Prof. Dr. Ralf Hartmut Güting, Prof. Dr. Martin Erwig

Kurs 01810

Übersetzerbau

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere das Recht der Vervielfältigung und Verbreitung sowie der Übersetzung und des Nachdrucks bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder ein anderes Verfahren) ohne schriftliche Genehmigung der FernUniversität reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Vorwort

Liebe Fernstudentin, lieber Fernstudent,

wir freuen uns, daß Sie am Kurs „Übersetzerbau“ teilnehmen und wünschen Ihnen viel Spaß und Erfolg beim Studium des Kurses und der Bearbeitung der Übungsaufgaben. Dieser Kurs bietet eine kompakte Einführung in die Grundlagen und Techniken des Übersetzerbaus. Übersetzer transformieren Texte einer Quellsprache, deren Struktur durch eine formale Grammatik beschrieben ist, in eine Zielsprache.

Die klassische Anwendung ist die Übersetzung höherer Programmiersprachen in Maschinensprache und damit die Implementierung solcher Sprachen. Dies ist ein grundlegendes Problem der Informatik, das sehr früh intensiv studiert wurde. Deshalb gibt es seit langem eine gut verstandene Theorie zu formalen Sprachen und Maschinen, die sie akzeptieren; die Techniken der Syntaxanalyse und ihre Verbindung mit Übersetzungsaktionen sind ausgefeilt, und es gibt eine klare Zerlegung der Gesamtaufgabe der Übersetzung in Teilaufgaben, die die Grundlage für eine entsprechende Modularisierung von Übersetzern bilden. Das Verständnis der Implementierung von Programmiersprachen gehört zweifellos zur „Allgemeinbildung“ eines Informatikers, ebenso wie man Grundlagen der Hardware oder Betriebssysteme verstehen sollte.

Nun gibt es nur ein paar weitverbreitete höhere Programmiersprachen, und nur wenige Informatiker werden heute an der Entwicklung von Compilern für solche Sprachen, wie z.B. C++ oder Java, mitarbeiten. Ist die Beschäftigung mit dem Thema daher etwa nur von akademischem Interesse bzw. kann sie solchen Spezialisten vorbehalten bleiben? Keineswegs. Ein wichtiges Ziel dieses Kurses besteht darin, den Blick zu schärfen für die vielseitige Verwendbarkeit von Techniken des Übersetzerbaus zur Realisierung von Komponenten von Software-Systemen. In vielen Anwendungen gibt es Beschreibungssprachen für spezielle Zwecke, wie etwa Dokumentstrukturen (LaTeX, HTML), Anfragesprachen für Datenbanksysteme, Protokolle in verteilten Systemen, vielerlei Formate für den Datenaustausch über das Internet (Browser-Plugins) usw. Ganz allgemein kann man mit Techniken des Übersetzerbaus Strukturen in Texten, Dateien oder Byte-Strömen identifizieren. Oft ist es nützlich, eine „kleine“ Sprache zu definieren und sie dann mit Hilfe von Übersetzerbau-Werkzeugen mit wenig Aufwand zu implementieren.

Dieser Kurs setzt deshalb folgende Akzente:

- Die Methoden des Übersetzerbaus, etwa für die lexikalische Analyse, Syntaxanalyse, syntaxgesteuerte Übersetzung oder Code-Optimierung werden angemessen behandelt.
- Die vielseitige Anwendbarkeit dieser Techniken wird betont. So wird z.B. bewußt als erste Anwendung die Übersetzung einer Dokument-Beschreibungssprache nach LATEX vorgeführt. Dies ist gleichzeitig ein Beispiel für eine vollständige Übersetzerimplementierung, d.h., der komplette dokumentierte Quellcode für diesen Compiler ist im Kurs wiedergegeben.
- Der Einsatz von Werkzeugen, nämlich von Scanner- und Parsergeneratoren, wird anhand von Lex und Yacc erklärt und gründlich eingeübt.
- Das klassische Thema der Implementierung imperativer Programmiersprachen wird detailliert behandelt. Auf der Basis einer abstrakten Maschine für 3-Adreß-Code wird die Übersetzung einer PASCAL-ähnlichen Sprache durch Angabe von Übersetzungsschemata präzise beschrieben.
- Schließlich zeigen wir die Implementierung funktionaler Programmiersprachen durch Interpretation und Übersetzung, um neben imperativen noch eine weitere Klasse von Programmiersprachen zu behandeln und so den Blick zu erweitern.

Der Kurs ist wie folgt aufgebaut: Nach einem einführenden Kapitel werden in den Kapiteln 2 bis 4 lexikalische Analyse, Syntaxanalyse und syntaxgesteuerte Übersetzung (auf der Basis attributierter Grammatiken) behandelt. Es folgen Anwendungen, nämlich die Übersetzung einer Dokument-Beschreibungssprache sowie die Übersetzung imperativer und schließlich funktionaler Programmiersprachen in den Kapiteln 5 bis 7. Das letzte Kapitel ist der Code-Optimierung gewidmet. Im Anhang findet sich ein vollständiger Compiler zu Kapitel 5.

Dieser Kurs ist auch als Buch im Springer-Verlag erschienen:

Güting, R.H., und M. Erwig, Übersetzerbau: Techniken, Werkzeuge, Anwendungen. Springer-Verlag, Berlin Heidelberg 1999. 398 S., ISBN 3-540-65389-9.

Die Autoren

Prof. Dr. Ralf Hartmut Güting, geb. 1955. Studium der Informatik an der Universität Dortmund. 1980 Diplom. 1981/82 einjähriger Aufenthalt an der McMaster University, Hamilton, Kanada, Forschung über algorithmische Geometrie. 1983 Promotion über algorithmische Geometrie an der Universität Dortmund. 1985 einjähriger Aufenthalt am IBM Almaden Research Center, San Jose, USA, Forschung im Bereich Büroinformationssysteme, Nicht-Standard-Datenbanken. Ab 1984 Hochschulassistent, ab 1987 Professor an der Universität Dortmund. Seit November 1989 Professor für Praktische Informatik (Lehrgebiet „Datenbanksysteme für neue Anwendungen“) an der FernUniversität. Forschungsgebiete: erweiterbare/konfigurierbare Datenbanksysteme, Geo-Datenbanksysteme, Graphen in Datenbanken, raum-zeitliche Datenbanken.

Prof. Dr. Martin Erwig, geb. 1963. Studium der Informatik an der Universität Dortmund. Diplom 1989. Seitdem wissenschaftlicher Mitarbeiter am Lehrgebiet Praktische Informatik IV an der Fernuniversität. Promotion 1994 zum Thema Graphen in Datenbanken. Von 1994 bis 2000 Hochschulassistent an der Fernuniversität. Seit September 2000 Associate Professor an der Oregon State University, Corvallis, OR, USA. Forschungsgebiete: Programmiersprachen, insbesondere funktionale Programmierung und visuelle Sprachen, sowie Geo-Datenbanken.

Weitere Informationen zu Personen, Forschungsgebieten, Veröffentlichungen, Lehrangeboten, Abschlußarbeiten usw. am Lehrgebiet „Datenbanksysteme für neue Anwendungen“ können Sie auch unseren WWW-Seiten entnehmen, URL:

`http://dna.fernuni-hagen.de/`

Voraussetzungen

Der Kurs setzt Grundkenntnisse der Informatik voraus. Von Bedeutung sind insbesondere Konzepte der Programmierung (etwa anhand der Kurse 01612, 01613 oder 01618) und von Datenstrukturen und Algorithmen (Kurs 01663 oder 01661). Grund begriffe der theoretischen Informatik zu formalen Sprachen und Maschinen (etwa aus Kurs 01654 oder 01658) sind nützlich, aber nicht unbedingt notwendig.

Zum Verständnis einiger Teile des Kurses brauchen Sie zumindest rudimentäre Kenntnisse der Programmiersprache C. Obwohl die Autoren des Kurses die Sprache C auch nicht gerade lieben, kann man ihr im Zusammenhang mit den Werkzeugen Lex und Yacc nicht aus dem Wege gehen, da diese C-Code erzeugen und man in Spezifikationen C-Programmstücke einbetten muß. Wenn Sie die Konzepte imperativer Sprachen gut verstehen und eine Sprache wie PASCAL oder Modula-2 beherrschen, ist es kein großes Problem, C-Programme zu verstehen (naja: je nachdem, wie schlimm sie programmiert sind) und einfache C-Programme selbst zu schreiben. In den Literaturhinweisen finden Sie ein C-Buch angegeben; andere Bücher tun es sicher auch.

Übungen

Wie immer ist die Bearbeitung der Selbsttest- und Einsendeaufgaben wesentlich, um ein gutes Verständnis des Kursinhalts zu erreichen. Wir ermutigen Sie sehr, in den Übungen kleine Teile von Übersetzern selbst zu schreiben, insbesondere mit den Werkzeugen Lex und Yacc (bzw. Flex und Bison unter Linux) zu arbeiten. Auf Ihrem PC zuhause können Sie LINUX installieren und so eine UNIX-Umgebung bekommen; Lex (Flex) und Yacc (Bison) sind Teil der Standardinstallation von LINUX. In jeder anderen UNIX-Umgebung stehen Lex und Yacc sowieso zur Verfügung, wie natürlich auch die Sprache C.

Nähere Informationen zum Ablauf der Übungen, Sprechstunden der Betreuer usw. finden Sie in einem gesonderten Anschreiben.

Literatur

Hinweise zu anderen Lehrbüchern und weiterführender Literatur finden sich in den Literaturhinweisen am Ende von Kapitel 1 und der weiteren Kapitel.

Inhalt

1 Einführung **Kurseinheit 1**

1.1 Anwendungsgebiete 1
1.2 Übersetzungsphasen 3
1.3 Die Systemumgebung des Compilers 11
1.4 Compiler und Interpreter, reale und abstrakte Maschinen 13
1.5 Werkzeuge 15
1.6 Struktur des Kurses 17
1.7 Literaturhinweise 17

2 Lexikalische Analyse 19

2.1 Beschreibung von Token durch reguläre Ausdrücke 20
2.2 Beschreibung von Token durch Zustandsdiagramme 23
2.3 Direkte Implementierung eines Scanners 28
2.4 Implementierung eines Scanners mit Lex 33
2.5 Literaturhinweise 37

---

3 Syntaxanalyse **Kurseinheit 2**

3.1 Kontextfreie Grammatiken und Syntaxbäume
3.2 Top-Down-Analyse

3.3 Bottom-Up-Analyse **Kurseinheit 3**

---

4 Syntax-gesteuerte Übersetzung **Kurseinheit 4**

5 Übersetzung einer Dokument-Beschreibungssprache

---

6 Übersetzung imperativer Programmiersprachen **Kurseinheit 5**

---

7 Übersetzung funktionaler Programmiersprachen **Kurseinheit 6**

---

8 Codeerzeugung und Optimierung **Kurseinheit 7**

Lehrziele

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie

- die Aufgaben und die grundsätzliche Arbeitsweise von Compilern und Interpretern erklären und die Vor- und Nachteile beider Techniken erläutern können;
- verstehen, daß Compilertechnik neben dem traditionellen Bereich der Übersetzung von Programmiersprachen auch andere Anwendungsgebiete hat;
- die verschiedenen Übersetzungsphasen mit ihren unterschiedlichen Aufgaben beschreiben können;
- die Systemumgebung eines Compilers und die Aufgaben ihrer einzelnen Komponenten erläutern können;
- anhand der bekannten Unix-Tools Lex und Yacc den Vorteil des Einsatzes von Werkzeugen (z.B. Scanner, Parser) für die einzelnen Übersetzungsphasen kennen;
- die lexikalische Analyse als erste Analysephase des Übersetzungsprozesses im Detail beschreiben können;
- Token durch reguläre Ausdrücke und durch Zustandsdiagramme darstellen können;
- für einfache Problemstellungen Lex-Spezifikationen zur Implementierung eines Scanners schreiben können.

Kapitel 1

Einführung

Thema dieses Kurses ist der Bau von Übersetzern für „formale" Sprachen, d.h. für Programmiersprachen im weitesten Sinne. Ein Übersetzer (engl. *Compiler*) erzeugt aus einem Quellprogramm in einer Sprache A ein Zielprogramm in einer Sprache B. Mindestens die Quellsprache A ist durch ein Regelsystem beschrieben, das der Übersetzer benutzt, um das Quellprogramm zu analysieren; so wird insbesondere die *Syntax* von A durch eine Grammatik definiert. Der Übersetzer ist damit auch in der Lage, Fehler im Quellprogramm zu erkennen, und die Ausgabe entsprechender Fehlermeldungen ist eine wichtige Teilaufgabe der Übersetzung. Die Aufgabe sieht also ganz grob so aus, wie in Abb. 1.1 gezeigt.

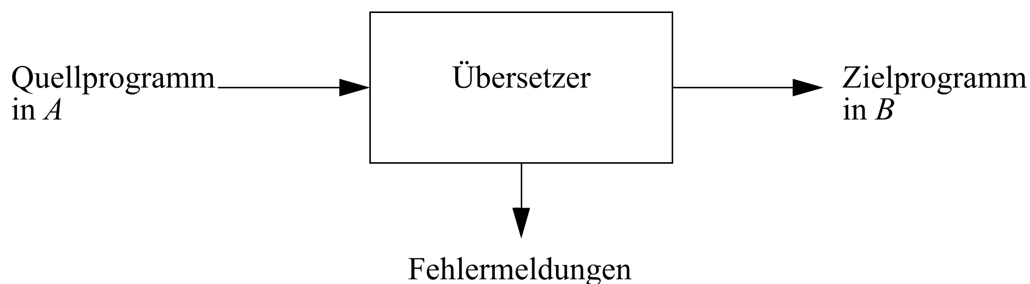


Abb. 1.1. Aufgabe eines Übersetzers

Warum möchte man überhaupt von A nach B übersetzen? Ganz allgemein deshalb, weil man aus irgendeinem Grund etwas besser in der Sprache A beschreiben kann als in B, man andererseits aber nur eine „Maschine" (einen Computer, ein Softwarepaket, …) zur Verfügung hat, die B versteht.

1.1 Anwendungsgebiete

Der klassische Anwendungsfall ist die Übersetzung einer höheren Programmiersprache (PASCAL, C, …) in die Maschinensprache eines Rechners. Dies ist sicherlich die wichtigste Anwendung, die auch die Entwicklung der Compilertechnik wie die der dahinterliegenden Theorie der formalen Sprachen von Anfang an motiviert

hat. Ein Ziel dieses Kurses besteht also darin, die Implementierung von Programmiersprachen zu erklären, die ja das grundlegende Werkzeug eines Informatikers darstellen. Ebenso, wie man verstehen sollte, wie Betriebssysteme oder Datenbanksysteme funktionieren, sollte man auch wissen, was im Innern eines Compilers vor sich geht und wie Programme letztendlich auf der Hardware ausführbar gemacht werden.

Neben diesem Ziel, gewissermaßen die Allgemeinbildung zu vervollständigen, gibt es aber auch sehr handfeste praktische Gründe, warum man sich mit Compilertechnik beschäftigen sollte. Zwar werden nur wenige Informatiker tatsächlich Compiler für C++, Eiffel oder Java schreiben. Es gibt aber viele Anwendungen, in denen Beschreibungssprachen für spezielle Zwecke gebraucht werden. Einige Beispiele:

- *Dokument-Beschreibungssprachen* wie LaTeX, SGML, HTML. Hier werden Textelemente wie Überschriften, Absätze, Aufzählungen ebenso wie Darstellungsattribute (*Kursivschrift*) oder Sonderzeichen in Textdateien beschrieben. Dieser Absatz etwa sieht in HTML so aus:

  ```
  <UL>
  <LI><EM>Dokument-Beschreibungssprachen</EM> wie LaTeX, SGML,
  HTML. Hier werden Textelemente wie &Uuml;berschriften,
  Abs&auml;tze, Aufz&auml;hlungen ebenso wie Darstellungsattri-
  bute (<EM>Kursivschrift</EM>) oder Sonderzeichen in Textdateien
  beschrieben. Dieser Absatz etwa sieht in HTML so aus:
  </UL>
  ```

  HTML ist die Sprache, in der Dokumente über das Internet (World Wide Web) verfügbar gemacht werden. Wer einen HTML-Browser implementieren will, braucht Compilertechnik, um die Struktur solcher Dokumente zu analysieren.

