**Inhalt der Vorlesung „Compiler-Konstruktion“im WS 2015/16**

## Einleitung

- Prinzipieller Aufbau eines Compilers, Aufgaben der einzelnen Phasen

## Attributierte Grammatik

- Definition einer attributierten Grammatik, synthetische und inherite Attribute, semantische Regeln.
- attributierter Ableitungsbaum, ausgewerteter (attributierter) Ableitungsbaum, Auswertbarkeit des attributierten Ableitungsbaums, das ausgewählte Attribut des Startsymbols.
- Abhängigkeitsgraphen von Produktionen, Abhängigkeitsgraphen für Ableitungsbäume, Zyklenfreiheit
- Prinzip der Übersetzung eines Wortes $w$ mit Hilfe einer attributierten Grammatik.
- SDTS als spezielle attributierte Grammatik
- Definition S-attributierte und L-attributierte Grammatiken.
- S-attributierte Grammatiken und Bottom-Up-Parsing
- L-attributierte Grammatiken und Top-Down-Parsing.
- Umformungen:
  - Verschieben von Aktionen in einem SDTS ans Ende der Produktion
  - Entfernen von Linksrekursionen in S-attributierten Grammatiken.

## Typ-Prüfung und Typ-Anpassung

- Typ einer Variablen, Typ-Ausdrücke, Typ-Prüfung, Typ-Inferenz für Konstrukte der Programmiersprache
- dynamische versus statische Typ-Prüfung.
- Äquivalenz von Typ-Ausdrücken (Strukturäquivalenz und Namensäquivalenz).
- Typ-Umwandlungen, Untertypen, Vererbung
- Überladene Funktionen, Bestimmung des „richtigen“ Typs
- polymorphe Funktionen, Typ-Variable, Unifikation

## Das Laufzeitsystem

- Gültigkeitsbereich und Lebensdauer von Objekten
- Laufzeit-Stack, Aktivierungsrecord, statische und dynamische Verkettung,
- zusätzliche Probleme bei Prozedurparametern (Closure)
- Speicherorganisation:
  - statischer Speicher,

  - dynamischer Speicher (Heap),
  - Laufzeit-Stack.
- Aufbau eines Aktivierungsrecords
- Formen der Parameterübergabe
- Speicherverwaltung in objektorientierten Systemen am Beispiel Smalltalk
  - Objekttabelle
  - Darstellung von Integer-Zahlen
  - Darstellung von Methoden
  - Darstellung von Klassen
- Heap Management
  - Speicherhierarchie
  - Aufgaben der Speicherverwaltung (Allokation, Deallokation)
  - Reference Count
  - Garbage Collection

### Zwischencode-Erzeugung

- verschieden Formen des Zwischencodes (Syntax-Bäume, Drei-Adress-Code, DAGs)
- Übersetzung von Deklarationen.
- Übersetzung arithmetischer Ausdrücke
- Übersetzung von Feldzugriffen, zeilen- spaltenweise Speicherung.
- Übersetzung Boolescher Ausdrücke, Möglichkeiten der Auswertung bei Booleschen Ausdrücken
  1. Darstellung der Booleschen Werte durch numerische Kodierung,
  2. Darstellung durch Steuerung des Programmablaufes.
     Lösung durch:
     (a) inherite Attribute, die das Ziel des Vorwärtssprungs angeben,
     (b) Backpatching unvollständiger Sprungbefehle
- Übersetzung von Steuerbefehlen, etwa
  - `if`-Anweisungen oder
  - `while`-Anweisungen.

