Abstrakte Syntaxbäume

Steffen Pingel
Universität Stuttgart
Institut für Softwaretechnologie
Breitwiesenstraße 20-22
D-70565 Stuttgart
steffenp@gmx.de

Studienprojekt A: IML Browser
Licensed under GNU Free Document License[2]

Überblick

Abstrakte Syntaxbäume haben vielfältige Einsatzmöglichkeiten im Compilerbau und Reverse Engineering. Im Folgenden wird sowohl die Erstellung von Syntaxbäumen betrachtet als auch die enthaltenen Informationen und deren Anwendungsmöglichkeiten.

Schlüsselwörter Abstrakter Syntaxbaum, Analyse, AST, Zwischendarstellung

1 Einleitung

Allgemein betrachtet sind Abstrakte Syntaxbäume (Englisch: Abstrakt Syntax Tree, AST) eine Zwischendarstellung (Englisch: Intermediate Representation, IR) von Quellcode.

Quellcode wird in der Regel zeilenweise als einfacher Text dargestellt. Durch Einrücken kann ein Programmierer eine bestimmte Struktur repräsentieren. Syntaxbäume gehen noch einen Schritt weiter und stellen den Code in seiner exakten syntaktischen Struktur in einem Baum dar.

Anwendungen finden sich hauptsächlich im Compilerbau, aber auch im Reengineering werden Syntaxbäume für Analysen eingesetzt (z.B. zur Klonerkennung).

2 Analyse

In den nächsten drei Abschnitten erfolgt ein Exkurs in die Grundlagen der Übersetzer. Es wird beispielhaft anhand eines kurzen Codestückes dargestellt wie ein Compiler bei der Übersetzung vorgeht.

Ein Compiler besteht in der Regel aus mehreren Teilen[1], die jeweils eine speziellen Aufgabe wahrnehmen. Jeder Teil erhält seine Eingabe von dem vorherigen Teil, transformiert diese und gibt sie an den nächsten Teil weiter.

Grob werden diese Teilaufgaben in zwei Phasen eingeteilt. Die Analyse erhält als Eingabe das Quellprogramm, erzeugt daraus eine Zwischendarstellung, die anschliessend von der Synthese in das Zielprogramm überführt wird.

Dieser Aufsatz beschränkt sich auf die Analyse und erklärt welche Transformationen hier durchgeführt werden. Die Analyse ist in 3 Module unterteilt: In die lexikalische, syntaktische und semantische Analyse.

Lexikalische Analyse

Die lexikalische Analyse erhält als Eingabe den Quelltext des zu übersetzenden Programms. Die erste Aufgabe ist es diesen in eine Folge von lexikalischen Einheiten, Symbole genannt, aufzuteilen. Die Symbole sind in üblichen Programmiersprachen Bezeichner, Leerzeichen, Umbrüche und andere Sonderzeichen wie kleiner und grösser Zeichen, Klammern, usw. Die Erkennung kann von deterministischen Automaten, die aus regulären Ausdrückern erzeugt werden, durchgeführt werden. Das Modul, das diese Aufgabe erfüllt wird als Scanner bezeichnet.

Der Scanner gibt die Symbole an den Sieber weiter (Englisch: Sifter). Der Sieber klassifiziert die Symbole. Er kann Operatoren erkennen und Schlüsselwörter von Bezeichnern unterscheiden in dem er z.B. in einer Symboltabelle nachschaut in der sämtliche Schlüsselwörter hinterlegt sind. Er kann auserdem feststellen welche Symbole ignoriert werden können wie Leerzeichen, die lediglich zur Trennung dienen, oder Kommentare, die keine für die Übersetzung not-

wendigen Informationen enthalten. Die Ausgabe des Siebers wird als Lexem oder Token bezeichnet.

Ein Beispiel für reguläre Ausdrücke ist hier angegeben (Terminale sind von Anführungszeichen umschlossen, | bedeutet alternativ):

id → 'a' | ... | 'z'
num → '1' | ... | '9'
int → **num num***
delim → ' '
assop → '='
multop → '*' | '/'
addop → '+' | '-'
sem → ';'

Als Eingabe dient die folgende Zuweisung:

a = 2+4*10;

Der Scanner erkennt folgende Symbole (fettgedruckt) mit den jeweiligen Ableitungen (in Klammern angegeben):

id(a) **delim**( ) **assop**(=) **delim**( ) **int**(2) **addop**(+) **int**(4) **multop**(*) **int**(10) **sem**(;)

Die Ausgabe des Siebers nach dem Entfernen der entsprechenden Symbole ist:

id(a) **assop**(=) **int**(2) **addop**(+) **int**(4) **multop**(*) **int**(10) **sem**(;)

Syntaktische Analyse

Der Parser führt die syntaktische Analyse durch. Das Ziel ist es die Struktur des Programms herauszufinden. Der Parser kann den Unterschied zwischen Ausdrücken, Anweisungen, Deklarationen usw. erkennen.

