Computergrundlagen
Programmieren – Fortran77/90

Maria Fyta

Institut für Computerphysik
Universität Stuttgart

Wintersemester 2012/13

Was sind Programme?

- In dieser Vorlesung: Python, C, Fortran, LaTeX, bash
- Sind das alle Typen von Programmiersprachen?

Wie kann man Programmiersprachen klassifizieren?

- Kein Prozessor versteht Python-, C-, Fortran- oder Shell-Befehle
- Was muss alles passieren, um ein Programm laufen zu lassen?

Wie wird aus unserem Programm etwas, dass der Prozessor ausführen kann?

Was soll das Program leisten?

Zum Beispiel

- eine Liste sortieren
- Echtzeitanalyse von Messwerten
- Computersimulation eines Moleküls

Top-down Analyse

- schrittweises Zerlegen in Teilprobleme
- gibt Einsicht in die Komplexität
- Welche Teile gibt es schon? Bibliotheken, ...

Wer kann mir helfen?

- Wer hat Erfahrung mit ähnlichen Problemen?

Aufwandsabschätzung

- Lohnt sich das Programmieren überhaupt?

Methodenwahl

Wahl der Rechenstrukturen/Algorithmen

- aufteilen in möglichst kleine sinnvolle Einheiten
- möglichst allgemein formulieren $\rightarrow$ wiederverwendbar
- welche bekannten Algorithmen kann ich benutzen?

Wahl von Datentypen und -strukturen

- Ganz-, Fließkommazahlen, Strings, ...
- Listen, Tupel, Wörterbücher, Klassen
- Datenbanken

Wahl der Programmiersprache

- Skriptsprachen für Automatisierung und kleine Aufgaben
- Compilersprachen für sehr rechenaufwändige und oft wiederkehrende Aufgaben

Implementation

Programmieren

- modular programmieren: Teilprobleme in getrennten Funktionen, Modulen, Klassen
- Probleme von außen nach innen bearbeiten (Top–Down)
- Kommentare nicht für „Reservecode“ missbrauchen

Dokumentieren

- schon während des Programmierens
- Kommentare im Code und Anleitung/Hilfe

Testen (Benchmarking)

- möglichst einzelne Teile getrennt testen
- so früh wie möglich
- Tests müssen schnell ausführbar sein

Testen des Programms

Funktion des Programms

- generische Fälle prüfen, etwa Zufallszahlen
- Extremwerte: leere Eingabe, 0, Grundzustand, ...
- Beispiele mit bekannter Lösung (benchmarking)
- z.B. harmonischer Oszillator, ideales Gas, ...
- bekannte implizite Eigenschaften
- z.B. Energieerhaltung, ...

Fehlerbehandlung

- alle Fehlermeldungen sollten in Tests ausgelöst werden
- Verhalten bei nicht erlaubten Eingaben überprüfen

Effizienz

- Laufzeit des Programms bei generischer Eingabe
- Fest- und Hauptspeicherbedarf

Programmiersprachen

Imperative Sprachen

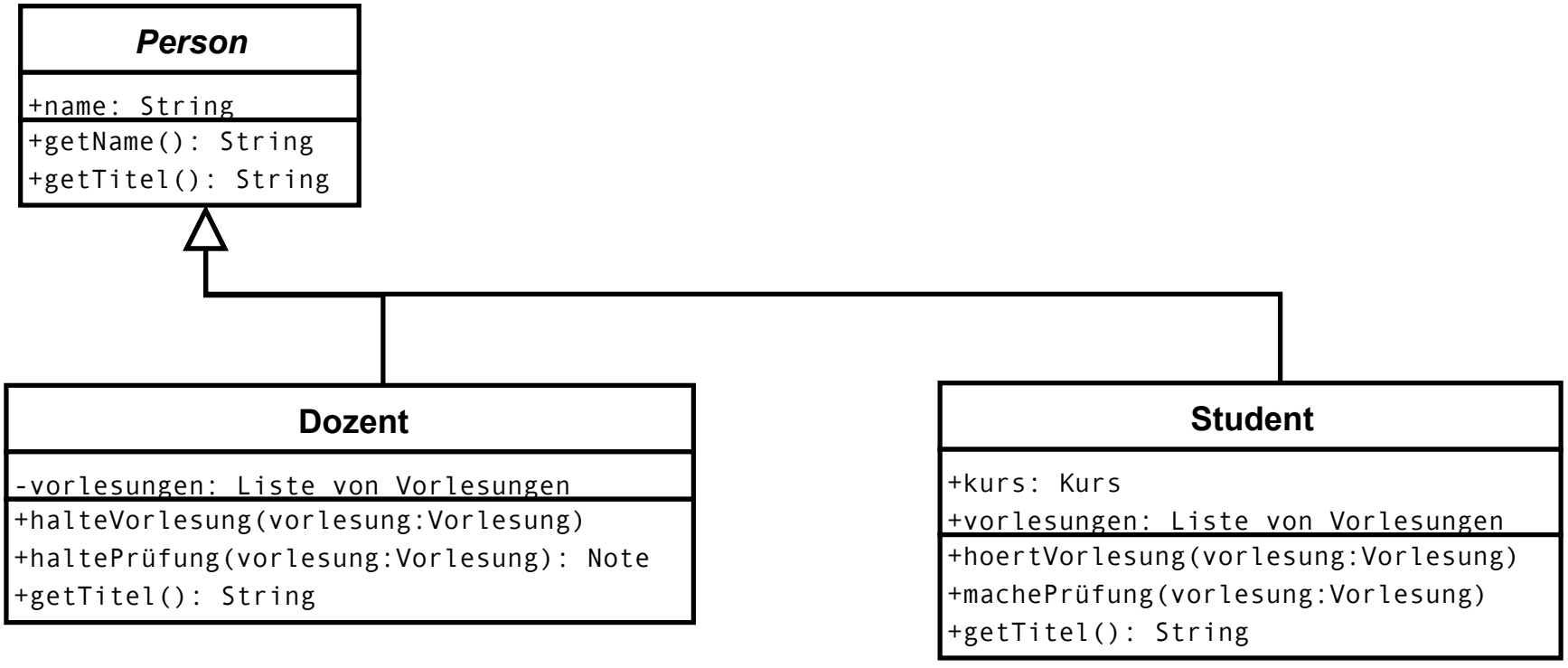
Beispiele: Python, C, Fortran, Shell, BASIC...

