

Tutorium 13: Actoren & Compiler

David Kaufmann

7. Februar 2023

Tutorium Programmierparadigmen am KIT

Actor Model

- Actors sind computation units mit State, Behaviour, Mailbox
- kommunizieren über Nachrichten
- verarbeiten immer nur eine Nachricht
- verarbeiten Nachrichten in der Reihenfolge in der sie empfangen wurden

```
public class HelloWorldActor extends AbstractActor{
    @Override
    public Receive createReceive() {
        return receiveBuilder()
            .match(String.class,
                message -> message.equals("printHello"),
                message -> System.out.println("Hello World!"))
            .matchAny(message -> unhandled(message))
            .build();
    }
}
```

Further Methods

- `preStart()`, `postStop()`, `preRestart()`,
`postRestart()`
- `getSelf()`: Referenz auf sich selbst
- `getContext()`: Context um weiter Actoren zu erzeugen
- `getSender()`: Sender der aktuell verarbeiteten Nachricht

Muss auf einen **Context** aufgerufen werden, entweder `ActorSystem` oder `getContext()` von innerhalb eines Actors.
Actor beaufsichtigt alle Aktoren die er erstellt hat

```
ActorSystem actorSystem = ActorSystem.create("MySystem");  
ActorRef helloWorldActor =  
actorSystem.actorOf(Props.create>HelloWorldActor.class));
```

Muss auf den Empfänger aufgerufen werden

- `tell(Object message, ActorRef sender):` asynchron, nicht blockierend
- `Future<?> Patterns.ask(ActorRef target, Object msg, Timeout timeout):` kann awaited werden, sollte man aber vermeiden

- `ActorSystem.create(String name)`: Erzeugt ein `ActorSystem`
- `void stop(ActorRef actorToStop)`: Muss auf eine `ActorRefFactory` aufgerufen werden (`ActorSystem`, `Context`)
- `PoisonPill.getInstance()`: Kann als Nachricht an einen `Actor` gesendet werden
- `ActorSystem.terminate()`: Terminiert `ActorSystem`

Klausuraufgabe SS21

Einführung in Compilerbau

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

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

Compiler

Compiler: Motivation

- Maschine(-nmodell) versteht i.d.R. eingeschränkten Instruktionssatz
- \leadsto Programme in Maschinensprache sind schwer les-/schreibbar

- Maschine(-nmodell) versteht i.d.R. eingeschränkten Instruktionssatz
- \leadsto Programme in Maschinensprache sind schwer les-/schreibbar
- 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, Scala, Kotlin \rightarrow Java-Bytecode
 - TypeScript \rightarrow JavaScript/WebAssembly

- Übersetzer für formale Sprachen nennt man *Compiler*
- Beispiele:
 - C, Haskell, Rust, Go → X86
 - Java, Scala, Kotlin → Java-Bytecode
 - TypeScript → JavaScript/WebAssembly
- 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 (Baum-)Struktur der bisher linear gelesenen Eingabe fest:
 - Blockstruktur von Programmen
 - Baumstruktur von HTML-Dateien
 - Header + Inhalt-Struktur von Mails
 - Verschachtelte arithmetische Ausdrücke
- Syntaktische Analyse ist das größte Compiler-Thema in PP.

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

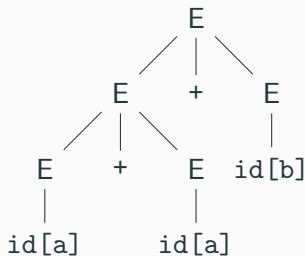
Beispiel: Arithmetische Ausdrücke

$a+a+b$

$a*c+b$

$a*c+b*d$

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

$$\begin{aligned} E &\rightarrow E + E \\ &\quad | E * E \\ &\quad | (E) \\ &\quad | id \end{aligned}$$


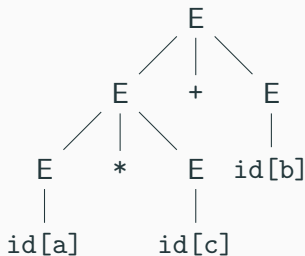
Beispiel: Arithmetische Ausdrücke

$a+a+b$

$a*c+b$

$a*c+b*d$

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

$$\begin{aligned} E &\rightarrow E + E \\ &\quad | E * E \\ &\quad | (E) \\ &\quad | id \end{aligned}$$