- *Datenbankanfragesprachen*. Hier ein Beispiel in *SQL*, mit dem aus einer Studententabelle die Namen und Adressen der Studenten ermittelt werden, die in Münster wohnen und im 10. oder höheren Semester Informatik studieren.

  ```
  select Name, Anschrift
  from Studenten
  where Stadt = "Muenster" and Fach = "Informatik" and
          Semester >= 10
  ```

  Auch hier werden Analysetechniken aus dem Compilerbau benutzt, um die Anfrage z.B. in einen Operatorbaum umzuwandeln, der dann vom Optimierer des Datenbanksystems weiterbehandelt wird.

- *VLSI-Entwurfssprachen*. Solche Sprachen beschreiben das Layout elektronischer Schaltungen auf einem Chip. Die Beschreibung umfaßt verschiedene Abstraktionsebenen, von komplexen Komponenten (Prozessor, Speicherbaustein, …) bis hin zu Basiselementen (Transistoren, …), die schließlich auf Rechtecke in den unterschiedlichen Materialschichten des Chips abgebildet werden.

- *Protokolle in verteilten Systemen*. Hier werden Zeichenströme über Rechnernetze ausgetauscht; die Struktur dieser Zeichenströme stellt ebenfalls eine spezielle Sprache dar, die auf der Empfängerseite analysiert werden muß.

Es ist schon wahrscheinlicher, daß man einmal vor der Aufgabe steht, eine solche spezielle Sprache zu implementieren. Grundsätzlich bieten einem die Analysetechniken des Compilerbaus und die zugehörigen Werkzeuge die Möglichkeit, Strukturen in Textdateien zu erkennen. Textdateien sind insbesondere dann von Bedeutung, wenn Daten zwischen verschiedenen Anwendungen ausgetauscht werden sollen, wenn Daten über Netze verschickt werden, wenn Daten zwischen Programmen in unterschiedlichen Programmiersprachen (mit inkompatiblen Laufzeitsystemen, etwa zwischen Miranda und C) übermittelt werden müssen. Wer Compilertechnik beherrscht, kann erkennen, daß es sich lohnt, hier eine kleine Sprache zunächst zu definieren und dann mit Hilfe vorhandener Standardwerkzeuge mit relativ wenig Aufwand zu implementieren. Damit ist es möglich, beliebig komplexe und flexible Datenstrukturen auf Texte abzubilden – das Herausschreiben ist zwar sowieso kein Problem; nun kann man aber auch die Struktur aus der Textdatei zurückgewinnen.

1.2 Übersetzungsphasen

Wir betrachten nun die innere Struktur eines Übersetzers. Die Gesamtaufgabe kann in eine Reihe von Teilaufgaben zerlegt werden, die wir *Phasen* nennen wollen. Jede Phase transformiert das zu übersetzende Programm von einer Darstellungsform in eine andere. Bei einem klassischen Compiler ist die Ausgangsform das Quellprogramm einer höheren Programmiersprache, die letzte erreichte Form das Zielprogramm in Maschinen- oder Assemblersprache. Bei den oben erwähnten anderen Anwendungen fallen ggf. einige der letzten Phasen weg. Abb. 1.2 stellt die Übersetzungsphasen im Überblick dar.

Die Phasen entsprechen logischen Schritten in der Übersetzung, was aber nicht heißt, daß sie unbedingt strikt nacheinander ablaufen müssen, etwa in 6 Durchgängen durch das Programm. Es ist durchaus möglich, sie zeitlich verzahnt ablaufen zu lassen, indem z.B. ein Übersetzermodul von seinem Vorgänger benötigte Zeichen oder Teilstrukturen anfordert. Parallel zu allen Phasen liegen die Aufgaben der *Verwaltung der Symboltabelle* und der *Behandlung von Fehlern*. Wir betrachten nun die einzelnen Phasen.

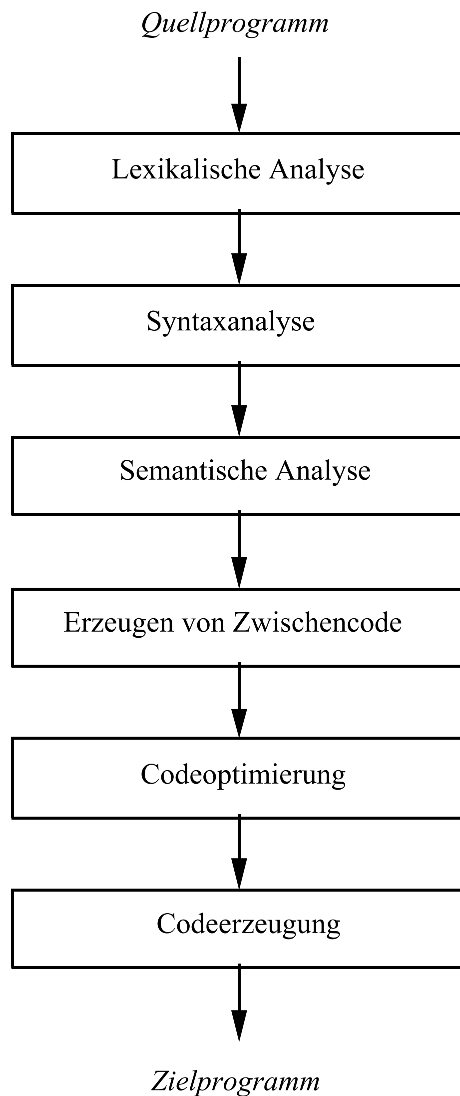


Abb. 1.2. Übersetzungsphasen

Lexikalische Analyse. Aus der Sicht der lexikalischen Analyse ist die Eingabe eine Folge von Zeichen (Buchstaben, Ziffern, Sonderzeichen). Das Ziel dieser Phase ist die Erkennung gewisser „Grundsymbole" in diesem Zeichenstrom. Für eine Programmiersprache sind dies etwa Wortsymbole wie *begin*, *if*, *while*, Variablennamen, numerische Konstanten usw.

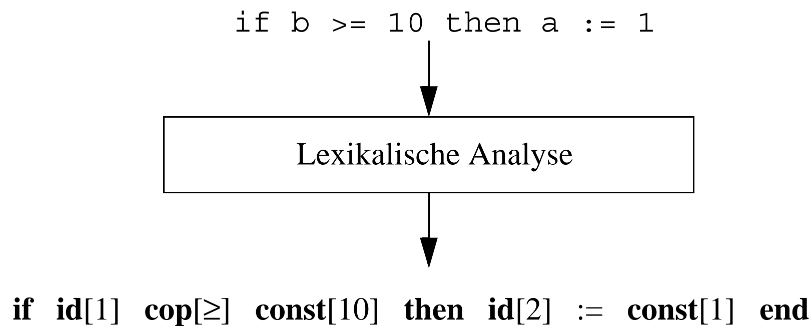


Abb. 1.3. Lexikalische Analyse: Berechnung der Tokenfolge zu einer Zeichenfolge

Abbildung 1.3 zeigt, wie eine bedingte Anweisung in eine Folge von Grundsymbolen, genannt *Token*, überführt wird. Die 22 Zeichen der Eingabe werden in folgende Token gruppiert:

1. Wortsymbol *if*.
2. Bezeichner *b*. Token können neben ihrem „Namen" zusätzliche Information (ein *Attribut*) zugeordnet bekommen. Wir notieren dies in eckigen Klammern hinter dem Token. Für einen Bezeichner (Token **id**) ist dies z.B. ein Index in eine Symboltabelle, in der man den Namen, später auch weitere Information wie den Typ, eine Speicheradresse usw. finden kann.
3. Vergleichsoperator „>=". Das Token **cop** beschreibt sämtliche derartigen Operatoren ($<, \leq, =, \neq, \ldots$); das Attribut gibt den konkreten Operator an.
4. Numerische Konstante 10. Das Attribut nimmt den Wert auf.
5. Wortsymbol *then*.
6. Bezeichner *a*.
7. Zuweisungssymbol „:=".
8. Numerische Konstante 1.
9. Wortsymbol *end*.

Die Struktur der Grundsymbole läßt sich mit relativ einfachen Mitteln beschreiben, nämlich mit sog. regulären Ausdrücken. Der Ausdruck

```
letter (letter | digit)*
```

könnte etwa die erlaubte Form von Bezeichnern in einer Programmiersprache definieren („ein Buchstabe, gefolgt von 0 oder mehr Buchstaben oder Ziffern"). – Ausgabe der lexikalischen Analyse ist also eine Folge von Token.

Syntaxanalyse. Aufgabe der Syntaxanalyse ist es, hierarchische Strukturen in Programmen (oder anderen Texten) zu erkennen. Solche Strukturen lassen sich nicht mehr mit regulären Ausdrücken beschreiben, wohl aber mit (kontextfreien) Grammatiken. Die Symbole einer Grammatik beschreiben größere Einheiten in Programmen wie arithmetische Ausdrücke, bedingte Anweisungen, Schleifen, Prozedurdeklarationen usw.

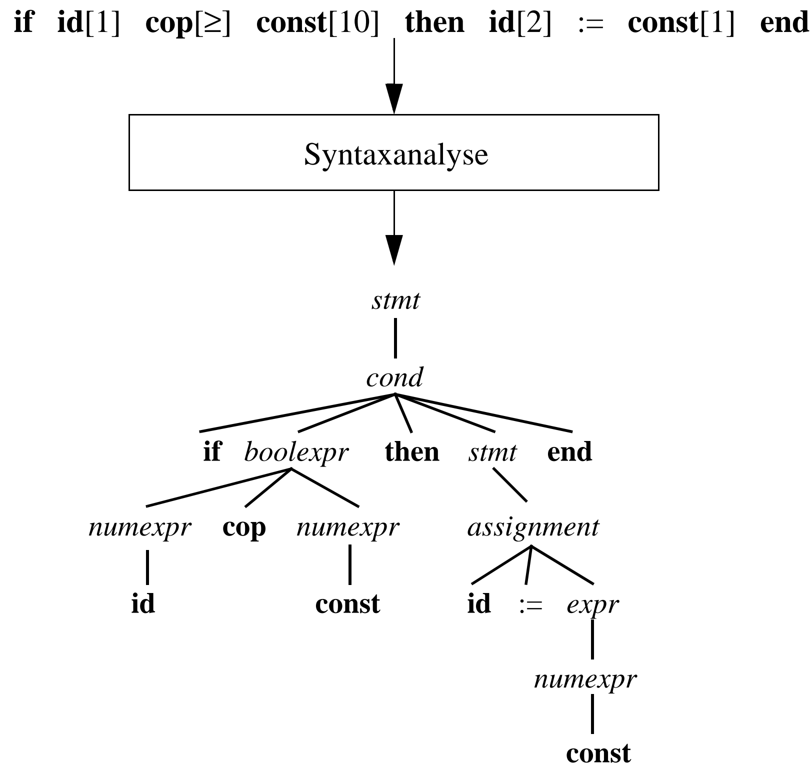


Abb. 1.4. Syntaxanalyse: Berechnung des Syntaxbaums zu einer Tokenfolge

Abbildung 1.4 zeigt den Baum, der die Struktur der bedingten Anweisung wiedergibt. Zwei Regeln der zugehörigen Grammatik sind etwa die folgenden:

stmt	::=	*assignment* | *cond*
cond	::=	**if** *boolexpr* **then** *stmt* **end** |
		if *boolexpr* **then** *stmt* **else** *stmt* **end**

Sie drücken aus, daß eine Anweisung (*stmt*) entweder eine Zuweisung (*assignment*) oder eine bedingte Anweisung (*cond*) sein kann und daß eine bedingte Anweisung jede der beiden gezeigten Formen annehmen kann.

Eingabe der Syntaxanalyse ist also eine Tokenfolge; Ausgabe ist ein Syntaxbaum.

Semantische Analyse. Nachdem die syntaktische Struktur des Programms bekannt ist, können in der Phase der semantischen Analyse weitere Informationen gesammelt werden, die in der Syntaxanalyse nicht erfaßte Korrektheitsaspekte betreffen oder für die folgende Codeerzeugung benötigt werden. Insbesondere gehört dazu die Typüberprüfung (*type checking*), bei der untersucht wird, ob Operationen auf passende Argumentausdrücke angewandt werden, wie auch das Auflösen überladener oder polymorpher Operationen. So ist z.B. die Addition überladen, und es muß ermittelt werden, ob eine ganzzahlige oder eine Gleitkommaaddition durchzuführen ist. Bei manchen Typinkonsistenzen muß der Compiler zusätzliche Operationen ein-

fügen (*type casting*); so kann z.B. beim Vergleich einer ganzen mit einer reellen Zahl die ganze Zahl zunächst in eine reelle umgewandelt werden, um dann zwei reelle Zahlen vergleichen zu können.

Für unsere Beispielanweisung

```
if b>=10 then a:=1 end
```

könnte in dieser Phase etwa festgestellt werden, daß die Variable *a* vom Typ *real* ist, während der Ausdruck auf der rechten Seite der Zuweisung

```
a:=1
```

vom Typ *integer* ist. Je nach Definition der Programmiersprache kann dies als Fehler aufgefaßt werden, oder die semantische Analyse könnte eine Typanpassungsoperation einfügen, die Zuweisung also umschreiben in

```
a:=makereal(1)
```

Die ersten drei Phasen waren der *Analyse* des Quellprogramms gewidmet, die damit abgeschlossen ist. Die folgenden drei Phasen dienen der *Synthese*, also dem Zusammensetzen, dem Konstruieren des Zielprogramms.

Erzeugen von Zwischencode. Insbesondere bei der Übersetzung höherer Programmiersprachen klafft eine gewaltige Lücke zwischen den dort vorhandenen Konzepten und den recht primitiven Möglichkeiten der Maschinensprache als Zielsprache. Um die Komplexität des Übersetzungsproblems beherrschbar zu machen, fügt man gern noch eine Zwischenebene ein, übersetzt also nicht direkt in die Maschinensprache, sondern in eine „abstrakte Maschinensprache" von etwas höherem Niveau.

In Abb. 1.5 wird unsere Beispielanweisung in ein Programm in sog. *3-Adreß-Code* überführt. Befehle im 3-Adreß-Code können bis zu drei Argumente (Adressen) enthalten. Es gibt z.B. folgende Befehlsformen:

$x := a$ op b
$x := a$
if a cop b goto L

Hier sind x, a, b und L die Argumente (die Adressen von Speicherzellen sein könnten). „op" ist Teil des „Befehlscodes" (ebenso „cop"); es gibt Versionen der Befehle für Addition, Subtraktion usw. (also op $\in \{+, -, \ldots\}$) bzw. für verschiedene Vergleichsoperatoren (cop $\in \{=, <, \leq, \ldots\}$). Darüber hinaus gibt es die Möglichkeit, Sprungmarken zu setzen wie in Assemblersprachen.

