Prof. Javier Esparza
Philipp Czerner, Martin Helfrich

Technische Universität München
Lehrstuhl für Theoretische Informatik

Einführung in die Theoretische Informatik

Sommersemester 2021 – Hausaufgabenblatt 6

- Diese Woche werden alle Aufgaben korrigiert.
- Wenn Sie einen Beweis aufstellen, von dem Sie wissen, das einzelne Schritte problematisch oder unvollständig sind, merken Sie dies bitte in Ihrer Lösung an, damit wir das bei der Korrektur positiv berücksichtigen können.

Aufgabe H6.1. (*Xuby*) 1+1+1+1 Punkte

Sei $\Sigma := \{a, b\}$, und sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik mit den Produktionen

$$\begin{aligned} S &\to XUb \mid Y & Y &\to ab \mid YY \\ X &\to UbY \mid UU & U &\to aU \mid \varepsilon \end{aligned}$$

Konvertieren Sie G mit dem aus der Vorlesung bekannten Verfahren schrittweise zur Chomsky-Normalform, indem Sie G so anpassen, dass für jede Produktion $\alpha \to \beta$ folgende zusätzlichen Einschränkungen gelten:

(a) $\beta \in \Sigma \cup V^*$ (b) $|\beta| \leq 2$ (c) $\beta \neq \varepsilon$ (d) $\beta \notin V$

Sie sollen also eine Grammatik G' angeben, die (a) und $L(G') = L(G)$ erfüllt, eine Grammatik G'', die (a), (b) und $L(G'') = L(G)$ erfüllt, usw.

Lösungsskizze.

(a) Wir fügen $A \to a$ und $B \to b$ hinzu, und ersetzten a und b in den anderen Produktionen jeweils durch A und B.

$$\begin{aligned} S &\to XUB \mid Y & U &\to AU \mid \varepsilon \\ X &\to UBY \mid UU & A &\to a \\ Y &\to AB \mid YY & B &\to b \end{aligned}$$

(b) Wir passen jeweils die erste Produktion von S und von X an.

$$\begin{aligned} S &\to XS_1 \mid Y & Y &\to AB \mid YY \\ S_1 &\to UB & U &\to AU \mid \varepsilon \\ X &\to UX_1 \mid UU & A &\to a \\ X_1 &\to BY & B &\to b \end{aligned}$$

(c) Zuerst betrachten wir $U \to \varepsilon$. Damit können wir folgende Ableitungen erzeugen:

$$\begin{aligned} S_1 &\to B \\ X &\to U \\ X &\to \varepsilon \\ X &\to X_1 \\ U &\to A \end{aligned}$$

Wir erhalten also insbesondere $X \to \varepsilon$, hiermit erzeugen wir

$$S \to S_1$$

Nach Entfernen der ε-Transitionen ergibt sich also

$$\begin{aligned} S &\to XS_1 \mid Y \mid S_1 & Y &\to AB \mid YY \\ S_1 &\to UB \mid B & U &\to AU \mid A \\ X &\to UX_1 \mid UU \mid U \mid X_1 & A &\to a \\ X_1 &\to BY & B &\to b \end{aligned}$$

(d) Nun entfernen wir die Kettenproduktionen. Aus den Kettenproduktionen $S \to Y$, $S \to S_1$, $S_1 \to B$, $X \to U$, $X \to X_1$ und $U \to A$ erhalten wir

$$\begin{aligned} S &\to AB \mid YY \mid UB \mid B \\ S_1 &\to b \\ X &\to AU \mid A \mid BY \\ U &\to a \end{aligned}$$

Des weiteren müssen wir $S \to S_1$ und $X \to U$ erneut anwenden (da S_1 und U zusätzliche Produktionen haben), und die neuen Kettenproduktionen $S \to B$ und $X \to A$ berücksichtigen. Letztere sind redundant und nur der Anschaulichkeit halber gezeigt – es ist nicht möglich, eine Produktion mehrfach zu besitzen.