- Die meisten Sprachen sind imperativ
- Programme erklären, *wie* ein Problem gelöst werden soll
- Umsetzung des von Neumann-Rechners: Befehle und Schleifen
- prozedural heißen Sprachen, die Prozeduren kennen
  Beispiele: alle außer einfachem BASIC

Deklarative Sprachen

- Keine von Neumann-artigen Befehle, kein innerer Zustand
- Rekursion anstatt Schleifen
- funktional, basierend auf Funktion im mathematischen Sinn
  Beispiele: Haskell, Erlang, Scheme...
- logisch, basierend auf Fakten, Axiomen und logischer Ableitung
  Beispiele: Prolog

Objektorientierte Sprachen

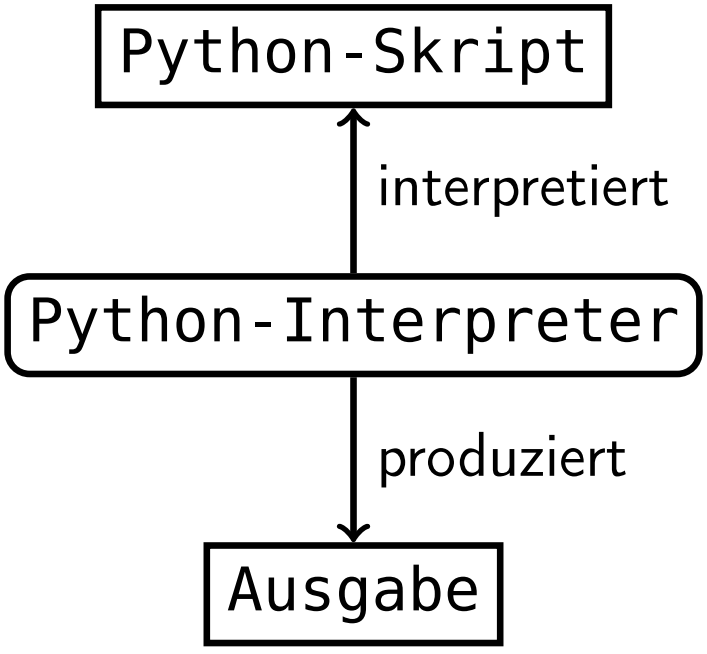


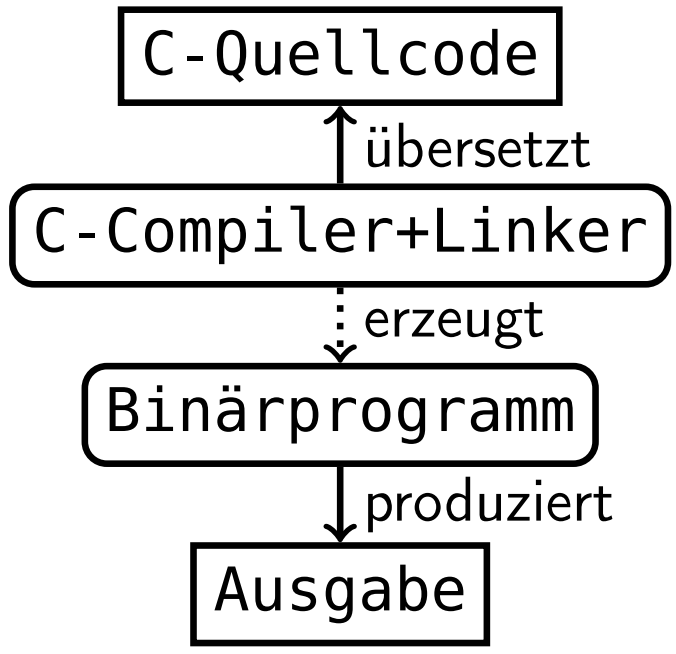
- Objektorientierung ist ein sprachunabhängiges Programmiermodell
- Ziel ist die bessere Wartbarkeit und Austauschbarkeit von Software durch starke Isolation von Teilproblemen
- Speziell darauf ausgelegt sind z.B. C++, Java, Python,...

