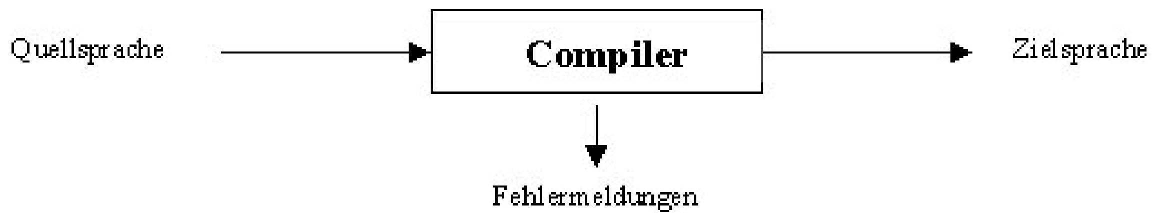
Compilerbau

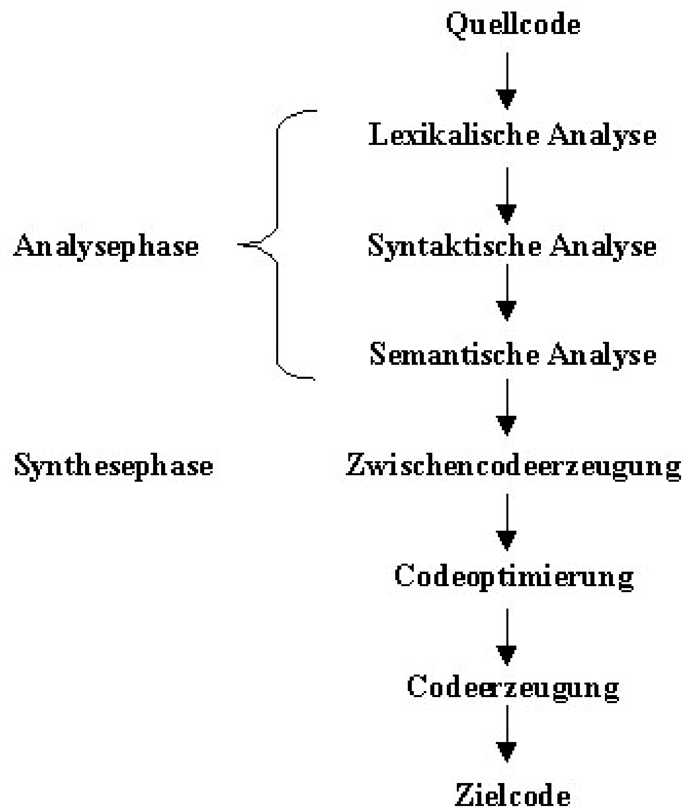
1. Einleitung

Der Compiler ist ein Programm, das ein in einer bestimmten Sprache (Quellsprache geschriebenes Programm liest und es in ein äquivalentes Programm einer anderen Sprache (Zielsprache übersetzt und weiters dem Benutzer Fehler die im Quellprogramm enthalten sind zu melden. Die ersten Compiler wurden in den frühen 50-er Jahren entwickelt und galten als schwer zu schreibende Programme (Erster Fortran-Compiler benötigte 18 Mannjahre).



Die Bandbreite der Quell- und Zielsprachen ist sehr groß. Die Bandbreite reicht von der Übersetzung von C++-Code in Assembler-Code bis zur Übersetzung von Postscript-Code in eine grafische Ausgabe. Der Aufbau dieser Compiler ist jedoch grundsätzlich der Selbe.

2. Aufbau eines Analyse-Synthese Compilers



Analysephase: Diese Phase zerlegt das Quellprogramm in seine Bestandteile und erzeugt eine Zwischendarstellung des Quellprogramms. Während der Analyse werden die im Quellprogramm enthaltenen Operationen bestimmt und in einem Baum (Syntaxbaum) angeordnet. Jeder Knoten stellt eine Operation dar. Die Kanten enthalten die Argumente der Operationen.

