Automatische Generierung effizienter Compiler

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Automatische Generierung effizienter Compiler

Josef Grosch

Jan. 12, 1989

Report No. 13

Copyright © 1989 GMD

Gesellschaft für Mathematik und Datenverarbeitung mbH
Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1
D-7500 Karlsruhe

Automatische Generierung effizienter Compiler

Josef Grosch
GMD Forschungsstelle für Programmstrukturen an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe 1

Zusammenfassung

Das Projekt UWE (Übersetzerbau-Werkzeuge) zielt auf die Erstellung von Generatorprogrammen, die es erlauben Compiler weitgehend automatisch zu erzeugen. Für die Analysephase von Compilern werden drei Generatoren vorgestellt: *Rex* ist ein Scanner-Generator auf der Basis regulärer Ausdrücke. Der Parser-Generator *Lalr* erzeugt aus LALR(1)-Grammatiken tabellengesteuerte Parser mit S-Attributierung und automatischer Fehlerbehandlung. Der Parser-Generator *Ell* erzeugt aus LL(1)-Grammatiken Parser nach dem Verfahren des rekursiven Abstiegs mit einem Mechanismus zur L-Attributierung und ebenfalls automatischer Fehlerbehandlung. Alle Generatoren sind in Modula-2 programmiert und erzeugen Programme in C oder Modula-2. Die herausragende Eigenschaft der erzeugten Programme ist ihre Geschwindigkeit. Auf einem MC 68020 Prozessor erreichen die Scanner bis zu 200.000 und die Parser um die 400.000 Zeilen pro Minute. Diese Geschwindigkeiten übertreffen die UNIX Generatoren *Lex* und *Yacc* um die Faktoren 4 bzw. 2 bis 3.

Abstract

The project UWE (compiler construction tools) has the goal to implement a complete set of tools for the largely automatic construction of compilers. Three tools for the analysis phase of compilers are presented: *Rex* is a scanner generator based on regular expressions. The parser generator *Lalr* generates table-driven parsers for LALR(1) grammars that include a mechanism for S-attribution and automatic error reporting, recovery, and repair. The parser generator *Ell* generates recursive descent parsers for LL(1) grammars with a mechanism for L-attribution and an automatic error recovery similar to *Lalr.* All the tools are implemented in Modula-2 and generate programs in C or Modula-2. The outstanding property of the generated programs is their speed. On a MC 68020 processor the scanners reach up to 200,000 and the parsers around 400,000 lines per minute. These speeds represent in comparison to the UNIX tools *Lex* and *Yacc* a clear improvement by factors of 4 respectively 2 to 3.

Projekt-Überblick

Das Projekt UWE (für Übersetzerbau-Werkzeuge) befaßt sich mit der Technik des Übersetzerbaus (Compiler-Baus), der Systematisierung der dabei verwendeten Methoden und Verfahren sowie deren Umsetzung in Generatorprogramme. Dies schließt die Entwicklung von Spezifikationstechniken für die verschiedenen Compiler-Teile ein. Das Ziel ist die Erstellung eines vollständigen Satzes von Werkzeugen oder Generatorprogrammen zur weitgehend automatischen Konstruktion von Compilern. Dabei wird gegenüber einer Compiler-Entwicklung von Hand angestrebt, den Konstruktionsaufwand deutlich zu reduzieren, die Fehlerfreiheit und Zuverlässigkeit zu steigern und eine vergleichbare Effizienz bzw. Übersetzungsgeschwindigkeit zu erreichen.

