Prof. Dr. Friedrich Steimann

Kurs 01888

Domänenspezifische Sprachen

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Inhaltsverzeichnis

Vorwort

Meiner Erfahrung nach bedarf es dreier Wiederholungen, bis ein Kurs sich eingeschwungen, also die Balance zwischen dem, was sein Autor vermitteln will, und dem, was seine Leser zu wissen wünschen, gefunden hat. Dabei gilt es nicht nur, zwischen Autor und Lesern auszutarieren, sondern auch unter den Lesern selbst, denn die Leserschaft ist selten homogen.

Bei dieser Version des Kurses handelt es sich um die erste; wir sind also bei Wiederholung 0. Es ist also zu erwarten, dass der Kurs noch nicht richtig rund läuft, sei es, weil das didaktische Konzept nicht klar wird, sei es, weil der rote Faden fehlt, oder sei es, weil die durch die Übungsaufgaben entstehende Arbeitsbelastung nicht passt. All das sind erwartete Baustellen.

All denen, denen der Kurs zu umfangreich scheint, möchte ich dennoch eines mitgeben: Dieser Kurs muss nach Maßgabe des ECTS einen Arbeitsaufwand von 300 Stunden verursachen. Dies entspricht siebeneinhalb 40-Stunden-Wochen, also einer Zeitspanne, die man — berufstätig zumal — in einem Semester nur schwer erübrigen kann. Seien Sie also nicht frustriert, wenn es nicht so schnell voran geht wie von Ihnen erhofft.

Basistext und Leittext

Dieser Kurs basiert auf dem Buch „DSL Engineering“ von Markus Völter. Dieses Buch wird im folgenden *Basistext* genannt. Dementsprechend handelt es sich bei dem hier vorliegenden Text um einen *Leittext*, der Sie durch den Basistext führen soll. Abweichend von anderen Basistext/Leittext-Paaren, die Sie vielleicht schon kennen, kann dieser Leittext aber weitgehend isoliert vom Basistext gelesen werden. Dabei soll der Leittext jedoch keinesfalls als Zusammenfassung des Basistextes verstanden werden, die zu beherrschen für eine erfolgreiche Prüfung ausreichend ist — er eröffnet eher eine andere Perspektive auf die Inhalte des Basistextes.

Der Basistext ist sog. Donationware. Sie können ihn für wenig Geld als Print-on-demand-Version kaufen oder über `http://dslbook.squarespace.com/` als PDF herunterladen.

Warum ein Kurs zu domänenspezifischen Sprachen?

Der Begriff der domänenspezifischen Sprache ist ein trendiger. Mit ihm wird eine bestimmte Menge sog. Softwaresprachen (also Programmier- und Modellierungssprachen) von Allzwecksprachen (engl. General Purpose Languages, GPL) abgegrenzt. Das Abgrenzungsmerkmal ist, wie die Benennung nahelegt, das Einsatzgebiet der Sprachen: Bei domänenspezifischen ist es eine bestimmte, vergleichsweise schmale Anwendungsdomäne, bei Allzwecksprachen ist kein spezielles Anwendungsgebiet vorgesehen.

Es gibt eine lang anhaltende Kontroverse bezüglich der Sinnhaftigkeit von DSLs. Auch wenn dieser Kurs sich dieser Debatte nicht unterwerfen will, so soll sie

doch nicht verschwiegen werden. Es stellt sich nämlich in der Tat die Frage, ob die Investition in die Entwicklung und die Pflege von DSLs sinnvoll ist, wenn man den technischen Aufwand, den Lern- und Lehraufwand sowie die Unwägbarkeit der zukünftigen Entwicklungen sowohl in der Anwendungsdomäne selbst als auch der mithilfe der DSLs dafür zu entwickelnden Produkte zusammenzählt:

- Zum technischen Aufwand zählt neben der Entwicklung eines Compilers oder Interpreters heute fraglos die Einbettung der Sprache in eine sog. integrierte Entwicklungsumgebung (engl. Integrated Development Environment, IDE) mit Funktionen wie Source-level debugger, automatischer Vervollständigung und Korrekturen sowie Refaktorisierungswerkzeugen.
- Zum Lern- und Lehraufwand zählt nicht nur die Zeit, die benötigt wird, eine neue (wenn auch beschränkte) Sprache zu erlernen, sondern auch der Aufwand, eine Sprachdokumentation zu erstellen, sowie der, der benötigt wird, ein Forum o. Ä. zu betreiben, in dem Nutzer der Sprache andere um Hilfestellungen und Tipps bitten können. Gerade für letzteres braucht man aber eine kritische Masse von Teilnehmern, die sich für eine bestimmte DSL eben nur schwer erreichen lässt.
- Nicht zuletzt ist eine DSL naturgemäß anfälliger für Änderungen in der Domäne — ergeben sich darin Änderungen, muss früher oder später auch die DSL nachgeführt werden. Dazu kommen Änderungen der Plattform (Betriebssystemwechsel etc.), an die Compiler oder Interpreter angepasst werden müssen. Bekommt man im Falle von Allzwecksprachen diese Anpassungen gewöhnlich im Rahmen von Serviceverträgen oder gar gratis, kann die Anpassung von DSLs ziemlich teuer werden. Zudem besteht die Gefahr, dass die Entwicklung einer DSLs der ihrer Umgebung hinterherhinkt. Schlechte Stimmung ist beinahe immer die Folge.

Interessanter als diese Diskussionen sind aber die Techniken, die zur Entwicklung und Umsetzung von DSLs heute herangezogen werden und die Sie in diesem Kurs kennenlernen sollen. Diese Techniken sind nämlich nicht an DSLs, sondern an formale Sprachen, oder *Softwaresprachen*, allgemein gebunden. DSLs, die in aller Regel eher kleine Sprachen darstellen, sind ein geeignetes Vehikel, um diese Techniken mit überschaubarem Aufwand kennenzulernen.

Im Kurs betrachtete Werkzeuge

Der Basistext setzt auf drei verschiedene Werkzeuge zur Umsetzung von DSLs: Spoofax, eine Werkzeug-Suite, die an der Technischen Universität Delft in den Niederlanden entwickelt wird, XText, ein im Eclipse-Umfeld weit verbreitetes Framework für Softwaresprachen, so wie das Meta Programming System (MPS) der Firma JetBrains, die ihr Geld und ihre Bekanntheit vor allem mit der Entwicklung einer Java-IDE (IntelliJ IDEA) erworben hat. Abweichend vom Basistext wird in diesem Leittext Spoofax durch einen sehr viel einfacheres, grundlegenderes Werkzeug ersetzt: die sogenannten definiten Klauselgrammatiken (engl. definite clause grammars, DCGs) der Programmiersprache Prolog. Dies soll ihnen ermöglichen, einfache Techniken der Sprachumsetzung zu erlernen,

die nicht auf einem (mitunter doch recht komplexen) Toolstack sogenannter *Language workbenches* aufbauen.

Warum ECLiPSe-Prolog?

Als Prolog-System verwenden wir ECLiPSe (`http://eclipseclp.org`), obwohl sich aus der aktuell vorliegenden Version des Kurses kein zwingender Grund dafür ergibt. Tatsächlich sollten Sie alle Darstellungen und Aufgaben auch mit einem anderen Prolog-Dialekt nachvollziehen bzw. bearbeiten können. Allerdings erlaubt ECLIiPSe-Prolog es (genau wie SWI-Prolog), die logische Programmierung mit dem Constraint-Programmieren zur sog. *constraint-logischen Programmierung* (*CLP*) zu verbinden. Auch wenn dies in der aktuellen Version des Kurses noch nicht ausgenutzt wird, steht es Ihnen natürlich frei, bei der Lösung der Aufgaben damit zu arbeiten.

Voraussetzungen

Dieser Kurs verlangt zumindest ein gewisses Grundverständnis davon, was formale Sprachen sind und welche Bedeutung sie in der Informatik haben. Für die Bearbeitung der Einsendeaufgaben sind weiterhin profunde Kenntnisse der Programmiersprache Java vonnöten. Hilfreich sind schließlich praktische Programmiererfahrungen, insbesondere die Wertschätzung von heutigen integrierten Entwicklungsumgebungen (und das Interesse daran was in ihnen steckt).

