Compilertechnologie

by

Dr. Günter Kolousek

(C/C++-) Übersetzungsmodell

1. Präprozessor (preprocessor)
   - Input: `.c`, `.cpp`, `.h`
   - Aufgabe: Headerdateien einbinden, Macros expandieren
   - Output: `.i` (C), `.ii` (C++)
2. Übersetzer (compiler)
   - Aufgabe: Übersetzung in Assembler (oder gleich Maschinenspracheanweisungen)
   - Output: `.s`
3. Assembler (assembler)
   - Aufgabe: Übersetzung in Maschinensprache
   - Output: `.o` (Windows: `.obj`)
4. Linker (linker, link editor)
   - optional zusätzlicher Input: `.a` (Windows: `.lib`)
   - Aufgabe: Erstellung eines ausführbaren Programmes
   - Output: keine Endung (Windows: `.exe`)

Präprozessor

- Aufgabe: Textersetzung
  - Headerdateien inkludieren

    ```
    #include ”mathutils.h”
    ```
  - Macros expandieren

    ```
    #ifndef MATHUTILS_H
    #define MATHUTILS_H

    #endif
    ```

Prinzip eines Compilers

- Compiler: Quelltext $\rightsquigarrow$ Übersetzer $\rightsquigarrow$ Zielcode
  - Quelltext, z.B.: C, C++, Go, Objective-C, D, Modula,…, Java, C#, Python, Ruby,…
  - Übersetzer: Fehlermeldungen sind wichtig!
  - Zielcode: Zwischencode, Assembler, Maschinencode
- Syntax und Semantik muss bekannt sein

Anwendungen eines Compilers

- Programmiersprachen: Compiler (vs. Interpreter)
- Pretty print Programme
- Query-Interpreter, z.B. SQL
- Silicon-Compiler: Programm in Schaltplan
- Strukturediteren, wie z.B. in Eclipse oder Netbeans
- Textformatierung mit Steuerzeichen, z.B. Postscript, PDF, RTF, LaTeX

Anforderungen an Compiler

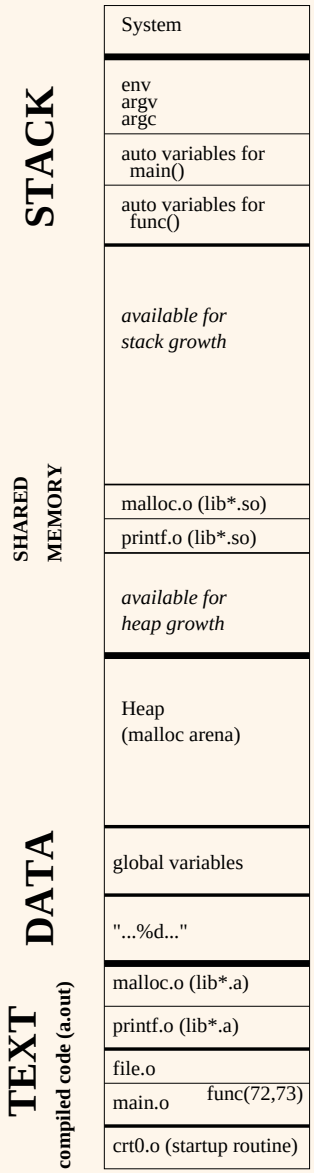
- *muss* fehlerfrei übersetzen
- *muss* jeden Fehler im Programmcode erkennen und zuverlässig melden
- *soll* möglichst effizienten Code erzeugen
- *soll* möglichst schnell übersetzen

Struktur des Übersetzungsmodelles

1. (Coding)
2. Preprocessor
   - Input: Programmtext (source code)
   - Output: Programmtext
3. Compiler
4. Assembler
5. Linkeditor (linker)
   - zusätzlicher Input: Bibliotheken (libraries)
   - statisches Linken vs. dynamisches Linken
   - Speicherlayout (memory layout)
6. (Loader)

Speicherlayout

Memory Layout (Virtual address space of a C process)



High memory

← mfp − frame pointer (for main)

← stack pointer (grows downward if func() calls another function)

library functions if dynamically linked (usual case)

← brk point

uninitialized data (bss)

initialized data

library functions if statically linked (not usual case)

← ra (return address)

Low memory

Stack illustrated after the call func(72,73) called from main(), assuming func defined by:

```
func(int x, int y) {
  int a;
  int b[3];
 /* no other auto variables */
```

Assumes int = long = char * of size 4 and assumes stack at high address and descending down.