Die Beschäftigung mit Übersetzerbau-Werkzeugen geht auf den Lehrstuhl für Übersetzerbau an der Universität Karlsruhe zurück, aus dem heraus die GMD Forschungsstelle Karlsruhe entstanden ist. Dort wurden bereits das Parser Generating System PGS (LALR(1)-Zerteiler-Generator), der Generator für Attribut-Grammatiken GAG (semantische

Analyse) und das Code Generator Synthese System CGSS (erzeugt Code-Generatoren aus Maschinenbeschreibungen) entwickelt. Diese Werkzeuge wurden an der GMD Forschungsstelle Karlsruhe weiterentwickelt und inzwischen weltweit an über 100 Institutionen weitergegeben. Diese quasi erste Generation von Werkzeugen erfüllte bereits die Forderung nach Steigerung der Zuverlässigkeit der erzeugten Compiler-Teile. Das kann man am erfolgreichen Einsatz für realistische Anwendungen sehen, wie etwa dem Karlsruher Ada Compiler oder der Norm für PEARL. Die im folgenden vorgestellten Werkzeuge der zweiten Generation schließen die Lücke im Bereich der lexikalischen Analyse und erzeugen für die Syntaxanalyse komfortable und effiziente Zerteiler.

Der Scanner-Generator Rex

Der Scanner-Generator *Rex* wurde mit dem Ziel entwickelt, die mächtige Spezifikationsmethode der regulären Ausdrücke mit der Generierung möglichst effizienter Scanner zu kombinieren. Der Name *Rex* steht für *regular expression tool* und spiegelt die Spezifikationsmethode wider. Eine Scanner-Spezifikation besteht im Prinzip aus einer Menge regulärer Ausdrücke wobei jedem eine semantische Aktion zugeordnet ist. Diese Aktionen sind beliebige Anweisungen, die in einer der Zielsprachen C oder Modula-2 programmiert sind. Immer wenn eine Zeichenkette, die durch einen regulären Ausdruck beschrieben wird, in der Eingabe das Scanners erkannt wird, werden die Anweisungen der zugehörigen Aktion ausgeführt. Um Zeichenketten auch in Abhängigkeit vom Kontext erkennen zu können stehen sogenannte Startzustände zur Behandlung von linkem Kontext zur Verfügung, und der rechte Kontext kann durch einen zusätzlichen regulären Ausdruck spezifiziert werden. Sollten mehrere reguläre Ausdrücke zu einer Eingabe passen, so wird derjenige bevorzugt, der die längste Zeichenfolge erkennt. Falls immer noch mehrere Möglichkeiten bestehen, wird davon der erste reguläre Ausdruck in der Spezifikation gewählt.

Mit *Rex* erzeugte Scanner stellen automatisch für jede erkannte Zeichenkette die Zeilen- und Spalten-Position zur Verfügung. Für Sprachen wie Pascal und Ada, wo Groß- und Kleinbuchstaben nicht unterschieden werden, gibt es eine Normalisierung auf Groß- oder Kleinbuchstaben. Vordefinierte Regeln gestatten das Überlesen unbedeutender Zeichen wie Leerzeichen, Tabulatoren oder Zeilenwechsel.

Die erzeugten Scanner sind tabellengesteuerte deterministische endliche Automaten. Da die Tabellen dünn besetzte Matrizen sind, werden sie mit der sogenannten "Kammvektor-Technik" komprimiert. Diese kombiniert große Speicherreduktion mit schnellem Tabellenzugriff. Der Generator *Rex* ist in Modula-2 programmiert und kann zur Zeit Scanner in den Sprachen C und Modula-2 erzeugen. *Rex* läuft inzwischen auf den Rechnern SUN/UNIX, PCS Cadmus/UNIX und VAX/BSD UNIX bzw. ULTRIX.

Die herausragende Eigenschaft von *Rex* ist die Geschwindigkeit. Die erzeugten Scanner verarbeiten bis zu 200.000 ohne und 150.000 Zeilen pro Minute mit Anwendung eines Hash-Verfahrens für Bezeichner. Dies ist die vierfache Geschwindigkeit gegenüber Scannern, die mit dem UNIX-Werkzeug *Lex* [Les75] erzeugt wurden. In typischen Fällen haben die generierten Scanner nur ein Viertel der Größe wie bei *Lex* (z. B. 11 KB für Modula-2). Gewöhnlich benötigt *Rex* nur 1/10 der Zeit von *Lex* um einen Scanner zu erzeugen. Die genannten Zahlen wurden auf einem MC 68020 Prozessor gemessen.

