OX: Ein Compilersystem für Attributierte Grammatiken

Kurt M Bischoff (Original)
René C. Kiesler (Übersetzung)

27. April 2004

Zusammenfassung

Das für Übersetzerbau dringend benötigte OX-Manual ist leider nur auf Englisch erhältlich. Ich habe es mir auf Deutsch übersetzt und stelle es unter http://www.kiesler.at jedem Interessenten zur Verfügung.

1 Wofür wird OX eingesetzt?

Lex und Yacc sind mächtige Tools, mit denen man gerne automatische Sprachersteller schreibt. Lex nimmt einen Satz von benutzerdefinierten regulären Ausdrücken und schreibt damit ein C-Programm. Dieses macht eine den regulären Ausdrücken entsprechende lexikalische Analyse. Yacc übersetzt benutzerdefinierte Grammatik-Regeln in C-Sourcecode für die Syntaxanalyse. Beide stellen zwar Ansatzpunkte für handgeschriebene semantische Aktionen in C zur Verfügung, viel mehr Unterstützung für automatische Sprachsemantik gibt es aber nicht.

Attributierte Parserbäume werden bei der Sprach-Evaluation gerne als Datenstruktur verwendet. Oft schreibt der Sprachenimplementierer den Code für das Erstellen, Traversieren und Auswerten von Bäumen sowie das Speichermanagement der dazugehörigen Datenstrukturen per Hand. Eine Yacc-Spezifikation definiert eine kontextfreie Sprache und eine Übersetzungstabelle von den gültigen Sätzen in den Parserbaum. Code für das Parserbaum-Management hingegen erstellt Yacc nicht automatisch.

Der Ox[1]-Benutzer kann einerseits Sprachen mit der Bekannten Lex und Yacc-Syntax erstellen. Andererseits kann er genauso eine bestehende Lex / Yacc-Spezifikation als Basis nehmen und zu dieser Semantik hinzufügen. Dies geschieht durch Hinzufügen von Deklarationen und Definitionen von typisierten Attributen der Parserbaum-Knoten.

Diese Spezifikation erzeugt eine *Attributierte Grammatik*, von welcher Ox automatisch einen *Evaluierer* in Yacc, Lex und C schreiben kann. Für eine gegebene Eingabe baut dieser Evaluierer einen Parsebaum auf, ermittelt die Evaluierungsreihenfolge für die Attribute des Baums, und führt für jedes einzelne Attribut die für dessen Evaluierung erforderliche Aktion durch. Dieser Parserbaum wird unabhängig von händisch mit C hinzugefügtem Code verwaltet. Sowohl der Parserbaum, als auch der benutzerdefinierte C Code können untereinander Informationen austauschen.

Außerdem kann der Ox-Benutzer ganz einfach Parserbaum-Traversierungen angeben, welche nach der Evaluierung der Baumattribute durchgeführt werden und auf solche Attribute verweisen. Mit solchen Traversierungen

[1] Ox war ursprünglich ein Homophone für ein anderes Akronym für „An Attribute Grammar Compiling System". Später hat man bemerkt, dass jedes Yak ein Ox ist und Ox die Funktionalität von Yacc generalisiert.

wird es deutlich einfacher, Codegenerierung und Übersetzungsstatistiken zu erhalten.

Der Sprachentwickler kann sich vom aufwendigen und fehleranfälligen Parserbaummanagement lösen. Die von Ox generierten Evaluierer sind deutlich schneller als gängigen, handgeschriebene Evaluierer, welche für jeden Knoten `malloc` aufrufen. Ox sorgt für Sicherheit, indem es die Definitionen auf Konsistenz und Vollständigkeit prüft. Dazu gehört auch die Erkennung, ob zirkulären Definitionen vielleicht die Evaluierung von Attributen verhindert haben.

Ox ist ein Präprozessor, der zwei oder mehrere Dateien entgegennimmt und dann diese in entsprechende Lex-[2] und Yacc-Dateien übersetzt. Mit wenigen Ausnahmen sind alle Lex-/Yacc Eingabepaare, die Erkenner oder Übersetzer darstellen, für Ox gültige Eingaben. Somit kann man bestehende Software mit Ox aufwerten. Außerdem ist es möglich, Ox-Dateien Schritt für Schritt händisch in „puren" Lex/Yacc Code zu übersetzen. Somit ist Ox gut für Sprachdesigner, Experimentierfreudige und Implementierer, die bereits mit Lex, Yacc und C vertraut sind.

