Formale Sprachen und Compiler

Überblick

Prof. Dr. Franz-Karl Schmatzer
schmatzf@dhbw-loerrach.de

Literatur

- C.Wagenknecht, M.Hielscher; Formale Sprachen, abstrakte Automaten und Compiler; 3.Aufl. Springer Vieweg 2022;
- U.Meyer; Grundkurs Compilerbau; Rheinwerkverlag, 1. Aufl. 2021
- A.V.Aho, M.S.Lam,R.Savi,J.D.Ullman, *Compiler – Prinzipien,Techniken und Werkzeuge*. 2. Aufl., Pearson Studium, 2008.
- Güting, Erwin; *Übersetzerbau –Techniken, Werkzeuge, Anwendungen*, Springer Verlag 1999

Grundlegende Konzepte

- Motivation
- Sprachprozessoren
- Struktur eines Compilers
- Entwicklung der Programmiersprachen
- Methodik des Compilerbaus
- Anwendung der Compilertechnologie
- Tools für diese Vorlesung

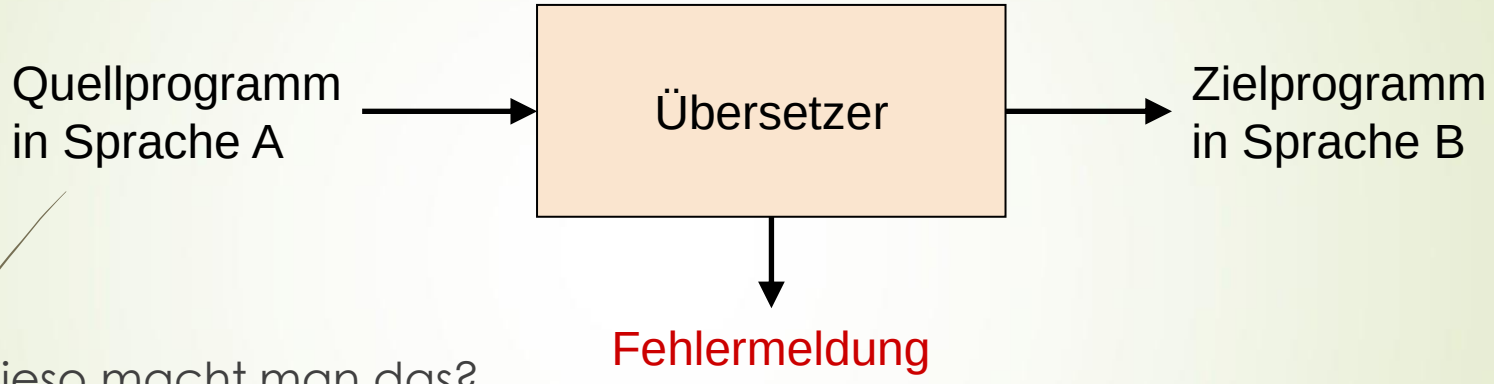
Motivation

Wieso wird heute noch Compilerbau gelehrt?

- Gehört zur Allgemeinbildung eines Informatikers wie
  - Datenbanken
  - Erlernen einer Programmiersprachen
  - Internet-Technologien
- Einzelne Techniken und Tools werden immer wieder verwendet:
  - Entwickeln von Beschreibungssprachen (LaTex, HTML, SGML)
  - Datenbankanfragesprachen (SQL, XQuery)
  - VLSI Entwurfsprachen (Layout von Chips)
  - Entwickeln von Protokollen in verteilten Systemen
  - Entwickeln von Sprachen für spezielle Systeme

Interpreter und Compiler

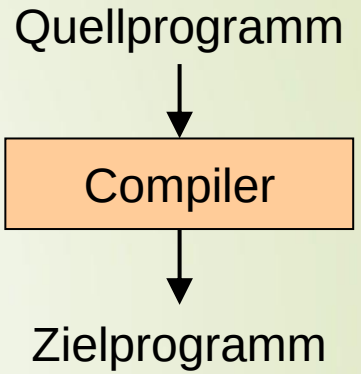
- Bau eines Übersetzers (Compiler) für formale Sprachen im weitesten Sinnes. D.h. Das Übersetzen von einem Quellprogramm in ein Zielprogramm und der Ausgabe einer Fehlermeldung.

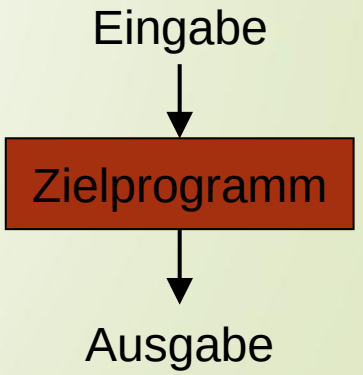


- Wieso macht man das?
  - Man kann etwas besser in Sprache A beschreiben, aber die Maschine versteht nur Sprache B, oder man hat nur eine Maschine die Sprache B versteht.
- Solche Systeme nennt man auch allgemein Sprachprozessoren
- Man unterscheidet i. W. 2 Typen von Sprachprozessoren
  - Compiler und Interpreter