Scanner-Spezifikation mit Rechts-Kontext

Die folgenden kleinen Beispiele zeigen wie Spezifikationen für *Rex* [Gro87] aussehen. Abbildung 1 spezifiziert den Aufbau ganzer und reeller Zahlen für die Sprache Modula-2. Die erste Zeile besagt, daß eine ganze Zahl aus Folgen von Zeichen aus der Menge 0 bis 9 besteht. In Modula-2 gibt es, wie in manchen anderen Sprachen auch, eine Stelle, an der die Strategie der

längsten Zeichenkette fehlschlägt. So sollte etwa die Eingabe `1..` in die Zeichenketten "`1`" und "`..`", nicht aber in "`1.`" und "`.`" zerlegt werden, was alles legale Modula-2-Symbole wären.

```
{0-9} +              : { RETURN SymDecimal; }
{0-9} + / ".."       : { RETURN SymDecimal; }
{0-9} + "." {0-9} * (E {+\-} ? {0-9} +) ?
                     : { RETURN SymReal   ; }
".."                 : { RETURN SymRange  ; }
"."                  : { RETURN SymDot    ; }
```

Abbildung 1: Scanner-Spezifikation mit Rechts-Kontext

Das Problem kann mit einem zusätzlichen regulären Ausdruck gelöst werden. Dieser beschreibt den rechten Kontext eines Symbols welches zu einer Ausnahme der Strategie der längsten Zeichenkette führt. In der zweiten Zeile trennt das Zeichen '/' zwei reguläre Ausdrücke und spezifiziert so, daß eine Ziffernfolge erkannt werden soll, wenn zwei Punkte folgen. Die restlichen Zeilen in Abbildung 1 beschreiben die weiteren Symbole, die an diesem Problem beteiligt sind.

Scanner-Spezifikation mit Startzuständen

Manche Probleme lassen sich mit regulären Ausdrücken allein nicht lösen. So erfordert etwa die Erkennung geschachtelter Kommentare in Modula-2 einen zusätzlichen Zähler und den Einsatz von Startzuständen (Abbildung 2). Der Zähler wird im Abschnitt GLOBAL deklariert und im Abschnitt BEGIN initialisiert. Der Abschnitt START vereinbart einen Startzustand namens *Comment.* Regeln, vor denen ein Startzustand steht, sind nur wirksam, wenn sich der Scanner in diesem Startzustand befindet. Die erste Regel erkennt öffnende Kommentarklammern, erhöht den Schachtelungszähler und schaltet in den Startzustand Comment um. In letzterem sind alle 3 Regeln wirksam. Entweder werden Kommentarzeichen überlesen oder bei schließenden Kommentarklammern der Schachtelungszähler erniedrigt. Erreicht dieser den Wert Null, so ist der Kommentar zu Ende, und es wird wieder in den vordefinierten Startzustand STD zurückgekehrt. Im Abschnitt EOF stehen Aktionen, die beim Erreichen des Eingabeendes auszuführen sind. Hier wird im Falle einer fehlenden Kommentarklammer ein entsprechender Fehler gemeldet.

Vergleich von Scanner-Generatoren

Die Tabelle 1 vergleicht *Rex* mit dem UNIX Scanner-Generator *Lex* und dem Lex-Nachbau *Flex* (für fast Lex). Die Tabelle vergleicht die Spezifikationstechniken, die Werkzeuge und die generierten Scanner. Die spezifikations-abhängigen Zahlen wie Generierungszeit und Scanner-

```
GLOBAL   {VAR NestingLevel: CARDINAL;}
BEGIN    {NestingLevel := 0;}
EOF      {IF yyStartState = Comment THEN Error ("unclosed comment"); END;}
DEFINE   CmtCh    = - {*(0.
START    Comment
RULES
            "(*" : {INC (NestingLevel); yyStart (Comment);}
#Comment#   "*)" : {DEC (NestingLevel);
                     IF NestingLevel = 0 THEN yyStart (STD); END;}
#Comment#   "(" | "*" | CmtCh + : {}
```

Abbildung 2: Scanner Spezifikation für geschachtelte Kommentare

Größe gelten für einen Modula-2-Scanner. Die Zahlen wurden auf einem MC 68020 Prozessor gemessen.

