Bericht 1/90

WRG – ein neuer Generator für Top-Down-Parser mit automatischer Fehlerbehandlung

Klaus Lagally

Zusammenfassung

WRG ist ein Parsergenerator, der aus einer EBNF-Beschreibung einer kontextfreien Grammatik einen "recursive-descent"-Parser in (Turbo-)Pascal erzeugt. Eine Syntaxfehlerbehandlung, die notfalls Teile der Eingabe überspringt, wird auf Wunsch automatisch generiert. Die Grammatik kann eine L-Attributierung tragen, die auf das Parameter-Konzept von Pascal abgebildet wird; ebenso sind semantische Aktionen, lokale Hilfsvariablen und Einschübe von Pascal-Deklarationen möglich. Ein "any"-Konstrukt gestattet das bequeme Überlesen von Teilen der Eingabe.

Inhaltsverzeichnis

Kapitel 1

Übersicht

WRG (Werkzeug zum Übersetzerbau durch Rechnergestützte Generierung) ist ein am Institut für Informatik der Universität Stuttgart entwickelter Parser-Generator. Er akzeptiert eine EBNF-Beschreibung einer kontextfreien Sprache, die den LL(1)-Bedingungen genügen sollte, und generiert daraus einen "recursive-descent"-Parser in (Turbo-)Pascal mit automatischer Syntax-Fehlerbehandlung.

Darüberhinaus enthält WRG folgende Erweiterungen:

- Für viele Grammatiken, welche die LL(1)-Bedingungen verletzen, wird dennoch ein "vernünftiger" Parser erzeugt, da die Klasse der durch "recursive descent" analysierbaren Sprachen die Klasse LL(1) echt umfaßt. Die LL(1)-Verletzungen werden erkannt und gemeldet, aber die Code-Erzeugung wird fortgesetzt; der Benutzer muß sich dann selbst durch Inspektion vergewissern, ob der erzeugte Parser das Gewünschte leistet. Näheres dazu unter "Konfliktbehandlung".

- Die Eingabe von WRG besteht aus einer Folge von Regeln. Jede Regel beschreibt eine *Produktionenklasse*, das sind alle Produktionen mit gleicher linker Seite. WRG erzeugt aus jeder Produktionenklasse eine Pascal-Prüfprozedur, welche eine aus der linken Seite ableitbare Folge von Terminalsymbolen erkennt. Bei mehreren möglichen Alternativen wird abhängig vom aktuellen Eingabesymbol der erste passende Zweig ausgewählt.

- Innerhalb einer Regel können, beispielsweise zur Berechnung von Attributen, an beliebiger Stelle semantische Aktionen eingefügt werden; diese werden dann ausgeführt, wenn sie beim Durchlaufen des betreffenden Zweiges erreicht werden. Sie dürfen aus beliebigen syntaktisch korrekten Pascal-Anweisungsfolgen bestehen und werden von WRG nicht überprüft, sondern unverändert übernommen. Für die Ablaufsteuerung des erzeugten Parsers sind sie bedeutungslos.

  Zu beachten ist dabei, daß eine semantische Aktion, mit der eine Alternative

beginnt, nur dann ausgeführt wird, wenn der betreffende Zweig tatsächlich durchlaufen wird, auch wenn die dafür maßgeblichen Grammatiksymbole erst weiter hinten stehen!

- Zur Übergabe von Attributen kann eine Regel eine Liste von formalen Parametern beliebigen Typs besitzen, auf die in semantischen Aktionen zugegriffen werden kann. Diese Liste wird in die erzeugte Prüfprozedur unverändert übernommen und ansonsten nicht überprüft. Ebenso können lokale Hilfsvariable beliebigen Typs deklariert werden, die bei Rekursion automatisch gekellert werden. Darüberhinaus können semantische Aktionen auch auf globale Größen zugreifen.

- Anstelle einer Regel kann auch ein Semantik-Einschub stehen, der aus beliebigen syntaktisch korrekten Pascal-Deklarationen bestehen darf. Er wird von WRG nicht überprüft, sondern unverändert übernommen. Die so deklarierten Objekte (Konstanten, Typen, Variablen, Prozeduren und Funktionen) sind global sichtbar.

