Aufnahme der Hybrid-Vorlesung

- **Hybrid-Vorlesung mit Aufnahme**
  - Die Aufnahme ist anschließend in Stud.IP verfügbar
  - Nutzen Sie die Gelegenheit zur Live-Veranstaltung!
- Wir nehmen auf
  - Folien, Dozent, Live-Audio sowie BBB-Audio
  - Ihre Stimme beim Fragen und Sprechen
  - **Durch aktive Teilnahme erklären Sie sich einverstanden!**
- Fragen: Live, im Chat, Sprechen in der BBB-Sitzung

Technische Universität Braunschweig

Institute of Operating Systems and Computer Networks
Reliable System Software

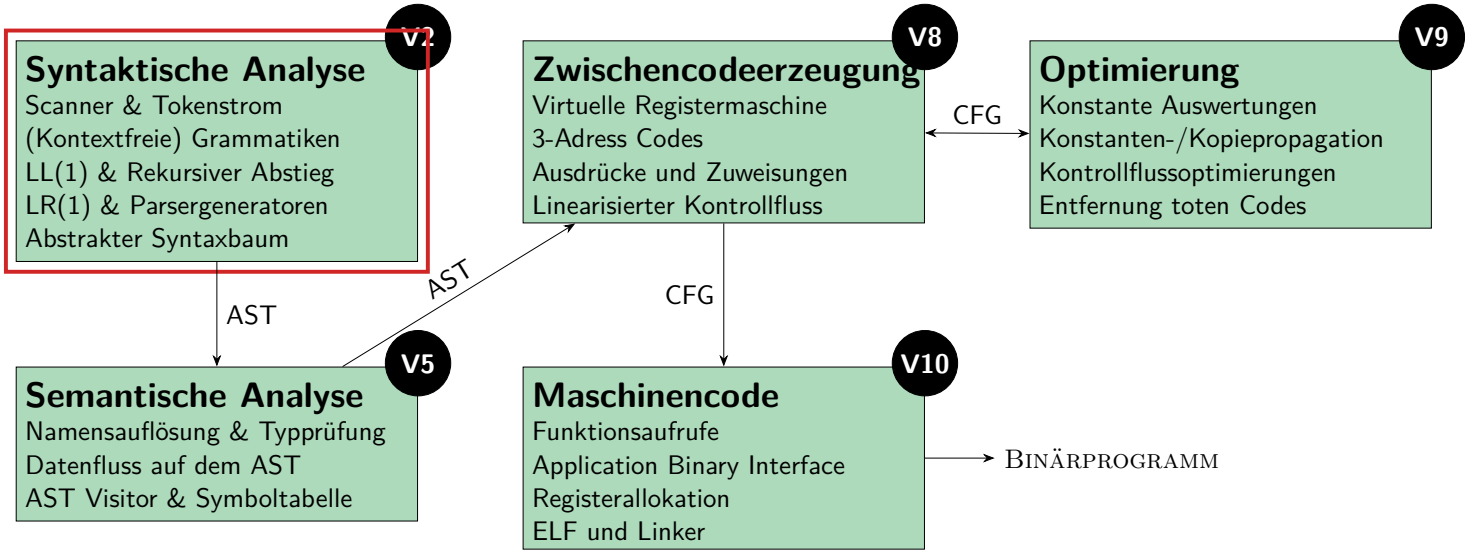
Programmiersprachen und Übersetzer

02 - Syntaktische Analyse

Christian Dietrich

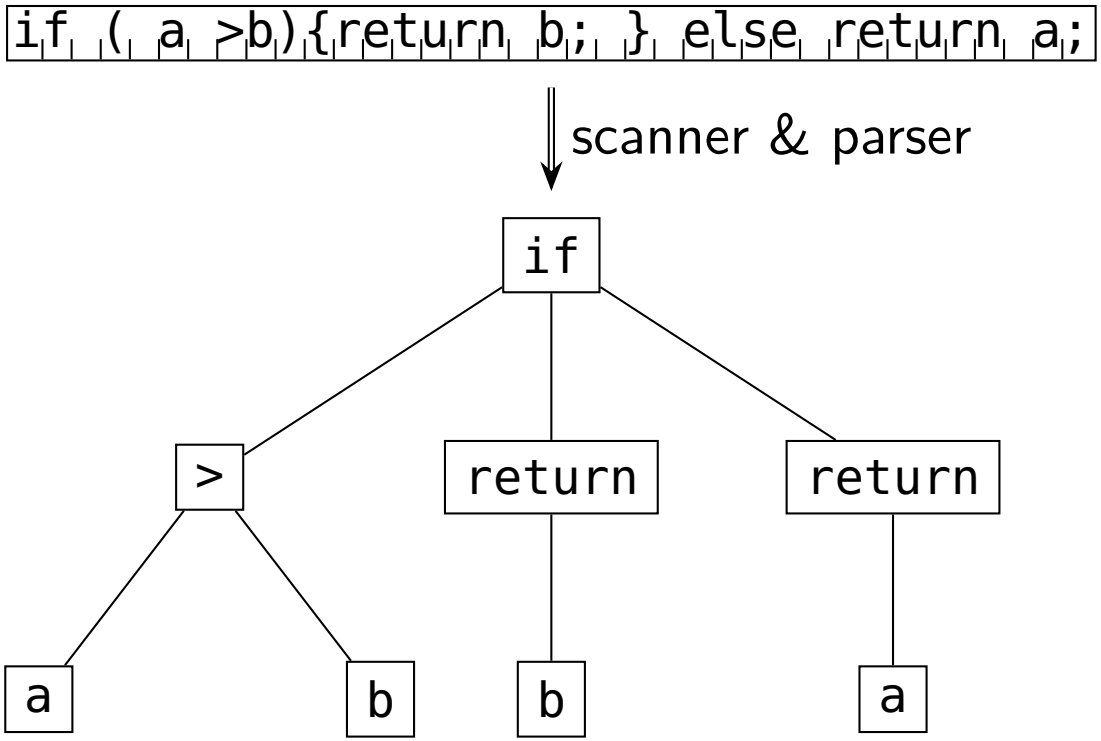
Sommersemester 2025

Einordnung in die Vorlesung: Syntaktische Analyse



- Die Syntaktische Analyse ist der erste Schritt in einem Übersetzer.
- Was sollte der *effektive* und *effiziente* Informatiker darüber Wissen?
  - Die Syntax ist die nur die Schreibweise, wie man die Konzepte notiert.
  - Rekursiv geschachtelte Elemente und Bäume passen zueinander.
  - Es gibt gute Formalismen und mächtige Werkzeuge zur Syntaktischen Analyse.

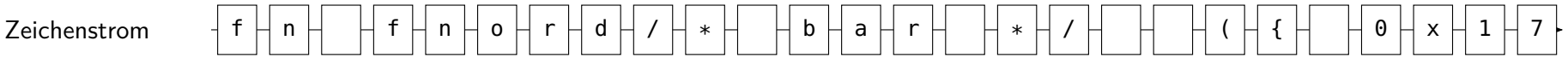
Was ist das Ziel der Syntaktischen Analyse?



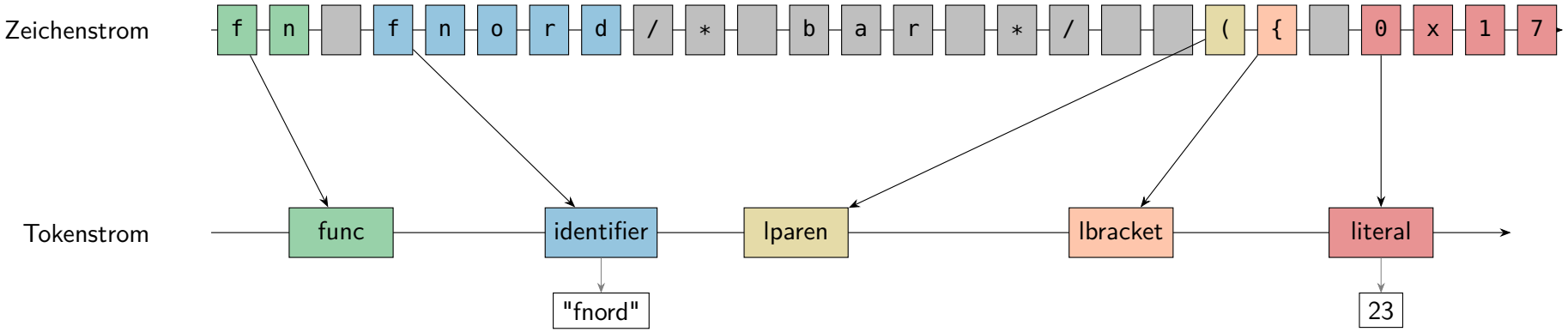
- Überprüfung der Syntaxregeln und Extraktion der Programmstruktur
  - Syntaktische Korrektheit ist ein Teil der Sprachregeln.
  - Übersetzer für Programmiersprachen müssen Zeichenketten verarbeiten.
  - Im Weiteren benötigen wir die Programmstruktur als abstrakten Syntaxbaum.

Scanner und Tokenstrom

Tokenstrom

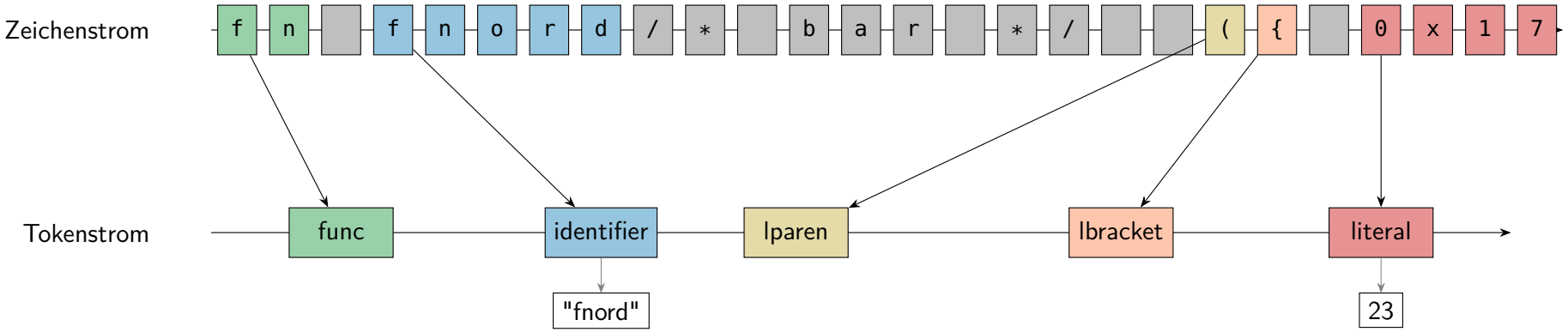


Tokenstrom



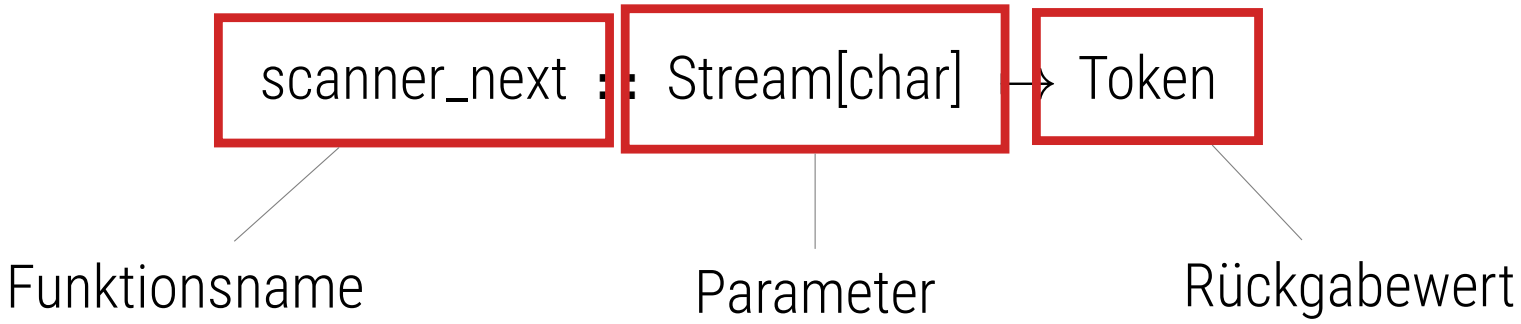
- Gruppierung von Zeichen zu sprachspezifischen Bausteinen (Token)
  - Token haben eine *Klasse* (func) und ggf. eine *Nutzlast* (`"fnord"`, 23).
  - Der Tokenstrom enthält keine Leerzeichen oder Kommentare.
  - Unterschiedliche Schreibweisen (0x17, 23, 027, 0b10111) sind vereinheitlicht.
  - Gemeinsame Präfixe werden zum längsten Match aufgelöst (`fn` vs. `fnord`).

Tokenstrom



- Gruppierung von Zeichen zu sprachspezifischen Bausteinen (Token)
  - Token haben eine *Klasse* (func) und ggf. eine *Nutzlast* (`"fnord"`, 23).
  - Der Tokenstrom enthält keine Leerzeichen oder Kommentare.
  - Unterschiedliche Schreibweisen (0x17, 23, 027, 0b10111) sind vereinheitlicht.
  - Gemeinsame Präfixe werden zum längsten Match aufgelöst (`fn` vs. `fnord`).
- Ziele der Abstraktion vom einzelnen Zeichen zum Token
  - Komplexitätsreduktion: Es gibt immer weniger Token als Zeichen.
  - Fokussierung auf die Sprachelemente, nicht auf ihre Schreibweise.
  - Vereinfachung des Parsens durch weniger Sonderfälle (Kommentare).

Scanner (oder Lexer)



Konvention: Wir verwenden Funktionssignaturen aus Haskell

Scanner (oder Lexer)

$$\texttt{scanner\_next} :: \text{Stream[char]} \mapsto \text{Token}$$

- Der Scanner (auch Abtaster oder Lexer) erzeugt den Tokenstrom.
  - `scanner_next` konsumiert Zeichen vom Anfang bis zum Ende eines Tokens.
  - Implementierung: manuell oder mittels regulärer Ausdrücke und Tooling

Scanner (oder Lexer)

$$\texttt{scanner\_next} :: \text{Stream[char]} \mapsto \text{Token}$$

- Der Scanner (auch Abtaster oder Lexer) erzeugt den Tokenstrom.
  - `scanner_next` konsumiert Zeichen vom Anfang bis zum Ende eines Tokens.
  - Implementierung: manuell oder mittels regulärer Ausdrücke und Tooling

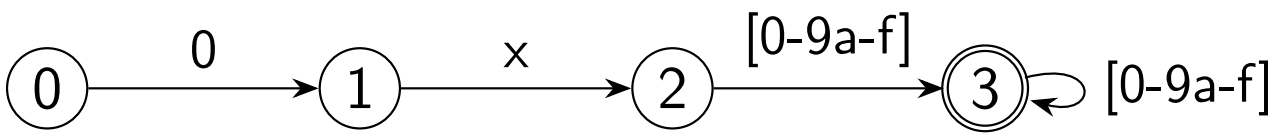
```Py
def scanner_next(stream):
    # Leerzeichen vom Anfang wegwerfen
    while stream.peek() in " \t":
        stream.read()

    ch = stream.read()
    if ch == '0': # literal: hex, oktal, oder binär
        ch = stream.read()
        if ch == 'x':
            hexdigits = "0123456789abcdefABCDEF"
            chars = stream.read_many(hexdigits)
            value = int(chars, base=16)
            return ["literal", value]
        # ....
        else:
            raise ScannerError("invalid character")
```

Wiederholung: Reguläre Sprachen und ihre Akzeptoren

- Reguläre Sprachen werden von Typ-3 Grammatiken erzeugt.
  - (Nicht-)Terminale: *nonterminal* $\rightarrow$ X
  - Alternativen: *hexdigit* $\rightarrow$ 0 | 1 | 2 | 3 | …| f
  - Konkatenierung: *hexprefix* $\rightarrow$ 0x
  - Wiederholung: *hexliteral* $\rightarrow$ *hexprefix hexdigit hexdigit** (Kleene Stern)
- Reguläre Ausdrücke sind eine Kurzschreibweise für diese Grammatiken.
  - Für hexadezimale Zahlen: 0x[0-9a-f][0-9a-f]*
  - Verwendung zum Pattern Matching in Texten (`grep(1)`, Py: `import re`, …)
  - Effizient implementierbar in $\mathcal{O}(n)$, wenn n die Eingabelänge