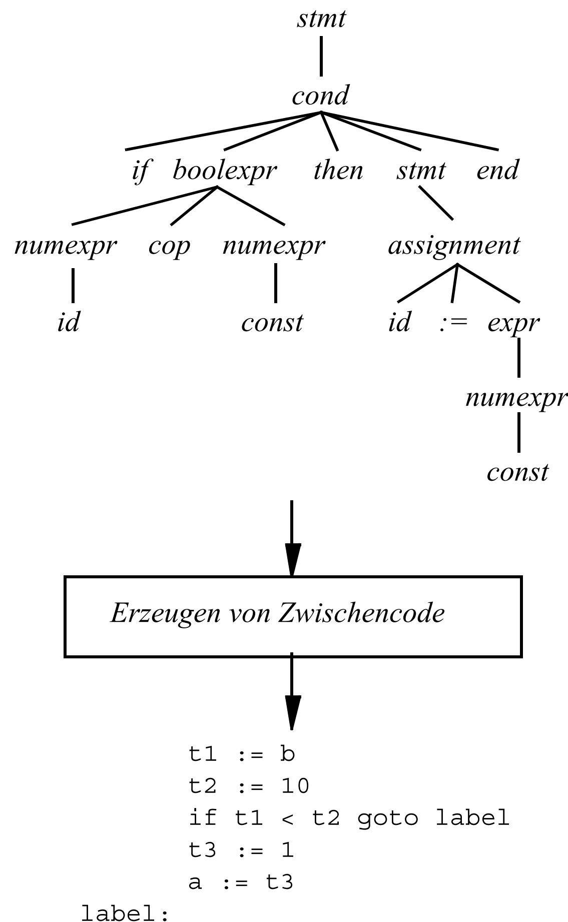


Abb. 1.5. Zwischencodeerzeugung

Das Erzeugen von Code in einer solchen Zwischensprache ist einfacher zunächst deshalb, weil das Sprachniveau etwas höher ist; insbesondere kann man geschachtelte arithmetische Ausdrücke sehr leicht übersetzen, da man jeden binären Operator direkt in eine Operation der Zwischensprache überführen kann. Zum Beispiel hat der Ausdruck

```
3 * a + (b - c) / (d - 5 * e)
```

die in der Syntaxanalyse erkannte Operatorbaumstruktur, die in Abb. 1.6 gezeigt ist. Durch Einsatz immer neuer temporärer Variablen läßt sich relativ mechanisch dafür der Zwischencode erzeugen:

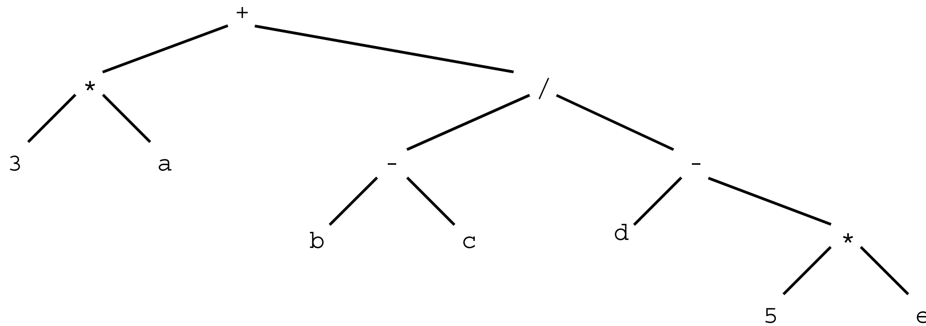


Abb. 1.6. Operatorbaum zum arithmetischen Ausdruck

```
t1 := 3 * a
t2 := b - c
t3 := 5 * e
t4 := d - t3
t5 := t2 / t4
t6 := t1 + t5
```

Das Erzeugen von Zwischencode ist andererseits einfacher, weil man hier noch vom speziellen Befehlssatz der Zielmaschine abstrahiert. Um guten Code für eine gegebene Zielmaschine zu erzeugen, muß man die dort vorhandenen Befehlsmöglichkeiten sehr genau kennen und gut ausnutzen; der vorhandene Befehlssatz zusammen mit Adressierungsmöglichkeiten ist aber oft sehr komplex. Es ist viel einfacher, diese Feinheiten bei der Übersetzung des Zwischencodes in die Maschinensprache zu berücksichtigen, also in der übernächsten Phase.

Codeoptimierung. Die relativ mechanische Art, mit der Zwischencode aus dem vorhandenen Syntaxbaum generiert wird, führt zu manchen Ungeschicklichkeiten, d.h. zu Ineffizienzen im erzeugten Zwischencode. Das Ziel der Codeoptimierungsphase besteht darin, solche Ineffizienzen aufzufinden und zu beseitigen.

Abbildung 1.7 zeigt, wie der Zwischencode vereinfacht werden kann. Wir nehmen an, daß die Übersetzungstechnik für die Übersetzung der Bedingung in einer bedingten Anweisung zunächst die zu vergleichenden Werte temporären Variablen zugewiesen hat. Das kann sinnvoll sein, weil dort ja jeweils auch komplexe Ausdrücke stehen könnten. In diesem Fall ist es allerdings ineffizient, weil man *b* und 10 auch direkt in einen 3-Adreß-Befehl aufnehmen kann. Ähnlich verhält es sich mit der folgenden Übersetzung einer Zuweisung; auch hier kann *t3* eliminiert werden.

In dieser Phase werden z.T. recht komplexe Analysen des erzeugten Zwischencodes durchgeführt. Dabei versucht man z.B., die Schleifenstruktur des Programms zu erkennen, um die in innersten Schleifen verwendeten Variablen in schnellen Registern zu halten.

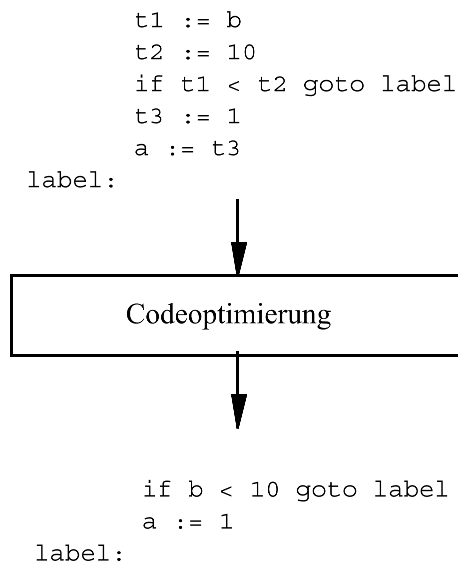


Abb. 1.7. Codeoptimierung: Verbessern des Zwischencodes

Codeerzeugung. In der letzten Phase wird aus dem optimierten Zwischencode Assembler- oder Maschinencode für die spezielle Zielmaschine generiert. Wichtige dabei zu lösende Probleme sind etwa die Speicherorganisation für das Zielprogramm, die Abbildung von Operationen des Zwischencodes auf die bestmöglichen Befehlsfolgen der Zielmaschine und die bereits erwähnte möglichst gute Zuteilung von Registern.

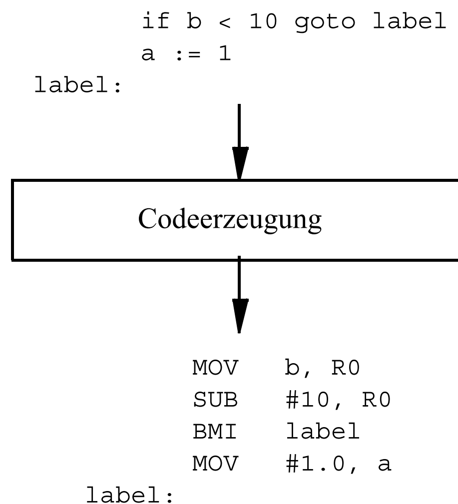


Abb. 1.8. Codeerzeugung: Generieren von Maschinencode

Abbildung 1.8 zeigt die Übersetzung unserer Beispielanweisung in eine hypothetische Maschinensprache (hier in Assembler dargestellt). Dabei bezeichnen *a* und *b* Speicheradressen, *Ri* das *i*-te Register, `#10` und `#1.0` innerhalb des Befehls direkt dargestellte Konstanten. `MOV` ist ein Transportbefehl, `SUB` die Subtraktion, und `BMI` steht für „branch on minus".

1.3 Die Systemumgebung des Compilers

Ein Übersetzer arbeitet mit anderen Programmen zusammen, um die zu übersetzende Sprache ausführbar zu machen. Für den Standardfall, die Übersetzung höherer Programmiersprachen, ist die Systemumgebung in Abb. 1.9 gezeigt.

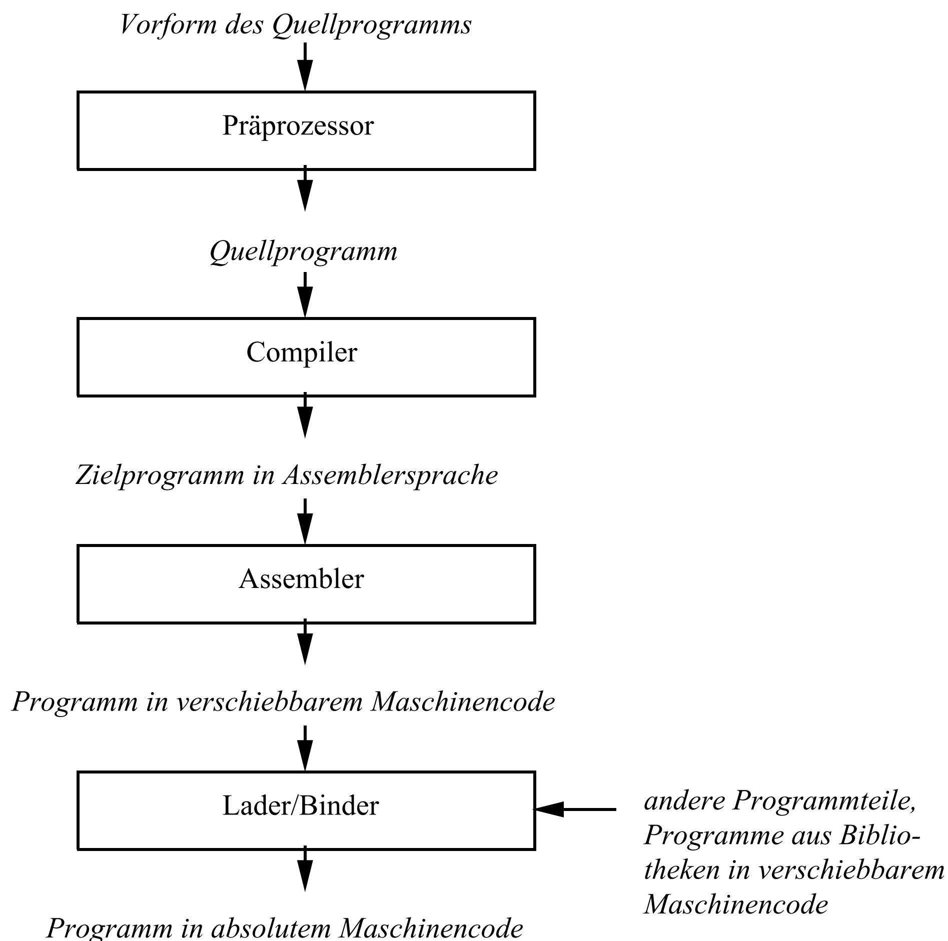


Abb. 1.9. Systemumgebung des Compilers

Die wichtigsten Programme dieser Umgebung sind folgende:

- Ein *Präprozessor* überführt eine „Vorform“ des Quellprogramms in die vom Compiler akzeptierte Sprache. Die Vorform kann z.B. *Makros* enthalten, das sind textuelle Definitionen, für die andere Texte einzusetzen sind. Das heißt, eine Makrodefinition hat grundsätzlich die Form

  ```
  macro alpha = beta
  ```

  wobei *alpha* und *beta* Zeichenketten sind. Der Präprozessor ersetzt dann jedes Auftreten von *alpha* im Text durch *beta*. Vornehmere Makrosprachen erlauben parametrisierte Makros, so daß beim Auftreten von *alpha* noch jeweils variable Texte eingebaut werden können. Die Dokument-Beschreibungssprache LATEX hat z.B. einen solchen Makromechanismus. In der Sprache C werden Konstanten auf diese Art definiert und dann vom Präprozessor ins Programm textuell einkopiert. Eine andere Anwendung von Präprozessoren sind Einbettungen von Datenbankanfragesprachen in Programmiersprachen; in der Vorform des Quellprogramms werden Datenbankkommandos geeignet markiert, z.B.

  ```
  EXEC SQL <DB-Kommando>
  ```

  Der Präprozessor erkennt lediglich solche Kommandos und ersetzt sie im Quelltext durch entsprechende Aufrufe von Funktionen der DB-Schnittstelle, die dann die in der Programmiersprache erlaubte Form haben.

- Ein *Assembler* überführt ein Programm in Assemblersprache in ein „verschiebbares“ Programm in Maschinensprache. Die Assemblersprache ist äquivalent zur Maschinensprache in dem Sinne, daß dort genau sämtliche Maschinenbefehle und Adressierungsmodi wiederzufinden sind. Der wesentliche Unterschied besteht darin, daß die Maschinensprache bekanntlich nur aus Nullen und Einsen besteht, während in der Assemblersprache kurze textuelle Codes für Befehle (z.B. „MOV“ anstelle von „00100110“) und auch symbolische Adressen, also Namen für Speicherzellen oder Register, verwendet werden. Darüber hinaus bieten Assemblersprachen z.B. auch die gerade beschriebenen Makromechanismen.

  Ein Compiler kann prinzipiell genausogut Maschinensprache erzeugen wie Assemblersprache, also die Aufgabe des Assemblers miterledigen. Assemblersprache ist für menschliche Leser besser genießbar; dies könnte z.B. für das Testen des Compilers eine Rolle spielen.

  Wichtig ist, daß zunächst in jedem Fall noch *verschiebbarer* Maschinencode erzeugt wird. Das bedeutet, daß alle im Maschinenprogramm verwendeten Adressen relativ zum Anfang des Programms zu verstehen sind; daneben kann auch indiziert über Basisregister adressiert werden. Es ist die Aufgabe des *Binders* und *Laders*, diese Adressen in absolute Maschinenadressen umzusetzen. Zu diesem Zweck können Maschinenbefehle, in denen Adressen umzusetzen sind, mit einem besonderen gesetzten Bit (*relocation bit*) markiert sein.

- *Binder* (engl. *link editor*) und *Lader* bauen eine Menge von verschiebbaren Maschinenprogrammen aus unterschiedlichen Dateien zu einem einzigen ausführbaren Programm zusammen und laden es in den Hauptspeicher, wo es dann unter Kontrolle des Betriebssystems ausgeführt wird. Dabei sind einerseits die relativen Adressen im verschiebbaren Programm entsprechend der vorgesehenen Ladeposition in absolute Adressen umzuwandeln. Zum anderen müssen externe Referenzen (*links*) aufgelöst werden. Externe Referenzen entstehen dadurch, daß die verschiedenen verschiebbaren Maschinenprogramme schließlich zusammenarbeiten sollen – andernfalls brauchte man sie nicht gemeinsam zu laden. Konkret bedeutet das, daß Programmteil *A* Zugriff auf Variablen des Programmteils *B* haben möchte oder Funktionen aus *B* aufruft. Der Binder verwendet zum Auflösen der Referenzen Tabellen von Namen und relativen Adressen, die jeweils mit dem verschiebbaren Maschinenprogramm gespeichert sind. Wenn also Programmteil *B* eine extern aufrufbare Funktion *squareroot* oder eine Variable *length* enthält, so beschreibt die Tabelle die Einstiegsadresse der Funktion bzw. die Adresse der Variablen:

```
squareroot      264
length           48
```

In anderen Anwendungsgebieten von Compilertechnik gibt es andere Programme, die die Ausgabe eines Compilers weiterverarbeiten. Zum Beispiel wird bei einem Cross-Compiler von C++ nach C die Weiterverarbeitung von einem C-Compiler übernommen. In Kapitel 5 werden wir einen kleinen Compiler konstruieren, der als Ausgabe LATEX erzeugt; dort wird die Ausgabe also vom TEX-System bearbeitet.

1.4 Compiler und Interpreter, reale und abstrakte Maschinen

Eine Alternative zur Übersetzung eines Quellprogramms in ein Zielprogramm, gefolgt von der Ausführung des Zielprogramms zu einer beliebigen späteren Zeit, ist die *Interpretation*. Ein Interpreter benutzt die gleichen Analysetechniken wie ein Compiler. Anstelle der Synthesephasen, in denen der Compiler das Zielprogramm generiert, tritt aber nun die direkte Ausführung. Der Interpreter hat ja aufgrund der Analyse „verstanden“, was zu tun ist, und er tut es auch unmittelbar. Analyse und Ausführung laufen also nun zeitlich verzahnt ab. Der Interpreter wertet beispielsweise in einer bedingten Anweisung die Bedingung aus, erhält den Wert *true*, analysiert daraufhin die erste Anweisung im *then*-Zweig, führt diese aus usw. Beispiele für Programmiersprachen, die in der Regel interpretiert werden, sind *Basic* oder *Lisp*.