Ausserdem wird geprüft, ob das Programm syntaktisch korrekt ist. Das ist oft nicht der Fall. Die meisten Programme werden mehrfach kompiliert und korrigiert bis die Syntax richtig ist.

Die Ausgabe des Parsers ist eine Zwischendarstellung des Programms, die strukturelle Informationen enthält. Es gibt mehrere Möglichkeiten die Struktur eines Programms darzustellen. Dieser Aufsatz beschränkt sich auf die Darstellung als Syntaxbaum.

Der Baum wird mit Hilfe einer kontext-freien Grammatik, die eine eindeutige Erkennung der Syntax ermöglicht, aus dem Tokenstrom vom Sieber aufgebaut und als Parsebaum oder konkreter Syntaxbaum bezeichnet. Dieser Baum enthält auf Grund der Ableitungsregeln der Grammatik viele Informationen, die für die weitere Verarbeitung des Programms irrelevant sind. Deshalb wird der Parsebaum abstrahiert.

Dabei werden sämtliche Nichtterminale und Schlüsselwörter entfernt, da diese keine semantische Bedeutung tragen. Auch Terminale, die die Präzendenz von Operatoren ausdrücken, werden entfernt, da die Präzendenz bereits durch die Schachtelung des Baumes ausgedrückt wird. Dieser komprimierte Baum wird als Abstrakter Syntaxbaum bezeichnet.

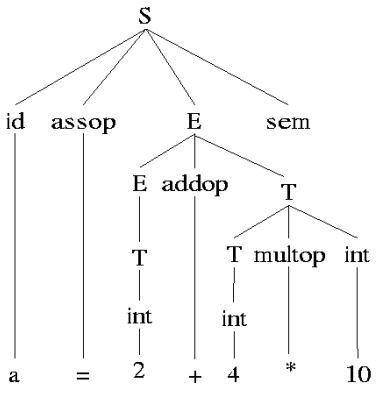
Hier ist eine kurze Beispiel Grammatik zur Erkennung von einfachen Ausdrücken angegeben, die allerdings Punkt- vor Strichrechnung berücksichtigt:

S → **id assop** E **sem**
E → E **addop** T | T
T → T **multop int** | **int**

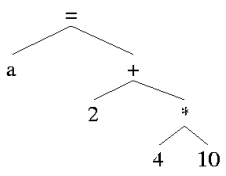
Im folgenden wird das Beispiel aus 2 exemplarisch Top-Down, also ausgehend von dem Startsymbol der Grammtik, in diesem Fall S, geparsed. E steht in der Grammatik für Expression und T für Term. Die fettgedruckten Symbole entsprechen den Regulären Ausdrücken aus 2.

Für S gibt es nur eine Produktion, also wird S zu id, asssignment operator, expression und semicolon abegeleitet. id wird zu a abgeleitet, hier durch eine Kante im Baum dargestellt. Das ergibt sich aus den Regulären Ausdrücken, die für die lexikalische Analyse verwendet wurden.

Das E verbleibt als Nicht-Terminal und und wird zu E, add operator, T abgeleitet usw. Das Verfahren endet, wenn alle Nicht-Terminale abgeleitet sind. Das Ergebniss ist hier abgebildete Parsebaum.



Wie schon erwähnt wird der Parsebaum anschliessend abstrahiert und die folgende Darstellung transformiert. Es ist gut sichtbar wie die Operatoren hoch gezogen wurden und die Kettenregeln entfernt wurden.



Semantische Analyse

In der semantischen Analyse wird nun der Abstrakte Syntaxbaum mit statischen semantischen Informationen angereichert. Im Gegensatz zu dynamischen Eigenschaften, die erst zur Laufzeit von Programmen ermittelt werden können. Die statischen Attribute sind in der Regel kontext abhängig.

