Informatik II - Übung 3

Pascal Schärli

IloveExercise3@pascscha.ch

09.10.2020

ETH zürich

Nachbesprechung

Bild: look again by corpus delicti from the Noun Project

Sortieren

```java
public String toString() {
    String s="[";
    for (int i=0; i<numbers.length; i++) {
        if (i!=0) {
            s = s + ", ";
        }
        s = s + numbers[i];
    }
    s = s + "]";
    return s;
}
```

Sortieren

```java
private void recursiveSort(int until) {
    if (until == 0) {
        return;
    } else {
        recursiveSort(until - 1);

        int index_max = until-1;
        for (int i=until; i<numbers.length; i++) {
           if (numbers[i] > numbers[index_max]) {
                index_max = i;
            }
        }
        swap(until - 1, index_max);
    }
}
```

Binärbäume als Arrays

```
1   private static void checkTree(char[] array) throws IllegalArgumentException
2   {
3     if(array.length==0)
4       throw new IllegalArgumentException("empty tree");
5     for(int i=0;i<array.length;i++) {
6       if((array[i]!=' ')&&(array[father(i)]==' '))
7         throw new IllegalArgumentException("empty father");
8     }
9   }
```

Binärbäume als Arrays

```java
private String toString(int node, String indentation){
  String res="";
  if(tree[node]==' ')
    return "";

  // add the first one to the node
  res=indentation+Character.toString(tree[node])+"\n";

  //Check if the child node is still inside
  if(leftChild(node)<tree.length&&leftChild(node)!=' ') {
      res=res+toString(leftChild(node),indentation+" ");
    }

    //also check if there is a right child
    if((rightChild(node)<tree.length)&&(rightChild(node)!=' ')) {
      res=res+toString(rightChild(node),indentation+" ");
    }

  return res;
}
```

Vorlesung

String vs StringBuffer

- Klasse **String**
  - immutable: Unveränderlich
  - final -> Operationen können gut optimiert werden (Zb. multithreading)
- Klasse StringBuffer
  - mutable: veränderbar
  - Mehr Platz
  - Gewisse Operationen sind aufwendiger

String vs StringBuffer

```
1 String myString = "hello";
2 myString = myString + " world";
```

"hello"

" world"

"hello world"

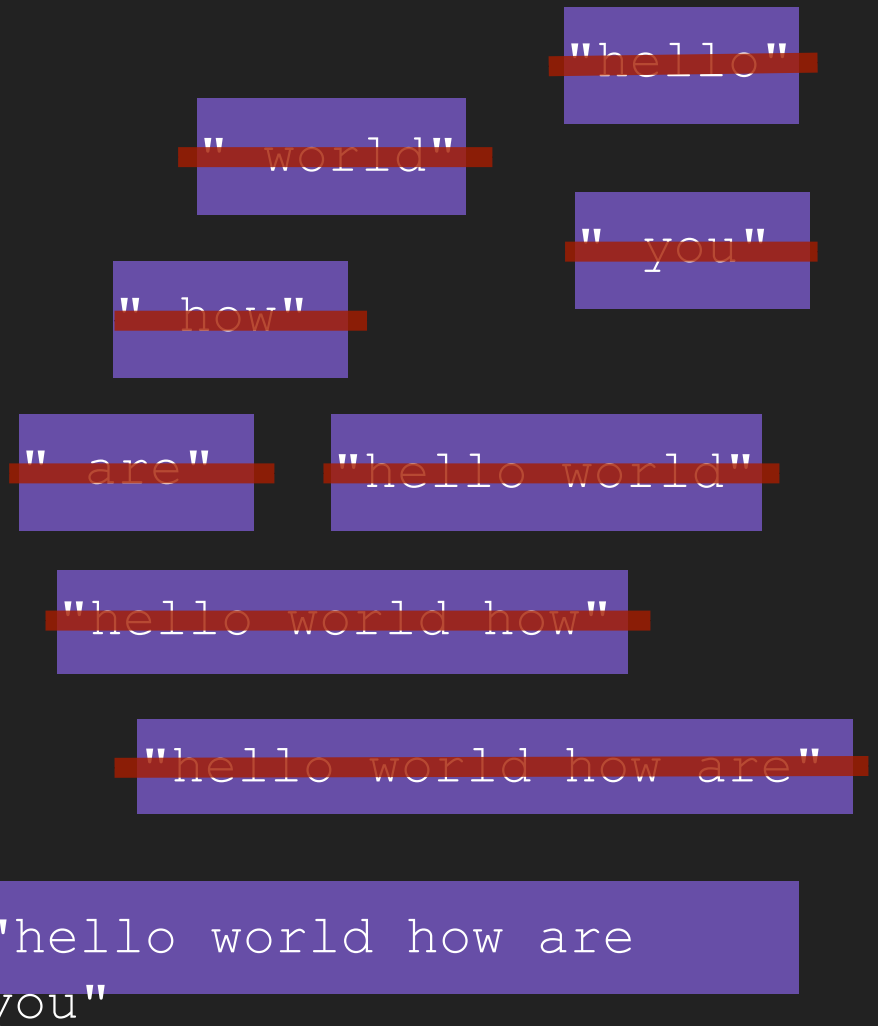
```
1 StringBuffer myStringBuffer = "hello";
2 myStringBuffer.append(" world");
```

"hello world"

" world"

Garbage Collector

```
1 String myString = "hello";
2 myString = myString+" world";
3 myString = myString+" how";
4 myString = myString+" are";
5 myString = myString+" you";
6 myString = myString+" today";
```



Garbage Collector

```
" today"
```

```
1 String myString = "hello";
2 myString = myString+" world";
3 myString = myString+" how";
4 myString = myString+" are";
5 myString = myString+" you";
6 myString = myString+" today";
```

```
"hello world how are you today"
```

```
"hello world how are you"
```

Schleifeninvarianten

Schleifeninvarianten sind ein Weg um die Korrektheit einer Funktion zu beweisen.

```
1  static int divide(int in, int div){
2
3      assert(in >= 0 && div >= 0);
4
5      int n = in;
6      int out = 0;
7
8      // [Vor Schleife]
9      while(n >= div){
10         // [Vor Schleifenkörper]
11     out += 1;
12     n -= div;
13     // [Nach Schleifenkörper]
14     }
15     // [Nach Schleife]
16
17     return out;
18 }
```

Schleife

Schleifen Körper

Definition:

1. Falls die Schleifeninvariante vor dem Schleifenkörper gilt, so gilt sie auch nach dem Schleifenkörper.
2. Wenn man dann zeigen kann, dass die Invariante vor der Schleife gilt, so gilt sie auch nach der Schleife.

Schleifeninvarianten

Schleifeninvarianten sind ein Weg um die Korrektheit einer Funktion zu beweisen.

```
1  static int f(int i, int j) {
2
3      assert(i >= 0 && j >= 0);
4
5      int u = i;
6      int z = 0;
7
8      // [Vor Schleife]
9      while (u > 0){
10         // [Vor Schleifenkörper]
11         z = z + j;
12         u = u - 1;
13         // [Nach Schleifenkörper]
14     }
15     // [Nach Schleife]
16
17     return z;
18 }
```

Schleife

Schleifen Körper

Definition:

1. Falls die Schleifeninvariante vor dem Schleifenkörper gilt, so gilt sie auch nach dem Schleifenkörper.
2. Falls man zeigen kann, dass die Invariante vor der Schleife gilt, so gilt sie auch nach der Schleife.