2 Einführung

Es wird davon ausgegangen, dass der Leser bereits mit Yacc, Lex und C umgehen kann. Die Ox-Anweisungen sind im wesentlichen Ergänzungen dieser Sprachen[3]. Es hilft außerdem, wenn man ein gewisses theoretisches Verständnis der Attributierten Grammatik mitbringt.

Ox benötigt zumindest zwei Eingabedateien. Einerseits eine Syntaxspezifikation (stellt eine Yacc-Eingabe dar und wird Y-Datei genannt), die Ox in eine reine Yacc-Datei umwandelt. Andererseits eine oder mehrere lexikalische Spezifikationen (stellen Lex-Eingaben dar und werden L-Dateien genannt), die Ox in reine Lex-Dateien umwandelt. Im Regelfall gibt es genau eine L-Datei. Jedoch kann man auch Evaluierer schreiben, die mehrere solche Dateien benutzen. Dieses Handbuch beschreibt die Ox-spezifischen

[2] Dieses Handbuch geht davon aus, dass Lex als lexikalischer Analysator eingesetzt wird. Man kann jedoch auch handgeschriebenen Code einsetzen.

[3] „Yacc", „Lex" und „C" steht im Rahmen dieses Handbuchs für „Yacc oder Bison", „Lex oder Flex" beziehungsweise „C oder C++"

Konstrukte, die in diesen Dateien vorkommen dürfen. Auch die dazugehörigen Hintergründe werden näher erläutert. Für die Erklärungen werden hauptsächlich Fragmente der drei Ox-Beispiele im Anhang verwendet.

Innerhalb von Ox-Anweisungen dürfen C- und C++ Kommentare überall dort erscheinen, wo Whitespace erlaubt ist. Ox benennt seine globalen Variablen mit führendem yy. Verzichtet man also auf die Verwendung von globalen Variablen, deren Name mit yy anfängt, wird man auch keine Namenskonflikte bekommen.

3 Attributdeklarationen

In einer Yacc-Datei findet sich der Deklarationsbereich vor der ersten %% Markierung. Hier kann der Benutzer das Startsymbol, Tokens, Assoziativitäten, unions, C-Code und so weiter deklarieren. Es dürfen hier im Falle von Ox nicht nur die klassischen Yacc-Anweisungen verwendet werden, sondern auch sogenannte *Attributdeklarationen* in Ox-Syntax. Solche Deklarationen bestehen aus dem reservierten Wort `@attributes`, welches von `{`, einer Liste von Attributsdeklarationen und von `}` gefolgt wird. Im Anschluß steht eine Liste von Grammatiksymbolen.

Nehmen wir an, unsere Grammatik hätte ein Symbol namens `bitliste` und die folgende Attributsdeklaration:

```
@attributes {float wert; int skalierung, laenge;} bitliste
```

Eine solche Deklaration veranlässt den von Ox generierten Evaluierer, für jeden Knoten von `bitliste` im Parserbaum Speicher für einen float namens `wert` und die beiden Integer `skalierung` und `laenge` anzufordern.

Eine Liste von Attributsdeklarationen (im vorigen Beispiel der Teil zwischen den geschwungenen Klammern) ist nichts anderes, als die von C bekannte struct-Definition. Hier können Zahlen und Bezeichner im C-Stil, sowie die folgenden Zeichen und reservierten Worte verwendet werden:

```
* : ; , char short int long float double
    signed unsigned struct union enum
```

Die geschwungenen Klammern dürfen nicht geschachtelt werden. Innerhalb einer Liste von Attributdeklarationen darf man somit keine structs und unions verwenden. Davon abgesehen darf aber jeder elementare oder abgeleitete Datentyp, der auch in einem C-Programm gültig wäre, verwendet werden. In Yacc-Dateien finden sich oft C-Fragmente, die zwischen `%{` und `%}` eingeschlossen wurden. Auch diese sind in der Ox-Datei erlaubt. Selbstverständlich ist auch die Verwendung von zuvor definierten Typnamen der Art `struct`, `union`, `typedef` und `#define` als Attributtyp erlaubt.

Die Grammatiksymbole (die Bezeichner nach `}`) ist eine Liste von Yacc-Tokens (einschließlich Zeichenkonstanten), die auch leer sein darf, sowie von Nonterminals. Die einzelnen Symbole werden durch Whitespace getrennt.