Vorteile der Interpretation liegen vor allem in der schnelleren Programmentwicklung. Nach einer Änderung im Quelltext kann das Programm vom Interpreter sofort ausgeführt werden; es entfällt die Wartezeit für den Übersetzungsvorgang. Bei großen Programmsystemen kann diese Wartezeit beträchtlich sein! Der Compiler muß auch stets das gesamte Programm analysieren und übersetzen, während sich die

Analyse im Interpreter auf die im Programmlauf tatsächlich erreichten Teile beschränkt.

Vorteil der Übersetzung ist, daß der Analyseaufwand nur einmal anfällt (na ja: im Programmentwicklungszyklus vermutlich doch etliche Male) und vor allem aus der Ausführungszeit des Programms herausgehalten wird. Ein Interpreter muß eine Anweisungsfolge in einer Schleife, die 1000mal ausgeführt wird, auch 1000mal analysieren. Übersetzte Programme laufen daher im allgemeinen sehr viel schneller als interpretierte.

Interessant ist, daß man beide Techniken auch miteinander kombinieren kann:

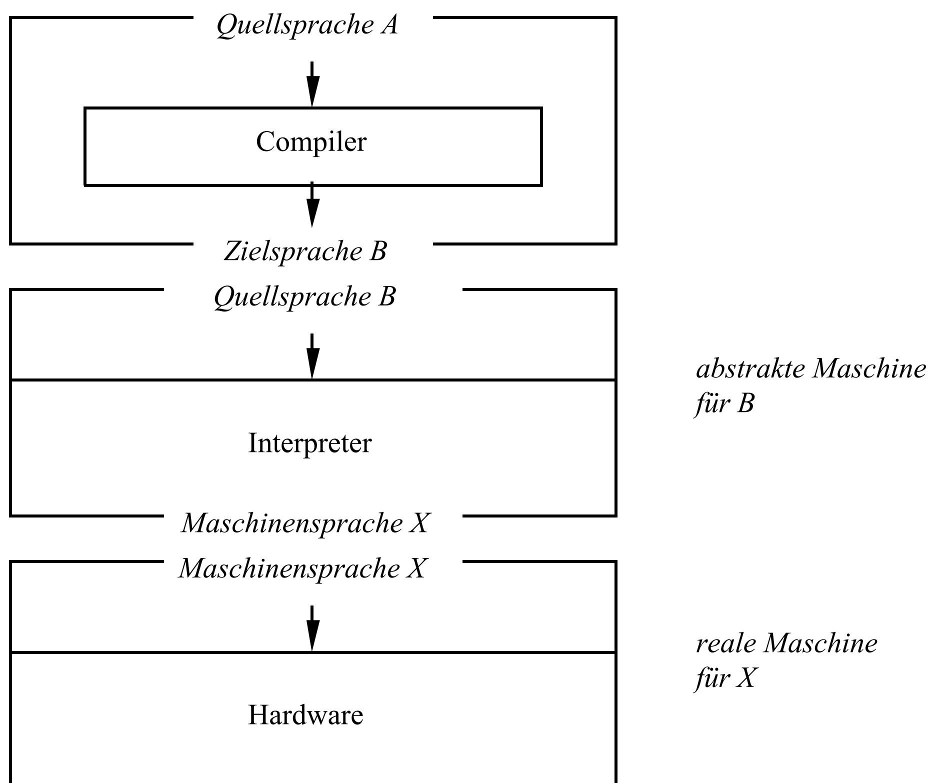


Abb. 1.10. Kooperation von Compiler und Interpreter

Man definiert eine Zwischensprache *B*, in die der Compiler übersetzt; der Interpreter ist in der Lage, Befehle von *B* direkt auszuführen. Der Interpreter selbst liegt in seiner übersetzten Form in einer Maschinensprache *X* vor. Die Zwischensprache *B* definiert man so, daß sie einerseits etwas an die Sprachkonzepte von *A* angepaßt ist, die Übersetzung also erleichtert wird, und daß sie andererseits relativ leicht und effizient zu interpretieren ist.

Bei der Definition der Sprache *B* hat man die Freiheit, sich ein Maschinenmodell auszudenken, das von realer Hardware deutlich abweicht. Ein beliebtes derartiges

Modell ist die *Stackmaschine*, bei der z.B. Transportbefehle Werte vom Speicher nicht in Register laden, sondern sie auf einen Stack (Stapel, Kellerspeicher) legen. Arithmetische Operationen verbrauchen grundsätzlich die obersten Stackelemente und legen das Ergebnis wieder auf den Stack, solche Befehle haben also keine weiteren Parameter. Die Vereinfachung liegt u.a. darin, daß der Stack beliebig groß werden darf, während die Menge der Register einer realen Maschine sehr beschränkt und die Registerzuteilung ein schwieriges Problem ist.

Da der Interpreter die Sprache *B* ebenso ausführen kann wie reale Hardware die Sprache *X*, hat man effektiv eine „abstrakte Maschine" für die Sprache *B* erhalten.

Der Hauptvorteil dieses Ansatzes ist Portabilität. Um die Sprache *A* auf eine neue Maschine *Y* zu bringen, braucht man nur einen Interpreter für *B* in *Y* zu realisieren. Wegen der Einfachheit der Sprache *B* ist das eine relativ leichte Aufgabe.

Die Technik wurde z.B. für die Implementierung von Pascal im UCSD P-System eingesetzt. Derzeit ist sie wieder aktuell als Implementierungsmethode für die Sprache Java, die als Einheitssprache für das Internet propagiert wird. Das Ziel ist, Programme über das Netz versendbar zu machen; sie sollten auf sämtlichen Hardwareplattformen (Workstations, PC, …) lauffähig sein. Der Java-Compiler übersetzt dazu Java in *Byte Codes* (die Zwischensprache); in dieser Form werden Programme über das Internet verschickt. Sämtliche Hardware-Plattformen besitzen Interpreter für die Byte Codes und können somit Java-Programme ausführen.

1.5 Werkzeuge

Der Bau von Übersetzern ist seit langer Zeit studiert worden, und es gibt eine weithin akzeptierte Zerlegung in Teilaufgaben, die wir in Abschnitt 1.2 beschrieben haben. Darüber hinaus kann man für viele dieser Teilaufgaben präzise Spezifikationen des zu lösenden Problems angeben. So wird z.B. die Struktur der zu erkennenden Symbole in der lexikalischen Analyse, wie oben kurz angedeutet, durch reguläre Ausdrücke beschrieben und die Struktur der zu erkennenden Syntaxbäume in der Syntaxanalyse durch Grammatiken. Alles das zusammen hat die Entwicklung von *Werkzeugen* ermöglicht, die heute Unterstützung für fast jede Phase der Übersetzung bieten. Die Eingabe für solche Werkzeuge ist typischerweise eine abstrakte Spezifikation, z.B. auf der Basis von regulären Ausdrücken oder Grammatiken. Ausgabe des Werkzeugs ist ein Programm, das die zu erfüllende Aufgabe löst, z.B. ein *Scanner* (lexikalischer Analysator) oder ein *Parser* („Zerteiler", ein Programm, das die Syntaxanalyse durchführt). Solche Werkzeuge, die Programme als Ausgabe erzeugen, werden *Generatoren* genannt. Es gibt Werkzeuge z.B. für folgende Teilaufgaben:

- Scannergenerator
- Parsergeneratoren für LALR(1)- oder LL(1)-Grammatiken (diese Grammatiktypen werden in Kapitel 3 erklärt)

- Generator für abstrakte Syntaxbäume
- Generator für Attributauswerter (*attributierte Grammatiken* sind in Kapitel 4 beschrieben)
- Transformation abstrakter Syntaxbäume
- Generator für Codegeneratoren

Die bekanntesten Vertreter derartiger Werkzeuge sind die UNIX-Tools *Lex* und *Yacc*; sie werden mit dem Betriebssystem UNIX zusammen verbreitet. *Lex* ist ein Scannergenerator. In Abschnitt 1.2 hatten wir die Struktur eines Bezeichners mit einem regulären Ausdruck so beschrieben:

```
letter (letter | digit)*
```

Eine entsprechender Ausschnitt einer Lex-Spezifikation könnte so aussehen:

```
letter          [A-Za-z]
digit           [0-9]
%%
{letter}({letter}|{digit})*      {return(IDENTIFIER)}
```

Die Spezifikation besteht aus zwei Teilen, die durch %% getrennt sind. Im ersten Teil werden lexikalische Symbole *letter* und *digit* definiert, jeweils durch Angabe eines Bereiches aus dem ASCII-Zeichensatz. *Letter* steht also nun für einen Buchstaben, *digit* für eine Ziffer. Im zweiten Teil wird auf der linken Seite die Struktur von Bezeichnern definiert. Auf der rechten Seite steht in den geschweiften Klammern eine *Aktion*, die auszuführen ist, wenn in der Eingabezeichenfolge die Struktur der linken Seite erkannt wird. Die Aktion ist in C formuliert, in diesem Fall ist lediglich ein Token *IDENTIFIER* auszugeben.

Yacc ist ein Parsergenerator (das Wort steht für „yet another compiler-compiler“). Die Regeln aus Abschnitt 1.2

stmt	::=	*assignment* | *cond*
cond	::=	**if** *boolexpr* **then** *stmt* **end** |
		if *boolexpr* **then** *stmt* **else** *stmt* **end**

könnte man in einer Yacc-Spezifikation fast genauso hinschreiben:

```
stmt    :       assigment               {…}
        |       cond                    {…}
        ;

cond    : IF boolexpr THEN stmt END{…}
        | IF boolexpr THEN stmt ELSE stmt END{…}
        ;
```

Dabei sind IF, THEN usw. Token, die aus der lexikalischen Analyse kommen, also z.B. von einem Lex-generierten Scanner stammen. Die Lex-Spezifikation für das Token IF sieht so aus:

```
if              {return(IF)}
```

Die geschweiften Klammern in der Yacc-Spezifikation enthalten ebenfalls Aktionen (sog. *semantische Regeln*), die auszuführen sind, wenn die entsprechende *rechte Seite* der Regel erkannt wird; dies ist hier nur angedeutet. Übrigens würde Yacc in seiner Analyse dieser Spezifikation die obigen Regeln für *cond* wohl mit einer Warnung (wegen Mehrdeutigkeit) versehen.

Diese Werkzeuge sind mächtige Hilfsmittel, mit denen man mit wenig Aufwand Scanner und Parser implementieren kann, und man sollte sie unbedingt kennen. Wir besprechen Lex und Yacc jeweils im Zusammenhang der Kapitel über lexikalische und Syntaxanalyse. In Kapitel 5 implementieren wir einen kompletten Compiler für eine Dokument-Beschreibungssprache mit Hilfe dieser Werkzeuge.

1.6 Struktur des Kurses

Die Gliederung des Kurses folgt im wesentlichen der in Abschnitt 1.2 beschriebenen Zerlegung in Phasen. In Kapitel 2 besprechen wir die lexikalische Analyse und den Einsatz des Werkzeugs Lex. Thema von Kapitel 3 ist die Syntaxanalyse; wir betrachten zunächst Top-down-Verfahren, die den Syntaxbaum von der Wurzel her aufbauen. Es folgen Bottom-up-Verfahren, die von den Blättern her Teilbäume erzeugen, bis schließlich das Wurzelsymbol erreicht wird. Diese Technik wird auch von Yacc-generierten Parsern verwendet; entsprechend wird Yacc hier mitbehandelt. In Kapitel 4 werden *attributierte Grammatiken* eingeführt; sie erlauben die Kopplung von Übersetzungsaktionen an das Erkennen von Teilbäumen in der Syntaxanalyse. Dies ist der theoretische Hintergrund der oben gezeigten „semantischen Regeln“, die in Yacc-Spezifikationen verwendet werden. In den folgenden drei Kapiteln wenden wir die eingeführten Methoden auf drei Einsatzgebiete von Compilertechnik an. In Kapitel 5 konstruieren wir einen Compiler für eine einfache Sprache zur Beschreibung von Textdokumenten unter Einsatz von Lex und Yacc. Die Kapitel 6 und 7 sind der Übersetzung klassischer imperativer bzw. funktionaler Programmiersprachen gewidmet. Schließlich behandelt Kapitel 8 die Phasen der Optimierung des Zwischencodes und der Codeerzeugung.

1.7 Literaturhinweise

Natürlich gibt es eine große Auswahl an Büchern zum Übersetzerbau. Wir wollen fünf Bücher herausheben und als Begleitlektüre besonders empfehlen. Sie sind nach der Reihenfolge ihrer Bedeutung für diesen Kurs geordnet.

1. Mit großem Abstand das wichtigste, klassische Buch zum Übersetzerbau ist (Aho et al. 2006), das sog. „Drachenbuch“ (so genannt wegen des Umschlagdesigns). Es hat auf die Darstellung des Gebietes in Lehrbüchern und Vorlesungen großen Einfluß gehabt, und auch wir orientieren uns in erster Linie an diesem

Buch. Es behandelt umfassend, in klarer Darstellung und mit gutem Blick für praktische Probleme alle grundlegenden Fragen des Übersetzerbaus. – Dieses Buch gibt es auch in deutscher Sprache (Aho et al. 2008).

2. Das Buch von Parsons (1992) folgt weitgehend der vom Drachenbuch vorgegebenen Struktur. Es besticht aber durch sehr gute, motivierende Erklärungen und Darstellungen, vielleicht deshalb, weil es für „Undergraduates“ (Studenten in den ersten Studienjahren) geschrieben wurde. In manchen Passagen ist die Darstellung klarer als im Drachenbuch.
3. Ein sehr gutes deutschsprachiges Buch mit einer originellen Struktur ist (Wilhelm und Maurer 2007). Während die meisten Bücher – wie das Drachenbuch – zunächst Übersetzungstechniken vorstellen (lexikalische Analyse, Syntaxanalyse, attributierte Grammatiken) und sich dann auf die Übersetzung imperativer Programmiersprachen konzentrieren bzw. beschränken, beginnt dieses Buch mit vier Kapiteln zur Implementierung imperativer, funktionaler, logischer und objektorientierter Programmiersprachen und betrachtet danach die Methodik der Übersetzung. Funktionale, logische und objektorientierte Programmiersprachen werden in den anderen genannten Büchern nicht betrachtet; insofern ist dieses Buch umfassender.
4. Eine weitere umfassende und gründliche Darstellung des Gebietes Compilerbau bietet (Waite und Goos 1984). Eine modulare Compiler-Struktur und der Einsatz attributierter Grammatiken werden besonders betont.
5. Das Buch von Sudkamp (2005) ist kein spezielles Buch zum Übersetzerbau, sondern ein gutes allgemeines Theorie-Buch. Im Unterschied zu manchen anderen derartigen Büchern enthält es Kapitel zu LL(k)-Grammatiken und LR(k)-Grammatiken. Bei einigen formalen Definitionen in Kapitel 3 haben wir uns an diesem Buch orientiert. Es ist also ein gutes Nachschlagewerk zu den formalen Grundlagen von Sprachen und Maschinen.

Weitere empfehlenswerte Bücher zum Thema Übersetzerbau sind z.B. (Alblas und Nymeyer 1996, Appel und Ginsburg 1997, Holmes 1995a, Holub 1990, Kastens 1990, Pittmann und Peters 1992, Wirth 2008). Speziell der Übersetzung objektorientierter Sprachen gewidmet ist (Bauer und Höllerer 1998). Compilerbau-Projekte, d.h. die einigermaßen vollständige Entwicklung eines Beispielcompilers für eine ausgewählte Anwendung bzw. einen Sprachausschnitt, werden u.a. in (Doberkat und Fox 1990, Holmes 1995b) und in (Wirth 2011) beschrieben.

Darüber hinaus ist es nützlich, Bücher verfügbar zu haben, um Details nachschlagen zu können zu Lex und Yacc, z.B. (Levine, Mason und Brown 1992), zur Programmiersprache C, z.B. (Kernighan und Ritchie 1990), und zu LATEX, z.B. (Lamport 1994).

Die Darstellung in diesem einführenden Kapitel orientiert sich im wesentlichen an (Aho et al. 2006).

Kapitel 2

Lexikalische Analyse

Die lexikalische Analyse ist die erste Analysephase innerhalb des Übersetzungsprozesses; das Ziel besteht darin, einen Strom von Eingabezeichen in eine Folge von Token umzuwandeln, die nützliche „atomare“ Einheiten für die nächste Phase, die Syntaxanalyse, darstellen (s. Abb. 1.3).

