Aufbau und Funktionsweisen von Compilern

1×1 des Compilerbaus

***Dr. Dietmar Nentwig, Christian Becker* • Üblicherweise stellt die Kompilierung von Programmen für Anwendungen, in denen Programme beliebig häufig ausgeführt werden müssen, die wirtschaftlichste Lösung dar. Was hinter der Aufgabe steckt, ein lauffähiges Programm zu erzeugen, und wie moderne Entwicklungstools diese Aufgabe lösen, zeigen wir in diesem Artikel.**

Das PC-Lexikon (aus der Doppelklick-Reihe von Rowohlt-Systhema) sagt zum Begriff Compiler folgendes: "Bezeichnung für ein Übersetzungsprogramm, das in der Lage ist, aus einem Quellprogramm ein speicherfähiges Objektprogramm zu übersetzen." Eine andere Definition ist in diesem Zusammenhang die Abgrenzung zu einem verwandten Begriff, dem Interpreter. Sowohl Compiler als auch Interpreter sind Programme, die andere Programme als Eingabe verarbeiten. Interpreter arbeiten dabei das Eingabeprogramm Schritt für Schritt ab und liefern als Ausgabe die gewünschten Aktionen der jeweils dahinterstehenden Befehle des Eingabeprogramms. Dazu führt der Interpreter jeden Befehl des Eingabeprogramms dadurch aus, indem er die zu dem Befehl gehörenden Unterprogramme (des Interpreters) aufruft. Diese Unterprogramme sind in der Regel in der für die Maschine, auf dem der Interpreter abläuft, bestimmten Maschinensprache abgefaßt. Diese schrittweise Abarbeitung des Eingabeprogramms durch den Interpreter erfolgt so lange, bis das Programm normal endet oder der Anwender die Ausführung abbricht.
Eine interpretierte Computersprache, die auch heute noch mit dem Betriebssystem MS-DOS beziehungsweise PC-DOS ausgeliefert wird, ist Basic (beziehungsweise QBasic), der "Beginners' All Purpose Symbolic Instruction Code". Daneben existieren aber auch Compiler zu Basic.
Im Gegensatz zu einem Interpreter überführt ein Compiler den gesamten Quellcode des Eingabeprogramms in ein Maschinenspracheprogramm (Objektcode). Dazu wird das Eingabeprogramm unter verschiedenen Gesichtspunkten vom Compiler analysiert und entsprechend umgewandelt. Das aus diesem Prozeß entstandene Maschinenprogramm kann dann auf der Zielmaschine direkt ausgeführt werden und benötigt dazu keine weiteren Hilfsprogramme.

■ Compiler und Interpreter

Anhand des folgenden kleinen Basic-Programms wird der Unterschied zwischen einem Interpreter und einem Compiler deutlich.

```
10 FOR Lauf=1 TO 1000
20 PRINT 'Testausgabe'
30 NEXT Lauf
```

Die drei Zeilen beginnen allesamt mit einer Ziffer, die eine Zeilennumerierung darstellt. Zeile 10 legt den Rahmen für eine Schleife fest. In diesem Fall soll der Befehl aus der folgenden Zeile (20) 1000mal ausgeführt werden. Zeile 30 bestimmt das Ende der Schleife.
Der Interpreter liest zunächst aus der ersten Zeile das Wort `FOR` (beziehungsweise eine interne Darstellung des Wortes FOR, Token genannt). Danach sucht er in einer Tabelle nach der Einsprungadresse für das zum Schleifenkonstrukt gehörende Unterprogramm. Dieses Unterprogramm legt die Anfangs- und Endwerte sowie den aktuellen Stand der Schleifenvariable `Lauf` für die FOR-Schleife an den entsprechenden Stellen ab. Nachdem das Unterprogramm abgearbeitet ist, liest der Interpreter die zweite Programmzeile ein und verfährt nach dem gleichen Schema wie in der ersten Zeile. In der dritten Programmzeile wird vom Interpreter nach dem Einlesen des Wortes `NEXT` das entsprechende Unterprogramm aufgerufen und die Laufvariable Lauf hochgezählt. Ist der in der ersten Zeile definierte Endwert erreicht, wird die Ausführung der Schleife an dieser Stelle vom Interpreter beendet. Ansonsten wird wieder die erste Zeile eingelesen und wie bisher beschrieben verfahren.

Es wird deutlich, daß das Interpretieren von Programmen eine recht zeitaufwendige Angelegenheit ist. Ein Compiler würde das gleiche Programm auf ähnliche Weise zunächst Zeile für Zeile analysieren, dann aber nicht die Unterprogramme aufrufen und direkt ausführen. Er erzeugt aus dem Eingabeprogramm eine "Zusammenstellung" der entsprechenden Unterprogramme in Maschinensprache. Diese Zusammenstellung ist dann das übersetzte Programm oder auch der sogenannte Objektcode, der direkt auf der Maschine ausgeführt werden kann. Sämtliche Zeiten für das Lesen in Interpreter-Tabellen und das Suchen des zugehörigen Unterprogramms fallen weg. Die Programme laufen dadurch erheblich schneller ab.