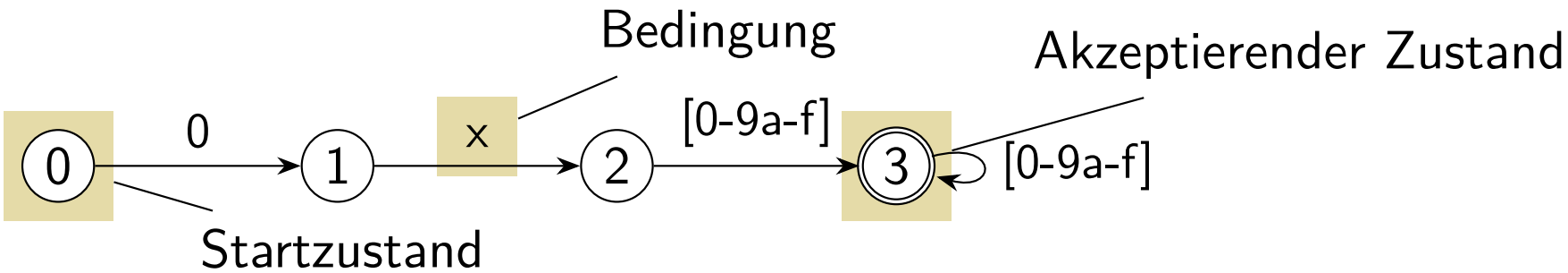
Wiederholung: Reguläre Sprachen und ihre Akzeptoren

- Reguläre Sprachen werden von Typ-3 Grammatiken erzeugt.
  - (Nicht-)Terminale: *nonterminal* $\rightarrow$ X
  - Alternativen: *hexdigit* $\rightarrow$ 0 | 1 | 2 | 3 | …| f
  - Konkatenierung: *hexprefix* $\rightarrow$ 0x
  - Wiederholung: *hexliteral* $\rightarrow$ *hexprefix hexdigit hexdigit** (Kleene Stern)
- Reguläre Ausdrücke sind eine Kurzschreibweise für diese Grammatiken.
  - Für hexadezimale Zahlen: 0x[0-9a-f][0-9a-f]*
  - Verwendung zum Pattern Matching in Texten (`grep(1)`, Py: `import re`, …)
  - Effizient implementierbar in $\mathcal{O}(n)$, wenn n die Eingabelänge
- Darstellung und Implementierung: (nicht-)deterministische Automaten

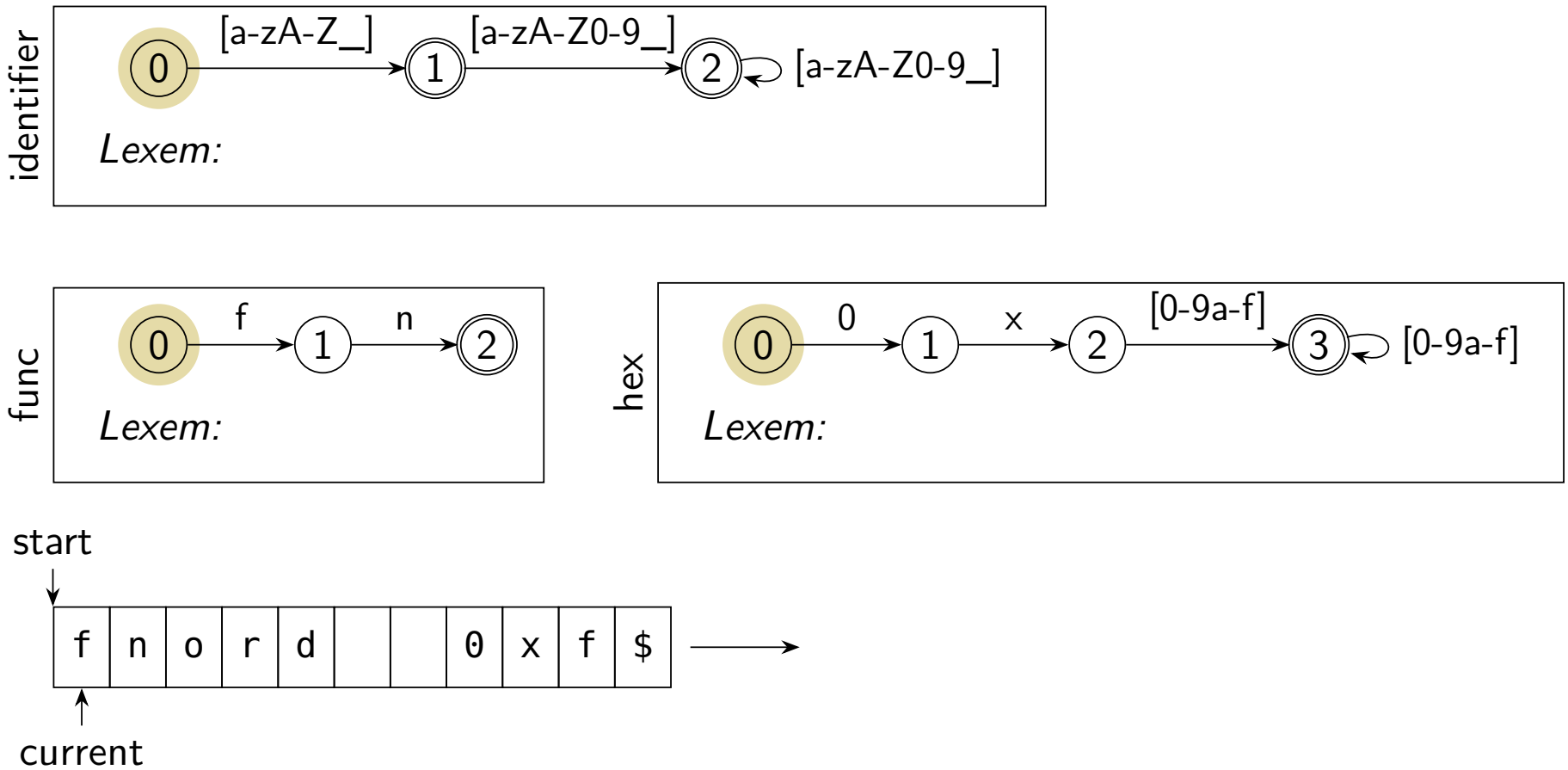


Wiederholung: Reguläre Sprachen und ihre Akzeptoren

- Reguläre Sprachen werden von Typ-3 Grammatiken erzeugt.
  - (Nicht-)Terminale: *nonterminal* → X
  - Alternativen: *hexdigit* → 0 | 1 | 2 | 3 | …| f
  - Konkatenierung: *hexprefix* → 0x
  - Wiederholung: *hexliteral* → *hexprefix hexdigit hexdigit** (Kleene Stern)
- Reguläre Ausdrücke sind eine Kurzschreibweise für diese Grammatiken.
  - Für hexadezimale Zahlen: 0x[0-9a-f][0-9a-f]*
  - Verwendung zum Pattern Matching in Texten (`grep(1)`, Py: `import re`, …)
  - Effizient implementierbar in $\mathcal{O}(n)$, wenn n die Eingabelänge
- Darstellung und Implementierung: (nicht-)deterministische Automaten

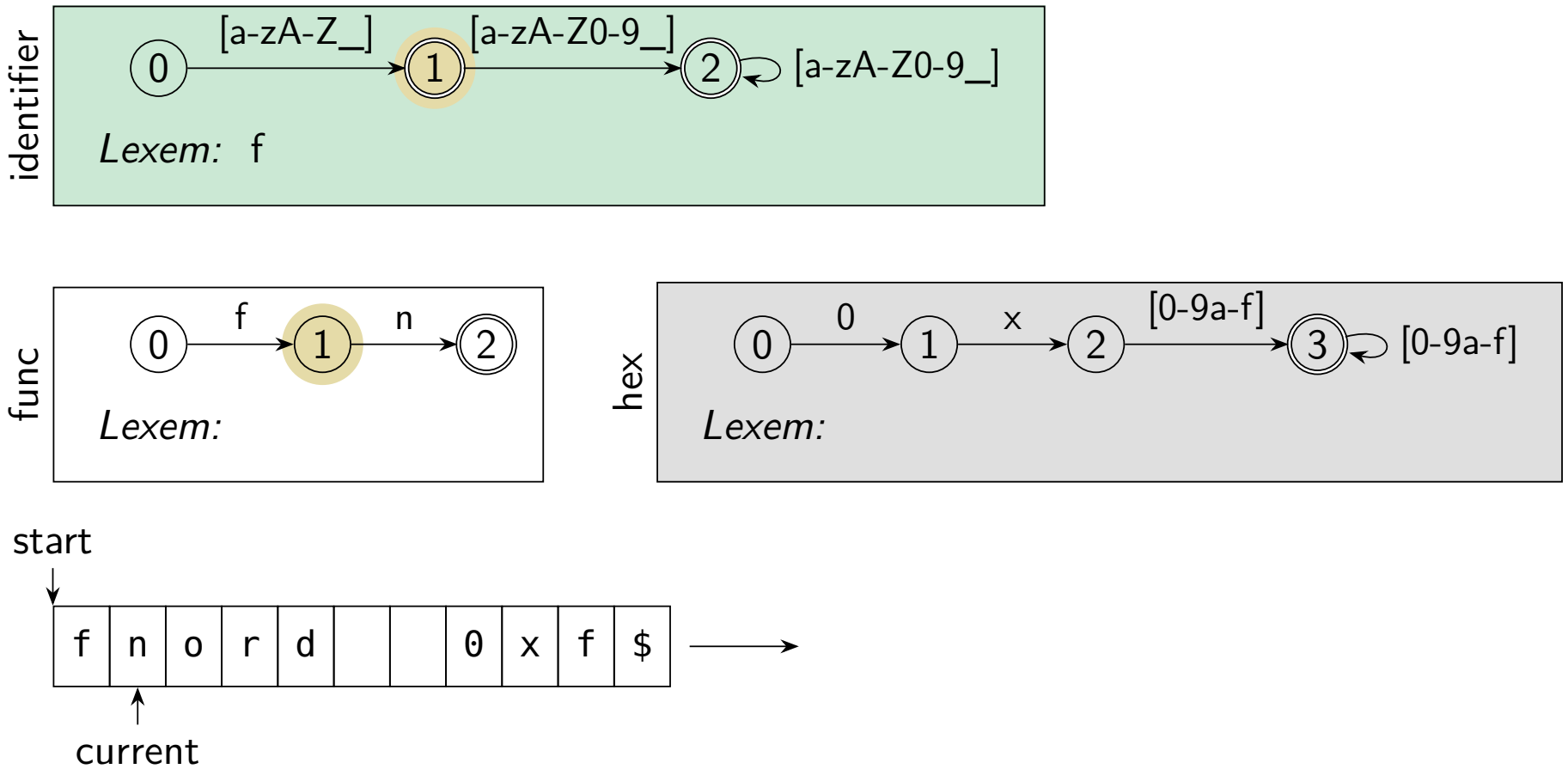


Scanner als Kombination von Regulären Automaten



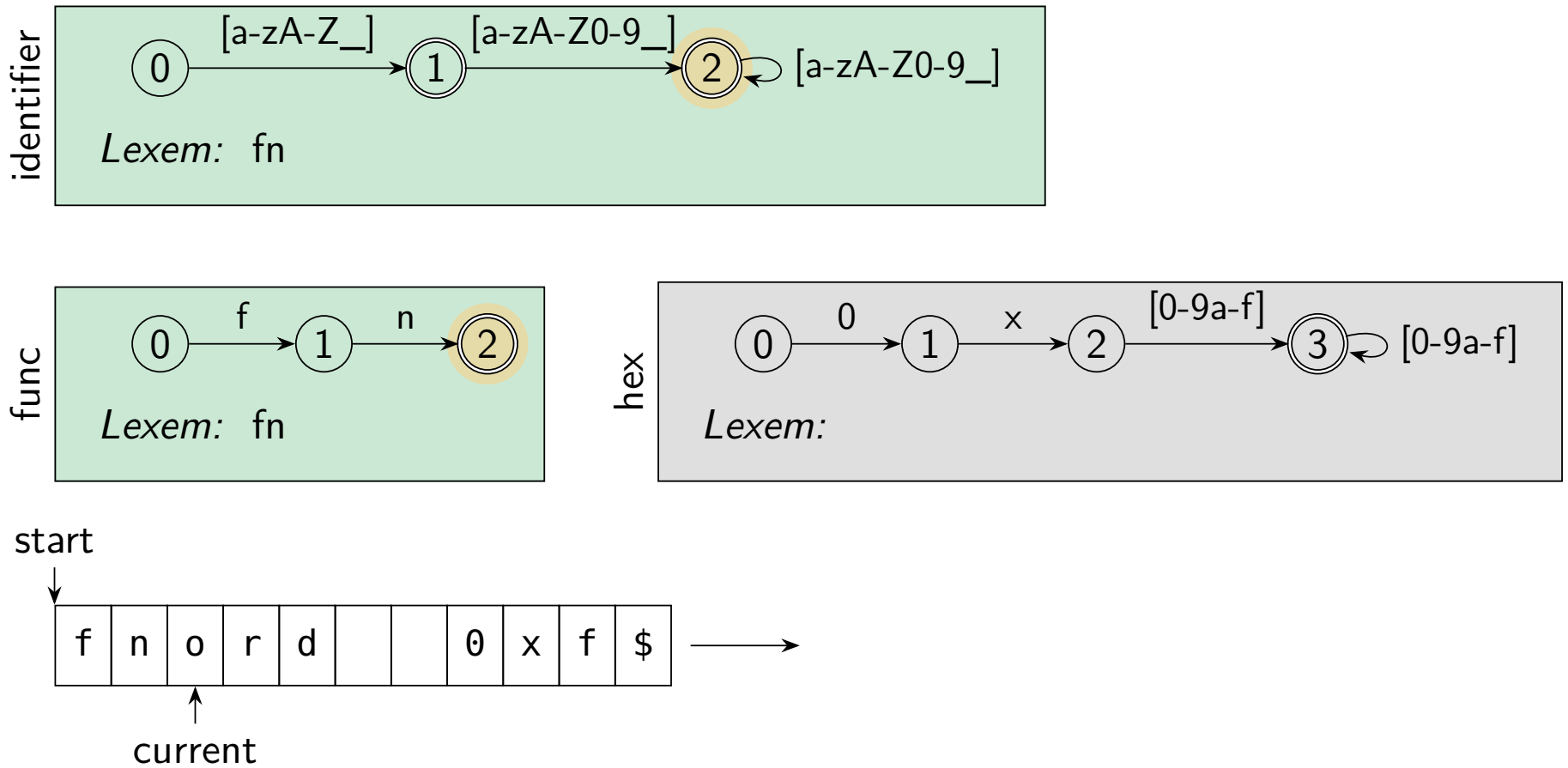
- Automaten parallel über die Eingabe ausführen
  - Longest Possible Match, Konfliktauflösung bei mehrfachem Match
  - Simulation des NFA, Umwandlung in *determinstic finite automata* (`flex(1)`)

Scanner als Kombination von Regulären Automaten



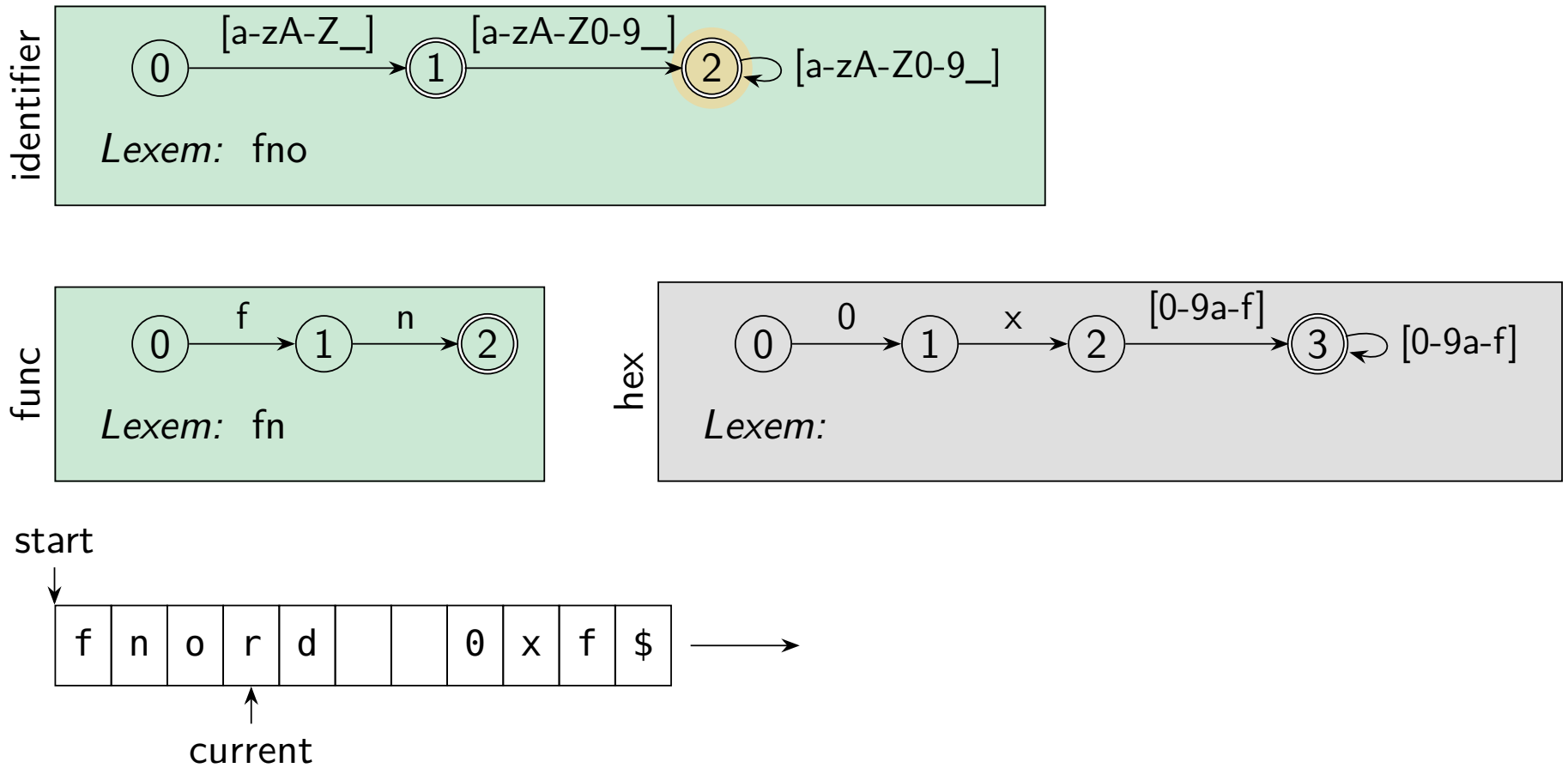
- Automaten parallel über die Eingabe ausführen
  - Longest Possible Match, Konfliktauflösung bei mehrfachem Match
  - Simulation des NFA, Umwandlung in *determinstic finite automata* (`flex(1)`)

