Der Compilergenerator Coco/R

Hanspeter Mössenböck, Albrecht Wöß, Markus Löberbauer

Johannes Kepler Universität Linz, Institut für Praktische Informatik,
Altenbergerstr. 69, 4040 Linz
{moessenboeck,woess,loeberbauer}@ssw.uni-linz.ac.at

Coco/R ist ein Compilergenerator, der aus einer attributierten Grammatik einen Scanner und einen Parser samt semantischen Aktionen erzeugt. Die erste Version dieses Werkzeugs entstand in den frühen achtziger Jahren an der Universität Linz unter der Leitung von Peter Rechenberg. Inzwischen existiert Coco/R in mehreren Versionen für moderne Programmiersprachen und wird weltweit verwendet.

In diesem Beitrag werden die Grundlagen, die Arbeitsweise und die Geschichte von Coco/R beschrieben. Ein Teil des Beitrags ist einem neuen Merkmal von Coco/R gewidmet, nämlich der Fähigkeit, Grammatiken verarbeiten zu können, die nicht LL(1) sind.

1 Einleitung

Die Übersetzung von Programmiersprachen – allgemein: die Verarbeitung strukturierter Datenströme – lässt sich mit *attributierten Grammatiken* spezifizieren. Die Syntax des Eingabestroms wird dabei durch eine kontextfreie Grammatik beschrieben und die nötigen Übersetzungsaktionen durch Attribute und semantische Aktionen, die an geeigneten Stellen in die Grammatik eingefügt werden.

Attributierte Grammatiken dienen aber nicht nur der *Beschreibung* eines Übersetzungsprozesses, sondern können auch als seine *Implementierung* auf hoher Abstraktionsebene aufgefasst werden. Ein Werkzeug kann daraus die wesentlichen Teile eines Compilers erzeugen, nämlich einen lexikalischen Analysator (Scanner) und einen Syntaxanalysator (Parser) samt semantischen Übersetzungsaktionen. Solche Werkzeuge werden *Compilergeneratoren* genannt.

Seit den frühen achtziger Jahren wird an der Universität Linz an solchen Werkzeugen gearbeitet. Unter anderem ging daraus der Compilergenerator *Coco/R* hervor, der auch heute noch weiterentwickelt wird. Eine dieser Weiterentwicklungen erlaubt es, auch Sprachen verarbeiten zu können, deren Grammatik nicht LL(1) ist.

Dieser Beitrag ist wie folgt aufgebaut: In Abschnitt 2 geben wir einen kurzen Überblick über attributierte Grammatiken und ihren Einsatz im Übersetzerbau. Abschnitt 3 beschreibt den Compilergenerator Coco/R. In Abschnitt 4 werden LL(1)-Konflikte und ihre Beseitigung behandelt und Abschnitt 5 erläutert, wie man mit Coco/R unvermeidbare LL(1)-Konflikte in den Griff bekommt. Abschnitt 6 erzählt schließlich die Geschichte der in Linz entwickelten Compilergeneratoren.

2 Attributierte Grammatiken

Attributierte Grammatiken wurden 1968 von *Knuth* eingeführt, um die Semantik von Programmiersprachen und ihre Übersetzung zu spezifizieren [Knuth68]. In ihrer ursprünglichen Form sind sie *deklarative Beschreibungen*, in denen der Syntaxbaum eines Programms mit Attributen (z.B. mit dem Typ eines Ausdrucks oder der Adresse einer Variablen) dekoriert wird, die sich durch Berechnungsregeln aus anderen Attributen ergeben.

Im Sinne von *Rechenberg* [Rech79, Rech84] sehen wir attributierte Grammatiken jedoch eher als *prozedurale Beschreibungen*, bei denen Grammatikregeln als Erkennungsprozeduren und Attribute als Parameter dieser Prozeduren aufgefasst werden. Diese Sichtweise kommt dem Übersetzerbauer entgegen, der aus ihr nahezu mechanisch einen Syntaxanalysator mit semantischen Aktionen ableiten kann.

Attributierte Grammatiken dieser Art bestehen im wesentlichen aus drei Teilen, die im Folgenden kurz erläutert werden.

