Fragenkatalog 1810 – Übersetzerbau

(2008 by Jörg Jacob)
Version 1
studium (at) snuablo (dot) com

Ich habe versucht, möglichst vollständig und fehlerfrei zu schreiben. Natürlich kann ich nicht ausschließlich, dass sich doch der ein oder andere Fehler eingeschlichen hat. Ich übernehme **keinerlei Haftung**! Dieses Dokument ist zur unterstützenden Prüfungsvorbereitung gedacht und NICHT als Ersatz zum Kurstext.

Top Themen:
Allgemeines
Lexikalische Analyse
Syntaxanalyse

Add-Ons (sehr kurz und teilweise unvollständig):
Semantische Analyse
Codeerzeugung

Aus eigener Erfahrung kann ich sagen, dass es immer gut ist, von sich aus zu erzählen – soviel wie möglich, was zur Frage passt. Der Prüfer unterbricht schon, wenn es ihm ausreicht. Nur einfach die Antworten runter rattern und „nächste Frage bitte“ ist nicht so gut. Zudem können dann auch mehr oder unangenehmere Fragen folgen. Wer durch Wissen glänzt wird (normalerweise) positiv bewertet und verbraucht gleichzeitig Prüfungszeit ;)...

Der hiermit vorgelegte Fragenkatalog zu Übersetzerbau wurde von mir zusammengestellt aus den von der Fachschaft herausgegebenen Prüfungsprotokollen. Zudem habe ich mir die ein oder andere Frage auch selbst ausgedacht bzw. einen Teil meiner eigenen Prüfungserfahrung einfließen lassen.

VIEL ERFOLG!

Allgemeines

1.) Was macht ein Übersetzer?
Überführung von Sprache A zu Sprache B. Es kann sinnvoller/einfacher sein, etwas in Sprache A auszudrücken. Die Maschine/Anwendung versteht aber nur Sprache B.

2.) Wofür werden Übersetzer eingesetzt?
Klassischerweise zur Übersetzung einer höheren Programmiersprache in Assembler/ Maschinensprache. Darüber hinaus Seiten- und Dokumentenbeschreibungssprachen, Datenbankabfragesprachen, Entwurfssprachen (VLSI).
Eigentlich überall dort, wo syntaktische Strukturen erkannt werden.

3.) Welche Sprachen eignen sich als Quellsprache?
Sprachen, die eine kontextfreie Grammatik haben, d. h., bei denen eine Produktion keine Abhängigkeiten auf der linken Seite hat.
Definition: Eine kontextfreie Grammatik ist ein Quadrupel $G = (N, \Sigma, P, S)$, wobei gilt:

(i) N ist ein Alphabet von *Nichtterminalen*.
(ii) Σ ist ein Alphabet von *Terminalen* $N \cap \Sigma = \{\}$ (N und Σ disjunkt).
(iii) $P \subseteq N \times (N \cup \Sigma)^*$ ist eine Menge von *Produktionsregeln*.
(iv) $S \in N$ ist das *Startsymbol*.

4.) Kann man auch natürliche Sprachen, wie Deutsch oder Japanisch, übersetzen?
Nein, da diese Sprachen keine kontextfreie Grammatik haben.

5.) Unterschied zwischen Compiler und Interpreter?

Compiler:	- übersetzt gesamte Eingabe - Ausführung erst zu späterem Zeitpunkt - schnelle Ausführungsgeschwindigkeit aber längere Programmentwicklung
Interpreter:	- übersetzt kleine Teile der Eingabe (z. B. Zeilen) - sofortige Ausführung - schnelle Programmentwicklung, aber langsamere Ausführungsgeschwindigkeit
Mischform:	Compiler übersetzt Programm in einfachen Zwischencode, der zur Laufzeit effizient von einem Interpreter verarbeitet werden kann (z. B. Java).

6.) Welche Phasen existieren bei der Übersetzung?

Grob:	Analyse und Synthese
Analyse:	lexikalische Analyse, syntaktische Analyse, semantische Analyse
Synthese:	Zwischencode-Erzeugung, Optimierung des Zwischencodes, Erzeugung von Maschinencode
Global:	Fehlerbehandlung, Verwaltung der Symboltabelle

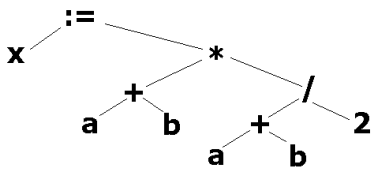
(Phasen genau auf nächster Seite beschrieben)

