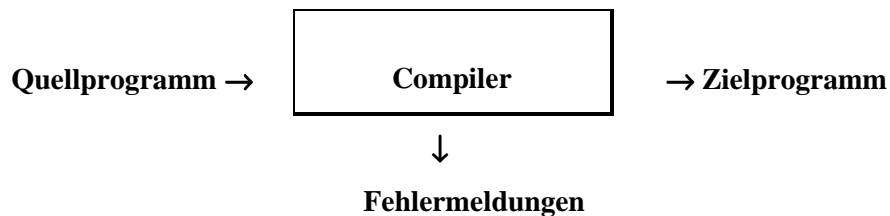


# 1 Einleitung

## 1.1 Compiler

**Def.:** Ein Compiler ist ein Programm, das ein in einer bestimmten Sprache (Quell-Sprache) geschriebenes Programm in ein äquivalentes Programm in einer anderen Sprache (Ziel-Sprache) übersetzt. Eine wichtige Teilaufgabe besteht darin, Fehler im Quellprogramm zu finden.



Es gibt in der Praxis eine Vielfalt an Compilern, was Programmiersprachen, Prozessoren und die Arbeitsweise angeht.

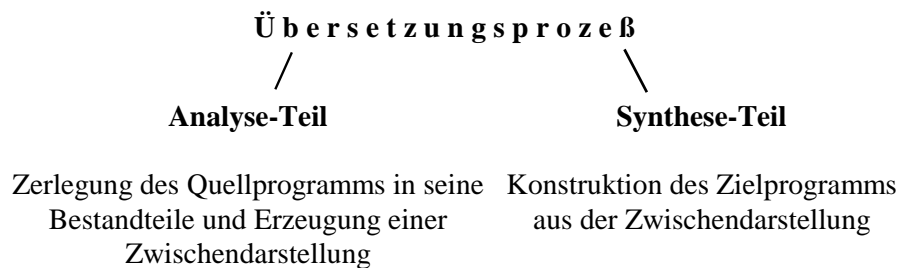
### Klassifikationen:

- Ein-Pass-Compiler
- Mehr-Pass-Compiler
- Load-and-Go-Compiler
- Interpreter
- optimierende Compiler

### Geschichte des Compilerbaus

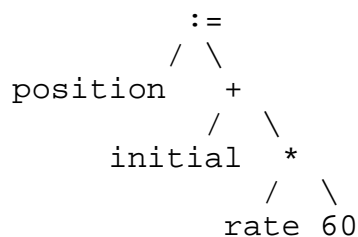
- Beginn zu Anfang der 50er Jahre
- FORTRAN-Compiler (Backus et al. [1957])  
Aufwand 18 Mannjahre
- Entwicklung systematischer Techniken
- Neue Implementierungssprachen
- Programmierumgebungen, Software-Werkzeuge, Compiler-Generatoren

## 1.2 Das Analyse-Synthese-Modell der Compilierung



Bei der Analyse werden die Operationen des Quellprogramms bestimmt und in einer Baumstruktur angeordnet, z.B. einem Syntaxbaum, in dem jeder Knoten eine Operation darstellt und die Söhne eines Knotens die Argumente einer Operation repräsentieren.

**Bsp.:** Der *Syntaxbaum* für das Statement `position := initial + rate * 60` einer fiktiven (Pascal-ähnlichen) Programmiersprache hat folgendes Aussehen



### Beispiele von Programmen, die eine Analyse eines Quellprogramms durchführen:

1. Struktureditor
2. Pretty Printer
3. Interpreter: Direkte Ausführung der Operationen eines Programms nach ihrer Analyse
4. Kommando-Interpreter von Betriebssystemen

### Weitere Anwendungen von Compilerbau-Techniken

Text-Formatierungsprogramm  
 Silicon-Compiler  
 Interaktive Programme

## 1.3 Die Analyse des Quellprogramms

Die Analyse besteht aus drei Teilen:

- lineare Analyse: *lexikalische* Analyse
- hierarchische Analyse: *syntaktische* Analyse
- inhaltliche Analyse: *semantische* Analyse

### Lexikalische Analyse

Der Strom von Zeichen, aus dem das Quellprogramm besteht, wird von links nach rechts gelesen und in Symbole (tokens) aufgeteilt. Ein Symbol ist eine Folge von Zeichen, die zusammen eine bestimmte Bedeutung haben.

