Übungsblatt 03

A4.1: Kontextfreie Grammatik

- First-Menge
  - $FIRST(S)$:
    - Aus $S \to 1AS$ folgt Terminal 1 in $FIRST(S)$
    - Aus $S \to 3$ folgt Terminal 3 also $FIRST(S) = \{1, 3\}$
  - $FIRST(A)$:
    - Aus $A \to 2AS$ folgt 2
    - Aus $A \to \epsilon$ folgt ϵ also $FIRST(A) = \{2, \epsilon\}$
- Follow-Menge
  - Startsymbol S enthält das End-of-input-Symbol $: $ \$ \in FOLLOW(S)$
    - Vorkommen von A: in $S \to 1AS$ und in $A \to 2AS$. In beiden Fällen folgt auf A das Nichtterminal S. Deshalb gilt → $FOLLOW(A) \supseteq FIRST(S) \backslash \{\epsilon\} = \{1, 3\}$. S kann nicht ϵ erzeugen, also keine zusätzliche Folge aus der ϵ-Situation. → $FOLLOW(A) = \{1, 3\}$
    - S kommt rechts in Produktionen vor , also hat $FOLLOW(S)$ mindestens das Follow des umgebenen Nichtterminals enthält, da S Startsymbol ist: $FOLLOW(S) = \{\$\}$
- Eignung für LL(1)-Eignung
  - Für jedes Nichtterminal müssen die First-menge der produktionen disjunkt sein, falls eine Produktion ϵ enthält, muss First disjunkt ovn Follow(Nichtterminal) sein.
    - Für S: Alternativen haben First-mengen {1} bzw. {3} disjunkt
    - Für A: Alternativen haben Firsts {2} und $\{\epsilon\}$. Da ϵ in der zweiten Alternative enthalten ist, prüfen wir FIRST(2AS) = {2} gegen FOLLOW(A) = {1, 3}. Da {2} und {1, 3} disjunkt sind ist es okay.

- Die Grammatik ist LL(1)

A4.2: Grammatik

- Akzeptierte Programme

```
(+ 1 1)
(/ 10 3)
(+ 1 2 3 4)
(+ (+ 1 2) 3)
```

```
42
"hello"
true
foo
```

```
(def x 42)
(+ x 7)

(defn fac (n)
   (if (< n 2)
      1
      (* n (fac (- n 1)))))

(print (fac 5))
```

- Nicht akzeptierende Programme

```
(+ 1 2
```

```
"test String
```

```
@#$%
```

- Grammatik

```
<program>   ::= { <sexpr> }
<sexpr>     ::= <literal>
              | IDENT
              | "(" { <sexpr> } ")"

<literal>   ::= INT | STRING | BOOL
```

A4.3: Lexer

- Code

```
public enum TokenType {
    LPAREN, RPAREN,
    INT, STRING, BOOL, IDENT,
    EOF
}

public class Token {
    public final TokenType type;
    public final String lexeme;
    public final int column;

    public Token(TokenType type, String lexeme, int line, int column) {
        this.type = type;
        this.lexeme = lexeme;
        this.line = line;
        this.column = column;
    }

    public String toString() {
```

```
        return String.format("%s(%s) @%d:%d", type, lexeme, line, column);
    }
}
```

- Lexer Verhalten
  - Whitespace (Space, Tab, CR, LF) überspringen (Zeilenzähler aktualisieren).
  - Kommentar: `;;` startet Kommentar bis Zeilenende → überspringen.
  - `(` → LPAREN, `)` → RPAREN.
  - STRING: Beginnt mit `"` — akzeptiere Zeichen bis schließendem `"` und verarbeite Escape-Sequenzen `\n`, `\"`, `\\`. Wenn EOF vor `"` → lexikalischer Fehler: "ungeschlossener String" mit Position.
  - INT: Folge von Ziffern `[0-9]+`.
  - BOOL: genau `true` oder `false` → Token BOOL (alternativ als IDENT behandeln; sinnvoll ist BOOL-Token).
  - IDENT: Buchstaben/Operatorzeichen gefolgt von Buchstaben/Ziffern/Operatorzeichen (z. B. `+`, `print`, `defn`).
  - Fehlerbehandlung: Abbruch bei unbekannten Zeichen

A4.6 — Recherche & Diskussion: Open-Source-Projekte mit handgeschriebenen recursive-descent Parsern

- In A4.6 werden Beispiele realer OS-Projekte untersucht, die handgeschriebene, recursive-descent Parser verwenden. Dazu gehören unter anderem Go und der Ruff-Linter, die sich bewusst gegen Parsergeneratoren wie ANTLR entschieden haben. Hauptgründe dafür sind höhere Leistung, präzisere Fehlermeldungen und größere Flexibilität bei der Sprachentwicklung. Gerade bei Sprachen mit vielen Sonderregeln, oder besonderen Syntaxstrukturen lassen sich handgeschriebene Parser besser anpassen. Zudem ermöglichen sie eine benutzerfreundlichere Fehlerdiagnose

und bessere Kontrolle über das Parsing Verhalten. Nachteile sind jedoch ein höherer Entwicklungs- und Wartungsaufwand sowie weniger Automatisierung bei komplexen Grammatiken. Insgesamt eignen sich handgeschriebene Parser besonders für kleinere oder performancekritische Sprachen, während Parsergeneratoren bei großen und komplexen Sprachen Vorteile bieten.