Tutorium 14: Compiler

Paul Brinkmeier

03. Februar 2020

Tutorium Programmierparadigmen am KIT

Einführung

Compiler in ProPa

- Ein bisschen...
  - Lexikalische Analyse (Tokenisierung)
  - Syntaktische Analyse (Parsen)
  - Semantische Analyse (Optimierung)
  - Codegenerierung

Compiler in ProPa

- Ein bisschen...
  - Lexikalische Analyse (Tokenisierung)
  - Syntaktische Analyse (Parsen)
  - Semantische Analyse (Optimierung)
  - Codegenerierung
- Klausur:
  - SLL(k)-Form beweisen
  - Rekursiven Abstiegsparser schreiben/vervollständigen
  - First/Follow-Mengen berechnen
  - Java-Bytecode

$$\mathit{SGML} \rightarrow \texttt{<}\ \texttt{id}\ \texttt{>}\ \ \mathit{Children}\ \ \texttt{<}\ \texttt{/}\ \texttt{>}$$

$$\mathit{Children} \rightarrow \epsilon \mid \mathit{SGML}\ \ \mathit{Children}$$

$$\{\texttt{<id><id></><id></></>},\ \ \texttt{<id></>},\ \ \ldots\} \in G$$

$$\mathit{SGML} \rightarrow \texttt{<}\ \texttt{id}\ \texttt{>}\ \mathit{Children}\ \texttt{<}\ \texttt{/}\ \texttt{>}$$

$$\mathit{Children} \rightarrow \epsilon \mid \mathit{SGML}\ \ \mathit{Children}$$

$$\{\texttt{<id><id></><id></></>},\ \ \texttt{<id></>},\ \ ...\} \in G$$

- Begründen Sie formal, dass die obige Grammatik nicht in SLL(1)-Form ist (3P.).

$$\textit{SGML} \rightarrow \texttt{<}\ \texttt{id}\ \texttt{>}\ \textit{Children}\ \texttt{<}\ \texttt{/}\ \texttt{>}$$

$$\textit{Children} \rightarrow \epsilon \mid \textit{SGML}\ \ \textit{Children}$$

$$\{\texttt{<id><id></><id></></>},\ \ \texttt{<id></>},\ \ ...\} \in G$$

- Begründen Sie formal, dass die obige Grammatik nicht in SLL(1)-Form ist (3P.).
- Entwickeln Sie für [eine linksfaktorisierte Version der obigen Grammatik] einen rekursiven Abstiegsparser (16P.).

Java-Bytecode (16SS)

Übersetzen Sie folgenden Java-Programmausschnitt in Java-Bytecode (10P.):

```
if (((a < b) || !((a < c) || (c < b))) && !(c < 0)) {
    c = b + a;
}
```

Java-Bytecode (16SS)

Übersetzen Sie folgenden Java-Programmausschnitt in Java-Bytecode (10P.):

```
if (((a < b) || !((a < c) || (c < b))) && !(c < 0)) {
    c = b + a;
}
```

Hinweise:

- Codeerzeugung für bedingte Sprünge: Folien 447ff.
- Um eine Bedingung der Form `!cond` zu übersetzen, reicht es, `cond` zu übersetzen und die Sprungziele anzupassen.

Compiler

Compiler: Motivation

- Maschine(-nmodell) versteht i.d.R. eingeschränkten Instruktionssatz
  - Es gibt/gab zwar auch mal CISC-Maschinen, heute ist sind aber RISC(-ähnliche) Prozessoren am weitesten verbreitet
  - Gründe: RISC-Prozessoren sind wesentlich einfacher (= billiger) zu bauen
- Programme in Maschinensprache sind i.d.R. für Menschen nicht einfach zu Schreiben.

Compiler: Motivation

- Maschine(-nmodell) versteht i.d.R. eingeschränkten Instruktionssatz
  - Es gibt/gab zwar auch mal CISC-Maschinen, heute ist sind aber RISC(-ähnliche) Prozessoren am weitesten verbreitet
  - Gründe: RISC-Prozessoren sind wesentlich einfacher ($=$ billiger) zu bauen
- Programme in Maschinensprache sind i.d.R. für Menschen nicht einfach zu Schreiben.
- Also: Erfinde einfacher zu Schreibende ($\approx$ mächtigere) Sprache, die dann in die Sprache der Maschine übersetzt wird.
- Diesen Übersetzungsschritt sollte optimalerweise ein Programm erledigen, da wir sonst auch einfach direkt Maschinensprache-Programme schreiben können.