Der Parser-Generator Lalr

Wie *Rex* wurde der Parser-Generator *Lalr* mit dem Ziel entwickelt eine mächtige Spezifikationstechnik für kontextfreie Grammatiken mit der Erzeugung effizienter Parser (Zerteiler) zu kombinieren. Der Name *Lalr* nimmt Bezug auf die Klasse der LALR(1)-Grammatiken, für die Zerteiler erzeugt werden können. Diese Grammatiken können in erweiterter BNF geschrieben sein. Jede Grammatikregel kann mit semantischen Aktionen versehen werden, welche wiederum aus beliebigen Anweisungen der Zielsprache bestehen. Wenn der erzeugte Zerteiler eine Grammatikregel erkannt hat, werden die zugeordneten Aktionen ausgeführt. Es steht ein Mechanismus zur S-Attributierung zur Verfügung, d. h. abgeleitete Attribute können während der Zerteilung berechnet werden.

Gehört eine Grammatik nicht zur LALR(1)-Klasse, so stellt der Generator LR-Konflikte fest. Das Problem dabei besteht darin, für einen Konflikt aussagekräftige Information zu liefern, um das Problem in der Grammatik finden zu können und dann für Abhilfe zu sorgen. Während andere Generatoren meist nur den Konflikt in Begriffen von internen Zuständen und Situationen melden gibt *Lalr* Ableitungsbäume aus, die die Konfliktbehebung wesentlich erleichtern. Syntaxfehler werden von den generierten Zerteilern vollautomatisch behandelt. Dies schließt

Tabelle 1: Vergleich einiger Scanner-Generatoren

	Lex	Flex	Rex
Spezifikationsmethode	reguläre Ausdrücke	reguläre Ausdrücke	reguläre Ausdrücke
semantische Aktionen	ja	ja	ja
Rechts-Kontext	ja	ja	ja
Links-Kontext (Startzustände)	ja	ja	ja
Konfliktlösung	längste Folge erste Regel	längste Folge erste Regel	längste Folge erste Regel
Quellposition	Zeile	-	Zeile + Spalte
Normalisierung	-	ja	ja
vordefinierte Regeln zum Überlesen	-	-	ja
mehrere Lösungen (REJECT)	ja	ja	-
Anpassung interner Datenstrukturen	von Hand	automatisch	automatisch
Scanneralgorithmus	tabellengesteuert	tabellengesteuert	tabellengesteuert
Tabellenkompression	Kammvektor	Kammvektor	Kammvektor
Implementierungssprache	C	C	Modula-2
Zielsprachen	C	C	C, Modula-2
Geschwindigkeit [Zeilen/Min.]			
ohne Hashing	36.400	139.000	182.700
mit Hashing	34.700	118.000	141.400
Tabellengröße [Bytes]	39.200	57.300	4.400
Scanner-Größe [Bytes]	43.800	64.100	11.200
Generierungszeit [Sek.]	73,7	7,2	4,9

Fehlermeldung, Wiederaufsetzen der Analyse und Fehlerreparatur ein. (Die genannten Eigenschaften werden unten näher ausgeführt.)

Die erzeugten Parser sind tabellengesteuert, wobei die Tabellen wie im Fall von *Rex* mittels Kammvektor-Technik komprimiert werden. Der Generator *Lalr* ist in Modula-2 programmiert und kann zur Zeit Zerteiler in C und Modula-2 erzeugen. Wie *Rex* läuft *Lalr* inzwischen auf den Rechnern SUN/UNIX, PCS Cadmus/UNIX und VAX/BSD UNIX bzw. ULTRIX.

Mit *Lalr* erzeugte Zerteiler sind 2 bis 3 Mal schneller als mit dem UNIX Werkzeug *Yacc* [Joh75] generierte. Sie erreichen eine Geschwindigkeit von bis zu 400.000 Zeilen pro Minute auf einem MC 68020 Prozessor, wobei die Zeit für den Scanner nicht berücksichtigt ist. Von der Größe her sind die Zerteiler im Vergleich zu *Yacc* etwas größer (z. B. 37 KB für Ada). Im folgenden werden einige Eigenschaften von *Lalr* näher vorgestellt.