■ Der Compilerlauf

Bild 1 veranschaulicht die Funktionsweise eines Compilerlaufs. Der Compiler liest das Quellprogramm und erzeugt daraus das Objektprogramm. Der Kompilationsvorgang läßt sich in fünf Phasen aufteilen: die lexikalische Analyse, die Syntaxanalyse, die semantische Analyse, die Optimierung und zuletzt die Codeerzeugung. Diese Phasen werden für bestimmte Compiler und Sprachen zu sogenannten Passes zusammengefaßt. Ein Pass liest dann entweder das Quellprogramm oder das vom vorhergehenden Pass erzeugte Zwischenfile. Innerhalb des. Pass werden die Pass-spezifischen Arbeitsphasen (syntaktische Analyse und so weiter) ausgeführt und eine neue Zwischendatei oder das Objektprogramm geschrieben.
Ein solches Zwischenfile enthält das in eine jeweils andere "Zwischensprache" transformierte Quellprogramm. Üblicherweise liefert die syntaktische Analyse als Ausgabe eine Art Baumstruktur, in der die Struktur des Quellprogramms in der sogenannten umgekehrten polnischen Notation (zuerst die Operanden, dann die zur Verknüpfung benötigten Operatoren) umgewandelt und gespeichert ist.
Die Anzahl der für die Kompilation benötigten Passes hängt zum einen von dem zur Verfügung stehenden Speicher (in der Regel heute kein Problem mehr) und zum anderen von der Programmiersprache des Quellprogramms ab. Je mehr Speicher zur Verfügung steht, desto mehr Programmteile für unterschiedliche Arbeitsphasen können gleichzeitig bearbeitet und müssen nicht auf externe Datenträger ausgelagert werden. Alte Programmiersprachen wie Algol oder PL/1 benötigen mindestens zwei Passes, da es aufgrund ihrer Deklaration erlaubt ist, Bezeichner für Variablen erst nach ihrer ersten Verwendung zu definieren. Der Compiler kann also zu dem Zeitpunkt des Auftauchens des Bezeichners nicht wissen, ob ein syntaktischer Fehler vorliegt oder ob die Deklaration noch später erfolgen wird.

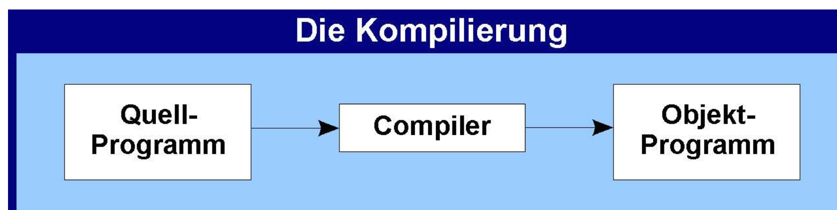

Bild 1. Arbeitsphasen bei der Kompilation

■ Verschränktes Arbeiten

Für Programmiersprachen wie C und Pascal ist dagegen theoretisch nur ein Pass notwendig, da grundsätzlich vor der Verwendung eines Bezeichners seine Deklaration erfolgt sein muß. Bei den dazugehörigen Compilern wird allerdings nicht nacheinander, sondern ineinander verschränkt gearbeitet. Die Syntaxanalyse steuert dabei sozusagen den gesamten Compiler. Zur Analyse eines Programmabschnitts ruft die syntaktische Analyse so oft wie für den Programmabschnitt nötig die lexikalische Analyse auf. Ist der Abschnitt vollständig analysiert, so wird von der Syntaxanalyse die semantische Analyse angestoßen. Abschließend werden für den aktuell bearbeiteten Programmteil noch eine Optimierung (falls möglich) sowie die Codeerzeugung durchgeführt. Die syntaktische Analyse übernimmt wieder die Kontrolle zur Analyse des anschließenden Programmteils. Dies wird bis zum Ende des Quellprogramms durchgeführt (Bild 2).
Für die Kompilation werden von den einzelnen Passes umfangreiche Tabellen angelegt, die allen anderen Passes zur Analyse und gemeinsamen Nutzung zur Verfügung stehen. Dort werden in der Hauptsache die im Quelltext verwendeten Bezeichner und deren interne Darstellung abgelegt. Mit diesen Bezeichnern sind Variablennamen, Prozedur- und Funktionsnamen, Datentypnamen und Sprungmarken gemeint.
Bei der Codeoptimierung wird noch zwischen einer globalen und einer lokalen Optimierung unterschieden. Die globale Optimierung bezieht sich in der Regel auf die Optimierung von Schleifenkonstrukten und ähnlichem. Sie wird üblicherweise vor der Codeerzeugung durchgeführt. Die lokalen Optimierungen erfolgen anschließend an die Codeerzeugung. Ein Beispiel für eine lokale Optimierung ist das Vereinfachen von arithmetischen Ausdrücken.
Compiler werden heute fast ausschließlich mit sich selbst generiert. Das bedeutet, daß zum Beispiel der Visual C++-Compiler in der aktuellen Version mit der Vorgängerversion desselben Compilers erzeugt wurde. Dann existieren sogenannte Compiler-Compiler. Das sind Programme, die aus bestimmten Vorgaben selbständig einen Compiler generieren. Diese Vorgaben beschreiben dann einerseits die Computersprache, für die der Compiler übersetzen soll, und andererseits die Zielmaschine, für die der zu erstellende Compiler übersetzen soll.