Die Yacc-spezifischen Wörter `%token`, `%left`, `%right` und `%nonassoc` müssen vor etwaigen Attributdeklarationen eingesetzt werden.

3.1 Semantik der Attributdeklarationen

Eine Attributdeklaration erklärt Ox, dass jedes so definierte Grammatiksymbol Attribute mit der Definition entsprechendem Namen und Typ hat. Wenn a in der Attributsdeklaration aufscheint und s in der Liste der Grammatiksymbole, sagt man, dass a zu s *gehört* bzw. a ein Attribut von s ist. Jedes Grammatiksymbol hat seinen eigenen Namensraum. Sobald der Evaluierer einen Knoten erzeugt, der den Namen eines der gelisteten Symbole trägt, wird für jedes benannte Attribut der entsprechende Speicher angefordert. Ein so angeforderter Speicher wird *Attribut Instanz* (kurz: eine *Instanz*) im Parserbaum genannt. Instanzen *gehören* zu den entsprechenden Knoten.

4 Regeln und Attributvorkommnisse

Yacc Grammatikregeln (Produktionen) und die Objekte der `return`-Statements der Lex-Datei (ein jedes stellt ein Token dar) werden hier generisch als *Regeln* bezeichnet. Nachdem Ox sowohl Yacc- als auch Lexkonstrukte entgegennimmt und unverändert in die Ausgabedateien schreibt, werden auch die ensprechenden Ox-Konstrukte *Regeln* genannt. Jede Regel ist eine Abfolge von Grammatiksymbolen. Das Return-Statement einer Lex-Datei besteht demnach aus nur einem einzelnen Grammatiksymbol. Das erste

Symbol (ganz links) einer Regel wird die *Linke Seite (LHS)* genannt. Die *Rechte Seite (RHS)* enthält die restlichen Symbole einer Regel. Die Stelle eines Symbols in einer Regel wird zusammen mit einem Attribut dieses Symbol ergibt ein *Attributvorkommnis* (kurz: *Vorkomnis*) in dieser Regel genannt. Wenn das fragliche Attribut a ist, ist das Vorkommnis ein *Vorkommnis von* a. Sehen wir uns mal die vorige Attributdeklaration zusammen mit folgender Regel an:

```
num : bitliste PUNKT bitliste
```

Das Attributvorkommnis von `skalierung` des ersten Erwähnung von `bitliste` wird in Ox als `bitliste.0.skalierung` angesprochen, wärend das Attributvorkomniss von `skalierung` der letzten Erwähnung der rechten `bitliste` als `bitliste.1.skalierung` angesprochen wird.

Allgemein werden Attributvorkommnisse mit einem Grammatiksymbol, gefolgt von einem Punkt, gefolgt von einer optionalen nichtnegativen Ganzzahl und einem weiteren Punkt, gefolgt von einem Attributnamen des Symbols. Die nichtnegative Ganzzahl und den zweiten Punkt benötigen wir nur, wenn ein bestimmtes Grammatiksymbol öfter als einmal vorkommt. In so einem Fall werden die Vorkommnisse von links nach rechts mit 0, 1, ... durchnummeriert. Für das Symbol `X` mit dem Attribut `a` ist `X.a` gleichwertig mit `X.0.a`.

Eine bestimmte Regel und ein dazugehöriges Attributvorkommnis in dieser Regel stellen ein *Attribut Vorkommnis* in der Grammatik dar.

5 Attributdefinitionen

Für jede Regel kann der Ox-Anwender einen *Attribut Referenzbereich*, der durch `@{` und `@}` begrenzt wird angegeben. Wenn erwünscht, können hier die Definitionen der Attributvorkommnisse der gegebenen Regel eingetragen werden. Attributvorkommnisse können hier sowohl durch andere Vorkommnisse, als auch durch C-Code (globale Variablen, Konstanten, Makros und Funktionsaufrufe) definiert werden.

5.1 Vererbte vs. Synthetisierte Attribute

Ein Attributvorkommnis v in der Regel R ist genau dann und nur dann *synthetisiert*, wenn

1. v auf der LHS von R ist und der Attribut Referenzbereich von R eine Definition von v enthält, oder
2. v auf der RHS von R ist und der Attribut Referenzbereich von R keine Definition von v enthält.

