Vorlesung Compilerbau
WS 2025/2026

Prof.Dr.U.Göhner

HS Kempten

Überblick

- Allgemeines, Motivation
- Einführung
  - Compiler/Interpreter
  - Bootstrapping
- Lexikalische Analyse
  - Adhoc-Scanner
  - Arbeitsweise Scanner-Generator
  - Lex/Flex

Überblick

- Syntaktische Analyse
  - Top-Down, Bottom-Up
  - Rekursiver Abstieg, LL-Parsing
  - Tabellengesteuerter Top-Down Parser
  - First- und Follow-Mengen
  - Kellerautomat
  - LR-Parsing
  - Parsergenerator yacc
  - Fehlerbehandlung

Überblick

- Semantische Analyse
  - Attributierter Syntaxbaum
  - Yacc
- Codeerzeugung
  - Codegenerierung
  - Codeoptimierung
- Natural Language Processing

Allgemeines

- Vorlesung für das 3.Semester, d.h. alle Voraussetzungen zum Eintritt ins 3. Semester müssen erfüllt sein.
- Voraussetzung: VL Theoretische Informatik (!), Algorithmen und Datenstrukturen

Organisatorisches

- 2 SWS Vorlesung
- 2 SWS Übung:
  - Übungsbeginn: Mittwoch, 15.10. / Freitag 17.10.25
  - 4 Gruppen
- PVL:
  - Aktive Mitarbeit an den Übungen
  - Votierkonzept wie bei THI und AuD:
    - 70% der Aufgaben votieren
    - 3* Vorführen von Aufgaben
    - Keine Anwesenheitskontrolle

Motivation

- Warum Compilerbau?
- Compiler-Hersteller:
  - IBM, Microsoft, Intel, PGI Group, Apple, Google,…..
- Neuere Sprachen, z.B.:
  - Google Logica (2021)
  - Microsoft Bosque (2019)
  - Microsoft Q# (2017) für Quantencomputer
  - Apple Swift (2014)
  - Julia (2012) – numerische Anwendungen
  - Google GO (2009)
  - Clojure (2007)
  - Scala (2003)
  - Groovy(2003)

Motivation

Spracherweiterungen:

- OpenMP
- MPI3
- AVX2, AVX512 (Intel)
- CUDA (NVIDIA)
- OpenCL

Nutzung moderner Prozessorarchitekturen:

- Parallelisierung
- GPU-Programmierung
- Quantencomputer

Natural Language Processing

Einführung

- Was ist Compilerbau?
  - Altes Teilgebiet der Informatik (ca. 35 Jahre)
  - Sehr gut erforscht
  - gute Überführung theoretischer Erkenntnisse (formale Sprachen, Automatentheorie) in die Praxis

Einführung

- **Wozu werden Compiler benötigt?**
  - Übersetzung von höheren Programmiersprachen in Assemblersprache
  - Bearbeitung von Text-Markup-Sprachen, z.B. html
  - Druckausgabe-Sprachen (PDF, Postscript)
  - Testwerkzeuge für automatisches Testen (automatisches Instrumentieren von Programmen)

Einführung

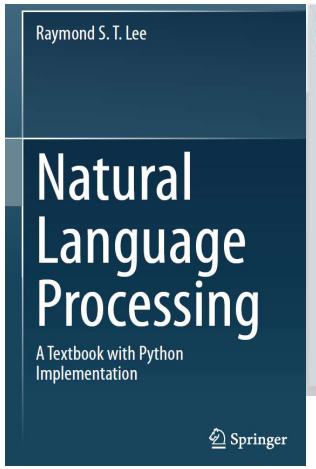
- Warum Compilerbau in der Vorlesung?
- Die Vorlesung Compilerbau befaßt sich mit:
  - SW-Werkzeugen rund um den Compiler
  - dem internen Aufbau von Compilern,
  - den wichtigsten Algorithmen, die in Compilern verwendet werden.
- Ziel der Vorlesung:
  - Erstellung und Funktionsweise eines Compilers zu erarbeiten
  - Einsatz von tools und Algorithmen für
    - Text processing
      - Tokenization
      - Sentence Segmentation
    - Linguistische Strukturen
      - Anreicherung von LLMs
      - Kontextfreie Grammatiken
      - Syntaxbaum

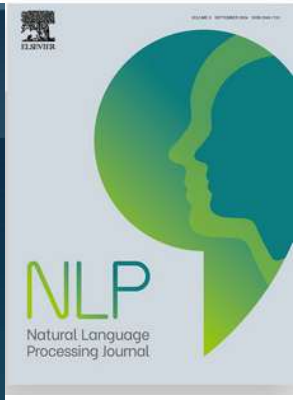
Literatur

- Aho, A.V., Sethi, R., Ullmann, J. D.: Compilerbau. Band 1 und 2, Addison-Wesley 1988 (grüner Drache)
- Aho, A.V., Lam, M.S., Sethi, R., Ullmann, J. D.: Compiler 2. Auflage, 2008, Addison-Wesley
- Levine, J. R., Mason, T., Brown, D.: lex & yacc. O'Reilly & Associates 1995
- Wilhelm, R., Seidl, H., Hack, S.: Übersetzerbau, Springer Vieweg Berlin Heidelberg, 2012.
- Wagenknecht, Ch.; Hielscher, M.: "Formale Sprachenm abstrakte Automaten und Compiler", Springer Vieweg, 2014.
- Böckenhauer, H.-J., Hromkovic, J.: "Formale Sprachen, Endliche Automaten, lexikalische und syntaktische Analyse", Springer Vieweg Wiesbaden, 2013.

Literatur

- Lee, R.S.T. (2024). Natural Language Processing.
  Springer, Singapore
- Natural Language Processing Journal
- Karlsson, Fred, Voutilainen, Atro, Heikkilae, Juha and Anttila, Arto. Constraint Grammar: A Language-Independent System for Parsing Unrestricted Text, Berlin, Boston: De Gruyter Mouton, 1995.





Einführung

- **Was ist ein Compiler?**
  - Programm, das Quell-Programme in Ziel-Programme übersetzt.
  - Die Quell-Programme müssen in der **Quell-Sprache** des Compilers geschrieben sein
  - Der Compiler erzeugt die Ziel -Programme in seiner **Ziel-Sprach**e.
  - Der Compiler läuft auf einem System das eine Implementierungssprache (IS) ausführt.

Einführung

- **Beispiele für Quell-Sprachen** von Compilern:
  - C, C++, C#, Java, Fortran, Cobol, Basic, Algol60, Pascal, Modula, Ada, Eiffel, Lisp, Prolog, Haskell, ...
  - Neue Programmiersprachen:
    - Google GO (Multicore-Konzept)
    - Scala (funktional+objektorientiert)
- Zielsprache des Compilers ist meistens **Maschinen-** oder eine **Assembler-Sprache**

Einführung

– Was ist ein Interpreter?

- Programm, das Programme übersetzt und ausführt
- erfordert **zwei Eingabe**n:
  1. Ein Programm P (das interpretiert, d.h. ausgeführt, werden soll)
  2. Eine Eingabe E für das Programm P (falls Eingabe erforderlich)
- Interpreter führt das Programm P mit der Eingabe E aus und liefert als Ergebnis die Ausgabe A (des Programms P für die Eingabe E).

