Domain-specific languages: Examples

Ausarbeitung zum Vortrag im Rahmen des Seminar
„Softwareentwicklung in der Wissenschaft“

Jonas Gresens

Betreuung durch Dr. Julian Kunkel

27. September 2015

1 Domain-specific Languages

Domain-specific language (noun): a computer programming language of limited expresspiveness focused on a particular domain.

∼ Martin Fowler, 2010

Als domain-specific language (DSL) wird eine spezielle (Programmier-)Sprache bezeichnet, die im Hinblick auf einen bestimmten Problembereich, die sog. *Domäne*, entwickelt wurde und den Zweck hat die Arbeit innerhalb dieser Domäne zu vereinfachen. Eine solche Domäne ist beispielsweise die Beschreibung von Regeln für den Aufbau von Zeichenfolgen mit regulären Ausdrücken als passende DSL.
Die zwingend vorhandene Domänenbindung unterscheidet DSLs von den mächtigeren (turing-vollständigen) *general purpose languages* (GPLs) wie z.B. C oder Java, die für jegliche Problemstellungen/-bereiche entwickelt wurden.
Die Implementierung der Lösung eines Problems in einer Domäne wird durch die Nutzung einer geeigneten DSL deutlich vereinfacht, da diese gerade auf solche Probleme zugeschnitten ist und die Implementierung dadurch weniger Aufwand benötigt: In der DSL MATLAB[1] benötigt beispielsweise die Multiplikation zweier Matrizen weder den Import von bestimmten Math-Headern, noch eine spezielle Typ-Definition der Matrizen oder die Nutzung einer bestimmten Funktion für die tatsächliche Matrixmultiplikation, wie es bei Benutzung der GPL C notwendig gewesen wäre.
Der durch die Nutzung einer DSL erzielte Langzeit-Effekt ist die *Produktivitätssteigerung* der domänentypischen Arbeit.

[1] MATLAB eignet sich sehr gut für mathematische Probleme

Als Ergebnis mehrerer Diskussionen mit meinem Betreuer unterscheide ich in der Domäne „Softwareentwicklung“ zwischen zwei verschiedenen Gruppen von DSLs:

- DSLs zur *abstrakteren Entwicklung eines Programms/-moduls* (kurz *abstrahierende* DSLs) vereinfachen die Implementierung des Programms, indem sie die Semantik des Codes von den technischen Details der Zielplattform entkoppeln. Abstrahierende DSLs werden daher oft fürs Prototyping und zur Entwicklung von möglichst plattformunabhängigem Code verwendet.
- DSLs zur *vereinfachten Handhabung der technischen Aspekte* der Implementierung (kurz *technische* DSLs) hingegen vereinfachen die zusätzlich zur Implementation anfallenden Arbeiten wie z.B. Code-Dokumentation, Tests und Parallelisierung, die unabhängig von der domänenspezifischen Semantik des Quellcodes sind. Technische DSLs wie z.B. XML werden darüber hinaus oft für den Austausch von Informationen zwischen verschiedenen Anwendungen verwendet.

1.1 Entwicklungsstufen

Speziell bei der Entwicklung von wissenschaftlicher Software existiert die Programm-Lösung mit der Zeit auf verschiedenen interdisziplinären Stufen.

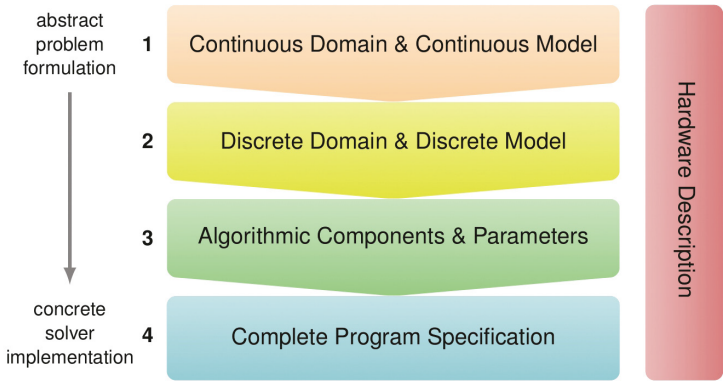

Abbildung 1: Schichten in ExaStencils [ES14]

Diese Stufen teilen sich wie folgt auf die verschiedenen, bei der Entwicklung involvierten, Personen auf:

- Stufe 1 & 2: wissenschaftliche Domäne, vom Fachwissenschaftler behandelt
- Stufe 2 & 3: mathematische Domäne, vom Mathematiker behandelt
- Stufe 3 & 4: technische Domäne, vom Informatiker behandelt

Parallel zu den einzelnen Stufen in der Entwicklung existieren entsprechende DSLs, die speziell auf die Anforderungen der jeweiligen Stufe zugeschnittene sind.