Ein Attributvorkommnis v in einer Regel R ist genau dann und nur dann *vererbt*, wenn

1. v auf der LHS von R ist und der Attribut Referenzbereich von R keine Definition von v enthält, oder
2. v auf der RHS von R ist und er Attribut Referenzbereich von R eine Definition von v enthält.

Sollte ein Attribut in der Grammatik sowohl vererbte als auch synthetisierte Vorkommnisse haben, gibt es eine Fehlermeldung. Ein Attribut ist dann und nur dann *synthetisiert*, wenn es zumindest einmal vorkommt und jedes Vorkommnis synthetisiert ist. Ein Attribut ist dann und nur dann *vererbt*, wenn es zumindest einmal vorkommt und jedes Vorkommnis vererbt wurde. Eine Folge davon ist, dass das Startsymbol der Grammatik nur synthetisierte Attribute haben darf. Rückgabetoken als Regeln zu bezeichnen hebt die Gleichwertigkeit von Token und Nichtterminalen insofern hervor, als das jede Art von Symbol (vom Startsymbol abgesehen) sowohl synthetisierte, als auch vererbte Attribute haben darf. Nachdem jedes Symbol einen eigenen Namensraum hat, sind gleichbenannte Attribute verschiedener Symbole verschiedene Attribute und dürfen somit auch in einem Fall synthetisiert, im anderen vererbt sein.

Es gibt nun zwei spezielle Regeln in der Ox-Datei, die für jeden Parsebaumknoten (von der Wurzel abgesehen) interessant sind. Die *Heimatregel* ist die Regel, die einem bestimmten Knoten zugewiesen wird. Das heißt: Die Regel, deren LHS die Bezeichnung des Knotens ist und deren RHS Symbole

die Bezeichnungen der dazugehörigen Kinder des Knotens sind. Die *Vaterregel* wiederum wird an den Vater der Knoten angewendet. Die Attributsdefinition einer synthetisierten Attributsinstanz eines bestimmten Knotens steht in Verbindung mit der Heimatregel des Knotens (sie steht demnach im Referenzbereich des Attributes). Die Definitionen von vererbten Attributen sind entsprechend mit der Vaterregel verbunden.

In einer gültigen Eingabedatei ist jedes Attribut der Symbole der Regeln entweder synthetisiert oder vererbt, niemals beides. Somit passen alle Definitionen der Attribute zusammen, ganz widerspruchsfrei.

5.2 Attribut Referenzbereich in der Y-Datei

Der *Regelbereich* einer YaccDatei kommt gleich nach der ersten %% Markierung und enthält die Produktionen (Regeln) der Grammatik. Wie zuvor angemerkt, kann der Ox-Benutzer jede Regel mit einem Attribut Referenzbereich aufwerten. Diese werden durch `@{` und `@}` begrenzt und enthalten null oder mehr *Attribut Definitionen*. Wenn vorhanden, ist der Attribut Referenzbereich der letzte Eintrag (vom Strichpunkt abgesehen) in einer Regel[4]. Eine Attributsdefinition hat zwar konzeptionell einen *Abhängigkeitsteil* und einen *Auswertungsteil*. Syntaktisch dürfen die beiden Teile aber genausogut kombiniert werden. Es gibt drei Möglichkeiten, Attributdefinitionen aufzuschreiben. Innerhalb eines Referenzbereichs dürfen unterschiedliche Arten verwendet werden. Jede Definition beginnt mit einer *Definitionsmodus Ankündigung* (`@e`, `@i` oder `@m`) und wird durch einen weiteren Modusankündiger oder durch `@}` abgeschlossen.

5.2.1 Der Explizite Modus

Dies ist der mächtigste Modus und gleichzeitig der, bei dem am meisten Schreibarbeit gefragt ist. Hier wird eine Attributionsdefinition mit `@e` (kurz für *explicit*) eingeleitet. Im Anschluss folgen ein *Abhängigkeitsausdruck* und ein *Auswertungsausdruck*. Im folgenden Beispiel besteht der Attribut Referenzbereich aus drei Attributsdefinitionen, welche alle im expliziten Modus formuliert wurden.

