Extractor Description Language

Daniel Dominik Janke

Universität Koblenz-Landau
Universitätsstr. 1, 56070 Koblenz, Germany

danijank@uni-koblenz.de

Zusammenfassung

Extractor Description Language (EDL) vereinfacht die im Software-Reengineering immer wiederkehrende Überführung von Quellcode in abstrakte Syntaxgraphen (ASG). EDL verwendet einen GLR-Parser, der im Unterschied zu den üblichen LL/LR(k)-Algorithmen die Verwendung von beliebigen, auch mehrdeutigen, kontextfreien Grammatiken erlaubt. Darüber hinaus vereinfachen Symboltabellen-Stacks und deklarative Knotenerzeugung den Aufbau des ASGs. Erste Zeitmessungen deuten darauf hin, dass der Aufwand trotz GLR im Verhältnis zur Eingabelänge linear ist.

1 Einleitung

Als geeignete Repräsentationsform von Daten wie beispielsweise Quellcode haben sich TGraphen bewährt, da sowohl Knoten als auch gerichtete Kanten typisiert sind und über Attribute verfügen können. Darüber hinaus besteht eine globale Anordnung aller Elemente des Graphen sowie eine Reihenfolge der Inzidenzen jedes Knotens.

Um die Überführung in einen TGraphen zu vereinfachen, wurde in [4] die „Extractor Description Language“ (EDL) [1] entwickelt, die den „Syntax Definition Formalism“ (SDF) [3, 6] um semantische Aktionen erweitert. In diesem Paper soll anhand eines Auszugs einer Extraktoren-Beschreibung für Java verdeutlicht werden, wie die in EDL enthaltenen Symboltabellen deklarativ geschachtelte Gültigkeitsbereiche unterstützen.

2 Workflow

Um eine Eingabe in einen ASG überführen zu können, müssen zunächst eine in Module gegliederte Extraktoren-Beschreibung und ein Schema erstellt werden, das den Aufbau des gewünschten ASGs beschreibt. Wie in Abbildung 1 gezeigt, generiert der `EDLPreprocessor` aus diesen Daten eine Parse-Tabelle und einen `GraphBuilder`, der die semantischen Aktionen ausführt.

Die soeben generierten Artefakte werden vom `EDLParser` genutzt, um aus Eingabedateien den gewünschten ASG zu extrahieren. Hierzu nutzt `EDLParser` „Stratego/XT“ [2]. Dieses Programm implementiert einen GLR-Algorithmus, der auf dem in [5] vorgestellten Verfahren beruht.

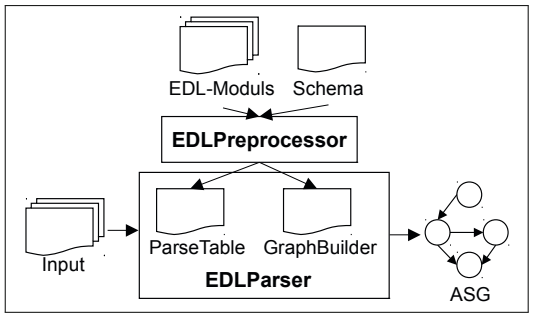

Abbildung 1: Der EDL-Workflow.

3 Anwendungsbeispiel

Das in Listing 1 gezeigte Modul einer Extraktoren-Beschreibung von Java soll die Verwendung von Symboltabellen veranschaulichen. Das zum Verständnis der semantischen Aktionen benötigte Schema des erzeugten TGraphen, ist in Abbildung 2 zu sehen. Die vom gezeigten `java/Example`-Modul importierten Module wurden aus Platzgründen durch ... ersetzt (Zeile 2). Die Deklaration der in diesem Modul verwendeten Symboltabelle `id2var`, die nur `Variable`-Knoten enthalten darf, geschieht in Zeile 4.

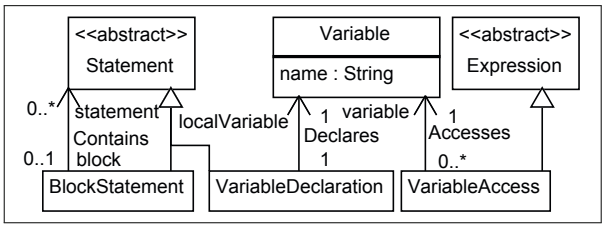

Abbildung 2: Das Java-Schema (Auszug).

Um die semantischen Aktionen der in den Zeilen 7 bis 22 definierten Regeln besser verstehen zu können, muss zunächst die Arbeitsweise von EDL erklärt werden: In einem ersten Schritt wird anhand der Regeln die Eingabe in einen Parse-Baum überführt. Danach wird er nach der „depth-first“-Methode traversiert. Dabei werden die semantischen Aktionen entsprechend der Position ausgeführt, an der sie in der jeweils angewendeten Regel stehen.

```
1  module java/Example
2  imports ...
3  symbol tables
4    id2var<Variable>
5  exports
6    context-free syntax
7      @Symboltable{id2var}
8      rule "{"
9          Statement #Contains($,$0);# *
10         "}" -> BlockStatement
11
12     rule Type Id ";" -> Statement
13       #$var = Variable();
14        $var.name = lexem($1);
15        id2var.declare($var.name, $var);
16        $ = VariableDeclaration();
17        Declares($, $var);#
18
19     rule Id -> Expression
20       #$var = id2var.use(lexem($0));
21        $ = VariableAccess();
22        Accesses($, $var);#
```

Listing 1: Ein Modul einer Java-Grammatik.