Phase	Eingabe	Ausgabe	Arbeitsschritte/Anmerkungen
lexikalische Analyse (Scanner)	Folge von Zeichen (Buchstaben, Ziffern, Sonderzeichen)	Folge von Token	- Erkennung gewisser Grundsymbole (Wortsymbole, wie `begin`, `if`, `while`, Variablennamen, numerische Konstanten etc.) - Beschreibung durch reguläre Ausdrücke
syntaktische Analyse (Parser)	Tokenfolge	Syntaxbaum	- Erkennung von hierarchischen Strukturen - Beschreibung in regulären Ausdrücken nicht möglich - aber mit kontextfreien Grammatiken - Beschreibung größerer Einheiten, wie arithmetische Ausdrücke, bedingte Anweisungen, Schleifen usw.
semantische Analyse	Syntaxbaum	attributierter Syntaxbaum	- Sammlung weiterer Informationen, die in der Syntaxanalyse nicht erfasste Korrektheitsaspekte betreffen oder für die Codeerzeugung notwendig sind - Typüberprüfung (type checking) - Auflösung überladener oder polymorpher Operationen (z. B. Addition bei int oder float) - type casting (float f = 1)
Erzeugung des Zwischencode	Syntaxbaum	Zwischencode	- Komplexität beherrschbar machen (große Lücke zwischen höherer Programmiersprache und Maschinensprache) - also eine Zwischenebene - abstrakte Maschinensprache - z. B. 3-Adress-Code (3AC)
Optimierung des Codes	Zwischencode	optimierter Zwischencode	- Ineffizienzen durch ungeschickten Zwischencode entfernen - komplexe Analysen des erzeugten Zwischencodes, z. B. durch Erkennen der Schleifenstruktur und Variablen innerer Schleifen in Register
Codeerzeugung	(optimierter) Zwischencode	Assembler-/ Maschinencode	- speziell für Zielmaschine - Speicherorganisation für Zielprogramm - Abbildung der Operationen des Zwischencodes auf die bestmögliche Befehlsfolge der Zielmaschine - gute Zuteilung von Registern

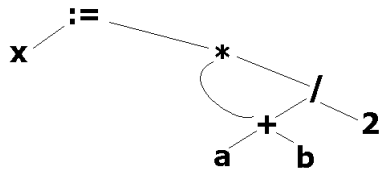
6.) Was ist eine Zwischensprache?
Eine *abstrakte Maschinensprache* von etwas höherem Niveau (im Vergleich zur tatsächlichen Maschinensprache).

7.) Welche Zwischensprachen kennen Sie?
- (abstrakte) Syntaxbäume
- gerichtete azyklische Graphen (DAG = directed acyclic graph)
- Postfix Notation
- 3-Adress-Code (3AC)

8.) Was sind (abstrakte) Syntaxbäume genau?
Ein Syntaxbaum ist ein Synonym zu Ableitungsbaum (konkrete Syntaxbäume). Abstrakte Syntaxbäume sind eine etwas vereinfachte Darstellung konkreter Syntaxbäume, bei denen die Struktur weniger an grammatischen Kategorien (Nichtterminalen) als an der Bedeutung des Konstrukts, d. h. den durchzuführenden Operationen orientiert ist.
Beispiel: Zuweisung x := (a + b) * (a + b) / 2



9.) Was sind DAGs?
Diese sind eng verwandt mit den Syntaxbäumen, unterscheiden sich aber darin, dass im Syntaxbaum mehrfach auftretende Teilstrukturen nur einmal vorkommen. Ein DAG für die oben aufgeführte Zuweisung würde wie folgt aussehen:



10.) Wie wird die Postfix Notation verarbeitet?
Operanden werden auf den Stack gepackt, sobald ein Operator in der Eingabe erscheint, werden die notwendigen Operanden vom Stack genommen, die Operation ausgeführt und das Ergebnis wieder auf den Stack gelegt. Die Zuweisung x := (a+b) * (a + b) / 2 würde so aussehen:
x a b + a b + 2 / * :=
Die Postfix Notation lässt sich zu einer Zwischensprache für eine abstrakte *Stack-Maschine* ausbauen.

11.) Was ist 3-Adress-Code (3AC)?
3AC ist eine einfache Zwischensprache mit Befehlen mit maximal drei Argumenten, z. B. binäre Operationen, Zuweisungen, Arrayzugriffe, bedingte und unbedingte Sprünge etc. Befehlsfolgen lassen sich leichter umordnen als bei Postfix Notation. Zudem können auch nach der Erzeugung des Codes bestimmte Variablen Maschinenregistern zugeordnet werden (Einsparung von Speicherzugriffen).
3AC ist abstrakt genug, um für eine Optimierung geeignet zu sein, aber andererseits so einfach, dass 3AC-Befehle durch einen oder wenige Maschinencode Befehle ersetzt werden können.

12.) Wie sehen 3AC Befehle aus?
3AC Befehle haben bis zu drei Argumente.

13.) Welche 3AC Befehle kennen Sie?
(hier ist die vollständige Liste zu können!)