Synthesephase: Dieser Teil konstruiert das gewünschte Zielprogramm aus der Zwischendarstellung

2.1Symboltabellen

In der Symboltabelle werden die Bezeichner des Quellprogramms und Informationen der Attribute des Bezeichners gespeichert. Die Attribute können den Speicherbedarf eines Bezeichners betreffen, seinen Typ, seinen Gültigkeitsbereich (wo im Programm ist er gültig) und bei Prozedurnamen die Anzahl und Typen der Argumente sowie den Typ des Rückgabewertes. Sobald die lexikalische Analyse einen Bezeichner im Quellprogramm erkennt wird er in die Symboltabelle eingetragen. Die Attribute eines Bezeichners können jedoch im allgemeinen nicht während der lexikalischen Analyse bestimmt werden.

Bsp.: var posotion, rate:real;

Wenn position bzw. rate gelesen wird ist der Typ real noch nicht bekannt. Die restlichen Phasen tragen ebenfalls Informationen über den Bezeichner in die Symboltabelle ein.

2.2Lexikalische Analyse (Scanner)

Liest die Zeichen aus denen das Quellprogramm besteht von links nach rechts durch und teilt es in Symbole (Token) auf. Ein Token stellt eine Folge von Zeichen dar, die zusammen eine bestimmte Bedeutung haben. Diese können ein Bezeichner, ein Schlüsselwort (if, while, ...) oder ein Satzzeichen (+,-,=, ...) sein. Die Zeichenfolge aus der ein Symbol besteht heißt „lexem“ für das Symbol.

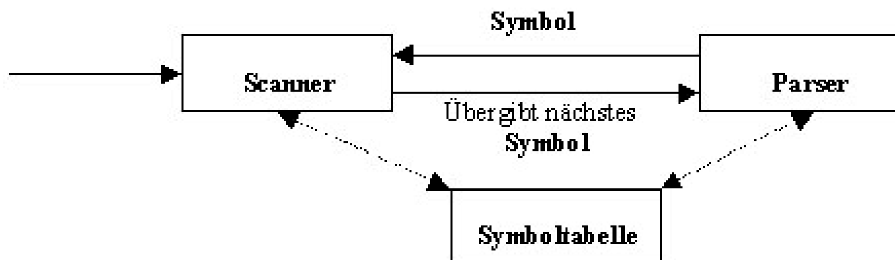
Bsp:

Position=initial + rate *60 wird in folgende Symbole gruppiert:

- Bezeichner position
- Zuweisungssymbol =
- Bezeichner initial
- Satzzeichen +
- Bezeichner rate
- Satzzeichen *
- Zahl 60

Die Leerzeichen werden bei der lexikalischen Analyse entfernt und die Bezeichner in einer Symboltabelle gespeichert.

2.2.1 Rolle des Scanners



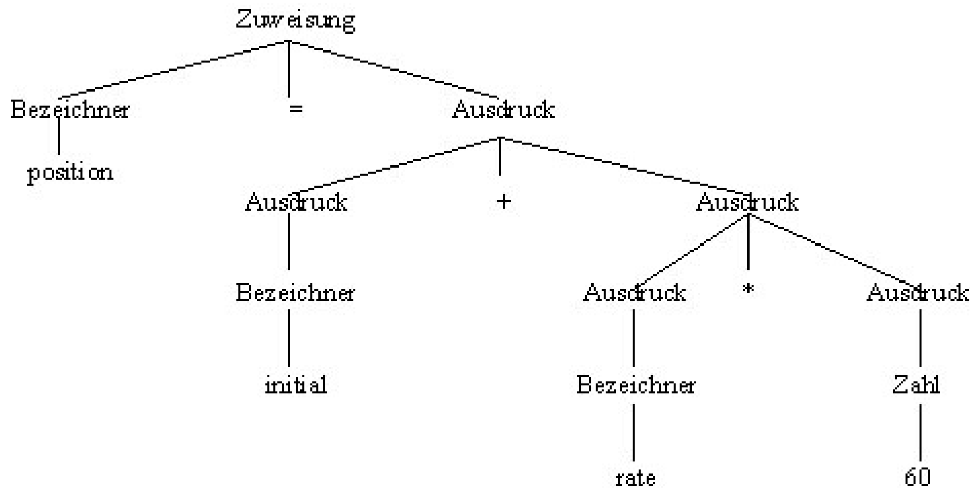
Der Scanner liest Eingabezeichen ein und erzeugt eine Folge von Symbolen, die der Parser syntaktisch analysiert. Er ist als Unterprogramm des Parsers implementiert. Nach Erhalt eines vom Parser gegebenen Kommandos liest der Scanner