Ziele

Als Minimalziel sollen Sie durch diesen Kurs einen Zugang zu den Problemen und Möglichkeiten der Entwicklung domänenspezifischer Sprachen (inkl. dazugehöriger Werkzeuge) mit heutigen Mitteln finden. Idealerweise haben Sie am Ende eine Anleitung gefunden, eine solche Sprache mit brauchbarer IDE-Unterstützung selbst zu entwickeln. Dieses Ziel erreichen Sie natürlich am ehesten, wenn Sie intrinsisch motiviert sind — insofern sollten Sie die Bearbeitung der freiwilligen Übungsaufgabe, nach der es gilt, eine DSL aus Ihrem Umfeld zu identifizieren und umzusetzen, ernsthaft in Betracht ziehen.

1 Einführung und Grundlagen von DSLs

In dieser Kurseinheit geht es um die Grundlagen von DSLs und ihre Implementierung, wie sie in den Kapiteln 2 („Introduction to DSLs“) und 3 („Conceptual Foundations“) des Basistextes behandelt werden. Ergänzend (und damit Sie durch den vielen Text und seinen Mangel an Handfestem nicht falsch eingestellt werden) lernen Sie in diesem Leittext die Programmiersprache Prolog kennen, soweit es für diesen Kurs (genauer: diesen Leittext) notwendig ist.

---

2 Elementare Datenstrukturen und Constraints

In dieser Kurseinheit geht es weiter um Grundlegendes, und zwar um die Freiheitsgrade und Leitplanken bei der Entwicklung von DSLs sowie Aspekte des Entwicklungsprozesses, wie sie in den Kapiteln 4 („Design Dimensions“), 5 („Fundamental Paradigms“) und 6 („Process Issues“) des Basistextes behandelt werden. Ergänzend lernen Sie in diesem Leittext (mit ähnlicher Motivlage wie in der vorhergehenden Kurseinheit), was genau Sie von Prolog wissen müssen, um es für die Umsetzung von DSLs verwenden zu können.

---

3 Spezifikation der Syntax von DSLs

In dieser Kurseinheit geht es endlich richtig los. Konkret geht es um die Darstellung von Sätzen einer DSL (in der Regel Programmen) in Form eines abstrakten Syntaxbaums und wie man dazu gelangt. Dies wird in Kapitel 7 („Concrete and Abstract Syntax“) des Basistextes unter Berücksichtigung der Language workbenches Xtext und MPS (Spoofax können Sie überspringen) handelt. Ergänzend lernen Sie in diesem Leittext, wie Sie mithilfe von definiten Klauselgrammatiken (engl. Definite clause grammars, DCGs*) in Prolog Sätze von kontextfreien und kontextsensitiven Grammatiken parsen und generieren und dabei den abstrakten Syntaxbaum gleich mitgewinnen können.*

---

4 Spezifikation der statischen Semantik von DSLs

In dieser Kurseinheit geht um zusätzliche Regeln und Bedingungen, denen Sätze einer DSL (Programme) über die ihrer Syntax hinaus genügen müssen. Dies wird in den Kapiteln 8 („Scoping and Linking“), 9 („Constraints“) und 10 („Type Systems“) des Basistextes behandelt. Ergänzend lernen Sie in diesem Leittext, wie diese Regeln und Bedingungen als Teil einer Attributgrammatik spezifiziert werden können.

---

5 Dynamische Semantik von DSLs: Interpreter und Übersetzer

In dieser Kurseinheit geht es um die Ausführung von Sätzen einer DSL, sofern die DSL dafür vorgesehen ist. Dafür haben sich zwei Wege etabliert: die Übersetzung in einen ausführbaren Formalismus (im Basistext Kapitel 11, „Transformation and Generation“) und die Interpretation (Kapitel 12, „Building Interpreters“). Beide Wege werden in diesem Leittext mit Prolog beschritten.

---

6 Code-Vervollständigung und -korrektur

In dieser Kurseinheit geht es um zusätzliche Dienste, die zur erwarteten Infrastruktur einer DSL gehören (im Basistext Kapitel 13 „IDE Services“). Der Leittext zeigt, wie diese Services in Prolog ohne großen Aufwand aus einer bestehenden Grammatik abgeleitet werden können.

---

7 Debugging

In dieser Kurseinheit geht es um das Testen und Debuggen von DSLs, wie es im Basistext, Kapitel 14 („Testing DSLs“) und 15 („Debugging DSLs“), behandelt wird. Dieser Leittext deutet an, wie Debugger für interpretierte und kompilierte Sprachen in Prolog implementiert werden können.

