Skript zur Vorlesung

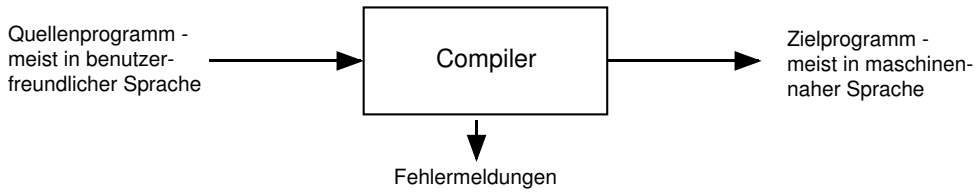
Compilerkonstruktion II

Wintersemester 2015/16

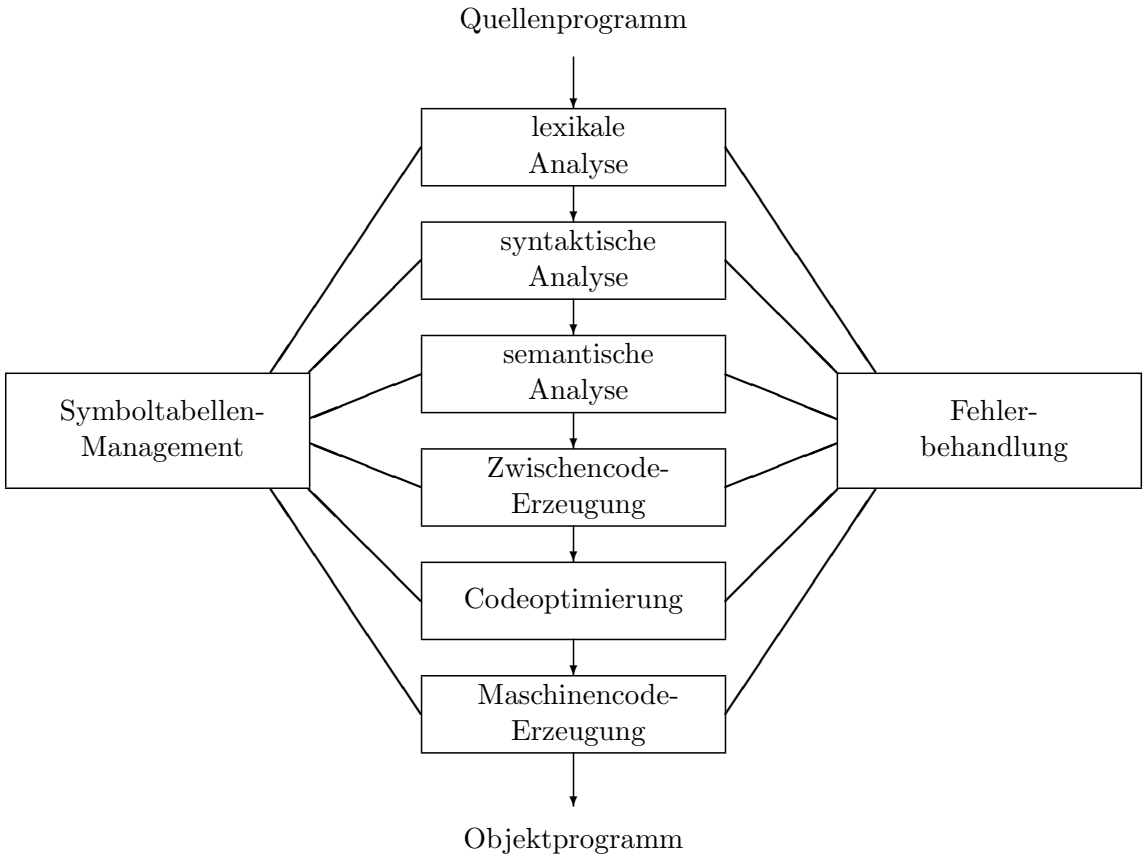
Institut für Praktische Informatik
FG Programmiersprachen und Übersetzer
Universität Hannover

1 Einleitung

Als Compiler bezeichnet ein Programm, das einen Text - meist einen Programmtext - geschrieben in einer Sprache A einliest und in einen äquivalenten Text in einer anderen Sprache B übersetzt.