Sprachprozessoren

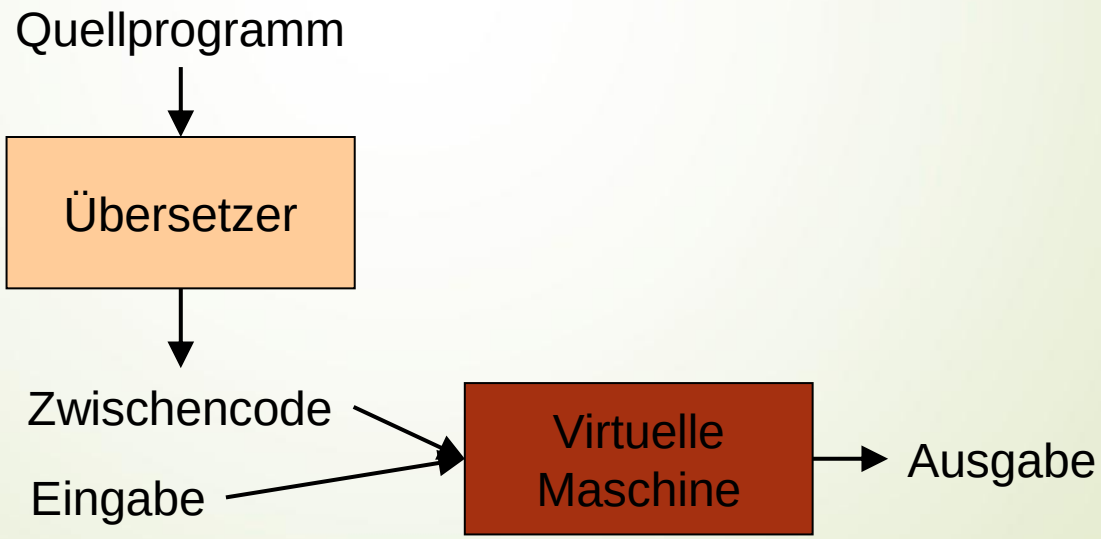
- Was ist ein Compiler?
  - Übersetzer eines Programms von einer Quellsprache in eine Zielsprache
  - Wichtige Rolle des Compilers ist eine Fehleranalyse
  - Ist die Zielsprache ein ausführbares Programm in Maschinensprache, kann es anschließend direkt aufgerufen werden.
- Was ist ein Interpreter?
  - ein andere Form eines Sprachprozessor, der die Operationen direkt ausführt, ohne ein Zielprogramm zu erstellen.
- Compilergenerierte Programme sind normalerweise viel schneller als eine Ausführung durch einen Interpreter
- Aber eine Fehleranalyse eines Interpreters ist meistens besser.





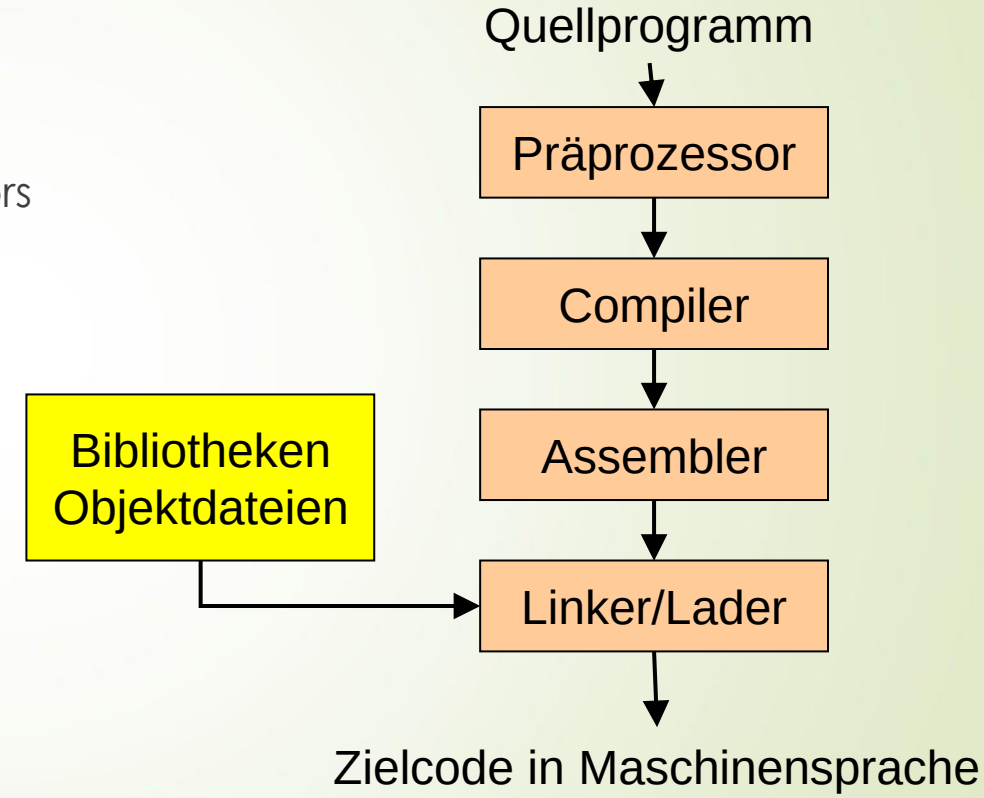
Sprachprozessoren – Beispiel 1

- Java Sprachprozessoren
  - kombinieren Kompilierung und Interpretation
  - Es wird eine Zwischenform , sogenannter Bytecode, erzeugt.
  - Der Bytecode wird von der virtuellen Maschine interpretiert



Sprachprozessoren – Beispiel 2

- Neben dem Compiler können noch andere Programme benötigt werden
  - Aufspalten in Module
  - Verwenden eines Präprozessors
  - Assembler
  - Linker/Lader