Expanded view of the stack

Stack

Offset from current frame pointer (for func())	main() auto variables	Contents
+12	73	y
+8	72	x
+4	ra	return address
0 ← frame pointer points here	mfp	caller's frame pointer
−4	garbage	a
−8	garbage	b[2]
−12	garbage	b[1]
−16	garbage	b[0]
← stack pointer (top of stack) points here		

All auto variables and parameters are referenced via offsets from the frame pointer.

The frame pointer and stack pointer are in registers (for fast access).

When funct returns, the return value is stored in a register. The stack pointer is move to the y location, the code is jumped to the return address (ra), and the frame pointer is set to mfp (the stored value of the caller's frame pointer). The caller moves the return value to the right place.

Speicherlayout – 2

- uninitialized data

```
int x;  // will be initialized by exec!
void f() {
    static double y; //initialized by exec!
}
int main() {}
```

- initialized data

```
int i{123};

char s[]{"abc"};  // inside read/write area

const char* pc{"abc"};  // inside read-only

int main() {}
```

Phasenmodell (des Compilers)

1. Analysephase
   - Überprüfung der Syntax
   - Aufbau der Zwischendarstellung
   - besteht aus:
     1.1 Syntaxanalyse
     1.2 Semantischer Analyse
2. Synthesephase
   - Erzeugung des Maschinencodes
   - eventuell: Codeoptimierung

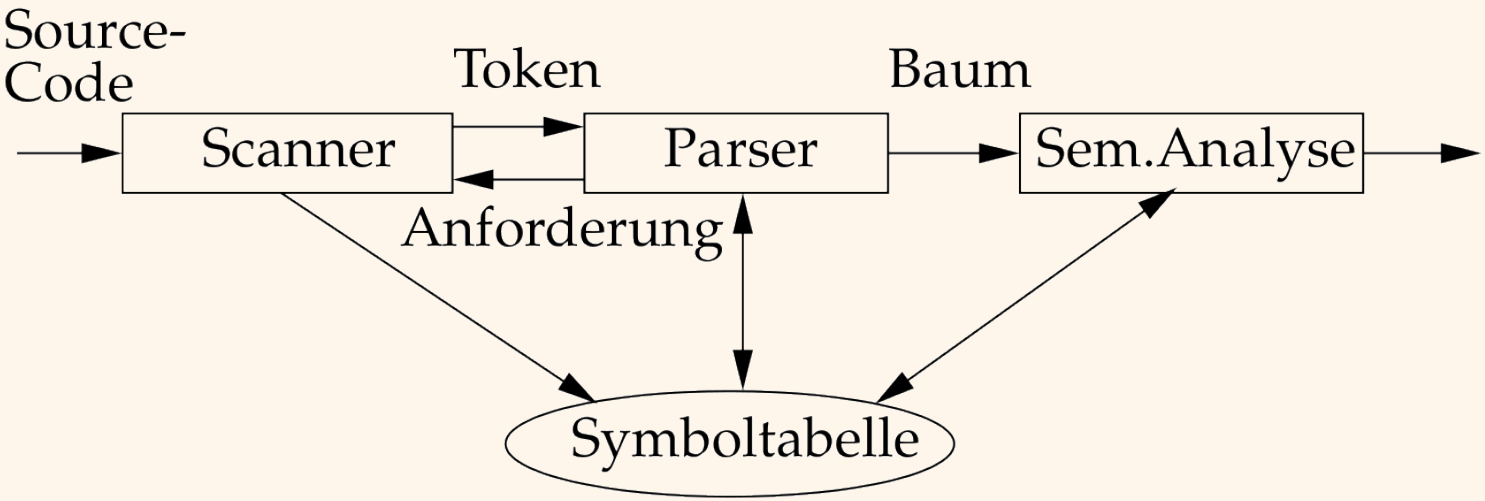
Syntaxanalyse

1. lexikalische Analyse (Scanner oder Tokenizer)
   - liest Quelltext als eine Folge von Zeichen
   - erzeugt Liste der lexikalische Tokens (Symbole)
     - Identifier
     - Keyword
     - Operator
     - Delimiter
     - Literal
   - baut Symboltabelle auf ($\leadsto$ Namen)
2. syntaktische Analyse (Parser)
   - überprüft Grammatik
     - Eingabe dann syntaktisch korrekt, wenn Syntaxbaum bzw. eine Ableitung gemäß Regeln der Grammatik konstruierbar.
   - erzeugt Syntaxbaum
   - ergänzt Symboltabelle um Art und Typ

Semantische Analyse

- inhaltliche Prüfung des Quelltextes
  - Überprüfung der Deklarationen
  - Berechnung von Typkonversionen
  - Eindeutigkeitsprüfung
  - Gültigkeitsprüfung
- erzeugt Zwischencode
- ergänzt Symboltabelle um Gültigkeitsbereich

Scanner-Parser-Semantische A.