x := y op z	Hier ist *op* ein binärer Operator, z. B. +, *, and usw. Diese Klasse enthält für jeden Operator einen Befehl. Die Werte von y und z werden mit *op* verknüpft und der Variablen z zugewiesen.
x := op y	Analog, nur ist *op* ein unärer Operator (z. B. not).
x := y	einfache Zuweisung
goto L	Sprung zum 3AC Befehl mit Sprungmarke L
if x cop y goto L	Hier ist *cop* ein Vergleichsoperator, z. B. =, <, > usw. Bedingter Sprung nach L, falls der Vergleich von x und y mit *cop true* ergibt. Auch hier gibt es einen Befehl für jeden Operator *cop*.
x := y[i] x[i] := y	Indizierte Zuweisung zur Übersetzung von Array-Zugriffen. Der erste Befehl weist x den Wert der Speicherstelle zu, die i Speicherzellen hinter y liegt. Der zweite weist der Speicherstelle, die i Zellen hinter x liegt, den Wert von y zu.
x := &y x := *y *x = y	Befehle zur Manipulation von Zeigervariablen. Der erste weist x die Adresse von y zu. Der zweite weist x den Wert der Speicherstelle zu, deren Adresse in y steht. Der dritte Befehl weist der Speicherstelle, deren Adresse in x steht, den Wert von y zu.
param x call p return	Diese Befehle dienen zur Übersetzung von Prozeduraufrufen. Ein Aufruf $p(x_1, ..., x_n)$ wird übersetzt in eine Folge von Befehlen param x1 ... param xn call p Der *return* Befehl steht im Code für die Prozedur und bewirkt die Rückkehr aus der Prozedur. Das Argument y ist optional und dient der Rückgabe des Ergebnisses in Funktionsprozeduren.

Lexikalische Analyse

14.) Welche Arten von Grundsymbolen (Token) hat eine Programmiersprache üblicherweise?
Schlüsselwörter, Operatoren, Bezeichner, Konstanten

15.) Wie lassen sich die Grundsymbole (Token) einer Quellsprache beschreiben?
Durch reguläre Ausdrücke.

16.) Was sind interessante Operationen auf Sprachen?

$L_1 \cup L_2$	$= \{ w \mid w \in L_1 \vee w \in L_2 \}$	Vereinigung
L_1L_2	$= \{ w_1w_2 \mid w_1 \in L_1, w_2 \in L_2 \}$	Konkationation
L^n	$= \{ x_1...x_n \mid x_i \in L, 1 \leq i \leq n \}$	alle Strings aus n Zeichen
L^*	$= \cup_{i \geq 0} L^i$	Abschluss – alle Strings mit ε
L^+	$= \cup_{i > 0} L^i = L^* \setminus \{\varepsilon\}$	positiver Abschluss – alle Strings ohne ε

17.) Was sind reguläre Sprachen?
Die *regulären Sprachen* über Σ werden durch folgende Regeln induktiv definiert:

(i) $\varnothing$ und $\{\varepsilon\}$ sind reguläre Sprachen.
(ii) Für jedes $a \in \Sigma$ ist $\{a\}$ eine reguläre Sprache.
(iii) Seien R und S reguläre Sprachen, dann sind auch $R \cup S$, RS und R^* reguläre Sprachen.
(iv) Nichts sonst ist eine reguläre Sprache.

18.) Was sind reguläre Ausdrücke?
Ein *regulärer Ausdruck r* beschreibt eine reguläre Sprache *L(r)*.
Reguläre Ausdrücke werden induktiv definiert:

(i) $\varnothing$ bzw. ε ist ein regulärer Ausdruck, der die reguläre Sprache $\varnothing$ bzw. $\{\varepsilon\}$ beschreibt.
(ii) Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck; er beschreibt die Sprache $\{a\}$.
(iii) Wenn *r* und *s* reguläre Ausdrücke sind, die die Sprachen *R* und *S* beschreiben, so ist auch
(r|s) ein regulärer Ausdruck, der $R \cup S$ beschreibt.
rs ein regulärer Ausdruck, der *RS* beschreibt.
r^* ein regulärer Ausdruck, der R^* beschreibt.
(iv) Nichts sonst ist ein regulärer Ausdruck.

19.) Wie lassen sich Grundsymbole (Token) erkennen?
Durch endliche Automaten.

20.) Was ist ein endlicher Automat?
Ein *deterministischer endlicher Automat* M ist gegeben als $M = (Q, \Sigma, \delta, s, F)$, wobei gilt:

(i) Q ist eine endliche, nichtleere Menge von *Zuständen*,
(ii) Σ ist ein Alphabet von *Eingabezeichen*,
(iii) $\delta: Q \times \Sigma \rightarrow Q$ eine *Übergangsfunktion*,
(iv) $s \in Q$ ist ein *Anfangszustand*,
(v) $F \subseteq Q$ ist eine Menge von *Endzuständen*.

Der Automat befindet sich jeweils in einem Zustand Q, zu Anfang im Zustand s. In jedem Schritt wird ein Eingabezeichen gelesen und der Automat geht in einen neuen Zustand, bestimmt durch δ, über. Sobald ein Zustand in F erreicht wird, gilt die bisher durchlaufene Zeichenfolge als akzeptiert.

