Grundlagen

- Man kann etwas in Sprache A besser beschreiben als in Sprache B, aber die Maschine versteht nur B

- Anwendungsgebiete:
  - Übersetzung einer höheren Programmiersprache in Maschinensprache
  - Dokumentenbeschreibungssprachen
  - Datenbankanfragesprachen
  - VLSI-Entwurfssprachen
  - Protokolle in verteilten Systemen

- Compiler:
  - Erzeugt aus einem Quellprogramm in Sprache A ein Zielprogramm in Sprache B (Übersetzer)
  - Quellsprache A ist ein durch ein Regelsystem (kontextfreie Grammatik) beschrieben, das der Übersetzer benutzt, um das Quellprogramm zu analysieren
  - Ggf. Erkennung von Fehlern im Programm und Ausgabe von Fehlermeldungen

- Compiler vs. Interpreter: schnellere Programmlaufzeit vs. schnellere Übersetzungszeit

Übersetzungsphasen

1) Analysephasen

a) Lexikalische Analyse (Scanner)

- Eingabe ist eine Folge von Zeichen (Buchstaben, Ziffern, Sonderzeichen)
- Ziel ist die Erkennung gewisser Grundsymbole (Token) in diesem Zeichenstrom, z. B. Wortsymbole (begin, if, while, Bezeichner etc.)
- Struktur der Token lässt sich durch reguläre Ausdrücke oder DEAs beschreiben
- Ausgabe ist eine Folge von Token, die als Eingabe der Syntaxanalyse dient

b) Syntaxanalyse (Parser)

- Aufgabe ist die Erkennung von hierarchischen Strukturen in Programmen
- Struktur lässt sich mit (kontextfreien) Grammatiken beschreiben
- Reguläre Ausdrücke haben kein Gedächtnis (Speicher), daher kontextfreie Grammatiken
- Symbole einer Grammatik beschreiben größere Einheiten in Programmen (arithmetische Ausdrücke, bedingte Anweisungen, Schleifen etc.)
- Eingabe: Tokenfolge, Ausgabe: Syntaxbaum

c) Semantische Analyse

- Eine Attributgrammatik ist eine kontextfreie Grammatik, die um Attribute sowie Regeln und Bedingungen erweitert ist
- Anwendung, um die Einhaltung von Regeln zu überprüfen, die mit kontextfreien Grammatiken (und somit während der Syntaxanalyse) nicht formuliert werden können
- Eingabe: Syntaxbaum, Ausgabe: Attributierter Syntaxbaum
- Beispiele für Regeln:
    - Jede Variable muss deklariert sein und ihrem Datentyp entsprechend verwendet werden (Typüberprüfung)
    - Operationen müssen auf passende Argumentausdrücke angewandt werden
    - Auflösen überladener oder polymorpher Operationen

2) Synthesephasen

d) Erzeugen von Zwischencode
- Gewaltige Lücke zwischen den Konzepten einer höheren Programmiersprache und den recht primitiven Möglichkeiten der Maschinensprache als Zielsprache
- Um Komplexität des Übersetzungsproblems beherrschbar zu machen, wird Zwischenebene eingefügt, also Übersetzung in abstrakte Maschinensprache von etwas höherem Niveau, nicht in die finale Maschinensprache
- Eingabe: Attributierter Syntaxbaum, Ausgabe: Zwischencode

e) Codeoptimierung
- Die relativ mechanische Art, mit der Zwischencode aus dem vorhandenen Syntaxbaum generiert wird, führt zu möglichen Ineffizienzen im erzeugten Zwischencode
- Ineffizienzen sollen gefunden und beseitigt werden
- Eingabe: Zwischencode, Ausgabe: Optimierter Zwischencode

f) Codeerzeugung
- Aus dem optimierten Zwischencode wird Assembler- oder Maschinencode für die spezielle Zielmaschine generiert
- Wichtigste zu lösende Probleme:
  - Speicherorganisation für das Zielprogramm, möglichst gute Zuteilung von Registern
  - Abbildung von Operationen des Zwischencodes auf die bestmöglichen Befehlsfolgen der Zielmaschine
- Eingabe: Optimierter Zwischencode, Ausgabe: Maschinencode

