ICBW

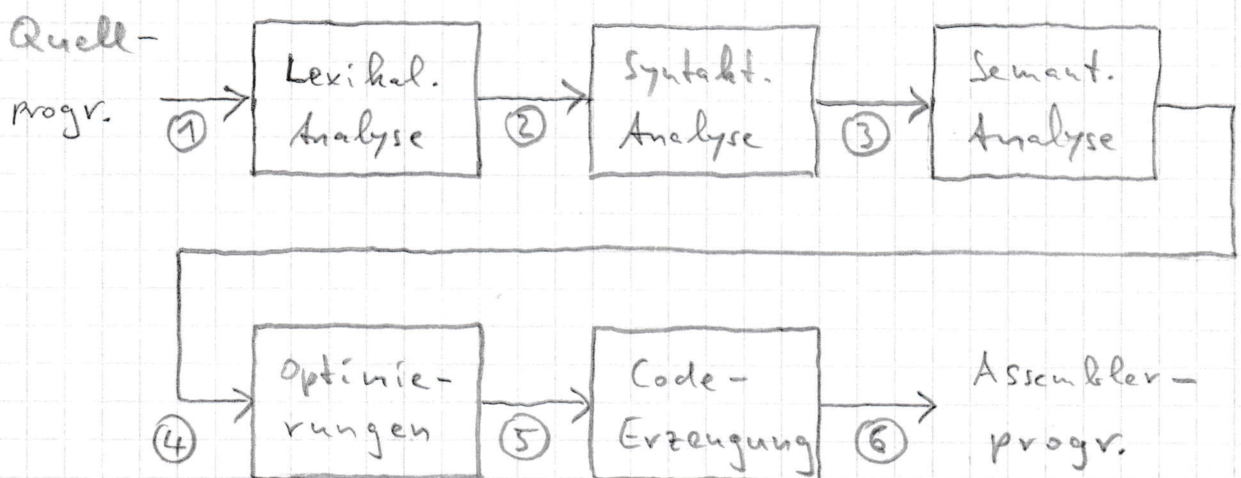
Implementierung von Compilerbau-Werkzeugen

0) Skripte der Vorlesungen "Compilerbau"
1) D. Grune, C.J.H. Jacobs: Parsing Techniques, Springer 2008
2) "Drachenbuch" (Aho, Sethi, Ullman: Compilers, Addison-Wesley 2013)

1. Grundlagen

1.1. Compiler

Compiler transformieren Programme aus einer Quellsprache in eine Zielsprache, meist in vielen Stufen ("Phasen"), ohne ihre Semantik ("Bedeutung": was wird berechnet) zu ändern. Sie gliedern sich ganz grob in zwei Teile, die Analyse- und die Synthesephase:



1: Characters
2: Tokens (Symbole)
3: Abstr. Syntax (Datenstruktur)
4: Abstr. Syntax
5: Zwischencode (Abstr. Maschineninstrukt.)
6: characters

Bem.: Wenn in der Synthesephase nicht Instruktionen erzeugt werden, die (später) eine Berechnung ausführen, sondern diese Berechnungen schon in der Synthesephase durchgeführt werden, nennt man das Programm nicht "Compiler", sondern "Interpreter".

Bsp.: demo.c, ein "Taschenrechner" mit Operatorprioritäten, Klammern, und einem Speicher mit 26 Speicherzellen; siehe Seiten DØ – D5. Erläuterungen:

a) Die lexikal. Analyse wird vom sog. "Scanner" durchgeführt, der hier durch die Funktion nextToken() implementiert wird. Diese Funktion liest solange Zeichen von der Eingabe, bis das nächste Token vollständig erkannt ist; dann wird die Tokenklasse (END_OF_INPUT, ..., STORE) als Integer an den Aufrufer zurückgegeben. Bei manchen Token muß auch noch ein "semantischer Wert" übermittelt werden (bei NUMBER z.B. der Zahlenwert); dafür gibt's die Variable "tokenValue". Wozu dient die Komponente "index" in tokenValue? Was bedeutet "ein Zeichen Vorausschau"?

b) Die syntakt. Analyse wird in Form des "rekursiven Abstiegs" durchgeführt: 6 wechselseitig rekursive Funktionen, die jeweils ein best. syntakt. Konstrukt erkennen können. Sie schauen sich dazu das nächste Token an, das sie durch Aufruf von nextToken() erhalten. Wenn ein Konstrukt erkannt wurde, wird sofort die zugehörige Aktion ausgeführt. Was passiert z.B. bei parseAssign(), bei parseTerm() und bei parsePrimary()? Was genau geben also die Parsing-Routinen zurück? (Wir werden die Konstruktion des Parsers noch genauer

besprechen, insbesondere wie dadurch die Prioritäten und Assoziativitäten der Operatoren abgebildet werden.)