Die entsprechenden Programme, also *Scanner* und *Parser*, arbeiten gewöhnlich verschränkt bzw. in einer Pipeline: Der Parser fordert vom Scanner jeweils ein Token an – der Scanner liest so lange Eingabezeichen, bis ein Token erkannt wurde, und liefert es dann dem Parser.

Bei der Überlegung, was Grundsymbole sind, die in der lexikalischen Analyse erkannt werden sollten, hat man eine gewisse Freiheit. In Kapitel 1 wurde schon kurz angedeutet, daß lexikalische Symbole mit regulären Ausdrücken spezifiziert werden können, während kontextfreie Grammatiken Basis der Syntaxanalyse sind. Nun sind kontextfreie Grammatiken der mächtigere Formalismus. Das heißt, alles, was man mit regulären Ausdrücken beschreiben kann, kann auch mit kontextfreien Grammatiken definiert werden – also auch die lexikalischen Symbole.

Ein Kriterium ist sicherlich die Effizienz. Die in Scannern ablaufenden Mechanismen sind einfacher als die in Parsern; von daher sollte man Textstrukturen soweit möglich im Rahmen der lexikalischen Analyse erkennen. Ein zweiter wichtiger Gesichtspunkt ergibt sich aus den Methoden, die zur Syntaxanalyse eingesetzt werden: Dort werden Entscheidungen, welche Regel einer Grammatik anzuwenden ist, gewöhnlich durch Vorausschau in der Eingabe-Tokenfolge getroffen, und zwar durch Vorausschau auf *genau ein Token*! Daraus folgt, daß es etwa zum Erkennen einer bedingten Anweisung zwingend erforderlich ist, daß *if* ein Token ist. Denn nach Lesen des Zeichens *i* kann der Parser eine solche Entscheidung nicht treffen. Dieses Kriterium sollte man also beim Entwurf der lexikalischen Analyse beachten.

Die lexikalische Analyse hat einen seit langem gut verstandenen theoretischen Hintergrund. Dieser läßt sich ganz kurz so zusammenfassen:

1. Die Struktur lexikalischer Symbole kann durch *reguläre Ausdrücke* beschrieben werden. Das heißt, die Menge der Zeichenketten, die auf ein Token abgebildet werden, ist die zum Ausdruck gehörige *reguläre Sprache*.
2. Reguläre Sprachen werden durch *rechtslineare* (oder linkslineare) *Grammatiken* erzeugt.

3. Sie werden von *nichtdeterministischen endlichen Automaten* erkannt.
4. Zu jedem nichtdeterministischen endlichen Automaten kann man auch einen *deterministischen endlichen Automaten* konstruieren, der die gleiche Sprache erkennt.

Wir werden die wichtigsten dieser Begriffe kurz wiederholen und uns auf die praktische Anwendung der Theorie im Compilerbau konzentrieren. In den folgenden Abschnitten betrachten wir die Beschreibung lexikalischer Symbole mit regulären Ausdrücken, die Beschreibung mit Zustandsdiagrammen (endlichen Automaten) und die „Handimplementierung" solcher Zustandsdiagramme. Wir führen die Sprache der *Lex-Spezifikationen* ein und zeigen die Implementierung eines kleinen speziellen Analysators mit *Lex*.

2.1 Beschreibung von Token durch reguläre Ausdrücke

Zunächst müssen wir den Begriff einer *formalen Sprache* präzisieren. Ein *Alphabet* ist eine endliche, nichtleere Menge. Für ein Alphabet Σ bezeichnet Σ^* die Menge aller Folgen von Elementen aus Σ. Die Elemente von Σ^* werden auch *Wörter* über Σ genannt. Die Menge Σ^* enthält insbesondere die leere Folge bzw. das *leere Wort*, das mit dem Symbol ε bezeichnet wird. Eine *Sprache über* Σ ist eine Teilmenge von Σ^*.

Sei z.B. $\Sigma = \{a, b, c\}$. Dann sind *a*, *bbabba, cc* und ε Wörter über Σ, und $L = \{a, bbabba, cc, \varepsilon\}$ ist eine Sprache über Σ. Die leere Menge $\emptyset$ ist auch eine Sprache über Σ; sie enthält kein einziges Wort, nicht einmal das leere.

Auf Wörtern gibt es eine Operation, die *Konkatenation*, d.h. das Aneinanderhängen. Sei *v* = *Kaffee* und *w* = *maschine*, dann ist *vw* = *Kaffeemaschine*, die Konkatenation von *v* und *w*. Das leere Wort ε ist das neutrale Element bezüglich der Konkatenation.

Interessante Operationen auf Sprachen sind Vereinigung, Konkatenation und Abschlußoperation. Seien also L, L_1 und L_2 Sprachen.

$L_1 \cup L_2$	Vereinigung
$L_1L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}$	Konkatenation
$L^n = \{x_1 \ldots x_n \mid x_i \in L, 1 \leq i \leq n\}$	
$L^* = \bigcup_{i \geq 0} L^i$	Abschluß
$L^+ = \bigcup_{i > 0} L^i = L^* \setminus \{\varepsilon\}$	positiver Abschluß

Definition 2.1: Die *regulären Sprachen* über Σ werden durch folgende Regeln induktiv definiert:

(i) $\varnothing$ und $\{\varepsilon\}$ sind reguläre Sprachen.
(ii) Für jedes $a \in \Sigma$ ist $\{a\}$ eine reguläre Sprache.
(iii) Seien R und S reguläre Sprachen, dann sind auch $R \cup S$, RS und R^* reguläre Sprachen.
(iv) Nichts sonst ist eine reguläre Sprache über Σ. □

Ein *regulärer Ausdruck r* beschreibt eine reguläre Sprache $L(r)$.

Definition 2.2: *Reguläre Ausdrücke* werden ebenfalls induktiv definiert:

(i) $\varnothing$ ist ein regulärer Ausdruck, der die reguläre Sprache $\varnothing$ beschreibt. ε ist ein regulärer Ausdruck, der die Sprache $\{\varepsilon\}$ beschreibt.
(ii) Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck; er beschreibt die Sprache $\{a\}$.
(iii) Wenn r und s reguläre Ausdrücke sind, die die Sprachen R und S beschreiben, so ist auch
  $(r \mid s)$ ein regulärer Ausdruck, der $R \cup S$ beschreibt,
  rs ein regulärer Ausdruck, der RS beschreibt,
  r^* ein regulärer Ausdruck, der R^* beschreibt.
(iv) Nichts sonst ist ein regulärer Ausdruck. □

Selbsttestaufgabe 2.1: Geben Sie reguläre Ausdrücke für die folgenden Mengen (Sprachen) an:

(a) $\{w \in \{\text{a, b, c, d}\}^* \mid$ die Zeichen in w sind lexikographisch aufsteigend von links nach rechts sortiert$\}$

(b) $\{w \in \{0, 1\}^* \mid w$ enthält höchstens eine $1\}$

(c) $\{w \in \{0, 1\}^* \mid w$ enthält mindestens eine $1\}$ □

Um die Spezifikation mit regulären Ausdrücken etwas bequemer zu machen, führen wir den Begriff der *regulären Definition* ein. Dabei werden Namen für reguläre Ausdrücke definiert, die in folgenden regulären Ausdrücken wie Zeichen des Alphabets verwendet werden können. Eine *reguläre Definition* hat die Form

$$\begin{array}{lcl} d_1 & \to & r_1 \\ d_2 & \to & r_2 \\ \ldots & & \\ d_n & \to & r_n \end{array}$$

Dabei ist jeweils d_i ein Name für den regulären Ausdruck r_i. Diese Namen nennen wir *reguläre Symbole*. Weiterhin gilt, daß in r_i nur die Namen $d_1, \ldots, d_{i-1}$ vorkommen dürfen. Andernfalls wären rekursive Definitionen möglich, die den Rahmen der regulären Sprachen sprengen würden.

Mit diesen Mitteln können wir die schon in Kapitel 1 im Beispiel gezeigte Struktur von Bezeichnern so definieren:

```
letter         ->     A | B | C | … | Z | a | b | c | … | z
digit          ->     0 | 1 | … | 9
identifier     ->     letter (letter | digit)*
```

Eine ganzzahlige Konstante hat die Beschreibung:

```
sign           ->     + | -
constant       ->     (sign | ε) digit digit*
```

Schlüsselwörter einer Programmiersprache:

```
if             ->     if
then           ->     then
begin          ->     begin
```

Bei der Beschreibung formaler Sprachen hat man oft das Problem, daß zwischen Zeichen der Sprache und Metasymbolen, die der Beschreibung dienen, unterschieden werden muß. Wir wählen Fettdruck, um das reguläre Symbol **then** von dem regulären Ausdruck *then* zu unterscheiden, der definitionsgemäß die Konkatenation der regulären Ausdrücke *t*, *h*, *e* und *n* darstellt und die Sprache {*then*} beschreibt. Weiterhin nehmen wir an, daß die Zeichen

```
(   )   |   *   ε
```

Metasymbole sind.

Wir führen noch einige weitere vereinfachende Notationen ein:

1. + ist ein unärer Postfixoperator und steht für *ein- oder mehrmaliges Auftreten* des vorangehenden Symbols. $r+$ bezeichnet also die Sprache $(L(r))^+$ (oben definiert).
2. ? ist ein unärer Postfixoperator und steht für *optionales Auftreten*. $r?$ bezeichnet damit $L(r) \cup \{\varepsilon\}$.
3. *Zeichenklassen*. [*axf*9&], wobei die Zeichen *a*, *x*, usw. aus Σ sind, steht abkürzend für (a | x | f | 9 | &). [A–Z] steht für einmaliges Auftreten eines Zeichens aus dem Bereich A bis Z. [^abcd] beschreibt das *Komplement* der Menge der Zeichen *a*, *b*, *c*, *d*, also alle anderen Zeichen, ^ ist der Komplementoperator. Mit [^A–Za–z0–9] kann man daher z.B. alle Zeichen beschreiben, die nicht Buchstaben oder Ziffern sind.

Damit vereinfachen sich die obigen Definitionen:

```
letter         ->     [A-Za-z]
digit          ->     [0-9]
identifier     ->     letter (letter | digit)*

sign           ->     + | -
constant       ->     sign? digit+
```

Reguläre Symbole, die an den Parser weitergereicht werden, sind gerade Token. Die Wörter der durch ein reguläres Symbol beschriebenen Sprache heißen auch *Lexeme* zu diesem Symbol bzw. Token.

2.2 Beschreibung von Token durch Zustandsdiagramme

Wie in der Einleitung dieses Kapitels schon erwähnt, sind Beschreibungen mit regulären Ausdrücken äquivalent zu solchen mit endlichen Automaten; d.h. mit endlichen Automaten kann man die durch reguläre Ausdrücke definierten Sprachen akzeptieren. Die klassische Vorgehensweise in der Theorie konstruiert zu einem regulären Ausdruck zunächst einen nichtdeterministischen endlichen Automaten (NEA) und wandelt diesen dann um in einen äquivalenten deterministischen endlichen Automaten (DEA). Ein DEA läßt sich relativ leicht in ein Analyseprogramm übersetzen.

Diese Vorgehensweise wird z.B. benötigt, um aus einer durch reguläre Ausdrücke gegebenen Scannerspezifikation (z.B. für *Lex*) einen Scanner zu generieren. In diesem Abschnitt wollen wir eine direkte „Handimplementierung" eines Scanners anstreben. Dazu ist es einfacher, die Struktur der Token direkt mit Hilfe von *Zustandsdiagrammen* anzugeben, die nichts anderes sind als eine graphische Notation für deterministische endliche Automaten.

Wir wiederholen kurz die Definition eines DEA:

Definition 2.3: Ein *deterministischer endlicher Automat M* ist gegeben als $M = (Q, \Sigma, \delta, s, F)$, wobei gilt:

(i) Q ist eine endliche nichtleere Menge von *Zuständen*,
(ii) Σ ist ein Alphabet von *Eingabezeichen*,
(iii) $\delta: Q \times \Sigma \rightarrow Q$ ist eine *Übergangsfunktion*,
(iv) $s \in Q$ ist ein *Anfangszustand*,
(v) $F \subseteq Q$ ist eine Menge von *Endzuständen*. □

Der Automat befindet sich jeweils in einem Zustand aus Q, zu Anfang im Zustand s. In jedem Schritt liest er ein Eingabezeichen und geht über in einen neuen Zustand, der durch δ bestimmt wird. Sobald ein Zustand in F erreicht wird, gilt die bisher durchlaufene Zeichenfolge als akzeptiert.

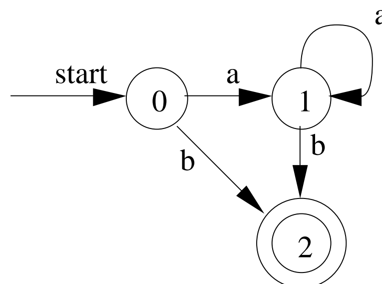


Abb. 2.1. Zustandsdiagramm eines endlichen Automaten

Abbildung 2.1 zeigt einen endlichen Automaten, dargestellt als Zustandsdiagramm. Hier ist $Q = \{0, 1, 2\}$, $\Sigma = \{a, b\}$, $s = 0$, $F = \{2\}$, und die Übergangsfunktion δ ist als Tabelle dargestellt:

Q	S	Q'
0	a	1
0	b	2
1	a	1
1	b	2

Die akzeptierte Sprache ist offensichtlich $(aa^*b \mid b)$, was man allerdings auch einfacher darstellen könnte (sowohl als Automat wie auch als regulärer Ausdruck, nämlich wie?). Wie man sieht, kennzeichnen wir in der graphischen Notation den Anfangszustand durch einen Pfeil „start“, Endzustände durch einen Doppelkreis.

Selbsttestaufgabe 2.2: Geben Sie für die durch die regulären Ausdrücke aus Aufgabe 2.1 definierten Sprachen deterministische endliche Automaten als Quintupel $M = (Q, \Sigma, \delta, s, F)$ und in Form von Zustandsdiagrammen an. Die Übergangsfunktion δ braucht nicht textuell (z.B. als Tabelle) dargestellt zu werden. □

Im folgenden wollen wir für eine einfache Anwendung die lexikalische Analyse mit Zustandsdiagrammen beschreiben und dann auch „von Hand“ implementieren. In der Anwendung geht es darum, mit „geschachtelten Listen“ hantieren zu können, wie sie in ähnlicher Form in der Programmiersprache *Lisp* vorkommen. Wir wollen z.B. Anfragen an ein Datenbanksystem in diesem Format stellen können oder auch die in einer Datenbank gespeicherte Information in diesem Format in eine Datei schreiben und daraus lesen können. Eine Anfrage könnte z.B. so aussehen:

```
(select
        cities
        (       fun
                (c city)
                (and    (>= (attribute c pop) 500000)
                        (= (attribute c state) "Germany")
                )
        )
)
```

Die Anfrage wählt aus einer Menge von Städten diejenigen aus, deren Einwohnerzahl („population“) mindestens 500 000 ist und die innerhalb Deutschlands liegen. Die genaue Struktur der Anfrage ist hier nicht weiter von Interesse. Wichtig sind aber die Konventionen für die textuelle Darstellung solcher geschachtelter Listen. Eine Liste hat die Form

$$(elem_1 \;\; elem_2 \;\; \ldots \;\; elem_n)$$

wobei jedes Element entweder ein „Atom“ ist oder wieder eine Liste der gleichen Form. Eine Liste darf auch leer sein, d.h. $n \geq 0$. Die Elemente einer Liste sind durch

„Leerraum“ getrennt; Leerraum darf eine beliebige Folge von Leerzeichen („blanks“), Tabulatorsymbolen oder Zeilenendesymbolen sein.

Die obige äußerste Liste besteht also aus drei Elementen, nämlich „select“, „cities“ und der darauf folgenden Liste, die selbst „fun“ als erstes Element hat.