Lexikalische Analyse

- Alphabet Σ ist eine endliche, nichtleere Menge, Σ* ist die Menge aller Folgen von Elementen aus Σ (Wörter)
- Sprache ist eine Teilmenge von Σ*
- Operationen auf Sprachen: Konkatenation, Vereinigung und Abschlussoperation

- Regulärer Ausdruck:

    - Beschreibt die Struktur eines Token
    - Induktive Definition:
        i. Die leere Menge Ø ist ein regulärer Ausdruck, der die reguläre Sprache Ø beschreibt. Das leere Wort ε ist ein regulärer Ausdruck, der die Sprache {ε} beschreibt.
        ii. Für jedes *a* ϵ Σ 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 sind jeweils auch die Konkatenation und Vereinigung von *r* und *s* sowie *r** reguläre Ausdrücke, die die jeweiligen regulären Sprachen beschreiben.
        iv. Nichts sonst ist ein regulärer Ausdruck

- Reguläre Sprache:

    - Menge der Zeichenketten, die auf ein Token abgebildet werden
    - Wird durch rechtslineare (oder linkslineare) Grammatiken erzeugt
    - Wird von (nicht-) deterministischen Automaten erkannt
    - Induktive Definition:
        i. Ø und {ε} sind reguläre Sprachen.
        ii. Für jedes *a* ϵ Σ ist {*a*} eine reguläre Sprache.
        iii. Seien *R* und *S* reguläre Sprachen, dann sind auch deren Konkatenation und Vereinigung sowie *R** reguläre Sprachen.
        iv. Nichts sonst ist eine reguläre Sprache über Σ.

- Beschreibung von Token durch Zustandsdiagramme

    - Durch reguläre Ausdrücke definierte Sprachen können mit endlichen Automaten akzeptiert werden
    - Zu einem regulären Ausdruck wird zunächst ein nichtdeterministischer endlicher Automat (NEA) konstruiert, der anschließend in einen äquivalenten deterministischen endlichen Automaten (DEA) umgewandelt wird
    - Ein DEA lässt sich relativ leicht in ein Analyseprogramm übersetzen
    - Handimplementierung eines Scanners, indem die Struktur der Token direkt mit Hilfe von Zustandsdiagrammen (graphische Notation für DEAs) angegeben wird
    - Ein DEA M ist gegeben als M = (*Q*, Σ, δ, *s*, *F*), wobei gilt:
        i. *Q* ist eine endliche nichtleere Menge von Zuständen,
        ii. Σ ist ein Alphabet von Eingabezeichen,
        iii. δ: *Q* x Σ → *Q* ist eine Übergangsfunktion,
        iv. *s* ϵ *Q* ist ein Anfangszustand,
        v. *F* ≤ *Q* ist eine Menge von Endzuständen.
    - DEAs zur Erkennung von Integer, Gleitkommazahl, String

Syntaxanalyse

- Aufgabe: Berechnung eines Syntaxbaumes (Ableitungsbaumes)
- Basis für die Syntaxanalyse ist eine Grammatik, die die Syntax der Quellsprache beschreibt, also die Struktur von Programmen
- Eine kontextfreie Grammatik ist ein Quadrupel $G = (N, \Sigma, P, S)$, wobei gilt:
  - N ist ein Alphabet von Nichtterminalen.
  - Σ ist ein Alphabet von Terminalen. Die Alphabete N und Σ sind disjunkt.
  - $P \subseteq N \times (N \cup \Sigma)^*$ ist eine Menge von Produktionsregeln.
  - $S \in N$ ist das Startsymbol.
- Ziel: Ableitungsbaum konstruieren zu einer gegebenen Grammatik und einer gegebenen Eingabesymbolfolge, die als Ergebnis der lexikalischen Analyse entstanden ist
- Zwei Strategien, um mit Hilfe der Grammatik den Syntaxbaum aus der Tokenfolge zu berechnen: Top down und Bottom up
- Struktur des Syntaxbaumes legt die Bedeutung des entsprechenden Wortes (oder Programmtextes) fest
- Existieren zu einem Terminalwort verschiedene Ableitungsbäume, so ist die zugrundeliegende Grammatik mehrdeutig; die Mehrdeutigkeit der Grammatik ist unter allen Umständen zu vermeiden