Die erste Regel in den Zeilen 8 bis 10 definiert ein `BlockStatement` als eine durch geschweifte Klammern umschlossene Folge von `Statements`. Anders als in EBNF werden Regeln in EDL wie bereits in SDF in der Form *Regelrumpf* `->` *Regelkopf* notiert. Der Tatsache, dass diese Blöcke Gültigkeitsbereiche von Variablen definieren, trägt die Annotation in Zeile 7 Rechnung. Sie stellt sicher, dass die innerhalb dieses Blocks in `id2var` eingefügten Variablen auch nur im durch die Anwendung dieser Regel aufgespannten Teil des Parse-Baums verfügbar sind.

Da es im Schema eine nicht-abstrakte Knotenklasse gibt, deren Name identisch mit dem Regelkopf `BlockStatement` ist, wird defaultmäßig ein Knoten dieses Typs erzeugt und als Ergebnis der Regelanwendung (synthetisiertes Attribut) deklariert. Durch die zwischen den `#` stehenden semantischen Aktionen wird dieser Knoten in Zeile 21 durch je eine `Contains`-Kante mit jedem durch die Anwendung einer `Statement`-Regel synthetisierten `Statement`-Knoten verbunden.

Die Regel in Zeile 12 beschreibt die gekürzte Syntax eine Variablendeklaration. Zunächst wird in Zeile 13 ein neuer `Variable`-Knoten erzeugt und der Variablen `$var` zugewiesen. In Zeile 14 wird das `name`-Attribut auf das Lexem gesetzt, das durch den Term mit der Position 1 im Regelrumpf (`Id`) erkannt wurde. Im Anschluss wird der soeben erzeugte `Variable`-Knoten in der Symboltabelle `id2var` unter seinem Namen registriert (Zeile 15). Sollte im aktuellen Gültigkeitsbereich bereits eine gleichnamige Variable existieren, wird eine Exception geworfen. Durch die Zuweisung an `$` wird der in Zeile 16 erzeugte `VariableDeclaration`-Knoten als synthetisiertes Attribut festgelegt. Schließlich wird er in Zeile 17 mit der `Variable`-Instanz per `Declares`-Kante verbunden.

Durch die Regel in Zeile 19 wird der Variablenzugriff in Java definiert. Bei jeder Anwendung dieser Regel wird in der Symboltabelle `id2var` nach dem `Variable`-Knoten gesucht, der unter dem durch den Term mit der Position 0 im Regelrumpf (`Id`) erkannten Lexem (dem Variablennamen) registriert ist (Zeile 20). Diese Suche beginnt beim innersten Gültigkeitsbereich und endet, sobald der erste Knoten dieses Namens gefunden wurde. Als Ergebnis dieser Regelanwendung wird ein `VariableAccess`-Knoten erzeugt (Zeile 21), der durch eine `Accesses`-Kante mit der zuvor identifizierten `Variable`-Instanz verbunden wird (Zeile 22).

4 Ergebnisse

Um den zeitlichen Aufwand abschätzen zu können, wurde eine vollständige Java-Grammatik erstellt, um aus real existierenden Projekten je einen ASG zu extrahieren und die dafür benötigte Zeit zu messen (siehe [4]). Die Granularität des ASG reicht bis zur Ebene der Methodensignaturen. Die Länge aller geparsten Dateien variiert je nach Projekt von 125.000 bis 14,5 Millionen Eingabezeichen. Der Extraktor benötigte 1,5 bis 216,6 Sekunden. Alle gemessenen Wert und die jeweils gemessene Zeit sind in Abbildung 3 in einem Diagramm mit logarithmischen skalierten Achsen dargestellt. Diese Darstellung legt einen linearen Aufwand nahe.

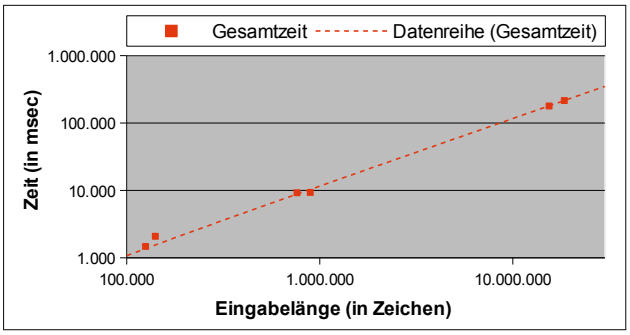

Abbildung 3: Gemessener Aufwand zum Parsen.

Darüber hinaus kann EDL ohne großen Aufwand so erweitert werden, dass neben TGraphen auch andere Repository-Technologien von den Extraktoren genutzt werden können.

Literatur

[1] `https://github.com/jgralab/edl`.

[2] `https://strategoxt.org/`.

[3] J. Heering and P. Klint. A syntax definition formalism. In *ESPRIT'86: Results and Achievements*, pages 619–630. North-Holland, 1986.

[4] D. D. Janke. Extractor description language. Master's thesis, Universität Koblenz-Landau, Campus Koblenz, 2012.

[5] M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1986.

[6] E. Visser. *Syntax definition for language prototyping*. Ponsen & Looijen, 1997.