Um die Attribute zu ermitteln gibt es verschiedene Strategien. Man kann Übesetzungschemata angeben, die das Quellprogramm bereits während der Syntaktischen Analyse in Zielcode übersetzen. Das funktioniert allerdings nur für einfache Sprachen. Für kompliziertere Sprachen kann man die Übersetzungregeln explizit Codieren oder Attributgrammatiken verwenden. Attributgrammatiken werden im nächsten Abschnitt näher erläutert.

Zu den statischen semantischen Informationen gehört die Namensbindung. Zu jedem Bezeichner wird festgestellt an welcher Stelle dieser deklarariert wurde. Da ca. ein Drittel der Lexeme Bezeichner sind, gehört dieses zu den Hauptaufgaben während der semantischen Analyse. Um den sogenannten Lookup möglichst effizient zu gestalten können die Bezeichner in einer Hashtabelle, genannt Symboltabelle, gehalten werden.

Ausserdem werden zu jeder Operation soweit das möglich ist die Typen der Operatoren bestimmt. Die Typbindung kann im trivialen Falles einfach anhand des Operators bestimmt werden, aber in der Regel ist es erforderlich den Kontext also die Operanden mit einzubeziehen. Bei Bezeichnern ist es einfach, da man anhand der Namensbindung ohne weiteres den Typ ermitteln kann. Wobei zu beachten ist, dass viele Programmiersprachen auch eine implizite Konvertierungen, z.B. von int nach float erlauben. Es ist also nicht immer ganz einfach zu entscheiden welche Typen eine Operation hat. Bei Überladung von Operatoren kann sogar die Operation erst nach der Bestimmung der Operanden-Typen ermittelt werden.

Ausserdem wird die Sichtbarkeit und damit auch die Überdeckung bestimmt. Falls eine lokale Variable den selben Bezeichner wie eine globale Variable hat überdeckt die lokale die globale Variable. Um die Sichtbarkeit zu bestimmen, kann man die Bezeichner auf einen Stack legen, wenn der Deklarationsknoten traversiert wird, und wieder herunter nehmen, wenn der Block in dem sie deklariert wurden verlassen wird. Damit sind nur jeweils die auf dem Stack liegenenden Bezeichner sichtbar.

Die gefunden Attribute werden im Syntaxbaum bei den entsprechenden Knoten abgelegt bzw. als semantische Kanten zum Baum hinzugefügt.

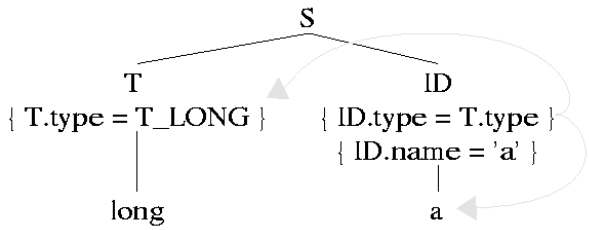
3 Attributgrammatiken

In den sogenannten Attributgrammatiken werden in die Grammatik Berechnungsvorschriften eingefügt. Bei einer Traversierung oder bereits bei der Erstellung des Syntaxbaumes werden diese Berechnungsvorschriften ausgeführt und die Knoten mit entsprechenden Attributen versehen. Die Lexeme sind im folgenden Beispiel fettgedruckt und die Berechnungsvorschriften in eckigen Klammern angegeben.

S → T ID [ID.type = T.type]
T → **long** [T.type = T_LONG]
T → **float** [T.type = T_FLOAT]
ID → **a** [ID.name = 'a']

Im Compilerbau bezeichnet man das als syntaxgesteuerte Übersetzung. Die Idee ist, die Codeerzeugung und Typüberprüfung bereits während der Syntaxanalyse durchzuführen was allerdings wie gesagt nur für einfache Sprachen möglich ist.

Man unterscheidet zwischen zwei verschiedene Arten von Attributen. Die zusammengesetzten, synthesized, Attribute setzen sich nur aus eigenen Attributen und den Attributen der Söhne zusammen. Wohingegen ererbte Attribute von Attributen des Vaterknotens und der Geschwisterknoten abhängen können.



Hier angegeben ist der Parsebaum zur der Deklaration "**long a**". Der Name von dem ID Knoten ist nur von seinem Kindknoten abhängig, also ist das ein zusammengesetztes Attribut. Wohingegen der Typ von ID vov T, also einem Geschwisterknoten, abhängig ist. Damit ist das ein ererbtes Attribut.

Den enstandenen Baum bezeichnet man als Attibutierten Syntaxbaum.