solange Eingabezeichen, bis das nächste Symbol gefunden wurde. Der Scanner ist derjenige Teil, der den Quellcode liest. Er entfernt auch Leerzeichen, Tabulatoren, Kommentare und Zeilenwechsel.

2.3Syntaxanalyse Analyse (Parser)

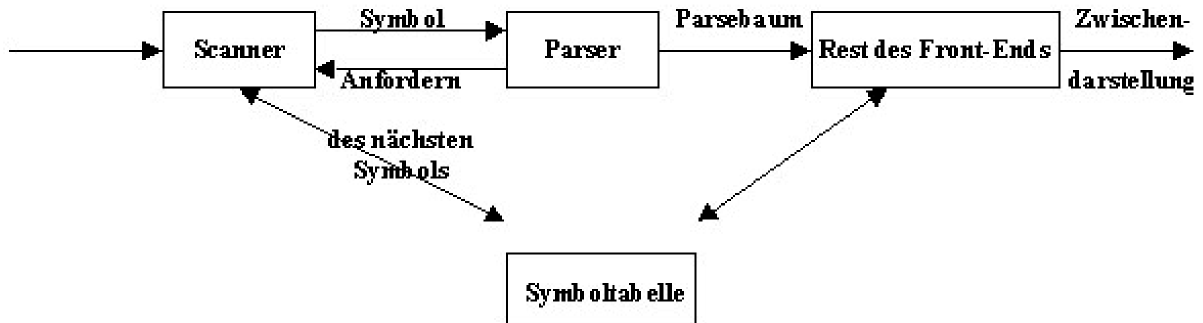
Die Systaxanalyse ist der Prozeß, der entscheidet, ob ein aus Symbolen bestehendes Programm von einer Grammatik erzeugt werden kann. Während der Syntaxanalyse wird ein Parese-Baum aufgebaut (Parse = allgemeiner Erkennungs-Algorithmus), sonst ist die Korrektheit der Übersetzung nicht gewährleiset. Grundsätzlich läßt sich für jede Grammatik ein Parser konstruieren. Die Aufgabe der Syntaxanalyse ist somit, das Quellprogramm zu grammatikalischen Sätzen zusammenzufassen, die der Compiler zur Erzeugung einer Ausgabe benutzt.

Bsp: Parsebaum für position=initial + rate*60



2.3.1 Die Rolle des Parsers

Der Scanner liefert dem Parser eine Folge von Symbolen. Der Parser prüft, ob diese Symbolfolge von der Grammatik der Quellsprache erzeugt werden kann. Außerdem liefert er aussagekräftige Meldungen bei Fehlern und zwar so, daß er auch den Rest der Eingabe bearbeiten kann (Recovering Strategie).



2.4Semantische Analyse

Hier wird die Funktion des Syntax analysiert. Die Funktion der Syntax ist die Beschreibung der Menge der Symbolfolgen, genannt Sätze, die zur Sprache gehören. Die zweite wichtige Funktion ist die Definition einer Satzstruktur. Diese spielt eine Rolle in der Erkennung der Bedeutung des Satzes. Syntax und Semantik sind sehr eng miteinander verbunden. Es wird überprüft, ob das Quellprogramm semantische Fehler hat. Typ-Informationen für die Codegenerierung werden gesammelt.

Bsp.: Array des Typs float angesprochen.

2.5Die Zwischencodeerzeugung

Manche Compiler erzeugen nach der syntaktischen und semantischen Analyse eine Zwischendarstellung des Quellprogramms. Sie sollte zwei wichtige Aufgaben erfüllen. Erstens soll sie leicht zu erzeugen sein und zweitens sollte sie leicht ins Zielprogramm zu übersetzen sein. Es gibt verschiedene Formen der Zwischencodeerzeugung.

