Kurz & bündig	
Inhalt	Domänenspezifische Sprachen mit ANTLR v3 unter .NET erstellen
Zusammenfassung	Domänenspezifische Sprachen werden für spezielle Problemfelder entworfen und erfordern die Implementierung eines robusten und zuverlässigen Parsers. Ein Parsergenerator aus dem Java-Umfeld bietet einfache Lösungen für .NET ohne dediziertes Know-how über den Bau von Compilern.
Quellcode	Ja (C#, ANTLR v3)

Veröffentlichung: 01/2008.

Domain-Specific-Languages mit ANTLR v3

Das Parsen von Sprachen erforderte bisher tiefgreifende Kenntnisse aus dem Bereich Compilerbau. Ein Schulterblick in das Java-Lager zeigt, wie einfach und intuitiv ein zuverlässiger Übersetzer für eine domänenspezifische Sprache implementiert werden kann. Mit vollständiger Unterstützung für C# und .NET!

Manu Carus

Language Recognition, die Erkennung von Computersprachen, galt lange Zeit als Königsklasse der Informatik. Begriffe wie „lexikalische Analyse“, „rekursiver Abstieg“, „LALR(k)-Parser“ und „Token-Streams“ waren lange Zeit nur sehr erfahrenen Entwicklern vorbehalten, die ausgerüstet mit den UNIX-Werkzeugen „lex“ und „yacc“ und der Programmiersprache C, umfangreiche Compiler entwickelten. Mit ANTLR v3 steht der Entwicklergemeinde nun ein Werkzeug zur Verfügung, das die Aufgabenstellung angenehm vereinfacht.

Domain-Specific Languages

Spezielle Probleme erfordern spezielle Lösungen. Um die zunehmend komplexer werdenden Schnittstellen zwischen IT-Systemen zu verarbeiten, werden *domänenspezifische Sprachen* (DSL) geschaffen. Darunter versteht man formale Sprachen, die für ein spezielles Problemfeld (= eine Domäne) entworfen werden. Beispiele sind:

- Gen-Sequenzen
- Erkennung von Proteinmustern
- Editoren für Benutzeroberflächen
- WebML für die Webseiten-Modellierung
- u.v.m.

In den vergangenen Jahren entdeckten die einzelnen Wirtschaftsbranchen den Wert der Domain-Specific Languages für sich: Um branchenspezifische Daten zu organisieren und diese zwischen IT-Systemen, externen Dienstleistern und Geschäftspartnern automatisiert auszutauschen, werden zunehmend formale Sprachen eingesetzt. Einfache Textformate reichen längst nicht mehr aus, um die Problemstellungen in der Telekommunikationsindustrie, im Kredit- und Versicherungsgewerbe, in der Energie- und Wasserversorgung und anderen Branchen zu beschreiben:

- CSV mangelt es an Struktur: Die Datensätze sind „flach“ organisiert, und es wird keine Aussage über die semantische Korrektheit und Vollständigkeit der Daten getroffen.

- Mit XML können die Daten zwar hierarchisch organisiert werden, und auch XML Schema unterstützt die syntaktische Beschreibung der Daten. Die verarbeitende Anwendung jedoch muss *generische APIs* einsetzen (DOM, SAX, Reader), um die in dem XML-Dokument enthaltenen Daten auszulesen, zu organisieren und weiterzuverarbeiten. Eine problemspezifische Verarbeitung der Daten ist erst nach Extrahieren der einzelnen Informationen aus dem XML-Dokument möglich (Push- oder Pull-Modell).

Der Vorteil einer DSL, im Vergleich zu allgemein gültigen Datenaustauschformaten, ist die Problemspezifität: Die jeweilige Sprache ist auf ein bestimmtes Problem zugeschnitten und kann nichts anderes als eine solche Problembeschreibung darstellen. Fachspezialisten sollen das mit Hilfe dieser Sprache beschriebene Problem ohne weitere Kenntnisse lösen können.

ad-hoc-Entwurf

In diesem Artikel wird eine beispielhafte DSL entworfen und ein entsprechende Übersetzer erzeugt. Abbildung 1 veranschaulicht zu diesem Zweck ein beispielhaftes Szenario aus der Telekommunikationsindustrie: Ein Verbund von mehreren Carriern lagert die IT an einen externen Dienstleister aus. Die bei den Carriern entstehenden Geschäftsprozesse werden elektronisch an den Dienstleister weitergeleitet und dort verarbeitet (z.B. ein Neuauftrag für einen Telefon-/Internetanschluss). Der Dienstleister verwaltet die Daten der angebundenen Unternehmen in jeweils getrennten Systemen:

- Customer Relationship Management (z.B. Oracle Siebel oder Microsoft CRM)
- Enterprise Resource Planning (z.B. SAP oder Navision)
- Produktmanagement (z.B. kundenindividuelle Lösung)
- Order Management (z.B. kundenindividuelle Lösung)

Im Zuge des Outsourcings übernimmt der Dienstleister die bisher implementierten Schnittstellen. Da verschiedene Systeme von mehreren Unternehmen vereint werden, müssen naturgemäß unterschiedliche Schnittstellentechnologien berücksichtigt werden:

- Stored Procedures (PL/SQL)
- SOAP
- native C++ Schnittstellen

Der Dienstleister möchte sicherstellen, dass die Carrier nur semantisch korrekte Daten überliefern, und spezifiziert daher eine domänenspezifische Sprache. Diese Sprache drückt formal korrekt aus, welche Geschäftsprozesse verarbeitet werden können und wie die zugehörigen Geschäftsdaten zusammengesetzt sein müssen. Listing 1 zeigt am Beispiel eines Telefonauftrags, welche Auftragsdaten zusammengestellt werden müssen und wie die dahinter liegenden Systeme zu provisionieren sind.