Schleifeninvarianten

1. Beweise, dass dies eine korrekte Scheifeninvariante ist:

`in = n + out * div && n >= 0 && (in - n) % div = 0`

```
1  static int divide(int in, int div)
2
3      assert(in >= 0 && div >= 0);
4
5      int n = in;
6      int out = 0;
7
8      // [Vor Schleife]
9      while(n >= div){
10         // [Vor Schleifenkörper]
11         out += 1;
12         n -= div;
13         // [Nach Schleifenkörper]
14     }
15     // [Nach Schleife]
16
17     return out;
18 }
```

Vor Schleifenkörper

1. $in = n_n + out_n \cdot div$
   && $n_n >= 0$
   && $(in - n_n) \% div = 0$
2. $n_n >= div$

(Schleifeninvariante)

(Sonst wären wir nicht in den while-loop gekommen)

Nach Schleifenkörper

1. $in = (n_{n+1} + div) + (out_{n+1} - 1) * div$
   $\Leftrightarrow in = n_{n+1} + out_{n+1} * div$
2. (n_{n+1}+div) >= div
   $\Leftrightarrow$ n >= 0
3. $(in - (n_{n+1} + div)) \% div = 0$
   $\Leftrightarrow (in - n_{n+1}) \% div = 0$

$\Rightarrow$ Schleifeninvariante gilt auch nach Schleifenkörper

Schleifeninvarianten

1. Beweise, dass dies eine korrekte Scheifeninvariante ist:

`in = n + out * div && n >= 0 && (in - n) % div = 0`

```
1  static int divide(int in, int div)
2
3      assert(in >= 0 && div >= 0);
4
5      int n = in;
6      int out = 0;
7
8      // [Vor Schleife]
9      while(n >= div){
10         // [Vor Schleifenkörper]
11         out += 1;
12         n -= div;
13         // [Nach Schleifenkörper]
14     }
15     // [Nach Schleife]
16
17     return out;
18 }
```

Vor Schleifenkörper

1. $in = n_n + out_n \cdot div$
   && $n_n >= 0$
   && $(in - n_n) \% div = 0$ (Schleifeninvariante)
2. $n_n >= div$ (Sonst wären wir nicht in den while-loop gekommen)

Nach Schleifenkörper

1. $in = (n_{n+1} + div) + (out_{n+1} - 1) * div$
   $\Leftrightarrow in = n_{n+1} + out_{n+1} * div$
2. (n_{n+1}+div) >= div
   $\Leftrightarrow$ n >= 0
3. $(in - (n_{n+1} + div)) \% div = 0$
   $\Leftrightarrow (in - n_{n+1}) \% div = 0$

$\Rightarrow$ Schleifeninvariante gilt auch nach Schleifenkörper

Schleifeninvarianten

2. Zeige, dass die Schleifeninvariante vor der Schleife gilt

`in = n + out * div && n >= 0 && (in - n) % div = 0`

1. 2. 3.

```
1  static int divide(int in, int div)
2
3      assert(in >= 0 && div >= 0);
4
5      int n = in;
6      int out = 0;
7
8      // [Vor Schleife]
9      while(n >= div){
10         // [Vor Schleifenkörper]
11         out += 1;
12         n -= div;
13         // [Nach Schleifenkörper]
14     }
15     // [Nach Schleife]
16
17     return out;
18 }
```

1. in = n + out * div
   in = in + 0 * div
   in = in ✓

2. n >= 0
   in >= 0 ✓

3. (in - n) % div = 0
   (in - in) % div = 0
   0 % div = 0 ✓

Schleifeninvarianten

3. Daraus folgt, dass sie auch nach der Schleife gilt.

`in = n + out * div && n >= 0 && (in - n) % div = 0`

```
1  static int divide(int in, int div)
2
3      assert(in >= 0 && div >= 0);
4
5      int n = in;
6      int out = 0;
7
8      // [Vor Schleife]
9      while(n >= div){
10         // [Vor Schleifenkörper]
11         out += 1;
12         n -= div;
13         // [Nach Schleifenkörper]
14     }
15     // [Nach Schleife]
16
17     return out;
18 }
```

1. `in = n + out * div`
   `&& n >= 0`
   `&& (in - n) % div = 0`
2. n < div

(Schleifeninvariante)

(Sonst wären wir nicht aus dem while-loop gekommen)

Daraus folgt:

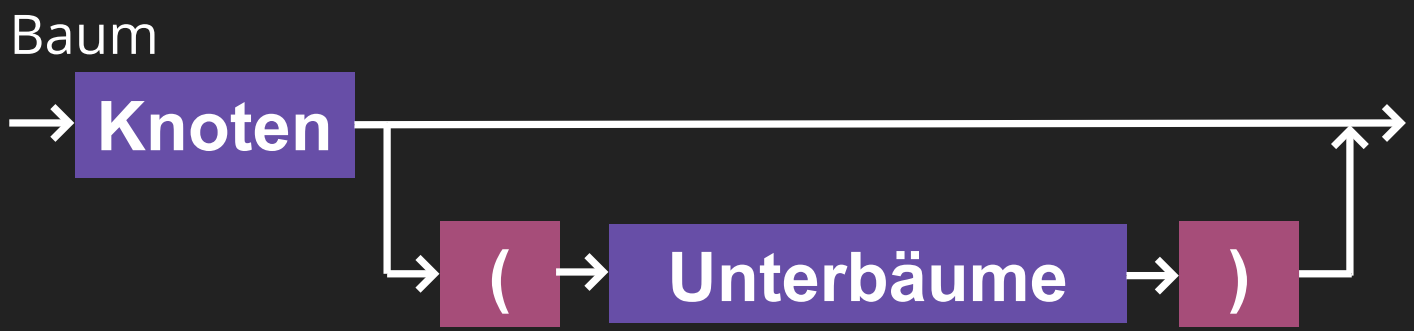
1. 0 <= n < div
   && (in - n) % div = 0
   && out = (in - n) / div
2. ⇒ out = ⌊in / div⌋

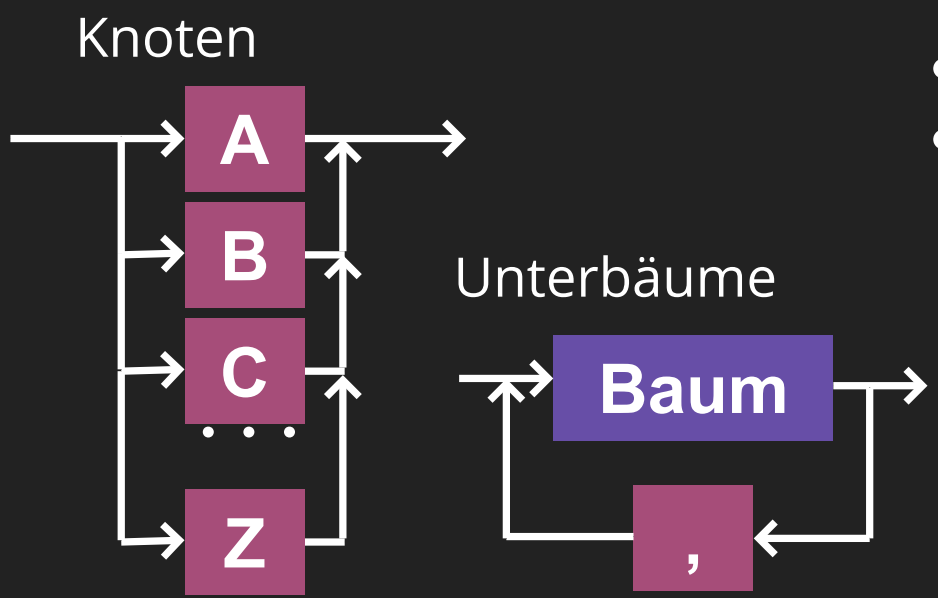
Schleifeninvarianten

Wie findet man eine Schleifeninvariante?

