Oliver Müller

Parsertechniken in C++

Oliver Müller, Kempten (Allgäu)
oliver@cogito-ergo-sum.org

Lektorat: Lutz Friedrich
Umschlagdesign: Oliver Müller
Umschlagfoto: © Olena Shkrob und Oliver Müller, 2007

Das Umschlagfoto zeigt den Liwadija-Palast auf der Krim (Ukraine).

Oliver Müller

Parsertechniken in C++

Inhaltsverzeichnis

Vorwort

Die Parsertechniken sind ein faszinierendes Gebiet. Die intensive Auseinandersetzung mit diesem Themenbereich bedurfte bislang einer Odyssee durch den Urwald der Fachliteratur. Die zumeist nur in englischer Sprache vorliegenden Werke befassen sich dabei häufig mit dem Parsing nur am Rande der formalen Sprachtheorie, des Compilerbaus oder im Zuge des Unix-Systems und dessen Tools lex und yacc. Hinzukommt, daß die meisten grundlegenden praxisorientierten Werke mittlerweile mindestens vor einem Jahrzehnt geschrieben wurden und somit neue Erkenntnisse wie das objektorientierte Paradigma nicht berücksichtigen.

Das Suchen, Finden und Beschaffen des entsprechenden Fachliteratur und die Zusammenführung der Erkenntnisse für eine umfassende Betrachtung der Thematik aus mehreren Blickwinkeln ist teilweise eine nicht triviale Aufgabe. Besonders dann nicht, wenn man noch den „State of the Art“ der Software-Entwicklung berücksichtigen will, also zum Beispiel Erkenntnisse der sich zur Philosophie auswachsenden Objektorientierung einbeziehen will.

Ich hoffe, daß dieses Buch Ihnen eine solide Basis an Grundwissen gibt, Ihnen aber auch die Weiterbildung Ihren Ambitionen entsprechend durch die Einstreuung von Verweisen auf die Fachliteratur erleichtern wird.

Ich wünsche Ihnen, verehrte Leserin, verehrter Leser, daß Sie durch das Studium dieses Buches ein Stück über sich hinauswachsen werden, so wie ich beim Schreiben dieses Werks und dem damit zwangsläufig verbundenen erneuten und vertiefenden Beschäftigen mit der Thematik meine Leistungsgrenzen wieder ein Stück verschieben konnte. Des weiteren wünsche ich Ihnen, daß Sie nicht nur Wissen aus diesem Buch beziehen, sondern auch eine gute Portion Spaß aus dem Experimentieren mit den Beispielen schöpfen werden.

Danksagung

An dieser Stelle möchte ich mich bei Herrn Friedrich für die Chance dieses Buch schreiben und veröffentlichen zu dürfen bedanken. Ebenso danke ich ihm und Herrn Nickles für das entgegengebrachte Vertrauen in diese Buchidee.

Bei der Redaktion des „PC Magazin“ im allgemeinen und Frau Kathrin Nagy im speziellen möchte ich mich für das schnelle und unkomplizierte Beschaffen von Kopien älterer Artikel bedanken.

Kempten, im Februar 1998

Oliver Müller

Dieses Buch widme ich meinen Eltern.

1 Einleitung

1.1 Über dieses Buch

Das Ziel dieses Buches ist es Ihnen einen soliden Grundstock an Basiswissen zur Programmierung von Parsern zu geben. Dabei ist das Hauptaugenmerk auf die praktischen Aspekte der Parsertechniken gelegt. Für eine ausführliche theoretische Betrachtung sei auf [19], [20] und [7] verwiesen. Die nötigsten Grundlagen in formalen Sprachen und Automatentheorie wird soweit eingegangen, daß Sie auch ohne tieferes theoretisches Grundwissen diesem Buch folgen können. Sie sollten jedoch über mathematische Grundkenntnisse, insbesondere der Mengenlehre, verfügen.

Das Buch arbeitet durchgehend mit Beispielen, die die Prinzipien der Parsing-Theorie erläutern. Diese Quelltexte sind dabei objektorientiert in C++ realisiert. Eine Intention dieses Werkes ist es die Parser aus dem Bereich der strukturierten Programmierung herauszuheben und dem moderneren objektorientierten Paradigma zuzuführen. Sie sollten daher mit der Programmiersprache C++ und dem Fachjargon der objektorientierten Techniken vertraut sein.

Eine objektorientierte Implementation eines Parsers mag zwar die Gefahr in sich bergen die Turnaround-Zeiten zu erhöhen, aber die Vorteile eines verbesserten Handlings der inhärenten Komplexität moderner Software-Systeme und damit einer besseren Wartbarkeit eines OO-Systems dürften selbst den „Taktzyklenjäger“ überzeugen.

Bei der heutigen Marktlage sind eben weniger die Taktzyklen entscheidend für den Erfolg einer Software, sondern die Produktzyklen.

1.2 Aufbau dieses Buches

Das Buch ist im wesentlichen so organisiert, daß sich jedes Kapitel in zwei Teile gliedert. Im ersten Teil werden die Grundlagen des behandelten Themenbereichs dargestellt, die im zweiten Part durch die praktische Anwendung vertieft werden.

Die praktischen Beispiele beginnen in Kapitel 2 mit der einfachen Grammatik eines kleinen Rechnerprogramms und der Implementierung des entsprechenden Parsers. In den Kapiteln 3 und 4 wird dieses Programm nach und nach zu einem komfortableren Interpreter mit Stack-Maschine ausgebaut, der seinen Climax in Kapitel 5 und 6 in Form einer kleinen Programmiersprache mit Kontrollstrukturen und Unterprogrammtechnik findet.

Kapitel 2 gibt einen Überblick über die Parsertechniken. Grundbegriffe wie die der formalen Sprachen und der grammatikalischen Analyse werden erläutert.

Kapitel 3 geht tiefer auf die lexikalische Analyse und die verschiedenen Konzepte zur Konstruktion von Scannern ein.

Kapitel 4 befaßt sich mit der Syntaxanalyse, dem eigentlichen Parsing. Hier werden die wesentlichen Strategien - Top-Down- und Bottom-Up-Ansatz - vorgestellt.

Kapitel 5 widmet sich wieder der lexikalischen Analyse, aber in Form des praktischen Einsatzes des Scannergenerators lex.

Kapitel 6 hat wiederum die syntaktische Analyse zum Thema. Hier wird allerdings der Parsergenerator yacc vorgestellt.

2 Einführung in die Parsing-Theorie

2.1 Einleitung

Bevor wir in die Tiefen der Parserprogrammierung hinabsteigen können, müssen wir zunächst einige Begriffe klären. Der zentrale Inhalt dieses Buches ist der *Parser*. [17] definiert einen Parser wie folgt:

„Analysierer, syntaktischer. (syntactic analyzer, parser) Als s. A. bezeichnet man ein Programm, das eine syntaktische Analyse durchführen kann und damit in der Lage ist, ein Primärprogramm in ein Maschinenprogramm umzuwandeln. Der s. A. ist ein wesentliches Element eines Übersetzungsprogramms."

In [1] findet sich unter dem Stichwort „Parser":

„An algorithm or program to determine the syntactic structure of a sentence or string of symbols in some language. A parser normally takes as input a sequence of tokens output by a lexical analyser. It may produce some kind of abstract syntax tree as output."

Ein Parser ist demnach ein Programm oder ein Teil eines Programms, das eine Texteingabe syntaktisch auswerten, das heißt anhand einer Grammatik zerlegen kann. Das geht mit der Übersetzung des englischen „to parse", das „grammatikalisch zerlegen" bedeutet, konform.

Die Anwendungsgebiete von Parsern sind dabei heute nicht mehr nur auf den Compilerbau beschränkt, obwohl dies immer noch der wichtigste Anwendungsbereich ist. Parsern begegnet man in den zu Compilern verwandten Interpretern von Programmier- und Makrosprachen, Tabellenkalkulationen und auch Kommandoprozessoren. Kurz gesagt in allen Bereichen, wo Daten anhand einer Grammatik analysiert werden müssen.

Eine Grammatik ist ein formales Regelsystem zur Beschreibung der syntaktischen Struktur einer Sprache. In der Sprachtheorie existieren mehrere Formen von Grammatiken und unterschiedliche Darstellungsarten der selben. Im Bereich der Parsertechniken sind für die syntaktische Analyse jedoch lediglich die sogenannten *kontextfreien Grammatiken* von praktischer Bedeutung.

2.2 Kontextfreie Grammatiken

Eine kontextfreie Grammatik G stellt sich mathematisch als ein Quadrupel (4-Tupel) G = (V, T, P, S) dar. Sie beschreibt eine kontextfreie Sprache L(G). V ist eine endliche Menge von sogenannten *nicht-terminalen Symbolen*, gelegentlich auch als *Variablen*, *Nichtterminale* und *syntaktische Kategorien* bezeichnet. Die Elemente dieser Menge V repräsentieren Symbole, die aus anderen Symbolen zusammengesetzt sind und in diese zerlegt werden können.

T ist die endliche Menge der *terminalen Symbolen*, auch als *Terminale* oder *primitive Symbole* bezeichnet. Die Elemente von T sind die atomaren Sprachbestandteile, d. h. sie können nicht weiter zerlegt werden. Im Falle einer Programmiersprache sind dies beispielsweise die Schlüsselwörter und Zeichen.

P ist ebenfalls eine endliche Menge und stellt den wesentlichen Charakter der kontextfreien Grammatiken dar. P beinhaltet als Elemente die *Produktionen* der kontextfreien Grammatik. Statt Produktionen findet sich auch gelegentlich die Bezeichnung *Regeln*.

Eine Produktion ist die Ableitungsvorschrift für ein nichtterminales Symbol. Es beschreibt in welche Elemente der Mengen V und T sich das Nichtterminal zerlegen läßt. Produktionen haben dabei die Form

```
linke Seite --> rechte Seite
```

linke Seite muß dabei das nicht-terminale Symbol, für welches die Produktion gültig ist, sein. *rechte Seite* ist eine Folge von terminalen und/oder nicht-terminalen Symbolen in die das nicht-terminale Symbol der linken Seite abgeleitet bzw. zerlegt werden kann. Die rechte Seite ist somit die *Ableitung* für das Nichtterminal der linken Seite.

S ist ein spezielles nicht-terminales Symbol aus V. Es ist das *Startsymbol*.

Die kontextfreie Grammatik, nach der ein einfacher Taschenrechner funktioniert, hat zum Beispiel folgendes Aussehen:

```
<Ausdruck> --> <Term> + <Ausdruck>
<Ausdruck> --> <Term> - <Ausdruck>
<Ausdruck> --> <Term>
<Term>     --> <Faktor> * <Term>
<Term>     --> <Faktor> / <Term>
<Term>     --> <Faktor>
<Faktor>   --> Zahl
<Faktor>   --> + Zahl
<Faktor>   --> - Zahl
<Faktor>   --> ( <Ausdruck> )
```

Alle nicht-terminalen Symbole sind in spitze Klammern eingefaßt. V ist demnach gleich { `<Ausdruck>`, `<Term>`, `<Faktor>` }. Die Terminale sind die Operatorzeichen und

`Zahl`. Strenggenommen könnte (oder müßte) auch `Zahl` durch eine Produktion angegeben werden. Damit die Grammatik jedoch in überschaubaren Dimensionen bleibt, soll hier darauf verzichtet werden. T ist also gleich { `+`, `-`, `(`, `)`, `*`, `/`, `Zahl` }.

Die Menge der Produktionen P setzt sich schlicht aus den obigen 10 Zeilen zusammen. Wie Sie sehen ist es in kontextfreien Grammatiken durchaus möglich Produktionen rekursiv zu definieren. `<Faktor>` kann als Bestandteil von `<Term>` und damit auch von `<Ausdruck>` selbst wieder `<Ausdruck>` und damit unter Umständen auch wieder sich selbst in seinen Produktionen enthalten.

Durch diese Möglichkeit der rekursive Definition kann die Grammatik analog zum Prinzip der rekursiven Programmierung auf die einfachsten Fälle konzentriert werden. Das software-technische Potential, das sich hiermit zum Aufbau von Parsern eröffnet, ist wohl mitverantwortlich für den Erfolg der kontextfreien Grammatiken in der Computer-Linguistik.

Man unterscheidet hierbei im wesentlichen vier Typen von rekusiven Definitionen:

1. *Linksrekursion*. Entspricht das am weitesten links stehende Symbol der rechten Seite dem Symbol der linken Seite, so spricht man von einer Linksrekursion.
   Beispiel: A --> A α Das Nichtterminal A ist *linksrekursiv*.
2. *Rechtsrekursion*. Ist das am weitesten rechts stehende Symbol der rechten Seite identisch mit dem Symbol der linken Seite, so liegt eine Rechtsrekursion vor.
   Beispiel: A --> α A Das Nichtterminal A ist *rechtsrekursiv*.
3. *Selbsteinbettung*. Ist ein Symbol der rechten Seite der Produktion identisch mit dem der linken Seite, so spricht man von Selbsteinbettung.
   Beispiel: A --> α A β Das Nichtterminal A ist *selbsteinbettend* (engl. self-embedding).
4. *Zykel*. Bettet sich ein Symbol über den Umweg über andere Produktionen selbst ein, so liegt ein Zykel vor.
   Beispiel: A --> α B; B --> β A

Das Startsymbol S ist `<Ausdruck>`. Im folgenden soll sofern nichts anderes angeben ist implizit immer gelten, daß das nicht-terminale Symbol, auf das sich die erste angegebene Produktion bezieht, das Startsymbol ist.

Gemäß der oben angegebenen kontextfreien Grammatik wäre ein Rechnerprogramm, das einen Parser dieser Grammatik besitzt, in der Lage

- die Operatorenpriorität („Punkt vor Strich“) korrekt zu interpretieren,
- etwaige Vorzeichen zuzuordnen und
- durch gesetzte Klammern die Operatorenpriorität entsprechend aufzuheben.

Wenn wir im Geist eine Top-Down-Syntaxanalyse für den Ausdruck 1+(2+5)*4 durchgehen, so erhalten wir folgende Situation:

```
<Ausdruck>
<Term> + <Ausdruck>
<Faktor> + <Term>
Zahl + <Faktor> * <Term>
Zahl + ( <Ausdruck> ) * <Faktor>
Zahl + ( <Term> + <Ausdruck> ) * Zahl
Zahl + ( <Faktor> + <Term> ) * Zahl
Zahl + ( Zahl + <Faktor> ) * Zahl
Zahl + ( Zahl + Zahl ) * Zahl
```

Jedes nicht-terminale Symbol des einen Schritts wird im nächsten Schritt durch die Ableitungsvorschrift einer passenden Produktion ersetzt. So wird das erste `<Ausdruck>` durch `<Term> + <Ausdruck>` ersetzt. Dies entspricht exakt der rechten Seite der ersten Produktion der kontextfreien Grammatik von oben. Im zweiten Schritt wird `<Term>` durch `<Faktor>` gemäß der sechsten Produktion und `<Ausdruck>` laut der dritten Regel durch `<Term>` ersetzt und so weiter. Nach und nach wird so der Text grammatikalisch zerlegt.

Wenn wir nun in umgekehrter Richtung jeweils die Zahlenwerte einsetzen und immer wenn sich mehrere Symbole zu einem Nichtterminal zusammenfügen, den Wert der arithmetischen Operation einsetzen, so erhalten wir zum Schluß, d. h. bei Erreichen der Zeile `<Ausdruck>`, das korrekte Rechnenergebnis:

```
1 + ( 2 + 5 ) * 4   : Zahl + ( Zahl + Zahl ) * Zahl
1 + ( 2 + 5 ) * 4   : Zahl + ( Zahl + <Faktor> ) * Zahl
1 + ( 2 + 5 ) * 4   : Zahl + ( <Faktor> + <Term> ) * Zahl
1 + ( 2 + 5 ) * 4   : Zahl + ( <Term> + <Ausdruck> ) * Zahl
1 + ( 7 ) * 4       : Zahl + ( <Ausdruck> ) * <Faktor>
1 + 7 * 4           : Zahl + <Faktor> * <Term>
1 + 28              : <Faktor> + <Term>
1 + 28              : <Term> + <Ausdruck>
29                  : <Ausdruck>
```

Da die Darstellung komplexerer Ableitungen in der oben gezeigten Form sehr leicht unübersichtlich wird, ist es nützlich den Parse-Vorgang als Baum darzustellen. Diese graphische Darstellungsform nennt man *Ableitungs-*, *Syntax-* oder *Parse-Baum*. In Abbildung 2.1 sehen Sie den Parse-Baum für das Beispiel 1+(2+5)*4.

Ein Parse-Baum ist ein Graph einer Ableitung einer kontextfreien Grammatik. Jeder Knoten repräsentiert entweder ein terminales oder nicht-terminales Symbol. Als Spezialfall kann auch das weiter unten erläuterte Symbol ε als Knoten, genauer als Blatt auftreten.

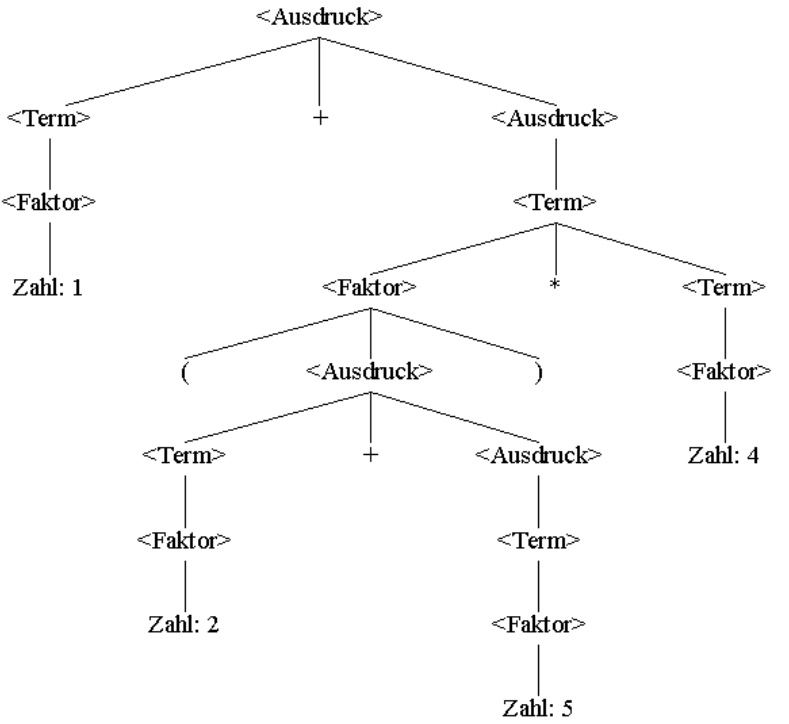


Abb 2.1: Parse-Baum für 1+(2+5)*4

Alle inneren Knoten sind Nichtterminale. Alle Blätter sind Terminale oder ε. Die Wurzel des Baumes stellt das Startsymbol S dar. Die *Front* α eines Parse-Baumes ist das Wort, welches durch die Konkatenation der Terminale an den Blättern von links nach rechts ergibt. ε-Ableitungen werden dabei als neutrales Element ignoriert.

Die Söhne eines Knotens von links nach rechts gelesen ergeben die rechte Seite einer Produktion (=Ableitung) dieses Knotens. Die Söhne der Wurzel `<Ausdruck>` aus Abbildung 2.1 ergeben beispielsweise `<Term> + <Ausdruck>`. Dies ist die rechte Seite der ersten Produktion der kontextfreien Grammatik.

ε, seltener auch mit λ oder *empty* bezeichnet, stellt in einer Grammatik ein leeres Symbol dar. Dies ist sinnvoll zur Formulierung von optionalen Konstrukten oder auch zum Aufbau von Parsern, die Leerzeilen verarbeiten, d. h. ignorieren, sollen.

Hier sei exemplarisch auf die Deklaration der Formalparameter in einer C-Funktion verwiesen. Diese kann entweder eine Parameterliste enthalten oder eine leere Klammer darstellen. In einer kontextfreien Grammatik könnte dies wie folgt ausgedrückt werden:

. . .

```
<Fktskopf>   --> <Typ> <Bez> ( <FormParam> )
<FormParam>  --> <Typ> <Bez> <FormParam2>
<FormParam>  --> ε
<FormParam2> --> , <Typ> <Bez> <FormParam2>
<FormParam2> --> ε
...
```

Ein Parser, der auf dieser Grammatik aufbaut, würde sowohl

```
void meineFunktion()
```

als auch

```
void meineFunktion(int a, int b)
```

akzeptieren.

Produktionen der Form <Nichtterminal> --> ε nennt man ε-*Produktionen*. Selbstverständlich darf ein ε nur als Blatt in einem Ableitungsbaum und damit nur auf der rechten Seite einer Produktion auftreten. Ein Abweichen von dieser Regel würde bedeuten, daß für ε eine Produktion vereinbart werden würde. Das Resultat dieser Abweichung wäre das Umdefinieren von ε. ε stünde dann nicht mehr für das leere Symbol und wäre somit ein Element der Menge V - ein gewöhnliches Nichtterminal!

2.3 Backus-Naur-Form

Die ursprüngliche Motivation zur Entwicklung der kontextfreien Grammatiken war die Beschreibung von natürlichen Sprachen. Die Komplexität, die natürlichen Sprachen zueigen ist, lies die kontextfreien Grammatiken allerdings sehr schnell als nicht adäquates Notationsmittel für die natürlichen Sprachen erscheinen. Für die Beschreibung von künstlichen (Computer-)Sprachen stellen Sie allerdings einen wichtigen Fortschritt dar.

In den Jahren 1959 und 1960 entwickelten J. Backus und P. Naur die nach Ihnen benannte *Backus-Naur-Form* (*BNF*). Diese auf dem Prinzip der kontextfreien Grammatiken aufbauende Beschreibungsform für syntaktische Zusammenhänge wurde erfolgreich zur Spezifikation und Dokumentation der Progammiersprache ALGOL 60 eingesetzt.

Der Erfolg der kontextfreien Grammatik beziehungsweise der BNF ist ohne Zweifel in der einfachen und logischen Weise, in der sich Grammatiken von Programmiersprachen damit beschreiben lassen, begründet. Der Entwurf und die folgende Codierung von Parsern wurde durch dieses Beschreibungsmittel erheblich vereinfacht.

Die BNF ist der Notation von kontextfreien Grammatiken sehr ähnlich. Nichtterminale werden ebenfalls in spitze Klammern eingefaßt und die Produktionen haben bis auf notationstechnische Details und Abkürzungen das selbe Aussehen.

Die Grammatik von oben liese sich als BNF wie folgt darstellen:

```
<Ausdruck> ::= <Term> ( "+" | "-" ) <Ausdruck>
             | <Term>
<Term>     ::= <Faktor> ( "*" | "/" ) <Term>
             | <Faktor>
<Faktor>   ::= Zahl
             | ( "+" | "-" ) Zahl
             | "(" <Ausdruck> ")"
```

Statt des --> aus der Notation der kontextfreien Grammatiken findet sich hier nun das ::=. In vielen Werken findet sich allerdings auch in BNF-Listings der Pfeil. Dies ist zwar nicht konform zur BNF, ist aber infolge des häufigen Auftretens nicht zu verurteilen und allgemein anerkannt.

Das augenfälligste Merkmal der BNF ist, daß es nicht mehr nötig ist für ein nicht-terminales Symbol mehrere Produktionen anzugeben. Durch den Oderoperator | lassen sich nun mehrere Regeln zusammenfassen.

In der BNF ist es auch möglich mehrere Parts einer Produktion durch Klammern zusammenzufassen. Dadurch läßt sich die Grammatik weiter vereinfachen und überschaubarer gestalten. Die Produktionen von `<Ausdruck>` für die Addition und Substration beispielsweise lassen sich so in einer Regel kombinieren.

Das Einführen weiterer Metasymbole wie der Klammern hat allerdings zur Folge, daß diese nicht mehr ohne weiteres als Terminale verwendet werden können. Daher empfiehlt es sich zumindest Sonderzeichen und ähnliche aus einem Zeichen bestehende Terminale in Anführungszeichen zu fassen.

Die Oderoperatoren der zusammengefaßten Produktionen müssen im übrigen nicht zwangsläufig unter dem ::= stehen. Sie könnten die alternativen Regeln auch fortlaufend in einer Zeile notieren. Der Übersichtlichkeit wegen bietet sich jedoch die obige Formatierung an.

2.3.1 EBNF

Da Informatiker wie Mathematiker gerne Abkürzungen einführen, um sich die Schreibarbeit zu erleichtern, dauerte es nicht lange bis auch in der BNF entsprechende Abkürzungen auftauchten. So führte beispielsweise IBM in der Definition von PL/I die eckigen Klammern zur Kennzeichnung von optionalen Symbolen ein. Wieder andere führten zusätzliche Metasymbole ein, um Wiederholungen beschreiben zu können. Diese Metasymbole stammen zum größten Teil aus dem Bereich der regulären Ausdrücke und lehnen sich an die unter Unix üblichen Notation der selben an.

Da diese Erweiterungen sehr nützlich für die praktische Arbeit sind, wurden diese später zur sogenannten *Erweiterten Backus-Naur-Form* (*EBNF*) zusammengefaßt.

Die zusätzlichen Metazeichen der EBNF gegenüber der BNF sind:

Metasymbol	*Aufgabe*
[sym]	Optionales Symbol sym
{s1, s2, s3}	Durch Kommata getrennte alternative Symbole
{sym}*	Kleene-Closure, deckt beliebig viele Vorkommen des Symbols sym ab (auch 0)
{sym}+	Plus-Closure, deckt beliebig viele Vorkommen des Symbols sym ab (min. 1)
$\{sym\}_n^m$	Interval-Closure, deckt zwischen n und m Vorkommen von sym ab
$\{sym\}^{n,m}$	dto.
{sym}n,m	dto. (sinnvoll bei Arbeitsplätzen, die lediglich Textmodus bieten)

Unter Zuhilfenahme der EBNF könnte die obige Grammatik noch weiter vereinfacht werden:

```
<Ausdruck> ::= <Term> [ ( "+" | "-" ) <Ausdruck> ]
<Term>     ::= <Faktor> [ ( "*" | "/" ) <Term> ]
<Faktor>   ::= [ ( "+" | "-" ) ] Zahl
             | "(" <Ausdruck> ")"
```

Oder:

```
<Ausdruck> ::= <Term> [ {"+","-"} <Ausdruck> ]
<Term>     ::= <Faktor> [ {"*","/"} <Term> ]
<Faktor>   ::= [ {"+","-"} ] Zahl
             | "(" <Ausdruck> ")"
```

2.4 Analyse eines Quelltextes

Die Analyse eines Quelltext gliedert sich in drei Phasen:

1. *Lineare* - oder *lexikalische Analyse*: Der Quelltext wird in die terminalen Symbole, sogenannte *Tokens* aufgeteilt. Dieser Vorgang wird *Scanning* oder neudeutsch *Scannen* genannt. Der Algorithmus, der die lexikalische Analyse durchführt, wird *Scanner* genannt.
2. *Hierarchische* - oder *syntaktische Analyse*: Die Tokens (oder Zeichen falls kein Scanner zur Anwendung kommt) werden gemäß den Produktionen zu Nichtterminalen zusammengefaßt und ausgewertet. Dieser Arbeitsschritt heißt *Parsing* oder *Parsen*. Der durchführende Programmteil ist der *Parser*.
3. *Semantische Analyse*: Die Bestandteile des Programms werden dahingehen überprüft, ob sie zusammenpassen, also in einer sinnvollen Konstellation vorliegen.

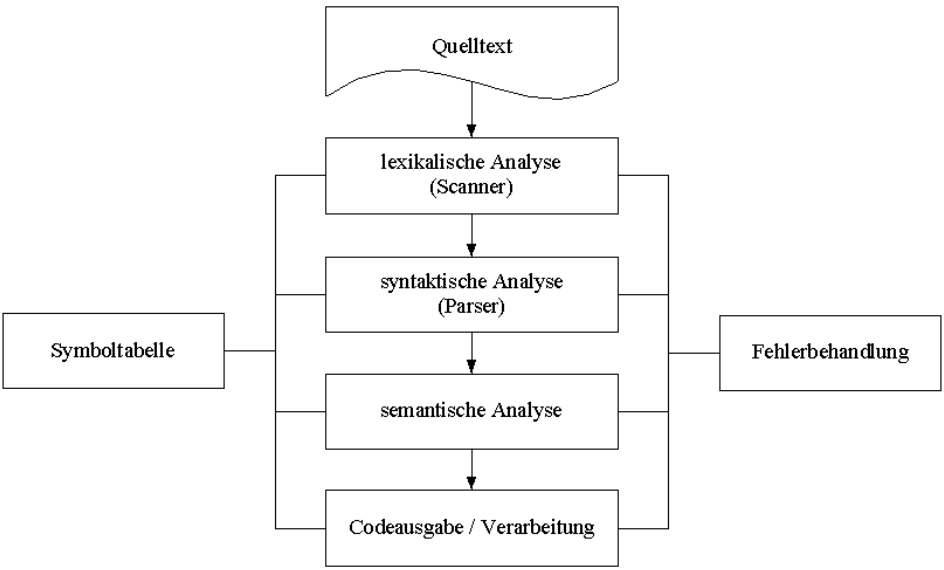


Abb. 2.2: Phasen bei der Analyse eines Quelltextes

Obwohl strenggenommen nur die Syntaxanalyse vom Parser vorgenommen wird, werden wir die lexikalische - und die semantische Analyse hier ebenfalls betrachten. Sie sind essentieller Bestandteil der Parsertechniken. Außerdem findet man gerade bei einfacheren Grammatiken und damit kleineren Parsern häufig in der Praxis diese drei Analyseschritte direkt im Parser implementiert.

Als Eingabestrom für die lexikalische Analyse dient der Quelltext. Die Ausgabe dieses Schrittes bildet sogleich die Eingabe der syntaktischen Analyse, die wiederum Ihre Ausgabe an den semantische Analysierer weiterreicht. Das Ergebnis der semantischen Analyse geht dann als Eingabe an einen Codegenerator oder -interpreter. Diesen Vorgang finden Sie in Abbildung 2.2 grafisch dargestellt.

Der Scanner, der Parser und der semantische Analysierer stehen während Ihrer Arbeit in Wechelbeziehungen mit der *Fehlerbehandlung* und der *Symboltabelle*, auch als *Wort-* oder *Tokendatenbank* bezeichnet. Diese zwei Module werden auch von folgenden Schritten wie beispielsweise der Codererzeugung genutzt.

2.4.1 Die Symboltabelle

Die Symboltabelle ist eine als Array organisierte Datenstruktur, die für jeden Bezeichner innerhalb des Quelltextes, also Variablen, Funktionen, Objekte, etc., einen Record enthält. In diesen Record werden die Attribute des Bezeichners wie zum Beispiel Typ, Speicherbedarf, Adresse (bei Compilern) und Wert (bei Interpretern) gespeichert.

Erkennt die lexikalische Analyse einen Bezeichner, so trägt sie ihn in diese Tabelle ein. Die Attribute können in der Regel vom Scanner aber meist nicht gesetzt werden. Denken Sie hier beispielsweise an folgende C-Anweisung:

```
int a, b;
```

`int` wird vom Scanner nur als Schlüsselwort erkannt; die Bedeutung dieses Token ist ihm jedoch nicht bekannt. Die Interpretierung des `int` als Einleitung einer Integer-Deklaration bleibt der syntaktischen Analyse, also dem Parser, vorbehalten. Dieser folgt allerdings erst nach der lexikalischen Analyse! Infolgedessen kann der Scanner die Attribute beim Erkennen von `a` und `b` als Bezeichner nicht setzen, denn er weiß (!) ja nicht, daß es sich hierbei um Integer-Variablen handelt.

Erst die folgenden Phasen ergänzen die fehlenden Informationen und verwenden sie nach Bedarf.

2.4.2 Organisation der Phasen

Sind die drei Phasen der Analyse - lexikalische -, syntaktische - und semantische Analse - jeweils getrennt implementiert, so sind für die Analyse des Quelltextes drei Durchläufe notwendig. Da dieses Vorgehen aber nicht absonderlich effektiv ist, ist es üblich mehrere Phasen zu einem *Lauf* oder *Pass* zusammenzufassen.

Die drei Analyse-Schritte werden meist in einem Pass organisiert werden. Der Parser übernimmt dabei die zentrale Rolle. Er fordert, wann immer er ein Symbol benötigt, dieses vom Scanner an und versucht es in die grammatikale Struktur einzuordnen. Ist eine in sich abgeschlossene Struktur erkannt worden, wird die Semantik geprüft und ein (Zwischen-)Codefragment für den nächsten Pass erzeugt.

Das Zusammenfassen von Phasen zu Läufen ist vorteilhaft, da eine geringe Anzahl von Passes die Geschwindigkeit erhöht. Dieser Geschwindigkeitsvorteil resultiert aus der Reduzierung des zeitintensiven Auslagern von Zwischenergebnissen in Dateien.

Es existiert allerdings kaum ein Vorteil, der nicht mit einem Nachteil erkauft wird; so auch dieser. Ein Problem kann ein erhöhter Speicherbedarf darstellen. Es kann erforderlich sein die Ergebnisse einzelner Phasen komplett im Speicher zu halten, da nicht grundsätzlich gewährleistet werden kann, daß die folgende Phase die Resultate in exakt der gleichen Reihenfolge benötigt.

Auch das dem Parser zurgrundeliegende Sprachkonzept wirkt sich zwangsläufig auf die Pass-Verteilung aus. So existieren klassische 2-Pass- und 1-Pass-Sprachen. PL/I und Algol beispielsweise erlauben es Bezeichner (Variablen, Unterprogramme) vor ihrer Deklaration zu verwenden. Hier ist es erforderlich zunächst in einem Lauf die Bezeichner in die Symboltabelle einzutragen, um im zweiten Pass bestimmen zu können, ob sich die grammatikalischen Konstrukte syntaktisch und semantisch sinnvoll sind und daraus Code generiert werden kann.

Hier ein veranschaulichendes Fragment eines PL/I-Programms:

```
PROC_A: PROCEDURE;
       ...
       PROC_B();
       ...
END PROC_A;

PROC_B: PROCEDURE;
       ...
END PROC_B;
```

Obwohl `PROC_B` erst nach `PROC_A` definiert wird, kann es schon in `PROC_A` verwendet werden.

Eine klassische für einen Pass ausgerichtete Sprache ist Pascal. Hier müssen alle Bezeichner zuvor eindeutig deklariert worden sein bevor sie verwendet werden können. Dies machte es auch erforderlich in Pascal das Schlüsselwort forward einzuführen. Rufen sich nämlich zwei Unterprogramme gegenseitig rekursiv auf, so muß mindestens das eine durch einen Prototypen per forward deklariert worden sein, da sonst ein Fehler aufgrund eines nicht definierten Bezeichners auftritt.

Der folgende Quelltext veranschaulicht dies:

```
procedure B(x: integer); forward;

procedure A(x: integer);
begin
       ...
       B(x);  { B wäre ohne forward-Deklaration hier }
              { unbekannt und würde einen Fehler }
              { verursachen. }
       ...
end;

procedure B(x: integer);
begin
       ...
```

```
        A(x);  { A ist infolge der vorherigen Deklaration }
               { und Implementation ohnehin bekannt. }
        ...
end;
```

Eine Sprache wie Pascal ist daher für die Zusammenfassung der Analyse-Phasen in einen Lauf geradezu prädestiniert.

Ein ähnliches Problem ergibt sich bei dem noch aus dem Bereich der konventionellen Programmierung der 50er und 60er Jahre stammenden Goto-Befehl. Sprungziele können aufgrund ihrer Bezeichner häufig nicht sofort erfaßt werden. Befindet sich das Sprungziel erst weiter unten im Quelltext, so ist dessen Adresse nicht bekannt.

Der semantische Aspekt ist bei Sprungzielen weniger komplex, als bei Variablen oder gar Unterprogrammen. Bei Sprungzielen existiert nur ein Typ, nämlich eines Bezeichners für eine eindeutige Adresse im Quelltext bzw. im Gültigkeitsbereich. Variablen- Routinenbezeichner hingegen können verschiedene Typen besitzen. In puncto Unterprogramme ist die Komplexität aufgrund der vielfältigen Formalparameterkonstellationen sogar noch größer.

Aufgrund dieser geringeren semantischen Komplexität bei Sprungzielen bzw. deren Bezeichner ist die Realisierung in 1-Pass-Parsern dennoch möglich. Trifft der Analysierer auf einen Sprungbefehl (goto xy), so kann er im Falle eines in der Symboltabelle noch nicht definierten Bezeichners einen Platzhalter, also Leereintrag, im Code definieren. Kommt er schließlich zur Stelle, an der die Sprungmarke definiert wird, so trägt er die Adresse anstatt des/der Platzhalter(s) ein.

2.4.3 Organisation in Front-End und Back-End

Es ist mittlerweile auch Standard im Compilerbau das System - zumeist ein Compiler - in der Praxis in einen vorderen Teil, das *Front-End*, und einen hinteren Teil, das *Back-End*, aufzuteilen.

Das Front-End kapselt im wesentlichen alle sprachabhängigen Parts, wohingegen das Back-End die maschinenabhängigen Teile beinhaltet. Die Schnittstelle schafft ein vom Front-End generierter und vom Back-End verarbeiteter Zwischencode.

Der Vorteil liegt auf der Hand. Durch diese Verfahrensweise ist es möglich einen Compiler für mehrere Hardware-Plattformen zu implementieren. Es ist hier lediglich nötig für ein Front-End ein dem jeweiligen Zielsystem angepaßtes Back-End zu programmieren.

Ein weiterer interessanter Aspekt, der aus dieser Praxis hervorgeht, ist die Möglichkeit für ein Back-End (und damit für ein bestimmtes Hardware-System) mehrere Front-End für verschiedene Programmiersprachen zu realisieren.

Ein Projekt, das sehr erfolgreich mit diesem Konzept ist, ist der gcc - GNU-C-Compiler. Für diesen Freeware-Compiler existieren mittlerweile mehrere Front-Ends, unter anderem für C, C++, Objective-C, Pascal, FORTRAN und Ada, sowie eine Fülle von Back-Ends, z. B. für 32-Bit-DOS, Win32, OS/2, AmigaOS, TOS, Linux, Solaris und andere Unix-Derivate.

Die Erwähnung des „Front-End-Back-End-Konzeptes“ sollte hier nur der Vollständigkeit im Zuge der Organisation von Phasen erfolgen. Da es sich hierbei um konkrete Praktiken des Compilerbaus handelt, soll im folgenden auf diesen Aspekt nur noch am Rande eingegangen werden. Eine ausführlichere Betrachtung kann [2] entnommen werden.

2.5 Das Parsing

Der Vorteil an kontextfreien Grammatiken ist, daß effiziente und einfach zu implementierende Parsing-Algorithmen existieren. Für den Aufbau von Parsern kommen theoretisch drei Verfahren infrage:

1. Universelle Methoden
2. Top-Down-Methoden
3. Bottom-Up-Methoden

Die universellen Methoden erlauben es für jede Form von kontextfreien Sprachen Parser zu konstruieren. Die Sprache muß allerdings durch eine kontextfreie Grammatik gemäß der *Chomsky-Normal-Form* gegeben sein.

Eine kontextfreie Grammatik ist eine Chomsky-Normal-Form, sofern Ihre Produktionen der Menge P folgende Form haben:

```
A --> BC, A --> a und S --> ε,
mit B, C ∈ V und a ∈ T.
Wenn S --> ε ∈ P, dann darf S nicht in der Ableitung einer
anderen Produktion vorkommen.
```

Eine Untersuchung des Zeitbedarfs ergibt für universelle Parser eine $O(n^3)$ für die Analyse eines Symbols aus n Zeichen. Eine kubische Ordnung ist jedoch nicht gerade effektiv. Deshalb nimmt man in der Praxis vom universellen Konzept Abstand und baut spezialisiertere, leicht analysierbare Grammatiken auf. Es ist praktisch für jede Programmier- oder Kommandosprache möglich einen im wesentlichen auf einem linearen Algorithmus basierenden Parser zu konstruieren.

Dies wird durch das Funktionsprinzip der meisten Parser ermöglicht. Parser für Programmiersprachen arbeiten überwiegend die Symbolreihenfolge von links nach rechts ab, wobei das nächste Symbol im Zuge eines Lookaheads bereits vor dem

eigentlichen Scannvorgang betrachtet wird. Lineare Größenordnungen sind für diese Parser aufgrund ihrer Eigenschaften keine Utopie, sondern eher die Regel.

Getreu dem Auspruch „nomen est omen“ spiegeln die Benennungen der beiden anderen Parsing-Methoden die Richtung wieder, in der die Knoten des Parse-Baumes konstruiert werden. Top-Down-Parser arbeiten sich von der Wurzel zu den Blättern vor; Bottom-Up-Systeme genau entgegengesetzt.

Top-Down- und Bottom-Up-Parser arbeiten wie die universellen Parser ebenfalls auf der Basis von kontextfreien Grammatiken mit spezifischen Eigenschaften. Sie sind jedoch nicht die Chomsky-Normal-Formen, deren praktische Ineffizienz sich in kubischer Ordnung niederschlägt.

Die Grammatiken für Top-Down-Parser dürfen keine linksrekursive Nichtterminale enthalten. Bottom-Up-Parser benötigen hingegen kontextfreie Grammatiken ohne ε-Produktionen und frei von Zykeln. Dem Designer von Parsern kommt aber eine Gesetzmäßigkeit kontextfreier Grammatiken entgegen: Jede kontextfreie Grammatik läßt sich in eine Äquivalente mit den gewünschten Eigenschaften umformen.

Das Top-Down-Prinzip ist hervorragend dafür geeignet effiziente Parser „von Hand“ zu implementieren. Bottom-Up-Verfahren lassen hingegen ein breiteres Spektrum an Grammatiken und Übersetzungsschemata zu. Bottom-Up-Methoden werden daher meist von Software-Tools zur automatischen Parser-Konstruktion verwendet.

2.6 LL(k)- und LR(k)-Parser

Die wichtigsten Teilklassen der Top-Down- beziehungsweise Bottom-Up-Parser sind die *LL(k)*- respektive *LR(k)-Parser*. LL(k) bedeutet hierbei: „Lesen der Eingabe von links nach rechts, produzieren einer Linksableitung und betrachten von k Symbolen als Lookahead auf die Eingabe.“ LR(k) arbeitet ähnlich, produziert allerdings eine Rechtsableitung.

Eine *Linksableitung* (bzw. *Rechtsableitung*) liegt dann vor, wenn durch die Produktionen einer Grammatik stets das am weitesten links (rechts) stehende Nichtterminal als erstes ersetzt wird. Für die Relation r_l für die Linksableitung bzw. r_r für die Rechtsableitung einer Produktion A --> ω bedeutet dies formal:

$$r_l(G) = \{ (xA\beta, x\omega\beta) \mid x \in T^*, \beta \in V^* \}$$
$$r_r(G) = \{ (\alpha Ay, \alpha\omega y) \mid \alpha \in V^*, y \in T^* \}$$

Wenn Sie sich diese Prinzipien anhand der Abbildungen 2.3 und 2.4 nachvollziehen, stellen Sie fest, daß aufgrund des Scanvorgangs von links nach rechts ein LL(k)-Parser zwangsläufig auf eine Top-Down-Analyse respektive das LR(k)-Pendant auf die Bottom-Up-Methode hinausläuft.

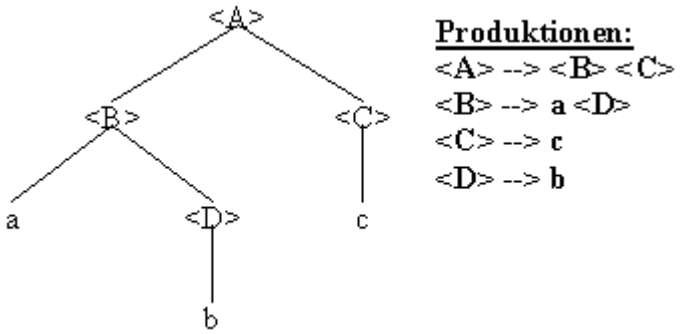


Abb. 2.3: Ableitungsbaum eines LL(k)-Analyse

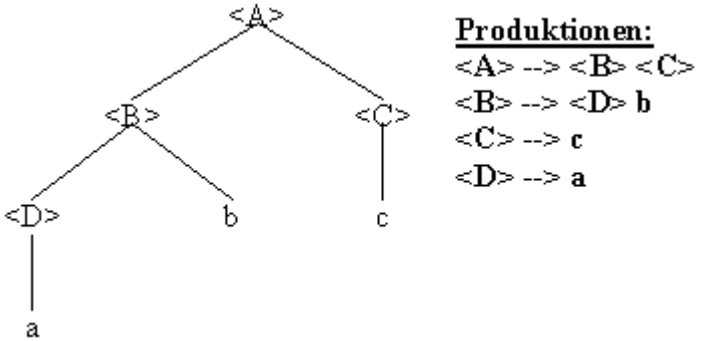


Abb. 2.4: Ableitungsbaum einer LR(k)-Analyse

Bei einer LL(k)-Analyse gemäß Abbildung 2.3 für die Eingabe `abc` kann das Symbol im allgemeinen sofort nach der Lesen in eine Produktion eingeordnet werden. Der Grund hierfür ist die gleiche Richtung bei Scanvorgang (von links nach rechts) und Ableitung (links). Betrachten Sie hierzu die Produktion `<B> --> a <D>`. Das Eingabesymbol und Terminal `a` kann sofort nach dem Lesen eingeordnet werden; die Produktion für `<D>` ist hierfür nicht relevant. Dieses direkt einordnende und damit immer tiefer absteigende Prinzip ist exakt das Top-Down-Verfahren.

Das LR(k)-Parsing verhält sich hier komplizierter. Aufgrund der rechtsableitenden Produktion `<B> --> <D> b` muß `<D>` wegen des Scannens von links nach rechts vor dem Ableiten von `b` betrachtet werden, da das Sybol `a` vor `b` gelesen wird. Das Eingabesymbol `a` muß sozusagen zurückgestellt werden und `b` betrachtet werden. Stimmt `b` kann `a` auf die Ebene `<D>` angehoben und zusammen mit `b` auf `<B>` reduziert und weiter angehoben werden.

Sie sehen schon aufgrund der verwendeten Verben „anheben“ und „reduzieren“, daß hier von unten nach oben gearbeitet wird. Ergo: Bottom-Up.

Unter dem *Lookahead* versteht man das Betrachten von Eingabesymbolen während des Scannings. Der Wert k der Links- und Rechts-Parser ist dabei ein zwangsläufig

ganzahliger und positiver Wert. Es ist dabei auch möglich diesen Wert auf 0 zu setzen. LL(0)- und LR(0)-Grammatiken sind jedoch für die Beschreibung von komplexeren Sprachen, wie sie im praktischen Einsatz grundsätzlich vorkommen, zu unelegant und uneffizient.

LL(k)-Parser können wie alle Top-Down-Analysierer relativ leicht durch deterministische Algorithmen beschrieben werden. Bei Bottom-Up-Systemen im allgemeinen und LR(k)-Parsern im speziellen sind schon tiefere Griffe in die Trickkiste der Informatik nötig, da es sich um nichtdeterministisches Konzept handelt. Hier ist der Einsatz automatentheoretischer Mittel (Kellerautomaten) unerläßlich.

Damit wäre auch die Behauptung von oben bestätigt, daß sich das Top-Down-Konzept gut für „handgestrickte“ Parser, das Bottom-Up-Verfahren eher für Parser-Generatoren eignet.

2.7 Ein prädiktiver Parser

Nach den bislang theoretischen Grundlagen geht es nun in medias res. Im folgenden wollen wir einen simplen Top-Down-Parser aus einer LL(1)-Grammatik implementieren. Es soll ein prädiktiver Parser sein, der einfache arithmetische Terme korrekt löst.

Die Basis der *prädiktiven Syntaxanalyse* ist der *rekursive Abstieg* (*recursive descent*). Hierbei wird jedem nichtterminalen Symbol der Grammatik eine Prozedur zugeordnet, die die entsprechenden Terminale der jeweiligen Produktion erkennt. Nichtterminale in der Ableitung werden auf Aufrufe der korrespondierenden Prozeduren abgebildet. Das Lookahead-Symbol bestimmt bei der prädiktiven Syntaxanalyse für jedes Nichtterminal eindeutig die auszuwählende Prozedur (prädiktiv = vorhersagend).

Der rekursive Abstieg besitzt einige Vorteile (nach [16]):

- Die Prozeduren ergeben sich direkt auf der Grammatik.
- Die Korrektheit des Verfahrens ist intuitiv klar.
- Die Implementierung ist einfach.
- Das Verfahren arbeitet relativ schnell.

2.7.1 Die Grammatik für den Parser

Als Basis verwenden wir für unseren Parser eine leicht modifizierte Variante der Grammatik von oben. Die EBNF erweitern wir hier um ein Konstrukt zur Vereinfachung der Notierung von umfangreicheren Mengen alternativer Symbole. {s1..sn} soll die kürzere Schreibweise für die Menge aller alternativen Symbole von s1 bis sn sein.

{"0".."9"} soll also dem Standardkonstrukt {"0","1","2","3","4","5","6","7","8","9"} entsprechen.

```
<list> ::= <expr> [{"+","-"} <list>]
<expr> ::= ( "(" <list> ")" | <num> ) [{"*","/"} <expr>]
<num>  ::= {"+","-"} [{"0".."9"}+] ["." [{"0".."9"}+]]
           [{"E","e"} {"+","-"} {"0".."9"}+]
```

Wie Sie sehen sind die Zahlen in unserer Grammatik nun auch (korrekterweise) durch ein nichtterminales Symbol und eine zugehörige Produktion auf Terminale realisiert. Das Format der Zahlen orientiert sich hierbei an den Fließkommakonstanten in C++.

2.7.2 Umsetzung der Grammatik in Programmcode

Da in den Neuzigern nicht mehr strukturiert, sondern objektorientiert programmiert wird, soll auch unser Parser in einer Objektklasse realisiert werden. Jedes der drei Nichtterminale ist demzufolge als Methode zu implementieren. Die nichtterminalen Symbole in den Ableitungen werden in die Rechenoperationen einbezogen. Die entsprechenden Methoden müssen demzufolge ein Ergebnis zurückliefern. Da sich für die Darstellung der Fließkommazahlen in C++ der Typ `double` aufgrund seiner hohen Genauigkeit anbietet, werden die Methoden diesen Typ zurückliefern.

In einem Parser ist leider nicht alles an den Idealzustand der Grammatik gebunden. Fehleingaben vonseiten des Benutzers müssen erkannt und behandelt werden. In unserem kleinen Beispielparser werden wir daher eine Fehlerbehandlung einführen. Dieses Error-Handling ist schlicht durch eine Methode realisiert, die beim Erkennen eines Fehlers aufgerufen wird und das Programm kurzerhand beendet. Zugegeben ist diese Fehlerbehandlung nur rudimentär und wäre in einem komplexeren Parser unzureichend, soll uns aber vorerst genügen.

Nach dem Hinzufügen einer öffentlichen Methode zum Starten des Rechners und eines Puffers, sowie einer Variablen, die den aktuellen Lookahead beschreibt, erhalten wir folgende Klassendeklaration:

```
class Calculator
{
      public:
            void Run();

      protected:
            void error();

            double list();
            double expr();
            double num();
```

```
        unsigned pos;
        char buffer[1000];
};
```

Auf einen expliziten Konstruktor kann bei dem kleinen Parser verzichtet werden. Ebenfalls auf den Destruktor.

Sie sehen, daß die Implementierung des Parsers bzw. die Umsetzung der Grammatik nicht öffentlich zugänglich ist. Lediglich durch die Methode `Run()` kann das System in Gang gesetzt werden. Dies ist unbedingt erforderlich, da nur so vermieden werden kann, daß in den Parser „quereingestiegen“ werden kann. Nur so ist immer ein klar definierter Zustand des Parsers garantiert.

Nicht einmal die Methode des Startsymbols darf öffentlich zugänglich sein. Etwaige Rekursionen und Zyklen verhindern nämlich hier das Initialisieren des Parsers, z. B. hier das Setzen von `pos` auf Null. Würde beispielsweise in `list() pos` immer wieder auf Null gesetzt, würde dies unweigerlich bei geklammerten Ausdrücken, aber auch bei Additionen respektive Subtraktionen, zu einer unterminierten Rekusion führen. Nach dem Scan von „(“, „+“ und „-“ würde der Lookahead immer wieder beim Eintritt in `list()` auf den Eingabeanfang zurückspringen und der Parser erneut mit der Analyse beginnen.

Die grundlegende Implementation des Parsers ist anhand der Grammatik nicht schwer. Gemäß dem Prinzip des rekursiven Abstiegs sind in den Methoden zu den Nichterminalen lediglich die Ableitungen zu Codieren:

- Sämtliche Nichtterminale werden zu Methoden-Calls und
- die Terminale und die Rückgabewerte der Methoden werden entsprechend interpretiert, also in unserem Fall arithmetisch laut den grammatikalischen Regeln verknüpft.

Die Methoden zeigen sich bei purer Betrachtung der Syntax, also ohne jegliche Interpretierung, wie folgt:

```
double Calculator::list()
{
        expr();

        if(buffer[pos] == '+' || buffer[pos] == '-') {
                pos++;
                list();
        }
}
```

```
double Calculator::expr()
{
	if(buffer[pos] == '(') {
		pos++;
		list();
		pos++; // die schließende Klammer überspringen
	} else
		num();

	if(buffer[pos] == '*' || buffer[pos] == '/') {
		pos++;
		expr();
	}
}

double Calculator::num()
{
}
```

Falls Sie sich wundern, daß `num()` keinen Code enthält, dann ist Ihre Skepsis berechtigt. `num()` müßte Code für die Terminale enthalten! Wir werden uns jedoch einer eleganten Möglichkeit bedienen die Analyse der Terminale (Fließkommazahlen in C++-Notation) an eine Funktion aus der C-Standardbibliothek zu delegieren.

Eine gute Hilfe bei der Umsetzung von Grammatiken in einen prädiktiven Parser ist, daß die Struktur sich nicht nur in Form der Methoden-Nichtterminal-Äquivalenz ausdrückt, sondern auch innerhalb der Methoden exakt wiedergespiegelt wird. So können Sie „Leichtsinnsfehler“ recht schnell finden, wenn Sie die Blöcke in der Grammatik mit denen der Implementierung vergleichen.

Alle in der EBNF durch runde - und eckige Klammern gekapselten Blöcke finden sich auch im Quelltext als Blöcke wieder. Betrachten wir hierzu `<list>` und `list()`.

Die Ableitung von `<list>` enthält zwei Blöcke, nämlich `<expr>` und `[{"+","-"} <list>]`. Die Methode `list()` auch:

```
double Calculator::list()
{
	/*****ANFANG BLOCK 1*****/
	expr();
	/******ENDE BLOCK 1******/

	/*****ANFANG BLOCK 2*****/
	if(buffer[pos] == '+' || buffer[pos] == '-') {
		pos++;
```

```
                list();
        }
        /******ENDE BLOCK 2******/
}
```

Alle in eckige Klammern gefaßten Blöcke der Grammatik müssen durch einen optionalen Block und alle in geschweiften Klammern durch alternative Blöcke im Algorithmus repräsentiert sein. Innerhalb der Blöcke setzt sich diese Struktur konsequent fort.

Daraus resultieren nun zwei Grundsätze:

1. Sie können implementierungstechnische Fehler anhand des Vergleichs der Struktur, die im wesentlichen 1:1 vorliegt, zwischen Grammatik und Quelltext leicht finden.
2. Der Erfolg der Implementierung ist von der Qualität der Grammatik direkt abhängig. Mit anderen Worten: Haben Sie die Grammatik nicht vollends durchdacht, übertragen Sie sämtliche logischen Fehler auf den Parser!

Sie sehen die Grammatik ist und bleibt der zentrale Punkt in der Parsing-Theorie. Der Erfolg Ihres Parsers steht und fällt unweigerlich mit Ihrem Grammatikentwurf.

2.7.3 Vollendung des Parsers

Nach dem Hinzufügen der Fehlerbehandlung und des Codes zur Verarbeitung der Terminale und der Berechnungsoperationen ergibt sich folgendes Bild:

```
void Calculator::error()
{
        cerr << "Syntaxfehler bei Position "
             << pos + 1 << endl;
        exit(1);
}

double Calculator::list()
{
        double a = expr(), b;

        if(buffer[pos] == '+') {
                pos++;
                b = list();
                return a + b;
        } else if(buffer[pos] == '-') {
                pos++;
```

```
                b = list();
                return a - b;
        } else if(buffer[pos] && buffer[pos] != ')')
                error();

        return a;
}

double Calculator::expr()
{
        double a, b;

        if(buffer[pos] == '(') {
                pos++;
                a = list();
                if(buffer[pos] != ')')
                        error();
                pos++;
        } else
                a = num();

        if(buffer[pos] == '*') {
                pos++;
                b = expr();
                return a * b;
        } else if(buffer[pos] == '/') {
                pos++;
                b = expr();
                return a / b;
        }

        return a;
}

double Calculator::num()
{
        char *endptr;
        double a = strtod(&buffer[pos], &endptr);
        unsigned len = (unsigned) (endptr - &buffer[pos]);

        if(!len)
                error();
```

```
        pos += len;

        return a;
    }
```

Die Ergebnisse der Methoden werden nun in `list()` und `expr()` arithmetisch gemäß den Produktionen verknüpft. Tritt jeweils ein unerwartetes Eingabesymbol auf, so wird die Fehlerbehandlung aktiviert.

Der Test in `list()` auf eine schließende Klammer als Lookahead-Symbol vor der Aktivierung des Error-Handlings ist nötig, da `list()` laut Grammatik auch eingefaßt in Klammern vorliegen kann. Ist dies der Fall, so kann als nächstes Eingabesymbol auch die schließende Klammer vorliegen.

Die obige Implementation läßt allerdings noch zu wünschen übrig, da auch ein Ausdruck wie `5+5)` akzeptiert und interpretiert wird. Doch die Lösung dieses Problem wollen wir zunächst noch verschieben.