Man sieht hier bereits, das die beschriebenen Phasen in einem Compiler nicht strikt getrennt sind. Hier wird während der Syntaxanalyse bereits ein Teil der semantischen Analyse durchgeführt.

4 Compiler Struktur

Die Analyse lässt sich zu den folgenden Schritten zusammenfassen: Das Quellprogramm wird vom Lexer in einen Tokenstrom transformiert. Anschliessend vom Parser in einen Parsebaum transformiert,

der zu einem Abstrakten Syntaxbaum zusammengefasst wird und schliesslich während der semantische Analyse in einen, mit semantischen Informationen, den Attributen, angereicherten, annotierten abstrakten Syntaxbaum umgewandelt. In Praxis werden die beschriebenen Phasen in einem Compiler allerdings nicht sequentiell, sondern parallel ausgeführt.

Die Analyse, also die Transformation des Quellcodes in eine eine generische Zwischendarstellung, typischerweise in eine Zwischensprache, wird von dem Programmiersprache nabhängigen Front-End durchgeführt. Nach der Analyse erfolgt die Synthese. Im Middle-End wird die Zwischendarstellung optimiert und anschliessend im Back-End in eine plattformabhängige Zielsprache übersetzt.

Um ein Front-End zu erstellen können Generatoren verwendet werden. Diese können aus Regulären Ausdrücken und Grammatiken Scanner und Parser erzeugen. Um einen Scanner zu erzeugen kann beispielsweise flex und um einen Parser zu erzeugen, bison eingesetzt werden.

Ein Compiler erzeugt, wie bereits erwähnt, nicht unbedingt nur einen Abstrakten Syntaxbaum als Zwischendarstellung, sondern es sind verschiedene Darstellungen mit unterschiedlichen Abstraktionsebenen möglich. GCC generiert aus dem Abstrakten Syntaxbaum beispielsweise eine generische Register Transfer Language, die anschliessend in die Zielsprache transformiert wird.

Im folgenden wird auf die Darstellung von Abstrakten Syntaxbaum im Rechner eingegangen und eine Einblick in die Anwendungsmöglichkeiten von Abstrakten Syntaxbäumen gegeben.

5 Interne Darstellung

Die Knoten eines Abstrakten Syntaxbaumes lassen sich z.B. objekt-orientiert modellieren. Anstatt Klassen zu verwenden eignen sich natürlich auch Varianten Rekords wie Unions in C.

Die Attribute lassen sich als Membervariablen speichern. Zur Optimierung werden diese, da sie sich oft wiederholen können, aber in der Regel in Symboltabellen gehalten und über Zeiger referenziert.

Die Namensbindung, also den Verweis an welcher Stelle ein Bezeichner Deklariert wurde, lässt sich geschickterweise auch als Kante auf den Deklarationsknoten im Syntaxbaum darstellen.

Zur persistenten Speicherung von annotierten Syntaxbäumen wurde die Interface Definition Language (IDL) definiert. Die IDL sieht ähnlich aus wie eine Bakkus-Naur-Form und enthält sowohl die syntaktischen als auch die semantischen Attribute. Die IDL eignet sich auch zum Austausch von Syntaxbäumen zwischen Werkzeugen.

6 Eigenschaften von ASTs

Syntaxbäume sind eine exakte Darstellung des Programms. Allerdings mit der Ausnahme, dass Kommentare und Trennzeichen wie Tabulatoren und Leerzeichen nicht berüksichtigt werden. Bei Abstrakten Syntaxbäumen können weitere Details wie Klammerung fehlen. Das verhindert im Zweifelsfall die exakte Rücktransformierung des Syntaxbaumes in den ursprünglichen Quellcode. Die Semantik bleibt allerdings auch bei einer Rücktransformierung genau die gleiche. Es gibt auch Ansätze den kompletten Quellcode im Syntaxbaum zu hinterlegen, um diesen wieder exakt rücktransformieren zu können.

Ein Abstrakter Syntaxbaum kann auf Grund seiner Baumstruktur effizient traversiert werden. Die Grösse ist in der Praxis ungefähr linear zu der Grösse des Quellprogramms. Auch die Erzeugung kann mit Hilfe einer Grammatik in linearer Zeit durchgeführt werden.

Durch Hinzufügen von semantischen Kanten können allerdings Zykel entstehen und der Baum sehr gross werden. Dann kann es schwierig werden den Baum schnell zu traversieren.