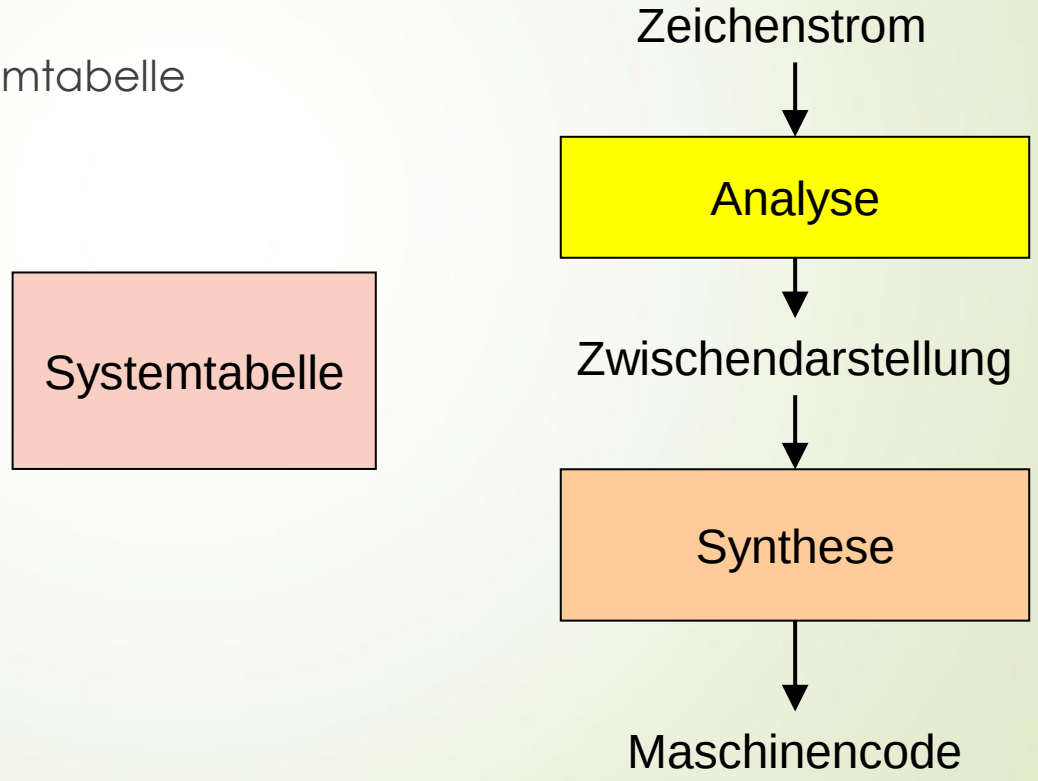
Interpreter und Compiler

Recherchieren Sie beliebte Computersprachen

- Welche sind Interpreter?
- Welche sind Compiler-Sprachen?
- Welche sind Auszeichnungssprachen?

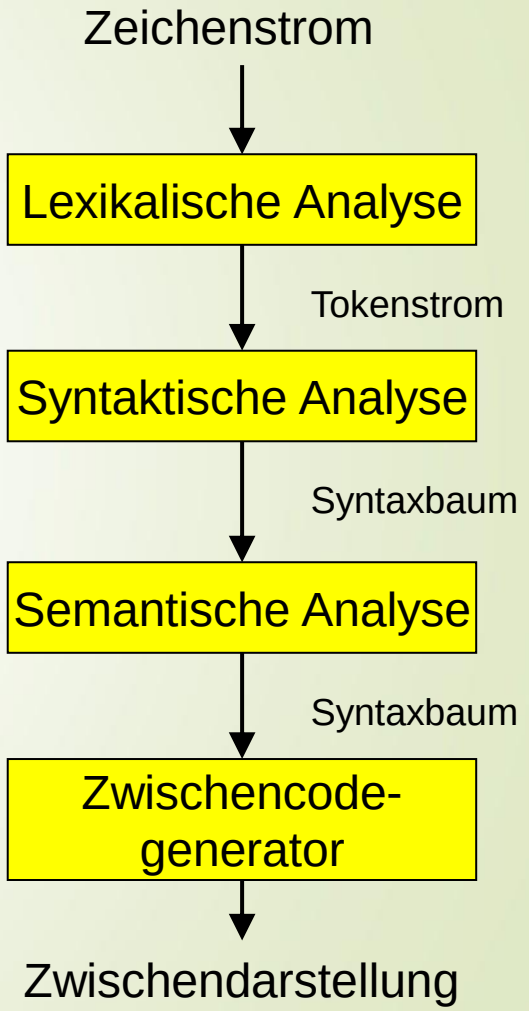
Struktur eines Compilers

- Ein Compiler teilt man in 2 große Blöcke
  - Analyse und
  - Synthese
  - mit einer Systemtabelle