S-Attributierung

Die Spezifikation eines Zerteilers [GrV88] folgt dem Stil einer *Rex* Spezifikation. Abbildung 3 zeigt ein einfaches Beispiel für einen Tischrechner, welcher arithmetische Ausdrücke akzeptiert, die aus den Operatoren + und * sowie Klammern und Zahlen aufgebaut sind. Die in den geschweiften Klammern stehenden semantischen Aktionen sorgen für die Berechnung der Ausdrücke. Dazu wird dem Nichtterminal *expr* und dem Terminal *number* das Attribut *value* zugeordnet. Mit $i wird in den semantischen Aktionen auf die Attribute zugegriffen. Dabei bedeuten $i das i-te Grammatiksymbol der rechten Seite und $$ das Nichtterminal der linken Seite einer Regel.

```
expr : expr '+' expr { $$.value := $1.value + $3.value; } .
expr : expr '*' expr { $$.value := $1.value * $3.value; } .
expr : '(' expr ')'  { $$.value := $2.value; } .
expr : number        { $$.value := $1.value; } .
```

Abbildung 3: Grammatikregeln mit S-Attributierung

Mehrdeutige Grammatiken

Die Grammatik in Abbildung 3 und die Regeln in Abbildung 4 sind typische Beispiele für mehrdeutige Grammatiken. Wie bei *Yacc* lassen sich daraus resultierende LR-Konflikte durch die Angabe von Priorität und Assoziativität für Terminale (Operatoren) lösen. Abbildung 5 zeigt ein Beispiel. Die Zeilen stellen zunehmende Prioritäten dar. LEFT, RIGHT und NONE spezifizieren links-assoziative, rechts-assoziative bzw. Operatoren ohne Assoziativität.

```
stmt : 'IF' expr 'THEN' stmt               PREC LOW
     | 'IF' expr 'THEN' stmt 'ELSE' stmt   PREC HIGH .
```

Abbildung 4: Mehrdeutige Grammatikregeln (Dangling Else Problem von Pascal)

```
OPER  LEFT '+'
      LEFT '*'
      NONE LOW
      NONE HIGH
```

Abbildung 5: Lösen von LR-Konflikten mit Priorität und Assoziativität

Beschreibung von LR-Konflikten

Zur leichteren Lokalisierung des Grundes für LR-Konflikte wurde eine von DeRemer und Pennello [DeP82] vorgeschlagene Methode aufgegriffen. Neben der Art des Konflikts und den beteiligten Situationen wie bei anderen LR-Generatoren wird zusätzlich ein Ableitungsbaum

ausgegeben. Eine Situation (item) ist eine Grammatikregel mit einem zusätzlichen Punkt in der rechten Seite, welcher angibt, wie weit diese Regel bereits analysiert wurde. Abbildung 6 zeigt ein Beispiel.

Der Baum zeigt wie die Situationen und die Vorschauzeichen in den Konfliktzustand gelangen. Im allgemeinen wird für jede beteiligte Situation ein eigener Baum ausgegeben. Sind die Bäume jedoch gleich, wie das in Abbildung 6 der Fall ist, so wird nur ein Baum ausgegeben. Jeder Baum besteht aus drei Teilen: Ein Anfangsteil beginnt mit dem Startsymbol. An einem bestimmten Knoten (Regel) erklären zwei weitere Teilbäume die Entstehung der Situation und der Vorschau.