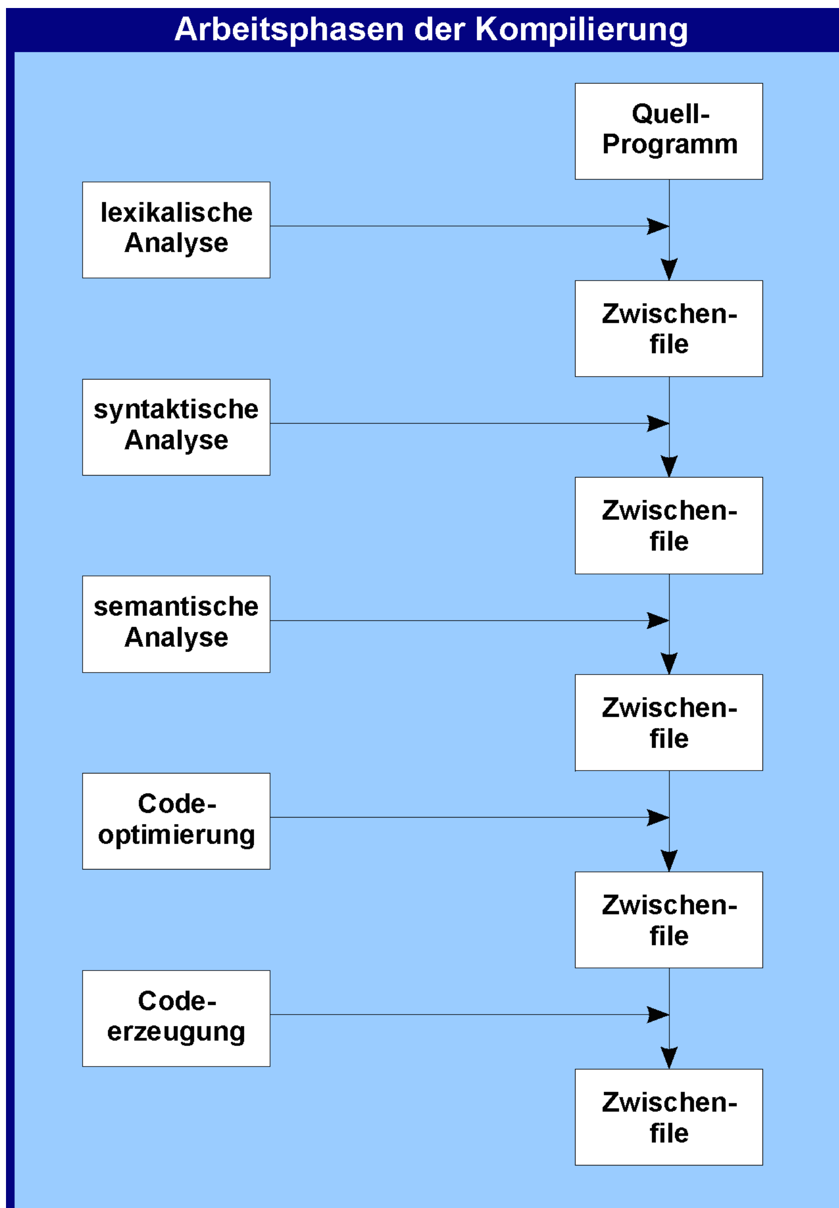

Bild 2. Nach jeder Arbeitsphase wird ein Zwischenfile erzeugt.

■ Lexikalische Analyse

Der Teil des Compilers, der die lexikalische Analyse des Quellprogramms durchführt, wird Scanner genannt. Der Scanvorgang zerlegt den Text des Quellprogramms in seine einzelnen Bestandteile wie Befehlswörter, Bezeichner für Variablen, Prozeduren und Funktionen sowie Operatoren (+, − und so weiter). Jeder vom Scanner erkannte Bestandteil wird vom Scanner entsprechend aufbereitet und in eine andere, für nachfolgende Analysestufen besser zu verarbeitende Form gebracht. Der Scanner wandelt dabei Befehlsworte, Bezeichner und Zahlen in interne Darstellungsformate um.
Dazu wird der Quelltext Zeichen für Zeichen eingelesen, analysiert und in seine Bestandteile zerlegt. Das Programmfragment `while A < 10 do` wird dabei in "`while | A[IC1] | < | 10 | do`" aufgeteilt und in die internen Darstellungsformen umgewandelt.
Einige Compiler führen die lexikalische zusammen mit der syntaktischen Analyse durch. Der Programmteil für die Syntaxanalyse fordert dann vom Scanner immer den nächsten zu analysierenden Bestandteil. Auf diese Weise wird eine Speicherung in einer Zwischendatei gespart und eine Geschwindigkeitssteigerung realisiert.

■ Syntax und Semantik

Der englischsprachige Begriff für die Analyse der Syntax lautet “parsing”, weshalb der Programmteil zur Analyse der Syntax Parser genannt wird. Das Rowohlt-Systhema PC-Lexikon bezeichnet Syntax “als die Gesamtheit der formalen Regeln einer Sprache, die sich vor allem auf die Art der verwendeten Zeichen, der Bildung von Zeichenkombinationen und so weiter beziehen (Information). Die Syntax einer Programmiersprache legt die Regeln über die Bildung und Verwendung der programmiersprachlichen Formulierungen fest, an die sich der Programmierer zu halten hat. Jeder Verstoß gegen diese Regeln führt zu einem Formfehler des Programms, der aber durch das Übersetzungsprogramm angezeigt wird, weil bei der Übersetzung eine vollständige Syntaxüberprüfung des Programms durchgeführt wird.” Im Unterschied dazu steht die Semantik, die in einer Sprache “die Beziehung zwischen dem Sprachzeichen, dem Wort und seinem Bedeutungsgehalt darstellt. Dies sind die Beziehungen zur realen Welt, die in den Bedeutungen begrifflich gefaßt werden. Bei Programmiersprachen ist das die Beziehung zwischen dem Befehlswort und der Elementarfunktion, die der Computer ausführen kann.”
Diese doch eher abstrakte Definition kann an folgendem Beispiel verdeutlicht werden: “Die Katze schreibt Bücher.” Grammatikalisch, also syntaktisch ist der Satz korrekt. Er ergibt jedoch keinen Sinn, was nicht am Aufbau des Satzes liegt, sondern an seiner Bedeutung, die ihm durch die Semantik der deutschen Sprache zugeordnet ist. Der Satz kann also als semantisch falsch angesehen werden. Ein weiteres, mehr formales Beispiel, ist die Definition zur Bildung von Binärzahlen. Unter Binärzahlen werden dabei Konstrukte wie 0, 1, 010, 011, 1101 verstanden. Zunächst sind diese Abfolgen von Nullen und Einsen rein syntaktische Konstrukte, denen eine Bedeutung fehlt. Man könnte theoretisch der Null den Wert 1 zuweisen und der Eins den Wert 0, dies entspräche jedoch nicht den Gepflogenheiten.
Für das Beispiel, in dem die Syntax und Semantik von Binärzahlen genau definiert werden, bleiben aber die gewohnten Bedeutungen der Null und der Eins erhalten. So können die Definitionen der Syntax und der Semantik an folgendem Binärzahlenbeispiel diskutiert werden.

