

Tutorium 14: Compiler

Paul Brinkmeier

03. Februar 2020

Tutorium Programmierparadigmen am KIT

Einführung

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

- 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

$$SGML \rightarrow < id > Children < / >$$
$$Children \rightarrow \epsilon \mid SGML \ Children$$
$$\{<id><id></><id></></>, <id></>, \dots\} \in G$$

$$SGML \rightarrow < id > Children < / >$$
$$Children \rightarrow \epsilon \mid SGML \ Children$$
$$\{<id><id></><id></></>, <id></>, \dots\} \in G$$

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

$$SGML \rightarrow < id > Children < / >$$
$$Children \rightarrow \epsilon \mid SGML \ Children$$
$$\{<id><id></><id></></>, <id></>, \dots\} \in G$$

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

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

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


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.

- 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

- 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

- 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"], ...
```

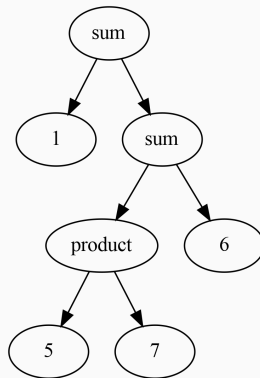
- Lexikalische Analyse (Tokenisierung) verarbeitet eine Zeichensequenz in eine Liste von *Tokens*.
- Tokens sind Zeichengruppen, denen eine Semantik innewohnt:
 - `int` — Typ einer Ganzzahl
 - `=` — Zuweisungsoperator
 - `x1` — Variablen- oder Methodenname
 - `123` — Literal einer Ganzzahl
 - `"123"`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

- Syntaktische Analyse 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 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

$$1 + 5 * 7 + 6$$



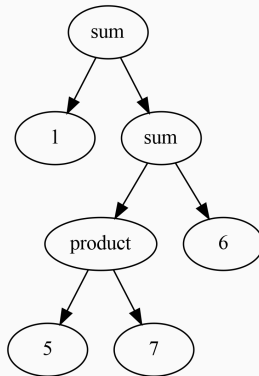
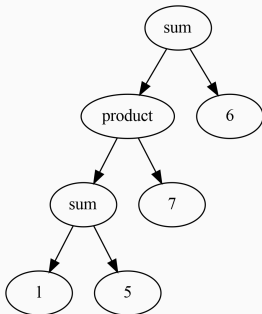
- Zu beachten: Punkt-vor-Strich (Präzedenz), Klammerung, etc.
- Nicht mehr mit regulären Ausdrücken lösbar
- „Offensichtliche“ Grammatik oft nicht einfach zu Parsen

Beispiel: Mathematische Ausdrücke

$$E \rightarrow \text{num} \mid (E) \mid E + E \mid E * E$$

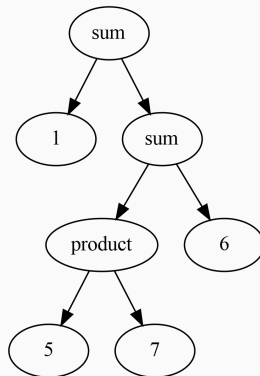
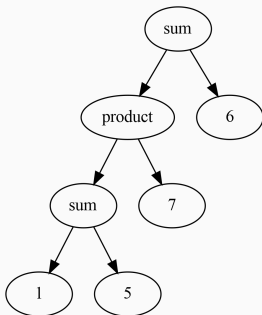
Beispiel: Mathematische Ausdrücke

$$E \rightarrow \text{num} \mid (E) \mid E + E \mid E * E$$



Beispiel: Mathematische Ausdrücke

$$E \rightarrow \text{num} \mid (E) \mid E + E \mid E * E$$



- Ableitungsbaum nicht eindeutig \rightsquigarrow schlecht
- Ableitungsbaum garantiert nicht Punkt-vor-Strich \rightsquigarrow schlecht

Wie zeichnen sich „gute“ Grammatiken aus?

Präzedenz, Linksfaktorisierung

Wie zeichnen sich „gute“ Grammatiken aus?

Operatorpräzedenz schon in Grammatik definiert:

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow \text{num} \mid (E)$$

Präzedenz, Linksfaktorisierung

Wie zeichnen sich „gute“ Grammatiken aus?

Operatorpräzedenz schon in Grammatik definiert:

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow \text{num} \mid (E)$$

Vermeidung von Linksrekursion (Linksfaktorisierung):

$$E \rightarrow T \ EList$$

$$EList \rightarrow \epsilon \mid + \ EList \mid - \ EList$$

$$T \rightarrow F \ TList$$

$$TList \rightarrow \epsilon \mid * \ TList \mid / \ TList$$

$$F \rightarrow \text{num} \mid (E)$$

$$EList \rightarrow \epsilon \mid + EList \mid - EList$$

Wie können wir bspw. bei *EList* entscheiden, welche Produktion anzuwenden ist?

First-/Followmenge, Indizmenge

$$EList \rightarrow \epsilon \mid + EList \mid - EList$$

Wie können wir bspw. bei $EList$ entscheiden, welche Produktion anzuwenden ist?