---

3 Spezifikation der Syntax von DSLs

In dieser Kurseinheit geht es endlich richtig los. Konkret geht es um die Darstellung von Sätzen einer DSL (in der Regel Programmen) in Form eines abstrakten Syntaxbaums und wie man dazu gelangt. Dies wird in Kapitel 7 („Concrete and Abstract Syntax") des Basistextes unter Berücksichtigung der Language workbenches Xtext und MPS (Spoofax können Sie überspringen) handelt. Ergänzend lernen Sie in diesem Leittext, wie Sie mithilfe von definiten Klauselgrammatiken (engl. definite clause grammars, DCGs*) in Prolog Sätze von kontextfreien und kontextsensitiven Grammatiken parsen und generieren und dabei den abstrakten Syntaxbaum gleich mitgewinnen können.*

3.1 Definite Klauselgrammatiken

Wenn man einen Satz einer Sprache als eine Liste von Worten auffasst, dann lässt sich in Prolog auf einfache Weise ein Parser und Generator bauen, der die Sätze der Sprache akzeptiert bzw. generiert. Weiterhin kann dieser Parser und Generator leicht so erweitert werden, dass er zu einem akzeptierten Satz den abstrakten Syntaxbaum bzw. zu einem abstrakten Syntaxbaum den dazugehörigen Satz liefert.

Wir beginnen mit einer einfachen, nicht *linksrekursiven*[8] (s. dazu Basistext, Seite 182 ff.) *Grammatik*

```
44    Expr ::= Term
45     | Term "+" Expr
46    Term ::= Fact
47     | Fact "*" Term
48    Fact ::= "0"
49     | "1"
```

für einfache arithmetische Ausdrücke (Addition und Multiplikation) über die Zahlen 0 und 1. (Die Namen der Nichtterminale `Term` und `Fact` sind die in der Literatur gebräuchlichen und haben nichts mit Prolog-Termen und -Fakten zu tun!) Wir erkennen in den Regeln die Struktur eines Prolog-Programms

```
50    expr :- term.
51    expr :- term, "+", expr.
52    term :- fact.
53    term :- fact, "*", term.
54    fact :- 0.
55    fact :- 1.
```

Dieses Programm ist aber nicht nur syntaktisch inkorrekt (Strings wie `"+"` sind keine Prädikate und können daher nicht auf oberster Ebene stehen), sondern kann auch nichts: Es akzeptiert keine Eingabe und produziert auch keine Ausgabe. Die folgende Änderung behebt dies:

---

8 Linksrekursivität führt in Prolog zum Stapelüberlauf — erkennen Sie, warum?

```
56    expr(In, Out) :- term(In, Out).
57    expr(In, Out) :- term(In, Tmp1), Tmp1 = [+|Tmp2], expr(Tmp2, Out).
58    term(In, Out) :- fact(In, Out).
59    term(In, Out) :- fact(In, Tmp1), Tmp1 = [*|Tmp2], term(Tmp2, Out).
60    fact(In, Out) :- In = [0|Out].
61    fact(In, Out) :- In = [1|Out].
```

Hier ist jedes Nichtterminal um zwei Argumente ergänzt worden, eines, das die Ein- bzw. Ausgabe (im folgenden nur noch als Restsatz bezeichnet) vor Anwendung des Nichtterminal hält und eines für danach; jedes Terminal ist ersetzt worden durch eine Unifikation des Restsatzes vor seiner Produktion mit einer Liste, die mit dem Terminal beginnt und auf das der neue Restsatz folgt. Innerhalb des Rumpfes einer Regel ist dann der Restsatz nach einer Anwendung des vorhergehenden Terminals oder Nichtterminals der Restsatz vor einer Anwendung des darauffolgenden Terminals oder Nichtterminals. Bevor Sie jetzt versuchen, dies anhand vorstehender Sätze zu verstehen, vollziehen Sie die Funktionsweise lieber anhand des Prolog-Interpreters „live“ nach.