Jede Zeile enthält die rechte Seite einer Grammatikregel. Normalerweise ist diese rechte Seite so eingerückt, daß sie unter dem Nichtterminal der linken Seite beginnt. Um zu lange Zeilen zu vermeiden, verweisen punktierte Linienzüge ebenfalls zum Nichtterminal der linken Seite und erlauben so am linken Rand fortzufahren. In Abbildung 6 besteht der Anfangsteil des Baumes aus 5 Zeilen, wobei die punktierten Zeilen nicht gezählt wurden. Die Symbole 'stmt' und 'ELSE' sind die Wurzeln der beiden weiteren Teilbäume. Diese Stelle ist mit einem "überflüssigen" Doppelpunkt gekennzeichnet. Nach einem Ableitungsschritt werden im linken Teilbaum die Konfliktsituationen erreicht. Der rechte Teilbaum besteht in diesem Fall nur aus dem Wurzelknoten (dem Symbol 'ELSE') und beschreibt das Vorschauzeichen. Im allgemeinen kann hier ein Baum von beliebiger Größe stehen. Der LR-Konflikt kann in diesem Baumfragment leicht abgelesen werden. Falls bedingte Anweisungen wie gezeigt geschachtelt werden liegt ein Lies-Reduziere-Konflikt vor.

Fehlerbehandlung

Die generierten Zerteiler enthalten Information und Algorithmen um Syntaxfehler vollautomatisch zu behandeln. Es wird die vollständige, rücksetzungsfreie Methode von Röhrich [Röh76, Röh80, Röh82] verwendet. Diese schließt Fehlermeldungen, Wiederaufsetzen und Fehlerreparatur ein. Jede syntaktisch fehlerhafte Eingabe wird gedachterweise in ein korrektes Programm transformiert. Dies bewirkt, daß nur korrekte Folgen von semantischen Aktionen ausgeführt werden. Dadurch brauchen spätere Compiler-Phasen wie die semantische Analyse auf Syntaxfehler keine Rücksicht nehmen. *Lalr* stellt einen Modul zur Sammlung oder Meldung von Fehlern zur Verfügung, der leicht an Benutzerwünsche anpaßbar ist. Ohne Modifikation werden Fehlermeldungen wie in Abbildung 7 ausgegeben. Die Fehlerbehandlung läuft in folgenden Schritten ab:

```
State 266
read reduce conflict
program End-of-Tokens
'PROGRAM' identifier params ';' block '.'
..............................:
:
labels consts types vars procs 'BEGIN' stmts 'END'
....................................:
:
stmt
'IF' expr 'THEN' stmt 'ELSE' stmt
                  :
reduce   stmt -> 'IF' expr 'THEN' stmt.  {'ELSE'}  ?
read     stmt -> 'IF' expr 'THEN' stmt.'ELSE' stmt  ?
```

Abbildung 6: Ableitungsbaum für einen LR-Konflikt (Dangling Else Problem)

Quellprogramm:

```
program test (output);
begin
   if (a = b] write (a);
end.
```

Fehlermeldungen:

```
3, 13: Error       syntax error
3, 13: Information expected symbols: ')' '*' '+' '-' '/' '<' '<='
                   '=' '<>' '>' '>=' 'AND' 'DIV' 'IN' 'MOD' 'OR'
3, 15: Information restart point
3, 15: Repair      symbol inserted : ')'
3, 15: Repair      symbol inserted : 'THEN'
```

Abbildung 7: Beispiel einer automatischen Fehlerbehandlung

- Die Position des Syntaxfehlers wird gemeldet.
- Alle Terminale, die eine korrekte Fortsetzung des Programms wären, werden berechnet und gemeldet.
- Alle Terminale, die zum Wiederaufsetzen der Zerteilung geeignet sind, werden berechnet. Eine minimale Folge von Terminalen wird überlesen, bis eines dieser Symbole gefunden wird.
- Die Position des Wiederaufsetzens wird gemeldet.
- Die Zerteilung wird im sogenannten Reparaturmodus fortgesetzt. In diesem Modus verhält sich der Zerteiler wie gewohnt ohne jedoch Terminale von der Eingabe zu lesen. Stattdessen wird eine minimale Folge von Terminalen generiert, die gedachterweise die überlesenen Terminale ersetzt. Diese generierten Terminale werden gemeldet. Der Zerteiler bleibt in diesem Modus, bis das Terminal am Aufsetzpunkt akzeptiert werden kann. Danach wird der Reparaturmodus verlassen und die Zerteilung normal fortgesetzt.