Wir erlauben folgende Arten von Atomen. Links steht der Name des Atoms, rechts Beispielzeichenketten (Lexeme):

```
Integer         12, -371
Real            3.14, 14.8E02
Boolean         TRUE, FALSE
String          "Hello, World!", "Germany"
Symbol          cities, select, fun, >=
Text            <text>Ein noch sehr kurzer "Text".</text--->
```

Um die Syntax von Atomen zu präzisieren, machen wir folgende Annahmen:

- Die *Integer*- und *Real*-Darstellungen sollen den in Programmiersprachen üblichen entsprechen, sagen wir, denen in Modula-2.
- Die *Boolean*-Darstellungen seien genau die beiden gezeigten Lexeme.
- Ein *String* ist eine in Anführungsstrichen eingeklammerte Folge von bis zu 48 beliebigen Zeichen, die allerdings keine (Doppel-)Anführungsstriche enthalten darf.
- Ein *Symbol* ist eine Folge von bis zu 48 Zeichen, die durch folgende reguläre Definitionen beschrieben wird:

  ```
  symchar -> [^A-Za-z0-9 \t\n()\"]
  symbol  -> letter (letter | digit)* | symchar+
  ```

  Das heißt, ein *Symbol* sieht entweder so aus wie ein Bezeichner in Programmiersprachen oder es besteht nur aus bestimmten Sonderzeichen (*symchar*). Als Sonderzeichen sind *nicht* zugelassen Buchstaben, Ziffern, Leerzeichen, Tabulatorsymbole (\t), Zeilenendesymbole (\n), öffnende und schließende Klammern und Doppelanführungsstriche. Die Notation für diese Symbole entstammt der Sprache C und ist auch in *Lex* zu verwenden. – Das zweite Format für *Symbole* würde z.B. die Darstellung von „>=“ erlauben.
- Ein *Text* ist eine beliebige und auch beliebig lange Zeichenfolge, die durch die Zeichenketten „<text>“ und „</text--->“ eingeklammert ist. Verlangt wird lediglich, daß in der Zeichenfolge nicht die schließende Klammer „</text--->“ vorkommt.

Die Wahl der Grundsymbole (Token) ist für dieses Beispiel relativ offensichtlich. Benötigt werden öffnende und schließende Klammern für die Listenebene. Leerraum sollte von der lexikalischen Analyse verschluckt werden, also nicht als Token weitergegeben werden. Jede der ersten 5 Arten von Atomen wird zu einem Token werden. Für *Text*-Atome werden wir zwei Token OPENTEXT und CLOSETEXT einführen, die die speziellen Klammerungszeichenketten darstellen; der Text inner-

halb dieser Klammern sollte unverarbeitet an den Parser weitergereicht werden. Man kann nicht gut *Text* als lexikalisches Symbol einführen, da die dargestellten Texte beliebig lang werden können und der Eingabepuffer der lexikalischen Analyse beschränkt ist.

Wir beschreiben die Token nun durch Zustandsdiagramme:

Abb. 2.2. Zustandsdiagramme für die Token OPEN und CLOSE

Die Diagramme für öffnende und schließende Klammern sind trivial. Wir markieren Endzustände noch mit der bei Erreichen auszuführenden Aktion. Hier sind also die Token OPEN bzw. CLOSE an den Parser zu liefern.

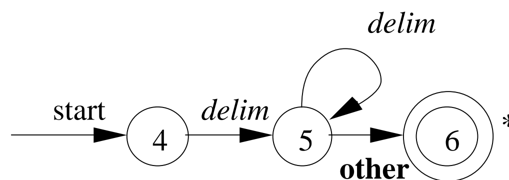


Abb. 2.3. Zustandsdiagramm zur Behandlung von Leerraum

Das Diagramm in Abb. 2.3 beschreibt die Behandlung von „Leerraum“. Hier ist *delim* eine Zeichenklasse, die als

```
delim   -> [ \t\n]
```

definiert ist. Im Syntaxdiagramm bedeutet das, daß für jedes dieser Zeichen der entsprechende Übergang definiert ist. Die Notation **other** steht für alle Zeichen, für die nicht andere Übergänge aus diesem Zustand definiert sind. Hier ist zu beachten, daß das Zeichen, mit dem der Übergang von Zustand 5 nach Zustand 6 erfolgt, selbst nicht mehr zu dem zu erkennenden Symbol (Leerraum) gehört. Nach dem Erreichen des Endzustandes muß daher der Eingabezeiger um ein Zeichen zurückgesetzt werden, damit dieses Zeichen noch in das nächste Token eingehen kann. Dies wird

durch den Stern am Endzustand ausgedrückt. – Man beachte, daß bei diesem Diagramm kein Token an den Parser zurückgegeben wird; insofern wird Leerraum „verschluckt". Das Zustandsdiagramm für INTEGER-Token ist auch noch relativ einfach (Abb. 2.4).

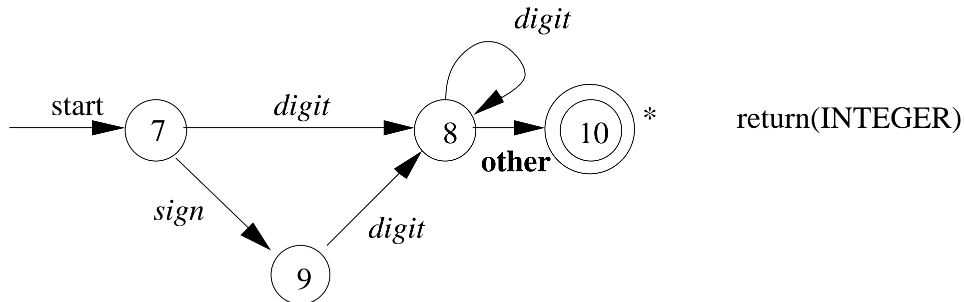


Abb. 2.4. Zustandsdiagramm für INTEGER-Token

Das Diagramm für REAL-Token (Abb. 2.5) ist das komplizierteste; hier sind die Ziffernfolge für Nachkommastellen und die Angabe eines Exponenten jeweils optional. Die Zeichenklasse *sign* ist hier, wie auch beim INTEGER-Diagramm (Abb. 2.4), als [+–] definiert.

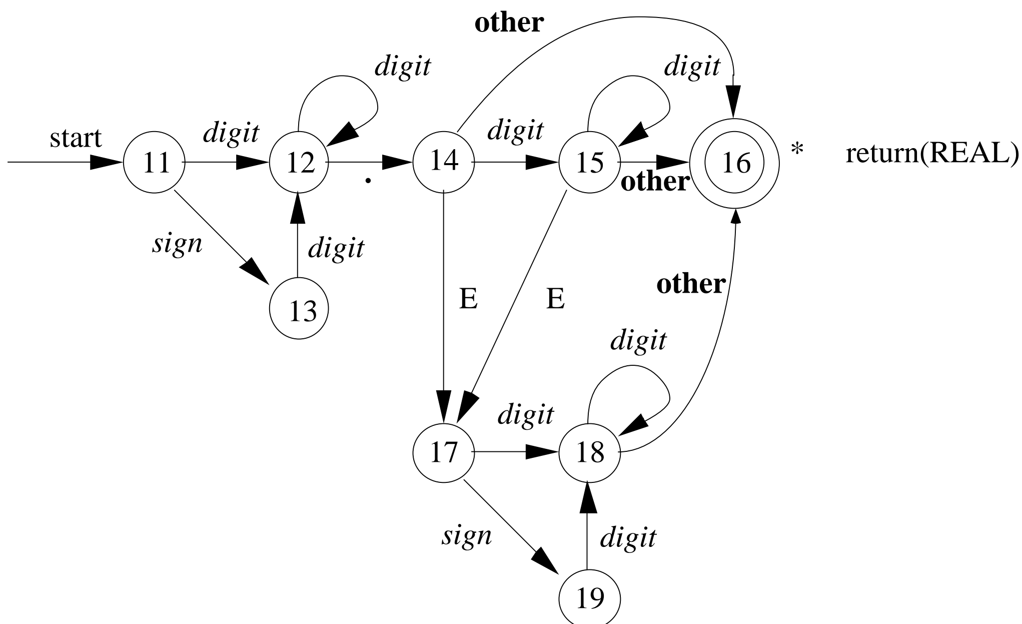


Abb. 2.5. Zustandsdiagramm für REAL-Token

Die übrigen Diagramme für BOOLEAN-Token und STRING-Token sind vergleichsweise einfach (Abb. 2.6 und Abb. 2.7).

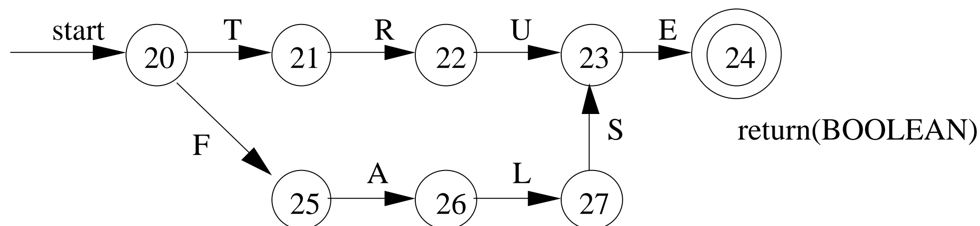


Abb. 2.6. Zustandsdiagramm für BOOLEAN-Token

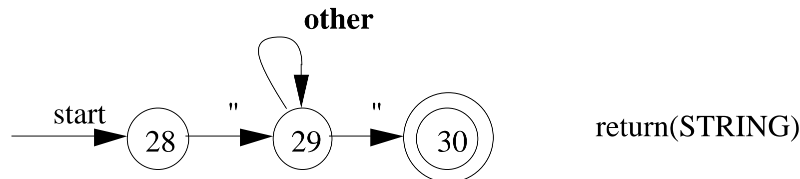


Abb. 2.7. Zustandsdiagramm für STRING-Token

Selbsttestaufgabe 2.3: Definieren Sie die Syntaxdiagramme für die SYMBOL-, OPENTEXT und CLOSETEXT-Token. □

2.3 Direkte Implementierung eines Scanners

Für eine gegebene Menge von Zustandsdiagrammen als Beschreibung der zu erkennenden Token läßt sich auf relativ einfache Art ein lexikalischer Analysator (Scanner) implementieren. Wir zeigen eine solche Implementierung in der Sprache C.

Zunächst werden die Token als Integer-Konstanten definiert:

```
#define OPEN            1000
#define CLOSE           1001
#define INTEGER         1002
#define REAL            1003
#define BOOLEAN         1004
#define STRING          1005
#define SYMBOL          1006
#define OPENTEXT        1007
#define CLOSETEXT       1008
```

Ein Scanner reicht im allgemeinen nicht nur explizit definierte Token an den Parser weiter, sondern gelegentlich auch einfache Zeichen der Eingabe (ein Zeichen ist damit sein eigenes Token). Daher ist es sinnvoll, die Werte für diese Konstanten so zu wählen, daß sie nicht mit den Werten der einzelnen Zeichen (ASCII-Codes) kollidieren. Deshalb haben wir hier Werte größer als 1000 benutzt.

Die Eingabezeichenfolge steht in einem Pufferbereich:

```
char buffer[BUFFERSIZE];
```

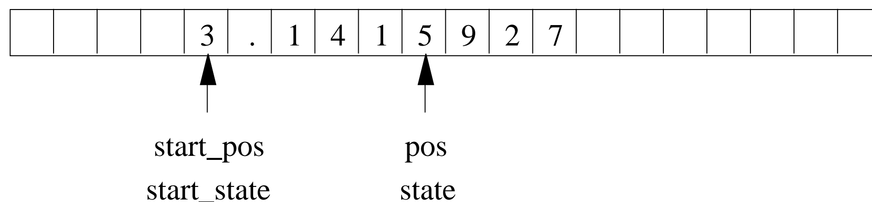


Abb. 2.8. Verwaltung von Zeigern auf Zeichenpuffer

Es gibt zwei Zeiger in diesen Puffer. Der Zeiger *pos* zeigt auf das nächste zu lesende Zeichen. Der Zeiger *start_pos* zeigt während der Analyse eines Tokens auf das Zeichen, das beim Einstieg in das gerade untersuchte Zustandsdiagramm aktuell war. Die Gesamtstrategie der lexikalischen Analyse sieht nämlich so aus, daß der Reihe nach Zustandsdiagramme untersucht, d.h. durchlaufen werden. Falls die Analyse in einem Zustandsdiagramm fehlschlägt, wird das nächste Diagramm untersucht. Dann muß aber der Zeiger *pos* auf die gemerkte Position *start_pos* zurückgesetzt werden.

Die Variablen *start_state* und *state* beschreiben in ähnlicher Weise den Zustand beim Einstieg in das aktuelle Zustandsdiagramm und den aktuellen Zustand. Der Anfangszustand *start_state* wird benutzt, um bei Fehlschlag den Übergang auf das nächste Zustandsdiagramm zu steuern. Wir haben also globale Variablen:

```
int start_pos = 0, pos = 0;
int start_state, state;
```

Eine Funktion *nextchar*() liefert jeweils das aktuelle Zeichen und erhöht *pos* um 1. Weiterhin haben wir oben gesehen, daß manchmal der Eingabezeiger um eine Position zurückgesetzt werden muß, weil das letzte gelesene Zeichen nicht mehr zum Token gehört; dies leistet eine Funktion *stepback*().

Die lexikalische Analyse wird dann durch eine Funktion *gettoken*() realisiert, die bei jedem Aufruf das nächste Token liefert. Diese Funktion simuliert einen endlichen Automaten, indem in einer Schleife in jedem Durchlauf ein Zustandsübergang durchgeführt wird. Die Zustände werden in einer großen Fallunterscheidung aufgelistet; für jeden Zustand wird gemäß Zustandsdiagramm für die möglichen Eingabezeichen der Folgezustand gesetzt. Wir zeigen ein Anfangsstück dieser Funktion, das

den Programmcode für die Zustandsdiagramme für die Token OPEN, CLOSE und INTEGER sowie für Leerraum enthält.

```
#define TRUE    1

int gettoken()
{
    int c;
    state = 0; start_state = 0;
    while (TRUE) {
        switch (state) {
        case 0: c = nextchar();
                if (c == '(')   state = 1;
                else            state = next_diagram();
                break;
        case 1: return(OPEN);
        case 2: c = nextchar();
                if (c == ')')   state = 3;
                else            state = next_diagram();
                break;
        case 3: return(CLOSE);
        case 4: c = nextchar();
                if isdelim(c)   state = 5;
                else            state = next_diagram();
                break;
        case 5: c = nextchar();
                if isdelim(c)   state = 5;
                else            state = 6;
                break;
        case 6: stepback(); state = 0;
                break;  /* empty space, return nothing */
        case 7: c = nextchar();
                if isdigit(c)   state = 8;
                else if issign(c) state = 9;
                else            state = next_diagram();
                break;
        case 8: c = nextchar();
                if isdigit(c)   state = 8;
                else            state = 10;
                break;
        case 9: c = nextchar();
                if isdigit(c)   state = 8;
                else            state = next_diagram();
                break;
        case 10:stepback();
                lex_value = IntAtom(atoi(&buffer[start_pos]));
                return(INTEGER);
                …
        case 52:return(CLOSETEXT);
        case 53:c = nextchar(); return(c);
        }
    }
}
```

Um zu testen, ob das aktuelle Zeichen zu einer Zeichenklasse gehört, werden entsprechende Testfunktionen *isdelim*(*c*), *isdigit*(*c*) usw. benutzt, deren Implementierung nicht schwierig ist.

In manchen Fällen muß zusätzlich zum Token ein Attribut an den Parser übergeben werden (vgl. Abb. 1.3). Da die Funktion *gettoken*() nur einen Wert (das Token) zurückgeben kann, benutzen wir dazu eine globale Variable:

```
int lex_value;
```

Wir benutzen eine Integer-Variable in Anlehnung an *Lex*-generierte Scanner; dort wird zur Übergabe von Attributen zwischen Scanner und Parser ebenfalls stets eine Integer-Variable benutzt (die dort *yylval* heißt). Man kann immer mit Integer-Werten für die Attribute auskommen, da z.B. Adressen mit solchen Werten kompatibel sind oder da man Indizes in eine Symboltabelle übergeben kann. In unserer Anwendung wird ein solches Attribut im Zustand 10 bei Erkennen eines INTEGER-Tokens übergeben. Hier erzeugt die Funktion *IntAtom* einen Knoten (für eine Baumstruktur, die die geschachtelte Liste darstellen soll), der selbst als Index in einen Array von Knoten beschrieben wird, also auch als Integer-Wert. Die C-Funktion *atoi* konvertiert eine Zeichenkette (deren Anfangsadresse sie bekommt) in einen Integer-Wert.