Synthesephase

1. Codeoptimierung
   - erzeugt optimierter Zwischencode
   - Beispiele der Optimierung
     - Algebraische Vereinfachung:
       `x * 1` $\rightsquigarrow$ `x`
       `x * 2 ** i` $\rightsquigarrow$ `x << i`
     - Unterdrückung von Laufzeitüberprüfungen:
       z.B. Indexüberprüfung, wenn Index immer im gültigen Bereich
     - Fortpflanzen von Zuweisungen:
       `x = a + b; y = a + b;` $\rightsquigarrow$ `x = a + b; y = x;`
     - Entfernen von redundanten Codeteilen
     - Entfernen von Codeteilen, die nicht durchlaufen werden
2. Codeerzeugung
   - erzeugt Zielcode

Arten von Fehlern

- lexikalische Fehler
  - falsches Anfangszeichen bei Identifier
  - Literal nicht im zulässigen Wertebereich
- syntaktische Fehler
  - schließende Klammer fehlt
- semantische Fehler
  - falsche Typen für Operator
- logische Fehler

Formen von Compilern

- Einteilung in Phasen (Durchgang, engl. pass)
  - gemäß folgender Kriterien
    - Anforderungen der Programmiersprache
    - Speicherbedarf (Code, Daten)
    - Optimierung
    - Zeitbedarf der Übersetzung
    - Modularität des Übersetzers

Formen von Compilern – 2

- Einteilung
  - Ein-Pass-Compiler
  - Mehr-Pass-Compiler
    - Front-End: hängen von Quellsprache ab, Analysephase bis tw. Codeoptimierung
    - Back-End: maschinenabhängig, tw. Codeoptimierung, Codeerzgeugung
  - Vor-/Nachteile: Geschwindigkeit, Anpassung an HW und Quellsprache
  - Beispiele:
    - Frontend: C, Backends: Win, Linux
    - Frontends: C, Java Backend: Linux

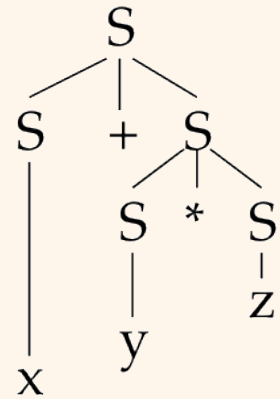
Formen von Compilern – 3

- Compiler
  - Beispiele: C, C++, D, Go, Eiffel, Fortran, Ada, Cobol
- Interpreter
  - Jede Anweisung und Deklaration: sequentiell analysiert und unmittelbar ausgeführt
    - Beispiele: Basic, Tcl, Bash
  - Zwischenform: Übersetzung in Zwischencode
    - interpretieren
    - tw. Übersetzung in Maschinensprache zur Laufzeit (JIT)
    - Beispiele: Java, C#, Python, Ruby, LISP, Prolog

Syntaktische Analylse – Beispiel

- $G = (\Phi, \Sigma, P, S)$
  $\Phi = \{S\}$
  $\Sigma = \{x, y, z\}$
  $P = \{S \to S + S | S * S | (S) | x | y | z\}$
  $S = S$
- Ableitung 1

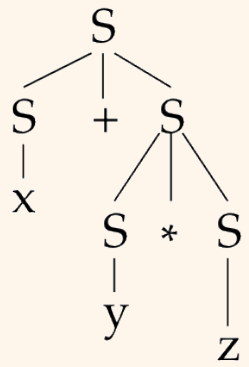
$$S \Rightarrow S + \underline{S} \Rightarrow S + S * \underline{S} \Rightarrow S + \underline{S} * z \Rightarrow \underline{S} + y * z \Rightarrow x + y * z$$



Syntaktische Analyse – Beispiel – 2

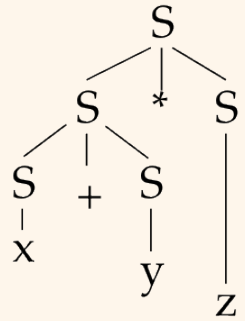
- Ableitung 2

$$S \Rightarrow \underline{S} + S \Rightarrow x + \underline{S} \Rightarrow x + \underline{S} * S \Rightarrow x + y * \underline{S} \Rightarrow x + y * z$$