2 Überblick

2.1 Klassifikation

DSLs lassen sich nach verschiedenen Eigenschaften klassifizieren:

- Jede DSL erfüllt einen *Zweck*, da sie im Hinblick auf die Problemstellungen einer bestimmten Domäne entwickelt wurde. Ein Beispiel ist die einfachere und automatisierte Kompilierung von geänderten Quellcode-Dateien über Buildsysteme wie Make.

- Passend zu den verschiedenen Arbeitsweisen der einzelnen Wissenschaftsrichtungen existieren verschiedene *Umsetzungsarten*:
  - Die Implementation als *Untermenge* einer bereits existierenden GPL wie C oder Java ist am einfachsten und erfordert vom erfahrenen Nutzer keine große Umstellung. Eine solche DSL teilt jedoch normalerweise alle Kritikpunkte ihrer Wirts-GPL, die jegliche Konventionen wie z.B. die Syntax vorgibt.
  - Eine andere Möglichkeit ist die *Erweiterung* einer bereits existierenden GPL durch neue Sprachelemente, die zum Beispiel die Details der parallelen Implementierung des Programms vor dem Nutzer verbergen. Diese neuen Sprachelemente erfordern zwar oftmals eine aufwendigere Anpassung des Compilers, ermöglichen jedoch andererseits einen direkteren Umgang mit der tatsächlichen Problemstellung.
  - Am ambitioniertesten ist die vollständige Entwicklung einer *neuen Sprache*, die auf die Domäne und deren typische Arbeitsart zugeschnitten ist. Dieser Ansatz ist jedoch auch der Aufwendigste, da eine neue Sprache neben der entsprechenden Tool-Chain (inklusive Compiler) auch von den Nutzern das Erlernen der Sprache selbst fordert.

- Ein weiteres Merkmal ist die *Einbettung* des DSL-Codes in den Programm-Code, welche auf zwei verschiedene Arten geschehen kann:
  - *code embedded* bedeutet, dass der DSL-Code in der gleichen Datei wie der restliche Code geschriebenen steht
  - *external* heißt, dass der DSL-Code in einer eigenen Datei steht und der erst beim Kompilieren mit anderem Code zusammengefügt wird

- Sollte die DSL als Erweiterung einer GPL implementiert worden sein, so ist ihre *Rückwärtskompatiblität* von großer Relevanz, da die DSL ansonsten die Portabilität von vermischtem Code stark einschränkt.

Zwei Anmerkungen zur Klassifikation:

! Die Implementierungsdetails verhindern manchmal eine eindeutige Klassifikation

! Turing-Vollständigkeit alleine ist kein Kriterium das DSLs von GPLs unterscheidet

2.2 Interne DSL

Interne DSLs sind *Untermengen* oder *Erweiterungen* einer GPL oder anderen DSL, die auf ihrer übergeordneten Sprache aufbauen, indem sie die schon existierende Sprachelemente, die Syntax, sowie Compiler wiederverwenden. Oftmals ist der Übergang zwischen interne DSLs und Frameworks fließend, da sich die Idee einer internen DSL auch direkt durch neue Datentypen und Funktionen innerhalb eines Frameworks umsetzen lässt.

Im folgenden werden kurz verschiedene interne technische DSLs vorgestellt:

2.2.1 javadoc

Die DSL javadoc dient der vollständig *rückwärtskompatiblen* annotation-basierten Dokumentation von Java-Klassen. Auf Basis dieser im Code befindlichen Kommentare (*code embedded*) kann z.B. eine HTML-basierte Dokumentation der API erzeugt werden.

```
/**
 * Get the absolut value of a number x.
 *
 * @param x
 * @return x or -x if x < 0
 */
int abs(int x)
```

Listing 1: javadoc Beispiel

2.2.2 XSD

XSD (XML Schema Definition) ist die auf XML basierende Sprache zur Beschreibung der Struktur und Regeln von XML-Daten. Mit XSD lassen sich z.B. explizit die in einem XML-Dokument erlaubten Datentypen (inklusive Invarianten) definieren.

```
<xs:simpleType name="monatInt">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="12"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="monate">
  <xs:list itemType="monatInt"/>
</xs:simpleType>

<monate>
  1 2 3 4 5 6 7 8 9 10 11 12
</monate>
```