Beispiel: Arithmetische Ausdrücke

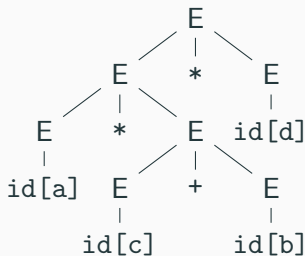
$a+a+b$

$a*c+b$

$a*c+b*d$

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

$$\begin{aligned} E &\rightarrow E + E \\ &\quad | E * E \\ &\quad | (E) \\ &\quad | id \end{aligned}$$



Beispiel: Mathematische Ausdrücke

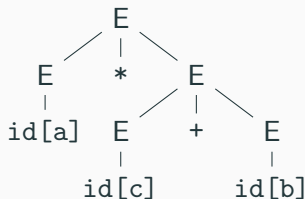
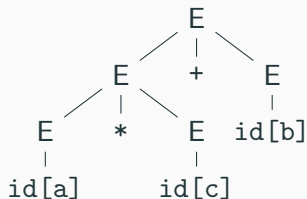
$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

a*c+b

Beispiel: Mathematische Ausdrücke

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

a*c+b



- Grammatik nicht eindeutig \leadsto schlecht
- Grammatik garantiert nicht Punkt-vor-Strich \leadsto schlecht
- Grammatik ist linksrekursiv \leadsto nicht einfach zu parsen \leadsto schlecht

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

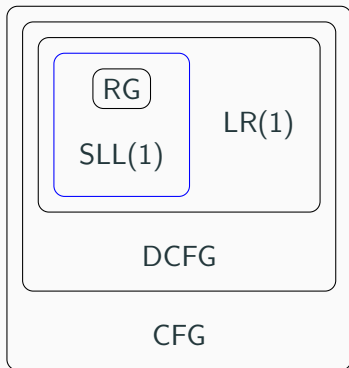
- Punkt-vor-Strich („Operatorpräzedenz“) wird von dieser naiven Grammatik nicht beachtet.
- Lösung: Ein Nichtterminal pro Präzedenzstufe:
 - *Summen von Produkten von Atomen.*
 - Herkömmliche Begriffe: Ausdruck, Term und Faktor.

$$\begin{aligned} \text{Expr} &\rightarrow \text{Expr} + \text{Term} \\ &\mid \text{Term} \end{aligned}$$

$$\begin{aligned} \text{Term} &\rightarrow \text{Term} * \text{Factor} \\ &\mid \text{Factor} \end{aligned}$$

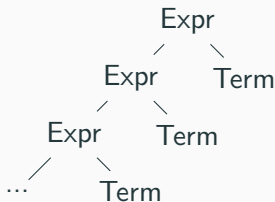
$$\text{Factor} \rightarrow (\text{Expr}) \mid \text{id}$$

Welche Art von Grammatik wollen wir denn genau?



- CFG-Parsen ist i.A. in $O(n^3)$, bspw. Earley-Algorithmus.
- Reguläre Grammatiken (\approx reg. Sprachen) sind uns nicht mächtig genug.
- **LR**: Left-to-right, Rightmost
- **LL**: Left-to-right, Leftmost
- SLL-Parsing $\in O(n)$

CFG: Context-Free Grammar/Kontextfreie Grammatik

$$\begin{aligned} \text{Expr} &\rightarrow \text{Expr} + \text{Term} \\ &\quad | \text{Term} \\ \text{Term} &\rightarrow \text{Term} * \text{Factor} \\ &\quad | \text{Factor} \\ \text{Factor} &\rightarrow (\text{Expr}) \mid \text{id} \end{aligned}$$


Problem: Die Linksableitung des Symbols *Expr* in dieser Grammatik ist eine endlose Schleife.