Kontextfreie Grammatik

Eine kontextfreie Grammatik beschreibt die Syntax der zu übersetzenden Sprache. Sie besteht aus *Terminalsymbolen* und *Nonterminalsymbolen* sowie aus Ableitungsregeln (sog. *Produktionen*), die für jedes Nonterminalsymbol angeben, wie es in Terminal- und Nonterminalsymbole abgeleitet werden kann. Als Schreibweise verwenden wir die von *Wirth* vorgeschlagene erweiterte Backus-Naur-Form (EBNF) [Wirth77], die wie folgt definiert ist:

Tabelle 1: Wirth'sche EBNF

Zeichen	Bedeutung	Beispiel	Erläuterung
=	trennt die Seiten einer Produktion	A = x y z.	
.	schließt eine Produktion ab	A = x y z.	
\| | trennt Alternativen | x \| y | x oder y |
( ) | fasst Alternativen zusammen | (x \| y) z | xz oder yz |
[ ] | wahlweises Vorkommen | [x] y | y oder xy |
{ } | 0- oder mehrmaliges Vorkommen | {x} y | y oder xy oder xxy oder ... |

Eines der Nonterminalsymbole ist das *Startsymbol*, aus dem alle Sätze der Sprache abgeleitet werden können. Bei den Terminalsymbolen unterscheidet man zwischen *einfachen Terminalsymbolen* (z.B. "while", ";"), die sich selbst bedeuten und *Terminalklassen* (z.B. *ident*, *number*), die in mehreren Ausprägungen vorliegen können. Terminalsymbole haben ebenfalls eine Struktur, die aber nicht zur Syntax der Sprache gehört, sondern auf lexikalischer Ebene (vom Scanner) analysiert wird.

Hier ist ein Beispiel einer kontextfreien Grammatik für vereinfachte Variablendeklarationen in C#:

```
VarDecl   = Type IdentList ";".
Type      = "int" | "char".
IdentList = ident {"," ident}.
```

Jede Produktion kann als Erkennungsprozedur aufgefasst werden. Die Vorkommen von *Type* und *IdentList* in der Produktion von *VarDecl* bedeuten den Aufruf der entsprechenden Erkennungsprozeduren für *Type* und *IdentList*.

Attribute

Terminal- und Nonterminalsymbole können Attribute haben, die semantische Werte bedeuten, welche bei der Erkennung dieses Symbols entstehen oder vom Kontext dieses Symbols an die Erkennungsprozedur mitgegeben werden. Im ersten Fall spricht man von *Ausgangsattributen* (*synthesized attributes*), im zweiten Fall von *Eingangsattributen* (*inherited attributes*). Die Symbole der obigen Grammatik für Variablendeklarationen könnten zum Beispiel folgende Attribute aufweisen:

VarDecl	keine Attribute
$\text{Type}\uparrow_{\text{size}}$	Ausgangsattribut *size*, das die Typgröße in Bytes beschreibt
$\text{IdentList}\downarrow_{\text{size}}$	Eingangsattribut *size*, das die Typgröße der Variablen in *IdentList* beschreibt
$\text{ident}\uparrow_{\text{name}}$	Ausgangsattribut *name*, das den Text eines Namens beschreibt

Attribute können als Parameter der Erkennungsprozeduren für die entsprechenden Syntaxsymbole aufgefasst werden. Terminalklassen dürfen nur Ausgangsattribute haben, deren Wert vom Scanner geliefert wird. Einfache Terminalsymbole haben keine Attribute.

Semantische Aktionen

Um die Eingabesprache in die Ausgabesprache zu übersetzen, müssen *semantische Aktionen* ausgeführt werden, die die Werte von Attributen berechnen, semantische Bedingungen (sog. *Kontextbedingungen*) prüfen, Datenstrukturen wie die Symbolliste verwalten und schließlich Sätze der Ausgabesprache erzeugen.

Semantische Aktionen sind Anweisungen in einer beliebigen Programmiersprache (z.B. in C#). Sie werden an jenen Stellen in die Produktionen eingefügt, an denen sie während der Erkennung ausgeführt werden sollen.

Das folgende Beispiel zeigt die Grammatik für Variablendeklarationen samt semantischen Aktionen. Als Übersetzungsergebnis soll hier eine Liste der deklarierten Variablen mit ihren Adressen ausgegeben werden. Semantische Aktionen werden dabei zwischen die Symbole (. und .) geschrieben.

```
VarDecl = Type↑size IdentList↓size ";".

Type↑size
= "int"              (. size = 4; .)
| "char"             (. size = 2; .) .

IdentList↓size
= ident↑name         (. Print(name, 0); int adr = size; .)
  { "," ident↑name   (. Print(name, adr); adr += size; .)
  }.
```

Attributierte Grammatiken sind nicht nur im Übersetzerbau nützlich, sondern in allen Situationen, in denen eine strukturierte Eingabe zu verarbeiten ist, wie zum Beispiel bei Personendaten, Stücklisten oder Konfigurationsdateien. Sobald die Syntax der Eingabe in Form einer Grammatik vorliegt, bildet diese das Gerüst der Übersetzung, das durch semantische Aktionen vervollständigt wird. Man spricht daher auch von *syntaxgetriebener Übersetzung* (*syntax-directed translation*).