Der Parser-Generator Ell

Der Parser-Generator *Ell* erzeugt aus LL(1)-Grammatiken Zerteiler nach dem Verfahren des rekursiven Abstiegs. Die Grammatiken können in erweiterter BNF geschrieben sein und semantische Aktionen enthalten. Innerhalb der Aktionen ist es möglich eine L-Attribut-Berechnung zu spezifizieren, die während der Zerteilung ausgewertet wird. Es können sowohl erworbene als auch abgeleitete Attribute benutzt werden, solange die Auswertung in einem Links-Rechts-Durchlauf des Ableitungsbaums möglich ist. Syntaxfehler werden wie bei *Lalr* vollautomatisch behandelt mit Fehlermeldungen, Wiederaufsetzen und Fehlerreparatur. Das Werkzeug ist ebenfalls in Modula-2 programmiert und kann Zerteiler in C und Modula-2 erzeugen. Die Geschwindigkeit der erzeugten Zerteiler liegt im Augenblick bei 450.000 Zeilen pro Minute. Zur Zeit wird an einigen Verbesserungen gearbeitet, wodurch dieser Wert noch steigen wird. Der Speicherbedarf liegt in der Größenordnung von tabellengesteuerten Zerteilern (z. B. 14 KB für Modula-2).

Abbildung 8 zeigt die Spezifikation eines einfachen Tischrechners mit 4 Grundrechenarten für *Ell.* Die Spezifikation erfolgt durch eine LL(1)-Grammatik in erweiterter BNF. Die in geschweifte Klammern eingeschlossenen semantischen Aktionen berechnen wiederum den Wert eines eingegebenen Ausdrucks und geben ihn aus. Der Zugriff auf Attribute erfolgt hier über die Namen der Grammatiksymbole. Da mehrere gleichnamige Symbole in einer Regel vorkommen können werden zur Unterscheidung an die Namen Zahlen angehängt. Die Zahl 0 bezeichnet das Nichtterminal der linken Seite. Mit Zahlen größer 0 werden gleichnamige Symbole der rechten

```
ExpList : ( Expr ';'     { Write (Expr1.value, 10);                          }
          ) *
        .
Expr    : ( ['+'] Term   { Expr0.value := Term1.value;                       }
          | '-' Term     { Expr0.value := - Term2.value;                     }
          )
          ( '+' Term     { INC (Expr0.value, Term3.value);                   }
          | '-' Term     { DEC (Expr0.value, Term4.value);                   }
          ) *
        .
Term    : Factor         { Term0.value := Factor1.value;                     }
          ( '*' Factor   { Term0.value := Term0.value * Factor2.value;       }
          | '/' Factor   { Term0.value := Term0.value DIV Factor3.value;     }
          ) *
        .
Factor  : Number         { Factor0.value := Number1;                         }
        | '(' Expr ')'   { Factor0.value := Expr1.value;                     }
        .
```

Abbildung 8: LL(1)-Grammatik für einfachen Tischrechner

Seite von links nach rechts durchgezählt.

Vergleich von Parser-Generatoren

Abschließend vergleicht die Tabelle 2 die Parser-Generatoren *Lalr* und *Ell* mit dem UNIX Werkzeug *Yacc,* mit dem Yacc-Nachbau *Bison* und mit PGS (siehe oben). Die Tabelle faßt die angesprochenen Eigenschaften zusammen und sollte selbsterklärend sein. Die sprachabhängigen Zahlen sind ohne Laufzeit und Speicherbedarf für Scanner und beziehen sich auf Experimente mit einem Parser für Modula-2.

Zusammenfassung

Die Übersetzerbau-Werkzeuge *Rex, Lalr* und *Ell* wurden mit dem Ziel entworfen, mächtige Spezifikationstechniken mit der automatischen Erzeugung effizienter Compiler-Teile zu kombinieren. Es können Scanner und Parser in C und Modula-2 erzeugt werden, die sich vor allem durch ihre Geschwindigkeit auszeichnen. Die Scanner verarbeiten bis zu 200.000 und die Parser etwa 400.000 Zeile pro Minute auf einem MC 68020 Prozessor. Zusammen erreichen Scanner und Parser mehr als 100.000 Zeilen pro Minute oder fast 2000 Zeilen pro Sekunde.

