Modellierung und Programmierung

Dr. Martin Riplinger

23.1.2012

Die kommerzielle Software MATLAB von MathWorks

MathWorks TAH Campuslizenz

Die Software darf von allen Studierenden der Universität des Saarlandes genutzt werden. Das umfasst einerseits beliebig viele Installationen auf dem Campus und andererseits auch die Nutzung auf privaten Computern.
Notwendig: Registrierung bei der Firma ASKnet AG. Weitere Informationen:
`https://unisb.asknet.de/cgi-bin/program/S1552`

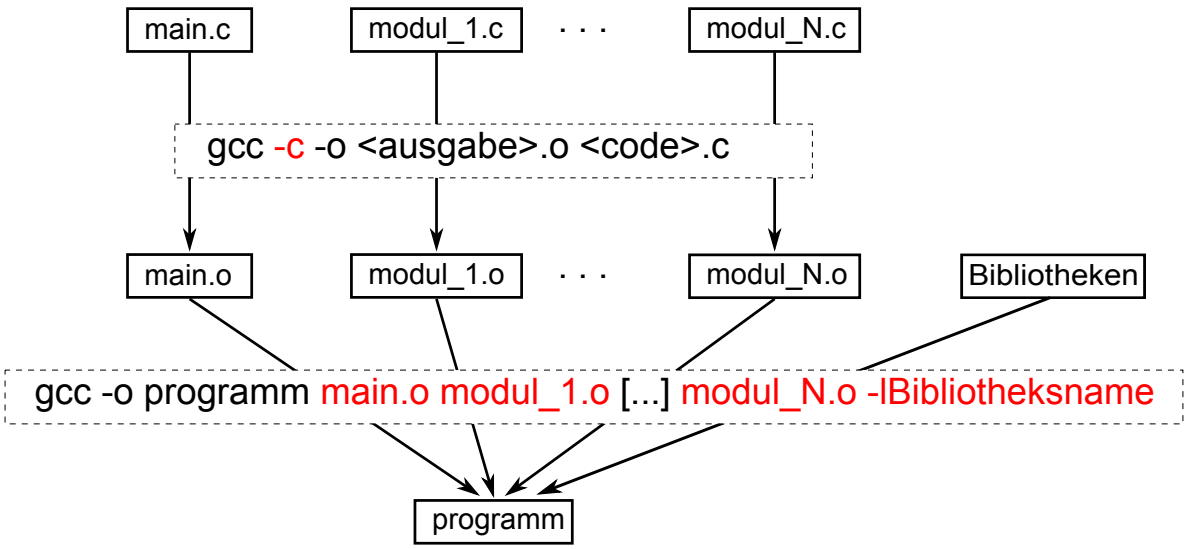
Weitere Informationen:
`http://www.hiz-saarland.de/informationen/arbeitsplatz/sw-lizenzen/`

Herstellerseite:
`http://www.mathworks.com`
`http://www.mathworks.de`

Programmierprojekte mit mehreren Dateien

- Die Präprozessordirektive `#include "eigener_header.h"` erlaubt es, Funktions**deklarationen**, `typedef`s, Definitionen von Makros usw. in einen externen Header auszulagern.
- Nach wie vor müssen jedoch alle Funktions**definitionen** im Hauptprogrammcode enthalten sein.
- Diese Tatsache führt zu langen und unübersichtlichen Codedateien und verhindert eine einfache Wiederverwendung der programmierten Funktionen (einzige Möglichkeit: copy&paste in andere Projekte).
- Wünschenswert wäre eine Möglichkeit, Funktionsdefinitionen (gruppiert nach Aufgabengebiet) in externe Quellcode-Dateien auszulagern.
- Mit Hilfe von **Objektdateien**, einem Zwischenprodukt beim Compilieren von Code, lässt sich diese Möglichkeit realisieren.

Programmierprojekte mit mehreren Dateien: Prinzip



Programmierprojekte mit mehreren Dateien: Prinzip

- Der Quellcode ist verteilt auf mehrere **Module** - getrennt nach Funktionalität - und das **Hauptprogramm**.
- Jede Codedatei wird separat **zu Objektcode compiliert** mit dem Aufruf `gcc -c -o Datei.o Datei.c`
- Mit `gcc -o programm Liste_von_Objektdateien` werden die Objektdateien zu einem ausführbaren Programm „zusammengebunden“ (gelinkt).
- **Erinnerung**:
  - Der **Compiler** braucht alle nötigen **Deklarationen** und gibt als Resultat Objektcode aus.
  - Erst der **Linker** muss die **Definitionen** zur Verfügung haben, sonst liefert er einen `undefined reference`-Fehler. Das Ergebnis des Link-Vorgangs ist schließlich das ausführbare Programm.