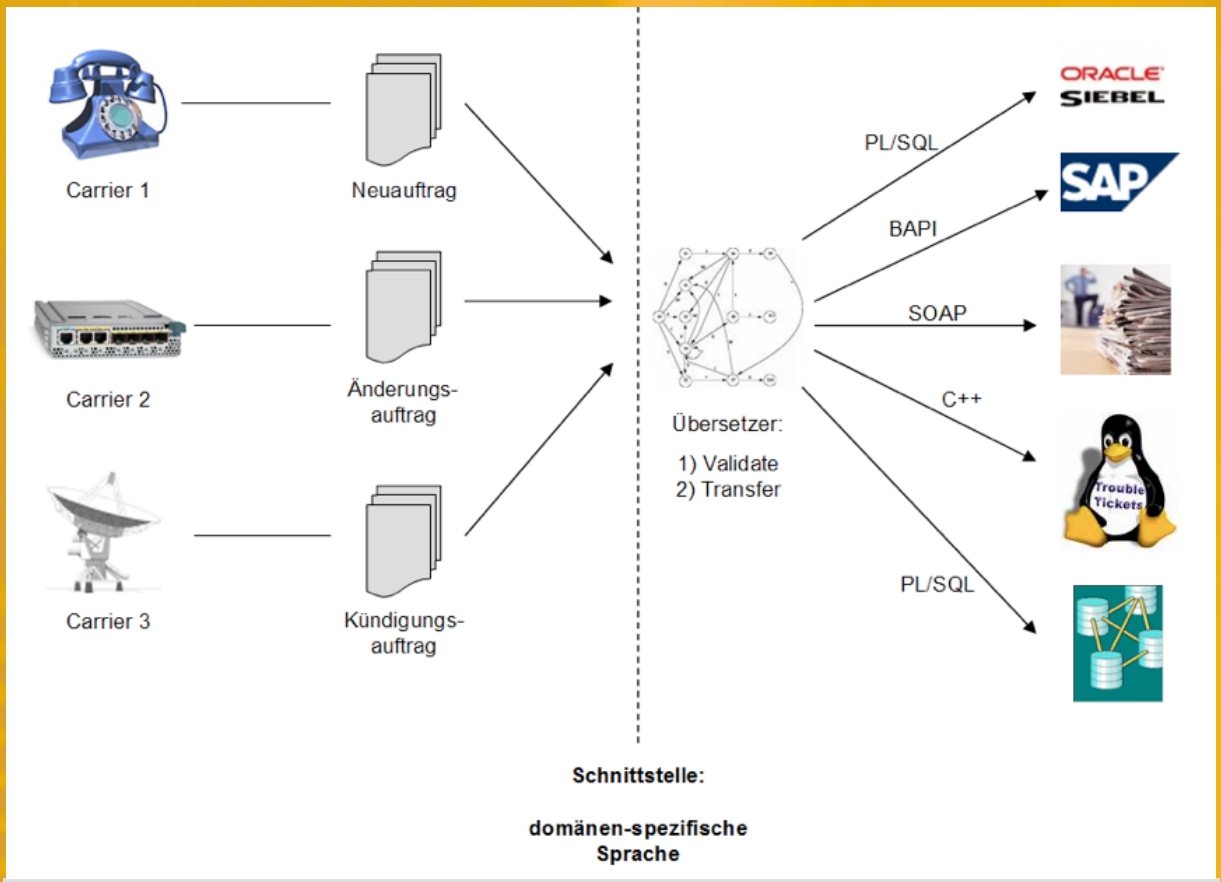

Abbildung 1: Anwendungsbeispiel

Listing 1: Telefonauftrag (formuliert in einer domänenspezifischen Sprache)

```
ORDER

  DECLARE Customer (12345678, Max, Mustermann, Musterstraße 1, 12345 Musterstadt);
  DECLARE Product  (DSL 6Mbit/s, Flat, 0221 / 1234567);
  DECLARE Features (Mailbox, EGN, Firewall, Spam-Filter);
  DECLARE Order    (0815, 03.08.2007, 16:23, Shop City);

  CREATE  Customer IN CRM USING PL/SQL (crmdb, customer.create());
  CREATE  Product  IN CRM USING PL/SQL (crmdb, product.create());
  CREATE  Features IN CRM USING PL/SQL (crmdb, features.create());
  CREATE  Order    IN CRM USING PL/SQL (crmdb, order.create());
  CREATE  Customer IN ERP USING BAPI   (sap,   CreateCustomer());
  CREATE  Product  IN PM  USING SOAP   (product-server.domain.de, customer:create);

  FORWARD Order    TO OM  USING PL/SQL (crmdb, order.forward());

  SET     Order    TO STATUS (in progress);

END.
```

In der DSL wird beschrieben, wie die einzelnen Auftragsdaten zu Geschäftsobjekten zusammenzufassen und in die Wirksysteme zu übertragen sind:

- Der Auftrag wird von den Schlüsselwörtern `ORDER` und `END.` umrahmt.
- Die Deklaration `DECLARE` definiert ein Geschäftsobjekt und fasst die zugehörigen Attribute zusammen. Ein Kunde wird bspw. über die Attribute (Kundennummer, Vorname, Nachname, Straße Hausnummer, PLZ Ort) beschrieben. Beispiel:

```
DECLARE Customer (12345678, Max, Mustermann, Musterstraße 1, 12345 Musterstadt);
```

- Das Statement `CREATE` bewirkt eine Datenübertragung in ein nachgelagertes System. Der Befehl
  ```
  CREATE Customer IN CRM USING PL/SQL (crmdb, customer.create());
  ```
  erzeugt bspw. ein neues Kundenobjekt im Customer Relationship Management-System (CRM) durch Aufruf der Stored Procedure `customer.create()` in der Datenbank `crmdb`.
- Ein Workflow wird durch Angabe des Statements `FORWARD` gestartet. Um einen Auftrag vom CRM-System in das Auftragsmanagementsystem (OM) weiterzuleiten, wird die Stored Procedure `order.forward()` in der Datenbank `crmdb` aufgerufen. Beispiel:
  ```
  FORWARD Order TO OM USING PL/SQL (crmdb, order.forward());
  ```
- Der Status eines Objekts wird über das Statement `SET` gesetzt. Bspw. kann der Auftragsstatus am Ende der Verarbeitung mit Hilfe des Befehls
  ```
  SET Order TO STATUS (in progress);
  ```
  auf „in Arbeit“ gesetzt werden:

Listing 1 zeigt nur einen kleinen beispielhaften Ausschnitt, verdeutlicht jedoch die Möglichkeiten, die domänenspezifische Sprachen bieten. Nach freiem Belieben können Befehle und Syntaxen formuliert werden, so dass neue Mini-Sprachen entstehen. Die Syntax dieser Sprachen ist auf das spezielle Problem reduziert, und die Lösung des Problems besteht in der Verarbeitung der aufgeführten Befehle.

Nun ähneln DSLs nicht zwingend kleineren Programmiersprachen, sondern können sich in einfachen Markup-Languages oder Mustererkennungen widerspiegeln.

Wie übersetzt man aber nun ein „Programm“ wie Listing 1? Wie können die einzelnen Schlüsselwörter erkannt werden? Wie geht man mit Leerzeilen, Einrückungen und Leerzeichen um? Wie extrahiert man einzelne Angaben, wie bspw. eine Kundennummer, und ordnet diese den Geschäftsobjekten zu? Wie verarbeitet man die angegebenen Befehle, indem entsprechende PL/SQL-, BAPI- oder SOAP-Aufrufe ausgeführt werden? Zu diesem Zweck wird ein robuster und zuverlässiger Übersetzer benötigt...

ANTLR

ANTLR [1] ist ein Akronym für „ANother Tool for Language Recognition“, in Anlehnung an die im UNIX-Bereich verwendeten Werkzeuge „lex“ und „yacc“ („yet another compiler compiler“) für die lexikalische und grammatikalische Analyse von Sprachen. ANTLR ist ein Parser-Generator, der für die Konstruktion von Interpretern, Compilern und anderen Übersetzern eingesetzt wird. Mit Hilfe dieses Werkzeugs können Grammatiken für vollwertige Programmiersprachen formuliert und professionelle Compiler entwickelt werden. ANTLR erzeugt Source-Codes wahlweise in Java, C#, C, C++, Objective-C, Python oder Ruby. Weitere Programmiersprachen sind in Bearbeitung [2].