- \rightsquigarrow definiere *Indizmenge* $IM_k(A \rightarrow \alpha) = \text{First}_k(\alpha \text{Follow}_k(A))$
- Wenn nächste k Token in $IM_k(EList \rightarrow \phi) \rightsquigarrow$ weiter mit ϕ
- Konkret:
 - $IM_1(EList \rightarrow \epsilon) = \text{First}_1(\epsilon \text{Follow}_1(EList)) = \{), \#\}$
 - $IM_1(EList \rightarrow + EList) = \text{First}_1(+ EList \text{Follow}_1(EList)) = \{+\}$
 - $IM_1(EList \rightarrow - EList) = \text{First}_1(- EList \text{Follow}_1(EList)) = \{-\}$

First-/Followmenge, Indizmenge

$$EList \rightarrow \epsilon \mid + EList \mid - EList$$

Wie können wir bspw. bei $EList$ entscheiden, welche Produktion anzuwenden ist?

- \rightsquigarrow definiere *Indizmenge* $IM_k(A \rightarrow \alpha) = \text{First}_k(\alpha \text{Follow}_k(A))$
- Wenn nächste k Token in $IM_k(EList \rightarrow \phi) \rightsquigarrow$ weiter mit ϕ
- Konkret:
 - $IM_1(EList \rightarrow \epsilon) = \text{First}_1(\epsilon \text{Follow}_1(EList)) = \{), \#\}$
 - $IM_1(EList \rightarrow + EList) = \text{First}_1(+ EList \text{Follow}_1(EList)) = \{+\}$
 - $IM_1(EList \rightarrow - EList) = \text{First}_1(- EList \text{Follow}_1(EList)) = \{-\}$
- $\text{First}_k(A)$: Menge an möglichen ersten k Token in A
- $\text{Follow}_k(A)$: Menge an möglichen ersten k Token nach A

Grammatik ist in SLL(k)-Form

$$:\Leftrightarrow \forall A \rightarrow \alpha, A \rightarrow \beta \in P : IM_k(A \rightarrow \alpha) \cap IM_k(A \rightarrow \beta) \neq \emptyset$$

- SLL(k): Bei einem Nichtterminal muss die zu wählende Produktion an den nächsten k Token wählbar sein.
- Nichtterminale mit nur einer Produktion sind hier irrelevant
- Schwierig daran: Follow-Mengen berechnen

SLL-Kriterium

Grammatik ist in SLL(k)-Form

$$:\Leftrightarrow \forall A \rightarrow \alpha, A \rightarrow \beta \in P : IM_k(A \rightarrow \alpha) \cap IM_k(A \rightarrow \beta) \neq \emptyset$$

- SLL(k): Bei einem Nichtterminal muss die zu wählende Produktion an den nächsten k Token wählbar sein.
- Nichtterminale mit nur einer Produktion sind hier irrelevant
- Schwierig daran: Follow-Mengen berechnen

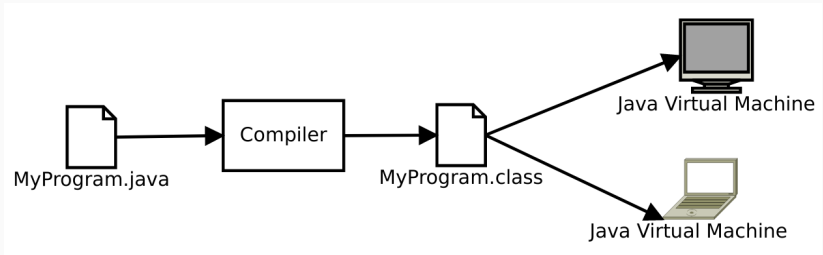
$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow \text{num} \mid (E)$$

- Begründet formal, dass obige Grammatik nicht SLL(1).
- Berechnet $\text{Follow}_1(N)$ für $N \in \{E, T, F\}$.

- PP beschäftigt sich (bis auf Typinferenz) nur kurz mit semantischer Analyse
- Hier geht es um Optimierungen, Typchecks, etc.
- \rightsquigarrow weiterführende (Master-)Vorlesungen am IPD



- Codegenerierung wird in PP am Beispiel Java-Bytecode demonstriert
 - „Jede_r kann Java“
 - Relativ übersichtliche „Maschinensprache“
 - Ggf. relevant, wenn seltsame Bugs in Java-Programmen auftreten

Java-Bytecode: Lokale Variablen, Operandenstack

- ?load_<x>, ?store_<x>
 - <x>: ID einer lokalen Variablen
- ?add
- ?sub
- ?mul
- bipush <c>
- ?const_<c>

?-Instruktionen gibt es für verschiedene Typen, bspw. iadd, fadd

- `getfield <field>, putfield <field>`
- `invokevirtual <m>, invokestatic <m>`
- `new <t>, newarray <t>`
- `?return`
 - `areturn` für Objekte

- `goto <label>`
- `if(eq|ne|lt|gt|le|ge) <label>`
- `if_?cmp(eq|ne|lt|gt|le|ge) <label>`

Kennen sollte man:

- Methodenaufrufe (`invokevirtual`)
- Zugriff auf Attribute (`getfield`, `putfield`)
- `if`, `while`
- Kurzauswertung (Laziness bei `&&` und `||`)
- Negation (Kein `not`-Befehl, nur veränderte Sprungziele)