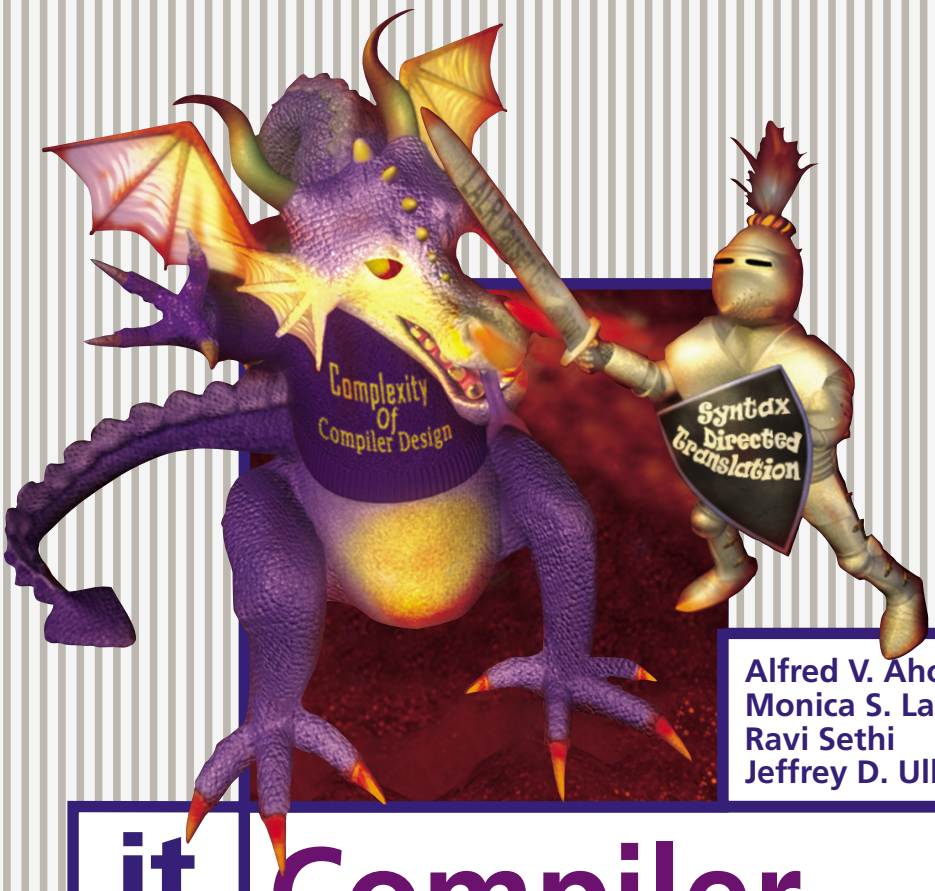

Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman

it
informatik

Compiler

Prinzipien, Techniken und Werkzeuge

2., aktualisierte Auflage

ADDISON-WESLEY

PEARSON
Studium

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* Typ double für den Yacc-Stack */
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines : lines expr '\n'   { printf("%g\n", $2); }
      | lines '\n'
      | /* leer */
      | error '\n' { yyerror("reenter previous line:");
                     yyerrok; }
      ;
expr  : expr '+' expr     { $$ = $1 + $3; }
      | expr '-' expr     { $$ = $1 - $3; }
      | expr '*' expr     { $$ = $1 * $3; }
      | expr '/' expr     { $$ = $1 / $3; }
      | '(' expr ')'      { $$ = $2; }
      | '-' expr %prec UMINUS { $$ = - $2; }
      | NUMBER
      ;
%%
#include "lex.yy.c"
```

Abbildung 4.61: Taschenrechner mit Fehlerbehebung

Außerdem befindet sich Zustand 0 immer am unteren Ende des Stacks. Der Parser verschiebt das Token **error** auf den Stack und fährt fort, Eingabezeichen zu überspringen, bis er eine Zeilenschaltung findet. Er verschiebt das Zeilenschaltungszeichen auf den Stack, reduziert **error** `'\n'` auf *lines* und gibt die Diagnosemeldung „reenter previous line:“ aus. Die `Yacc`-Sonderroutine `yyerrok` setzt den Parser auf seinen normalen Betriebsmodus zurück. ◻

Übungen zu Abschnitt 4.9

! Übung 4.9.1: Schreiben Sie ein `Yacc`-Programm, das als Eingabe boolesche Ausdrücke übernimmt [wie durch die Grammatik in Übung 4.2.2 (g) gegeben] und den Wahrheitswert der Ausdrücke ermittelt.