- Das Wortsymbol "eps" bezeichnet das leere Wort. Es dient nur der leichteren Lesbarkeit und kann genausogut weggelassen werden.

  Eine Alternative, die bis auf semantische Aktionen (allenfalls) aus "eps" besteht, sollte i.a. der letzte Zweig einer Produktionenklasse sein; dies verbessert die Effizienz des Parsers und erleichtert bei Grammatiken, die nicht in der Klasse LL(1) liegen, in der Regel die Konfliktbehandlung.

- Das Wortsymbol "any" bezeichnet ein beliebiges einzelnes Terminalsymbol. "any" kann mit Vorteil verwendet werden, um bei einer partiellen Analyse Teile der Eingabe zu überlesen, die im Augenblick nicht interessieren, und wird nur dann angesprochen, wenn es zum aktuellen Eingabesymbol keine andere passende Zuordnung gibt.

Kapitel 2

Funktionsweise

2.1 Aufbau

WRG besteht aus 4 Pässen:

- Pass 1 liest die Eingabe ein, prüft sie auf syntaktische Korrektheit und schreibt sie auf eine Zwischendatei. Für jede Produktionenklasse wird eine "forward"-Deklaration abgesetzt. Außerdem wird ein Geflecht erzeugt, welches die Ablaufstruktur des zu erzeugenden Parsers nachbildet.
- Pass 2 analysiert das Geflecht, berechnet die Anfangs- und Folge-Mengen der vorkommenden Nichtterminalzeichen, überprüft die LL(1)-Eigenschaft und meldet gefundene Konflikte. Außerdem werden Hilfsmengen zur Ablaufsteuerung und zur Fehlerbehandlung berechnet. Wahlweise werden die berechneten Strukturen lesbar ausgegeben.
- Pass 3 liest die Zwischendatei wieder ein und erzeugt aus ihr und den in Pass 2 berechneten Informationen die Parser-Routinen. Dabei werden auch Parameter, lokale Deklarationen und Semantik-Einschübe an den passenden Stellen eingefügt.
- Pass 4 erzeugt im Fehlerfall stets und ansonsten auf Wunsch ein Protokoll der Eingabe, evtl. mit eingestreuten Fehlermeldungen. Wahlweise kann als Testhilfe die interne Darstellung der Ablaufstruktur mit den Steuermengen in lesbarer Form ausgegeben werden.
- Pass 2 und 3 werden bei syntaktischen Fehlern nicht aktiviert.

2.2 Konfliktbehandlung

Bei erkannten LL(1)-Konflikten geht der erzeugte Parser folgendermaßen vor:

- First-First-Verletzung:

  Zwei Alternativen können mit demselben Terminalsymbol beginnen: die in der Aufschreibung frühere Alternative wird ausgewählt.

- First-Follow-Verletzung:

  Es gibt einen Zweig, aus dem das leere Wort ableitbar ist, und ein Terminalzeichen kann sowohl Folgesymbol wie Anfangssymbol desselben Zweiges sein: das Zeichen wird als Anfangssymbol interpretiert.

2.3 Bedeutung von "any"

- Eine mit "any" beginnende Alternative wird nur dann ausgewählt, falls kein anderer Zweig auf das aktuelle Eingabezeichen paßt.
- "any" als einzige Alternative wird demnach immer gewählt.
- "any" innerhalb einer Option oder einer Wiederholung wird nur dann ausgewählt, wenn es keinen zur Eingabe passenden Parallelzweig gibt und wenn das anstehende Eingabezeichen kein legales Folgesymbol ist.
- Beispiel: das Konstrukt `"(" { any } ")"` überliest nach `"("` alle Zeichen bis vor `")"`.

Kapitel 3

Praktische Erfahrungen

3.1 Stand der Realisierung

WRG ist zum Einsatz auf MS-DOS-Rechnern (IBM-PC und Kompatible) gedacht und benötigt für einen vernünftigen Einsatz einen Hauptspeicher-Ausbau von mindestens 512 kB.