Programmierprojekte mit mehreren Dateien: Beispiel – `matrix.h`

```
1  #ifndef __MATRIX_H__
2  #define __MATRIX_H__
3
4  // Legt Speicher fuer eine neue Matrix an und liefert einen Zeiger auf die
5  // erste Zeile; bei Misserfolg NULL
6  double **neueMatrix(int n_zeilen, int n_spalten);
7
8  // Gibt die Matrix am Bildschirm aus
9  void zeigeMatrix(double **matrix, int n_zeilen, int n_spalten);
10
11 // Gibt den Speicherplatz der Matrix frei
12 void loescheMatrix(double **matrix, int n_zeilen);
13
14 // Reserviert Speicher fuer eine Matrix und liest die Elemente aus der
15 // TEXTdatei, die unter dem Pfad zu finden ist; bei Misserfolg NULL
16 double **neueMatrixAusDatei(char *pfad, int *n_zeilen, int *n_spalten);
17
18 // Schreibt die Matrix im Textmodus in die Datei im angegebenen Pfad
19 void schreibeMatrixInDatei(char *pfad, double **matrix,
20                            int n_zeilen, int n_spalten);
21
22 // Reserviert Speicher fuer eine Matrix und liest die Elemente aus der
23 // BINAERdatei, die unter dem Pfad zu finden ist; bei Misserfolg NULL
24 double **neueMatrixAusBinDatei(char *pfad, int n_zeilen, int n_spalten);
25
26 // Schreibt die Matrix im Binaermodus in die Datei im angegebenen Pfad
27 void schreibeMatrixInBinDatei(char *pfad, double **matrix,
28                               int n_zeilen, int n_spalten);
29 #endif
```

Programmierprojekte mit mehreren Dateien: Beispiel – matrix_funktionen.c

```c
#include <stdio.h>
#include <stdlib.h>

double **neueMatrix(int n_zeilen, int n_spalten)
{
  int i, j;
  double **matrix;

  matrix = (double **)malloc(n_zeilen * sizeof(double *));
  if (matrix == NULL)
    return NULL;

  for (i = 0; i < n_zeilen; i++)
  {
    matrix[i] = (double *)malloc(n_spalten * sizeof(double));
    if (matrix[i] == NULL)
    {
      for (j = 0; j < i; j++)
        free(matrix[j]);
      free(matrix);
      return NULL;
    }
  }
  return matrix;
}

// Und alle weiteren Funktionsdefinitionen
```

Programmierprojekte mit mehreren Dateien: Beispiel – matrix_test_main.c

```c
#include "matrix.h"

int main(void)
{
  int n, m;
  double **G, **H;

  G = neueMatrixAusDatei("Matrix_kommentiert.dat", &n, &m);
  H = neueMatrixAusBinDatei("Matrix.bin", n, m);

  zeigeMatrix(G, n, m);
  zeigeMatrix(H, n, m);

  loescheMatrix(G, n);
  loescheMatrix(H, n);

  return 0;
}
```

Kommandos:

- `gcc -c -o matrix_test_main.o matrix_test_main.c` (Compiler)
- `gcc -c -o matrix_funktionen.o matrix_funktionen.c` (Compiler)
- `gcc -o matrix_test matrix_test_main.o matrix_funktionen.o` (Linker)

Programmierprojekte mit mehreren Dateien

Anmerkungen:

- Das Modul `matrix_funktionen.c` benötigt die Systemheader `stdio.h` und `stdlib.h`, damit die dort deklarierten Funktionen (z. B. `printf` oder `malloc`) verwendet werden können. Sind sie nicht eingebunden, liefert der Compiler (`gcc -c`) einen Fehler.
- Der Header `matrix.h` braucht hingegen **nicht** inkludiert zu werden.
- Im Hauptprogramm genügt es, `matrix.h` einzufügen. Gäbe es Statements, die beispielsweise `printf` enthalten, so müsste zusätzlich `stdio.h` inkludiert werden.