Bsp.: Drei-Adreß-Code

Dieser Code ist sehr Assembler-ähnlich. Der Drei-Adreß-Code besteht aus einer Folge von Instruktionen, von denen jede höchstens drei Operanden hat.

Bsp.:
temp1=inttoreal(60)
temp2=id3*temp1
temp3=ide2+temp2
id1=temp3

Wenn der Compiler diese Befehle erzeugen will muß er eine Reihenfolge festlegen, in der die Operatioenen auszuführen sind. Er muß die Priorität der Operatoren berücksichtigen (Multiplikation vor Addition). Der Compiler muß auch temporäre Namen erzeugen. Einige Drei-Adreßen-Befehle haben auch weniger als drei Operanden (erster und letzer in Beispiel).

2.6Code-Optimierung

Die Phase der Code-Optimierung (Code-verbessernde Transformationen) versucht, den Zwischencode zu verbessern. Das Ergebnis soll ein effizienterer Maschinencode sein. Die besten Programm-Transformationen sind jene, aus denen man mit dem geringsten Aufwand den größten Nutzen zieht.

Kriterien für Code-Optimierung:

Die Bedeutung des Programmes muß gewahrt bleiben.
Es dürfen keine zusätzlichen Fehler verursacht werden.
Es soll die durchschnittliche Geschwindigkeit eines Programmes meßbar erhöht werden.
Es soll der vom compilierten Code belegte Speichersplatz reduziert werden.

Nähere Erklärung anhand des Beispiels oben.

Der Compiler kann erkennen, daß die Umwandlung der Zahl 60 von der Integer in die Real-Darstellung ein für allemal zur Compilerzeit durchgeführt werden kann und damit die Operation inttoreal eingespart wird. Außerdem wird der Name temp3 nur einmal verwendet, um seinen Wert an id1 zu übergeben. Man kann deshalb gefahrlos temp3 durch id1 ersetzen, wodurch die letzte Anweisung überflüssig wird.

Ergebnis:

temp1=id3*60
id1=id2+temp1

2.7Code-Erzeugung

Die letzte Phase ist die Erzeugung des Zielcodes, der im allgemeinen aus absolutem, verschiebbaren Maschinen- oder Assemblercode besteht. Jeder im Programm benutzten Variable wird Speicherplatz zugeordnet. Danach wird jede Instruktion der Zwischendarstellung in eine funktionserhaltende Folge von Maschinenbefehlen übersetzt. Entscheidender Punkt ist die Zuordnung von Registern zu Variablen.

Bsp.:

MOV id3,R2
MULF #60.0,R2

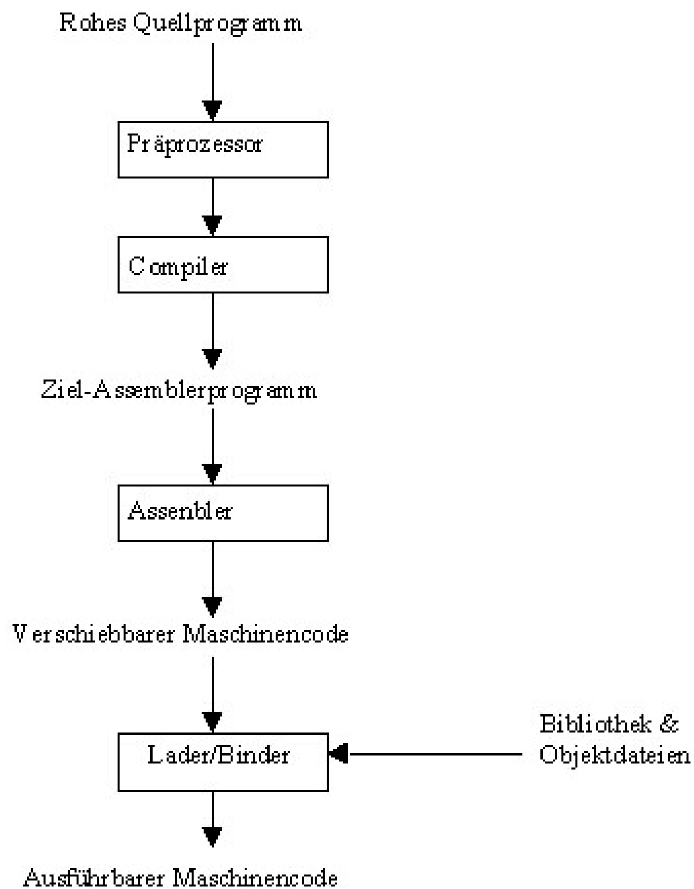
MOVF id2,R1
ADDF R2, R1
MOV R1,id1

Der erste Operand jeder Instruktion bezeichnet die Quelle, der zweite das Ziel. Das F in den Instruktionen sagt aus, daß sie Gleitkommazahlen behandeln. Der obige Code ist äquivalent mit dem obigen Beispiel.

3. Die Umgebung des Compilers

Um ein ausführbares Programm zu erzeugen sind neben einem Compiler meistens noch verschiedene andere Programme notwendig.

Bsp:



3.1Präprozessor

Sie erzeugen die Eingabe für einen Compiler. Sie kännen folgende Funktionen übernehmen:

- **Makrobearbeitung:** Kopiert Makros an die richtige Stelle bzw. fügt sie ein.
- **Einkopieren** von Dateien: z. B. Header Dateien in C.
- **Spracherweiterungen:** z. B. ESQL: Anweisungen dir mit $ beginnen werden vom Präprozessor als Anweisungen für den Datenbankzugriff verstanden

3.2Assembler

Einige Compiler erzeugen Assemblercode, der einem Assembler zur Weiterverarbeitung übergeben wird. Andere Compiler erledigen die Arbeit des Assemblers selber und erzeugen selbst einen verschiebbaren Maschinencode (Code der an jeder Stelle des Speichers geladen werden kann). Dieser kann direkt dem Linker übergeben werden.

3.3Binder/Lader

Der Prozeß des Ladens besteht darin, die verschiebbaren Adressen im verschiebbaren Maschinen so zu verändern, daß die geänderten Befehle und Daten an den richtigen Stellen im Speicher abgelegt werden. Der Binder ermöglicht es, verschiedene Dateien, die jeweils verschiebbaren Maschiniencode enthalten, zu einem eigenem Programm zusammenzufassen.

4. Allgemeines zu Compilern

4.1Front- und Back-End

Oft werden die Phasen in einen vorderen Teil, das Front-End und in einen hinteren Teil, das Back-End aufgeteilt.

Das Front-End besteht aus denjenigen Phase, die in erster Linie von der Quellsprache abhängen und weitgehend von der Zielmaschine unabhängig sind.

Dazu zählen:

Lexikalische Analyse
Syntaktische Analyse
Erstellen der Symboltabelle
Semantische Analyse
Zwischencode Erzeugung
Fehlerbehandlung die diese Phase betreffen

Das Back-Ende beinhaltet diejenigen Teile des Compilers, die sich auf die Zielmaschine beziehen. Im allgemeinen hängen diese Phase nicht von der Quellsprache ab, sondern nur von der Zielsprache.

Dazu zählen:

Code-Optimierung
Code-Erzeugung
Fehlerbehandlungen
Symboltabellenoperationen

4.2Erkennen und Melden von Fehlern

Jede Phase kann auf Fehler stoßen. Diese Fehler müssen irgendwie behandelt werden. Ziel ist es, die Übersetzung fortzuführen, um danach weitere Fehler im Quellprogramm zu finden. Ein Compiler, der nach dem ersten Fehler abbricht ist nicht besonders hilfreich. Besser wäre es, wenn er weiterarbeiten würde. Die Phasen der syntaktischen und semantischen Analyse behandeln den größten Teil der Fehler. Die lexikalische Phase kann die Fehler erkennen, bei der die in der Eingabe befindlichen Zeichen kein Symbol der Sprache bilden. Fehler in der Syntax werden von der Syntaxanalyse-Phase entdeckt. Die semantische Analyse erkennt die zwar syntaktisch korrekten, deren Operation jedoch keinen Sinn ergibt.