Listing 2: XSD Beispiel für den Typ monatInt mit Daten

2.2.3 OpenMP

OpenMP (Open Multi-Processing) ist streng genommen eine API für die Shared-Memory-Programmierung auf Multiprozessor-Systemen und besteht aus speziellen Präprozessor-Direktiven (*code embeddede* DSL[2]) sowie einer kleinen Bibliothek besteht.
OpenMP ermöglicht die inkrementelle Parallelisierung von sequenziellem Programmcode, indem der Compiler datentechnisch voneinander unabhängige Codeabschnitte, die mit OpenMP-Direktiven makiert sind, durch die Nutzung von mehreren Threads parallelisiert. Da OpenMP auf Präprozessor-Direktiven basiert, ist es vollständig *rückwärtskompatibel*.

```
1  #include <omp.h>
2
3  // ...
4
5  #pragma omp parallel for
6  for (int i = 1; i <= 42; i++) {
7      // ...
8  }
```

Listing 3: OpenMP Beispiel

2.2.4 weitere Beispiele

- JUnit
- HMPPCG
- Reguläre Ausdrücke (Regex) in GPLs

[2]OpenMP ist eine interne DSL, da die Präprozessordirektiven indirekt ein Teil der GPL sind.

2.3 Externe DSL

Externe DSLs sind spezielle neue Sprachen, die im Idealfall perfekt an ihre Domäne und Anforderungen angepasst sind, dafür erfordert jedoch ihre Entwicklung zusätzlich zur Sprache selbst auch mindestens die Programmierung eines geeigneten Compilers. Externe DSLs sind sehr vielseitig und werden quasi überall verwendet, wo eine einfache und trotzdem gleichzeitig ausdrucksstarke Sprache nützlicher und angebrachter ist als eine vollständige GPL.

Im folgenden werden kurz verschiedene externe technische DSLs vorgestellt:

2.3.1 SQL

SQL ist eine Datenbanksprache zur Kommunikation mit einem Database-Management-System (DBMS) und ermöglicht so unter anderem den Zugriff auf die, in einer Datenbank enthaltenen, Daten. SQL-Anweisungen können sowohl *external* in Skripten, als auch *code embedded* in anderem DSL-/GPL-Code genutzt werden.

```
SELECT * FROM Station
    WHERE 50 < (SELECT AVG(Temp_C) FROM Stats)
    WHERE Station.ID = Stats.ID);
```

Listing 4: SQL Beispiel

2.3.2 HTML

Das im Internet allgegenwärtige HTML ist eine Auszeichnungssprache zur Beschreibung der Strukturierung und des Inhalts von Internetseiten. HTML steht normalerweise *external* in einer eigenen Datei, kann allerdings auch *code embedded* z.B. in E-Mails vorkommen.

```
<head>
<title>HTML Sample Page</title>
<head>
<body>
<a href="http://www.google.com/">Google</a>
</body>
```

Listing 5: HTML Beispiel

2.3.3 Bash

Bash ist strenggenommen die freie Unix-Shell des GNU-Projekts und damit die traditionelle Benutzerschnittstelle unter unixoiden Betriebssystemen. Bash dient jedoch auch der Verarbeitung von Skripten, die in der turing-vollständigen Bash-DSL geschrieben sind und der automatischen Steuerung von Programmaufrufen dienen. Der DSL-Code kann sowohl *external* als auch z.B. *code embedded* in Makefiles stehen.

```
#!/bin/sh
set -e -x
module load gcc/5.0
./configure --prefix=$HOME/libxc/2.1.2/
make -j 24
make install
```

Listing 6: Bash Beispiel

2.3.4 weitere Beispiele

- MATLAB / GNU Octave
- LaTeX/ Markdown / MediaWiki / reStructuredText
- Makefiles
- reguläre Ausdrücke (Regex)

3 Wissenschaftliche DSLs

Alle im folgenden vorgestellten Sprachen sind abstrahierende DSLs, die entworfen wurden um Wissenschaftlern aus verschiedenen Domänen die Entwicklung von optimierten Programmen zu vereinfachen und dienen der Arbeit mit Stencil-Codes.