Der Parser-Generator ANTLR wurde von Terence Parr (Professor für Computer Science) an der University of San Francisco entwickelt und im Mai 2007 in Version 3 unter der BSD-License als Open-Source zur Verfügung gestellt. ANTLR hat sich bereits in der Community durchgesetzt und wird mit einer funktional vollständigen IDE namens „ANTLRWorks“ ausgeliefert, einer Entwicklungsumgebung für die Implementierung und den Test von Grammatiken und Übersetzern.

Mit der Mächtigkeit, die diesem Werkzeug zugrunde liegt, ist ANTLR insbesondere für die Formulierung und Implementierung domänenspezifischer Sprachen geeignet.

Schnellstart

Um ANTLR einsetzen zu können, sind zunächst ein paar einfache Installationsschritte erforderlich (Kasten 1).

Kasten 1: ANTLR-Installation

ANTLR erfordert Java 1.5 oder höher.

Für eine vollständige Installation sind daher die folgenden Schritte erforderlich:

Installationsschritt	Beschreibung
1. Installation des JDK 6 Update 2	Das Installationspaket kann unter [3] bezogen werden. Nach der Ausführung des Setups befindet sich in dem Installationsverzeichnis `C:\Program Files\Java\jdk1.6.0_02\` die Java-Laufzeitumgebung (JRE) sowie das Development-Kit (JDK) mit dem Java-Compiler und weiteren Werkzeugen und Bibliotheken.

2. Pfad anpassen	Um eine ANTLR-Grammatik übersetzen zu können, muss der Java-Compiler auf der Kommandozeile aufgerufen werden können. Dies kann am einfachsten durch die folgende Pfadangabe erreicht werden:
	`Path = C:\Program Files\Java\jdk1.6.0_02\bin;`
	`<old_path>`

3. Installation von ANTLR v3	ANTLR v3 wird unter [4] bereitgestellt. Am besten wählt man auf der Internen-Seite in dem zweiten Kasten „ANTLR v3“ das Installationspaket „ANTLR source distribution“. Dahinter verbirgt sich die Datei http://www.antlr.org/download/antlr-3.0.tar.gz, die mittels „unzip“ und „untar“ entpackt werden muss (z.B. mit Hilfe von WinZIP oder WinRar). I.f. wird angenommen, dass die Archiv-Dateien in das Verzeichnis `c:\antlr-3.0\` entpackt wurden.

4. Classpath anpassen	Um der Java-Entwicklungsumgebung die mitgelieferten ANTLR-Bibliotheken bekanntzugeben, muss die Umgebungsvariable `CLASSPATH` angepasst werden:
	`CLASSPATH = C:\antlr-3.0\lib\antlr-2.7.7.jar;`
	`C:\antlr-3.0\lib\antlr-3.0.jar;`
	`C:\antlr-3.0\lib\antlr-runtime-3.0.jar;`
	`C:\antlr-3.0\lib\stringtemplate-3.0.jar;`
	`<old_classpath>;`

5. Installation von ANTLRWorks 1.1.2	ANTLRWorks ist eine IDE, die die Entwicklung und das Debuggen von Grammatiken vereinfacht. Das Installationspaket findet sich unter [5] („Download ANTLRWorks 1.1.2“ ⇨ „For Windows, Linux and Mac OS X“). Die dahinter liegende Datei http://www.antlr.org/download/antlrworks-1.1.2.jar ist ein selbst-ausführendes Java-Archiv und wird z.B. in dem Verzeichnis `c:\antlrworks\` abgelegt.

6. Desktop-Verknüpfung einrichten	ANTLRWorks kann durch Aufruf des Kommandozeilenbefehls `java –jar antlrworks-1.1.2.jar` gestartet werden. Am einfachsten ist es, eine Verknüpfung mit diesem Kommando auf dem Desktop einzurichten.

Nach dem Start von ANTLRWorks kann für einen einfachen Test die folgende Grammatik eingegeben werden: Über das Menu „File | New“ ist ein neues Fenster zu öffnen, in dessen Inhalt der folgende Text eingegeben wird:

```
grammar Hello;

hello   :       'hello' ID '!' {System.out.println("hello " + $ID.text + "!");} ;

ID      :       'a'..'z'+ ;
WS      :       (' ' | '\r' | '\n' | '\t')+ {$channel=HIDDEN;} ;
```

Diese Grammatik muss unter dem Dateinamen `Hello.g` abgespeichert werden (Abbildung 2). Die Grammatikregel *hello* spezifiziert eine recht triviale „Sprache“:

- Nach dem Wort *hello* folgt ein Identifizierer *ID*, gefolgt von einem Ausrufezeichen.
- Der Identifizierer *ID* setzt sich aus mehreren Kleinbuchstaben zusammen (`'a'..'z'+`).
- Whitespaces werden ignoriert (`{$channel=HIDDEN;}`).
- Nach Erkennung des Satzes „hello *ID*!“ wird auf der Konsole ein Echo ausgegeben.

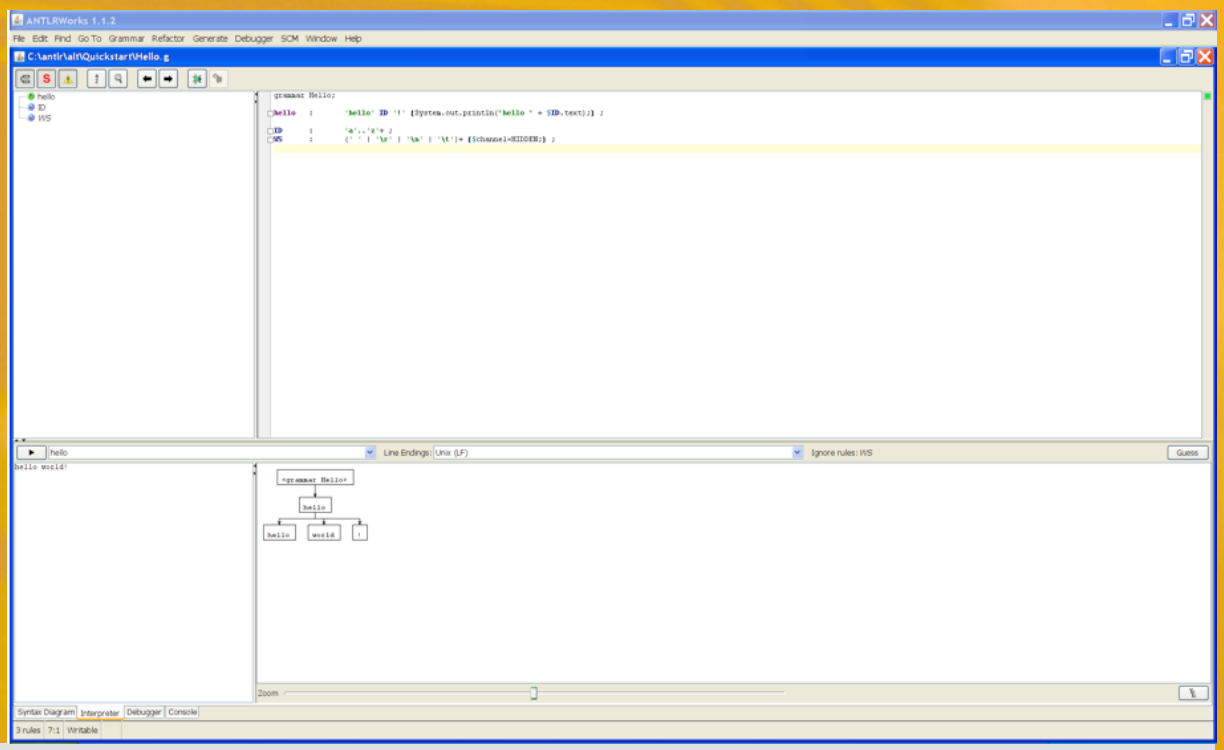

Abbildung 2: ANTLRWorks

Über die Registerkarte „Interpreter“ kann die Grammatik für eine Beispiel-Eingabe getestet werden: Nach Eingabe von „hello world!“ erzeugt die IDE eine Baumstruktur, in der die zur Erkennung des Textes verwendeten Regeln (*hello*) und Symbole („hello“, „world“ und „!“) graphisch angezeigt werden.