Einführung

- **Vorteile von Interpreter gegen Compiler**
  - Sofortige Ausführung des Programms
  - Programmänderungen können während der Ausführung vorgenommen werden, danach kann die Ausführung fortgesetzt werden
  - Nur benötigte Programmteile auswerten

Einführung

Vorteile von Compiler gegenüber Interpreter

- Einmal übersetzen – oft ausführen
- Programmoptimierung lohnenswert
- Ausführung des übersetzten Codes ist im allgemeinen effizienter:
  - Code ist schneller
  - Code benötigt weniger Speicherplatz
  - Verwendung von Programmbibliotheken in übersetzter Form möglich
  - Code-Optimierungen sind möglich:
    - Parallelisierung / Vektorisierung
    - Entfernung von überflüssigem Code

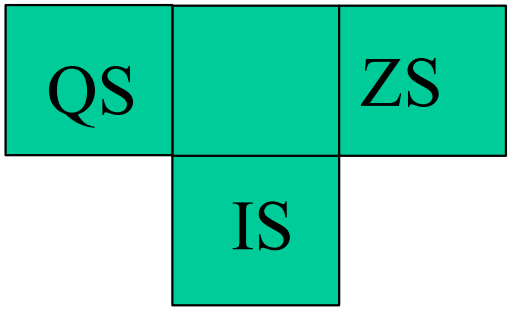
Einführung

- **Mischformen aus Compiler gegenüber Interpreter**
  - Typisch für Java, C#,…
  - Quelltext wird in Zwischencode übersetzt
  - Zwischencode (Bytecode) wird von virtueller Maschine interpretiert
  - Interpretation übersetzten Zwischencodes
  - Just-In-Time-Compiler (JIT-compiler)

Einführung

- **Anwendungen Compiler-Techniken**
  - Assembler
  - Implementierung von Text-Markup-Sprachen (XML, SGML, LaTeX)
  - Model Driven Architecture
  - Analysetools für Reverse-Engineering (automatisierte Generierung von Strukturinformation aus bestehendem Code)
  - Natural Language Processing

T-Diagramm



QS: Quellsprache
ZS: Zielsprache
IS: Implementierungs-sprache

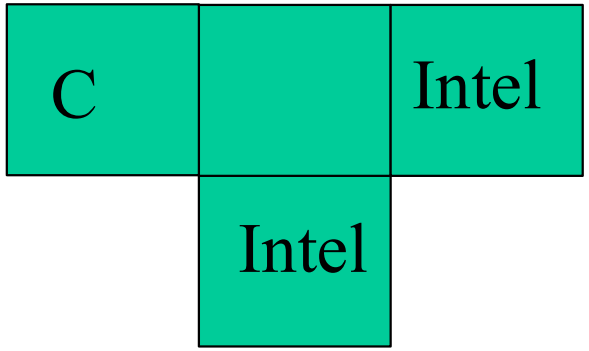
- Das T-Diagramm stellt den Compiler schematisch dar
- Der Compiler **übersetzt** Programme, die in der Quellsprache vorliegen, in die Zielsprache
- Die **Laufzeitumgebung**, bzw. die Sprache in der der Compiler geschrieben ist, ist die Implementierungssprache
- T-Diagram = Tombstone-Diagram

T-Diagramm

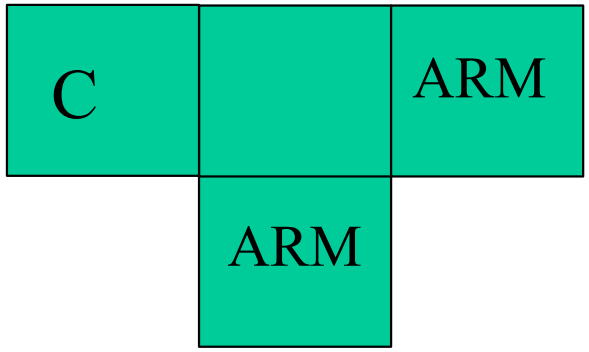
Bootstrapping:
- Technik, um Compiler schrittweise aus anderen vorgegebenen Compilern zu entwickeln.
- T-Diagrammen zeigt die notwendigen Übersetzungsschritte

Beispiel:

gegeben:
C-Compiler auf Intel

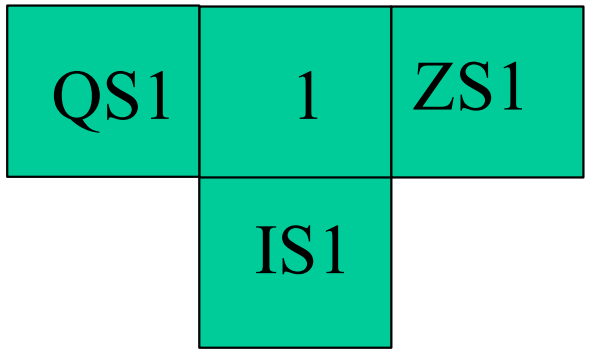


gesucht:
C-Compiler auf ARM

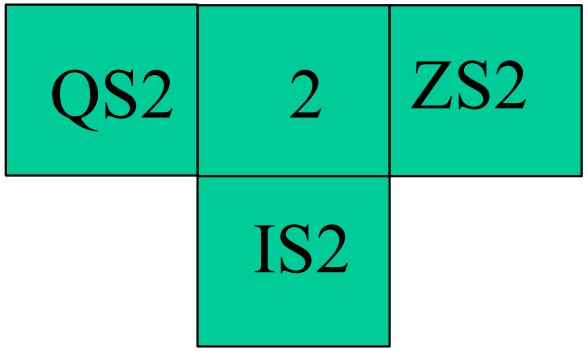


T-Diagramm

Vorgehensweise beim Bootstrapping:
Compiler 1 ist gegeben:
- übersetzt von QS1 nach ZS1
- lauffähig auf System, das IS1 ausführen kann
- Compiler ist geschrieben in Sprache IS1

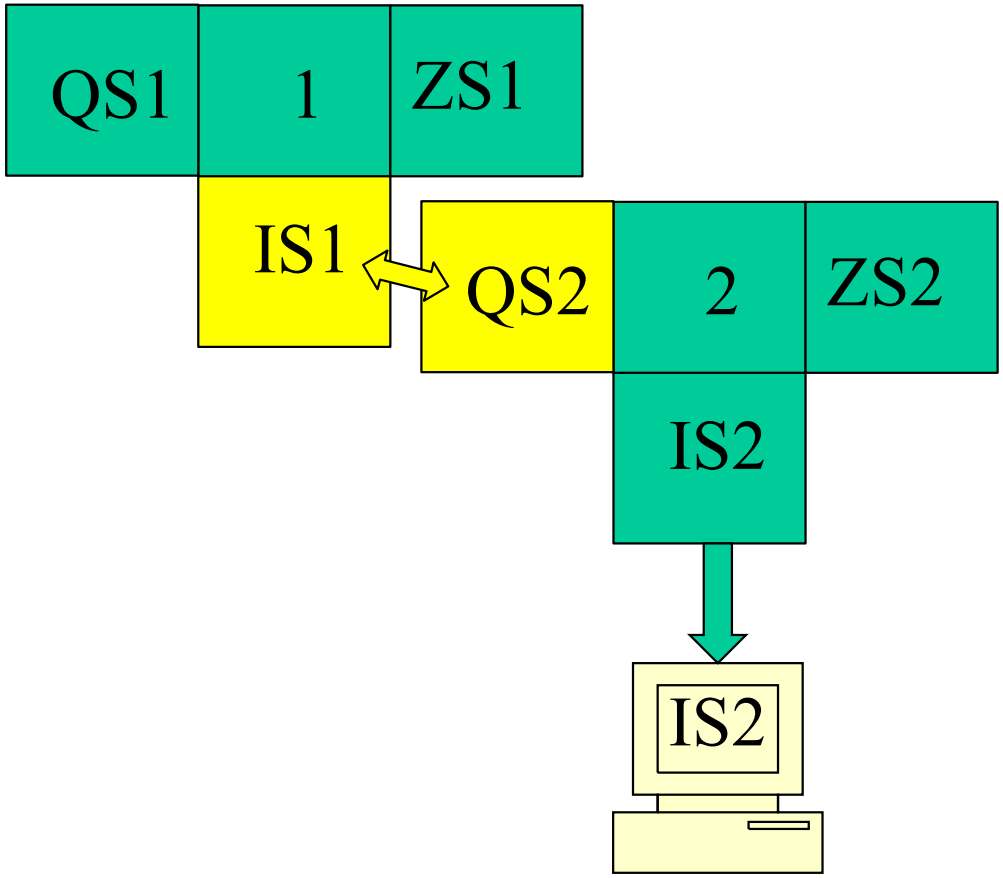


Compiler 2 ist gegeben:
- übersetzt von QS2 nach ZS2
- lauffähig auf System, das IS2 ausführen kann
- Compiler ist geschrieben in Sprache IS2