Struktur eines Compilers
Analyse

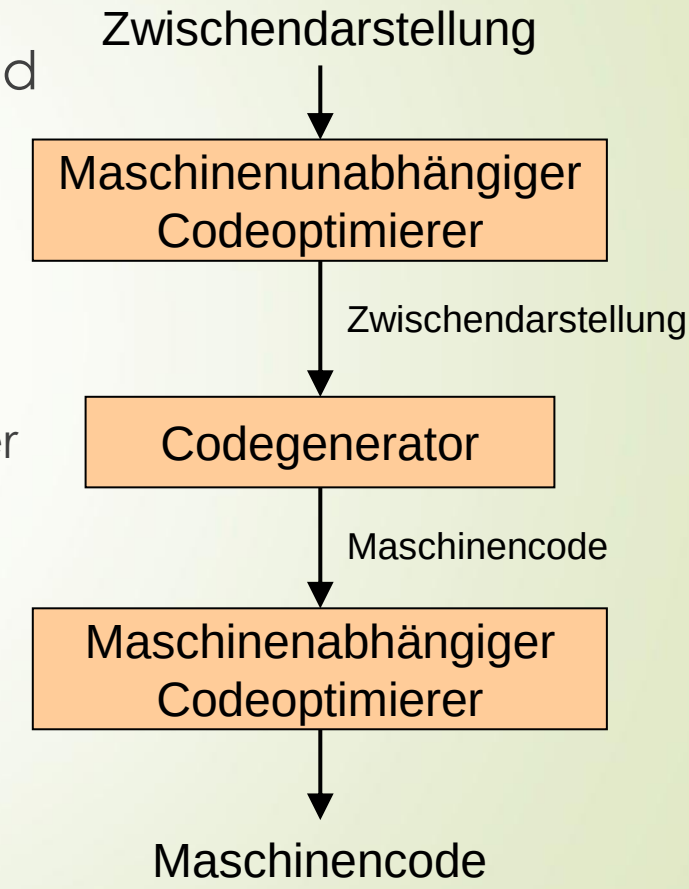
- Hier wird das Programm zerlegt und mit grammatischen Strukturen versehen.
- Dann wird eine Zwischendarstellung des Programms erstellt.
- Im Fehlerfall (syntaktischer oder Semantischer Art) wird eine Fehlermeldung generiert.
- Außerdem wird eine Symboltabelle erstellt.
- Die Analyse wird üblicherweise weiter verfeinert in
  - Lexikalische Analyse
  - Syntaktische Analyse
  - Semantische Analyse
  - Zwischencodegenerator
- Der Analyseteil des Compilers nennt man Front-End



Struktur eines Compilers - Synthese

Synthese

- Hier wird aus der Zwischendarstellung und der Symboltabelle das gewünschte Zielprogramm, der Maschinencode konstruiert.
- Die Synthese wird weiter verfeinert in
  - Maschinenunabhängiger Codeoptimierer
  - Codegenerator
  - Maschinenabhängiger Codeoptimierer
- Diesen Teil des Compilers nennt man Back-End.



Lexikalische Analyse

- Erste Phase einer Kompilierung.
- Nennt man auch Scannen.
- Der Zeichenstrom wird in sinnvolle Sequenzen, den Lexemen eingeteilt.
- Für jedes Lexem wird ein Token folgender Form ausgegeben

<Tokenname, Attributwert>

Tokenname:	Abstraktes Symbol
Attributwert:	Zeigt auf den Eintrag in die Symboltabelle

- Die Einträge in der Symboltabelle werden für die semantische Analyse und die Codegenerierung verwendet.

Lexikalische Analyse - Beispiel

- Im Quellprogramm finden Sie folgenden Code:

  position = initial + rate * 60

- Aufbau der Lexeme:

  position → <id,1> (id ist ein abstraktes Symbol für Bezeichner)

  = → <=> (wird auf das Token "=" abgebildet und kein Attributwert)

  initial → <id,2>

  + → <+>

  rate → <id,3>

  * → <*>

  60 → <number,4>

1	position	….
2	initial	….
3	rate	….
4	60	….

Symboltabelle

- Es wird folgender Strom von Tokens erzeugt:

  <id,1> <=> <id,2> <+> <id,3> <*> <number,4>

Beispiel ANTLR
Token-Generierung

- Grammatik

```
grammar Hello;
r : 'hello' ID;
ID : [a-z]*;
WS : [ \t\r\n]+ -> skip;
```

Eingabe: hello part

Token:

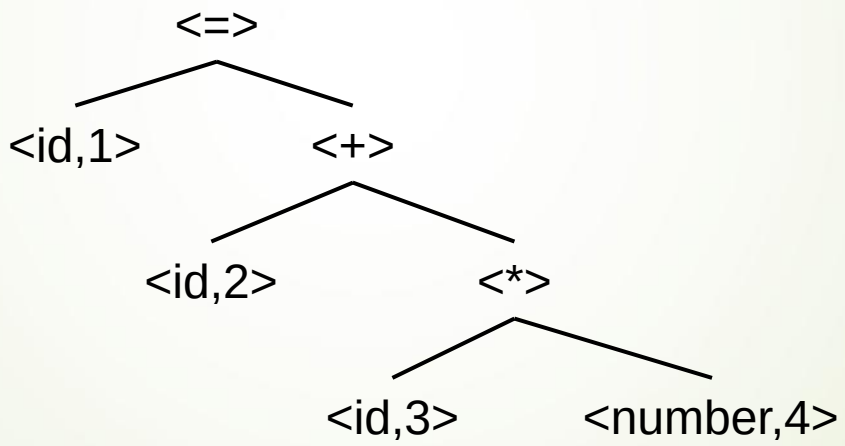
```
[@0,0:4='hello',<'hello'>,1:0]
[@1,6:9='part',<ID>,1:6]
[@2,12:11='<EOF>',<EOF>,2:0]
```

Syntaxanalyse

- Die zweiten Phase der Analyse ist die Syntaxanalyse.
- Dies nennt man auch Parsing.
- Aus dem Tokenstrom wird eine baumartige Zwischenstruktur generiert, welche die Grammatik zeigt.
- Ein typischer Baum dazu ist der Syntaxbaum
  - Innere Knoten sind Operationen
  - Kinderknoten sind die Argumente
- Der Baum zeigt die Reihenfolge, in der die Operationen der Zuweisungen durchgeführt werden.