- Syntaxdefinition:
  1. Die Zeichen 0 und 1 sind Binärzahlen.
  2. Ist b eine Binärzahl, so sind auch b0, 0b, bl, 1b Binärzahlen.
- Semantikdefinition:
  1. Das Zeichen “0” stellt den Wert 0, das Zeichen “1” den Wert 1 dar.
  2. Stellt n als Binärzahl den Wert x dar, so erhält n0 den Wert 2 * x und n1 den Wert 2 * x + 1.

An diesem Beispiel wird deutlich, wie eng Syntax und Semantik miteinander verknüpft sein können. Hier hat sich die Definition der Semantik stark am induktiven Aufbau der Syntax orientiert. Es wird daran auch deutlich, daß es oft nicht wünschenswert ist, die Definition der Syntax von der Definition der Semantik vollständig zu trennen. Insbesondere bei der Programmierung sollen durch Elemente der Programmiersprache die Lesbarkeit und das Verständnis für das in der Programmiersprache geschriebene Programm erhöht werden. Man versucht dabei mit Mitteln der Syntax, die Bedeutung des Programms (Semantik) hervorzuheben.
Das Fragment `read(a); write(3/a);` aus einem PASCAL-Programm verdeutlicht die Schwierigkeit, an dieser Stelle zu entscheiden, ob das Programm semantisch oder syntaktisch falsch ist, wenn für “a” vom Anwender der Wert 0 eingegeben wird. Der undefinierte Wert 3/0 müßte eigentlich ausgegeben werden, ist jedoch nicht definiert.

■ Formale Definition

Es existieren heute viele Verfahren, um einen Parser für eine Programmiersprache automatisch erstellen zu lassen. Dem UNIX-vertrauten Leser dürfte das Tool YACC (Yet Another Compiler Compiler) ein Begriff sein. Mit diesem Tool wird aus einem Eingabefile, der die formale Beschreibung einer Sprachsyntax enthält, der Quellcode für einen Parser derselben Sprache erzeugt. Diese Eingabedatei muß jedoch einer bestimmten formalen Schreibweise genügen. Für die Festlegung eines Formalismus zur Syntaxbeschreibung von Programmiersprachen ist es zunächst wichtig, sich zu verdeutlichen, aus welchen Elementen eine Programmiersprache überhaupt besteht. In der theoretischen Informatik gibt es dazu mehrere Definitionen. Eine Definition wurde von Chomsky vorgenommen. Eine Sprache ist ein n-Tupel (n=4, auch Quadrupel genannt), welches aus 3 Mengen und einem speziellen Symbol besteht: (T, N, P, S).

- Die Menge “T” wird im allgemeinen als das Alphabet der Sprache bezeichnet. Die Elemente der Menge werden terminale Symbole genannt. Diese Menge schließt also die Grundeinheiten einer jeden Sprache ein. Das können zum einen Buchstaben des deutschen Alphabets, Ziffern und alle weiteren Satz- und Sonderzeichen sein, zum anderen aber auch – speziell auf Programmiersprachen bezogen – die Kommando beziehungsweise Befehlswörter der betreffenden Programmiersprache (`while`, `for` und so weiter). Die Befehlswörter zählen aus dem Grund zu den terminalen Symbolen, da sie sich im Sinne der Syntaxdefinition nicht weiter in ihre Bestandteile zerlegen lassen und somit atomare Sprachbestandteile darstellen (terminal).
- Die Menge “N” beschreibt die sogenannten nicht-terminalen Symbole (auch non-terminal Symbol) einer Sprache. Darunter werden Zwischensymbole verstanden, die weiter definiert und in Bestandteile zerlegt werden können. Das non-terminal Symbol “identifier” aus der Sprachdefinition von PASCAL (beispielsweise die Bezeichner für Variablen, Konstanten und Prozeduren) kann aus einer beliebigen Buchstaben und Ziffernfolge bestehen, muß jedoch als Einschränkung mit einem Buchstaben beginnen.

- Die Menge “P” wird als die Menge der Produktionen oder Ableitungsregeln beschrieben. Eine Produktion definiert die Art und Weise, wie eine Folge terminaler und non-terminaler Symbole weiter abgeleitet beziehungsweise zerlegt werden kann.
  Im Kasten “Kontextfreie Produktionen” wird festgelegt, wie das Zwischensymbol “identifier” durch eine Buchstaben- und Ziffernfolge abgeleitet werden kann (beginnend mit einem Buchstaben, wodurch wiederum die Reihenfolge wichtig ist). Die Richtung des Ableitungsprozesses wird dabei durch den Pfeil bestimmt. COMPILERBAU ist demnach eine gültige Ableitung, 0COMPILERBAU nicht, da sie nicht mit einem Buchstaben, sondern einer Ziffer beginnt.
- Das spezielle Symbol, welches noch zum 4-Tupel gehört, ist das sogenannte Startsymbol “S”. Mit diesem Symbol, welches zusätzlich zur Menge der non-terminalen Symbole gehört, beginnen sämtliche Ableitungen. In der Sprachdefinition von PASCAL ist das Wort “program” das Startsymbol.