Scanner als Kombination von Regulären Automaten



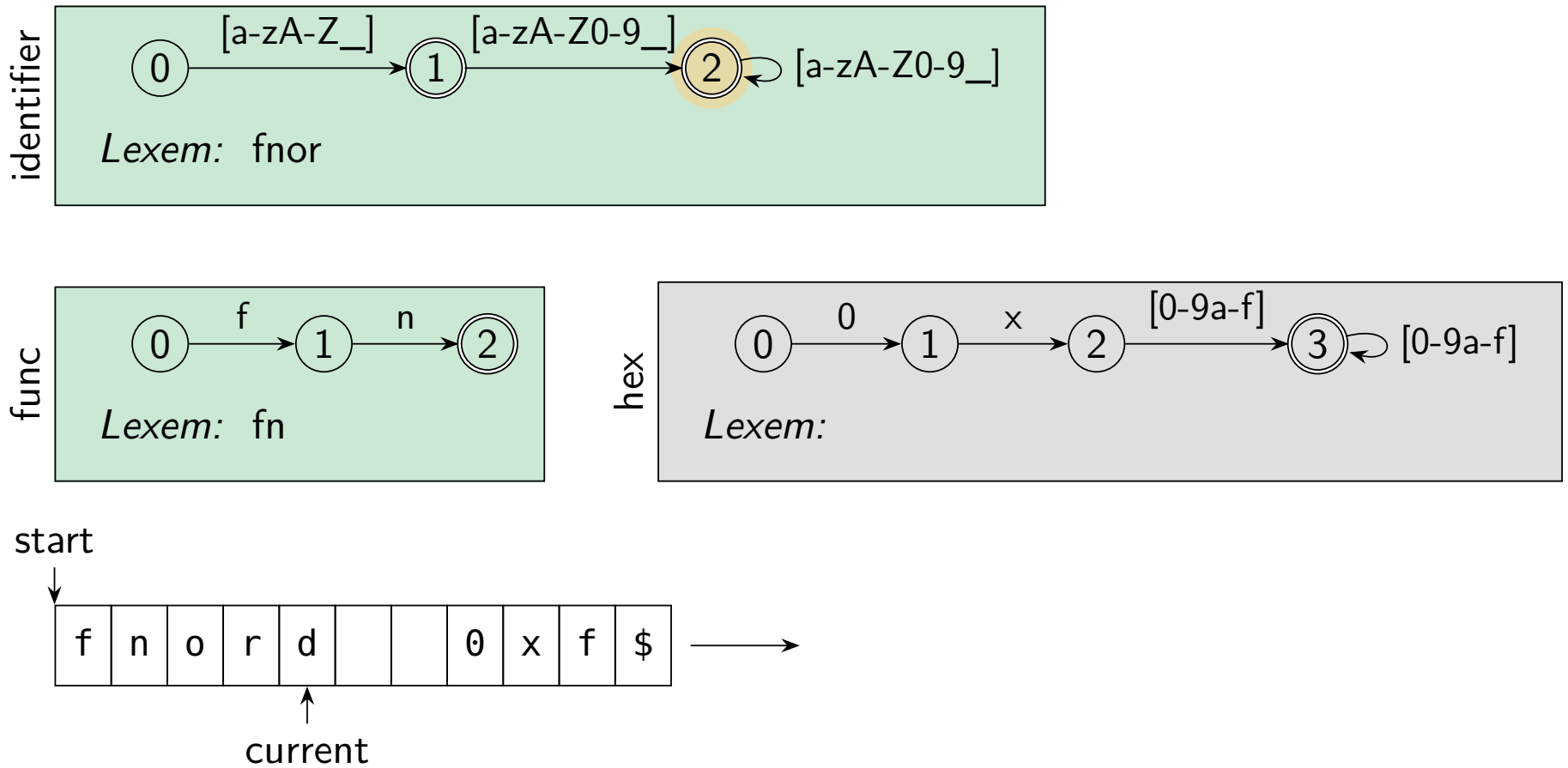
- Automaten parallel über die Eingabe ausführen
  - Longest Possible Match, Konfliktauflösung bei mehrfachem Match
  - Simulation des NFA, Umwandlung in *determinstic finite automata* (`flex(1)`)

Scanner als Kombination von Regulären Automaten



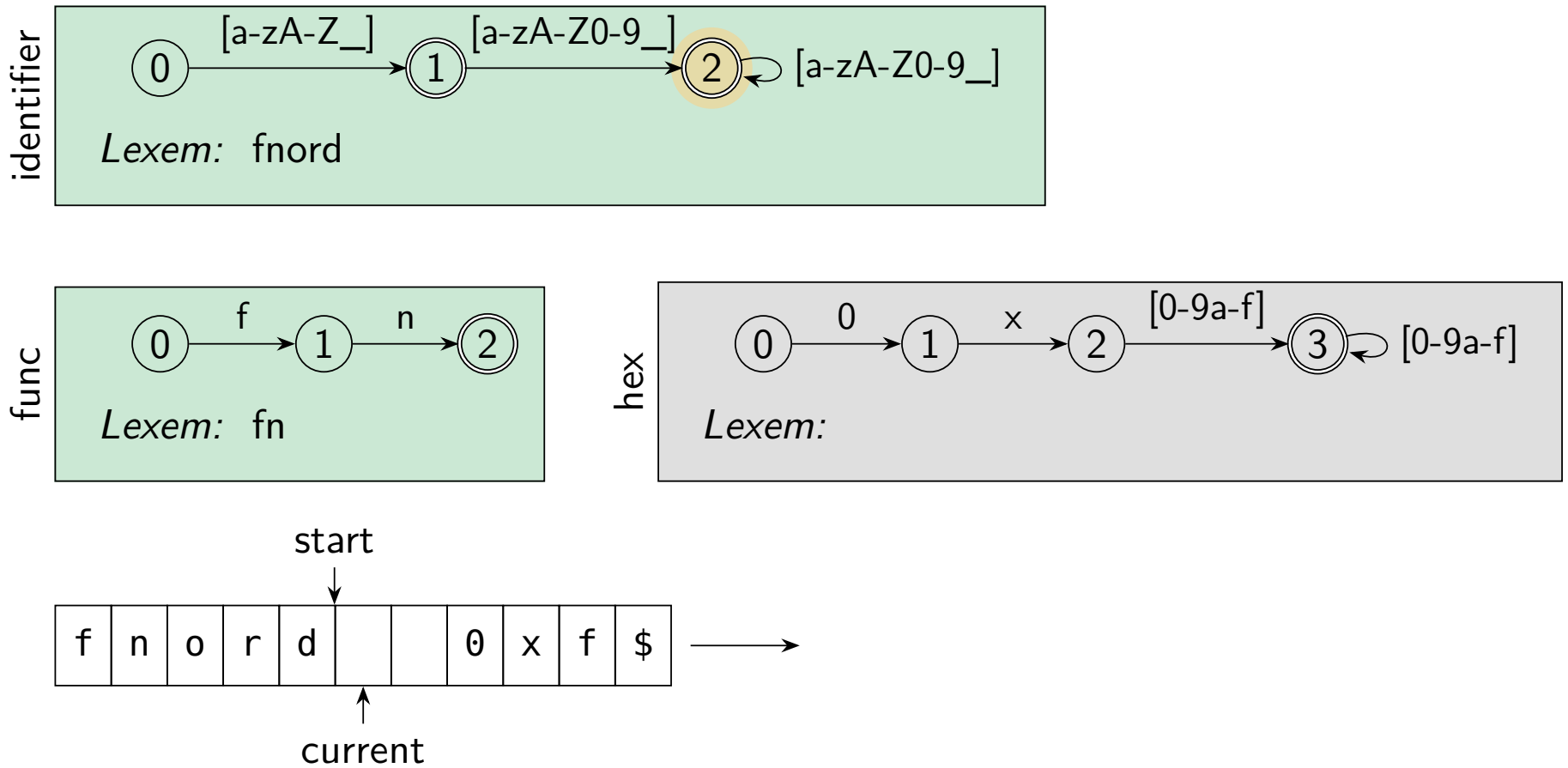
- Automaten parallel über die Eingabe ausführen
  - Longest Possible Match, Konfliktauflösung bei mehrfachem Match
  - Simulation des NFA, Umwandlung in *determinstic finite automata* (`flex(1)`)

Scanner als Kombination von Regulären Automaten



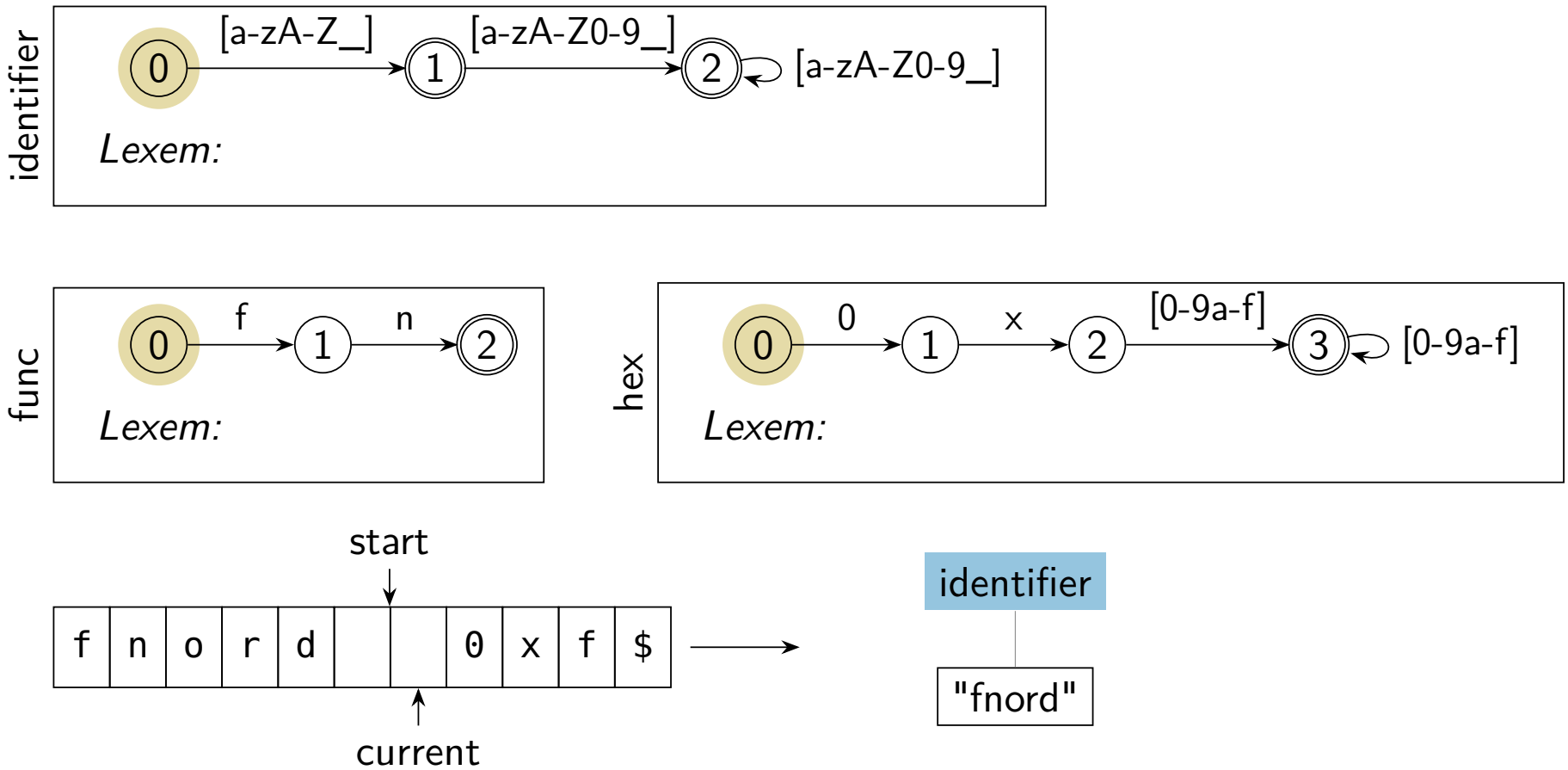
- Automaten parallel über die Eingabe ausführen
  - Longest Possible Match, Konfliktauflösung bei mehrfachem Match
  - Simulation des NFA, Umwandlung in *determinstic finite automata* (`flex(1)`)

Scanner als Kombination von Regulären Automaten



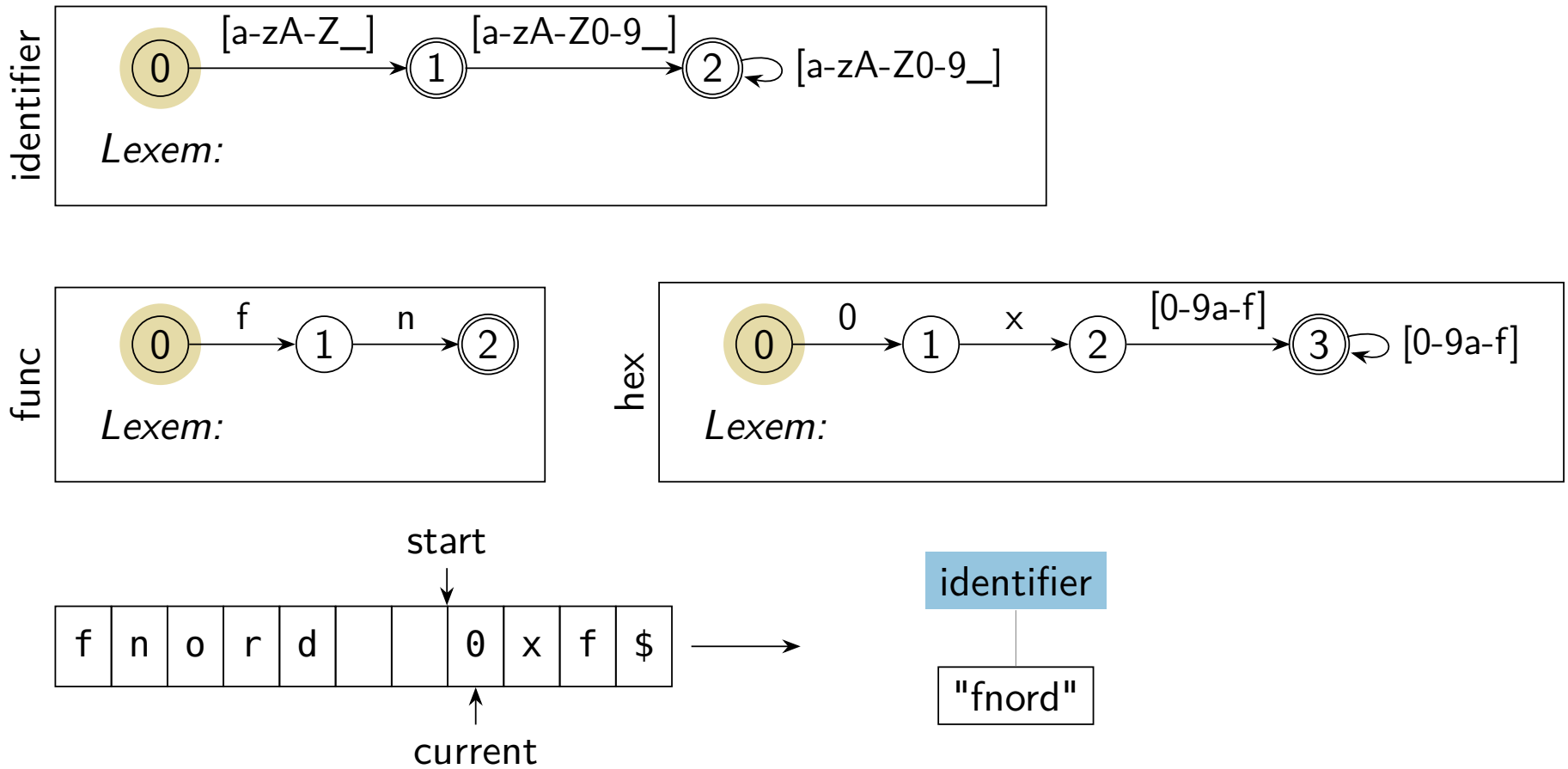
- Automaten parallel über die Eingabe ausführen
  - Longest Possible Match, Konfliktauflösung bei mehrfachem Match
  - Simulation des NFA, Umwandlung in *determinstic finite automata* (`flex(1)`)