! Übung 4.9.2: Schreiben Sie ein `Yacc`-Programm, das Listen übernimmt (wie durch die Grammatik in Übung 4.2.2 (e) definiert, aber mit einem beliebigen einzelnen Zeichen als Element, nicht nur mit *a*) und eine lineare Darstellung derselben Liste ausgibt, also eine einzige Liste der Elemente in der Reihenfolge, wie sie in der Eingabe erscheinen.

! Übung 4.9.3: Schreiben Sie ein `Yacc`-Programm, das feststellt, ob es sich bei der Eingabe um ein *Palindrom* handelt (eine Folge von Zeichen, die vorwärts wie rückwärts gelesen gleich lautet).

!! Übung 4.9.4: Schreiben Sie ein `Yacc`-Programm, das reguläre Ausdrücke übernimmt (wie durch die Grammatik in Übung 4.2.2 (d) definiert, aber mit einem beliebigen einzelnen Zeichen als Argument, nicht nur mit *a*) und eine Übergangstabelle für einen nichtdeterministischen endlichen Automaten ausgibt, der dieselbe Sprache erkennt.

ZUSAMMENFASSUNG

- **Parser.** Ein Parser übernimmt Eingabezeichen von einem Lexer und behandelt die Tokennamen als Terminalsymbole einer kontextfreien Grammatik. Anschließend erstellt er einen Parse-Baum für die eingegebene Tokenfolge; dies kann bildlich (als Durchlaufen der betreffenden Ableitungsschritte) oder im Wortsinn erfolgen.
- **Kontextfreie Grammatiken.** Eine Grammatik spezifiziert eine Menge von Terminalsymbolen (Eingaben), eine weitere Menge von Nichtterminalen (Symbole, die syntaktische Konstrukte darstellen) und eine Menge von Produktionen, die jeweils eine Methode angeben, mit der sich Strings, die durch ein einzelnes Nichtterminal dargestellt werden, aus Terminalsymbolen und Strings, die durch bestimmte andere Nichtterminale dargestellt werden, konstruieren lassen. Eine Produktion besteht aus einem Kopf (dem zu ersetzenden Nichtterminal) und einem Rumpf (dem Grammatiksymbolstring, der den Kopf ersetzt).
- **Ableitungen.** Das Verfahren, mit dem ersten Nichtterminal einer Grammatik zu beginnen und es Zug um Zug durch den Rumpf einer ihrer Produktionen zu ersetzen, wird als Ableitung bezeichnet. Wenn immer das am weitesten links (bzw. rechts) stehende Nichtterminal ersetzt wird, handelt es sich um eine Linksableitung (bzw. Rechtsableitung).
- **Parse-Bäume.** Ein Parse-Baum ist ein Bild einer Ableitung, in dem es für jedes Nichtterminal in der Ableitung einen Knoten gibt. Die Kinder eines Knotens sind die Symbole, durch die das Nichtterminal in der Ableitung ersetzt wird. Zwischen Parse-Bäumen, Links- und Rechtsableitungen desselben Terminalstrings besteht eine 1:1-Entsprechung.
- **Mehrdeutigkeit.** Eine Grammatik, für die ein Terminalstring zwei oder mehr unterschiedliche Parse-Bäume oder entsprechend zwei oder mehr Links- bzw. Rechtsableitungen aufweist, wird als mehrdeutig bezeichnet. In den meisten für die Praxis interessanten Fällen lässt sich eine solche Grammatik zu einer eindeutigen Grammatik für dieselbe Sprache umgestalten. Mehrdeutige Grammatiken führen jedoch gelegentlich zu effizienteren Parsern, wenn sie mit bestimmten Tricks eingesetzt werden.