1.1 Prinzipieller Aufbau eines Compilers



Die Aufgaben der einzelnen Komponenten kann man kurz wie folgt charakterisieren:

lexikale Analyse: Aufgabe der lexikalen Analyse ist das zeichenweise Lesen der Eingabe und das Zusammenfassen der Zeichen zu größeren Einheiten. Eine derartige Einheit wird auch **Lexem** genannt. Als interne Darstellung wird für jedes Lexem ein **Token** erzeugt. Token repräsentieren eine Folge von Eingabezeichen, die als logische Einheit zu betrachten sind; z. B. ist die Zeichenfolge `1234` als eine Zahl und *nicht* als Folge von vier Zahlen aufzufassen. An diesem Beispiel erkennt man auch, dass ein Token aus zwei Komponenten besteht,

nämlich der **Tokenklasse** oder dem **Tokennamen** (in diesem Fall `Zahl`) und dem **Tokenwert** (in diesem Fall der Zahlwert `1234`). Übliche Tokenklassen bei Programmiersprachen sind z. B. Schlüsselworte wie etwa `do`, `begin`, `end` usw. oder `Zahl` oder `Identifier` (Namen). Was als Token zu klassifizieren ist, hängt von der beabsichtigten Anwendung ab und ist in das Ermessen des Compilerkonstrukteurs gestellt. Häufig werden Schlüsselworte und Namen von Variablen (Identifier) aus Effizienzgründen vom Scanner in eine Tabelle aller auftretenden Worte eingetragen. Man erreicht dadurch, dass auftretende Identifier-Tokens eindeutig sind; zum Beispiel sollen zwei unterschiedliche Auftreten eines Identifiers `abc` zu einem eindeutigen Token führen. Außerdem werden Leerzeichen und Kommentare von dieser Komponente des Compilers, die auch als **Lexical Scanner** oder einfach **Scanner** bezeichnet wird, überlesen.

Als theoretisches Modell zur Beschreibung der Tokenklassen dienen reguläre Sprachen bzw. reguläre Ausdrücke und endliche, erkennende Automaten.

syntaktische Analyse: Diese Komponente des Compilers wird auch als **Parser** bezeichnet. Ihre Aufgabe ist:

1) Eine Prüfung, ob die von der lexikalen Analyse gelieferten Token in einer Reihenfolge auftreten, die der (syntaktischen) Definition der Programmiersprache entspricht. (Z. B. gibt `identifier + * identifier` in Java einen Fehler, in C jedoch nicht!)
2) Die Gruppieren der Token zu größeren syntaktischen Strukturen, etwa in Form eines Baumes (**Syntaxbaum**). Ein interner Knoten des Syntaxbaums stellt eine Operationen dar, die auf Werte angewendet wird, die durch die Teilbäume unter dem internen Knoten repräsentiert werden.
3) Aufbau und Verwaltung einer Symboltabelle, in der die Token mit entsprechenden Typ-Informationen abgelegt werden.

Als theoretisches Modell zur Beschreibung der korrekten Syntax einer Programmiersprache werden kontextfreie Grammatiken benutzt. Dementsprechend bilden verschiedene Arten von Kellerautomaten ein Modell für den Parser.

semantische Analyse: Wichtigste Aufgabe der semantischen Analyse ist die Überprüfung der Typen von Konstanten und Variablen bezüglich der auf sie angewendeten Operationen. (Z. B. sollte ein Feldname mit einem Index vom Typ `real` einen Fehler liefern.) Falls es die Programmiersprache erlaubt, kann in dieser Phase des Compilers auch eine automatische Typanpassung von Variablen eingefügt werden. Weiterhin muss, sofern nicht schon in der syntaktischen Analyse geschehen, überprüft werden, ob verwendete Identifier auch im Deklarationsteil des Programms definiert worden sind.

Bei objektorientierten Sprachen muss die semantische Analyse anhand der Klassenhierarchie prüfen, ob eine Methoden auf einen Identifier anwendbar ist und gegebenenfalls, falls zur Übersetzungszeit bereits möglich, die „passende“ Methode ausgewählt werden. Schwieriger wird es bei generischen Klassen und speziell bei komplizierteren Typ-Systemen, insbesondere bei Typ-Inferenzsystemen, muss man mit allgemeinen Typ-Ausdrücken und Unifikationalgorithmen arbeiten.

Bei einfacheren Programmiersprachen kann die semantische Analyse parallel zur syntaktischen Analyse erfolgen, bei komplexeren Typs-Systemen ist die semantische Analyse in mehrere Teilphasen zerlegt, die nacheinander abgearbeitet werden.

Zwischencode-Erzeugung: Aufgabe der Zwischencode-Erzeugung ist die Übersetzung in einen Zwischencode, d. h. in die Maschinensprache einer abstrakten Maschine, die möglichst einfache Struktur hat.

Dieser Zwischenschritt ist besonders bei optimierenden Compilern wichtig. Dieser Zwischencode (Syntax-Bäume, Bytecode, 3-Adress-Befehle usw.) sollte so aufgebaut sein, dass er leicht von der höheren Programmiersprache aus produzierbar, aber auch leicht in Maschinensprache übersetzbar oder aber auch interpretierbar ist, wie etwa der Bytecode der Java Virtual Machine.

Häufig ist diese Phase der Übersetzung in mehrere einzelne Schritte aufgeteilt, in denen verschiedene Zwischencodes erzeugt werden.