Jedenfalls lassen sich mithilfe dieses Prolog-Programms Sätze wie etwa `[0, +, 1, *, 1]` akzeptieren, die dazu allerdings in eben dieser Listenform vorliegen müssen.[9] Dazu ruft man das Startsymbol der Grammatik, `expr`, mit der obigen Liste und einer leeren Liste als Argumente auf, also

```
62    :- expr([0, +, 1, *, 1], []).
```

Die leere Liste sorgt dafür, dass auch tatsächlich alle Terminalsymbole des Satzes akzeptiert werden müssen.

rücksetzender Parser

Wenn Sie das Akzeptieren der Eingabe `[0, +, 1, *, 1]` durch das obige Prolog-Programm nachvollziehen, stellen Sie fest, dass der Parse-Vorgang auf das *Zurücksetzen* (Backtracking) angewiesen ist. So führt nämlich zunächst die Anwendung der ersten Regel (Zeile 56) und rekursiv der ersten Regeln für `term` (Zeile 52) und `fact` (Zeile 54) zum Erfolg: `0` ist tatsächlich das erste Atom der Eingabeliste. Da aber die Restliste nicht leer ist, scheitert diese Anwendung letztlich doch und der Parser muss zur nächsten Regel zurücksetzen. Für den Autor der Grammatik ist das kein Problem; für den Nutzer kann es aber eines werden, weil das rücksetzende Parsen im schlimmsten Fall exponentielle Komplexität besitzt und somit sehr langsam werden kann. Rücksetzende Parser sind daher bei Compilerbauern nicht besonders beliebt; für unsere Zwecke (DSLs!) sollen sie aber reichen.

Selbsttestaufgabe 3.1

Ersetzen Sie in Zeile 62 die leere Liste `[]` durch `X`. Ermitteln Sie alle Ausgaben und erklären Sie Ihre Beobachtung.

---

[9] Zum Akzeptieren von Strings s. Kapitel 2.1.3.

Ruft man das Startsymbol hingegen mit `expr(S, [])` auf, so liefert der Prolog-Interpreter per Backtracking alle Sätze der Sprache als Werte von `S`. Wie Sie also sehen, fungiert das Prolog-Programm, das die obige Grammatik umsetzt, sowohl als Parser als auch als Generator.

akzeptieren und generieren mit derselben Grammatik

Selbsttestaufgabe 3.2

Überlegen Sie sich, ob Suche beim Parsen oder beim Generieren die größere Rolle spielt. Woran machen Sie das fest?

Die obige Form der Implementierung einer Grammatik in Prolog ist nicht besonders schön — die beiden Argumente, die immer mitgeschleppt bzw. weitergereicht werden müssen, behindern die Lesbarkeit. Wie ihre generische Benennung jedoch nahelegt, sind die Argumente auch gar nicht notwendig: Sie können automatisch generiert werden.

Genau dies steckt hinter der Einführung sog. **definiter Klauselgrammatiken** (engl. *definite clause grammars*, **DCGs**) in Prolog. Deren Definitionen sehen zunächst wie ganz normale Prozedurdefinitionen aus, verwenden aber statt des „`:-`“ ein „`-->`“, um eine Produktionsregel darzustellen. Außerdem werden Terminalsymbole als Listen angegeben (die allerdings in der Regel nur ein Element, das jeweilige Wort, enthalten). So wird aus obiger Grammatik

```
63    expr --> term.
64    expr --> term, [+], expr.
65    term --> fact.
66    term --> fact, [*], term.
67    fact --> [0].
68    fact --> [1].
```

Wie man sieht, wird hier auf die Ein- und Ausgabeargumente verzichtet — diese werden beim Einlesen der Regeln vom Prolog-Interpreter automatisch hinzugefügt. Um ein solches DCG-Programm zur Ausführung zu bringen, bedarf es allerdings der Verwendung eines speziellen Prädikats; in ECLiPSe-Prolog heißt dies `phrase` und ist dreistellig. So parst obige Grammatik etwa mittels des Aufrufs `phrase(expr, [0, +, 1, *, 1], [])` den Satz, der ihm im mittleren Argument übergeben wurde.