- Es gibt keinen Weg eine Schleifeninvariante zu finden, welche den gewünschten Beweis ermöglicht.
- Es ist nur möglich zu prüfen ob eine Gegebene Invariante funktioniert.
- Probiert eine Invariante aus, schaut ob der Beweis funktioniert.
- Falls nicht, passt eure Invariante an und versucht es nochmals, bis es funktioniert.

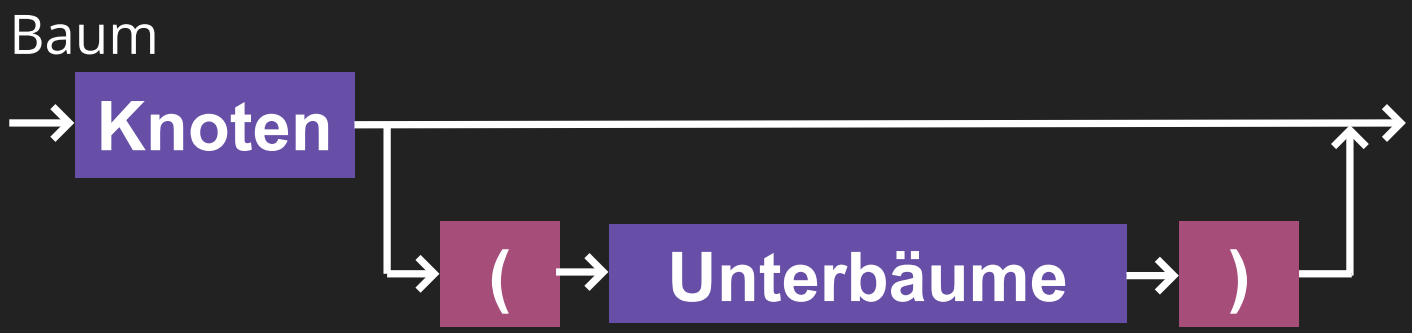
Syntaxdiagramme

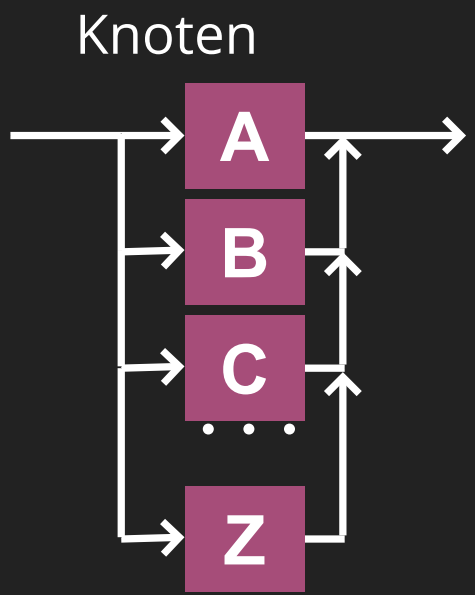


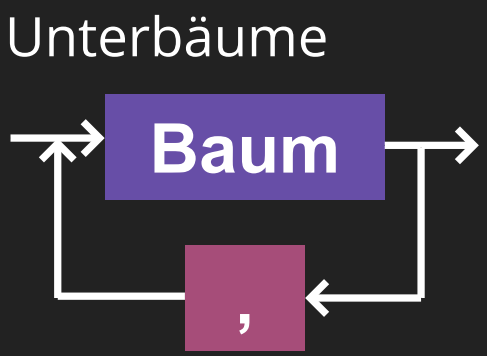


- Gleiches Konzept wie BNF
- Ein Ausdruck ist erzeugbar, wenn es einen Weg durch das Diagramm gibt, welcher diesen beschreibt