3 Der Compilergenerator Coco/R

Coco/R [CocoR] ist ein Werkzeug, das eine attributierte Grammatik in einen Scanner und einen Parser übersetzt, der nach der Methode des rekursiven Abstiegs [Wirth96] arbeitet. Der Benutzer muss noch ein Hauptprogramm hinzufügen, das den Parser aufruft, sowie Klassen, die von den semantischen Aktionen des Parsers benutzt werden (z.B. Klassen für die Symbolliste, die Codeerzeugung oder die Optimierung; siehe Abb. 1). Wir beschreiben hier die Version von Coco/R, die in C# implementiert ist und Scanner und Parser in C# erzeugt. Es gibt aber auch Versionen für Java, C/C++, Pascal, Modula und Oberon.

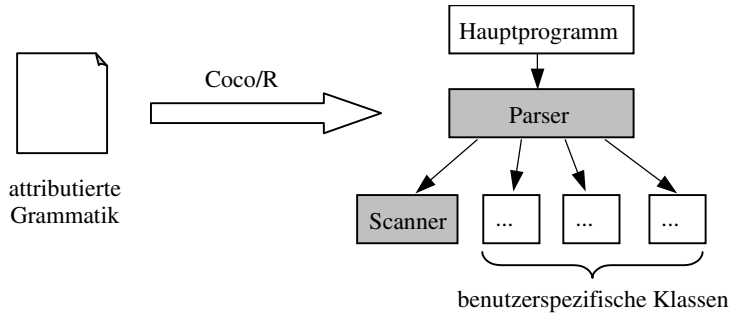

Abb. 1 *Arbeitsweise von Coco/R*

Neben der attributierten Grammatik, die den Parser und seine semantischen Aktionen spezifiziert, enthält die Eingabe für Coco/R auch eine Scannerbeschreibung sowie semantische Deklarationen, die wir nun der Reihe nach erklären.

```
CocoInput = "COMPILER" ident
            SemanticDeclarations
            ScannerDescription
            ParserDescription
            "END" ident ".".
```

Scannerbeschreibung

In diesem Teil der Compilerbeschreibung wird die Struktur aller Terminalklassen (z.B. *ident*) definiert. Coco/R erzeugt daraus einen Scanner in Form eines deterministischen endlichen Automaten, der vom Parser aufgerufen wird und ihm die Symbole des Eingabestroms liefert. Einfache Terminalsymbole (z.B. "while") müssen nicht in

der Scannerbeschreibung deklariert werden, sondern man kann sie einfach in den Produktionen verwenden.

Neben den Terminalklassen werden auch die verwendeten Zeichenmengen (z.B. Buchstaben und Ziffern) definiert sowie die Struktur von Kommentaren und jene Zeichen, die der Scanner ignorieren soll. Hier ist ein Beispiel einer Scannerbeschreibung:

```
CHARACTERS
  letter = 'A'..'Z' + 'a'..'z'.
  digit  = '0'..'9'.
  cr     = '\u000d'.
  lf     = '\u000a'.
TOKENS
  ident  = letter {letter | digit}.
  number = digit {digit}.
COMMENTS FROM "/*" TO "*/" NESTED
IGNORE cr + lf
```

Jedes vom Scanner gelieferte Terminalsymbol ist ein Objekt der Klasse *Token*:

```
class Token {
  int kind;     // Symbolnummer
  int pos;      // Quelltext-Position des Symbols
  int line;     // Zeilennummer des Symbols
  int col;      // Spaltennummer des Symbols
  string val;   // Symbol als Zeichenkette
}
```

Bei Dateiende wird ein Symbol mit der Nummer 0 geliefert, für unbekannte Symbole wird ein Symbol mit der vordefinierten Nummer *noSym* geliefert.

Parserbeschreibung

Die Parserbeschreibung besteht aus einer attributierten Grammatik der zu übersetzenden Sprache. Attribute werden zwischen spitze Klammern geschrieben, semantische Aktionen zwischen "(." und ".)". Hier ist ein Beispiel für die Produktion *IdentList* aus Abschnitt 2.

```
IdentList<int size>     (. string name; int adr; .)
= Ident<out name>       (. Print(name, 0); adr = size; .)
  { "," Ident<out name> (. Print(name, adr); adr += size; .)
  }.
```

In der Eingabesprache von Coco/R liefern Terminalsymbole keine Attribute. Daher wurde im obigen Beispiel statt *ident* ein Hilfsnonterminalsymbol *Ident* verwendet, das wie folgt implementiert ist:

```
Ident<out string name>
= ident     (. name = t.val; .) .
```

Der von Coco/R erzeugte Parser speichert das zuletzt erkannte Terminalsymbol in der globalen Variablen *t* vom Typ *Token*. Im obigen Beispiel steht daher der semantische Wert von *ident* in *t.val*.