Die Analyse der Zahlen übernimmt in `num()` die Funktion `strtod()`, die in `stdlib.h` deklariert ist. Durch den Zeiger `endptr` und die aktuelle Position des Lookahead kann die Länge des Zahlausdrucks bestimmt werden und der Lookahead neu bestimmt werden.

Sollte `len` gleich Null sein (`!len`), so konnte `strtod()` keine Fließkommazahl aus dem Eingabepuffer lesen. In diesem Fall liegt ein Syntaxfehler vor, da laut Grammatik eine solche Fließkommazahl erwartet wird. Die Fehlerbehandlung wird aktiviert.

Zum Starten des Parsers existiert die öffentliche Methode `Run()`. Die zuvor implementierten Routinen opertieren immer über einem String, der exakt einen arithmetischen Term beschreibt. Die Aufgabe von `Run()` ist es nun vom Standardeingabekanal immer einen solchen Term, das heißt konkret einen String, zu lesen, dem zuvor initialisierten Parser zu übergeben und das Ergebnis dieser Operation auf dem Standardausgabekanal auszugeben. Dieser Vorgang wird so oft wiederholt bis das Dateiendezeichen gelesen wird oder ein Syntaxfehler auftritt.

Für `Run()` ergibt sich demnach folgender Aufbau:

```
void Calculator::Run()
{
        while(!cin.eof()) {
                cin.getline(buffer, sizeof(buffer));
                pos = 0;
                cout << buffer << " = " << list() << endl;
        }
}
```

`Run()` arbeit also immer eine gesamte Eingabedatei ab, die über die Standardeingabe eingeht.

2.7.4 Verbessung des Parsers

Bevor der Test des Parsers anlaufen kann, muß noch ein Hauptprogramm geschrieben werden:

```
int main()
{
       cout << "Calculator calc1 - "
            << "Copyright (c) 1997 by Oliver Mueller"
            << endl << endl;
       Calculator calculator;
       calculator.Run();
       cout << endl << "Bye!" << endl;
       return 0;
}
```

Nach der Ausgabe einer Copyright-Angabe, folgt der Parsing-Vorgang. Als Abschluß wird noch ein freundliches „Bye!“ den Bildschirm zieren; vorausgesetzt es tritt kein Fehler auf.

Nachdem Start offenbart sich die Korrektheit des Parsers und der Grammatik des kleinen Rechners. Es zeigen sich jedoch auch kleinere Unschönheiten, die entweder der Implementation oder sogar der Grammatik zuzuschreiben sind:

1. Das altbekannte Problem der geschlossenen Klammer:
   `1+1)` wird beispielsweise interpretiert, obwohl es laut Grammatik nicht korrekt ist.
2. Steht das Dateiendezeichen als erstes Zeichen in einer Zeile, also allein in der Zeile, so wird dies als Syntaxfehler gewertet. Als Abschluß eines Terms wird es jedoch akzeptiert:

   ```
   1+1<EOF>
   1+1 = 2


   Bye!
   ```

   im Gegensatz zu

   ```
   1+1
   1+1 = 2
   <EOF>
   Syntaxfehler bei Position 1
   ```

   Hinweis: `<EOF>` ist hier jeweils als das Dateiendezeichen zu werten (eof = end of file). Da nicht alle Systeme, wie DOS, Windows und OS/2, (Strg)+(Z) als Dateiendezeichen definieren (Linux beispielsweise verwendet (Strg)+(D)), soll im folgenden jeweils `<EOF>` dieses Zeichen neutral repräsentieren.

3. Leerzeichen und Tabulatoren innerhalb eines Terms werden als Syntaxfehler gewertet:

```
1   +   1
Syntaxfehler bei Position 2
```

Problem Nr. 1 ist ein nur die Implementierung betreffendes Problem und recht einfach zu beseitigen. Zur Erkennung, ob ein geklammertes `<list>` vorliegt genügt es eine geschütztes Attribut vom Typ `unsigned` einzuführen. Wir wollen es `brackets` nennen.

`brackets` soll mit Null initialisiert werden. Diese Initialisierung findet in `Run()` für jede neue Zeile statt. Nach jedem Lesen einer öffnenden Klammer soll dieses Attribut um eins inkrementiert, nach jeder schließender Klammer wieder um eins reduziert werden. Somit gibt diese Variable die Klammerebene an. Null ist hier der Indikator für keine Klammerebene. In diesem Fall ist eine schließende Klammer ein Syntaxfehler in `list()`. Bei einem Wert ungleich Null ist diese schließende Klammer als korrektes Lookahead zu akzeptieren.

Damit ergibt sich die modifizierten Methoden `list()` und `expr()`:

```
double Calculator::list()
{
	double a = expr(), b;

	if(buffer[pos] == '+') {
		pos++;
		b = list();
		return a + b;
	} else if(buffer[pos] == '-') {
		pos++;
		b = list();
		return a - b;
	} else if(buffer[pos])
		if(buffer[pos] == ')') {
			// Lookahead ist eine schließende
			// Klammer. Falls der Zähler brackets
			// Null ist liegt ein Syntaxfehler vor.
			if(!brackets)
				error();
		} else
			error();

	return a;
}
```

```
double Calculator::expr()
{
	double a, b;

	if(buffer[pos] == '(') {
		pos++;

		// Neue Klammernebene -> brackets erhöhen
		brackets++;

		a = list();
		if(buffer[pos] != ')')
			error();

		// Neue Klammernebene beendet
		//   -> brackets erniedrigen
		brackets--;

		pos++;
	} else
		a = num();

	if(buffer[pos] == '*') {
		pos++;
		b = expr();
		return a * b;
	} else if(buffer[pos] == '/') {
		pos++;
		b = expr();
		return a / b;
	}

	return a;
}
```

Das zweite Problem ist in der Grammatik begründet. Ein <EOF> allein in einer Zeile bedeutet doch nichts anderes als eine Leerzeile. Doch die Grammatik unseres kleinen Rechners ist nicht auf die Verarbeitung eines „leeren Terms“ aufgelegt. Hier muß eine ε-Produktion zum Einsatz kommen. Doch wo soll diese eingebaut werden?

Die Aufnahme als neue Produktion direkt in `<list>` zieht Fehler nach sich:

```
<list> ::= ε | <expr> [{"+","-"} <list>]
```

Da `<list>` eingefaßt in runde Klammern auch in einer Produktion von `<expr>` auftreten kann, wäre auch `()` ein syntaktisch gültiger Ausdruck. Mathematisch ist dies jedoch Unsinn in purster Form.

Doch was soll erreicht werden? Es sollen entweder arithmetische Terme ausgewertet werden können, oder leere Zeilen ignoriert werden können. Was liegt also näher, als ein neues Startsymbol einzuführen, daß entweder auf eine Ableitung `<list>` oder ein ε-Produktion verweist? Damit ergeben sich keine Probleme, da nicht in die Grammatik für die mathematischen Terme eingegriffen wird.

Somit ergibt sich eine neue Produktion für ein nichtterminales Symbol `<entry>`, das zugleich als neues Startsymbol definiert wird:

```
<entry> ::= ε | <list>
```

`<entry>` wird nun nach dem Prinzip der prädiktiven Syntaxanalyse wieder als neue (geschützte) Methode aufgenommen. Der Rückgabewert der Methode wird ebenfalls, wie der der anderen Methoden, als `double` deklariert. Zusätzlich wollen wir jedoch noch einen formalen Parameter deklarieren. Dieser Parameter ist eine Referenz auf eine Boolesche Variable. Dieser Wert gibt nach dem Parsing an, ob ein Term analysiert (`true`) oder eine ε-Produktion erkannt wurde (`false`).

```
inline double Calculator::entry(bool& term)
{
       if(!buffer[0]) {
              term = false;
              return 0.0;
       }
       term = true;
       return list();
}
```

Falls Ihr C++-Compiler noch nicht das aktuelle Draft-Paper von ANSI/ISO ([3]) unterstützen sollte, also die neuen Schlüsselworte `inline` und `bool` nicht akzeptiert werden, können Sie die Include-Datei `ansi.hpp` aus Anhang B verwenden.

Die Deklaration von `entry()` als inline erfolgt als Optimierung. Es dürfte also nicht zur Verwendung als reale Methode kommen. Höchstwahrscheinlich wird `entry()` in den verwendeten Methoden inline-expanded, also als Code eingefügt werden. (Geben Sie hierzu diese Methode vor allen anderen an!)

Das dritte Problem ist ebenfalls leicht zu lösen. Eine kontextfreie Grammatik definiert immer nur Symbole und ihre syntaktischen Beziehungen. Es ist dadurch nicht definiert, ob die terminalen Symbole (=*Tokens*) durch sogenannte *Whitespaces*, also Leerzeichen und Tabulatoren, getrennt sind.

In den meisten Computersprachen ist dies jedoch der Fall. Auch wir wollen unseren Rechner so auslegen, daß er etwaige Whitespaces schlicht als Trenner wertet und ignoriert.

Da in der Grammatik des Rechnerprogramms der Trennung durch derartige Whitespaces keine relevante Bedeutung zukommt, können wir vor dem eigentlichen Parsing-Vorgang alle Leer- und Tabulatorzeichen aus dem Eingabepuffer eliminieren.

Hierzu definieren wir eine neue geschützte Methode:

```
void Calculator::removeWhiteSpaces()
{
	register unsigned i = 0, j = 0;
	while(buffer[i]) {
		if(isspace(buffer[i]))
			i++;
		else if(i != j)
			buffer[j++] = buffer[i++];
		else {
			i++;
			j++;
		}
	}
	buffer[j] = '\0';
}
```

Diese Methode liest die Zeichen des Eingabepuffers von links nach rechts. Dabei wird mit zwei Zeigern gearbeitet. Einer (`i`) gibt die aktuelle Leseposition an. Ein anderer (`j`) die aktuelle Schreibposition.

`removeWhiteSpaces()` arbeitet nach einem Prinzip des Entfernens und Komprimierens. Wird ein Whitespace gelesen, wird zwar der Lesezeiger erhöht, der Schreibzeiger jedoch nicht. Wird ein anderes Zeichen als ein Whitespace gelesen, dann wird dieses an die aktuelle Schreibposition geschrieben und beide Zeiger erhöht.

Aufgrunddessen kommt es zur Entfernung aller Whitespaces und zugleich zur Verdichtung des Strings.

Um die drei Probleme nun endgültig zu lösen, ist die Methode Run() zu modifizieren.

```
void Calculator::Run()
{
	while(!cin.eof()) {
		// Zeile in Eingabepuffer lesen
		cin.getline(buffer, sizeof(buffer));

		// Whitespaces entfernen
```

```
                removeWhiteSpaces();

                // Lookahead initialisieren
                pos = 0;

                // Zähler der Klammernebene initialisieren
                brackets = 0;

                // Indikator für Empty-Produktion deklarieren
                bool calculated;

                // Parsing
                double result = entry(calculated);

                // Wenn keine leere Zeile vorlag,
                // Ergebnis ausgeben.
                if(calculated)
                       cout << buffer << " = "
                            << result << endl;
        }
    }
```

Das Startsymbol ist nun `<entry>`. Demnach wird jetzt als Einstieg in die Syntaxanalyse die Methode `entry()` verwendet. Der Zähler für die Klammernebene wird initialitisiert und die Whitespaces aus dem Eingabepuffer entfernt.

Das Ergebnis der Rechenoperation wird nun nur noch ausgegeben, wenn wirklich ein Term vorlag. Dies wird durch den neu eingeführten Indikator `calculated` möglich, der als Referenzargument beim Aufruf von `entry()` dient.

2.8 Fehlerbehandlung

Die Fehlerbehandung des kleinen Rechnerprogramms ist wie schon erwähnt nur rudimentär und unzulänglich. Von einer Fehlerbehandlung in Parsern im allgemeinen und in Compilern im besonderen wird wesentlich mehr erwartet.

Gerade von Compilern erwartet man keine aus der Anwendungsprogrammierung bekannte Fehlerbehandlung von ein *Error-Recovery*, also eine „Erholung vom Fehler“. Es wird nicht erwartet, daß nach dem ersten Auftreten eines Fehlers die Compilierung abbricht. Vielmehr ist es sinnvoll das System wieder in einen definierten Zustand zu führen, also eventuell zu interpretieren, was gemeint sein könnte. Danach können weitere Fehler gesucht und ebenfalls ausgegeben werden. Der Benutzer kann diesen Fehler in einem Bearbeitungsschritt nachgehen und muß nicht für jeden einzelnen Fehler den Compiler neu starten, um ihn zu finden.

Ihr C++-Compiler wird sich nach diesen Regeln verhalten. Haben Sie mehrere Fehler in einem Quelltext, so werden alle fehlerhaften Zeilen angegeben. Allerdings existieren auch Compiler, denen ein solches Wiederherstellen eines definierten Zustands nach einem Fehler ein Fremdwort ist. Borlands Turbo Pascal und Delphi 1 brachen die Compilierung jeweils nach dem ersten aufgetretenen Fehler ab. Erst mit der Einführung von Delphi 2 wurde dieses Manko behoben.

Die Intelligenz moderner Compiler zeigt sich sogar darin, daß nicht nur Fehler gefunden werden, sondern auch Hinweise und Warnungen auf eventuelle Fehler gegeben werden.

Ein derartig komplexes Recovery ist im Falle des kleinen Rechners noch nicht nötig, da jede Zeile ein insich geschlossenes und sehr kurzes Programm darstellt und nicht mit den anderen Zeilen in Kontext steht.

Es ist aber sicherlich kein Fehler genauere Hinweise auf die Fehlerart zu geben und den Rechner nicht nach jedem Fehler zu beenden.

2.8.1 Fehlerbehandlung durch C++-Exceptions

Für die Fehlerbehandlung wollen wir den Exception-Mechanismus von C++ verwenden. Alternativ könnten hier auch die strukturierten Exceptions aus C oder auch die Routinen der ebenfalls aus C stammenden Library `<setjmp.h>` Anwendung finden. Kernighan und Pike setzen beispielsweise für hoc (=high order calculator) auf `<setjmp.h>` ([8]).

Es wird ein Mechanismus benötigt, der das System in einen zuvor eindeutig definierten Zustand rückführen kann. Welches der in C++ angebotenen Konzepte konkret verwendet wird hängt im wesentlichen vom angewendeten Paradigma und dem Geschmack des Entwicklers ab.

Innerhalb des kleinen Rechners können zwei Arten von Fehlern auftreten. Die erste Art ist eine „Division durch Null“, die allerdings nicht den Parser betrifft, sondern die Ausführung. Parser und Ausführung sind in diesem Minimal-Interpreter jedoch in einander verschmolzen. Die zweite Fehlergattung ist der altbekannte „Syntaxfehler“, der eindeutig im Parser auftritt.

Die Fehlerbehandlung soll so organisiert sein, daß im Falle eines Fehlers eine passende Exception geworfen wird. Im Falle eines „Division durch Null“-Fehlers, der nur bei einer Produktion des Nichtterminals `<expr>` auftreten kann, soll schlicht eine entsprechende Exception ausgelöst wird. Bei Syntaxfehlern, die in unserem Parser wesentlich häufiger auftreten, wird nach wie vor `error()` aufgerufen. `error()` instanziiert dann die entsprechende Exception und wirft sie.

`error()` soll im folgenden auch genauere Auskunft über die Ursache des Fehlers erhalten. Diese Information wird `error()` an das Exception-Objekt weitergeben.

Hierzu ist nicht nur das Modifizieren der Konstruktion dieser Methode erforderlich, sondern auch das Verändern der Spezifikation.

Die zentrale Auswertungsstelle der Fehlerbehandlung bzw. der Ort, an den alle Fehler zurückfallen, soll `Run()` sein. Diese Methode wird die auftretenden Exception zu den oben beschriebenen Fehlerarten fangen, auswerten und das System in einen definierten Zustand rückführen.

Die Wahl von `Run()` für diese Aufgabe ist deshalb sinnvoll, da hier ein fester Zustand existiert respektive dieser leicht hergestellt werden kann. Ein Term, der einen Fehler verursachte, wird schlicht mit einer Fehlermeldung beantwortet und anschließend irgnoriert. Danach befindet sich der Parser wieder in einem definierten Zustand und kann den nächsten Term analysieren. Fehlerbehaftete Eingaben können hier leicht gänzlich ignoriert werden, da das Zurückfinden in einen definierten Zustand innerhalb eines fehlerhaften Terms wohl kaum sinnvoll ist.

Die Implementation dieser Fehlerbehandlung stellt einen tiefen Eingriff in das Parser-System dar. `error()`, `Run()` und die prädiktiven Methoden, die die Nichtterminale repräsentieren, müssen modifiziert werden. Desweiteren sind noch entsprechende Klassen für die Exceptions zu konstruieren.

Doch zunächst zu den Exception-Klassen. Wir wollen diesen Klassen eine einheitliche Schnittstelle geben. Diese Schnittstelle kann idealerweise in Form einer abstrakten Basisklasse, die als Elternklasse für die konkreten Exception-Klassen dienen soll, definiert werden. Diese abstrakte Klasse deklariert lediglich zwei virtuelle Methoden:

- `why()` gibt Auskunft darüber, warum diese Exception auftrat. Es wird schlicht ein String mit der Fehlermeldung zurückgegeben.
- `raise()` löst die Exception aus. Diese einheitliche Methode ist auch eine komfortable Möglichkeit diese Exception weiterzureichen.

Damit ergibt sich folgende abstrakte Klasse:

```
class xException
{
      public:
            virtual const char *why() = 0;
            virtual void raise() = 0;
};
```

Die Klasse für die Exception „Division durch Null“ ist recht einfach gestrickt. Sie implementiert lediglich die beiden oben genannten Methoden der Elternklasse.

```
class xDivZero : public xException
{
      public:
```

```
                const char *why();
                void raise();
};

const char *xDivZero::why()
{
        return "Division durch Null.";
}

void xDivZero::raise()
{
        throw *this;
}
```

Diese beiden Methoden sind so einfach gestaltet, daß sie keiner weiteren Erläuterung bedürfen.

Die Klasse für Syntaxfehler hat dagegen schon eine etwas kompliziertere Erscheinungsform. `why()` soll drei Informationen ausgeben:

1. Die Fehlerposition als Zahlenwert.
2. Die Fehlerart.
3. Die Fehlerposition quasi-grafisch aufbereitet.

Im praktischen Einsatz soll dies beispielsweise für die fehlerhafte Eingabe `1+1)` folgende Ausgabe liefern:

```
Syntaxfehler bei Position 4: Unerwartete ).
1+1)
   ^
```

Der erste Part der ersten Zeile enthält die bekannte Ausgabe der Fehlerposition als Zahlenwert wie sie schon in der rudimentären Implementation vorhanden war. Der zweite Teil enthält jetzt eine genauere Spezifikation des aufgetretenen Fehlers, die dem Benutzer beim Fehlerbeheben helfen soll.

Die zweite und dritte Zeile stellt die „quasi-grafische“ Aufbereitung der Fehlerposition dar. In der zweiten Zeile wird der Ausdruck - im übrigen befreit von Whitespaces - ausgegeben. Das Circumflex in der dritten Zeile zeigt auf das Zeichen, das den Syntaxfehler hervorrief.

An das Exception-Objekt müssen somit drei Informationen vom Parser übergeben werden:

- Die Position innerhalb des Eingabepuffers, an der der Syntaxfehler auftrat.

- Die Fehlerart bzw. die Fehlermeldung.
- Der Inhalt des Eingabepuffers.

Diese Informationen werden bei der Instanziierung der Exception per Konstruktor übergeben. Innerhalb des Objektes sind diese Informationen geschützt vor äußerem Zugriff zu speichern. Damit ergibt sich folgender Klassenaufbau:

```
class xSyntax : public xException
{
      public:
            xSyntax(unsigned pos, const char *errmsg,
                    const char *term);

            const char *why();
            void raise();

      protected:
            unsigned err_pos;    // Position des Fehlers
            char msg[2000],      // Medlungspuffer
                 err_msg[100],   // Fehlerart
                 err_term[1000]; // Fehlerhafter Term
};
```

Für den gegen etwaige Nullzeiger abgesichterten Konstruktor ergibt sich ein recht simpler Aufbau:

```
xSyntax::xSyntax(unsigned pos, const char *errmsg, const
char *term)
{
      err_pos = pos;

      if(errmsg)
            strcpy(err_msg, errmsg);
      else
            strcpy(err_msg, "Unbekannter Fehler.");

      if(term)
            strcpy(err_term, term);
      else
            err_term[0] = '\0';
}
```

Ebenso für `raise()`:

```
void xSyntax::raise()
```

```
{
        throw *this;
}
```

Die Methode `why()` beinhaltet den wahren Kern der Exception:

```
const char *xSyntax::why()
{
        char num[10];
        itoa(err_pos, num, 10);
        strcpy(msg, "Syntaxfehler bei Position ");
        strcat(msg, num);
        strcat(msg, ": ");
        strcat(msg, err_msg);
        if(err_term[0]) {
                strcat(msg, "\n");
                strcat(msg, err_term);
                strcat(msg, "\n");
                register unsigned n, m;
                m = strlen(msg);
                for(n = 0; n < err_pos-1; n++, m++)
                        msg[m] = ' ';
                msg[m] = '^';
                msg[m+1] = '\0';
        }
        return msg;
}
```

Die Methode operiert über dem Attribut `msg`, welches einen Puffer für die Nachricht darstellt. In diesem Puffer `msg` wird die dreizeiligen Fehlermeldung gemäß der oben genannten Form sukzessive aufgebaut.

Hinweis: `msg` ist als Attribut der Klasse `xSyntax` deklariert, da eine Realisierung als lokale Variable in `why()` unangenehme Nebeneffekte erzeugen kann. `msg` wäre nicht persistent, das heißt in diesem Fall es würde den Lebenszyklus der Methode `why()` nicht überdauern. Wenn im folgenden beim Fangen der Exception auf den von `why()` zurückgelieferten Wert zugegriffen wird, ist zwar ein Zeiger auf das String-Objekt `msg` vorhanden und höchstwahrscheinlich wäre auch die Werte in `msg` unverändert, aber der Speicherbereich vom `msg` könnte vom System schon erneut verwendet werden. Bei unglücklichen Konstellationen kann dies zu unerfreulichen Nebeneffekten führen. Alternativ könnte `msg` in `why()` als static deklariert werden.

Sämtliche auftretende Fehler sollen zu `Run()` zurückführen, um dort das Error-Recovery durchzuführen. Hierzu ist der Eintritt in den Parser in einen Try-Block zu fassen und ein entsprechendes Catch für das Fangen der Exceptions anzubringen.

```
void Calculator::Run()
{
       while(!cin.eof()) {
              cin.getline(buffer, sizeof(buffer));
              removeWhiteSpaces();
              pos = 0;
              brackets = 0;
              try {
                     bool calculated;
                     double result = entry(calculated);
                     if(calculated)
                            cout << buffer << " = "
                                 << result << endl;
              }
              catch(xException& X) {
                     cerr << X.why() << endl;
              }
       }
}
```

Wie Sie sehen ist hier nur ein Catch angegeben, obwohl zwei Exceptions - `xDivZero` und `xSyntax` - erwartet werden müssen. Hier rückt die durch die abstrakte Basisklasse eingeführte einheitliche Schnittstelle in den Mittelpunkt des Interesses. Beide Exception-Klassen `xDivZero` und `xSyntax` stammen beide von der Klasse `xException` ab. Durch die virtuellen Methodendeklarationen in der Basisklasse wird hier jeweils beim Senden einer Nachricht an das Objekt auf die entsprechende Methode der Kindklasse zugegriffen - egal, ob das Objekt als Referenz der abstrakten - oder der konkreten Klasse aufgefaßt wird.

Daraus folgt nun, daß die Instanzen der beiden Exceptions `xDivZero` und `xSyntax` vereinheitlich als Instanzen der abstrakten `xException` angesehen werden können. Wird an diese „vereinheitlichten“ Objekte eine Nachricht gesendet, so kommt es dennoch zum korrekten und konkreten Methoden-Call der Kindklasse.

`X.why()` im Catch-Block wird also dem Kontext entsprechend ausgewertet. Dieses C++-Statement kann sich demnach auf eine Instanz von `xDivZero` ebenso beziehen wie auf ein Objekt von `xSyntax`.

Der Catch-Block befindet sich innerhalb der While-Schleife und führt die beiden erwarteten Exceptions elegant an das Ende der selben. Dadurch kann die Schleife nach

einem aufgetretenen Fehler und dessen Behandlung an den Anfang zurückgeführt werden und dort die nächste Eingabe (oder EOF) lesen.

Nach einer Fehlerbehandlung herrschen somit die gleichen Bedingungen wie bei einer fehlerfreien Analyse. Das Programm kehrt also in einen definierten Zustand zurück, von dem aus es fortgesetzt werden kann (=Error-Recovery).

2.8.2 Implementation der Fehlerbehandlung in den Parser

Zuvor wurde schon erwähnt, daß in und an `error()` einige Umformungen anstehen. Diese Methode zum Melden von Syntaxfehlern muß jetzt als Parameter die Fehlerart bzw. die Fehlermeldung erwarten. Hierzu werden wir einen komfortablen Mechanismus von C++ nutzen - variable Argumentenlisten. `error()` soll es ähnlich der C-Funktion `printf()` durch einen Format-String und eine variable Parameterliste auf einfache Weise erlauben die Fehlermeldungen zu formatieren.

Variable Parameterlisten haben zwar Vorteile, aber auch Nachteile. Probleme wie Sie beispielsweise nachfolgender Aufruf von `printf()` bereitet, sind dann auch bei error() nicht ausgeschlossen.

```
printf("%s\n");
```

Hier wird im Format-String ein Argument als String referenziert (%s), aber dieses Argument existiert nicht. Die verursachten Probleme sind dem C/C++-Kenner hinreichend bekannt.

Allerdings ist die Implementierung mittels variabler Argumentenliste der einfachste Weg zur Lösung eines Problems, daß sich in der Methode `list()` ergibt. Hier kann der Fehler „Unerwartetes Zeichen“ auftreten. Im Zuge einer ausführlichen, benutzerfreundlichen Fehlerbehandlung muß hier das fehlerverursachende Zeichen angegeben werden. Dieses ist jedoch variabel. Die einfachste Lösung ist hier die Angabe einer Fehlermeldung als Format-String „Unerwartetes Zeichen (%c)“ und des Zeichens als Argument der variablen Parameterliste.

Aufgrund dessen wollen wir die Gefahren, die zweifelsohne mit einer variablen Parameterliste verbunden sind, in Kauf nehmen.

```
void Calculator::error(const char *errmsg, ...)
{
       char err_msg[100];
       va_list ap;

       va_start(ap, errmsg);
       vsprintf(err_msg, errmsg, ap);
       va_end(ap);
```

```
        xSyntax xsyn(pos + 1, err_msg, buffer);
        xsyn.raise();
}
```

Die Verarbeitung des Formatvorschrift in `errmsg` und der etwaig folgenden Parameter übernimmt die Standard-C-Funktion `vsprintf()` aus `<stdio.h>`. Sie schreibt das Ergebnis in den Puffer `err_msg`, der dann die vollständige Fehlermeldung enthält.

Nach der Instanziierung von `xSyntax` wird diese Exception mittels `raise()` geworfen. Der Fehler ist damit gemeldet und wird in `Run()` gefangen und ausgewertet.

Von dieser Modifikation der Methode `error()` sind `list()`, `expr()` und `num()` des Parsers betroffen. In Ihnen können Syntaxfehler auftreten, die mit Hilfe von `error()` gemeldet werden. Darüberhinaus kann in `expr()` eine „Division durch Null“ auftreten.

Für diese drei Methoden ergibt sich das folgende neue Aussehen:

```
double Calculator::list()
{
        double a = expr(), b;

        if(buffer[pos] == '+') {
                pos++;
                b = list();
                return a + b;
        } else if(buffer[pos] == '-') {
                pos++;
                b = list();
                return a - b;
        } else if(buffer[pos])
                if(buffer[pos] == ')') {
                        if(!brackets)
                                error("Unerwartete ).");
                } else
                        error("Unerwartetes Zeichen (%c).",
                                buffer[pos]);

        return a;
}

double Calculator::expr()
{
        double a, b;
```

```
        if(buffer[pos] == '(') {
                pos++;
                brackets++;
                a = list();
                if(buffer[pos] != ')')
                        error(") erwartet.");
                brackets--;
                pos++;
        } else
                a = num();

        if(buffer[pos] == '*') {
                pos++;
                b = expr();
                return a * b;
        } else if(buffer[pos] == '/') {
                pos++;
                if((b = expr()) == 0)
                        throw xDivZero();
                return a / b;
        }

        return a;
}

double Calculator::num()
{
        char *endptr;
        double a =  strtod(&buffer[pos], &endptr);
        unsigned len = (unsigned) (endptr - &buffer[pos]);

        if(!len)
                error("Fliesskommazahl erwartet.");
        pos += len;

        return a;
}
```

Einen Überblick über die einzelnen Syntaxfehler und die Nichtterminale, in denen Sie auftreten können, verschafft folgende Tabelle:

Nichtterminal	*Fehlerart*
<list>	Unerwartete).
<list>	Unerwartetes Zeichen (x).
<expr>	) erwartet.
<num>	Fließkommazahl erwartet.

3 Lexikalische Analyse

Die Aufgabe der lexikalischen Analyse ist es aus dem Eingabestrom die terminalen Symbole, die sogenannten *Tokens*, zu destillieren und diese an den Parser weiterzuleiten. Der lexikalische Analysierer oder Scanner entfernt während seiner Arbeit etwaige Trenn- oder Leerzeichen (=Whitespaces) und Kommentare, die für die syntaktisch-semantische Struktur keine Relevanz haben. Die erkannten Tokens wie Konstanten verschiedener Formen (Integer, String, etc.), Operator- und Sonderzeichen, Bezeichner und Schlüsselworte werden in für den Parser adäquate Darstellungsformen überführt.

Der Scanner entlastet den Parser und macht teilweise erst effiziente Implementierungen möglich. Der Parser erhält vom Scanner grundsätzlich kompakte Tokens, um deren lexikalische Struktur oder Herkunft er sich nicht mehr kümmern muß. Der Scanner entlastet den Parser gravierend in zwei Punkten:

- Die Lexem-Darstellung muß vom Parser nicht verarbeitet werden.
- Die Whitespaces müssen nicht berücksichtigt werden.

Symbole einer Grammatik oder Wörter einer Sprache setzen sich entweder aus mehreren Zeichen zusammen oder bestehen aus einem einzelnen Eingabesymbol. Ein sich aus einer Folge von Eingabezeichen zusammensetzendes Wort heißt *Lexem*. Um diese Lexem-Darstellung der Tokens muß sich der Parser nicht mehr kümmern.

Die Betrachtung von Whitespaces im Parser ist bei komplexeren Grammatiken kaum mehr effizient und überschaubar umzusetzen. Bei jedem Ändern des Lookaheads müßten etwaig auftretende Trennzeichen ausgewertet werden. Dies würde ein immenses Aufblähen des Parsers bedeuten. Es wäre zum einen nicht einfach zu implementieren und zum anderen würde die Effizienz und die Wartbarkeit des Parsers darunter leiden. Hier ist ein Scanner unerläßlich!

3.1 Funktionsprinzip von Scannern

Ein Scanner führt im wesentlichen drei Schritte aus:

- Entfernen von Whitespaces und Kommentaren.

- Umsetzen von Konstanten in Tokens, dargestellt durch n-Tupel.
- Erkennen von Bezeichnern und Schlüsselworte.

Whitespaces haben in der lexikalischen Analyse die Aufgabe Tokens von einander zu trennen. Unter Whitespaces sind man im allgemeinen im Bereich der Computerlinguistik Seitenvorschübe, Zeilentrenner, Leer- und Tabulatorzeichen. Allerdings ist es durchaus nicht abwegig in Sprachen andere Zeichen als Trenner und damit als Whitespaces zu deklarieren. Dies ist eben im wesentlichen von der lexikalischen Spezifikation der Sprache abhängig.

Dieser Umstand, daß Whitespaces nicht unbedingt die oben genannten Zeichen gleichzusetzen sind, wird leider oft nicht berücksichtigt. [17] definiert ein Token beispielsweise wie folgt:

„In der linguistischen Datenverarbeitung wird als T. eine Folge beliebiger Zeichen verstanden, die zwischen jeweils zwei Leerzeichen stehen."

Dies mag auf die symbolischen Programmiersprachen durchaus zutreffen. Allerdings ist diese Aussage sehr vereinfacht. Wenn Sie beispielsweise an Maschinensprache denken, so sind die einzelnen Bytes der Eingabe durchaus als Tokens zu betrachten. Diese sind jedoch nicht in Leerzeichen eingeschlossen.

[1] legt sich in seiner Definition nicht einmal auf Whitespaces als Grundvoraussetzung fest. Ein Token ist hier

„A basic, grammatically indivisible unit of a language such as a keyword, operator or identifier."

Denken Sie beispielweise an einen Betriebssystememulator. Hier wird ein Parser benötigt, der den Bytecode der zu emulierenden Plattform interpretiert. Hier sind die Tokens jeweils - abhängig von der Anweisung (sozusagen ein prädiktives System) - Bytes oder Bytefolgen. Whitespaces - in welcher Form auch immer - existieren hier nicht!

Fazit: Whitespaces sind demnach spezielle Eingabesymbole, die zum Trennen von Tokens verwendet werden können. Tokens sind schlicht die atomaren Bestandteile der Grammatik. Sie können entweder direkt als die terminalen Symbole, oder als interne Darstellungen der selben im Parsing-System verstanden werden.

Diese Whitespaces geben dem Scanner die Grenzen der Tokens an und sind nur für die lexikalische Struktur wichtig. Sie werden von diesem entfernt und somit vom Parser ferngehalten.

Die Kommentare sollten in der Praxis aus mehreren Gründen in der Funktion innerhalb der lexikalischen Struktur mit Whitespaces gleichgesetzt werden. Einerseits sind Sie für

den Parser ebenfalls nicht von Bedeutung, andererseits ist die Auffassung als Trenner aus zwei Gründen sinnvoll:

a)Es sollte im Interresse des Benutzer vermieden werden, daß Bezeichner durch Kommentare zwar optisch geteilt werden, aber logisch dennoch eine Einheit bilden: `Der/* Kommentar */Bezeichner` wäre ein Symbol `DerBezeichner`.

b)Der Scanner ist einfacher zu implementieren.

Kommentare könnten demnach als Whitespace-Lexeme, also als aus mehreren Eingabesymbolen bestehende Trenner, aufgefaßt werden.

Konstanten können in einer Grammatik durch entsprechende Produktionen dargestellt werden. Die Terminale sind in diesem Fall die einzelnen Eingabesymbole, aus denen sich die Lexeme zusammensetzen. Die Produktion für `<num>` aus dem vorhergehenden Kapitel stellt beispielsweise Fließkommakonstanten dar.

Diese Produktion ist schon sehr kompliziert. Sie können sich vorstellen wie unübersichtlich Grammatiken für komplexe Sprachen werden, wenn jede Konstante durch eine eigene Produktion innerhalb der Grammatik ausgedrückt wird.

Es empiehlt sich daher für diese Konstanten eigene Symbole einzuführen, diese jedoch nicht als Nichtterminale auszufassen, sondern innerhalb der Grammatik als Terminal. Die Definition könnte dann zwar ebenfalls durch Produktionen notiert werden, jedoch als lexikalische Spezifikation unabhängig von der Grammatik. Sozusagen eine EBNF für den Parser und eine für den Scanner. Damit können Sie schon im Vorfeld zwischen lexikalischen und syntaktischen Anforderungen unterscheiden. Durch diese Modularisierung erhalten Sie außerdem überschaubare Teile, die zugleich eine gute Grundlage für die Design-Phase darstellen.

Der Scanner legt für diese lexikalischen Symbole jeweils n-Tupel an, die zum einen den Typ des Token (das Symbol) und die Attribute enthalten. In der Regel genügen für die Darstellung von Konstanten ein Attribut, nämlich der Wert der Konstanten. Somit dergeben sich 2-Tupel der Form (Symbol, Wert) für Konstanten.

Aus den Anweisungen

```
1 + 1
"Ein " + "Test"
```

würden beispielsweise folgende Tupelfolgen

```
(Integer, 1) (+,) (Integer, 1)
(String, "Ein ") (+,) (String, "Test")
```

Für die syntaktische Analyse sind lediglich die Symbole wichtig. Die Werte treten erst bei der Übersetzung ins Licht des Interesses. Der Operator + wird ebenfalls als Tupel dargestellt. Ein Attribut ist für dieses Symbol nicht erforderlich.

Beim Erkennen von Bezeichnern und Schlüsselwörtern ist es sinnvoll, wenn der Scanner ebenfalls diese Tupel-Darstellung verwendet. Die Bezeichner werden ebenfalls durch ein Symbol eindeutig klassifiziert und durch ein Attribut näher beschrieben. Dieses Attribut ist hier dann ein Verweis auf den Eintrag in der Symboltabelle, in die der Bezeichner vom Scanner eingetragen wird. Die Attribute des Bezeichners werden im Symboltabelleneintrag gespeichert. Wie schon im vorherigen Kapitel erwähnt können diese Attribute der Bezeichner in der Regel vom Scanner nicht festgelegt werden, da diese sich erst bei der Syntaxanalyse ergeben und diese folgt erst im Parser.

Auch hier ist das Attribut aus dem 2-Tupel erst für die Übersetzung nach der abgeschlossenen Analyse des Quelltextes von Bedeutung.

In modernen Sprachen existieren Lexeme mit fester Funktion als Satzzeichen (Pascal: begin und end) oder zum Aufbau von algorithmischen Konstrukten (if, C: switch, Basic, Fortran 90: SELECT CASE, Pascal: case of). Diese Schlüsselwörter sind im allgemeinen nach den Regeln für Bezeichner gebildet. Es wird demnach einen Mechanismus benötigt, um zwischen Schlüsselwörtern und Bezeichnern zu unterscheiden.

Hier existieren zwei Konzepte. Entweder die Schlüsselwörter werden reserviert und dürfen nicht als Bezeichner verwendet werden, e. g. Pascal und C++, oder werden nicht reserviert und können als Bezeichner auftreten, z. B. Fortran und PL/I.

Das erste Konzept ermöglicht es schon bei der lexikalischen Analyse Schlüsselwörter als solche zu erkennen. Die können infolgedessen direkt als Symbol ohne Attribut als 2-Tupel abgebildet werden. Die zweite Verfahrensweise macht die Analyse schon komplizierter. Hier kann das Schlüsselwort nicht unbedingt vom Scanner, sondern eventuell erst beim Parsing erkannt werden, denn die Bedeutung des Lexems wird erst aus dem Kontext, der syntaktischen Struktur, gewonnen.

Fortran erlaubt wie bereits erwähnt die Verwendung von Schlüsselwörtern als Variablen. Die folgenden Anweisungen sind beide korrekte FORTRAN-77-Statements:

```
IF(I)=70
IF(I) 30,40,50
```

Im ersten Fall wird der Array-Variablen `IF` mit dem Index `I` der Wert 70 zugewiesen. Das zweites Statement ist ein arithmetische Verzweigung und prüft den Inhalt von `I` auf negativ, gleich Null und positiv und verzweigt entsprechend an die Sprungziele `30`, `40` oder `50`.

In diesem Fall wäre durch ein zusätzliches Lookahead ein Erkennen der Bedeutung des Lexems `IF` durch den Scanner möglich. Es muß schlicht geprüft werden, ob ein Operator oder Sonderzeichen (`IF` ist Variable) bzw. ob ein Bezeichner (`IF` ist Schlüsselwort) folgt.

Ein anderes sehr extremes Beispiel für diese Problematik findet sich in [2]. Die Regeln zur Unterscheidung zwischen Schlüsselwörtern und Bezeichnern sind in PL/I sehr kompliziert:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
```

Entgegen der allgemeinen Auffassung, daß die Unterscheidung zwischen Bezeichner und Schlüsselwort im Scanner stattzufinden hat, sollte hier in Erwägung gezogen werden die Entscheidung eventuell erst im Parser zu treffen.

Die einfacher zu implementierende Möglichkeit ist eindeutig die Reservierung. Dies und die Tatsache, daß diese Mehrdeutigkeiten ein großes Fehlerpotential in sich bergen, mag der Grund dafür sein, daß in modernen Sprachen grundsätzlich Schlüsselwörter reserviert sind.

Ein anderes Problem bei der lexikalischen Analyse ist das Erkennen von Lexemen, die mit dem gleichen Eingabesymbol starten, jedoch unterschiedliche Bedeutungen haben. Ein Beispiel wären die Operatoren > und >=.

Hier können automatentheoretische Verfahren angewendet werden ([2]). Die automatentheoretischen Mittel zur Beschreibung von Lexemen sind jedoch ein wichtiges Hilfsmittel in der lexikalischen Analyse. Doch hierzu später mehr.

Zu obigen Beispiel gibt es allerdings eine einfachere Lösung. Nach dem Lesen von > aus dem Eingabepuffer findet schlicht ein Lookahead auf das nächste Zeichen statt. Ist dieses Eingabesymbol das =, so liegt >= vor und als Token wird das Symbol für „größer als oder gleich“ zurückgeliefert. Entpuppt sich dieses zusätzlich betrachtete Zeichen nicht als =, so wird das Symbol für „größer als“ korrespondierend zu > zurückgegeben. Das „zuviel“ gelesene Zeichen wird vom Scanner einfach wieder zurück in den Eingabestrom gestellt.

Das Prinzip des Scanners kann schlicht analog zu Abbildung 3.1 wie folgt zusammengefaßt werden: Der Scanner wandelt einen zeichenbasierten Eingabestrom in einen für den Parser tauglichen auf Tupeln aufbauenden Symbolstrom um. Er übernimmt somit eine Vermittlerrolle zwischen Eingabe und Parser. [2] spricht hier treffend von einer *Erzeuger-Verbraucher-Beziehung*: „Der Scanner erzeugt Symbole, der Parser verbraucht sie.“

Die Interaktion zwsichen Scanner und Parser wird durch die Größe des Puffers für die Symbole beschränkt. Der Scanner kann erst wieder weiterarbeiten, wenn dieser Puffer durch eine Anfrage des Parsers geleert wird. Im allgemeinen kann dieser Puffer lediglich ein Symbol aufnehmen. In der bisherigen Fachliteratur wird deshalb empfohlen, den Scanner als Unter- oder Coroutine des Parsers zu implementieren, die vom Parser aufgerufen wird, wenn dieser ein Token benötigt.

Im Zuge einer modernen Betrachtung kann man heute sagen, daß der Scanner als Objekt implementiert werden sollte, an welches das Paser-Objekt eine Nachricht sendet, wenn ein Token benötigt wird.

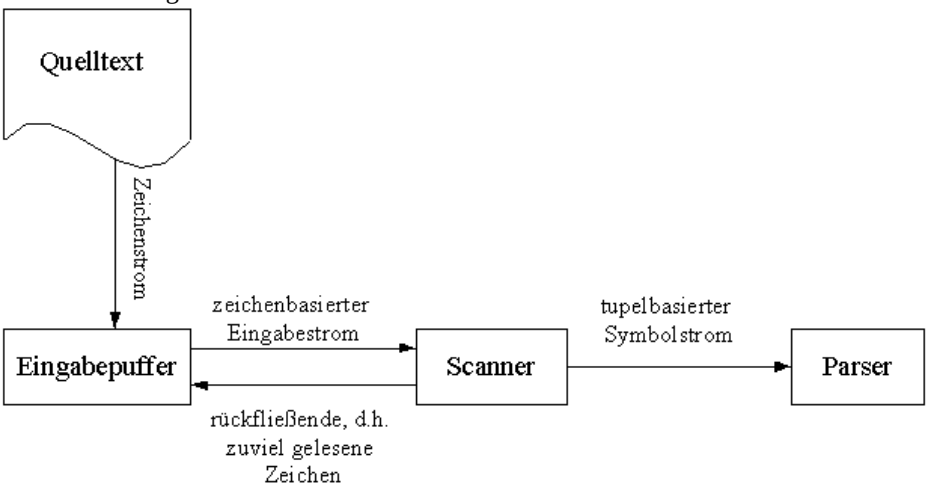


Abb. 3.1: Der Scanner als Vermittler zwischen Eingabe und Parser

Eine sehr interessanter Ansatz ergibt sich mit den von aktuellen Betriebssystemen bereitgestellten Multithreading. Hier wäre es möglich Scanner und Parser in jeweils einen Thread zu legen. Der Scanner könnte somit im Hingrund die lexikalische Struktur des/der Quelltexte(s) analysieren und diese Struktur in einen Puffer schreiben. Der Parser könnte aus diesem Puffer jeweils nebenläufig die Tokens auslesen. Eine genauere Betrachtung wäre wohl nicht nur vom theoretischen Standpunkt aus interessant.

Die Eingabe erfolgt in der Regel über einen Eingabepuffer. Dieser Puffer wird jeweils mit Zeichen aus dem Quelltext gefüllt. Ein Zeiger, der über diesem Puffer operiert, gibt das aktuell betrachtete Zeichen an (Lookahead). Der Rückfluß von Zeichen kann durch Rücksetzen dieses Zeigers erreicht werden.

Die Eingabepufferung ist aus mehereren Gründen sinnvoll. Einerseits müssen für die Fehlerbehandlung die Positionen der Eingabezeichen nachvollziehbar sein, da schließlich angegeben werden muß, wo der Fehler auftrat. Andererseits ist es wesentlich effizienter Daten in Blöcken in einen Puffer einzulesen und durch Operation über diesem Puffer zu verarbeiten, als jedes einzelne Zeichen einzulesen. Doch auch hierzu später mehr.

3.2 Motivation für Scanner

Weshalb sollte man einen Scanner implementieren? - Einige Punkte wurden bereits angesprochen, sollen jedoch der Vollständigkeit wegen hier erneut Erwähnung finden:

- Der Umgang mit der Komplexität des Parsing-Systems wird erleichtert. Die Trennung zwischen Scanner und Parser macht das Analyse-Modul wesentlich überschaubarer.
- Der Parser kann sich effizienter um seine Aufgabe, nämlich die Analyse der Syntax, kümmern. Whitespaces und Kommentare werden vom Parser durch den Scanner ferngehalten.
- Die Trennung zwischen lexikalischen - und syntaktischen Analysierer steigert die Effizienz des Systems. Für beide Bereiche existieren effiziente Algorithmen, für die bei einer ungetrennten Implementierung Abstriche gemacht werden müßten.

Die Portabilität des grammatikalischen Analysierers wird erhöht. Plattformabhängige Eigenschaften, wie beispielsweise unterschiedliche Zeichensätze, können innerhalb des Scanners gekapselt werden und betreffen den Parser, das wohl komplexere Werk, nicht mehr.

3.3 Aufbau von Scannern

Wie bei einem Parser, sind auch bei der Implementierung eines Scanners entsprechende Vorüberlegungen zu treffen. So wie eine Grammatik die Struktur oder das Funktionsweise eines Parsers vorgibt, definiert diese Spezifikation die Struktur des Scanners.

So wie bei der Syntaxspezifikation die kontextfreie Grammatiken oder die EBNF existieren, gibt es für die Festlegung der lexikalischen Struktur ebenfalls entsprechende Notationen. Hier können Sie auf reguläre Defintionen und regulären Ausdrücken zurückgreifen. Diese beiden Notationen werden Sie etwas später kennenlernen.

Die mit diesen Hilfsmitteln aufgestellte lexikalische Spezifikation kann direkt in einen Scanner überführt werden. So entstehen sehr effiziente Scanner.

Eine andere Möglichkeit ist der Einsatz eines Scannergenerators, wie lex, der später in diesem Buch gesondert betrachtet wird. Werkzeuge wie lex ermöglichen es direkt durch reguläre Ausdrücke Lexeme zu spezifizieren. Der Vorteil ist hier eindeutig, daß der Anwender zwar aktuelle und effiziente Algorithmen zum Pattern-Matching, also der Mustererkennung, einsetzen kann, ohne mit die automatentheoretischen Hintergründen vertraut sein zu müssen.

Die Nähe zur Eingabe und damit zum Quelltext prädestiniert den Scanner Aufgaben der Benutzerführung zu übernehmen. Der Scanner könnte die Zeilenwechsel zählen, um so für die Fehlerbehandlung die Nummer der Zeile bereitzuhalten, in der der Fehler auftrat. Auch Makroprozessor- oder Aliasaufgaben könnten auf ihn übertragen werden.

Gelegentlich werden Scanner sogar in zwei Phasen aufgeteilt. Die erste übernimmt das Lesen und Puffern des Quelltextes und die zweite übernimmt die eigentliche lexikalische Analyse. Bei einer komplexeren Lexik der Sprache ist dieses Vorgehen durchaus empfehlenswert.

3.3.1 Begriffsabgrenzung

In der lexikalischen Analyse tauchen drei Begriffe auf, die wir analog zu [2] verwenden wollen. Es sind *Symbol*, *Muster* und *Lexem*. Verschiedene Zeichenfolgen treten in der Syntax durch das gleiche Symbol auf. Diese Zeichenfolgen nennen sich Lexeme und sind durch ein entsprechendes, grundlegendes Muster beschrieben.

Ein Lexem gehört zu einer Menge, die den Namens des Symbols trägt und durch die Regeln des Musters eindeutig beschrieben ist. 25 und 6 sind beispielsweise beides Lexeme des Typs Integer (=Symbol) und werden durch das Muster „Folge von Ziffern“ lexikalisch beschrieben.

Das Symbol ist also sozusagen der Typ, das Muster die Bildungsvorschrift und Lexem ist die konkret im Quelltext vorliegende Zeichenfolge, die als lexikalische Einheit betrachtet wird.

Die (lexikalischen) Symbole repäsentieren in der Grammatik, die bekanntlich die Grundlage des Parsers darstellt, Terminale. Diese Symbole werden in der Implementierung häufig durch Integerwerte beschrieben. Fordert der Parser ein Token an, so erhält er vom Scanner einen eindeutigen Integerwert, sowie einen Attributwert, zusammengefaßt als 2-Tupel. Es hat sich in der Praxis gezeigt, daß ein Attribut ausreicht, um ein Symbol zu beschreiben. Die Attribute werden vom Parser an folgende Phasen übergeben, da für die syntaktische Analyse lediglich das Symbol - der Integerwert - von Interesse ist.

Die Muster zur Beschreibung der Symbole können je nach Grammatik recht komplex sein. Schlüsselworte werden beispielseweise recht einfach durch die Konkatenation von Zeichen beschrieben (if, while, break). Bezeichner hingegen können entsprechenden Konventionen folgend vom Benutzer frei definiert werden. „Das erste Zeichen muß ein Unterstrich oder ein beliebiger Groß- oder Kleinbuchstabe aus dem englischen Alphabet sein. Danach kann eine Reihe von weiteren Groß- und Kleinbuchstaben aus dem englischen Alphabet, Unterstrichen und Ziffern folgen.“ Dieses Muster beschreibt zum Beispiel alle in C++ erlaubten Bezeichner. Das Muster für Fließkommazahlen aus C++ ist dagegen sogar noch komplizierter.

Darüberhinaus kann auch die Position des Lexems im Quelltext von Bedeutung sein. Formatfeste Sprachen wie FORTRAN 77 oder COBOL 85 definieren die Gültigkeit oder die grammatikalische Funktion eines Lexems auch anhand er Spaltenposition an der es beginnt und endet. Moderne Sprachen und Sprachüberarbeitungen (z. B. Fortran 90) sind heute formatfrei, so daß dieser Aspekt der lexikalischen Analyse immer mehr an Bedeutung verliert.

3.4 Fehlerbehandlung im Scanner

Die Möglichkeiten zur frühzeitigen Fehlererkennung in der lexikalischen Analyse sind begrenzt. Da die Tokens nicht im Zusammenhang gesehen werden können, denn dieser ergibt sich erst aus syntaktisch-semantischen Kontext, ist die Fehlerbehandlung im Scanner begrenzt.

Ein Scanner kann beispielsweise im folgenden nicht erkennen, daß hier das Schlüsselwort `while` falsch geschrieben wurde:

```
whnile(*a) *a++;
```

Der Scanner würde `whnile` als korrekten Bezeichner identifizieren und an den Parser zurückgeben. Dieser würde anschließend lediglich feststellen, daß gemäß den syntaktischen Regeln ein Semikolon nach der schließenden Klammer fehlt. Er würde es als Prozeduraufruf interpretieren.

Eine Möglichkeit festzustellen, daß hier lediglich das die Schleife einleitende Schlüsselwort falsch geschrieben ist, besteht nicht. Im Scanner erst recht nicht.

Der Scanner kann nur auf reine lexikalische Fehler reagieren. Dies kann sein, weil auf das aktuelle Präfix der Eingabe, also das betrachtete Lookahead, auf kein Muster paßt. Die einfachste Strategie des Error-Recovery ist hier das *panische Fortsetzen* (=*panic mode recovery*). Der Scanner ignoriert und überspringt dabei sukzessive alle nicht passenden Zeichen bis ein einem Muster entsprechendes Symbol konstruiert hat. Aus dem fehlerhaften Bezeichner `Überschall` würde dann `berschall`.

Zur Erinnerung: Dieses Error-Recovery wird nur angewendet, damit im folgenden etwaige weitere Fehler erkannt werden können und ebenfalls darauf hingewiesen werden kann. Durch diesen Fehler wird in der Regel kein Patching durchgeführt, damit der Quelltext trotzdem arbeiten kann!

Das panische Fortsetzen mag dem Parser die Arbeit erschweren, aber für interaktive Systeme ist es gut geeignet (vgl. [2]).

Insgesamt existieren sechs mögliche Recovery-Möglichkeiten:

- Löschen fehlerhafter Zeichen (panisches Fortsetzen).
- Löschen eines überzähligen Zeichens.

- Einfügen eines erwarteten Zeichens.
- Ersetzen eines fehlerhaften Zeichens.
- Vertauschen zweier benachbarter Zeichen[1].
- Betrachten einer fehlerhaften Zeichenfolge zwischen erkennbar korrekten Symbolen mittels fehlertoleranter Suche innerhalb der Schlüsselwort- und der Symboltabelle.

Die einfachste Strategie besteht darin den Fehler durch die Anwendung einer der oben geannten Strategien in ein gültiges Lexem zu wandeln. In der Praxis bestätigt sich häufig, daß ein lexikalischer Fehler lediglich das Produkt einer einzigen Fehlertransformation ist. Lediglich einige Fehlersituationen entstehen durch mehrere Fehlertransformationen. Diese zu betrachten wäre in der Praxis kaum akzeptabel, da dies zu ineffizient wäre.

In der Theorie existiert zwar der Grundsatz, daß ein fehlerhaftes Programm k Fehler enthält, wenn die kürzeste Folge von Fehlertransformationen die Länge k hat ([2]). In der Praxis wird dieses Prinzip jedoch nicht eingesetzt, da die Effizient des Systems leiden würde.

3.5 Eingabepufferung

Das Lesen von einzelnen Zeichen von einem Datenträger ist nicht effektiv. Für jedes Zeichen müssen diverse Bibliotheks- und/oder Systemaufrufe erfolgen, die dann noch zusätzlich Hardware in Gang setzen müssen. Dieses relativ zeitintensive Verfahren auf jedes Zeichen eines Quelltextes anzuwenden ist extrem ineffektiv und selbst bei guten Vertrauen in das Mooresche Gesetz nicht zu akzeptieren.

Die lexikalische Analyse ist zudem die einzige Phase, die den Quelltext Zeichen für Zeichen abarbeiten muß. Spätere Phasen sind zwar aufgrund ihrer algorithmischen Struktur wesentlich komplexer, aber das oben beschriebene, förmlich zeitfressende Leseverfahren ist dennoch eine nicht zu unterschätzende Größe des Analyseprozesses. Der Zeitbedarf der Eingabeprozedur darf deshalb nicht vernachlässigt werden. Außerdem sitzt hier durch die Möglichkeit der Pufferung ein erhebliches Optimierungspotential, das schlicht genutzt werden muß.

Viele Sprachentwürfe verlangen das Betrachten von mehreren Zeichen als Lookahead, um ein Symbol zu erkennen. In der Fachliteratur wimmelt es bei Beispielimplementierungen von LL(1)-Parsern, die im Zuge eines Lookaheads zuviel gelesene Zeichen schlicht durch die C-Funktion `ungetc()` zurückstellen. Als Referenz

1 Das Vertauschen zweier benachbarter Zeichen ist im übrigen der häufigste Fehler beim Tippen mit Zehnfingersystem.

sei exemplarisch auf das einführende Beispiel im „Drachenbuch“ ([2]) und auf den Scanner von hoc bei Kernighan und Pike ([8]) verwiesen.

Bei komplexeren Systemen reicht jedoch ein einzelnes Lookahead nicht immer aus. Unter diesem Aspekt betrachtet gewinnt die Eingabepufferung weiter an Bedeutung.

Eine sehr leistungsstarke und gebräuchliche Lösung ist das Arbeiten mit *zweigeteiltem Puffer mit Wächtern*. Das Prinzip des zweigeteilten Puffers ist - wie die meisten genialen Ideen - recht simpel. Ein Puffer wird in zwei Hälften mit je einer Aufnahemkapazität von n Zeichen zerlegt. Um das optimierungstechnische Potential voll ausschöpfen zu können, ist n die Größe einer Zuordnungseinheit (=Cluster) bzw. eines Blocks des Dateisystems. Werte wie 1024, 2048 oder 4096 sind hier üblich. Dadurch wird es möglich ganze Datenblöcke, die garantiert nicht fragmentiert sind, schnell in den Speicher zu laden. Diese Blöcke haben nämlich eine Größe, die die kleinste verwaltbare Einheit des Dateisystems darstellt - sozusagen die Atome der Dateiverwaltung.

In die Pufferhälften jeweils in einem Schritt n Zeichen eingelesen. Ist die gelesene Zeichenzahl geringer als n, so wird das Ende durch ein EOF gekennzeichnet.

Über den Puffern arbeiten zwei Zeiger. Einer markiert den Anfang des aktuellen Lexems. Der andere markiert das Ende des Lexems. Bei Beginn der Analyse zeigen beide Zeiger auf das erste noch nicht zu einem Lexem gehörende Zeichen. Im folgenden werden die Zeichen solange sukzessive analysiert und der Lookahead-Zeiger erhöht bis ein Lexem erkannt ist. Zwischen den beiden Zeigern findet sich dann das Lexem, daß dann als Symbol an den Parser übergeben werden kann. Daraufhin werden die Zeiger auf das Zeichen, das dem Lexem direkt folgt, gesetzt. (Vergleiche Abbildung 3.2.)

Wird ein Ende einer Pufferhälfte erreicht, wird die nächste Pufferhälfte mit n Zeichen aus dem Eingabestom aufgefüllt. Der Lookahead-Zeiger wird anschließend auf den Anfang der neu gefüllten Puffers gesetzt.

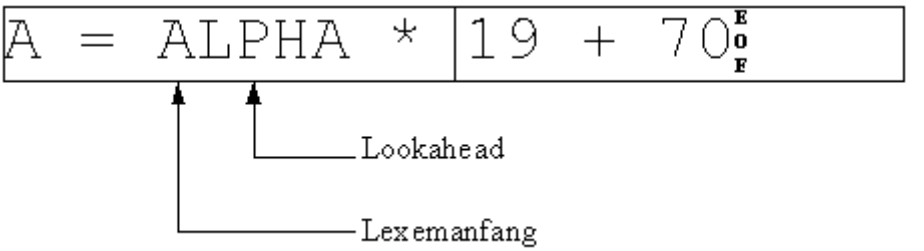


Abb. 3.2: Prinzip des zweigeteilten Puffers.

Bei der algorithmischen Umsetzung des Prinzips ergibt sich eine kleine Schwäche. Bei jedem Verschieben des Lookahead-Zeigers muß geprüft werden, ob dieser sich am Ende einer Pufferhälfte befindet. Dies bedeutet, daß für jedes Eingabezeichen je ein Tests für die beiden Pufferhälften stattfindet.

Durch das Einführen von sogenannten *Wächern* kann der Aufwand minimiert werden. Ans Ende jeder Pufferhälfte setzt man jeweils ein EOF (Abbildung 3.3). Somit lautet die Vorschrift nicht mehr „wenn die Pufferhälfte erreicht wurde, soll die andere Pufferhälfte erneut gefüllt werden“, sondern „wenn das nächste Zeichen ein EOF ist und die Pufferhälfte erreicht ist, soll die andere Pufferhälfte erneut gefüllt werden“. Damit ist für die Analyse eines Eingabezeichens nur noch ein zusätzlicher Test erforderlich. Es wird nur noch getestet, ob ein EOF vorliegt oder nicht. Liegt dieses vor, wird eben noch getestet, ob das Pufferende oder das wirkliche Dateiende erreicht wurde.

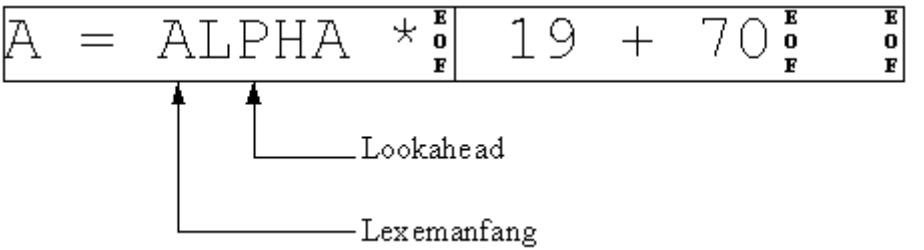


Abb. 3.3: Zweigeteilter Puffer mit Wächtern.

Hier existiert für das EOF nur noch eine Bedingung, die beim Inkrementieren des Lookahead-Zeigers getestet werden muß. Vorher waren es zwei, nämlich „Ist das Ende der ersten Hälfte erreicht?“ und „Ist das Ende der zweiten Hälfte erreicht?“.

Der interessante Aspekt an diesem Prinzip des zweigeteilten Puffers ist, daß Ketten von Whitespaces und Kommentare als Lexeme aufgefaßt werden können. Diese Lexeme haben lediglich die Besonderheit, daß sie nicht in Symbole gewandelt und an den Parser zurückgegeben werden. Sie werden stattdessen einfach verworfen, die Zeiger neu positioniert und der lexikalische Analyseprozeß wird fortgesetzt.

In der Regel funktioniert diese Puffertechnik in der Praxis recht gut. Allerdings darf man nicht aus den Augen verlieren, daß die maximale Lexemlänge beschränkt ist. Gerade bei Sprachdefinitionen, in den die Schlüsselwörter nicht reserviert sind, kann es zu Komplikationen kommen. Hier müssen teilweise sehr viele Zeichen betrachtet werden, um ein Lexem einordnen zu können.

3.6 Musterspezifizierung durch reguläre Ausdrücke

Das gebräuchlichste Beschreibungsmittel für Muster sind sogenannte *reguläre Ausdrücke*. Ein regulärer Ausdruck beschreibt eine Menge von Zeichenketten. Es beschreibt also die Gemeinsamkeiten einer Menge von Strings, gibt also das zugrundeliegende Muster an. Reguläre Ausdrücke stammen wie die kontextfreien Grammatiken aus dem Bereich der formalen Sprachtheorie. Sie sind gerade durch das UNIX-System in den unterschiedlichsten Anwendungen sehr weit verbreitet.

Mit Hilfe von regulären Ausdrücken lassen sich *Hüllen bilden*, d. h. Wiederholungmuster spezifizieren und *alternative* Ausdrücke angeben.

Zu jedem regulären Ausdruck r existiert eine Sprache L(r) mit dem Alphabet Σ, die die Menge Σ* aller möglichen Strings beschreibt. Strings sind schlicht Konkatenationen von Zeichen aus Σ gemäß den Bildungsgesetzen der Sprache L(r), also r.

Für reguläre Ausdrücke gilt folgendes:

- Ein Symbol $c \in \Sigma$, dann ist c ein regulärer Ausdruck und bezeichnet die Menge {c}.
- ε ist ein regulärer Ausdruck und bezeichnet die Menge aller leeren Strings {ε}.
- Zur Verknüpfung zweier regulärer Ausdruck r_1 und r_2, die die Sprachen $L(r_1)$ und $L(r_2)$ bezeichnet, existieren folgende Möglichkeiten:

  a) Alternation: $r_1|r_2$ bezeichnet $L(r_1) \cup L(r_2)$. | ist der Oderoperator, für den gilt:
  a|b = b|a => | ist kommutativ
  a|(b|c) = (a|b)|c => | ist assoziativ

  b) Konkatenation: r_1r_2 bezeichnet $L(r_1)L(r_2)$. Es gilt:
  (ab)c = a(bc) => Konkatenation ist assoziativ
  (a|b)c = ac|bc => Konkatenation ist distributiv bezüglich |
  εa = a => ε ist das neutrale Element der Konkatenation

- Regulären Ausdrücke unterstützen die Hüllenbildung. Wenn r ein regulärer Ausdruck ist, dann gilt:
  Kleene-Hülle: r* bezeichnet (L(r))*.
  Positive Hülle: r+ ist äquivalent zu rr* und bezeichnet L(r)(L(r))*.
  Es existiert ein Sonderfall:
  Optionales Element: r? ist äquivalent zu r|ε und bezeichnet $L(r) \cup L(\varepsilon)$.
- Klammerung: (r) = r und bezeichet L(r).

Diese Leistungsfähigkeit mit der (Text-)Muster durch regulären Ausdrücke beschrieben werden können prädestiniert diese „Musterbeschreibungssprache“ geradezu als Hilfsmittel für die lexikalische Analyse. Da reguläre Ausdrücke darüberhinaus in äquivalente endliche Automaten transformiert werden können, ist auch die direkte algorithmische Umsetzung keine unüberwindliche Hürde. Dies ist auch der Grund weshalb Scannergeneratoren wie lex auf diesen regulären Ausdrücken basieren.

In der Praxis erfährt die obige theoretische Basis, einige Erweiterungen, die das Leben des Entwicklers erleichtern. Im Einsatz findet man häufig folgende Operatoren bzw. Metazeichen:

Ausdruck	*Funktion*
c	Das Zeichen c ohne Sonderbedeutung.
\c	Hebt die Sonderbedeutung des Zeichens c auf bzw. beschreibt eine Escape-Sequenz.
^	Zeilenanfang.
$	Zeilenende.
.	Beliebiges Zeichen, außer Zeilentrenner.
[...]	Zeichenklasse: Ein Zeichen aus ..., wobei auch Bereiche wie a-z erlaubt sind.
[^...]	Zeichenklasse: Ein Zeichen nicht aus ...; ebenfalls Bereiche erlaubt.
r*	Null oder mehr Vorkommen des Ausdrucks r.
r+	Ein oder mehr Vorkommen des Ausdrucks r.
r?	Null oder ein Vorkommen des Ausdrucks r.
r1r2	Auf den Ausdruck r1 folgt r2.
r1|r2	Entweder r1 oder r2.
(r)	Entspricht dem Ausdruck r. Klammerungen können geschachtelt werden.

Das Muster für einen C++-Bezeichner könnte durch den regulären Ausdruck

```
[_A-Za-z][_A-Za-z0-9]*
```

ausgedrückt werden. Auf einen Buchstaben oder Unterstrich (1. Zeichenklasse) folgen beliebig viele (auch Null) weitere Buchstaben, Unterstriche oder Ziffern (2. Zeichenklasse mit Kleene-Operator).

Eine Stringkonstante, wie sie in C++ akzeptiert werden würde, könnte wie folgt aufgebaut werden:

```
"([^"\\]|\\.)*"
```

Das Einschließen in Anführungszeichen dürfte klar sein. Die Kleene-Hülle ebenfalls, da in einem String beliebig viele oder ggf. auch keine Zeichen enthalten sein dürften. Doch die runde Klammer mutet etwas verwirrend an.

Diese Klammer kapselt eine Alternation, d. h. zwei per Oderoperator | verknüpfte reguläre Ausdrücke. Der erste Ausdruck ist eine negierte Zeichenklasse `[^"\\]`. Sie deckt alle Zeichen außer Anführungszeichen und Backslashes ab. Anführungszeichen können innerhalb der Stringkonstanten nicht ohne weiteres verwendet werden, da sie zum Einfassen der selben verwendet werden. Das Ausschließen des Backslashs ist eine kleine Versicherung. Wenn wir uns dem zweiten alternativen Ausdruck zuwenden, so deckt dieser eine Konkatenation von Backslash und einem beliebigen Zeichen ab. Damit ist es möglich in Stringkonstanten auch beispielsweise das Anführungszeichen durch \" zuverwenden. Die Versicherung im Ausschließen des Backslashs im anderen alternativen Ausdruck besteht darin, daß dort nicht fälschlicherweise ein Backslash

abgedeckt werden kann, da dieser immer (!) durch den zweiten Ausdruck abgedeckt werden muß.

Die Angabe der Backslashes durch \\ ist nötig, da der Backslash in der Notation der regulären Ausdrücke eine Sonderstellung einnimmt. Er kann die Sonderbedeutung von Metazeichen aufheben. \. deckt beispielsweise den einfachen Punkt ab. \\. hingegen beschreibt die Verkettung von Backslash und einem beliebigen Zeichen.

Für näheres zur formal-mathematischen und sprachtheoretischen Betrachtung von regulären Ausdrücken sei auf [2], [19] und [7] verwiesen.

3.7 Reguläre Definitionen

So wie die Beschreibung von syntaktischen Zusammenhängen durch Produktionen ausgedrückt werden kann, so läßt sich die Lexik einer Sprache durch *reguläre Definitionen* ausdrücken. Was die kontextfreie Grammatik in der syntaktischen Analyse ist, das ist die *reguläre Grammatik* in der lexikalsichen Analyse.

Jedem regulären Ausdruck kann auf diese Weise ein Name zugeordnet werden. Innerhalb dieser Ableitungen können wieder die Namen anderer regulärer Definitionen referenzieren. Es ist sinnvoll die Namen der regulären Definitionen wie bei kontextfreien Grammatiken in spitze Klammern zu fassen.

Der reguläre Ausdruck für C++-Bezeichner kann durch eine reguläre Defintion wie folgt übersichtlich gestaltet werden:

```
<letter> --> [_A-Za-z]
<number> --> [0-9]
<id>     --> <letter>(<letter>|<number>)*
```

Der Name `<id>` bezeichnet somit das gesamte Muster für einen C++-Bezeichner.

Der reguläre Ausdruck für die Stringkonstante ist regulär wie folgt definiert:

```
<strchar> --> [^"\\]
<esc>     --> \\.
<string>  --> "(<strchar>|<esc>)*"
```

Die einzelnen Namen der regulären Defintionen, die dem Token gleichzusetzen sind, die korrespondierenden regulären Ausdrücke und die Attributwerte trägt man sinnvollerweise in eine Tabelle ein.

Für die beiden obigen Lexeme würde diese Tabelle wie folgt aussehen:

Regulärer Ausdruck	*Symbol*	*Attributwert*
[_A-Za-z][_A-Za-z0-9]*	id	Verweis auf Symboltabelle
"([^"\\]|\\.)*"	string	Zeiger auf String zwischen den beiden "

3.8 Schwachpunkte in regulären Grammatiken

Nicht alle möglichen Sprachen können durch reguläre Ausdrücke respektive -Definitionen beschrieben werden. Perioden und Abhängigkeiten innerhalb von Ausdrücken können nicht beschrieben werden. Diese Schwächen existieren allerdings bei kontextfreien Grammatiken ebenfalls.

Schachtelungen können ebenfalls durch reguläre Ausdrücke nicht beschrieben werden. Dieses Manko ist bei kontextfreien Grammatiken nicht zu finden.

Beispiele für nicht beschreibbare Strukturen:

- Perioden: Ein Muster für einen String der formal durch {αcα | α ist eine Folge von Kleinbuchstaben} definiert werden kann, kann durch einen regulären Ausdruck nicht beschrieben werden, da α nach dem Symbol c wiederkehrt. Der konkrete String kehrt wieder nicht nur das (Teil-)Muster zu α!

  Die einzige Ausnahme hierzu bildet ein fest durch einen String als Muster definiertes α. Beispiel: α := abcdef.

- Abhängigkeiten: In älteren Versionen von Fortran beispielsweise gab es noch den sogenannten Hollerith-String der Form $nHa_1a_2a_3...a_n$. Diese Lexeme können nicht durch reguläre Ausdrücke beschrieben werden, da die Anzahl der Symbole nach dem H vom Wert n abhängt.

- Schachtelungen: Es gibt keinen regulären Ausdruck, der die korrekten Klammernebenen in C++ beschreiben kann.

Diese Einschränkungen sind jedoch nicht so gravierend, als das die Idee der Verwendung von regulären Ausdrücken bei der Implementierung von Scannern verworfen werden müßte.

Die Verwendung von Periodika ist selten nötig. Falls ja, kann man immer noch eine entsprechende Speziallösung implementieren.

Abhängigkeiten können durch eine nachgeschaltete Validierung oder einen modifizierten endlichen Automaten gelöst werden.

Schachtelungen sind wohl eher ein Fall der Syntax, denn der Lexik. Demzufolge ist diese Problematik an den Parser zu delegieren und kann dort leicht in der kontextfreien Grammatik berücksichtigt werden.

3.9 Endliche Automaten

Wie bereits zuvor schon angedeutet, existiert zu jedem regulären Ausdruck ein korrespondierender endlicher Automat, der die gleiche Sprache erkennt. Ein endlicher Automat wird von [1] wie folgt definiert:

„Finite State Machine (FSM or "Finite State Automaton", "transducer"): An abstract machine consisting of a set of states (including the initial state), a set of input events, a set of output events and a state transition function. The function takes the current state and an input event and returns the new set of output events and the next state. Some states may be designated as "terminal states". The state machine can also be viewed as a function which maps an ordered sequence of input events into a corresponding sequence of (sets of) output events."

Ein endlicher Automat besteht demnach aus einer endlichen Menge von Zuständen, einer endlichen Menge von Eingabeereignissen und einer endlichen Menge von Ausgabeereignissen. Den Übergang zwischen den Zuständen ist eindeutig durch ein Eingabeereignis geregelt. Jeder Automat besitzt einen Startzustand und eventuell mehrere Endzustände.

Formal stellt sich ein solcher endlicher Automat M als Quintupel $M(Q,\Sigma,\delta,q_0,F)$ dar. Q ist die endliche Menge aller Zustände. Σ ist das endliche Eingabealphabet. Dieses ist identisch mit der gleichbezeichneten Menge des regulären Ausdrucks. q_0 ist ein Element der Menge Q und ist der Startzustand des Automaten M. F als echte Teilmenge von Q ist die Menge der Endzustände. δ ist die Transitionsfunktion, die $Q \times \Sigma$ auf Q abbildet. $\delta(q,a)$ mit $q \in Q$ und $a \in \Sigma$ ermittelt also für ein Eingabesymbol a und einen Zustand q den folgenden Zustand (bei deterministischen Automaten) oder die folgenden Zustände (bei nichtdeterministischen).

Endliche Automaten werden im allgemeinen durch *Transitionsdiagramme* dargestellt. Ein solches Diagramm ist ein gerichteter Graph, dessen Knoten die Zustände des Automaten repräsentieren. Die Kanten sind die *Übergänge* - die *Transitionen*. Diese Kanten werden mit Symbolen des Eingabealphabets Σ *markiert*. Damit sind die Wege der Funktion $\delta(q,a)$ eindeutig beschrieben. Der Zustand q ist durch den Koten dargestellt, von dem der Pfeil ausgeht, a ist repräsentiert durch die Marke auf der Kante und der Folgezustand, d. h. das Resultat der Funktion δ, ist durch den Knoten, bei dem der Pfeil endet, vertreten.

Eine Kante, die von keinem Knoten ausgeht, wird mit *Start* beschriftet. Sie zeigt auf den Knoten, der den Startzustand q_0 des Automaten darstellt. Die Endzustände der Menge F, die erreicht werden, wenn ein Lexem erkannt wurde, werden durch einen umrandeten Knoten vertreten. Ist ein solcher Endzustand mit einem Stern markiert, so deutet dies darauf hin, daß ein zuviel gelesenes Zeichen in den Eingabestrom zurückfließen muß.

Kanten mit der Markierung *other* deuten dies ebenfalls an. Hier liegt irgendein Zeichen im Eingabestrom, das nicht vom aktuellen Muster abgedeckt wurde.

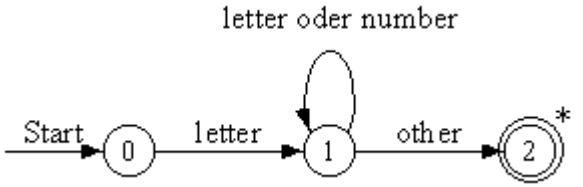


Abb. 3.4: Transitionsdiagramm des Automaten für das Muster für Bezeichner.

Die Kanten können auch mit Namen für reguläre Definitionen versehen werden. Es muß allerdings darauf geachtet werden, daß dieser Name wirklich nur ein einzelnes Eingabesymbol, sprich Zeichen, darstellt. Dieses Verfahren mag sinnvoll sein, wenn beispielsweise auf eine Zeichenklasse hingewiesen werden soll.

In den Abbildungen 3.4 und 3.5 sehen Sie beispielsweise die Transitionsdiagramme für die Automaten für die bekannten Muster für Bezeichner und Stringkonstanten.

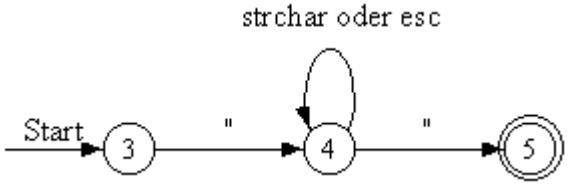


Abb. 3.5: Transistionsdiagramm des Automaten für das Muster für Stringkonstanten

Ein String wird durch einen Automaten erkannt, wenn für diesen ein Pfad vom Start- zu einem Endzustand im Automaten existiert. Ein *Pfad* ist dabei die Verkettung von Kanten (in Pfeilrichtung), deren Markierungen den Symbolen des Strings entsprechen.

Das Erkennen des Lexems "abc" durch den Automaten aus Abbildung 3.5 ist in Abbildung 3.6 dargestellt. Vom Startzustand mit der Nummer 3 existiert für das erste Eingabesymbol " eine entsprechend markierte Kante nach Zustand 4. Nach Übergang von 3 nach 4 wird das nächste Symbol betrachtet. a ist in strchar enthalten. Demnach führt die Kante, die mit strchar oder esc beschriftet ist, von 4 nach 4. Für b und c erfolgt die selbe Transition. Für " existiert eine entsprechend markierte Kante von Zustand 4 nach 5. Nummer 5 ist ein Endzustand. Für den String "abc" existiert ein Pfad vom Startzustand 3 nach Endzustand 5 und wird somit von diesem Automaten erkannt. Der Pfad besteht aus Übergängen (=Transitionen) zwischen den Zuständen 3, 4, 4, 4 und 5.

Die hier betrachteten Automaten sind *deterministisch*. Damit ist gemeint, daß bei dieses endlichen Automaten aus keinem Zustand mehr als eine Kante für ein bestimmtes Symbol herausführt. Automaten, die Zustände enthalten, die für mehr als eine Kante mit der selben Markierung Ausgangspunkt sind, nennt man *nichtdeterministisch*. Man

unterscheidet demzufolge zwischen *deterministischen endlichen Automaten*, kurz *DEA* genannt, und *nichtdeterministischen endlichen Automaten*, abgekürzt *NEA*. Enthält ein NEA auch noch Transitionen für das Symbol ε, so spricht man von einem NEA mit ε-Transitionen.

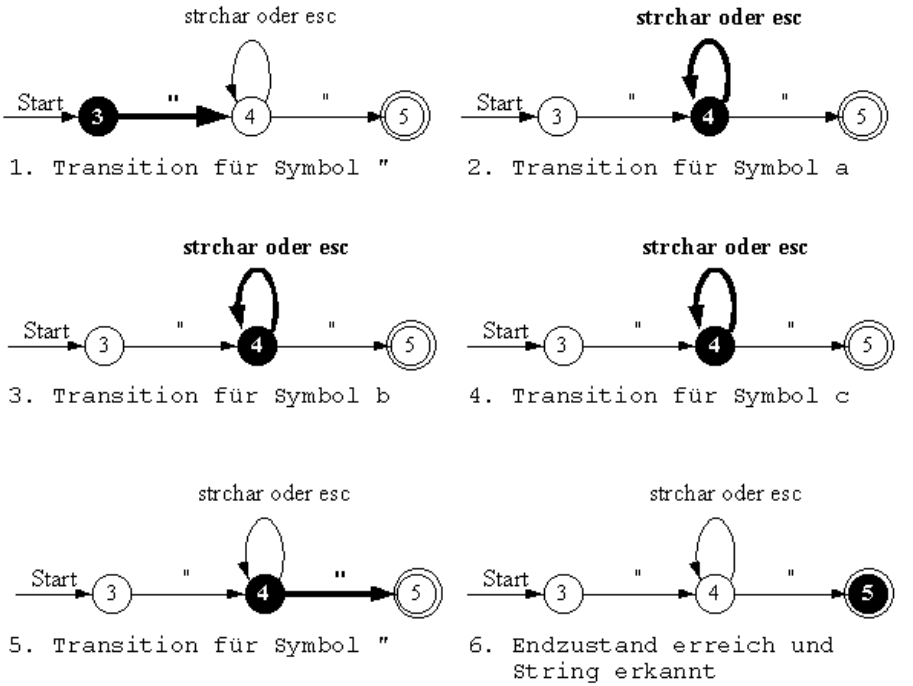


Abb. 3.6: Erkennen des Strings "abc"

Für nähere Informationen zur Automatentheorie sei auf [7], [19] und [2] verwiesen. Implementierungsbeispiele bzw. -hilfen finden sich in [18], [12] und [2], sowie in einfacher rein algorithmenbasierter Betrachtung auch in [5].

3.10 Schlüsselworterkennung

Prinzipiell gibt es zwei Möglichkeiten Schlüsselwörter im Scanner zu erkennen:

1. Behandeln wie Bezeichner und Vergleich mit Schlüsselworttabelle.
2. Einbau in den zentralen Erkennungsautomaten.

Die eine häufig in der Fachliteratur empfohlene Erkennungstechnik ist das Behandeln wie reservierte Bezeichner. Schlüsselwörter gehorchen in der Regel den Bildungsgesetzen für Bezeichner einer Sprache. Deshalb werden Sie meist durch den

gleichen Automaten wie Bezeichner erkannt und anschließend mit einer eigens aufgestellten Schlüsselworttabelle verglichen. Findet sich das erkannte Lexem nicht in dieser Schlüsselworttabelle, so ist es schlicht ein Bezeichner.

Diese Methode hat den Vorteil, daß das System überschaubar und leicht erweiterbar bleibt. Sollten neue Schlüsselwörter hinzukommen, so müssen diese nur in die Schlüsselworttabelle eingetragen werden. Der Nachteil ist, daß das Lexem zweimal analysiert werden muß. Zum einen muß der Automat dieses erkennen und zum anderen muß es mit einer Schlüsselworttabelle verglichen werden.

Eine andere Methode ist das Aufbauen eigener Automaten. Da die einzelnen Teilautomaten im Scanner ohnehin zusammengeführt werden müssen und ein leistungsstarker, zentraler Gesamtautomat entsteht, liegt hier ein großes Optimierungspotential. Die Überschaubarkeit und Wartbarkeit auf reiner Implementierungsebene leidet allerdings darunter. Durch moderne Dokumentationsmethoden läßt sich dieses Manko allerdings teilweise ausgleichen.

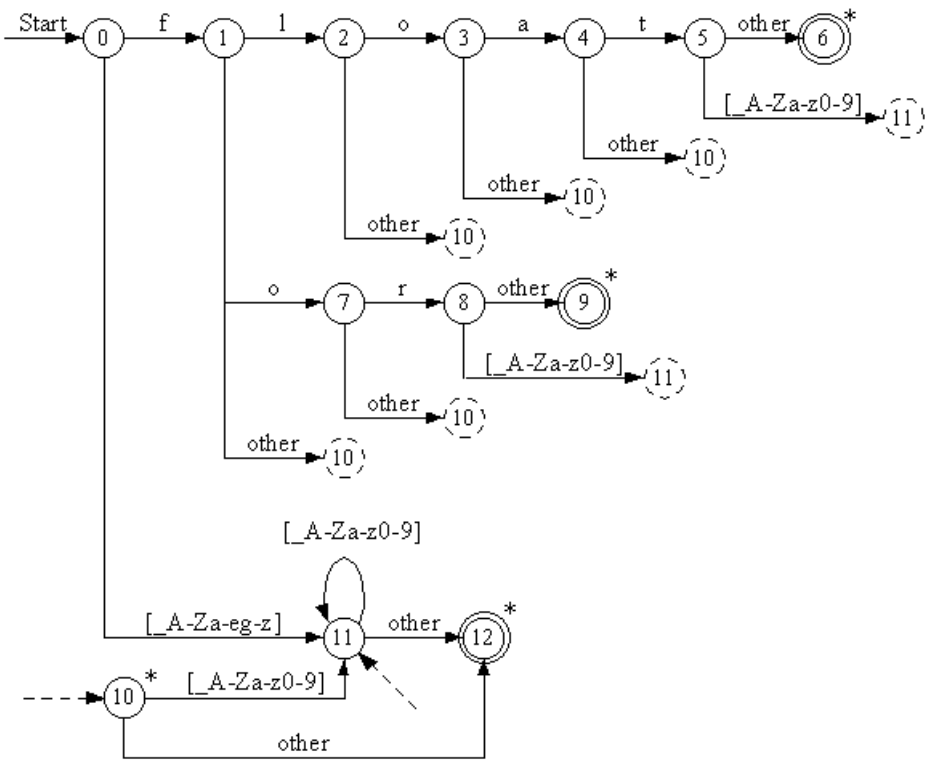


Abb. 3.7: Ein Automat zum Erkennen von float, for und Bezeichnern

Diese Technik basiert im wesentlichen auf der Idee der Multiple-String-Suche wie sie beispielsweise von Binstock und Rex in [5] beschrieben wird. Abbildung 3.7 zeigt einen solchen zentralen Erkennungsautomaten, der Bezeichner und die Schlüsselworte for und float abdeckt. Dieser Automat geht allerdings von reservierten Schlüsselwörtern aus. Bei nicht reservierten Schlüsselwörtern ergeben sich entsprechend komplexere Automaten.

Zugegeben, das Transitionsdiagramm in Abbildung 3.7 ist kompliziert, aber Optimierungen gehen selten mit einer Verringerung der Komplexität einher.

Die Knoten respektive Zustände 10 und 11 erscheinen in diesem Diagramm mehrfach. Einmal mit durchgezogener Linie und mehrfach gestrichelt. Damit soll die Übersichtlichkeit gewahrt bleiben. Normalerweise müßten von den Zuständen 1 bis 4 und 7 direkte Kanten zu 10 und von 5 und 8 nach 11 existieren. Da dies das Diagramm sehr unübersichtlich werden lies, sind für diese Fälle hier die gestrichelt gezeichneten Knoten als „Links“ zum realen Knoten zu verstehen. An den realen Knoten ist das Vorhandensein von Links durch die gestichelten Kanten angedeutet.

Zustand 10 wird immer erreicht, wenn bei teilweise erkannten Schlüsselwörtern ein unerwartetes Zeichen auftritt, das dieses Lexem ggf. als Bezeichner ausweist. In Zustand 11 wird immer übergegangen, wenn ein Zeichen auftritt, das den String eindeutig als Bezeichner ausweist.

Es zeigt zwei Methoden auf den Fall eines Nichtschlüsselwortes zu reagieren, wenn ein teil des Lexem bereits auf ein Schlüsselwort hindeutete. Bei der Knoten-10-Lösungen, landen alle Zeichen, die nicht zum Schlüsselwort gehören, in Zustand 10. Dies umfaßt auch Zeichen, die nicht zu einem Bezeicher gehören. Deshalb wird auch in Zustand 10 der Lookahead zurückgesetzt, um das Zeichen erneut zu betrachten (* beim Knoten). Diese Methode erleichtert dem Entwickler den Umgang mit der Komplexität, da er nicht jeden Spezialfall schon vor Eintritt in den folgenden Zustand (hier 10) berücksichtigen muß. Stattdessen verzweigt er eben für alle unerwarteten Eingabesymbole in diesen Zustand. Dies ist nicht so effektiv, da eine zusätzliche Transition mit Prüfung durch die Funktion δ betrachtet werden muß.

Bei der anderen Lösungen wird hingegen gezielter übergegangen. Für das Zeichen aus der explizit spezifizierten Menge der Bezeichenersymbole, das das erkannte Lexem zum Bezeichner macht, wird nach Zustand 11 übergegangen. Am Ende eines Schlüsselwortes ist dieses Vorgehen auch nötig, da andere Folgezeichen, wie Whitespaces oder Satzzeichen, das Lexem eindeutig als Schlüsselwort identifizieren.

Die erste Lösung bietet sich für häufige Transitionen, wie eben bei jedem Abbruch einer Schlüsselworterkennung, an. Die zweite Technik hingegen eher bei weniger häufig auftretenden Fällen und insbesondere beim Erreichen des Endes eines Schlüsselwortes.

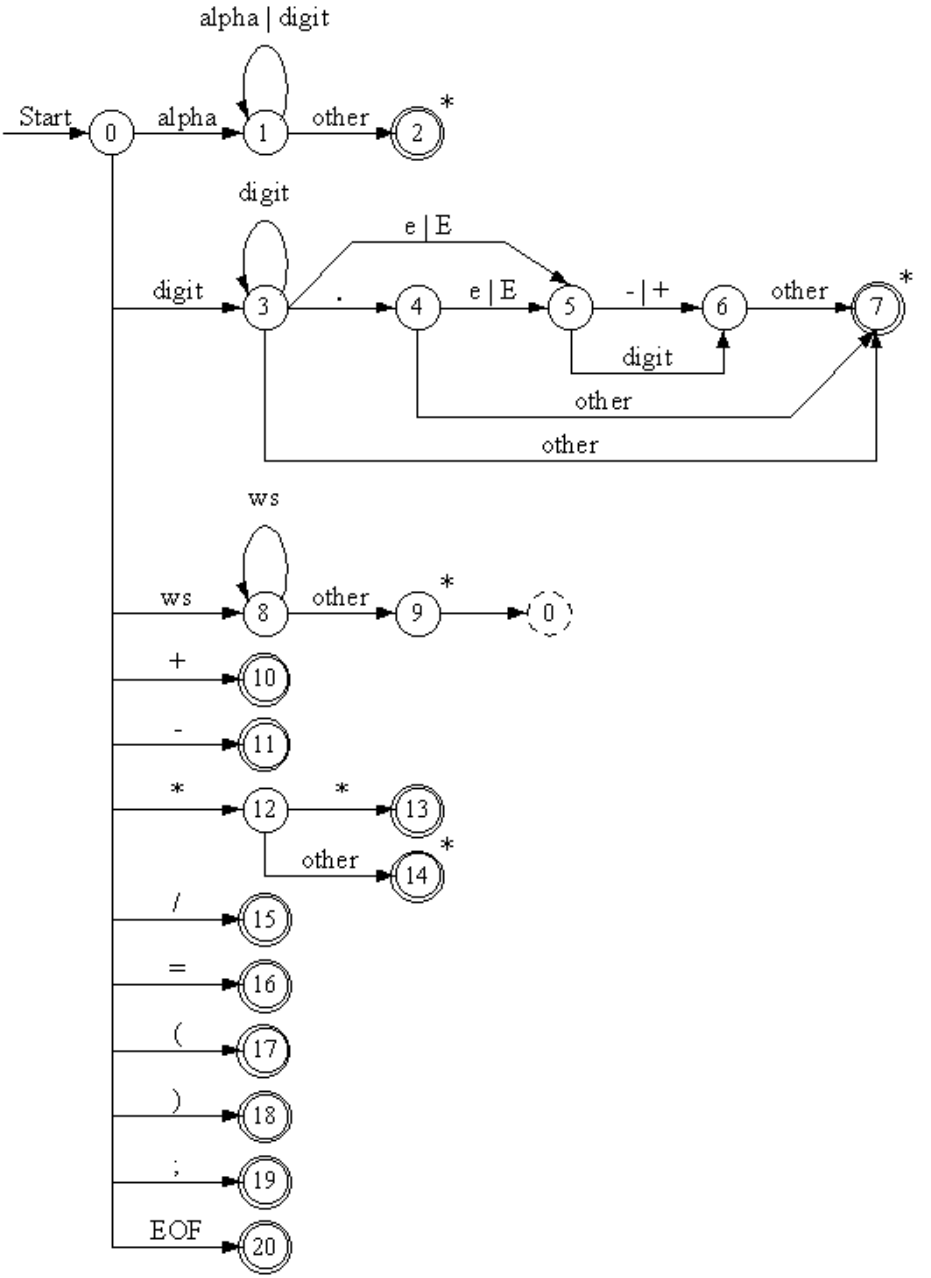


Abb. 3.8: Der endliche Automat für den Scanner.

Insgesamt läßt sich die Methode Schlüsselwörter direkt als Automat zu repräsentieren, ohne den Umweg über eine Schlüsselworttabelle, auf folgenden Nenner bringen: Die Information aus bereits gelesene Zeichen wird direkt weiter berücksichtigt und die Redundanz einer erneuten Überprüfung dieser Zeichen kann unterbleiben. Dieses Prinzip ist auch bei der Stringsuche bekannt. Das Boyer-Moore-Verfahren arbeitet ebenfalls nach diesem Prinzip.

Der einzige Schwachpunkt ist, daß die Erweiterbarkeit ohne ausreichend durchdachtes Design und ohne gute Dokumentation der Entwicklungsphasen darunter leidet. Doch gerade bei Systemen mit einer großen Menge von Schlüsselwörtern dürfte dieses Technik interessant sein, da die Geschwindigkeit bei guten Implementierungen gesteigert werden kann. Außerdem ist diese Methode gut für die Implementierung von Scannergeneratoren geeignet.

3.11 Ein Scanner

Im folgenden wollen wir einen Scanner implementieren, der die obigen Erkenntnisse - wann immer dies sinnvoll - berücksichtigt. Dieser Scanner soll den ersten Schritt in die Richtung einer kleinen mathematischen Programmiersprache darstellen. Als Lexeme sollen Bezeichner für Variablen und Fließkommazahlen nach den Bildungsgesetzen in C++ unterstützt werden. Darüberhinaus sollen die Operatoren und Satzzeichen `+`, `-`, `*`, `**` (Potenzoperator aus Fortran), `/`, `=`, `;` (zum Abschluß von Anweisungen), `(` und `)`, sowie als Whitespaces Zeilenvorschub, Wagenrücklauf, Leerzeichen, horizontaler - und vertikaler Tabulator erkannt werden können.

Als reguläre Definition ergibt sich:

```
<alpha>     --> [_A-Za-z]
<digit>     --> [0-9]

<id>        --> <alpha>(<alpha>|<digit>)*
<num>       --> <digit>*[\.<digit>*][(E|e)[-|\+]<digit>*]
<ws>        --> [ \n\r\v\t]+
<plus>      --> \+
<minus>     --> -
<times>     --> \*
<power>     --> \*\*
<div>       --> /
<assign>    --> =
<bracket1>  --> (
<bracker2>  --> )
<semicolon> --> ;
```

Die abgesetzten untere Definitionen sind die Tokens. <alpha> und <digit> sind lediglich Hilfsdefintionen zur überschaubareren Definition von <id> und <num>. Als spezielles zusätzliches „Quasi-Token“ wird im Scanner noch eof definiert. Dieses Token wird bei Erreichen des Endes der Eingabedatei zurückgegeben.

Wie Sie in Abbildung 3.8 sehen, sind einige Teilautomaten recht komplex, die meisten jedoch sehr schlicht gehalten. Bei der Umsetzung der regulären Defintion in ein Transitionsdiagramm genügen ganze 21 Zustände, um einen Scanner mit den oben genannten Eigenschaften zu implementieren.

Nun stellt sich die Frage, wie Transitionsdiagramme in eine Programmiersprache überführt werden können. Eine recht übersichtliche und nachvollziehbare Variante besteht im wesentlichen aus einer Schleife, in der der aktuellen Zustand ermittelt wird und die entsprechenden Aktionen ausgeführt werden. Diese Aktionen bestehen bei „gewöhnlichen“ Zuständen aus dem Test, welche Transition für welches Eingabesymbol möglich ist. Wird ein passender Übergang gefunden, wird der aktuelle Zustand gemäß der Transition angepaßt, d. h. geändert und die Schleife beginnt von neuem. Falls keine passende Transition existiert, wird das Error-Recovery aktiv. Bei Endzuständen hingegen wird das Token aufgebaut, die Schleife verlassen und das Symbol an den Parser zurückgeliefert.

Damit ergibt sich folgender prinzipieller Aufbau für Scanner, die aus Transitionsdiagrammen gewonnen werden:

```
Wiederhole bis Lexem erkannt wurde:
      Aktueller Zustand ist...
            0: -- gewöhnlicher Zustand
                  Lies Eingagesymbol
                  Wenn Eingabesymbol == a1 dann
                        Transition durchführen:
                              Aktueller Zustand = q1
                  Wenn Eingabesymbol == a2 dann
                        Transition durchführen:
                              Aktueller Zustand = q2
                  ...
                  Wenn Eingabesymbol == aN dann
                        Transition durchführen:
                              Aktueller Zustand = qN
                  Sonst
                        Starte Error-Recovery
            1: -- gewöhnlicher Zustand
                  ...
            ...
            15: -- Endzustand
```

```
                    Lexem ist als Token1 erkannt
                          Setze Attribut
                          Liefere Token1 zurück
               16: -- Endzustand
                    ...
               ...
          Ende der Auswahl
     Ende der Schleife
```

Die Größe derartig codierter Scanner ist direkt proportional zur Anzal der Knoten und Kanten im Transitionsdiagramm. Jeder Knoten respektive Zustand wird durch einen eigenes Codessegment vertreten.

3.11.1 Design und Spezifikation

Bevor wir die konkrete Implementierung betrachten, wollen wir noch einige Fragen zum Design klären und eine Spezifikation aufstellen. Auch der Scanner soll objektorientiert programmiert werden, d. h. er soll in einer gesonderten Objektklasse realisiert werden.

Ein Scanner-Objekt besteht aus folgenden Teilen:

- Lexikalischer Analysierer, der von außen, d. h. vom Parser ansprechbar sein muß (=öffentlich).
- Eingabepufferung, die lediglich im Scanner-Objekt verfügbar sein darf, also geschützt sein muß.
- Geschützte Schnittstelle zum Error-Recovery.
- Geschütztes Interface zur Symboltabelle.

Die Funktionalität der lexikalischen Analyse werden wir in die Methode `getToken()` kapseln. Der gesamte Analysierer kann darin eingebettet werden.

Da unser Scanner für die Eingabepuffer einen zweigeteilten Puffer mit Wächtern verwenden soll, ist hierfür erheblich mehr Verwaltungsaufwand möglich. Die Methode `readChar()` soll hierbei dem Lesen und `unreadChar()` dem Rückgeben eines Zeichens in den Eingabestrom dienen. Darüberhinaus werden eine Reihe von Attributen existieren, die die Administrationsinformationen aufnehmen werden.

Das Error-Recovery ist innerhalb eines Parsing-Systems ein eigenständiges Objekt, das mit Scanner, Parser und folgenden Analysephasen in Assoziationsbeziehung steht. Deshalb ist es nur sinnvoll für dieses selbständige Untersystem eine gesonderte Klasse aufzubauen. Im Scanner-Objekt werden wir daher eine Instanz von ErrorHandler referenzieren und über eine geschützte Methode `error()` ansprechen. Die Implementierung von `ErrorHandler` wollen wir allerdings noch auf einen späteren

Zeitpunkt verschieben. `error()` wird dabei wieder wie `printf()` per variabler Parameterzahl das Formatieren der Fehlermeldungen unterstützen. Dieses Feature macht `error()` sinnvoll, da sich so Redundanzen vermeiden lassen. Andernfalls müßten bei jedem Aufruf des Error-Recovery diese Formatierung jeweils an der betreffenden Stelle erfolgen.

Die Symboltabelle ist ebenfalls als Untersystem aufzufassen. Sie steht ebenfalls in Assoziationsbeziehung mit Scanner, Parser und den folgenden Phasen. Im Scanner-Objekt wird sie daher ebenfalls referenziert. Ihre Klasse wird `SymbolTable` sein. Eine spezielle Methode als Schnittstelle ist hier nicht nötig.

Damit ergibt sich folgende Klasse:

```
class Scanner
{
       public:
              Scanner(SymbolTable *stable,
                      ErrorHandler *ehand);
              ~Scanner();

              TOKEN getToken();

              inline void setInputFile(istream *file) {
                     line = 1;
                     column = 0;
                     inpfile = file;
                     curPart = &part1;
                     part1.alr_read = false;
                     part2.alr_read = false;
                     begin = lookahead = BUF_PART_SIZE;
              }

              inline unsigned long getLine() {
                     return line;
              }
              inline unsigned long getColumn() {
                     return column;
              }

       protected:
              void unreadChar();
              char readChar();

              SymbolTable *symtable;
```

```
        ErrorHandler *errhand;

        istream *inpfile;

        unsigned long line,
                     column;

        void error(const char *format, ...);

        // Zweigeteilter Puffer
        unsigned begin, lookahead;
        char buffer[BUF_PART_SIZE*2+2];
        struct BufferPart {
               char *buffer;
               BufferPart *next;
               bool alr_read;
        } part1, part2, *curPart;

        unsigned lexem_len;
        char *lexem;
        char *getLexem();
};
```

Neben den oben genannten Methoden, existieren noch einige Hilfen und Zusätze. Über die Methode `setInputFile()` läßt sich die Datei festlegen, die als Eingabe dienen soll. `getLine()` und `getColumn()` liefern die aktuelle Zeilen- bzw. Spaltennummer zurück.

Die Methode `getLexem()` operiert über den Attributen `lexem_len` und `lexem`. Dieser Teil von Scanner arbeitet eng mit der Eingabepufferung zusammen. Hat die Methode `getToken()` ein Lexem erkannt, so ruft Sie `getLexem()` auf und erhält so das erkannte Lexem, das durch die Attribute `begin` und `lookahead` der Eingabepufferung beschrieben ist. Das Lexem wird in `lexem` gespeichert und hat die Länge `lexem_len`. Der Rückgabewert von `getLexem()` ist schlicht der Inhalt von `lexem`.

Der Datentyp `TOKEN` ist schlicht das 2-Tupel zur Aufnahme des Symbols und des Attributs eines Token. Er ist wie folgt definiert:

```
typedef struct {
       int sym;
       void *attrib;
} TOKEN;
```

Die Konstante `BUF_PART_SIZE` ist eine per Präprozessordirektive `#define` festgelegter Wert, der die Größe einer Hälfte des zweigeteilten Eingabepuffers angibt. Dieser Wert berücksichtigt nicht den am Ende der Pufferhälfte zu findenen Wächter.

Diese Konstante sollte wie bereits oben erwähnt der Größe eines typischen Blocks des zugrundeliegenden Dateisystems haben, also 1024, 2048 oder 4096.

Die Tokens werden durch Ihren Symbolwert (`sym` in `TOKEN`) beschrieben. Dies ist ein Integer-Wert aus folgender Tabelle. Die Konstanten können mit `#define` angelegt werden.

Wert	*Token*
tEOF	spezielles „EOF-Token“
tNUM	num - Fließkommazahl
tID	id - Bezeichner
tPLUS	plus - Additionsoperator
tMINUS	minus - Subtraktionsoperator
tTIMES	times - Multiplikationsoperator
tDIV	div - Divisionsoperator
tPOWER	power - Potenzoperator
tASSIGN	assign - Zuweisungsoperator
tBRACKET1	bracket1 - Klammer auf
tBRACKET2	bracker2 - Klammer zu
tSEMICOLON	semicolon - Strichpunkt

3.11.2 Implementierung des Scanners

In diesem Abschnitt wollen wir den schrittweisen Aufbau des Scanners, d. h. der Methode `getToken()` betrachten. Das grundlegende Gerüst der Methode ist eine Schleife, in die eine Fallunterscheidung für die jeweiligen Zustände eingebettet ist. Zur Speicherung des aktuellen Zustand wird eine Variable state benötigt. Darüberhinaus müssen noch Variablen für das gerade analysierte Zeichen `c` und zur Aufnahme des aufzubauenden und zurückzuliefernden Tokens angelegt werden.

Infolge der 21 Zustände des endlichen Automaten aus Abbildung 3.8 ergibt sich folgendes Gerüst:

```
TOKEN Scanner::getToken()
{
       int state = 0;
       TOKEN token;
       char c;

       for(;;)
              switch(state) {
                     case 0:
                     case 1:
```

```
                    case 2:
                    ...
                    case 20:
                }
    }
```

Alle Zustände, aus denen Kanten herausführen, müssen das nächste Zeichen für die folgende Analyse lesen. Dies geschieht durch die Methode `readChar()`. Das Ergebnis wird in `c` gespeichert. Anschließend muß eine Fallunterscheidung erfolgen und ggf. zum entsprechenden Folgezustand verzweigt werden. Dieser Übergang zu einem anderen Zustand ist schlicht durch das Zuweisen der Nummer dieses Zustands an die Variable `state` zu bewerkstelligen. Existieren bei diesen Zuständen ein Übergänge für `other`, so ist für diese Transition schlicht in der Fallunterscheidung bei `else` das entsprechende Folgezustand anzugeben. Gibt es jedoch keinen Übergang `other`, so ist für `else` das Error-Recovery aufzurufen.

Alle Endzustände, die mit dem Stern für einen Eingabezeichenrückfluß markiert sind, müssen das Zeichen zuletzt gelesene Zeichen in den Eingabestrom zurückstellen. Hierzu existiert die Methode `unreadChar()`.

In allen Endzuständen ist das Token aufzubauen und zurückzuliefern, d. h. `token.sym` mit dem Symbolwert und `token.attrib` mit dem korrespondierenden Attribut zu belegen, sowie per `return token` das Token an den Parser zurückzugeben.

Im Startzustand sind außerdem die Werte für `lexem_len` und `begin` zu initialisieren.

Damit ergibt sich folgender Code:

```
TOKEN Scanner::getToken()
{
 int state = 0;
 TOKEN token;
 char c;

 for(;;)
  switch(state) {
   case 0:
       lexem_len = 0;
       c = readChar();
       begin = lookahead - 1;
       if(!c)
              state = 20;
       else if(isalpha(c) || c == '_')
              state = 1;
       else if(isdigit(c))
              state = 3;
```

```
    else if(isspace(c)) {
          state = 8;
          if(c == '\n') {
                line++;
                column = 0;
          }
    } else if(c == '+')
          state = 10;
    else if(c == '-')
          state = 11;
    else if(c == '*')
          state = 12;
    else if(c == '/')
          state = 15;
    else if(c == '=')
          state = 16;
    else if(c == '(')
          state = 17;
    else if(c == ')')
          state = 18;
    else if(c == ';')
          state = 19;
    else {
          if(isprint(c))
                error("Zeichen %c nicht erlaubt.", c);
          else
                error("Zeichen 0x%.2x nicht erlaubt.",
                      (unsigned)c);
    }
    break;
  case 1:
    c = readChar();
    if(isalnum(c) || c == '_')
          state = 1;
    else
          state = 2;
    break;
  case 2:
    unreadChar();
    token.sym = tID;
    token.attrib = symtable->update(getLexem());
    return token;
```

```
case 3:
   c = readChar();
   if(isdigit(c))
          state = 3;
   else if(c == '.')
          state = 4;
   else if(c == 'e' || c == 'E')
          state = 5;
   else
          state = 7;
   break;
case 4:
   c = readChar();
   if(isdigit(c))
          state = 4;
   else if(c == 'e' || c == 'E')
          state = 5;
   else
          state = 7;
   break;
case 5:
   c = readChar();
   if(c == '-' || c == '+')
          state = 6;
   else if(isdigit(c))
          state = 6;
   else {
          error("Auf 'E' muss bei Zahlenkonstanten"
                " ein Wert folgen.");
          state = 6;
   }
   break;
case 6:
   c = readChar();
   if(isdigit(c))
          state = 6;
   else
          state = 7;
   break;
case 7:
   unreadChar();
   token.sym = tNUM;
```

```
      token.attrib = (void*) new double;
      *((double*)token.attrib) = atof(getLexem());
      return token;
   case 8:
      c = readChar();
      if(isspace(c)) {
            if(c == '\n') {
                  line++;
                  column = 0;
            }
            state = 8;
      } else
            state = 9;
      break;
   case 9:
      unreadChar();
      state = 0;
      break;
   case 10:
      token.sym = tPLUS;
      token.attrib = 0;
      return token;
   case 11:
      token.sym = tMINUS;
      token.attrib = 0;
      return token;
   case 12:
      c = readChar();
      if(c == '*')
            state = 13;
      else
            state = 14;
      break;
   case 13:
      token.sym = tPOWER;
      token.attrib = 0;
      return token;
   case 14:
      unreadChar();
      token.sym = tTIMES;
      token.attrib = 0;
      return token;
```

```
      case 15:
         token.sym = tDIV;
         token.attrib = 0;
         return token;
      case 16:
         token.sym = tASSIGN;
         token.attrib = 0;
         return token;
      case 17:
         token.sym = tBRACKET1;
         token.attrib = 0;
         return token;
      case 18:
         token.sym = tBRACKET2;
         token.attrib = 0;
         return token;
      case 19:
         token.sym = tSEMICOLON;
         token.attrib = 0;
         return token;
      case 20:
         token.sym = tEOF;
         token.attrib = 0;
         return token;
    }
  }
```

Zum Erkennen der Zeichen kommen die Makros bzw. Funktionen aus `<ctype.h>` `isalpha()`, `isdigit()`, `isalnum()` und `isspace()` zum Einsatz. Hier muß teilweise gerade bei deutschsprachigen Compilern darauf geachtet werden, daß `isalpha()` bzw. `isalnum()` lediglich A bis Z und a bis z und nicht Umlaute als Buchstaben akzeptieren. Das einzige, was sonst passieren kann, ist, daß zum Beispiel `Müller` trotz Umlaut als Bezeichner akzeptiert wird. Im allgemeinen können Sie jedoch dieses Verhalten der Bibliothek Ihres Compilers modifizieren. Falls nicht, so können Sie auf die Makros in Anhang D ausweichen.

Beim Erkennen von Whitespaces mittels `isspace()` (Zustände 1 und 8) wird überprüft, ob ein „Newline“ vorliegt. Ist dies der Fall, so wird der Zeilenzähler `line` inkrementiert.

3.11.3 Eingabepufferung

Der Puffer selbst wird durch das Attribut `buffer` vertreten. Die Verwaltung der Pufferhälften wird durch `part1`, `part2` und `curPart` realisiert. `part1` und `part2` enthalten dabei die zur Administration der Eingabepufferung relevanten Informationen zu den Pufferhälften. Das Element `buffer` dieser Attriubte ist ein Zeiger auf die jeweilige Pufferhälfte von `Scanner::buffer`. `next` zeigt auf die jeweils andere Verwaltungsstruktur, also `part1.next` zeigt auf `part2`, `part2.next` auf `part1`. Das boolesche `alr_read` steht für „already read“, also „bereits gelesen“. Dieses Flag gewinnt im Zusammenhang des Zurückstellens eines Zeichens in den Eingabestrom an Bedeutung. `curPart` („current part“ = „aktueller Teil“) zeigt auf die Verwaltungsstruktur des jeweils gerade aktuellen Pufferteils, also entweder auf `part1` oder `part2`. Abbildung 3.9 veranschaulicht die Zusammenhänge zwischen `part1`, `part2` und `buffer`.

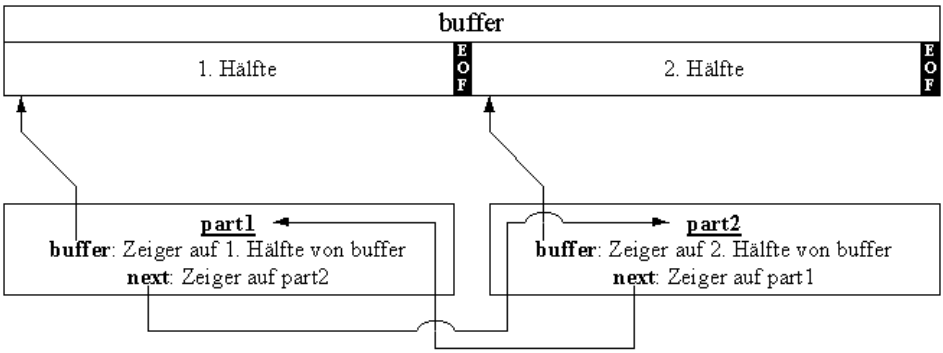


Abb. 3.9: Die Zusammenhänge zwischen dem Puffer und seinen Verwaltungsstrukturen

Die Initialisierung der Pufferverwaltung findet sinnvollerweise im Konstruktor bzw. beim Setzen der Eingabedatei statt:

```
Scanner::Scanner(SymbolTable *stable, ErrorHandler *ehand)
{
       errhand = ehand;
       symtable = stable;
       lexem = 0;
       buffer[BUF_PART_SIZE] = '\0';
       buffer[BUF_PART_SIZE*2+1] = '\0';
       curPart = 0;
       part1.next = &part2;
       part1.buffer = buffer;
       part2.next = &part1;
       part2.buffer = &buffer[BUF_PART_SIZE+1];
}
```

```
...
// Definiert in der Klassendeklaration:
inline void setInputFile(istream *file) {
       line = 1;
       column = 0;
       inpfile = file;
       curPart = &part1;
       part1.alr_read = false;
       part2.alr_read = false;
       begin = lookahead = BUF_PART_SIZE;
}
...
```

Als Wächterzeichen verwenden wir das Nullbyte '\0', da dieses zum einen in den ASCII-Quelltexten nicht vorkommen darf und ohnehin einen Fehler bedeuten würde, zum anderen bei der ASCIIZ-String-Darstellung von C/C++ ohnehin reserviert sein muß. Im Konstruktor werden diese Wächer jeweils am Ende der Pufferhälften in `buffer` positioniert. Außerdem werden die Zeiger `next` der Verwaltungsstrukturen auf die jeweils andere Struktur und die Pointer `buffer` auf die entsprechenden Pufferhälften gesetzt.

`curPart` wird auf Null gesetzt, da eine Pufferhälfte ohne zugeordnete Eingabedatei ohnehin nicht sinnvoll ist. Dies bedeutet zwar grundsätzlich eine Fehlerquelle, aber mit dieser Problematik wollen und können wir uns im Rahmen dieses Buches nicht befassen.

Wird eine Eingabedatei per `setInputFile()` zugeordnet, werden `curPart` auf die Verwaltungsstruktur der ersten Pufferhälfte gesetzt, die Flags `alr_read` der Pufferstrukturen auf `false` gesetzt. `begin` und `lookahead` werden auf den Wächter der ersten Pufferhälfte gesetzt, damit wird `readChar()` beim Lesen automatisch veranlaßt den zweiten Puffer mit Daten aus der Eingabedatei zu füllen. Es ist also nicht notwendig diesen Spezialfall der neu zugeordneten Datei in `readChar()` berücksichtigen. Die entsprechende Abfrage würde erstens ein spezielles Flag und zweitens Abfragecode in `readChar()` erfordlich machen. Dieser Code wäre in den meisten Fällen des Aufrufs von `readChar()` redundant. Sind zur Analyse eines Quelltextes n Lesevorgänge nötig, wäre diese Abfrage in n - 1 Fällen schlicht zeitfressender Overhead!

Zeichen lesen

Dank des zuvor genannten kleinen Optimierungstricks zeigt sich `readChar()` relativ schlank:

```
char Scanner::readChar()
{
```

```
    if(!buffer[lookahead]){
        curPart = curPart->next;
        if(!curPart->alr_read) {
            inpfile->read(curPart->buffer, BUF_PART_SIZE);
            curPart->buffer[inpfile->gcount()] = '\0';
        } else
            curPart->alr_read = false;
        lookahead = curPart->buffer - buffer;
    }
    lexem_len++;
    column++;
    return buffer[lookahead++];
}
```

Ist das aktuell zu betrachtende Zeichen ein Nullbyte, so wird zum erneuten Einlesen verzweigt. Ein Nullbyte ist entweder ein Wächter oder das gekennzeichnete Ende eines Resultats eines Lesevorgangs, der nicht mehr die Länge `BUF_PART_SIZE` liefern konnte.

Zunächst wird zur anderen Pufferhälfte und deren Verwaltungsstruktur verzweigt. Es wird geprüft, ob diese Hälfte schon gefüllt wurde. Dies kann der Fall sein, wenn ein Zeichen zurückgestellt wurde und diese Operation die Puffertrenner überschritt. Doch hierzu später mehr.

Ist die Pufferhälfte bereits gefüllt, so wird lediglich das Flag `alr_read` auf `false` zurückgesetzt, andernfalls werden Daten in den Puffer eingelesen. Das Ende des eingelesen Datenbestands wird mit einem Nullbyte gekennzeichnet. Konnten `BUF_PART_SIZE` Zeichen gelesen werden, wird damit der Wächter überschrieben, was keinen Unterschied macht, da dieser ohnehin ein Nullbyte ist. Wurden hingegen weniger Zeichen gelesen, wird das auf die gelesenen Daten folgende Byte im Puffer mit Null überschrieben. Dies markiert das Ende.

Der Lookahead-Zeiger wird auf das erste Zeichen der Pufferhälfte gesetzt. Die Operation `curPart->buffer - buffer` ergibt für `curPart = &part1` einfach Null und für `curPart = &part2` `BUF_PART_SIZE` + 1.

Was ist aber wenn schon das Ende der Datei erreicht wurde, also die Bedingung in der if-Abfrage sich nicht auf einen Wächter, sondern auf eine Endemarkierung innerhalb des Puffers bezog?

Diese Frage ist einfach beantwortet. Beim Einlesen der Daten in die Pufferhälfte reagiert der Eingabe-Stream `inpfile` damit, schlicht die Anzahl der gelesen Zeichen auf Null zu setzen. Bei der Abfrage mittels `inpfile->gcount()` liefert er Null, damit wird das erste Zeichen der Pufferhälfte auf Null gesetzt. Dieses Nullbyte wird dann am Ende der Methode zurückgegeben. Ein Nullbyte als Zeichen interpretiert der Scanner dann als Dateiendezeichen und generiert das Token `tEOF`. Fazit: Ein korrektes Ergebnis.

Vor dem Beenden der Methode werden schließlich noch die Lexemlänge und die aktuelle Spaltenposition aktualisiert, sowie der Lookahead angepaßt.

Zeichen zurück in den Eingabestrom geben

Durch die zweigeteilte Pufferung ist das Zurückgeben eines Zeichens in den Eingabestrom keine triviale Aufgabe. Ein einfaches Dekrementieren des Lookahead-Zeigers ist nicht immer möglich. Befindet sich der Lookahead nämlich auf dem ersten Zeichen eines Puffers, so würde dieser beim simplen Erniedrigen entweder auf dem Wächter der ersten Hälfte landen, oder in fatalerweise im Datennirvana landen (Unsigned-Wert 0 - 1 ist maximale speicherbare Größe!). Daher ist eine dreifache bedingte Verzeigung für die Fälle

- erstes Zeichen der ersten Pufferhälfte,
- erstes Zeichen der zweiten Pufferhälfte und
- sonstige Position im Puffer

erforderlich.

Im ersten und zweiten Fall muß der Lookahead auf den Wächter der zweiten respektive der ersten Pufferhälfte gesetzt werden. Hier kommt das Flag `alr_read` der Verwaltungsstrukturen ins Spiel. Damit beim erneuten Lesen des Wächters kein erneutes Lesen erfolgt, wird dieses Flag auf `true` gesetzt und in `readChar()` entsprechend berücksichtigt.

Im dritten Fall ist ein Zeichen schlicht irgendwo innerhalb einer Pufferhälfte zu finden. Hier genügt es den Lookahead um eins zu reduzieren.

Damit ergibt sich folgender Code:

```
void Scanner::unreadChar()
{
       if(!lookahead) {
              lookahead = BUF_PART_SIZE*2+1;
              part1.alr_read = true;
       } else if(lookahead == BUF_PART_SIZE+1) {
              lookahead = BUF_PART_SIZE;
              part2.alr_read = true;
       } else
              lookahead--;
       column--;
}
```

`column`, der Spaltenzähler, kann einfach dekrementiert werden, da hier grundsätzlich nach dem Lesen eines Zeichens ein Wert größer oder gleich eins zu finden ist. Also findet sich hier nach dieser Operation ein Wert größer oder gleich null. Null bedeutet dann, daß aus der aktuellen Zeile kein Zeichen bislang gelesen wurde. Dies ist korrekt, wenn es das erste Zeichen der neuen Zeile war.

3.11.4 Ermitteln eines erkannten Lexems

Mit Hilfe der Methode `getLexem()` läßt sich ein Lexem begrenzt durch begin und `lookahead - 1` im Puffer in einen String transformieren. Operiert wird hierzu über dem Attribut `lexem`. Infolge der Zweiteilung des Puffers ist die Implementierung dieser Methode auch nicht in einfacheren Dimensionen zu suchen.

Prinzipiell existieren drei Möglichkeiten, wie sich ein Lexem zusammensetzen kann:

1. Es befindet sich komplett in einer Pufferhälfte.
2. Es beginnt in der unteren und setzt sich in der oberen Hälfte fort.
3. Es beginnt in der oberen und setzt sich in der unteren Pufferhälfte fort.

Die erste Situation ist die einfachste. Hier sind ledliglich alle Zeichen zwischen `begin` und `lookahead - 1` in einen String zu kopieren.

Das zweite Szenario ist ähnlich. Die zu kopierenden Stringhälfte im Puffer sind durch ein Nullbyte - den Wächter der ersten Pufferhälfte - unterbrochen. Beim Kopieren sind Nullbytes einfach zu ignorieren, womit das oben genannte Prinzip angewendet werden kann.

Die dritte Möglichkeit erfordert zwei Kopierschritte. Zunächst sind alle Zeichen der oberen Pufferhälfte, also von `begin` bis zum Wächter, in den String zu übertragen. Anschließend müssen alle Zeichen der unteren Pufferhälfte, also von Null bis `lookahead - 1`, an den String angehängt werden.

Damit ergibt sich ein durch zwei alternative Blöcke geteilter Algorithmus:

```
char *Scanner::getLexem()
{
       if(lexem)
              delete[] lexem;
       lexem = new char[lexem_len+1];
       unsigned n, m = 0;
       if(begin < lookahead) {
              for(n = begin;n < lookahead;n++)
                     if(buffer[n])
```

```
                        lexem[m++] = buffer[n];
        } else {
                for(n = begin;buffer[n];n++)
                        lexem[m++] = buffer[n];
                for(n = 0;n < lookahead;n++);
                        lexem[m++] = buffer[n];
        }
        lexem[m] = '\0';
        return lexem;
}
```

Hinweis: Der an lexem zugewiesene Speicher muß im Destruktor von Scanner freigegeben werden:

```
Scanner::~Scanner()
{
        if(lexem)
        delete[] lexem;
}
```

3.11.5 Fehlerbehandlung

Die Möglichkeiten der Fehlererkennung in der lexikalischen Analyse sind bekanntlich begrenzt. Sie kann sich lediglich auf eindeutig fehlerhafte Lexeme beschränken, die sich in keine Regel einordnen lassen.

Im Beispiel-Scanner existieren nur zwei Punkte, in denen auf Fehler geschlossen werden kann. Nur in diesen beiden Fällen ist ein Error-Recovery denkbar. Diese Stellen finden sich beim Übergang von Zustand 5 nach 6 im Lexem für Fließkommazahlen und bei den Transitionen ausgehend vom Startzustand 0. Beide Zustände - 5 und 0 - besitzen keine Übergänge für `other`. Hier ist die Menge der zugelassenen Zeichen eindeutig.

In beiden Fällen wird beim else der Fallunterscheidung die Fehlerbehandlung aktiviert. Das Error-Recovery besteht schlicht daraus alle ungültigen Zeichen wie Whitespaces zu ignorieren.

Dieses Verfahren erschwert jedoch dem Parser die Arbeit. Denken Sie beispielsweise an einen falsch geschriebenen Bezeichner wie `K,l.ang`. Dies wird wie drei Bezeichner getrennt durch Whitespaces `K l ang` interpretiert. Da allerdings jeweils eine Fehlermeldung für das Komma und den Punkt ausgegeben wird, sind Fehlermeldungen des Parsers infolge inkorrekter Syntax akzeptabel. Die Geschwindigkeit des Error-Recoveries darf schließlich nicht aus den Augen verloren gehen.

Der Aufbau von `error()`, welche als Schnittstelle zum Fehlerbehandlungsobjekt dient, ist ähnlich zu dem der gleichnamigen Methode des Parser-Objekts aus dem vorherigen Kapitel:

```
void Scanner::error(const char *format, ...)
{
       char err_msg[200];
       va_list ap;

       va_start(ap, format);
       vsprintf(err_msg, format, ap);
       va_end(ap);

       errhand->error(line, column, ErrorHandler::etError,
                      err_msg);
}
```

Der einzige Unterschied besteht darin, daß keine Exception mehr geworfen wird. Stattdessen wird die Methode `error()` des Error-Handlers aufgerufen. Neben Zeilen und Spaltennummer verlangt diese auch die Art der Meldung (Warnung, Fehler oder fataler Fehler). Doch hierzu später mehr.

Der wichtige Punkt ist, daß der Error-Handler den Ablauf des Programms nicht unterbricht. Nachdem `ErrorHandler::error()` beendet ist, wird regulär fortgesetzt. Lediglich bei einem fatalen Fehler wird das Programm beendet. Das Error-Recovery hat also im fehlererkennenden Part - hier im Scanner - zu erfolgen.

3.12 Erweiterung des Scanners um Schlüsselworte

Die meisten Computersprachen kennen Schlüsselworte. Im folgenden soll nun gezeigt werden, wie man einen Scanner einerseits durch den Ausbau des Automaten und andererseits durch das Hinzufügen einer Schlüsselworttabelle erweitert.

3.12.1 Erweiterung des Automaten

Die Erweiterung eines Scanners um Schlüsselworte anhand des direkten Integrierens in den Erkennungsautomaten, also ohne den Umweg über Bezeichnererkennung/Schlüsselworttabelle ist Thema dieses Abschnitts. Sie werden sehen, daß infolge der guten Dokumentation mittels Transistionsdiagramm eine Erweiterung kein allzu schwieriges Unterfangen darstellt.

Die folgende Änderung des Automaten soll fähig sein, zusätzlich zu den bereits möglichen Lexemen auch die folgenden Schlüsselwörter zu erkennen:

- *proc* zum Einleiten von Unterprogrammdeklarationen
- *return* zur Rückgabe von Werten aus Unterprogrammen
- *if* und *else* für bedingte Verzweigungen
- *while* für Schleifen
- *read* zum Lesen von Werten von der Tastatur
- *write* zum Ausgeben von Werten auf den Standardausgabekanal

Bevor die Implementation erfolgen kann, muß zunächst das Transitionsdiagramm aus Abbildung 3.8 erweitert werden. Die Modifikationen sehen Sie in Abbildung 3.10, wobei nur der Teilautomat für Bezeichner übertragen wurde. Die anderen Teilautomaten sind ohnehin von den Erweiterungen nicht betroffen.

Der Übergang vom Startzustand 0 zum Zustand 1, also in den Teilautomaten für Bezeichner, ist verändert. Dieser ist nun nur für Eingabesymbole möglich, die der regulären Definition `<alpha>` genügen und zugleich durch die Zeichenklasse `[^ieprw]` definiert sind. Die Konjunktion wird durch das & zwischen den Ausdrücken beschrieben. Da wir mit deterministischen Automaten arbeiten, müssen wir unbedingt so vorgehen. Würde diese Einschränkung nicht vorgenommen, würden sich anderfalls Mehrdeutigkeiten ergeben, die auf den Nichtdeterminismus hindeuten.

Das Zeichen i beispielweise ist in `<alpha>` ebenfalls enthalten. Somit wäre ohne die obige Einschränkung für dieses Eingabesymbol ein Übergang von 0 nach 1 und zugleich von 0 nach 21 möglich. Ein deterministischer endlicher Automat zeichnet sich jedoch dadurch aus, daß für jeden Zustand für ein Eingabesymbol eindeutig nur eine einzige Transition und damit Folgezustand existieren darf.

Die Transitionsvorschrift `other (alpha | digit)` bedeutet, daß hier für jedes andere Zeichen aus `<alpha>` oder `<digit>` verzweigt werden kann. Es ist sozusagen ein auf `(alpha | digit)` beschränktes `other`. Diese Markierung erlaubt es das Diagramm etwas übersichtlicher zu gestalten, da sonst für jeden Fall explizit `(alpha | digit) & [^c]` angegeben werden müßte, wobei `c` das/die Zeichen darstellt, für die bereits Transitionen existieren. Damit würde sich die Anzahl der „Links“ auf den Zustand 1 inklusive entsprechender Kanten um ein erhebliches Maß erhöhen. Das Diagramm wäre nicht mehr zu überschauen.

Der direkte Einbau von Schlüsselwörtern in den Erkennungsautomaten erfordert, da diese den Regeln für Bezeichner gehorchen, mehr Aufwand als für andere Lexeme. Kommt ein mit einem Buchstaben markierter Übergang in einem solchen Teilautomaten bei der Erkennung nicht in Betracht, da das Eingabesymbol mit dem für die Transition benötigten nicht übereinstimmt, so deutet alles auf einen Bezeichner hin.

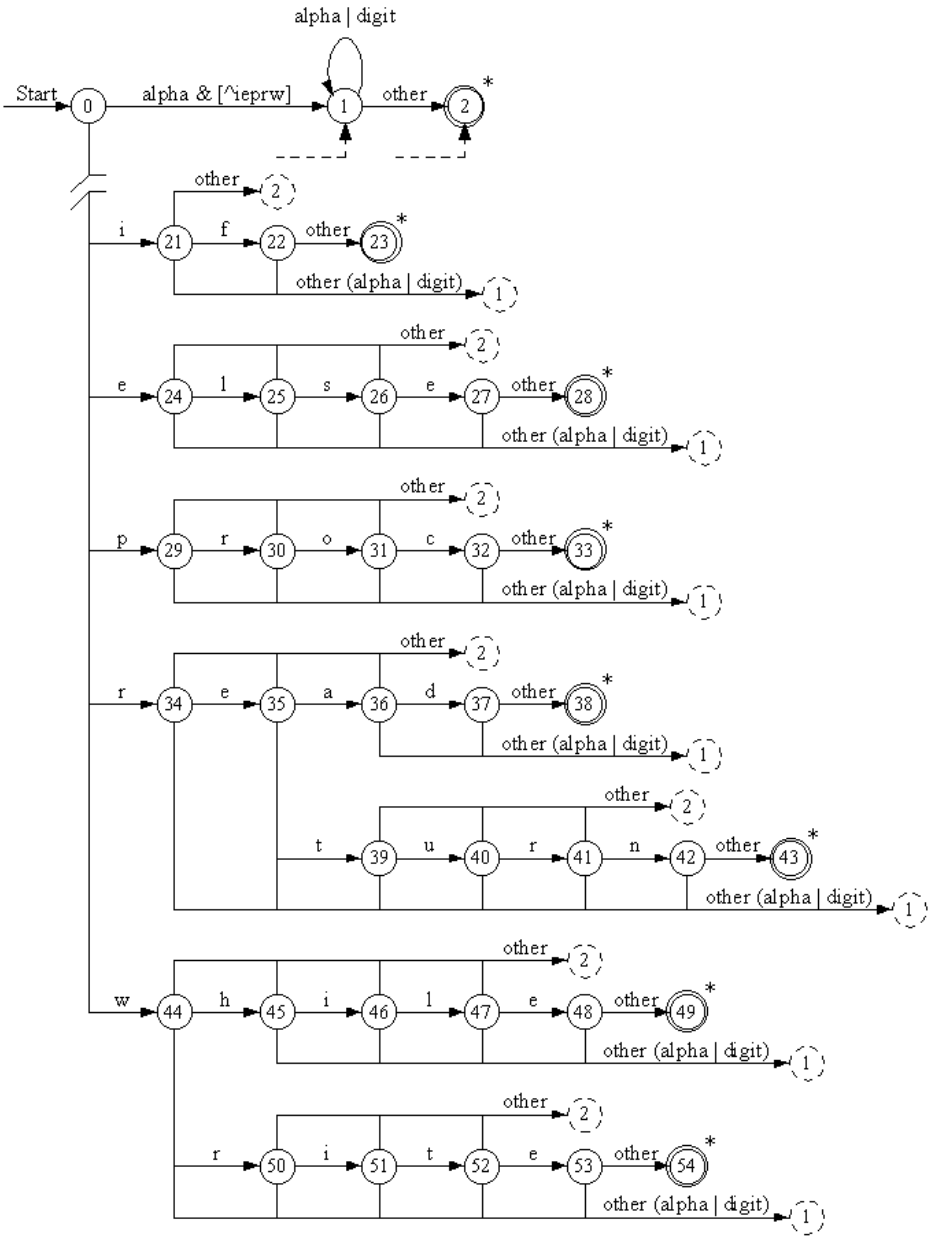


Abb. 3.10: Die Erweiterungen für die Schlüsselwörter

In einem solchen Fall muß in den Teilautomaten für Bezeichner verzweigt werden. Hier kommen prinzipiell die Zustände 1 und 2 in Betracht. Ist das Eingabesymbol ein alphanumerisches Zeichen oder ein Unterstrich, so führt die Transition zu Zustand 1. Ist es hingegen irgendein anderes, dann muß zu Zustand 2 übergegangen werden, da dann das gesamte Lexem, der Bezeichner, bereits vollständig erkannt wurde.

Die Darstellungsmöglichkeiten eines sehr komplexen Automaten für die lexikalische Analyse sind zugegebenermaßen begrenzt. Hier empfiehlt es sich weitere Abkürzungen zu den Links einzuführen. Beispielsweise können Zustände, für die ein Übergang für other (alpha | digit) in den Zustand 1 führt rot und für die eine Transition other nach Zustand 2 existiert blau gekennzeichnet werden. Die Kanten könnten somit auf das nötigtste reduziert werden. Die Erweiterungen für Transitionsdiagramme könnten in einer Legende oder Spezifikation zusammengefaßt werden.

3.12.2 Erweiterung der Implementierung

Die Erweiterung der Implementierung verläuft sehr einfach, aber es ist Tippakrobatik gefragt, da die Realisierung der einzelnen Zustände nach vier Schemata aufgebaut ist.

„Gewöhnliche“ Zustände mit einem Übergang für ein konkretes Zeichen, müssen mit Code nach folgendem Schema aufgebaut werden:

```
case <Zustandsnummer>:
       c = readChar();
       if(c == <Markierung der Kante>)
              state = <Folgezustand>;
       else if(isalnum(c) || c == '_')
              state = 1;
       else
              state = 2;
       break;
```

Die Zustände 35 und 44 mit Transistionen für zwei konkrete Zeichen erfordern die Erweiterung des obigen Codefragments um eine Abfrage für die Markierung der zweiten Kante inklusive Festsetzung des entsprechenden Folgezustands.

Bei Zuständen vor dem Endzustand, also 22, 27, 32, 37, 42, 48 und 53, wird hingegen die Abfrage nach einem konkreten Zeichen ersatzlos gestrichen, woraus sich folgender Codeteil ergibt:

```
case <Zustandsnummer>:
              c =  readChar();
              if(isalnum(c) || c == '_')
                     state = 1;
              else
```

```
                    state = <entsprechender Endzustand>;
                break;
```

Die Endzustände 23, 28, 33, 38, 43, 49 und 54 bauen wieder das entsprechende Token auf. Hierzu müssen neue Symbole bzw. korrespondierende Integer-Konstanten definiert werden. Diese zusätzlichen Konstanten sind

Wert	*Token*
tIF	Schlüsselwort if
tELSE	Schlüsselwort else
tPROC	Schlüsselwort proc
tREAD	Schlüsselwort read
tRETURN	Schlüsselwort return
tWHILE	Schlüsselwort while
tWRITE	Schlüsselwort write

Die Wertzuordnung kann wieder per Präprozessordirektive `#define` erfolgen.

Damit ergibt sich folgender Rahmencode für die sieben neuen Endzustände:

```
case <Zustandsnummer>:
       unreadChar();
       token.sym = <Token-Konstante>;
       token.attrib = 0;
       return token;
```

Nach konsequenter Anwendung dieser Code-Schablonen ergibt sich die folgende, modifizierte Variante der Methode getToken(). Da sich die Zustände aus den angegebenen Rahmencodes leicht aufbauen lassen, sind hier nicht alle Zustände angegeben.

```
TOKEN Scanner::getToken()
{
  int state = 0;
  TOKEN token;
  char c;

  for(;;)
    switch(state) {
      case 0:
             lexem_len = 0;
             c = readChar();
             begin = lookahead - 1;
             if(!c)
                    state = 20;
```

```
        else if(isalpha(c) || c == '_')
                switch(c) {
                        case 'i': state = 21; break;
                        case 'e': state = 24; break;
                        case 'p': state = 29; break;
                        case 'r': state = 34; break;
                        case 'w': state = 44; break;
                        default : state = 1;
                }
        else if(isdigit(c))
                state = 3;
        else if(isspace(c)) {
                state = 8;
                if(c == '\n') {
                        line++;
                        column = 0;
                }
        } else if(c == '+')
                state = 10;
        else if(c == '-')
                ...
        break;
case 1:
        ...
case 21:
        c = readChar();
        if(c == 'f')
                state = 22;
        else if(isalnum(c) || c == '_')
                state = 1;
        else
                state = 2;
        break;
case 22:
        c =  readChar();
        if(isalnum(c) || c == '_')
                state = 1;
        else
                state = 23;
        break;
case 23:
        unreadChar();
```

```
            token.sym = tIF;
            token.attrib = 0;
            return token;
      case 24:
            ...
      case 44:
            c = readChar();
            if(c == 'h')
                  state = 45;
            else if(c == 'r')
                  state = 50;
            else if(isalnum(c) || c == '_')
                  state = 1;
            else
                  state = 2;
            break;
      case 45:
            c = readChar();
            if(c == 'i')
                  state = 46;
            else if(isalnum(c) || c == '_')
                  state = 1;
            else
                  state = 2;
            break;
      case 46:
            ...
      case 48:
            c =  readChar();
            if(isalnum(c) || c == '_')
                  state = 1;
            else
                  state = 49;
            break;
      case 49:
            unreadChar();
            token.sym = tWHILE;
            token.attrib = 0;
            return token;
      case 50:
            c = readChar();
            if(c == 'i')
```

```
                        state = 51;
                else if(isalnum(c) || c == '_')
                        state = 1;
                else
                        state = 2;
                break;
        case 51:
                ...
        case 53:
                c =  readChar();
                if(isalnum(c) || c == '_')
                        state = 1;
                else
                        state = 54;
                break;
        case 54:
                unreadChar();
                token.sym = tWRITE;
                token.attrib = 0;
                return token;
    }
}
```

Die nachträgliche Erweiterung eines Erkennungsautomaten ist kein Hexenwerk - vorausgesetzt eine entsprechende Dokumentation (idealerweise ein Transitionsdiagramm) liegt vor.

3.12.3 Alternative Ansätze mittels Transitionsdiagramm

Diese Methode der direkten Umsetzung eines Transitionsdiagramms in Code hat allerdings auch ihre Grenzen. Die Grenzen liegen hier zwar eher darin begründet, daß der Quelltext sehr schnell unüberschaubar werden kann.

Technische Grenzen, d. h. Grenzen des Systems, sind wohl eher kaum zu befürchten. Selbst beim mittlerweile betagten Systemveteran DOS dürfte das 64-KB-Limit wohl nicht so schnell in Kraft treten - von modernen Systemen zu schweigen. Auch die Umsetzung des switch-Befehls von C++ wird von aktuellen Compilern per Sprungtabelle gelöst, so daß zur Verzweigung zu einem Zustand lediglich eine Abfrage nötig ist (im Gegensatz zu verschachtelten if-Statements!).

Zugegeben, je mehr Zustände, desto größer wird der Code. Der Geschwindigkeitsvorteil ist jedoch nicht von der Hand zu weisen. Womit wieder die grundlegende Frage nach dem Mittelmaß zwischen Größe und Geschindigkeit eines Programms aufgeworfen wäre.

Eine andere Möglichkeit ein Transitionsdiagramm umzusetzen besteht darin, die Zustände in einer Tabelle zu erfassen. Wie wir gesehen haben, können Zustände in Kategorien bzw. Klassen eingeordnet und durch ähnlichen Code ausgewertet werden. Trotz einer großen Zahl von Zuständen, sind die Eigenschaften einiger Zustände doch ähnlich und daraus ergeben sich nur wenige Klassen. Somit ist ein üblicher Ansatz die Zustände mit Ihren Eigenschaften in einer Tabelle zu erfassen. Der Scanner operiert bei diesem Ansatz über dieser Tabelle. Die ständigen Zugriffe auf eine im Speicher liegende Tabelle erhöht jedoch den Laufzeitbedarf des Algorithmus. Gerade durch das derzeitige Leistungsgefälle zwischen Prozessorleistung und RAM-Geschwindigkeit darf dieses Problem nicht vernachlässigt werden. Gerade mit dem Scanner steht und fällt bekanntlich die Effizient eines Parsing-Systems. Hier sollte man also genau abwägen.

Auch ein objektorientierter Ansatz für die Verwaltung der Zustände als Objekte wäre denkbar, könnte sich allerdings sehr negativ auf die Effizienz des Scanners auswirken.

3.12.4 Analyse anhand Schlüsselworttabelle

Last but not least soll nun noch die Analyse anhand einer Schlüsselworttabelle gezeigt werden. Schlüsselwörter sollen also vom Automaten als Bezeichner erkannt werden und mittels einer Schlüsselworttabelle identifiziert werden.

Die Modifikationen für diesen Mechanismus sind naturgemäß gering, da der Erkennungsalgorithmus in Form des Teilautomaten für Bezeichner bereits existiert. Es wird lediglich eine Tabelle mit den Schlüsselwörtern und den korrespondierenden Interger-, sprich Symbolwerten und ein darüber Algorithmus benötigt, der das erkannte Bezeichnerlexem mit den Schlüsselwörtern vergleicht.

Die Schlüsselworttabelle muß dabei nicht in die Scanner-Klasse gelegt werden, da nicht jedes Scanner-Objekt seine eigende Schlüsselworttabelle benötigt. Vielmehr bietet sich eine statische, konstante Implementierung im Modul der Scanner-Klasse an. Der Vergleichsalgorithmus wird idealerweise in jedem Scanner-Objekt eigens instanziiert, da über dem Attribut `lexem` operiert werden muß.

Diesen Vergleichsalgorithmus legen wir in einer geschützten Methode idType() der Scanner-Klasse an. Vor dieser Methode definieren wir außerdem die Schlüsselworttabelle:

```
static const struct {
       const char *keyword;
       int sym;
} keytable[] = {
       { "if", tIF },
       { "else", tELSE },
       { "proc", tPROC },
```

```
        { "read", tREAD },
        { "return", tRETURN },
        { "while", tWHILE },
        { "write", tWRITE },
        { 0, 0 }
};

int Scanner::idType()
{
        int n = 0;
        getLexem();
        while(keytable[n].keyword) {
                if(!strcmp(lexem, keytable[n].keyword))
                        return keytable[n].sym;
                n++;
        }
        return tID;
}
```

In dem Algorithmus wird in einer Schleife sukzessive das Lexem mit den Einträgen der Schlüsselworttabelle verglichen. Kommt es zu einer Übereinstimmung wird der betreffende Symbolwert zurückgegeben, andernfalls `tID` für Bezeichner.

Die Veränderungen an `getToken()` beschränken sich auf den Zustand 2, den Endzustand des Teilautomaten für die Bezeichnererkennung:

```
case 2:
        unreadChar();
        if((token.sym = idType()) == tID)
                token.attrib = symtable->update(lexem);
        else
                token.attrib = 0;
        return token;
```

Identifiziert `idType()` das Lexem als Bezeichern, so wird in das Attribut des Token mit dem Verweis auf den Eintrag der Symboltabelle gefüllt. Im Falle eines Schlüsselwortes ist dieses Token-Attribut nicht wichtig und wird auf Null gesetzt.

Aus dieser Technik resultieren gut überschaubare und leicht nachvollziehbare Scanner. Sie ist auch sehr flexibel, da Änderungen im Schlüsselwortschatz sehr schnell und ohne Probleme vorgenommen werden können. Bei der direkten Realisierung der Schlüsselwörter als Automaten ist dies nicht der Fall. Allerdings wird dieser Vorteil mit einem Geschwindigkeitsnachteil erkauft.

Gerade in experimentellen Umgebungen und Prototypen ist diese Technik sehr komfortabel.

4 Syntaktische Analyse

Die Aufgabe eines Parsers ist es den Symbolstrom des Scanners dahingehend zu überprüfen, ob diese Symbolfolge von der Grammatik erzeugt werden kann. Es macht aus den Wörtern der lexikalischen Analyse Sätze mit Aussagekraft. Der Parser liefert für spätere Analysephasen ein geeignetes Format.

4.1 Funktionsprinzip von Parsern

An einen Parser werden gewisse Forderungen gestellt, deren Erfüllung für den Erfolg des späteren Produkts entscheidend sein können. So wie das Gewicht des Scanners auf seinem Laufzeitbedarf liegt, so beruhen die Erwartungen an den Parser neben seiner Turn-Around-Zeit auf seiner Kommunikativität gegenüber dem Benutzer. Die Basis eines korrekten und effizienten Parsing-Prozesses vorausgesetzt, beruht der Erfolg eines Parsers auf seiner Auskunftsfreudigkeit bei - und Behebungsfähigkeit von Fehlern.

Die Leistungsfähigkeit eines syntaktischen Analysierers resultiert im wesentlichen aus der Grammatik und der Art wie er im Fehlerfall reagiert. Sie repräsentieren zwei Hauptkriterien moderner Software - funktioniale Möglichkeiten und Anwenderfreundlichkeit. Der dritte Gesichtpunkt, dem sich moderne Programme stellen müssen, ist die Effektivität mit der Probleme unter Zuhilfenahme der Software gelöst werden können. Die Effektivität resultiert dabei aus dem funktionalen Möglichkeiten und der Effizienz der Implementierung, also im Falle von Parsern aus der algorithmentechnischen Umsetzung der Grammatik.

Da die Qualität eines Parsers direkt von der Grammatik abhängt, muß diese

- frei von Fehlern sein und
- in einer geeigneten Form für die gewählte Parsing-Technik vorliegen.

Das heißt also, um einen effizienten Parser konstruieren zu können, benötigt der Entwickler Methoden zur Überprüfung der Fehlerfreiheit der Grammatik und gegenenfalls zur Transformation der Grammatik in geeignete Formen für die gewählte Parsing-Strategie. Sie wissen bereits aus dem zweiten Kapitel, daß nicht jede Grammatik mit jeder Parsing-Methode arbeiten kann. So setzen Top-Down-Parser beispielsweise Grammatiken ohne Linksrekursionen voraus.

Die Fehlerbehandlung - ein dem Benutzer sehr nahes Teilsystem - sollte

- aussagekräftig, d. h. präzise und verständlich, sein,
- nicht nach dem ersten Fehler abbrechen (Error-Recovery) ,
- sich von Fehler möglichst schnell erholen und
- die Analyse korrekter Programme nicht unverhältnismäßig verlangsamen.

Diese Anforderungen an die Fehlerbehandlung deuten schon an, daß hier eine Gradwanderung nötig ist. Es gilt Ausführlichkeit versus Geschwindigkeit abzuwägen.

4.2 Grammatiken überprüfen

Die Grundlage für einen korrekt arbeitenden Parser ist eine fehlerfreie Grammatik. In der Praxis wird kaum die gesamte Grammatik einer Sprache verifiziert. Man beschränkt sich darauf eventuelle Problemherde zu überprüfen.

Für diese problematischen Teile erstellt oder destilliert man kleine kompakte Grammatiken bzw. Grammatikparts. Diese Teilgrammatiken werden danach mit mathematischen Methoden analysiert. Hierzu wird der Beweis geführt, daß die Grammatik G auch wirklich die Sprache L(G) erzeugt.

Dieser Beweis besteht darin, allgemein zu zeigen, daß jeder String w, den die Grammatik G erzeugt, in L(G) enthalten ist und vice versa, daß jeder String w aus L(G) auch tatsächlich durch G erzeugt wird.

Betrachten wir hierzu die Grammatik G = (V, T, P, S) mit V = {A}, T = {a, b, ε}, P = { A --> aAb, A --> ε} und S = A. Es handelt sich hier um das Problem, ob durch diese Grammatik die korrekt in a und b geklammerten Ausdrücke beschrieben werden. Das Problem ist dabei auf den einfachsten Fall reduziert. Etwaige Konkatenationen von geklammerten Ausdrücken werden hier nicht berücksichtigt.

Hier ist nun ein Beweis durch vollständige Induktion über einen String w zu führen. Der Induktionsstart |w| = 0 (Länge des Strings ist Null) ist sehr simpel. Für |w| = 0 gilt die Ableitung A => ε, also der Leerstring, was korrekt ist.

Für eine beliebige Länge 2n mit $n \geq 1$ ergibt sich die Ableitungsfolge

$A => aAb => aaAbb => ... => a^{n.-1}Ab^{n-1} => a^nAb^n => a^n\varepsilon b^n => a^nb^n$

Somit gilt |w| = 2n. Die Produktion A --> aAb wird n Mal angewendet, gefolgt von einem einmaligen Anwenden der Produktion A --> ε. Bei jeder Anwendung der ersten Produktion bleibt die Anzahl von A gleich. Nachdem die zweite Produktion zur Anwendung kam, ist Zahl von A im Satz um eins vermindert. Das Nichtterminal A spielt

in der folgenden Betrachtung keine Rolle mehr. Da ε das neutrale Element ist, kann es entfallen. Somit resultieren wirklich nur korrekt in a und b geklammerte Ausdrücke, die die Länge 2n haben. Die Grammatik G bezeichnet demnach L(G) = { $a^n b^n$ | $n \geq 1$ }.

Vice versa gilt, daß jeder korrekt geklammerten String w mit |w| = 2n für $n \geq 1$ aus G herleitbar und in L(G) enthalten ist. w beginnt ohne Zweifel mit einem a und endet mit einem b. w kann demnach äquivalent als axb beschrieben werden, wobei x wiederum ein korrekt geklammerter String ist. Da |x| = 2(n-1) < 2n gilt, kann x infolge der Induktionsannahme durch A hergeleitet werden: w => A => aAb => axb

Damit ist bewiesen, daß auch w = axb durch A herleitbar ist.

4.3 Grammatiken transformieren

Grammatiken müssen häufig in andere Formen überführt werden, da zum Beispiel die gewählte Parsertechnik entsprechende Ansprüche an die Eigenschaften einer Grammatik stellt, oder die Grammatik mehrdeutig ist. In diesem Abschnitt werden wir uns mit der Transformation von Grammatiken befassen.

4.3.1 Mehrdeutigkeiten eliminieren

Ein Problem, daß beim Entwurf von Grammatiken immer wieder auftaucht, sind Mehrdeutigkeiten. Eine mehrdeutige Grammatik definieren Sippu und Soisalon-Soininen in [19] wie folgt:

„A grammar G is *ambiguous* if some sentence in L(G) has more than one leftmost derivation in G. Otherwise, G is *unambiguous*."

Sehr deutlich wird der Begriff der Mehrdeutigkeit anhand folgender Produktion einer Grammatik G:

```
<stmt> --> if <cond> then <stmt>
         | if <cond> then <stmt> else <stmt>
         | <other>
```

Eine Anweisung aus L(G) wie

```
if c1 then if c2 then s1 else s2
```

hätte zwei mögliche Ableitungen:

```
<stmt> => if <cond> then <stmt>
       => if <cond> then if <cond> then <stmt> else <stmt>
       => if c1 then if c2 then s1 else s2
```

und

```
<stmt> => if <cond> then <stmt> else <stmt>
```

```
=> if <cond> then if <cond> then <stmt> else <stmt>
=> if c1 then if c2 then s1 else s2
```

Die Unterschiede zwischen diesen beiden Ableitungen tritt in den Ableitungsbäumen, dargestellt in den Abbildungen 4.1 und 4.2, gut hervor.

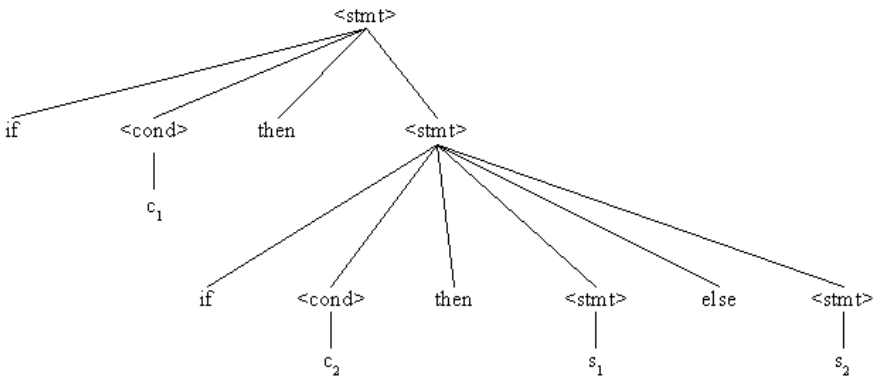


Abb. 4.1: Erste mögliche Ableitung für „if c_1 then if c_2 then s_1 else s_2“

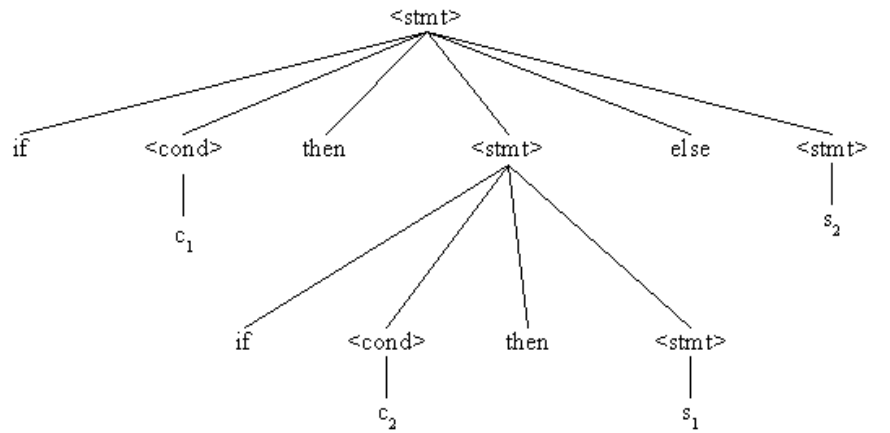


Abb. 4.2: Zweite mögliche Ableitung für „if c_1 then if c_2 then s_1 else s_2“

Ein sehr extremer Fall der Mehrdeutigkeit wäre beispielsweise folgende Grammatik:

```
A --> a
    | A
```

Für den schlichten Satz a wären hier unendlich viele Ableitungen denkbar:

```
A => a
A => A => a
```

```
A => A => A => a
...
```

Zugegeben dieser konkrete Fall ist rein theoretischer Natur. Aber durch Zyklen in der Grammatik kann es zu derartigen Problemen kommen.

Das ein Parser, der solche Mehrdeutigkeiten zuläßt, nicht das „Gelbe vom Ei“ sein kann dürfte einleuchten. Zugegeben bei der Implementierung muß sich der Entwickler ohnehin entscheiden, wie er ein solches Konstrukt wie die obigen behandeln will. Aber laut dieser Grammatik könnte jeder Entwickler selbst entscheiden, wie er einen solchen Fall angehen wollte.

Stellen Sie sich vor die Sprache C++ würde als Standard mit einer solchen Produktion für if definiert. Es gäbe keine Quelltexte, die mit jedem Compiler zusammenarbeiten.

Ergo: Derartige Mehrdeutigkeiten gilt es zu eliminieren.

Alle Programmiersprachen, die bedingte Verzweigungen in der obigen Form zulassen, orientieren sich am Parse-Baum in Abbildung 4.1. Mit anderen Worten: Das „lose“ (=dangling) Else wird grundsätzlich dem letztmöglichen If zugeordnet.

Diese Lösung läßt sich durch folgende Grammatik beschreiben (nach [2]):

```
<stmt>  --> <stmt1>
          | <stmt2>
<stmt1> --> if <cond> then <stmt1> else <stmt1>
          | <other>
<stmt2> --> if <cond> then <stmt>
          | if <cond> then <stmt1> else <stmt2>
```

Mehrdeutigkeiten können prinzipiell in zwei Kategorien eingeordnet werden:

1. Durch Rekursionen und Zykeln verursachte Mehrdeutigkeiten.
2. Durch redundante Mehrfachdefinitionen hervorgerufene Mehrdeutigkeiten.

Erstere können, wie oben geschehen, durch geschicktes Aufspalten in mehrere eigenständige Produktionen der Grammatik und/oder durch Setzen von prädiktierenden Terminale, z. B. Schlüsselwörter, ausgelöst werden.

Im zweiten Fall ist hingegen ein Zusammenführen der Produktionen, die im wesentlichen das gleiche beschreiben nötig. Hier muß darauf geachtet werden, daß sich nicht wieder Mehrdeutigkeiten des ersten Falls ergeben.

4.3.2 Linksrekursionen eliminieren

Top-Down-Parser benötigen Grammatiken, die frei von Linksrekursionen sind. Da das Groß der per „Hand“ implementierten Parser nach dem Top-Down-Prinzip arbeiten, kommt dem Entfernen von Linksrekursionen in Grammatiken eine wichtige Rolle zu.

Betrachten wir hierzu folgende linksrekursive Grammatik G für arithmetische Ausdrücke:

```
<expr>   --> <expr> + <term>
           | <term>
<term>   --> <term> * <factor>
           | <factor>
<factor> --> num
           | ( <expr> )
```

Ein Top-Down-Parser kann die resultierende Sprache L(G) nicht deterministisch analysieren.

Da die Sprache L(G) sich jedoch selbst für eine Top-Down-Analyse eignet, da Sie eine LL(1)-Sprache ist, muß lediglich eine Transformation G' der Grammatik G gefunden werden.

Diese LL(1)-Grammatik G' wäre:

```
<expr>   --> <term> <expr'>
<expr'>  --> ε
           | + <term> <expr'>
<term>   --> <factor> <term'>
<term'>  --> ε
           | * <factor> <term'>
<factor> --> num
           | ( <expr> )
```

Die Methode, nach der die rechtsrekursive Grammatik G' gewonnen wurde, ist sehr simpel. Eine linksrekursive Produktion der Form

```
A --> A α
    | β
```

wird in Regeln der Form

```
A  --> β A'
A' --> ε
     | α A'
```

umgewandelt.

Bei Linksrekursion über Zykeln gestaltet sich das Auflösen etwas komplizierter. Betrachten wir hierzu folgende allgemein formulierte Grammatik G:

```
A --> B α
    | ε
B --> A β
    | γ
```

Die beiden Nichtterminale A und S sind jeweils über das Symbol andere Symbol linksrekursiv definiert. Eine äquivalente Grammatik G' wäre hier:

```
A --> A β α
    | γ α
    | ε
B --> A β
    | γ
```

Die Linksrekusion des Symbols B ist damit beseitigt. Sämtliche Produktionen von B wurden anstatt des Nichtterminals B direkt in A eingebaut. Nun kann die Linksrekurivität von A durch das einfache Anwenden der obigen Methode beseitigt werden. Damit entsteht die Grammatik G":

```
A  --> γ α A'
     | A'
A' --> β α A'
     | ε
B  --> A β
     | γ
```

Sie sehen, daß die Produktion A --> ε gestrichen wurde. Durch A --> A' und A' --> ε entsteht automatische eine indirekte ε-Produktion für A. Die direkte ε-Produktion für A ist hier redundant.

Das Nichtterminal B und seine Produktionen können im übrigen entfallen, sofern sie nicht an anderer Stelle noch referenziert werden.

4.3.3 Linksfaktorisierung

Bei prädiktiven Parsern ergeben sich teilweise Probleme, wenn für ein Nichtterminal zwei alternative Produktionen vorliegen, aber durch den aktuellen Lookahead nicht entschieden werden kann, welche Alternative nun angewendet werden muß. Selbst bei einer Auflösung etwaiger Mehrdeutigkeiten ist das prädiktive Analysesystem nicht in der Lage den korrekten Weg aufgrund seines Eingabesymbols festzulegen.

Betrachten wir hierzu erneut das Dangling-Else-Problem. Die aufgrund des Token `if` kann der prädiktive Parser nicht festellen, welche alternative Produktion von `<stmt>` nun korrekt ist:

```
<stmt> --> if <cond> then <stmt>
         | if <cond> then <stmt> else <stmt>
```

```
          | <other>
```

Selbst nach dem Auflösen der Mehrdeutigkeiten in der folgenden Grammatik

```
<stmt>  --> <stmt1>
          | <stmt2>
<stmt1> --> if <cond> then <stmt1> else <stmt1>
          | <other>
<stmt2> --> if <cond> then <stmt>
          | if <cond> then <stmt1> else <stmt2>
```

, ist der Parser nicht fähig zu entscheiden, ob `<stmt1>` oder `<stmt2>` richtig ist. Auch bei einer Entscheidung zugunsten `<stmt2>` steht er wieder vor der gleichen Frage.

Betrachten wir daher im folgenden die einfacher zu überschauende erste Grammatik.

Die Lösung des Problems ist die sogenannten *Linksfaktorisierung*, eine Grammatiktransformation speziell für prädiktive Parser. Hierzu wird der Produktionsteil, der den Unterschied definiert in eine neue Produktion expandiert. Die Produktionen werden die Unterschiede reduziert durch das neu gewonnene Nichtterminal ersetzt. Somit ergeben sich identische Produktionen, die nur noch durch eine Produktion ausgedrückt werden müssen.

Formal bedeutet dies folgendes: Die prädiktiv nach Lesen von α nicht analysierbare Produktion

```
A --> α β1
    | α β2
```

wird zu den Produktionen

```
A  --> α A'
A' --> β1
     | β2
```

transformiert.

Für das obige Dangling-Else-Problem entsteht konkret die folgende Grammatik:

```
<stmt> --> if <cond> then <stmt> <else>
         | <other>
<else> --> else <stmt>
         | ε
```

Bei einem prädiktiven Parser läßt sich die Mehrdeutigkeit dieser Grammatik sehr komfortabel lösen. Da das prädiktive Parsing-Prinzip darauf aufbaut aufgrund des aktuell betrachteten Symbols die korrekten Produktionen zu finden, kann ein prädiktiver Parser das Dangling-Else-Problem leicht der Ableitung gemäß Abbildung 4.1 den Vorzug geben. Trifft er nämlich auf das Else-Token schreibt er dieses dem aktuellen If, also dem letztmöglichen, zu.

Dies Ändert jedoch nichts an der Mehrdeutigkeit der Grammatik. Die Mehrdeutigkeit wird lediglich in der Implementierung behoben. Im Sinne einer eindeutigen und zweifelsfreien Sprachspezifikation sollte jedoch die Mehrdeutigkeit bereits in der Grammatik behoben werden.

4.3.4 Kontextsensitve Konstrukte

Es wurde bereits an anderer Stelle angedeutet, daß es sprachliche Konstrukte bzw. Sprachen gibt, die weder durch einen regulären Ausdruck, noch durch eine kontextfreie Grammatik angegeben werden können. Hierbei handelt es sich um kontextsensitive Sprachkonstrukte.

Diese speziellen kontextsensitiven Konstrukte müssen von Parser oder Scanner entsprechend überprüft werden. Im folgenden wollen wir einige dieser Fälle exemplarisch betrachten.

Der Hollerith-String aus älteren Fortran-Versionen ist nicht kontextfrei. Die Form $nHa_1a_2a_3...a_n$ des Hollerith-Strings verlangt, daß die Länge des nach dem H folgenden Substring dem vor dem H angegebenen Zahl entspricht. Hier ist eindeutig eine Abhängigkeit vom Kontext gegeben. Die Prüfung, ob dieses Token „Hollerith-String“ korrekt ist, könnte direkt im Scanner erfolgen.

Der folgende Teil einer kontextfreien Grammatik beschreibt den Aufruf einer C-Prozedur.

```
<proc_call> --> id ( <args> )
<args>      --> ε
              | <args> , <expr>
              | <expr>
```

Die Syntax ist damit allgemein beschrieben. Ein Call wie

```
myproc(10, 10);
```

wäre syntaktisch korrekt. Allerdings ist dieser Sprachteil davon abhängig, ob `myproc()` überhaupt Argumente unterstützt und wenn ja, ob es dann wirklich zwei sind und jeweils Integers akzeptiert werden.

Diese kontextsensitiven Aspekte müssen vom Parser bzw. von einem semantischer Analysierer gesondert Betrachtung finden.

Auch müssen in Sprachen, wie C++ oder Pascal, überprüft werden, ob ein Bezeichner überhaupt zuvor deklariert wurde und ob er hier verwendet werden darf (Sichtbarkeit, „Information Hiding“).

4.4 Top-Down-Syntaxanalyse

Die Top-Down-Syntaxanalyse beruht darauf für einen Strom von Eingabesymbolen Linksableitungen zu finden. Anders ausgedrückt: Beginnend beim Startsymbol wird ein Parse-Baum von der Wurzel in Richtung der Blätter konstruiert.

4.4.1 Recursive-Descent-Analyse

Die allgemeinste Form der Top-Down-Syntaxanalyse ist die *Recursive-Descent-Analyse*, die nach dem Prinzip des rekursiven Abstiegs arbeitet. Hier ist es im Gegensatz zur prädiktiven Syntaxanalyse nötig die Eingabe gelegentlich zurückzusetzen (=*Backtracking*), also ein Eingabesymbol mehrfach zu lesen und zu betrachten. Bei der *prädiktiven Syntaxanalyse* kann der Parser für jedes Eingabesymbol eindeutig bestimmen, welche Produktion anzuwenden ist.

Betrachten wir hierzu folgende Grammatik:

```
<A> -->  x <B> y
<B> --> a b
      | a c
```

Für den Eingabestring `xacy` wollen wir eine Recursive-Descent-Analyse durchgehen. Für das Startsymbol `<A>` ergibt sich die Ableitung `x<B>y`. Daraufhin testet der Parser für `<B>` die Ableitung `ab`. Nach der Überprüfung des Eingabesymbol `c` stellt er jedoch fest, daß diese Produktion nicht anwendbar ist. Nun erfolgt ein Backtracking, d. h. die Eingabesymbole `a` und `c` werden wieder in den Eingabestrom zurückgestellt und die andere Produktion für `<B>` wird untersucht. Nun wird festgestellt, daß die Symbole `a` und `c` auf diese Ableitung passen. Nach dem Test des letzten Eingabesymbols `y` wird eine erfolgreiche Analyse gemeldet.

Zu beachten ist bei der Recursive-Descent-Anaylse, daß auch hier trotzt Backtracking die Grammatiken frei von Linksrekursionen sein müssen. Selbst durch die Möglichkeit des Rücksetzens sind Recursive-Descent-Parser vor unterminierten Rekursionen, d. h. Endlosschleifen, nicht gefeit!

Die Recursive-Descent-Analyse mit Backtracking ist in der Praxis der Computersprachen nicht von Bedeutung. Jede Grammatik einer künstlichen Sprache kann in eine Form überführt werden, die sich für die prädiktive Syntaxanalyse eignet. Ein ineffizientes Backtracking ist nicht nötig.

4.4.2 Prädiktive Syntaxanalyse

Wie zuvor erwähnt ist bei der prädiktiven Syntaxanalyse kein Backtracking nötig, da dieses Verfahren für jedes Eingabesymbol eindeutig die passende Produktion ermitteln kann. Damit jedoch diese Analysetechnik zum Einsatz kommen kann, muß die

Grammatik zusätzliche Eigenschaften aufweisen. Hierzu sind gegenenfalls Grammatiktransformationen durchzuführen.

Eine Grammatik für einen prädiktiven Parser muß nicht nur frei von Linkrekursionen, sondern auch linksfaktorisiert sein.

Betrachten wir erneut die Grammatik aus dem vorherigen Abschnitt. Das Problem sind hier die Produktionen für <B>, die eine prädiktive Vorgehensweise verhindern. Durch einfaches Linksfaktorisieren läßt sich dieses jedoch lösen:

```
<A>  --> x <B> y
<B>  --> a <B'>
<B'> --> b
       | c
```

Nun kann für jedes Eingabesymbol eindeutig die anzuwendende Produktion ermittelt werden. Liegt erneut der Eingabestring xacy vor, so erhalten wir die eindeutige Ableitung:

```
<A> => x<B>y => xa<B'>y => xacy
```

Durch Betrachten eines einizigen Symbols kann nun auf die anzuwendende Produktion geschlossen werden.

4.4.3 Transitionsdiagramme in der Syntaxanalyse

Im vorherigen Kapitel haben wir die automatentheoretischen Zusammenhänge der lexikalischen Analyse durch Transistionsdiagramme dargestellt. Diese Diagramme können auch für die Darstellung des Plans oder Datenflusses der Produktionen in der syntaktischen Analyse genutzt werden.

Dies ist in keiner Weise überraschend, da sich die Analyse-Prozesse für die Lexik und die Syntax einer Sprache von einer abstrakteren Ebene betrachtet nicht gravierend unterscheiden. In der lexikalischen Analyse werden die Tokens durch eigene Sprachen, die durch reguläre Grammatiken bestimmt sind, beschrieben. Die Syntaxanalyse beschreibt hingegen den gesamten analysierbaren Text durch eine umfassende Sprache, die *Zielsprache*. Diese Zielsprache kann wiederum als Aggregat von Teilsprachen, nämlich den einzelnen Produktionen und den Lexemen.

Verfolgt man diese Idee konsequent weiter, so kommt man schließlich zu dem Schluß, daß die lexikalische Analyse den Mikroprozeß und die syntaktische Analyse den Makroprozeß des Parsings darstellt. Diese etwas unkonventionellere Sichtweise auf die Thematik schlägt sich in der Fachliteratur sonst nur indirekt in der Aussage nieder, daß die lexikalische Analyse in einer Subroutine des Parsers zu kapseln ist, also einen Unterprozeß der Syntaxanalyse bildet.

In der objektorientierten - oder auch modularen Sichtweise kann die Teilung in Scanner und in Parser somit als Mittel zum Umgang mit einer Grundeigenschaft von Software, der inhärenten Komplexität, angesehen werden.

Die Grundfunktion von lexikalischer - und syntaktischer Analyse ist demnach die selbe. Die Unterschiede beschränken sich auf ihrer Komplexität. Für die Beschreibung der Lexik genügen reguläre Grammatiken, obwohl auch kontextfreie Grammatiken herangezogen werden können. Die Definition der Syntax hingegen bedarf der kontextfreien Grammatik. Der theoretische Analyseprozeß anhand dieser Grammatik ist jedoch aus einer abstrakten Ebene betrachtet nicht wesentlich unterschiedlich. Die verwendete Strategie des Prozesses wird jedoch von der Komplexität diktiert.

Im Grunde kann die lexikalische Analyse als abgespeckte Top-Down-Analyse betrachtet werden. Es wird von links nach rechts gelesen und versucht Linksableitungen (reguläre Defintion --> Ausdruck) zu bilden. Für diesen Prozeß können automatentheoretische Darstellungen in Form von Transitionsdiagrammen genutzt werden.

Was liegt also näher, als für die syntaktische Top-Down-Analyse ebenfalls dieses Beschreibungsmittel einzusetzen?

Die Transitionsdiagramme für die Syntaxanalyse unterscheiden sich, aufgrund der höhreren Komplexität der Syntax gegenüber der Lexik, in einigen Punkten von denen der lexikalischen Analyse:

- Für jedes Nichtterminal existiert ein eigenes Diagramm.
- Die Kanten können sowohl mit Terminalen als auch Nichtterminalen beschriftet sein. Im Falle eines Terminals muß schlicht, ähnlich wie bei den „lexikalischen Transitionsdiagrammen“, das Eingabesymbol übereinstimmen. Das Eingabesymbol ist beim Parsing allerdings kein simples Zeichen, sondern ein Token. Bei Nichtterminalen muß in den jeweiligen Automaten verzweigt werden. Dies ist in der Implementierung durch einen Methoden-Call realisiert.

In einem solchen Transistionsdiagramm für ein Nichtterminal existiert somit für jede Produktion ein Pfad vom Start- zum Endzustand des Teilautomaten. Die Beispielgrammatik des vorherigen Abschnitts sehen Sie als Automaten in den Transitionsdiagrammen der Abbildung 4.3 dargestellt. Die „Verzweigungen“ in die anderen Teilautomaten sind hier zur Unterscheidung von Tokens nach wie vor in spitze Klammern gesetzt.

Bei der Analyse startet der prädiktive Parser im Startzustand des Startsymbols (in Abbildung 4.3 durch den Pfeil „Start“ angedeutet). Trifft er auf eine Kante mit passendem Token, so setzt er den Eingabezeiger um eine Position nach vorn und geht in den entsprechenden Folgezustand über. Kommt er zu einer Kante, die mit einem Nichtterminal beschriftet ist, so geht er in den entsprechenden Teilautomaten über. Er

geht zum Startzustand des Automaten des Nichtterminals über. Erreicht er in einem solchen Teilautomaten den Endzustand kehrt er zurück in den ursprünglichen Automaten und führt dort die mit dem Nichtterminal beschriftete Transition durch, geht also in den dortigen Folgezustand über. Für ε-Transitionen kann direkt in den Folgezustand übergegangen werden, der Eingabezeiger wird hier selbstverständlicht nicht inkrementiert. Die Syntaxanalyse gilt als erfolgreich abgeschlossen, wenn er den Endzustand des Automaten des Startsymbols erreicht hat.

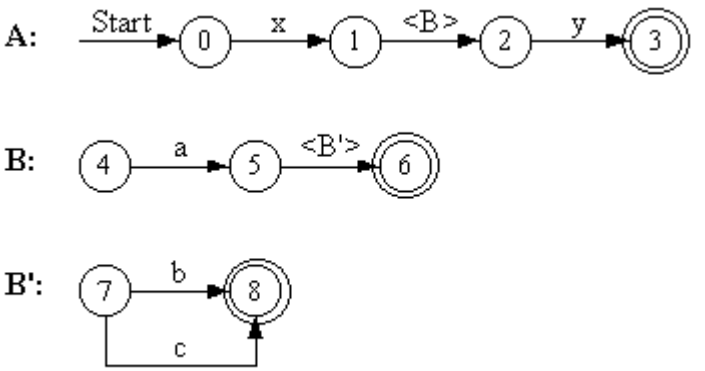


Abb. 4.3: Ein Transitionsdiagramm für die Syntaxanalyse

Aufgrund des „vorhersagenden“ Analyseprinzips müssen die Automaten für prädiktive Parser deterministisch sein. Beim allgemeinen Prinzip der Recursive-Descent-Analyse, also mit Backtracking, sind auch nichtdeterministische Automaten denkbar. Die Rückkehr zu einem früheren Zustand ist bei NEAen, die bekanntlich mehrere von einem Zustand ausgehende Kante mit identischer Markierung erlauben, erforderlich, wenn sich eine gewählte Transition nicht als die richtige entpuppt. Diese Rückkehr im Automaten ist mit dem Backtracking gleichzusetzen.

Die prädiktive Analyse ist prinzipiell ein deterministischer Vorgang. Die Recursive-Descent-Analyse kann hingegen auch Nichtdeterminismen enthalten.

4.4.4 Vereinfachen und Optimieren der Automaten

Die Automaten, die Sie direkt aus der kontextfreien Grammatik gewinnen können, sind selten die optimale Lösung. Kleinere Teilautomaten, wie Sie aus Produktionen, die durch die Linksfaktorisierung entstanden, gewonnen werden, können häufig in die „aufrufenden“ Automaten eingesetzt werden. Bei jeder dieser Optimierungen entfallen so zwei Zustände, die sonst bei der Syntaxanalyse zu durchlaufenden wären.

Abbildung 4.4 zeigt beispielsweise wie der Automat aus Abbildung 4.3 in vier Schritten von ursprünglich neun Zuständen auf fünf reduziert werden kann. Kleine Teilautomaten

werden in größere eingesetzt und anschließend die resultierenden Zustände mit reinen ε-Transitionen entfernt.

In Kapitel 2 haben wir bereits einen prädiktiven Parser implementiert. Die Grammatik haben wir dort zunächst in der Notation der kontextfreien Grammatiken notiert und sukzessive vereinfacht, bis sie als EBNF vorlag.

Ausgangsbasis für unseren kleinen Taschenrechner war folgende kontextfreie Grammatik:

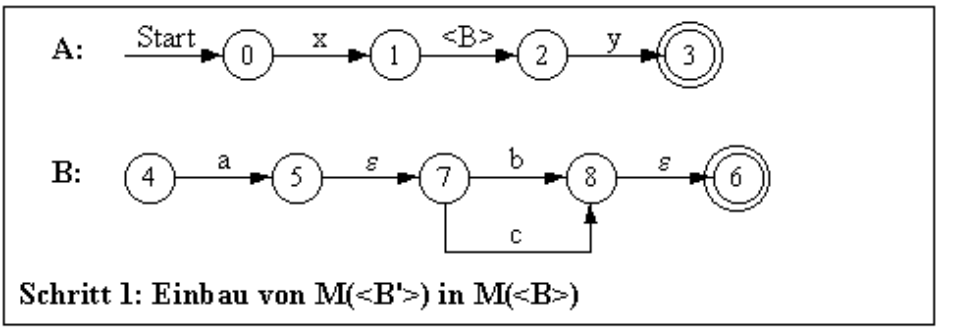
```
<Ausdruck> --> <Term> + <Ausdruck>
<Ausdruck> --> <Term> - <Ausdruck>
<Ausdruck> --> <Term>
<Term>     --> <Faktor> * <Term>
<Term>     --> <Faktor> / <Term>
<Term>     --> <Faktor>
<Faktor>   --> Zahl
<Faktor>   --> + Zahl
<Faktor>   --> - Zahl
<Faktor>   --> ( <Ausdruck> )
```

In dieser Form ist diese Grammatik nicht für die prädiktive Syntaxanalyse geeignet. Es liegen zwar keine Linksrekursionen vor, aber die Grammatik ist nicht linksfaktorisiert. Die Produktionen für die Nichtterminale `Ausdruck` und `Term` sind hier das Problem.

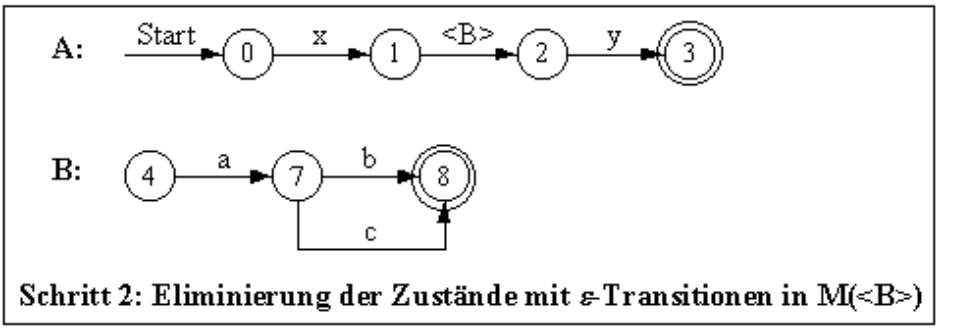
Die EBNF, die schließlich in leicht modifizierter Form auch die Grundlage für unseren prädiktiven Parser bildete, hat diese Probleme nicht mehr:

```
<Ausdruck> ::= <Term> [ {"+","-"} <Ausdruck> ]
<Term>     ::= <Faktor> [ {"*","/"} <Term> ]
<Faktor>   ::= [ {"+","-"} ] Zahl
             | "(" <Ausdruck> ")"
```

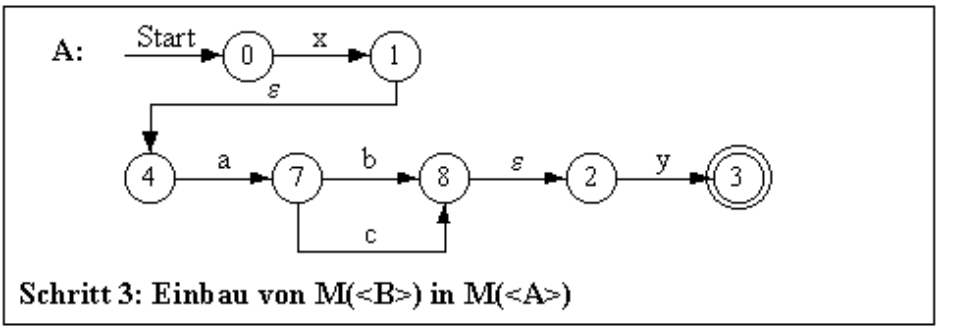
Die korrespondierenden Automaten sehen Sie in Abbildung 4.5. Aus der EBNF ergeben sich lediglich drei Automaten. Hätte man hingegen die kontextfreie Grammatik von oben linksfaktoriert, so hätten sich aufgrund der prädiktiv nicht analysierbaren Konstrukte in `Ausdruck` und `Term` zwei zusätzliche Automaten ergeben. In Abbildung 4.5 sind diese jedoch schon in den Automaten für `Ausdruck` und `Term` integriert. Es handelt sich um die Teilautomaten zwischen den Zuständen 1 und 3 respektive 5 und 7.



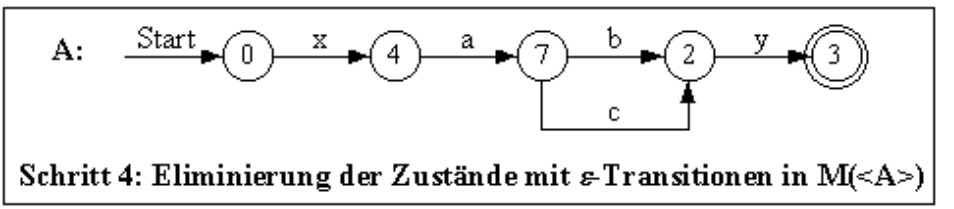
Schritt 1: Einbau von M(<B'>) in M(<B>)



Schritt 2: Eliminierung der Zustände mit ε-Transitionen in M(<B>)



Schritt 3: Einbau von M(<B>) in M(<A>)



Schritt 4: Eliminierung der Zustände mit ε-Transitionen in M(<A>)

Abb. 4.4: Optimierung der Automaten

Die Optimierungen und Vereinfachungen, die sich schon beim Umsetzen einer kontextfreien Grammatik in eine EBNF ergeben, reduzieren den Arbeitsaufwand für die Optimierung der Automaten erheblich. Linksrekursionen müssen zwar selbst bei der EBNF durch entsprechende Transformationen beseitigt werden, die Linksfaktorisierung kann jedoch teilweise schon implizit bei der Umsetzung in die EBNF erfolgen. Linksfaktorisierung, wie sie durch optionale Konstrukte (vergleiche Dangling-Else-Problem) nötig werden würden, lassen sich komplett durch die Beschreibungsmittel der EBNF ohne zusätzliche Produktionen auflösen.

Die Automaten in Abbildung 4.5 sind dadurch, daß sie aus einer EBNF gewonnen wurden, nicht zwangsläufig optimal. Gerade die integrierten Teilautomaten der optionalen Konstrukte in `Ausdruck` und `Term` können vereinfacht werden. Die Rekursion kann in diesen Automatenteilen direkt ausgedrückt werden, also ohne eine Kante, die mit `Ausdruck` respektive `Term` markiert ist. Die entsprechenden Optimierungen, die die Anzahl der Zustände im System von 13 auf 11 reduziert sehen Sie in Abbildung 4.6.

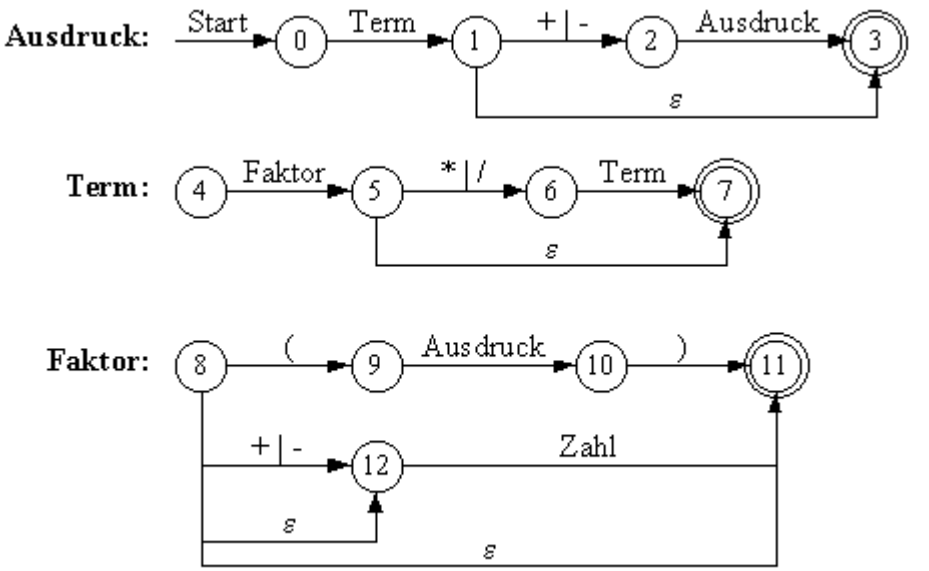


Abb. 4.5: Der Automat für die Taschenrechner-EBNF

Diese offensichtliche Optimierung durch Reduzierung der Zustände und damit des Codes wirkt sich zur Laufzeit noch viel stärker positiv aus. Der simple Ausdruck 1+1 wird vom Automatensystem aus Abbildung 4.5, durch die Transitionsfolge

```
0 -> 1 -> 2 -> 0 -> 1 -> 3 -> 3
```

ausgewertet. Durch den Automaten aus Abbildung 4.6 hingegen genügt die folgende Zustandsfolge:

```
0 -> 1 -> 0 -> 1 -> 3
```

Die Rekursion über eine Verzeigung in den ursprünglichen Automaten erweist sich somit als ineffizent. Das Anfertigen derartiger Transitionsdiagramme für den Parser ist also nicht nur eine Darstellungsmöglichkeit, sondern ist eine hervorragende Methode Optimierungen aufzustöbern und umzusetzen.

Die Optimierungsmöglichkeiten sind mit dem Transitionsdiagramm in Abbildung 4.6 noch nicht erschöpft. Der Automat für das Nichtterminal Faktor könnte beispielsweise ohne Probleme in dern Automaten für Term eingesetzt werden. Auch Term könnte in Ausdruck integriert werden. Sie sollten allerdings bedenken, daß größer angelegte Optimierungen immer eine Quelle für Fehler sein können. Außerdem darf die Übersichtlichkeit des Programms im Hinblick auf die Wartung und Erweiterung nicht zu sehr leiden. Hier ist wieder eine Gradwanderung zu bewerkstelligen.

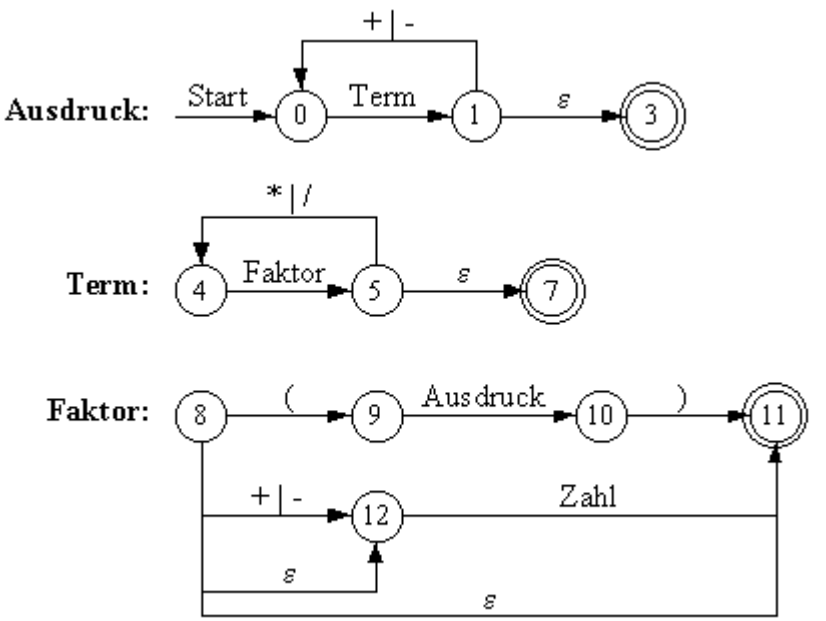


Abb. 4.6: Die optimierten Automaten

4.4.5 Rekursive prädiktive Parser

Die rekursive Implementierung eines prädiktiven Parsers haben Sie bereits in Kapitel 2 kennengelernt. Das zugrundeliegende Prinzip ist schlicht für jedes Nichtterminal eine Prozedur anzulegen, die die jeweilige(n) Ableitung(en) testet bzw. umsetzt. Terminale in den Ableitungen werden mit dem Lookahead verglichen und bestimmen (=prädiktives Prinzip) die anzuwendenden Produktionen. Nichtterminale Symbole in den Ableitungen werden in entsprechende Prozeduraufrufe umgesetzt.

Die Basis für die Implementierung kann allerdings auch statt einer kontextfreien Grammatik ein optimiertes Automatensystem wie die zuvor besprochenen sein. Die einzelnen Automaten werden dabei jeweils in Prozeduren umgesetzt. Aus dem Transistionsdiagramm kann die algorithmische Struktur ersehen werden.

Zustände mit mehreren ausgehenden Kanten stellen bedingte Verzweigungen dar. ε-Transitionen repräsentieren im Zusammenspiel mit anderen Kanten, die vom gleichen Zustand ausgehen, die Else-Konstrukte in bedingten Verzweigungen oder identifizieren die anderen Kanten als Optionen. Reine ε-Transitionen, d. h. vom Zustand geht nur eine mit ε markierte Kante aus, werden durch einfache Zustandsänderungen, die an keine Bedingung geknüpft sind, implementiert. Mit Nichtterminalen markierte Kanten werden zu Prozeduraufrufen. Abbildung 4.7 zeigt die zu den Konstrukten korrespondierenden Codeteile.

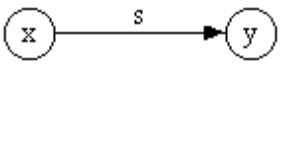
4.4.6 Nichtrekursive prädiktive Parser

Die rekursive Implementation prädiktiver Syntaxanalysierer ist schnell und einfach. Es ist jedoch auch möglich diese Analysetechnik nichtrekursiv umzusetzen. Die Grundidee für nichtrekursive prädiktive Parser ist, statt der impliziten Stapelverwaltung der verwendeten Programmiersprache und des Betriebssystems infolge rekursiver Routinenaufrufe, über einem explizit realisierten Stapel zu operieren. Die Zuordnung von Produktionen zu einem Nichtterminal, also die Implementierung der Linksableitung, erfolgt hier durch eine *Parse-Tabelle*. Syntaxanalysesysteme, die nach diesem Prinzip arbeiten, nennt man *tabellengesteuerte prädiktive Parser*.

Ein solcher tabellengesteuerter prädiktiver Parser verfügt über folgende Komponenten:

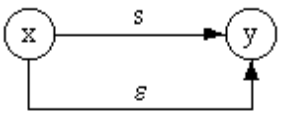
- Ein Eingabepuffer zur Übergabe der Symbole. Er ist durch ein Symbol oder Quasi-Token EOF zum Kennzeichnen des Endes abgeschlossen. Dieser Puffer kann durch einen Symbolstrom liefernden Scanner ersetzt werden.
- Ein Stapel mit einem Endezeichen zum Markieren der untersten Zelle. Die anderen Zellen nehmen Nichtterminale und Terminle auf.
- Eine Parse-Tabelle M(v, t) für Nichtterminale v und Terminale t oder EOF.

- Ein Ausgabestrom, der die Ergebnisse der Syntaxanalyse an folgende Phasen weiterleitet.



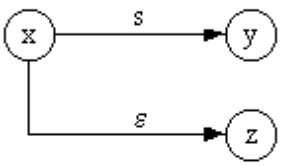
Einfache Transition:

```
if Lookahead = s then
  Zustand = y
else
  Fehler
```



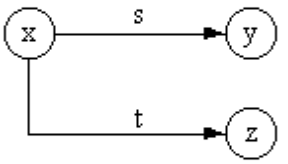
Alternativer Block (Option):

```
if Lookahead = s then
  Option verarbeiten
Zustand = y
```



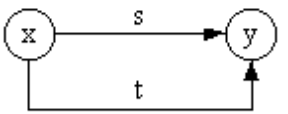
Bedingte Verzweigung mit Default:

```
if Lookahead = s then
  Zustand = y
else
  Zustand = z
```



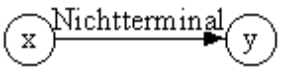
Bedingte Verzweigung ohne Default:

```
if Lookahead = s then
  Zustand = y
else if Lookahead = t
  Zustand = z
else
  Fehler
```



Alternative Blöcke ohne Default:

```
if Lookahead = s then
  Alternative verarbeiten
  Zustand = y
else if Lookahead = t
  Alternative verarbeiten
  Zustand = y
else
  Fehler
```



Prozeduraufruf:

```
Nichtterminal()
Zustand = y
```

Abb. 4.7: Codegewinnung aus Transitionsdiagrammen

Der Eingabepuffer enthält die zu analysierende Symbolkette. Der Stapel enthält eine Folge von Grammatiksymbolen. Das Endezeichen EOF markiert das untere Ende des Stapels. Bei Beginn der Analyse enthält dieser Stapel lediglich das Endezeichen und darüber das Startsymbol der Grammatik. Die Parse-Tabelle M(v, t) ist ein zweidimensionales Feld. Sie bildet auf der Grundlage der Grammatik G(V, T, P, S) die Terminale $t \in T$ und die Nichtterminale $v \in V$ aufeinander ab.

Abbildung 4.8 verdeutlicht die Zusammenhänge zwischen den einzelnen Komponenten.

Der Parser arbeitet nach folgendem Algorithmus: Er liest das oberste Symbol x vom Stapel und das erste Eingabesymbol c. Die Auswertung dieser beiden Symbole geschieht wie folgt:

1.$x \in T \cup \{ EOF \}$. x ist ein Terminal oder EOF:

a) $x = c = EOF$. Das Ende der Syntaxanalyse ist erreicht. Der erfolgreiche Abschluß wird gemeldet.

b) $x = c \neq EOF$. Das Terminal a ist erkannt. x wird vom Stapel entfernt und der Eingabezeiger um eins erhöht bzw. ein neues Eingabesymbol c gelesen.

2.$x \in V$. x ist ein Nichtterminal. Der Eintrag M(x, a) wird gelesen:

a) $M(x, a) \in P$. Der Eintrag der Parse-Tabelle ist eine Produktion. Der Stapeleintrag für x wird entfernt. Ist die Produktion keine ε-Produktion, dann werden die Symbole der Ableitung auf den Stapel gelegt. Ist M(x, a) = x --> ABC, so werden die Ableitungssymbole umgekehrt auf den Stapel gelegt, also CBA. Dies ist nötig, damit als nächstes, d. h. als oberstes Stapelelement, A betrachtet wird. Im Falle einer ε-Produktion wird lediglich x vom Stapel entfernt.

b) M(x, a) = error. Der Eintrag in der Parse-Tabelle verweist auf einen Syntaxfehler. Die Fehlerbehandlung wird aktiviert.

3.$x \notin V \vee x \neq c$ => error. Da x weder ein Nichtterminal ist, noch mit c übereinstimmt, liegt ein Syntaxfehler vor. Die Fehlerbehandlung wird aktiviert.

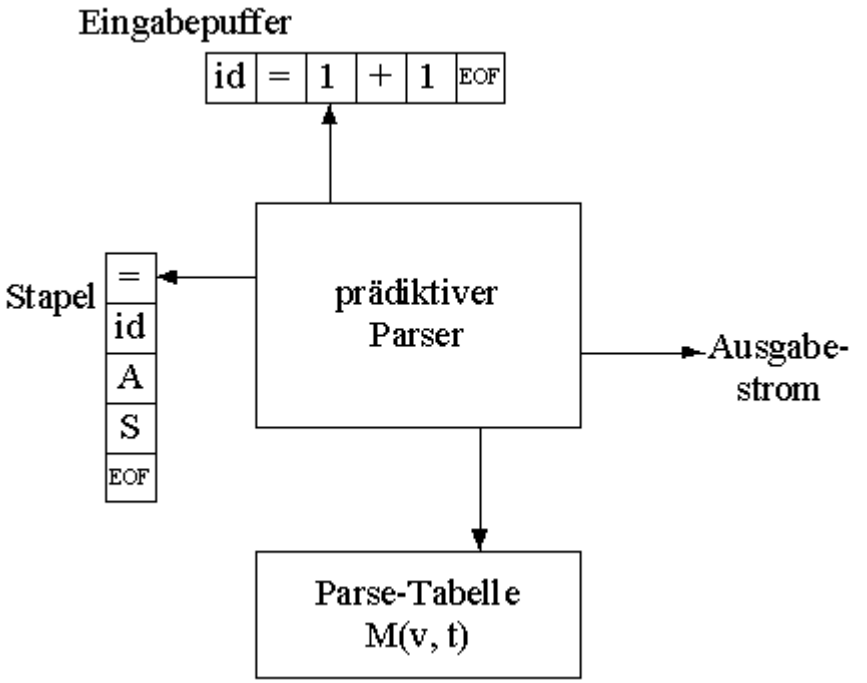


Abb. 4.8: Modell eines tabellengesteuerten prädiktiven Parsers.

Eine prädiktive Syntaxanalyse

Geben sei die Grammatik G:

```
<Ausdruck>  --> <Term> <Ausdruck'>
<Ausdruck'> --> + <Ausdruck>
              | - <Ausdruck>
              | ε
<Term>     --> <Faktor> <Term'>
<Term'>    --> * <Term>
             | / <Term>
             | ε
<Faktor>   --> Zahl
             | + Zahl
             | - Zahl
             | ( <Ausdruck> )
```

Es handelt sich um eine linksfaktorisierte Variante der Grammatik aus Abschnitt 4.4.4. Tabelle 4.1 zeigt die Parse-Tabelle für die Grammatik.

Terminale	***Nichtterminale***				
	Ausdruck	Ausdruck'	Term	Term'	Faktor
Zahl	Term Ausdruck'	**error**	Faktor Term'	**error**	Zahl
+	Term Ausdruck'	+ Ausdruck	Faktor Term'	ε	+ Zahl
-	Term Ausdruck'	- Ausdruck	Faktor Term'	ε	- Zahl
*	**error**	**error**	**error**	* Term	**error**
/	**error**	**error**	**error**	/ Term	**error**
(	Term Ausdruck'	**error**	Faktor Term'	**error**	(Ausdruck)
)	**error**	ε	**error**	ε	**error**
EOF	EOF	ε	**error**	ε	**error**

Tab. 4.1: Eine Parse-Tabelle

Die Einträge der Tabelle entsprechen den jeweils für ein Eingabesymbol anzuwendenen Produktionen. Ist keine Produktion für den entsprechenden Fall vorgesehen, so liegt ein Syntaxfehler vor. In der Tabelle ist dies durch die error-Einträge dargestellt.

Tabelle 4.2 zeigt die Syntaxanalyse für den Ausdruck `(1+1)*2`, der durch die lexikalische Analyse in die Terminalfolge `(Zahl+Zahl)*Zahl` transformiert wurde. Damit die Angabe in der Tabelle so kompakt wie möglich erfolgen konnte, wurden Abkürzungen verwendet.

E steht für EOF und Z für Zahl. Die anderen Terminale wurden als die Zeichen, durch die sie dargestellt werden, übernommen. Die Nichtterminale wurden wie folgt abgekürzt:

- A = Ausdruck
- A' = Ausdruck'
- T = Term
- T' = Term
- F = Faktor

rm in der dritten Spalte ist ein besonderes Symbol. Es deutet an, daß ein Terminal erkannt wurde und der Eingabezeiger um eine Position vorgesetzt wird. rm steht für „remove“, da das Eingabesymbol vom Stapel und von der Eingabe quasi entfernt wird.

In der Spalte „Stapel“ ist jeweils das am weitesten rechts stehende Symbol das oberste Symbol auf dem Stapel. Bei der Spalte Eingabe hingegen ist das am weitesten links stehende Symbol das gerade betrachtete.

Stapel	*Eingabe*	*Aktion*
E A	(Z + Z) * Z E	--> T A'
E A' T	(Z + Z) * Z E	--> F T'
E A' T' F	(Z + Z) * Z E	--> (A)
E A' T') A (	(Z + Z) * Z E	rm
E A' T') A	Z + Z) * Z E	--> T A'
E A' T') A' T	Z + Z) * Z E	--> F T'
E A' T') A' T' F	Z + Z) * Z E	--> Z
E A' T') A' T' Z	Z + Z) * Z E	rm
E A' T') A' T'	+ Z) * Z E	--> ε
E A' T') A'	+ Z) * Z E	--> + A
E A' T') A +	+ Z) * Z E	rm
E A' T') A	Z) * Z E	--> T A'
E A' T') A' T	Z) * Z E	--> F T'
E A' T') A' T' F	Z) * Z E	--> Z
E A' T') A' T' Z	Z) * Z E	rm
E A' T') A' T'	) * Z E	--> ε
E A' T') A'	) * Z E	--> ε
E A' T')	) * Z E	rm
E A' T'	* Z E	--> * T
E A' T *	* Z E	rm
E A' T	Z E	--> F T'
E A' T' F	Z E	--> Z
E A' T' Z	Z E	rm
E A' T'	E	--> ε
E A'	E	--> ε
E	E	EOF
Syntaxanalyse erfolgreich beendet		

Tab. 4.2: Syntaxanalyse für (Zahl+Zahl)*Zahl

Die Zeilen dieser Tabelle sind wie folgt zu lesen: In der ersten Zeile steht in der ersten Spalte der aktuelle Inhalt des Stapels, in der zweiten der der Eingabe. Das oberste Stapelelement ist A und das aktuell betrachtete Eingabesymbol ist (. Diese beiden Informationen sind die Indizes für die Parse-Tabelle. Der jeweilige Eintrag dieser Parse-Tabelle findet sich als Aktion in Spalte drei. Diese Aktion angewendet auf Stapel und ggf. auch Eingabe ergibt die Situationen in der Stapel- und der Eingabe-Spalte der folgenden Zeile mit der resultierenden Aktion in Spalte drei und so fort.

4.4.7 Parse-Tabellen erstellen

Die Arbeitsanweisung zum Aufbau einer Parse-Tablle ist sehr einfach formuliert: „Finden Sie für die Nichtterminale jeweils alle Terminale, mit denen die Produktionen beginnen können, und tragen Sie diese in die Parse-Tabelle ein.“ Die Durchführung dieser Anweisung kann jedoch sehr mühsam sein, ja sogar bei komplizierten und komplexen Grammatik sich zur Sisyphusarbeit entwickeln.

Zum Aufbau von Parse-Tabelle geben Aho, Sethi und Ullman in [2] ein praktikables Verfahren an, das dem Entwickler die Arbeit erleichtert, da es auf systematischen, algorithmischen Überlegungen beruht.

Im folgenden wollen wir diese Methode betrachten.

Die Funktionen FIRST und FOLLOW

Die Funktionen FIRST und FOLLOW sind für die Technik nach Aho, Sethi und Ullman unerläßlich. Deshalb wollen auch wir unseren Exkurs in diese Methode mit der Betrachtung dieser Funktionen beginnen.

α sei eine beliebige Folge von Grammatiksymbolen, d. h. die rechte Seite einer Produktion. FIRST(α) liefert dann die Menge aller Terminale, mit denen ein aus α herleitbarer Eingabestring beginnen kann. Ist aus α ε herleitbar, so findet sich ε in dieser Menge. Dies ist dann von Interesse, wenn sich keine anderen Terminale durch FIRST(α) ergeben (vgl. M(Term', +) aus Tabelle 4.1).

A sei ein Nichtterminal. FOLLOW(A) liefert dann die Menge aller Terminale, die in einer Satzform direkt rechts neben A stehen können. Für jedes a aus FOLLOW(A) und A gilt, daß sie in einem Eingabestring aufeinanderfolgend sind: Aa. Es ist zu beachten, daß aus anderen Herleitungen Symbole zwischen A und a gestanden haben, die beim Analyseprozeß entfernt wurden. Existieren auch herleitbare Strings, bei denen A ganz rechts stehen kann, so gehört auch die Endemarke EOF der Eingabe zu FOLLOW(A).

Für alle Symbole X einer Grammatik G(V, T, P, S), sind für FIRST(X) folgende Regeln solange anzuwenden, bis kein weiteres Terminal oder ε zur FIRST-Menge mehr hinzukommt:

1. Gilt $X \in T$, dann ist FIRST(X) = { X }.
2. Gilt $X \in V \wedge (X \dashrightarrow \varepsilon) \in P$, dann füge ε zu FIRST(X) hinzu.
3. Gilt $X \in V \wedge (X \dashrightarrow Y_1\ Y_2\ Y_3 \ldots Y_n) \in P$, dann nimm a zu FIRST(X) hinzu, falls a für irgendein i in FIRST(Y_i) und ε in allen FIRST(Y_1) bis FIRST(Y_{i-1}) enthalten ist. Falls ε in allen FIRST(Y_1) bis FIRST(Y_j), mit j = 1, 2, 3, ..., n, enthalten ist, nimm ε zu FIRST(X) hinzu.

Für ein Nichtterminal A $\in$ V einer Grammatik G(V, T, P, S) sind FOLLOW(A) folgende Regeln solange anzuwenden, bis sich die FOLLOW-Menge nicht mehr vergrößern läßt:

1. Nimm EOF in FOLLOW(S) auf.
2. Für eine Produktion (A --> αBβ) $\in$ P, mit A $\in$ V $\wedge$ B $\in$ V und α und β als beliebige Folgen von Symbolen aus G, nimm jedes Element von FIRST(β) / { ε } in FOLLOW(B) auf.
3. Wenn gilt (A --> αB $\vee$ A --> αBβ) $\in$ P $\wedge$ ε $\in$ FIRST(β), mit A $\in$ V $\wedge$ B $\in$ V und α und β als beliebige Folgen von Symbolen aus G, dann gehört jedes Element von FOLLOW(A) auch zu FOLLOW(B).

Aus dem oben gesagten gilt resultieren für die Nichtterminale aus der Grammatik aus Abschnitt 4.4.6 folgende Mengen:

```
FIRST(Ausdruck) = FIRST(Term) = FIRST(Faktor) =
  = { Zahl, +, -, ( }
FIRST(Ausdruck') = { +, -, ε }
FIRST(Term') = { *, /, ε }
FOLLOW(Ausdruck) = FOLLOW(Ausdruck') = { ), EOF }
FOLLOW(Term) = FOLLOW(Term') = { +, -, ), EOF }
FOLLOW(Faktor) = { +, -, *, /, ), EOF }
```

Konstruktion von prädiktiven Parse-Tabellen

Nach diesen zugegeben etwas komplizierteren Vorarbeiten, kann nun der eigentliche Algorithmus aus [2] vorgestellt werden.

Die grundlegende Idee des Verfahrens der Autoren des „Drachenbuchs“ ist folgende: Wenn A --> α eine Produktion ist und das Terminal a ist in FIRST(α), dann leitet der Parser A zu α ab, wenn a das aktuelle Eingabesymbol ist. Als Komplikation gegen Aho, Sethi und Ullman den Fall an, daß α = ε ist oder α sich zu ε ableitet. Hier muß A erneut zu α expandiert werden, wenn das Eingabesymbol a in FOLLOW(A) enthalten ist oder wenn a = EOF ist und EOF ein Element von FOLLOW(A) ist.

Der folgende Algorithmus erwartet als Eingabe eine Grammatik G(V, T, P, S) und erzeugt als Ausgabe eine Parse-Tabelle M für diese Grammatik G. Um für eine Grammatik G eine solche prädiktive Parse-Tabelle zu erzeugen, sind folgende Schritte durchzuführen:

1. Für alle Produktionen (A --> α) $\in$ P führe die Schritte 2 und 3 durch.
2. Für jedes Terminal a $\in$ T aus FIRST(α) trage die Produktion A --> α in M(A, a) ein.

3. Gilt ε ∈ FIRST(α), so trage A --> α für jedes Terminal b ∈ FOLLOW(A) an der Stelle M(A, b) ein. Gilt weiter, daß EOF in FOLLOW(A) enthalten ist, dann trage zusätzlich A --> α in M(A, EOF) ein.

4. Für alle undefinierten Einträge in M trage error ein.

Wenn Sie diesen Algorithmus auf die Grammatik aus Abschnitt 4.4.6 anwenden, erhalten Sie die Parse-Tabelle aus Tabelle 4.1.

Der Algorithmus kann prinzipiell über jeder Grammatik operieren und für diese Parse-Tabellen erstellen. Bei linksrekursiven oder mehrdeutigen Grammatiken kann es jedoch dazu führen, daß sich in der Parse-Tabelle mehrere Einträge an der gleichen Stelle ergeben.

Ein solcher Kandidat ist die folgende Grammatik des bekannten Dangling-Else-Problems:

```
<stmt> --> if <cond> then <stmt> <else>
         | <other>
<else> --> else <stmt>
         | ε
```

Die Grammatik ist bekanntlich mehrdeutig, da sie für verschachtelte If-Konstrukte unter Umständen mehrere Ableitungsbäume für den gleichen Eingabestring hat (vgl. Abbildungen 4.1 und 4.2). Da hier nicht immer eine Else-Struktur eindeutig einer If-Bedingung zugeordnet werden kann, hätte die die Parse-Tabelle an der Stelle M(<else>, else) zwei Einträge:

1. `<else> --> else <stmt>`
2. `<else> --> ε`

Grund hierfür ist FOLLOW(<else>), denn es ist { else, EOF }. Dieses Problem läßt sich mit einer Entscheidung zugunsten der ersten Ableitung `<else> --> else <stmt>` lösen. Im übrigen entspricht dies der Strategie ein Else-Konstrukt immer dem letzten If-Block zuzuordnen.

Leider lassen sich nicht alle Proble diese Art so einfach lösen. Penner umgeht dieses Problem in [16] indem er für den obigen Algorithmus nur LL(1)-Grammatiken zuläßt. Allerdings ist dies nur eine Verschiebung des Problems, da sich nicht alle Programmiersprache in LL(1)-Grammatiken beschreiben lassen.

Eine LL(1)-Grammatik ist eine Grammatik, deren Parse-Tabelle keine Mehrfacheinträge enthält. Das erste L steht wieder für ein Lesen des Eingabestroms von links nach rechts. Das zweite L bedeutet, daß Linksableitungen gebildet werden. Der Lookahead ist 1.

Eine LL(1)-Grammatik ist grundsätzlich nicht linksrekursiv und nicht mehrdeutig. Hat ein Nichtterminal A einer LL(1)-Grammatik zwei Produktionen A --> α und A --> β, so muß gelten:

1. Aus String, der aus α und β hergeleitet werden kann, darf nicht mit dem gleichen Nichtterminal beginnen.
2. Ein Leer-String darf nicht zugleich aus α und β hergeleitet werden können.
3. Wenn ε aus β ableitbar ist, dann darf ein aus α herleitbarer String nicht mit einem Terminal aus FOLLOW(A) beginnen.

Sie sehen die obige Grammatik mit dem Dangling-Else-Problem ist keine LL(1)-Grammatik, da sie hauptsächlich gegen Regel 3 und ggf. auch gegen Regel 2 verstößt.

4.5 Bottom-Up-Syntaxanalyse

Die Bottom-Up-Syntaxanalyse arbeitet nach dem Prinzip für eine Folge von Tokens die korrekte Produktion zu ermitteln und durch inverse Rechtsableitungen auf die Nichtterminale abzubilden. Der Parse-Baum wird demnach beginnend bei den Blättern in Richtung Wurzel aufgebaut. Eine Symbolfolge wird also nach und nach auf das Startsymbol, welches bekanntlich die Wurzel des Parse-Baums darstellt, reduziert.

Hinweis: Die Rechtsableitung bei der Bottom-Up-Syntaxanalyse ist deshalb invers, weil sie nicht ein Nichtterminal auf eine Produktion, sondern eine Produktion auf ein Nichtterminal abbildet.

Stimmt eine Symbolfolge mit der rechten Seite einer Produktion überein, so wird diese Folge durch das Nichtterminal der linken Seite ersetzt. Damit „klettern“ die Symbole im Parse-Baum nach oben und werden auf eine weniger große Anzahl von Nichtterminalen reduziert bis das Startsymbol erreicht ist. Dort endet die Syntaxanalyse erfolgreich.

Die Bottom-Up-Syntaxanalyse ist auch unter dem Begriff *Shift-Reduce-Syntaxanalyse* bekannt.

Betrachten wir als Beispiel folgende Grammatik:

```
<Ausdruck> --> <Term> + <Ausdruck>
             | <Term> - <Ausdruck>
             | <Term>
<Term>     --> <Faktor> * <Term>
             | <Faktor> / <Term>
             | <Faktor>
<Faktor>   --> Zahl
             | + Zahl
```

```
                | - Zahl
                | ( <Ausdruck> )
```

Der Eingabestring `(1+1)*2` wird von der lexikalischen Analyse in `(Zahl+Zahl) *Zahl` umgewandelt. Ein Bottom-Up-Parser reduziert diesen Satz durch die folgenden Schritte auf das Startsymbol `<Ausdruck>`:

```
(Zahl+Zahl)*Zahl
(<Faktor>+Zahl)*Zahl
(<Term>+Zahl)*Zahl
(<Term>+<Faktor>)*Zahl
(<Term>+<Term>)*Zahl
(<Term>+<Ausdruck>)*Zahl
(<Ausdruck>)*Zahl
<Faktor>*Zahl
<Faktor>*<Faktor>
<Faktor>*<Term>
<Term>
<Ausdruck>
```

4.5.1 Grundlegender Ansatz

Bevor wir einen grundlegenden Ansatz zur eigentlichen Bottom-Up-Syntaxanalyse betrachten können, sind einige Begriffe zu klären.

Im Zusammenhang mit Rechtsableitungen spielt der Begriff *Handle* eine wichtige Rolle. Ein Handle ist der Substring einer Satzform oder eines konkreten Satzes, der der rechten Seite einer Produktion entspricht und folglich durch das Nichtterminal der linken Seite ausdrückt werden kann. Allerdings muß sich nach dem Ersetzen des Substrings durch das Nichtterminal eine vom Startsymbol (rechts-)ableitbare Symbolfolge, d. h. Satzform, ergeben.

Eine *Satzform* ist eine Folge von Terminalen und Nichtterminalen, die aus dem Startsymbol direkt oder über einen Ableitungspfad mehrerer angewendeter Produktionen herleitbar ist. Der *konkrete Satz* oder auch nur *Satz* ist hingegen ein String, der durch die Grammatik erzeugt werden kann und somit in der Sprache enthalten ist. Eine Eingabezeichenkette, die durch den Parser erkannt werden kann ist damit ein Satz. Die „Zwischenprodukte“ während der Analyse sind demnach Satzformen.

Im Ausdruck `Zahl+Zahl` kann beispielsweise der Substring `+Zahl` in der obigen Grammatik durch die Produktion `<Faktor> --> + Zahl` ausgedrückt werden. Dennoch handelt es sich hier nicht um ein Handle, da das Resultat `Zahl<Faktor>` durch keine gültige Produktion ausgedrückt, ergo auch nicht vom Startsymbol `<Ausdruck>` abgeleitet werden kann.

Der Substring `Zahl` im Ausdruck `Zahl+Zahl` ist hingegen ein Handle der Produktion `<Faktor> --> Zahl`, da sich nach der Substitution `Zahl+<Faktor>` ergibt. Dieses Resultat der Ersetzung ist über einen Ableitungspfad, bestehend aus mehreren angewendeten Produktionen, vom Startsymbol `<Ausdruck>` herleitbar.

Für den Ausdruck `Zahl+Zahl*Zahl` ergibt sich gemäß der Grammatik für arithmetische Ausdrücke von oben folgende Rechtableitung:

```
(1) <Ausdruck> => <Term> + <Ausdruck>
(2)            => <Term> + <Term>
(3)            => <Term> + <Faktor> * <Term>
(4)            => <Term> + <Faktor> * <Faktor>
(5)            => <Term> + <Faktor> * Zahl
(6)            => <Term> + Zahl * Zahl
(7)            => <Faktor> + Zahl * Zahl
(8)            => Zahl + Zahl * Zahl
```

Für die folgende Erläuterungen sind die Handles jeweils fettgedruckt und die Zeilen durchnumeriert. `<Term> + <Ausdruck>` in der ersten Zeile ist beispielsweise ein Handle der rechten Satzform, die aus der Produktion `<Ausdruck> --> <Term> + <Ausdruck>` hergeleitet werden kann. `Zahl` in der fünften Zeile ist zum Beispiel ein Handle der rechten Satzform, die aus der Produktion `<Faktor> --> Zahl` abgeleitet werden kann.

Für die Bottom-Up-Syntaxanalyse ist es nun notwendig eine Rechtsableitung, wie die oben angegebene, zu invertieren, d. h. in umkehrter Richtung herzuleiten. Für das obige Beispiel würde dies bedeuten, daß die Zeilen von 8 nach 1 zu erzeugen sind.

Durch das sogenannte *Handle-Pruning* (vgl. [2]) läßt sich die Rechtsableitung in umgekehrter Richtung erstellen (=inverse Rechtsableitung).

Betrachten wir die folgende noch unbekannte, allgemein formulierte Rechtsableitung:

$S = \gamma_0 => \gamma_1 => \ldots => \gamma_{n-1} => \gamma_n = w$

Aus dem Startsymbol S kann der konkrete Satz w durch sukzessives Anwenden der Produktionen der Grammatik hergeleitet werden.

Durch das Handle-Pruning kann nun ausgehend von w auf S geschlossen werden und damit die Rechtsableitung in umgekehrter Richtung aufgebaut werden.

Der Handle β_n in γ_n, der Satzform, die w repräsentiert wird ermittelt und durch das Nichtterminal der anzuwendenden Produktion $A_n \dashrightarrow \beta_n$ (=*Reduktionsproduktion*) ersetzt. Damit ergibt sich die rechtsableitende Satzform γ_{n-1}. In dieser wird nun der Handle β_{n-1} ermittelt und durch das Nichtterminal der entsprechenden Produktion $A_{n-1} \dashrightarrow \beta_{n-1}$ substituiert, woraus die Satzform γ_{n-2} resultiert. Dieses Verfahren „ermitteln eines Handle, ersetzen des Handle durch das Nichtterminal und erhalten der vorherigen

Satzform der Rechtsableitung“ wird solange fortgesetzt bis das Startsymbol resultiert oder ein Fehler auftritt. Im ersten Fall ist die Syntaxanalse erfolgreich beendet und im zweiten wird die Fehlerbehandlung aktiviert. Tabelle 4.3 zeigt hierzu ein Beispiel für den Ausdruck `Zahl+Zahl*Zahl`. Die Symbole sind hier der einfacheren Darstellung wegen durchnumeriert und analog zu Abschnitt 4.4.6 abgekürzt.

Satzform	***Handle***	***reduzierende Produktion***	***resultierende Satzform***
Z1+Z2*Z3	Z1	<F> --> Z	<F1>+Z2*Z3
<F1>+Z2*Z3	<F1>	<T> --> <F>	<T1>+Z2*Z3
<T1>+Z2*Z3	Z2	<F> --> Z	<T1>+<F2>*Z3
<T1>+<F2>*Z3	Z3	<F> --> Z	<T1>+<F2>*<F3>
<T1>+<F2>*<F3>	<F3>	<T> --> <F>	<T1>+<F2>*<T2>
<T1>+<F2>*<T2>	<F2>*<T2>	<T> --> <F> * <T>	<T1>+<T3>
<T1>+<T3>	<T3>	<A> --> <T>	<T1>+<A1>
<T1>+<A1>	<T1>+<A1>	<A> --> <T> + <A>	<A2>
<A2>			

Tab. 4.3: Reduktionen eines Shift-Reduce-Parsers für Zahl+Zahl*Zahl

Implementierung des Handle-Pruning

Das zentrale Problem beim Handle-Pruning ist nun wie die Handles erkannt werden und einer Produktion zugeordnet werden können. Ein Shift-Reduce-Parser operiert hierzu über einem Stapel und einem Eingabepuffer (vgl. Abbildung 4.9).

Der Eingabepuffer enthält wieder die syntaktisch zu analysierende Zeichenkette, wobei der Puffer auch wieder direkt durch einen Scanner ersetzt werden kann. Auf dem Stapel werden die Grammatiksymbole gespeichert. Der Boden des Stapels ist wieder durch das EOF markiert.

Bei Beginn der Syntaxanalyse ist der Stapel leer, also nur das EOF liegt vor. Im Eingabepuffer liegt der zu analysierende String. Bei erfolgreichem Abschluß der Analyse liegt auf dem Stapel lediglich das Startsymbol und der Eingabepuffer ist leer bzw. der Lesezeiger liegt auf dem EOF, der Endemarke des Puffers.

Wie der Name schon andeutet, kennt der Shift-Reduce-Parser zwei grundlegende Operationen:

1. *Schieben*. Diese Aktion wird ausgeführt, wenn das nächste Eingabesymbol gelesen wird. Der Lesezeiger wird um eins erhöht und das gelesene Symbol auf den Stapel gelegt.

2. *Reduzieren*. Diese Aktion wird ausgeführt, wenn der Parser feststellt, daß das rechte Ende eines Handle als oberstes Element auf dem Stapel liegt. Er sucht das linke Ende des Handle und ermittelt die reduzierende Produktion. Anschließend wird die Symbolfolge des Handle vom Stapel entfernt und das korrespondierende Nichtterminal auf den Stapel gelegt.

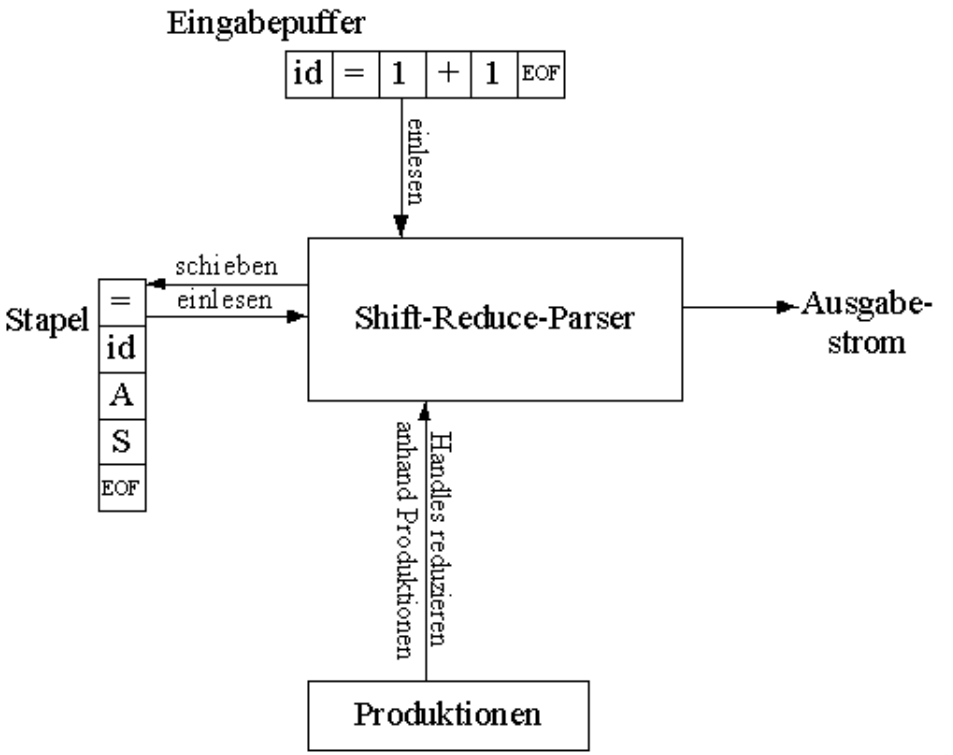


Abb. 4.9: Modell eines Shift-Reduce-Parsers

Der Stapel die ideale Datenstruktur für Shift-Reduce-Parser. Der Grund ist die Tatsache, daß der Handle immer (!) nur oben auf Stapel liegen kann: Sowie das rechte Ende eines Handle auf dem Stapel liegt, so wird dieser Handle umgehend durch das entsprechende Nichtterminal ersetzt. Somit kann kein Handle unter dem aktuell erkannten liegen, denn andere rechte Enden von Handles können nur nach dem aktuell betrachteten folgen. Weiter unten im Stapel liegende Handles wurden ja bereits durch die korrespondierenden Nichtterminale ersetzt!

Tabelle 4.4 zeigt die einzelnen Aktionen und die daraus resultierenden Zustände eines Shift-Reduce-Parsers bei der Analyse des Ausdrucks `Zahl+Zahl*Zahl`. Die Symbole wurden wieder abgekürzt. E enspricht in dieser Tabelle der Endemarke EOF.

Stapel	***Eingabe***	***Aktion***
E	Z+Z*ZE	Z schieben
EZ	+Z*ZE	reduzieren gemäß <F> --> Z
E<F>	+Z*ZE	reduzieren gemäß <T> --> <F>
E<T>	+Z*ZE	+ schieben
E<T>+	Z*ZE	Z schieben
E<T>+Z	*ZE	reduzieren gemäß <F> --> Z
E<T>+<F>	*ZE	* schieben
E<T>+<F>*	ZE	Z schieben
E<T>+<F>*Z	E	reduzieren gemäß <F> --> Z
E<T>+<F>*<F>	E	reduzieren gemäß <T> --> <F>
E<T>+<F>*<T>	E	reduzieren gemäß <T> --> <F> * <T>
E<T>+<T>	E	reduzieren gemäß <A> --> <T>
E<T>+<A>	E	reduzieren gemäß <A> --> <T> + <A>
E<A>	E	
Endzustand erreicht: Syntaxanalyse erfolgreich abgeschlossen		

Tabelle 4.4: Schritte eines Shift-Reduce-Parser bei der Analyse von Zahl+Zahl*Zahl

Die Spalte „Stapel“ zeigt den aktuellen Zustand des Stapels, wobei das am weitesten rechts stehende Symbol das zu oberst liegende ist. Die Spalte „Eingabe“ zeigt den Inhalt des Eingabepuffers. Das am weitesten links stehende Symbol ist hier das aktuell betrachtete. Die letzte Spalte „Aktion“ listet die gemäß des Zustands, der sich aus der Momentaufnahme des Stapels und Eingabepuffers ergibt, die auszuführende Operation auf. Das Resultat dieser Aktion bilden die veränderten Zustände des Stapels und der Eingabe aus der folgenden Zeile.

Auch Shift-Reduce-Parser können nicht alle grammtikalischen Konstrukte analysieren. Mehrdeutigkeiten und Rekursionen bzw. Zyklen mit ε-Produktionen sind solche Kandidaten.

Ein Shift-Reduce-Parser kann bei entsprechenden Grammatiken in Siuationen kommen, bei denen aufgrund des aktuellen Zustands des Stapels und der Eingabe nicht entscheiden kann, welche Aktion auszuführen ist. Prinzipiell unterscheidet man zwischen zwei Konflikten, die auftreten können:

- Shift-Reduce-Konfikt. Der Parser kann nicht entscheiden, ob er schieben oder reduzieren soll.
- Reduce-Reduce-Konflikt. Der Parser kann aufgrund mehrerer passender Prodiktionen nicht entscheiden, welche er zur Reduktion anwenden soll.

Betrachten wir zunächst folgende allgemein formulierte Produktion:

```
<A> --> <A> α
      | ε
```

Hier kommt es zu einem besonderen Shift-Reduce-Konflikt, wenn ein Handle passend zu α vorliegt. Liegt α auf dem Stapel kann keine Reduktion stattfinden, da zur Reduktion <A>α auf dem Stapel liegen müßte. Diese Situation kann jedoch im konkreten Fall niemals eintreten, da hierzu ε auf den Stapel gelegt und zu <A> reduziert werden müßte. Doch wann sollte ein ε auf den Stapel gelegt werden, also eine Markierung für kein Zeichen?

Also anstatt α auf <A> zu reduzieren, werden immer weiter Symbole geschoben (shift statt reduce => besonderer Shift-Reduce-Konflikt).

Theoretisch ist ein Parser denkbar, der jeweils beim Handle-Test alle Nichtterminale mit ε-Produktionen einfach zwischen die Stapelelemente schiebt und überprüft, ob dies einem Handle entspricht. Dieses Vorgehen ist jedoch nicht effektiv.

Ein anderes Problem ergibt sich wieder mit dem Dangling-Else. Auch hier kommt es zu einem Shift-Reduce-Konflikt:

```
<stmt> --> if <cond> then <stmt>
         | if <cond> then <stmt> else <stmt>
         | other
```

Liegt als nächste Eingabe ein Token `else` an, so kann der Parser nicht entscheiden, ob er den Handle auf dem Stapel `if <cond> then <stmt>` auf `<stmt>` reduzieren, oder `else` auf den Stapel schieben soll.

Dieser Konflikt läßt sich in einem Shift-Reduce-Parser sehr leicht lösen, in dem dem Schieben der Vorzug eingeräumt wird. Dadurch wird das Else-Konstrukt wieder dem letzten If-Statement zugeordnet. So wie es allgemein üblich ist.

Folgende Grammatik ist ein Beispiel für einen Reduce-Reduce-Konflikt:

```
<expr> --> <term> + <expr>
         | <term> - <expr>
         | <term>
<term> --> <fact> * <term>
         | <fact> / <term>
         | <fact> <term>
         | <fact>
<fact> --> Zahl
         | + Zahl
         | - Zahl
         | ( <expr> )
```

Die obige Grammatik enthält eine kleines „Schmankerl“ durch die Produktion `<term> --> <fact> <term>` lassen sich auch Ausdrücke wie 8(9+(-2)) auswerten, d. h. als ob dort stünde 8*(9+(-2)). Allerdings birgt dies einen typischen Reduce-Reduce-Konflikt insich.

Angenommen es liegt die Symbolfolge `Zahl-Zahl` vor. Hier kann nicht entschieden werden, ob eine Multiplikation einer positiven und einer negativen Zahl oder eine Differenz zweier positiver Zahlen vorliegt, denn es existieren zwei gültige Rechtsableitungen für diesen Satz:

```
(1) <expr> => <term> - <expr> => <term> - <term> =>
           => <term> - <fact> => <term> - Zahl =>
           => <fact> - Zahl => Zahl - Zahl
(2) <expr> => <term> => <fact> <term> => <fact> <fact> =>
           => <fact> - Zahl => Zahl - Zahl
```

4.5.2 LR(k)-Syntaxanalyse

Die effizienteste Bottom-Up-Syntaxanalysetechnik ist die *LR(k)-Syntaxanalyse*. Die Bezeichnung LR(k) steht für

- L = lesen von links nach rechts
- R = bilden einer Rechtsableitung und
- k = Anzahl der im voraus betrachteten Symbole (=Lookahead).

Häufig findet sich auch die gekürzte Bezeichnung LR-Syntaxanaylse. Hier wird von einem Lookahead k = 1 ausgegangen.

Die Implementierung von LR(k)-Parsern für eine konkrete Sprache „von Hand“ ist nur genialen Seelen mit nicht vom Terminkalender diktiertem Leben zu empfehlen. Der

Normalsterbliche wird für die Konstruktion von LR(k)-Parsern ein entsprechendes Hilfsmittel, sprich *Compilergenerator* bzw. *Parsergenerator*, verwenden. Ein soclehs Werkzeug, yacc genannt, werden wir etwas später betrachten.

Wozu sich dann überhaupt mit LR(k)-Parsing beschäftigen? - Zugegeben eine berechtigte Frage. Das Grundlagenwissen, welches hier gewonnen wird, ermöglicht einen Blick hinter die Kulissen von Parsergeneratoren und kann somit helfen das eine oder andere delikate Problem im praktischen Einsatz zu lösen.

Das LR(k)-Parsing ist die wohl allgemeinste Form der Shift-Reduce-Syntaxanalyse ohne Backtracking, aber dennoch die effizienteste. Diese Analysetechnik zeichnet sich dadurch aus, daß jedes sprachliche Konstrukt, das durch eine kontextfreie Grammatik beschrieben werden kann, von ihr analysiert werden kann. Damit sind die Grammatiken, die vom LR(k)-Parsing analysiert werden können, eine echte Obermenge der Grammtiken, die von der prädiktiven Syntaxanalyse verarbeitet werden können. Außerdem ist die LR(k)-Syntaxanalyse durch das Lesen von links nach rechts in der Lage syntaktische Fehler so früh wie möglich zu erkennen.

Kurzum: Die LR(k)-Syntaxanalyse ist sehr mächtig und effizient. Schon deshalb lohnt sich eine nähere Betrachtung.

Grundlegender Algorithmus

Das Modell eines LR-Parsers sehen Sie in Abbildung 4.10. Er besteht wie zu erwarten war aus einem Eingabepuffer, einem Ausgabestrom, einem Stack, einem Analyseprogramm und das gekapselten Grammatik. Die Grammatik ist durch eine zweigeteilte Syntaxanalysetabelle repräsentiert. Diese Tabelle enthält die gesamten sprachabhängigen Teile. Hieraus läßt sich auch der bevorzugte Einsatz von LR-Parsern in Parsergeneratoren ersehen. Der gesamte Algorithmus und die grundlegenden Datenstrukturen bleiben immer erhalten. Ändert sich die Sprache, muß lediglich die Syntaxanalysetabelle ausgetauscht werden.

Damit erklärt sich auch, warum LR-Parser nicht die erste Wahl für Einzelimplementierungen „von Hand“ sind. Das gesamte Konzept ist auf Wiederverwendung ausgelegt. Durch einfachen Austausch der Syntaxanalysetabelle läßt sich eine andere Sprache analysieren. Die Nutzung des Konzept für lediglich eine einzige Sprache käme einer Verwendung eines Kugelschreibers mit wechselbarer Miene als Einwegschreiber gleich.

Auf dem Stapel werden Zustände s_i verwaltet. Jeder Zustand gibt Auskunft über die bereits analysierten und gelesenen Symbole der Grammatik; kurz den Status, den Zustand der Analyse. Er ist quasi eine Zusammenfassung der bereits erfolgten Analyse.

Durch den akutellen Zustand zu oberst auf dem Stapel und dem Zeichen aus dem Eingabepuffer wird die Teiltabelle Aktion der Syntaxanalysetabelle indiziert. Der so

eindeutig beschriebene Eintrag bestimmt das weitere Vorgehen, d. h. die auszuführende Aktion.

Jede Zelle der Teiltabelle Aktion kann dabei eine der folgenden vier Aktionen deuten:

1. schieben eines Zustands,
2. reduzieren mittels Produktion der Grammatik,
3. akzeptieren der Eingabe, d. h. Syntaxanalyse erfolgreich beendet und
4. aktivieren der Fehlerbehandlung aufgrund Syntaxfehler.

Die Teiltabelle Sprung der Syntaxanalysetabelle wird durch die Zustände und durch die Grammatiksymbole indiziert. Sie gibt den nächsten Zustand nach einer Reduktion an.

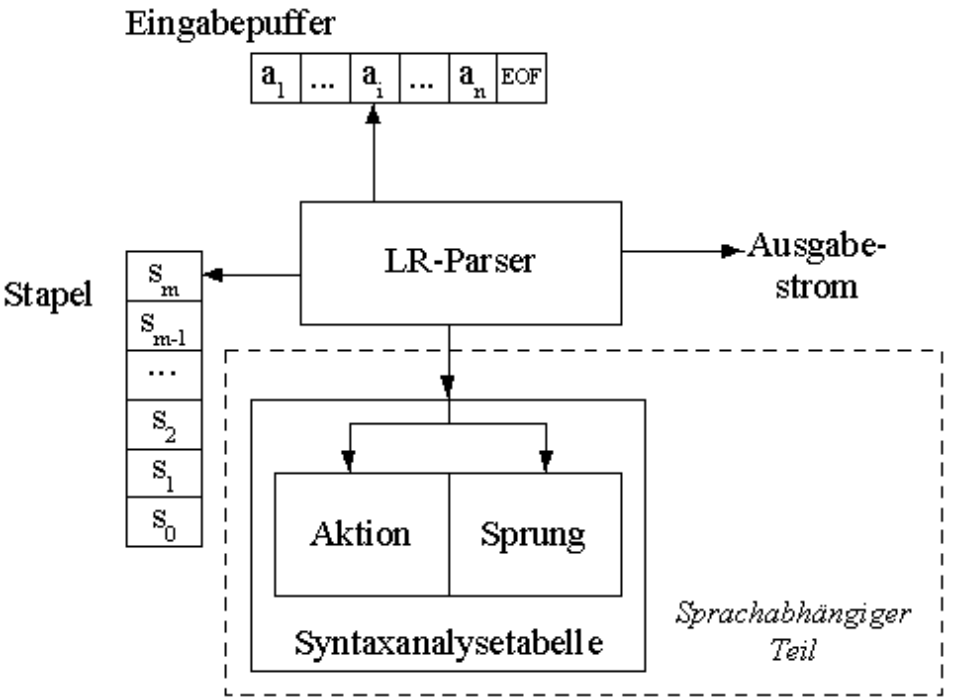


Abb. 4.10: Das Modell eines LR-Parsers

Das Prinzip nach dem LR-Parser nun arbeiten ist folgendes. Nach Betrachten des aktuellen Symbols a_i aus dem Eingabepuffer und des obersten Zustand s_m auf dem Stapel, wird Aktion(a_i, s_m) aus der Syntaxanalysetabelle bestimmt und entsprechend verfahren:

1. Aktion(a_i, s_m) = „schieben“. Folgende Schritte sind auszuführen:

a) Zustand Sprung(s_m, a_i) auf den Stack legen.

b) Lesezeiger der Eingabe um eins inkrementieren, also das nächste Eingabezeichen betrachtet.

2. Aktion(a_i, s_m) = „reduziere“ gemäß einer Produktion der allgemeinen Form A --> β. Hier sind folgende Schritte auszuführen:

a) Ermitteln von r = Anzahl der gemäß der Ableitung β vom Stapel zu entfernenden Zustände.

b) Entfernen der r Zustände vom Stapel, so daß der Zustand s_{m-r} sichtbar wird.

c) Zustand Sprung(s_{m-r}, A) auf den Stapel legen (A ist das Nichtterminal der Produktion).

3. Aktion(a_i, s_m) = „akzeptiere“. Die Syntaxanalyse ist erfolgreich beendet.

4. Aktion(a_i, s_m) = „Synataxfehler“. Fehlerbehandlung aufrufen.

Diese vier Schritte werden so oft wiederholt, bis ein akzeptierender Zustand erreicht wird oder ein Syntaxfehler auftritt.

Betrachten wir hierzu folgende Grammatik:

```
(1) <A> --> <T> AS <E>
(2)       | <T>
(3) <T> --> <F> MD <T>
(4)       | <F>
(5) <F> --> ( <A> )
(6)       | Z
```

Die Terminale AS und MD stehen in dieser Grammatik allgemein für + und - bzw. * und /. Sonst gelten wieder die bekannten Abkürzungen. Zusätzlich sind die Produktionen diesmal durchnumeriert.

Zustand	*Z*	*AS*	*MD*	*(*	*)*	*EOF*
0	s	**err**	**err**	s	**err**	**err**
1	**err**	s	**err**	**err**	**err**	OK
2	**err**	r (2)	s	**err**	r (2)	r (2)
3	**err**	r (4)	r (4)	**err**	r (4)	r (4)
4	s	**err**	**err**	s	**err**	**err**
5	**err**	r (6)	r (6)	**err**	r (6)	r (6)
6	s	**err**	**err**	s	**err**	**err**
7	s	**err**	**err**	s	**err**	**err**
8	**err**	s	**err**	**err**	s	**err**
9	**err**	r (1)	s	**err**	r (1)	r (1)
10	**err**	r (3)	r (3)	**err**	r (3)	r (3)
11	**err**	r (5)	r (5)	**err**	r (5)	r (5)

Tab. 4.5: Aktionstabelle

Zustand	*Z*	*AS*	*MD*	*(*	*)*	*EOF*	*<A>*	*<T>*	*<F>*
0	5			4			1	2	3
1		6							
2			7						
3									
4	5			4			8	2	3
5									
6	5			4				9	3
7	5			4					10
8		6			11				
9			7						
10									
11									

Tab. 4.6: Sprungtabelle

Die Teile der Syntaxanalysetabelle zur Grammatik sehen Sie in den Tabellen 4.5 und 4.6. Die fettgedruckten Einträge **err** in der Aktionstabelle weisen auf Syntaxfehler hin. OK ist ein Eintrag, der den akzeptierenden Zustand repäsentiert. s und r sind die Abkürzungen für „schieben“ und „reduzieren“. Bei Reduktionsaktionen ist zusätzlich die Nummer der reduzierenden Produktion angegeben.

Betrachten wir nun die syntaktische Analyse des konkreten Satz `1+1*2`. Den kompletten Durchlauf sehen Sie in Tabelle 4.7. Der Satz `1+1*2` wird vom Scanner in die Symbolfolge `Z+Z*Z` umgewandelt. Streng genommen sogar in `Z AS Z MD Z`, aber der Übersichtlichkeit wegen geht Tabelle 4.7 von ersterer Symbolfolge aus, auch wenn dies

nicht ganz korrekt ist. E ist in dieser Tabelle wiederum die Endemarke im Eingabepuffer bzw. das spezielle Token EOF des Scanners.

In der Stapel-Spalte ist wieder das am weitesten rechts stehende Symbol das oberste Element und in der Spalte „Eingabe“ ist wieder das am weitesten links stehende Symbol das aktuell betrachtete Zeichen.

Stapel	*Eingabe*	*Aktion*
0	Z+Z*ZE	schiebe (push 5)
0 5	+Z*ZE	reduziere gemäß <F> --> Z (1 x pop, push 3)
0 3	+Z*ZE	reduziere gemäß <T> --> <F> (1 x pop, push 2)
0 2	+Z*ZE	reduziere gemäß <A> --> <T> (1 x pop, push 1)
0 1	+Z*ZE	schiebe (push 6)
0 1 6	Z*ZE	schiebe (push 5)
0 1 6 5	*ZE	reduziere gemäß <F> --> Z (1 x pop, push 3)
0 1 6 3	*ZE	reduziere gemäß <T> --> <F> (1 x pop, push 9)
0 1 6 9	*ZE	schiebe (push 7)
0 1 6 9 7	ZE	schiebe (push 5)
0 1 6 9 7 5	E	reduziere gemäß <F> --> Z (1 x pop, push 5)
0 1 6 9 7 10	E	reduziere gemäß <T> --> <F> MD <T> (3 x pop, push 9)
0 1 6 9	E	reduziere gemäß <A> --> <T> AS <A> (3 x pop, push 1)
0 1	E	OK

Tab. 4.7: LR-Parsing von 1+1*2

Die syntaktische Analyse beginnt mit der Ausgangskonfiguration[1]:

- Auf dem Stapel liegt lediglich der Startzustand 0 und
- Der Lesezeiger ist auf das erste Symbol positioniert.

Aktion(0, Z) ist „schieben“, d. h. der Wert von Sprung(0, Z) muß auf den Stapel gelegt werden. Deshalb findet sich in der ersten Zeile der Tabelle 4.7 auch in Aktion die in klammern gefaßte Anweisung „push 5“, was soviel wie „lege 5 auf den Stapel“ bedeutet. Nach jedem Schieben muß auch der Lesezeiger der Eingabe inkrementiert werden, woraus die neue Kofiguration von Stapel und Eingabe der nächsten Zeile resultiert.

In der nächsten Zeile ergibt sich für Aktion(5, +) bzw. Aktion(5, AS) „reduzieren“ gemäß <F> --> Z. Nun ist eine bestimmte Anzahl an Zuständen vom Stapel zu nehmen, die von der reduzierenden Produktion abhängt. Diese Anzahl ist im Fall des Beispiel-LR-Parsers schlicht die Zahl der Grammatiksymbole der rechten Seite der Produktion.

1 Unter Konfiguration versteht man die Konstellation aus den aktuellen Zuständen des Stapels und des Eingabepuffers.

Somit ergeben sich die Stapeloperationen „1 x pop, push 3“, also „nimmt einen Wert vom Stapel, lege 3 auf den Stapel.

Die folgenden Zeilen der Tabelle sind analog hierzu zu lesen, bis in der letzten Zeile der Endzustand erreicht wird und die Syntaxanalyse damit erfolgreich abgeschlossen wurde.

Damit wollen wir zum einen den Bereich der LR-Parser, aber auch der Bottom-Up-Syntaxanalyse abschließen. Auf die Kontruktion von Syntaxanalysetabellen und die damit verbundenen speziellen Formen der LR-Parser wie SLR, LALR, etc. wollen wir hier verzichten. Einerseits wäre der Nutzen für den „Otto-Normal-Anwender“ sehr gering, da LR-Parser praktisch nur im Bereich der Parsergeneratoren von Bedeutung sind und andererseits dies den Umfang dieses Buches schlicht sprengen würde.

Interessierte seien auf [19], [20] und [7] für die theoretisch-mathematische Betrachtung und auf [2] für die praktische Darstellung verwiesen.

4.6 Fehlerbehandlung

Die Behandlung von Fehlern ist einem Parser ein nicht zu vernachlässigendes Subsystem. Da Parser im allgemeinen von Menschen eingegebe Anweisungen grammatikalisch analysieren und nun einmal „erare humanum est“, irren menschlich ist, muß ein Parser gegen Fehleingaben abgesichert sein. Dieses Problem ist aber inhärent für alle mit dem Menschen interagierende Software.

Die Anforderungen an die Fehlerbehandlung wurden zwar bereits in Abschnitt 4.1 erwähnt, sollen jedoch der Vollständigkeit halber hier noch einmal angeführt werden. Ein Fehlerbehandlungssubsystem sollte folgende Punkte erfüllen:

- Es muß präzise und verständlich den Fehler dokumentieren.
- Es sollte nicht nach dem ersten Fehler abbrechen (Error-Recovery).
- Es sollte sich von Fehler möglichst schnell erholen.
- Es darf die Analyse korrekter Programme nicht unverhältnismäßig verlangsamen.

Die Umsetzung aller dieser Anforderungen ist schwierig. Allerdings ist das Groß der in der Praxis vorkommenden Fehler nicht sonderlich kompliziert, so daß man sich in einer Fehlerbehandlung auf diese Fälle konzentrieren kann. In den wenigen wirklich schwer nachzuvollziehenden Problemen, in denen die Ursachenfindung in angemessenem Zeitbedarf ohne die Analyse von korrekten Programmen erheblich zu verlangsamen kaum möglich ist, bleibt selten eine andere Wahl als ungenauere Fehlermeldungen - vielleicht auch mit falschen Rückschlüssen - zu liefern und sich auf die Erfahrung des Anwenders zu verlassen.

Dem Parser-Konstrukteur kommt jedoch entgegen, daß manche Analysemethoden, so beispielsweise die LL- und LR-Methoden, Syntaxfehler schon zum frühest möglichen Zeitpunkt erkennen. Diese besondere Eigenschaft nennt sich im Fachjargon *viable-prefix-Eigenschaft*. Ein Fehler wird schon dann erkannt, wenn das gelesene Präfix nicht mehr Präfix eines Wortes der Sprache sein kann, also nicht mehr in denn aktuellen Kontext paßt.

Prinzipiell kann zwischen vier Fehlerarten unterschieden werden:

- *Lexikalische Fehler*. Das Lexem ist nicht im Wortschatz der Sprache. Hierzu gehören falsch geschriebene Bezeichner, Schlüsselwörter, unzulässige Zeichen, etc.
- *Syntaktische Fehler*. Für die gelesene Symbolfolge existiert keine gültige Ableitung. Beispiele wären ein Else-Block ohne If-Konstrukt oder nicht abgeschlossene String-konstanten.
- *Semantische Fehler*. Der syntaktisch korrekte Ausdruck enthält Symbole, die in diesem Kontext nicht miteinander verwendet werden dürfen. Semantische Fehler sind z. B. zuviel übergebene Argumente beim Aufruf eines Unterprogramms oder auch die Zuweisung einer String-Konstante an eine Integer-Variable.
- *Logische Fehler*. Dies sind Fehler im logischen Ablauf der Programms. Hier sind unterminierte Rekursionen oder auch niemals erreichte Codeblöcke zu nennen.

Semantische und logische Fehler sind äußerst schwer bis gar nicht zu erkennen. Häufig muß man sich hier auf abgeschwächte Meldungen, die Warnungen, beschränken.

Wohingegen die Zuweisung einer String-Konstanten an eine Integer-Variable eindeutig als Fehler erkannt und gemeldet werden kann, muß man sich beim Zuweisen des Inhalts einer Signed-Integer-Variablen an eine Unsigned-Integer-Variable allenfalls - wenn überhaupt - auf die Ausgabe einer Warnung beschränken. Borlands C++-Compiler gibt hier vorbildlich die Meldung

```
Conversion may lose significant digits
```

aus.

Bei logischen Fehlern kann man sich ebenfalls nur auf Warnungen beschränken. Allerdings läuft man hier sehr schnell Gefahr sich vom hundertsten ins tausende zu verlieren. Außerdem können logische Fehler nur in wenigen Fällen vom Parser erkannt werden, schließlich kann ein Parser nicht die Idee im Kopf des Anwenders erahnen. (Vielleicht ändert sich dies ja einmal mit ausgereifteren „Brain-Interfaces“...)

Die Fehlererkennung während der lexikalischen Analyse ist sehr begrenzt (vgl. vorheriges Kapitel). Dementsprechend liegt das größte Potential für die Fehlererkennung und das Error-Recovery in den syntaktischen Fehlern und somit beim Parser.

Entsprechende repräsentative Statistiken geben außerdem ein sehr interessantes Bild, welcher Art die in der Praxis auftretenden Syntaxfehler sind. Zwar sind Statistiken geduldig und können bei kompliziert zu handhabenden Sprachen eventuell schwanken, dennoch kann man aus diesen Statistiken folgende Schlüsse ziehen (im wesentlichen nach [2]):

- Vier von fünf fehlerhaften Anweisungen enthalten lediglich einen, das Groß der übrigen zwei Fehler.
- Neun von zehn Fehler betreffen lediglich ein einzelnes Symbol.
- Die meisten Fehler betreffen die Zeichensetzung (60%), gefolgt von Operatoren und Operanden (20%) und Schlüsselwörtern (15%).

Das Error-Recovery ist immer eine Gradwanderung. Es muß zwischen effektiven Nutzen für Anwender, der „Erkennungsgüte“ und dem Geschwindigkeitsverlust durch die Fehlerüberprüfung abgewogen werden. Unter *Erkennungsgüte* möchte ich die Qualität, Treffsicherheit und Intelligenz der Fehlererkennung zusammenfassen. Der Idealfall hoher Erkennungsgüte wäre das alle Fehler exakt bestimmt und eindeutig zugeordnet werden können. Da aber Parser nicht in die Köpfe der Anwender sehen können, ist dieser Idealfall schlicht eine theoretische Größe.

Prinzipiell kann man jedoch mit Aussage von Aho, Sethi und Ullman in [2] d'accord gehen: „Abgesehen vielleicht von Fällen, in denen Studienanfänger kleine Programme schreiben, sind die Kosten bei extensiver Fehlerreparatur höher als ihr Gewinn.“

4.6.1 Strategien zum Error-Recovery

Es wurden eine Reihe von allgemein verwendbaren Strategien zum Error-Recovery erarbeitet. Keine davon konnte sich jedoch als „der Standard“ schlechthin durchsetzen, da alle ihre Vor- und Nachteile haben. Im folgenden sollen nun die gebräuchlichsten vorgestellt werden.

Panisches Fortsetzen

Das *panische Fortsetzen*, auch *panic recovery* oder *panic mode* genannt, ist eine sehr einfache Methode des Error-Recovery. Tritt ein Syntaxfehler auf, so werden alle folgenden Symbole solange überlesen, bis ein Symbol aus einer sogenannten *Synchronisationsmenge* gelesen wird.

Die Symbole der Synchronisationsmenge sind Tokens, nach denen im allgemeinen der Parser wieder ein definierter Zustand vorfindet. Überlicherweise werden hierfür Symbole wie das Semikolon, welches Anweisungen abschließt, oder auch Klammerungen wie end oder } herangezogen.

Der Vorteil dieser Methode ist ihre Einfachheit und Schnelligkeit. Außerdem ist es ausgeschlossen, daß ein solches Recovery-System in eine Endlosschleife gerät. Andere Error-Recovery-Techniken sind dagegen nicht immer gefeit.

Der Nachteil ist, daß eine ganze Reihe von Symbolen ignoriert und nicht auf ihre syntaktische Korrektheit überprüft werden.

Gerade für interaktive Systeme ist diese Methode jedoch zu empfehlen.

Das panische Fortsetzen eignet sich für die meisten Parse-Strategien.

Phrasen-Level-Recovery

Die Grundidee beim *Phrasen-Level-Recovery* ist in beim Auftreten des Fehlers die Symbolfolge, welche im Verdacht steht den Syntaxfehler zu enthalten bzw. zu verursachen, innerhalb der aktuellen Satzform zu isolieren. Anschließend wird diese fehlerträchtige Stelle durch ein Nichtterminal durch anwenden einer Reduktionsproduktion ersetzt, so daß eine korrekte Ableitung gebildet werden.

Diese Methode eignet sich besonders für Bottom-Up-Parser. Eine gute Abhandlung zu hierzu findet sich [20]. Einen Blick wert ist auch die Erweiterung *Forward-Move-Recovery* von Sippu und Soisalon-Soininen; ebenfalls in [20] zu finden.

Lokale Korrektur

Die *lokale Korrektur* wird häufig mit der Phrasen-Level-Recovery in einem Zug genannt. Sippu und Soisalon-Soininen trennen in [20] zwischen diesen beiden Recovery-Strategien. Wo die Phrasen-Level-Recovery auf fehlerhafte Symbolfolgen innerhalb einer Satzform ansetzt, beschränkt sich die lokale Korrektur darauf fehlerhafte Symbole durch syntaktisch richtige zu ersetzen, überflüssige zu löschen oder fehlende einzufügen.

Die Wahl der lokalen Korrekturen ist Sache des Parser-Entwerfers. Dieser muß vor allem sehr darauf achten, daß das Recovery-Subsystem nicht in eine Endlosschleife gerät. Dies kann beispielsweise dann passieren, wenn vom dem aktuellen Eingabesymbol immer wieder ein Symbole eingefügt wird.

Die Methode der lokalen Korrektur hat den Nachteil, daß in Situationen, in sich ein Fehler erst später auswirkt als er aufgetreten ist, sich das System in Probleme verstrickt und nicht mehr aussagekräftig arbeiten kann.

Die lokale Korrektur ist sowohl für Top-Down-Parser ([2], dort allerdings konstruktorientiertes Recovery genannt), als auch für Bottom-Up-Parser ([20]) geeignet.

Fehlerproduktionen

Sind die Fehler, die häufig auftreten, bekannt, so können diese Fehlersituationen formalisiert werden. Hierzu werden die Fehlersituationen durch eigens entwickelte Produktionen in die Grammatik aufgenommen. Der auf dieser erweiterten Grammatik aufbauende Parser kann diese Syntaxfehler sehr einfach und in der Regel schnell behandeln.

Tabellengesteuerte Parser wie nichtrekursive prädiktive Parser oder auch LR-Parser profitieren von dieser Formalisierung der Fehlersituationen ohne Geschwindigkeitsverlust, da in den Tabellen die Syntaxfehler schon beschrieben sind. In die Syntaxtabellen sind an der betreffenden Stellen lediglich die Produktionen anzuwenden.

Im allgemeinen ist dies für die Implementierung eine sehr saubere Lösung. Leider lassen sich aber nicht alle Fehler auf diese Art beschreiben.

Globale Korrektur

Die *globale Korrektur* basiert darauf einen fehlerhaften String in einen ähnlichen String umzuformen, der syntaktisch korrekt ist. Es wird angesprebt möglichst wenige Änderunge durchzuführen, um folgende Symbol möglichst unverfälscht noch analysieren zu können. Dies ist ein Schritt in Richtung ideale Erkennungsgüte von oben. Allerdings sind die aktuellen Algorithmen für diese Error-Recovery-Strategie zu zeitaufwendig, als das sie von praktischem Interesse sein könnten.

4.6.2 Fehlermeldung

In jedem Fall eines Fehlers muß das Error-Recovery des Parsers eine entsprechende Meldung ausgeben bzw. ein anderes Subsystem (Fehlerbehandlung) damit beauftragen. Fehlermeldungen können in zwei Klassen eingeteilt werden:

- Deklarative Fehlermeldungen.
- Operationale Fehlermeldungen.

Die deklarative Meldung gibt Auskunft über die Position und die Art des Fehlers im Verständis der Grammatik wieder. Operationale Meldungen hingegen geben Hinweise auf die angewendete Recovery-Aktion des Error-Recovery.

Die operationale Fehlermeldung kann als Abbild oder Momentaufnahme der Datenstrukturen im Parser verstanden werden. Sie geben sozusagen einen Dump, der lediglich mit ausreichender Kenntnis des Parsings und der Recovery-Routine ausgewertet werden kann. Diese Meldungen dürften zwar für die Testphase eines

Parsers dem Programmierer sehr nützlich sein, dem Anwender bringen sie in der Regel keine Vorteile.

Die deklarativen Meldungen sind gemeinhin ausreichend eine Fehlersituation zu beschreiben. Dadurch wird zwar die Gefahr eingegangen, daß keine eindeutige oder korrekte Fehlerdiagnosen gegeben werden, da ggf. andere fehlerhafte Konstrukte überprungen (Panic-Mode) oder nicht im ursprünglich vom Anwender vorgesehenen Sinne erkannt (Phrase-Level-Mode und lokale Korrektur) wurden.

Im folgenden wollen wir daher einige Worte zu deklarativen Fehlermeldungen in Top-Down-Parsern verlieren. Für Bottom-Up-Parser sei auf [20] verwiesen.

Prinzipiell können bei Top-Down-Parsern zwei Fehler auftreten:

- *Erwartungsfehler*. Das gelesene Eingabesymbol stimmt mit der erwarteten Terminal nicht überein.
- *Produktionsfehler*. Ein Nichtterminal kann nicht expandiert werden.

Während Erwartungsfehler innerhalb einer Satzform auftreten, nachdem schon ein Präfix, also die Einleitung der Produktion, erkannt wurden, treten Produktionsfehler auf, wenn kein gültiges Präfix einer Produktion für dieses Nichtterminal erkannt werden kann.

Die Form von Meldungen für Erwartungsfehler sind aus entsprechenden Compilern hinreichend bekannt. Es wird schlicht ausgegeben, welches Symbol erwartet wird, z. B.

```
end erwartet
Bezeichner erwartet
:= erwartet
```

Für Produktionsfehler findet sich in [20] die allgemein gefaßte Meldung, daß dieses sprachliche Kostrukt nicht mit dem aktuellen Symbol beginnen kann. Beispiele wären hier:

```
Die rechte Seite der Zuweisung kann nicht mit if beginnen
Keine Anweisung kann mit else beginnen
```

In der Praxis wird man diese Fehler jedoch noch genauer spezifizieren, in dem man das in der Eingabe anliegende Symbol genauer analysiert:

```
if ist in einer Zuweisung nicht erlaubt
else ohne if
```

Letzterer Fall ist im übrigen ein Kandidat für eine Fehlerproduktion.

4.7 Ein Parser

Nach diesen theoretischen Grundlagen folgt nun der Ausflug in die Anwendung. In diesem Abschnitt wollen wir einen prädiktiven Parser für arithmetische Operationen implementieren. Die Basis wird der rekursive Parser aus dem zweiten Kapitel bilden.

Dieser Parser wird ein Schritt in Richtung Programmiersprache darstellen. Er wird sowohl mit Fließkommakonstanten, als auch mit Variablen operieren können. Die Operationen werden dabei auf der artihmetischen Seite Addition, Subtraktion, Multiplikation, Division und Potenzierung umfassen. Ergänzt werden diese Operationen durch die Zuweisung von Werten an Variablen. Darüberhinaus werden Schlüsselwörter zum Einlesen von vom Benutzer eingegebenen Werten in Variablen und zum Ausgeben von Ergebnissen existieren.

Die direkte Codierung der Operationen im Parser ist in der Praxis nicht üblich und auch nicht sinnvoll. Deshalb implementieren wir eine Stack-Maschine, die von Parser generierten Code interpretieren kann. Diese Stack-Maschine ist die Grundvoraussetzung für die Verarbeitung von Unterprogrammen und Kontrollstrukturen, die in den folgenden Kapiteln hinzukommen werden.

Die Error-Recovery-Strategie wird ein in diesem Fall angemessenes panisches Fortsetzen sein. Die Synchronisationsmenge enthält dabei lediglich ein Zeichen, nämlich das Semikolon. Dieses Zeichen dient zum Abschluß der Anweisungen in unserem kleinen Arithmetikinterpreter.

Als Scanner wird dieser Interpreter die Variante mit Schlüsselworttabelle aus Abschnitt 3.12.4 verwenden. In einem Experimentiersystem ist diese Implementierung sehr komfortabel.

4.7.1 Design-Überblick

Der Interpreter ist in sechs Module zuzüglich Hauptmodul mit Hauptprogramm aufgeteilt. In diesen Modulen sind die neun Klassen des Anwendungsbereich gekapselt. Tabelle 4.8 gibt einen Überblick.

Instanzen von `ErrorHandler`, `Scanner`, `SymbolTable` und `StackMachine` sind Aggregate von `Parser`, stehen also in einer Whole-Part-Beziehung zur Parser-Klasse. Die Parser-Klasse avanciert somit zur Abstraktion des gesamten Interpreters. `ErrorHandler` und `SymbolTable` stehen in Assoziation mit dem `Scanner`, der sie verwendet. `ErrorHandler` und `StackMachine` stehen ebenfalls in Beziehung zueinander, da die Fehlerbehandlung von der Stapelmaschine verwendet wird.

`Symbol` wird in `SymbolTable` referenziert. `Variable` ebenfalls, aber indirekt über Elternklasse `Symbol`. `Variable` und `Symbol` werden in `StackMachine` als Teil des generierten Codes verwendet.

Modul	*Klasse*	*Aufgabe der Klasse*
scanner	Scanner	Scanners
parser	Parser	Parsers
parser	xRecovered	Exception; wird nach Error-Recovery geworfen
machine	StackMachine	Stack-Maschine für Interpretation des Codes
symtable	SymbolTable	Symboltabelle
symtable	Symbol	Abstrakte Basisklasse für Symbole des Symboltabelle
symbols	Variable	Konkrete Klasse für Variablen; stammt von Symbol ab
errhand	ErrorHandler	Fehlerbehandlung
errhand	xFatal	Exception; wird bei fatalen Fehlern geworfen

Tab. 4.8: Die Abstraktion des Interpreters

4.7.2 Die Grammatik

Die Grammatik des Arithmetikinterpreters enthält einige interessante Details und Punkte, die bislang nicht angesprochen wurden.

Der Arithmetikinterpreter arbeitet nach folgender Grammatik:

```
<entry> ::= ε
          | <write> ";"
          | <read> ";"
          | <list> ";"
<write> ::= write <list>
<read>  ::= read id
<list>  ::= <expr> { {"+","-"} <expr> }*
<expr>  ::= <power> { {"*","/"} <power> }*
<power> ::= [{"+","-"}] ( (id "=" <list>) |
                ("(" <list ")" | id | num) ["**" <power>] )
```

Assoziativität der Operatoren

In der obigen Grammatik sind die Operatoren +, -, * und / aus `<list>` und `<expr>` *linksassoziativ*. Die Vorzeichen + und -, sowie der Fortran-Potenzoperator ** aus `<power>` sind *rechtsassoziativ*.

Betrachten wir hierzu den Ausdruck `60/6/2`. Bei linksassoziativer Auswertung, also von links her, ergibt sich:

```
60/6/2 = (60/6)/2 = 10/2 = 5
```

Bei rechtsassoziativer Betrachtung des Divisionsoperator, also von rechts herangehend, resultiert hingegen:

```
60/6/2 = 60/(6/2) = 60/3 = 20
```

Die Zuordnung der Assoziativität an die Operatoren der obigen Grammatik orientiert sich an der allgemein üblichen Regelung in Programmiersprachen, die sich nach den arithmetischen Gesetzmäßigkeiten richtet.

Der Aufbau der Produktionen für `<list>` und `<expr>` ist imgrunde das Resultat einer Linksfaktorisierung. Wenn Sie die EBNF entsprechend auflösen, erhalten sie:

```
<list>  --> <expr> <list'>
<list'> --> ε
          | - <expr> <list'>
          | + <expr> <list'>
<expr>  --> <power> <expr'>
<expr'> --> ε
          | / <power> <expr'>
          | * <power> <expr'>
```

Nach Umkehrung der Linksfaktorisierung erhalten Sie folgende linksrekursive Produktionen:

```
<list> --> <list> - <expr>
         | <list> + <expr>
<expr> --> <expr> / <power>
         | <expr> * <power>
```

Diese Produktionen können von einem prädiktiven Top-Down-Parser nicht verarbeitet werden, zeigen aber deutlich die Linksassoziativität der Operatoren. Die Satzform „60/6/2“ wird durch folgende Linksableitung erkannt:

```
<list> => <list> / <expr> =>
       => <list> / <expr> / <expr> =>
       => <expr> / <expr> / <expr> =>
       => <power> / <expr> / <expr> =>
       => num / <expr> / <expr> =>
       => num / <power> / <expr> =>
       => num / num / <expr> =>
       => num / num / <power> =>
       => num / num / num = 60/6/2
```

Hier wird deutlich, daß zunächst die zwei ersten Operanden verknüpft werden und dann das Ergebnis mit dem dritten Operanden.

Wenn wir hingegen die Satzform „2**2**3“ betrachten, so ergibt diese linksassoziativ ausgewertet den Wert

```
2**2**3 = (2**2)**3 = 4**3 = 64
```

Rechtsassoziativ Berechnet ergibt sich hingegen

```
2**2**3 = 2**(2**3) = 2**8 = 256
```

Letzteres Ergebnis liefert auch die obige Grammatik:

```
<power> => num ** <power> => num ** num ** <power> =>
        => num ** num ** num = 2**2**3
```

Die Produktion für `<power>` ist aufgrund der rechtsassoziativen Bewertungsweise von vornherein nicht linksrekursiv und verursacht somit in einer prädiktiven Top-Down-Implemetierung keine Probleme.

Fazit: Linksassoziativität erfordert in der Grammatik Linksrekursionen; Rechtsassoziativität Rechtsrekursionen.

Priorität der Operatoren

Die Priorität der Operatoren („Punkt vor Strich“) ist in der obigen Grammatik selbstverständlich auch berücksichtigt. Wir wollen hier nicht erneut die grammatikalische Umsetzung betrachten. Hierfür sei auf der zweite Kapitel dieses Buches verwiesen.

Die Operatorpriorität kann jedoch ebenfalls auf ein einfaches Fazit wie das der Assoziativität gebracht werden: Je höher die Priorität des Operators desto tiefer ist er in der Grammatik zu finden. Die Produktionen des Operators bildet dabei keine Zyklen mit den Produktionen der Operatoren nierigerer Priorität. Die einzige Ausnahme bilden geklammerte Ausdrücke, in denen die Operatorenpriorität lokal aufgehoben wird.

4.7.3 Der Scanner

Der Scanner ist mit der Implementierung aus Abschnitt 3.12.4 identisch. Lediglich die Tokens und die Schlüsselworttabelle wurden reduziert.

Die Schlüsselworttabelle präsentiert sich vom Volumen der Einträge her recht spartanisch:

```
static const struct {
       const char *keyword;
       int sym;
} keytable[] = {
       { "read", tREAD },
       { "write", tWRITE },
       { 0, 0 }
};
```

Alle anderen Terminale sind bereits fest in den Automaten des lexikalischen Analysierers implementiert (vgl. vorheriges Kapitel).

Insgesamt existieren lediglich vierzehn Tokens.

4.7.4 Die Stack-Maschine

Die Stack-Maschine ist eine weitere Phase in unserem Interpreter-System. Hier gewinnt auch die Ausgabe des Parsers als Datenstrom an Bedeutung. Statt direkt die einzelnen Operationen auszuführen, generiert der Parser Code für ein einfaches *Computer-Modell*. Dieser Code wird in der auf die grammatikalische Analyse folgenden Interpreter-Phase ausgeführt - sofern beim Parsing kein Fehler auftrat. Den Code können Sie mit dem Byte-Code der Java-Virtual-Machine vergleichen, wenngleich unser Interpreter mit wesentlich plumperen Mitteln arbeitet.

Die *Stack-Maschine* operiert über zwei Datenstrukturen:

- dem Code-Speicher und
- dem Stapel.

Der Code-oder Programmspeicher enthält die Anweisungen, die von der grammatikalischen Analyse erzeugt wurden. Die einzelnen Anweisungen werden in folgender Datenstruktur `Instruction` (definiert in `"machine.hpp"`) gespeichert:

```
typedef struct {
       enum Type {
              itStop  =  0, // Code-Ende
              itVar   =  1, // Variable auf Stapel
              itConst =  2, // Konstante auf Stapel
              itAdd   =  3, // Additionsoperation
              itSub   =  4, // Subtraktionsoperation
              itMul   =  5, // Multiplikationsoperation
              itDiv   =  6, // Divisionsoperation
              itPower =  7, // Potenzoperation
              itNeg   =  8, // Negationsoperation
              itEval  =  9, // Evaluierung
              itAsgn  = 10, // Zuweisung
              itWrite = 11, // Ausgabe
              itRead  = 12  // Eingabe
       } type;       // Instruktionstyp
       Symbol *sym;  // Attribut: Zeiger auf Symbol
       double val;   // Attribut: Wert
} Instruction;
```

Eine Instruktion im Programmspeicher ist demnach ein Tripel aus Typ, Symbol und Wert. Der Typ ist in jeder Instruktion wichtig, denn er identifiziert die auszuführende

Operation. Symbol und Wert sind Attribute, deren Belegung und Bedeutung vom Instruktionstyp abhängt.

Für einen Satz a = b / 2 wird beim Parsing beispielsweise folgender Code generiert:

```
(1) {itVar, a, *}   Variable a auf den Stapel
(2) {itVar, b, *}   Variable b auf den Stapel
(3) {itEval, *, *}  Variable durch ihren Wert ersetzen
(4) {itConst, *, 2} Konstante 2 auf den Stapel
(5) {itDiv, *, *}   Division ausführen
(6) {itAsgn, *, *}  Wert in Variable speichern
(7) {itStop, *, *}  Code-Ende
```

Attribute, die von der jeweiligen Instruktion nicht verwendet werden sind durch einen Stern * gekennzeichnet. Bei `itVar` sind als Symbolattribut die Bezeichner eingetragen. In Wirklichkeit handelt es sich hierbei jedoch um Zeiger auf die entsprechenden Einträge in der Symboltabelle.

Bei der Interpretation des obigen Programms werden die Instruktionen ausgeführt. Es wird hierzu über einem Stapel operiert, auf dem die einzelnen Operanden verwaltet werden. Die Zellen des Stapels haben folgenden Typ aus "`machine.hpp`":

```
typedef struct {
      double val;
      Symbol *sym;
} Operand;
```

Ein Operand kann demnach entweder ein Fließkommawert (Elementvariable `val`) oder ein Verweis auf einen Symboltabelleneintrag (`sym`) sein.

Die Stack-Maschine positioniert zur Interpretation des obigen Programms den *Program-Counter*, kurz pc, auf die erste Anweisung und führt diese aus. Danach findet sich auf dem Stapel das Operanden-Bitupel { *, b }. Der pc wird inkrementiert und die nächste Anweisung ausgeführt. Diese Anweisung itEval bewirkt, daß der Operand vom Stapel gelesen wird und durch seinen Wert ersetzt wird. Ist der Variablen b beispielsweise der Wert 10 zugewiesen, findet sich anschließend dieser Wert auf dem Stapel. Auf dem Stapel liegt also das 2-Tupel { 10, * }. Der pc wird erhöht und die nächste Instruktion betrachtet und so fort. Den gesamten Ablauf sehen Sie in Tabelle 4.9.

Die Instruktion `itAsgn` weist nicht nur der Variablen den Wert zu, sondern legt diesen Wert auch wieder zurück auf den Stapel. Dies ist ermöglicht Anweisungen wie in C++, in denen Zuweisungen schlicht rechtsassoziative Operationen sind.

Folgende C++-Anweisung bewirkt beispielsweise, daß die Variablen a, b und c alle den Wert 5 enthalten:

```
a = b = c = 5;
```

Diese Möglichkeit bietet auch der kleine Arthmetikinterpreter.

In Pascal ist dies beispielsweise nicht möglich, da der Zuweisungsoperator keine Assoziativität besitzt. Eine Anweisung

```
a := b := c := 5;
```

führt in Pascal zu einem Syntaxfehler.

Stapel	***auszuführende Instruktion***	***Aktion***
	{ itVar, a, * }	Variable a auf den Stapel legen.
{ *, a }	{ itVar, b, * }	Variable b auf den Stapel legen.
{ *, b } { *, a }	{ itEval, *, * }	Variable c1 (=b) vom Stapel holen und ihren Wert 10 auf den Stapel legen.
{ 10, * } { *, a }	{ itConst, *, 2 }	Konstante 2 auf den Stapel legen.
{ 2, * } { 10, * } { *, a }	{ itDiv, *, * }	Konstante c2 vom Stapel holen. Konstante c1 vom Stapel holen. c1 = c1 / c2 berechnen. c1 zurück auf den Stapel legen.
{ 5, * } { *, a }	{ itAsgn, *, * }	Konstante c2 vom Stapel holen. Variable c1 vom Stapel holen. c1 (=a) den Wert c2 (=5) zuweisen. Den Wert c2 zurück auf den Stapel legen.
{ 5, * }	{ itStop, *, * }	Ausführung beendet. Stapel leeren.

Tab. 4.9: Interpretation eines Programms mittels Stack-Maschine.

Am Ende einer Interpretation kann es allerdings dann vorkommen, daß noch ein Wert auf dem Stapel liegt. Dieses Problem wird durch ein Entleeren des Stapel nach der Ausführung einer Anweisung behoben.

Im Aufbau des Programms der kleinen Zuweisung sehen Sie, daß die Instruktionen zwischen (2) und (5) auf dem Stapel einen Wert generieren. Dieser Wert ist der zweite, der Wert aus Instruktion (1) der zweite Operand für die Instruktion (6). Anhand dieses kleinen Beispiels sehen Sie schon die Mächtigkeit des Prinzips der Stapel-Maschine. Kleine Einzelinstruktionen lassen eine Fülle von Kombinationen zu, aus denen komplexe Operationen zusammengesetzt werden. Dieses Prinzip ist im Grunde identisch mit der Arbeitsweise der Maschinensprache von Mikroprozessoren. Sie können also die Stack-Maschine als vituellen Prozessor auffassen.

Die Stack-Maschine ist in einer Klasse folgenden Aufbaus implementiert:

```
class StackMachine
{
        public:
                StackMachine(ErrorHandler *eh);
```

```
            // Methode zur Codegenerierung
            void code(Instruction i);
            // Methode zur Interpretation
            double execute(Instruction *prg);

            // Zurücksetzen und löschen des Programms
            inline void reset() {
                  progptr = prog;
                  prog[0].type = Instruction::itStop;
                  stackptr = stack;
            }

            // Liefert die erste Instruktion und damit den
            // Eintrittpunkt für das Programm
            inline Instruction *getProg() { return prog; }

      protected:
            // Methoden der Stapelverwaltung
            void push(Operand c);
            Operand pop();
            inline bool isEmpty() {
                  return (stackptr <= stack);
            }

            // Datenstrukturen der Programmverwaltung
            Instruction prog[PROGSIZE], *progptr;
            // Datenstrukturen der Stapelverwaltung
            Operand stack[STACKSIZE], *stackptr;
            // Zeiger auf das Fehlerbehandlungsobjekt
            ErrorHandler *errHand;
};
```

Code-Generierung

Die Methode `code()` zum Erzeugen einer Instruktion im Programmspeicher ist recht simpel aufgebaut. Der Programmspeicher ist durch das Attribut `prog`, welches ein Array zur Verwaltung von Daten des Typs `Instruction` ist, realisiert. Die Größe wird durch die Konstante `PROGSIZE` bestimmt. `progptr` ist ein Zeiger auf die nächste freie Instruktionszelle im Programmspeicher. Mit Hilfe dieses Zeigers legt `code()` Instruktionen im Programmspeicher `prog` ab:

```
void StackMachine::code(Instruction i)
```

```
{
      if(progptr >= &prog[PROGSIZE])
            errHand->error(0, 0, ErrorHandler::etFatal,
                        "Programm zu gross");
      *progptr++ = i;
}
```

Sollte der Programmspeicher voll sein, so wird die Fehlerbehandlung aktiviert. Die Fehlerart ist hier `ErrorHandler::etFatal`, also ein fataler Fehler. Hierunter versteht man einen Fehler, der das System in eine Situation bringt, in der es die Arbeit nicht fortsetzen kann. Da `code()` vom Parser verwendet wird, bedeutet dies hier, daß im Falle eines fatalen Fehlers die grammatikalische Analyse abgebrochen wird.

Interpretation des Codes

Die Interpretation des Codes übernimmt die Methode `execute()`. Sie besteht aus einer Schleife, die solange alle Instruktionen abarbeitet bis eine Stop-Instruktion (`itStop`) gelesen wird. Als Argument erwartet sie den Zeiger auf die erste Instruktion des abzuarbeitenden Programms. In dieser Interpreter-Implementierung ist dies schlicht immer der Programspeicher `prog`. Bei der Ausführung im Parser-Objekt, das ja als laut Design als Abstaktion des gesamten Interpreter-Systems gesehen werden kann, wird prog schlicht per `getProg()` ermittelt und an `execute()` übergeben.

Warum aber überhaupt dieser Umweg über das Argument? Warum nicht gleich ohne Parameter und direkter Zugriff auf `prog`? - Denken Sie daran, daß Sie den Interpreter auch dahingehend erweitern können, daß er auch Unterprogramme verarbeitet. Diese werden dann zur Ausführung als Argument an `execute()` weitergereicht. Dazu jedoch später mehr.

Innerhalb der Schleife von `execute()` findet sich im wesentlichen ein Switch-Konstrukt, das für die einzelnen Instuktionstypen eine Fallunterscheidung trifft und entsprechende Operationen über dem Stapel ausführt.

Hier der gekürzte Quelltext:

```
double StackMachine::execute(Instruction *prg)
{
 if(errHand->isError())
      return 0.0;
 Operand c1, c2;
 Instruction *pc;
 for(pc = prg; pc->type != Instruction::itStop; pc++)
  switch(pc->type) {
      case Instruction::itVar:
            c1.sym = pc->sym;
```

```
                push(c1);
                break;
        case Instruction::itConst:
                c1.val = pc->val;
                push(c1);
                break;
        case Instruction::itAdd:
                c2 = pop();
                c1 = pop();
                c1.val += c2.val;
                push(c1);
                break;
...
        case Instruction::itNeg:
                c1 = pop();
                c1.val = -c1.val;
                push(c1);
                break;
        case Instruction::itEval:
                c1 = pop();
                c1.val = ((Variable*)c1.sym)->getValue();
                push(c1);
                break;
        case Instruction::itAsgn:
                c2 = pop();
                c1 = pop();
                ((Variable*)c1.sym)->setValue(c2.val);
                push(c2);
                break;
        case Instruction::itWrite:
                c1 = pop();
                cout << c1.val << endl;
                break;
        case Instruction::itRead:
                c1 = pop();
                cout << ">> ";
                cin >> c1.val;
                ((Variable*)c1.sym)->setValue(c1.val);
   }
  return (isEmpty() ? 0.0 : (pop()).val);
}
```

Die Operationen der einzelnen Instruktionen sind recht einfach. Per `pop()` werden Operanden vom Stapel geholt und mit `push()` werden die Ergebnisse wieder zurück auf diesen gelegt. Die einzigen zwei Instruktionen, die keine Werte zurück auf den Stapel legen, sind `itWrite` und `itRead`. Die Ablage etwaiger Statuswerte auf dem Stapel, die hier theoretisch übergeben werden könnten, macht keinen Sinn, da die Grammatik es ohnehin nicht erlaubt dieses Rückgabewerte von `write` und `read` auszuwerten.

`execute()` gibt den eventuell auf dem Stapel liegenden Wert einer Anweisung nach der Stop-Instruktion an den aufrufenden Programmteil zurück. Diesen Rückgabewert wertet unser Interpreter nicht aus. Sie können diesen Wert jedoch für eigene Experimente nutzen.

Die Verwaltung des zur Interpretation notwendigen Stapels ist ebenfalls einfach. Die grundlegende Datenstruktur wird über das Attribut `stack` angesprochen. Es ist ein Array der Größe `STACKSIZE`. Es beinhaltet Elemente vom Typ `Operand`. `stackptr` ist ein Zeiger auf die nächste freie Zelle in `stack`.

Die beiden Methoden `pop()` und `push()` gestalten sich dementsprechend einfach:

```
void StackMachine::push(Operand c)
{
      if(stackptr >= &stack[STACKSIZE])
            errHand->error(0, 0, ErrorHandler::etFatal,
                           "Stapelueberlauf");
      *stackptr++ = c;
}

Operand StackMachine::pop()
{
      if(stackptr <= stack)
            errHand->error(0, 0, ErrorHandler::etFatal,
                           "Stapelunterlauf");
      return *--stackptr;
}
```

Die fatalen Fehler, die in diesen Methoden auftreten können, deuten allerdings eindeutig auf logische Fehler hin. Hat der Programmierer bei der Code-Generierung im Parser keinen Fehler gemacht, können Stapelüberlauf und -unterlauf niemals auftreten!

Hinweis: Falls Sie diesen Quelltext unter DOS zu einer 16-Bit-EXE compilieren, kann es je nach Speichermodell zu Problemen mit den Zeigervergleichen in `code()`, `pop()` und `push()` kommen. Sicherheitshalber sollten Sie die Zeiger als `huge` deklarieren, da hier automatisch bei jeder Zeigerarithmetik die Segmentkomponente normalisiert wird.

4.7.5 Die Symboltabelle

Die Symboltabelle ist als gesonderte Klasse realisiert. Sie verwaltet die einzelnen Symbole als dynamische verkettete Liste. Dies mag uneffektiv klingen, ist es jedoch nicht. Muß überprüft werden, ob ein Bezeichner bereits als Symbol in der Tabelle existiert muß ohnehin der gesamte Datenbestand überprüft werden. Hier ist die verkettete Liste kein Hindernis. Innerhalb des Codes des Programmspeichers werden die Symbole dann sowieso direkt durch Ihre Symbolobjekte angesprochen, die über ein Handle der Symboltabelle direkt angesprochen werden. Hier tritt die Datenstruktur der verketteten Liste nicht in Aktion.

Der Vorteile der dynamischen verketteten Liste liegt auf der Hand: Keine theoretische Größenbeschränkung. Lediglich der real verfügbare Arbeitsspeicher setzt hier Grenzen.

Über die Methode `update()`, die bereits im vorherigen Kapitel im Scanner zum Einsatz kam, wird die Symboltabelle aktualisiert. Nicht registrierte Symbole werden eingetragen und zu bereits existierenden das Handle ermittelt.

Mittels `getSymbol()` und `setSymbol()` läßt sich das zu einem Handle gehörende Symbolobjekt ermitteln respektive setzen.

Damit ergibt sich folgende Klassendeklaration:

```
class SymbolTable
{
       public:
              SymbolTable();
              ~SymbolTable();

              SYMHANDLE update(const char *symbol);

              Symbol *getSymbol(SYMHANDLE sh);
              void setSymbol(SYMHANDLE sh, Symbol *s);

              void clear();

       protected:
              struct SymbolEntry {
                     char *id;
                     Symbol *sym;
                     SymbolEntry *next;
              } *table;
};
```

SYMHANDLE ist schlicht als untypisierter Zeiger definiert:

```
typedef void *SYMHANDLE;
```

Hinter diesen Handles verbirgt sich im Grunde nichts anderes als Elemente der verketteten Liste table, also Zeiger vom Typ SymbolTable::SymbolEntry. Im Zuge des Information-Hiding wird diese Information allerdings verschleiert.

Die Methode clear() dient zum Löschen aller Symbole in der Tabelle. Dies ist beispielsweise nötig, wenn der Parser einen neuen Quelltext analysieren soll. Die Symbole des vorherigen Programms machen dann recht wenig Sinn.

Die Klasse Symbol ist eine abstrakte Basisklasse für die einzelnen Symboltypen. Diese Klasse macht es möglich unterschiedliche Symbole in diese Tabelle zu verwalten. Sie werden schlicht alle als Instanzen von Symbol eingetragen.

```
class Symbol
{
        public:
                enum {
                        type_var = 0
                };

                virtual int getType() = 0;
};
```

Die virtuelle Methode getType() von Symbol gibt den jeweiligen Typ des Symbol zurück. In dieser Ausbaustufe des Arithmetikinterpreters ist lediglich der Typ Variable (type_var) definiert. Diese virtuelle Methode getType() wird erst in den konkreten Kindklassen von Symbol implementiert.

Die Methode update() der Symboltabellenklasse ist recht unspektakulär. Zunächst wird die Symboltabelle durchforstet, ob unter dem als Argument vorliegenden Bezeichner bereits ein Symbol definiert wurde. Ist dies nicht der Fall wird eines angelegt. Zurückgegeben wird der Zeiger des Tabellenelements, allerdings „verschleiert“, d. h. konvertiert in einen Wert vom Typ SYMHANDLE.

