EINFÜHRUNG

(Montag, den 28. April 2003)

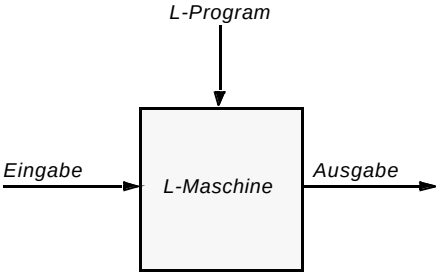
Implementierung von Programmiersprachen

Implementierung von Übersetzern

***Nur zur Information*: Das PL0-System**

Maschinenmodell

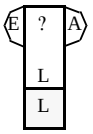
universelle programmierbare Rechenmaschine:



Charakteristika

Kann beliebige Programme einer Sprache L ausführen
d.h. kann Ausgaben für beliebige Eingaben berechnen

Uns interessiert primär die Sprache L,
die die Maschine ausführen kann
Ausführung eines Programms wird so symbolisiert:



konkrete Maschine

Wie kann man eine L-Maschine bauen?

für eine höhere Programiersprache L

Idee

Baue L in *Hardware* (konkrete Maschine)

Bemerkungen

“höhere Maschinensprachen”
sind im allgemeinen zu teuer und zu kompliziert
Außerdem:
solche Maschinen sind nicht mehr “flexibel”
Ausnahme
Silicon compilers

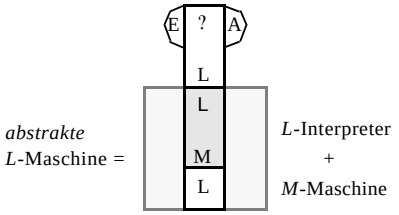
bessere Idee

realisiere die L-Maschine
als Programm für eine konkrete Maschine M

Interpreter

Simuliere L mir einem M-Programm

Ein Interpreter definiert eine *abstrakte* L-Maschine:



Vorteil:

flexible konkrete Maschine M
spezielle *abstrakte* Maschine

Nachteil

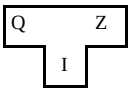
Programme und Eingabe werden *gleichzeitig* bearbeitet (*overhead*)

Trotzdem

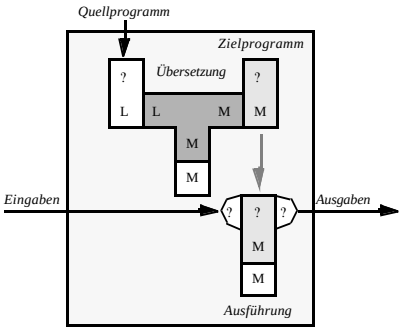
Für einfache Sprachen interessant
einfach zu implementieren

Übersetzer

Übersetze L-Programme in M-Programme



Auch Übersetzer definieren *abstrakte* Maschinen:



Vorteil

Quell-Programm wird nur *einmal* bearbeitet

kein *overhead* bei der Ausführung des Ziel-Programms

Nachteil

etwas aufwendiger

(man muß ein *korrektes* Programm konstruieren!)

Implementierungssprache

In welcher Sprache soll man Übersetzer schreiben?

Anforderungen:

Rekursion

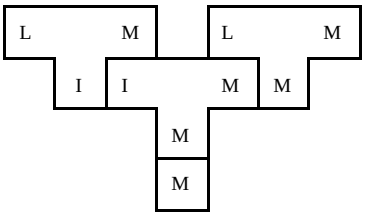
komplexe Datenstrukturn (Bäume, Graphen, Tabellen)

Modularität

Folgerung: in höheren Programmiersprachen!

Aber

Dann braucht man für den Übersetzer einen Übersetzer!



Übertragung von Übersetzern

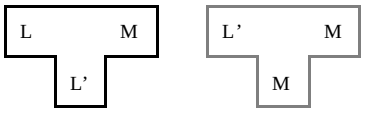
re-hosting

re-targetting

“re-sourcing” (Übersetzer-erzeugende Systeme)

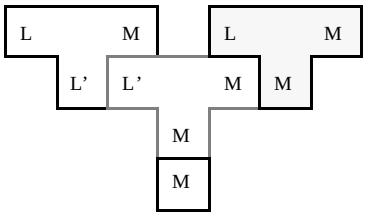
Bootstrapping (*re-hosting*)

Gegeben:

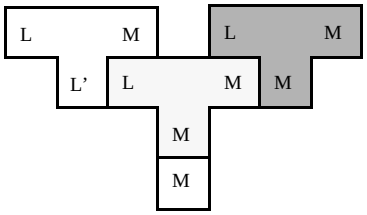


Quellübersetzer “Einweg”-Übersetzer

Erstübersetzung:



Selbstübersetzung:



“Einweg”-Übersetzer:

implementiert nur die benutzte Teilsprache

kann sehr ineffizient sein

Beispiel: der SYSTEAM Ada-Übersetzer

(Ada ? Pascal? Maschinensprache)

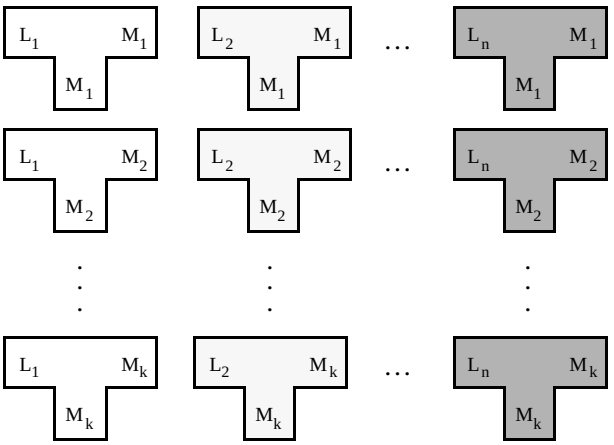
re-hosting und *re-targetting*

Problem

n Sprachen L_1 L_2 .... L_n

k Rechner M_1 M_2 .... M_k

Man braucht *n*k* Übersetzer

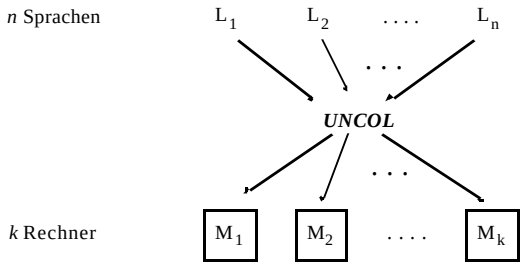


Wie kann man diesen Aufwand reduzieren?

Der *UNCOL*-Ansatz

(UNiversal Communication Oriented Language, ~1960)

Definiere eine universelle Zwischensprache:



Vorteil

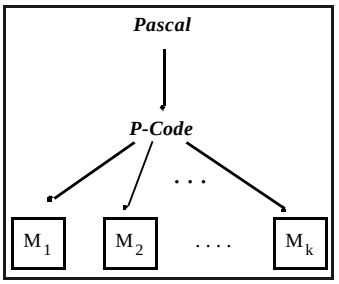
Man braucht nur *n* Übersetzer nach *UNCOL*
und *k UNCOL*-Übersetzer für alle Maschinen!

Leider gibt es keine *effiziente* Sprache *UNCOL*

Sprachen sind zu verschieden
Rechner sind zu verschieden

Definition einer *abstrakten* Maschine für die Quellsprache

Das Beispiel Pascal:



Damit kann man gut leben:

man braucht nur einen Übersetzer von Pascal nach P-Code
und P-Code-Übersetzer oder Interpreter
für alle Maschinen

Ziele beim Definieren der Zwischensprache:

maschinen-unabhängig
möglichst effizient auf allen Zielmaschinen realisierbar

Das ist schwierig

Werkzeuge im Übersetzerbau

“re-sourcing”

Wiederverwendung für andere Quellsprachen

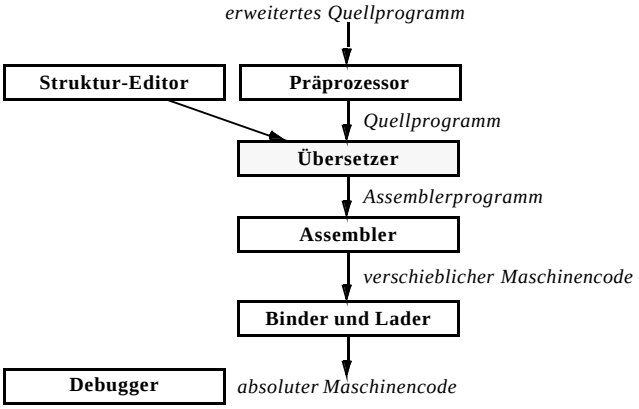
Übersetzer-erzeugende Systeme
(*compiler generators*, *compiler compilers*)

Für Phasen der Übersetzung werden Werkzeuge entwickelt, die
bestimmte Moduln des Übersetzers
aus speziellen Spezifikationen erzeugen