Coco/R erzeugt aus der attributierten Grammatik einen Parser im rekursiven Abstieg. Jede Produktion wird in eine Parsermethode übersetzt. Attribute werden zu Parametern. Semantische Aktionen werden an der Stelle ihres Vorkommens in den Parsercode eingefügt. Für die Produktion *IdentList* ergibt sich somit folgende Parsermethode (in C#):

```
static void IdentList(int size) {
  string name; int adr;
  Ident(out name);
  Print(name, 0); adr = size;
  while (sym == comma) {
    Scan();
    Ident(out name);
    Print(name, adr); adr += size;
  }
}
```

Eine attributierte Grammatik ist meist kürzer und übersichtlicher als die entsprechenden Parsermethoden, da Syntax und Semantik deutlicher getrennt sind, Wiederholungen und Optionen kompakter codiert werden und auch die Fehlerbehandlung nicht explizit spezifiziert werden muss. Für die Fehlerbehandlung gibt man in der Grammatik einfach Synchronisationsstellen an, an denen der Parser im Fehlerfall versucht, den Eingabestrom wieder mit der Grammatik zu synchronisieren [CocoR].

Semantische Deklarationen

Am Anfang einer Compilerbeschreibung kann man statische Variablen und Methoden deklarieren, die man in den semantischen Aktionen der attributierten Grammatik benutzen möchte. Sie werden von Coco/R in die erzeugte Parserklasse integriert. Die in *IdentList* verwendete Methode *Print* könnte man zum Beispiel wie folgt deklarieren:

```
COMPILER ...

  static void Print(string name, int adr) {
    Console.WriteLine(name + ": " + adr);
  }

CHARACTERS ...
```

Natürlich kann man von semantischen Aktionen aus auch Objekte anderer Klassen (z.B. der Symbolliste) ansprechen.

Grammatikprüfungen

Coco/R erzeugt nicht nur einen Scanner und einen Parser, sondern überprüft auch die Vollständigkeit und Eindeutigkeit der Grammatik. So wird zum Beispiel sichergestellt, dass jedes Nonterminalsymbol durch eine Produktion definiert wird, dass die Grammatik zyklenfrei ist und dass sie die LL(1)-Bedingung erfüllt (siehe nächster Abschnitt). Dies ist vielleicht der größte Vorteil eines Werkzeugs wie Coco/R gegenüber einer manuellen Compilerimplementierung, da insbesondere LL(1)-Verletzungen ohne Werkzeuge äußerst mühsam festzustellen sind.

4 LL(1)-Konflikte und ihre Beseitigung

Um einen Parser im rekursiven Abstieg erzeugen zu können, muss die zugrunde liegende Grammatik LL(1) sein. Eine Grammatik heißt LL(1) (d.h. analysierbar von **l**inks nach rechts mit **l**inkskanonischen Ableitungen und **1** Vorgriffssymbol), wenn an jeder Stelle, an der man zwischen mehreren Alternativen wählen kann, gilt, dass die terminalen Anfänge dieser Alternativen paarweise disjunkt sind. Mit anderen Worten: der Parser muss jederzeit mit einem einzigen Vorgriffssymbol entscheiden können, welche von mehreren möglichen Alternativen er wählen soll.

In EBNF-Grammatiken können LL(1)-Konflikte bei expliziten Alternativen, bei Optionen und bei Iterationen vorkommen. Folgende Aufstellung zeigt, was in diesen Situationen zu prüfen ist (griechische Symbole bedeuten dabei beliebige EBNF-Ausdrücke wie z.B. `a[b]c`; *first(α)* bezeichnet die Menge der terminalen Anfänge des EBNF-Ausdrucks α; *follow(A)* bezeichnet die terminalen Nachfolger des Nonterminalsymbols `A`):

- **Explizite Alternativen** (z.B. A = α | β | γ.)
  es muss gelten:
  $first(\alpha) \cap first(\beta) = \{\} \wedge first(\alpha) \cap first(\gamma) = \{\} \wedge first(\beta) \cap first(\gamma) = \{\}$
- **Optionen**
  z.B.: A = [α] β. es muss gelten: $first(\alpha) \cap first(\beta) = \{\}$
  z.B.: A = [α]. es muss gelten: $first(\alpha) \cap follow(A) = \{\}$
- **Iterationen**
  z.B. A = {α} β. es muss gelten: $first(\alpha) \cap first(\beta) = \{\}$
  z.B. A = {α}. es muss gelten: $first(\alpha) \cap follow(A) = \{\}$

Viele Grammatiken von heute verwendeten Programmiersprachen wie Java, C oder C# sind nicht LL(1). Allerdings kann man LL(1)-Konflikte oft durch Transformationen beseitigen, was wir im Folgenden zeigen.