Grammatiken

Es ist grundsätzlich einfacher, eine Grammatik zu schreiben als einen Compiler zu entwickeln. Die Syntax einer formalen Grammatik setzt sich aus einigen einfachen Konstrukten zusammen:

1. Regeln bestehen aus einer Folge von Symbolen (*Tokens*), Aktionen und inneren Regeln.
2. Symbole sind Literale (z.B. `'hello'`) oder Bezeichner (z.B. *ID*).
3. Bezeichner konstruieren komplexe Zeichenketten, ähnlich den regulären Ausdrücken, die sich aus Wiederholungen, Alternativen, optionalen Anteilen und Zeichenausschlüssen definieren (z.B. `'a'..'z'+`).
4. Aktionen sind Code-Fragmente in der Zielsprache des zu erzeugenden Übersetzers, die an beliebiger Stelle innerhalb einer Grammatikregel oder an deren Ende eingefügt werden. Bei der Konstruktion eines Übersetzers kopiert ANTLR das Code-Fragment genau an diejenige Stelle im Ziel-Code, die der Position der Aktion in der Grammatikregel entspricht. In dem o.a. Beispiel der Regel *hello* erfolgt die Systemausgabe am Ende der Regel, d.h. nachdem alle Tokens erkannt wurden.

Der Übersetzer, den ANTLR auf Basis einer Grammatik konstruiert, durchläuft bei der Spracherkennung die folgenden Phasen:

1. Grammatikalische Analyse
   Beginnend mit der Startregel werden die in der Grammatik definierten Regeln angewandt. Diese setzen sich aus Tokens, Aktionen und inneren Regeln zusammen. Tokens werden lexikalisch analysiert, Aktionen in den Ziel-Code kopiert. Innere Regeln werden rekursiv aufgelöst.
2. Lexikalische Analyse
   Die Eingabedatei (z.B. Listing 1) wird zeichenweise eingelesen. Dabei werden die eingelesenen Zeichen nach den Vorgaben der Grammatik zu Tokens zusammengefasst. Aus den sieben aufeinanderfolgenden Zeichen „DECLARE“ wird somit das Token *DECLARE*.
3. Übersetzung
   Unterliegt die Eingabe den Regeln der Grammatik, dann konnte die Spracherkennung erfolgreich durchgeführt werden. In diesem Fall wurde parallel zur Spracherkennung der in den Grammatikregeln hinterlegte Code ausgeführt, was dem Vorgang der Übersetzung entspricht.

Die einzelnen Phasen werden nicht sequentiell ausgeführt, sondern ineinander verschränkt. Die Analyse einer Grammatikregel erfordert, dass der *Lexer* (lexikalische Analyzer) zur Erkennung von Tokens einbezogen wird. Die Übersetzung, d.h. die Ausführung von Code, erfolgt zeitgleich mit der Erkennung der Regeln durch den *Parser*.

Ohne auf die komplizierten Einzelheiten der Spracherkennung einzugehen sei angemerkt, dass die lexikalische und grammatikalische Analyse jeweils sehr komplexe Vorgänge sind: Da sich die Definition von Tokens und Regeln meist überschneiden, muss der Übersetzer verschiedene Definitionen ausprobieren und sich ggf. wieder auf den vorherigen Zustand zurück setzen („Backtracking“). In vielen Fällen müssen Zeichen aus der Eingabe vorgelesen werden („lookahead“). Und wurde eine falsche Grammatikregel ausprobiert, muss der gesamte rekursive Vorgang aufgelöst werden, damit in dem Ursprungszustand die nächstmögliche Regel angewendet werden kann.

Mit diesem Rüstzeug ausgestattet kann nun eine domänenspezifische Sprache für das in Listing 1 aufgeführte Beispiel entwickelt werden.

Grammatikalische Analyse

Listing 2 spezifiziert die Grammatikregeln für die Beispiel-DSL.

Listing 2: Grammatikregeln

```
grammar Order;

// parser

start     : 'ORDER' content 'END' '.' ;

content   : declare* statement* ;

declare   : 'DECLARE' entity ';' ;

entity    : customer | product | features | order;

customer  : 'Customer' '(' id=NUM_IDENT ','
                           first_name=CHAR_IDENT ','
                           last_name=CHAR_IDENT ','
                           street=CHAR_IDENT house=NUM_IDENT ','
                           zipcode=NUM_IDENT city=CHAR_IDENT
                         ')' ;

product   : 'Product' '(' PRODUCT ','
                          TARIFF ','
                          preselection=NUM_IDENT '/' callnumber=NUM_IDENT
                        ')' ;

features  : 'Features' '(' feature+=CHAR_IDENT (',' feature+=CHAR_IDENT)* ')' ;

order     : 'Order' '(' id=NUM_IDENT ',' DATE ',' TIME ',' SHOP ')' ;

statement : (create | forward | set) ';' ;

create    : 'CREATE' ('Customer' | 'Product' | 'Features' | 'Order') 'IN'
            SYSTEM 'USING' contract ;

contract  : plsql | bapi | soap ;

plsql     : 'PL/SQL' '(' db=CHAR_IDENT ','
                         package=CHAR_IDENT '.' procedure=CHAR_IDENT '()'
                       ')' ;

bapi      : 'BAPI' '(' erp=CHAR_IDENT ',' method=CHAR_IDENT '()' ')' ;

soap      : 'SOAP' '(' sub_domain=CHAR_IDENT '.'
                       domain=CHAR_IDENT '.'
                       tld=CHAR_IDENT ','
                       obj=CHAR_IDENT ':' action=CHAR_IDENT
                     ')' ;

forward   : 'FORWARD' 'Order' 'TO' 'OM' 'USING' plsql ;

set       : 'SET' 'Order' 'TO' 'STATUS' '(' STATUS ')' ;

// lexer (siehe Listing 3)
```

Der Inhalt dieser Datei kann direkt in *ANTLRWorks* eingegeben werden („File | New“, Copy & Paste, „File | Save As...“ unter dem Dateinamen *Order.g*). Abbildung 3 zeigt ANTLRWorks in Aktion:

- Regeln, Tokens und Literale werden mit Hilfe von Syntax-Highlighting unterschieden.
- Alle Definitionen werden in dem Bereich links oben aufgelistet (P = Produktion, L = Lexer-Symbol). Bei Klick auf einen Identifizierer springt die IDE sofort zu der entsprechenden Definition in der Grammatik.
- Im unteren Bereich wird das Syntaxdiagramm für die markierte Regel angezeigt. Aus diesem Diagramm lassen sich Wiederholungen und optionale Anteile leicht ablesen, da sie durch Pfeile dargestellt werden, die einen Pfad für die Definition festlegen.