Somit sind alle Teile des 4-Tupels definiert. Ein Programm in einer Programmiersprache besteht demnach aus einer Folge von terminalen Symbolen (Buchstaben, Ziffern, Operatoren, Sprachbefehlen, Sonderzeichen).
Ein Programm ist nun im Sinne der Definition der Sprache syntaktisch korrekt, wenn sie sich mittels der Produktionen der Menge “P” aus dem Startsymbol “S” ableiten läßt. Das festzustellen ist nun die Aufgabe eines Parsers (Syntax-Analysator).

Kontextfreie Produktionen

Definition: *identifier* → Buchstaben/Zifferfolge

Beispiele: 1. *identifier* → COMPILERBAU
2. *identifier* → 0COMPILERBAU

Kontextsensitive Produktionen

Definition: *identifier* ::= arithmetischer Ausdruck →
Buchstaben/Zifferfolge ::= arithmetischer Ausdruck

Klassenunterschiede

Programmiersprachen werden grob in zwei große Klassen eingeteilt, die kontextfreien und die kontextsensitiven (kontextabhängigen) Sprachen. Eine Sprache heißt kontextfrei, wenn ihre Produktionen kontextfrei sind beziehungsweise wenn die Symbolfolgen, von denen aus abgeleitet wird, nur aus einem einzigen non-terminalen Symbol bestehen. Besteht die abzuleitende Symbolfolge aus einer Folge von terminalen und non-terminalen Symbolen, heißt die Produktion sowie die zugehörige Programmiersprache kontextsensitiv.
Die Produktion des Kastens kontextfreie Produktionen ist kontextfrei, weil “Identifier” genau ein einzelnes non-terminales Symbol ist. Der Zusammenhang, in dem die Produktion verwendet wird (zum Beispiel Schleifenkonstrukt, Konstruktur) ist nicht entscheidend (kontextfrei). Die Produktion aus dem Kasten kontextsensitive Produktion ist kontextsensitiv, da “identifier” immer nur im Zusammenhang mit einer Wertzuweisung verwendet werden kann.
Für kontextfreie Sprachen lassen sich Algorithmen finden, mit denen einwandfrei entscheidbar ist, ob eine Folge von terminalen Symbolen vom Startsymbol ableitbar ist und damit zur Sprache gehört, also syntaktisch korrekt ist. Dies ist mittels eines mathematischen Beweises belegbar, der jedoch an dieser Stelle zu weit führen würde. Für alle kontextfreien Sprachen existieren somit Parser, die ein Quellprogramm auf syntaktische Korrektheit hin überprüfen können.
Natürliche Sprachen, wie beispielsweise die deutsche Sprache, sind nicht kontextfrei (also kontextsensitiv). Ein und dasselbe Wort kann in unterschiedlichen Zusammenhängen vollkommen verschiedene Bedeutungen besitzen. Diese Tatsache macht auch eine grammatische Überprüfung von Texten in natürlichen Sprachen und auch die Erfassung ihrer Bedeutung mit dem Computer besonders schwierig und teilweise unmöglich.

Backus-Naur-Form

Bisher wurden die Ableitungen mittels eines Pfeils dargestellt. Es gibt jedoch eine geläufige andere Art der Darstellung von Ableitungen, die Backus-Naur-Form (BNF). Sie ist eine erheblich kompaktere und leistungsfähigere Notation für die Darstellung der Syntax einer Programmiersprache. Die BNF ist eine sogenannte Meta-Sprache, also eine Sprache zur Beschreibung formaler Sprachen. Der Kasten “Meta-Symbole der BNF und EBNF” zeigt die von der Backus-Naur-Form und die der sogenannten erweiterten Backus-Naur-Form (EBNF) verwendeten Metasymbole zur Syntaxbeschreibung. Die EBNF ist die um eine Iteration erweitere BNF. Diese geschieht mittels des Metasymbols {...}*.
Das Symbol “::=” bedeutet “wird definiert durch”. Das “|” bedeutet sprachlich ein “oder ”, also eine verkürzende Schreibweise für mehrere Produktionen. b ::= 0b | b0 | 1b | b1 ließe sich auch in die unterschiedlichen Produktionen b ::= 0b, b ::= b0, b ::= 1b, b ::= b1 aufteilen. Das Komma “,” dient zur

Trennung. Der Ausdruck, der durch geschweifte Klammern “{...}*” eingeschlossen ist, wird beliebig oft wiederholt. Die normalen runden Klammer “(...)” klammern zusammengehörige Symbole. Die eckigen Klammer “[...]” dienen der Kennzeichnung von Optionen.
Eine EBNF-Darstellung für Binärzahlen kann beispielsweise folgendermaßen erfolgen: Die Menge der terminalen Symbole T besteht aus den Elementen 0 und 1. Die Menge der non-terminalen Symbole N besteht lediglich aus dem Startsymbol S. Die Produktionen legen fest, wie aus dem Startsymbol S die zu den Binärzahlen gehörenden Ableitungen gebildet werden können (siehe Kasten EBNF-Darstellung für Binärzahlen).