- Ableitung 3

$$S \Rightarrow \underline{S} * S \Rightarrow \underline{S} + S * S \Rightarrow x + \underline{S} * S \Rightarrow x + y * \underline{S} \Rightarrow x + y * z$$



Syntaktische Analyse – Beispiel – 3

- Ableitung 1 ist ungleich Ableitung 2
- Ableitung 1 ist nur unwesentlich verschieden zu Ableitung 2
  - da Syntaxbäume gleich sind
- Syntaxbaum 3 ist ungleich Syntaxbaum 1 und ungleich Syntaxbaum 2

Begriffe

- Eine KF Grammatik heißt **mehrdeutig**, wenn
  - es zu mind. 1 einem ableitbaren Wort 2 verschiedene Syntaxbäume gibt.
- Eine Ableitung heißt **linkskanonisch**, wenn
  - in einem Ableitungsschritt das jeweils am weitesten links stehende Non-Terminalsymbol ersetzt wird.
- Es gilt: Eine KF Grammatik ist **mehrdeutig**, wenn
  - es für mind. 1 ein Wort 2 verschiedene linkskanonische Ableitungen gibt.
- A $\rightarrow \sigma$ heißt rekursiv, wenn A in σ vorkommt.
- A $\rightarrow \sigma$ heißt linksrekursiv, wenn σ mit A beginnt.
- Top-Down-Analyse: eine Ableitung der Tokenfolge des Quellprogrammes aus dem Startsymbol zu bilden, die nur Linksableitungen enthält.

Vorgang der Top-Down-Analyse

1. Wurzel wird mit dem Startsymbol markiert.
2. Wiederhole:
   - 2.1 Aktuellen Knoten betrachten:
     - Wenn ein NT-Symbol: in Abhängigkeit des nächsten zu lesenden Zeichens eine der Produktionen wählen und für jedes Grammatiksymbol der rechten Seite einen Knoten als Nachfolger anlegen.
     - Wenn ein T-Symbol und dieses stimmt mit dem zu nächsten zu lesenden Zeichen überein, dann: im Parsebaum als auch in der Eingabe einen Schritt weiter gehen.
   - 2.2 Nächsten zu behandelnden Knoten suchen: Sind die Nachfolger eines Knoten erzeugt, so behandeln wir als nächstes diese Nachfolger von links nach rechts.

Beispiele

- $G = (\Phi, \Sigma, P, S)$
  $\Phi = \{S\}$
  $\Sigma = \{x, y, z, (,), +, *\}$
  $P = \{S \to S + S | S * S | (S) | x | y | z\}$
  $S = S$
  ges.: Ableitung und Syntaxbaum für `x + y * z`
- $G = (\Phi, \Sigma, P, S)$
  $\Phi = \{S\}$
  $\Sigma = \{a, b, c, +, -\}$
  $P = \{S \to a + S | b - S | (S) | c\}$
  $S = S$
  ges.: Ableitung und Syntaxbaum für `a + b - c`

Probleme

Parser sind so zwar einfach zu konstruieren, aber:

1. Wenn es zu einem abzuleitenden NT-Symbol mehrere Ableitungsregeln gibt, muss anhand des aktuellen Eingabetokens entschieden werden, welche Regel anzuwenden ist. Dadurch: Sackgassen $\leadsto$ Backtracking!
   - x + y * z mit vorhergehender Grammatik $\leadsto$ S $\Rightarrow$ S + S $\Rightarrow$ x + S $\Rightarrow$ x + y!!!
2. Weiters können Endlosschleifen bei links-rekursiven Produktionen auftreten
   - S $\Rightarrow$ $\underline{\text{S}}$ + S $\Rightarrow$ $\underline{\text{S}}$ + S $\Rightarrow$ ...

Top-Down-Parser – Impl.

1. recursive-descent: Jedes NT-Symbol entspricht einer Prozedur. Das Anhängen von Knoten an den Parse-Baum geschieht durch einen Prozeduraufruf.
2. tabellengesteuert: funktioniert mit einer Grammatik-spezifischen Tabelle und die Verwaltung des Ableitungsprozesses wird mit Hilfe eines Stacks realisiert.

