

PQ4R Exzerpte

Marco Maniscalco (maniscalco.twoday.net)

①

Andrew W. Appel, Jens Palsberg. Modern Compiler Implementation in Java, Second Edition.
Cambridge: Cambridge University Press 2002

Preview

- JavaCC Parser Generator
- Predictive Parsing
- Using Parser Generators
- Semantic Actions
- Visitors
- Intermediate Representation Trees
- Translation into Trees

Question

1. Wie kann Code ausgeführt werden?
2. Wie verbindet man semantische Aktionen mit Grammatiken?
3. Wie repräsentiert man einen abstrakten Syntaxbaum in Zwischencode?

Read

1. Code kann in einem *one-pass* Compiler ausgeführt werden. Hier wird lexikalische Analyse, Parsen, semantische Analyse und Ausführung in einem Schritt parallel erledigt. Das bringt mit sich, dass semantische Aktionen in der *EBNF* mit ‚angeheftet‘ werden müssen. Das unterstützen nicht alle Generatoren. Komplexe Grammatiken werden so sehr schwer lesbar und sind schwer zu debuggen. Die Alternative hierzu ist ein generierter Parse-Tree. Der Vorteil hier ist dann, dass man nicht alle Schritte auf einmal machen muss und die verschiedenen Aspekte separieren kann. Nachdem die Lexikalische Analyse ausgeführt wurde, kann z.B. ein Optimierungsschritt, Zwischencode-Erzeugung oder die eigentliche Traversierung (Ausführung) durchgeführt werden.
2. JavaCC bietet die Möglichkeit semantische Aktionen an Regeln zu binden. Diese werden dann im generierten Parsen an die jeweiligen Stellen geschrieben.
3. Man kann einen abstrakten Syntaxbaum als Kette von Operatoren darstellen, die jeweils eine Regel im Recursive Descent Parser repräsentieren. Die Operanden werden jeweils hinter einen Operator geschrieben. Dies stellt ein Lösungs-Schema für Blöcke dar.

Reflect

Kapitel 3,4 und 7 liefern gute Ideen zum Thema Parse-Trees, und 2 wichtige Paradigmen zur Ausführung von Code in Form von abstrakten Syntax Bäumen. Speziell wird hier JavaCC abgehandelt und aufgezeigt, wie semantische Aktionen in den generierten Parser

gelangen können. In Kapitel 7 wird konkret auf die Darstellung von Bäumen in Zwischencode eingegangen. Mit Vorkenntnissen in diesem Thema sind alle Aussagen gut nachvollziehbar, jedoch muss teilweise der Praxisbezug beim Leser vorhanden sein, um die eher einfach gehaltenen Beispiele transportieren zu können.

Recite

Man kann bei JavaCC semantische Aktionen direkt in *EBNF* einbinden. Das bedeutet, in der Grammatik werden an den Stellen Aktionen hinterlegt, die sich auch im generierten Parser an der jeweiligen Stelle befinden. Dieses Paradigma erscheint effizient, da im Vergleich zu generierten Parse-Trees im Idealfall nicht so viele Klassen wie Nicht-Terminale generiert werden müssen, die unter Umständen lediglich leere Funktionsrümpfe besitzen. Komplexe Grammatiken werden somit jedoch schwer zu überblicken und schwer zu debuggen, da man den semantischen Code im generierten Parser wieder findet und generierter Code ist für Menschen überwiegend schwer zu lesen. Bei Parse-Trees hingegen hinterlegt man den Semantischen Code in den erzeugten Semantik-Klassen, die im Parser beim Parsen instantiiert werden. Hingegen ein *one-pass* Compiler führt alle semantische Aktionen beim Parsen parallel durch. Mit Hilfe von Visitors können semantische Aktionen sauber getrennt werden und nacheinander, auch selektiv, mehrfach ausgeführt werden. Lässt man einen Parse-Tree generieren, der Traversierungs Funktionen bereitstellt, kann hier eine semantische Aktion hinterlegt werden, die den Baum verlustfrei in eine andere Datenstruktur überführt, so genannter Zwischencode. Zwischencode besitzt ein wesentliches charakteristisches Merkmal, ein Datenstrom schneller interpretiert werden als es beim eigentlichen Parsing-Vorgang der Fall ist.