$$\begin{aligned} X &\to a \mid a \\ S &\to b \mid b \end{aligned}$$

Schließlich entfernen wir die nun unnötigen Kettenproduktionen.

$$\begin{aligned} S &\to XS_1 \mid AB \mid YY \mid UB \mid b & Y &\to AB \mid YY \\ S_1 &\to UB \mid b & U &\to AU \mid a \\ X &\to UX_1 \mid UU \mid AU \mid BY \mid a & A &\to a \\ X_1 &\to BY & B &\to b \end{aligned}$$

Aufgabe H6.2. (*Matririrkelnununummern*) 2+2 Punkte

Eines Morgens erhalten Sie einen panischen Anruf von einem alten Bekannten[1], dem Rektor der Technischen Hochschule Estlingen-Oberfeld. Nachdem eine Horde wütender Studierender das Verwaltungsgebäude stürmte und die Wiedereinführung der traditionellen Estlinger Matrikelnummern verlangte, hat sich der Rektor nun im Keller verbarrikadiert und sucht verzweifelt nach Lösungen, um diese automatisch verarbeiten zu können. In einem alten Kabinett hat er einen Bausatz für Automaten gefunden, mit denen man anscheinend beliebige kontextfreie Sprachen akzeptieren kann. Sind diese „Kellerautomaten“ vielleicht die Lösung?

(a) Sei $\Sigma := \{0, ..., 9\}$. Zeigen Sie, dass die Sprache $L := \{\#\}\{w \in \Sigma^* : \sum_{i \geq 1} w_i \text{ prim}\}$ nicht kontextfrei ist.

[1] Siehe H3.2

Nachdem sich seine letzte Hoffnung in Luft aufgelöst hat, beschließt der Rektor, durch einen der vielen unterirdischen Tunnel zu fliehen. Diese werden leider von der Bergbaufakultät kontrolliert, und können nur mit bestimmten Klopfzeichen betreten werden, die einem komplizierten System folgen, von dem der Rektor schlicht überfordert ist. Aber vielleicht kann er sich ja einen Kellerautomaten dafür bauen...

(b) Zeigen oder widerlegen Sie: $L := \{a^i b^j c^k : i, j, k \in \mathbb{N} \text{ und } j^2 \leq ik\}$ ist kontextfrei.

Lösungsskizze. **(a)** Wir wenden das Pumping-Lemma für kontextfreie Sprachen an. Sei also $n \in \mathbb{N}$ beliebig. Wir wählen das Wort $z := \#1^p$, wobei $p \geq n$ eine Primzahl ist. Nun erhalten wir eine Zerlegung $z = uvwxy$. Falls $\# \in v$ oder $\# \in x$ gilt, ist $uv^0wx^0y = uwy \notin L$, da uwy nicht mit einer Raute beginnt. Ansonsten enthalten v und x nur das Zeichen 1, somit gilt $vx = 1^k$ für $k := |vx|$. Sei nun $i := 1 + p$. Wir erhalten

$$uv^iwx^iy = \#1^{p+kp}$$

Da aber $p + kp = p(k + 1)$ und $k > 0$ nach den Eigenschaften der Zerlegung gilt, besitzt $p(k + 1)$ mindestens zwei Teiler (p und $k + 1$), ist keine Primzahl, und $uv^iwx^iy \notin L$.

In beiden Fällen haben wir ein i gefunden, sodass $uv^iwx^iy \notin L$. Damit gilt die Eigenschaft des PL nicht, und L kann nicht kontextfrei sein.