  Merkregel: Jede Codedatei benötigt genau diejenigen Header, die nötig sind, damit **alle Variablen, Konstanten, Funktionen, ... innerhalb dieser Datei** korrekt deklariert sind.
- Wird der Code eines Moduls geändert, so müssen **alle Objekte und Programme** neu compiliert und gelinkt werden, die **von diesem Modul abhängen**.
  In diesem Beispiel müssen nach einer Änderung in `matrix_funktionen.c` alle Schritte von neuem durchgeführt werden!
- Bei größeren Projekten mit vielen Dateien wird es zunehmend schwierig nachzuvollziehen, welche Aktionen nach welcher Änderung vollzogen werden müssen. Hierfür gibt es die elegante Lösung der **Makefiles**.

Mehrdateiprojekte mit `make`

- `make` ist das mit Abstand wichtigste Entwicklungs-Tool bei Softwareprojekten.
- Wir betrachten hier die am weitesten verbreitete Implementierung **GNU Make** (`www.gnu.org/software/make/`)
- **Aufgabe**: Neuübersetzung **genau derjenigen** Programmteile, die von einer Änderung am Code betroffen sind.

  Hintergrund: Die „Brechstangen"-Methode, bei jeder Änderung alles neu zu übersetzen, ist aus zwei Gründen nicht praktikabel:

  1. Umfangreiche Projekte benötigen Minuten bis Stunden zur Komplettübersetzung
  2. Für jede einzelne Datei gcc aufzurufen (mit individuellen Optionen) ist extrem mühsam.

- **Funktionsweise**: In einer **Steuerungsdatei** mit dem Namen `Makefile` gibt man alle Abhängigkeiten zwischen den Dateien in Form von **Erstellungsregeln** (engl. *make rules*) an.

Mehrdateiprojekte mit `make`: Erstellungsregeln

Aufbau:

```
Ziel:  Abhaengigkeiten \
       weitere Abhaengigkeiten
------>Befehl1
------>...
------>BefehlN
```

- `Ziel` bezeichnet „das, was getan werden soll“, also entweder einen Dateinamen (z. B. `matrix_funktionen.o`) oder eine abstrakte Aktion (z. B. `clean`).
- Als Abhängigkeiten werden sämtliche **Dateien** angegeben, von deren Änderung das `Ziel` abhängt bzw. abhängen soll. Lange Zeilen können mit einem Backslash am Ende umgebrochen werden.
- Die Befehle definieren, was `make` unternehmen soll, um das Ziel zu erstellen.

  Wichtig: Befehlszeilen **müssen immer** mit einem **Tabulator** (TAB-Taste) beginnen, sonst meldet `make` einen Fehler.

Mehrdateiprojekte mit `make`: Beispiel Matrix-Projekt

Bisheriger Inhalt: `matrix.h`, `matrix_funktionen.c` und `matrix_test_main.c`

Jede Quellcodedatei `.c` wird zu einer Objektdatei `.o` kompiliert. Diese Objekte werden schließlich zu einem ausführbaren Programm zusammengebunden.

Abhängigkeiten:

- `matrix_funktionen.o`: Entsteht aus `matrix_funktionen.c`
- `matrix_test_main.o`: Entsteht aus `matrix_test_main.c` und inkludiert `matrix.h`
- `matrix_test`: Entsteht aus `matrix_test_main.o` und `matrix_funktionen.o`