21.) Wie wird ein endlicher Automat per Hand implementiert?
Hierzu werden erst die Zustandsdiagramme entworfen und die Aktion festgelegt (z. B. C Code). Damit lässt sich auf relativ einfache Art ein lexikalischer Analysator (Scanner) implementieren.
- Token werden als Integer Konstanten definiert (> 255, da 0 – 255 ASCII Zeichen sind)
- Zwei Zeiger auf den Puffer: *pos* auf nächstes zu lesendes Zeichen, *start_pos* auf das Zeichen, dass beim Einstieg in das gerade untersuchte Zustandsdiagramm, aktuell war.
- *state* beschreibt das aktuelle Zustandsdiagramm, *start_state* wird benutzt, um beim Fehlschlag auf den nächsten gültigen Anfangszustand zu wechseln.
- Eine Funktion *nextchar()* liefert das nächste Zeichen.
- eine Funktion *stepback()* ist notwendig, um ggf. im Zeichenstrom eins zurückzuwandern
- die lexikalische Analyse wird durch die Funktion *gettoken()* realisiert (Fallunterscheidung).
- zur Übergabe von Attributen wird eine globale Variable verwendet (der Rückgabewert der Funktion liefert schon das Token an sich).
- es muss noch beachtet werden, in welcher Reihenfolge die Zustandsdiagramme durchlaufen werden. Dabei muss z. B. REAL vor INTEGER geprüft werden, da die Integer Struktur ein Präfix von Real ist. Ansonsten sollte so sortiert werden, dass häufig benutzte Token weiter vorne stehen.
- eine Funktion *next_diagram()* wählt das nächste zu prüfende Zustandsdiagramm (bzw. dessen Startpunkt) aus, falls notwendig.

22.) Wie wird ein endlicher Automat normalerweise (also nicht per Hand) implementiert?
Statt einen Scanner komplett per Hand zu implementieren, wird normalerweise ein Scannergenerator verwendet. Der bekannteste ist lex.
Mit diesem ist es möglich, eine Spezifikation der lexikalischen Analyse in Form regulärer Definitionen anzugeben und an diese auszuführende Aktionen als C-Programmcode anzuhängen.

Lex-Spezifikation (lex.l)	→ [lex] →	lex.yy.c
lex.yy.c	→ [C-Compiler] →	a.out
Eingabezeichenfolge	→ [a.out] →	Ausgabe Tokenfolge

Natürlich kann das C-Programm auch mit einem Programm gebunden werden. Dann erfolgt der Aufruf über die Funktion *int yylex()*, welche bei jedem erneuten Aufruf das aktuelle Token zurückgibt.
Aufbau von Lexprogrammen:

```
Deklarationen
%%
Tokendefinitionen und Aktionen
%%
Hilfsprozeduren
```

Der *Deklarationsteil* enthält die Definitionen von Konstanten als Tokendarstellungen sowie *reguläre Definitonen*.

```
%{
#define OPEN 1000
...
%}
sign      [+-]
digit     [0-9]
```

Der Abschnitt *Tokendefinitionen und Aktionen* enthält selbige:

```
{sign}?{digit}+     return( INTEGER );
```

Die Zeichenkette, die zu einem Token gehört, wird über zwei Variablen *char* yytext* und *int yyleng* zur Verfügung gestellt.
Im dritten Teil wird zusätzlicher Programmcode hinterlegt, z. B. Hilfsprozeduren, die innerhalb der Aktionen benötigt werden. Diese werden mit in lex.yy.c kopiert.
Unterschied Zeichen/Metazeichen: Zeichen ist alles außer . $ ^ [ ] - ? * + | ( ) / { } < > „ \

[]	Zeichenklassen	[aby&]
-	Bereich	[A-Z]
^	Komplement	[^0-9]
?	optional	[+-]?
|	Alternative	a|bc
+	ein- oder mehrmals	{digit}+
*	kein oder mehrmals	{letter}*
( )	normale Klammerung	(a|bc)*
.	alles außer \n	
^$	Zeilenanfang bzw. -ende	^hallo \t$
/	Vorschau	-/{digit}+
{}	Kennzeichnung reguläres Symbol	{digit}
„ \	Escape Zeichen	„...“ \“

Der Lex-generierte Analysator akzeptiert bei Mehrdeutigkeit stets eine Zeichenfolge maximaler Länge.
yylval enthält schließlich Attribute, z. B. den Wert eines Integers oder eine Nummer eines Eintrages in der Symboltabelle.

23.) Warum überhaupt Beschreibung in regulären Ausdrücken, wenn kontextfreie Grammatiken der mächtiger Formalismus ist?
Effizienz: Die Mechanismen, die in Scannern ablaufen sind einfacher als die, die in Parsern ablaufen. Textstrukturen sollten soweit möglich im Rahmen der lexikalischen Analyse erkannt werden.
Ein zweiter wichtiger Punkt ergibt sich aus den Methoden, die zur Syntaxanalyse eingesetzt werden: Die Entscheidung, welche Regel anzuwenden ist, wird gewöhnlich durch Vorausschau in der Eingabe-Tokenfolge getroffen – und zwar durch Vorausschau auf *genau ein Token*! Daraus folgt, dass zum Erkennen einer bedingten Anweisung zwingend erforderlich ist, das *if* ein Token ist. Nur das Lesen von *i* reicht hier nicht aus!

Syntaxanalyse

24.) Welche Verfahren der Syntaxanalyse kennen Sie?
Top-Down bzw. Bottom-Up Ansatz

25.) Wie funktioniert die Top-Down Analyse?
Erstellung des Ableitungsbaums von der Wurzel (Startsymbol) her zu den Blättern durch Auswahl der richtigen Ableitung.