Terminologie

- Klasse: beschreibt Eigenschaften und Methoden von Objekten
  Beispiel: Klassen sind z.B. Dozent, Student, Person
- Objekt: eine Instanz einer Klasse. Ein Objekt hat stets eine Klasse, aber es kann viele Instanzen geben
  Beispiel: Axel Arnold ist ein Dozent, Maria Fyta auch
- Eigenschaften: Datenelemente von Objekten
  Beispiel: Dozent hat Name, Titel und zu haltende Vorlesungen
- Methoden: Funktionen einer Klasse
  Beispiel: Personen können ihren Namen sagen
- Datenkapselung: Eigenschaften werden nur durch Funktionen der Klasse verändert
  Beispiel: der Name einer Person kann nicht geändert werden
- Vererbung: Klassen können von anderen abgeleitet werden, erben dadurch deren Eigenschaften und Funktionen
  Beispiel: Jede Person hat einen Namen

Interpreter- und Compilersprachen

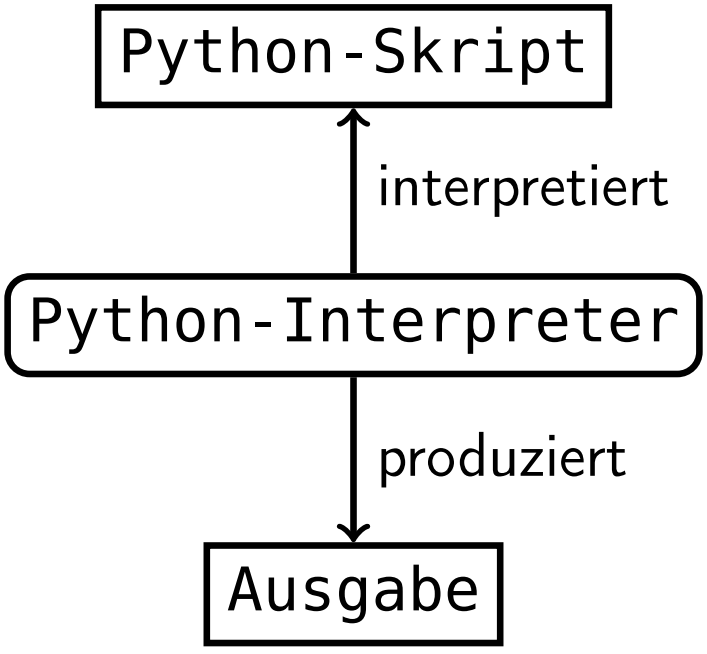


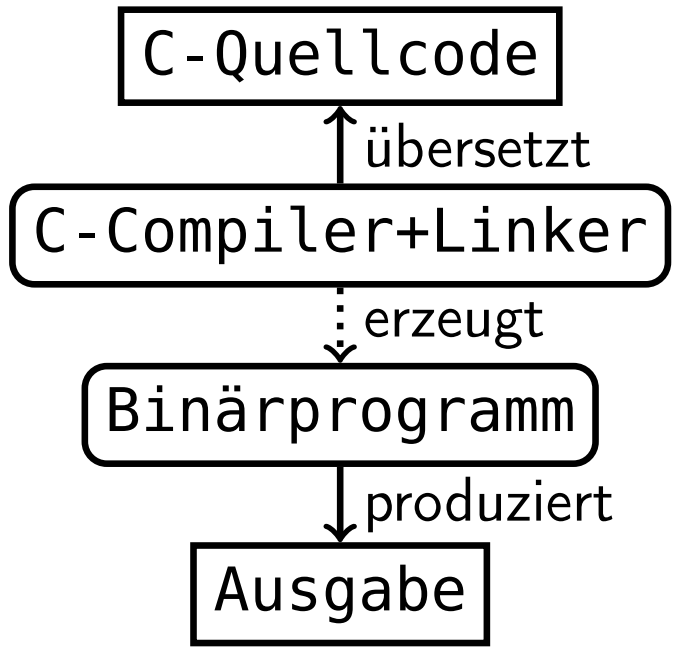


Interpretersprachen

- Z.B. Python, Java, C#, Basic, PHP, Shellsprachen
- Das Programm wird vom Interpreter gelesen und ausgeführt
- Es wird nie in Maschinencode für den Prozessor übersetzt
- Ausnahme: Just-in-Time (JIT) Compiler

