.
Die zweite wichtige Komponente ist der Parser. Dieser erhält den Strom von Terminalsymbolen und prüft, ob ihre Kombination gültig im Sinne der Grammatik ist. Der Parser kann weiterhin einen der Definition folgenden AST aufbauen. Seine Grammatik besteht aus Nichtterminalsymbolen, die über verschiedene Kombinationen von anderen Symbolen definiert werden. Eine Kette von Definitionen muss dabei mit Regeln enden, die sich auf Kombinationen von Terminalsymbolen bezieht.
Bei der Definition des Parsers und des Lexers kann zu jeder Regel eine Aktion definiert werden, die bei Erkennung des Symbols durchgeführt wird. Diese Aktion wird in geschweiften Klammern angegeben und enthält Java Code, der direkt in den Code des Parsers/Lexers generiert wird. Ein einfaches Beispiel aus (Mil05) für einen Parser, der mit dieser Methode arbeitet findet sich in Listing 3.

```
1  class SimpleParser extends Parser;
2
3  entry : (d:DOB n:NAME a:AGE(SEMI)
4          {
5             System.out.println(
6               "Name: " +
7               n.getText() +
8               ", Age: " +
9               a.getText() +
10              ", DOB: " +
11              d.getText()
12            );
13         })*
14         ;
```

Listing 3: Einfacher Parser mit Aktionen zur Ausgabe.

Es wäre möglich, beim Erkennen von Sensordaten eine entsprechende Klasse zu schreiben, die die Daten in das gewünschte Format bringt (z.B. in eine Datenbank schreibt). Hierbei müssen im Zustandsspeicher des Parsers (oder einer zusätzlichen Klasse) die Informationen aus dem Abschnitt “@Meta” der Datei gespeichert werden. Beim Parsen der Daten würde der eingefügte Java Code mit den gespeicherten Zustandsinformationen die gelesenen Daten zuordnen und verarbeiten können. Diese Methode erfordert gute Schnittstellen und zusätzliche Klassen, damit möglichst wenig Java Code in der Definition des Parsers geschrieben werden muss.

3.2 Erweiterungen von ANTLR

Die obige Methode ist ziemlich umständlich und eigentlich nicht das, wofür ANTLR entwickelt wurde. Man erhält eine Mischung von Grammatikdefinition und Java Code, die sehr unübersichtlich uns schwer wartbar werden kann. Die normale Vorgehensweise wäre auch hier, einen AST generieren zu lassen und diesen für die eigentliche Datenverarbeitung zu durchlaufen.
Es gibt verwandte Probleme, für die bereits Lösungen im Internet diskutiert werden. Eine Arbeit (Par03) befasst sich damit, einen Stream von Tokens (Terminalsymbolen) in eine neue Form zu bringen, ohne einen AST zu erzeugen. Die in dem Artikel vorgestellte Rewrite Engine wurde in das ANTLR Paket aufgenommen. Es handelt sich dabei um eine Unterstützung, Tokens des Streams zu manipulieren und neue Tokens einzufügen. Dieses Paket ist jedoch hauptsächlich darauf ausgelegt, Streams zu manipulieren und bietet daher keine Unterstützung bei der hier gestellten Aufgabe.
Eine andere Erweiterung namens StringTemplate wird in (Par05) vorgestellt. Es handelt sich dabei um eine mächtige Möglichkeit, Sprachen zu transformieren. Über Templates können erkannte Konstrukte aus der Sprache in eine andere Form gebracht werden. Es handelt sich bei der neuen Form leider wieder um Strings. Diese Methode ist also auch nicht einfach zu verwenden. Es wäre möglich Templates zu schreiben, die direkt SQL Code erzeugen, um die Daten in die Datenbank zu schreiben. Es müsste jedoch weiterhin eine Möglichkeit gefunden werden, den Zustand des Parsers zu speichern und es würde die Möglichkeit verloren gehen, einfache Veränderungen oder Prüfungen an den Daten durchzuführen.

3.3 Spezielle Java Programme

Eine Möglichkeit, die am Anfang eigentlich vermieden werden sollte, ist es die gewünschten Java Programme nicht generieren zu lassen, sondern selber zu schreiben. Für jede mögliche Grammatik wäre es denkbar, eine Art Filter zu programmieren, der nur auf die Besonderheiten der Log Datei abgestimmt ist. Durch Hinzufügen eines neuen Filters wäre ein weiterer Dialekt abgedeckt. Die Qualität und Wartbarkeit dieser Lösung hängt stark von den entwickelten Schnittstellen ab. Es ist jedoch sinnvoll, auch diese Lösung zu betrachten, da das Lexen und Parsen der Daten sich größtenteils nur ein Splitten von Zeichenketten ist und so nicht unbedingt Werkzeuge wie generierte Parser und Lexer benötigt.

4 Zusammenfassung

Diese Arbeit hat gezeigt, dass für das Einlesen von Daten Dateien und Protokollen einer bestimmten Größe das Framework MontiCore nicht geeignet ist. Die Menge der Objekte, die beim Parsen mit einer einfachen Grammatik erzeugt werden, ist zu groß für den normalen Heap Speicher einer Virtuellen Maschine von Java (64MB). Die Struktur eines AST wird für die relevanten Daten (Messergebnisse) nicht benötigt.
Andere untersuchte Alternativen auf der Basis von ANTLR erbringen nicht die gehoffte Vereinfachung des Problems. Es scheint daher am Sinnvollsten, eine "Filter"-basierte Lösung zu wählen, in der einzelne Java Programme Plugin-ähnlich genutzt werden, um verschiedene Log Dateitypen einzulesen. Da die Hauptarbeit des Parsers darin besteht, Zeichenketten zu splitten, ist ein mächtiges Tool wie ANTLR nicht unbedingt nötig.