Syntaxdiagramme

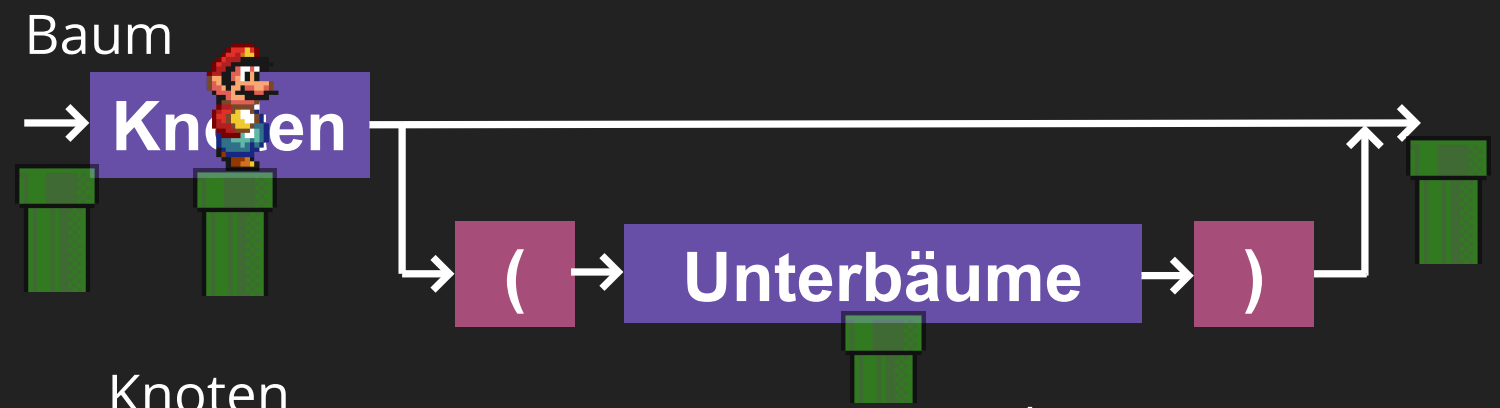


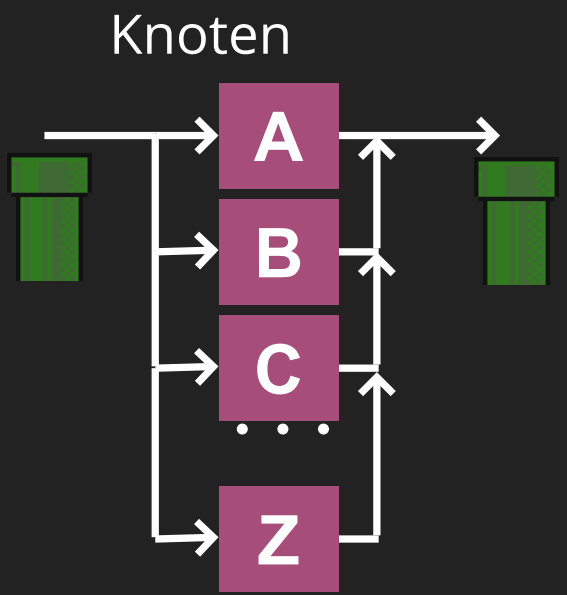


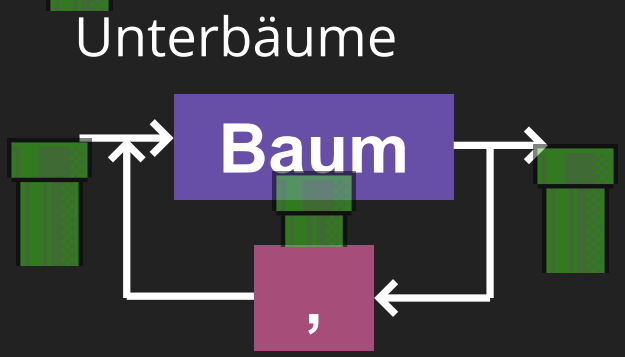



```
A(B(C,D))
```

Syntaxdiagramme