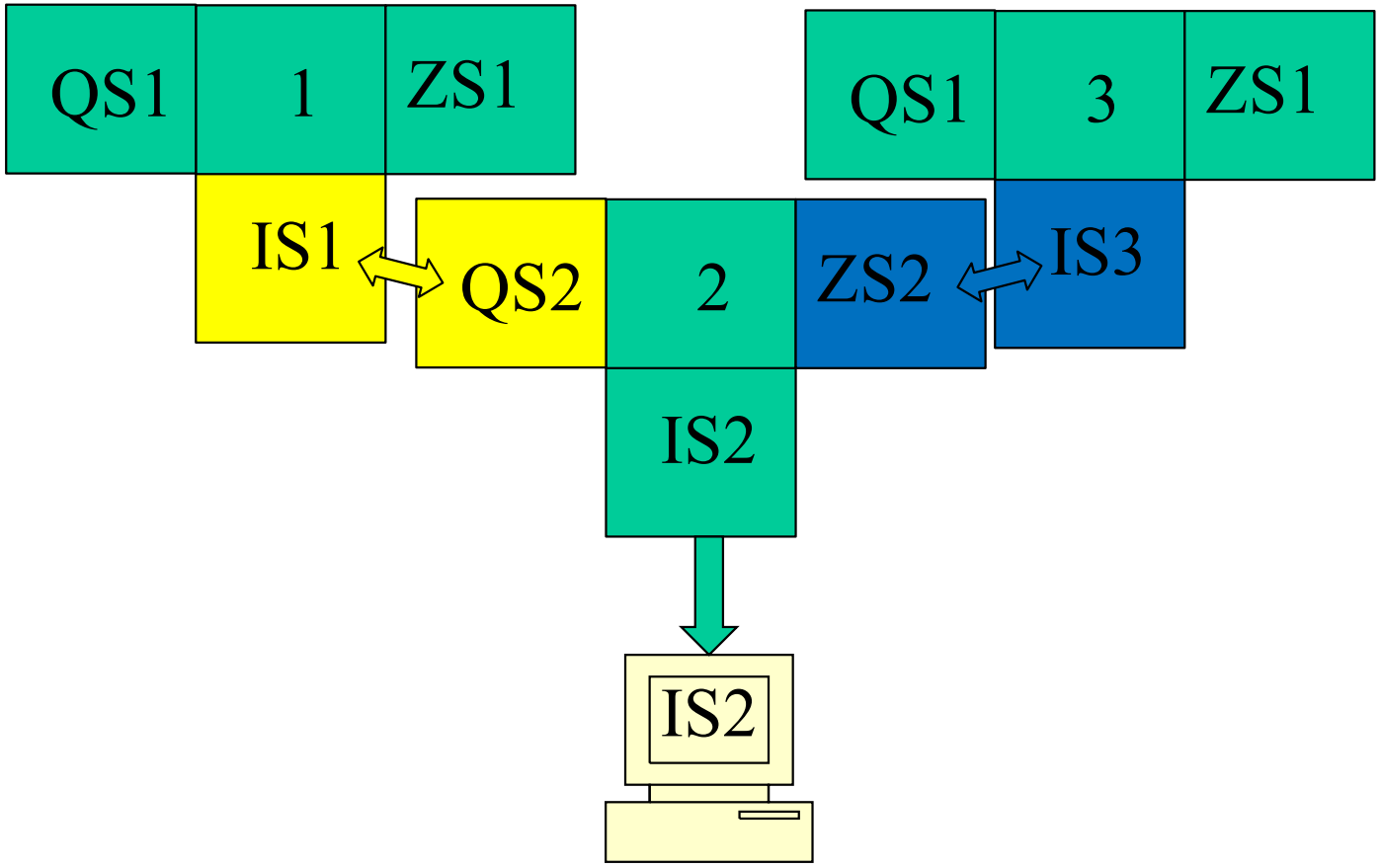
T-Diagramm

Vorgehensweise beim Bootstrapping:
Compiler 1 – Quelltext geschrieben in IS1-Sprache
wird mit Compiler 2 übersetzt:
- nur möglich falls IS1=QS2
- Benötigt System, das Compiler2 ausführt (Laufzeitsystem für IS2)



T-Diagramm

Vorgehensweise beim Bootstrapping:
Ergebnis des Compilierungslaufs ist Compiler 3:
- Übersetzt auch von QS1 nach ZS1, aber IS3 ist nun ZS2



T-Diagramm

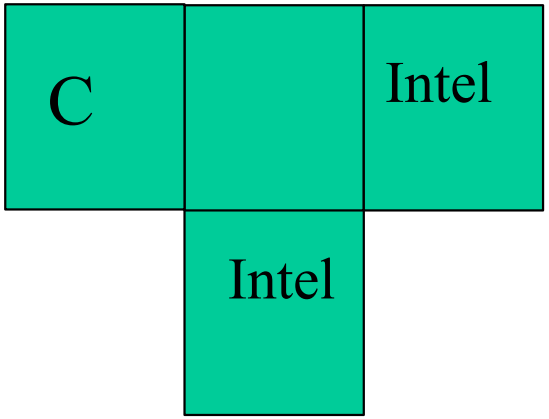
Beispiel Bootstrapping:

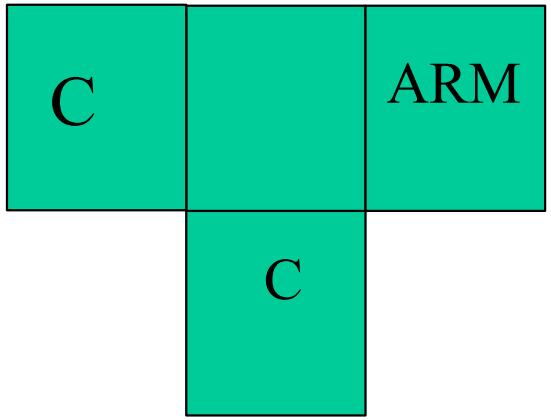
- beschriebene Vorgehensweise wird mehrmals angewendet
- Mehrstufige T-Diagramme veranschaulichen die notwendigen Compilierungschritte