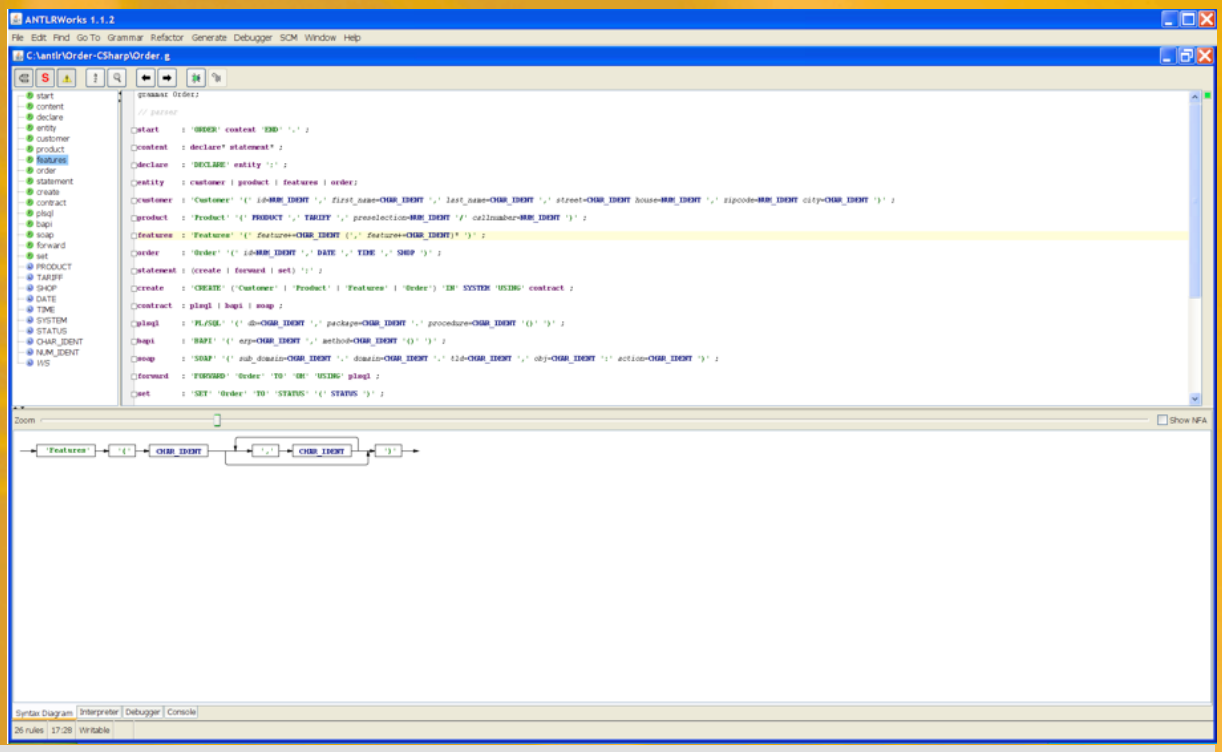

Abbildung 3: Grammatikregeln in ANTLRWorks

Die Regeln der domänenspezifischen Sprache sind *straight-forward* definiert und lassen sich einfach verstehen:

- Die Startregel *start* beschreibt den Rahmen eines Auftrags (`start: 'ORDER' content 'END' '.'`).
- Der Inhalt des Auftrags wird durch die Regel *content* festgelegt: ein Auftrag besteht aus Deklarationen und Anweisungen (`content: declare* statement*`).
- Eine Deklaration besteht aus dem Token `DECLARE`, gefolgt von der Regel *entity* und einem Semikolon (`declare : 'DECLARE' entity ';'`).
- Eine Entität ist ein Kunde, ein Produkt, eine Feature-Liste oder ein Auftrag (`entity : customer | product | features | order`).
- `Eine Kundendefinition umfaßt die Kundennummer, den Vor- und Nachnamen, die Straße, Hausnummer, PLZ und Stadt:`

```
customer : 'Customer' '(' id=NUM_IDENT ','
                          first_name=CHAR_IDENT ','
                          last_name=CHAR_IDENT ','
                          street=CHAR_IDENT house=NUM_IDENT ','
                          zipcode=NUM_IDENT city=CHAR_IDENT
                       ')'
```

Nach dem Schlüsselwort `Customer` folgt in Klammern eine Auflistung der Kundenattribute. Numerische Identifizierer werden über das Token *NUM_IDENT* erkannt, alphanumerische Identifizierer über das Token *CHAR_IDENT*. Die einzelnen Attribute werden durch Kommata voneinander getrennt, nur Straße und Hausnummer sowie PLZ und Ort sind durch Whitespaces getrennt. Um die erkannten Inhalte bei dem nachfolgenden Übersetzungsvorgang auseinanderhalten zu können, werden den Tokens *NUM_IDENT* und *CHAR_IDENT* eindeutige Bezeichner zugeordnet (z.B. *first_name*, *last_name*, usw.).

- Eine Produktdefinition enthält nach dem Schlüsselwort `Product` in Klammern die Produktbezeichnung, den Tarif sowie die bisherige Telefonnummer des Kunden:

```
product : 'Product' '(' PRODUCT ','
                        TARIFF ','
                        preselection=NUM_IDENT '/' callnumber=NUM_IDENT
                      ')'
```

Vorwahl und Rufnummer werden durch einen Schrägstrich getrennt (z.B. `0221 / 1234567`). Für die aufgeführten Produkte und Tarife sind keine freien Bezeichner zugelassen; stattdessen wird in der lexikalischen Definition eine Auswahl der zugelassenen Schlüsselwörter definiert (s.u.).

- Die mit dem Telefonanschluss bestellten Merkmale (z.B. Anrufbeantworter, Einzelverbindungsnachweis, Firewall und Spam-Filter) werden in einer komma-getrennten Liste zusammengefasst:

```
features  : 'Features' '(' feature+=CHAR_IDENT (',' feature+=CHAR_IDENT)* ')'
```

Die einzelnen Merkmale werden über die freien Bezeichner *CHAR_IDENT* zu einer Liste *feature* vom Typ *System.Collections.ArrayList* zusammengefasst.

Die weiteren Regeln in Listing 2 lesen sich nach den gleichen Vorgaben. Um die Grammatik zu vervollständigen, müssen nun die in den Regeln aufgeführten Tokens definiert werden.

Lexikalische Analyse

Listing 3 enthält die Tokendefinitionen für alle in der DSL zugelassenen Symbole:

- Produkte sind wohldefiniert und müssen einer bestimmten Grunddatenliste entnommen werden:

```
PRODUCT : ('DSL 2Mbit/s' | 'DSL 6Mbit/s' | 'DSL 18Mbit/s')
```

Gleiches gilt für die Tarife, Shops, Systeme und den Status (*TARIFF*, *SHOP*, *SYSTEM*, *STATUS*).