Scanner als Kombination von Regulären Automaten



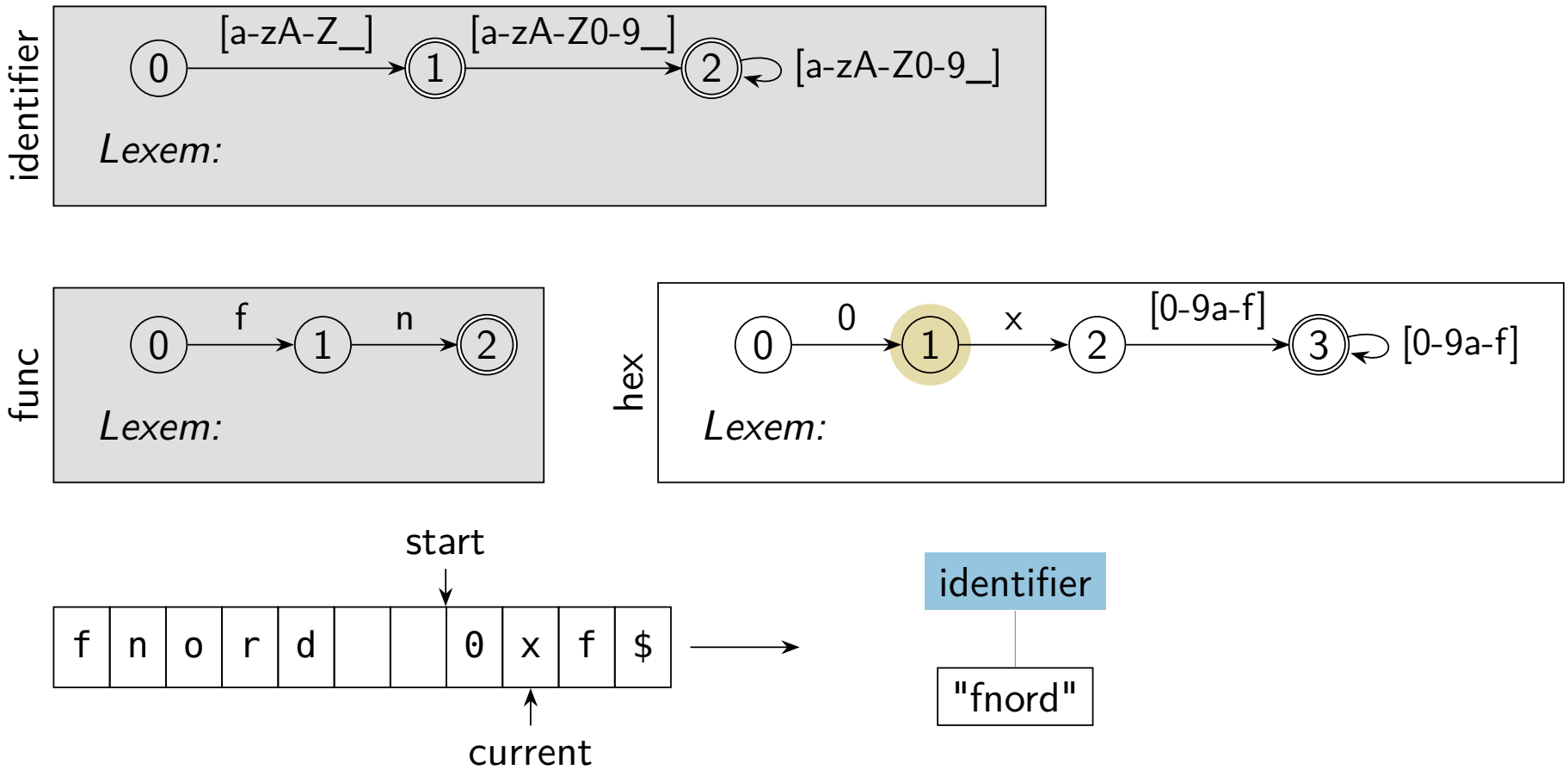
- Automaten parallel über die Eingabe ausführen
  - Longest Possible Match, Konfliktauflösung bei mehrfachem Match
  - Simulation des NFA, Umwandlung in *determinstic finite automata* (`flex(1)`)

Scanner als Kombination von Regulären Automaten



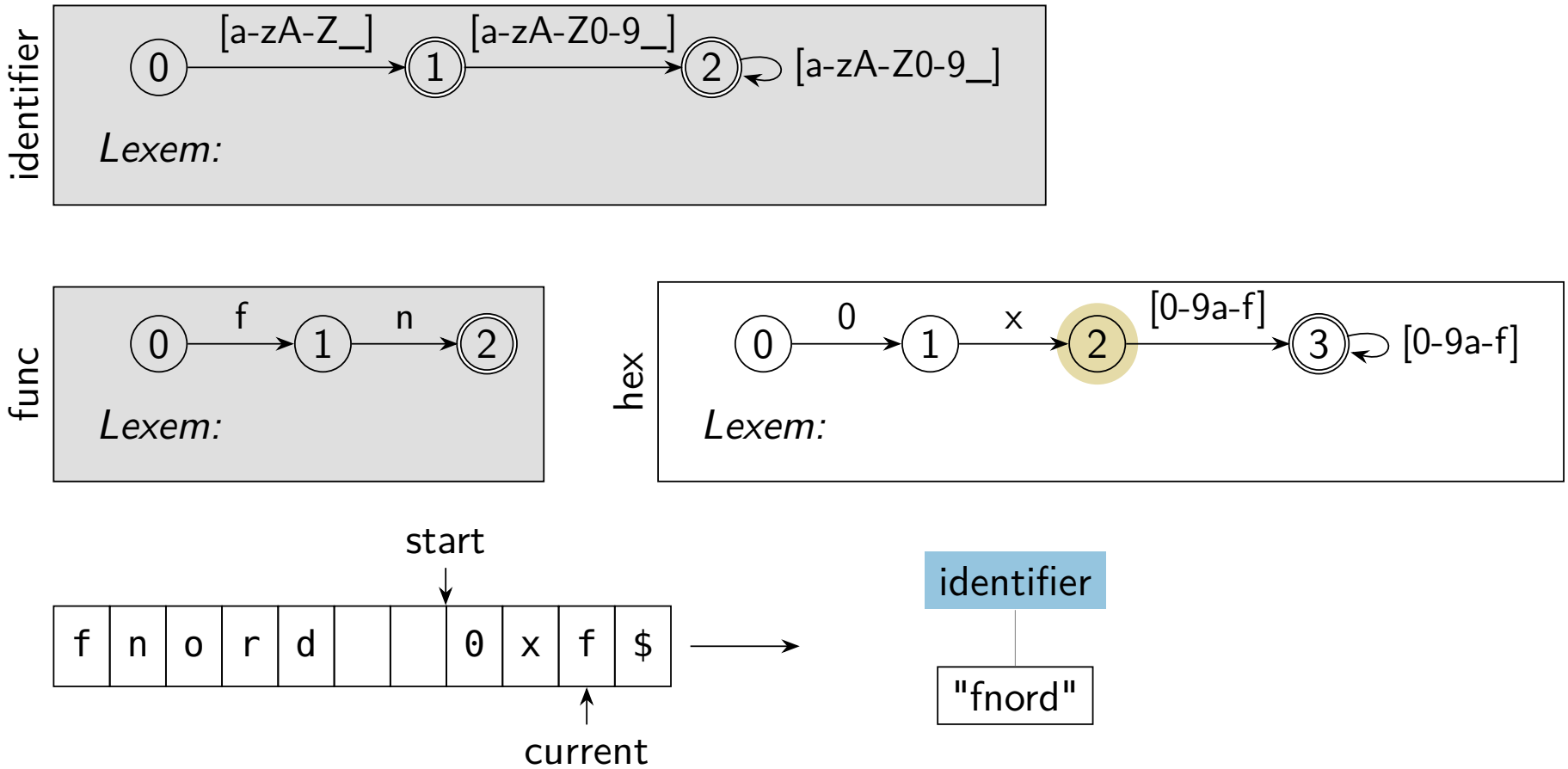
- Automaten parallel über die Eingabe ausführen
  - Longest Possible Match, Konfliktauflösung bei mehrfachem Match
  - Simulation des NFA, Umwandlung in *determinstic finite automata* (`flex(1)`)

Scanner als Kombination von Regulären Automaten



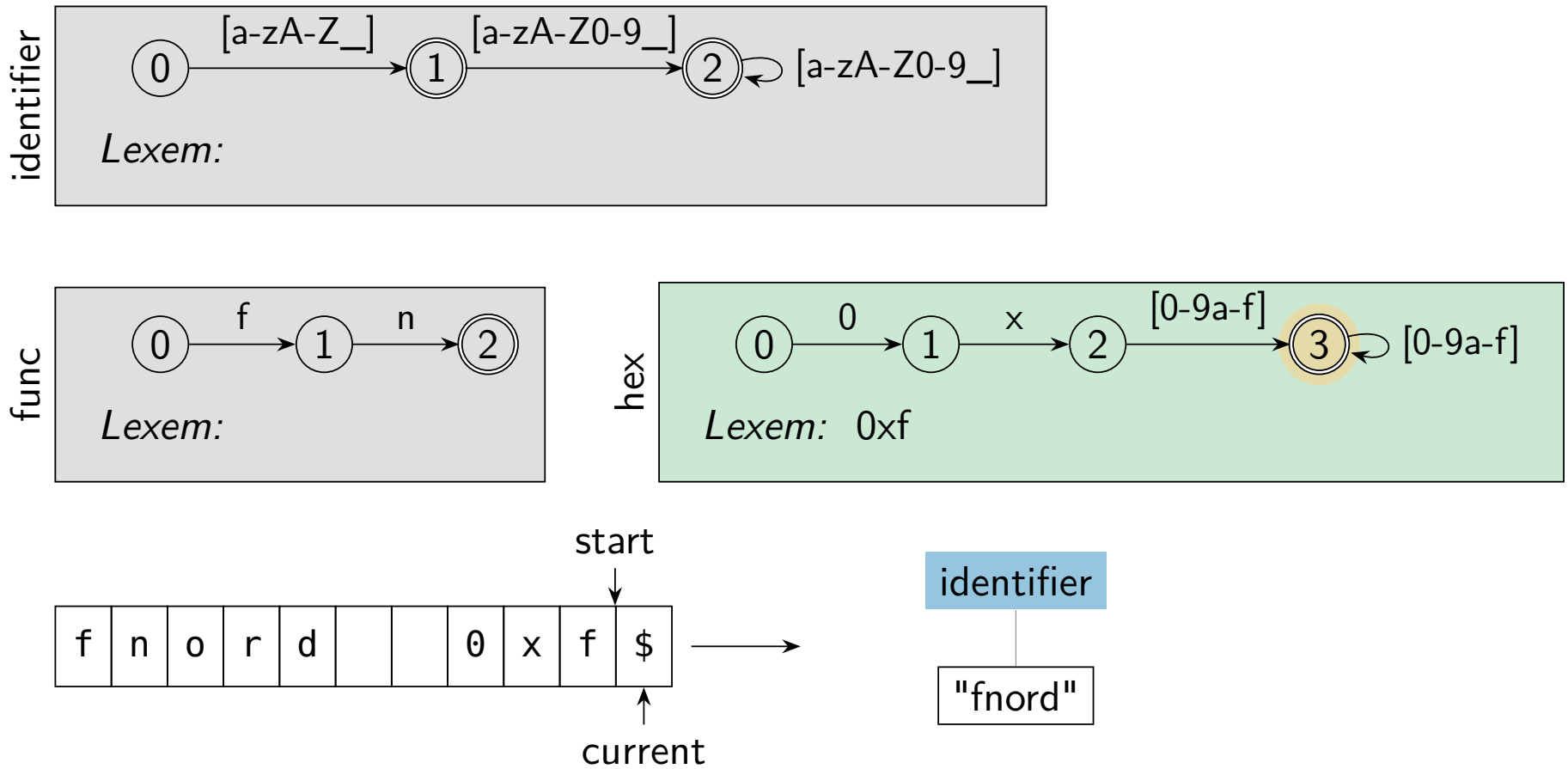
- Automaten parallel über die Eingabe ausführen
  - Longest Possible Match, Konfliktauflösung bei mehrfachem Match
  - Simulation des NFA, Umwandlung in *determinstic finite automata* (`flex(1)`)

Scanner als Kombination von Regulären Automaten



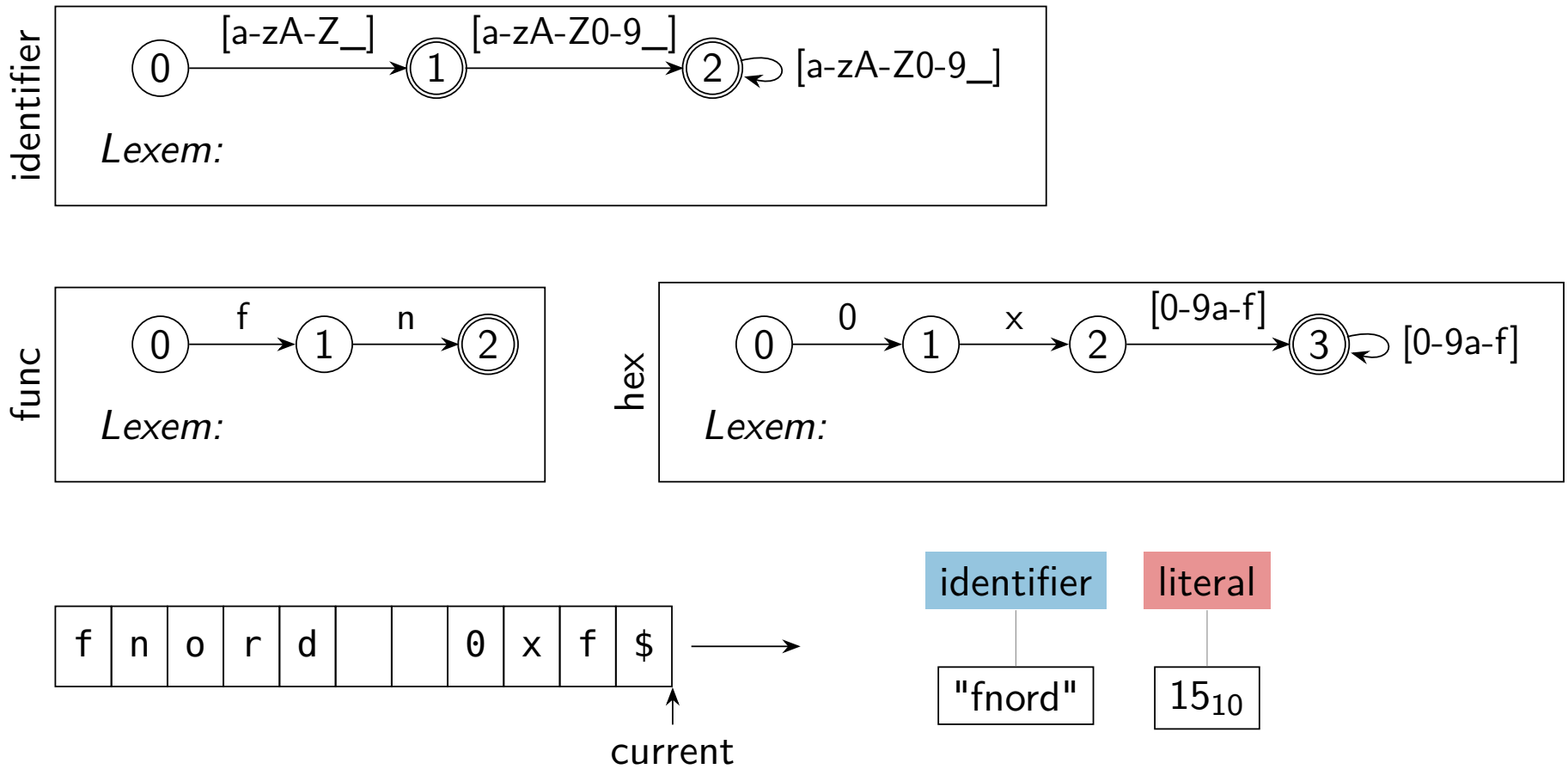
- Automaten parallel über die Eingabe ausführen
  - Longest Possible Match, Konfliktauflösung bei mehrfachem Match
  - Simulation des NFA, Umwandlung in *determinstic finite automata* (`flex(1)`)

Scanner als Kombination von Regulären Automaten



- Automaten parallel über die Eingabe ausführen
  - Longest Possible Match, Konfliktauflösung bei mehrfachem Match
  - Simulation des NFA, Umwandlung in *determinstic finite automata* (`flex(1)`)

Scanner als Kombination von Regulären Automaten



- Automaten parallel über die Eingabe ausführen
  - Longest Possible Match, Konfliktauflösung bei mehrfachem Match
  - Simulation des NFA, Umwandlung in *determinstic finite automata* (`flex(1)`)

Parser
und
Syntaxbaum

Blockstruktur und Klammerung

- **Erinnerung**: Wir wollen Syntaxbaum aus Tokenstrom erzeugen.