Interpreter- und Compilersprachen

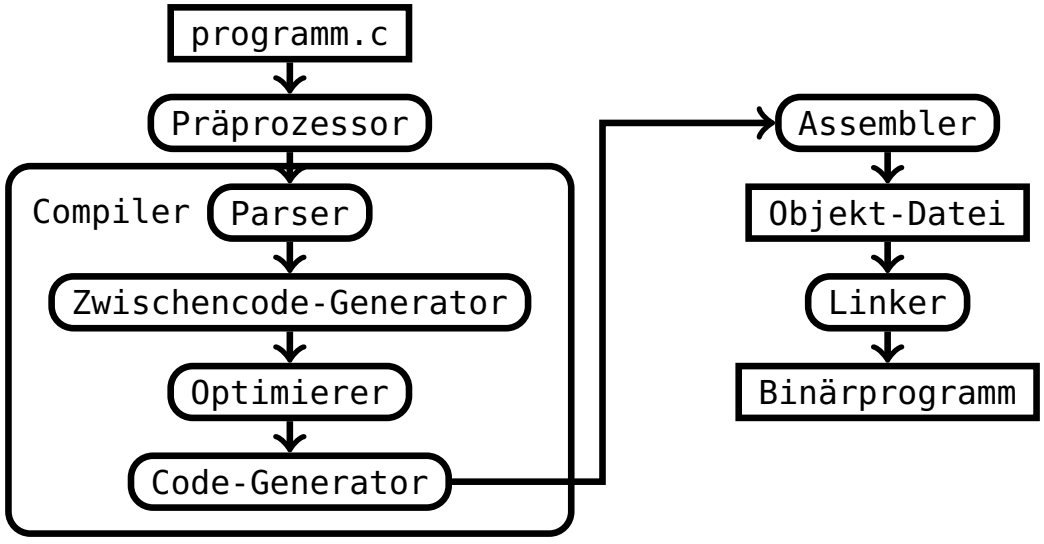




Compilersprachen

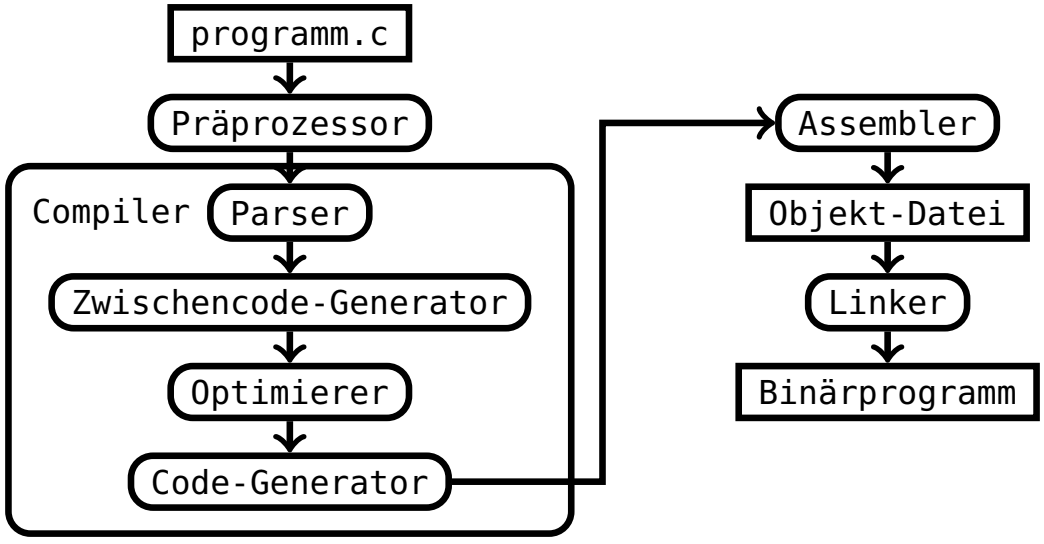
- Z.B. C/C++, Fortran, Pascal/Delphi
- Compiler übersetzt in maschinenlesbaren Code
- Nicht portabel, erschwerte Fehlersuche, deutlich schneller
- Interpreter selbst müssen compiliert werden

Vom Sourcecode zum Programm



- Ein Programm durchläuft viele Schritte bis zur fertigen ausführbaren Datei
- Präprozessor, Compiler, Assembler und Linker sind meist separate Programme
- meist mehrere Objektdateien aus verschiedenen Quelltextdateien

Vom Sourcecode zum Programm



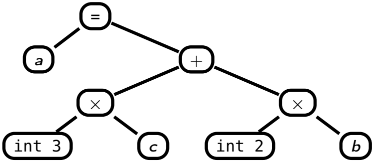
Präprozessor

- Im Prinzip sprachunabhängig, rein textbasiert
- Bindet weitere Quelltextdateien ein
- Erlaubt den Einsatz von *Makros* (Ersetzungen)
  Beispiel: TEST(...) wird überall durch **`if (...)`** *ersetzt*