- Top down-Analyse:

  - Der Syntaxbaum wird von der Wurzel aus zu den Blättern hin aufgebaut

  - Die Blattfolge des bisher erzeugten Ableitungsbaumes wird mit der Eingabesymbolfolge verglichen, d. h., beide Symbolfolgen werden von links nach rechts gelesen

  - Solange beide gleiche Terminalsymbole enthalten, kann man weiterlesen

  - Enthält die Eingabefolge ein Terminalsymbol und das entsprechende Blatt des Baumes ein Nichtterminal, so wird diejenige Produktion der Grammatik ausgewählt, die auf dieses Nichtterminal anwendbar ist; dadurch wird die Blattfolge des Ableitungsbaumes lokal verändert

  - Falls zwei nicht übereinstimmende Terminalsymbole angetroffen werden, so ist entweder eine vorher getroffene Auswahl einer Produktion falsch gewesen und rückgängig zu machen, oder die Eingabefolge ist syntaktisch nicht korrekt

  - Liegt eine Linksrekursion (linksrekursive Produktion) vor, gelangt man in eine Endlosschleife und die Analyse ist unmöglich
    - Beseitigung der Linksrekursion:
      $A \rightarrow A\alpha \mid \beta \quad => \quad A \rightarrow \beta A' \quad \text{und} \quad A' \rightarrow \alpha A' \mid \varepsilon$

  - Allgemeinste Form der Top-down-Analyse: Top-down-Analyse mit Backtracking (wenn man sich in Sackgassen verläuft) -> ineffizient
    - Lösungsmöglichkeit zur Vermeidung von Backtracking: Predictive Parsing auf Basis von LL(*k*)-Grammatiken
      (ggf. erst Linksfaktorisierung, um Predictive Parsing überhaupt erst zu ermöglichen; dadurch gibt es immer nur eine Produktion, die ein bestimmtes Zeichen enthält)
    - Linksfaktorisierung anwenden, wenn die rechten Seiten verschiedener *A*-Produktionen ein gemeinsames Präfix haben

- Wie muss eine kontextfreie Grammatik beschaffen sein, damit eine sackgassenfreie Analyse unter Vorausschau auf die jeweils nächsten k Zeichen möglich ist? Die im Analyseprozess zu treffende Entscheidung, durch welche rechte Seite ein Nichtterminal expandiert werden soll, muss eindeutig sein

- Steuermenge wird benötigt zwecks Entscheidung, welche rechte Seite ausgewählt werden soll, wenn ein Nichtterminal A expandiert werden soll:
  - $FIRST_k(\alpha)$ beschreibt gerade die Anfangsstücke bis zur Länge *k* von aus α ableitbaren Terminalworten
  - $FOLLOW_k(A)$ beschreibt Terminalzeichenfolgen bis zur Länge *k*, die innerhalb von Ableitungen in G auf das Nichtterminal *A* folgen können
  - Wenn ein Wort aus $FIRST_k(\alpha_i)$ kürzer als *k* ist, dann wird die Vorausschau auf die nächsten *k* Zeichen noch Zeichen enthalten, die nicht aus α_i abgeleitet sind, sondern aus der Umgebung (FOLLOW-Menge), in der das Nichtterminal *A* stand.

    WOFÜR WERDEN FOLLOW-MENGEN BENÖTIGT?

  - Die Steuermenge, die die Konkatenation der FIRST und FOLLOW-Mengen ist, muss disjunkt sein (damit Eindeutigkeit gewährleistet ist)!