(b) L ist nicht kontextfrei. Wir wenden wieder das Pumping-Lemma für kontextfreie Sprachen an und fixieren ein beliebiges n. Als Wort wählen wir $w := a^n b^n c^n$. Da $n^2 = nn$ gilt $w \in L$. Nun erhalten wir eine Zerlegung $w = uvwxy$. Da $|vwx| \leq n$, gilt entweder $a \notin vx$ oder $c \notin vx$. Wir betrachten den Fall $c \notin vx$, der Fall $a \notin vx$ lässt sich dann analog beweisen. Nun betrachten wir folgende Teilfälle:

1. $ab \in v$ oder $ab \in x$: Diese Zerlegung erlaubt es uns, die Struktur des Wortes zu zerstören. Es gilt insbesondere $uv^2wx^2y \notin L(a^*b^*c^*)$ und somit $uv^2wx^2y \notin L$.
2. $b \notin vx$: Aufpumpen würde einfach die Anzahl a erhöhen, aber durch abpumpen kommen wir unter die Schranke. Sei also $z' := uv^0wx^0y$. Somit gilt $|z'|_a = n - |vx| < n$ und $|z'|_b = |z'|_c = n$. Da aber $n^2 > n(n - |vx|)$, ergibt sich $z' \notin L$.
3. $a \notin vx$: Dieser Fall ist ähnlich zum vorherigen, nur dass Aufpumpen die Anzahl b erhöht. Sei $z' := uv^2wx^2y$. Somit gilt $|z'|_b = n + |vx| > n$ und $|z'|_a = |z'|_c = n$. Da aber $(n + |vx|)^2 > n \cdot n$, ergibt sich $z' \notin L$.
4. $v = a^r$, $x = b^s$ mit $r, s \geq 1$: Nun erhöht Aufpumpen sowohl die Anzahl an a als auch die Anzahl b. Letztere zählt jedoch quadratisch, wir müssen also nur oft genug pumpen. Sei $i := 1 + rn$ und $z' := uv^iwx^iy$. Wir erhalten $|z'|_a = n + r(rn)$, $|z'|_b = n + s(rn)$ und $|z'|_c = n$. Es gilt

   $$\big(|z'|_b\big)^2 - |z'|_a|z'|_c = (n + srn)^2 - (n + r^2n)n = n^2\big((1 + sr)^2 - (1 + r^2)\big)$$

   Nun verwenden wir $r, s \geq 1$ und erhalten

   $$(1 + sr)^2 - (1 + r^2) = 1 + 2sr + s^2r^2 - 1 - r^2 = 2sr + (s^2 - 1)r^2 > 0$$

   Damit gilt $\big(|z'|_b\big)^2 > |z'|_a|z'|_c$, und $z' \notin L$.

Aufgabe H6.3. ((()((()))))))) 3+1 Punkte

S-Expressions sind eine Notation für verschachtelte Listen. Für unsere Zwecke verwenden wir das Alphabet $\Sigma := \{a, ..., z, 0, ..., 9\}$, und ein Atom ist ein Wort in Σ^*. Eine *S-Expression* ist entweder ein Atom, oder eine Liste $(w_1\ w_2\ ...\ w_n)$, wobei $w_1, ..., w_n$ S-Expressions sind. Beispiele:

```
theo
(theo 2021)
((1) (2 3) (4 5) () (6))
```

Wir definieren uns nun eine Programmiersprache, die wir theoLISP nennen, aus folgenden Komponenten.

- Arithmetische Operationen:

  `(add` x y`)` $\to x + y$
  `(sub` x y`)` $\to x - y$
  `(mul` x y`)` $\to xy$

- Bedingungen und Schleifen:

  `(if` x y z`)` Wenn $x > 0$, wird y ausgeführt, sonst z
  `(while` x y`)` Solange $x > 0$, wird y ausgeführt.

- Variablenzuweisungen:

  `(set` x y`)` Setzt die Variable mit Namen x auf den Wert von y

Ein Programm ist eine Liste an Befehlen, die Ausgabe eines Programms ist der Wert der Variablen `result`. Alle Variablen sind mit 0 initialisiert, und können Werte in $\mathbb{Z}$ annehmen.