Der Compiler

- Parser übersetzt den Quellcode in einen Syntaxbaum

$a = 2 * b + 3 * c$ ⇢

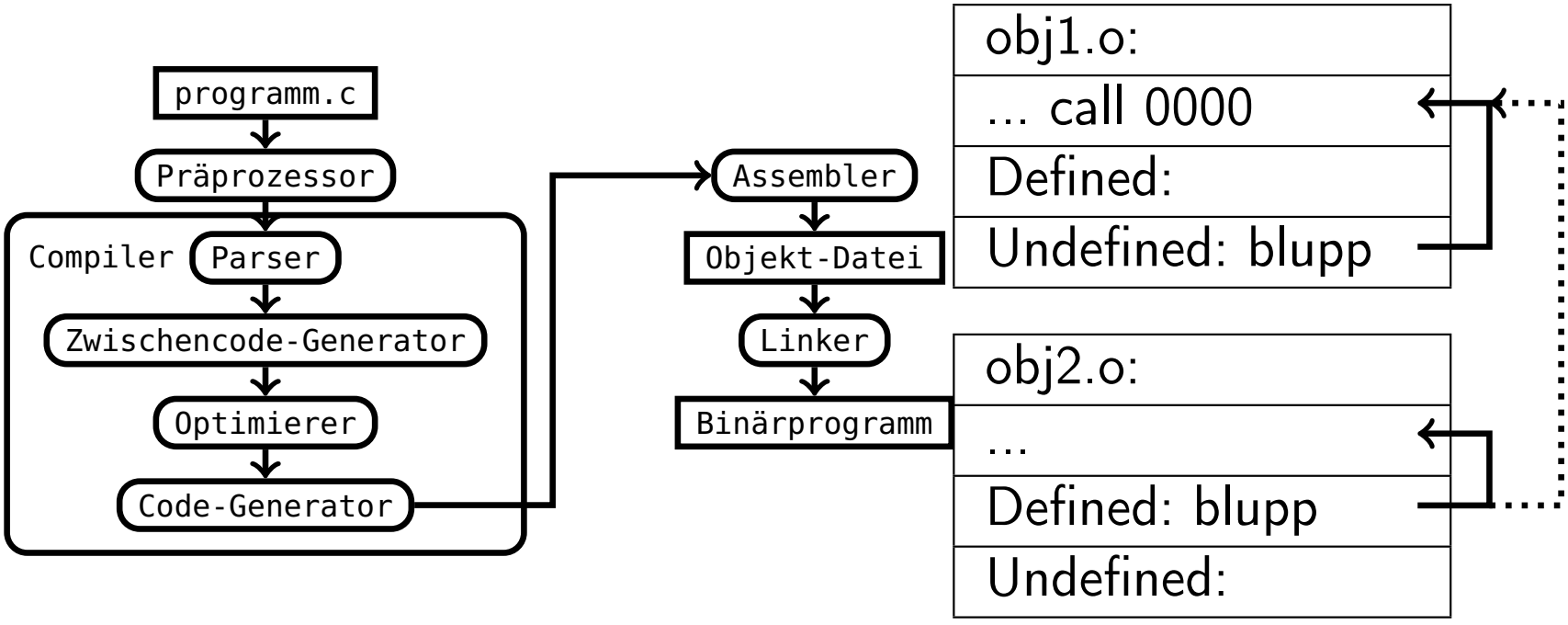


- Zwischencode-Generator erzeugt daraus Pseudocode, etwa:

$r1 = b$	$r2 = r1 \times 2$	$r3 = c$	$r4 = r3 \times 3$	$r5 = r2 + r4$	$a = r5$

- Optimierer versucht, diesen zu verbessern, z.B. durch
  - Prozessorregisterzuweisung
  - Einfügen kurzer Funktionen, Entrollen von Schleifen
  - Suche nach gemeinsamen Termen, ...
- Code-Generator erzeugt Assembler aus dem Zwischencode