Transformationen zur Beseitigung von LL(1)-Konflikten

Faktorisierung. LL(1)-Konflikte können oft durch Faktorisierung beseitigt werden, indem man die gemeinsamen Teile der in Konflikt stehenden Alternativen heraushebt. Zum Beispiel kann die Produktion

```
A = a b c | a b d.
```

in folgende Produktion umgewandelt werden, in der es keinen Konflikt mehr gibt:

```
A = a b (c | d).
```

Linksrekursion. Linksrekursive Regeln stellen immer einen LL(1)-Konflikt dar, da der rekursive und nichtrekursive Teil gleiche terminale Anfänge haben. In der Produktion

```
A = A b | c.
```

beginnen zum Beispiel beide Alternativen mit *c*, da *first(A)* = {*c*}. Linksrekursion kann jedoch immer in Iteration umgewandelt werden, z.B.:

```
A = c {b}.
```

Probleme bei der Beseitigung von LL(1)-Konflikten

Harte Konflikte. Es gibt LL(1)-Konflikte, die man auch durch Transformation nicht so einfach beseitigen kann, wie zum Beispiel in folgender (vereinfachter) Produktion aus der C#-Grammatik:

```
IdentList = ident {"," ident} [","].
```

Der Konflikt tritt auf, weil sowohl die Iteration als auch die Option mit einem Komma beginnen. Das lässt sich auch durch Umformungen nicht so leicht ändern. Eine Lösungsmöglichkeit besteht darin, mehrere Symbole vorauszulesen, um zu sehen, was auf das Komma folgt: folgt *ident*, dann muss die Iteration gewählt werden, folgt ein Nachfolger von *IdentList*, dann muss man die Option wählen. Wir werden im nächsten Abschnitt zeigen, wie man dieses Problem mit Coco/R lösen kann.

Lesbarkeit. Manchmal kann eine Transformation auch die Lesbarkeit einer Grammatik beeinträchtigen. Betrachten wir dazu folgendes Beispiel (wieder in vereinfachter Form aus der C#-Grammatik entnommen):

```
UsingClause = "using" [ident "="] Qualident ";".
Qualident   = ident {"." ident}.
```

Der Konflikt tritt in *UsingClause* auf, da sowohl der optionale Teil als auch *Qualident* mit *ident* beginnen. Obwohl man diesen Konflikt auf folgende Weise durch Umformung beseitigen könnte

```
UsingClause = "using" ident
              ( {"." ident}
              | "=" Qualident
              ) ";" .
```

würde die Lesbarkeit deutlich darunter leiden, und die semantischen Aktionen für *Qualident* müssten in *UsingClause* wiederholt werden. Es ist in diesem Fall besser, den Konflikt mit den Mitteln zu lösen, die wir im nächsten Abschnitt vorstellen.

Probleme mit der Semantikverarbeitung. In manchen Fällen kann man einen LL(1)-Konflikt nicht durch Faktorisierung beseitigen, da die beiden zu faktorisierenden Alternativen auf unterschiedliche Weise semantisch verarbeitet werden müssen, z.B.:

```
A = ident (. x = 1; .) {"," ident (. x++; .) } ":"
  | ident (. P(); .) {"," ident (. Q(); .) } ";".
```

Auch dieses Problem lässt sich mit den Techniken lösen, die im nächsten Abschnitt vorgestellt werden.

5 Lösung von LL(1)-Konflikten in Coco/R

Coco/R erzeugt Parser im rekursiven Abstieg und verlangt daher, dass die zugrunde liegende Grammatik LL(1) ist. Um auch Grammatiken von Sprachen wie Java oder C# verarbeiten zu können, die nicht LL(1) sind, haben wir Coco/R um sogenannte *Konfliktlöser* erweitert, wie sie in ähnlicher Form auch in anderen Compilergeneratoren (z.B. in *JavaCC* [JCC] oder *ANTLR* [ANTLR]) vorkommen.

Ein Konfliktlöser ist ein boolescher Ausdruck, der am Beginn einer in Konflikt stehenden Alternative in die Grammatik eingefügt wird und mittels Vorgriff oder semantischer Abfragen prüft, ob diese Alternative im Eingabestrom vorliegt. Liefert der Konfliktlöser *true*, so wird die Alternative gewählt, liefert er *false*, wird die nächste Alternative untersucht. Ein Konfliktlöser hat die Form

```
Resolver = "IF" "(" {ANY} ")".
```

wobei `{ANY}` ein beliebiges Programmstück bedeutet, das einen booleschen Ausdruck darstellt. In den meisten Fällen wird in diesem Programmstück eine Funktionsmethode aufgerufen, die *true* oder *false* zurückgibt.