Erklärung: Stencil-Codes

Als *Stencil-Codes* werden rechenintensive Iterations-basierte Algorithmen bezeichnet, die auf grid-basierten Daten rechnen. Ein grid (dt. Gitter) ist z.B. die Partitionierung (Diskretisierung) eines Raumes in eine Menge von Zellen/Datenpunkten mit fest definierter Nachbarschaft. So wird in einem Software-Windkanal ein Gitter um das zu untersuchende Objekt gelegt und in den Zellen bestimmte physikalische Eigenschaften des umliegenden Raums gespeichert.

Stencil-Codes haben die folgende typischen Merkmale:

- Der namensgebende *Stencil* (engl. Stempel), welcher das feste Zugriffsmuster auf die Daten der vorherigen Iteration beschreibt
- Die *Datenunabhängigkeit* innerhalb jeder Iteration, da sich die Daten jeder neuen Iteration komplett aus denen der Vorherigen berechnen und die Reihenfolge der Berechnungen dabei nicht relevant ist
- Die Kombination aus Datenunabhängigkeit und festem Zugriffsmuster ermöglicht eine extrem *effiziente parallele Implementierung*

Stencil-Codes eignen sich für viele verschiedene Einsatzbereiche, insbesondere finden sie Anwendung in numerischen Simulationen von physikalischen Vorgängen (wie Windkanälen), sowie in der Programmierung von PDE-Solvern.

3.1 ExaStencils

ExaStencils ist ein junges Projekt mehrerer deutschen Universitäten (Passau, Wuppertal und Erlangen-Nürnberg) mit dem Ziel eine DSL zur Implementation von Stencil-Codes für Exascale-Systeme zu entwickeln. ExaStencils soll mit (semi-)structured grids umgehen können und selbst eine automatische Auswahl geeigneter Lösungsverfahren für das spezifizierte Problem treffen.
Die DSL soll aus 4 Abstraktionsschichten bestehen und geeignete Sprachelemente auf jeder Schicht anbieten. Aus den Informationen im DSL-Code soll dann zusammen mit der zusätzlichen Einbindung von Wissen über die Zielhardware möglichst effizienter Code erzeugt werden.
Da die Entwicklung von Exastencils noch nicht abgeschlossen ist, ließen sich keine geeigneten Code-Beispiele finden.

3.2 Liszt

Die DSL Liszt ist ein Projekt der Universität Stanford für die Entwicklung von mesh[3]-basierten PDE[4]-Solvern, die beispielsweise die Ausbreitung von Wärme auf einer Herdplatte berechnen.
Liszt bietet die Möglichkeit aus einer Codebasis automatisch verschiedene Implementierungen zu generieren, die für verschiedene Plattformen (SMPs[5], Cluster und GPUs) optimiert sind. Der dafür nötige Liszt-Code ist dabei quasi sequentiell und plattformunabhängig und ermöglicht schnelles Prototyping selbst durch unerfahrenere Entwickler.

3.2.1 DSL

Die externe DSL Liszt ist als Spracherweiterung zu Scala implementiert, sodass sich die Liszt-Anweisungen *code embedded* im Scala-Code befinden. Aufgrund von neuen Sprachelementen in Liszt ist dieser Code jedoch *nicht abwärtskompatibel*.

Liszt erweitert Scala um folgende Sprachelemente:

- Eine *spezielle Deklaration der Datenfelder* der Gitterpunkte, die die Lesbarkeit fördert und die programmtechnische Handhabung der Daten vereinfacht

```
1 //Initialize data storage
2 val Position = FieldWithLabel[Vertex,Float3]("position")
3 val Temperature = FieldWithConst[Vertex,Float](0.f)
4 val Flux = FieldWithConst[Vertex,Float](0.f)
5 val JacobiStep = FieldWithConst[Vertex,Float](0.f)
```

- *Feste Methoden für den Zugriff auf die Daten*, die zur Compile-Zeit zur Berechnung der Stencilform verwendet werden

```
1 //read   edge e
2 val v1 = head(e)
3 val v2 = tail(e)
4 val dP = Position(v2) - Position(v1)
5
6 //write   vertex v
7 Temperature(v) = 1000.0f
```

- Und eine *eigene for-each Schleife*, die aufgrund der Datenunabhängigkeit zur automatischen Parallelisierung des Codes verwendet wird (vgl. OpenMP)

```
1 //for all vertices v in mesh
2 for (v <- vertices(mesh)) {
3     //...
4 }
```