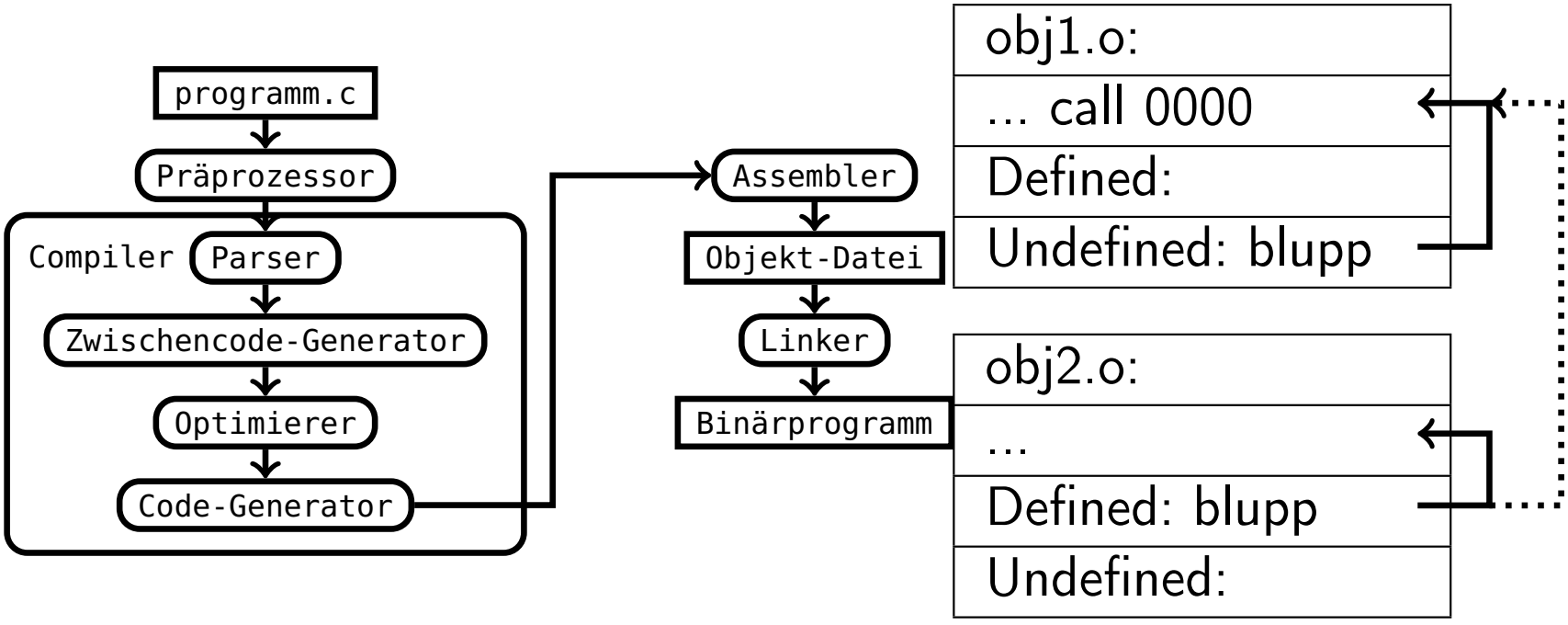
Der Assembler



Der Assembler erzeugt eine Objektdatei mit

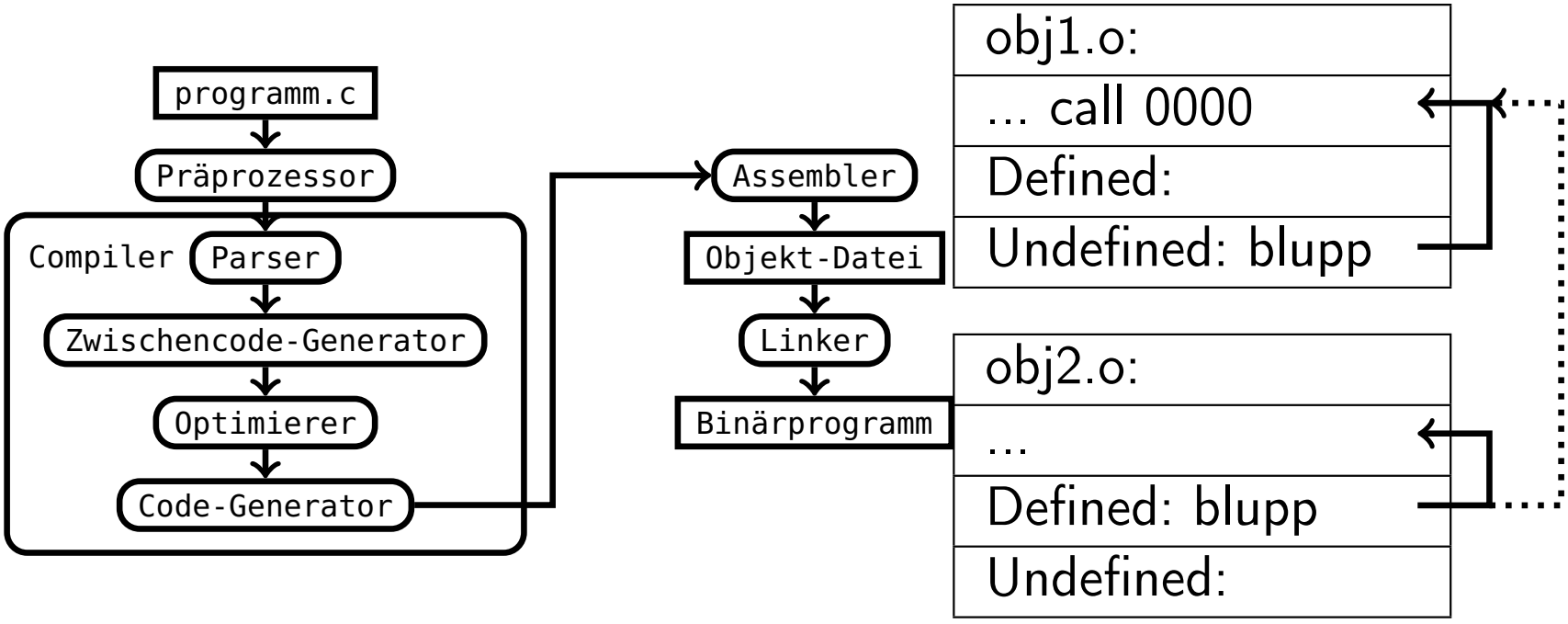
- Maschinencode
- den Speicherpositionen von globalen Variablen und Funktionen
- Stellen, an denen Stellen Information über solche Positionen aus andere Objektdateien benötigt wird

Der Linker



- Verbindet mehrere solcher Objektdateien zu einer ausführbaren Datei (Binary) oder einer Bibliothek
- Bibliotheken sind Sammlungen von Objektdateien
- Der Linker verbindet Zugriffe zwischen Objektdateien
- Nur wenn alle Zugriffe aufgelöst wurden, entsteht ein Binary

Dynamisches Linken



- Beim dynamischen Linken wird das Linken gegen die Bibliotheken erst beim Starten des Programms erledigt.
- Dadurch können Teile des Programms geupdated werden
- Und mehrfach benutzte Bibliotheken werden nur einmal geladen
- Erfordert spezielle dynamische Bibliotheken (.so, .dylib oder .dll)

Programmiersprache Fortran (77/90)

Literaturempfehlung:
Vorlesungsskript von Heidrun Kolinsky zu FORTRAN 90/95:
[www.rz.uni-bayreuth.delehrefortran90vorlesungindex.html]
G. Schmitt (1996): Fortran–90–Kurs technisch orientiert, R. Oldenbourg Verlag, München
W. Brainerd, C. Goldberg and J. Adams (1996): Programme's Guide to Fortran 90, Springer–Verlag M. Kallweit & F. Kindermann, 'Crashkurs Fortran'

Fortran (FORmula TRANslation)

Geschichte

- die erste jemals tatsächlich realisierte höhere Programmiersprache (1953: J.W. Backus – IBM)
- Fortran ist eine prozedurale und in ihrer neuesten Version zusätzlich eine objektorientierte Programmiersprache
- Fortran wird insbesondere für numerische Berechnungen eingesetzt
- mehrmals erweitert (FORTRAN I, FORTRAN II, FORTRAN IV, FORTRAN-66, FORTRAN-77, Fortran 90, Fortran 95, Fortran 2000, Fortran 2003, Fortran 2008 und zuletzt Fortran 2010)
- zahlreiche Sprachelemente wurden aus später entstandenen Programmiersprachen übernommen
- verschiedene Varianten; darunter HPF (high performance Fortran)

Eigenschaften

- für numerische Berechnungen vorgesehen und optimiert; umfangreiche Bibliotheken
- restriktiver als C und kann leichter optimiert werden
- Potenz-Operator ** (in vielen Hochsprachen nicht vorhanden)
- Fortran90: Vektor- udn Matrix-Operationen wurden standardisiert
- Compiler unterscheidet nicht zwischen Groß- und Kleinschreibung (case insensitive")
- Ab Fortran90 ist die dynamische Speicherverwaltung standard

Compiler

- Kommerzielle: IBM SUN, HP, Intel, Absoft, PGI, NAG
- Freie: GNU Fortran, OpenWatcom
- Transcompiler(f2c): Übersetcung von Fortran77 in C (NAG benutzt als Zwischensprache C)
- `ftnchek` Programmierwerkzeug für eine separate Prüfung der Übereinstimmung von Argumentlisten

Namensyntax

- Keine Schlüsselwörter (z.B. program, integer, real, do, if, else, ...) dürfen verwendet werden.
- Ein Name darf in Fortran bis zu 31 Zeichen lang sein und muss mit einem Buchstaben beginnen.
- Für die auf das erste Zeichen folgenden Zeichen dürfen alle alphanumerischen Zeichen sowie Unterstriche verwendet werden.