Review

Die Autoren Appel und Palsberg benennen praxistaugliche Ideen zum Thema „Ausführung von Code“ mittels generierten Parse-Trees und dem Visitor Pattern, sowie die Vor- und Nachteile von *one-pass* Compilern. Unabhängig hiervon wird verdeutlicht, wie ein Syntaxbaum in eine andere Datenstruktur transformiert werden kann, die charakteristische Vorteile mit sich bringt.

②

T. J. PARR, R. W. QUONG. ANTLR: A Predicated $LL(k)$ Parser Generator. 1995

Preview

- Parsing
- Compiler
- Parser generator
- Predicates
- $LL(k)$ parser

Question

1. Wie ist die grundlegende Funktionsweise von ANTLR?
2. Wie kann Code ausgeführt werden?
3. Wie verbindet man semantische Aktionen mit der Grammatik?

Read

1. ANTLR ist ein $LL(k)$ Parser Generator, der wie JavaCC Predicates unterstützt, das heißt semantische Aktionen können an beliebigen Stellen eingefügt werden. Es können abstrakte Syntaxbäume automatisch generiert werden. Um die Lookahead Problematik zu lösen, wird eine ‚lineare Approximation‘ der Lookahead Tabelle berechnet. Würde man eine Tabelle berechnen, wäre die Anzahl der Tokens k polynomiell in die Größe der Tabelle enthalten. Durch den schwächeren Algorithmus der die lineare Approximation berechnet, können nicht alle $LL(k)$ Probleme gelöst werden. Zusätzlich unterstützt der Generator für genau diese Fälle das Erzeugen solcher exakten Tabellen, damit wieder die gesamte Menge der $LL(k)$ Probleme geparkt werden können. Die Grammatiken werden in *EBNF* erstellt.
2. Code kann während des Parsing Vorgangs direkt ausgeführt werden, indem man Semantic Predicates einfügt. Des Weiteren kann anhand einer Tree-Parser Grammatik, ein Baum erzeugt werden, der im generierten Parser verwendet wird.
3. Semantische Aktionen können direkt als eingebetteter Java Code in den *EBNF* Grammatiken eingebunden werden. Im erzeugten Recursive Descent Parser, wird für jedes Nicht-terminal eine Funktion erzeugt, die zum einen den Parser Code enthält, als auch den semantischen Code.

Reflect

Der Artikel bietet eine komplette Übersicht über die Grundlegenden Funktionen von ANTLR. Es wird ersichtlich, wie die Grammatiken mit dem generierten Parser zusammenhängen, und wie semantische Aktionen einfließen. Man bekommt des Weiteren einen Einblick in die interne Funktionsweise und die theoretischen Grenzen des Generators.

Recite

ANTLR ist ein $LL(k)$ Parser Generator, der keine statischen Lookahead Tabellen erzeugt, und stattdessen ein dynamisches Lookahead verwendet. Das dynamische Lookahead wird mit einer linearen Approximation erreicht, wodurch auch die Mächtigkeit von $LL(k)$ mo-

derat eingeschränkt wird. In vielen natürlichen Grammatiken ist diese Einschränkung nicht von Bedeutung. Natürliche Grammatik heißt in diesem Kontext, dass eine Grammatik überwiegend $LL(1)$ ist, und an Knackpunkten ein $k > 1$ verwendet.

Anhand einer speziellen Tree-Walker Grammatik kann ein AST automatisch erzeugt werden, der direkt vom Parser verwendet werden kann. Semantische Aktionen können ähnlich wie bei JavaCC direkt in der *EBNF* Grammatik eingebunden werden. Dadurch muss der generierte Parser nicht editiert werden.

Review

Parr und Quong bringen die grundlegenden Techniken von ANTLR anhand von konkreten Beispielen nahe. Hier wird speziell auf die verwendete Parser Technik und deren Grenzen eingegangen. Die Verwendung von semantischem Code in one-pass Compilern sowie das Generieren von Parse-Trees wird erläutert.

③

Terence Parr, Russell Quong. LL and LR Translator Need k . *SIGPLAN Notices* Volume 31 #2 February 1996.

Preview

- $LL(k)$ Translators
- $LR(k)$ Translators
- Example Grammars needing $k > 1$
- Implementing Issues

Question

1. Wie unterscheiden sich LL und LR Parser?
2. Warum ist $k > 1$ überhaupt interessant?
3. Wie kann $LL(k > 1)$ realisiert werden?
4. Wie wird $LL(k)$ in ANTLR realisiert?