Für bestimmte Analysen kann es auch erforderlich sein den Baum rückwärts zu traversieren. Für diesen Zweck kann der Baum auch mit Rückwärtskanten ausgestattet werden.

Die Attributierbarkeit ist besonders im Reverse Engineering wichtig, da zu den einzelnen Kanten und Knoten zusätzliche Informationen gespeichert werden können. Das ermöglicht das Programmwissen, das während der Analyse stetig zunimmt, mit menschlicher Intuition zu verbessern.

7 Analysen auf Syntaxbäumen

Auf Abstrakten Syntaxbaum lassen sich verschiedenste Analysen durchführen. Allgemein lässt sich sagen, dass die Analysen weitere Informationen zum Syntaxbaum hinzufügen z.B. bei einer Datenabhängigkeitsanalyse in Form von zusätzlichen Set-Use, Use-Set und Set-Set Beziehungen. Teilweise wird der Syntaxbaum auch weiter abstrahiert, wenn beispielsweise bei der Kontrollflussanalyse nur die Beziehungen zwischen Basisblöcken betrachtet werden.

Für Analysen, die eine grob-granulare Repräsentation des Programms benötigen, sind Abstrakte Syntaxbäume nicht geeignet, da sie eine exakte Darstellung des Programms sind. Für solche Analysen sind Entity-Relationship Graphen wie z.B. ein Resource Flow Graph besser geeignet. Eine Beispiel für eine solche Analyse wäre eine Cliche Recognition, um z.B. einen Stack zu erkennen.

8 Varianten

Um die Erstellung von Compilern zu vereinfachen und die Interoperatibilität von Analyse Werkzeugen zu ermöglichen gibt es Bestrebungen die ASTs innerhalb von Programmiersprachen zu *vereinheitlichen*.

Für Ada gibt es beispielsweise ASIS. Den Nachfolger des erfolglosen DIANA. Dieses Schema wird auch tatsächlich in der Ada Community eingesetzt. Es wäre unrentabel für jeden Compiler oder jedes Analysewerkzeug eine neues Frontend zu schreiben, das die komplette Ada Spezifikation implementiert.

Für C gibt es beispielsweise das Datrix Schema und Columbus Schema[5]. Beide haben ihre Vor- und Nachteile, ähneln sich aber stark. In der Praxis hat konnte sich allerdings bisher keines durchsetzen.

Ein anderer Ansatz ist es die Abstrakten Syntaxbäume Programmiersprachen *unabhängig* zu machen. IML[3] hat dies erfolgreich für C und ein Subset von Ada geschafft.

Dabei werden Konstrukte wie z.B. for Schleifen, die sich in Ada und C stark unterscheiden, durch eine generische Implementation mit einer If-Bedingung und Gotos repräsentiert. Dieser Syntaxbaum wird entsprechend als Generalized Abstrakt Syntax Tree bezeichnet.

Aus der IML lassen sich Ansichten verschiedener granularitäts Stufen erzeugen. Das Function Level, also die grob-granulare Ansicht, kann z.B. durch einen Entity-Relationship Graphen (ERG) repräsentiert werden.

Auf Grund der multiplen Abstraktionsebenen kann ein breites Spektrum von Analysen abgedeckt werden, die für das Reverse Engineering benötigt werden. Desweitern kann der ERG durch zusätzliche Informationen über Pipes, Remote Procedure Calls oder Shared Memory angereichert werden.

9 Zusammenfassung

Syntaxbäume sind eine exakte fein-granulare Zwischendarstellung von Quellcode. Abstrakte Syntaxbäume enthalten sowohl syntaktische als auch semantische Informationen, die für vielfältige Analysen genutzt werden können.

Literatur

[1] ALFRED V. AHO, RAVI SETHI, JEFFREY D. ULLMANN: *Complierbau*. Oldenbourg, 1999.

[2] *http://www.gnu.org*.

[3] RAINER KOSCHKE, JEAN-FRANCOIS GIRARD und MARTIN WÜRTHNER: *An Intermediate Representation for Integrating Reverse Engineering Analyses*. Presented at Working Converence on Reverse Engineering, Honolulu, HI, October 12-14 1998.

[4] REINHARD WILHELM, DIETER MAURER: *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer, 1992.

[5] RUDOLF FERENC, SUSAN ELLIOTT SIM, RICHARD C. HOLT RAINER KOSCHKE TIBOR GYIMOTHY: *Towards a Standard Schema for C/C++*.