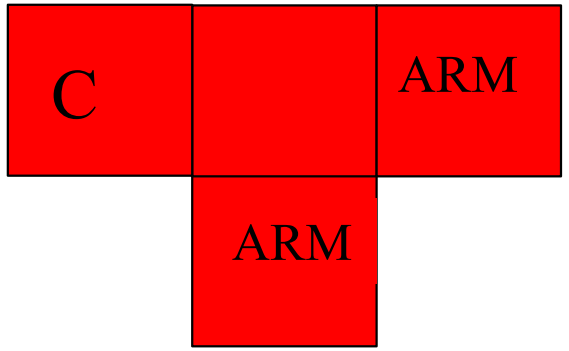
gegeben:
C-Compiler für Intel

gegeben:
C-Source-Text
C-Compiler für ARM

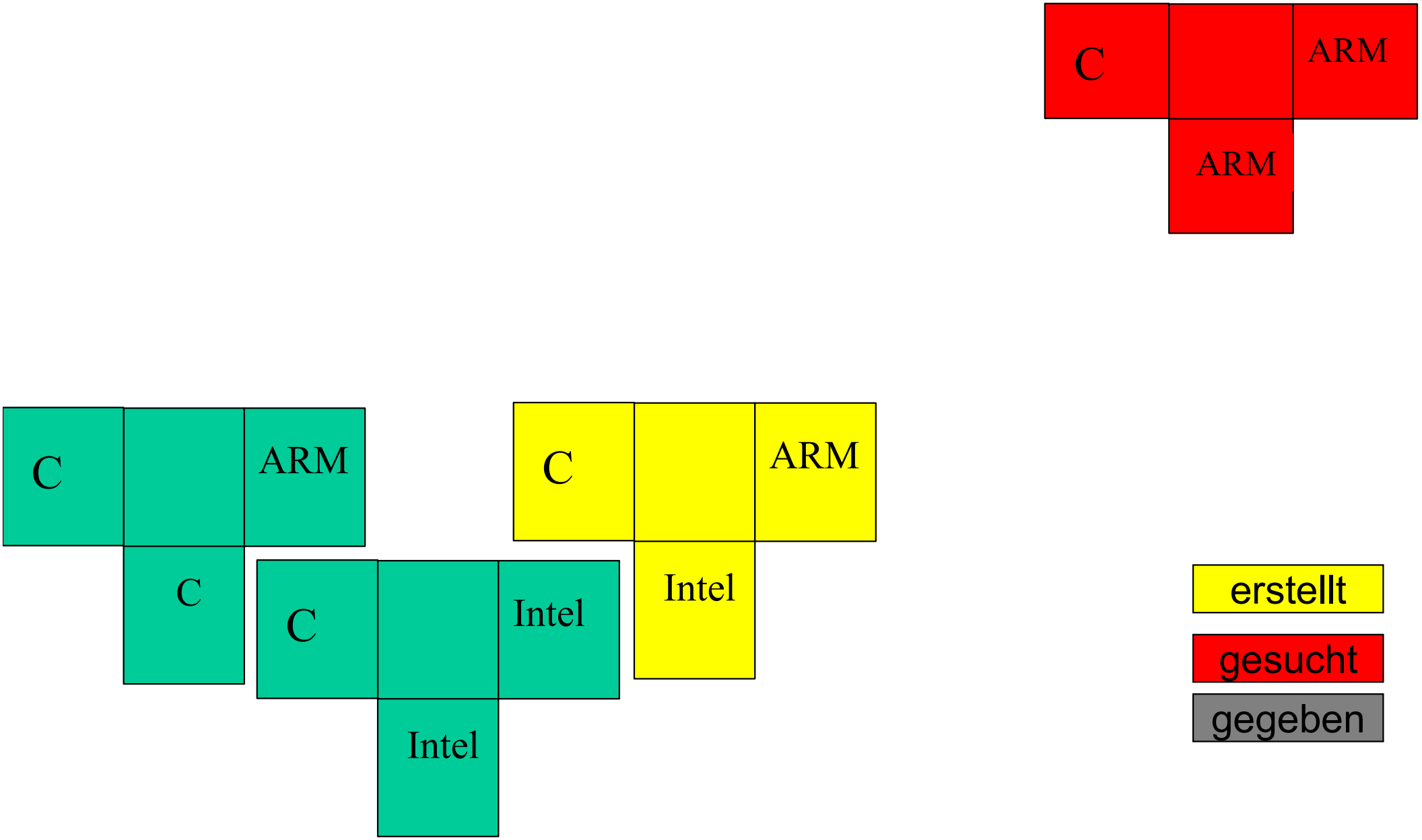
gesucht:
C-Compiler auf ARM



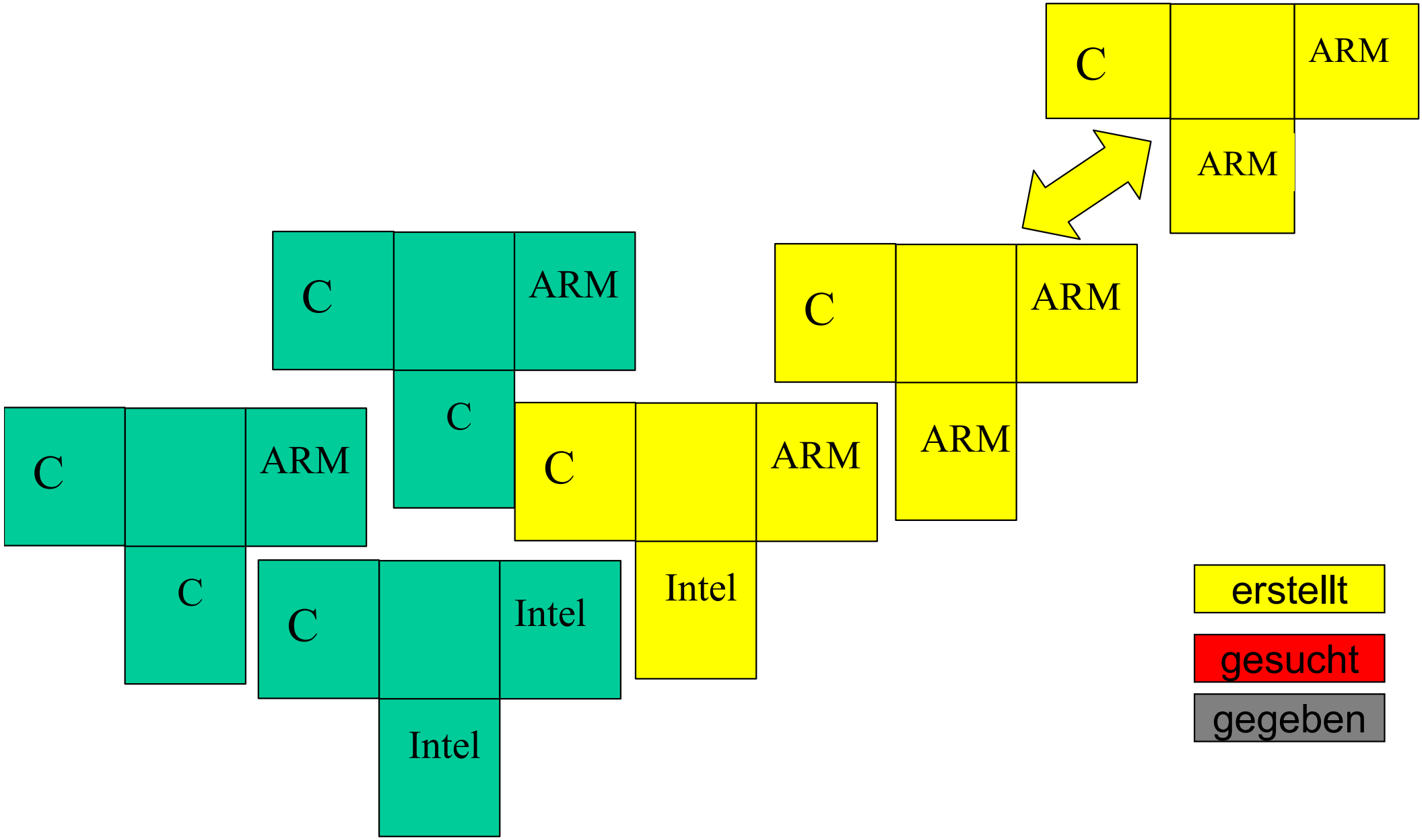




T-Diagramm


C
ARM
ARM
C
ARM