```
SYMHANDLE SymbolTable::update(const char *symbol)
{
        SymbolEntry *ptr = table;
        while(ptr) {
                if(!strcmp(ptr->id, symbol))
                        break;
                ptr = ptr->next;
        }
        if(!ptr) {
                ptr = new SymbolEntry;
```

```
                ptr->sym = 0;
                ptr->id = new char[strlen(symbol)+1];
                strcpy(ptr->id, symbol);
                ptr->next = table;
                table = ptr;
        }
        return ptr;
}
```

Beim Neuanlegen des Bezeichners, das ja im Scanner geschieht, wird dem Bezeichner noch kein Symbolobjekt zugeordnet. In der aktuellen Ausbaustufe des Interpreters existieren zwar nur Variablen, aber im Hinblick auf die Einführung anderer Symboltypen, z. B. Unterprogramme, sollte man bei der Implementierung davon ausgehen, daß eine Reihe anderer Symbole existieren. Der Scanner kann allerdings aus den bekannten Gründen nicht feststellen, welchen Typs ein Bezeichner ist (vgl. Kapitel 3).

Das Zuordnen einer Symbol-Instanz erfolgt per `setSymbol()`:

```
void SymbolTable::setSymbol(SYMHANDLE sh, Symbol *s)
{
        SymbolEntry *ptr = (SymbolEntry*)sh;
        if(ptr->sym)
                delete ptr->sym;
        ptr->sym = s;
}
```

Wie hier klar zu sehen ist, werden die Symbole nur über das Handle vom Typ `SYMHANDLE` von außen angesprochen. Dies ist im Sinne der Datenabstraktion.

Die Abfrage der Symbol-Instanz geschieht mittels `getSymbol()`:

```
Symbol *SymbolTable::getSymbol(SYMHANDLE sh)
{
        return ((SymbolEntry*)sh)->sym;
}
```

Diese Methode ist so einfach, daß Sie sogar inline definiert werden könnte. Allerdings würde dies dem Anwender verraten, welche Information hinter `SYMHANDLE` steht und würde somit einen Konflikt in der Philosophie des Information-Hiding verursachen.

Last but not least noch die Methode clear(), die ebenfalls sehr „einfach gestrickt“ ist:

```
void SymbolTable::clear()
{
        SymbolEntry *ptr;
        while(table) {
                ptr = table;
```

```
                table = table->next;
                if(ptr->id)
                        delete[] ptr->id;
                if(ptr->sym)
                        delete ptr->sym;
                delete ptr;
        }
}
```

4.7.6 Das Variablensymbol

Die konkrete Klasse `Variable` ist ein Kind von `Symbol`. Sie dient der Administration der Informationen zu einer Variablen. Zusätzlich enthält `Variable` Methoden zum Verwalten des Variablenwerts. Der Wert wird im geschützten Attribut `value` einer Variable vom Type `double` gespeichert.

Insgesamt enthält Variable einen Konstruktor und drei Methoden, die selbsterklärend sind:

```
Variable::Variable()
{
        value = 0.0;
}

double Variable::getValue()
{
        return value;
}

void Variable::setValue(double val)
{
        value = val;
}

int Variable::getType()
{
        return type_var;
}
```

Die Idee zu `getType()` ist nun folgende: Wird einer Instanz von `Symbol` aus der Symboltablle die Nachricht zum Aufruf dieser Methode gesendet, wird nicht die Methode von `Symbol` gestartet - das im übrigen nicht geht, da die Implementierung auf die Kindklassen verschoben wurde, sondern die Methode der jeweiligen Kindklasse (=virtuelle Methode).

4.7.7 Die Fehlerbehandlung

Die Fehlerbehandlung ist in der Klasse `ErrorHandler` gekapselt. Die Implementierung dieser Klasse ist ebenfalls recht einfacher Natur:

```
class ErrorHandler
{
 public:
      enum ErrorType {
            etWarning,  // Warnung
            etError,    // Fehler
            etFatal     // Fataler Fehler
      };
      void error(unsigned long line, unsigned long
                  column, ErrorType et, const char *msg);
      inline void reset() { error_appeared = false; }
      inline bool isError() { return error_appeared; }
 protected:
      bool error_appeared;
};
```

Die Fehlerbehandlung wird durch die Methode `error()` aktiviert. Sie gibt die entsprechenden Meldungen aus. Im Falle eines fatalen Fehlers wird zusätzlich ein Objekt der Klasse `xFatal` geworfen. Diese Klasse ist nur rudimentär, aber ausreichend definiert:

```
class xFatal {};
```

Wie oben zu sehen ist unterscheidet die Fehlerbehandlung zwischen drei Fehlerarten:

- *Warnung*: Möglicher Fehler.
- *Fehler*: Tatsächlicher Fehler, Error-Recovery setzt ein.
- *Fataler Fehler*: Der Fehler kann durch das Error-Recovery nicht behoben werden.

Die Fehlerbehandlung enthält das Error-Recovery nicht. Dies hat einen einfachen Grund: Das Error-Recovery muß auf die grammatikalischen Strukturen zurückgreifen können. Dies kann jedoch nur der Parser bzw. in gewissen Schranken der Scanner.

Ein externes Fehlerbehandlungsobjekt kann von außen nicht effektiv auf die nötigen grammatikalischen Informationen zurückgreifen. Eine dennoch erfolgende Implementierung des Error-Recovery in die Fehlerbehandlungsinstanz würde zwei Probleme aufwerfen:

1. Durch die ständige Korrespondenz mit dem Parser per Methoden würde die Geschwindigkeit des Systems leiden.

2. Durch die feste Verstrickung des Parsers mit der Fehlerbehandlung würde die objektorientierte Philosophie verletzt. Laut objektorientiertem Paradigma sind die einzelnen Entitäten des Anwendungsbereichs zu trennen und die Schnittstellen auf das nötigste zu beschränken.

Somit begrenzen wir die Fehlerbehandlung auf die Ausgabe von Meldungen. Allerdings wird beim ersten Auftreten eines Fehlers - nicht bei Warnungen - der Fehler registiert. `error_appeared` wird auf true gesetzt. Dieses Flag kann durch die Methode `isError()` abgefragt werden. Dies wird beispielsweise vom Parser genutzt, um die Ausführung des fehlerhaften Quelltextes zu unterdrücken. Doch hierzu später mehr.

Die Methode `error()` erklärt sich nach dem bereits gesagten fast von selbst:

```
void ErrorHandler::error(unsigned long line,
      unsigned long column, ErrorType et, const char *msg)
{
      switch(et) {
            case etWarning:
                  cerr << "Warnung";
                  break;
            case etError:
                  cerr << "Fehler";
                  error_appeared = true;
                  break;
            case etFatal:
                  cerr << "Fatal";
                  error_appeared = true;
      }
      if(line != 0)
            cerr << " (Zeile " << line << ", Spalte "
                 << column << "): " << msg << endl;
      else
            cerr << ": " << msg << endl;
      if(et == etFatal)
            throw xFatal();
}
```

Falls die Zeilennummer nicht Null ist, wird die Position des Fehlers im Quelltext ausgegeben. Dies ist bei lexikalischen und syntaktischen Fehlern der Fall.

Bei Fehlern, die erst bei der Ausführung festgestellt werden, liegen die Positionsdaten nicht mehr vor bzw. der Fehler könnte auch keiner konkreten Position im Quelltext zugeordnet werden.

4.7.8 Der Parser

Nach diesen Basisinformationen nun zum eigentlichen Parser. Der Parser ist als rekursiver prädiktiver Top-Down-Parser mit Panic-Mode-Error-Recovery realisiert.

Das Parser ist eine erweiterte Variante des syntaktischen Analysierers aus Kapitel 2:

```
class Parser
{
       public:
              Parser(istream *file);
              void Run();

              inline void setInputFile(istream *file) {
                     File = file;
              }

       protected:
              void error(const char *errmsg, ...);
              void tokenClear();
              const char *getTokenName();

              void entry(bool& term);
              void list();
              void expr();
              void power();
              void write();
              void read();

              unsigned brackets;
              TOKEN token;

              SymbolTable symTable;
              ErrorHandler errHand;
              Scanner scanner;
              StackMachine machine;
              istream *File;
};
```

Sie sehen, daß der Konstruktor des Parsers als Argument eine Datei in Form eines istream-Objekts erwartet. Es handelt sich hierbei um die Quelldatei, die per `setInputFile()` geändert werden kann. `istream` wurde deshalb gewählt, da hierdurch sowohl die komfortablen Instanzen von `ifstream`, als auch `istream_withassign` - die Klasse des Objekts `cin` - genutzt werden kann. Dadurch

ist es möglich sowohl den Standardeingabekanal als auch jedes andere Dateiobjekt als Eingabe für den Parser zu nutzen.

Als Aggregate im Zuge einer Whole-Part-Beziehung finden sich

- die Symboltabelle als `symTable`,
- die Fehlerbehandlung als `errHand`,
- der Scanner als `scanner` und
- die Stack-Maschine als `machine`.

Umsetzung der Grammatik

Die Grammatik ist in den Methoden `entry()`, `list()`, `expr()`, `power()`, `write()` und `read()` umgesetzt. Gemäß dem Recursive-Descent ist jedes Nichtterminal als Methode umgesetzt.

Die Implementierung ist kein Hexenwerk:

```
inline void Parser::entry(bool& term)
{
  machine.reset();
  token = scanner.getToken();
  if(token.sym == tEOF) {
       term = false;
       return;
  }
  term = true;
  if(token.sym == tWRITE) {
       token = scanner.getToken();
       write();
  } else if(token.sym == tREAD) {
       token = scanner.getToken();
       read();
  } else
       list();
  if(token.sym != tSEMICOLON)
       error("Semikolon erwartet.");
  Instruction inst;
  inst.type = Instruction::itStop;
  machine.code(inst);
}
```

```
void Parser::list()
{
  Instruction inst;
  expr();

  while(token.sym == tPLUS || token.sym == tMINUS)
       if(token.sym == tPLUS) {
              token = scanner.getToken();
              expr();
              inst.type = Instruction::itAdd;
              machine.code(inst);
       } else if(token.sym == tMINUS) {
              token = scanner.getToken();
              expr();
              inst.type = Instruction::itSub;
              machine.code(inst);
       } else if(token.sym != tEOF &&
                  token.sym != tSEMICOLON)
              if(token.sym == tBRACKET2) {
                     if(!brackets)
                            error("Unerwartete ).");
              } else
                     error("Unerwartetes Token: %s.",
                           getTokenName());
}

void Parser::expr()
{
  Instruction inst;

  power();

  while(token.sym == tTIMES || token.sym == tDIV)
       if(token.sym == tTIMES) {
              token = scanner.getToken();
              power();
              inst.type = Instruction::itMul;
              machine.code(inst);
       } else if(token.sym == tDIV) {
              token = scanner.getToken();
              power();
```

```
                inst.type = Instruction::itDiv;
                machine.code(inst);
        }
}

void Parser::power()
{
  Instruction inst;
  bool neg = false;

  if(token.sym == tMINUS) {
        neg = true;
        token = scanner.getToken();
  } else if(token.sym == tPLUS)
        token = scanner.getToken();

  if(token.sym == tBRACKET1) {
        token = scanner.getToken();
        brackets++;
        list();
        if(token.sym != tBRACKET2)
                error(") erwartet.");
        brackets--;
        token = scanner.getToken();
  } else if(token.sym == tID) {
        inst.type = Instruction::itVar;
        Symbol *sym =
                 symTable.getSymbol((SYMHANDLE)token.attrib);
        if(!sym) {
                sym = new Variable;
                symTable.setSymbol((SYMHANDLE)token.attrib,
                                        sym);
        }
        inst.sym = sym;
        machine.code(inst);
        token = scanner.getToken();
        if(token.sym == tASSIGN) {
                token = scanner.getToken();
                list();
                inst.type = Instruction::itAsgn;
                machine.code(inst);
                if(neg) {
```

```
                    inst.type = Instruction::itNeg;
                    machine.code(inst);
              }
              return;
        } else {
              inst.type = Instruction::itEval;
              machine.code(inst);
        }
  } else if(token.sym == tNUM) {
        inst.type = Instruction::itConst;
        inst.val = *((double*)token.attrib);
        machine.code(inst);
        delete token.attrib;
        token = scanner.getToken();
  } else
        error("Fliesskommakonstante oder Bezeichner "
              " erwartet.");

  if(token.sym == tPOWER) {
        token = scanner.getToken();
        power();
        inst.type = Instruction::itPower;
        machine.code(inst);
  }

  if(neg) {
        inst.type = Instruction::itNeg;
        machine.code(inst);
  }
}

void Parser::write()
{
  if(token.sym != tEOF)
        list();
  Instruction inst;
  inst.type = Instruction::itWrite;
  machine.code(inst);
}

void Parser::read()
{
```

```
    if(token.sym != tID)
        error("Bezeichner erwartet.");

    Instruction inst;
    inst.type = Instruction::itVar;
    Symbol *sym =
            symTable.getSymbol((SYMHANDLE)token.attrib);
    if(!sym) {
        sym = new Variable;
        symTable.setSymbol((SYMHANDLE)token.attrib, sym);
    }
    inst.sym = sym;
    machine.code(inst);

    token = scanner.getToken();

    inst.type = Instruction::itRead;
    machine.code(inst);
}
```

Die markantesten Unterschiede zur Implementation aus Kapitel 2 sind - neben dem erweiterten Sprachumfang - zum einen das Scanner als Eingabequelle und zum anderen die Code-Generierung anstatt direkter Berechnung der Terme.

Die algorithmische Struktur dieser Methoden ergibt sich direkt aus der Umsetzung der Grammatik aus Abschnitt 4.7.2. Die Regeln für die Transformation der Grammatik in Programmcode wurde bereits in Abschnitt 2.7.2 dargestellt. Im folgenden wollen wir daher nur einige interessante Stelle in diesem Quelltext herausgreifen.

Die Behandlung eines Bezeichners in `power()` und auch in `read()` ist eine solche Stelle. Wurde das Token `tID` erkannt, so wird zunächst die Instruktion für „Variable auf den Stapel legen“ (`Instruction::itVar`) generiert und der Stack-Maschine übergeben.

Das Symbol-Objekt, das als Attribut für die Instruktion benötigt wird, muß jedoch erst ermittelt bzw. eventuell sogar zuerst angelegt werden. Hierzu wird die Symboltabelle mit dem Attribut des Token, dem Handle des Symbols, frequentiert. Ergibt sich jedoch aus dem Aufruf von `getSymbol()` ein Nullwert, so muß das Variablen-Objekt erst angelegt werden. Hier verfährt der Interpreter wie ein BASIC-System. Die Variable wird ungeachtet ihres Kontextes im Quelltext erzeugt und per Konstruktor mit dem Wert 0.0 initialisiert.

Nachdem `inst.type` und `inst.sym` mit den entsprechenden Werten belegt wurden, wird der Stack-Maschine die Instruktion übergeben.

Im Falle der Methode `power()` kommt nun eine weitere interessante Stelle. Liegt eine Zuweisung vor, also ist das Token `tASSIGN` gelesen worden, so liegt eine Produktion von `power()` vor, wo kein „powered by“ (engl. für „hoch“ eine Zahl) vorkommen kann.

Hier wird schlicht zu `list()` verzweigt und anschließend ein etwaiges negatives Vorzeichen behandelt. Danach wird die Methode `power()` schlicht verlassen. Hier wäre es auch denkbar bei korrrekter Syntax dennoch weiterzuverfahren, da kein Potenzoperator mehr vorkommen kann, da dieser dann schon beim Call von `list()` ausgewertet worde wäre. Die Auswertungen für den Potenzoperator kann man sich an dieser Stelle jedoch aus optimierungstechnischen Gründen sparen.

Eine andere nicht unwichtige Stelle findet sich ebenfalls in `power()` bei der Auswertung des Token `tNUM`. Zunächst wird die Instruktion für „Konstante auf den Stapel legen“ (`Instruction::itConst`) erzeugt. Beim Übergeben des Wertes ist eine Cast-Anweisung nötig, die aus dem untypisierten Zeiger des Token-Attributs zunächst einen Zeiger auf `double` macht und diesen dann dereferenziert, damit die Zuweisung vonstattengehen kann.

Nachdem die Übergabe der Instruktion an die Stapel-Maschine erfolgt ist, kommt eine trivial klingende Stelle: Der dynamisch allokiierte Speicher des Attributs des Token muß freigegeben werden. Dieser Punkt ist für sich betrachtet ist nicht weiter der Erwähnung wert. Im Zuge des Error-Recovery gewinnt dieser Aspekt jedoch besondere Bedeutung. Tokens müssen beim panischen Fortsetzen solange gelesen werden bis ein synchronisierendes Token gelesen wird. Hierbei müssen etwaig dynamisch allokiierte Speicherbereiche im Token freigegeben werden!

Error-Recovery

Das Error-Recovery ist in der Methode `error()` gekapselt. Wie zuvor schon erwähnt arbeitet es nach dem Prinzip des panischen Fortsetzens. Wird also ein fehlerhaftes Token gelesen, so werden alle Tokens solange überlesen bis ein Token der Synchronisationsmenge angetroffen wird.

Die Synchronisationsmenge besteht hier lediglich aus den Tokens `tSEMICOLON`, also dem Strichpunkt und `tEOF` für „Dateiende erreicht“.

```
void Parser::error(const char *errmsg, ...)
{
  char err_msg[100];
  va_list ap;

  va_start(ap, errmsg);
  vsprintf(err_msg, errmsg, ap);
  va_end(ap);
```

```
    errHand.error(scanner.getLine(), scanner.getColumn(),
                  ErrorHandler::etError, err_msg);

    while(token.sym != tEOF && token.sym != tSEMICOLON) {
         token = scanner.getToken();
         tokenClear();
    }
    throw xRecovered();
}
```

Zunächst wird nach der vsprintf-Methode eine Fehlermeldung generiert und mit der Position im Quelltext an den Error-Handler `errHand` übergeben. Anschließend folgt das Error-Recovery in Form einer schlichten Schleife.

Es werden alle Tokens bis zum Synchronistationssymbol ausgelesen und ignoriert. Die Methode `tokenClear()` hat nun die bereits oben angesprochene Aufgabe eventuell reservierten Speicher im Token freizugeben. Sie ist recht simpel implementiert, da bisland nur das Token `tNUM` Speicher belegen kann:

```
void Parser::tokenClear()
{
      if(token.sym == tNUM)
            delete token.attrib;
}
```

Nachdem das panische Fortsetzen ein Ende gefunden hat, muß nun zu einem fest definierten Zustand im Parser zurückgekehrt werden. Hierzu wird eine Instanz von `xRecovered` als Exception geworfen. Dieses Exception-Objekt wird an anderer Stelle gefangen, wo dieser definierte Zustand wieder hergestellt werden kann. Um es schon vorweg zu nehmen, dieser Punkt wird in der Methode `Run()` liegen.

Die Klasse `xRecovered` ist wieder nur rudimentär definiert:

```
class xRecovered {};
```

Die Methode `getTokenName()` der Klasse Parser gibt zum aktuellen einen aussagekräfigen Namen als String zurück. Diese Methode wird in `list()` benötigt. Da diese Methode nur aus einem Switch-Konstrukt besteht und nichts anderes als String-Konstanten zurückliefert, soll auf den Abdruck hier verzichtet werden.

Interpreterbetrieb

Der eigentliche Interpreter ist in der Methode `Run()` gekapselt. Neben Initialisierungscode besteht er hauptsächlich aus einer Schleife. Diese Schleife arbeitet solange bis das EOF-Token gelesen wurde. Bei jedem Schleifendurchlauf wird für eine Anweisung Code durch den Aufruf von `entry()` generiert und - sofern nicht das Dateiende erreicht wurde - dieser Code durch die Stack-Maschine ausgeführt. Danach folgt die nächste Anweisung.

```
void Parser::Run()
{
  symTable.clear();
  errHand.reset();
  scanner.setInputFile(File);

  token.sym = tSEMICOLON;
  while(token.sym != tEOF) {
       brackets = 0;
       try {
             bool calculated;
             entry(calculated);
             if(calculated)
                   machine.execute(machine.getProg());
       }
       catch(xRecovered) {}
       catch(xFatal) {
             break;
       }
  }
}
```

Die Exceptions vom Typ `xRecovered` und `xFatal` werden gefangen, falls sie Auftreten. Bei `xRecovered` genügt es durch `catch` einen Punkt anzugeben, von wo aus die Schleife erneut starten kann, denn schließlich sollen eventuell folgende (Syntax-)Fehler ebenfalls entdeckt werden können. Im Falle von `xFatal` liegt bekanntlich eine Situation vor, in der kein weiteres Fortsetzen sinnvoll und möglich ist. Deshalb wird hier die Schleife schlicht durch `break` beendet.

4.7.9 Das Hauptprogramm

Das Hauptprogramm kennt zwei Modi:

- Quelltexte wurden per Kommandozeile übergeben und

- es wurden keine übergeben, ergo wird von der Standardeingabe gelesen.

Damit ergibt sich folgender Quelltext:

```
#include <iostream.h>
#include <fstream.h>
#include "parser.hpp"

int main(int argc, char *argv[])
{
       cout << "Calculator calc7 - Copyright (c) 1997 "
            << "by Oliver Mueller" << endl << endl;
       if(argc > 1) {
              ifstream file;
              file.unsetf(ios::skipws);
              Parser calculator_file(&file);
              int n;
              for(n=1;argv[n];n++) {
                     file.open(argv[n]);
                     calculator_file.Run();
                     file.close();
              }
       } else {
              Parser calculator_cin(&cin);
              calculator_cin.Run();
       }
       cout << endl << "Bye!" << endl;
       return 0;
}
```

5 Scannergenerator lex

Die Programmierung eines Scanners von Hand ist nicht gerade zeitdefensiv, deshalb wurden Werkzeuge - sogenannte *Scannergeneratoren* - entwickelt, die diese Arbeit automatisieren. In den 70er Jahren entstand der wohl gebräuchlichste Scannergenerator *lex*. Er wurde von M. E. Leck und E. Schmidt als Zusatz zu yacc, einem Parsergenerator, entwickelt.

lex arbeitet nach dem Prinzip regulären Ausdrücken C-Code zuzuordnen. Der Benutzer muß hierzu ledlich den regulären Ausdruck angegeben und lex generiert automatisch den Automaten für die Erkennung. Zu jedem regulären Ausdruck muß der Benutzer dann ledilich noch auszuführende Aktionen in C angeben und im Zuge dieser zugeordneten Aktionen das Token aufbauen.

flex

Auf der dem Buch beiliegenden CD-ROM finden Sie den freiverfügbaren flex, den lex-Compiler des GNU-Projekts. Dieses Programm flex ist mittlerweile für eine breite Palette von System- und Hardware-Plattformen verfügbar. Außerdem ist der Quelltext ebenfalls - in den Schranken der GNU General Public Licence - frei verfügbar. Falls für Ihr System kein Executable auf der CD-ROM enthalten sein sollte, können Sie immernoch den Quelltext compilieren.

Kurzum: Durch flex können Sie sofort in medias res gehen und das hier vermittelte Wissen - ohne zusätzliche Kosten - umsetzen.

5.1 Einführung in lex

lex ist eine Art Compiler, der lex-Programme in C-Quelltexte umsetzt. Die so gewonnen C-Quelltexte können dann anschließend mit einem C-Compiler zu ausführbaren Dateien compiliert werden.

lex-Quelltexte haben für gewöhnlich die Extension .l. Sie werden durch einfaches ausführen des Kommandos

```
lex datei.l
```

in die Datei lex.yy.c „compiliert“, wobei datei.l der lex-Quelltext ist. In lex.yy.c findet sich die Funktion `yylex()`, die den Scanner kapselt. Sie gibt jeweils einen Integerwert

zurück, der, wenn er ungleich Null ist, ein Token identifiziert und EOF signalisiert, wenn er gleich Null ist.

Hinweis: Bei den DOS-Clonen von lex, insbesondere der DOS-Version von flex, heißt die Ausgabedatei nicht lex.yy.c, sondern aufgrund der 8+3-Beschränkung von Dateinamen lexyy.c. Im folgenden wird diese Datei jedoch wie unter Unix üblich als lex.yy.c bezeichnet.

Diese Datei lex.yy.c kann mit dem C(++)-Compiler compiliert werden. Welche Optionen beim Start von lex und später beim Compilieren von lex.yy.c anzugeben sind, hängt vom verwendeten lex-Clone ab. Hier sei auf die jeweilige Dokumentation verwiesen.

5.1.1 Aufbau eines lex-Quelltextes

Ein lex-Programm setzt sich aus drei Abschnitten zusammen, die jeweils durch ein %% voneinander getrennt sind:

```
Definitionsteil
%%
Regelteil
%%
Routinenteil
```

Definitions- und Routinenteil sind optional. Der Regelteil ist jedoch für die Spezifikation der Lexik einer Sprache praktisch unerläßlich, denn was sollte ein Scanner ohne zu erkennende Lexeme?

Es ist zwar prinzipiell ohne weiteres möglich ein lex-Programm ohne Regelteil zu schreiben, aber welchen praktischen Nutzen soll dieses Programm haben?

Falls der Routinenteil entfällt, kann auch das zweite %% entfallen. Das erste %% ist jedoch unerläßtlich, selbst wenn kein Definitionsteil angegeben wurde.

Der *Definitionsteil* enthält Deklarationen von C-Variablen, Prototypen von Routinen, Präprozessordirektiven (z. B. #include oder #define) und reguläre Definitionen. Die regulären Definitionen ermöglichen es, wie die aus Kapitel 3 bekannten, reguläre Ausdrücke zu „benennen“. Damit können reguläre Ausdrücke im Regelteil einfacher und übersichticher spezifiziert werden.

Der *Regelteil* enthält Anweisungen der Form

```
Pattern1      Aktion1
Pattern2      Aktion2
...
Patternn      Aktionn
```

Jedes `Pattern`$_i$ ist dabei ein regulärer Ausdruck und jede `Aktion`$_i$ ist ein zugehöriger Code-Block, der ausgeführt werden soll, wenn ein auf `Pattern`$_i$ passendes Lexem erkannt wurde.

Der *Routinenteil* bietet die Möglichkeit direkt im Anschluß an das lex-Programm entsprechende Funktionen und Prozeduren in C(++) zu implementieren, die für die Auswertung der Lexeme, d. h. in den Aktionen, benötigt werden. Es ist allerdings auch denkbar diese Routinen in gesonderten Modulen zu implementieren und per Linker hinzuzubinden.

5.1.2 Der Definitionsteil

Im Definitionsteil können Festlegungen für das folgende lex-Programm getroffen werden. Im folgenden wollen wir die in der Praxis häufigst gebrauchtesten Features dieses Parts betrachten. Für tiefergehende Informationen und Feinheiten sei auf [6] verwiesen.

C-Code einbinden

Im Definitionsteil ist es möglich C-Code einzubinden. Hierzu müssen die entsprechenden Zeilen zwischen die Zeilen

```
%{
```

und

```
%}
```

eingeschlossen werden.

Zwischen diese Zeilen kann jeder in C(++) gültige Code - auch Präprozessordirektiven - stehen.

Ein Beispiel wäre

```
%{
#include <stdio.h>
#include <stdlib.h>

unsigned n, m;
void help();
%}
```

Es muß lediglich darauf geachtet werden, daß `%{` und `%}` jeweils allein in einer Zeile stehen.

regulärer Ausdruck	*Bedeutung*	*Beispiel*
^	Zeilenanfang	^A deckt A am Zeilenanfang ab
$	Zeilenende	A$ paßt auf A am Zeilenende
.	Beliebiges Zeichen außer \n	
[...]	Zeichenklasse	[+-] entspricht + oder - [0-9] deckt jede Ziffer ab
[^...]	Komplementzeichenklasse	[^+-] entspricht jedem Zeichen außer + und -
ab	Konkatenation	AB deckt A unmittelbar von B gefolgt ab
a|b	Alternation	A|B paßt entweder auf A oder B
(a)	Klammerung	
a*	Kleene-Hülle von a	A* deckt beliebige Wiederholung (auch Null) von A ab.
a+	Positive Hülle von a	A+ deckt beliebige Wiederholung (mindestens 1 Vorkommen) von A ab.
a?	Optionales a	A? = optionales A
\c	Sonderbedeutung von c aufheben oder Escapesequenz	* entspricht * \n entspricht „New Line“ \\ entspricht „Backslash“
"..."	Stringbildung, entspricht C-String	"a+b" entspricht a+b und nicht ab, aab, etc.
a{n,m}	n- bis m-fache Wiederholung von a	A{2,7} entspricht einer Folge von 2 bis 7 A
{rd}	Reguläre Defintion mit Namen rd einbinden	
a/b	Kontextsensitivität, deckt a nur ab, wenn b folgt	A/B deckt A ab, wenn B folgt; B gehört aber nicht zum abgedeckten Lexem!
<SB>a	a wird nur erkannt, wenn Startbedingung SB gilt	

Tab. 5.1: Reguläre Ausdrücke in lex

Reguläre Definitionen

Jede Zeile im Defintionsteil, die nicht in `%{` und `%}` eingeschlossen ist und mit einem Buchstaben beginnt, wird als *reguläre Defintion* aufgefaßt. Reguläre Defintionen sind in lex mit Bezeichnern belegte reguläre Ausdrücke.

Eine reguläre Defintion steht immer allein in einer Zeile folgenden Formats:

```
Name    regulärer_Ausdruck
```

Name und regulärer Ausdruck müssen mindestens durch ein Leer- oder Tabulatorzeichen getrennt sein. Durch {Name} können diese regulären Defintionen

später in den Patterns des Regelteils verwendet werden. Doch hierzu an gegebener Stelle mehr.

Tabelle 5.1 gibt Auskunft über die von lex akzeptierten regulären Ausdrücke.

Beispiele für reguläre Defintionen wären:

```
Letter    [A-Za-z]
Digit     [0-9]
WS        [ \n\r\t\f\v]
```

`Letter` bezeichnet den regulären Ausdruck, der Buchstaben abdeckt. `Digit` steht für alle Ziffern und `WS` bezeichnet Whitespaces.

5.1.3 Der Routinenteil

Der *Routinenteil* von lex ermöglicht es C-Code direkt im lex-Programm einzugeben. Alles was am Anschluß an das zweite %% folgt wird unverändert an lex.yy.c angehängt. Klammerungen wie durch %{ und %} wie im Defintionsteil sind hier nicht nötig.

Die Möglichkeit C-Code im Defintionsteil einzubinden ist im wesentlichen dafür vorgesehen, entsprechende Prototypen von Routinen anzugeben, Variablen zu deklarieren oder auch Headerfiles einzubinden. Im Routinenteil werden hingegen die - eventuell als Prototypen im Defintionsteil deklaraierten - Unterprogramme implementiert.

5.1.4 Der Regelteil

Das Herz von lex ist der Regelteil. Hier werden die Regeln für die Lexeme angegeben, also die Lexik spezifiziert.

Jede Regel beginnt in einer neuen Zeile folgenden Formats:

```
Pattern    Aktion
```

`Pattern` und `Aktion` sind dabei durch mindestens ein Leer- oder Tabulatorzeichen getrennt. `Pattern` ist ein nach Tabelle 5.1 gültiger regulärer Ausdruck. `Aktion` ist ein Code-Block in C, der wenn er mehr als eine C-Anweisung enthält oder sich über mehrere Zeilen erstreckt in geschweifte Klammern gefaßt wird. Dieser Code-Block wird immer dann ausgeführt, wenn ein auf `Pattern` passendes Lexem erkannt wurde.

Die Aktionen in einem lex-Programm haben die gleiche Aufgabe wie die Aktionen bei erreichen des Endzustand in den automatenbasierten Scannern des dritten Kapitels. Die Hauptaufgabe ist hier das Token aufzubauen und es an den Parser zurückzuliefern.

lex ist ein Werkzeug, daß nicht nur bei der Parserprogrammierung oder im Compilerbau Anwendung findet. Daher resultiert eine Eigenheit, die, wenn man sie nicht beachtet, unangenehme Nebeneffekte verursacht.

Wird ein Zeichen gelesen, daß auf keinen der in den Regeln angegebenen regulären Ausdrücke paßt, so schaltet sich die *Default-Echo-Funktion* ein. Mit anderen Worten: Das Zeichen wird schlicht über den Standardausgabekanal ausgegeben. Um diesen unangenehmen Nebeneffekt zu vermeiden, müssen alle möglichen (und unmöglichen) Zeichen in den Regeln beachtet werden.

Nichts ist jedoch so schlecht, daß man nicht auch etwas positives daran entdeckt. Der Programmierer ist daran gebunden alle Zeichen zu beachten, auch die lexikalisch falschen. Im Falle von ungültigen Zeichen kann (muß) auf diese Weise direkt die Fehlerbehandlung aktiviert werden.

Der folgende Quelltext zählt beispielsweise alle Buchstaben und Ziffern in einer Datei:

```
%{
#include <iostream.h>
unsigned long letters = 0, digits = 0;
%}

%%

[A-Za-zÖöÜüÄäß]  letters++;
[0-9]            digits++;
.                ;
\n               ;

%%

int main()
{
       yylex();
       cout << "Buchstaben: " << letters << endl
            << "Ziffern   : " << digits << endl;
       return 0;
}
```

Sowie ein Buchstabe erkannt wird, wird die Variable letters inkrementiert. Im Falle einer Ziffer wird digits erhöht. Die beiden letzten Regeln für jedes beliebige Zeichen außer \n und für \n enthalten sogenannte *Leeraktionen*, da sie keinen Code enthalten. Solche minimalen „Aktionen“ werden durch einen einzelnes Semikolon dargestellt.

Sie sehen das „Default-Echo-Problem“ tritt hier nicht auf, da durch die letzten beiden Regeln sichergestellt ist, daß kein Zeichen unabgedeckt bleibt.

Die beiden letzten Regeln könnten auch durch das *Aktionszeichen* | ausgedrückt werden. Da für beide regulären Ausdrücke die gleichen Aktionen - nämlich keine - auszuführen sind, liegt es nahe die Regeln miteinander zu verbinden.

```
.       |
\n      ;
```

Dies ist zum Beispiel dann sinnvoll, wenn ein größerer Code-Block angebeben wurde und dieser für beide Pattern identisch wäre. Diese Redundanz läßt sich durch das Aktionszeichen entfernen.

Angenommen Sie haben drei reguläre Ausdrücke, die verschiedene Anreden beschreiben und für alle muß die gleiche Variable inkrementiert und die gleiche Routine aufgerufen werden. Sie können dies durch folgenden Quelltext verkürzt ausdrücken:

```
Frau    |
Herr    |
Firma   { Anrede++; AnredeRoutine(); }
```

statt

```
Frau    { Anrede++; AnredeRoutine(); }
Herr    { Anrede++; AnredeRoutine(); }
Firma   { Anrede++; AnredeRoutine(); }
```

yytext und yyleng

Reguläre Ausdrücke haben bekanntlich die Eigenschaft nicht nur konkrete Strings, sondern auch Muster, also die Gemeinsamkeiten einer Menge von Strings, zu beschreiben. Hier ist es in lex häufig unerläßlich in den Aktionen auf das tatsächlich erkannte Lexem zuzugreifen.

lex speichert das aktuell erkannte Lexem in der Variable `yytext`, welche ein char-Array ist. Diese Variable können Sie mit dem Attribut `lexem` und der Methode `getLexem()` der Scanner-Klasse aus Kapitel 3 vergleichen.

Die Länge des Lexems speichert lex in der Variablen `yyleng`. Diese Variable ist vergleichbar mit dem Attribut `lexem_len` der Scanner-Klasse.

5.2 Ein Scanner in lex

Zunächst wollen wir einen Scanner programmieren, der die gleichen Tokens erkennt wie der Scanner aus dem vorherigen Kapitel. Im nächsten Kapitel wird dann ein Parser in yacc hinzukommen, der die gleiche Sprache erkennt wie der Parser aus dem letzten Kapitel. Lediglich die Fehlerbehandlung werden wir in dieser Ausbaustufe stiefmütterlich behandeln.

Insgesamt wird es drei Ausbaustufen des Scanners in lex und des Parsers in yacc geben:

1. Funktionalität wie das System aus dem vorherigen Kapitel.
2. Auf 1. aufbauende arithmetische Programmiersprache mit Kontrollstrukturen.
3. Auf 2. aufbauende arithmetische Programmiersprache mit Unterprogrammen.

Um Sie jedoch mit lex (und später mit yacc) bekannt zu machen, ist es günstig zunächst ein System zu entwerfen, daß die gleiche Leistungsfähigkeit bereitsstellt wie das Ihnen bereits vertraute System des vorherigen Kapitels. Dadurch, daß sich keine Neuerungen an der Sprache ergeben, werden Sie eine gute Kontroll- und Vergleichsmöglichkeit haben.

Die Stack-Maschine, die Fehlerbehandlung und die Symboltabelle werden wir in den folgenden System weiterverwenden bzw. erweitern.

5.2.1 Stufe 1

Die erste Stufe das Scanners präsentiert sich recht schlicht. Da die Fehlerbehandlungsmöglichkeiten in der lexikalischen Analyse bekanntlich sehr begrenzt sind und auch der Einsatz von lex an diesem Umstand nichts ändert, wird sämtliche Fehlerbehandlung an den Parser weitergereicht. Selbst ungültige Zeichen werden nicht im Scanner behandelt. Wie Sie feststellen werden, wird dies weder den Parser unnötig aufblähen, noch an der Güte der Fehlermeldung etwas ändern. Der einzige Effekt ist, daß der Scanner bzw. das lex-Programm schlank gehalten wird.

Der Quelltext:

```
%{
#include "symtable.hpp"
#include "y.tab.h"

extern SymbolTable symTable;
%}

WS     [ \t]
D      [0-9]
L      [_A-Za-z]
LD     [_A-Za-z0-9]

%%

write  return WRITE;
```

```
read   return READ;
\*\*   return POWER;
{WS}   ; /* Whitespaces ignorieren */
({D}+\.?|{D}*\.{D}+)([eE][+-]?{D}+)? {
       sscanf(yytext, "%lg", &yylval.val);
       return NUM;
       }
{L}{LD}* {
       yylval.sym = symTable.update(yytext);
       return ID;
       }
\n     ;
.      return yytext[0];
```

Im Defintionsteil findet sich ein geklammerter C-Code. Die Headerdatei "`symtable.hpp`" ist wegen der Deklaration der Klasse `SymbolTable` eingebunden. Durch die extern-Anweisung wird ein später im Parser deklarierte Instanz von `SymbolTable` eingebunden.

lex (wie auch yacc) stammt leider noch aus Zeiten als die strukturierte Programmierung noch der Stand der Dinge war. Deshalb kann der Scanner nicht als Klasse definiert werden. Wie schon zuvor erwähnt wird von lex eine Funktion namens `yylex()` generiert, die den Scanner kapselt. Wegen dieser „strukturierten Altlast“ müssen wir auch die Symboltabelle als eigenständige Variable deklarieren. Sämtliche Schutzmechanismen des Information-Hiding entfallen somit.

Die Headerdatei "`y.tab.h`" ist ein Novum. Diese Datei wird von yacc generiert und enthält sämtliche Integerwerte für die Tokens. Um die Verwaltung dieser Werte müssen Sie sich nicht mehr wie in Kapitel 3 selbst kümmern.

Hinweis: Unter DOS wird diese Datei y_tab.h genannt.

Nun folgen noch reguläre Defintionen:

- `WS` für Whitespace-Zeichen,
- `D` für Ziffern,
- `L` für Buchstaben und
- `LD` für Buchstaben und Ziffern.

Nun folgt der Regelteil. Die ersten beiden Regeln decken die Schlüsselwörter `write` und `read` ab. Da Schlüsselwörter keine Token-Attribute benötigen, genügt es hier den Symbolwert zurückzugeben. Dies geschieht mittels der C-Anweisung `return` und dem Tokenwert aus y.tab.h. Sie müssen sich vorstellen, daß der gesamte Regelteil lediglich Part der Funktion `yylex()` ist, die Sie durch

```
return Tokenwert;
```
verlassen können. Der Rückgabewert wird dann vom Parser ausgewertet.

Die nächste Regel ist der Potenzoperator **. Da der Stern für Kleene-Hüllen reserviert ist, muß diese Sonderbedeutung durch vorangestellten Backslash aufgehoben werden.

Alternativ könnten Sie diese Regel auch durch folgende Zeile

```
"**"   return POWER;
```
ausdrücken.

Nun folgend die Whitespaces unter Verwendung der regulären Defintion `WS`. Wie Sie sehen, werden reguläre Defintionen in das Pattern durch Umfassen mit geschweiften Klammern eingebunden.

Whitespaces sollen ignoriert werden, deshalb die Leeraktion.

Die nun folgenden Regeln sind etwas komplexer. Sie behandeln die Fließkommakonstanten á la C und die Bezeichner. Diese beiden regulären Ausdrücke haben gemein, daß sie Strings variabler Länge beschreiben und Tokens mit Attributen (den Wert der Konstante bzw. den Verweis auf den Symboltabelleneintrag) zurückliefern.

Hier kommt jeweils `yytext` zu Einsatz. Allerdings wird auch eine andere Variable - `yylval` - eingesetzt. Diese globale dient zur Aufnahme der Attributwerte. Ihre Struktur wird vom Parser, also im yacc-Programm des nächsten Kapitels, festgelegt.

In unserem yacc-Programm werden wir diese Variable als Strukturtyp mit folgenden Elementvariablen deklarieren:

```
double val;
SYMHANDLE sym;
```
Für jeden Attributtyp werden wir also eine eigene Elementvariable deklarieren. Die Notwendigkeit dieses Weges anstatt - wie im dritten Kapitel - einer einzigen Attributvariable, wird bei der Betrachtung von yacc ersichtlich werden.

Die nächste Regel dient zur Eleminierung von Zeilentrennern.

Die letzte Regel hat zwei Aufgaben:

a)Ungültige Zeichen werden an den Parser weitergereicht und dort der Fehlerbehandlung zugeführt.

b)Die jeweils ein Zeichen umfassenden Tokens für die Operatoren werden dem Parser übergeben ohne gesonderte Regeln aufstellen zu müssen.

In b) fungieren die ASCII-Werte dieser Zeichen als Integertokenwerte. Hierbei ist eine Eigenheit von yacc zu erwähnen. Alle Tokenwerte, die in y.tab.h deklariert werden,

haben einen Wert größer 257. ASCII-Werte sind jedoch maximal 8 Bit breit und haben somit einen Wertbereich von 0 bis 255 und können damit direkt als Integerwerte fpr Tokens genutzt werden.

Ein Routinenteil entfällt in diesem Quelltext. Der Scanner der Stufe 1 ist implementiert.

5.2.2 Stufe 2

Die zweite Stufe des Parsing-Systems wird die Möglichkeiten der Stack-Maschine erstmals nutzen können bzw. die Stack-Maschine wird die Grundvoraussetzung für Stufe 2 sein. Die Neuerungen gegenüber der ersten Stufe werden Kontrollstrukturen und wieder ein aussagekräftiges Fehlerbehandlungs- und -Recovery-Subsystem sein.

Die Kontrollstrukturen werden bedingte Verzweigungen (if) und Schleifen (while) sein. Der Aufbau des If-Konstruktes wird dabei wie folgt gestaltet sein:

```
if Bedingung then
       Anweisungen;
else
       Answeisungen;
endif;
```

bzw.

```
if Bedingung then
       Anweisungen;
endif;
```

Die While-Schleifen werden wie folgt aufgebaut sein:

```
while Bedingung do
       Anweisungen;
wend;
```

Damit ergeben sich die neuen Schlüsselwörter

- `if`,
- `then`,
- `else`,
- `endif`,
- `while`,
- `do` und
- `wend`.

Die Bedingungen werden folgende Operatoren unterstützen:

- Vergleiche mittels >, >=, <, <=, == und !=.
- Logische Verknüpfungen mit den Schlüsselwörtern `and`, `or` und `not`.

Damit ergibt sich der modifzierte Quelltext:

```
%{
#include "symtable.hpp"
#include "machine.hpp"
#include "y.tab.h"

extern SymbolTable symTable;
extern unsigned long lineno, colno;
%}

WS     [ \t]
D      [0-9]
L      [_A-Za-z]
LD     [_A-Za-z0-9]
%%

and    { colno += 3; return AND; }
if     { colno += 2; return IF; }
else   { colno += 4; return ELSE; }
endif  { colno += 5; return ENDIF; }
do     { colno += 2; return DO; }
not    { colno += 3; return NOT; }
or     { colno += 2; return OR; }
while  { colno += 5; return WHILE; }
wend   { colno += 4; return WEND; }
write  { colno += 5; return WRITE; }
read   { colno += 4; return READ; }
then   { colno += 4; return THEN; }
">="   { colno += 2; return GE; }
">"    { colno++; return GT; }
"<="   { colno += 2; return LE; }
"<"    { colno++; return LT; }
"=="   { colno += 2; return EQ; }
"!="   { colno += 2; return NE; }
\*\*   { colno += 2; return POWER; }
{WS}   { colno++; }
```

```
({D}+\.?|{D}*\.{D}+)([eE][+-]?{D}+)? {
       sscanf(yytext, "%lg", &yylval.val);
       colno += yyleng;
       return NUM;
       }
{L}{LD}* {
       yylval.sym = symTable.update(yytext);
       colno += yyleng;
       return ID;
       }
\n     { lineno++; colno = 0; }
.      { colno++; return yytext[0]; }
```

Für die Fehlerbehandlung wollen wir wieder die Fehlerposition durch Zeilen- und Spaltennummer spezifizieren. Hierzu werden im Parser-Modul die Variablen `lineno` und `colno` eingeführt, die im obigen Definitionsteil als extern referenziert sind.

Bei jedem erkannten Lexem wird `colno` aktualisiert. `lineno` und `colno` werden lediglich beim Lesen eines Zeilenvorschubs (\n) korrigiert.

5.2.3 Stufe 3

Die dritte Stufe unterscheidet sich von der zweiten Stufe im wesentlichen dadurch, daß sie auch die benutzerdefinierte Unterprogramm gestattet. Diese Unterprogramme werden allerdings eher Makrocharakter haben, da sie keine Parameter und Rückgabewerte zulassen.

Unterprogramme werden wie folgt definiert werden können:

```
proc Bezeichner is
       Anweisungen;
end;
```

Aufgerufen werden sie durch folgendes Statement:

```
call(Bezeicher);
```

Damit ergeben sich vier neue Schlüsselwörter:

- `proc`,
- `is`,
- `end` und
- `call`.

Der lex-Quelltext erfährt damit keine allzugroßen Änderungen:

```
...
%%

and    { colno += 3; return AND; }
call   { colno += 4; return CALL; }
if     { colno += 2; return IF; }
is     { colno += 2; return IS; }
else   { colno += 4; return ELSE; }
endif  { colno += 5; return ENDIF; }
end    { colno += 3; return END; }
do     { colno += 2; return DO; }
not    { colno += 3; return NOT; }
or     { colno += 2; return OR; }
proc   { colno += 4; return PROC; }
while  { colno += 5; return WHILE; }
wend   { colno += 4; return WEND; }
write  { colno += 5; return WRITE; }
read   { colno += 4; return READ; }
then   { colno += 4; return THEN; }
">="   { colno += 2; return GE; }
...
```

6 Parsergenerator yacc

In den frühen 70er Jahren herrschte eine wahre Welle der Euphorie für Parsergeneratoren bzw. Compiler-Compiler wie sie auch genannt werden. Endlich konnte man durch einfache Spezifikationssprachen Parser automatisch erstellen, ohne sich mühselig um die Codierung von Grammatiken zu kümmern. Diese Popularitätswelle für Parsergeneratoren führte allerdings auch dazu, daß es damals eine unüberschaubare Menge dieser Tools gab.

Als S. C. Johnson von den Bell Labs 1972 einen weiteren Vertreter dieser Werkzeugzunft dem Licht der Welt aussetzte, trug er diesem Umstand durch die von Selbstironie geprägte Bezeichnung „yet another compiler compiler“, kurz yacc, Rechnung. Also „noch ein anderer Compiler-Compiler“ der im Werkzeugdjschungel drohte unterzugehen? - Nein, yacc ist eines der wenigen Tools, das bis heute überlebte.

yacc ist jetzt bei Erscheinen dieses Buches bereits über 25 Jahre alt - ein Viertel Jahrhundert. 1972: Unix war noch ein Geheimtip, DOS noch nicht erfunden, der Umgang mit Lochkarten war Alltag und die strukturierte Programmierung feierte damals die explosionsartigen Erfolge, die heute das objektorientierte Paradigma erreicht.

Das Erfolgsrezept dieses Veteranen der Computergeschichte, das es ihm ermöglichte ein für die Informationstechnik biblisches Alter zu erreichen, läßt sich in wenigen Worten skizzieren: Die Spezifikationssprache von yacc ist der BNF sehr ähnlich. Jeder der mit kontextfreien Grammatiken umgehen kann, findet sich in yacc schnell zurecht.

6.1 Einführung in yacc

Wie im Kapitel über lex werden wir auch hier nur die grundlegenden Elemente von yacc besprechen. Jedes Detail und jede Feinheit würde den Rahmen dieses Buches sprengen. An weitergehenden Informationen Interessierte seinen wiederum an [6] verwiesen.

Wie lex genertiert auch yacc C-Code. Die Quelltextdateien von yacc tragen für gewöhnlich die Erweiterung .y. Die Ausgabedatei nennt sich y.tab.c. Bei Angabe der Option -d erzeugt yacc auch die Datei y.tab.h, die alle Token-Konstanten enthält (vgl. Kapitel 5).

Die „Compilierung“ einer yacc-Datei erfolgt durch folgendes Kommando

```
yacc Datei.y
```

bzw.

```
yacc -d Datei.y
```

, wenn y.tab.h auch erzeugt werden soll.

Hinweis: Unter DOS werden die Dateien y.tab.c und y.tab.h in der Regel aufgrund der 8+3-Beschränkung des FAT-Systems y_tab.c respektive y_tab.h genannt.

bison

Auch zu yacc bietet das GNU-Projekt eine hervorragende kostenlose Alternative, die sie auch auf der beiliegenden CD-ROM vorfinden. Etwas scherzhaft nennt sich dieser yacc-Clone *bison*. Der Quelltext ist ebenfalls wie bei flex frei verfügbar.

Hinweis zu bison: Der frei verfügbare yacc-Clone bison hat seine kleinen Eigenheiten. bison nennt die Ausgabedateien für eine Eingabedatei Datei.y nicht y.tab.c und y.tab.h, sondern Datei.tab.c und Datei.tab.h. Dies mag eine sinnvolle Erweiterung sein, aber ist mit dem Verhalten von yacc nicht kompatibel. Deshalb bietet bison eine zusätzliche Kommandozeilenoption -y, die bison in den yacc-Kompatibilitätsmodus schaltet.

```
bison -y Datei.y
```

erzeugt also y.tab.c und

```
bison -y -d Datei.y
```

erzeugt y.tab.c und y.tab.h.

6.1.1 Aufbau eines yacc-Quelltextes

yacc-Programme gliedern sich wie lex-Quelltexte in drei Teile:

```
Defintionsteil
%%
Regelteil
%%
Routinenteil
```

Der *Defintionsteil* enthält wieder geklammerten C-Code und Vereinbarungen für yacc. Die Tokens und die mit ihnen verbunden Integerwerte, die von lex genutzt werden, werden hier festgelegt; ebenso die Priorität und Assiziativität von Operatoren.

Der *Regelteil* enthält die kontextfreie Grammatik. Zu jedem Symbol können hierbei Aktionen festgelegt werden.

Der *Routinenteil* dient wieder der Implementierung von Unterprogrammen und sonstigen C(++)-Konstrukten, die vom Regelteil benötigt werden.

6.1.2 Der Definitionsteil

Der Defintionsteil enthält, wie der Name schon sagt, Festlegungen für die im Regelteil folgende Grammatik bzw. deren Implementierung im Parser. Hier kann wie bei lex in %{ und %} C(++)-Code angegeben werden. An dieser Stelle sollte man sich allerdings wieder auf Deklarationen beschränken, die eigentliche Implementierung von Unterprogrammen sollte im Routinenteil erfolgen.

Die wesentlichen Aufgaben des Definitionsteils aus der Sicht von yacc sind jedoch

- Token- und Typverwaltung,
- Festlegung der Operatorenpriorität und
- Defintion der Assoziativität der Operatoren.

Tokens und Typen

Tokens werden in yacc durch die %token-Klausel bestimmt:

```
%token Token1 Token2 Token3 ...
```

bzw.

```
%token Token1
%token Token2
%token Token3
...
```

Für jedes so definierte Token wird von yacc eine Integer-Konstante generiert und diese bei Angabe der Option -d in der Datei y.tab.h gespeichert. Diese Konstanten beginnen bei 257 und werden für jedes weitere Token um eins hochgezählt. Die Werte 0 bis 255 sind für die ASCII-Werte von „Ein-Byte-Tokens“ vorgesehen (vgl. Kapitel 5). Der Wert 256 wird von yacc für interne Zwecke - genauer die Fehlerbehandlung - verwendet.

Um beispielsweise die Tokens NUM und ID zu definieren, ist schlicht

```
%token NUM ID
```

anzugeben.

Prinzipiell ist es möglich auch Zeichenkonstanten als Tokens zu deklarieren, e. g.

```
%token '+'
```

Dies ist jedoch überflüssig, da diese Angabe an der Funktionsweise des yacc-Programms nichts ändert oder irgendeinen Vorteil verspricht.

Das Startsymbol kann durch die Anweisung

```
%start Nichtterminal
```

festgelegt werden.

Findet sich kein entsprechendes Statement im Defintionsteil, so wird einfach das Nichtterminal der ersten Produktion im Regelteil als Startsymbol deklariert.

Die so definierten Tokens haben bislang kein Attribut vom Programmierer festgelegtes Attribut erhalten. Implizit geht yacc davon aus, daß die Tokens ein Attribut vom Typ int besitzen. Die im vorherigen Kapitel bereits angesprochene Variable `yylval` ist demnach implizit als int deklariert. Für sehr einfache Grammatiken ist dies akzeptabel, aber für komplexere Systeme ist dies nicht sinnvoll.

yacc bietet daher einerseits die Möglichkeit die Variable `yylval` als Strukturdatentyp umzudeklarieren und die Elementvariablen den Tokens als Attribute zuzuweisen.

Zur Deklaration existiert die Klausel %union. Sie hat folgenden Aufbau:

```
%union {
       Typ1   Komponente1;
       Typ2   Komponente2;
       ...
}
```

Typ_i ist ein gültiger C(++)-Datentyp. $Komponente_i$ ein gültiger Bezeichner.

Angenommen Ihr Parser soll String- und Integer-Konstanten verarbeiten, dann ist es unerläßlich für die korrespondierenden Tokens Attriubte mit den jeweiligen Typen festzulegen. Ein Token für eine String-Konstante würde beispielsweise als 2-Tupel {STRING, "Oliver"} dargestellt. Ein Token einer Integer-Konstante hingegen vielleicht {INTEGER, 250674}. Somit ist es nötig für die Speicherung der Attribute geeignete Elementvariablen in yylval einzuführen. Dies geschieht zum Beispiel durch

```
%union {
       char *str;
       int val;
}
```

Von yacc wird diese Anweisung in ein C-Konstrukt wie

```
typedef union {
       char *str;
       int val
} YYSTYPE;
```

umgesetzt, auf das die Zeile

```
YYSTYPE yylval;
```

in y.tab.c folgt. Ähnliche Anweisungen finden sich auch in y.tab.h, damit lex ebenfalls mit dieser Variable umgehen kann.

Um den Tokens nun die jeweiligen Attributtypen zuzuweisen, gibt man schlicht in <...> zwischen %token und den folgenden Tokens den Namen der Komponente an:

```
%token <Komponentenname> Token1 Token2 ...
```

String- und Integer-Konstanten würden zum Beispiel durch die folgenden Anweisungen entsprechend deklariert:

```
%token <str> STRING
%token <val> INTEGER
```

In Grammatiken ist es auch häufig nötig Nichtterminalen einen Typ zuzuweisen, der mit einem Attribut gleichzusetzen ist. Betrachten Sie hierzu beispielseweise die folgende kontextfreie Grammatik:

```
<expr>   --> <expr> '+' <factor>
           | <expr> '-' <factor>
           | <factor>
<factor> --> <factor> '*' <term>
           | <factor> '/' <term>
           | <term>
<term>   --> NUM
           | ID
```

Wenn wir diese Grammatik in einen Parser umsetzen, der wie in Kapitel 2 direkt die Verarbeitung implementiert, also keine Stack-Maschine für die Berechnungen verwendet, so müssen den Nichtterminalen der Typ NUM zugewiesen werden. Es muß yacc mitgeteilt werden, daß zum Beispiel `<term>` einen Zahlenwert zurückgibt und nicht etwa den Namen eines Bezeichners (ID) als String.

Hierfür existiert in yacc die Anweisung %type, die mit der %token-Klausel gewisse Ähnlichkeiten aufweist:

```
%type <Komponentenname> Nichtterminal1 Nichtterminal2 ...
```

Um nun die Nichtterminale von oben mit den entsprechenden Werten zu belegen, muß folgende Anweisung angegeben werden:

```
%type <val> expr term factor
```

`val` ist hierbei der Komponentenname, der dem Attribut von NUM entspricht.

Priorität und Assoziativität

Die Assoziativität von Tokens und Zeichenkonstanten, die als Operatoren fungieren, kann durch folgende Klauseln bestimmt werden:

- Linksassoziativität:

  ```
  %left Operator1 Operator2 Operator3 ...
  ```

- Rechtsassoziativität:

  `%right Operator`$_1$ `Operator`$_2$ `Operator`$_3$ `...`

- Keine Assoziativität:

  `%nonassoc Operator`$_1$ `Operator`$_2$ `Operator`$_3$ `...`

Die Operatorpriorität wird durch die Folge der Anweisungen bestimmt. Zuletzt angegebene Assoziativitätsanweisungen geben den jeweiligen Operatoren die höchste Priorität.