Meta-Symbole der BNF und EBNF	
::=	wird definiert durch
,	Trennung
|	Alternation
{...}*	Iteration
[...]	Option
(...)	Gruppierung

EBNF-Darstellung für Binärzahlen

T = {0,1}
N = {S}
Startsymbol S
Produktion P:
S ::= (0 | 1) {0 | 1}*

■ Syntaxdiagramme

Syntaxdiagramme sind unter Umständen eine erheblich übersichtlichere Form der Darstellung von Produktionen einer Programmiersprache. Bild 3 zeigt die Regeln der Überführung einer Darstellung von EBNF in ein Syntaxdiagramm. Ein terminales Symbol wird durch einen Kreis dargestellt, ein non-terminales Symbol durch ein Rechteck. Optionen [o] finden eine Darstellung, wie in Punkt 3 gezeigt. Eine Iteration {i}* wird durch den Syntaxgraphen Punkt 4 beschrieben. Sequenzen und Alternativen zeigen die Syntaxgraphen 5 und 6.

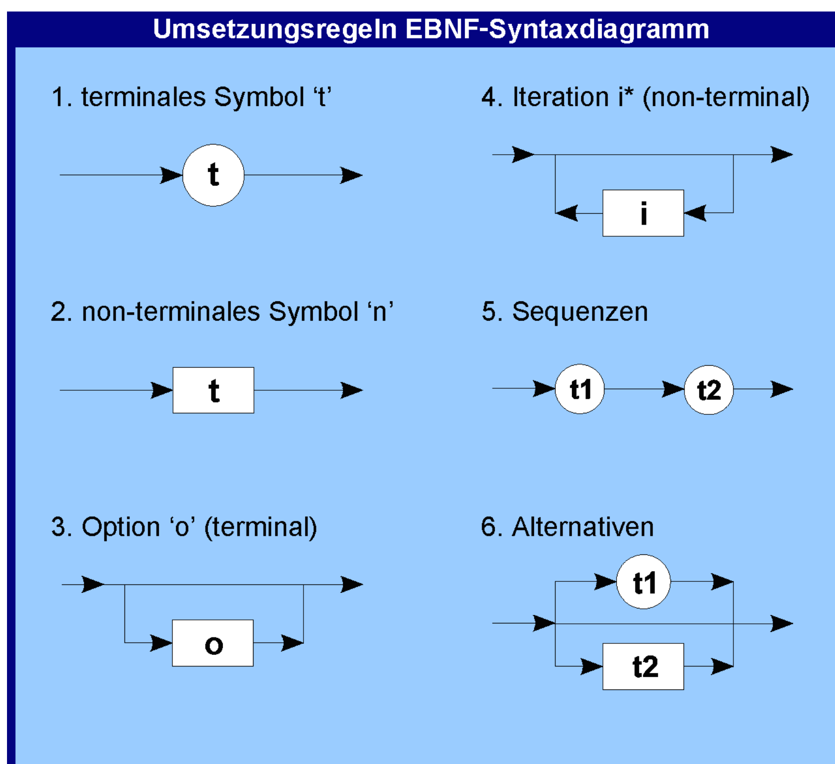

Bild 3. Regeln für die Umsetzung von EBNF in ein Syntaxdiagramm

■ Syntaxanalyse

Um nun die in einem Programm vorab formal beschriebene Syntax von einem Parser analysieren zu lassen, existieren zwei grundsätzlich verschiedene Vorgehensweisen: die Top-down- und die Bottom-up-Analyse.
Das Top-down-Analyseverfahren beginnt bei dem Startsymbol der betroffenen Programmiersprache (beispielsweise `program` bei PASCAL oder `main` bei C). Ein Top-down-Parser versucht dann durch sukzessive Anwendung der vorgegebenen Ableitungsregeln der Sprache zu den terminalen Symbolen der Sprache zu gelangen.
Ein Bottom-up-Parser arbeitet in umgekehrter Richtung. Ausgehend von der vorgegebenen Zeichenfolge versucht er, durch Rückwärtsanwendung der Produktionen zum Startsymbol zu gelangen. Man kann das Verfahren auch mit einer Reduktion bezeichnen. Es wird beispielsweise bei der Operatoren-Vorrang-Analyse verwendet, mit der die syntaktische Korrektheit arithmetischer Ausdrücke kontrolliert wird. Eine bekannte Regel daraus ist die "Punkt- vor Strich-Rechnung" bei Rechenausdrücken.
Eine Gleichberechtigung der Operatoren wird in der Tabelle (Bild 4) durch ein Gleichheitszeichen ausgedrückt, der Vorrang durch das Größer-Zeichen und die Unterordnung durch das Kleiner-Zeichen. Bei der Analyse eines arithmetischen Ausdrucks anhand der Tabelle müssen vier Regeln beachtet werden:

- Der zu untersuchende Text wird symbolweise von links nach rechts eingelesen und verarbeitet.
- Die vom Scanner gelieferten Bezeichner für Variablen und Konstanten werden auf einem Stapelspeicher abgelegt. Die vom Scanner erkannten Operatoren werden in gleicher Weise auf einem getrennten Stapel zur weiteren Verarbeitung abgelegt.
- Die Ablage der erkannten Konstanten, Variablen und Operatoren auf den Stapelspeichern wird so lange durchgeführt, bis der Scanner einen Operator liest, dessen Bedeutung kleiner oder gleich der Bedeutung des letzten abgelegten Operators auf dem Operatorenstapel ist. In diesem Fall werden sowohl der Operator als auch die zugehörigen Konstanten und Variablen vom jeweiligen Stapelspeicher entfernt. Da beim obigen Beispiel alle Operatoren zweistellig sind, werden immer die beiden obersten Variablen vom Stapel entfernt. An die Stelle der beiden Bezeichner wird das zu ihrem Teilausdruck gehörende non-terminale Symbol auf dem Stapelspeicher. Welches Symbol gespeichert wird, ergibt sich aus der Rückwärtsanwendung der Ableitungsregeln. Dieses Vorgehen wird so lange verfolgt, bis der Stapel einen Operator liefert, der dem zuletzt gelesenen Operator untergeordnet ist.
- Ist der Operatorenstapel am Ende der Analyse vollkommen geleert und befindet sich im Variablenstapel nur das Startsymbol der Sprachdefinition, dann war der analysierte Ausdruck syntaktisch korrekt.