C
C
Intel
Intel
C
ARM
Intel
erstellt
gesucht
gegeben

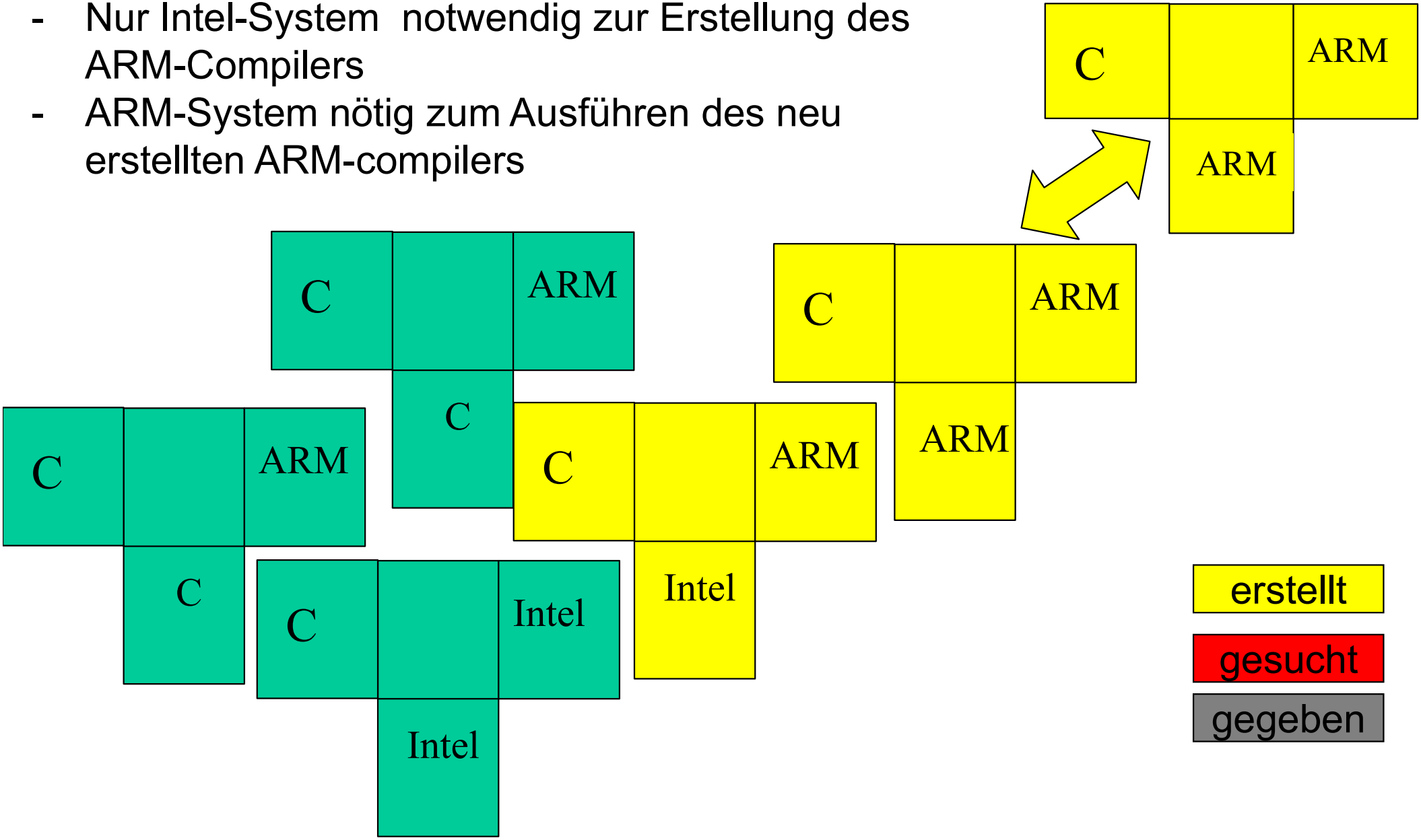
T-Diagramm


C
ARM
ARM
C
ARM
C
ARM
C
C
ARM
C
ARM
ARM
C
C
Intel
Intel
erstellt
gesucht
gegeben

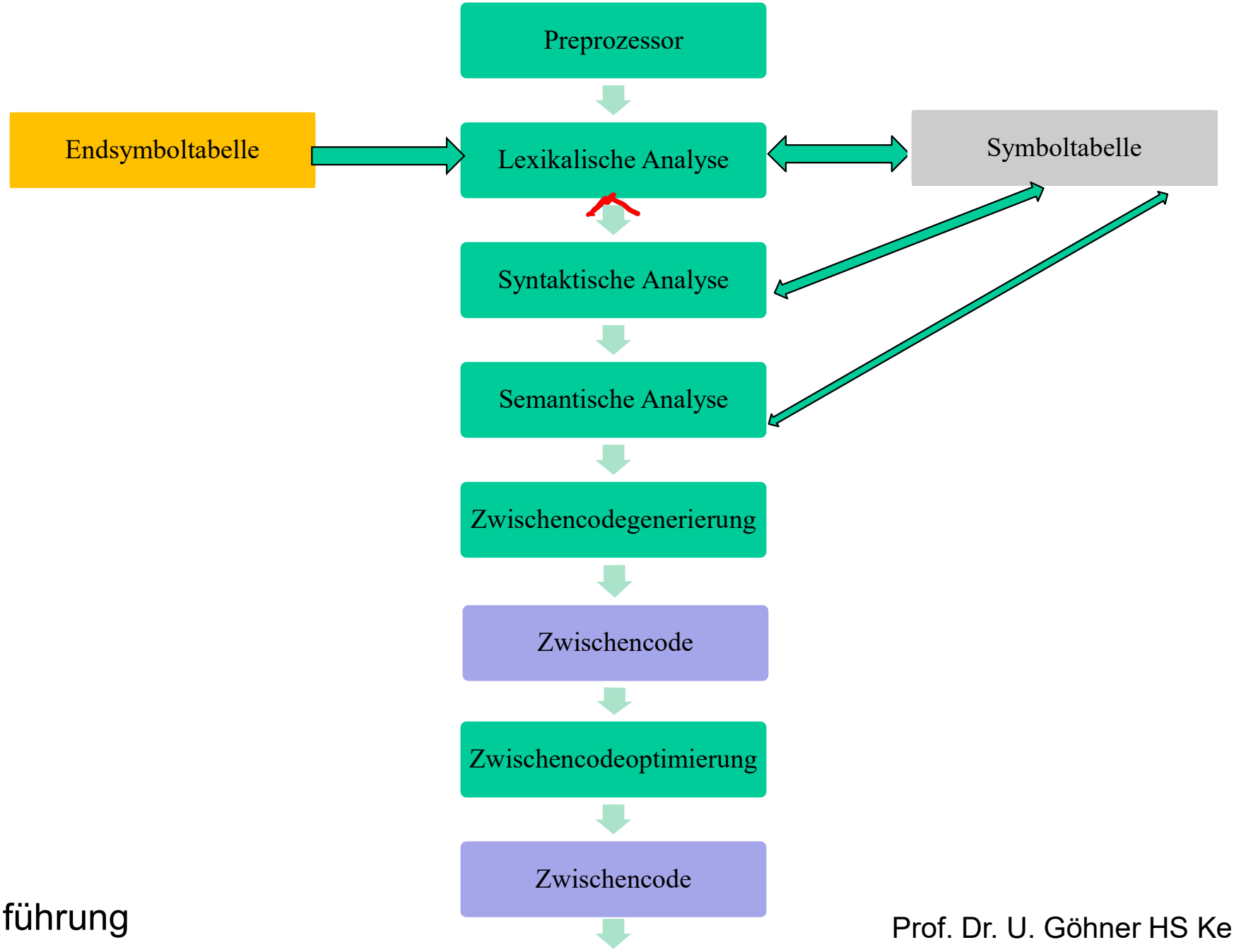
T-Diagramm

Laufzeitsystem:

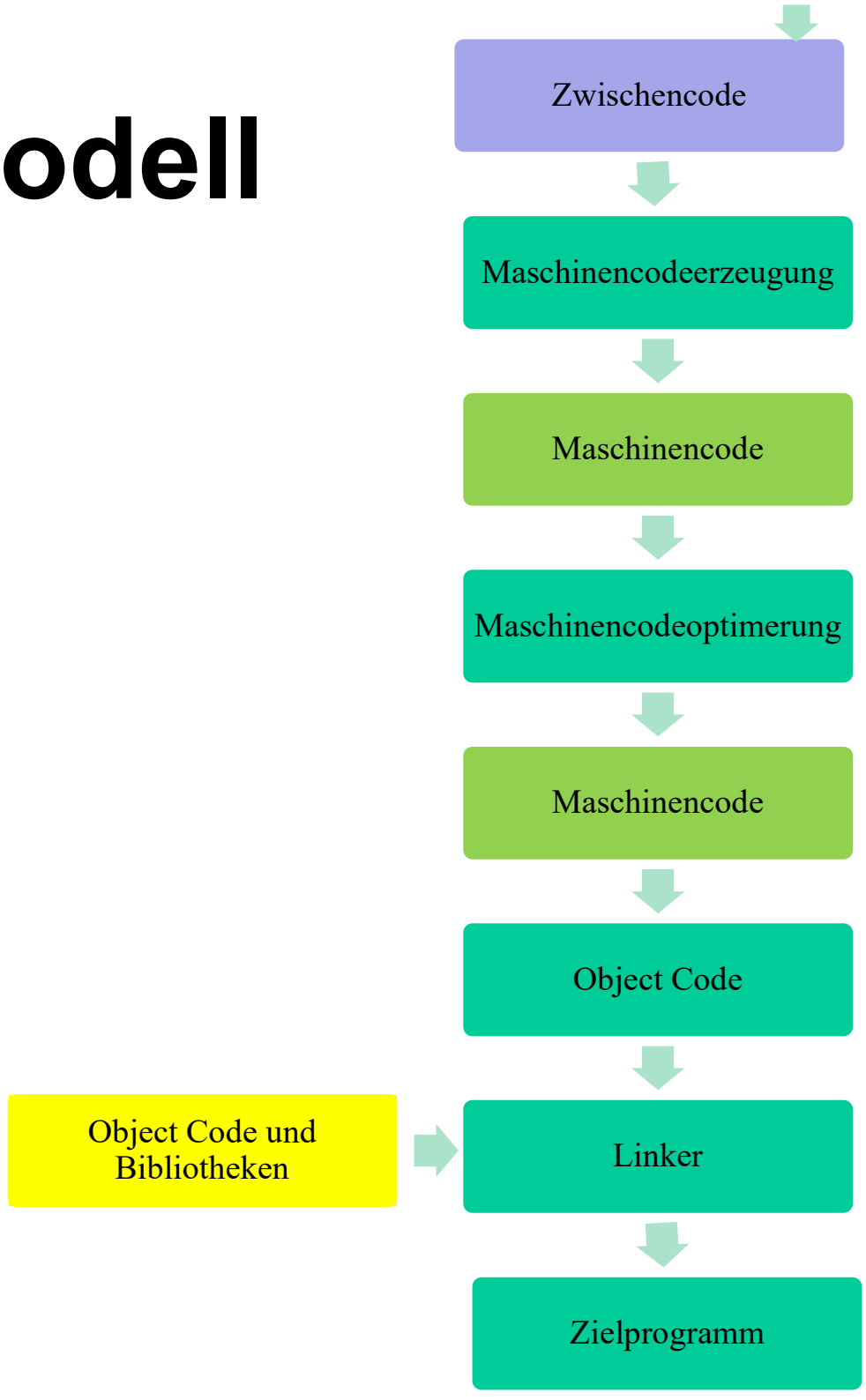
- Nur Intel-System notwendig zur Erstellung des ARM-Compilers
- ARM-System nötig zum Ausführen des neu erstellten ARM-compilers



Phasenmodell



Phasenmodell



Forderung an einen guten Compiler

- Erzeugt bei fehlerfreien Programmen korrekten Code
- kein Absturz des Compilers bei fehlerhaften Quell-Programmen
- findet alle Fehler im Quellprogramm
- Ausgabe von aussagekräftigen Fehlermeldungen
- Folgefehler werden als solche erkannt
- trotz Fehler wird die Analyse fortgeführt
- Fehlerbehandlung:
  - Umgang mit fehlerhaften Programmen
  - Ausgabe einer Fehlermeldung und Analyse des restlichen Quellprogramms
- Testwerkzeuge für Compiler: Compiler muss mehr fehlerhafte als korrekte Quelltexte übersetzen

Scanner

- Aufgaben des Scanners:
  - Eingabetext nach Mustern absuchen und Zuweisung von Token zu erkannten Mustern
  - Ignorieren von Leerzeichen, Tab, Kommentare
  - Einfache Berechnungen (z.B. Wert einer Konstanten)
  - Übergabe der Tokenklasse und der berechneten Werte (Zahl, Bezeichner,…) an den Parser
  - Eintrag der Bezeichner in die Symboltabelle

Scanner-Implementierung

Scanner werden entweder ad hoc programmiert oder mittels Scanner-Generator automatisch erzeugt

1. Ad hoc-Verfahren:
   - Scanner wird als Funktion implementiert, die vom Parser aufgerufen wird, um das nächste im Quelltext stehende Token zu erkennen.
   - Scanner liefert eine eindeutige Token-Identifikation als return-value
   - Falls der Scanner einen neuen Bezeichner gefunden hat, wird der Bezeichner in die Symboltabelle eingetragen

Scanner-Implementierung

- Scanner muss sehr effizient programmiert werden, da Quellprogramme aus sehr vielen Tokens bestehen und entsprechend viele Scanneraufrufe bei der Compilierung erfolgen.
- Pro eingelesenem Zeichen soll möglichst wenig Bearbeitungszeit benötigt werden
- Pufferung der Eingabe ist für effektive Bearbeitung beim Einlesen des Quelltexts sinnvoll
- Zur Erkennung des Token-Endes ist Vorausschau erforderlich (es genügt die Vorausschau um ein Zeichen)
- Um das Ende eines Bezeichners zu erkennen, muss der Scanner das Zeichen hinter dem Bezeichner gelesen haben (z.B. Space, Newline, Operator). Dieses Zeichen muss wieder „zurückgeschoben werden“ (unget)

Scanner-Implementierung

Am einfachsten wird die Einleseschleife, wenn folgende Vor- und Nachbedingung für den Scanneraufruf erfüllt werden:

- Beim Scanner-Aufruf und beim Rücksprung aus dem Scanner ist das aktuelle Zeichen im Eingabestrom entweder
  - das erste Zeichen des nächsten Token oder
  - ein Whitespace-Zeichen vor dem nächsten Token oder
  - das Eingabeende.
- zur Vorausschau gelesenes Zeichen, das nicht mehr zum Token gehört, wird in den Eingabestrom zurückgeschoben (unget- oder putback-Operation).

Scanner-Implementierung

- Schlüsselwörter und Bezeichner
  - In der Symboltabelle steht zu jedem Bezeichner die korrekte Klassifikation.
  - Schlüsselwörter sind in der Endsymboltabelle abgelegt
  - die korrekte Klassifikation (z.B. Variablentypisierung) wird dem Parser überlassen

Systematische Scanner-Implementierung

Grundlage für die Automatisierung der Scanner-Entwicklung ist das Konzept der endlichen Akzeptoren, die zur Prüfung der Token-Syntax bzw. Erkennung von Tokens verwendet werden.

Endliche Automaten / Endliche Akzeptoren (vgl THI)

- Endliche Automaten (EA) sind besonders einfache formale Maschinenmodelle, die im Compilerbau zur Spezifikation der lexikalischen Analyse dienen
- Das Verhalten eines EA wird durch Eingabe und Zustand bestimmt.
- Charakteristisch ist die Endlichkeit der Zustandsmenge, die einen EA von komplexeren Maschinenmodellen unterscheidet.

Systematische Scanner-Implementierung

DEA kann leicht in Programm-Code überführt werden

NEA:

- Alle möglichen Übergänge müssen durchprobiert werden
- Als Grundlage für Scannergenerierung sehr ineffektiv!

Da NEA für die Scanner-Implementierung zu ineffektiv ist, wird als Grundlage für den Scannergenerator ein DEA erstellt

Systematische Scanner-Implementierung

Generelle Vorgehensweise des Scannergenerators:

Gegeben reguläre Sprache, beschrieben durch regulären Ausdruck r

1. Erzeuge NEA aus regulärem Ausdruck r mittels Thompson-Algorithmus
2. Wandle NEA in DEA um
3. Minimiere die Anzahl der Zustände des DEAs
4. Erzeuge Scanner-Code aus DEA

Systematische Scanner-Implementierung

Thompson-Algorithmus erzeugt NEA aus regulärem Ausdruck r:

1. Erzeuge zunächst Teilautomaten für alle Eingabesymbole in r
2. Parallel/Reihenschaltung der Teilautomaten als analoge Automatentransformation für Konkatenation und oder - Operator
3. Rückführung als analoge Automatentransformation für die */+-Operatoren

Systematische Scanner-Implementierung

Vom nichtdeterministischen zum deterministischen Akzeptor

- Nichtdeterministisches Verhalten ist aufwändiger zu implementieren und ineffektiv
- nichtdeterministischen Akzeptoren werden in deterministische überführt.

Satz (siehe THI)

Zu jedem (nichtdeterministischen) endlichen Akzeptor A existiert ein äquivalenter deterministischer endlicher Akzeptor A_0 (d.h. A und A_0 akzeptieren die gleiche Sprache).