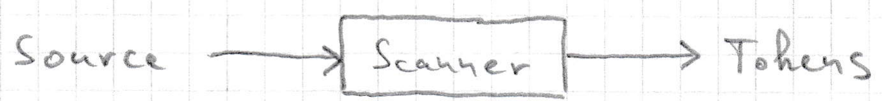
c) Der Treiber des Interpreters, die Funktion main(), ist selbsterklärend.

Bem.: In unserem Demo-Programm sind also alle Aktionen direkt in den Parser eingebettet, ohne daß eine Abbildung der eingegebenen Ausdrücke in Datenstrukturen ("Abstrakte Syntax") stattfindet. Das macht man in realen Compilern aus zwei Gründen anders:

- Die Aktionen müssen ohne Datenstruktur in exakt der Reihenfolge stattfinden, in der das Parsen abläuft.
- Das "Hineinquetschen" der großen Phasen "Semant. Analyse", "Optimierung" und "Codeerzeugung" in die syntakt. Analyse macht einen Compiler unwartbar.

1.2. Transformatoren und Generatoren, Bootstrap

Wir betrachten z.B. die Lexikal. Analyse, den "Scanner":



Der Scanner transformiert einen Strom von Zeichen in einen Strom von Tokens. Wenn die Transformation genügend starren Regeln folgt, kann der Transformationsmechanismus (hier der Scanner, also selbst ein Programmfragment) aus diesen Regeln abgeleitet werden. Programme, die solche Ableitungen vornehmen, heißen "Generatoren". Im Compilerbau sind insbesondere Scanner-Generatoren, Parser-Generatoren und Codegenerator-Generatoren in Gebrauch.

Ein Generator ist also selber ein kleiner Compiler: er übersetzt Regeln (für den Transformationsprozeß) in ein Programm, das diese Regeln in einem konkreten Compiler auf zu übersetzende Programme anwendet.
Wie bei jedem Compiler haben wir es auch bei einem Generator mit drei Sprachen zu tun: die Eingabesprache, die Ausgabesprache und die Implementierungssprache.
Es ist von Vorteil, wenn bei einem Compiler die Eingabe- und die Implementierungssprache die gleichen Sprachen sind: dann kann der Compiler in seiner "eigenen Sprache" (die, die er übersetzt) geschrieben werden. Vorteil: Wenn die Eingabesprache erweitert wird oder eine bessere Übersetzungsmethode angewendet werden soll, kommt das dem Compiler selbst zugute. Dieser "Bootstrap" geschieht in zwei Schritten: Zuerst wird die Änderung in den bestehenden Compiler eingebaut, mit den schon vorhandenen Sprachmitteln. Im zweiten Schritt benutzt man die neuen Features im Compiler selber. Nachteil: Der allererste Bootstrap benötigt immer noch eine Implementierung des Compilers in einer Hilfssprache.
Sowohl der Scanner- als auch der Parsergenerator besitzen einen Scanner sowie einen Parser für ihre jeweiligen Eingabesprachen. Also kann man einen Bootstrap in jedem der beiden Tools getrennt voneinander vornehmen (der Scannergenerator erzeugt seinen eigenen Scanner, der Parsergenerator seinen eigenen Parser).
Das geht aber auch "über Kreuz": der Scannergenerator

erzeugt den Scanner des Parsergenerators und der Parsergenerator den Parser des Scannergenerators. Damit eliminiert man die Abhängigkeit von handgeschriebenen (oder mit anderen Tools erzeugten) Scannern und Parsern in den beiden Generatoren vollständig.

1.3. Formale Sprachen und Grammatiken

Was ist eine "Sprache"? Gegeben sei eine endliche Menge von Zeichen (auch Symbole oder Buchstaben genannt), meist mit T bezeichnet. Bsp.: $T = \{0, 1\}$, das "Binäralphabet".
Dann bezeichnet $T*$ die Menge aller Zeichenketten über T, einschließlich der leeren Zeichenkette (meist ε genannt).
Eine Sprache L ist nun einfach eine Teilmenge von $T*$.
Im Allgemeinen ist $|L| = \infty$, obwohl $|T| < \infty$. Deswegen wird man i.A. ein anderes Mittel als die Aufzählung benötigen, um die zu L gehörenden Zeichenketten zu spezifizieren.
Eine "Grammatik" ist genau dazu da: sie "erzeugt" exakt alle zu L gehörenden Strings (und keinen, der nicht zu L gehört).
Eine Grammatik ist ein 4-Tupel (N, T, P, S), wobei

- T: Menge der Buchstaben ("Terminalsymbole"), $|T| < \infty$
- N: Menge der Nichtterminalsymbole ("syntaktische Variablen") mit $N \cap T = \{\}$ und $|N| < \infty$
- P: Menge der Produktionen, $|P| < \infty$. Jede Produktion $p \in P$ beschreibt eine Abbildung von Strings aus $(T \cup N)*$:

$P: (T \cup N)^* N (T \cup N)^* \to (T \cup N)^*$

Beachte: Links steht mind. ein Nichtterminalsymbol!