Ein Top-down-Parser wird in der Regel nach der Methode des sogenannten rekursiven Abstiegs realisiert. Um einen Top-down-Parser für eine gegebene Programmiersprache zu entwickeln, können einige Regel definiert werden.

- Jedem non-teminalen Symbol wird eine Prozedur zugeordnet, zum Beispiel `procedure EXPRESSION;` , wobei Expression ein non-terminales Symbol aus der Sprache der arithmetischen Ausdrücke ist.
- Alle terminalen Symbole "t" werden in folgendes Abfrageschema übersetzt: `if T_SYMBOL = t then SCANNER else ERROR`. Nach jeder Überprüfung wird also entweder der Scanner aufgerufen (Überprüfung o.k.) oder eine Fehlerbehandlungsroutine gestartet.
- Alternativen werden als geschachtelte if-Abfrage realisiert.
- Eine sequentielle Folge von non-terminalen Symbolen (n1 ... nn) wird folgendermaßen übertragen: `begin n1; n2; ... nn; end`
- Jede Iteration wird in eine einfache while-Schleife übersetzt.
- Optionen werden als einfache if-Abfragen gestaltet.
- Schließlich werden die einzelnen Prozeduren sowie der Scanner zu einem Hauptprogramm (Parser) zusammengefaßt.

Um die Effizienz der Erkennung zu steigern, ergeben sich an einigen Stellen bei der eben beschriebenen Umsetzung ein paar Einschränkungen. Beispielsweise sollten bei mehreren Alternativen in einer Ableitungsregel die dazugehörigen Anfangssymbole keine gemeinsamen Bestandteile enthalten, da der Parser ansonsten bei der Analyse auf die Trial-and-Error-Methode angewiesen ist.

Operatoren-Vorrang-Analyse				
	+	-	*	/
+	=	=	<	<
-	=	=	<	<
*	>	>	=	=
/	>	>	=	=

Bild 4. Operatoren-Tabelle zur Analyse arithmetischer Ausdrücke

■ Fehlerbehandlung

Problematisch für eine Fehlerprozedur ist es, exakt zu erkennen, um welche Art Fehler es sich handelt (unbekannter Befehl, falsche Klammerung und so weiter). Die Fehlerposition kann der Scanner leicht durch einen mitlaufenden Zeiger realisieren, der sich die aktuelle Position im Quelltext merkt. Es ist nicht immer eindeutig möglich, die Anfangsposition des Fehlers festzustellen. Der Parser kennzeichnet immer die Position, an der der Fehler seine ersten Auswirkungen zeigt (Ableitung nicht nachvollziehbar).

■ Semantische Analyse

Die Semantik einer Programmiersprache läßt sich in der Regel nicht so formal definieren wie eine Syntax. Dadurch ergibt sich Spielraum für Interpretationen, obwohl es auch Versuche gibt, formale Beschreibungsmöglichkeiten für Semantiken von Programmiersprachen zu finden. Diese Versuche basieren allesamt auf der Verwendung von sogenannten attributierten Grammatiken. Semantische Anforderungen an ein Programm werden mit dem Begriff "statische Semantik" beschrieben. Sie ist nicht mit Methoden der Syntaxanalyse zu analysieren, da sie teilweise stark kontextsensitiv ist und sich dadurch auch nicht in EBNF ausdrücken läßt, mit der nur die kontextfreien Teile einer Sprache beschreibbar sind. Attributierte Grammatiken sind ein Versuch, eine formale Beschreibungsmöglichkeit für Semantiken zu finden. Sie sind allerdings nicht nur für die formale Definition von semantischen Anforderungen an ein Programm wichtig. Alle möglichen semantischen Aktionen (beispielsweise die Erzeugung von Maschinencode oder die Ausführung einer Multiplikation) können mit attributierten Grammatiken beschrieben werden. Eine tiefergehende Beschreibung attributierter Grammatiken würde den Rahmen dieses Artikels sprengen, jedoch sei zur Veranschaulichung noch ergänzt, daß es sich im Prinzip um Grammatiken (also formale Sprachdefinitionen) handelt, deren Elemente (hier Terminalsymbole) mit Eigenschaften beziehungsweise Attributen versehen sind. Beispielsweise könnte man einer Variablen die Attribute Wert und Typ zuweisen.

■ Codeerzeugung

Wie schon angedeutet, läßt sich die Codeerzeugung ebenfalls mit attributierten Grammatiken steuern, da sie eine spezielle Form einer semantischen Aktion ist. Jeder Prozessor besitzt einen unterschiedlichen Befehlsumfang. Deshalb ist die Codeerzeugung maschinenabhängig.
Bei der Codeerzeugung muß der Compiler nicht nur den Maschinencode erzeugen. Er muß insbesondere auch dafür Sorge tragen, daß für Konstanten, Variablen und Felder Speicher reserviert wird. Dies ist je nach Programmiersprache mehr oder weniger aufwendig. Rekursion und dynamische Datenstrukturen komplizieren dies unter Umständen.
Heute wird von vielen Compilern, insbesondere auf PC- und Workstation-Plattformen, nicht direkt Maschinencode, sondern ein Zwischencode erzeugt. Auf diese Weise wird es möglich, daß unterschiedliche Programmteile in verschiedenen Programmiersprachen geschrieben werden können und letztendlich von einem sogenannten Binder (Linker) zu einem ablauffähigen Programm zusammengefaßt werden.