Zu klären ist noch, wie der Übergang von einem Zustandsdiagramm in das nächste gesteuert wird. Dazu müssen wir uns zunächst im Entwurf überlegen, in welcher Reihenfolge die Diagramme untersucht werden müssen. In unserer Anwendung gilt:

- BOOLEAN muß vor SYMBOL untersucht werden, da die Darstellungen von TRUE und FALSE auch der Symbol-Struktur entsprechen.
- OPENTEXT und CLOSETEXT müssen vor SYMBOL untersucht werden, da sonst die Zeichenfolgen „<“ und „</“ als SYMBOL erkannt würden.
- REAL muß vor INTEGER untersucht werden, da die Integer-Struktur ein Präfix der Real-Struktur darstellt, also sonst ein Anfangsstück einer Real-Darstellung als Integer erkannt würde.

Ansonsten sollte die Reihenfolge der Zustandsdiagramme so gewählt werden, daß häufig benutzte Token zuerst kommen. Wir legen die Reihenfolge deshalb wie folgt fest:

Token	Anfangszustand
OPEN	0
CLOSE	2
(Leerraum)	4
BOOLEAN	20
OPENTEXT	35
CLOSETEXT	42
SYMBOL	31

STRING	28
REAL	11
INTEGER	7
(sonstiges Zeichen)	53

Sonstige Zeichen kommen innerhalb von Text-Atomen vor. Die gewählte Reihenfolge wird nun in der Funktion *next_diagram*() beschrieben:

```
int next_diagram()
{
    pos = start_pos;
    switch (start_state) {

        case 0:         start_state = 2; break;
        case 2:         start_state = 4; break;
        case 4:         start_state = 20; break;
        case 20:        start_state = 35; break;
        case 35:        start_state = 42; break;
        case 42:        start_state = 31; break;
        case 31:        start_state = 28; break;
        case 28:        start_state = 11; break;
        case 11:        start_state = 7; break;
        case 7:         start_state = 53;

    }
    return(start_state);
}
```

Die Funktion wird jeweils aufgerufen, wenn innerhalb eines Zustandsdiagramms kein Übergang mehr möglich ist. Dann wird der Eingabezeiger *pos* zurückgesetzt und anhand des gemerkten Anfangszustands für das aktuelle Diagramm der Anfangszustand für das nächste Diagramm gesetzt und als Folgezustand zurückgegeben.

Wir haben in diesem Abschnitt gesehen, wie man auf relativ einfache Art einen Scanner auf der Basis einer Menge von Zustandsdiagrammen implementieren kann; die Zustandsdiagramme werden sequentiell untersucht. Bei dieser Implementierungstechnik braucht man nicht viel nachzudenken. Andererseits gibt es dadurch einige Ineffizienzen, die man mit etwas „Tuning" noch beseitigen könnte. Dazu gibt es z.B. folgende Möglichkeiten:

- Bei Zustandsdiagrammen, die mit unterschiedlichen Zeichen(klassen) beginnen, kann man die Anfangszustände problemlos verschmelzen.
- Endzustände, die nur über eine Kante erreicht werden, kann man weglassen und die entsprechenden Aktionen am vorherigen Zustand ausführen.
- Für TRUE und FALSE braucht man eigentlich keine eigenen Diagramme. Statt dessen schaut man nach Erkennen eines SYMBOL-Tokens in einer Tabelle nach, ob dort das entsprechende Lexem gespeichert ist. Dies ist übrigens die Standardtechnik, mit der in Programmiersprachen Bezeichner und Wortsym-

bole wie *if*, *then*, … unterschieden werden, die ja die gleiche syntaktische Struktur haben. Wortsymbole und bereits bekannte Bezeichner finden sich also in der Tabelle.

- Das Lesen der Zeichen von Text-Atomen ist relativ ungeschickt. Nach Erkennen eines OPENTEXT-Tokens sollte der Scanner in einen Modus übergehen, in dem ohne weitere Prüfung Zeichen an den Parser übertragen werden, solange nicht das Zeichen „<“ oder auch das ganze CLOSETEXT-Lexem auftreten.
- Das Erkennen von INTEGER-Token könnte innerhalb des REAL-Diagramms miterledigt werden.

Andererseits ist es sowieso einfacher, einen lexikalischen Analysator mit *Lex* zu implementieren, was wir im folgenden Abschnitt betrachten.

2.4 Implementierung eines Scanners mit Lex

Das UNIX-Tool *Lex* ist wohl das bekannteste Beispiel für einen Scannergenerator. Man kann damit eine Spezifikation der lexikalischen Analyse in Form regulärer Definitionen angeben und an diese auszuführende Aktionen als C-Programmcode anhängen. Lex benutzt man, wie in Abb. 2.9 gezeigt.

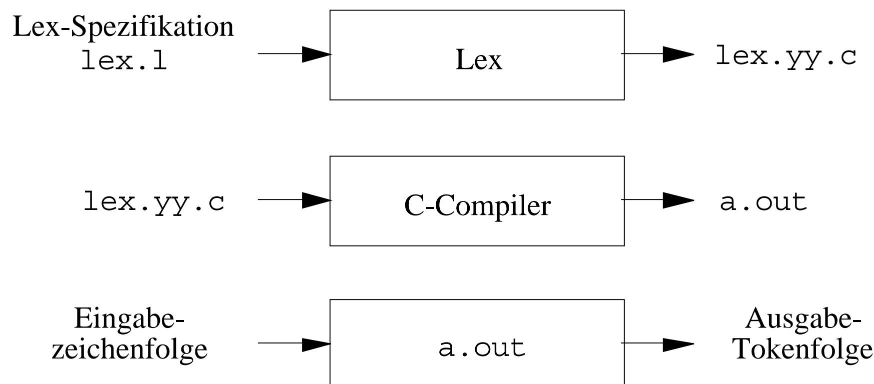


Abb. 2.9. Generierung eines Scanners mit Lex

Das Werkzeug Lex liest also eine Spezifikation aus einer Datei *lex.l* und generiert daraus ein C-Programm *lex.yy.c*. Der C-Compiler überführt dieses in ein ausführbares Programm *a.out*, das den erzeugten Scanner darstellt. Man kann das erzeugte Programm auch mit anderen Programmen zusammenbinden; in diesem Fall steht der Analysator als Funktion mit der Deklaration

```
int yylex()
```

zur Verfügung, die bei jedem Aufruf ein Token liefert. Das Spezifikationsfile *lex.l* hat folgende Struktur:

```
Deklarationen
%%
Tokendefinitionen und Aktionen
%%
Hilfsprozeduren
```

Der *Deklarationsteil* enthält Definitionen von Konstanten als Tokendarstellungen (entsprechend den Definitionen am Anfang von Abschnitt 2.3), diese werden in spezielle Klammern gesetzt und von Lex einfach in das Programm *lex.yy.c* kopiert:

```
%{
#define OPEN    1000
#define CLOSE   1001
...
%}
```

In diesen geklammerten Abschnitt kann man darüber hinaus *#include*-Anweisungen und Variablendeklarationen setzen. Es folgen innerhalb des Deklarationsteils *reguläre Definitionen*, wie in Abschnitt 2.1 eingeführt, z.B.

```
sign            [+-]
digit           [0-9]
```

Der zweite Teil „Tokendefinitionen und Aktionen“ enthält Paare (regulärer Ausdruck, auszuführende Aktion), z.B.

```
{sign}?{digit}+         return(INTEGER);
```

wobei die auszuführende Aktion, wie schon erwähnt, C-Code ist. Falls dort mehrere Anweisungen nötig sind, werden sie eingeklammert (mit geschweiften Klammern, wie in C üblich). Die Zeichenfolge, die zu einem erkannten Token gehört, wird von Lex über zwei Variablen *yytext* und *yyleng* verfügbar gemacht, die deklariert sind als

```
char    *yytext
int     yyleng
```

Dabei zeigt *yytext* auf das erste Zeichen und *yyleng* gibt die Länge an.

Der dritte Teil schließlich enthält zusätzlichen Programmcode, z.B. Hilfsprozeduren, die innerhalb der Aktionen benötigt werden. Auch dieser Teil wird von Lex wieder nach *lex.yy.c* kopiert.

Es bleibt zu erklären, wie hier reguläre Ausdrücke notiert werden können. Das wesentliche Problem ist die Unterscheidung zwischen Zeichen und „Metazeichen“. Zunächst einmal steht jedes Zeichen für sich selbst mit Ausnahme der folgenden Zeichen:

```
. $ ^ [ ] - ? * + | ( ) / { } < > " \
```

Wenn man diese als normale Zeichen benutzen will, müssen sie in Doppelanführungsstriche gesetzt werden. Die Bedeutung der meisten dieser Zeichen als Metazeichen kennen wir schon aus Abschnitt 2.1:

```
[ ]     Zeichenklassen          [aby&]
-       Bereich                 [A-Z]
^       Komplement              [^0-9]
?       optional                [+-]?
|       Alternative             a|bc
+       ein- oder mehrmals      {digit}+
*       0 oder mehrmals         {letter}*
( )     normale Klammerung      (a|bc)*
```

Konkatenation wird auch hier durch schlichtes Hintereinanderschreiben notiert. Die übrigen Zeichen haben folgende Bedeutung:

```
.       alles ausser \n
```

Der Punkt paßt auf jedes einzelne Zeichen mit Ausnahme des Zeilenendesymbols (dargestellt durch \n).

```
^ $     Ausdruck an Zeilenanfang oder -ende     ^hallo  \t$
```

Der Ausdruck „^hallo“ paßt auf das Wort „hallo“, wenn es am Anfang einer Zeile steht. Der zweite Ausdruck „\t$“ paßt auf ein Tabulatorsymbol am Ende einer Zeile. Das „^“-Zeichen hat hier eine zweite Bedeutung, für Komplement steht es nur innerhalb eckiger Klammern.

```
/       Vorausschau     -/{digit}+
```

Manchmal möchte man eine Zeichenfolge nur dann als ein bestimmtes Token erkennen, wenn eine andere Zeichenfolge folgt. Der obige Ausdruck paßt auf das Zeichen „–“ genau dann, wenn sich eine Ziffernfolge anschließt. Die Ziffernfolge selbst wird aber nicht ins Token eingeschlossen.

```
{ }     Kennzeichnen eines regulären Symbols    {digit}
```

Bereits definierte Symbole können so eingeklammert werden, um sie von der direkten Bedeutung als Zeichenfolge zu unterscheiden. Denn

```
digit
```

steht für die Zeichenfolge „digit“ und nichts sonst. Schließlich dienen

```
" \     Escape-Zeichen          "..."   \"
```

dazu, den Metazeichen ihre normale Bedeutung zu geben, wobei \ sich auf genau ein folgendes Zeichen bezieht ("..." bezeichnet also eine Folge von drei Punkten, \" einen Doppelanführungsstrich). – Auf die Erklärung der Bedeutung der spitzen Klammern wollen wir hier verzichten.

Die Lex-Spezifikation für unsere Anwendung aus den Abschnitten 2.2–2.3 (geschachtelte Listen) sieht dann so aus:

```
%{
#define OPEN            1000
#define CLOSE           1001
/* usw. wie oben */
#include "NestedList.h"
#define TRUE            1
#define FALSE           0
%}
sign            [+-]
letter          [A-Za-z]
digit           [0-9]
digits          [0-9]+
symchar         [^A-Za-z0-9 \t\n()\"]
emptyspace      [ \t\n]+
string          [^\"]*

%%
(               return(OPEN);
)               return(CLOSE);
{emptyspace}    ;
{sign}?{digits} {yylval = IntAtom(atoi(yytext));
                return(INTEGER);}
{sign}?{digits}"."{digits}?(E{sign}?{digits})?
                {yylval = RealAtom(atof(yytext));
                return(REAL);}
TRUE            {yylval = BoolAtom(TRUE); return(BOOL);}
FALSE           {yylval = BoolAtom(FALSE); return(BOOL);}
\"{string}\"    { if (yyleng-2 > 48) /* handle error */
                  else { s = strncpy(s, &yytext[1], yyleng-2);
                         yylval = StringAtom(s);
                         return(STRING);}}
({letter}({letter}|{digit})*)|symchar+
                { if (yyleng > 48) /* handle error */
                  else { s = strncpy(s, yytext, yyleng);
                         yylval = SymbolAtom(s);
                         return(SYMBOL);}}
"<text>"        return(OPENTEXT);
"</text--->"    return(CLOSETEXT);
.               return(yytext[0]);

%%
char buffer[50]; char *s;
s = &buffer;
```

Ein Lex-generierter Analysator akzeptiert bei Mehrdeutigkeiten stets eine Zeichenfolge maximaler Länge. Es schadet also nichts, daß die INTEGER-Beschreibung ein Präfix der REAL-Beschreibung ist.

Einige kurze Erklärungen zur obigen Spezifikation:

- Die C-Funktionen *atoi* bzw. *atof* erzeugen aus einer Zeichenkette einen Integer-Wert bzw. einen Wert des C-Typs *double* (der Gleitkommazahlen darstellt).

Diese Funktionen lesen von selbst nur so viele Zeichen, wie zur Zahlendarstellung gehören, deshalb braucht man *yyleng* nicht anzugeben.

- Die Funktionen *StringAtom* bzw. *SymbolAtom* erwarten als Argumente '0C'-terminierte Strings, deshalb muß vorher einmal kopiert werden; die Bibliotheksfunktion *strncpy* kopiert gerade so viele Zeichen, wie im letzten Parameter angegeben, und fügt dann ein '0C'-Zeichen an. Bei Stringatomen werden die Doppelanführungsstriche selbst nicht mitkopiert.

Wenn man den Aufwand für die – noch nicht einmal vollständig gezeigte – Handimplementierung aus den Abschnitten 2.2–2.3 mit dem für die Lex-Implementierung vergleicht, dann wird deutlich, wie sehr es sich lohnt, das Werkzeug Lex zu kennen und zu benutzen.

Die Implementierung eines Scannergenerators folgt prinzipiell der aus der Theorie bekannten Strategie:

- Erzeugen eines nichtdeterministischen endlichen Automaten (NEA) für jede Tokendefinition,
- Vereinigung der Automaten zu einem einzigen NEA (mit ε-Übergängen kein Problem),
- Konstruktion des entsprechenden DEA,
- Darstellung des DEA als große Tabelle,
- Simulation des DEA durch eine Prozedur, die jeden Übergang durch Nachschlagen in der Tabelle realisiert.

Natürlich sind dabei noch manche Details zu klären, u.a. im Hinblick auf Effizienz (z.B. kompakte Darstellung der Tabelle), auf die wir hier aber nicht weiter eingehen wollen.

Selbsttestaufgabe 2.4: Das Unix-Dienstprogramm *wc* (*w*ord *c*ount) zählt die Zeilen, Worte und Zeichen einer Datei oder der Standardeingabe. Geben Sie eine Lex-Spezifikation an, mit der sich ein Wortzählprogramm ähnlich *wc* generieren läßt.

Hinweis: Bei dieser Problemstellung wird Lex etwas anders benutzt, als oben besprochen, da nicht für jedes erkannte lexikalische Symbol ein einzelnes Token zurückgeliefert wird. Statt dessen wird die Funktion *yylex*() nur einmal aufgerufen und terminiert erst, wenn die Eingabe erschöpft ist. Man erreicht das, indem in den semantischen Aktionen keine **return**-Anweisungen eingefügt werden. □

2.5 Literaturhinweise

Wir sind in der Darstellung in diesem Kapitel davon ausgegangen, daß Ihnen der formale Hintergrund der lexikalischen Analyse bereits bekannt ist (etwa aus einem Theoriekurs) . Deshalb wurden die Grundbegriffe und Definitionen nur kurz wiederholt und die Schritte bei der Konstruktion eines Scanners aus regulären Ausdrücken nur stichwortartig aufgezählt. Diesen formalen Hintergrund kann man z.B. in (Sud-

kamp 2005) nachlesen. Das klassische Buch zu diesem Thema ist (Hopcroft, Motwani und Ullman 2007).