26.) Welche Probleme können bei der Top-Down Analyse entstehen?
Verlaufen in Sackgassen: Problem ist die Auswahl der richtigen Ableitung. Durch Verlaufen in „Sackgassen“ wird die Übersetzung ineffizient.
Linksrekursion: macht Analyse sogar unmöglich; bei Linksrekursion taucht das zu ersetzende Nichtterminal auf der rechten Seite der Produktion wieder ganz links auf.

27.) Was ist die Besonderheit des predictive parsing?
Vorherige Analyse der Grammatik und Erstellung einer Tabelle mit Regeln, welche Produktion bei welchen Token der Eingabefolge zu verwenden ist. Damit läuft diese Form der Top-Down Analyse nicht in Sackgassen und kommt ohne *Backtracking* aus. Allerdings kann durch Anschauen des ersten Symbols oft nicht entschieden werden, welche Produktion die Richtige ist:
cond → **if** *boolexpr* **then** *stmt* **fi** |
if *boolexpr* **then** *stmt* **else** *stmt* **fi**
Die Klasse von Grammatiken, bei denen durch Ansehen der nächsten k Terminalsymbole die richtige Entscheidung getroffen werden kann, nennt man LL(k)-Grammatiken (Lesen von links nach rechts, erkennen einer Linksableitung unter Vorausschau auf die nächsten k Zeichen).

28.) Was macht man bei Linksrekursion?
Folgende Szenarien möglich: direkt: $A \rightarrow A\alpha$ bzw. indirekt: $A \Rightarrow^* A\alpha$.
$A \rightarrow A\alpha \mid \beta$
Neue Produktionen: $A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$
Hier ändert sich die Assoziativität, d. h. bei Linksrekursion sind Operationen linksassoziativ, bei Rechtsrekursion umgekehrt.
Konkretes Beispiel: numexpr → numexpr + term | term
Daraus entsteht: numexpr → term numexpr'
numexpr' → + term numexpr' | ε

29.) Was macht man, um das Verlaufen in Sackgassen zu vermeiden?
Linksfaktorisierung.
Beispiel:
A $\rightarrow \alpha\beta_1 \mid \alpha\beta_2$
Produktion wird nun so umgeschrieben, dass zunächst eine eindeutige ausgewählt werden kann.
A $\rightarrow \alpha A'$
A' $\rightarrow \beta_1 \mid \beta_2$
Konkret:
cond → **if** *boolexpr* **then** *stmt cond-rest*
cond-rest → **fi** | **else** *stmt* **fi**

30.) Welche Möglichkeiten hat man, die Top-Down Analyse mit Vorausschau durchzuführen?
Mit Analyse-Tabelle und rekursiven Abstieg.

31.) Erstellung einer Steuermenge aus einer gegebenen Grammatik
FIRST-Menge:
Eingabe: Grammatik G und Menge von A Produktionen $A \rightarrow \alpha_1 \mid ... \mid \alpha_n$
Ausgabe: FIRST(A) und initiale Steuermengen $D_1, ... D_n$, wobei $D_i = FIRST(\alpha_i)$ mit i = 1...n

FIRST-Mengen	**Produktionen**	**Steuermengen**
{b, ε}	B → b | ε	{b} {ε}
{a, b, ε}	A → a | B	{a} {b, ε}
{a, b, c}	S → Abcd	{a, b, c}

Für 1 bis i gilt also, - wenn $\alpha_i = \varepsilon$, dann $D_i = \{\varepsilon\}$
- falls X_i ein Terminalsymbol ist oder Nichtterminal aus dem nicht ε abzuleiten ist, so ist man fertig
- nur wenn X_i ε enthält, muss X_{i+1} hinzugenommen werden

FOLLOW-Menge:
1. Initialisierung der Menge mit dem Startsymbol {$} (wobei $ ein spezielles Symbol sein soll, das die Eingabefolge abschließt).
   (2-3 solange ausführen, solange sich FOLLOW noch ändert
2. Für jede Produktion $A \rightarrow \alpha B\beta$ mit $\beta \neq \varepsilon$ füge alle Symbole in FIRST(β) außer ε in FOLLOW(B) ein.
3. Für jede Produktion $A \rightarrow \alpha B$ und jede Produktion $A \rightarrow \alpha B\beta$, bei der gilt, $\varepsilon \in FIRST(\beta)$, füge alle Symbole aus FOLLOW(A) in FOLLOW(B) ein.

Damit erhalten wir abschließend eine komplette Version der Grammatik mit endgültigen Steuermengen.

32.) Was ist $start_k$?
Sei $L \subseteq \Sigma^*$ eine beliebige Sprache und sei k > 0. Dann ist
$start_k(L) := \{w \mid (w \in L \text{ und } |w| < k) \text{ oder (es existiert } wu \in L \text{ und } |w| \ 0\ k)\}$
Für ein Wort $v \in \Sigma^*$ sei

$$start_k(v) := \begin{cases} v & \text{falls } |v| < k \\ u & \text{falls u,t existieren mit } |u| = k,\ ut = v \end{cases}$$

33.) Was ist $FIRST_k(\alpha)$?
Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik, $\alpha \in (N \cup \Sigma)^*$ und $k > 0$. Dann ist
$FIRST_k(\alpha) := start_k(\{w \mid \alpha \Rightarrow^* w\})$
Die Menge $FIRST_k(\alpha)$ beschreibt also gerade die Anfangsstücke bis zu Länge k von aus α ableitbaren Terminalworten.