[4]Somit steht der Referenzbereich nicht vor irgendeiner Yacc-Aktion oder dem reservierten Yacc-Wort %prec in der Regel. Alle folgenden Bezeichner müssen die LHS der nächsten Regel sein

```
num   :   bitliste PUNKT bitliste @{
       @e num.wert  :  bitliste.0.wert     bitliste.1.wert  ;
         @num.wert@ = @bitliste.0.wert@ + @bitliste.1.wert@ ;

       @e bitliste.0.skalierung  :   ;
         @bitliste.0.skalierung@ = 0 ;

       @e bitliste.1.skalierung  :   bitliste.1.laenge  ;
         @bitliste.1.skalierung@ = -@bitliste.1.laenge@ ;
      @}
      ;
```

Ein Abhängigkeitsausdruck zeigt explizit die Auswertungsreihenfolge und ist eine nichtleere Liste von Attributvorkommnissen der Regel, gefolgt von einem Doppelpunkt, gefolgt von einer gegebenenfalls leeren Liste und einem abschließenden Strichpunkt. Die Vorkommnisse links des Doppelpunkt werden *abhängig von* denen zur Rechten (und somit *Abhängige*) genannt und sind die *definierten* Vorkommnisse in der gegebenen Attributsdefinition. Die Vorkommnisse zur Rechten heißen *dependees*[5] von denen der linken. Ein Auswertungsausdruck ist im wesentlichen ein C Codefragment, welches *Attributreferenzen* enthalten kann, von welchen jede ein mit `@` eingeschlossenes Attributvorkommnis ist. Attributreferenzen verhalten sich wie C Variablen, somit sind auch sämtliche arithmetische, logische und Zeigeroperationen gültig. Der Auswertungsausdruck folgt gleich nach dem Strichpunkt des Abhängigkeitsausdrucks.

Der von Ox generierte Evaluierer wählt die Auswertungsreihenfolge so, dass die Auswertungsausdrücke für alle dependees in der Definition vor denen der Abhängigen ausgeführt wird. Normalerweise gibt es nur einen Abhängigen pro Attributsdefinition. Manchmal kann der Code allerdings, der Kompaktheit wegen, auch aus mehreren Attributsvorkommnissen in einer Abhängigkeitsliste bestehen. Der Abhängigkeitsausdruck wird im Context der Abhängigkeiten *als Ganzes* ausgeführt, und nicht etwa für jeden einzelnen Abhängigen. Diese Vorgehensweise wird auch *auflösen* der Vorkommnisse der Attributsinstanzen dieses Sets genannt.

[5] (?)habe leider weder im PONDS Wörterbuch, noch in Babylon, noch auf leo.org noch im TU Chemniz-Wörterbuch eine passende Übersetzung gefunden. Was ist das Gegenteil von Abhängige?

5.2.2 Der Implizite Modus

Der *implizite Modus*, welcher meistens für Attibutsdefinitionen verwendet wird, verbindet syntaktisch den Abhängigkeitsteil mit dem Auswertungsteil. Das folgende Ox-Beispiel entspricht dem vorigem expliziten Beispiel.

```
num   :   bitliste PUNKT bitliste @{
       @i @num.wert@ = @bitliste.0.wert@ + @bitliste.1.wert@;
       @i @bitliste.0.skalierung@ = 0;
       @i @bitliste.1.skalierung@ = -@bitliste.1.laenge@;
      @}
      ;
```

In diesem Modus hat eine Attributsdefinition die Form `@i`, gefolgt von einem Auswertungsausdruck. Der Modusankündiger `@i` informiert Ox, dass die Definition nur eine Abhängigkeit hat – genau das erste Attributsvorkommnis des Auswertungsausdrucks. Die dependees in der Definition bestehen aus allen *anderen* Attributsvorkommnissen, auf die im Auswertungsausdruck verwiesen wurde.

5.2.3 Der gemischte Modus

Definitionen im *gemischten Modus* werden durch das reservierte Wort `@m` angekündigt. Es folgen eine oder mehrere Abhängige, ein Strichpunkt und ein Auswertungsausdruck. Die Vorkommnisse, auf die im Auswertungsausdruck verwiesen wird (von denen abgesehen, die auch zwischen `@m` und Strichpunkt vorkommen) werden als dependees der Definition angesehen. Demnach werden die Abhängigkeiten explizit dargestellt und die dependees implizit. Wieder das gleiche Beispiel wie zuvor, diesmal im gemischten Modus:

```
num   :   bitliste PUNKT bitliste @{
       @m  num.wert ;
           @num.wert@ = @bitliste.0.wert@ + @bitliste.1.wert@;
       @i @bitliste.1.skalierung@ = - @bitliste.1.laenge@;
       @m bitliste.0.skalierung ; @bitliste.0.skalierung@ = 0;
      @}
      ;
```