Programmstruktur

Programmkopf: `program` <Programmname>
Deklarationsteil: besteht aus einer Reihe von Konstanten- und Variablenvereinbarungen.
Anweisungsteil: eine strikte Aneinanderreihung von auszuführenden Befehlen (kann mit Hilfe von Flussdiagrammen modelliert werden); Zeilennummern können auf Anweisungen hinweisen
Programmende:`end program` <Programmname>

Program Beispiele

Hello World

```
program hello
write (* ,*) 'Hello World !'
end program hello
```

Umrechnungstabelle: Fahrenheit nach Celsius

```
program celsius_table
implicit none
real :: Fahrenheit , Celsius
write (* ,*) ' Fahrenheit Celsius '
write (* ,*) ' ----------'
do Fahrenheit = 30.0 , 220.0 , 10.0
Celsius = (5.0/9.0) * ( Fahrenheit -32.0)
write (* ,*) Fahrenheit , Celsius
end do
end program celsius_table
```

Variablendeklaration

- Variablen: standardmäßig über ihren Anfangsbuchstaben deklariert

  Bezeichner die mit einem Charakter von i-n beginnen stehen für eine INTEGER-Variable oder einen INTEGER-Funktionswert
  alle übrigen Bezeichner stehen für REAL-Werte

  - Änderung der Vordefinition von Variablen:
    `implicit` <Variablentyp> <Variablenbereich>
  - Beispiele
    - IMPLICIT NONE: keine implizite Deklaration. Durch diesen Befehl müssen alle benutzten Variablen explizit definiert werden
    - IMPLIZIT CHARACTER(LEN=20) (H-I,L-N): alle nichtdeklarierte Bezeichnet, deren erster Buchstabe H-I oder L-N ist, bezeichnet ein Zeichen
    - IMPLICIT COMPLEX (x,c,z) : alle nichtdeklarierten Bezeichner, deren erster Buchstabe x oder c oder z ist, bezeichnen komplexe Zahlen
    - IMPLIZIT REAL*8 (A-F): alle nichtdeklarierten Bezeichner, deren erster Buchstabe A-F ist, bezeichnen reelle Zahlen

Deklarationen

Variablentypen und-wertebereiche

Datentyp	Erweiter.	Erläuterung	Wertebereich
logical			
integer		ganze Zahlen	(-2^7+1) - (2^7+1)
		ganze Zahlen	$(-2^{15}+1)$ - $(2^{15}+1)$
	(Standard)	ganze Zahlen	$(-2^{31}+1)$ - $(2^{31}+1)$
real	*4 (Standard)	reelle Zahlen	$-10^{38.53}$-$10^{38.53}$
	*8	reelle Zahlen	$-10^{308.25}$-$10^{308.25}$
character	(Standard)	einzelnes Zeichen	alle möglichen Zeichen
	(LEN=x)	Zeichenkette Länge x	alle möglichen Zeichen

Quelle: M. Kallweit & F. Kindermann, 'Crashkurs Fortran', Uni. Würzburg

Variablenvereinbarung:

<Variablentyp> :: <Variable>

Konstantenvereinbarung:

`parameter` (<Variable>=<Variablenwert>,...)

Eingabe/Ausgabe

Open/Close

```
open (unit,file,status,action,iostat)
```

- `unit=<unitnumber>`: eine nichtnegative integer-Zahl
- `file=<Dateiname>`: Dateiname in Form einer Zeichenkette mit den entsprechenden Anführungs und Schlusszeichen
- `status=<Schlüsselwort>`: Zustand der Datei. Optionen:'old', 'new', 'unknown','replace','scratch'
- `action=<Aktionsangabe>`: was mit der Datei geschehen soll. Optionen: 'read', 'write' 'readwrite'
- `iostat=<Name einer Variablen vom Datentyp integer>)` : wie erfolgreich das Betriebssystem, eine Verknüpfung zwischen der unit number und der angegebenenen Datei auf der Festplatte herstellen konnte (reibungslos $\rightarrow$ Rückgabewert=0)

Eingabe/Ausgabe

Open/Close

close (unit=<unit number> oder `close <unit number>`

`Delete` option:
`close (unit=<unit number>,status='delete'`

- Standard input: 5 und 100
- Standard output: 6 und 101
- Standard error: 0 und 102

Read/write/close

- read(unit,file,error,iostat,slist)
- write(unit,format,iostat)
- close(unit)

`slist:iostat, status, err, form, blank,...`

Falls keine 'unit' für `read` gegeben, erzeugt z.B. write(20,*) eine fort20.dat Datei.

- Option `end`
  Beispiel: `read(10,*,end=100)` $\rightarrow$ `read` am Ende der Datei wird die Anweisung mit Zeilennummer 100 in der Quelldatei durchgeführt (z.B. 100 print*,'end of file').

Print

`print*, 'Text', Variable`

[Ausgabe nur auf dem Bildschirm.]