- **Top-Down- und Bottom-Up-Syntaxanalyse.** Parser unterscheiden sich im Allgemeinen darin, ob sie von oben nach unten arbeiten (vom Startsymbol der Grammatik ausgehen und den Parse-Baum von oben her erstellen) oder von unten nach oben (von den Terminalsymbolen ausgehen, die die Blätter des Parse-Baumes bilden, und den Baum von unten her aufbauen). Zu den Top-Down-Parsern gehören die rekursiv absteigenden und die LL-Parser, während die häufigsten Formen der Bottom-Up-Parser LR-Parser sind.
- **Entwurf von Grammatiken.** Grammatiken, die sich für die Top-Down-Syntaxanalyse eignen, sind häufig schwerer zu entwerfen als solche, die von Bottom-Up-Parsern eingesetzt werden. Es ist erforderlich, die Linksrekursion zu beseitigen – eine Situation, in der ein einzelnes Nichtterminal zu einem String abgeleitet wird, der mit demselben Nichtterminal beginnt. Außerdem müssen wir eine Linksfaktorisierung durchführen – also Produktionen für dasselbe Nichtterminal mit einem gemeinsamen Präfix im Rumpf in Gruppen zusammenfassen.

ZUSAMMENFASSUNG

► Fortsetzung

- **Rekursiv absteigende Parser.** Solche Parser verwenden für jedes Nichtterminal eine Prozedur. Diese untersucht die Eingabe und entscheidet, welche Produktion für ihr Nichtterminal anzuwenden ist. Zum geeigneten Zeitpunkt werden Terminale im Rumpf der Produktion mit der Eingabe verglichen, während Nichtterminale im Rumpf einen Aufruf ihrer Prozedur auslösen. Falls die falsche Produktion gewählt wurde, ist eine Zurückverfolgung möglich.
- **LL(1)-Parser.** Eine Grammatik, bei der es möglich ist, die richtige Produktion zum Expandieren eines bestimmten Nichtterminals allein anhand des nächsten Eingabesymbols zu bestimmen, wird als LL(1)-Grammatik bezeichnet. Solche Grammatiken bieten die Möglichkeit, eine prädiktive Parsertabelle zu erstellen, die für die auszuwählende Produktion jedes Nichtterminal- und jedes Lookahead-Symbol korrekt angibt. Die Fehlerbehebung lässt sich vereinfachen, indem einige oder alle Tabelleneinträge, die keine zulässige Produktion haben, durch Fehlerroutinen ersetzt werden.
- **Shift/Reduce-Syntaxanalyse.** Bottom-Up-Parser gehen allgemein so vor, dass sie auf der Grundlage des nächsten Eingabesymbols (dem Lookahead-Symbol) und des Stackinhalts wählen, ob sie die nächste Eingabe auf den Stack verschieben oder einige Symbole reduzieren, die oben auf dem Stack liegen. Ein Reduzierungsschritt ersetzt einen Produktionsrumpf, der oben auf dem Stack liegt, durch den Kopf der Produktion.
- **Sinnvolle Präfixe.** Bei der Shift/Reduce-Syntaxanalyse ist der Stackinhalt immer ein sinnvolles Präfix – also ein Präfix einer rechten Satzform, das nicht weiter reicht als bis zum rechten Ende des Handle dieser Satzform. Der Handle ist der Teilstring, der im letzten Schritt der Rechtsableitung dieser Satzform eingeführt wurde.
- **Gültige Items.** Ein Item ist eine Produktion mit einem Punkt an einer Stelle des Rumpfes. Es ist für ein sinnvolles Präfix gültig, wenn die Produktion dieses Items dazu dient, den Handle zu generieren, und das sinnvolle Präfix alle Symbole links vom Punkt umfasst, aber nicht die darauf folgenden.