- Ein Datum wird in dem Format *DD.MM.YYYY* angegeben und daher als Folge von Ziffern und Punkten spezifiziert:

```
DATE : '0'..'9' '0'..'9' '.'
       '0'..'9' '0'..'9' '.'
       '0'..'9' '0'..'9' '0'..'9' '0'..'9'
```

Gleiches gilt für die Angabe von Uhrzeiten (*TIME*).

- Die restlichen Bezeichner in der Grammatik können frei gewählt werden. Das Token *NUM_IDENT* definiert numerische Bezeichner wie bspw. Hausnummer und Postleitzahl, das Token *CHAR_IDENT* alle alphanumerischen Bezeichner (z.B. Straße und Ort).

- Whitespaces werden ignoriert und können an beliebigen Stellen zwischen den Symbolen in der Eingabe eingefügt werden:

```
WS : (' ' | '\t' | '\n' | '\r')+   {$channel = HIDDEN;}
```

Listing 3: Lexikalische Regeln

```
grammar Order;

// parser (siehe Listing 2)

// lexer

PRODUCT     : ('DSL 2Mbit/s' | 'DSL 6Mbit/s' | 'DSL 18Mbit/s') ;

TARIFF      : ('Flat' | 'Minute') ;

SHOP        : ('Shop City' | 'Shop Nord' | 'Shop Süd' | 'Shop West' | 'Shop Ost') ;

DATE        : '0'..'9' '0'..'9' '.'
              '0'..'9' '0'..'9' '.'
              '0'..'9' '0'..'9' '0'..'9' '0'..'9' ;

TIME        : '0'..'9' '0'..'9' ':'
              '0'..'9' '0'..'9' ;

SYSTEM      : ('CRM' | 'ERP' | 'PM') ;

STATUS      : ('new' | 'clearance' | 'in progress' | 'finished') ;

CHAR_IDENT  : ('a'..'z'|'A'..'Z'|'ß'|'ä'|'ö'|'ü'|'Ä'|'Ö'|'Ü'|'-')+ ;

NUM_IDENT   : ('0'..'9')+ ;

WS          : (' ' | '\t' | '\n' | '\r')+   {$channel = HIDDEN;} ;
```

ANTLRWorks bietet für lexikalische Definitionen hinsichtlich Syntaxdiagrammen, Navigation und Syntax-Highlighting die gleiche Unterstützung wie für grammatikalische Regeln (Abbildung 4).

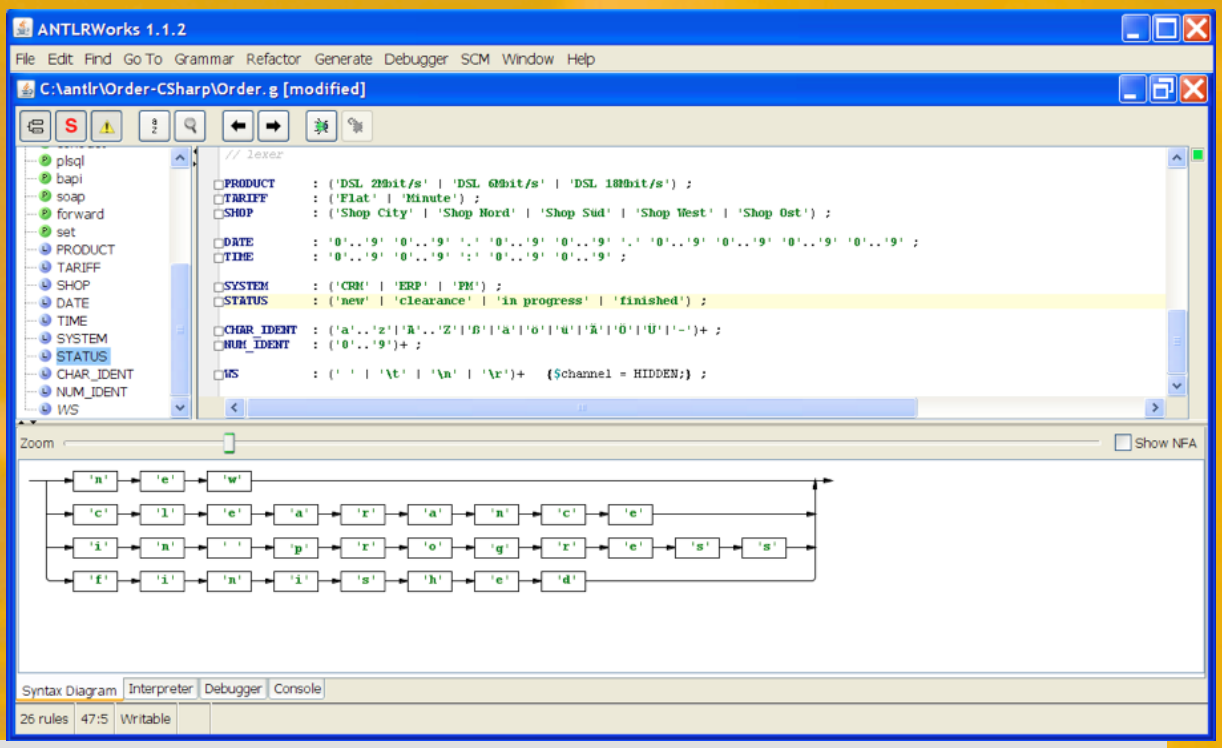

Abbildung 4: Lexikalische Regeln in ANTLRWorks

Übersetzung

Die Syntax formaler Grammatiken gestattet die Code-Ausführung nicht nur am Ende einer Regel, sondern bereits nach der Erkennung eines jeden Bestandteils der Regel, d.h. nach jedem einzelnen Token und nach jeder inneren Regel, die erfolgreich angewandt werden konnte. Aus diesem Grund erfolgt die Übersetzung bereits im Zuge der Spracherkennung. Soll in der Startregel *start* bspw. als Vorbereitung für die weitere Verarbeitung eine neue Auftragsnummer generiert und ausgegeben werden, muss die Regel wie folgt definiert werden:

```
start     : { string strOrderId = System.Guid.NewGuid().ToString(); }
            'ORDER'
            { Console.Out.WriteLine("beginning new order with id '" + strOrderId +
                                    "'"); }
            content
            'END'
            '.'
            { Console.Out.WriteLine("end of order '" + strOrderId + "'"); }
```

Das Beispiel zeigt, wie Tokens, innere Regeln und Code-Ausführungen in einer Regeldefinition miteinander verschränkt werden: Stellt der Übersetzer fest, dass die Regel *start* angewendet werden kann, weil das Token *ORDER* erkannt wurde, die Regel *content* erfüllt ist und anschließend die beiden Tokens *END* und . (Punkt) folgen, wird der Lesezeiger auf der Eingabedatei entsprechend weiter positioniert, und mit jeder Bewegung des Lesezeigers wird das entsprechende Code-Fragment im Zielcode des Übersetzers erzeugt.

Der vollständige Übersetzer für die Beispiel-Sprache, die diesem Artikel zugrunde liegt, ist aufgrund des Umfangs der Heft-CD in Listing 4 beigelegt.