Beispielprogramme mit Ihren Ausgaben:

```
((set x 42)
 (set result (sub x 2021)))
```

$\to -1979$

```
((set x 1)
 (while (sub 16 i) (
     (set x (mul x 2))
     (set i (add i 1))))
 (set result x))
```

$\to 65536$

```
((set a 247) (set b 299)
 (while a (
     (set a (sub b a))
     (if (sub a b) (
         (set t a) (set a b) (set b t))
         () )))
 (set result b))
```

$\to 13$

Folgende sechs Programme sind nicht gültig:

```
((print 42))
((set 5 7))
(set result 42)
((if (while x ()) () ()))
((set result (if x 16 42)))
((if ((add 20 21)) () ()))
```

Insbesondere erwartet z.B. `set` als erstes Argument einen Variablennamen, keine Zahl; und `if` als erstes Argument eine Variable, Zahl, oder arithmetische Operation, und keine Anweisung oder Liste an Anweisungen.

(a) Konstruieren Sie eine kontextfreie Grammatik, die genau die theoLISP Programme erzeugt. Es genügt, wenn Sie nur die Programme erzeugen, die keine unnötigen Leerzeichen oder Zeilenumbrüche enthalten. Das erste Beispielprogramm wäre also

```
((set x 42) (set result (sub x 2021))
```

Sie können Leerzeichen mit `.` notieren.

(b) Geben Sie einen Syntaxbaum Ihrer Grammatik für folgendes Programm an:

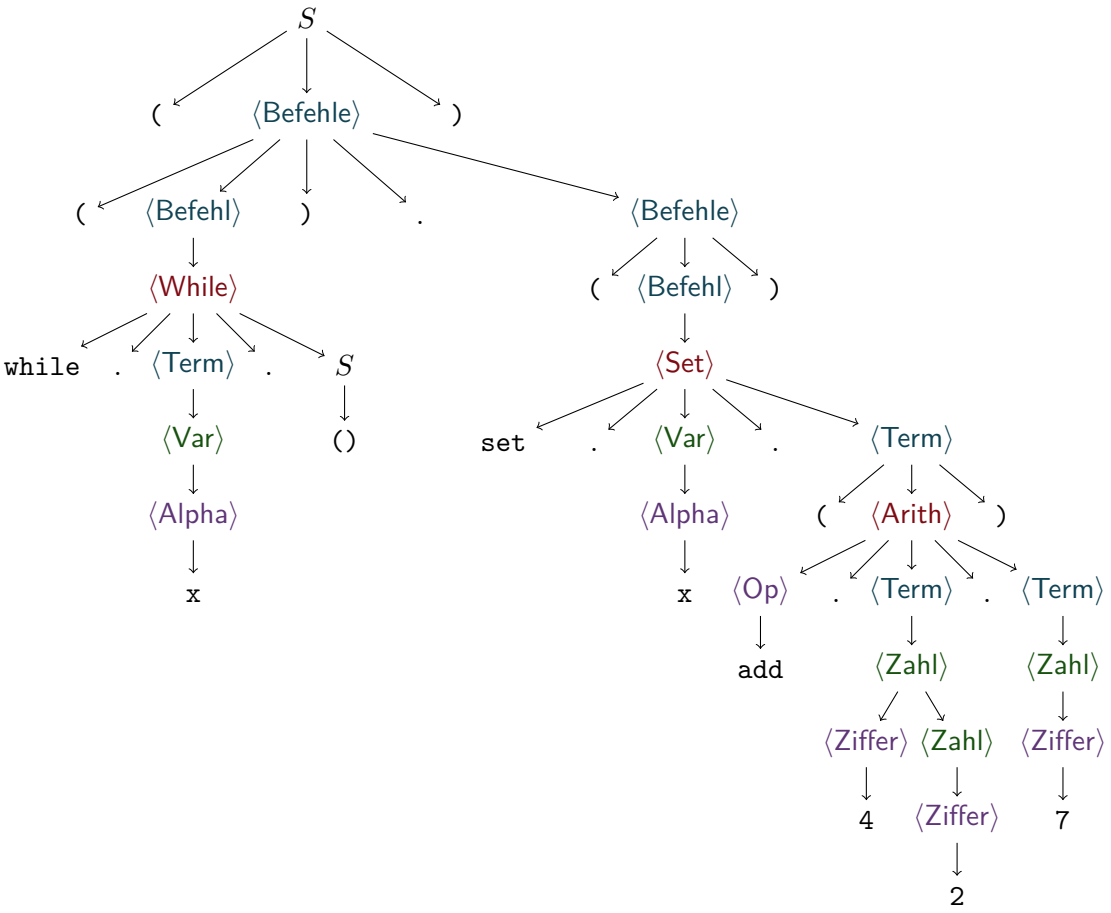
```
((while x ()) (set x (add 42 7)))
```