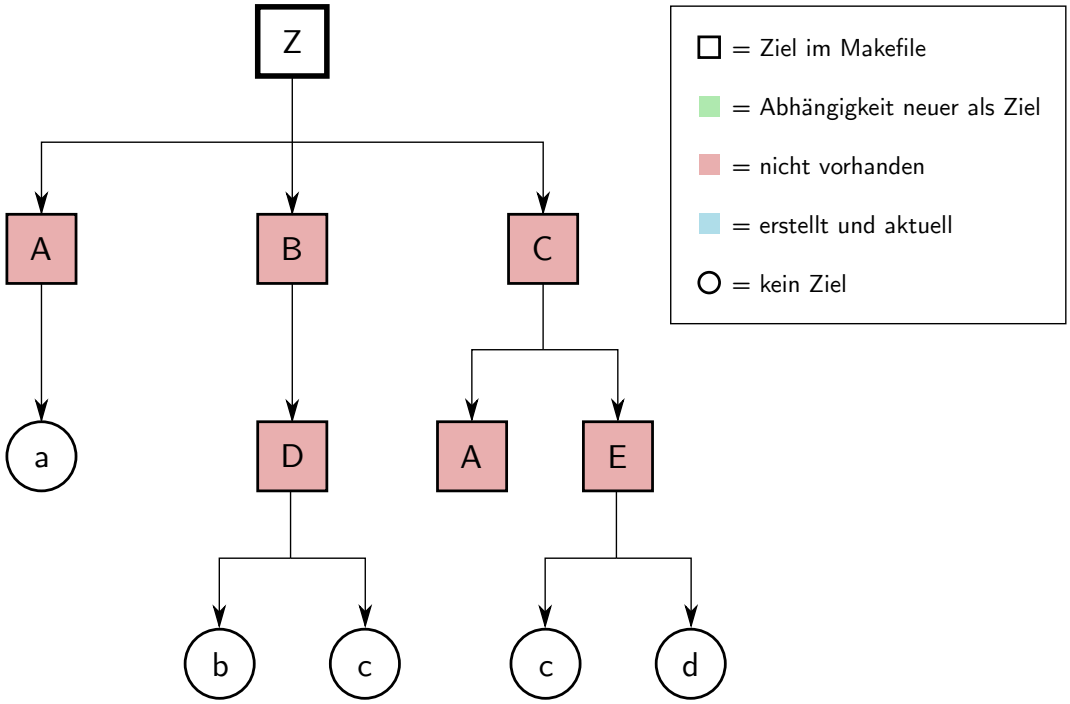
Daraus ergeben sich folgende Erstellungsregeln:

```
1  matrix_test: matrix_test_main.o matrix_funktionen.o
2  ------>gcc -o matrix_test matrix_test_main.o matrix_funktionen.o
3
4  matrix_test_main.o: matrix_test_main.c matrix.h
5  ------>gcc -c -o matrix_test_main.o matrix_test_main.c
6
7  matrix_funktionen.o: matrix_funktionen.c
8  ------>gcc -c -o matrix_funktionen.o matrix_funktionen.c
```

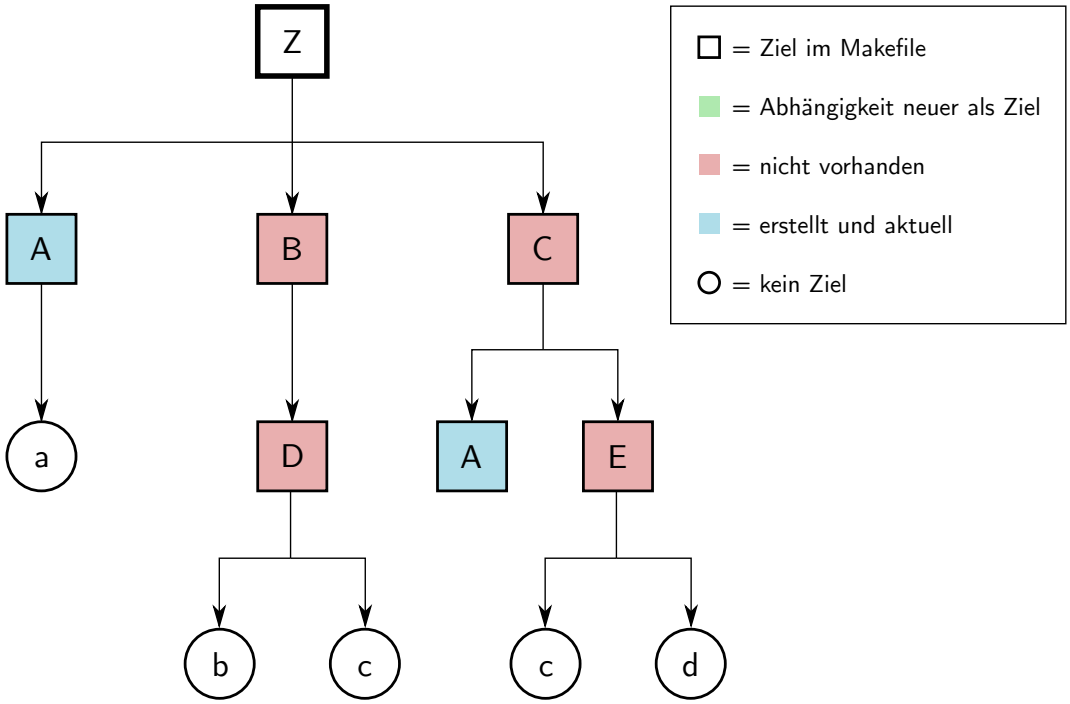
Von nun an genügt es, nach jeder Änderung auf der Kommandozeile `make` aufzurufen:

```
$ make
gcc -c -o matrix_test_main.o matrix_test_main.c
gcc -c -o matrix_funktionen.o matrix_funktionen.c
gcc -o matrix_test matrix_test_main.o matrix_funktionen.o
```