Der in Abschnitt 4 angesprochene LL(1)-Konflikt in *UsingClause* kann damit zum Beispiel folgendermaßen gelöst werden:

```
UsingClause = "using" [IF (IsAlias()) ident "="]
              Qualident ";".
```

IsAlias ist eine vom Benutzer geschriebene Methode, die zwei Symbole vorausliest. Folgt auf *ident* das Symbol "=", liefert sie *true*, andernfalls *false*.

Vorgriff um mehrere Symbole

Der von Coco/R erzeugte Parser merkt sich das zuletzt erkannte Terminalsymbol sowie das aktuelle Vorgriffssymbol in den folgenden beiden globalen Variablen:

```
Token t;   // zuletzt erkanntes Terminalsymbol
Token la;  // Vorgriffssymbol (lookahead symbol)
```

Will man weiter als ein Symbol vorausschauen, so bietet der von Coco/R erzeugte Scanner dafür zwei Methoden an:

- *StartPeek()* initialisiert die Vorausschau im Scanner.
- *Peek()* liefert bei jedem Aufruf das jeweils nächste Symbol in Form eines *Token*-Objekts (beginnend mit dem Symbol nach *la*). Die mit *Peek* gelieferten Symbole werden jedoch nicht aus dem Eingabestrom entfernt, so dass sie der Scanner später beim normalen Weiterlesen nochmals liefert.

Die Methode *IsAlias* kann mit Hilfe dieser beiden Methoden folgendermaßen implementiert werden:

```
static bool IsAlias() {
  Scanner.StartPeek();
  Token x = Scanner.Peek();
  return la.kind == Tokens.ident && x.kind == Tokens.eql;
}
```

Der ebenfalls in Abschnitt 4 beschriebene Konflikt

```
A = ident (. x = 1; .) {"," ident (. x++; .)} ":"
  | ident (. P(); .) {"," ident (. Q(); .)} ";".
```

lässt sich durch folgende Grammatikregel

```
A = IF (FollowedByColon())
    ident (.x = 1;.) {"," ident (.x++;.)} ":"
  | ident (. P(); .) {"," ident (. Q(); .)} ";".
```

und folgende Implementierung der Funktion *FollowedByColon* lösen:

```
static bool FollowedByColon() {
  Scanner.StartPeek();
  Token x = la;  // la ... lookahead token
  while (x.kind == Tokens.ident || x.kind == Tokens.comma)
    x = Scanner.Peek();
  return x.kind == Tokens.colon;
}
```

Konfliktlösung durch semantische Informationen

Ein Konfliktlöser kann seine Prüfung nicht nur auf Grund mehrerer Vorgriffssymbole, sondern auch auf Grund beliebiger anderer Informationen vornehmen. Dabei kann er zum Beispiel auf die Symbolliste oder auf andere semantische Informationen zugreifen.

In vielen Sprachen tritt zum Beispiel folgender LL(1)-Konflikt zwischen Zuweisungen und Deklarationen auf:

```
Statement = Type IdentList ";"
          | ident "=" Expression ";"
          | ...
Type      = ident | ...
```

Beide Alternativen von *Statement* beginnen mit *ident*. Der Konflikt kann gelöst werden, indem man prüft, ob *ident* einen Typ oder eine Variable bezeichnet:

```
Statement = IF (IsType()) Type IdentList ";"
          | ident "=" Expression ";".
          | ...
```

Die Implementierung von *IsType* sucht *ident* in der Symbolliste und liefert *true*, wenn es sich um einen Typnamen handelt:

```
static bool IsType() {
  if (la.kind = Tokens.ident) {
    Object obj = SymTab.find(la.val);
    return obj != null && obj.kind == Type;
  } else return false;
}
```

Übersetzung von Konfliktlösern

Konfliktlöser werden von Coco/R ähnlich wie semantische Aktionen behandelt und an der Stelle ihres Auftretens in den Quellcode des erzeugten Parsers eingefügt. Die obige Produktion für *UsingClause* führt daher zu folgender Parsermethode:

```
static void UsingClause() {
  Expect(Tokens.using);
  if (IsAlias()) {
    Expect(Tokens.ident); Expect(Tokens.eql);
  }
  Qualident();
  Expect(Tokens.semicolon);
}
```

6 Geschichte von Coco/R

Im Rahmen dieser Festschrift soll hier ein kurzer Abriss der Geschichte von Coco/R gegeben werden, der jene Personen würdigt, die an dieser Entwicklung beteiligt waren.

Die Forschung an compilererzeugenden Werkzeugen hat an der Universität Linz eine lange Tradition. Sie beginnt mit den Arbeiten von *Peter Rechenberg* an attributierten Grammatiken [Rech79, Rech84], die zunächst als reine Spezifikationen von Übersetzungsprozessen betrachtet und manuell in einen Parser übersetzt wurden.