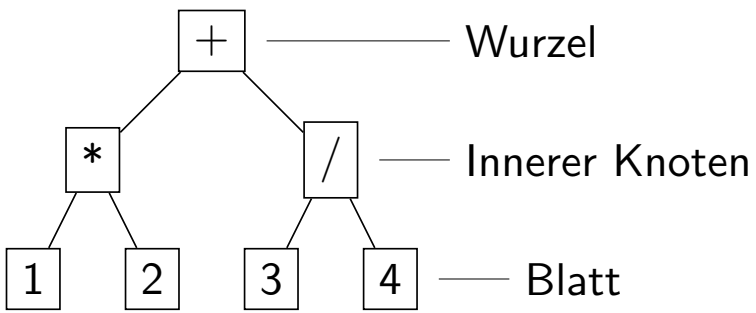
- Programmiersprachen enthalten *geschachtelte* oder *rekursive* Konstrukte
  - Geklammerte Ausdrücke: ((1*2)+(3/4)-(5*6)) *explizit*
  - Operatorpräzedenz: 1*2+3/4-5*6 *implizit*
  - Blockstrukturen: if (…) { if (…) { while (…) { …} } } *explizit*
  - Grammatik: *expr* → (*expr*) | *expr* + *expr*

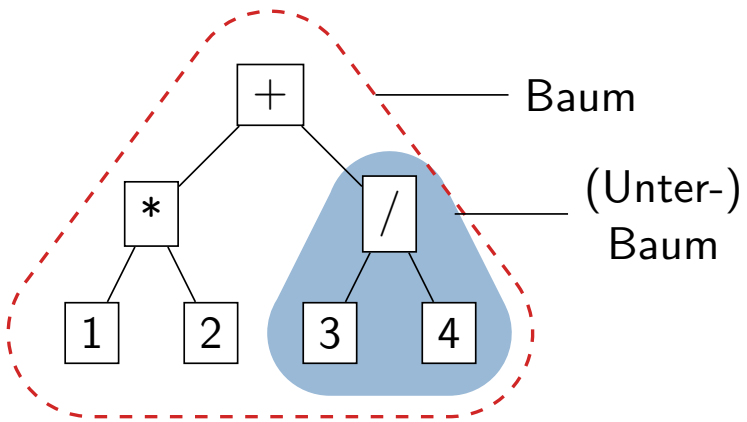
Blockstruktur und Klammerung

- **Erinnerung**: Wir wollen Syntaxbaum aus Tokenstrom erzeugen.

- Programmiersprachen enthalten *geschachtelte* oder *rekursive* Konstrukte
  - Geklammerte Ausdrücke: ((1*2)+(3/4)-(5*6)) *explizit*
  - Operatorpräzedenz: 1*2+3/4-5*6 *implizit*
  - Blockstrukturen: if (…) { if (…) { while (…) { …} } } *explizit*
  - Grammatik: *expr* → (*expr*) | *expr* + *expr*

- Bäume eignen sich, diese Strukture zu erfassen!



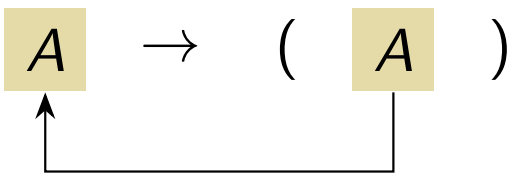


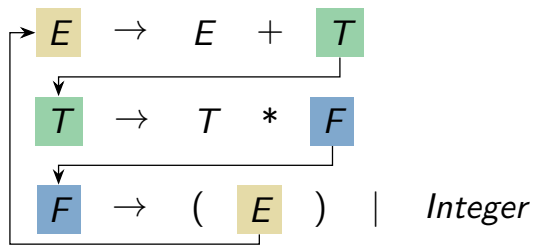
  - Token werden hierarchisch geordnet. Implizite Regeln werden explizit.
  - Bäume sind rekursive Datenstrukturen. Unterbäume sind auch Bäume!

Wiederholung: Kontextfreie Grammatiken

- Reguläre Grammatiken nicht fähig rekursive Strukturen zu erfassen
  „Klammernzählen“ ist verboten: $A \rightarrow (A) \mid \epsilon$

- Kontextfreie Sprachen werden von Typ-2 Grammatiken erzeugt.
  - Weiterhin erlaubt: Alternativen (|), Konkatenierung (Aa), Wiederholung ($A*$)
  - Zusätzlich: Nichtterminale dürfen sich selbst referenzieren.



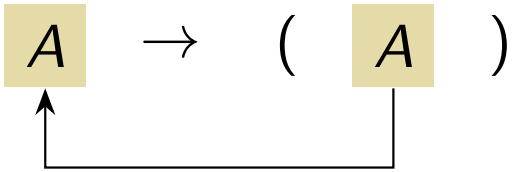


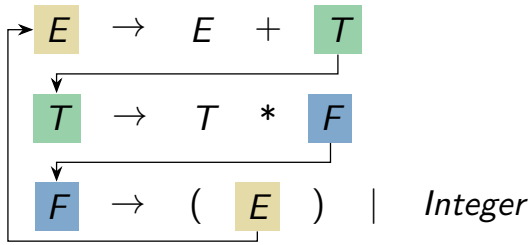
- Kontextfreie Sprachen können von Kellerautomaten erkannt werden.
  - Kellerautomat: Endlicher Zustandsautomat + Unendlicher Stapelspeicher
  - Erkennung immer in $\mathcal{O}(n^3)$ möglich: Cocke-Younger-Kasami Algorithmus
  - Für Programmiersprachen relevante Grammatiken: $\mathcal{O}(n)$

Wiederholung: Kontextfreie Grammatiken

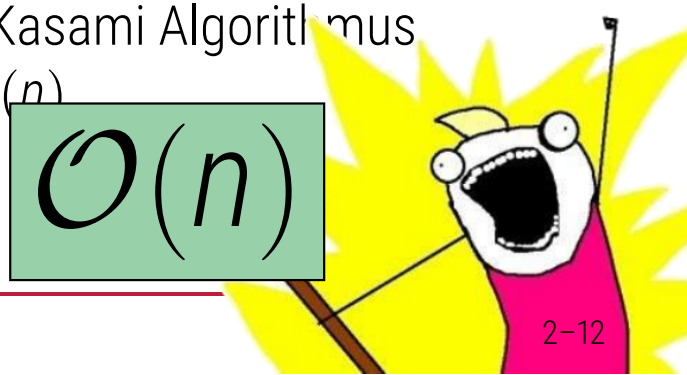
- Reguläre Grammatiken nicht fähig rekursive Strukturen zu erfassen
  „Klammernzählen“ ist verboten: $A \rightarrow (A) \mid \epsilon$

- Kontextfreie Sprachen werden von Typ-2 Grammatiken erzeugt.
  - Weiterhin erlaubt: Alternativen (|), Konkatenierung (Aa), Wiederholung ($A*$)
  - Zusätzlich: Nichtterminale dürfen sich selbst referenzieren.





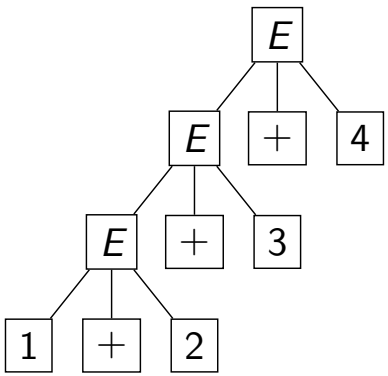
- Kontextfreie Sprachen können von Kellerautomaten erkannt werden.
  - Kellerautomat: Endlicher Zustandsautomat + Unendlicher Stapelspeicher
  - Erkennung immer in $\mathcal{O}(n^3)$ möglich: Cocke-Younger-Kasami Algorithmus
  - Für Programmiersprachen relevante Grammatiken: $\mathcal{O}(n)$



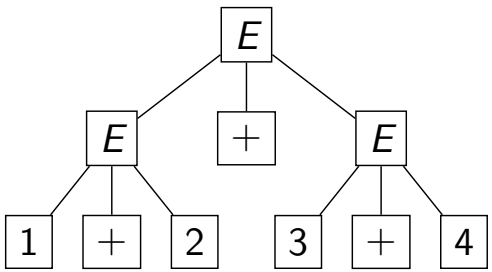
Deterministische Kontextfreie Grammatiken

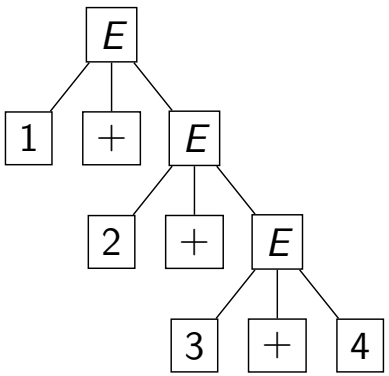
- Kontextfrei bedeutet nicht direkt, dass der Baum eindeutig ist.
  - Linksableitung: Das linkeste Nicht-Terminal wird zuerst abgeleitet.

$E \rightarrow E + E \mid \text{Integer}$ $\oplus$ $1 + 2 + 3 + 4$ $\Rightarrow$



Linksableitung





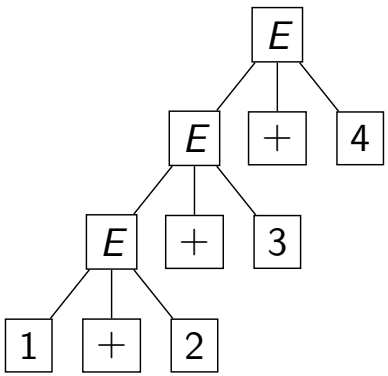
Rechtsableitung

- Wir werden nur deterministische Grammatiken verwenden.
  - LL-Grammatiken: **L**inks lesend, **L**inksableitend
  - LR-Grammatiken: **L**inks lesend, **R**echtsableitend
  - LL(k)/LR(k): Mit k Zeichen Look-Ahead Parsebar

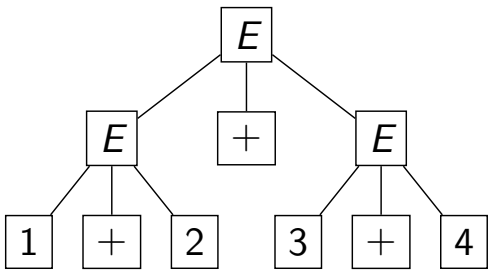
Deterministische Kontextfreie Grammatiken

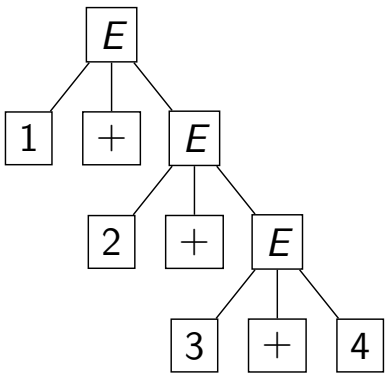
- Kontextfrei bedeutet nicht direkt, dass der Baum eindeutig ist.
  - Linksableitung: Das linkeste Nicht-Terminal wird zuerst abgeleitet.

$$E \rightarrow E + E \mid \text{Integer} \oplus 1 + 2 + 3 + 4 \Rightarrow$$



Linksableitung





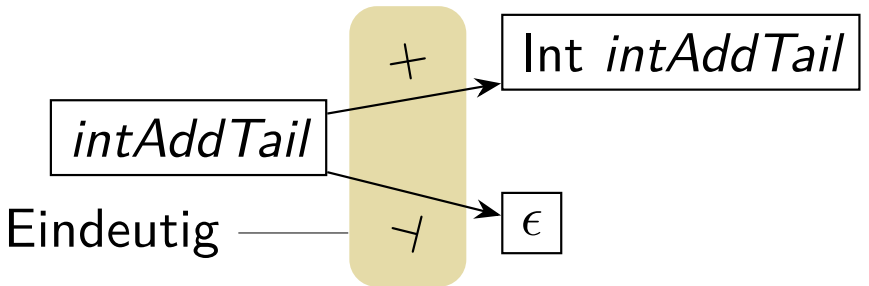
Rechtsableitung

- Wir werden nur deterministische Grammatiken verwenden.
  - LL-Grammatiken: **L**inks lesend, **L**inksableitend $\Rightarrow$ Recursive-Descent Parser
  - LR-Grammatiken: **L**inks lesend, **R**echtsableitend $\Rightarrow$ Parsergeneratoren
  - LL(k)/LR(k): Mit k Zeichen Look-Ahead Parsebar LL(1) $\ll$ LR(1)

Deterministische Linksableitende Grammatiken

Informell: Bei einer LL(1)-Grammatik kann man am aktuellen Zeichen sicher erkennen, welche Regel als nächstes angewendet werden soll.

intAdd $\rightarrow$ Int *intAddTail*
intAddTail $\rightarrow$ + Int *intAddTail*
intAddTail $\rightarrow$ $\dashv$



- Diese Sprachklasse kann mit *rekursivem Abstieg* geparst werden.
  - Eine Funktion für jedes Nicht-Terminal:
    ```
    def intAddTail(token_stream)
    ```
  - Entscheidung über angewendete Regel anhand des Look-Aheads:
    ```
    if token_stream.peek() == TOK_PLUS: ...
    ```
  - Recursive-Descent Parser werden manuell erstellt oder generiert.