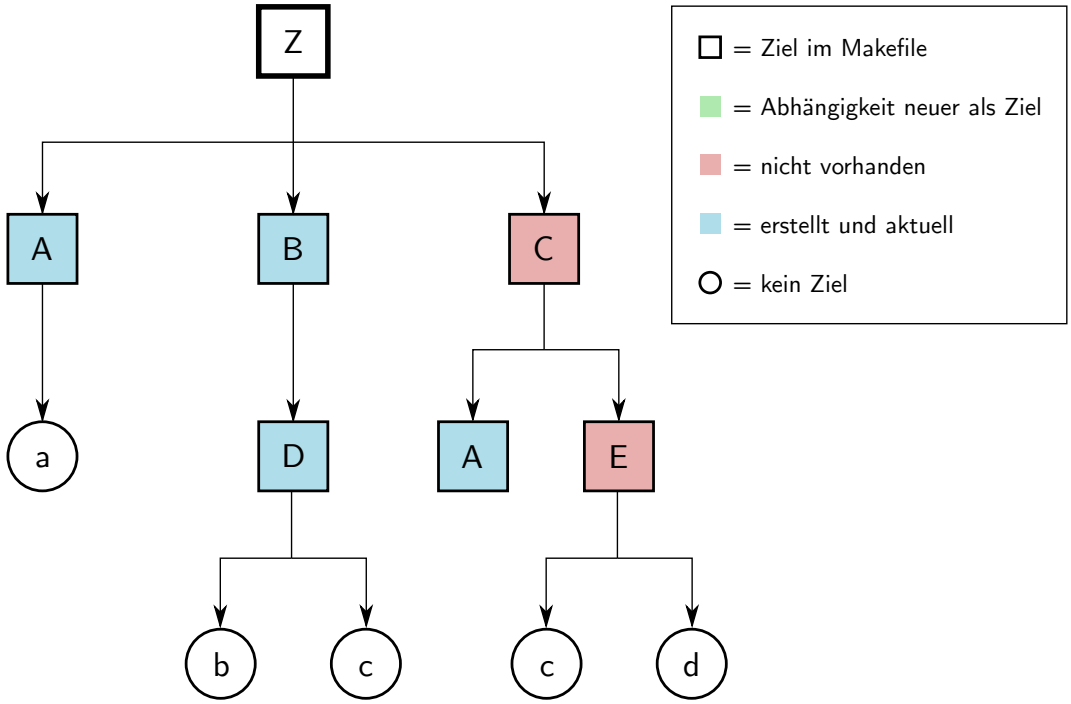
Auflösung der Abhängigkeiten: Erster Aufruf von `make` mit Ziel „Z“