Die vorliegenden Versionen (ab 1.1) akzeptieren den vollen oben beschriebenen Sprachumfang. Pass 1 und Pass 3 sind durch Bootstrap aus je einer EBNF-Grammatik für die Eingabesprache mittels einer Vorversion des Systems erzeugt worden und können daher später leicht modifiziert werden; die restlichen Teile sind von Hand in Turbo-Pascal (Version 4) geschrieben. Die erreichte Effizienz ist für den ersten praktischen Einsatz ausreichend, aber wohl verbesserungsfähig.

Als Erweiterung kann ein Parser ohne Fehlerbehandlung erzeugt werden. Ein solcher Parser hat erheblich geringeren Umfang, läuft wesentlich schneller und ist interessant für spätere Pässe eines Compilers, welche voraussetzen können, daß ihre Eingabe formal korrekt ist.

3.2 Bewertung

Der erzeugte Parser ist durch den Aufwand, den die Prozeduraufrufe mit sich bringen, notwendigerweise weniger effizient als ein expliziter Kellerautomat. Der Zusatzaufwand läßt sich durch konsequente Ausnutzung der Möglichkeiten von EBNF in vernünftigen Grenzen halten.

Die automatische Fehlerbehandlung ist nicht billig zu haben, zumal sie sich stark auf das Mengen-Konstrukt in Pascal abstützt, dessen Realisierung in Turbo-Pascal recht

aufwendig ist. Die derzeitige Implementierung ist robust, aber nicht optimal. Hier sind Verbesserungen möglich und geplant.

Von einer Nachbearbeitung des generierten Parser-Codes muß, soweit sie die Ablaufsteuerung und die Fehlerbehandlung betrifft, wegen derer enger Verzahnung in der Regel abgeraten werden. Dies gilt nicht für Semantik- Einschübe, die aber genausogut bereits in der Eingabe verbessert werden können.

Kapitel 4

Quellen

WRG baut auf einer Reihe von Vorläufern auf; viele Ideen wurden explizit oder sinngemäß übernommen aus:

- G.Goos: Programm SMG (Syntax Machine Generator), Interner Bericht TU München, ca. 1966
- A.C.Hartmann: A Concurrent Pascal Compiler for Minicomputers, Lecture Notes in Computer Science 50, Springer Verlag 1977
- P.Rechenberg, H.Mössenböck: Ein Compiler-Generator für Mikrocomputer, Carl Hanser Verlag 1985
- N.Wirth: Compilerbau, Teubner Verlag 1977

Eine Reihe von weiteren Ideen, deren genaue Quelle sich nicht mehr feststellen läßt, stammen aus der allgemeinen Folklore; auf YACC und ähnliche Produkte, die nach dem Bottom-Up-Prinzip arbeiten, haben wir nicht bewußt zurückgegriffen.

Wir haben uns bemüht, die Schwächen der uns bekannten Ansätze (die hier nicht weiter ausgeführt werden sollen) nach Möglichkeit zu vermeiden; dafür sind mit Sicherheit neue Fehler hineingekommen, für deren Mitteilung wir sehr dankbar sind.

Anhang A

Benutzungsanleitung

A.1 Aufruf des Generators

Der Parsergenerator wird folgendermaßen aufgerufen:

```
WRG <name>.<ext> <options>
```

Die Namenserweiterung `.<ext>` ist beliebig und darf auch fehlen. Fehlende Angaben `<name>.<ext>` und `<options>` werden interaktiv erfragt.

WRG erzeugt eine Textdatei `<name>.INC`, welche die generierten Parser-Routinen enthält. Im Fehlerfall und auf Wunsch wird ein Protokoll in die Datei `<name>.LST` geschrieben.

Die Dateinamen `<name>.IMC`, `<name>.ERR`, `<name>.TMP`, `<name>.GRA` werden von WRG intern verwendet und sind reserviert; die zugehörigen Dateien sind nach einem Lauf von WRG normalerweise gelöscht.