Beispiel: Parser mit rekursivem Abstieg

Technische Universität Braunschweig

```
def intAdd(token_stream):
    # Nur eine Regel: intAdd -> Int intAddTail
    ## 'Int': Das erste Token muss ein INT sein
    int_load  = token_stream.read(expected=TOK_INT)
    ## 'intAddTail': Rekursiver Abstieg
    tail_tree = intAddTail(token_stream)
    # Baumstruktur als geschachtelte Listen
    return ["intAdd", int_load, tail_tree]
```

```
def intAddTail(token_stream):
    # Verwende Look-Ahead zur Regelauswahl
    if token_stream.peek() == TOK_PLUS:
        # intAddTail -> '+' Int intAddTail
        ## '+': consume the PLUS
        _          = token_stream.read(expected=TOK_PLUS)
        ## 'Int': capture the integer
        int_load  = token_stream.read(expected=TOK_INT)
        ## 'intAddTail': Rekursiver Abstieg!
        tail_tree = intAddTail(token_stream)
        # Baumstruktur zurückgeben
        return ["intAddTail", int_load, tail_tree]
    elif token_stream.peek() == TOK_EOF: #...
```

Beispiel: Parser mit rekursivem Abstieg

```
def intAdd(token_stream):
    # Nur eine Regel: intAdd -> Int intAddTail
    ## 'Int': Das erste Token muss ein INT sein
    int_load  = token_stream.read(expected=TOK_INT)
    ## 'intAddTail': Rekursiver Abstieg
    tail_tree = intAddTail(token_stream)
    # Baumstruktur als geschachtelte Listen
    return ["intAdd", int_load, tail_tree]
```

```
def intAddTail(token_stream):
    # Verwende Look-Ahead zur Regelauswahl
    if token_stream.peek() == TOK_PLUS:
        # intAddTail -> '+' Int intAddTail
        ## '+': consume the PLUS
        _          = token_stream.read(expected=TOK_PLUS)
        ## 'Int': capture the integer
        int_load  = token_stream.read(expected=TOK_INT)
        ## 'intAddTail': Rekursiver Abstieg!
        tail_tree = intAddTail(token_stream)
        # Baumstruktur zurückgeben
        return ["intAddTail", int_load, tail_tree]
    elif token_stream.peek() == TOK_EOF: #...
```

Beispiel: Parser mit rekursivem Abstieg


```
def intAdd(token_stream):
    # Nur eine Regel: intAdd -> Int intAddTail
    ## 'Int': Das erste Token muss ein INT sein
    int_load  = token_stream.read(expected=TOK_INT)
    ## 'intAddTail': Rekursiver Abstieg
    tail_tree = intAddTail(token_stream)
    # Baumstruktur als geschachtelte Listen
    return ["intAdd", int_load, tail_tree]
```

```
def intAddTail(token_stream):
    # Verwende Look-Ahead zur Regelauswahl
    if token_stream.peek() == TOK_PLUS:
        # intAddTail -> '+' Int intAddTail
        ## '+': consume the PLUS
        _          = token_stream.read(expected=TOK_PLUS)
        ## 'Int': capture the integer
        int_load  = token_stream.read(expected=TOK_INT)
        ## 'intAddTail': Rekursiver Abstieg!
        tail_tree = intAddTail(token_stream)
        # Baumstruktur zurückgeben
        return ["intAddTail", int_load, tail_tree]
    elif token_stream.peek() == TOK_EOF: #...
```

Beispiel: Parser mit rekursivem Abstieg

Technische Universität Braunschweig

```
def intAdd(token_stream):
    # Nur eine Regel: intAdd -> Int intAddTail
    ## 'Int': Das erste Token muss ein INT sein
    int_load  = token_stream.read(expected=TOK_INT)
    ## 'intAddTail': Rekursiver Abstieg
    tail_tree = intAddTail(token_stream)
    # Baumstruktur als geschachtelte Listen
    return ["intAdd", int_load, tail_tree]
```

```
def intAddTail(token_stream):
    # Verwende Look-Ahead zur Regelauswahl
    if token_stream.peek() == TOK_PLUS:
        # intAddTail -> '+' Int intAddTail
        ## '+': consume the PLUS
        _          = token_stream.read(expected=TOK_PLUS)
        ## 'Int': capture the integer
        int_load  = token_stream.read(expected=TOK_INT)
        ## 'intAddTail': Rekursiver Abstieg!
        tail_tree = intAddTail(token_stream)
        # Baumstruktur zurückgeben
        return ["intAddTail", int_load, tail_tree]
    elif token_stream.peek() == TOK_EOF: #...
```

Beispiel: Parser mit rekursivem Abstieg

Technische Universität Braunschweig

```
def intAdd(token_stream):
    # Nur eine Regel: intAdd -> Int intAddTail
    ## 'Int': Das erste Token muss ein INT sein
    int_load  = token_stream.read(expected=TOK_INT)
    ## 'intAddTail': Rekursiver Abstieg
    tail_tree = intAddTail(token_stream)
    # Baumstruktur als geschachtelte Listen
    return ["intAdd", int_load, tail_tree]
```

```
def intAddTail(token_stream):
    # Verwende Look-Ahead zur Regelauswahl
    if token_stream.peek() == TOK_PLUS:
        # intAddTail -> '+' Int intAddTail
        ## '+': consume the PLUS
        _         = token_stream.read(expected=TOK_PLUS)
        ## 'Int': capture the integer
        int_load  = token_stream.read(expected=TOK_INT)
        ## 'intAddTail': Rekursiver Abstieg!
        tail_tree = intAddTail(token_stream)
        # Baumstruktur zurückgeben
        return ["intAddTail", int_load, tail_tree]
    elif token_stream.peek() == TOK_EOF: #...
```

`addInt(Scanner("1+2"))` ⇒ `['intAdd',1,['intAddTail',2,['intAddTail',]]]`

LL(1): PREDICT-Mengen

Für jede **Regel** enthält die PREDICT-Menge jene Token/Terminale, die im Look-Ahead dazu führen, dass die Regel angewendet wird.

$$\text{PREDICT}(A \to a) \in T^n$$

Regel	PREDICT
intAdd $\to$ Int *intAddTail*	{ Int }
intAddTail $\to$ + Int *intAddTail*	{ + }
intAddTail $\to$ $	{ $ }
qualifier $\to$ const $\mid \epsilon$	{ const }
funcDef $\to$ *qualifier* func identifier ...	{ const, func }

Test: Ist eine Grammatik in LL(1)?

Alle PREDICT-Mengen mit der gleichen **linken** Seite müssen schnittfrei sein.
Für jedes Nicht-Terminal ist Regelauswahl eindeutig.

Parser-Generator für Parser mit rekursivem Abstieg

- Mit den PREDICT-Mengen können wir einen LL(1)-Parser generieren.
  - Der Parser-Generator (grau) iteriert über Nichtterminale und Regeln.
  - Er erzeugt und emittiert, mittels Codeschablonen (gelb), den Parsercode.
  - Innerhalb der Schablonen werden Ersetzungen (grün) vorgenommen.

```
# Für jedes Nicht-Terminal erzeugen wir eine Funktion.
for NT in nonterminals:
    def NT.name(token_stream):
    # Die \PREDICT entscheidet, ob eine Regel genommen wird.
    for rule in NT.rules:
        if token_stream.peek() in PREDICT(rule):
        for symbol in rule.righthandside:
            if is_terminal(symbol):
            ..._load = token_stream.read(expect=symbol.name)
            else:
            ..._tree = symbol.name(token_stream)
        # Return Parse Tree
            return [NT.name, ..._load, ..._tree, ...]
```

- Dieser Parser-Generator ist ein Übersetzer von „Grammatik“ → Python.

PREDICT: Berechnung von EPS und FIRST

Bisher: Die PREDICT-Menge fällt vom Himmel bzw. ist trivial.

- Berechnung von PREDICT mittels EPS, FIRST und FOLLOW.
  - EPS(α) $\equiv$ **if** $\alpha \Longrightarrow^* \epsilon$ **then** true **else** false
    Kann aus dem Wort α, durch Ableitung(en), das leere Wort entstehen?
  - FIRST(α) $\equiv \{c \mid c \in T, \alpha \Longrightarrow^* c\beta\}$
    Die Menge aller Terminale, die bei den Ableitungen von α links auftreten.
- Beispiel für EPS und FIRST:

1+(3+4)+foo(2+5)

term	$\rightarrow$ *factor factor_tail*
factor_tail	$\rightarrow$ + *factor_tail* | ϵ
factor	$\rightarrow$ Int | *trans* (*term*)
trans	$\rightarrow$ Ident | ϵ

α	EPS(α)	FIRST(α)
term	false	{Int, Ident, (}
factor_tail	true	{+}
factor	false	{Int, Ident, (}
trans	true	{ Ident }

PREDICT: Berechnung der FOLLOW-Menge

- $\text{FOLLOW}(A) \equiv \{c \mid c \in T, S \Longrightarrow^{+} \alpha A c \beta\}$
  - Die Menge aller Terminale, die bei einer Ableitung vom Startsymbol S direkt auf A folgen können. α und β sind beliebige Teilwörter.

Grammatik mit Endsymbol ⊣

S → *term* ⊣
term → *factor factor_tail*
factor_tail → + *factor_tail* | ϵ

factor → Int | trans (*term*)
trans → Ident | ϵ

A	FOLLOW(*A*)	Beispielhafte Satzformen
term	{),⊣}	(*term*) | *term*⊣
factor_tail	{),⊣}	(1+*factor_tail*) | 1+*factor_tail* ⊣
factor	{+,),⊣}	1+*factor***+**3 | (5+*factor***)** | 4+*factor* ⊣
trans	{(}	*trans* **(**1+2)

PREDICT = FIRST ⊕ FOLLOW

- $PREDICT(A \to \alpha) \equiv \text{FIRST}(\alpha) \cup (\textbf{if } \text{EPS}(\alpha) \textbf{ then } \text{FOLLOW}(A) \textbf{ else } \emptyset)$

Intuition: Ein Terminal c sagt eine Regel dann voraus, wenn:

1. Die Ableitung der rechten Seite mit dem Terminal c startet (FIRST).
2. Die rechte Seite α kann ϵ werden, daher muss man auf die Terminal schauen, die der Regel folgen können (FOLLOW).

$A \to \alpha$	PREDICT	EPS(α)	FIRST(α)	FOLLOW(A)
$S \to term \dashv$	Int, Ident, (	0	Int, Ident,(	
$term \to factor\ factor\_tail$	Int, Ident, (	0	Int, Ident, (	), ⊣
$factor\_tail \to +\ factor\_tail$ $factor\_tail \to \epsilon$	+), ⊣	0 1	+ ∅	), ⊣
$factor \to \text{Int}$ $factor \to \text{trans}(term)$	Int Ident, (	0 0	Int Ident, (	+,), ⊣
$trans \to \text{Ident}$ $trans \to \epsilon$	Ident (	0 1	Ident ∅	(

PREDICT = FIRST ⊕ FOLLOW

- $PREDICT(A \to \alpha) \equiv \text{FIRST}(\alpha) \cup (\textbf{if } \text{EPS}(\alpha) \textbf{ then } \text{FOLLOW}(A) \textbf{ else } \emptyset)$

Intuition: Ein Terminal c sagt eine Regel dann voraus, wenn:

1. Die Ableitung der rechten Seite mit dem Terminal c startet (FIRST).
2. Die rechte Seite α kann ϵ werden, daher muss man auf die Terminal schauen, die der Regel folgen können (FOLLOW).

$A \to \alpha$	PREDICT	EPS(α)	FIRST(α)	FOLLOW(A)
$S \to term \dashv$	Int, Ident, (	0	Int, Ident,(	
$term \to factor\ factor\_tail$	Int, Ident, (	0	Int, Ident, (	), ⊣
$factor\_tail \to +\ factor\_tail$ $factor\_tail \to \epsilon$	+), ⊣	0 1	+ ∅	), ⊣
$factor \to \text{Int}$ $factor \to \text{trans}(term)$	Int Ident, (	0 0	Int Ident, (	+,), ⊣
$trans \to \text{Ident}$ $trans \to \epsilon$	Ident (	0 1	Ident ∅	(

PREDICT = FIRST ⊕ FOLLOW

- $PREDICT(A \to \alpha) \equiv \text{FIRST}(\alpha) \cup (\textbf{if } \text{EPS}(\alpha) \textbf{ then } \text{FOLLOW}(A) \textbf{ else } \emptyset)$

Intuition: Ein Terminal c sagt eine Regel dann voraus, wenn:

1. Die Ableitung der rechten Seite mit dem Terminal c startet (FIRST).
2. Die rechte Seite α kann ϵ werden, daher muss man auf die Terminal schauen, die der Regel folgen können (FOLLOW).

$A \to \alpha$	PREDICT	EPS(α)	FIRST(α)	FOLLOW(A)
S → *term*⊣	Int, Ident, (	0	Int, Ident,(	
term → *factor factor_tail*	Int, Ident, (	0	Int, Ident, (	), ⊣
factor_tail → + *factor_tail*	+	0	+	), ⊣
factor_tail → ϵ	), ⊣	1	∅	
factor → Int	Int	0	Int	+,), ⊣
factor → trans(*term*)	Ident, (	0	Ident, (	
trans → Ident	Ident	0	Ident	(
trans → ϵ	(	1	∅	

PREDICT = FIRST ⊕ FOLLOW

- $PREDICT(A \to \alpha) \equiv \text{FIRST}(\alpha) \cup (\textbf{if } \text{EPS}(\alpha) \textbf{ then } \text{FOLLOW}(A) \textbf{ else } \emptyset)$

Intuition: Ein Terminal c sagt eine Regel dann voraus, wenn:

1. Die Ableitung der rechten Seite mit dem Terminal c startet (FIRST).
2. Die rechte Seite α kann ϵ werden, daher muss man auf die Terminal schauen, die der Regel folgen können (FOLLOW).

$A \to \alpha$	PREDICT	EPS(α)	FIRST(α)	FOLLOW(A)
$S \to term \dashv$	Int, Ident, (	0	Int, Ident,(	
$term \to factor\ factor\_tail$	Int, Ident, (	0	Int, Ident, (	), ⊣
$factor\_tail \to +\ factor\_tail$ $factor\_tail \to \epsilon$	+), ⊣	0 1	+ ∅	), ⊣
$factor \to \text{Int}$ $factor \to \text{trans}(term)$	Int Ident, (	0 0	Int Ident, (	+,), ⊣
$trans \to \text{Ident}$ $trans \to \epsilon$	Ident (	0 1	Ident ∅	(

⇒ In der Übung: konkreter Algorithmus für EPS, FIRST, FOLLOW

LL(1)-Probleme: Gemeinsame Präfixe

- **Erinnerung:** Nur, wenn alle PREDICT-Mengen mit der gleichen linken Seite (=A-Produktionen) schnittfrei sind, ist die Grammatik LL(1).
- Oft haben zwei A-Produktionen das gleiche Präfix und ein Look-Ahead von einem Token würde nicht reichen.

stmt → Ident := *expr*	PREDICT ={Ident}
stmt → Ident (*argument_list*)	PREDICT ={Ident}

- Mittels *Linksfaktorisierung* kann man das Problem umgehen.

stmt → Ident *stmt_tail*	
stmt_tail → := *expr*	PREDICT ={:=}
stmt_tail → (*argument_list*)	PREDICT ={(}

LL(1)-Probleme: Gemeinsame Präfixe

- **Erinnerung:** Nur, wenn alle PREDICT-Mengen mit der gleichen linken Seite (=A-Produktionen) schnittfrei sind, ist die Grammatik LL(1).
- Oft haben zwei A-Produktionen das gleiche Präfix und ein Look-Ahead von einem Token würde nicht reichen.

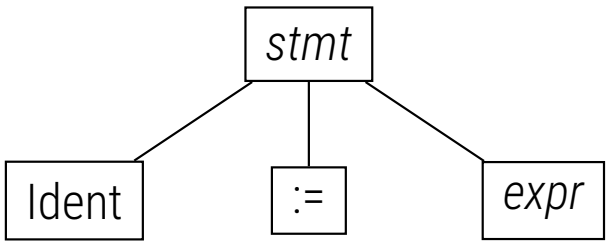
  stmt → Ident := *expr* PREDICT ={Ident}

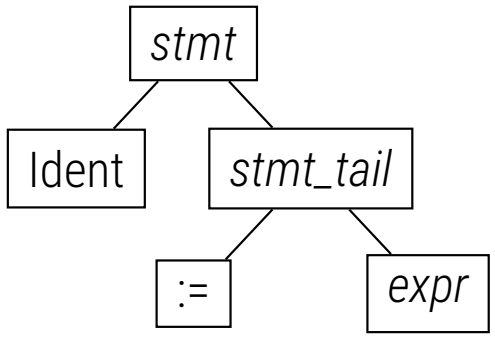
  stmt → Ident (*argument_list*) PREDICT ={Ident}
- Mittels *Linksfaktorisierung* kann man das Problem umgehen.

  stmt → Ident *stmt_tail*

  stmt_tail → := *expr* PREDICT ={:=}

  stmt_tail → (*argument_list*) PREDICT ={(}
- Linksfaktorisierte Grammatik ist nicht äquivalent! (Syntaxbäume)





LL(1)-Probleme: Linksrekursion

- LL(1) verbietet, dass eine Regel auf der linken Seite rekursiv ist.

  intAdd $\rightarrow$ Int PREDICT ={Int}

  intAdd $\rightarrow$ *intAdd* + Int PREDICT ={Int}

- **Informell**: Der Parser mit rekursivem Abstieg weiß nicht, welche Regel er anwenden soll, wenn er einen Int sieht bzw. kommt in eine endlose Rekursion, wenn er die zweite Regel auswählt.

```
def addInt(token_stream):
    if token_stream.peek() == TOK_INT:
       addInt(token_stream)
    ...
```

LL(1)-Probleme: Linksrekursion

- LL(1) verbietet, dass eine Regel auf der linken Seite rekursiv ist.