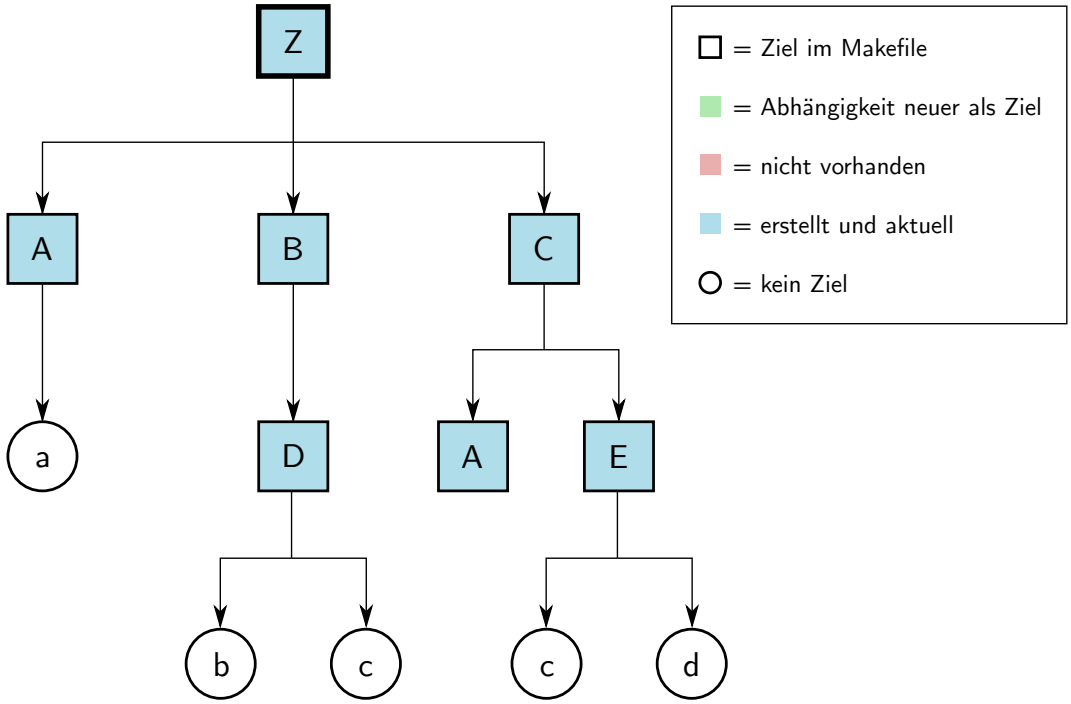
Auflösung der Abhängigkeiten: Erster Aufruf von `make` mit Ziel „Z“



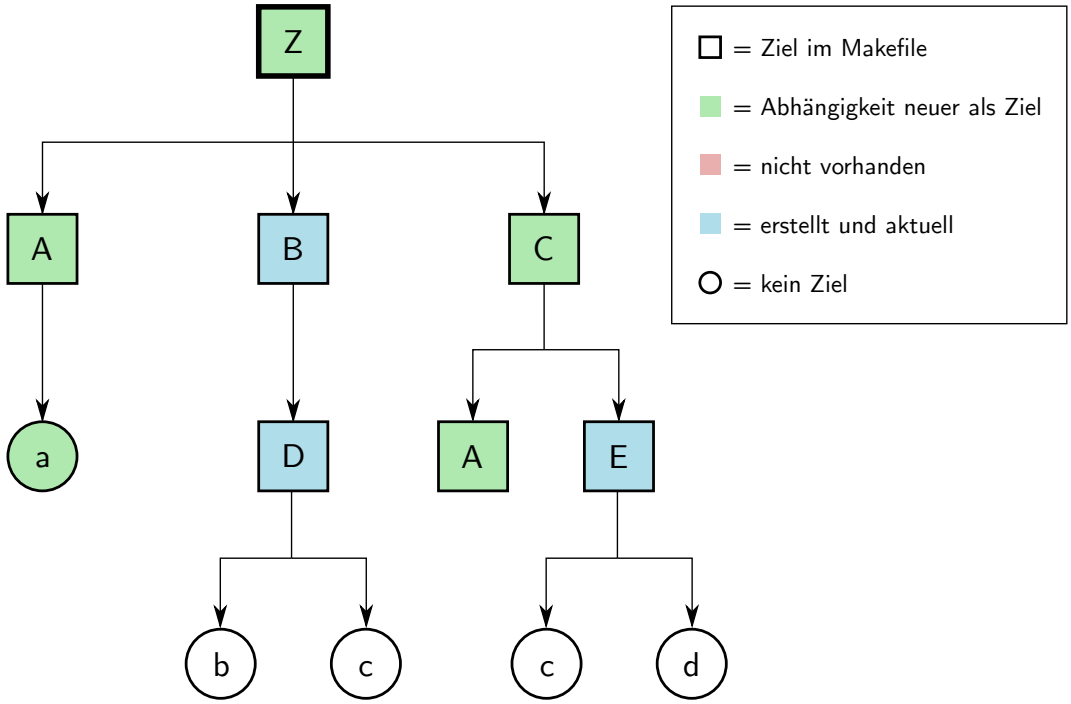
Auflösung der Abhängigkeiten: Erster Aufruf von `make` mit Ziel „Z“