Syntaxanalyse - Beispiel

- Beispiel:
  - position = initial + rate * 60 erzeugt den Tokenstrom
  - <id,1> <=> <id,2> <+> <id,3> <*> <number,4>

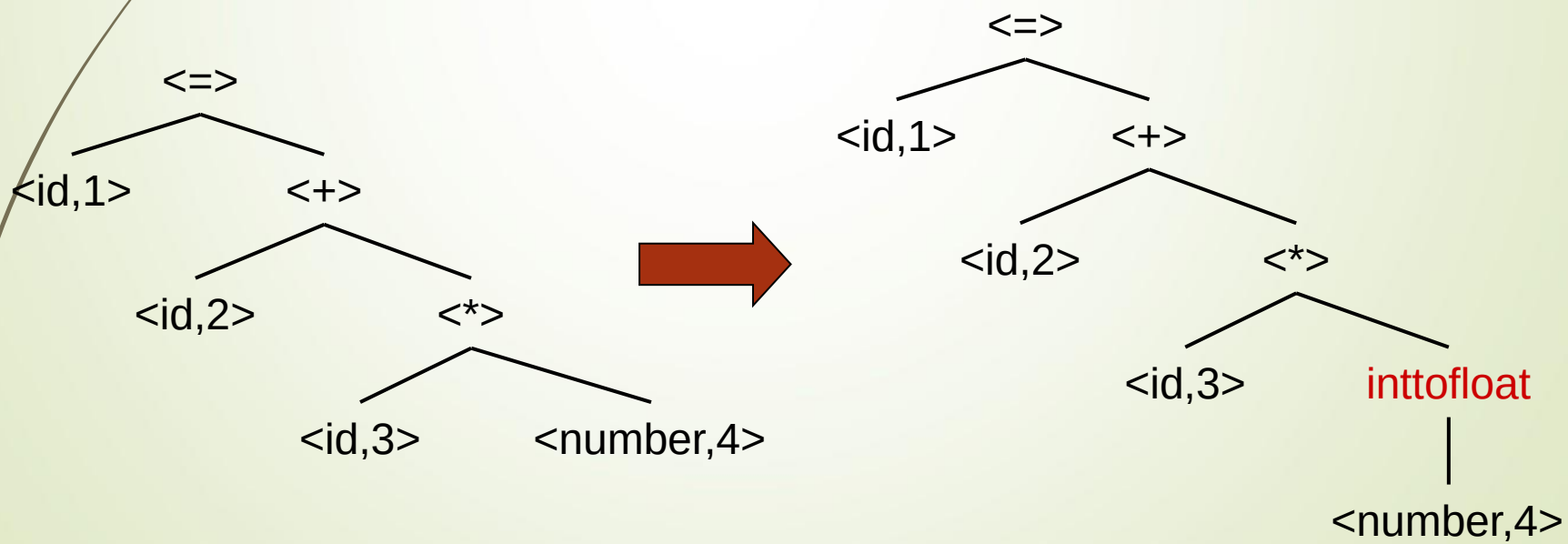


Semantische Analyse

- Die Semantische Analyse nimmt:
  - Syntaxbaum und
  - die Informationen in der Symboltabelle
- Die semantische Konsistenz mit der Sprachdefinition wird analysiert.
- Typinformationen werden gesammelt und abgespeichert
  - im Syntaxbaum oder
  - direkt in der Symboltabelle
- Wichtige Aufgabe dieser Phase:
  - Typüberprüfung
  - Typkonvertierung

Semantische Analyse - Beispiel

- Beispiel:
  - rate in <id,3> sei ein Fließkommazahl
  - 60 in <number,4> ist eine Integerzahl
  - D.h. 60 muss in eine Fließkommazahl konvertiert werden, bevor die Operation <*> ausgeführt werden kann.

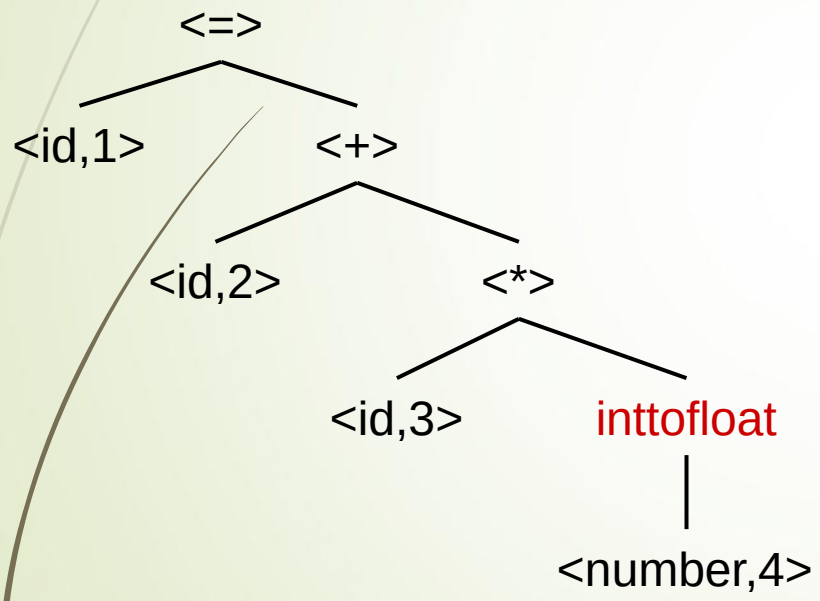


Zwischencodegenerator

- Viele Compiler erzeugen ausdrücklich eine Zwischendarstellung niedriger Ebene (maschinennahe).
- Wichtige Eigenschaft dieser Zwischendarstellung
  - Sie lässt sich einfach erstellen und
  - unkompliziert in die Zielmaschine übersetzen.
- Eine wichtige Zwischendarstellung ist der aus assembler- ähnlichen Befehlen aufgebaute *Drei-Adress-Code*
- Eigenschaften des 3-Adress-Codes:
  - Höchstens ein Operator auf der rechten Seite.
  - Compiler muss temporäre Namen anlegen, um Zwischenergebnisse speichern zu können
  - Einige Befehle haben auch weniger als 3 Operanten