Die direkte Implementierung eines Scanners auf der Basis von Zustandsdiagrammen wird z.B. in (Aho et al. 2006), (Alblas und Nymeyer 1996) und (Parsons 1992) gezeigt. Diese Bücher beschreiben auch aus Compilerbau-Sicht die vollständige Vorgehensweise bei der Konstruktion eines Scanners aus regulären Spezifikationen, also die Basis der Implementierung von Lex. Natürlich wird dies auch in vielen anderen Büchern zum Übersetzerbau behandelt.

Endliche Automaten wurden zuerst von McCulloch und Pitts (1943) als Modell für Nervenaktivität verwendet; Kleene (1956) führte reguläre Ausdrücke ein und zeigte die Äquivalenz mit den von endlichen Automaten akzeptierten Sprachen. Eine frühe Arbeit zur Generierung von Scannern aus regulären Ausdrücken ist (Johnson *et al.* 1968).

Der Scannergenerator Lex stammt von Lesk (1975). Als Teil des UNIX-Systems wird er noch immer weithin verwendet. Beschreibungen zur Benutzung findet man in (Levine, Mason und Brown 1992) oder in UNIX-Systemhandbüchern. Ein ähnlicher Public-Domain Scannergenerator namens Flex stammt von Paxson (1995) und ist Teil der LINUX-Distribution. Andere Beispiele für Scannergeneratoren sind Alex (Mössenböck 1986) und Rex (Grosch 1989). Auch das System Eli, ein modularer „Werkzeugkasten" für den Compilerbau (Kastens und Waite 1994) enthält einen Scannergenerator. Nähere Informationen zu Eli findet man unter `http://www.cs.colorado.edu/~eliuser/`.

Während generierte Scanner i.allg. tabellengesteuert arbeiten, ist es auch möglich, aus Beschreibungen von Zustandsdiagrammen (endlichen Automaten) direkt ausführbaren Code zu erzeugen, mit case-Anweisungen wie in der Handimplementierung. In (Gray 1988) wird gezeigt, daß auf diese Art sehr effizient arbeitende Scanner zu erhalten sind.

Lösungen zu den Selbsttestaufgaben

Aufgabe 2.1

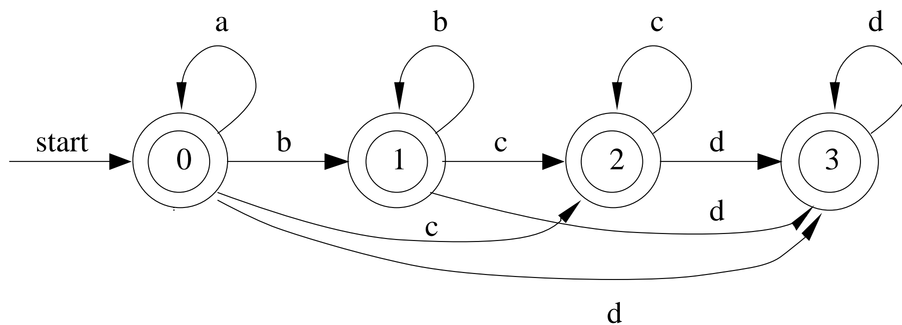
(a) $a^* b^* c^* d^*$

(b) $0^* (1 \mid \varepsilon) 0^*$

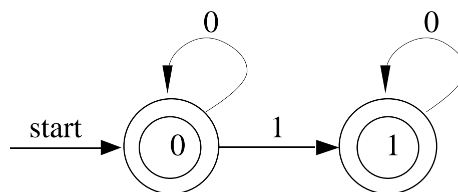
(c) $0^* 1 (0 \mid 1)^*$

Aufgabe 2.2

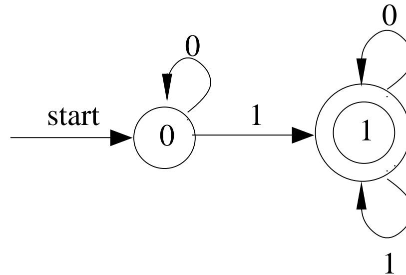
(a) $Q = \{0, 1, 2, 3\}$; $\Sigma = \{a, b, c, d\}$; $s = 0$; $F = Q$



(b) $Q = \{0, 1\}$; $\Sigma = \{0, 1\}$; $s = 0$; $F = \{0, 1\}$

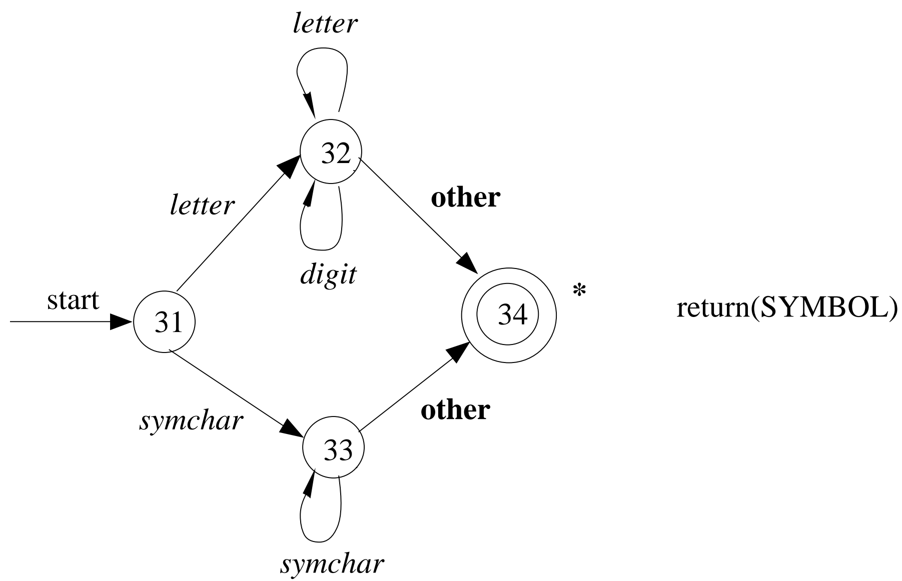


(c) $Q = \{0, 1\}$; $\Sigma = \{0, 1\}$; $s = 0$; $F = \{1\}$

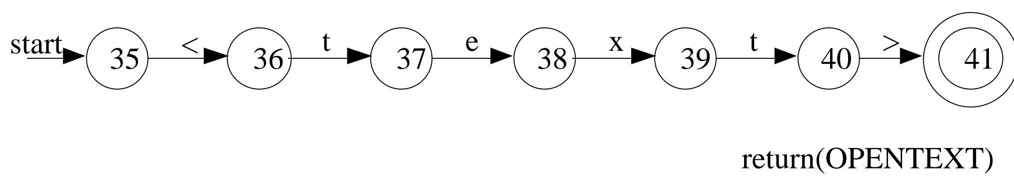


Aufgabe 2.3

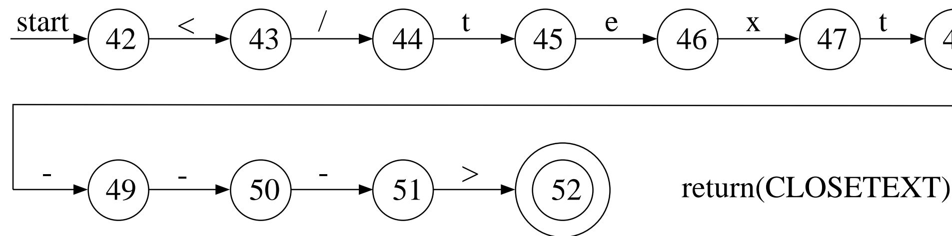
Zustandsdiagramm für SYMBOL-Token:



Zustandsdiagramm für OPENTEXT-Token:



Zustandsdiagramm für CLOSETEXT-Token:



Aufgabe 2.4

Eine Lex-Spezifikation für ein Wortzählprogramm kann beispielsweise wie folgt aussehen:

```
%{
   unsigned lineCount = 0, wordCount = 0, charCount = 0;
%}

word    [^ \t\n]+
eol     \n

%%
{word}  { wordCount++; charCount += yyleng; }
{eol}   { charCount++; lineCount++; }
.       { charCount++; }

%%
main()
{
   yylex();
   printf("%8d%8d%8d\n", lineCount, wordCount, charCount);
}
```

Der Deklarationsteil, der wortwörtlich in die Datei *lex.yy.c* kopiert wird, enthält zum einen drei Variablen, die im Programm benötigt werden und die die Anzahl der Zeilen, Worte und Zeichen festhalten. Im Anschluß daran folgen zwei reguläre Definitionen. Die erste Definition beschreibt ein Wort als eine nichtleere Kombination von Zeichen mit Ausnahme des Leerzeichens, des Tabulators und des Zeilenendezeichens. Die zweite Definition beschreibt das Zeilenendezeichen.

Der zweite Teil enthält Tokendefinitionen und Aktionen. In diesem Fall werden allerdings keine neuen Token definiert, sondern es wird Bezug auf die regulären Symbole des Deklarationsteils genommen. Lex ersetzt die Token innerhalb geschweifter Klammern durch die tatsächlichen regulären Ausdrücke im Deklarationsteil. Wird ein vollständiges Wort erkannt, so wird als Aktion die Anzahl der Worte um eins und die Anzahl der Zeichen um *yyleng* erhöht. Die Variable *yyleng* gibt die Länge der Zeichenfolge an, die zu einem erkannten Token gehört. Wird ein

Zeilenendezeichen erkannt, so wird die Anzahl der Zeichen und der Zeilen jeweils um eins erhöht. Bei jedem anderen Zeichen (dies kann nur ein Leerzeichen oder ein Tabulator sein!) wird die Anzahl der Zeichen um eins erhöht.

Der dritte Teil der Lex-Spezifikation, der ebenfalls wortwörtlich in die Datei *lex.yy.c* kopiert wird, beinhaltet Hilfsprozeduren (C-Quelltext). In diesem Fall ist es die Hauptroutine *main*, die zunächst den Analysator *yylex*() aufruft und dann das Ergebnis ausgibt.

Im folgenden wird gezeigt, wie auf einer Unix-Maschine der Scanner erzeugt und aufgerufen wird. Die obige Lex-Spezifikation steht in der Datei *my_wc.l*. Eine Datei *text* dient als Eingabe und hat diesen Absatz zum Inhalt. Eingegebene Kommandos werden fett und Ausgaben normal gedruckt.

```
% lex my_wc.l
% cc lex.yy.c -o my_wc -ll
% my_wc < text
       4      42     284
%
```

Literatur

Aho, A.V., Lam, M.S., Sethi, R. und Ullman, J.D. (2006). *Compilers: Principles, Techniques, and Tools*. 2nd Edition, Addison-Wesley, Reading, MA.

Aho, A.V., Lam, M.S., Sethi, R. und Ullman, J.D. (2008). *Compiler: Prinzipien, Techniken und Werkzeuge*. 2. Auflage, Pearson Studium.

Alblas, H. und Nymeyer, A. (1996). *Practice and Principles of Compiler Building with C*. Prentice-Hall International, London, UK.

Appel, A. und Ginsburg, M. (1997). *Modern Compiler Implementation in C: Basic Techniques.* Cambridge University Press, Cambridge, UK.

Bauer, B. und Höllerer, R. (1998). *Übersetzung objektorientierter Programmiersprachen.* Springer-Verlag, Berlin.

Doberkat, E.E. und Fox, D. (1990). *Praktischer Übersetzerbau*. Teubner-Verlag, Stuttgart.

Gray, R.W. (1988). γ–GLA – A Generator for Lexical Analyzers that Programmers Can Use. *USENIX Conf. Proceedings*. USENIX Association, Berkeley, CA, S. 147-160.

Grosch, J. (1989). Efficient Generation of Lexical Analyzers. *Software – Practice and Experience 19*, 1089-1103.

Holmes, J. (1995a). *Object-Oriented Compiler Construction*. Prentice-Hall International, London, UK.

Holmes, J. (1995b). *Building Your Own Compiler with C++*. Prentice Hall, Englewood Cliffs, NJ.

Holub, A.I. (1990). *Compiler Design in C*. Prentice-Hall International, London, UK.

Hopcroft, J.E., Motwani, R., und Ullman, J.D. (2007). *Introduction to Automata Theory, Languages, and Computation*. 3rd Edition, Addison-Wesley, Reading, MA.

Johnson, W.L., Porter, J.H., Ackley, S.I., und Ross, D.T. (1968). Automatic Generation of Efficient Lexical Processors Using Finite State Techniques. *Communications of the ACM 11*, 805-813.

Kastens, U. (1990). *Übersetzerbau*. Oldenbourg-Verlag, München.

Kastens, U. und Waite, W.M. (1994). Modularity and Reusability in Attribute Grammars. *Acta Informatica 31* (7), 601-627.

Kernighan, B.W., und Ritchie, D.M. (1990). *Programmieren in C*. Zweite Ausgabe ANSI C. Hanser-Verlag, München und Prentice-Hall International, London, UK.

Kleene, S.C. (1956). Representation of Events in Nerve Nets and Finite Automata. In: Shannon, C.E. und McCarthy, J. (Hrsg.), *Automata Studies*. Princeton University Press, Princeton, NJ, S. 3-42.

Lamport, L. (1994). *LATEX. A Document Preparation System*. 2nd Edition, Addison-Wesley, Reading, MA.

Lesk, M.E. (1975). Lex – A Lexical Analyzer Generator. Technical Report 39, AT & T Bell Laboratories, Murray Hill, NJ.

Levine, J.R., Mason, T. und Brown, D. (1992). *lex & yacc*. 2nd Edition, O'Reilly & Associates, Sebastopol.

McCulloch, W.S. und Pitts, W. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics 5*, 115-133.

Mössenböck, H. (1986). Alex – A Simple and Efficient Scanner Generator. *ACM SIGPLAN Notices 21*, 139-148.

Parsons, T.W. (1992). *Introduction to Compiler Construction*. W.H.Freeman & Co Ltd.

Paxson, V. (1995). Flex – Fast Lexical Analyzer Generator. Lawrence Berkeley Laboratory, Berkeley, CA, `ftp://ftp.ee.lbl.gov/flex-2.5.4.tar.gz`

Pittman, T. und Peters, J. (1992). *The Art of Compiler Design*. Prentice-Hall International, London.

Sudkamp, T. A. (2005). *Languages and Machines*: An Introduction to the Theory of Computer Science. 3rd Edition, Addison-Wesley, Reading, MA.

Waite, W.M. und Goos, G. (1984). *Compiler Construction*. Springer-Verlag, Berlin.

Wilhelm, R. und Maurer, D. (2007). *Übersetzerbau: Theorie, Konstruktion, Generierung*. 2. Aufl., Springer-Verlag, Berlin.

Wirth, N. (2011). *Grundlagen und Techniken des Compilerbaus*. 3. Aufl., Oldenbourg.

Index

A
Alphabet 20
Analyse 7
Assembler 12
attributierte Grammatik 17
B
Binder 13
C
Codeerzeugung 10
Codeoptimierung 9
Compiler 1
D
deterministischer endlicher Automat 23
F
formale Sprache 20
G
Generator 15
Grammatik 5
I
Interpretation 13
Interpreter 13
K
Konkatenation 20
L
Lader 13
leeres Wort 20
Lex 16, 33
Lexem 22
lexikalische Analyse 4, 19
link editor 13
M
Makro 12
P
Parser 15
Präprozessor 12
R
reguläre Definition 21
reguläre Sprache 19, 21
regulärer Ausdruck 5, 19, 21
reguläres Symbol 21
S
Scanner 15
semantische Analyse 6
semantische Regel 17
Symboltabelle 3
Syntax 1
Syntaxanalyse 5
Syntaxbaum 6
T
Token 5, 22
type casting 7
type checking 6
Typüberprüfung 6
U
Übersetzer 1
Übersetzungsphase 3
W
Werkzeug 15
Wort 20
Y
Yacc 16
Z
Zeichenklasse 22
Zustandsdiagramm 23
Zwischencode 7