- **LR-Parser.** LR-Parser beliebiger Art legen zuerst die Mengen gültiger Items (die sogenannten LR-Zustände) für alle möglichen sinnvollen Präfixe an und verfolgen den Zustand der einzelnen Präfixe auf dem Stack. Die gültige Item-Menge bestimmt die Entscheidung bei der Shift/Reduce-Syntaxanalyse. Wir bevorzugen die Reduktion, wenn ein gültiges Item mit dem Punkt am rechten Ende des Rumpfes vorliegt; steht jedoch das Lookahead-Symbol in einem gültigen Item direkt rechts neben dem Punkt, verschieben wir es lieber auf den Stack.
- **Einfache LR-Parser (SLR-Parser).** Bei einem SLR-Parser führen wir eine Reduktion durch, die von einem gültigen Item mit einem Punkt am rechten Ende ausgelöst wurde, vorausgesetzt, das Lookahead-Symbol kann in einer Satzform auf den Kopf der betreffenden Produktion folgen. Hierbei handelt es sich um eine SLR-Grammatik. Diese Methode kann angewendet werden, wenn es keine Konflikte zwischen Parseraktionen gibt, d. h., wenn es für keine Item-Menge und für kein Lookahead-Symbol zwei Produktionen gibt, durch die eine Reduktion erfolgen kann, und wenn nicht die Option besteht, zu reduzieren oder zu verschieben.

ZUSAMMENFASSUNG ▶ Fortsetzung

- **Kanonische LR-Parser.** Diese komplexere Form der LR-Parser nutzt Items, die um die Menge der Lookahead-Symbole erweitert sind, welche auf die Anwendung der zugrunde liegenden Produktion folgen können. Reduktionen werden nur vorgenommen, wenn ein gültiges Item mit einem Punkt am rechten Ende vorliegt und das aktuelle Lookahead-Symbol für diese Produktion zulässig ist. Ein kanonischer LR-Parser kann einige Aktionskonflikte vermeiden, die bei SLR-Parsern auftreten, umfasst aber häufig mehr Zustände als der SLR-Parser für dieselbe Grammatik.
- **Lookahead-LR-Parser.** LALR-Parser bieten viele Vorteile, die SLR- und kanonische LR-Parser haben, indem sie Zustände mit denselben Kernels zusammenfassen (Item-Mengen, wobei die zugehörigen Lookahead-Mengen ignoriert werden). Die Anzahl der Zustände ist also dieselbe wie bei einem SLR-Parser, aber einige Parseraktionskonflikte können beseitigt werden. LALR-Parser haben sich in der Praxis zur Methode der Wahl entwickelt.
- **Bottom-Up-Syntaxanalyse mehrdeutiger Grammatiken.** In vielen wichtigen Situationen, zum Beispiel bei der Analyse arithmetischer Ausdrücke, können wir eine mehrdeutige Grammatik verwenden und Zusatzinformationen wie die Präzedenz von Operatoren nutzen, um Konflikte zwischen Verschiebung und Reduktion bzw. zwischen zwei verschiedenen Produktionen für die Reduktion zu lösen. LR-Parsertechniken betreffen daher zahlreiche mehrdeutige Grammatiken.
- **Yacc.** Der Parsergenerator Yacc baut aus einer (möglicherweise) mehrdeutigen Grammatik sowie aus Informationen zur Konfliktlösung die LALR-Zustände auf. Anschließend erstellt er eine Funktion, die mithilfe dieser Zustände eine Bottom-Up-Syntaxanalyse durchführt und bei jeder Reduktion eine zugehörige Funktion aufruft.

Literatur zu Kapitel 4

Der Formalismus der kontextfreien Grammatik entstand im Rahmen einer Studie über natürliche Sprachen durch Chomsky [5]. Die Idee wurde außerdem bei der Beschreibung der Syntax zweier früher Sprachen verwendet: Fortran in Backus [2] und Algol 60 in Naur [26]. Der Gelehrte Panini erfand eine gleichwertige syntaktische Notation, um zwischen 400 und 200 v. Chr. die Regeln der Sanskrit-Grammatik zu formulieren [19].