34.) Was ist $FOLLOW_k(A)$?
Wenn ein Wort aus $FIRST_k(\alpha)$ kürzer ist als k, dann wird die Vorschau auf die nächsten k Zeichen noch Zeichen enthalten, die nicht aus α_i abgeleitet sind, sondern aus der Umgebung, in der das Nichtterminal A stand.
$FOLLOW_k(A)$: Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik, $A \in N$, $k > 0$.
$FOLLOW_k(A) := \{w \mid S \Rightarrow^* uAv \text{ und } w \in FIRST_k(v)\}$
$FOLLOW_k(A)$ beschreibt Terminalzeichenfolgen bis zur Länge k, die innerhalb von Ableitungen in G aus das Nichtterminal A folgen können.

35.) Wozu wird bei der Definition der Steuermengen noch die FOLLOW-Menge benötigt?
Weil es sein kann, dass man aus α ε ableiten kann.

36.) Wie wird das dann implementiert?
Direkt oder mit rekursiven Abstieg. Dabei kann man sich den entstandenen Parser wie eine abstrakte Maschine vorstellen (Bandmaschine) mit Eingabeband, Ausgabeband, Stack und Analysetabelle.

37.) Wie wird es genau direkt implementiert?
Es wird eine Analysetabelle erstellt, mit einem Eintrag an jeder Stelle (A, a). A = Stack oben, a aktuelles Token in der Eingabefolge. Der Eintrag verweist auf die nächste Produktionsnummer oder auf error. Bei ($,$) ist die Analyse erfolgreich durchgeführt worden.
Zu jedem Zeitpunkt befindet sich ein a in der Eingabefolge, A oben auf dem Stack. Ist A ein Nichtterminal und a in der Eingabefolge, so wird der Eintrag (A, a) in der Tabelle gesucht. Bei error Abbruch, ansonsten wird das Nichtterminal nach der Produktionsnummer in der Tabelle expandiert und die Produktion umgekehrt auf den Stack geschoben. Ist oben auf dem Stack ein Terminal, wird dieses mit der Eingabe verglichen. Bei Gleichheit wird in der Eingabefolge eins weiter gegangen, das Terminal oben vom Stack entfernt, sonst Fehlerbehandlung. Das ganze wie eine Maschine vorstellbar.

38.) Wie wird es durch rekursiven Abstieg implementiert?
Hier wird für jedes Nichtterminal der LL(1) Grammatik eine Prozedur angelegt. Am Anfang wird die Prozedur des Startsymbols aufgerufen und die Verarbeitung beginnt. Der Baum wird aus wechselseitigen Prozeduraufrufen aufgebaut.
Beispiel:

```
procedure assignment;
begin
  if symbol = id then
    output(4); match(id); match(:=); expr;
  else error
  fi
end;
```

39.) Können Sie die Bottom-Up Analyse konkret beschreiben?
Die Blätter werden von *unten* zu größeren Strukturen bis zur Wurzel zusammengesetzt. Eingabe wird auf den Stack geschoben und versucht *Handles* zu erkennen. Reduzierung bis zum Startsymbol.
40.) Was sind Handles?
Handles sind eine Folge von Terminalen die reduziert werden können. Dabei ist die Schwierigkeit, wann ein komplettes Handle vorliegt:
E + E * E
Hier wäre nicht etwa E + E das Handle, sondern E * E (Operatorvorrang)
Formal definiert: Sei *G* eine kontextfreie Grammatik und sei

$$S \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha \beta w$$

eine Rechtsableitung in *G*. Dann heißt β ein *Handle* der Rechtsatzform αβw.

41.) Welches ist das einfachste Verfahren der Bottum-Up Analyse?
Operator-Vorrang Analyse / Operator-Präzedenz Parser. Schwächstes Verfahren, nur eine recht eingeschränkte Klasse von Grammatiken kann behandelt werden. Kann dafür relativ einfach von Hand implementiert werden.

42.) Welche Aktionen führt so ein Parser (Bottum-Up) durch?
1. *shift*: Entnimm das nächste Symbol der Eingabefolge und lege es auf den Stack.
2. *reduce*: Ein Handle β einer Produktion A → β bildet das obere Ende des Stacks. Ersetze β auf dem Stack durch A.
3. *accept*: Auf dem Stack liegt nur noch das Startsymbol; die Eingabefolge ist leer. Akzeptiere die Eingabefolge.
4. *error*: Entscheide, dass ein Syntaxfehler vorliegt.

Da *shift/reduce* die wesentlichen Aktionen sind, werden solche Parser auch *shift-reduce* Parser genannt.

43.) Welche anderen Verfahren existieren?
LR-Analyse: mächtigste Klasse von Shift-Reduce Parsern. LR(k) Parser liest von links nach rechts und erkennt eine Rechtsableitung unter Vorausschau auf die k Zeichen. Aus folgenden Gründen attraktiv:
- Praktisch alle in Programmiersprachen vorkommende Konstrukte können analysiert werden
- allgemeinste Shift-Reduce Technik ohne Backtracking; effiziente Implementierung möglich
- echt mächtiger als LL-Parser
- frühestmögliche Fehlererkennung

kanonische LR-Parser: noch mächtiger als die LR-Parser
LALR(1)-Parser: Konstruktion durch Werkzeug wie *yacc*.