- Bottom up-Analyse:

  - Der Syntaxbaum wird von den Blättern her bis zur Wurzel aufgebaut

  - Grundidee: So lange Token einlesen und merken, bis eine vollständige rechte Seite einer Grammatikregel (Produktion) gelesen worden ist; eine solche vollständige rechte Seite wird als Handle bezeichnet

  - Aber: Kriterium „vollständige rechte Seite vorhanden“ nicht ausreichend: das zentrale Problem der Bottom-Up-Analyse besteht darin, zu entscheiden, ob bei Vorliegen einer vollständigen rechten Seite reduziert werden soll oder ob zunächst noch weitere Zeichen hinzugenommen werden sollen
    - wenn man nicht zum Startsymbol gelangt, war es kein Handle
    - bei einer eindeutigen Grammatik hat jede Rechtssatzform genau ein Handle

  - Implementierung mit Hilfe eines Stacks:
    - Eingabesymbole werden jeweils auf den Stack gelegt
    - Sobald am oberen Ende des Stacks ein Handle β einer Produktion $A \rightarrow \beta$ erscheint, wird reduziert, d.h., die Symbole von β werden vom Stack entfernt und *A* wird an ihrer Stelle auf den Stack gelegt

  - Bottom up-Parser führt 4 Aktionen durch:
    - Shift: Entnimm das nächste Symbol der Eingabefolge und lege es auf den Stack
    - Reduce: Ein Handle β einer Produktion $A \rightarrow \beta$ bildet das obere Ende des Stacks, dann ersetze β auf dem Stack durch *A*
    - Accept: Auf dem Stack liegt nur noch das Startsymbol; die Eingabefolge ist leer. Dann akzeptiere die Eingabefolge
    - Error: Entscheide, dass ein Syntaxfehler vorliegt

- Wie kann der Parser Handles erkennen: Er müsste feststellen, dass nach einem Shift-Schritt ein Handle auf dem Stack liegt, und zweitens die Anfangsposition (das erste Symbol) des Handles erkennen
    - Operator-Vorrang-Analyse:
        - Grundidee: Definition dreier Relationen, die zwischen aufeinanderfolgenden Symbolen auf dem Stack bestehen können: <, =, > (spitze Klammern zeigen Grenzen des Handles an
        - Man shifted solange, wie zwischen dem oberstem Stacksymbol und dem nächsten Eingabesymbol die Beziehung < oder = besteht
        - Sobald die Beziehung > auftritt, wird reduced

- LR-Parser (bei Bottom up) sind mächtigste Klasse von Shift-Reduce-Parsern:
    - Praktisch alle in Programmiersprachen vorkommenden Konstrukte können damit analysiert werden
    - Allgemeinste Shift-Reduce-Technik, die ohne Backtracking auskommt
    - LR-Parser sind echt mächtiger als LL-Parser
        - Für alle Grammatiken, für die man Predictive Parser konstruieren kann, kann man auch LR-Parser bauen
        - Es gibt Konstrukte, die mit LR-Parsern, nicht aber mit LL-Parsern analysierbar sind
    - Man kann Fehler frühestmöglich erkennen
    - Ein LR-Parser entscheidet, dass in der Ableitung eines zu analysierenden Wortes die Produktion $A \rightarrow \beta$ angewandt wurde, nachdem er alles gesehen hat, was aus den Symbolen von β abgeleitet wurde (zu jedem Symbol aus β liegt der Ableitungsbaum schon auf dem Stack) sowie die nächsten *k* Zeichen der Eingabe. Ein LL-Parser muss diese Entscheidung treffen, nachdem er nur die ersten k Zeichen des aus β abgeleiteten Terminalwortes gesehen hat
    - Nachteil: Analysetabelle von Hand nur sehr schwer konstruierbar, daher Werkzeuge wie Yacc

Semantische Analyse

- Erläuterung „Wofür Typechecking“: Compilerbau 8 – Typechecking, ca. Minute 26
- Erläuterung Symboltabelle: Compilerbau 8 – Typechecking, ca. Minute 45

- Methodisches Problem, Übersetzungsaktionen mit dem Erkennen von Teilstrukturen des Übersetzungsbaumes zu verbinden, soll gelöst werden; Übersetzungsschritte werden mit der Analyse verzahnt
- Daher syntaxgesteuerte Übersetzung, deren formale Grundlage attributierte Grammatiken bilden
- Man hat seinen Ableitungsbaum und hängt an den relevanten Stellen Codefragmente an