Das Phänomen der Mehrdeutigkeit wurde zuerst von Cantor [4] und Floyd [13] beobachtet. Die Chomsky-Normalform (Beispiel 4.34) stammt aus [6]. Die Theorie der kontextfreien Grammatiken wird in [17] zusammengefasst.

Die rekursiv absteigende Syntaxanalyse war die Methode der Wahl für frühe Compiler wie [16] und compilerverfassende Systeme wie META [28] und TMG [25]. LL-Grammatiken wurden von Lewis und Stearns [24] eingeführt. Übung 4.4.5, die linear mit der Zeit verlaufende Simulation der rekursiv absteigenden Analyse, stammt von [3].

Eine der ersten Analysetechniken, die wir Floyd [14] verdanken, nutzte die Präzedenz von Operatoren. Die Idee wurde von Wirth und Weber allgemeiner gefasst und auf Teile der Sprache übertragen, die nichts mit Operatoren zu tun haben [29]. Diese Techniken werden heute wenig eingesetzt, können aber als erste in einer Kette von Verbesserungen der LR-Syntaxanalyse angesehen werden.

LR-Parser wurden von Knuth [22] eingeführt, von dem auch die kanonischen LR-Parsertabellen stammen. Dieser Ansatz galt als nicht praktisch, weil die Parsertabellen größer waren als die Hauptspeicher der Computer jener Zeit, bis Korenjak [23] eine Methode zum Erstellen von Parsertabellen sinnvoller Größe für gängige Programmiersprachen vorstellte. DeRemer entwickelte die LALR- und die SLR-Methode [8 bzw. 9], die heute in Gebrauch sind. Die Konstruktion von LR-Parsertabellen für mehrdeutige Grammatiken geht auf [1] und [12] zurück.

`Yacc` von Johnson zeigte sehr schnell, dass es möglich war, Parser für Produktionscompiler mit einem LALR-Parsergenerator zu erzeugen. Das Handbuch für den Parsergenerator `Yacc` finden Sie unter [20]. Die Open-Source-Version, `Bison`, ist in [10] beschrieben. Ein ähnlicher auf LALR beruhender Parsergenerator mit dem Namen `CUP` [18] unterstützt in Java geschriebene Aktionen. Generatoren für Top-Down-Parser sind zum Beispiel `Antlr` [27], der rekursiv absteigende Parser generiert und Aktionen in C++, Java oder C# akzeptiert, und `LLGen` [15], der auf LL(1) basiert.

Dain [7] liefert eine Bibliografie über die Behandlung von Syntaxfehlern.

Literatur zu Kapitel 4

Der in Übung 4.4.9 beschriebene Allzweck-Parseralgorithmus für dynamische Programmierung wurde unabhängig voneinander von J. Cocke (unveröffentlicht), Younger [30] und Kasami [21] erfunden und wird daher als „CYK-Algorithmus“ bezeichnet.

Es gibt einen komplexeren Allzweckalgorithmus von Earley [11], der LR-Items für jeden Teilstring der vorliegenden Eingabe in Tabellenform darstellt; er benötigt zwar im Allgemeinen auch $O(n^3)$ an Zeit, für eindeutige Grammatiken aber nur $O(n^2)$.

1. Aho, A. V., S. C. Johnson und J. D. Ullman, *Deterministic parsing of ambiguous grammars*, Comm. ACM 18:8 (Aug., 1975), S. 441–452.

2. Backus, J. W, *The syntax and semantics of the proposed international algebraic language of the Zurich-ACM-GAMM Conference*, Proc. Intl. Conf. Information Processing, UNESCO, Paris, (1959) S. 125–132.

3. Birman, A. und J. D. Ullman, *Parsing algorithms with backtrack*, Information and Control 23:1 (1973), S. 1–34.

4. Cantor, D. C., *On the ambiguity problem of Backus systems*, J. ACM 9:4 (1962), S. 477–479.

5. Chomsky, N., *Three models for the description of language*, IRE Trans. on Information Theory IT-2:3 (1956), S. 113–124.

