Compilerkonstruktion

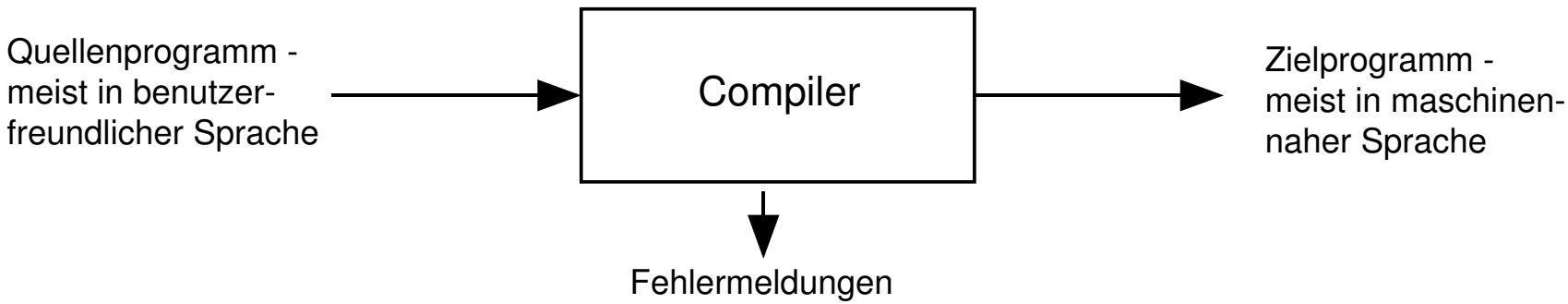
Wintersemester 2015/16

Prof. Dr. R. Parchmann

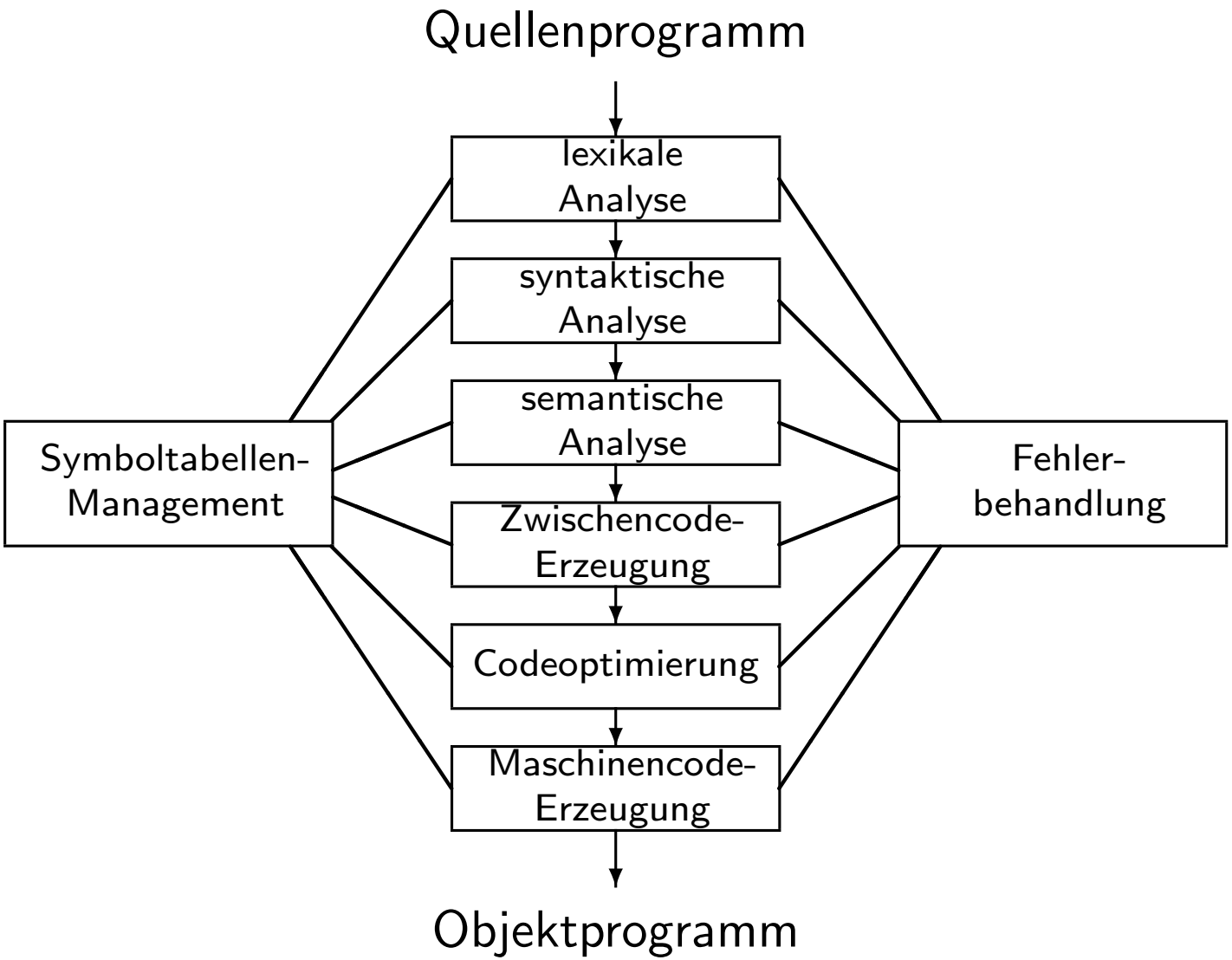
13. Oktober 2015

Einleitung

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



Schematischer Aufbau eines Compilers



Aufgaben der lexikalen Analyse (Scanner)