LL(1) Grammatiken

- ohne Linksrekursion für die eine Sackgassen-freie Top-Down-Analyse möglich ist.
- Wenn Produktion X $\rightarrow \sigma_1|\sigma_2|\ldots$, dann muss alleine durch Betrachten des nächsten Zeichens (look-ahead symbol) klar sein, welche der Alternativen zu wählen ist.
- Ein LL(1) Parser für eine LL(1) Grammatik
  - liest und untersucht das Eingabewort von links nach rechts.
  - liefert immer eine linkskanonische Ableitung, wenn eine Ableitung möglich ist.
  - liest genau 1 Zeichen voraus.

LL(1) Grammatiken – 2

- Satzform: eine beliebige Folgen von T bzw. NT-Symbolen der Grammatik G
- First-Menge
  - FIRST(σ) =
    - $\{t \in \Sigma | \sigma \Rightarrow^* t...\} \cup \{\varepsilon\}$, falls $\sigma \Rightarrow^* \varepsilon$
    - $\{t \in \Sigma | \sigma \Rightarrow^* t...\}$ anderenfalls
- Follow-Menge
  - $A \in \Phi$, FOLLOW(A) $=$
    - FOLLOW(A) $= \{t \in \Sigma | S \Rightarrow^* \alpha A t \beta, \alpha, \beta$beliebig$\}$
- Beispiel
  - $G = (\Phi, \Sigma, P, S)$
    $\Phi = \{S\}$
    $\Sigma = \{a, +\}$
    $P = \{S \rightarrow S + a | \varepsilon\}$
    $S = S$
    ges.: FIRST($S + a$), FIRST(ε), FOLLOW(S)

LL(1) Grammatiken – 3

- Eine KF Grammatik ist eine LL(1) Grammatik, wenn die beiden folgenden Bedingungen gelten:
  1. Falls zu einem NT-Symbol N zwei alternative Produktionen $N \rightarrow \sigma_1$ und $N \rightarrow \sigma_2$ gibt, muss gelten:
     - $\text{FIRST}(\sigma_1) \cap \text{FIRST}(\sigma_2) = \{\}$
  2. Falls aus einem NT-Symbol der Leerstring ε abgeleitet werden kann, muss gelten:
     - $\text{FIRST}(N) \cap \text{FOLLOW}(N) = \{\}$
- Beispiel
  - $G = (\Phi, \Sigma, P, S)$
    $\Phi = \{A, T, F\}$
    $\Sigma = \{x, y, +, -, *, /, (,)\}$
    $P = \{A \rightarrow A + T | A - T | T, T \rightarrow T * F | T/F | F, F \rightarrow x | y | (A)\}$
    $S = A$

Recursive-Descent Parser

G ist LL(1)

$\phi = \{S, S_1, S_2, ...S_n\}, S = S$

Parser hat folgende Struktur:

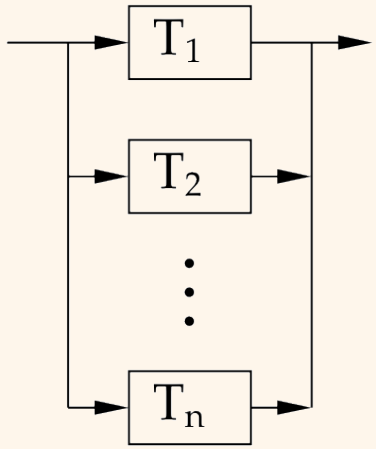
```
symbol = 0  # Token als ganze Zahl
def error():
    pass

def s1():
    pass
...
def sn():
    pass
def s():
    pass

def main():
    getsym()  # erstes Zeichen von Scanner
    s()       # Los geht's
```

Recursive-Descent Parser – 2

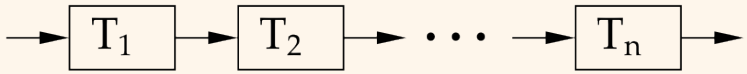
- Auswahl




```
if symbol in FIRST(T1): # T1 ... Teilgraph 1 der Syntax
    P(T1)  # Code fuer Teilgraph 1
elif symbol in FIRST(T2):
    P(T2)
...
elif symbol in FIRST(Tn):
    P(Tn)
else:
    error()
```

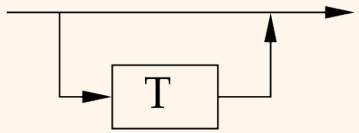
Recursive-Descent Parser – 3

- Sequenz




```
P(T1)
P(T2)
...
P(Tn)
```

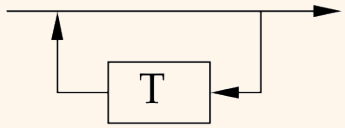
- Optionale Produktion




```
if symbol in FIRST(T):
    P(T)
```