  $intAdd \rightarrow$ Int PREDICT ={Int}

  $intAdd \rightarrow intAdd$ + Int PREDICT ={Int}

- **Informell**: Der Parser mit rekursivem Abstieg weiß nicht, welche Regel er anwenden soll, wenn er einen Int sieht bzw. kommt in eine endlose Rekursion, wenn er die zweite Regel auswählt.

```
def addInt(token_stream):
    if token_stream.peek() == TOK_INT:
       addInt(token_stream)
    ...
```

- **Definition**(Linksrekursiv): Mind. eine Regel hat folgende Ableitung.

$$A \Longrightarrow^* A\alpha$$

LL(1)-Probleme: Linksrekursion

- LL(1) verbietet, dass eine Regel auf der linken Seite rekursiv ist.

  intAdd $\rightarrow$ Int PREDICT ={Int}

  intAdd $\rightarrow$ *intAdd* + Int PREDICT ={Int}

- **Informell**: Der Parser mit rekursivem Abstieg weiß nicht, welche Regel er anwenden soll, wenn er einen Int sieht bzw. kommt in eine endlose Rekursion, wenn er die zweite Regel auswählt.

```
def addInt(token_stream):
    if token_stream.peek() == TOK_INT:
       addInt(token_stream)
    ...
```

- **Definition**(Linksrekursiv): Mind. eine Regel hat folgende Ableitung.

$$A \Longrightarrow^* A\alpha$$

- „Reparatur“ nur manchmal möglich. (LL(1)$\ll$LR(1))

  intAdd $\rightarrow$ Int + *intAdd*

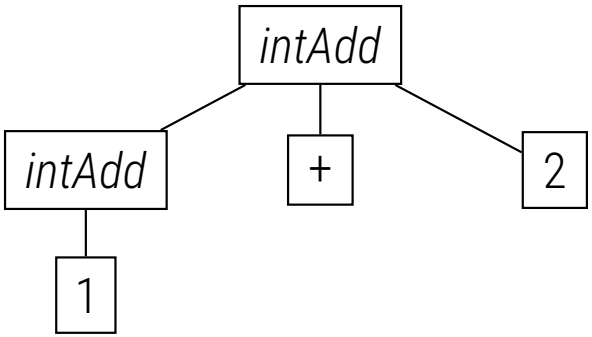
  intAddTail $\rightarrow$ Int

$\Rightarrow$ Umformung zur Rechtsrekursion + Linksfaktorisierung

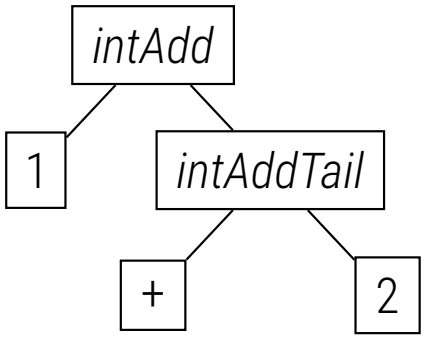
LL(1)-Probleme: „Hässliche“ Syntaxbäume

- Faktorisierung und tail-Umformung zerstören Lokalität des Syntaxbaums.

Mit Linksrekursion:



Ohne Linksrekursion:



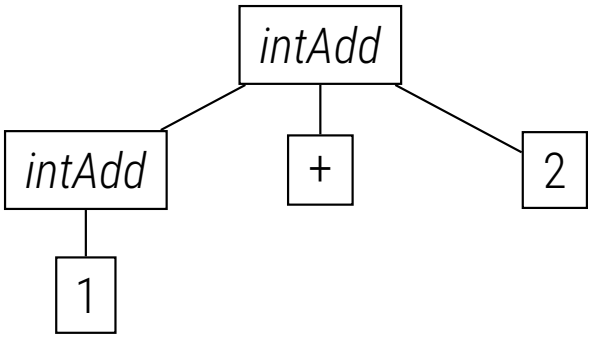
Mit Linksrekursion reicht oft Propagation alleinstehender Blätter.

- Parsergeneratoren können diese Probleme verstecken. ($\Rightarrow$ `antlr`)

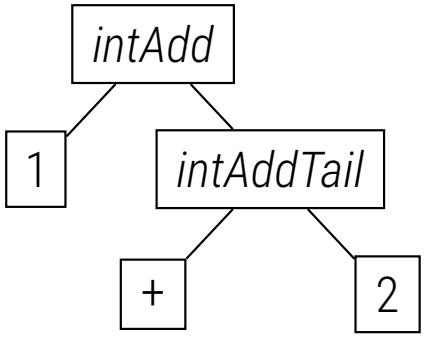
LL(1)-Probleme: „Hässliche“ Syntaxbäume

- Faktorisierung und tail-Umformung zerstören Lokalität des Syntaxbaums.

Mit Linksrekursion:



Ohne Linksrekursion:

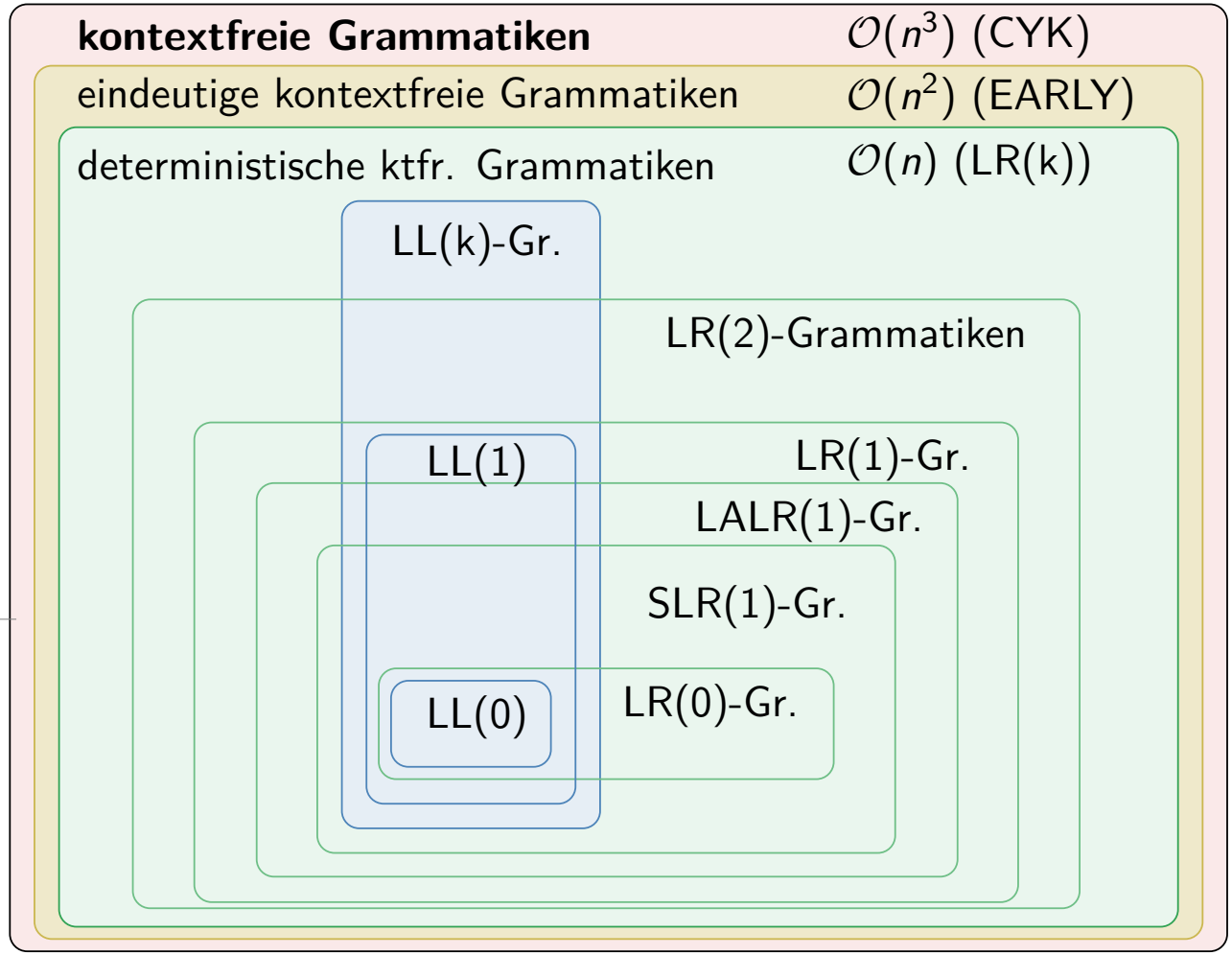


Mit Linksrekursion reicht oft Propagation alleinstehender Blätter.

- Parsergeneratoren können diese Probleme verstecken. (⇒ `antlr`)

- **Alternativ**: Verwendung der mächtigeren Sprachklasse LR(1).
  - LR(1) erlaubt gemeinsame Präfixe und Linksrekursion.
  - Konstruktion von LR(1) Parsern ist komplexer (⇒ Parsergenerator)
  - Parser sind Tabellen-gestützte Automaten mit explizitem Stapelspeicher.

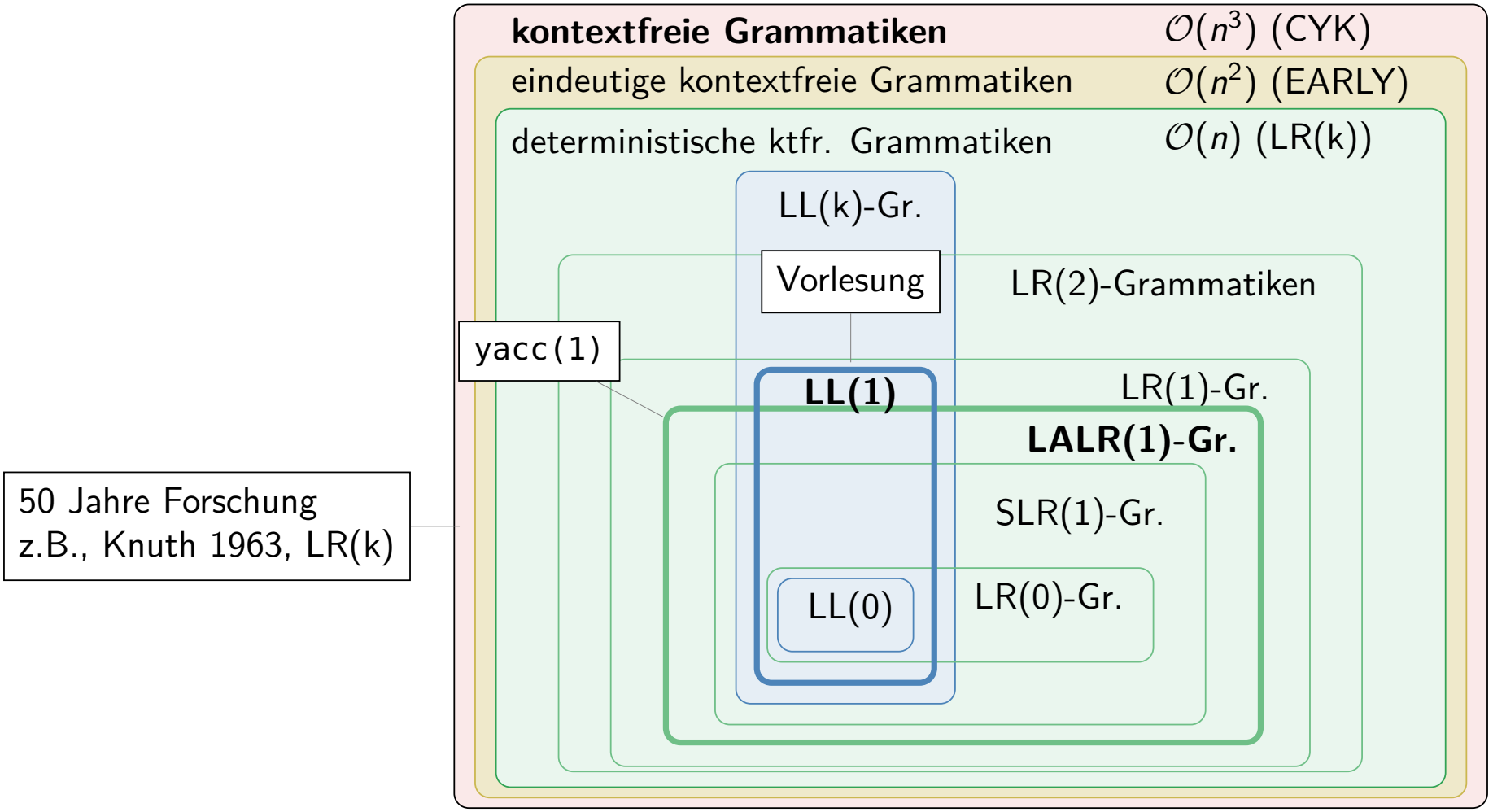
Einordnung von kontextfreien Grammatiken



50 Jahre Forschung
z.B., Knuth 1963, LR(k)

- Theorie von kontextfreien Grammatiken und Parsern ist gut erforscht.

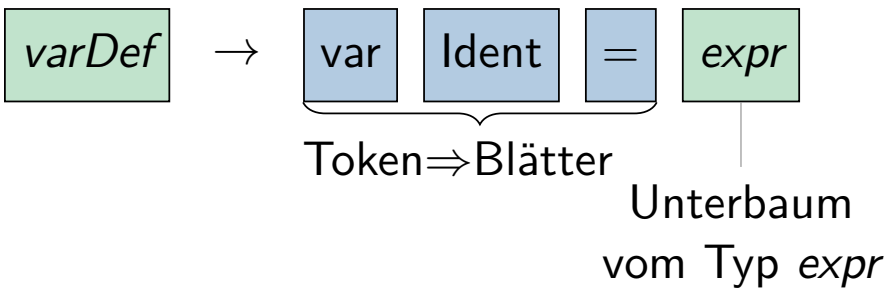
Einordnung von kontextfreien Grammatiken

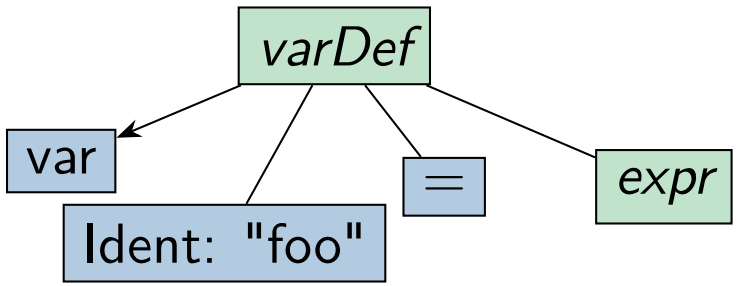


- Theorie von kontextfreien Grammatiken und Parsern ist gut erforscht.
- Moderne Generatoren akzeptieren beinahe jede ktfr. detr. Grammatik und liefern gute Fehlermeldungen. **Nutzen Sie diese!**

Ableitungsbäume

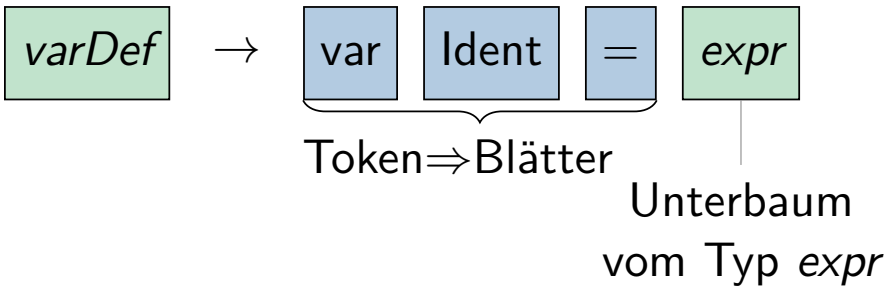
- **Bisher:** Wir haben angenommen, dass der Parser ”magisch” einen benutzbaren Syntaxbaum erzeugt. Aber nach welchen Regeln?
- **Ableitungsbaum**: Jede Anwendung einer Regel wird zu einem Knoten. Die Blätter überdecken alle Token.

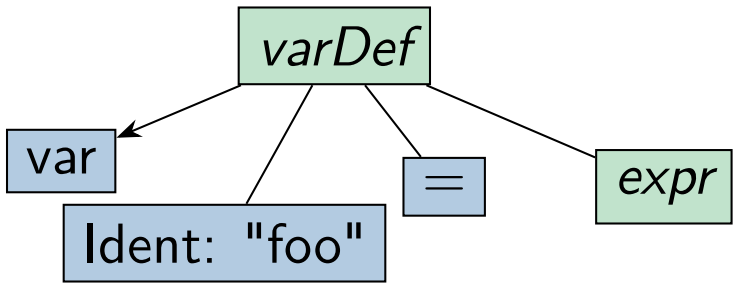




Ableitungsbäume

- **Bisher:** Wir haben angenommen, dass der Parser "magisch" einen benutzbaren Syntaxbaum erzeugt. Aber nach welchen Regeln?
- **Ableitungsbaum**: Jede Anwendung einer Regel wird zu einem Knoten. Die Blätter überdecken alle Token.