Zwischencodegenerator - Beispiel

- Beispiel



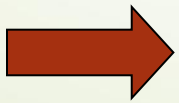
3-Adress-Code

t_1 = inttofloat(60)
t_2 = id3 * t_1
t_3 = id2 + t_2
id1 = t_3

Codeoptimierung

- In der maschinenunabhängigen Codeoptimierung wird versucht den Zwischencode zu verbessern.
- Besser kann heißen:
  - schneller oder
  - kürzer oder
  - weniger Ressourcen-Verbrauch
  - ....
- Beispiel:
  - Compiler ersetzt 60 direkt in die Fließkommazahl 60.0 um und spart sich dadurch die Konvertierung
  - Entfernen unnötiger Zwischenergebnisse

t_1 = inttofloat(60)
t_2 = id3 * t_1
t_3 = id2 + t_2
id1 = t_3

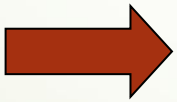


t_1 = id3 * 60.0
id1 = id2 + t_1

Codeerzeugung

- Der Codegenerator nimmt die Zwischendarstellung und bildet auf die Zielsprache ab.
- Wichtige Aufgaben
  - Verwalten und Zuordnen der Register zu den Variablen.
- Beispiel für eine fiktive Maschine

t_1 = id3 * 60.0
id1 = id2 + t_1



```
LDF     R2, id3          (R2 ← id3)
MULF    R2, R2, #60.0    (R2 ← R2*60.0)
LDF     R1, id2          (R1 ← id2)
ADDF    R1, R1, R2       (R1 ← R1+R2)
STF     id1, R1          (id1 ← R1)
```

Entwicklung der Programmiersprachen I

- Die ersten elektronischen Rechner in den 1940er Jahren
  - Programmiert in 0/1
  - Aufgaben
    - Daten verschieben,
    - Daten addieren/multiplizieren
    - Daten vergleichen
  - Programme waren speziell für die Maschine geschrieben und schwer zu verstehen
- Erste Entwicklung
  - Mnemonischer Assembler in den 1950 Jahren mit Makrobefehlen
- Erste Programmiersprachen Mitte 1950
  - Fortran
  - Cobol
  - Lisp
- In den folgenden Jahrzehnten entwickelten sich viele weitere Programmiersprachen

Entwicklung der Programmiersprachen II

- Einteilung der Programmiersprachen
  - 1. Generation (Maschinensprachen)
  - 2. Generation (Assemblersprachen)
  - 3. Generation (höhere Programmiersprachen wie Fortran, C, C++, C#, …)
  - 4. Generation (Sprachen für besondere Anwendungen wie SQL, PostScript,…)
  - 5. Generation (Logik und bedienungsorientierte Sprachen wie Prolog, …)
- Ein andere Einteilung
  - Imperative Sprachen (Wie ein Berechnung durchgeführt wird)
    - (C, Fortran, C++, Java, C#, ….)
  - Deklarative Sprache (Was für eine Berechnung durchgeführt wird)
    - (Haskel, Prolog)

Methodik des Compilerbaus

- Compilerbau zeigt beispielhaft wie man ein komplexes Problem mit Hilfe geeigneter mathematischer Modelle beschreiben kann und dann zu einer handbaren Lösung kommt.
- Auswahl der richtigen Methoden und Algorithmen
  - endliche Automaten, reguläre Ausdrücke (Lexer)
  - kontextfreie Grammatiken, Kellerautomaten (Parser)
  - Bäume, und Traversierung von Bäume (Syntaxgerichtete Übersetzung)
- Codeoptimierung
  - Graphentheorie, Syntaxbäume,....

Tools für die Vorlesung Compiler

- Für den Lexer und Parser nutzen wir das Tool ANTLR und teilweise FLACI:
  - https://www.antlr.org/
  - http://lab.antlr.org/
  - https://flaci.com/home/

ANTLR

ANTLR

Download v4.12.0 | Dev Tools | Book | Doc | About ANTLR | Support | Bugs

What is ANTLR?

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.

Terence Parr is a tech lead at Google and until 2022 was a professor of data science / computer science at Univ. of San Francisco. He is the maniac behind ANTLR and has been working on language tools since 1989.

Check out Terence impersonating a machine learning droid: explained.ai

Quick Start

To try ANTLR immediately, jump to the *new* ANTLR Lab!