6. Chomsky, N., *On certain formal properties of grammars*, Information and Control 2:2 (1959), S. 137–167.

7. Dain, J., *Bibliography on Syntax Error Handling in Language Translation Systems*, 1991. Available from the comp.compilers newsgroup;
   Siehe `http://compilers.iecc.com/comparch/article/91-04-050`.

8. DeRemer, F., *Practical Translators for LR(k) Languages*, Doktorarbeit, MIT, Cambridge, MA, 1969.

9. DeRemer, F., *Simple LR(k) grammars*, Comm. ACM 14:7 (Juli, 1971), S. 453–460.

10. Donnelly, C. und R. Stallman, *Bison: The YACC-compatible Parser Generator*, `http://www.gnu.org/software/bison/manual/`.

11. Earley, J., *An effcient context-free parsing algorithm*, Comm. ACM 13:2 (Feb., 1970), S. 94–102.

12. Earley, J., *Ambiguity and precedence in syntax description*, Acta Informatica 4:2 (1975), S. 183–192.

Literatur zu Kapitel 4

13. Floyd, R. W., *On ambiguity in phrase-structure languages*, Comm. ACM 5:10 (Okt., 1962), S. 526–534.

14. Floyd, R. W., *Syntactic analysis and operator precedence*, J. ACM 10:3 (1963), S. 316–333.

15. Grune, D und C. J. H. Jacobs, *A programmer-friendly LL(1) parser generator*, Software Practice and Experience 18:1 (Jan., 1988), S. 29–38.
Siehe auch http://www.cs.vu.nl/~ceriel/LLgen.html.

16. Hoare, C. A. R., *Report on the Elliott Algol translator*, Computer J. 5:2 (1962), S. 127–129.

17. Hopcroft, J. E., R. Motwani und J. D. Ullman, *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*, Pearson Studium, München, 2003.

18. Hudson, S. E. et al., *CUP LALR Parser Generator in Java*,
erhältlich unter http://www2.cs.tum.edu/projects/cup/.

19. Ingerman, P. Z., *Panini-Backus form suggested*, Comm. ACM 10:3 (März 1967), S. 137.

20. Johnson, S. C., *Yacc - Yet Another Compiler Compiler*, Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975.
Erhältlich unter http://dinosaur.compilertools.net/yacc/.

21. Kasami, T., *An effcient recognition and syntax analysis algorithm for context-free languages*, AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

22. Knuth, D. E., *On the translation of languages from left to right* Information and Control 8:6 (1965), S. 607–639.

23. Korenjak, A. J., *A practical method for constructing LR(k) processors*, Comm. ACM 12:11 (Nov., 1969), S. 613–623.

24. Lewis, P. M. II und R. E. Stearns, *Syntax-directed transduction*, J. ACM 15:3 (1968), S. 465–488.

25. McClure, R. M., *TMG - a syntax-directed compiler*, proc. 20th ACM Natl. Conf. (1965), S. 262–274.

26. Naur, P. et al., *Report on the algorithmic language ALGOL 60*, Comm. ACM 3:5 (Mai, 1960), S. 299–314. Siehe auch Comm. ACM 6:1 (Jan., 1963), S. 1–17.

Literatur zu Kapitel 4

27. http://www.antlr.org/

28. Schorre, D. V., *Meta-II: a syntax-oriented compiler writing language*, Proc. 19th ACM Natl. Conf. (1964) S. D1.3-1–D1.3-11.

29. Wirth, N. und H. Weber, *Euler: a generalization of Algol and its formal defnition: Part I*, Comm. ACM 9:1 (Jan., 1966), S. 13–23.

30. Younger, D. H., *Recognition and parsing of context-free languages in time n3*, Information und Control 10:2 (1967), S. 189–208.

Syntaxgerichtete Übersetzung

5

ÜBERBLICK