Lösungsskizze.

(a)

$$
\begin{aligned}
S &\to \texttt{(}\ \langle\text{Befehle}\rangle\ \texttt{)} \mid \texttt{(}\ \texttt{)} \\
\langle\text{Befehle}\rangle &\to \texttt{(}\ \langle\text{Befehl}\rangle\ \texttt{)}\ \texttt{.}\ \langle\text{Befehle}\rangle \mid \texttt{(}\ \langle\text{Befehl}\rangle\ \texttt{)} \\
\langle\text{Befehl}\rangle &\to \langle\text{If}\rangle \mid \langle\text{While}\rangle \mid \langle\text{Set}\rangle \\
\langle\text{Term}\rangle &\to \texttt{(}\ \langle\text{Arith}\rangle\ \texttt{)} \mid \langle\text{Zahl}\rangle \mid \langle\text{Var}\rangle \\
\langle\text{Arith}\rangle &\to \langle\text{Op}\rangle\ \texttt{.}\ \langle\text{Term}\rangle\ \texttt{.}\ \langle\text{Term}\rangle \\
\langle\text{If}\rangle &\to \texttt{if}\ \texttt{.}\ \langle\text{Term}\rangle\ \texttt{.}\ S\ \texttt{.}\ S \\
\langle\text{While}\rangle &\to \texttt{while}\ \texttt{.}\ \langle\text{Term}\rangle\ \texttt{.}\ S \\
\langle\text{Set}\rangle &\to \texttt{set}\ \texttt{.}\ \langle\text{Var}\rangle\ \texttt{.}\ \langle\text{Term}\rangle \\
\langle\text{Var}\rangle &\to \langle\text{Alpha}\rangle\langle\text{Var}\rangle \mid \langle\text{Alpha}\rangle \\
\langle\text{Zahl}\rangle &\to \langle\text{Ziffer}\rangle\langle\text{Zahl}\rangle \mid \langle\text{Ziffer}\rangle \\
\langle\text{Op}\rangle &\to \texttt{add} \mid \texttt{mul} \mid \texttt{sub} \\
\langle\text{Alpha}\rangle &\to \texttt{a} \mid \ldots \mid \texttt{z} \\
\langle\text{Ziffer}\rangle &\to \texttt{0} \mid \ldots \mid \texttt{9}
\end{aligned}
$$

(b)


S
(⟨Befehle⟩)
(⟨Befehl⟩) .
⟨Befehle⟩
⟨While⟩
(⟨Befehl⟩)
while . ⟨Term⟩ . S
⟨Set⟩
⟨Var⟩
()
set . ⟨Var⟩ . ⟨Term⟩
⟨Alpha⟩
⟨Alpha⟩
(⟨Arith⟩)
x
x
⟨Op⟩ . ⟨Term⟩ . ⟨Term⟩
add
⟨Zahl⟩
⟨Zahl⟩
⟨Ziffer⟩ ⟨Zahl⟩
⟨Ziffer⟩
4
⟨Ziffer⟩
7
2