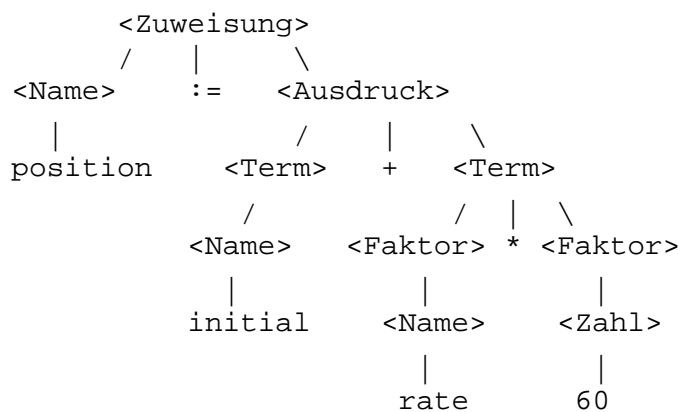
**Bsp.:** Zuweisung wird in folgende Symbole aufgeteilt:

1. Name (position)
2. Zuweisungssymbol :=
3. Name (initial)
4. Plus-Zeichen +
5. Name (rate)
6. Multiplikations-Zeichen \*
7. Zahl 60

### Syntaxanalyse (Parsing)

Die Symbole des Quellprogramms werden zu Gruppen, d.h. Sätzen der Grammatik der Programmiersprache zusammengefaßt. Diese Sätze können durch einen sog. *Parsebaum* oder *Ableitungsbaum* dargestellt werden.

**Bsp.:**



Die hierarchische Struktur eines Programms wird normalerweise durch rekursive Regeln ausgedrückt. Teil der Syntaxanalyse ist auch die *Erkennung von Fehlern*, wenn die eingegebene Symbolfolge nicht der Grammatik der Sprache entspricht.

### Semantische Analyse

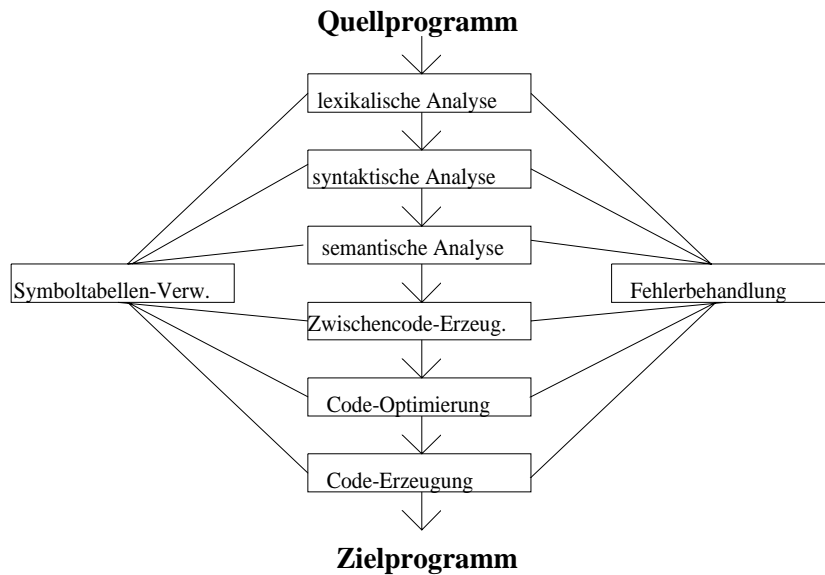
Das Programm wird auf semantische Richtigkeit überprüft, wobei die in der Syntaxanalyse ermittelte hierarchische Struktur zugrunde gelegt wird. Wesentliche Elemente der semantischen Analyse sind etwa

- Typprüfungen
- Eindeutigkeitsprüfungen
- Gültigkeitsprüfungen (Scope)

**Bsp.:** Notwendigkeit der Typumwandlung: integer → real

# 1.4 Die Phasen eines Compilers

**Modellvorstellung:** Ein Compiler arbeitet in *Phasen*, von denen jede das Quellprogramm von *einer* Darstellung in eine *andere* überführt.



## Symboltabellenverwaltung

**Aufgabe:** Speicherung der im Quellprogramm benutzten Namen und von Informationen über deren Attribute (Speicherbedarf, Typ, Gültigkeitsbereich, Wert, ...).

Wann wird was in die Symboltabelle eingetragen?