[3] meshes sind unstrukturierte Gitter und damit in ihrer Anwendung sehr flexibel
[4] partial differential equation (dt. partielle Differentialgleichung)
[5] Symmetric MultiProcessing

3.2.2 Beispielprogramm

```
//Initialize data storage
val Position = FieldWithLabel[Vertex,Float3]("position")
val Temperature = FieldWithConst[Vertex,Float](0.f)
val Flux = FieldWithConst[Vertex,Float](0.f)
val JacobiStep = FieldWithConst[Vertex,Float](0.f)

//Set initial conditions
val Kq = 0.20f
for (v <- vertices(mesh)) {
    if (ID(v) == 1)
        Temperature(v) = 1000.0f
    else
        Temperature(v) = 0.0f
}

//Perform Jacobi iterative solve
var i = 0;
while (i < 1000) {
    for (e <- edges(mesh)) {
        val v1 = head(e)
        val v2 = tail(e)
        val dP = Position(v2) - Position(v1)
        val dT = Temperature(v2) - Temperature(v1)
        val step = 1.0f/(length(dP))
        Flux(v1) += dT*step
        Flux(v2) -= dT*step
        JacobiStep(v1) += step
        JacobiStep(v2) += step
    }

    for (p <- vertices(mesh)) {
        Temperature(p) += 0.01f*Flux(p)/JacobiStep(p)
    }

    for (p <- vertices(mesh)) {
        Flux(p) = 0.f; JacobiStep(p) = 0.f;
    }

    i += 1
}
```

Listing 7: Liszt Beispielprogramm

Dieses Beispielprogramm simuliert die Ausbreitung von Wärme in einem homogenen Raum, diskretisiert durch ein unstrukturiertes Gitter (mesh) und zeigt wie einfach ein parallelisierter PDE-Solver in Liszt geschrieben werden kann.

3.2.3 Kompilierung

Die Kompilierung von eines Liszt-Programms ist ein anschauliches Beispiel für die Komplexität der Generierung von optimiertem plattformspezifischem Code:

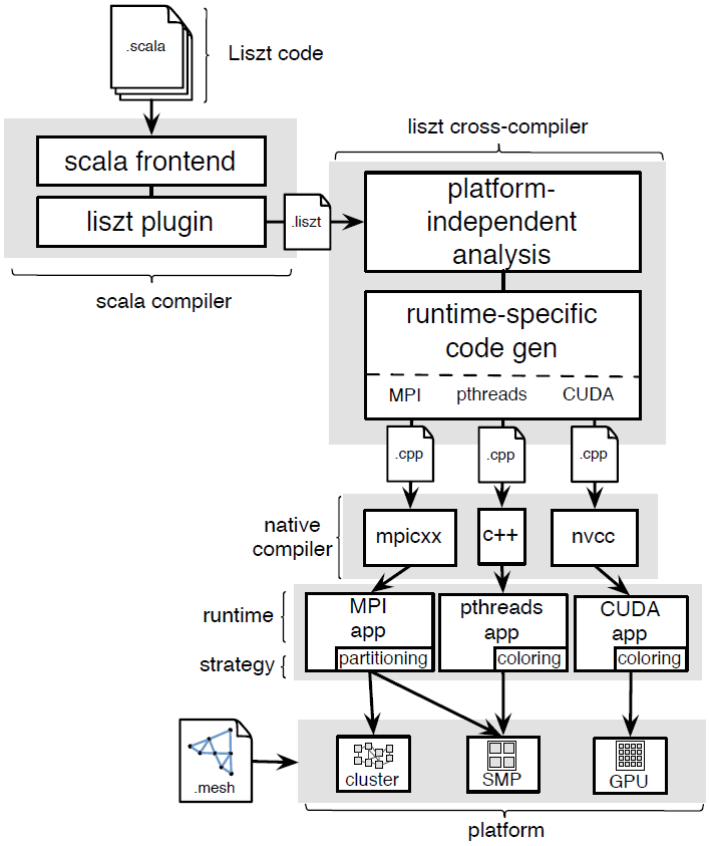

Abbildung 2: Kompilierung von Liszt-Code [L11]

Die Kompilierung des Liszt-Codes findet auf mehreren Stufen statt:

1. .scala → .liszt
   Der mit Liszt-Anweisungen erweiterte Scala-Code wird mit einem speziell für Liszt modifizierten Scala-Compiler zu purem Liszt-Code übersetzt.