Listing 4: Übersetzung

```
grammar Order;

options {language=CSharp;}

// parser

start     : { string strOrderId = System.Guid.NewGuid().ToString(); }
            'ORDER'
            { Console.Out.WriteLine("\nbeginning new order with id '" +
                                    strOrderId + "'"); }
            content
            'END'
            '.'
            { Console.Out.WriteLine("\nend of order '" + strOrderId + "'"); } ;

content   : declare* statement* ;

declare   : 'DECLARE' entity ';' ;

entity    : customer | product | features | order;

customer  : 'Customer' '(' id=NUM_IDENT ','
                           first_name=CHAR_IDENT ','
                           last_name=CHAR_IDENT ','
                           street=CHAR_IDENT house=NUM_IDENT ','
                           zipcode=NUM_IDENT city=CHAR_IDENT ')'

            { Console.Out.WriteLine("\nCustomer: " + $id.Text);
              Console.Out.WriteLine("           " + $first_name.Text);
              Console.Out.WriteLine("           " + $last_name.Text);
              Console.Out.WriteLine("           " + $street.Text);
              Console.Out.WriteLine("           " + $house.Text);
              Console.Out.WriteLine("           " + $zipcode.Text);
              Console.Out.WriteLine("           " + $city.Text);
            } ;

product   : 'Product' '(' PRODUCT ','
                          TARIFF ','
                          preselection=NUM_IDENT '/' callnumber=NUM_IDENT
                    ')'

            { Console.Out.WriteLine("\nProduct:  " + $PRODUCT.Text);
              Console.Out.WriteLine("           " + $TARIFF.Text);
              Console.Out.WriteLine("           " + $preselection.Text);
              Console.Out.WriteLine("           " + $callnumber.Text);
            } ;

features  : 'Features' '(' feature+=CHAR_IDENT (',' feature+=CHAR_IDENT)* ')'

            { int i=1;
              Console.Out.Write("\nFeatures: ");
              foreach(Object objFeature in $feature)
              {
                CommonToken tokenFeature = (CommonToken)objFeature;
                if (i>1) Console.Out.Write("           ");
                Console.Out.WriteLine(tokenFeature.Text);
                i++;
              }
            } ;

order     : 'Order' '(' id=NUM_IDENT ','
                        DATE ','
                        TIME ','
                        SHOP ')'

            { Console.Out.WriteLine("\nOrder:    " + $id.Text);
              Console.Out.WriteLine("           " + $DATE.Text);
              Console.Out.WriteLine("           " + $TIME.Text);
              Console.Out.WriteLine("           " + $SHOP.Text + "\n");
            } ;

statement : (create | forward | set) ';' ;
```

```
create    : 'CREATE' ('Customer' | 'Product' | 'Features' | 'Order')
            'IN' SYSTEM 'USING' contract ;

contract  : plsql | bapi | soap ;

plsql     : 'PL/SQL' '(' db=CHAR_IDENT ','
                         package=CHAR_IDENT '.' procedure=CHAR_IDENT '()' ')'

            { Console.Out.WriteLine("execute stored procedure '" +
                                    $package.Text + "." + $procedure.Text + "()' " +

                                    "in database '" + $db.Text + "'");

            } ;

bapi      : 'BAPI' '(' erp=CHAR_IDENT ','
                       method=CHAR_IDENT '()' ')'

            { Console.Out.WriteLine("call bapi method '" + $method.Text + "()' " +
                                    "in system '" + $erp.Text + "'");
            } ;

soap      : 'SOAP' '(' sub_domain=CHAR_IDENT '.'
                       domain=CHAR_IDENT '.'
                       tld=CHAR_IDENT ','
                       obj=CHAR_IDENT ':'
                       action=CHAR_IDENT ')'

            { Console.Out.WriteLine("invoke soap method '" +
                                    $obj.Text + ":" + $action.Text + "()' " +
                                    "at server '" + $sub_domain.Text + "." +
                                    $domain.Text + "." + $tld.Text + "'");
            } ;

forward   : 'FORWARD' 'Order' 'TO' 'OM' 'USING'
            { Console.Out.WriteLine("forward order to order management: "); }
            plsql ;

set       : 'SET' 'Order' 'TO' 'STATUS' '(' STATUS ')'
            { Console.Out.WriteLine("set order status to '" + $STATUS.Text + "'");
            } ;

// lexer

PRODUCT     : ('DSL 2Mbit/s' | 'DSL 6Mbit/s' | 'DSL 18Mbit/s') ;
TARIFF      : ('Flat' | 'Minute') ;
SHOP        : ('Shop City' | 'Shop Nord' | 'Shop Süd' | 'Shop West' | 'Shop Ost') ;

DATE        : '0'..'9' '0'..'9' '.' '0'..'9' '0'..'9' '.' '0'..'9' '0'..'9' '0'..'9' '0'..'9'
;
TIME        : '0'..'9' '0'..'9' ':' '0'..'9' '0'..'9' ;

SYSTEM      : ('CRM' | 'ERP' | 'PM') ;
STATUS      : ('new' | 'clearance' | 'in progress' | 'finished') ;

CHAR_IDENT  : ('a'..'z'|'A'..'Z'|'ß'|'ä'|'ö'|'ü'|'Ä'|'Ö'|'Ü'|'-')+ ;
NUM_IDENT   : ('0'..'9')+ ;

WS         : (' ' | '\t' | '\n' | '\r')+   {$channel = HIDDEN;} ;
```

Abbildung 5 zeigt die Ausgabe des Übersetzers, nach dem der Interpreter die Eingabe aus Listing 1 erkannt und *ANTLRWorks* einen entsprechenden Syntaxbaum konstruiert hat.