Die Optionenangabe `<options>` ist eine Zeichenfolge ohne Zwischenraum, deren Einzelzeichen in beliebiger Reihenfolge stehen dürfen und mögliche Optionen bezeichnen. Bislang sind festgelegt:

l	Protokollierung auch bei fehlerfreier Ausführung
q	Erzeugung einer schnellen Parser-Version ohne Fehlerbehandlung
s	schnellste Parser-Version, die korrekte Eingabe voraussetzt
g	kurze lesbare Darstellung des Geflechts ins Protokoll
d	ausführliche Darstellung des Geflechts nach `<name>.gra`
i	die Zwischendatei `<name>.IMC` bleibt erhalten
e	die Fehlerdatei `<name>.ERR` bleibt erhalten
-	keine zusätzlichen Optionen

Alle Options-Angaben außer “l”, “q”, “s” und “-” sind nur als Testhilfen des Generators gedacht und erzeugen sehr umfangreiche Informationen, deren Format hier nicht beschrieben wird.

Bei katastrophalen internen Fehlern des Generators bleiben alle internen Dateien erhalten; der Benutzer wird dann aufgefordert, sie sicherzustellen, um die Fehlersuche zu erleichtern.

Bei Betriebsmittel-Engpässen können Pascal-Laufzeitfehler auftreten, insbesondere:

101 Überlauf Platte
202 Überlauf Halde: zuwenig Hauptspeicher
203 Überlauf Stapel: zuwenig Hauptspeicher

Hier kann nur der Benutzer allenfalls Abhilfe schaffen.

A.2 Eingabeformat

Die Eingabe des Generators ist eine kontextfreie Grammatik, die in einer modifizierten erweiterten Backus-Naur-Form (EBNF) geschrieben ist. Ihre genaue Syntax wird im folgenden ebenfalls in EBNF beschrieben, dazu kommen verbale Erläuterungen; dabei bedeuten:

```
"X"                 X muss woertlich so dastehen
<X>                 verbale Beschreibung von X
(X)   Gruppierung   Zusammenfassung zusammengehoeriger Teile
[X]   Option        X kann vorhanden sein oder fehlen
{X}   Wiederholung  X kann beliebig oft auftreten oder fehlen
X|Y   Auswahl       genau eines von X und Y muss vorhanden sein
```

Jede Grammatikregel hat das Format:

```
<linke Seite> ":" <rechte Seite> "."
```

Terminalsymbole und Nichtterminalsymbole werden durch vom Benutzer frei gewählte Pascal-Bezeichner benannt. Die Terminalsymbole müssen außerhalb des erzeugten Parsers deklariert sein und können benannte Konstanten oder Elemente eines Aufzählungstyps sein. Aus den Nichtterminalbezeichnern werden die Namen der erzeugten Parser-Routinen abgeleitet.

Folgende Bezeichner sind reserviert:

```
grammar, eps, any, loc, endloc, sem, endsem
```

Diese Bezeichner müssen mit Kleinbuchstaben geschrieben werden.

Kommentare beginnen mit einem Prozent-Zeichen (%) und gehen bis zum Zeilenende. Leerzeichen, Zeilenenden und Kommentare trennen Bezeichner und sind ansonsten bedeutungslos, außer in Semantik-Einschüben, die bis auf Kommentare unverändert übernommen werden.

Die Eingabe muß folgender Syntax genügen:

```
ebnf : head { item } .

head : "grammar" name "(" axiom "," endsym ")" .

name : ident .

axiom : ident .

endsym : ident .

ident : <Bezeichner nach Pascal-Syntax, Unterstrich ist
         ausser als erstes Zeichen zugelassen> .

item : <folgt weiter unten> .
```