Compiler

- Übersetzer für formale Sprachen nennt man *Compiler*
- Beispiele:
  - C, Haskell, Rust, Go $\rightarrow$ X86
  - Java, Clojure, Kotlin $\rightarrow$ Java-Bytecode
  - TypeScript $\rightarrow$ JavaScript
  - Python $\rightarrow$ Python-AST

Compiler

- Übersetzer für formale Sprachen nennt man *Compiler*
- Beispiele:
  - C, Haskell, Rust, Go $\rightarrow$ X86
  - Java, Clojure, Kotlin $\rightarrow$ Java-Bytecode
  - TypeScript $\rightarrow$ JavaScript
  - Python $\rightarrow$ Python-AST
- Interpreter kann man auch als Compiler kategorisieren, sie zählen aber i.A. nicht dazu

Compiler

- Übersetzer für formale Sprachen nennt man *Compiler*
- Beispiele:
  - C, Haskell, Rust, Go $\rightarrow$ X86
  - Java, Clojure, Kotlin $\rightarrow$ Java-Bytecode
  - TypeScript $\rightarrow$ JavaScript
  - Python $\rightarrow$ Python-AST
- Interpreter kann man auch als Compiler kategorisieren, sie zählen aber i.A. nicht dazu
- Single-pass vs. Multi-pass
  - Single-pass: Eingabe wird einmal gelesen, Ausgabe währenddessen erzeugt (ältere Compiler)
  - Multi-pass: Eingabe wird in Zwischenschritten in verschiedene Repräsentationen umgewandelt
    - Quellsprache, Tokens, AST, Zwischensprache, Zielsprache

Lexikalische Analyse

```
int x1 = 123;
print("123");
```

```
int, id[x1], assign,
intlit[123], semi,
id[print], lp,
stringlit["123"], ...
```

- Lexikalische Analyse (Tokenisierung) verarbeitet eine Zeichensequenz in eine Liste von *Tokens*.
- Tokens sind Zeichengruppen, denen eine Semantik innewohnt:
  - `int` — Typ einer Ganzzahl
  - `=` — Zuweisungsoperator
  - `x1` — Variablen- oder Methodenname
  - `123` — Literal einer Ganzzahl
  - `"123"` — String-Literal
  - etc.
- Lösbar mit regulären Ausdrücken, Automaten

Syntaktische Analyse

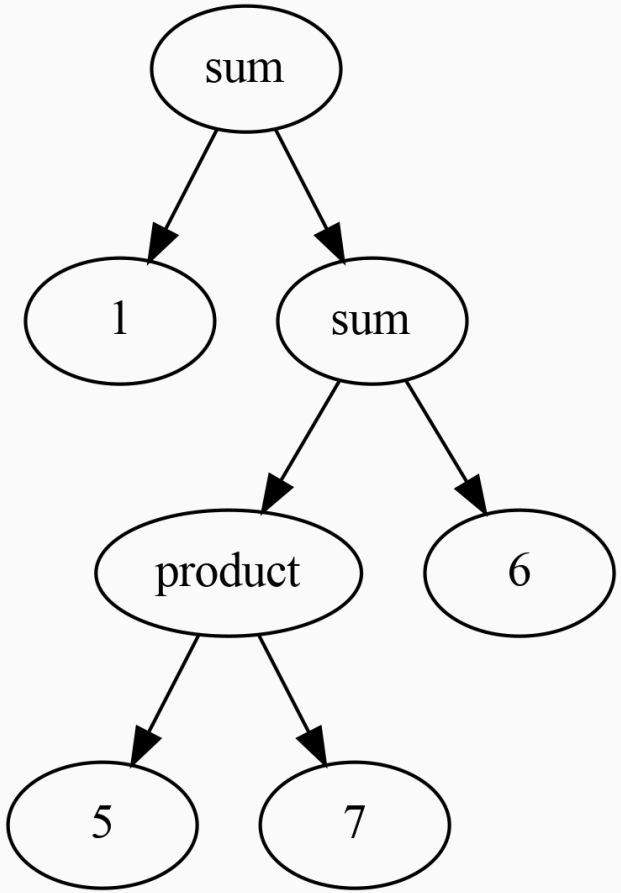
- Syntaktische Analyse stellt die unterliegende Struktur der bisher linear gelesenen Eingabe fest:
  - Blockstruktur von Programmen
  - Baumstruktur von HTML-Dateien
  - Header, Inhalt-Struktur von Mails
  - Verschachtelte mathematische Ausdrücke
- Syntaktische Analyse ist das größte Compiler-Thema in PP.

Syntaktische Analyse

- Syntaktische Analyse stellt die unterliegende Struktur der bisher linear gelesenen Eingabe fest:
  - Blockstruktur von Programmen
  - Baumstruktur von HTML-Dateien
  - Header, Inhalt-Struktur von Mails
  - Verschachtelte mathematische Ausdrücke
- Syntaktische Analyse ist das größte Compiler-Thema in PP.
- Übliche Vorgehensweise (in PP):
  - Grammatik G erfinden
  - ggf. G in einfache Form G' bringen
  - rekursiven Abstiegsparser für G' implementieren
- Alternativ: Parser-Kombinatoren, Yacc, etc.

Beispiel: Mathematische Ausdrücke

$1 + 5 * 7 + 6$



- Zu beachten: Punkt-vor-Strich (Präzedenz), Klammerung, etc.
- Nicht mehr mit regulären Ausdrücken lösbar

Links-/Rechtsrekursion

First-/Followmenge

SLL-Kriterium

Semantische Analyse

Codegenerierung