- Synthetisierte Attribute: Attribut wird aus Attributen der Kinderknoten berechnet (Kinderknoten sind unabhängig voneinander)
- Vererbte Attribute: Attribut wird aus Attributen der Eltern- oder Geschwisterknoten berechnet (Attribute auf der rechten Seite einer Produktion hängen voneinander ab); Beispiel Typisierung: Erster Wert auf der rechten Seite der Produktion ist ein Integer und zweiter Wert auf der rechten Seite der Produktion muss ebenfalls Integer sein

- S-Attributgrammatik: Alle Attribute werden synthetisiert (keine Interaktion zwischen Teilbäumen)
- L-Attributgrammatik: Nur erben von linken Geschwisterknoten

- Symboltabelle: Datenstruktur, die Typinformationen speichert (z. B. Hashmap oder Record) und das Problem der lokalen Sichtbarkeit von Symbolen löst (Variablen, Funktionen)

Zwischencode

- Speicherorganisation: Code, statistischer Speicher, Stack, Heap

- (abstrakte) Syntaxbäume: vereinfachte Darstellungen konkreter Syntaxbäume, bei denen die Struktur weniger an grammatikalischen Kategorien (Nichtterminalen) als an der Bedeutung des Konstrukts, d.h. den durchzuführenden Operationen, orientiert ist

- Gerichtete azyklische Graphen (DAG): Syntaxbaum, in dem mehrfach auftretende Teilstrukturen nur einmal vorkommen

- Postfix-Notation (z. B. PostScript): zunächst werden die Operanden und danach die auszuführende Operation hingeschrieben. Stack-Maschine: Operanden werden auf den Stack geladen, Operationen entnehmen die obersten Elemente auf dem Stack als Argumente und legen ihr Ergebnis wieder auf den Stack

- 3-Adress-Code: Befehle mit bis zu drei Argumenten. Befehlsfolgen lassen sich leichter umordnen, da mit expliziten Variablen gearbeitet wird. Klassen von Befehlen:

x := y op z	op ist binärer Operator (+, *, and, usw.)
x := op y	op ist unärer Operator (z. B. not)
x := y	einfache Zuweisung
goto L	Sprung mit Sprungmarke L
if x cop y goto L	cop ist Vergleichsoperator (=, <, <=, usw.)
x := y[i] und x[i] := y	indizierte Zuweisung zur Übersetzung von Array-Zugriffen
x := &y	Manipulation von Zeigervariablen, x := Adresse von y
x := *y	x := Wert der Speicherzelle, deren Adresse in y steht
*x := y	Speicherzelle, deren Adresse in x steht, wird der Wert von y zugewiesen
param x	Übersetzung von Prozeduraufrufen
call(p)	$p(x_1, \ldots, x_n)$ = param x_1, …, param x_n, call p
return(y)	Rückkehr aus der Prozedur, Rückgabeargument y ist optional

Codeoptimierung

1) Maschinenunabhängige Optimierung

   - Lokale Optimierung

     (O1) Konstantenpropagation und Konstantenfaltung
     x := 3, y := 4, z := x+y => z := 3*4 (Propagation) => z := 12 (Faltung)

     (O2) Kopierpropagation
     x := y, z := a * x => z := a * y

     (O3) Reduzierung der Stärke von Operatoren
     Komplexe Operationen werden in Spezialfällen durch einfachere ersetzt

     (O4) In-Line Expansion

     (O5) Elimination redundanter Berechnungen

   - Schleifenoptimierung

     (O6) Verlagerung von Schleifeninvarianten

     (O7) Vereinfachung von Berechnungen mit Schleifenvariablen

     (O8) Schleifenentfaltung

   - Globale Optimierung

     (O9) Elimination toten Codes

     (O10) Code Hoisting
     Wenn man mittels Datenflussanalyse feststellen kann, dass jede beteiligte Variable innerhalb des betrachteten Bereiches immer den gleichen Wert hat, so kann man Teilausdrücke auch über Basisblöcke hinweg faktorisieren

2) Maschinenabhängige Optimierung

   (O11) Anweisungsreihenfolge und Registerauswahl

   (O12) Befehlsauswahl

   (O13) Peephole Optimization
   Redundante Anweisungen eliminieren (z.B. Ladebefehl überflüssig, weil Wert schon in einem Register steht oder Zusammenfassung von Sprüngen)