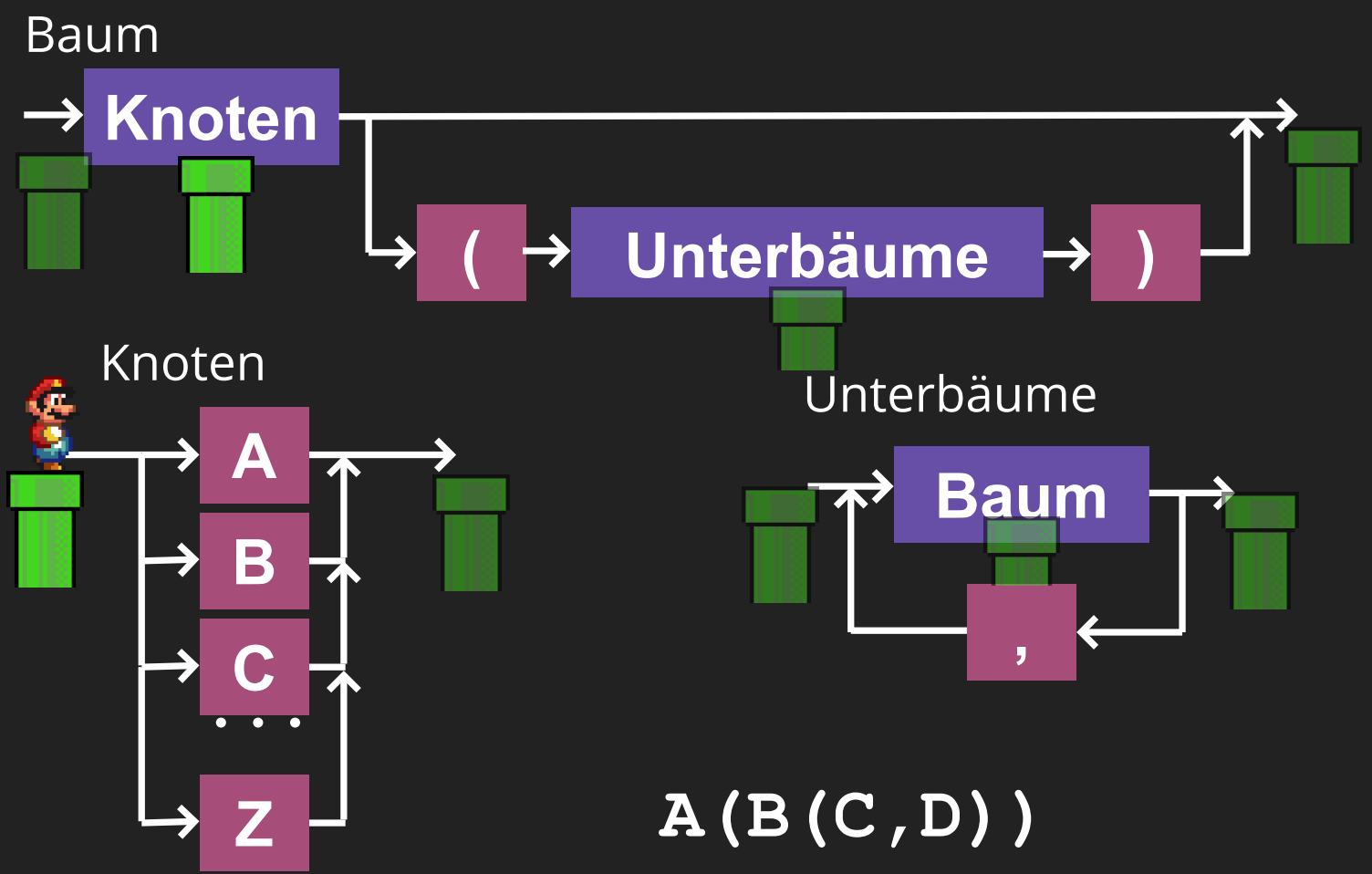






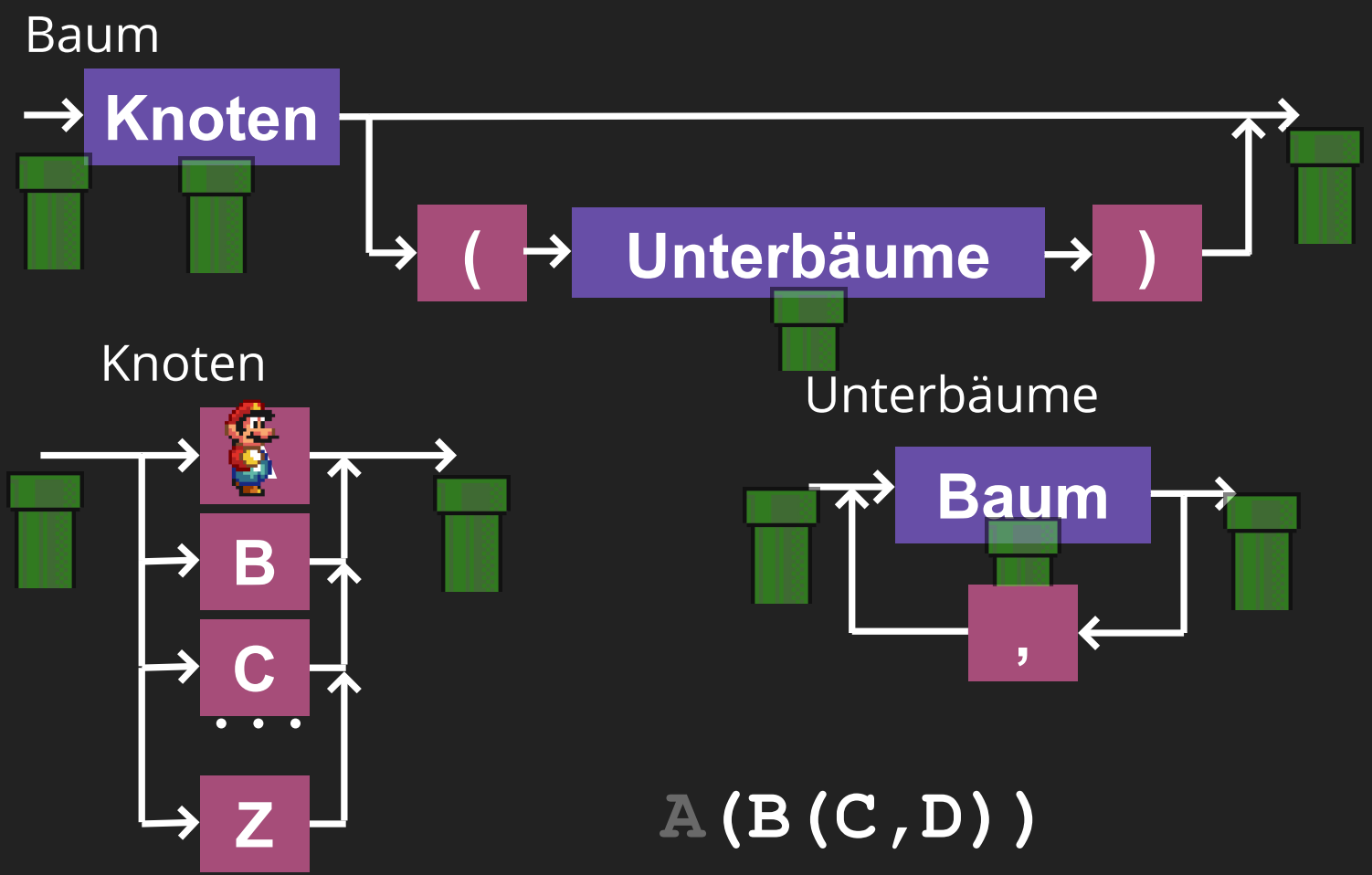

```
A(B(C,D))
```

Syntaxdiagramme



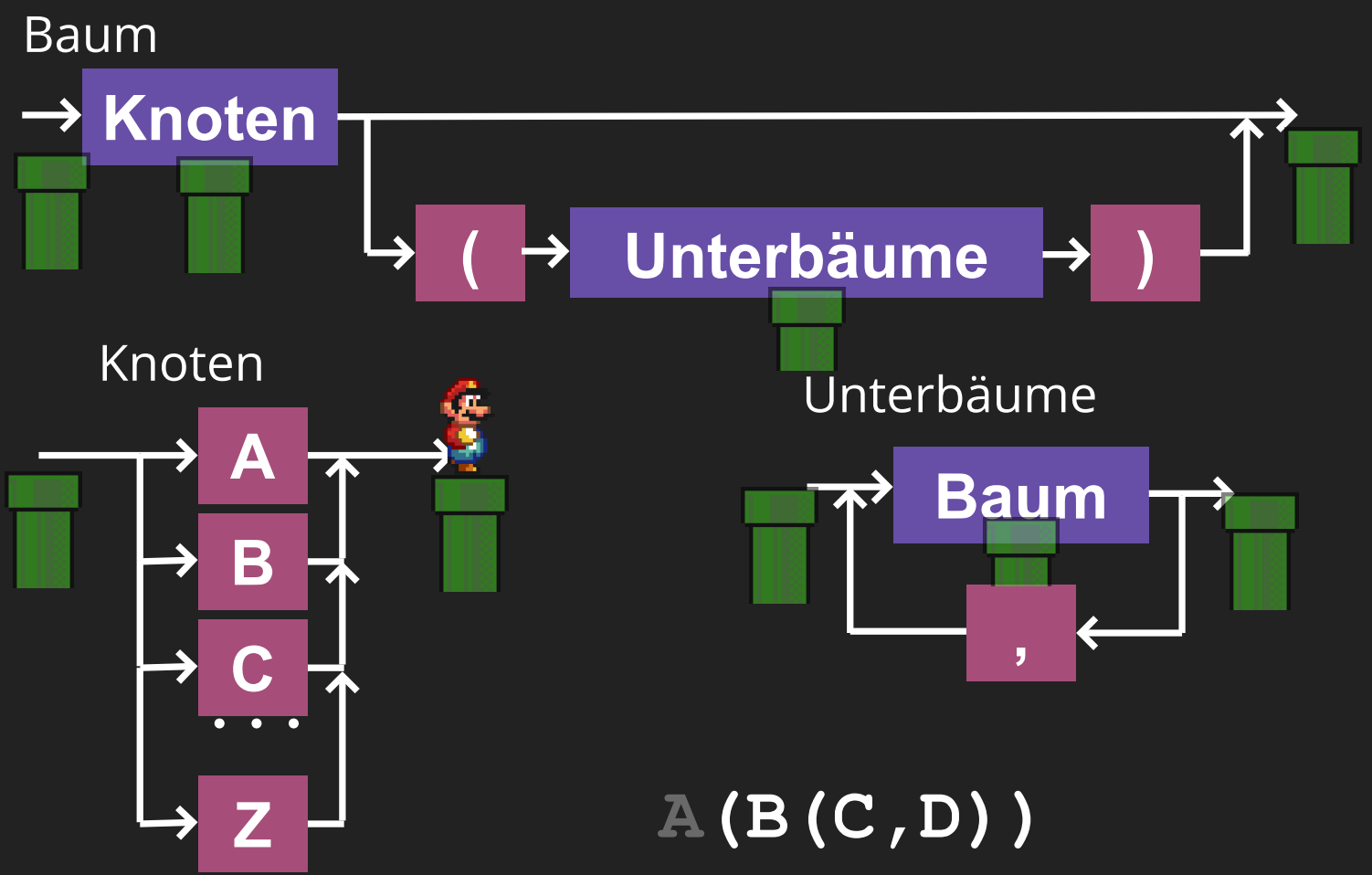

```
A(B(C,D))
```

Syntaxdiagramme