Da erfahrungsgemäß der Scanner einen Großteil der gesamten Übersetzungszeit verbraucht, hoffen wir vollständige Compiler mit einer Geschwindigkeit von 1000 Zeilen pro Sekunde automatisch erzeugen zu können. Im Augenblick wird an der Vervollständigung des Werkzeugsatzes gearbeitet. Dies betrifft Werkzeuge für die semantische Analyse auf der Basis attributierter Grammatiken, Werkzeuge für die Transformation wie etwa die Erzeugung einer Zwischensprache und Werkzeuge für die Codeerzeugung auf der Basis von Musterabgleich (pattern matching).

Der Generator *Lalr* wurde von Bertram Vielsack programmiert, der auch die experimentellen Ergebnisse für den Vergleich der Parser-Generatoren beitrug. Der Generator *Ell* wurde von Doris Kuske programmiert.

Literatur

[DeP82] F. DeRemer and T. Pennello, Efficient Computation of LALR(1) Look-Ahead Sets, *ACM Trans. Prog. Lang. and Systems 4*, 4 (Oct. 1982), 615-649.

Tabelle 2: Vergleich einiger Parser-Generatoren

	Bison	Yacc	PGS	Lalr	Ell
Spezifikationsmethode	BNF	BNF	EBNF	EBNF	EBNF
Grammatikklasse	LALR(1)	LALR(1)	LALR(1) LR(1) SLR(1)	LALR(1)	LL(1)
semantische Aktionen	ja	ja	ja	ja	ja
S-Attributierung	numerisch	numerisch	symbolisch	numerisch	-
L-Attributierung	-	-	-	-	symbolisch
Konfliktmeldung	Zustand, Situation	Zustand, Situation	Zustand, Situation	Ableitungs- baum	-
Konfliktlösung	Priorität Assoziativität	Priorität Assoziativität	Modifikation	Priorität Assoziativität	
Kettenregelelimination	-	-	ja	-	-
Fehlerbehandlung	von Hand	von Hand	automatisch	automatisch	automatisch
Fehlerreparatur	-	-	ja	ja	ja
Zerteilungsverfahren	tabellen- gesteuert	tabellen- gesteuert	tabellen- gesteuert	tabellen- gesteuert	rekursiver Abstieg
Tabellenkompression	Kammvektor	Kammvektor	Kammvektor	Kammvektor	-
Implementgs.-Sprache	C	C	Pascal	Modula-2	Modula-2
Zielsprachen	C	C	C Modula-2 Pascal Ada	C Modula-2	C Modula-2
Geschwindigkeit [Zeilen/Min.]	105.000	184.000	200.000	385.000	437.000
Tabellengröße [Bytes]	8.004	10.364	11.268	11.795	-
Zerteilergröße [Bytes]	11.136	12.548	17.616	17.416	14.344
Generierungszeit [Sek.]	5,0	19,6	69,5	29,6	6,4

[Gro87] J. Grosch, Rex - A Scanner Generator, Compiler Generation Report No. 5, GMD Forschungsstelle an der Universität Karlsruhe, Dec. 1987.

[GrV88] J. Grosch and B. Vielsack, The Parser Generators Lalr and Ell, Compiler Generation Report No. 8, GMD Forschungsstelle an der Universität Karlsruhe, Apr. 1988.

[Joh75] S. C. Johnson, Yacc — Yet Another Compiler-Compiler, Computer Science Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, July 1975.

[Les75] M. E. Lesk, LEX — A Lexical Analyzer Generator, Computing Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

[Röh76] J. Röhrich, Syntax-Error Recovery in LR-Parsers, in *Informatik-Fachberichte*, vol. 1, H.-J. Schneider and M. Nagl (ed.), Springer Verlag, Berlin, 1976, 175-184.

[Röh80] J. Röhrich, Methods for the Automatic Construction of Error Correcting Parsers, *Acta Inf. 13*, 2 (1980), 115-139.

[Röh82] J. Röhrich, Behandlung syntaktischer Fehler, *Informatik Spektrum 5*, 3 (1982), 171-184.

Inhalt