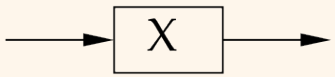
Recursive-Descent Parser – 4

- Iteration



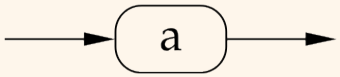

```
while symbol in FIRST(T): P(T)
```

- Non-Terminalsymbol




```
X()
```

- Terminalsymbol




```
if symbol == a:
    getsym()
else:
    error();
```

Bottom-Up Analyse

- Man geht von einem gegebenen Satz aus und reduziert diesen schrittweise bis Startsymbol erreicht ist.
- LR(k) Grammatik: Eingabestring von (l)inks nach rechts gelesen und Reduktionen anwenden, sodass rechtskanonische Ableitungen entstehen. k ist die Anzahl der Look-ahead Symbole.
- Bottum-Up Parser sind *immer* tabellengesteuert
- Aufwand zum Erstellen der Tabellen groß!
  - daher eigene Programme (klassisch: lex, yacc) $\rightarrow$ ply

Tools

- lex und yacc (Yet Another Compiler Compiler)
  - bzw. flex und bison (für viele Programmiersprachen)
- ply
  - einfache Umsetzung von lex und yacc in Python
  - http://www.dabeaz.com/ply/
- ANTLR
  - LL(k)
  - in Java
  - für: Java, C#, Python, C++, JavaScript, Go, Swift
  - http://www.antlr.org/
- PEGTL (Parser Expression Grammar Template Library)
  - https://github.com/taocpp/PEGTL

Assembler

- Aufgabe: Assemblerbefehle in Maschinensprachebefehle
- Beispiel (von wikipedia)
  - Übersetze: `MOV AL, 61h`
    - verschiebe den Wert 0x61 (8 Bitwert) in das Register AL
  - In: `10110000 01100001`
    - `10110000` … lade 8-Bitwert in Register AL
      `10011` … lade 8-Bitwert
      `000` … Register AL
    - `01100001` … 0x61

Linker

- Linker (dt. Binder) oder Link Editor
  - verbindet ein oder mehrere Objektdateien und Bibliotheken zu einem ausführbaren Programm (executable) oder einer Bibliothek
- Der Vorgang wird als Linken (engl. linking) bezeichnet
  - statisches Linken
  - dynamisches Linken
- Loader (oder linker and loader)

Artefakte

- Artefakt: ein Produkt im SW-Entwicklungsprozess
- Objektdateien
  - Dateinamenerweiterung: `.o` (Unix) bzw. `.obj` (Windows)
- Arten von Bibliotheken
  - statische Bibliothek (static library)
    - Linken zur Übersetzungszeit
    - Dateinamenerweiterung: `.a` (Unix) bzw. `.lib` (Windows)
    - unter Unix: Dateiname beginnt mit `lib`
  - dynamische Bibliothek (dynamic library, shared library, shared object)
    - Linken zur Laufzeit
    - Dateinamenerweiterung: `.so` (Unix) bzw. `.dll` (Windows)

Statisches Linken

- Vorteile
  - leichtere Portabilität
    - keine dynamische Bibliothek erforderlich
  - keine Installation notwendig
- Nachteile
  - wenn mehrere Programme dieselbe Library verwenden:
    - größere Executables: $\rightarrow$ mehr Hauptspeicher- und Festplattenverbrauch
  - erneutes Linken und Ausliefern bei jeder kleiner Änderung

Dynamisches Linken

- Vorteile
  - wenn mehrere Programme dieselbe Library verwenden:
    - kleinere Executables: $\rightarrow$ weniger Hauptspeicher- und Festplattenverbrauch
  - Plugins können leichter realisiert werden
- Nachteile
  - fehlende Bibliotheken bei der Ausführung
  - falsche Versionen der Bibliotheken bei der Ausführung
  - etwas größere Zeiten beim Starten des Programmes

Loader

- Programm starten
  1. Rechte überprüfen
  2. Speicher reservieren
  3. Programm von der Festplatte laden
     - wenn nicht schon einmal geladen (gestartet)
  4. benötigte dynamische Bibliothek laden
     - wenn nicht schon geladen
  5. Linker anstoßen
  6. u.U. Kommandozeilenparameter auf Stack
  7. Prozess starten