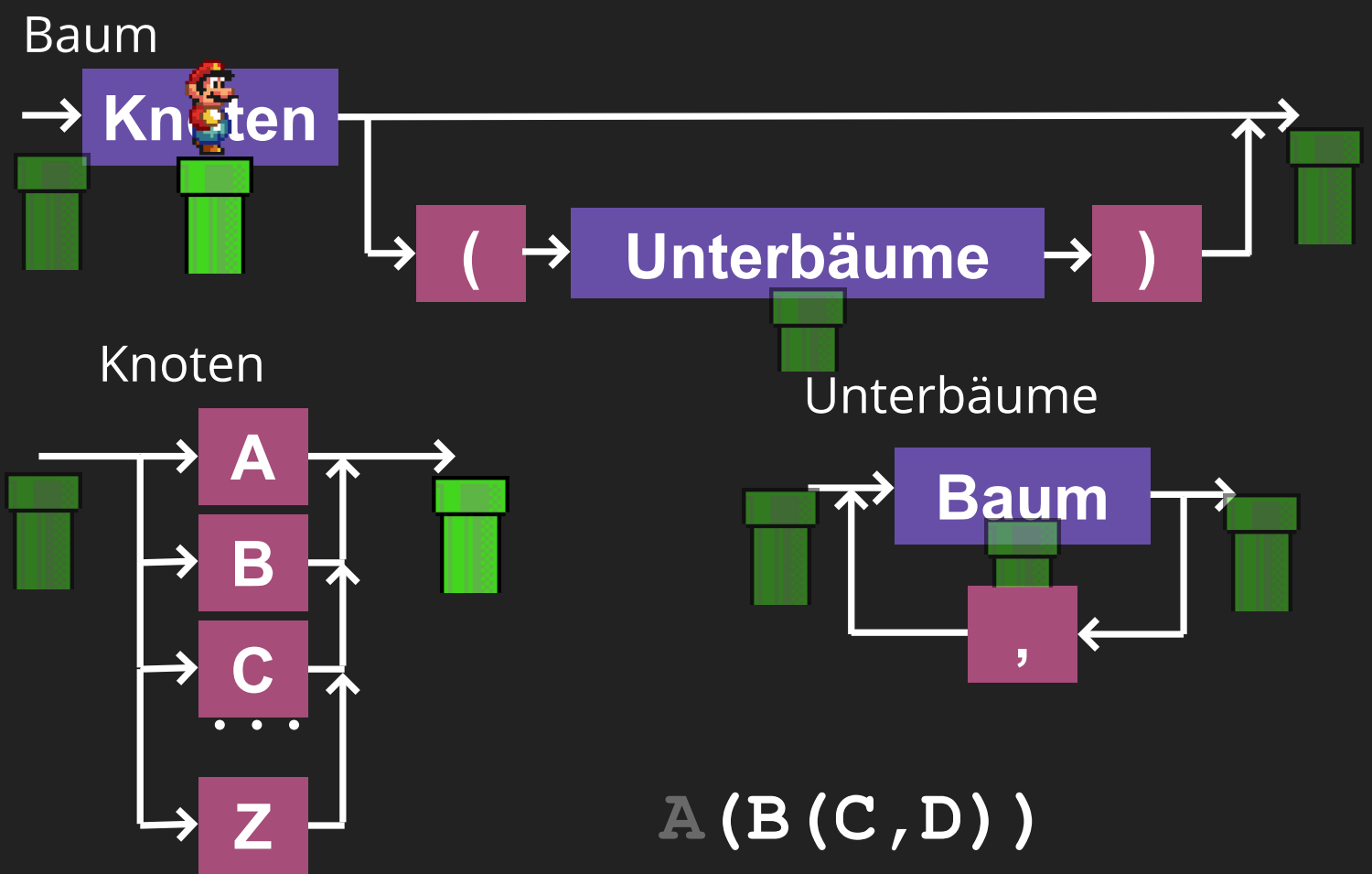

```
A(B(C,D))
```

Syntaxdiagramme



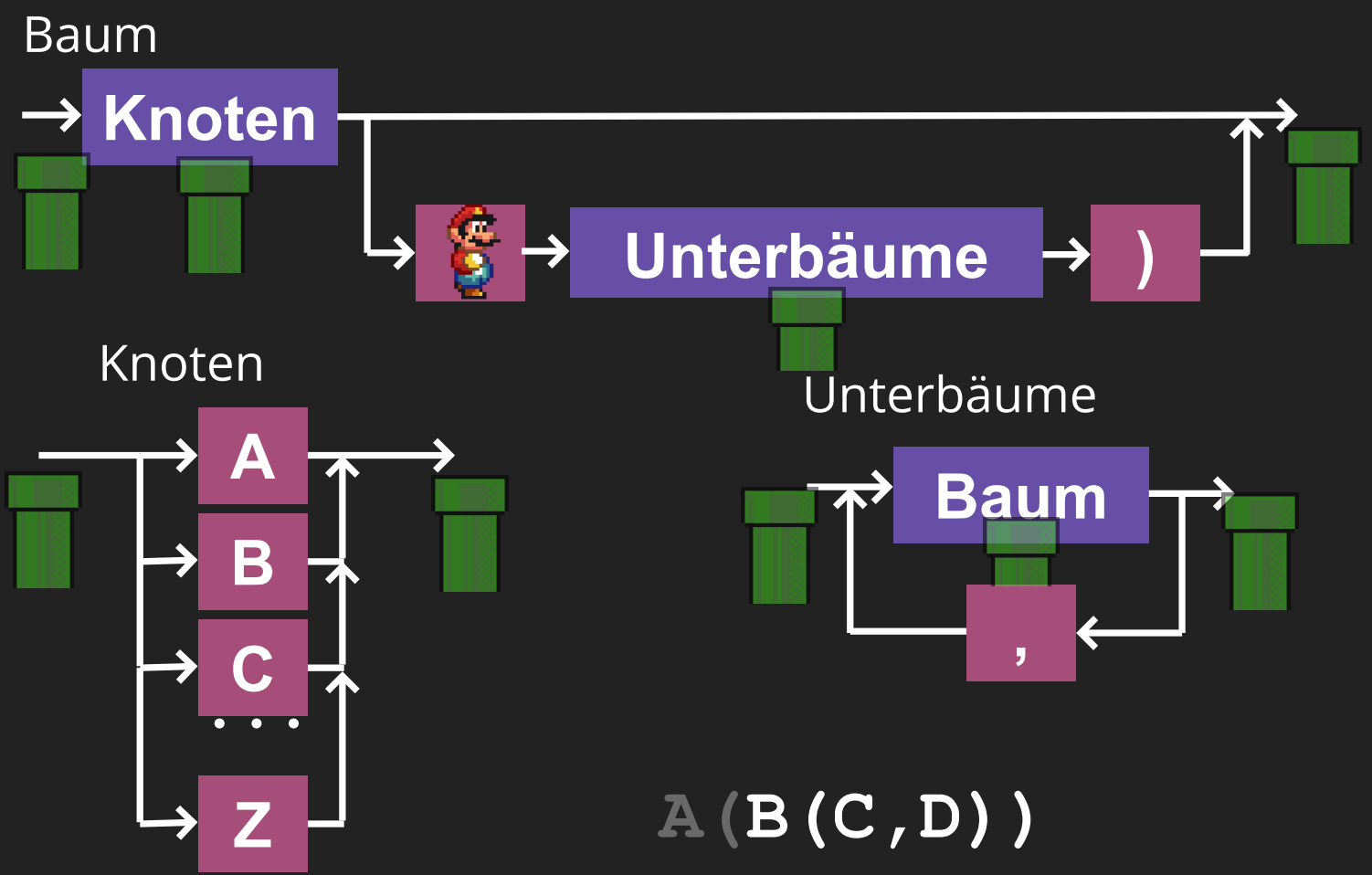

```
A(B(C,D))
```

Syntaxdiagramme



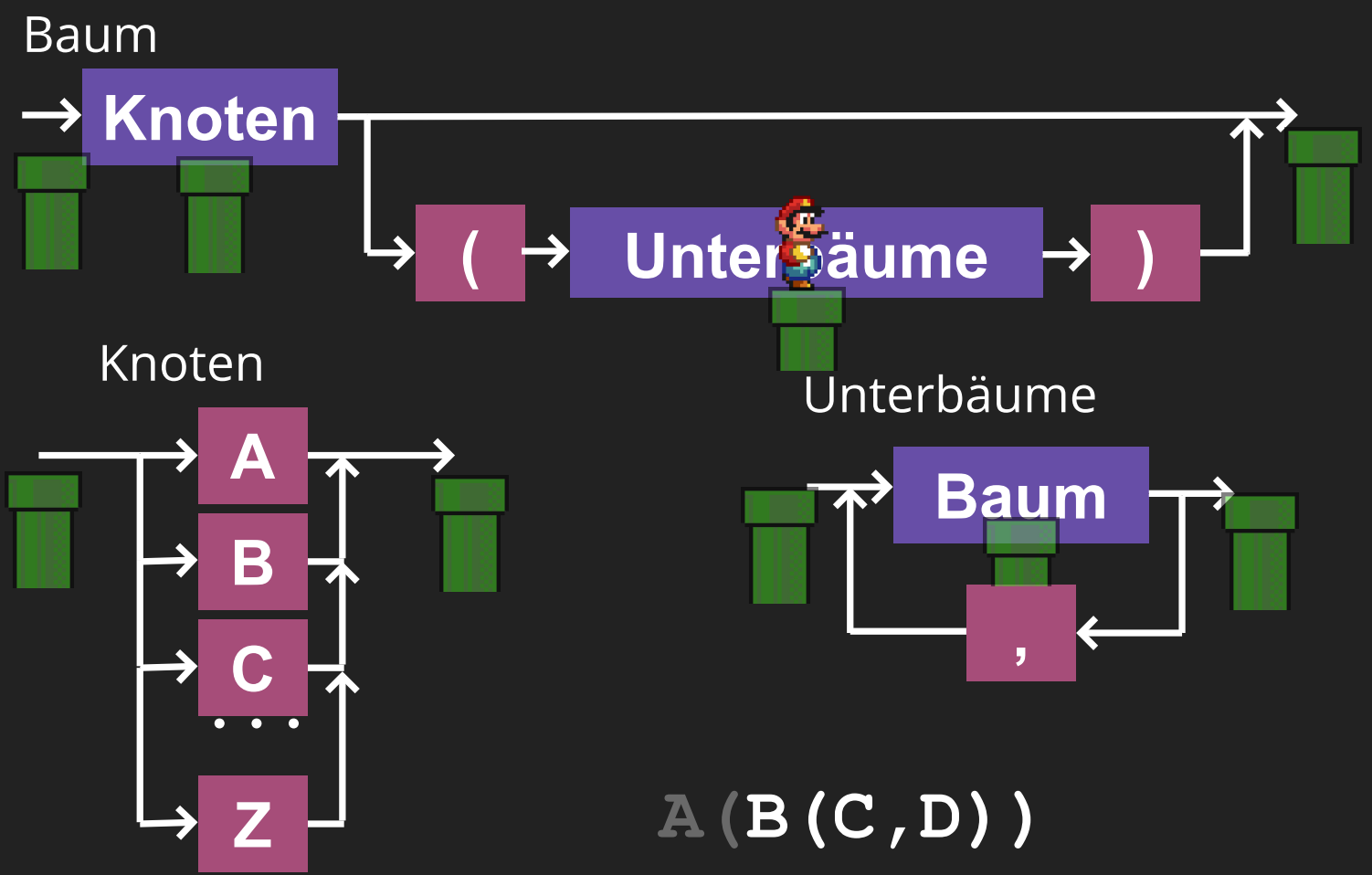