44.) Welche Einschränkungen unterliegt diese Art (Operator-Vorrang) der Analyse?
Nur eine recht eingeschränkte Klasse von Grammatiken kann damit behandelt werden.
- Auf keiner rechten Seite einer Produktion darf es zwei aufeinander folgende Nichtterminale geben.
- Es darf keine Produktionen mit gleicher rechter Seite geben.
- Es darf keine ε-Produktionen geben.
- für jedes Paar von Terminalen darf nur eine Relation gelten.

45.) Beschreiben Sie die Operator-Vorrang Analyse genauer
Vorgaben: Siehe 44.
Festlegung der Relationen:

1. Wenn eine Folge Ab auf der rechten Seite einer Produktion erscheint und wenn aus A eine Folge αa ableitbar ist, dann muss gelten a $\cdot>$ b.
2. Wenn eine Folge aA auf der rechten Seite einer Produktion erscheint und wenn aus A eine Folge bα ableitbar ist, dann muss gelten a $<\cdot$ b.
3. Wenn auf der rechten Seite einer Produktion eine Folge aAb erscheint, dann muss gelten a $\doteq$ b

Semantische Analye

46.) Wie kommen wir jetzt vom Parser zur Übersetzung?
Attributierte Grammatik: Erweiterung der Symbole um Attribute und der Regeln, um die Funktionen zur Attributberechnung.

47.) Was sind Attributierte Grammatiken?
Eine *attributierte Grammatik* enthält für jedes Symbol $X \in (N \cup \Sigma)$ eine Menge von *Attributen* A(X) und für jede Produktion p: $X_0 \rightarrow X_1 \ldots X_m$ ($X_i \in N \cup \Sigma$) eine Menge von *semantischen Regeln* R(p) der Form

$X_i.a := f(X_j.b, ..., X_k.c)$

wobei X_i usw. in der Regel vorkommende Symbole darstellen und $X_i.a$ usw. deren Attribute. Für jedes Auftreten von X in einem Ableitungsbaum ist höchstens eine Regel anwendbar, um X.a zu berechnen, für alle $a \in A(X)$.

48.) Welche Arten von Attributen exisiteren?
synthetisierte und vererbte Attribute

49.) Erklären Sie die synthetisierten und vererbten Attribute genauer – was sind die Unterschiede?
p: $X_0 \rightarrow X_1 \ldots X_m$
Das Attribut p heißt *synthetisiert*, falls es Attribut von X_0 ist. Es heißt *vererbt*, falls es Attribut eines X_i auf der rechten Seite von p ist.

50.) Wie werden synthetisierte und vererbte Attribute während der Analyse behandelt?
synthetisierte Attribute: Zu jeder Produktion werden die semantischen Regeln festgelegt. Auf dem Stack werden die Darstellungen der Symbole verwaltet, die ihre Attribute beinhaltet. Vor dem Reduktionsschritt stehen die Attribute der rechten Seite auf dem Stack zur Verfügung, die zur Produktion gehörige semantische Regeln können ausgewertet werden.
vererbte Attribute: für die geeignete Reihenfolge zur Auswertung der semantischen Regeln, wird der Abhängigkeitsgraph topologisch sortiert (nur für azyklische Graphen möglich).

51.) Welche speziellen Formen von attributierten Grammatiken kennen Sie?
S-attributierte und L-attributierte Grammatiken, dabei ist die S-attributierte besonders gut für Bottum-Up Parser geeignet.

52.) Erklären Sie S-attributierte und L-attributierte Definition – Unterschied?
Eine S-attributierte Definition ist eine syntaxgesteuerte Definition, in der nur synthetisierte Attribute vorkommen. Eine solche Definition ist besonders einfach zu behandeln, da für jeden Knoten des Ableitungsbaums seine Attribute bottom-up aus den Attributen seiner Söhne zu berechnen sind.
Eine syntaxgesteuerte Definition heißt L-attributiert, wenn jedes vererbte Attribut eines Symbols X_j auf der rechten Seite einer Produktion $X_0 \rightarrow X_1 \ldots X_m$ nur abhängt von

(i) Attributen der Symbole $X_1, \ldots, X_{j-1}$, und
(ii) vererbten Attributen von X_0.

Das L steht für „von links nach rechts“.
Dies sind Attribute, die durch rekursiven Abstieg (Tiefendurchlauf durch Syntaxbaum) berechnet werden können.

53.) Wie genau implementieren Sie eine L-attributierte Grammatik im rekursiven Abstieg?
Hier ist es wichtig, dass die synthetisierten Attribute zu den Rückgabewerten und die vererbten Attribute zu den Eingabeparametern der Funktion pro Nichtterminal werden.