Bonusaufgabe H6.4. (*S-Parser*) 1+1+1 Bonuspunkte

Bei Ausgrabungen im antiken Rom findet der Archäologe Jasper Vazarie eine kuriose Marmortafel mit folgender Beschriftung:

```
((set n 339) (while (sub 1 result) ((set nn (sub n 1)) (while (
sub 1 flag) ((set dd 0) (while (sub nn 1) ((set dd (add dd 1)) (
set nn (sub nn 2)))) (if nn ((set flag 1)) ((set nn dd) (set d dd
) (set r (add r 1)))))) (set k 4) (while k ((set a (mul (add a 1)
89)) (while ((sub a 112)) ((set a (sub a 113)))) (set x 1) (set
set dd d) (while dd ((set x (mul x a)) (while (sub x (sub n 1)) (
(set x (sub x n)))) (set dd (sub dd 1)))) (set xx (mul (sub x 1)
(sub x (sub n 1)))) (set flag 0) (if (mul xx xx) ((set rr (sub r
1)) (while rr ((set x (mul x x)) (while (sub x (sub n 1)) ((set x
(sub x n)))) (if (sub (sub n 1) x) () ((set flag 1) (set rr 0)))
(set rr (sub rr 1))))) ((set flag 1))) (if (sub 1 flag) ((set k 0
)) ()) (set k (sub k 1)))) (if (add k 1) ((set result n)) ((set n
(add n 2))))))))
```

Jasper vermutet, dass es sich hierbei um ein theoLISP Programm handelt, da dies eine beliebte Programmiersprache in der damaligen hispanischen Provinz Gallaecia war. Die Tafel stammt aus dem Jahre 339 n. Chr. und wurde, so die Überlieferung, dazu verwendet, die Geburt des nächsten großen Kaisers vorherzusagen. Leider litt auch die Marmortafel unter dem Zahn der Zeit und es befinden sich einige Fehler im Programm.

Helfen Sie Jasper, und korrigieren das Programm! Implementieren Sie hierzu einen Recogniser, der entscheidet, ob ein Wort ein syntaktisch korrektes theoLISP Programm ist, wie in Aufgabe H6.3 definiert. (Sie dürfen annehmen, dass das Programm keine unnötigen Leerzeichen und Zeilenumbrüche enthält.)

Hinweise: Verwenden Sie zur Lösung dieser Aufgabe einen Computer. Beschreiben Sie bitte Ihren Ansatz in *natürlicher Sprache* und illustrieren die wesentlichen Schritte Ihrer Lösung mit geeigneten Codefragmenten. Sie können (müssen aber nicht), Ihrer Lösung Ihren vollständigen Programmcode beifügen, jedoch steht es den Korrektoren frei, diesen zu ignorieren. Beschränken Sie sich in Ihrer Implementierung auf grundlegende Funktionalitäten verbreiteter Programmiersprachen. Insbesondere ist die Verwendung von Library-Funktionen für reguläre Ausdrücke oder andere Parser-Generatoren nicht gestattet.

Obiges Programm hat genau drei Fehler. Um Ihr Programm zu implementieren, mag es sinnvoll sein, Ihre Lösung zu Aufgabe H6.3(a) zu verwenden, und für jede Variable V eine Funktion zu schreiben, die (rekursiv) ein maximales Präfix aus $L_G(V)$ einliest. Zum Finden der Syntaxfehler bietet es sich an, dass Ihr Recogniser zusätzliche Informationen ausgibt, wenn er einen Fehler feststellt.

Die Beispielprogramme auf diesem Blatt finden Sie unter diesem Link. Im Ordner `nospace` finden Sie jeweils Versionen ohne unnötige Leerzeichen und Zeilenumbrüche. Die syntaktisch korrekten Beispiele beginnen mit einem `a`, die inkorrekten Beispiele mit einem `b`, und obiges Programm heißt `c1`.