Ein theoretisches Modell für einen allgemeinen Übersetzungsprozess bilden zum Beispiel attributierte Grammatiken oder Syntax-gesteuerte Übersetzungsschemata.

Codeoptimierung: In dieser Phase des Compilers wird versucht, den Zwischencodes in Hinblick auf Speicherplatzbedarf und Laufzeitverhalten zu verbessern. Dazu dienen Algorithmen zum Entfernen überflüssiger Berechnungen, Ersetzung von Berechnungen zur Laufzeit durch Berechnungen zur Übersetzungszeit, Erkennen von schleifeninvarianten Berechnungen und Verschieben dieser Teile vor den Schleifenbeginn usw. Hier geht meist schon die Struktur der Zielmaschine entscheidend ein.

Um diese Veränderungen am Zwischencode ohne Veränderung der Bedeutung des Programms durchführen zu können, werden notwendige Informationen in dieser Phase durch das Lösen komplexer Datenfluss- und Kontrollfluss-Gleichungen gewonnen.

Auch diese Phase des Compilers läuft meist in mehreren nacheinander folgenden Schritten ab, die teilweise mit der Zwischencode-Erzeugung verzahnt sind.

Code-Erzeugung: Erzeugung von verschieblichen Maschinencode (d. h. eines Objektprogramms bzw. eines Assemblerprogramms.

Die größte Schwierigkeit besteht dabei darin, die Zuordnung der wenigen freien Register so zu wählen, dass möglichst wenig Speicherzugriffe nötig sind. (Speziell wichtig und kompliziert bei modernen RISC-CPUs oder Parallelrechnern.) sowie die Auswahl der „passenden“ Maschinenbefehle.

Als Modell verwendet man in dieser Compilerstufe häufig Baumgrammatiken in Verbindung mit dynamischer Programmierung mit einer speziellen Form des Template-Matchings oder des Pattern-Matchings.

Symboltabelle: Die Aufgabe der Symboltabelle ist das Sammeln aller Informationen über die im Programm auftretenden Variablen. Dazu gehört

- Typ-Informationen
- Speicherbedarf und Speicherort
- Gültigkeitsbereich
- Bei Prozeduren Anzahl und Typ der Parameter und der Rückgabewerte

Die Symboltabelle muss eine einfache Möglichkeit bieten, neue Informationen aufzunehmen und nach Informationen zu suchen. Außerdem muss die Symboltabelle Sichtbarkeitsregeln in der Programmiersprache, etwa eine Blockstruktur, berücksichtigen.

Die notwendigen Informationen werden von verschiedenen Teilen des Compilers geliefert und eingetragen.

Hinweis::
Die ersten Stufen eines Compilers sind relativ gut formalisierbar und die zugehörigen theoretischen Modelle sind intensiv studiert worden. Dies hat zur Folge, dass es eine Vielzahl von relativ guten Programmen zur automatischen Erzeugung von lexikalen Scannern und Parsern gibt. (etwa `yacc`, `lex`, `flex`, `bison`, `ANTLR` usw.)

Bei den restlichen Stufen sind die Trennungslinien nicht so ausgeprägt. Die Abhängigkeit der einzelnen Stufen voneinander und auch die Abhängigkeit von der Zielmaschine erschweren die Formalisierbarkeit.

Es hat in der Vergangenheit viele Versuche gegeben, auch diese Stufen eines Compilers automatisch auf einer Spezifikation zu generieren, jedoch hat sich kein Ansatz so richtig durchgesetzt. Die Schwierigkeiten und die Komplexität dieser Stufen hat sich in der Vergangenheit eher erhöht, denn

1) Die modernen Rechner haben hierarchisch geordnete Speicherstrukturen (verschiedene Cache-Stufen), so dass man nicht mehr von konstanten „Kosten“ für eine Lade- bzw. Speicheroperation ausgehen kann.

2) Die in modernen Prozessoren vorhandenen Einheiten zur Bearbeitung von Vektoroperationen oder die effiziente Ausnutzung mehrfach vorhandener Funktionseinheiten stellen zusätzliche schwierige Probleme an die letzten Stufen des Compilers.

3) Die „Kosten“ eines Maschinenbefehls sind bei modernen RISC-Prozessoren *nicht* mehr kontext-unabhängig; man denke an Pipelining und voneinander unabhängige Funktionseinheiten.

Als Beispiel soll folgendes Programmfragment[1] betrachtet werden. Man erkennt sofort, dass gleiche Operationen in unterschiedlichem Kontext unterschiedliche „Kosten“ haben können.

```
do i = 1,1024*step,step
    a[i] := a[i] + c;
end do;
```

`step`	rel. Geschw.
1	100%
2	83%
4	63%
8	40%
16	23%
64	19%
256	12%

[1] aus D.F.Bacon, Compiler Transformations for High-Performance Computing, ACM Comp. Surveys 26, 1994