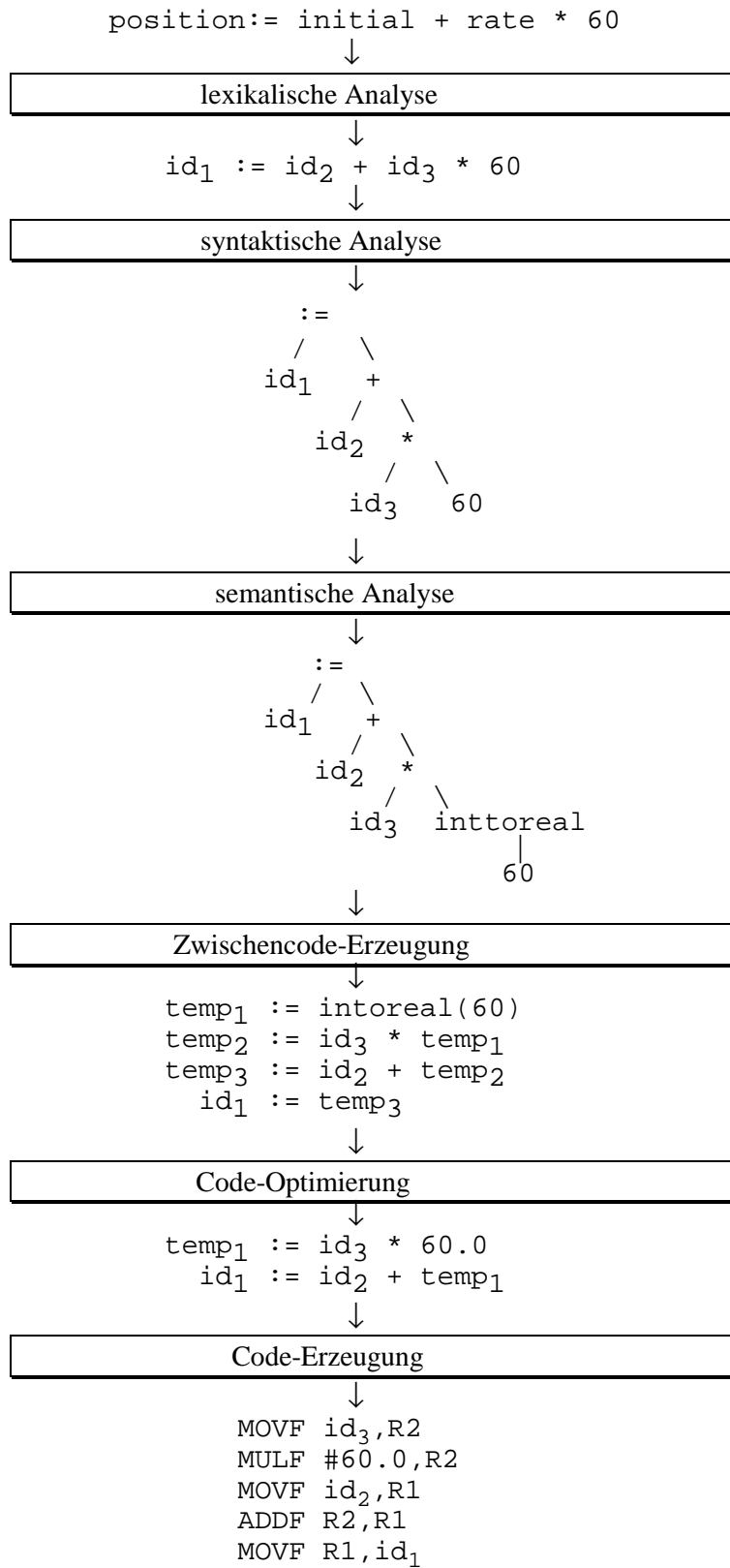
- lexikalische Analyse → Namen
- syntaktische Analyse → Attribute bezügl. Typ, Gültigkeitsbereich
- semantische Analyse →
- Code-Erzeugung → Attribute bezügl. Speicherplatz

## Fehlerbehandlung

Jede Phase kann auf Fehler stoßen. Ein Fehler muß auf geeignete Weise behandelt werden, damit die Compilation fortgesetzt werden kann, um weitere Fehler zu finden.

## Die Analyse-Phase

Wir betrachten hier den Fall der Verarbeitung eines schon früher verwendeten Statements.



Die interne Darstellung der Bäume nach der Syntaxanalyse bzw. der semantischen Analyse kann durch verzweigte Record-Strukturen erfolgen, aber auch nur implizit durch Prozeduraufrufe gesehen.

### Zwischencode-Erzeugung

*Bsp.:* Drei-Adreß-Code

Jede Instruktion besteht aus höchstens drei Operanden.

Jede Instruktion ist eine Zuweisung.

Neben der Zuweisung gibt es höchstens noch einen weiteren Operator.