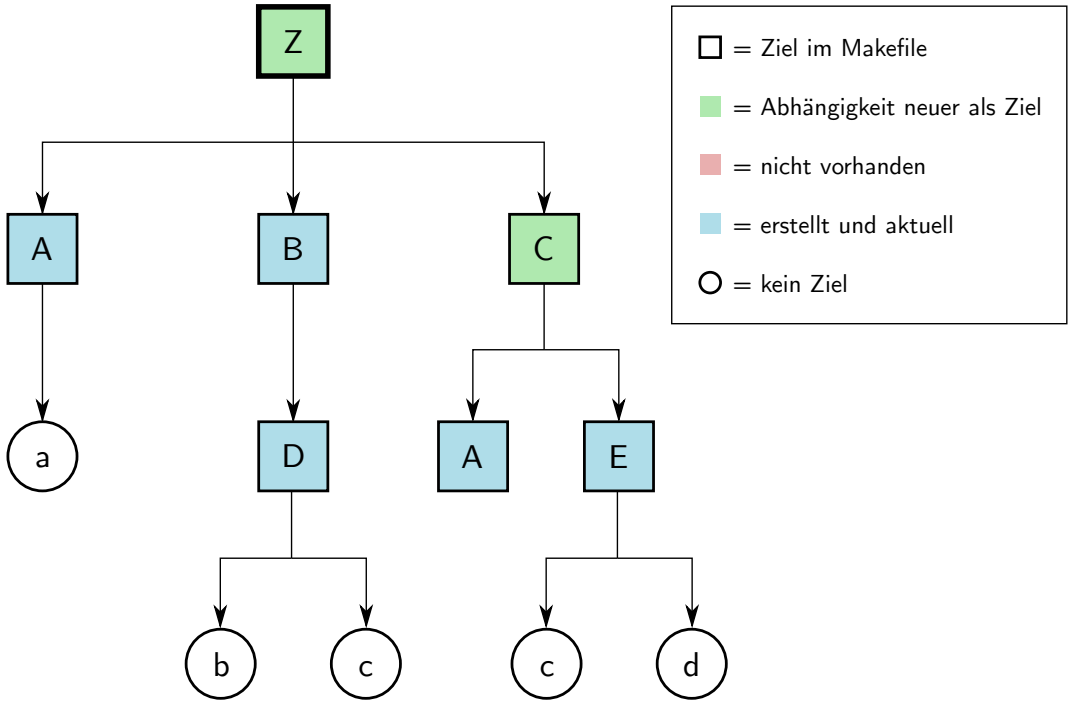
Auflösung der Abhängigkeiten: Erster Aufruf von `make` mit Ziel „Z“



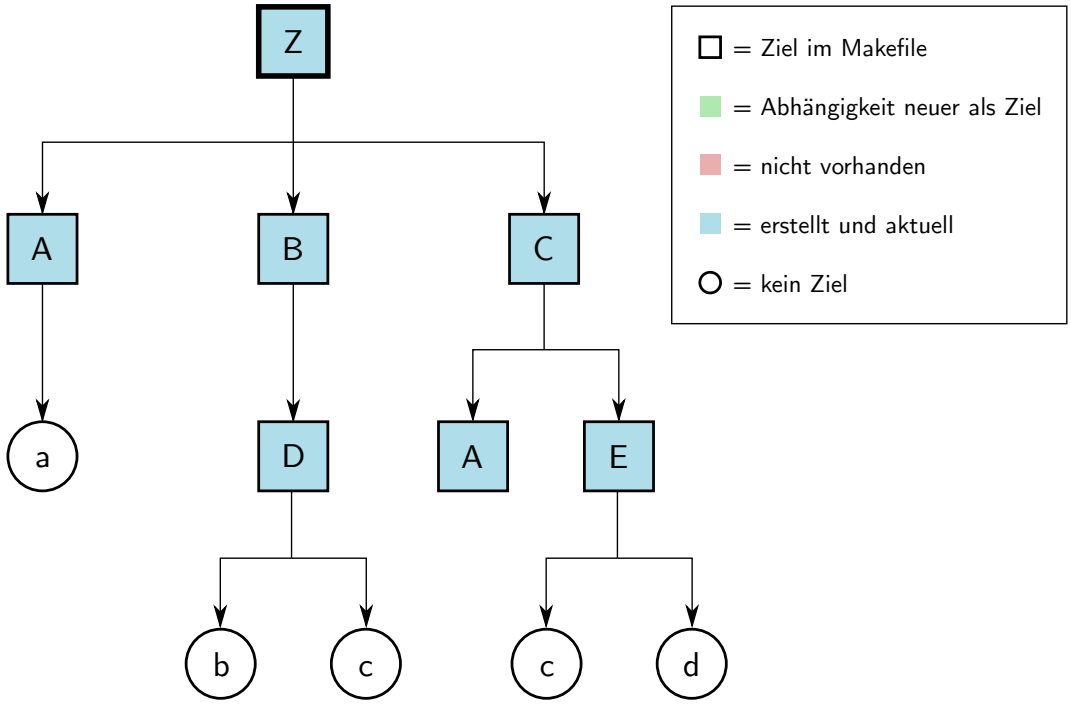
Auflösung der Abhängigkeiten: Nach Aktualisierung der Datei „a“