Read

1. Beide Parser, LL und LR benutzen Endliche Automaten und die Stärke der Übersetzungsfunktion hängt in beiden Fällen maßgeblich von der Größe des verwendeten Kontextes und der Größe des Lookaheads k ab. LL Parser arbeiten *top-down* wohingegen LR Parser *bottom-up* arbeiten. Die *bottom-up* Methode ist stärker, da bei dieser Methode während des Parsens schon mehr Kontext zur Verfügung steht. Hier können dann mehrere Ableitungsregeln simultan geprüft werden. Die *top-down* Variante muss mit einem Lookahead explizit hinter die bisher gelesenen Tokens sehen können. Man kann bei $LL(k)$ Grammatiken oft auf $LL(k-1)$ Grammatiken durch Linksfaktorisierung reduzieren, wobei es auch Grammatiken gibt, für die keine Reduktion existiert.
2. Es liegt nahe, Grammatiken so zu vereinfachen, dass sie mit $LL(1)$ oder $LR(1)$ erkannt werden können, denn daraus resultieren einfachere Parser. In realen Problemen kommt man jedoch durch diverse Probleme nicht um $k > 1$ herum. Ein Punkt ist die schlechte Lesbarkeit einer $k=1$ Grammatik, ein anderer sind diejenigen Konstrukte, die mit $k=1$ schlicht nicht trivial lösbar sind. Ein berühmtes Problem ist das „*Dangling-Else-Problem*“.
3. Lookahead Information für $k > 1$ benötigt $O(|T|^k)$ Speicherplatz. $|T|$ ist die Anzahl der Token Typen und k die Größe des Lookahead. $LL(k > 1)$ kann einerseits durch Tabellen realisiert werden, die jeweils die Token Tupel beinhaltet. Das ist für $k \gg 1$ wegen dem Platzverbrauch nicht machbar. Ein anderer Weg ist eine Heuristik die ‚lineare Approximation‘ genannt wird und den Aufwand auf $O(|T| * k)$ reduziert.
4. $LL(k)$ ist in ANTLR mit einer linearen Approximation realisiert.

Reflect

Der Artikel gibt Aufschluss über den Unterschied von LL und LR Parsern und den Problemen eines Lookaheads mit $k > 1$. Es wird eine Lösung vorgestellt, lineare Approximation, die Aussagen über den Speicher-/Platzverbrauch von Parsern und die Mächtigkeit trifft. Der Sinn und Zweck von $k > 1$ sind gewollte Sprachelemente, die in manchen Fällen nicht in $k=1$ darstellbar sind. Gleichzeitig wird ausgesagt, dass es heute (damals 1996) schon praktische Einsatzgebiete für große Lookaheads gibt und die Rechner dies auch bewältigen könnten.

Recite

LL und *LR* Parser werden mit Automaten realisiert. Konkret wird hier von „*Recursive Descent Parsen*“ gesprochen, für die jeweils pro Nichtterminal eine Funktion existiert, die im inneren ein Switch/Case mit den Lookahead Information bereithält. Der *LR* Ansatz ist *bottom-up*, hier wird ein „*Push Down Automat*“ genutzt. *LR* Parser können sehr viel schwerer zu debuggen sein. Ein *LL* Parser entspricht eher dem, was man auch von ‚Hand‘ implementieren würde und einfacher zu debuggen.

Eine Lookahead Größe $k > 1$ ist interessant, da es moderne Sprach-Features gibt, die schlicht nicht auf $k = 1$ reduzierbar sind und eine Grammatik mit $k > 1$ für den Menschen einfacher lesbar wird.

Wenn man die Geschichte der Compiler betrachtet, war in den Anfangsjahren $k > 1$ technisch nicht machbar. Zur heutigen Zeit jedoch gibt es Heuristiken, die verhindern, dass komplette Lookahead Tabellen erzeugt werden müssen. Man kann, wenn die Komplexität und Platzverbrauch keine Rolle spielt natürlich solche Tabellen generieren.

In ANTLR werden dynamische Lookahead Informationen erzeugt. Der Generator erkennt die Stellen, an denen ein Lookahead nötig ist, und berechnet die Approximation Tabelle.

Review

Parr und Quong schildern den praktischen Nutzen von *LL*($k > 1$) Parsern und zeigen Möglichkeiten für die Implementierung auf. Die Aussagen basieren maßgeblich auf der Phd. Thesis von Terence Parr, die auch als Meilenstein in der modernen Spracherkennung gilt.