- Eine Grammatik besteht aus einem Kopf, der globale Informationen enthält, und einer Folge von Grammatikregeln und Einschüben (Items).
- "name" bedeutet den Grammatik-Namen, mit dem alle intern erzeugten Bezeichner "bez" in der Gestalt "name_bez" eindeutig gemacht werden. Das Präfix "name_" sollte also sonst vom Benutzer nicht verwendet werden.
- "axiom" bedeutet das Startsymbol der Grammatik.
- "endsym" bedeutet das letzte Zeichen, das ein lexikalischer Analysator (Scanner) liefern muß, wenn die Eingabe abgearbeitet ist. Es ist vom Benutzer frei wählbar.

```
item : prod | insert .

prod : lhs ":" rhs "." .

lhs : ident [ formals ] .

rhs : [ locals ] regex .
```

```
formals : "(" <Zeichenfolge, die ")" nicht enthaelt> ")" .

locals : "loc"
         <Zeichenfolge, die "endloc" nicht enthaelt>
         "endloc" .

regex : <folgt weiter unten> .

insert : "sem"
         <Zeichenfolge, die "endsem" nicht enthaelt>
         "endsem" .
```

- Zu jedem Nichtterminalsymbol der Grammatik gehört eine Produktionenklasse, die in eine Pascal-Prüfroutine übersetzt wird.
- Nichtterminalsymbole können mit Attributen versehen werden, die durch Parameter der zugehörigen Prüfroutinen realisiert werden. Die Zeichenfolgen zwischen "(" und ")" in "formals" müssen jeweils eine syntaktisch korrekte formale Pascal-Parameterliste sein. Dies wird nicht geprüft.
- Die Zeichenfolgen zwischen "loc" und "endloc" in "locals" müssen jeweils aus einer syntaktisch korrekten abgeschlossenen Folge von Pascal-Vereinbarungen bestehen, die lokal zur erzeugten Prüfroutine gelten. Dies wird nicht geprüft.
- Die Zeichenfolgen zwischen "sem" und "endsem" in "insert" müssen jeweils aus einer syntaktisch korrekten abgeschlossenen Folge von Pascal-Vereinbarungen bestehen, die globale Gültigkeit haben. Dies wird nicht geprüft.

```
regex : sequence { "|" sequence } .

sequence : { element [ "," ] } .

element : terminal | nonterminal | "eps" | "any"
          | group | option | loop | semant .

terminal : "=" ident .

nonterminal : ident [ actuals ] .

actuals : "(" <Zeichenfolge, die ")" nicht enthaelt> ")" .

group : "(" regex ")" .
```

```
option : "[" regex "]" .

loop : "{" regex "}" .

semant : "sem"
         <Zeichenfolge, die "endsem" nicht enthaelt>
         "endsem" .
```

- Von den Alternativen in "regex" wird die erste passende in der Reihenfolge der Aufschreibung ausgewählt, wenn sie nicht ohnehin eindeutig festliegt.
- Die Elemente einer "sequence" werden der Reihe nach geprüft.
- Terminalsymbole werden durch vorgestelltes "=" gekennzeichnet.
- Nichtterminalsymbole können Attribute tragen, die als Parameter der zugehörigen Prüfroutinen angegeben werden. Die Zeichenfolgen zwischen "(" und ")" in "actuals" müssen jeweils eine syntaktisch korrekte aktuelle Pascal-Parameterliste sein, die zur entsprechenden formalen Parameterliste paßt. Dies wird nicht geprüft.
- "eps" bezeichnet das leere Wort und kann auch weggelassen werden.
- "any" bezeichnet ein beliebiges Terminalzeichen, das auf keine andere Alternative paßt.
- Die Bedeutung von "group", "option" und "loop" ist oben bereits beschrieben.
- Die Zeichenfolgen zwischen "sem" und "endsem" in "semant" müssen jeweils aus einer syntaktisch korrekten Folge von Pascal-Anweisungen bestehen. Dies wird nicht geprüft.