### Codeoptimierung

- Forderungen an die Code-Optimierung
  1. keine Änderung an der Funktion eines Programms
  2. Verbesserung der Laufzeit
  3. Kosteneffizienz

- Möglicher Aufbau eines Codeoptimierers
- Definition und Konstruktion einfacher Blöcke
- Flussgraphen.
- Definition und Gebrauch von Variablen.
- Lebendigkeit von Variablen
- Entfernen gemeinsamer Teilausdrücke (lokale Optimierung mit DAGs).
- Datenflussanalyse
  - Transferfunktionen für Drei-Adress-Befehle und einfache Blöcke
  - Vorwärtsanalyse, Rückwärtsanalyse.
  - zugehörige Datenflussgleichungen
- Reaching Definitions (verfügbare Definitionen)
  1. Datenfluss-Gleichungen (Vorwärtsanalyse):

$$\begin{aligned} \texttt{out}[\texttt{B}] &= \texttt{gen}[\texttt{B}] \cup (\texttt{in}[\texttt{B}] - \texttt{kill}[\texttt{B}]) \\ \texttt{in}[\texttt{B}] &= \bigcup_{\texttt{P} \in \text{pred}(\texttt{B})} \texttt{out}[\texttt{P}] \end{aligned}$$

  2. Bedeutung der Mengen `in`, `out`, `gen` und `kill`.
  3. Wie löst man ein derartiges Gleichungssystem?
- Lebendigkeit von Variablen
  1. Datenfluss-Gleichungen (Rückwärtsanalyse):

$$\begin{aligned} \texttt{in}[\texttt{B}] &= \texttt{use}[\texttt{B}] \cup (\texttt{out}[\texttt{B}] - \texttt{def}[\texttt{B}]) \\ \texttt{out}[\texttt{B}] &= \bigcup_{\texttt{S} \in \text{succ}(\texttt{B})} \texttt{in}[\texttt{S}] \end{aligned}$$

  2. Bedeutung der Mengen `in`, `out`, `def` und `use`.
  3. Wie bestimmt man die `def`- und die `use`-Menge?
- Available Expressions (verfügbare Ausdrücke)
  1. Datenfluss-Gleichungen (Vorwärtsanalyse):

$$\begin{aligned} \texttt{out}[\texttt{B}] &= \texttt{e\_gen}[\texttt{B}] \cup (\texttt{in}[\texttt{B}] \texttt{ - } \texttt{e\_kill}[\texttt{B}]) \\ \texttt{in}[\texttt{B}] &= \bigcap_{\texttt{P} \in \text{pred}(\texttt{B})} \texttt{out}[\texttt{P}] \end{aligned}$$

  2. Bedeutung der Mengen `in`, `out`, `e_gen` und `e_kill`.
- Analyse der Ablaufstruktur
  - Definition der dom-Relation

- Bestimmung der dom-Relation (Vorwärtsanalyse)

$$\begin{aligned} \texttt{out[B]} &= \{\texttt{B}\} \cup \texttt{in[B]} \\ \texttt{in[B]} &= \bigcap_{\texttt{P} \in \text{pred}(\texttt{B})} \texttt{out[P]} \end{aligned}$$

  mit Initialisierung `out[B]` = Knotenmenge des Flussgraphen,
  `out[entry]` = {`entry`}.
- Rückwärtskanten
- natürliche Schleifen

### Maschinencode Erzeugung

- Formen des Maschinencodes
- Benutzung der Informationen über Lebendigkeit und nächsten Gebrauch von Variablen bei der Erzeugung von Maschinencode
- Maschinencode-Erzeugung für Syntaxbäume (Ershov-Zahlen)
- Verwendung des Prinzips der dynamischen Programmierung
  - Berechnung der Kostenvektoren
  - Code-Erzeugung
- globale Register Allokation
  - über Usage Counts
  - über Graph-Färbung
- Prinzipielle Arbeitsweise von Generatoren für die Maschinencode-Erzeugung

### Übungsstoff

- Aufbau des lexikalen Scanners?
- Bearbeitung des Deklarationsteils eines Programms.
- Übersetzung in einen Syntaxbaum
- Prinzipieller Arbeitsweise der TreeWalker. Implementation in Java.
- Typ-Prüfung und Anpassung, Typ-Informationen?
- Transformieren der Feld-Zugriffe
- Code-Erzeugung, break-Anweisung
- Verringerung der Zahl der Label und der temporären Variablen.