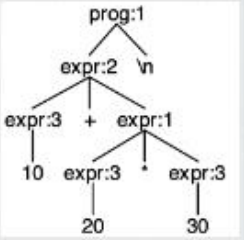
To install locally, use antlr4-tools, which installs Java and ANTLR if needed and creates `antlr4` and `antlr4-parse` executables:

```
$ pip install antlr4-tools
```

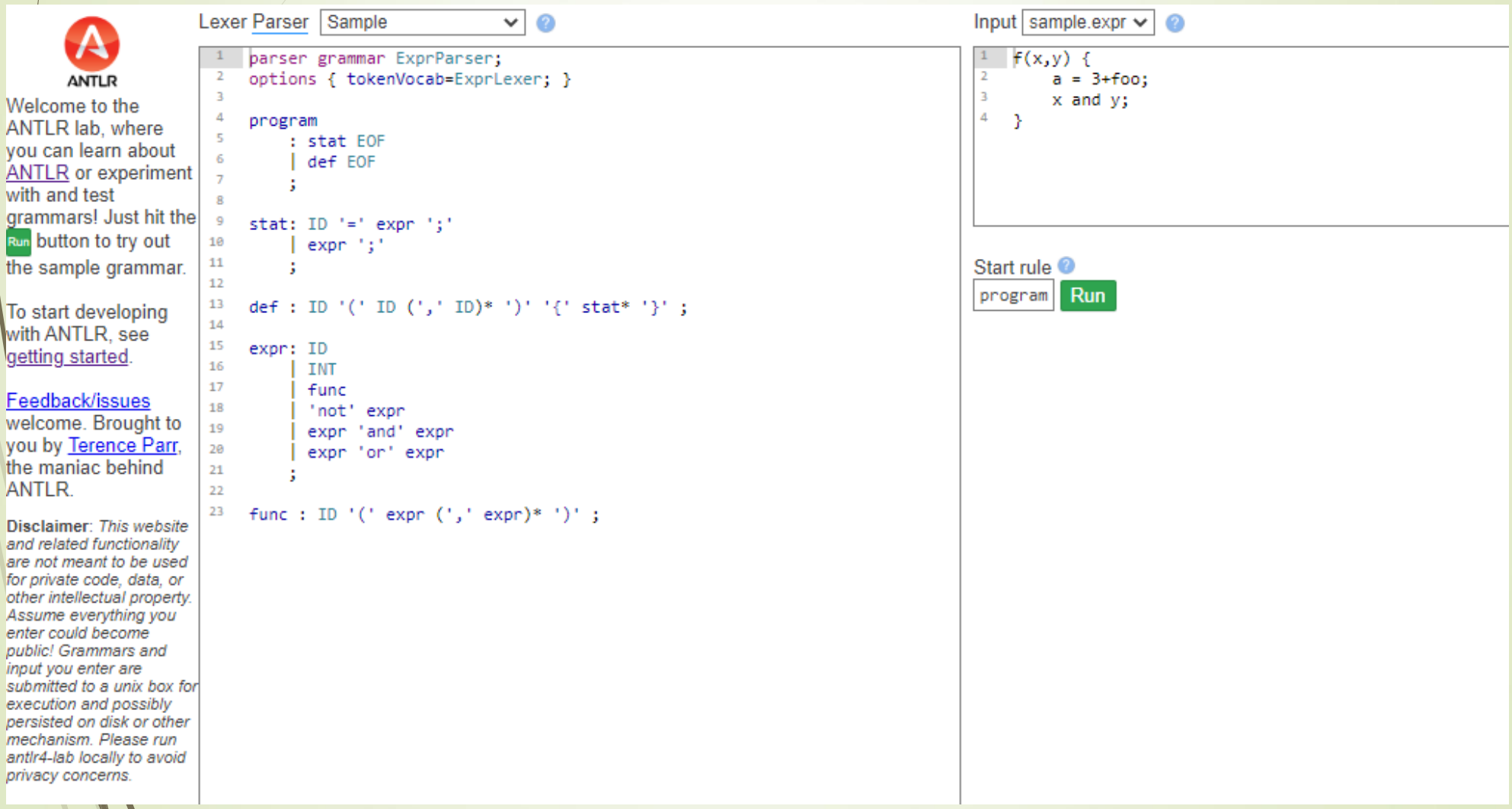
(Windows must add `..\LocalCache\local-packages\Python310\Scripts` to the PATH). See the Getting Started doc. Paste the following grammar into file `Expr.g4` and, from that directory, run the `antlr4-parse` command. Hit control-D on Unix (or control-Z on Windows) to indicate end-of-input. A window showing the parse tree will appear.

```
grammar Expr;
prog:   (expr NEWLINE)* ;
expr:   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   INT
    |   '(' expr ')'
    ;
NEWLINE : [\r\n]+ ;
INT     : [0-9]+ ;
```

```
$ antlr4-parse Expr.g4 prog -gui
10+20*30
^D
$ antlr4 Expr.g4  # gen code
$ ls ExprParser.java
ExprParser.java
```