54.) Erstellen Sie aus folgendem eine Procedure im Pseudocode, die den rekusiven Abstieg implementiert:
Vorgabe: $i_1 : A \rightarrow \alpha_1 | D_1$
$i_2 : A \rightarrow \alpha_2 | D_2$
Eine Funktionsprozedur für jedes Nichtterminal A (ein formaler Parameter für jedes vererbte Attribut; liefert die Werte der synthetisierten Attribute zurück).

```
if symbol ∈ D1 then
  bearbeite α1
elsif symbol ∈ D2 then
  bearbeite α2
...
else error
fi
```

Verfeinerung der rechten Seite (also bearbeite α_i); die rechte Seiten sind Folgen von Terminalsymbolen, Nichtterminalen und semantischen Regeln:
(i) Terminalsymbol X: falls X Attribute hat, dessen Werte hier benötigt werden, erzeuge Anweisung, die diese Werte in lokale Variablen übergibt, z. B.:

```
Xx = symbol.x
```

anschließend `match(x)`
(ii) Nichtterminal X: Erzeuge Zuweisungen

```
Xs := X(Xv1, ..., Xvn)
```

Xs ist lokale Variable zur Aufnahme des synthetisierten Attributs von X; rechts Aufruf der Funktionsprozedur für X.
(iii) Semantische Regel b := $f(c_1, \ldots, c_k)$:
Kopiere Regel in den Programmtext, wobei alle Bezüge auf Attribute durch entsprechende Variablen ersetzt werden.

55.) Was ist eine syntaxgesteuerte Definition?
Eng verwandt, aber etwas allgemeiner als die attributierte Grammatik. Sie erlaubt Seiteneffekte, wie z. B. Manipulation einer globalen Symboltabelle. Man kann hier beliebige Anweisungen benutzen.

56.) Was ist ein Übersetzungsschema?
Ein *Übersetzungsschema* ist eine kontextfreie Grammatik mit Attributen zu Grammatiksymbolen, in der rechten Seiten von Produktionen Folgen von Grammatiksymbolen und semantischen Regeln sind.
Die Reihenfolge der Aktionen wird genau festgelegt.

expr → *term rest*
rest → + *term* {*write*(„+“)} *rest*
| - *term* {*write*(„-“)} *rest*
| ε
term → **num** {*write*(**num**.*value*)}

Codeerzeugung

57.) Welche Aufgaben fallen bei der Verwaltung von Speicherplatz für Variable an?
- Bedeutung verschiedener Variablen
- Lebensdauer von Variable
- Sichtbarkeit von Bezeichnern
- Verwaltung von statischem Speicher (für globale und statische Variablen). Sequenzielle Aneinanderreihung der Variablen.

58.) Wie sieht das Speicherlayout aus?
Code, statischer Speicher, Stack und Heap, wobei Stack und Heap aufeinander zulaufen.

59.) Wie funktioniert die Verwaltung von Prozedurinkarnationen?
Verwaltung auf dem Keller

Rückgabeparameter
aktuelle Parameter
optionaler Kontrollzeiger
optionaler Zugriffszeiger
gesicherter Maschinenstatus
lokale Variablen
temporäre Variablen
Bereich für dynamische Arrays

Platz für *aktuelle Parameter* und *Rückgabeparameter* dienen der Parameterübergabe zwischen Aufrufer und aufgerufener Prozedur.
Kontrollzeiger zeigt auf den Beginn des Prozedurrahmens.
Zugriffszeiger zeigt auf den Prozedurrahmen einer Vorgängerprozedur p auf dem Stack, deren lexikalischer Scope den Scope des Textes der Prozedur q direkt einschließt.
Gesicherter Maschinenstatus: enthält Statusinformationen des Aufrufers, wie Programmzähler, Registerinhalte; damit kann die Umgebung bei Rückkehr wiederhergestellt werden.
Platz für *lokale Variablen*.
Speicherplatz für *dynamische Arrays* soweit in der Sprache vorhanden.

60.) Wie werden unterschiedliche Sichtbarkeitsbereiche implementiert?
Prozedurrahmen oder Displaymechanismus

61.) Wie können Optimierungsmethoden klassifiziert werden?
Unterscheidung nach Sprachniveau:
- *algebraische Optimierung*: Transformation auf Ebene der Quellsprache.
- *maschinenunabhängige Optimierung*: arbeitet auf der Ebene der Zwischensprache
- *maschinenabhängige Optimierung*: spezielle Eigenschaften des Zielprozessors

62.) Wie können maschinenunabhängige Optimierungsmethoden noch unterschieden werden?
- *lokale Optimierung*: stark begrenzter Ausschnitt des Programms
- *globale Optimierung*: weitaus mächtiger, aber zusätzliche Informationen notwendig, z. B. über Gültigkeit von Variablen, die die sog. *Datenflussanalyse* liefert.

63.) Was ist ein Basisblock?
Ein Basisblock ist eine maximale Folge von Anweisungen, die in jedem Fall nacheinander ausgeführt werden. D. h. keine Verzweigung in den Basisblock hinein oder heraus. Eindeutiger Blockanfang und eindeutiges Blockende. Hingegen ist die Reihenfolge in der die Basisblöcke durchlaufen werden unterschiedlich (je nach Parametern etc.).
Die Basisblöcke geben statische Informationen über den Programmfluss.

JJ 2008