```
%right '='
%left '+' '-'
%left '*' '/'
%right POWER
```

Der Zuweisungsoperator = und der Potenzoperator POWER sind rechtsassoziativ, die übrigen linksassoziativ. Der Zuweisungsoperator hat hier die geringste Priorität, POWER die höchste.

In Assoziativitätsanweisungen können Sie ebenfalls Komponenten von `yylval` zuweisen. Dies geschieht wie bei %token und %type durch <...>.

6.1.3 Der Routinenteil

Der Routinenteil ist wieder für die Aufnahme von C(++)-Code vorgesehen. Dieser Teil wird unverändert an die Datei y.tab.c angehängt.

In der Praxis werden hier Routinen definiert, die von den Produktionen des Regelteils benötigt werden. Drohen die Aktionen des Regelteils zu komplex zu werden, ist es oft sinnvoll hierfür entsprechende Unterprogramme zu definieren.

6.1.4 Der Regelteil

Die Regeln des Regelteils sind ähnlich aufgebaut wie die der BNF. Auf der linken Seite findet sich das Nichtterminal und auf der rechten Seite die Produktion. Die Trennung in linke und rechte Seite erfolgt durch einen Doppelpunkt. Die gesamte Produktion wird durch einen Strichpunkt abgeschlossen.

Eine Regel in yacc hat demnach folgenden Aufbau:

```
Nichterminal: Symbole der Produktion;
```

Mehrere Produktionen für ein Nichtterminal können durch | miteinander verkettet werden. Statt

```
A: B C;
```

```
A: D E F;
A: G H I J;
```

kann

```
A: B C
 | D E F
 | G H I J
 ;
```

geschrieben werden. Die Syntax der Regeln in yacc ist im übrigen formatfrei, d. h. Leerzeichen, Tabulatoren und Zeilentrenner werden ignoriert. Kommentare können in /* und */ geklammert angegeben werden.

ε-Produktionen können ebenfalls angegeben werden. Hierzu ist schlicht kein Symbol auf der rechten Seite anzugeben. Häufig werden jedoch solche ε-Produktionen durch einen Kommentar gekennzeichnet. Beispiele wären:

```
A: ;
B: /* empty */
 ;
C: /* leer */
 | D E
 ;
```

Die Unterscheidung zwischen terminalen und nichtterminalen Symbolen nimmt yacc anhand der %token-Klausel vor. Alle Symbole, die in den Regeln vorkommen und nicht durch %token definiert wurden, werden als Nichtterminale aufgefaßt. Zeichenkonstanten werden dabei ebenfalls als Tokens aufgefaßt.

Da die reine Spezifikation von Grammatiken wenig Sinn macht, müssen zu diesen Regeln Aktionen angegeben werden können, die die Verarbeitung der erkannten syntaktischen Struktur übernehmen. Ist beispielsweise ein programmiersprachliches Konstrukt wie die Implementierung einer Routine erkannt worden, muß angegeben werden, wie der entsprechende Code generiert werden soll.

In yacc kann für diese Aufgabe zu jedem Symbol eine Aktion angegeben werden. Aktionen werden in geschweifte Klammern gesetzt, um sie von Symbolen zu unterscheiden. Die Anweisungen in den Aktionen sind im wesentlichen als gültiger C (++) abzufassen.

Damit ergibt sich prinzipiell folgender Aufbau:

```
Nichtterminal: Symbol10 {Aktion10} Symbol11 {Aktion11} ...
             | Symbol20 {Aktion20} Symbol21 {Aktion21} ...
             ...
             ;
```

$Symbol_i$ und $Aktion_i$ sind jeweils optional. Es ist also durchaus möglich eine Aktion ohne vorangestelltes Symbol und auch ein Symbol ohne nachgestellte Aktion anzugeben.

In den Aktionen ist auch nötig, auf die Werte von Symbolen - also auf die Attribute von Tokens oder von typsierten Nichtterminale zuzugreifen. Hierzu existieren die Pseudovariablen $x, wobei x ein ganzzahliger Wert größer oder gleich eins ist. $1 bezeichnet also das erste Symbol in der rechten Seite einer Regel, $2 das zweite und so fort. $$ ist eine besondere Pseudovariable und beschreibt den Rückgabewert des Nichtterminals der linken Seite einer Produktion. Wird dieses $$ nicht explizit in einer Aktion auf einen Wert gesetzt, so gilt $$ = $1.

Für Tokens zu denen ein Wert gehört wird $x durch die entsprechende Elementvariable von `yylval` ersetzt. Bei typisierten Nichtterminalen (%type) ist dies der Rückgabewert und wird ebenfalls durch die jeweilige Komponente von `yylval` dargestellt.

Ein wichtiger Aspekt ist jedoch bei den Pseudovariablen $x zu beachten: Aktionen zählen mit! In der Regel

```
A: B { setSym($1); } C { setSym($3); } ;
```

gelten folgende Entsprechungen:

```
$1 = B
$2 = { setSym($1); }
$3 = C
$4 = { setSym($3); }
```

In der ersten Aktion wird also der Wert von `B` an die Routine `setSym()` übergeben und in der zweiten Aktion fungiert der Wert von `C` als Argument der Routine `setSym()`. Da keine explizite Zuweisung eines Rückgabewertes über $$ erfolgt, gibt `A` - sofern es typisiert ist - den Wert von `B` (=$1) zurück.

yacc generiert einen LR-Parser (genauer einen LALR-Parser). Das Lesen der Symbole erfolgt also von links nach rechts. Dies hat zur Folge, daß Sie nicht in einer Aktion auf folgende Symbole per Pseudovariablen zugreifen können. Die Aktion

```
A: B { $$ = $1 - $3; } C ;
```

scheitert, da $3 das Symbol C referenziert. Dieses Symbol ist jedoch noch nicht erkannt und folglich nicht in einem definierten Zustand. Korrekt müßte es lauten:

```
A: B C { $$ = $1 - $2; } ;
```

6.1.5 Fehlerbehandlung

yacc verfügt über eine rudimentäre Fehlerbehandlung. Tritt ein Fehler auf, so wird schlicht „syntax error“ oder „parse error“ ausgegeben und die Syntaxanalyse abgebrochen. Diese Art Fehler zu behandeln ist für die meisten Systeme unzureichend.

yacc bietet sehr leistungsfähige Elemente Syntaxfehler zu behandeln. Diese Möglichkeiten sollen hier kurz vorgestellt werden, wobei eine ausführliche Betrachtung bis ins letzte Detail im Zuge dieses Buches leider nicht möglich ist.

yyerror()

Die Funktion, die yacc zur Fehlermeldung aufruft, nennt sich `yyerror()`. Sie ist in alter „Kernighan & Richtie C"-Manier wie folgt definiert:

```
yyerror(s)
        char *s;
```

Der implizite Integer-Rückgabewert wird nicht ausgewertet - man erkennt eindeutig, daß yacc mittlerweile einige Jährchen alt ist.

In der Bibliotheksimplementierung von `yyerror()` wird der String `s` schlicht auf dem Standardausgabe- oder dem Standardfehlerkanal ausgeben. Sie können jedoch diese Routine überschreiben und selbstdefinierte Fehlermeldungen ausgeben.

Der formale Parameter `s` wird in eigenen Fehlermeldungen zumeist ignoriert, da dieser ohnehin nur zu allgemein gefaßte Meldungen von yacc enthält. Bei Syntaxfehlern ruft yacc schlicht `yyerror()` durch

```
yyerror("syntax error");
```

auf. Die Meldung, daß ein Syntax- oder Parse-Fehler vorliegt, ist schlicht nicht aussagekräftig. Daher ist `s` eben in der Praxis zu ignorieren.

yychar

Zum Aufbau von aussagekräftigen Fehlermeldungen in `yyerror()` ist es sinnvoll das Token zu kennen, welches den Fehler verursachte. `yychar` ist eine Integer-Variable, die den Wert des aktuellen Token enthält. Diese Variable ist geradezu prädestiniert um in `yyerror()` eingesetzt zu werden.

error

yacc bietet einen einfachen, aber nicht minder leistungsfähigen Mechanismus zur Erkennung von Fehlern an. Innerhalb des Regelteils können durch das Symbol `error` Fehlerproduktionen aufgebaut werden.

Beim Erkennen eines Syntaxfehlers schaltet der Parser in den Fehlerbehandlungsmodus. In diesem Modus werden alle fehlerhaften Symbole durch das Symbol `error` abgedeckt. Durch das geschickte Einsetzen dieses Symbols innerhalb der Grammatik können Fehler einfach und unproblematisch behandelt werden. Der Parser kann durch Anweisungen in Aktionen nach dem error-Symbol in einen definierten Zustand rückgeführt werden.

Prinzipiell sieht dies so aus:

```
A: B
 | error { Anweisungen zur Fehlerbehandlung }
 ;
```

yyerrok

Nach der Behandlung eines Fehlers und der Rückführung in einen definierten Zustand muß der Parser angewiesen werden den Fehlerbehandlungsmodus zu verlassen. Dazu existiert in yacc `yyerrok`.

Durch die Angabe von

```
yyerrok;
```

in einer Aktion nach dem error-Symbol kann der Fehlerbehandlungsmodus verlassen werden.

Es ist jedoch unbedingt zu beachten, daß mit dieser Anweisung nicht leichtfertig umgegangen werden sollte, da der Parser sich wirklich wieder in einem Zustand befinden muß, an dem die Analyse fortgesetzt werden kann. Es bietet sich daher an, dem error-Symbol ein synchronisierendes Token folgen zu lassen. Durch dieses Token gelangt der Parser dann automatisch an einen Punkt, an dem die syntaktische Analyse wieder ansetzen kann.

Somit ergibt sich folgendes Schema, wobei hier das Semikolon das synchronisierende Symbol ist:

```
A: B ';'
 | error ';' { Anweisungen zur Fehlerbehandlung; yyerrok; }
 ;
```

Die korrekte Plazierung von `error` und `yyerrok` innerhalb der Grammatik ist konform zu [6] Erfahrungssache. Dennoch können folgende Regeln aufgestellt werden:

1. `error` dort einfügen, wo Rekursionen vorkommen.
2. `yyerrok` erst nach den Synchronisationssymbolen angeben, also am Ende der Produktion als letzte Aktion.
3. `yyerrok` nur an Stellen verwenden, an denen die Syntaxanalyse wieder erneut ansetzen kann.

6.2 Ein Parser in yacc

Im folgenden werden wir nun die Parser der drei Ausbaustufen, wie in Abschnitt 5.2 schon erwähnt, in yacc implementieren.

6.2.1 Stufe 1

Die erste Stufe soll die gleiche Sprache wie der Parser aus Kapitel 4 erkennen und die Anweisungen ebenfalls über die Stack-Maschine ausführen.

Der Definitionsteil

Der Defintionsteil präsentiert sich unspektakulär. Der meiste belegte Platz bezieht sich auf C++-Code:

```
%{
#include <iostream.h>
#include <stdlib.h>
#include "symtable.hpp"
#include "symbols.hpp"
#include "errhand.hpp"
#include "machine.hpp"

ErrorHandler errHand;
StackMachine machine(&errHand);
SymbolTable symTable;

void code_inst(Instruction::Type type);
void code_var(SYMHANDLE sh);
void code_const(double val);

int yylex();
int yyerror(char *s);
%}

%union {
       double val;
       SYMHANDLE sym;
}

%token <val> NUM
%token <sym> ID
%token WRITE READ

%right '='
%left '+' '-'
%left '*' '/'
%left UNARYMINUS UNARYPLUS
```

```
%right POWER
```

Die drei Objekte `errHand`, `machine` und `symTable` haben die gleiche Bedeutung wie in Kapitel 4, repräsentieren also die Fehlerbehandlung, die Stack-Maschine und die Symboltabelle.

Die Prototypen code_xxx() deklarieren Routinen, die in den Aktionen die Code-Erzeugung erleichern werden. Die Aufgaben sind:

- code_inst(type): Codiert eine Instruktion vom Typ type.
- code_var(sh): Codiert die Instruktion Instruction::itVar mit dem Symbolhandle sh als Attribut. (Instruction::itVar = „Variable auf den Stapel legen“)
- code_const(val): Codiert die Instruktion Instruction::itConst mit dem Fließkommawert val als Attribut. (Instruction::itConst = „Konstante auf den Stapel legen“)

Die %union-Anweisung definiert den Typ von `yylval`, der „Attribut-Variable“. Vorgesehen sind Fließkommazahlen (`val`) und Verweise auf die Symboltabelle (`sym`). Als Tokens existieren `NUM` mit dem Attribut `val` und `ID` mit `sym`, sowie die Schlüsselwörter `WRITE` und `READ`.

Nun folgen die Operatoren. Die Priorität und die Assoziativität bedarf keiner ausführlichen Betrachtung mehr. Lediglich die linksassoziativen Operatoren `UNARYMINUS` und `UNARYPLUS` sind scheinbar neu. Keiner der Scanner aus Abschnitt 5.2 gibt diese Operatoren zurück, doch wie sollen sie dann berücksichtigt werden? - Diese Operatoren sind die Vorzeichen - und +. Da sie sich lexikalisch nicht von den Subtraktions- und Additionsoperatoren unterscheiden, müssen sie anders behandelt werden. Die Einführung dieser speziellen Tokens - `UNARYMINUS` und `UNARYPLUS` - wird in der Syntaxanalyse bei der Unterscheidung von Subtraktions- und Additionsoperator erfolgen. Die ausführliche Betrachtung sei deshalb auf den folgenden Abschnitt verschoben.

Der Regelteil

Da die Festlegung der Operatorpriorität und -assoziativität schon im Defintionsteil erfolgt ist, muß dies nicht mehr in der Grammatik durch eine bestimmte Produktionsfolge berücksichtigt werden. Die Operatoren können scheinbar gleichberechtigt in den Produktionen eines Nichtterminals eingetragen werden.

Damit genügen ganze vier Nichtterminale gegenüber sechs in Kapitel 4:

```
entry: /* empty */
       | entry ';'
       | entry write ';'
```

```
        | entry read ';'
        | entry expr ';'
        | entry error ';
        ;

write: WRITE expr
       ;

read:  READ ID
       ;

expr:    NUM
       | ID
       | ID '=' expr
       | '(' expr ')'
       | expr '+' expr
       | expr '-' expr
       | expr '*' expr
       | expr '/' expr
       | expr POWER expr
       | '+' expr %prec UNARYPLUS
       | '-' expr %prec UNARYMINUS
       ;
```

Sie sehen, daß die Fehlerbehandlung mittels `error` in `entry` mit dem Semikolon als synchronisierendes Symbol realisiert wurde. Dies entspricht im wesentlichen dem panischen Fortsetzen aus Kapitel 4. Die komfortable Fehlerbehandlung aus Kapitel 4 werden wir hier jedoch noch nicht implementieren, sondern damit bis zur zweiten Stufe warten.

`entry` ist in Ermangelung einer %start-Anweisung im Defintionsteil implizit als Startsymbol deklariert.

Die Regeln sind im Grunde leicht nachzuvollziehen. Die einzigen Stellen, die einer gesonderten Erläuterung bedürfen sind die Vorzeichenproduktion in `expr`.

%prec ist ein Schlüsselwort, das der Produktion die Priorität und Assoziativität des Token nach dem %prec verleiht. %prec ist am Ende einer Regel, also entweder vor der letzten Aktion oder vor dem Regelabschluß (; oder |) anzugeben.

Dem Token '+' ('-') wird also, wenn es als Vorzeichen gebraucht wird, die Priorität und Assoziativität von `UNARYPLUS` (`UNARYMINUS`) zugewiesen.

Bislang findet sich in der yacc-Defintion des Parsers lediglich der grammatikalische Aspekt, die Aktionen fehlen. Es müssen jetzt noch diese Aktionen eingefügt werden. Die Aktionen bestehen bei diesem Parser aus der Code-Erzeugung für die Stack-Maschine.

Für jedes Symbol sind, nach dessen erkennen, die entsprechenden Instruktionen zu erzeugen. Damit ergibt sich folgendes Bild für den Regelteil:

```
entry: /* empty */
       | entry ';'
       | entry write ';' { code_inst(Instruction::itStop);
         return 1; }
       | entry read ';'  { code_inst(Instruction::itStop);
         return 1; }
       | entry expr ';'  { code_inst(Instruction::itStop);
         return 1; }
       | entry error ';' { yyerrok; }
       ;

write: WRITE expr { code_inst(Instruction::itWrite); }
       ;

read:  READ ID { code_var($2);
         code_inst(Instruction::itRead); }
       ;

expr:    NUM { code_const($1); }
       | ID { code_var($1);
         code_inst(Instruction::itEval); }
       | ID { code_var($1); } '=' expr {
         code_inst(Instruction::itAsgn); }
       | '(' expr ')'
       | expr '+' expr { code_inst(Instruction::itAdd); }
       | expr '-' expr { code_inst(Instruction::itSub); }
       | expr '*' expr { code_inst(Instruction::itMul); }
       | expr '/' expr { code_inst(Instruction::itDiv); }
       | expr POWER expr {code_inst(Instruction::itPower);}
       | '+' expr %prec UNARYPLUS
       | '-' expr %prec UNARYMINUS {
         code_inst(Instruction::itNeg); }
       ;
```

Die Code-Generierung erfolgt analog zu Kapitel 4. Um den Regelteil jedoch übersichtlich zu halten, wurden die Routinen `code_xxxx()` eingeführt.

Nach dem Erkennen einer kompletten Anweisung in `entry` wird die Stop-Instruktion erzeugt und in die Stapelmaschine eingetragen. Allerdings findet sich hier auch die Anweisung

```
return 1;
```

Diese Anweisung bewirkt, daß die Parse-Routine `yyparse()` beendet wird und der entsprechende Integer-Wert an den aufrufenden Prozeß zurückgeliefert wird.

Sie kennen die Arbeitsweise des Interpreters bereits aus Kapitel 4. Nach jedem Erkennen einer kompletten Anweisung wird die Syntaxanalyse beendet und die Stack-Maschine wird zur Ausführung des generierten Codes veranlaßt. Danach erfolgt die Syntaxanalyse der nächsten Anweisung und danach wieder die Ausführung des erzeugten Codes. Dies wird solange fortgesetzt bis das Dateiende erreicht wurde.

Im Falle des Dateiendes wird `yyparse()` automatisch beendet und 0 zurückgeliefert. Beim erfolgreichen Erkennen einer Anweisung beenden wir `yyparse()` jedoch mit der Rücklieferung des Wertes 1. Damit haben wir einen einfachen, aber hervorragenden Indikator für generierten Code oder nicht bzw. Dateiende oder nicht.

Diese Aufgaben Übernahmen im vierten Kapitel das Flag `term` und die zusätzliche Abfrage des Dateiendes der Eingabedatei.

Der Routinenteil

Der Routinenteil enthält vier Unterprogramme und das Hauptprogramm. Die ersten drei Routinen sind die code_xxxx()-Prozeduren:

```
void code_inst(Instruction::Type type)
{
       Instruction inst;
       inst.type = type;
       machine.code(inst);
}

void code_var(SYMHANDLE sh)
{
       Instruction inst;
       inst.type = Instruction::itVar;
       Symbol *sym = symTable.getSymbol(sh);
       if(!sym) {
              sym = new Variable;
              symTable.setSymbol(sh, sym);
       }
       inst.sym = sym;
       machine.code(inst);
}

void code_const(double val)
{
```

```
        Instruction inst;
        inst.type = Instruction::itConst;
        inst.val = val;
        machine.code(inst);
}
```

Diese Routinen sind kein kompliziertes Hexenwerk. `code_inst()` legt die spezifizierte Instruktion in den Programmspeicher der Stack-Maschine. `code_var()` baut die Instruktion für „Variable auf den Stapel legen“ auf. Hierbei ist wieder das Symbol ggf. entsprechend anzupassen. `code_const()` erzeugt die Instruktion für „Konstante auf den Stapel legen“ und übergibt sie an die Stack-Maschine.

Das nächste Unterprogramm implementiert eine rudimentäre Fehlermeldungsroutine:

```
int yyerror(char *s)
{
        cerr << s << endl;
}
```

Das Hauptprogramm besteht lediglich aus einer Schleife, die jeweils die Syntaxanalyse mittels `yyparse()` startet und den Code durch `machine.execute()` ausführt:

```
int main()
{
        for(;yyparse();machine.reset())
                machine.execute(machine.getProg());
        return 0;
}
```

Die Klassen und Module der Fehlerbehandlung, Symbole, Symboltabelle und Stack-Maschine werden unverändert aus Kapitel 4 übernommen.

6.2.2 Stufe 2

Stufe 2 der „arithmetischen Programmiersprache“ wird gegenüber der ersten Stufe zwei Änderungen aufweisen:

a)Erweiterung um Konstrollstrukturen in Form von If-Abfragen und While-Schleifen

b)Verbesserte Fehlerbehandlung

Hierzu ist nicht nur eine Modifikation des Parser notwendig, sondern auch der Stack-Maschine.

Die Kontrollstrukturen - Änderung der Stack-Maschine

Die Implementierung der Konstrollstrukturen bedarf eines tieferen Eingriffs in die Stack-Maschine. Es müssen neue Instruktionen für If- und While-Konstrukte eingebaut werden. Die Implementierung dieser neuen Konstrukte ist jedoch nicht so simpel wie die der bereits vorhandenen. Bei If muß der Programmlauf verzweigt werden, bei While ein Code-Teil - im Grunde ein Programm für sich - mehrfach durchlaufen werden.

Beim If-Konstrukt müssen drei Code-Parts berücksichtigt werden:

1. die Bedinung,
2. der Then-Teil und
3. der Else-Teil.

Die Bedinung muß von der Stack-Maschine ausgewertet werden und je nach Ergebnis zum Then- oder Else-Teil verzweigt werden. Nachdem der Then- oder Else-Teil ausgewertet wurde, muß danach zum Code nach der If-Anweisung übergegangen werden.

Die neu einzuführende Instruktion `Instruction::itIf` muß somit drei Positionen im Programmspeicher speichern:

- Position des Then-Parts.
- Position des Else-Parts und
- Position der Anweisung nach dem If-Konstrukt.

Abbildung 6.1 zeigt das Code-Modell des If-Konstrukts. Für die Speicherung der drei Positionen im Programmspeicher werden Stop-Instruktionen zweckentfremdet. Das Symbolattriubt `sym` vom Typ `SYMHANDLE` wird schlicht als Zeiger auf Positionen im Programmspeicher zweckentfremdet. Damit wird zwar die Typsicherheit von C++ unterlaufen, aber da sich die Instruktion als Instruction::itStop ausweist, kann selbst bei fehlerhafter Programmierung kein Schaden entstehen. Da das If-Konstrukt „weiß“, daß es sich hierbei nicht um Verweise auf die Symboltabelle (`=SYMHANDLE`), sondern um Zeiger innerhalb des Programmspeicher handelt, ergeben sich hier keine Probleme.

Die Instruktionanweisung, die von der Stack-Maschine ausgelesen wird, ist `Instruction::itIf`. Daraufhin werden die folgenden Anweisungen im Programmspeicher von der Stapelmaschine als Zeiger auf Then- und Else-Part, sowie nächste Anweisung interpretiert.

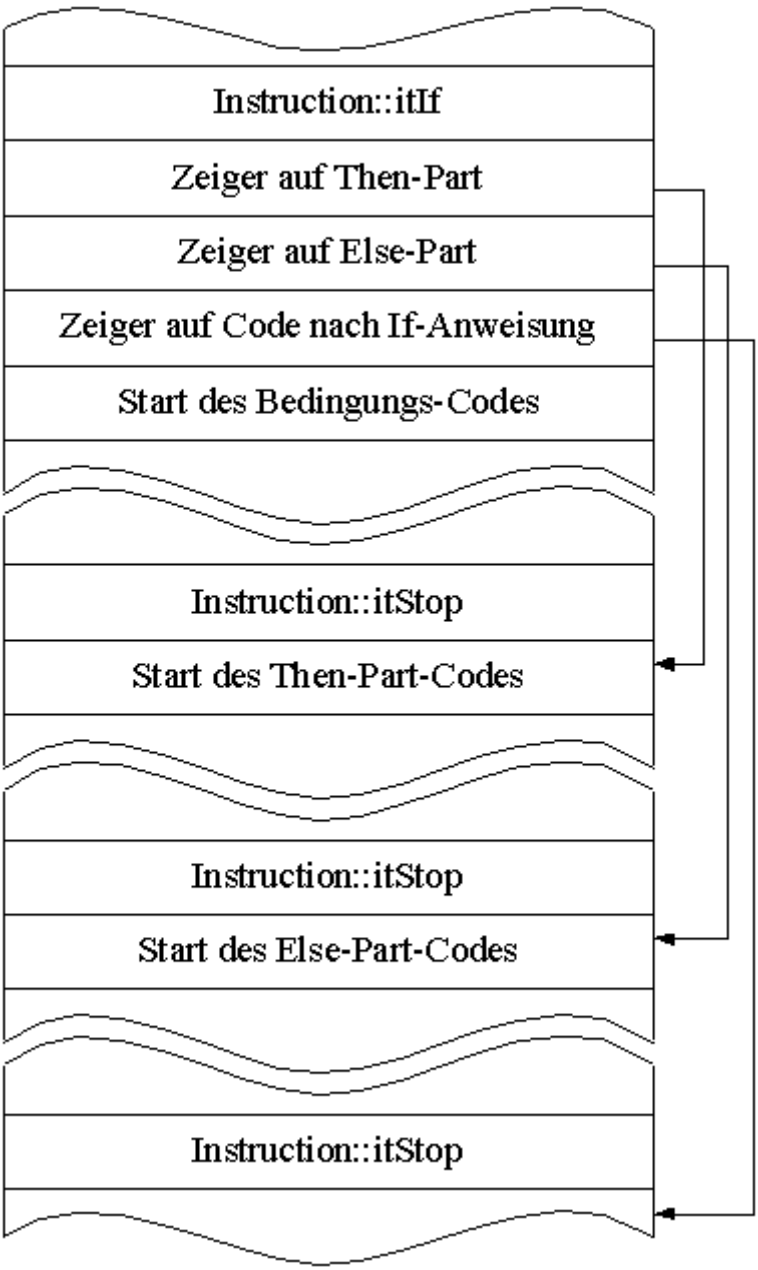


Abb. 6.1: Code-Modell des If-Konstrukts

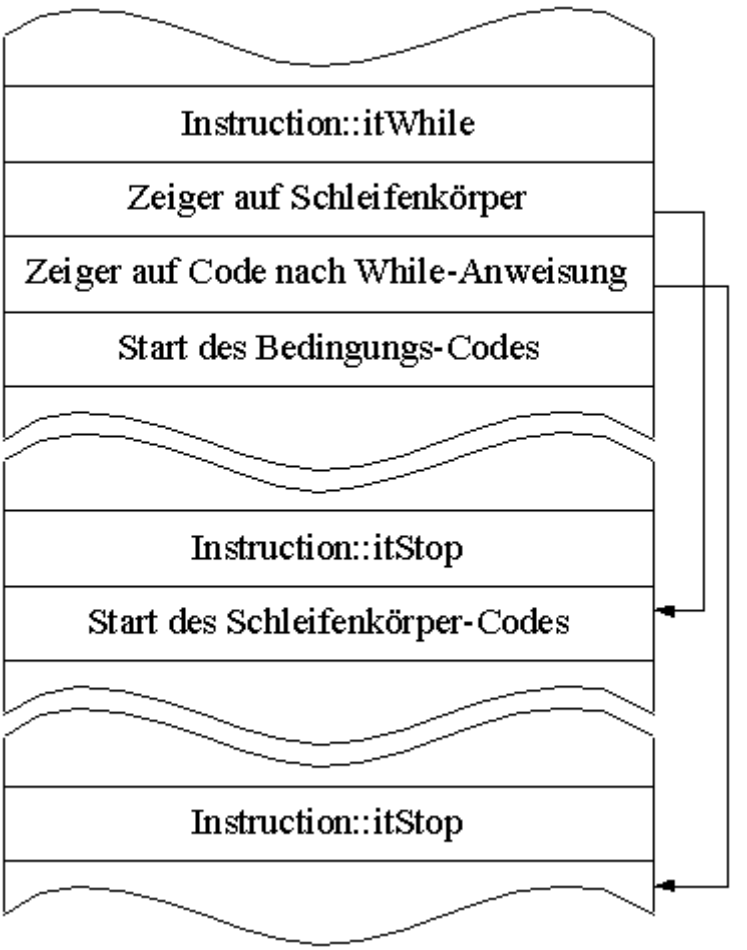


Abb. 6.2: Code-Modell des While-Konstrukts

Die auf den „nächste Anweisung“-Zeiger folgende Instruktion ist im übrigen der Beginn des Bedingungscodes. Dieser Code wird von der Stack-Maschine ausgeführt und liefert auf dem Stapel den Wert der Bedinung zurück. Je nachdem ob dieser Wert Null ist oder nicht, wird dann der Then- oder Else-Teil ausgeführt. Danach wird der „program counter“ (`pc`) auf den neuen Wert, nämlich den „nächste Anweisung“-Zeiger gesetzt, womit der If-Code abgearbeiet wäre.

Das Code-Modell des While-Konstrukts präsentiert sich ähnlich (Abbildung 6.2). Es benötigt jedoch nur zwei Zeiger, die als zweckentfremdete Stop-Instruktionen dargestellt werden. Diese Zeiger sind:

- Zeiger auf Schleifenkörper und
- Zeiger auf die dem While-Konstrukt folgende Anweisung.

Die Stack-Maschine führt bei Erkennen der Instruktion `Instruction::itWhile` den Bedinungscode aus und startet den Code des Schleifenkörpers, falls die Bedinung zutrifft. Danach wird wieder die Bedingung ausgeführt und so fort, bis die Bedinung nicht mehr zutrifft. In diesem Fall wird der „program counter“ (`pc`) auf den „nächste Anweisung“-Zeiger gesetzt.

Die Änderungen für die Instruktionen `Instruction::itIf` und `Instruction::itWhile` in `StackMachine::execute()` sind vergleichsweise gering:

```
case Instruction::itWhile:
       c1.val = execute(pc+3);
       while(c1.val) {
              execute((Instruction*)((pc+1)->sym));
              c1.val = execute(pc+3);
       }
       pc = (Instruction*)((pc+2)->sym);
       break;
case Instruction::itIf:
       c1.val = execute(pc+4);
       if(c1.val)
              execute((Instruction*)((pc+1)->sym));
       else if((pc+2)->sym)
              execute((Instruction*)((pc+2)->sym));
       pc = (Instruction*)((pc+3)->sym);
       break;
```

In das Case-Konstrukt sind lediglich die beiden obigen neuen Fallunterscheidungen einzutragen. Wie Sie sehen verwendet diese Implementierung nun rekursive Aufrufe von `execute()`. Für die - jeweils mit Stop-Instruktionen - abgeschlossenen Code-Parts wie Bedingung, Then-, Else- und Schleifenkörper-Teil werden jeweils zur Auswertung respektive Verarbeitung die Ausführungsmethode `execute()` neu instanziiert. Somit sind diese Teile sozusagen „Programme im Programm“. Sie sehen, daß dies einiges im Algorithmus vereinfacht.

Hinweis: Das optionale Else wird durch einen Nullzeiger signaliert, wenn es entfällt.

Diese Verarbeitung von „Programmen im Programm“ zieht allerdings die Notwendigkeit nachsich die Methode `isEmpty()` zu erweitern. Um entscheiden zu können, ob das Ende des Stapels für den aktuellen Prozeß von `execute()` erreicht ist genügt es nicht mehr einfach zu überprüfen, ob `stackptr == &stack[0]` ist. Der Stapelanfang ist schließlich von execute()-Call zu execute()-Call verschieden, da vom aufrufenden Prozeß bereits Teile des Stapels belegt sein können und sich für den Folgeprozeß der (relative) Stapelanfang nach oben verschiebt.

Am Ende von `execute()` hat demnach die Anweisung

```
return (isEmpty() ? 0.0 : (pop()).val);
```

dem Statement

```
return (isEmpty(stackptr_backup) ? 0.0 : (pop()).val);
```

zu weichen, wobei `stackptr_backup` als `Operand*` deklariert ist und bei Eintritt in `execute()` der aktuelle Inhalt des Stapelzeigers `stackptr` in dieser Variable gespeichert wird.

Die Methode `isEmpty()` in "machine.hpp" muß ebenfalls noch aktualisiert werden:

```
inline bool isEmpty(Operand *cmp) {
       return (stackptr <= cmp);
}
```

Damit die Bedingungen sinnvoll formuliert werden können, muß die Stackmaschine auch Vergleichs- und Verknüpfungsoperatoren verarbeiten können. Deshalb führen wir für diesen Zweck die folgenden neuen Instruktionen ein:

- `Instruction::itGt` für „größer als“
- `Instruction::itGe` für „größer oder gleich“
- `Instruction::itLt` für „kleiner als“
- `Instruction::itLe` für „kleiner oder gleich“
- `Instruction::itEq` für „gleich“
- `Instruction::itNe` für „nicht gleich“
- `Instruction::itAnd` für „und“
- `Instruction::itOr` für „oder“
- `Instruction::itNot` für „nicht“

Die Implementierung dieser Operatoren unterscheidet sich von den anderen bereits aus Kapitel 4 bekannten nicht sonderlich. Exeplarisch seien hier die folgenden gezeigt:

```
        case Instruction::itGt:
              c2 = pop();
              c1 = pop();
              c1.val = (c1.val > c2.val ? 1.0 : 0.0);
              push(c1);
              break;
...
        case Instruction::itAnd:
              c2 = pop();
              c1 = pop();
              c1.val = (c1.val != 0.0 && c2.val != 0.0 ?
                        1.0 : 0.0);
              push(c1);
              break;
...
        case Instruction::itNot:
              c1 = pop();
              c1.val = (c1.val == 0.0 ? 1.0 : 0.0);
              push(c1);
              break;
```

Die Definition der Konstanten für die neuen Instruktionen in `Instruction::Type` ist so trivial, daß wir uns dies hier die Betrachtung sparen wollen.

Erweiterung des Definitionsteils

Damit die neuen grammatikalischen Konstrukte und eine solide Fehlerbehandlung eingeführt werden können, sind auch Änderungen im Defintionsteil nötig. Hier das fragmentarische Listing:

```
%{
...
unsigned long lineno, colno;
...
%}

%union {
        double val;
        SYMHANDLE sym;
        Instruction *base;
}
```

```
%token <val> NUM
%token <sym> ID
%token IF THEN ELSE ENDIF WHILE DO WEND WRITE READ
%type <base> else preexpr prog stmt stmtlist stop

%right '='
%left OR
%left AND
%left GT GE LT LE EQ NE
%left '+' '-'
%left '*' '/'
%left UNARYMINUS UNARYPLUS NOT
%right POWER
```

Zum Speichern der Zeilen- und Spaltennummer sind nun im C++-Code-Teil die Variablen `lineno` und `colno` hinzugekommen.

Neben den neu aufgenommenen Schlüsselwort-Tokens und den Operatoren finden sich nun auch die Deklarationen von typisierten Nichtterminalen (%type). Diesen typisierten Nichtterminalen wurde das neue Attribut `base`, ein Zeiger auf eine Instruktion, zugeordnet. Die Bedeutung dieses Schrittes werden wir bei der Modifikation des Regelteils noch ausführlich behandeln.

Erweiterung des Regelteils

Die Eingriffe in die Grammatik sind nicht gerade einfacher Natur. Es müssen die If- und While-Konstrukte eingeführt werden und die neuen Operatoren eingebaut werden.

Beginnen wir mit der einfacheren Erweiterung - dem Einbau der Operatoren. Wie ihre „arithmetischen Kollegen“ sind die neuen Operatoren als Produktionen von `expr` realisiert. Auch ihre Implementierung erfolgt nach dem gleichen Schema:

```
expr:    NUM { code_const($1); }
       ...
       | expr GT expr { code_inst(Instruction::itGt); }
       | expr GE expr { code_inst(Instruction::itGe); }
       | expr LT expr { code_inst(Instruction::itLt); }
       | expr LE expr { code_inst(Instruction::itLe); }
       | expr EQ expr { code_inst(Instruction::itEq); }
       | expr NE expr { code_inst(Instruction::itNe); }
       | NOT expr { code_inst(Instruction::itNot); }
       | expr AND expr { code_inst(Instruction::itAnd); }
       | expr OR expr { code_inst(Instruction::itOr); }
       ;
```

Für die anderen Konstrukte ist erheblich mehr Aufwand nötig. Das Startsymbol entry reduziert seinen Produktionsumfang. Alle entfallenden Produktionen werden zu

```
entry: stmt ;
```

zusammengelegt:

```
entry: /* empty */
       | entry ';'
       | entry stmt ';'  { code_inst(Instruction::itStop);
         return 1; }
       | entry error ';' { yyerrok; }
       ;
```

Das Nichtterminal stmt nimmt arithmetische Ausdrücke, write- und read-Anweisungen, sowie die Kontrollstrukturen auf. Um jedoch den Code für die Stack-Maschine korrekt erzeugen zu können sind einige „Hilfsproduktionen“ nötig:

```
prog:  /* empty */ { $$ = machine.getProgPtr(); }
       ;

preexpr: prog expr
       | prog write
       | prog read
       ;

stmt:  preexpr
       | WHILE { code_inst(Instruction::itWhile);
       } stop stop expr DO stop stmtlist WEND stop {
             $3->sym = (Symbol*)$8;
             $4->sym = (Symbol*)$10;
       }
       | IF { code_inst(Instruction::itIf);
       } stop stop stop expr THEN stop stmtlist
         else ENDIF stop {
             $3->sym = (Symbol*)$9;
             $4->sym = (Symbol*)$10;
             $5->sym = (Symbol*)$12;
       }
       ;

else:  /* empty */ { $$ = (Instruction*)0; }
       | ELSE stop stmtlist { $$ = $3; }
       ;

stmtlist: prog stmtelements { $$ = $1; }
```

```
        ;

stmtelements:
        | stmt ';' stmtelements
        ;

stop:  /* empty */ { $$ = machine.getProgPtr();
          code_inst(Instruction::itStop); }
        ;
```

Das Nichtterminale `prog` verwendet eine neue Methode der Stack-Machine, um den aktuellen Inhalt des progptr-Zeiger auszulesen. Die Methode wird sinnvoll als inline in der Deklaration von `StackMachine` in "machine.hpp" definiert:

```
inline Instruction *getProgPtr() { return progptr; }
```

`prog` ist - wie alle im obigen Listing-Ausschnitt gezeigten Nichtterminal außer `stmtelements` - ein typisiertes Nichtterminal, daß Zeiger auf Instruktionen zurückgegeben kann. `prog` wird als Hilfe verwendet, wenn der Eintrittspunkt für generierten Code festgehalten werden muß.

Eine solche Stelle folgt bereits in der nächsten Produktion `preexpr`. Bevor dort in die Nichtterminale `expr`, `write` und `read` verzweigt wird, wird der aktuelle progptr ausgelesen und als implizit als Rückgabewert ($$=$1) festgelegt. Damit wird der Eintrittspunkt im Programmspeicher für die folgenden Konstrukte `expr`, `write` und `read` ermittelt. Diese Information ist unerläßlich für If- und While-Anweisungen.

Das Nichtterminal `stop` ist ebenfalls eine solche Hilfe. Es erzeugt eine Stop-Instruktion und liefert einen Zeiger auf diese Instruktion zurück. Dies ist wichtig, um für If- und While-Konstrukte „Platzhalter“ für die Zeiger auf die verschiedenen Parts (Then, Else, Schleifenkörper) zu schaffen.

`stmtlist` ist ein Nichtterminal, das eine Folge von Statements (`stmt`) beschreibt. Als Rückgabe wird der Eintrittspunkt in diesen Code-Block geliefert. Diese Liste von Anweisungen ist wieder für While- und If-Konstrukte nötig, da in ihnen ganze Code-Blöcke angegeben werden können.

Betrachten wir nun die While-Produktion in `stmt`. Nach dem Erkennen des Schlüsselworts `while` bzw. dessen Token `WHILE` wird in der folgenden Aktion die While-Instruktion im Programmspeicher angelegt. Die folgenden zwei Symbole vom Typ `stop` erzeugen die Platzhalter für die Zeiger auf den Schleifenkörper und auf das dem While folgenden Statement.

Danach wird die Bedingung mittels `expr` aufgebaut. Die Bedingung wird vom Schleifenkörper durch das Schlüsselwort `do` bzw. dessen Token `DO` getrennt. Das nun folgende `stop` erzeugt die Stop-Instruktion zum Abschluß der Bedingung. Der Rückgabewert ist im folgenden nicht mehr relevant.

Das nun folgende `stmtlist` analysiert den Anweisungsblock syntaktisch und erzeugt den entsprechenden Code im Programmspeicher. Der Rückgabewert von `stmtlist` ist der Eintrittspunkt in den Code-Block.

Das Schlüsselwort `wend` respektive das Token `WEND` schließt den Schleifenkörper syntaktisch ab. Das anschließende stop erzeugt die Stop-Instruktion für den Schleifenkörper und schließt ihn somit im Programmspeicher ab.

Die Schlußaktion trägt nun die „platzhaltenden stop-Symbole“ die Positionen des Schleifenkörpers ($8) und des auf das While folgende Statement ($10) ein.

Der Aufbau der If-Produktion ist ähnlich. Da das Else optional ist, wird ein Nullwert für den Zeiger auf den Else-Part eingetragen, wenn kein Else-Block angegeben wurde.

Die Fehlerbehandlung

Für die Behandlung von Syntaxfehlern wollen wir in Stufe 2 auch wieder das Fehlerbehandlungsobjekt `errHand` verwenden. `errHand` verfügt über die Methode `error()` zur Meldung eines Fehlers. Diese Methode wollen wir nun in `yyerror()` einsetzen.

Um aussagekräftige Meldungen an `errHand.error()` übergeben zu können müssen wir die Ursache des Fehlers ergründen. Hier liegt es nahe die Variable `yychar` zu verwenden, die das Token enthält, welches den Fehler auslöste:

```
int yyerror(char *s)
{
 char *msg;
 switch(yychar) {
   case NUM:
       msg = "Fliesskommazahl hier nicht erlaubt."; break;
   case ID: msg = "Bezeichner hier nicht erlaubt."; break;
   case IF: msg = "if hier nicht erlaubt."; break;
   case THEN: msg = "then ohne if."; break;
   case ELSE: msg = "else ohne if."; break;
   case ENDIF: msg = "endif ohne if."; break;
   case WHILE: msg = "while hier nicht erlaubt."; break;
   case DO: msg = "do ohne while."; break;
   case WEND: msg = "wend ohne while."; break;
   case WRITE: msg = "write hier nicht erlaubt."; break;
   case READ: msg = "read hier nicht erlaubt."; break;
   case OR:
   case AND:
   case GT:
       ... hier folgen die übrigen Operatoren ...
```

```
      case '*':
          msg = "Operator ohne gueltige Operanden."; break;
      case ')': msg = ") nicht erwartet."; break;
      case '(': msg = "( hier nicht erlaubt."; break;
      case ';': msg = "Unvollstaendige Anweisung."; break;
      default: msg = "Ungueltiges Zeichen."; break;
    }
    errHand.error(lineno, colno, ErrorHandler::etError, msg);
    return 0;
  }
```

Nachdem die Meldung aufgebaut ist, wird sie zusammen mit der Zeilen- und Spaltennummer an das Fehlerbehandlungobjekt übergeben. Weitere Aktionen sind nicht nötig, da die restliche Fehlerbehandlung und das Error-Recovery in der Grammatik durch das Symbol `error` implementiert sind.

Da es bei der Ausführung des generierten Codes durch die Stack-Maschine zu fatalen Fehler kommen kann und als Folge daraus die Exception `xFatal` geworfen wird, muß dies das Interpretersystem ebenfalls berücksichtigen. Außerdem müssen `lineno` und `colno` initialisiert werden.

Damit ergibt sich das folgende modifizierte Hauptprogramm:

```
int main()
{
       lineno = 1;
       colno = 0;
       try {
              for(;yyparse();machine.reset())
                     machine.execute(machine.getProg());
       }
       catch(xFatal){}
       return 0;
}
```

6.2.3 Stufe 3

Die dritte Stufe zeichnet sich gegenüber Stufe 2 dadurch aus, daß sie auch Unterprogramme zuläßt. Damit sind wieder Änderungen an der Stack-Maschine und dem Parser nötig.

Objekt für Unterprogramme

Unterprogramme werden über Bezeichner identifiziert. Bezeichner werden in der Symboltabelle verwaltet und damit auch Unterprogramme. Es muß also ähnlich wie bei

Variablen eine Objektklasse entworfen werden, durch die Unterprogramme in der Symboltaballe organisiert werden können. Diese Klasse wird, wie `Variable`, eine Kindklasse von `Symbol` sein.

Das Unterprogramm wird in einem Programmspeicher wie dem der Stack-Maschine gespeichert. Neben der typischen Methode `getType()` wird darüberhinaus eine Auskunftmethode `getProg()` existieren, die den Programmspeicher `prog` zurückliefert.

```
class Procedure : public Symbol
{
        public:
                Procedure();
                ~Procedure();

                inline Instruction *getProg() {
                        return &prog[2]; }
                int getType();

        protected:
                Instruction *prog;
                friend StackMachine;
};
```

Um den Zugriff auf diese neue Klasse von der Stack-Maschine aus nicht unnötig zu erschweren, wurde `StackMaschine` als Friend von `Procedure` deklariert. Weshalb `getProg()` nicht `prog`, sondern `&prog[2]` zurückliefert und weswegen `prog` dynamisch verwaltet wird, wird im folgenden Abschnitt erläutert.

Die Methode getType(), der Konstruktor und der Destruktor sind sehr einfach implementiert:

```
Procedure::Procedure()
{
        prog = 0;
}

Procedure::~Procedure()
{
        if(prog)
                delete[] prog;
}

int Procedure::getType()
{
```

```
        return type_proc;
    }
```

Unterprogramme - Änderung der Stack-Maschine

Zur Implementierung der Unterprogrammtechnik in die Stack-Maschine sind zwei neue Intruktionen hinzuzufügen. Zum einen die Instruktion „Unterprogramm anlegen“ (`Instruction::itProc`) und „Unterprogramm ausführen“ (`Instruction::itCall`).

Die Call-Instruktion ist ohne größere Änderungen implementierbar. Die Unterprogramme werden wie die Code-Blöcke der Kontrollstrukturen als „Programme im Programm“ aufgefaßt, d. h. `StackMachine::execute()` wird rekursiv aufgerufen. Als Argument wird an execute() dabei der Programmspeicher aus dem Symboltabelleneintrag übergeben. Zuvor ist jedoch zu testen, ob dieser Programmspeicher gültig, d. h. über ein solches in dem betreffenden Symbol angelegt wurde:

```
    case Instruction::itCall:
        if(!((Procedure*)pc->sym)->getProg())
            errHand->error(0, 0, ErrorHandler::etFatal,
                "Prozedur nicht implementiert.");
        execute(((Procedure*)pc->sym)->getProg());
        break;
```

`Instruction::itProc` ist ebenfalls einfach zu implementieren, wenn man den Aufbau dieser Instruktion beachtet. Abbildung 6.3 zeigt den Aufbau des Programmspeichers für diese Instruktion. Die erste Anweisung ist der Befehl „Unterprogramm anlegen“ mit dem Attribut `sym`, das auf den Symboltabelleneintrag verweist. Das Unterprogramm liegt ab `prog[2]` komplett in ausführbarer Form bereits vor. Es würde also genügen diesen Programmspeicher an den Bezeichner in der Symboltabelle zu binden.

Der einfachste Weg ist, auch in der Klasse `StackMachine` den Programmspeicher dynamisch durch einen Zeiger zu verwalten. Den allokiierten Speicher könnte man durch einfaches Zuweisen mit dem Symboltabelleneintrag verknüpfen. Danach müßte für die Stack-Maschine lediglich ein neuer Programmspeicher angefordert werden.

Damit ergibt sich folgender Programmcode in `execute()` für die Instruktion `Instruktion::itProc`:

```
    case Instruction::itProc:
        if(((Procedure*)pc->sym)->prog)
            delete[] ((Procedure*)pc->sym)->prog;
        ((Procedure*)pc->sym)->prog = prog;
        prog = new Instruction[PROGSIZE];
```

Dieses Vorgehen Bedarf nun noch einer Änderung von prog in StackMachine von

```
Instruction proc[PROGSIZE];
```

zu

```
Instruction *proc;
```

Außerdem muß nun im Konstruktor der entsprechende Speicherplatz reserviert werden und ein Destruktor eingeführt werden:

```
StackMachine::StackMachine(ErrorHandler *eh)
{
       stackptr = stack;
       prog = new Instruction[PROGSIZE];
       progptr = prog;
       prog[0].type = Instruction::itStop;
       errHand = eh;
}

StackMachine::~StackMachine()
{
       delete[] prog;
}
```

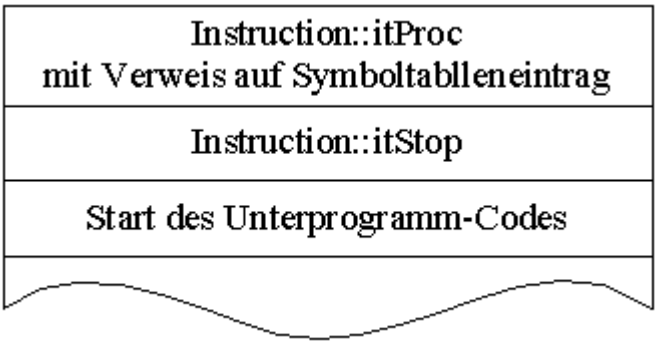


Abb. 6.3: Code-Modell für Instruktion „Unterprogramm anlegen“

Damit erklärt sich nun auch die Rückgabe von `&prog[2]` bei `Procedure::getProg()`. Bei `prog[2]` startet der Code. Die anderen beiden Instruktionen sind gemäß Abbildung 6.3 lediglich die Anweisungen, die das Anlegen des Unterprogramms bewirken. Diese Instruktionen dürfen beim Ausführen des Unterprogramms nicht mehr berücksichtigt werden.

Erweiterung des Definitionsteils

Der Erweiterung des Defintionsteils beschränkt sich für die Unterprogrammtechnik lediglich auf die Angabe der neuen Tokens `PROC`, `IS` und `END` in der %token-Zeile. Außerdem sind im C++-Code-Block die Prototypen von zwei neuen Routinen anzugeben:

```
void code_proc(SYMHANDLE sh);
void code_call(SYMHANDLE sh);
```

Diese Routinen erzeugen die Proc- und Call-Instruktionen.

Weitere Änderungen sind nicht erforderlich.

Erweiterung des Regelteils

Die neuen Produktionen sind ebenfalls nicht sehr kompliziert. Lediglich entry und stmt sind zu erweitern und ein neues Nichtterminal routine ist anzulegen.

```
entry: /* empty */
       | entry ';'
       | entry stmt ';'  {
            code_inst(Instruction::itStop); return 1; }
       | entry routine ';' {
            code_inst(Instruction::itStop); return 1; }
       | entry error ';' { yyerrok; }
       ;

routine: PROC ID { code_proc($2); } stop IS stmtlist
         END stop
       ;

...

stmt:  preexpr
       | CALL '(' ID ')' { code_call($3); }
       ...
       ;
```

Die Code-Generierung wird durch die Methoden `code_proc()` und `code_call()` veranlaßt, die beide einen Verweis auf einen Eintrag in der Symboltabelle als Argument erwarten.

Erweiterung des Routinenteils und Fehlerbehandlung

Die semantischen Aspekte sind in diesem Interpretersystem gewachsen. Das Token `ID` kann sowohl eine Variable, als auch ein Unterprogramm sein. Hier kann es zu semantischen Fehlern kommen, nämlich dann wenn ein Bezeichner eines Unterprogramms als Variable oder ein Bezeichner einer Variablen als Unterprogramm verwendet wird.

Daher müssen wir nun auch die semantischen Fehler berücksichtigen. Hierfür führen wir die Exception-Klasse `xSemantic` ein, die immer dann geworfen wird, wenn ein semantischer Fehler auftritt. Hier genügt wiederum eine rudimentäre Deklaration zu Beginn des Routinenteils:

```
class xSemantic {};
```

Die Routinen `code_proc()` und `code_call()` sind neu zu implementieren. Da im Unterprogramm `code_var()` ebenfalls Bezeichner verarbeitet werden, muß dieses modifiziert werden, um den semantischen Aspekt berücksichtigen zu können.

Der Code:

```
void code_call(SYMHANDLE sh)
{
       Instruction inst;
       inst.type = Instruction::itCall;
       Symbol *sym = symTable.getSymbol(sh);
       if(!sym) {
              sym = new Procedure;
              symTable.setSymbol(sh, sym);
       } else if(sym->getType() != Symbol::type_proc) {
              errHand.error(lineno, colno,
  ErrorHandler::etError, "Bezeichner ist keine Prozedur.");
              throw xSemantic();
       }
       inst.sym = sym;
       machine.code(inst);
}

void code_proc(SYMHANDLE sh)
{
       Instruction inst;
       inst.type = Instruction::itProc;
       Symbol *sym = symTable.getSymbol(sh);
       if(!sym) {
              sym = new Procedure;
              symTable.setSymbol(sh, sym);
```

```
        } else if(sym->getType() != Symbol::type_proc) {
              errHand.error(lineno, colno,
  ErrorHandler::etError, "Bezeichner ist keine Prozedur.");
              throw xSemantic();
        }
        inst.sym = sym;
        machine.code(inst);
}

void code_var(SYMHANDLE sh)
{
        Instruction inst;
        inst.type = Instruction::itVar;
        Symbol *sym = symTable.getSymbol(sh);
        if(!sym) {
              sym = new Variable;
              symTable.setSymbol(sh, sym);
        } else if(sym->getType() != Symbol::type_var) {
              errHand.error(lineno, colno,
  ErrorHandler::etError, "Bezeichner ist keine Variable.");
              throw xSemantic();
        }
        inst.sym = sym;
        machine.code(inst);
}
```

Darüberhinaus ist noch `yyerror()` für die vier neuen Schlüsselwörter zu modifizieren:

```
int yyerror(char *s)
{
 char *msg;
 switch(yychar) {
        case CALL: msg = "call hier nicht erlaubt."; break;
        case PROC: msg =
"proc-Deklaration nicht innerhalb von Anweisungsbloecken.";
              break;
        case END: msg = "end ohne proc."; break;
        case IS: msg = "is ohne proc."; break;
 ...
}
```

Damit auch die Exception `xSemantic` gefangen werden kann, muß das Hauptprogramm ebenfalls überarbeitet werden:

```
int main()
```

```
{
	lineno = 0;
	colno = 0;
	for(;;) {
		try {
			if(!yyparse())
				break;
			machine.execute(machine.getProg());
			machine.reset();
		}
		catch(xSemantic) {}
		catch(xFatal) {
			break;
		}
	}
	return 0;
}
```

Anhang

Anhang A: Literaturverzeichnis

[1] The Free Online Dictionary of Computing.
http://wagner.princeton.edu/foldoc/about.html

[2] Aho, Alfred V. / Sethi, Ravi / Ullman, Jeffrey D.: Compilerbau.
Bonn, München, Reading (Mass.): Addison Wesley 1988

[3] ANSI X3J16 / ISO WG21: Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++.
Doc No: X3J16/96-0225 December 2nd 1996

[4] Bachmann, Peter: Mathematische Grundlagen der Informatik.
Berlin: Akademie Verlag 1992

[5] Binstock, Andrew / Rex, John: Practical Algorithms for Programmers.
Reading (Mass.): Addison Wesley 1995

[6] Herold, Helmut: lex und yacc - Lexikalische und syntaktische Analyse.
Bonn, München, Reading (Mass.): Addison Wesley 2. Auflage 1995

[7] Hopcroft, John E. / Ullman, Jeffrey D.: Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.
Bonn, München, Reading (Mass.): Addison Wesley 3. Auflage 1996

[8] Kernighan, Brian W. / Pike, Robert: Der UNIX-Werkzeugkasten - Programmieren mit UNIX.
München: Carl Hanser Verlag 1986

[9] Leckebusch, Johannes: DOS-Informatik - Parsertechniken.
DOS International 12/93; Seiten 216 ff

[10] Müller, Oliver: Entwicklungshilfe - Lexikalische und syntaktische Analyse mit lex und yacc.
mc extra (DOS Interational) 12/95; Seiten 32 ff

[11] Müller, Oliver: command.com im Eigenbau - Entwicklung eines Kommandointerpreters.
mc extra (DOS International) 4/96; Seiten 31 ff

[12]Müller, Oliver: Mustererkennung in Delphi - Suchen und Ersetzen von regulären Ausdrücken.
mc extra (DOS International) 7/96; Seiten 38 ff

[13]Nentwig, Dr. Dietmar / Becker, Christian: 1 x 1 des Compilerbaus - Aufbau und Funktionsweise von Compilern.
mc extra (DOS Internationa) 8/95; Seiten 34 ff

[14]Nickles, Michael / Uhlmann, Kerrin: DOS-Informatik - Die Parser-Programmierung (1).
DOS International 8/91; Seiten 230 ff

[15]Nickles, Michael / Uhlmann, Kerrin: DOS-Informatik - Die Parser-Programmierung (2).
DOS International 9/91; Seiten 274 ff

[16]Penner, Volker: Konzepte und Praxis des Compilerbaus.
Braunschweig, Wiesbaden: Vieweg 1994

[17]Schulze, Hans Herbert: Computerenzyklopädie - Lexikon und Fachwörterbuch für Datenverarbeitung und Telekommunikation.
Reinbeck bei Hamburg: Rowohlt Taschenbuchverlag Februar 1990

[18]Sedgewick, Robert: Algorithmen.
Bonn, München, Reading (Mass.): Addison Wesley 1991

[19]Sippu, Seppo / Soisalon-Soininen, Eljas: Parsing Theory - Volume I: Languages and Parsing.
Berlin, Heidelberg, New York: Springer 1988

[20]Sippu, Seppo / Soisalon-Soininen, Eljas: Parsing Theory - Volume II: LR(k) and LL(k) Parsing.
Berlin, Heidelberg, New York: Springer 1990

[21]Wirth, Niklaus: Grundlagen und Techniken des Compilerbaus.
Bonn, München, Reading (Mass.): Addison Wesley 1995

Anhang B: ansi.hpp

Die folgende Include-Datei ermöglicht es auch C++-Compilern ohne Unterstützung des aktuellen Draft-Papers von ANSI/ISO ([3]) zu nutzen. Die einzelnen Einträge sind dem C++-Compiler ggf. anzupassen.

```
#ifndef __ANSI_HPP_OGM
#define __ANSI_HPP_OGM

// Schlüsselwort inline wird nicht unterstützt:
#define inline

// Typ bool wird nicht unterstützt:
typedef short bool;
#define false 0
#define true !false

#endif
```

Anhang C: Aufbau der CD-ROM

Verzeichnis \LEXYACC

Dieses Verzeichnis enthält die Quelltexte und Executables der GNU-Tools flex und bison, die Alternativen zu lex und yacc darstellen. Soweit nicht anders angegeben gelten die Bestimmungen der „GNU General Public Licence“, die sich in der Datei COPYING findet.

Verzechnis \LEXYACC\BIN

Hier finden sich die Executables von flex und bison für folgende Betriebsysteme:

- Amiga: Leider nur bison.
- Atari: bison und flex.
- DOS: bison und flex.
- Linux: bison und flex (ELF-Versionen).
- OS/2: Leider nur flex.
- Win32: bison und flex in einer vom Autor dieses Buches durchgeführten Quick-and-Dirty-Portierung für Windows 9x/NT/2000.

Verzeichnis \LEXYACC\SOURCE

Dieses Verzeichnis enthält die Quelltexte zu flex und bison als TAR-Archiv.

Verzeichnis \LEXYACC\TAR

Hier findet sich eine Portierung von TAR auf OS/2 und DOS inklusive Quelltext.

Verzeichnis \SOURCE

Im Verzeichnis \SOURCE und dessen Unterverzeichnissen finden sich alle Quelltexte des Buches:

- CALC1: Parser aus den Abschnitten 2.7.2 und 2.7.3
- CALC2: Parser aus Abschnitt 2.7.4
- CALC3: Parser aus Abschnitt 2.8
- CALC4: Scanner aus Abschnitt 3.11
- CALC5: Scanner aus den Abschnitten 3.11.1 und 3.11.2
- CALC6: Scanner aus Abschnitt 3.12.4
- CALC7: Interpreter aus Abschnitt 4.7
- CALC8: Scanner aus 5.2.1 und Parser aus 6.2.1
- CALC9: Scanner aus 5.2.2 und Parser aus 6.2.2
- CALC10: Scanner aus 5.2.3 und Parsr aus 6.2.3
- ANSI.HPP: Headerdatei ansi.hpp aus Anhang B
- ECTYPE.HPP: Headerdatei ectype.hpp aus Anhang D
- DOS2UNIX: Da die Quelltexte im DOS-Format gespeichert sind, findet sich hier ein DOS-nach-UNIX-Konverter in lex.

Anhang D: ectype.h

Falls die Bibliotheken Ihres Compilers die Makros von <ctype.h> nicht dem englischen Sprachgebrauch entsprechend definieren, d. h. auch Umlaute akzeptieren, können die folgenden Makros verwenden.

```
#ifndef __ECTYPE_H_OGM
#define __ECTYPE_H_OGM

#define isalpha(c) ((c>='A' && c<='Z')||(c>='a' && c<='z'))
#define isdigit(c) (c>='0' && c<='9')
#define isalnum(c) (isalpha(c)||isdigit(c))
#define isspace(c) (c=='\n' || c=='\r' || c=='\v' \
  || c=='\h' || c == ' ')

#endif
```