■ Optimierung

Eingangs wurden schon lokale und globale Optimierungen angeschnitten. An dieser Stelle werden nun einige Beispiele aufgeführt, die diese Begriffe verdeutlichen sollen. Üblicherweise werden die Optimierungsmöglichkeiten von Compilern überschätzt. Die wichtigste und beste Optimierung kann nur vom Programmierer selbst erfolgen, indem er möglichst effiziente Algorithmen optimal in die Zielsprache umsetzt. Unnötige Berechnungen sollten dann vermieden werden, wenn zeitkritische Probleme zur Lösung anstehen.

- Potenzen werden in der Regel über Logarithmen berechnet. Hierfür werden unter Umständen aufwendige numerische Verfahren verwendet. Der Compiler kann also bei der Optimierung möglicherweise Zweier- und eventuell auch noch Viererpotenzen in entsprechende Multiplikationen umsetzen und dadurch optimierend wirken. Lokale Optimierungen erstrecken sich hauptsächlich auf die Vereinfachung und optimale Berechnung von arithmetischen Ausdrücken.
- Multiplikationen mit Null und Eins müssen gar nicht durchgeführt werden, denn das Ergebnis ist in jedem Fall mathematisch fix definiert beziehungsweise einer der Multiplikanden selbst. Additionen und Subtraktionen um Eins können in der Regel durch Befehle auf Maschinensprachenebene umgesetzt werden (INCrement, DECrement).
- Tritt in einem arithmetischen Ausdruck ein Term mehrfach auf, so kann der Compiler die Berechnung so optimieren, daß die Berechnung des Terms nur noch einmal erfolgen muß.

Das folgende Beispiel zeigt eine globale Optimierung, die beim Compiler schon einen gewissen Grad an "Intelligenz" voraussetzt.

```
  for i:=1 to 100 do
  begin
    Prozedur_1;
    x:=3 + 8;
    Prozedur_2;
  end;
```

Die Berechnung der Variablen x innerhalb der Schleife ist eigentlich nur einmal notwendig, wenn sie im Umfeld der Schleife (also in Prozedur_1 und Prozedur_2) nicht verändert wird. In diesem Fall kann die Berechnung aus der Schleife herausgezogen werden und somit (in diesem Fall 99) überflüssige Berechnungen gespart werden.
Manche Compiler besitzen die Möglichkeit, rekursive Strukturen in iterative Strukturen umzusetzen. Üblicherweise ist die Abarbeitung rekursiver Strukturen mit einem höheren Rechen- und Verwaltungsaufwand für den Prozessor verbunden als die Abarbeitung iterativer Strukturen. Für Rekursionen wird die Verwaltung eines Stapelspeichers benötigt, der die Rücksprungadressen verwaltet. Das fällt in der Regel bei iterativen Strukturen weg.

■ Zusammenfassung

Die automatische Überprüfung des formulierten Programmcodes ist eine grundlegende Voraussetzung bei der Softwareentwicklung. Compiler als Übersetzungsprogramme für die Generierung speicherfähiger Objektprogramme sind heute aus dem Gebiet der Softwareentwicklung nicht mehr wegzudenken. Der Trend dabei geht eindeutig in Richtung komplexer Modellierungs- und Entwicklungstools (beispielsweise Borland Delphi, Bild 5), mit deren Hilfe ein Programmierer auch in kurzer Zeit umfangreiche Programmpakete erstellen kann.

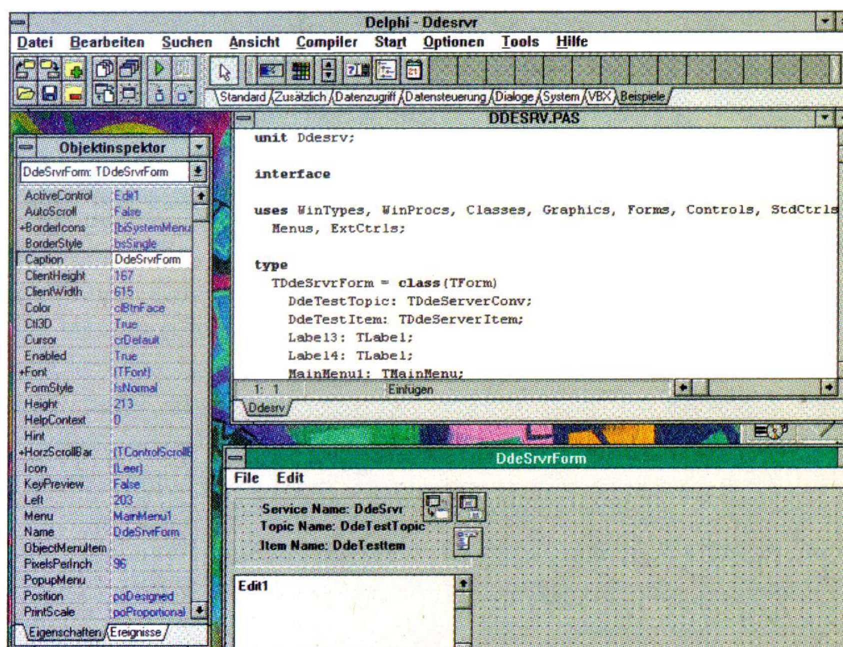

Bild 5. Borland Delphi als modernes Entwicklungstools für die Programmentwicklung