S: Ein ausgezeichnetes Element aus N, das "Startsymbol" (auch "Satzsymbol" genannt)

<u>Bsp.:</u> $N = \{S, A, B\}$

$T = \{a, b\}$

$P = \{ S \to AB$

$S \to \varepsilon$

$A \to aS$

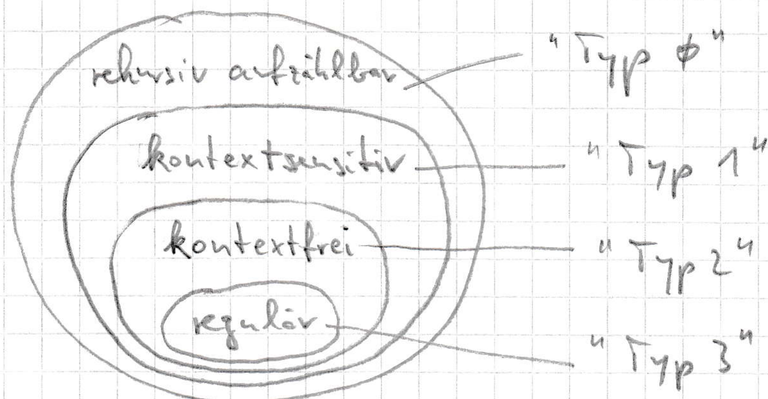
$B \to b \}$

Diese Grammatik definiert die Sprache $L = \{a^n b^n \mid n \in \mathbb{N}_0\}$.

Verschiedene Grammatiken können die gleiche Sprache beschreiben:

$P = \{S \to aSb, S \to \varepsilon\}$ erzeugt die gleiche Sprache L.

Es gibt (mindestens) vier Klassen von Sprachen, die eine echte Hierarchie bilden ("Chomsky-Hierarchie"):



Die entspr. Grammatiken unterscheiden sich nur darin, welche Produktionen erlaubt sind ($\alpha, \beta, \gamma \in (T \cup N)^*$):

Typ 0: $\alpha \to \beta$ (α enthält mind. ein Nichtterminal)

Typ 1: $\alpha A \beta \to \alpha \gamma \beta$ ($A \in N$, $\gamma \neq \varepsilon$)

Typ 2 : $A \to \gamma$

Typ 3 : $A \to a$, $A \to aB$ } "rechtsregulär"

alternativ:

$A \to a$, $A \to Ba$ } "linksregulär"

1.4. Automaten

Automaten dienen zum Erkennen, ob ein gegebener String zu einer gegeb. Sprache gehört. Die verschiedenen Klassen von Grammatiken benötigen verschieden mächtige Automaten:

Typ 0 : Turing-Maschine

Typ 1 : Turing-Maschine mit linear begrenztem Band

Typ 2 : Nichtdeterministischer Kellerautomat

Typ 3 : Endlicher Automat (deterministisch oder nichtdeterministisch ist gleichgültig, da die Automatenklassen gleich mächtig sind)

Im Compilerbau werden überwiegend Typ-3- und Typ-2-Grammatiken benutzt(*), mit der Einschränkung, daß anstelle des nichtdeterministischen Kellerautomaten ein deterministischer eingesetzt wird und damit nur eine Teilmenge der kontextfreien Sprachen erkannt werden kann – Grund: besseres Laufzeitverhalten ($O(n)$ vs. $O(n^3)$).

(*) (zur Spezifikation v. Scannern bzw. Parsern)

2. Aufgaben im Kurs

Scannergenerator	Parsergenerator
Def. d. Spezifikationssprache	Def. d. Spezifikationssprache
Scanner	Scanner
Parser	Parser
Abstr. Syntax	Abstr. Syntax
Semantik-Prüfungen	Semantik-Prüfungen
NEA	FIRST-Mengen, NEA
DEA	LR(0)-DEA
Minimierung	LALR(1)-DEA
Tabellenausgabe	Tabellenausgabe
Treiber	Treiber
1. Bootstrap: Scanner	1. Bootstrap: Parser
2. Bootstrap: Parser	2. Bootstrap: Scanner

Vorschlag: Zweiergruppen, jeweils ein Verantwortlicher für Scanner- und Parsergenerator

Bearbeitungszeit: ca. 1 Woche / Teilaufgabe