A.3 Schnittstelle des erzeugten Parsers

Ein von WRG aus einer Grammatik mit dem Kopf

```
grammar X (Y, Z)
```

erzeugter Parser setzt folgende Umgebung voraus:

- ein lexikalischer Analysator muß vorhanden sein. Er faßt die Zeichen der Eingabe zu Terminal-Symbolen des Typs `X_token` zusammen, die bei Bedarf mittels der Prozedur `X_nextch` abgerufen werden; in der Regel wird dabei noch lexikalische Zusatzinformation in globale Variablen abgelegt, auf die in Semantik-Einschüben zugegriffen werden kann.
- folgende Pascal-Objekte werden verwendet:

```
type X_token = <Aufzaehlung oder Unterbereich>;
        (* evtl. Umbenennung eines vorhandenen Typs *)

const Z: X_token;
        (* Eingabe-Ende-Symbol, benannte Konstante *)

var X_ch: X_token;
        (* aktuelles Eingabesymbol *)

procedure X_nextch;
        (* besetzt X_ch und evtl.
           globale Zusatzinformation neu *)
```

- Im Fehlerfall wird außerdem aufgerufen:

```
procedure X_error(s: string);
        (* gibt den Fehlertext "s" aus *)

function X_conv(t: X_token): string;
        (* liefert eine geeignete externe Darstellung
           fuer das Terminalsymbol "t" aus *)
```

- Alle in der Grammatik verwendeten Terminal-Bezeichner müssen als benannte Konstanten vom Typ `X_token` erklärt sein. Daneben kann in Semantik-Einschüben beliebig auf globale Größen zugegriffen werden.

Der erzeugte Parser enthält eine Prozedur mit dem Kopf:

```
procedure X_;
```

Bei ihrem Aufruf geschieht das folgende:

- `X_nextch` wird aufgerufen und liest das erste Eingabezeichen;
- die dem Startsymbol Y entsprechende Prüfprozedur wird aktiviert;
- nach deren Ende wird geprüft, ob das Ende-Symbol Z gefunden wurde. Andernfalls wird eine Fehlermeldung abgesetzt.

Anhang B

Anwendungsbeispiel

```
% Beispiel fuer die Verwendung von WRG:
% Auswertung arithmetischer 'real'-Ausdruecke

% akzeptiert eine Folge von Ausdruecken
% gibt die Ausdruecke und deren Werte aus
% Annahme: der Scanner besetzt 'ch' und 'k' global

grammar Arith (S, eof)

S :
  loc var wert: real; endloc
  { E(wert)      sem writeln(' = ', wert:10:5) endsem } .

E(var wert: real) :
  loc var wert1: real; op: token; endloc
        sem op := plus endsem
  [ addop(op) ]
    T(wert)
        sem if op = minus then wert := - wert endsem
  { addop(op)
    T(wert1)
        sem if op = plus then wert := wert + wert1
                         else wert := wert - wert1 endsem } .
```

```
addop(var op: token) :
% Annahme: der Scanner liefert fuer '+' und '-'
% deren ASCII-Verschluesselung nach 'ch'
        sem op := ch; write(' ', chr(op), ' ') endsem
  ( =plus
  | =minus ) .

T(var wert: real) :
% Annahme: bei Division durch 0 wird mit dem Zaehler weitergerechnet,
% eine Fehlermeldung geht ins Eingabeprotokoll
  loc var wert1: real; op: token; endloc
    F(wert)
  { mulop(op)
    F(wert1)
        sem if op = mal then wert := wert * wert1
            else if wert1 = 0 then error('Division durch Null')
            else wert := wert / wert1 endsem } .

mulop(var op: token) :
% Annahme: der Scanner liefert fuer '*' und '/'
% deren ASCII-Verschluesselung nach 'ch'
        sem op := ch; write(' ', chr(op), ' ') endsem
  ( =mal
  | =strich ) .

F(var wert: real) :
    =kla   sem write('(') endsem
    E(wert)
    =klz   sem write(')') endsem
  | CON(wert) .

CON(var wert: real) :
% Annahme: bei 'intcon' und 'realcon' wird 'k' vom Scanner mit einem
% Schluessel fuer die externe Darstellung der Konstanten besetzt
% 'getstring' liefert diese Darstellung als String aus
% 'val' konvertiert den String in die 'real'-Darstellung
% 'y' und 'z' sind hier bedeutungslos
  loc var s: string; y: integer; z: word; endloc
        sem getstring(k, s, y); write(s); val(s, wert, z) endsem
  ( =intcon
  | =realcon ) .
```