Blatt 04 – A4.2 Grammatik

Modul: Compilerbau — Gruppe: Adrian Kramkowski, Abdelhadi Fares, Yousef Al Sahli, Abdelraoof Sahli

---

A4.2 – Beispielprogramme und Grammatik für die Zielsprache

1) Beispielprogramme und Fehlerkategorien

```
; ----------------------------------------
; Einfache Ausdrücke (gültige Programme)
; ----------------------------------------

42
"hello"
true
false
foo
(+ 1 2)
(/ (+ 10 2) (+ 2 4))

; ----------------------------------------
; Nutzung von print und str (gültig)
; ----------------------------------------

(print "hello world")
(print (str "one: " 1 ", two: " 2))

; ----------------------------------------
; Variablen und Funktionen (gültig)
; ----------------------------------------

(def x 42)
(+ x 7)

(defn hello (n)
  (str "hello " n))

(hello "world")
```

```
; ----------------------------------------
; let, do und if (gültig)
; ----------------------------------------

(let (x 1
      y 2)
  (+ x y 10))

(do
  (print "wuppie")
  (print "fluppie")
  (print "foo")
  (print "bar"))

(if (< 1 2)
    (do (print "true")
        (print "WUPPIE"))
    (print "false"))

; ----------------------------------------
; Listen und Rekursion: Länge einer Liste
; ----------------------------------------

(defn length (lst)
  (if (< (nth lst 0) 0)          ; Dummy-Bedingung, nur symbolisch
      0
      (+ 1 (length (tail lst))))) ; rekursiver Aufruf

(def v (list 1 2 3))
(length v)

; ----------------------------------------
; Ungültige Programme (syntaktische Fehler)
; ----------------------------------------

(1 2 3)
; Fehler: erster Eintrag in der Liste ist kein Operator/Funktionsname

(def x)
; Fehler: def erwartet genau zwei Argumente: (def name value)

(defn f n) (print n))
; Fehler: fehlende Klammern um Parameterliste und Funktionskörper
```

```
(if (< 1 2) (print "ok")
; Fehler: schließende Klammer für if-Ausdruck fehlt

(print "unclosed string)
; Fehler: String-Literal nicht korrekt abgeschlossen

; ----------------------------------------
; Mögliche Fehlerkategorien
; ----------------------------------------

1. Klammerfehler
   - Zuviele oder zu wenige schließende/öffnende Klammern
   - Beispiel: (if (< 1 2) (print "ok")

2. Falsche Anzahl von Argumenten (Arität)
   - Beispiel: (def x)          ; zu wenige Argumente
   - Beispiel: (if cond then)    ; else-Zweig fehlt (je nach Definition)

3. Falsche Form von S-Expressions
   - erster Eintrag ist kein Operator/Funktionsname:
     (1 2 3)

4. Ungültige Token
   - z.B. ungeöffnete oder ungeschlossene Strings:
     "hello

5. Typische Tippfehler bei Schlüsselwörtern
   - z.B. (prit "x") statt (print "x")
   (Diese können vom Lexer/Parser meist nur als "unbekannter Bezeichner"
    erkannt werden, nicht als spezieller Tippfehler.)
```

2) Grammatik für die Lisp-artige Zielsprache

```
; Hinweis:
; - Großgeschriebene Namen sind Terminale (Token des Lexers), z.B. INT, STRING,
; - Klein geschriebene Namen sind Nichtterminale.
; - Die Grammatik ist so gestaltet, dass sie sich gut für einen LL(1)-Parser
;   mit rekursivem Abstieg eignet (keine Linksrekursion, klare Startsymbole).

; ----------------------------------------
```

```
; Programmstruktur
; ----------------------------------------

program
  : expression_list
  ;

expression_list
  : expression expression_list
  | /* ε: leeres Programm erlaubt */
  ;

; ----------------------------------------
; Ausdrücke (S-Expressions)
; ----------------------------------------

expression
  : literal
  | IDENT                    ; Variablen- oder Funktionsname allein
  | LPAREN form RPAREN       ; S-Expression in Klammern
  ;

literal
  : INT
  | STRING
  | BOOL
  ;

; ----------------------------------------
; Formen innerhalb von (...)
; Der erste Token nach '(' entscheidet die Form.
; ----------------------------------------

form
  : IF expression expression opt_expression      ; (if cond then [else])
  | DO expression_list                           ; (do e1 e2 ...)
  | DEF IDENT expression                         ; (def name value)
  | DEFN IDENT LPAREN param_list RPAREN expression
                                                 ; (defn name (params) body)
  | LET LPAREN let_bindings RPAREN expression    ; (let (name value ...) body)
  | LIST expression_list_nonempty                ; (list e1 e2 ...)
  | NTH expression expression                    ; (nth list-expr index-expr)
  | HEAD expression                              ; (head list-expr)
```

```
  | TAIL expression                    ; (tail list-expr)
  | function_call                     ; allgemeiner Funktions-/Operatoraufruf
  ;

opt_expression
  : expression
  | /* ε: kein else-Zweig */
  ;

; ----------------------------------------
; Funktions- und Operatoraufrufe
; ----------------------------------------

function_call
  : operator expression_list_nonempty      ; z.B. (+ 1 2 3)
  | IDENT  expression_list_nonempty        ; z.B. (foo 1 2)
  ;

operator
  : PLUS        ; '+'
  | MINUS       ; '-'
  | TIMES       ; '*'
  | DIV         ; '/'
  | EQ          ; '='
  | LT          ; '<'
  | GT          ; '>'
  | PRINT       ; 'print'
  | STR         ; 'str'
  ;

; Liste mit mindestens einem Ausdruck (für Argumentlisten usw.)
expression_list_nonempty
  : expression expression_list
  ;

; ----------------------------------------
; Parameter- und let-Bindungen
; ----------------------------------------

param_list
  : IDENT param_list_tail
  | /* ε: keine Parameter */
  ;
```

```
param_list_tail
  : IDENT param_list_tail
  | /* ε */
  ;

let_bindings
  : IDENT expression let_bindings_tail
  ;

let_bindings_tail
  : IDENT expression let_bindings_tail
  | /* ε */
  ;
```

Die Grammatik trennt bewusst zwischen `expression` und den speziellen `form`-Konstrukten innerhalb von Klammern. Dadurch kann der Parser mit einem einzigen Lookahead-Token sehr gut entscheiden, welche konkrete Regel anzuwenden ist (z. B. `IF`, `DEF`, `LIST`, Operator, normaler Funktionsname usw.). Das passt zu der in der Vorlesung gezeigten Idee des rekursiven Abstiegs.