Lesen und schreiben

read(*,*)<Variable>	Liest den vom Benutzer einzugebenden Wert der <Variable> von der Konsole ein
write(*,*)<Variable>	Schreibt den Wert der <Variable> unformatiert auf die Konsole
write(*,*)'<Text>'	Schreibt den eingegebenen <Text> auf die Konsole
write(*,'(f6.3)')<Variable>	Schreibt den Wert der <Variable> im Format f6.3 auf die Konsole

Formatierungsbefehle

- i4 : Integerzahl mit maximal 4 Stellen
- a7 : Character mit maximal 7 Stellen
- f6.3 : Realzahl mit insgesamt 6 Stellen (inkl. Trennzeichen) und 3 Nachkommastellen
- 3x : 3 Leerzeichen

Wertzuweisungen

Befehl:
<Variable>=<Variablenwert>
Der <Variablenwert> muss nicht unbedingt eine feste Zahl/Zeichenkette sein. Er kann sich genauso aus der Verknüpfung meherer anderer Variablen zusammensetzen.
Beispiele:

```
x=3
x=y*exp(z)
x=y+z+abs(c)
a=b+sqrt(c)
a=max(b,c)
d=mod(3,2)*sign(-1)
...
```

Kontrollstrukturen

Schleifen

do-Schleifen realisieren die Wiederholung von Anweisungen

- Befehl:

```
do <Laufindex>=<von>,<bis>(,<Schrittweite>)
<Anweisungsblock>
enddo
```

  Standardmässig ist als Schrittweite 1 eingestellt.

- Beispiel:

```
do i=2,5
write(*,*) i,i**i,log(i)
enddo
```

Kontrollstrukturen

Verzweigungen

Sollen Anweisungen nur unter einer Bestimmten Bedingung ausgeführt werden, so können diese Anweisungen in ein `if`-Statement geschrieben werden.

- Befehl1:
```
if(<Bedingung1>)then
(elseif(<Bedingung2>)then )
else <Anweisungsblock3>)
endif
```
- Befehl2:
alternativ: `if(<Bedingung>)<Anweisung>`

Verzweigung - Beispiele

- ```
  if(i.ge.0)then
  param=1
  else
  param=2
  endif
  ```
  Wenn i größer als 0 ist, dann nimmt param1 die Zahl 1
- ```
  j=1; sum=0
  do
  summe=summe+j ; j=j+1
  if(j.gt.i)exit
  enddo
  ```

Unterprogramme

[Unterprogramme müssen deklariert werden]

SUBROUTINE

- Programmabschnitte können definiert werden, die in einem Programm mehrmals in gleicher Form verwendet werden können.
- Programmabschnitte können definiert werden, die in einem Programm mehrmals in gleicher Form verwendet werden können
- die Übergabe von Variablen ist möglich
- besitzt jedoch keinen Rückgabewert

FUNCTION

- dient zur Berechnung eines Wertes
- an eine Funktion können mehrere Werte übergegeben werden, die zur Bestimmung des Funktionswerts benötigt werden.
- alle eingegebene Variablen müssen deklariert werden

Unterprogramme - Befehle

SUBROUTINE

- subroutine <Subroutinenname>((<Variable1>,...))
  <Variablendeklaration>
  <Programmabschnitt>
  end subroutine <Subroutinenname>

FUNCTION

- <Variablentyp> function <Funktionsname>((<Variable1>,...))
  <Variablendeklaration>
  <Werteberechnung>
  end function <Funktionsname>

SUBROUTINE: Beispiel

```
subroutine sum1(a,b)
integer a,b
b=b+a
print*,a,b
end subroutine sum1
```

FUNCTION: Beispiel

```
integer function sum1(a,b)
integer :: a,b
sum1=a+b
end function sum1
```

Aufruf von SUBROUTINE und FUNCTION

```
!sum.f90
 !Performs summations and saves input and summation in a file
 program summation
 implicit none
 integer :: sum, a, sum1
 print*, 'This program performs summations. Enter 0 to stop.'
 open(unit=10, file="sum.dat")
 sum = 0
 do
 print*, "Value:"
 read*, a
 if (a == 0) then
 exit
 else
 call sum1(a,sum) !<== subroutine call
 sum=sum1(sum,a) ! <== function call
 end if
 write(10,*) a
 end do
 print*, "Sum =", sum
 write(10,*) "Sum =", sum
 close(10)
 end program summation
```

Sonstige Anweisungen

- `call system ((<Variable1>,...))` : durchführt einen Befehl des Betriebsystems
- `COMMON block (Fortran77), Module (Fortran90)`: Informationen zwischen einzelnen Programmeinheiten (Hauptprogramm und Unterprogrammen) teilen und austauschen
  - Beispiel: π

```
module kreiszahl
implicit none
save
real , parameter :: pi = 3.141593
end module kreiszahl
subroutine kreisumfang (r,u)
use kreiszahl
```

usw...

Sonstige Anweisungen

- `Do while`: Anweisungsblock ausgeführt wenn logischer Ausdruck den Wert .true. aufweist ; dann erneut geprüft

```
do while (< logischer Ausdruck >)
Anweisungsblock
end do
```

```
program do_while
implicit none
integer :: i = 1, n = 1
do while (write (* ,*) i, n n = n + i
i = i + 1
end do
end program do_while
```

- `Go to (Fortran77), cycle (Fortran90)` realisiert bedingungsabhängige Sprünge innerhalb des Anweisungsteils einer Programmeinheit

Sonstige Anweisungen

- `ALLOCATABLE/ALLOCATE/DEALLOCATE`: dynamische Speicherzuweisung

```
 MODULE data_array
INTEGER n
REAL, DIMENSION(:,:,:), ALLOCATABLE :: data
END MODULE
PROGRAM main
USE data_array
READ (input, *) n
ALLOCATE(data(n, 2*n, 3*n), STAT=status)
:
DEALLOCATE (data)
```

- ...und vieles mehr...

Frohe Weinachten
und
einen guten Rutsch!!!