Konstruktion des DEA über Teilmengenbildung (THI)

Systematische Scanner-Implementierung

Anmerkung

- Zur Implementierung eines Scannergenerator benötigt man einen Parser für reguläre Ausdrücke
- Implementierung wird einfacher, wenn man nur einen Endzustand zulässt (ggf. neuen Endzustand und ε-Übergänge von den ursprünglichen Endzuständen in den neuen einführen)
- Man kann aus dem NEA durch Teilmengen-Konstruktion einen DEA ableiten (siehe THI)
- Es gibt andere Verfahren, die direkt einen DEA konstruieren (hier nicht behandelt)

Systematische Scanner-Implementierung

Satz (neu!)

Zu jedem (nichtdeterministischen) endlichen Akzeptor existiert ein äquivalenter deterministischer endlicher Akzeptor mit minimaler Zustandsmenge. Bis auf die Zustandsbenennung ist der minimale Akzeptor eindeutig bestimmt.

(ohne Beweis)

Systematische Scanner-Implementierung

Algorithmus:

Die Zustandsmenge von A_0 wird mit S bezeichnet. Jeder Zustand $z \in S$ repräsentiert eine Klasse äquivalenter Zustände aus der Zustandsmenge S des Ausgangsautomaten A.

Die Äquivalenzklasseneinteilung wird sukzessive durch Verfeinerung berechnet:

1. (Initialisierung der Äquivalenzklassen) S := { F, S \ F }

(Für $w=\varepsilon$ sind alle Endzustände und alle Nicht-Endzustände unterscheidbar)

2. Verfeinerung der Äquivalenzklassen (siehe nächste Seite)
3. Solange neue Klasseneinteilung feiner, wiederhole Verfeinerung, falls Klasseneinteilung sich nicht ändert, beende Verfeinerung
4. Zustandsübergänge sind durch die Übergänge der Repräsentanten definiert

Systematische Scanner-Implementierung

Algorithmus:

Verfeinerung der Äquivalenzklassen:

Für jeden Zustand z $\in$ S verteile die Zustände in der Klasse z so auf Teilklassen, dass zwei Zustände des Ausgangsautomaten s1, s2 $\in$ S genau dann in der gleichen Teilklasse z sind:

wenn für alle Eingabesymbole x $\in$ A gilt:

next(s1, x) und next(s2, x) sind in der gleichen Klasse von S

Führe die Aufteilung der Zustände in Teilklassen so durch, dass möglichst wenig Teilklassen entstehen

Ersetze die Zustände S durch die neu gebildeten Teilklassen.

Systematische Scanner-Implementierung

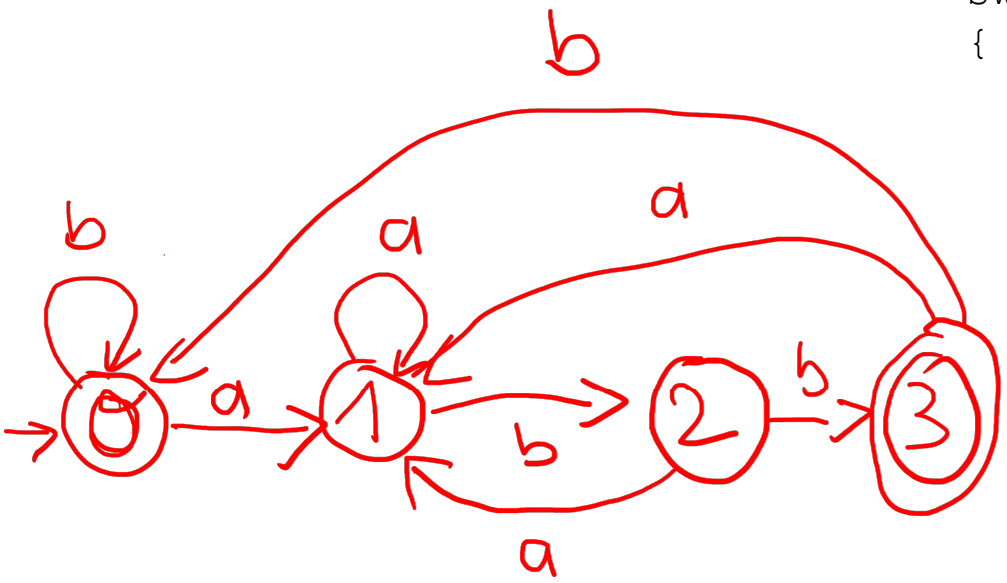
Implementierung des minimalen DEAs:

- Zustände durchnumerieren
- Ende der Eingabe-Zeichen beachten
- Eingabe Akzeptieren:
  - Automat ist in einem Endzustand
  - Eingabe ist vollständig gelesen (Ende der Eingabe ist aktuelles Zeichen)

Systematische Scanner-Implementierung

Implementierung des Automaten:

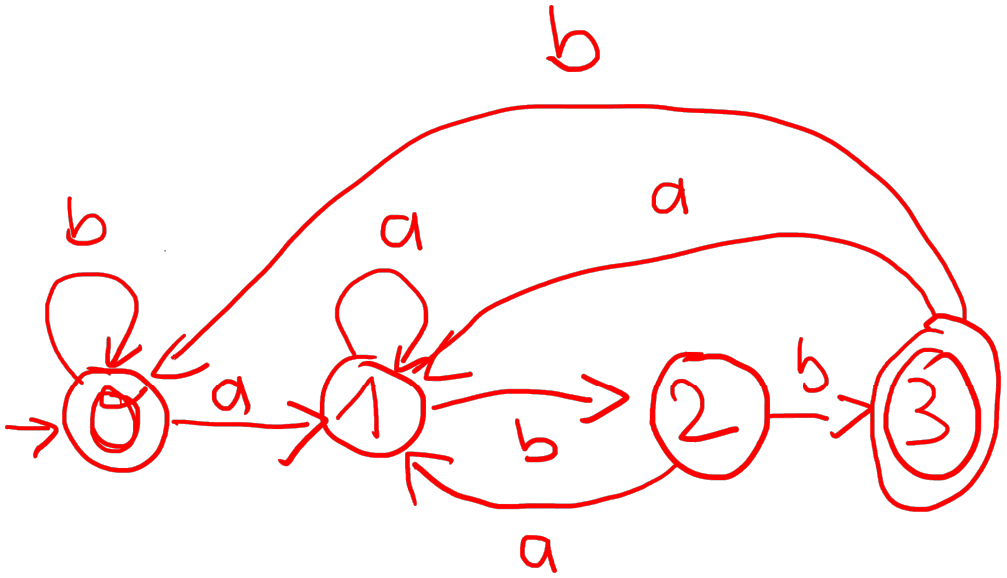
- Fallunterscheidung für Zustände
- Int-Variable zustand




```
#include "scanner.h"
int scanner()
{
  int zustand=0; // Anfangszustand
  char zeichen='0';
  while (zeichen !='\n')
  {
    zeichen = getchar(); // aktuelles Zeichen
    switch (zustand)
    {
      case 0:
        if (zeichen =='a')
         zustand=1;
         else if (zeichen =='b')
          zustand=0;
         else
          zustand=99; // Fehlerzustand = 99
        break;
```