$$\begin{aligned} \text{Expr} &\rightarrow \text{Expr} + \text{Term} \\ &\quad | \text{Term} \\ \text{Term} &\rightarrow \text{Term} * \text{Factor} \\ &\quad | \text{Factor} \\ \text{Factor} &\rightarrow (\text{Expr}) \mid \text{id} \end{aligned}$$
$$\begin{aligned} \text{Expr} &\rightarrow \text{Term Expr}' \\ \text{Expr}' &\rightarrow + \text{Term Expr}' \\ &\quad | \epsilon \\ \text{Term} &\rightarrow \text{Term Expr}' \\ \text{Term}' &\rightarrow * \text{Factor Term}' \\ &\quad | \epsilon \\ \text{Factor} &\rightarrow (\text{Expr}) \mid \text{id} \end{aligned}$$

Lösung: Linksrekursion eliminieren, durch folgendes Umschreiben der Grammatik:

Grammar Engineering 2 — Linksrekursion eliminieren

$$\begin{aligned} \text{Expr} &\rightarrow \text{Expr} + \text{Term} \\ &\quad | \text{Term} \\ \text{Term} &\rightarrow \text{Term} * \text{Factor} \\ &\quad | \text{Factor} \\ \text{Factor} &\rightarrow (\text{Expr}) \quad | \quad \text{id} \end{aligned}$$
$$\begin{aligned} \text{Sym} &\rightarrow \text{Sym } \alpha \\ &\quad | \beta \end{aligned}$$
$$\begin{aligned} \text{Expr} &\rightarrow \text{Term Expr}' \\ \text{Expr}' &\rightarrow + \text{Term Expr}' \\ &\quad | \epsilon \\ \text{Term} &\rightarrow \text{Term Expr}' \\ \text{Term}' &\rightarrow * \text{Factor Term}' \\ &\quad | \epsilon \\ \text{Factor} &\rightarrow (\text{Expr}) \quad | \quad \text{id} \end{aligned}$$
$$\begin{aligned} \text{Sym} &\rightarrow \beta \text{Sym}' \\ \text{Sym}' &\rightarrow \alpha \text{Sym}' \\ &\quad | \epsilon \end{aligned}$$

$$E \rightarrow T \text{ EList}$$

$$\text{EList} \rightarrow + T \text{ EList}$$

$$| \epsilon$$

$$T \rightarrow T \text{ EList}$$

$$\text{TList} \rightarrow * F \text{ TList}$$

$$| \epsilon$$

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

- Grammatik ist eindeutig ✓
- Grammatik erzeugt nur korrekte Terme ✓
- Grammatik enthält keine Linksrekursion ✓

First-/Followmenge, Indizmenge

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

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

First-/Followmenge, Indizmenge

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

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

- \leadsto definiere *Indizmenge* $IM_k(A \rightarrow \alpha) = \text{First}_k(\alpha \text{Follow}_k(A))$
- Wenn nächste k Token in $IM_k(EList \rightarrow \phi) \leadsto$ weiter mit ϕ

First-/Followmenge, Indizmenge

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

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

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

First-/Followmenge, Indizmenge

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

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

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

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

- SLL(k): Bei jedem 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

Grammatik ist in SLL(k)-Form

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

- SLL(k): Bei jedem 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 $N \in \{E, T, F\}$.

$$E \rightarrow T \text{ } EList$$

$$EList \rightarrow \epsilon \mid + \text{ } T \text{ } EList \mid - \text{ } T \text{ } EList$$

$$T \rightarrow F \text{ } TList$$

$$TList \rightarrow \epsilon \mid * \text{ } F \text{ } TList \mid / \text{ } F \text{ } TList$$

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

$$E \rightarrow T \ EList$$

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

$$T \rightarrow F \ TList$$

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

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

- Was bringt uns das diese Grammatik in SLL(1)-Form ist?

Rekursive Abstiegsparser

$$E \rightarrow T \text{ EList}$$
$$\text{EList} \rightarrow \epsilon \mid + T \text{ EList} \mid - T \text{ EList}$$
$$T \rightarrow F \text{ TList}$$
$$\text{TList} \rightarrow \epsilon \mid * F \text{ TList} \mid / F \text{ TList}$$
$$F \rightarrow \text{num} \mid (E)$$

- Was bringt uns das diese Grammatik in SLL(1)-Form ist?
- G ist jetzt einfach ausprogrammierbar:
 - 1 Methode per Nichtterminal: `parseE()`, `parseEList()`, ...
 - `lexer.lex()` konsumiert das aktuelle Token
 - `lexer.current` gibt nicht konsumierenden Zugriff auf das aktuelle Token