5 Anhang: Listings

```
1      ident INT "('0'..'9')+";
2
3      //InternalFormatFile ist Wurzel der Datei
4      InternalFormatFile = Daten:Daten;
5
6      //Symbole für die Datenerfassung
7      Zeit = Stunde:INT ":" Minute:INT ":" Sekunde:INT;
8
9      Datum = Tag:INT "." Monat:INT "." Jahr:INT;
10
11     //für den Datenblock
12     Daten = "@Data" ";"+ "\\r\\n" Zeilen:DatenZeile*;
13
14     DatenZeile = datum:Datum ";" zeit:Zeit ";" Zellen:DatenZelle+;
15
16     Float = vorzeichen:"-"? vorkomma:INT ("," nachkomma:INT)?;
17
18     DatenZelle = inhalt:Float? (";" | "\\r\\n");
```

Listing 4: MontiCore-Grammatik für einen Datenblock

```
1  @Data;;;;;;;
2  02.07.2005;03:14:48;15,915;15,986;33,809;54;22,585;22,465
3  02.07.2005;03:29:48;15,963;16,034;33,887;53,925;22,561;22,465
4  02.07.2005;03:44:48;15,939;16,058;33,835;53,85;22,585;22,489
5  02.07.2005;03:59:48;16,082;16,106;34,019;54,112;22,585;22,489
6  02.07.2005;04:14:48;16,082;16,106;34,045;54,3;22,585;22,489
7  02.07.2005;04:29:48;16,177;16,153;34,334;54,716;22,585;22,513
8  02.07.2005;04:44:48;16,201;16,129;34,334;54,792;22,585;22,513
9  02.07.2005;04:59:48;16,225;16,153;34,281;54,526;22,585;22,489
10 02.07.2005;05:14:48;16,249;16,201;34,334;54,792;22,513;22,393
11 02.07.2005;05:29:48;16,296;16,249;34,81;54,945;22,106;21,987
```

Listing 5: Beispiel eines Datenblocks

```
1   @sse;;;
2   ;;Vergabe SSE1;Vergabe SSE2
3   @Meta;;;
4   GLT-ID;;PWCB1: HOBO Dofa, Wert 1;PWCB1: HOBO Dofa, Wert 2
5   IGSname;;104B1xy0DOFAT01;104B1xy0DOFAT02
6   Klartext;;freier Text 1;freier Text 2
7   Art des Werts;;Lufttemp Dofa;Lufttemp Dofa
8   Einheit;;°C;°C
9   Skala;;-20 bis 40;-20 bis 40
10  Platzhalter 1;;;
11  Platzhalter 2;;;
12  Platzhalter 3;;;
13  @Data;;;
14  30.06.2005;13:14:48;23,641;24,195
15  30.06.2005;13:29:48;23,809;24,388
16  30.06.2005;13:44:48;23,881;24,677
17  30.06.2005;13:59:48;23,978;24,315
18  30.06.2005;14:14:48;24,026;24,653
19  30.06.2005;14:29:48;24,339;25,307
20  30.06.2005;14:44:48;24,388;25,355
21  30.06.2005;14:59:48;24,002;24,363
22  30.06.2005;15:14:48;24,412;24,968
23  30.06.2005;15:29:48;24,677;24,774
24  30.06.2005;15:44:48;25,04;24,968
25  30.06.2005;15:59:48;24,968;25,258
26  30.06.2005;16:14:48;24,992;25,162
27  30.06.2005;16:29:48;25,113;25,355
28  30.06.2005;16:44:48;25,283;25,355
29  30.06.2005;16:59:48;24,919;24,968
30  30.06.2005;17:14:48;24,508;24,532
```

Listing 6: Beispiel zu verarbeitender Log-Daten

Literatur

[GKR06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, Steven Völkel . *MontiCore 1.0, Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen* . TU-Braunschweig, Institut für Software Systems Engineering, Informatik-Bericht 2006-04 . 2006 .

[Mil05] Ashley J.S Mills . *ANTLR Tutorial.* University Of Birmingham . 2005 .
`http://supportweb.cs.bham.ac.uk/docs/tutorials/docsystem/build/tutorials/antlr/antlr.pdf`

[Par03] Terence Parr . *Syntax Directed TokenStream Rewriting* . University of San Francisco . 2003 .
`http://www.antlr.org/article/rewrite.engine/index.tml`

[Par05] Terence Parr . *Language Translation Using ANTLR and StringTemplate* . University of San Francisco . 2005 .
`http://www.codegeneration.net/tiki-read_article.php?articleId=77`

[Rou02] Vladimir Roubtsov. *Java Tip 130: Do you know your data size?.* JavaWorld . 16/08/2002 .
`http://www.javaworld.com/javaworld/javatips/jw-javatip130.html`

[Rou03] Vladimir Roubtsov. *Sizeof for Java.* JavaWorld . 26/12/2003 .
`http://www.javaworld.com/javaworld/javaqa/2003-12/02-qa-1226-sizeof.html?page=1`

[WWWa] Sun Microsystems . *java - The Java application launcher.*
`http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/java.html`

[WWWb] Webseite von ANTLR . *ANother Tool for Language Recognition.*
`http://www.antlr.org/`