Unter der Leitung von Rechenberg wurde 1981 erstmals ein Parsergenerator namens *TDG* (Topdown-Graph-Generator) entwickelt, der EBNF-Grammatiken in tabellengesteuerte Topdown-Parser übersetzte [Möss81]. Attribute gab es keine. Semantische Aktionen konnten durch Nummern angegeben werden. Der Benutzer hatte eine Semantikauswerteprozedur zu schreiben, die vom Parser für jede semantische Aktion aufgerufen wurde und an Hand der Aktionsnummer eines von mehreren Codestücken auswählte, das dann als semantische Aktion ausgeführt wurde. Produktionen hatten keine lokalen Variablen, sondern alle Variablen waren global, so dass man bei rekursiven Grammatikregeln mit Hilfe eines Kellers selbst dafür sorgen musste, dass lokal benutzte Variablenwerte nicht überschrieben wurden. TDG war in PL/M geschrieben, einer PL/I-ähnlichen Sprache für Mikrocomputer.

Als Fortsetzung dieser Arbeiten entstand im Rahmen einer Diplomarbeit der Compiler-Compiler *Coco*, der zunächst eigentlich nur ein Parser-Generator war, aber bereits attributierte Grammatiken mit Attributen und semantischen Aktionen verarbeiten konnte [Möss83]. Coco erzeugte wie TDG tabellengesteuerte Topdown-Parser [Rech83]. Die semantischen Aktionen wurden automatisch durchnummeriert und wie bei TDG in einer Semantikauswerteprozedur zusammengefasst. Auch in Coco gab es noch keine lokalen Variablen in Produktionen, sondern globale Variablen mussten mit Hilfe eines Kellers über rekursive Produktionen hinweg gerettet werden.

Coco wurde zunächst in PL/M, später in Pascal und Modula-2 implementiert und anderen Universitäten und Firmen kostenlos zur Verfügung gestellt. Da dies großen Anklang fand, wurden Coco und die in ihm verwendeten Übersetzerbau-Techniken in

einem Buch beschrieben [ReMö85], das später auch ins Englische [ReMö88a] und sogar ins Japanische [ReMö88b] übersetzt wurde.

Als nächster Schritt wurde auch die Erzeugung von Scannern automatisiert. Der Scannergenerator *Alex* erzeugte aus einer Spezifikation der Terminalsymbole und ihrer Struktur einen Scanner in Form eines endlichen Automaten [Möss86]. Der Benutzer hatte jedoch selbst dafür zu sorgen, dass die vom Scanner gelieferten Terminalsymbolnummern mit denen übereinstimmten, die der Parser erwartete.

Da Coco Topdown-Parser erzeugte, war sein Einsatz auf LL(1)-Grammatiken beschränkt. Daher wurde ein weiterer Compiler-Generator namens *Smart* implementiert, der nahezu dieselbe Eingabesprache wie Coco aufwies, aber LALR(1)-Parser erzeugte [Möss87]. Die Besonderheit von Smart war, dass die Syntaxanalyse im erzeugten Parser bottomup erfolgte (und somit mächtigere Sprachklassen analysieren konnte), während die semantischen Aktionen topdown an Hand eines vom Parser erstellten und kompakt codierten Syntaxbaums ausgeführt wurden.

Eine offensichtliche Schwäche von Coco und Alex bestand darin, dass Scanner und Parser durch getrennte Werkzeuge erzeugt wurden und die Schnittstelle zwischen den beiden Teilen manuell abgeglichen werden musste. So lag es nahe, Coco und Alex zusammenzuführen und aus der Compilerbeschreibung sowohl einen Scanner als auch einen Parser zu erzeugen. Dies führte zu *Coco-2* [DoPi90], einer Weiterentwicklung von Coco, die eine attributierte Grammatik mit Scannerbeschreibung zu einem Scanner und einem tabellengesteuerten Topdown-Parser verarbeitete. In Coco-2 konnten die Terminalklassen vom Scanner berechnete Attribute und die Produktionen auch lokale Variablen enthalten, so dass die Notwendigkeit entfiel, Werte über rekursive Produktionen hinweg mit Hilfe eines Kellers zu retten. Außerdem wurden sogenannte "Syntaxaktionen" eingeführt, die es ähnlich den in Abschnitt 5 besprochenen Konfliktlösern gestatteten, LL(1)-Konflikte auf Grund von weiterem Vorgriff oder Zusatzinformationen (etwa aus der Symbolliste) zu beseitigen ([Dobler91]).