Lösungsskizze. (Der Code lässt sich unter diesem Link herunterladen.) Zuerst lesen wir das Programm ein und entfernen unnötige Leerzeichen und Zeilenumbrüche.

```
import sys
```

```
with open(sys.argv[1], 'r') as f: code_raw = f.read()

code = ''; last = None
for c in code_raw:
    if c.isspace(): c = ' '

    if   last == '(' and c == ' ': continue
    elif last == ' ' and c == ' ': continue
    elif last == ' ' and c == ')':
        code = code[:-1] + c
        last = c
    else:
        code += c
        last = c
while code[-1] == ' ': code = code[:-1]
```

(Dies ist nicht notwendig.)

Die Variable `code` enthält den Code, den wir noch einlesen müssen. Am Anfang ist dies dementsprechend das gesamte Programm. Für das eigentliche Parsen implementieren wir zwei Hilfsfunktionen, `eat` und `eat_maybe`. Die Idee ist, dass z.B. `eat('while')` die Zeichenkette `while` vom Anfang des verbleibenden Programms einliest, und einen Fehler ausgibt, falls dies nicht möglich ist. Die Prozedur `eat_maybe` ist ähnlich, aber, statt einen Fehler zu werfen, gibt sie zurück, ob die Operation erfolgreich war.

```
def eat_maybe(s):
    global code
    if code.startswith(s):
        code = code[len(s):]
        return True
    else:
        return False

def eat(s): assert eat_maybe(s)
```

Nun lassen sich die Produktionen der Grammatik ohne viel Aufwand umsetzen.

$S \to$ `(` ⟨Befehle⟩ `)` | `(` `)`

```
def S():
    eat('(')
    if not eat_maybe(')'):
        Befehle()
        eat(')')
```

Für S gibt es zwei mögliche Produktionen, die beider mit `(` beginnen. Also lesen wir die öffnende Klammer zuerst ein, und entscheiden danach, welche der Produktionen wir ausführen. Falls direkt danach eine schließende Klammer kommt, wird die mit `eat_maybe(')')` eingelesen und wir sind fertig, ansonsten lesen wir ⟨Befehle⟩ und danach die schließende Klammer ein. Die anderen Produktionen werden auf ähnliche Art implementiert.

⟨Befehle⟩ → `(` ⟨Befehl⟩ `)` `.` ⟨Befehle⟩ | `(` ⟨Befehl⟩ `)`

```
def Befehle():
    eat('('); Befehl(); eat(')')
    if eat_maybe(' '): Befehle()
```

Interessant ist lediglich, wie wir entscheiden, welche Produktion wir ausführen. Hierfür reicht es allerdings immer, sich das nächste Zeichen anzuschauen. Bei ⟨Befehl⟩, ⟨Term⟩ und ⟨Op⟩ wird dies besonders deutlich:

⟨Befehl⟩ → ⟨If⟩ | ⟨While⟩ | ⟨Set⟩

```
def Befehl():
    if   code[0] == 'i': If()
    elif code[0] == 'w': While()
    elif code[0] == 's': Set()
    else: assert False
```

⟨Term⟩ → `(` ⟨Arith⟩ `)` | ⟨Zahl⟩ | ⟨Var⟩

```
def Term():
    if eat_maybe('('): Arith(); eat(')')
    elif code[0].isdigit(): Zahl()
    elif code[0].isalpha(): Var()
    else: assert False
```

⟨Op⟩ → `add` | `mul` | `sub`

```
def Op():
    if   code[0] == 'a': eat('add')
    elif code[0] == 'm': eat('mul')
    elif code[0] == 's': eat('sub')
    else: assert False
```

Die Nichtterminale ⟨Arith⟩, ⟨If⟩, ⟨While⟩ und ⟨Set⟩ sind leicht zu implementieren, da es hierfür nur jeweils eine Produktion gibt.