3.2 Konstruktion des abstrakten Syntaxbaums

Für die Implementierung einer DSL reicht es in der Regel nicht aus, zu wissen, ob ein gegebener Satz zur Sprache gehört — man will auch wissen, wie der dazugehörige abstrakte Syntaxbaum (S. 178 ff. im Basistext) aussieht. Dazu muss man dann doch Argumente an die Symbole der Grammatik anhängen, die den Baum aufnehmen, der die (erfolgreiche) Abarbeitung der Regeln repräsentiert (ganz so, wie es im Beispiel von Kapitel 2.1.1 schon vorgeführt wurde).

Die Erweiterung kann zunächst schematisch erfolgen und sieht dann so aus:

```
69    expr(exprN(Term)) --> term(Term).
```

```
70    expr(exprN(Term, +, Expr)) --> term(Term), [+], expr(Expr).
71    term(termN(Fact)) --> fact(Fact).
72    term(termN(Fact, *, Term)) --> fact(Fact), [*], term(Term).
73    fact(factN(0)) --> [0].
74    fact(factN(1)) --> [1].
```

Hierbei ist der Name der Funktoren, die die Knoten des Baums repräsentieren sollen, jeweils aus dem Namen des expandierten Nichtterminals und „N“ (für Node) zusammengesetzt. Der Aufruf `phrase(expr(T), [0, +, 1, *, 1], [])` liefert dann

```
75    T = exprN(termN(factN(0)), +, exprN(termN(factN(1), *, termN(factN(1)))))
```

Dabei handelt es sich aber lediglich um den konkreten Ableitungs- oder Syntaxbaum — eine Abstraktion ist nicht erkennbar. Diese erfolgt, indem man überflüssige Knoten (Funktionsanwendungen) weglässt (wobei sich natürlich die Frage stellt, was genau überflüssig ist — dies entscheidet in der Regel erst die Verwendung des AST, oder die Semantik der Sprache). Wenn wir davon ausgehen, dass das Ziel lediglich die Auswertung des Ausdrucks ist, für dessen Darstellung die Terminalsymbole der Operatoren `+` und `*` als Knoten ausreichen, kann man die Regeln zu

```
76    expr(Term) --> term(Term).
77    expr(+(Term, Expr)) --> term(Term), [+], expr(Expr).
78    term(Fact) --> fact(Fact).
79    term(*(Fact, Term)) --> fact(Fact), [*], term(Term).
80    fact(0) --> [0].
81    fact(1) --> [1].
```

vereinfachen, was bei obiger Anfrage zu

```
82    T = +(0, *(1, 1))
```

führt.[10] Im allgemeinen Fall wird man aber mehr Information als nur die Terminalsymbole in den Knoten halten wollen; so kann beim Parsen beispielsweise Typinformation anfallen (0 und 1 sind Integer), die bei der weiteren Verarbeitung ausgenutzt werden kann (um zum Beispiel bei der Auswertung eine Wahl zwischen Integer- und Fließkommaoperationen zu treffen). Auch ist es denkbar, dass ein Knoten des konkreten Syntaxbaums (der ja in letztem Beispiel nur noch implizit im Baum des bisherigen Programmablaufs codiert ist) durch mehrere im abstrakten ersetzt wird, beispielsweise um zwei syntaktische Varianten desselben Sprachkonstrukts zu vereinheitlichen.

---

[10] Tatsächlich ist die Ausgabe `0 + 1 * 1` , da in Prolog die Funktoren `+` und `*` standardmäßig als Infix-Operatoren definiert sind und der Interpreter diese Definition bei Ein- und Ausgabe berücksichtigt.

Selbsttestaufgabe 3.3

Können Sie obige DCG auch zum Generieren von Ausdrücken anhand eines vorgegebenen ASTs verwenden? Probieren Sie es aus!

3.3 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 3.1 (Seite 24)

Da nun kein leerer Satz mehr übrigbleiben muss, können auch Satzanfänge akzeptiert werden, die dafür allerdings ebenfalls für sich genommen wohlgeformt sein müssen.

Selbsttestaufgabe 3.2 (Seite 25)

Beim Parsen, den hier kommt das Zurücksetzen, das ja das Anzeichen einer Suche ist, regelmäßig zum Einsatz (beim Generieren nur, wenn der nächste Satz erzeugt werden soll).

Selbsttestaufgabe 3.3 (Seite 27)

Ja, es geht! Man kann also die Grammatik auch zum Linearisieren eines abstrakten Syntaxbaums verwenden (sog. *Pretty printing*).