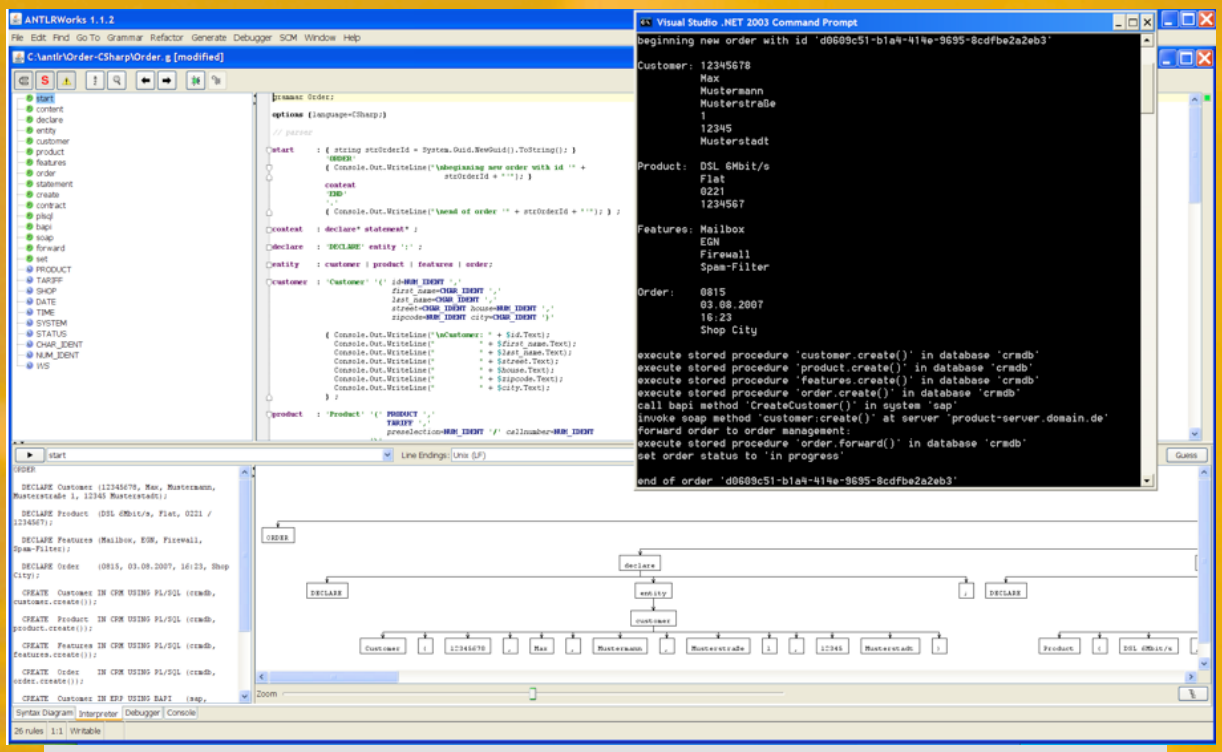

Abbildung 5: Übersetzung

ANTLR und C#

ANTLRWorks ist das Werkzeug der Wahl, wenn es um die Entwicklung und den Test formaler Grammatiken geht. Die IDE umfasst alle wesentlichen Features, die man sich von einer graphischen Entwicklungsoberfläche im Compilerbau wünscht:

- Syntaxdiagramme
- interaktiver Interpreter
- Syntaxbäume
- integrierter Debugger
- vollständiger Zugriff auf den generierten Lexer- und Parser-Code
- Visualisierung von nicht-auflösbaren Mehrdeutigkeiten
- Visualisierung der erstellten endlichen Zustandsautomaten
- Syntax-Highlighting

Zum aktuellen Zeitpunkt kann *ANTLRWorks* ausschließlich Java-Code erzeugen. Diese Einschränkung gilt allerdings nicht für *ANTLR v3*: Der Parser-Generator, der die Basis für die IDE bildet, bietet vollständige Unterstützung für C# und .NET! Für die Konstruktion eines Übersetzers in C# bietet es sich daher an, die Definitionen der Tokens und Regeln in *ANTLRWorks* vorzunehmen und anschließend von der IDE auf die Kommandozeile zu wechseln. Hierzu sind lediglich ein paar einfache Schritte notwendig (Kasten 2).

Kasten 2: ANTLR und C#

ANTLR unterstützt neben Java auch C#.

Um auf der Kommandozeile einen Testtreiber für eine Grammatik auszuführen, sind ein paar einfache Schritte erforderlich:

Schritt	Beschreibung
1. Sprache „C#“ einstellen	In der Grammatik ist die Option `options {language=CSharp;}` anzugeben (vergleiche Listing 4).
2. Übersetzer erzeugen	Der Übersetzer wird auf der Kommandozeile über den folgenden Befehl erzeugt: `java org.antlr.Tool Order.g` ANTLR generiert für die Grammatik *Order.g* einen Lexer und einen Parser in den C# Dateien *OrderLexer.cs* und *OrderParser.cs*.

3. Testtreiber implementieren	Um den Übersetzer auf der Kommandozeile mit der Beispiel-Datei in Listing 1 testen zu können, muss ein Testtreiber implementiert werden. Beispiel (*OrderTest.cs*):
	`using System;`
	`using Antlr.Runtime;`
	
	`public class OrderTest`
	`{`
	`  public static void Main(String[] args)`
	`  {`
	`  ANTLRFileStream input = new ANTLRFileStream(args[0]);`
	`  OrderLexer lexer = new OrderLexer(input);`
	`  CommonTokenStream tokens = new CommonTokenStream(lexer);`
	`  OrderParser parser = new OrderParser(tokens);`
	`  parser.start();`
	`  }`
	`}`
	Der Name der Testdatei wird dem Testtreiber als Kommandozeilenparameter übergeben.

4. Projekt kompilieren	Last but not least müssen Lexer und Parser in den Testtreiber kompiliert werden:
	`csc /target:exe`
	`  /r:C:\antlr-3.0\runtime\CSharp\bin\net-1.1\Antlr3.Runtime.dll`
	`  /r:C:\antlr-3.0\runtime\CSharp\bin\net-1.1\Antlr3.Utility.dll`
	`  /r:C:\antlr-3.0\runtime\CSharp\bin\net-1.1\antlr.runtime.dll`
	`  /r:C:\antlr-3.0\runtime\CSharp\bin\net-1.1\StringTemplate.dll`
	`  OrderLexer.cs OrderParser.cs OrderTest.cs`

5. Testtreiber starten	Anschließend kann der Testtreiber über die Kommandozeile gestartet werden: `OrderTest.exe Order.txt` Das Ergebnis der Testtreiber-Ausgabe kann Abbildung 5 entnommen werden.

Fazit

Für die Erstellung domänenspezifischer Sprachen bietet der Markt zahlreiche Tools (bspw. *Microsoft Visual Studio 2005 SDK including Domain-Specific Language Tools*). Visuelle Tools und Code-Generierungswerkzeuge unterstützen den Designer und den Entwickler bei der Spezifikation, der Implementierung und dem Test einer DSL.

ANTLR entstammt dem klassischen Compilerbau und eignet sich aufgrund seiner mächtigen Syntax ausgezeichnet für die Erstellung domänenspezifischer Sprachen. Die generierten Parser sind robust, zuverlässig und wartbar und können in verschiedenen Programmiersprachen erzeugt werden (u.a. Java, C#, C++). Die IDE *ANTLWorks* visualisiert die spezifizierten Regeln in Form endlicher Zustandsautomaten, zeichnet Syntaxbäume für die erkannten Spracheingaben und bietet ein interaktives Debugging.

Wer ein professionelles Werkzeug für die Erkennung domänenspezifischer Sprachen sucht, wird mit *ANTLR* und *ANTLRWorks* sehr schnelle und effiziente Ergebnisse erzielen können. Eine erfrischend lesbare Einführung in die Anwendung des Werkzeugs kann dem Buch „The Definitive ANTLR Reference“ von Terence Parr entnommen werden.

Manu Carus ist freiberuflicher IT-Berater. Für seine Diplomarbeit hätte er vor Jahren gerne auf ein Werkzeug wie ANTLRv3 zurückgegriffen, mußte sich jedoch der Klassiker „lex“ und „yacc“ bedienen. Sie erreichen ihn unter manu.carus@ethical-hacking.de.

Links & Literatur

[1] ANTLR Parser Generator:	http://www.antlr.org/
[2] ANTLR Code-Generation Targets:	http://www.antlr.org/wiki/display/ANTLR3/Code+Generation+Targets
[3] JDK 6 Update 2:	http://java.sun.com/javase/downloads/index.jsp
[4] ANTLR v3:	http://www.antlr.org/download.html
[5] ANTLRWorks 1.1.2:	http://www.antlr.org/works/index.html