Das wird hier nicht weiter behandelt

Umgebung eines Übersetzers

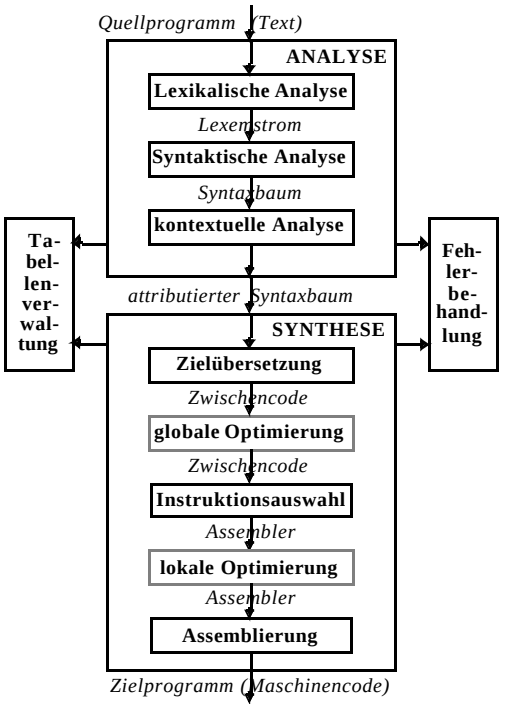
Programme rund um einen Übersetzer



Phasen der Übersetzung

***Analyse* des Quellsprache Q**

***Synthese* der Zielsprache Z**



Pässe

***Pässe* sind nacheinander ablaufende Teilprogramme des Übersetzers**

Phasen sind nur *konzeptionelle logische* Pässe

Phasen werden nach Effizienz-Gesichtspunkten zu Pässen zugeordnet

Beispiel

Lexikalische und syntaktische Analyse werden oft zu einem Pass zusammengefaßt.

Sie kommunizieren über einen Puffer;

Der Scanner liefert ein weiteres Symbol, wenn der Parser es verlangt

Aber

Es gibt kein *feedback* zwischen den Phasen!

z. B. sollte der Kontext-Prüfer das Verhalten des Scannners nicht beeinflussen

Sonst gibt es *backtracking* (und eine unübersichtliche Struktur)

Zwischensprachen

Programm -Darstellung in der für die Phase günstigsten Form

Lexemstrom, Syntaxbaum, Zwischencode

Zwischensprache müssen nicht unbedingt Texte sein

z.B. Lexem*folge*, Syntax*baum*, Fluß*graph,* Code-*Feld*

Zwischensprachen müssen nicht vollständig sequenziell konstruiert werden

(siehe Beispiel Lexikalische und syntaktische Analyse)

Tabellen

Einige Informationen werden oft und in mehreren Phasen benutzt

In vielen Sprachen sind dies *Bezeichner* (*identifier*)
Sie werden in Tabellen abgespeichert

Beispiele

- Repräsentationstabelle
- Deklarationstabelle
- Adreßtabelle

Die Tabellen sollen helfen, diese Information schnell wiederzufinden

Für jede Phase der Übersetzung sind andere Eigenschaften der Bezeichner relevant:

lexikalische Analyse: Repräsentation *(String Table)*
kontextuelle Analyse: Art und Typ der Vereinbarung, ggf. weitere globale (überdeckte) Vereinbarungen *(Declaration Table)*
Adressberechnung: Level und relative Adresse, ggf. Register/Speicherobjekt *(Adreßtabelle)*

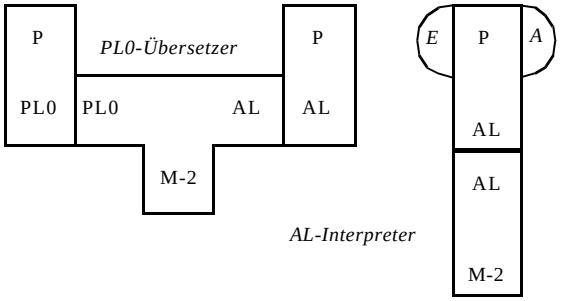
Beispiel PL0-Übersetzer

Deklarations- und Adreßtabelle sind zu einer *ObjectTable* verschmolzen.

Grobstruktur des PL0-System

Das PL0-System wurde von Niklaus Wirth für sein Buch *Compilerbau* (4. Aufl., Teubner 1986) "erfunden"
Seine Struktur ähnelt dem Züricher Pascal-Übersetzer

Maschinenstruktur:



Eine Schleife liest Programmdateien, übersetzt sie und führt sie mit Eingabewerten aus
PL0 wird in eine abstrakten Maschinensprache *AL* übersetzt
die AL-Programme werden interpretiert
DerAL-Code wird in einem Feld gespeichert

Die Sprache AL reicht aus,
um auch PL5-Programme zu übersetzen

Der AL-Interpreter muß also *nicht* erweitert werden.