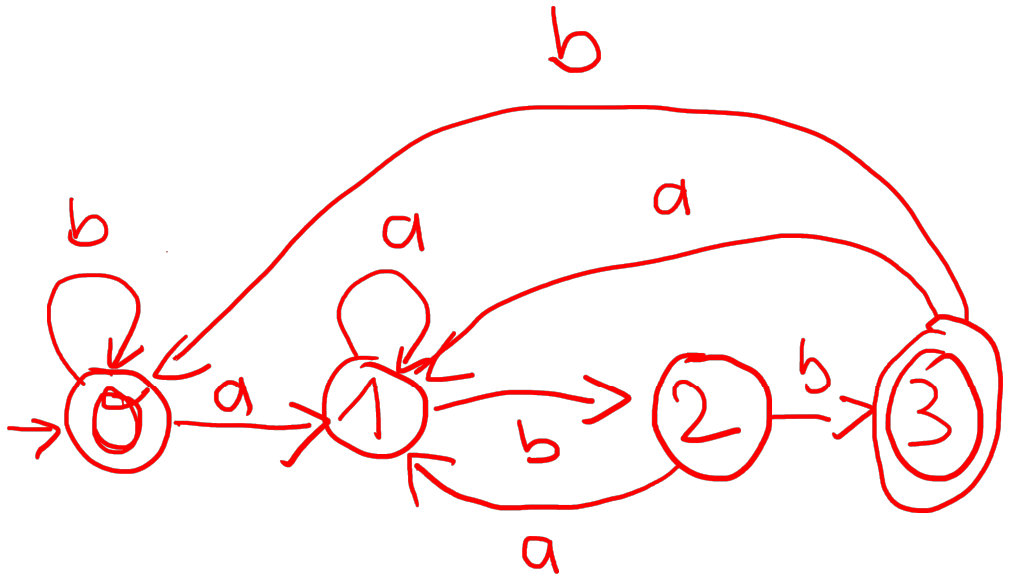
Systematische Scanner-Implementierung

```
case 1:
    if (zeichen =='a')
      zustand=1;
    else if (zeichen =='b')
      zustand=2;
    else
      zustand=99;
    break;
case 2:
    if (zeichen =='a')
      zustand=1;
    else if (zeichen =='b')
      zustand=3;
    else
      zustand=99;
    break;
```



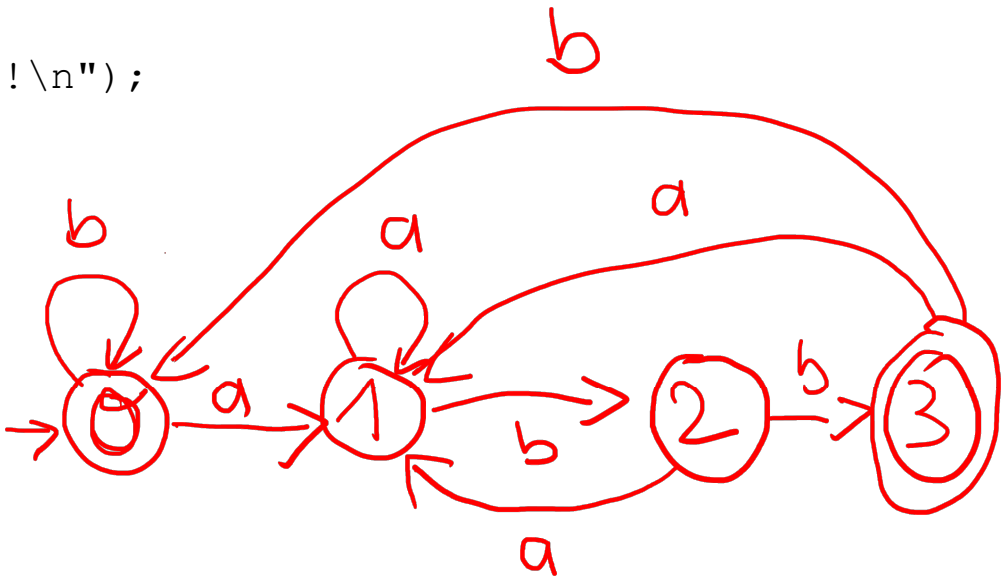
Systematische Scanner-Implementierung

```
case 3:
    if (zeichen =='a')
      zustand=1;
    else if (zeichen =='b')
      zustand=0;
    else if (zeichen =='\n') // Eingabeende nur im Endzustand erlaubt
      zustand=3;
    else
      zustand=99;
    break;
```



Systematische Scanner-Implementierung

```
  default:
    zustand=99;
     break;
  }
}
if ((zustand==3) && (zeichen =='\n')) // Endzustand && Eingabe gelesen
   printf("Eingabe akzeptiert!\n");
else
   printf("Eingabe nicht akzeptiert!\n");
return 0;
}
```



Systematische Scanner-Implementierung

Regeln beim Scanner-Bau

- nächstes Token wird stets als maximale mögliche gültige Zeichenkette bestimmt
- Die Bestimmung der Tokenklasse erfolgt entsprechend einer priorisierten Liste (d.h. Reihenfolge ist signifikant)

Systematische Scanner-Implementierung

Erzeugen von Scannern aus Grammatiken

- Wichtiges Einsatzgebiet von regulären Grammatiken sind Scanner
- Generierung des Scanner-Codes basierend auf Sprachbeschreibung mit regulären Ausdrücken

Systematische Scanner-Implementierung

Scanner-Generatoren:

- erzeugen aus einer formalen Beschreibung der lexikalischen Syntax (reguläre Ausdrücke) einen Scanner.
- lex, flex,... sind auf vielen Rechnerplattformen verfügbar.
- erzeugen den Scanner in Form eines C-Quelltexts, so dass man an dem generierten Scanner noch Modifikationen auf der C-Ebene vornehmen kann.
- Auch mittels Option auf C++ umschaltbar
- Visual CC (Compiler Compiler) enthält Scanner- und Parsergenerator

Systematische Scanner-Implementierung

Weitere Scanner-Generatoren:

- Jflex: Java-Implementierung von Flex
- VisualCC (AtoCC, Java, C++, C#)
- JavaCC (Java Compiler Compiler)
- Coco/R (Java, C++, C#)
- ANTLR (Another Tool for Language Recognition)

Systematische Scanner-Implementierung

Funktionsweise von lex:

- Eingabedatei gibt für jedes Token an:
  - die Syntax in Form eines regulären Ausdrucks
  - eine Aktion (C-Anweisung), die bei Erkennung des Tokens vom Scanner ausgeführt werden soll
- Der Generator erzeugt aus der Eingabedatei den Quelltext des Scanners, der aus der Definition der Scanner-Funktion yylex und den benötigten Datenstrukturen und Hilfsfunktionen besteht.
- Der generierte Scanner wird compiliert und das erzeugte Objektmodul zusammen mit dem Parser und den anderen Modulen zum lauffähigen Compiler gebunden.

Systematische Scanner-Implementierung

grober Aufbau des lex-input-files (<name>.l)

Erster Teil: optional, besteht aus:

Vorspann wird im Programm von lex unverändert eingesetzt:

meist #includes, #defines, C-Code

Abkürzungen

Abkürzende Bezeichnungen für reguläre Ausdrücke -> nur für Lex intern benutzt

%%

Zweiter Teil: obligatorisch

Definition der **regulären Ausdrücke** für die Tokens

%%

Dritter Teil: optional

Nachspann, meist C Prozeduren, die 1:1 übertragen werden.

Systematische Scanner-Implementierung

Aufbau im Detail:

Erster Teil:

Steuerzeichen innerhalb des ersten Teils:

%{ für Beginn Vorspann

%} für Ende Vorspann

Danach folgen Definitionen für abkürzende Schreibweise in den regulären Ausdrücken des zweiten Teils in folgender Form:

Bezeichner regulärer Ausdruck

z.B.: Ziffer [0-9]

Bezeichner werden mittels { Bezeichner } referenziert.

Zahl {Ziffer}+

Systematische Scanner-Implementierung

Aufbau im Detail:

Zweiter Teil:

Folge von Zeilen der Form:

regulärer Ausdruck {Aktionen }

{Aktionen} sind C++-statements, die ausgeführt werden, wenn regulärer Ausdruck erkannt wird

Die Aktionen werden unverändert in lex.yy.c übernommen.

Systematische Scanner-Implementierung

Aufbau im Detail:

Dritter Teil:

enthält Hilfsfunktionen, z.B. main oder yywrap

(aufgerufen am Ende der Eingabe)

Systematische Scanner-Implementierung

Konflikte bei der Auswertung von Mustern:

Es gelten folgende Regeln:

- Falls mehrere reguläre Ausdrücke anwendbar sind, dann wird der ausgewählt, der zum längsten Eingabe-String passt.
- Passt eine Zeichenkette (identischer EingabeString) zu mehreren regulären Ausdrücken, dann führt lex die Aktion der ersten passenden Regel aus (bei gleicher Stringlänge).