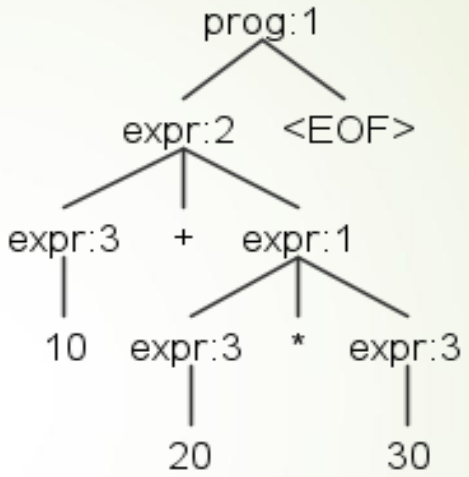


ANTLR-Lab


ANTLR
Welcome to the ANTLR lab, where you can learn about ANTLR or experiment with and test grammars! Just hit the Run button to try out the sample grammar.
To start developing with ANTLR, see getting started.
Feedback/issues welcome. Brought to you by Terence Parr, the maniac behind ANTLR.
Disclaimer: This website and related functionality are not meant to be used for private code, data, or other intellectual property. Assume everything you enter could become public! Grammars and input you enter are submitted to a unix box for execution and possibly persisted on disk or other mechanism. Please run antlr4-lab locally to avoid privacy concerns.
Lexer Parser Sample
parser grammar ExprParser;
options { tokenVocab=ExprLexer; }
program
: stat EOF
| def EOF
;
stat: ID '=' expr ';'
| expr ';'
;
def : ID '(' ID (',' ID)* ')' '{' stat* '}' ;
expr: ID
| INT
| func
| 'not' expr
| expr 'and' expr
| expr 'or' expr
;
func : ID '(' expr (',' expr)* ')' ;
Input sample.expr
f(x,y) {
a = 3+foo;
x and y;
}
Start rule
program
Run

Beispiel aus der Vorlage

```
grammar Expr;
prog:   (expr NEWLINE)* ;
expr:   expr ('*' | '/') expr
    |   expr ('+' | '-') expr
    |   INT
    |   '(' expr ')'
    ;
NEWLINE : [\r\n]+ ;
INT     : [0-9]+ ;
```



antlr4-parse Expr.g4 prog –gui
10+20*30
^D (unter Linux) bzw. ^Z (unter Windows)

FLACI


FLACI
Formale Sprachen, abstrakte Automaten, Compiler und Interpreter
Formale Sprachen
Ein Alphabet A ist eine endliche, nichtleere Menge von Zeichen.
Wählen Sie eines der Beispiel-Alphabete zum Experimentieren aus.
A1 = { 0, 1 }
A2 = { a, b, c, ..., z }
A3 = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
A4 = { , , , , , }
A5 = { begin , end , for , while , do , repeat , until }
Interaktive Begriffserklärungen zu Sprache, Wort und Alphabet.
Reguläre Ausdrücke
Der einfachste reguläre Ausdruck ist ein einzelnes Zeichen a. Er beschreibt die Sprache L = {a} . Reguläre Ausdrücke lassen sich verketten: a gefolgt von b, gefolgt von c wird kurz abc geschrieben.
Was passiert, wenn man den regulären Ausdruck zu ei oder ein verändert?
Regulärer Ausdruck
e
Suchtext
Das ist ein Beispieltext, der einige Wörter enthält.
Syntax-Diagramm
e
Interaktive Experimentierumgebung für reguläre Ausdrücke.

Grundbegriffe

- Alphabet und Zeichen
- Worte, Wortlänge und Verkettung
- Wortmenge
- Die Sprache

Alphabet und Zeichen

- Ein Alphabet ist eine beliebige endliche, nichtleere Menge. Die Elemente dieser Menge heißt Zeichen.
- Beispiel:
  - A_1 = {a,b,c,d,…,z}
  - A_2 = {(,),[,],+,+,*,/,a}
  - A_3 = {begin, end, for, while, do ,repeat, until}

Worte, Wortlänge und Verkettung

- Irgendeine (auch leere) Zeichenmenge ist ein Wort. Man sagt:
  - eine Zeichenkette „w“ ist ein Wort über dem Alphabet Σ, wenn sämtliche Zeichen von w aus Σ stammen.
- Das Symbol für das leere Wort ist ε.
- Die Länge eines Wortes w, kurz |w| ist bestimmt durch die Anzahl der aller Zeichen, die das Wort w enthält.
- Die Verkettung • von zwei Zeichen ergibt ein Wort.
  - Σ ={a,b,c} dann w=a•b = ab ist ein Wort.
  - Verkettung zwei Worte w_1 = ab mit w_2 = bc : w3 = w_1 • w_2 = abbc

Wortmenge

Σ und Σ*

- Die Menge aller Wörter über Σ nennt man die Wortmenge Σ*. Das leere Wort ε gehört auch dazu.
- Das Eingabealphabet Σ

  Σ = $\{e_1,..,e_n\}$ eine nicht leere Menge von Zeichen

  z.B. {0,1} oder {a,b,c}
- Worte
  - Endliche Zeichenfolge die aus dem Eingabealphabet gebildet werden können.

  z.B w = 0101101101 oder w = abbabccbacbb
- Σ* : = die Menge aller Wörter, die über das Alphabet gebildet werden können.

Wortmenge

Formale Definition von Σ^*

Formale Definition von Σ^* (rekursiv definiert)

1. Das leere Wort ε gehört zu Σ^*, d.h. $\varepsilon \in \Sigma^*$
2. Jeder Buchstabe $e \in \Sigma$ ist in Σ^*, d.h. $e \in \Sigma^*$
3. Sei $v,w \in \Sigma^*$ dann ist auch $vw \in \Sigma^*$

   (Konkatenierung von v mit w)

- Beispiel:

  $\Sigma = \{a,b\}$ dann ist

  $\Sigma^* = \{\varepsilon, a , b, aa, ab, ba, bb, aaa, \ldots\ldots\}$

Die Sprache L(A)

- Sei Σ ein Alphabet. Jede Teilmenge $L \subseteq \Sigma^*$ heißt Sprache über Σ.
- Eine Sprache besteht aus Wörter
  - $L_1 = \varnothing$ oder $L_2 = \Sigma^*$ sind Sprachen
- Sprachen können endlich aber unendliche viele Worte enthalten