Auflösung der Abhängigkeiten: Nach Aktualisierung der Datei „a“



Auflösung der Abhängigkeiten: Nach Aktualisierung der Datei „a“



Mehrdateiprojekte mit `make`

Bemerkungen:

- Das Kommando `make Ziel` erstellt `Ziel` anhand der Abhängigkeiten in der Datei `Makefile`. Der Befehl `make` ohne Argument erstellt das **erste Ziel** im Makefile.
- Eine Erstellungsregel muss nicht unbedingt Abhängigkeiten besitzen.

  Beispiel: Regel zum Aufräumen

  ```
  clean:
  ————→rm *.o matrix_test -f
  ```

  Mit `make clean` können nun alle Sicherungs- und Objektdateien sowie das Programm gelöscht werden.
- Auch Befehle müssen nicht zwingend in einer Regel enthalten sein.

  Beispiel: Alles erstellen

  ```
  all: matrix_test matrix_test_main.o matrix_funktionen.o
  ```

  Durch die Auflistung sämtlicher Objekt- und Programmdateien als Abhängigkeiten von `all` lässt sich mit dem Aufruf `make all` das gesamte Projekt aktualisieren.

Mehrdateiprojekte mit make: implizite Regeln und Variablen

GNU Make verfügt über eine große Menge von impliziten Regeln, die im Makefile nicht neu definiert werden müssen.

Wichtigstes Beispiel:

```
%.o: %.c
⟶$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

In Worten:

- Jedes Objekt `Datei.o` hängt vom entsprechenden Code `Datei.c` ab.
- Zur Erstellung eines solchen Objekts ist folgender Befehl aufzurufen:

```
Compiler -c Compileroptionen Präprozessoroptionen Datei.c -o Datei.o
```

Zusätzliche Abhängigkeiten können als **Regel ohne Befehl** angegeben werden.

Beispiel: `matrix_test_main.o: matrix.h`

Mit Hilfe der Variablen `CC`, `CFLAGS` und `CPPFLAGS` lassen sich Compilername und Optionen anpassen.

Beispiel:

```
CC = gcc
CFLAGS = -Wall
CPPFLAGS = -DDEBUG_AN
```

resultiert im generellen Compilerbefehl

```
gcc -c -Wall -DDEBUG_AN Datei.c -o Datei.o
```

Mehrdateiprojekte mit make: Vereinfachtes Matrix-Makefile

Ursprüngliches explizites Makefile:

```
1  matrix_test: matrix_test_main.o matrix_funktionen.o
2  ⟶gcc -Wall -o matrix_test matrix_test_main.o matrix_funktionen.o
3
4  matrix_test_main.o: matrix_test_main.c matrix.h
5  ⟶gcc -c -Wall -o matrix_test_main.o matrix_test_main.c
6
7  matrix_funktionen.o: matrix_funktionen.c
8  ⟶gcc -c -Wall -o matrix_funktionen.o matrix_funktionen.c
```

Angepasstes Makefile:

```
1  CC = gcc
2  CFLAGS = -Wall
3
4  matrix_test: matrix_test_main.o matrix_funktionen.o
5  ⟶$(CC) $(CFLAGS) $^ -o $@
6
7  matrix_test_main.o: matrix.h
```

Bemerkung:
Wichtige vordefinierte Variablen sind beispielsweise:

$@	Name des Ziels
$^	Liste aller Abhängigkeiten ohne Wiederholungen
$+	Liste aller Abhängigkeiten
$<	erste Abhängigkeit

Resultat des angepassten Makefiles:

```
$ make
gcc -Wall -c -o matrix_test_main.o matrix_test_main.c
gcc -Wall -c -o matrix_funktionen.o matrix_funktionen.c
gcc -Wall matrix_test_main.o matrix_funktionen.o -o matrix_test
```