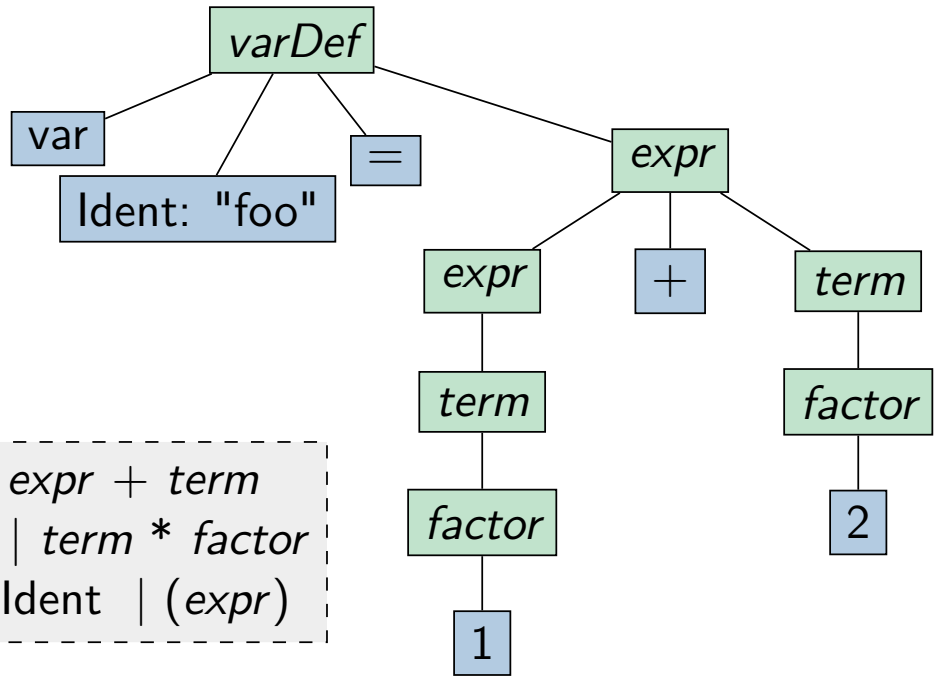


- Ein erzeugter Parser kann den Ableitungsbaum recht einfach liefern.

```
def varDef(token_stream):
    if token_stream.peek() == TOK_VAR:
        # varDef -> var Ident = expr
        ...
        expr_tree = expr(token_stream)
        # Ableitungsbaum als Python-Tupel
        return ("varDef", t_var, t_id, t_eq, expr_tree)
```

Abstrakte Syntaxbaum

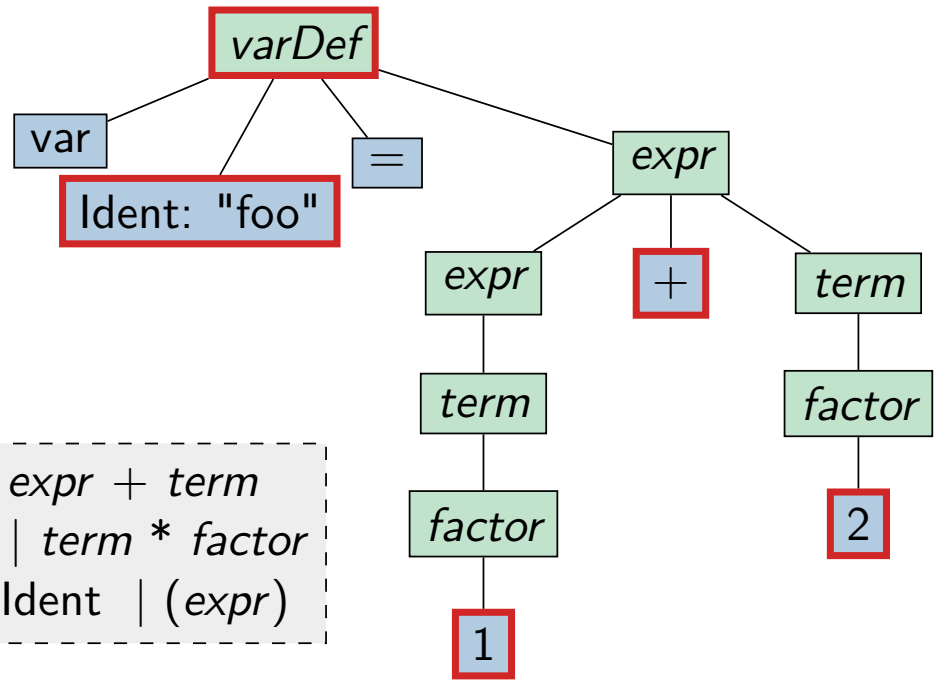
- Oft ist der Ableitungsbaum zu detailliert.
  - Viele Token sind nur für den Parser (Klammerung).
  - Regelumformungen, um die Grammatik parsebar zu machen (Faktorisierung)
  - Präzedenz von Operatoren wird über hierarchische Regeln ausgedrückt.



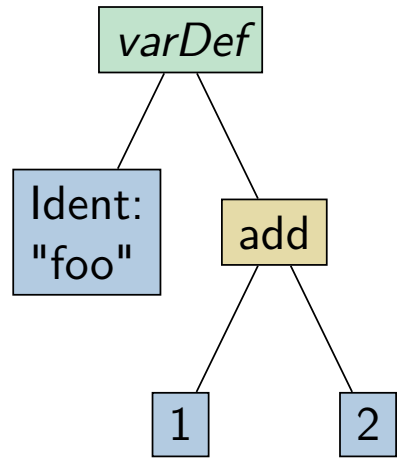
$$
\begin{aligned}
&\mathit{expr} \rightarrow \mathit{term} \mid \mathit{expr} + \mathit{term} \\
&\mathit{term} \rightarrow \mathit{factor} \mid \mathit{term} * \mathit{factor} \\
&\mathit{factor} \rightarrow \mathsf{Int} \mid \mathsf{Ident} \mid (\mathit{expr})
\end{aligned}
$$

Abstrakte Syntaxbaum

- Oft ist der Ableitungsbaum zu detailliert.
  - Viele Token sind nur für den Parser (Klammerung).
  - Regelumformungen, um die Grammatik parsebar zu machen (Faktorisierung)
  - Präzedenz von Operatoren wird über hierarchische Regeln ausgedrückt.



$$
\begin{aligned}
&\mathit{expr} \rightarrow \mathit{term} \mid \mathit{expr} + \mathit{term} \\
&\mathit{term} \rightarrow \mathit{factor} \mid \mathit{term} * \mathit{factor} \\
&\mathit{factor} \rightarrow \mathsf{Int} \mid \mathsf{Ident} \mid (\mathit{expr})
\end{aligned}
$$



- Der **Abstrakte Syntaxbaum** ist kondensiert auf das Wesentliche.
  - „Wesentlich“ ist abhängig von den folgenden Übersetzerschritten.
  - Zusammenfalten Unterbäumen und Erzeugung semantischer Knoten (add).

Parsingprobleme mit realen Sprachen

- Sprachen mit optionalem `else` können das *dangling-else problem* haben.

  stmt $\rightarrow$ `if` *condition then_clause else_clause* | `{` *stmt_list* `}` | …
  then_stmt $\rightarrow$ `then` *stmt*
  else_stmt $\rightarrow$ `else` *stmt* | ϵ

  - Grammatik ist nicht eindeutig! Wohin gehört bei `if-if-else` das `else`?
    Die Sprache muss definieren, welche Variante gewählt wird.

```
if A then  if B then doAB();  else doX();
```

Parsingprobleme mit realen Sprachen

- Sprachen mit optionalem `else` können das *dangling-else problem* haben.

  stmt $\rightarrow$ `if` *condition then_clause else_clause* | `{` *stmt_list* `}` | …
  then_stmt $\rightarrow$ `then` *stmt*
  else_stmt $\rightarrow$ `else` *stmt* | ϵ

  - Grammatik ist nicht eindeutig! Wohin gehört bei `if-if-else` das `else`?
    Die Sprache muss definieren, welche Variante gewählt wird.

```
if A then if B then doAB(); else doX();
```

Parsingprobleme mit realen Sprachen

- Sprachen mit optionalem `else` können das *dangling-else problem* haben.

  stmt → `if` *condition then_clause else_clause* | `{` *stmt_list* `}` | …
  then_stmt → `then` *stmt*
  else_stmt → `else` *stmt* | ϵ

  - Grammatik ist nicht eindeutig! Wohin gehört bei `if-if-else` das `else`?
    Die Sprache muss definieren, welche Variante gewählt wird.

```
if A then if B then doAB(); else doX();
```

Parsingprobleme mit realen Sprachen

- Sprachen mit optionalem `else` können das *dangling-else problem* haben.

  stmt → `if` *condition then_clause else_clause* | `{` *stmt_list* `}` | …
  then_stmt → `then` *stmt*
  else_stmt → `else` *stmt* | ϵ

  - Grammatik ist nicht eindeutig! Wohin gehört bei `if-if-else` das `else`?
    Die Sprache muss definieren, welche Variante gewählt wird.

```
if A then if B then doAB(); else doX();
```

- Oft reicht ein Token Look-Ahead nicht aus, um Konstrukte zu erkennen.
  - Java: Felder und Methoden erlauben unterschiedliche Modifer:

```
public abstract int foo = 0;
public abstract int foo() {…}
```

Syntaxfehler lässt sich nicht am „`abstract`“ erkennen.

  - Wir lassen beliebige Modifier zu, akzeptieren invalide Programme, und verschieben das Problem auf die semantische Analyse.

Parsingprobleme mit realen Sprachen: `typedef`

- Die Definition von eigenen Typnamen führt zu kontextsensitivität.

  typedef_decl → **`typedef`** *type* Ident ⇒ Ident wird ein Typname

  cast_expr → (*type_name*) expr ⇒ Referenz von Typnamen

- Ohne Kontext können wir Aufrufe und Casts nicht unterscheiden.

```
int bar;
int foo(int a) {};
...
// Funktionsname in Klammern
(foo)(bar); // = foo(bar)
```

```
int bar;
typedef char foo;
...
// Typname in Klammern
(foo)(bar); // = (char)(bar)
```

Parsingprobleme mit realen Sprachen: `typedef`

- Die Definition von eigenen Typnamen führt zu kontextsensitivität.

  typedef_decl → **`typedef`** *type* Ident ⇒ Ident wird ein Typname

  cast_expr → (*type_name*) expr ⇒ Referenz von Typnamen

- Ohne Kontext können wir Aufrufe und Casts nicht unterscheiden.

```c
int bar;
int foo(int a) {};
...
// Funktionsname in Klammern
(foo)(bar); // = foo(bar)
```

```c
int bar;
typedef char foo;
...
// Typname in Klammern
(foo)(bar); // = (char)(bar)
```

- „Lösung“: Der Lexer Hack vermeidet kontextsensitives Parsing.
  - Der Scanner führt eine Liste von bereits definierten Typnamen mit.
  - Er emittiert unterschiedliche Token für Typname und Bezeichner.
  - Nun ist allerdings der Scanner kontextsensitiv!

Parsingprobleme mit realen Sprachen: `typedef`

- Die Definition von eigenen Typnamen führt zu kontextsensitivität.

  typedef_decl → **`typedef`** *type* Ident ⇒ Ident wird ein Typname

  cast_expr → (*type_name*) expr ⇒ Referenz von Typnamen

- Ohne Kontext können wir Aufrufe und Casts nicht unterscheiden.

```c
int bar;
int foo(int a) {};
...
// Funktionsname in Klammern
(foo)(bar); // = foo(bar)
```

```c
int bar;
typedef char foo;
...
// Typname in Klammern
(foo)(bar); // = (char)(bar)
```

- „Lösung“: Der Lexer Hack vermeidet kontextsensitives Parsing.
  - Der Scanner führt eine Liste von bereits definierten Typnamen mit.
  - Er emittiert unterschiedliche Token für Typname und Bezeichner.
  - Nun ist allerdings der Scanner kontextsensitiv!

⇒ Echte Programmiersprachen sind nur auf den ersten Blick kontextfrei!

Parsing und Sicherheit

Parser als Einfallstore für Angreifer

Technische Universität Braunschweig

- Nutzer, wie auch Angreifer, schicken Anfragen als Zeichenstrom.
  - Parser extrahieren Typ, Argumente und strukturierte Daten.
  - Komplexe Protokoll- und Dateiformate (ASN.1, PDF, JPEG)
  - Die Eingaben müssen validiert und transformiert werden (XML: XSD, XSLT)
- Kleine (Programmier-)sprachen sind überall
  - Format Strings: `printf("%3$*1$.*2$f", 10, 2, 1.11111);`
  - Browser: HTML5, JavaScript, CSS, RDF, Embedded SVG, MathML, …
- Handgeschriebene für komplexe Formate neigen zu Schwachstellen.
  - DNP3: Kommunikationsprotokoll im US Stromnetz (> 100 Seiten Standard)
  - Untersuchung von 20 DNP3 Implementierung durch Fuzzing (2013-2014)
  - 31 bekannte Schwachstellen in DNP3 Parsern

Parser als Einfallstore für Angreifer

- Nutzer, wie auch Angreifer, schicken Anfragen als Zeichenstrom.
  - Parser extrahieren Typ, Argumente und strukturierte Daten.
  - Komplexe Protokoll- und Dateiformate (ASN.1, PDF, JPEG)
  - Die Eingaben müssen validiert und transformiert werden (XML: XSD, XSLT)
- Kleine (Programmier-)sprachen sind überall
  - Format Strings: `printf("%3$*1$.*2$f", 10, 2, 1.11111);`
  - Browser: HTML5, JavaScript, CSS, RDF, Embedded SVG, MathML, …
- Handgeschriebene für komplexe Formate neigen zu Schwachstellen.
  - DNP3: Kommunikationsprotokoll im US Stromnetz (> 100 Seiten Standard)
  - Untersuchung von 20 DNP3 Implementierung durch Fuzzing (2013-2014)
  - 31 bekannte Schwachstellen in DNP3 Parsern

(Benutzer-)Eingaben sind Lava! Parsergeneratoren!

Ungewollt Turing-Vollständig: Eingabevalidierung

- Syntaktische (und semantische) Analysen validieren eine Eingabe.
  - Dazu wenden sie, teils komplexe, Regeln auf die Eingabedaten an.
  - Was, wenn durch komplexe Regeln eine virtuelle Maschine entsteht?
  - Der Übersetzer selbst wird zum Sprachprozessor.
- „C++-Templates are Turing Complete“, Veldhuizen, 2003
  - Flexible Definition von Datentypen; aber auch Berechnung von π.
  - Maschinenmodell: Typen sind Objekte, Rekursive Templates die Operationen.
  - Es gibt sogar Standardbibliotheken zur C++-Template-Programmierung.

```
// Rekursion: F<N> -> N * F<N-1>
template <int N> struct F {
    enum { value = N * F<N - 1>::value };
};

// Rekursionsabbruch: F<0>
template <> struct F<0> { enum { value = 1 }; };

int main(void) { return F<4>::value; } // == 24
```

Ungewollt Turing-Vollständig: Eingabevalidierung

- Syntaktische (und semantische) Analysen validieren eine Eingabe.
  - Dazu wenden sie, teils komplexe, Regeln auf die Eingabedaten an.
  - Was, wenn durch komplexe Regeln eine virtuelle Maschine entsteht?
  - Der Übersetzer selbst wird zum Sprachprozessor.

- „C++-Templates are Turing Complete“, Veldhuizen, 2003
  - Flexible Definition von Datentypen; aber auch Berechnung von π.
  - Maschinenmodell: Typen sind Objekte, Rekursive Templates die Operationen.
  - Es gibt sogar Standardbibliotheken zur C++-Template-Programmierung.

```
// Rekursion: F<N> -> N * F<N-1>
template <int N> struct F {
    enum { value = N * F<N - 1>::value };
};

// Rekursionsabbruch: F<0>
template <> struct F<0> { enum { value = 1 }; };

int main(void) { return F<4>::value; } // == 24
```

Maschinen die π berechnen, führen auch Exploits aus.

Zusammenfassung

- Die syntaktische Analyse extrahiert die Struktur eines Programms
  - Die rekursive Natur von Bäumen eignet sich Schachtelungen zu greifen.
  - Syntaktische Korrektheit ist notwendig, jedoch nicht hinreichend.
- Scanner (Lexer) partitionieren den Zeichenstrom in einen Tokenstrom.
  - Token haben eine Klasse und eine Nutzlast: `(Int, 23)`
  - Zerteilung mittels regulärer Ausdrücke und endlichen Zustandsautomaten
- Parser ordnen die Token, mittels Grammatikregeln, in einem Baum an.
  - Die Sprachklasse LL(1) kann mittels rekursivem Abstieg erkannt werden.
  - Die PREDICT-Menge und der Look-Ahead entscheiden die nächste Regel.
  - Im Parsebaum sind alle Token Blätter und alle Regeln Knoten.
- Strukturierte Sprachen sind allgegenwärtig und sicherheitsrelevant!
  - Jede Validierung von Eingabedaten erfordert strukturelles Verstehen.
  - Formale Grammatiken und Parsergeneratoren helfen Lücken zu vermeiden.