```
A(B(C,D))
```

Syntaxdiagramme



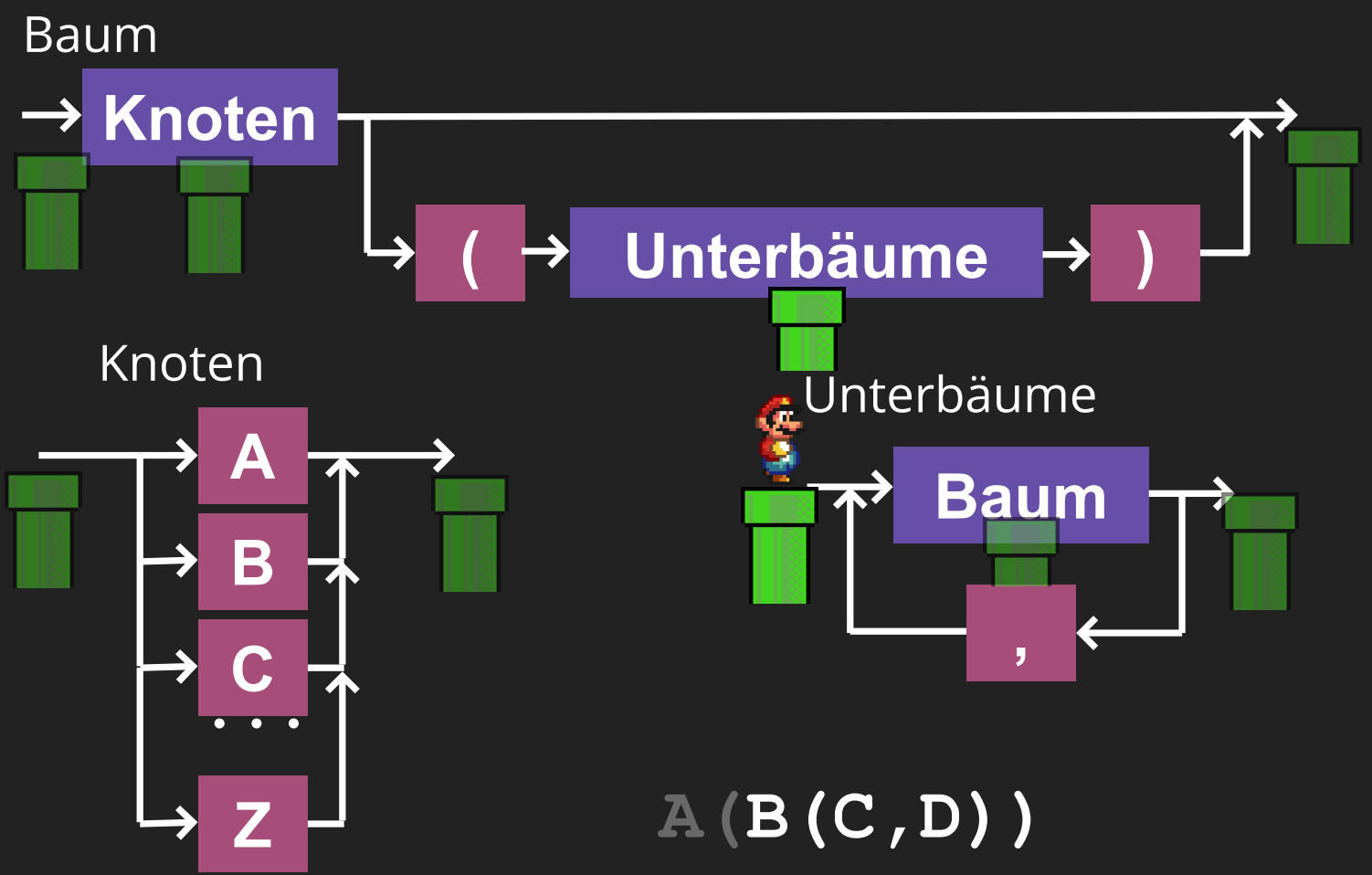

```
A(B(C,D))
```

Syntaxdiagramme



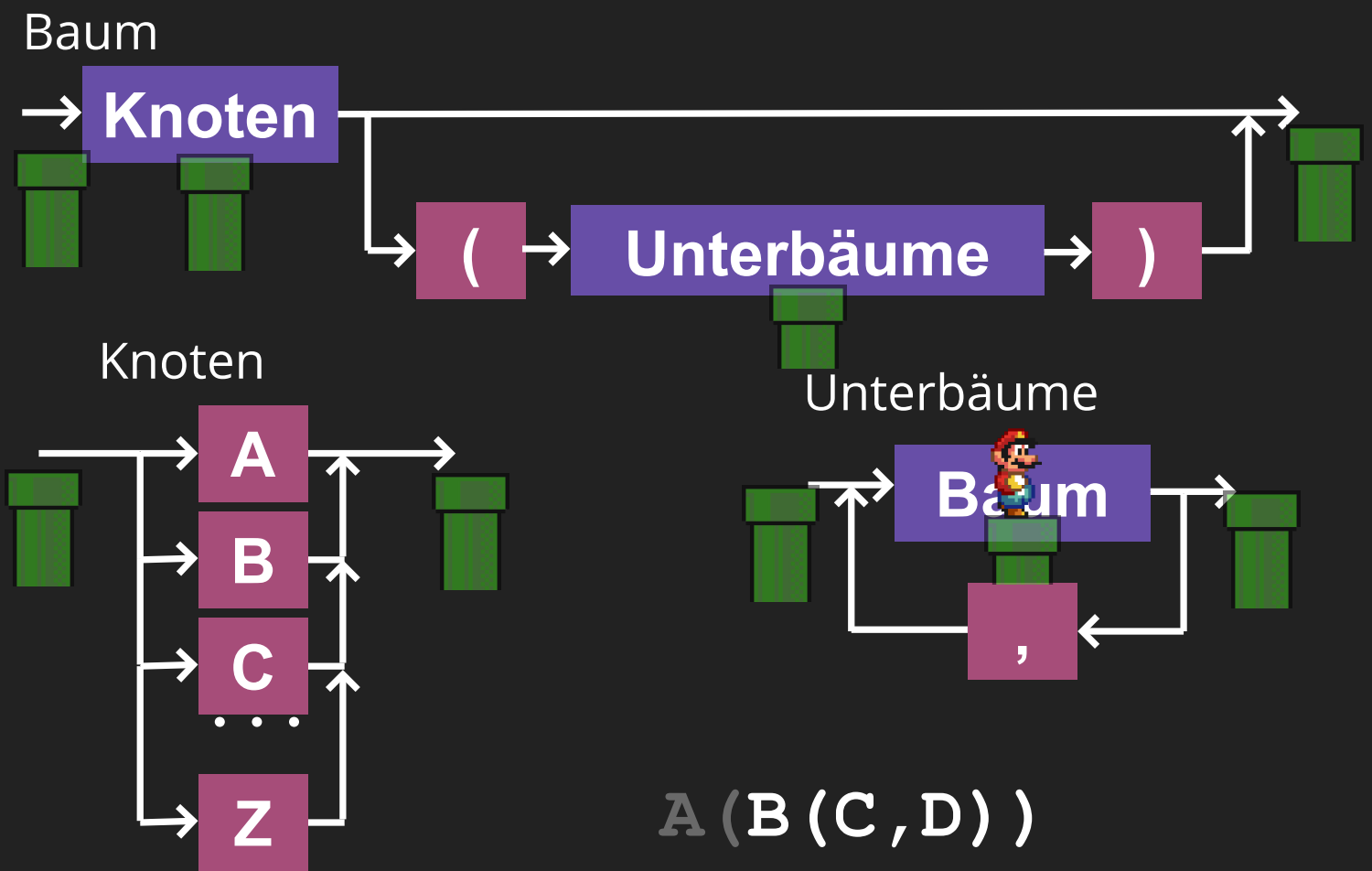

```
A(B(C,D))
```

Syntaxdiagramme