2. .liszt → .cpp
   Dieser plattformunabhängige Liszt-Code wird mit Liszts eigenem Cross-Compiler analysiert und daraufhin in plattformspezifischen C++-Code umgewandelt.

3. .cpp → binary
   Der C++-Code wird mit dem entsprechenden nativen Compiler zu einer für die Ziel-Plattform optimierten ausführbaren Anwendung übersetzt.

Für jede der Plattformen wird die geeignetste Technologie zur Parallelisierung genutzt:

- Auf Cluster wird *MPI*,
- auf SMP-Rechner werden *pthreads*
- und für die Berechnung auf GPUs wird *CUDA* verwendet.

Das genutzte mesh wird von der Anwendung zur Laufzeit analysiert und nach einer geeigneten Strategie im Hinblick auf die Parallelisierung und Eigenschaften der Plattform aufgeteilt.

3.3 ICON DSL

ICON DSL ist ein Gemeinschaftsprojekt der Universität Hamburg, des Max-Planck-Instituts für Meteorologie und des Deutschen Klimarechenzentrums (DKRZ) zur Entwicklung einer externen DSL für das ICON-Klimamodell. Die Entwicklung mit ICON DSL findet *code embedded* in Fortran statt und nutzt zur Übersetzung einen Source-to-Source-Compiler, der nativen Fortran-Code erzeugt. Durch die mit ICON-DSL erreichte Hardwareunabhängigkeit soll die Softwareentwicklung durch Klimawissenschaftler produktiver machen.

4 Ausblick

4.1 Entwicklung

Die jüngere Geschichte zeigt deutlich, dass sich die Nutzung von geeigneten DSLs in jeden Teilbereich der Informatik ausgebreitet hat und inzwischen in eigenen Domäne selbst mit GPLs konkurriert: So wurde in der Vergangenheit die Produktivitätssteigerung in der Softwareentwicklung durch DSLs wie Regex und modernere Sprachen wie OpenMP und Liszt immer deutlicher.

In den nächsten Jahren wird die Grenze zwischen DSLs und GPLs vermutlich weiter verschwimmen, sodass sich die Softwareentwicklung noch mehr in der Verwendung von geeigneten Frameworks und DSLs wiederfinden wird. Diese mögliche Entwicklung führt schließlich zur Annahme, dass wir in Zukunft immer mächtigere DSLs verwenden werden und Programme in reinem Pseudocode schreiben können. Werden die Begriffe Zukunft und Domäne etwas gedehnt, dann könnten z.B. sogar (Klima-)Simulationen in Holodeck-ähnlichen Systemen durch die rein natursprachliche Beschreibung der genutzten Modelle geschehen.

4.2 Blick über den Tellerrand

Das Konzept von DSLs lässt sich nicht nur in der Softwareentwicklung, sondern in quasi jeglicher Anwendung von Sprachen finden:

- Chomsky-Hierarchie: regulär $\subset$ kontextfrei $\subset$ kontextsensitiv $\subset$ rekursiv aufzählbar
  Jede Sprach-Untermenge ist eine interne DSL der übergeordneten Sprache, die vollständig zur Beschreibung aller möglichen Sprachen ihrer Domäne ausreicht.

- Fachsprache: Juristendeutsch $\subset$ Deutsch
  Auf eine bestimmte Domäne zugeschnittene und optimierte Teilsprachen sind interne DSLs basierend auf natürlichen Sprachen wie z.B. dem Deutschen.

- Natürliche Sprachen: Deutsch $\subset$???, Englisch $\subset$???
  Alle natürlichen Sprachen sind externe DSLs mit der Domäne „Kommunikation zwischen Menschen“, was die Frage nach der entsprechenden übergeordneten GPL aufwirft ...

Literatur

ES14 C. Lengauer, S. Apel, M. Bolten et all: „ExaStencils: Advanced Stencil-Code Engineering. First Project Report“. Technical Report, MIP-1401, Universität Passau, Juni 2014

L11 Z. DeVito, N. Joubert, F. Palacios et all: „Liszt: A Domain Specific Language for Building Portable Mesh-based PDE solvers“ in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'11. New York, NY, USA: ACM, 2011

- R. Torres, L. Linardakis, J. Kunkel, T. Ludwig: „ICON DSL: A Domain-Specific Language for climate modeling“.