⟨Arith⟩ → ⟨Op⟩ `.` ⟨Term⟩ `.` ⟨Term⟩
⟨If⟩ → `if` `.` ⟨Term⟩ `.` S `.` S
⟨While⟩ → `while` `.` ⟨Term⟩ `.` S
⟨Set⟩ → `set` `.` ⟨Var⟩ `.` ⟨Term⟩

```
def Arith(): Op(); eat(' '); Term(); eat(' '); Term()
def If():    eat('if ');     Term(); eat(' '); S(); eat(' '); S()
def While(): eat('while '); Term(); eat(' '); S()
def Set():   eat('set ');    Var();  eat(' '); Term()
```

(Beachten Sie, dass `eat('if ')` nach dem `if` auch ein Leerzeichen liest.) Schließlich verbleiben noch ⟨Var⟩ und ⟨Zahl⟩, wobei wir hier leicht von der Grammatik abweichen,

und diese direkt implementieren.

$$\langle \text{Var} \rangle \to \langle \text{Alpha} \rangle \langle \text{Var} \rangle \mid \langle \text{Alpha} \rangle$$
$$\langle \text{Zahl} \rangle \to \langle \text{Ziffer} \rangle \langle \text{Zahl} \rangle \mid \langle \text{Ziffer} \rangle$$
$$\langle \text{Alpha} \rangle \to \texttt{a} \mid \ldots \mid \texttt{z}$$
$$\langle \text{Ziffer} \rangle \to \texttt{0} \mid \ldots \mid \texttt{9}$$

```
def Var():
    assert code[0].isalpha()
    while  code[0].isalpha(): eat(code[0])
def Zahl():
    assert code[0].isdigit()
    while  code[0].isdigit(): eat(code[0])
```

Damit ist die Grammatik implementiert. Um einen Recogniser zu erhalten, müssen wir nun `S()` aufrufen und danach sicherstellen, dass auch tatsächlich der gesamte Code eingelesen wurde.

```
try:
    S()
    assert len(code) == 0
    print('Correct')
except:
    print('Incorrect')
    sys.exit(1)
```

Nun haben wir bereits einen funktionierenden Recogniser. Damit es leichter wird, die Fehler zu finden, erweitern wir unseren Recogniser, sodass er ein maximales Präfix angibt, das zu einem korrekten Programm erweitert werden kann. Beispiel:

```
((set result (if x 16 42)))
```

Die Zeichen bis zum `if` können zu einem korrekten Programm erweitert werden, etwa

```
((set result (add 1 1)))
```

Unseren Recogniser so zu erweitern ist leicht – wir speichern die Zeichen, die wir bereits eingelesen haben, und geben diese dann aus. Hierzu passen wir die `eat_maybe` Prozedur an:

```
parsed = ''
def eat_maybe(s):
    global code
    global parsed
    if code.startswith(s):
        parsed += s
        code = code[len(s):]
        return True
    else:
        return False
```

Wenn es einen Fehler gibt, können wir nun den Inhalt von `parsed` ausgeben.

```
try:
    S()
    assert len(code) == 0
    print('Correct')
except:
    print('Incorrect')
    print(parsed)
    sys.exit(1)
```

Nun lassen sich die Fehler leicht finden und beheben.

```
...
89)) (while ((sub a 112)) ((set a (sub a 113)))) (set x 1) (set
set dd d) (while dd ((set x (mul x a)) (while (sub x (sub n 1)) (
...
(add n 2)))))))
```

- In Zeile 5 müssen bei `((sub a 112))` die äußeren Klammern entfernt werden
- Am Ende von Zeile 5 beginnt `(set set dd d)` – hier sollte nur ein `set` sein
- Am Ende des Programms befindet sich eine schließende Klammer zu viel