Die "Programmier"-Sprache PL0

Die Sprache ist klein, aber geeignet,
wichtige Prinzipien der Übersetzung zu demonstrieren

PL0-Programme sind *Blöcke*
Blöcke bestehen aus *Vereinbarungen* (*declarations*) und einer *Anweisung* (*statement*)
Als Datenobjekte können ganzzahlige *Variable* und *Konstanten* vereinbart werden.
Zudem gibt es *Prozeduren* ohne Parameter
(deren Rumpf Blöcke sind und lokale Vereinbarungen enthalten)
Anweisungen sind

- Wertzuweisung *v := e*
- Lese-Anweisung *? v*
- Schreib-Anweisung *! e*
- Verbund-Anweisung **begin** s; ... **end**
- bedingte Anweisung **if** c **then** s;
- Wiederholungsanweisung **while** c **do** s;
- Prozeduraufruf **call** p

Bedingungen (*condition*)
können mit Vergleichsoperationen aufgebaut werden
Ausdrücke (*expression*)
bestehen aus *Bezeichnern* (*identifiers*)
und ganzen Zahlen mit den üblichen arithmetischen Operationen

Die Syntax von PL0

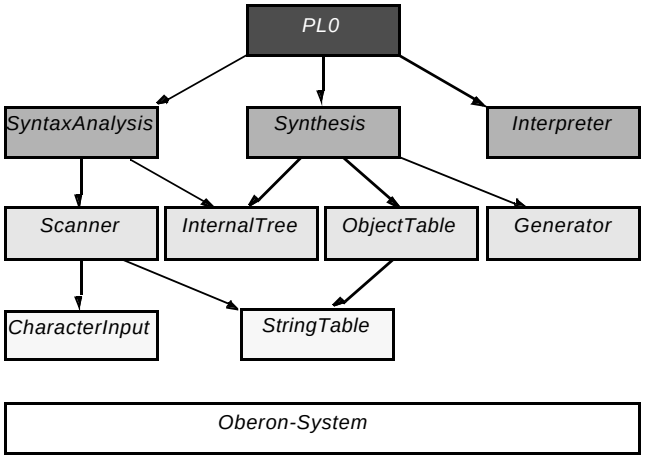
```
program  ::= block .

block    ::= { declaration }
             begin
             statement { ; statement }
             end

declaration   ::= const      ident = number
                             {, ident = number } ;
                | var        ident {,  ident} : integer;
                | procedure ident ; block ;

statement::=
            [ ident := expression
            | call ident
            | ?    ident
            | !    expression
            | begin statement {; statement} end
            | if condition then statement
            | while condition do statement
            ]

condition  ::= odd expression
             |  expression (=|#|<|>|<=|>=) expression

expression ::= [+|-] term { (+|-) term}

term       ::= factor { (*|/) factor}

factor     ::= ident
             |  number
             |  ( expression )
```


Ein PL0-Programm
(Größter gemeinsamer Teiler)

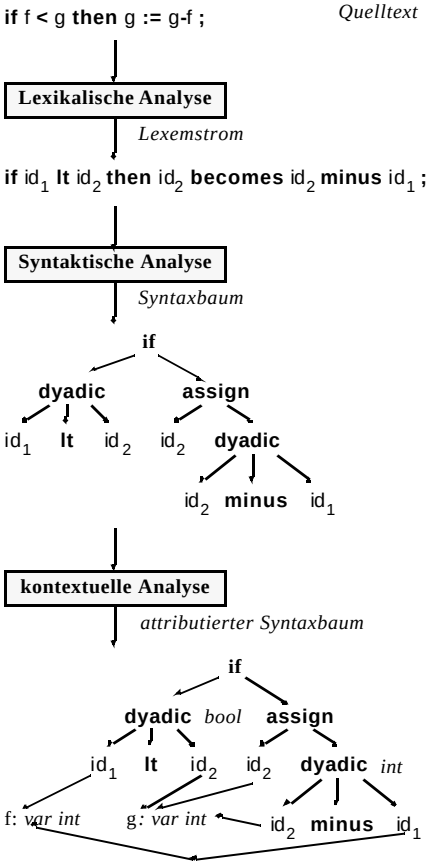
```
(* größter gemeinsamer Teiler *)
var x, y: INTEGER ;
begin
   ? x; ? y;
   if x > 0 then
      if y > o then
         begin
         ! x ; ! y ;
         while x # y do
            begin
            if x < y then y := y-x ;
            if x > y then x := x-y ;
            end
         end ;
   ! x
end.
```

Phasenstruktur des PL0-Systems

Modulstruktur:



Analyse von PL0



Synthese von AL