- Zusammenfassen der Eingabezeichen zu sogenannten Lexemen.
- Für jedes Lexem wird ein Token erzeugt, das aus einem Tokennamen (Tokenklasse) und einem Tokenwert besteht.
- Entfernen von Kommentaren und Zwischenräumen (white space).
- Aufbau einer Tabelle aller Schlüsselworte und aller auftretenden Namen von Variablen.

Als theoretisches Modell zur Beschreibung der Tokenklassen dienen reguläre Sprachen und als Modell für den Scanner endliche Automaten.

Aufgaben der syntaktischen Analyse (Parser)

- Prüfen, ob die vom vom Scanner gelieferten Token in einer Reihenfolge auftreten, die der syntaktischen Struktur der Programmiersprache entspricht.
- Parallel dazu wird eine interne Repräsentation des eingegeben Programms aufgebaut, auf der die nachfolgenden Phasen des Compilers arbeiten. Häufig werden hier Syntax-Bäume verwendet, bei denen die Blätter Operanden und die internen Knoten Operationen entsprechen.
- Aufbau und Verwaltung einer Symboltabelle

Als theoretisches Modell zur Beschreibung der Syntax dienen kontextfreie Grammatiken und als Modell für den Parser verschiedene Formen von Kellerautomaten.

Aufgaben der semantischen Analyse

- Überprüfung der Typen von Konstanten und Variablen auf Verträglichkeit mit den auf sie angewendeten Operationen.
- Eine eventuelle von der Sprache erlaubte automatischer Typanpassung.
- Prüfung des Programms auf Konsistenz mit der Sprachdefinition unter Verwendung des Syntaxbaumes und der Symboltabelle.
- Auflösen von generischen Definitionen
- Bei komplizierteren Sprachen Typ-Inferenz mittels Unifikation

Aufgabe der Zwischencode-Erzeugung

- Übersetzung des Syntax-Baumes in einen Zwischencode, den man als Maschinencode eines hypothetischen Rechners ansehen kann.
- Mögliche Formen des Zwischencodes sind
  - 3-Adress-Befehle (zur weiteren Optimierung)
  - Byte-Codes (zur Interpretation mit einer virtuellen Maschine)
  - verfeinerte Syntax-Bäume
  - viele weitere Möglichkeiten
- Häufig ist diese Phase in mehrere Schritte mit verschiedenen Formen des Zwischencodes aufgeteilt.

Als theoretisches Modell für einen Übersetzungsprozess dienen attributierte Grammatiken und Syntax-gesteuerte Übersetzungsschemata.

Aufgabe der Codeoptimierung

- Verbesserung des Zwischencodes in Hinsicht auf Speicherplatzbedarf und Laufzeit.
- Entfernen überflüssiger Berechnungen, Ersetzen von Laufzeit- durch Compilezeit-Berechnungen.
- Erkennung schleifeninvarianter Berechnungen

Die dazu notwendigen Informationen werden durch das Lösen von komplexen Daten- und Kontrollfluss-Gleichungen gewonnen.
Diese Phase läuft meist in mehreren nacheinander folgenden Schritten ab, die häufig mit der Zwischencode-Erzeugung verzahnt sind.

Aufgabe der Maschinencode-Erzeugung

- Register-Zuordnung und Speicherzuordnung für Variable
- Erzeugen eines äquivalenten Maschinencode-Programms aus transformierten Syntax-Bäumen oder einem Zwischencode-Programm in 3-Adress-Code.
- Eventuell nachfolgende Maschinencode-Optimierung.

Als Modell dienen Baumgrammatiken mit speziellen Formen des Template-Matchings oder des Pattern-Matchings in Verbindung mit dynamischer Programmierung.

Aufgabe der Symboltabelle

- Sammeln aller Informationen über die im Programm auftretenden Variablen. Dazu gehören
  - Speicherplatz
  - Typ-Informationen
  - Gültigkeitsbereich
  - bei Prozeduren Anzahl und Typ der Parameter, Rückgabewert.
- Einfache Möglichkeit zur Aufnahme neuer Informationen und zum Suchen nach Informationen.
- Berücksichtigung der Blockstruktur und der Sichtbarkeitsregeln der Programmiersprache.

Die notwendigen Informationen werden von verschiedenen Teilen des Compilers geliefert.

Probleme bei modernen Prozessoren

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 letzen 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.

Beispiel

Betrachtet man folgendes Programmfragment, so erhält man für verschiedene Werte von `step` unterschiedliche Laufzeiten für die Schleife:

```
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%