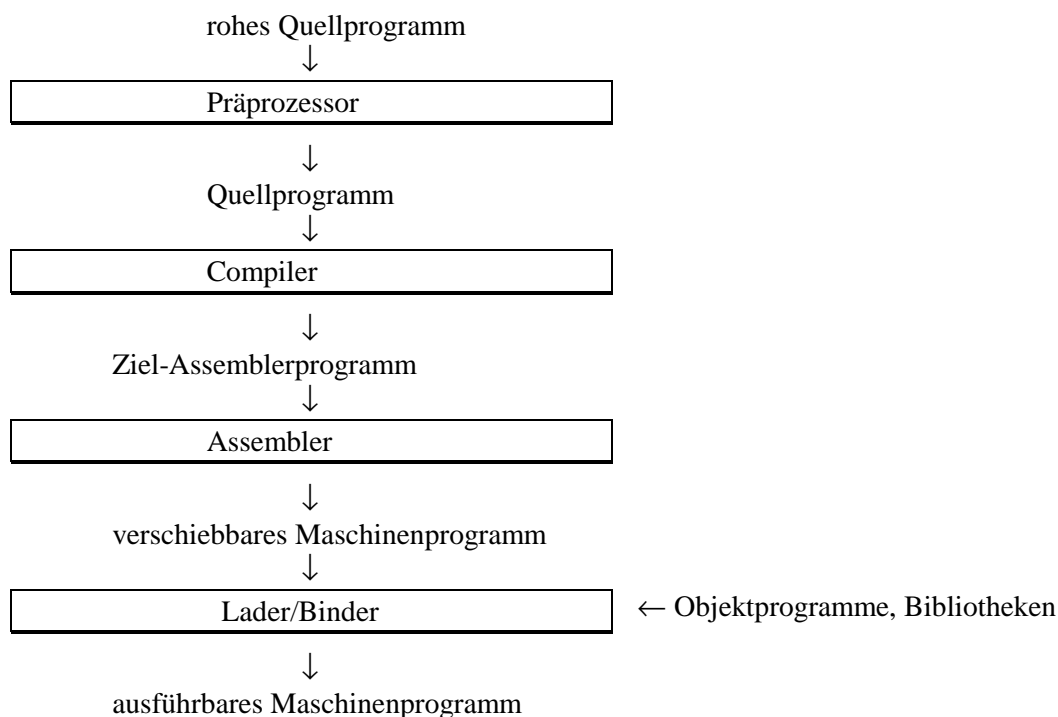
In unserem Beispiel steht F für Floating-Point-Operationen (MULF), # bezeichnet eine Konstante.

### Code-Erzeugung

Erzeugung von Zielcode in Maschinencode oder Assemblercode. Jeder Variablen wird Speicherplatz zugeordnet. Jede Instruktion des Zwischencodes wird in eine funktional gleiche Folge von Maschinenbefehlen übersetzt. Entscheidend ist dabei u.a. die Zuordnung von Variablen zu Registern.

## 1.5 Umgebung eines Compilers

### Typisches Beispiel



### Präprozessor

1. Textersatz: Einfügen von Include-Dateien, Makros, ... (bei C, C++).
2. Bsp. RATFOR → Spracherweiterung.

### Assembler

Assembler-Code: leichter verständliche Form des Maschinencodes mit Namen für Instruktionen und Adressen anstelle binärer Codierungen.

## 1.6 Zusammenfassung von Phasen

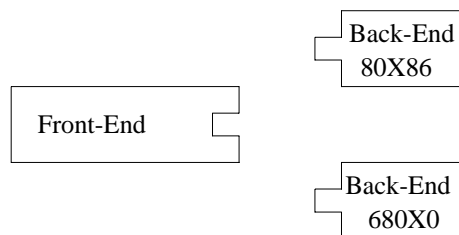
### Front-End / Back-End

*Front-End:* besteht aus denjenigen Phasen, die in erster Linie von der Quellsprache abhängen und weitgehend von der Zielmaschine unabhängig sind: lexikalische Analyse, Erstellung von Symboltabellen, semantische Analyse, Zwischencode-Erzeugung, teilweise Code-Optimierung.

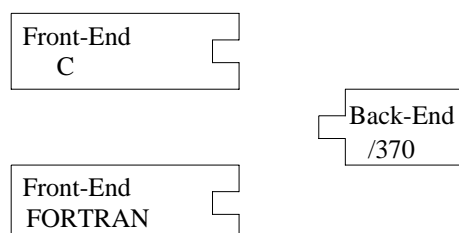
*Back-End:* besteht aus denjenigen Phasen, die sich auf die Zielmaschine beziehen; teilweise Code-Optimierung, Code-Erzeugung.

### Möglichkeiten der Vereinfachung bei der Entwicklung

#### Ein Front-End, mehrere Back-Ends



#### Mehrere Front-Ends, ein Back-End



### Läufe (Pässe)

Üblicherweise werden mehrere Phasen in einem Compiler-Paß zusammengefaßt, der eine *Eingabedatei* liest und eine *Ausgabedatei* erzeugt.

Extremfall: Ein-Paß-Compiler

**Aspekte:** Anforderungen der Sprache, Speicherbedarf (Code, Dateien), Optimierung, Zeitbedarf, Modularität.