An der ETH Zürich wurde ebenfalls an der Erweiterung von Coco gearbeitet, und es entstand die erste Version von *Coco/R* [Möss90]. Wie Coco-2 erzeugte auch Coco/R sowohl einen Scanner als auch einen Parser. Der Unterschied zu Coco-2 lag vor allem darin, dass der erzeugte Parser nach der Methode des rekursiven Abstiegs arbeitete und nicht nach einem tabellengesteuerten Verfahren (eine spätere Version von Coco-2 erzeugte ebenfalls Parser im rekursiven Abstieg).

Inzwischen war die Zeit des Internet angebrochen. Coco/R wurde kostenlos über FTP zur Verfügung gestellt und verbreitete sich rasch. Die ursprüngliche Version, die in Oberon geschrieben war, wurde von *Pat Terry* an der University of Grahamstown nach Pascal übertragen [Terry], von *John Gough* an der Queensland University of Technology nach Modula-2, von *Francisco Arzu* an der Universidad del Valle de Guatemala nach C und C++ und von *Pat Connors* nach Delphi [Conn]. An der Universität Linz entstanden die heute aktuellsten Versionen von Coco/R für Java und C# [CocoR]. Als neueste Weiterentwicklung wurden von den Verfassern dieses Beitrags die in Abschnitt 5 beschriebenen Konfliktlöser in die C#-Version von Coco/R integriert.

Coco/R wird heute weltweit verwendet. Nicht nur Universitäten und Firmen setzen dieses Werkzeug ein, sondern es gibt auch Übersetzerbau-Bücher, die auf Coco/R

beruhen [Terry97]. Allen Personen, die an Coco/R oder einer seiner Varianten gearbeitet haben, insbesondere Prof. Rechenberg, von dem die Idee zu Coco ausging, sei an dieser Stelle herzlich gedankt.

Literatur

[ANTLR] Online-Dokumentation von ANTLR. http://www.antlr.org

[CocoR] Mössenböck, H.: Online-Dokumentation von Coco/R. http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/

[Conn] Coco/R für Delphi: http://parserbuilder.sourceforge.net/

[Dobler91] Dobler H.: Topdown Parsing in Coco-2. ACM SIGPLAN Notices 26 (1991)

[DoPi90] Dobler H., Pirklbauer K.: Coco-2 - A New Compiler-Compiler. ACM SIGPLAN Notices 25 (1990)

[JCC] Online-Dokumentation von JavaCC. http://www.webgain.com/products/java_cc/

[Knuth68] Knuth, D.E.: Semantics of Context-free Languages. Mathematical Systems Theory, vol 2, pp.127-145, 1968

[Möss81] Mössenböck H.: TDG - Ein Generator für Syntaxanalysetabellen. Technischer Bericht 4/81. Universität Linz, Institut für Informatik, Abteilung Software 1981.

[Möss83] Mössenböck H.: Coco – Ein Compiler-Compiler. Diplomarbeit. Universität Linz, Institut für Informatik, 1983

[Möss86] Mössenböck H.: Alex – A Simple and Efficient Scanner Generator. ACM SIGPLAN Notices 21(1986)5: 69-75.

[Möss87] Mössenböck H.: Compilererzeugende Systeme für Mikrocomputer. Dissertationen der Johannes Kepler Universität Linz, Verlag VWGÖ, Wien 1987

[Möss90] Mössenböck H.: A Generator for Production Quality Compilers. 3rd Intl. Workshop on Compiler Compilers (CC'90), Schwerin, LNCS 477, Springer-Verlag 1990

[Rech79] Rechenberg P.: Übersetzung mit attributierten Grammatiken In: Brockhaus, Pomberger, Rechenberg, Schauer: Prinzipien des Übersetzerbaus. Österreichische Computergesellschaft 1979

[Rech83] Rechenberg P.: Ein tabellengesteuerter LL(1)-Syntaxanalysator mit automatischer Fehlerbehandlung. Technischer Bericht 1/83, Universität Linz, Institut für Informatik, Abteilung Software 1983

[Rech84] Rechenberg P.: Attributierte Grammatiken als Methode der Softwaretechnik. Elektronische Rechenanlagen 26(1984)3: 111-119

[ReMö85] Rechenberg P., Mössenböck H.: Ein Compiler-Generator für Mikrocomputer. Hanser-Verlag 1985

[ReMö88a] Rechenberg P., Mössenböck H.: A Compiler Generator for Microcomputers. Prentice-Hall 1988

[ReMö88b] Rechenberg P., Mössenböck H.: A Compiler Generator for Microcomputers (in Japanese). Information & Computing 1988

[Terry] http://www.scifac.ru.ac.za/coco/

[Terry97] Terry, P.: Compilers and Compiler Generators – An Introduction Using C++. International Thomson Computer Press, 1997.

[Wirth77] Wirth, N.: What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? Communications of the ACM, November 1977

[Wirth96] Wirth N.: Grundlagen und Techniken des Compilerbaus. Addison-Wesley 1996