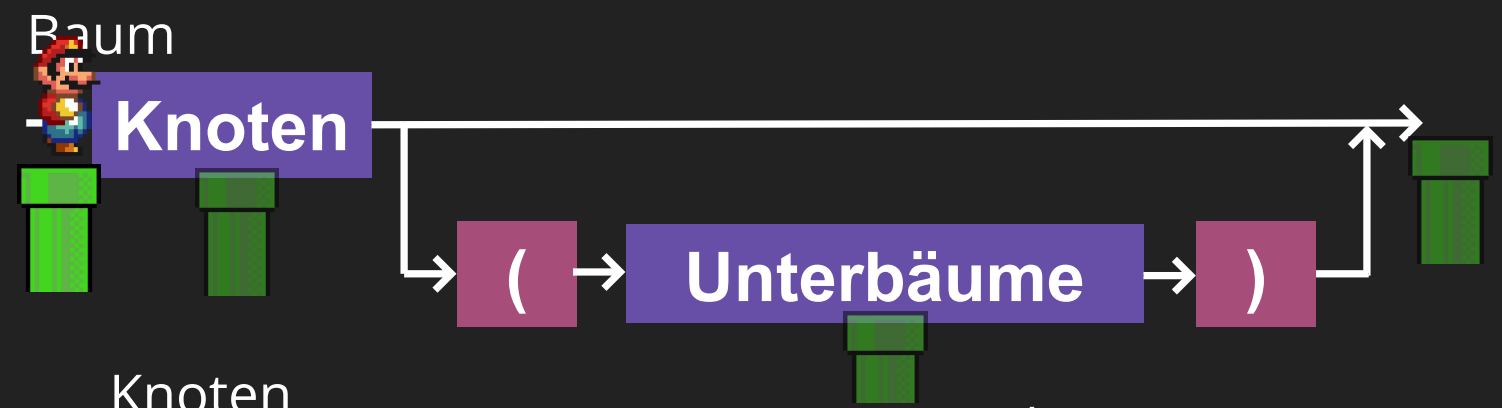

```
A(B(C,D))
```

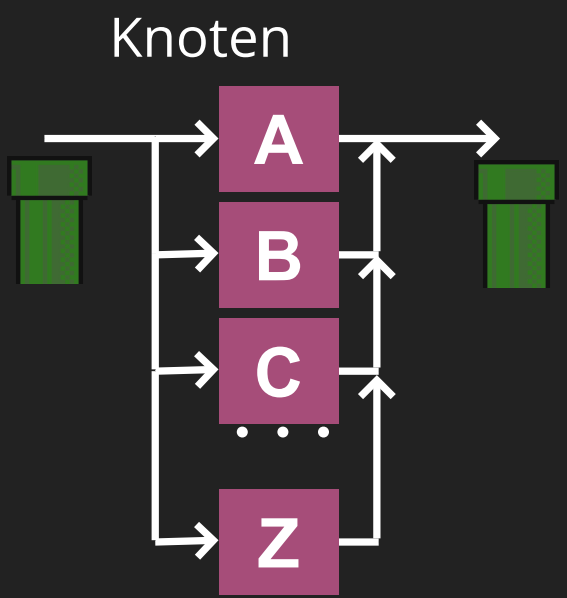
Syntaxdiagramme

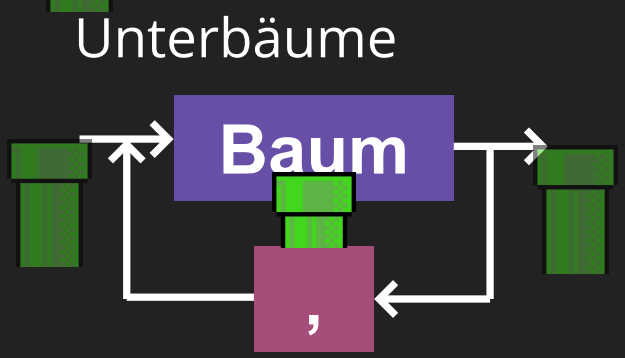



```
A(B(C,D))
```

Syntaxdiagramme

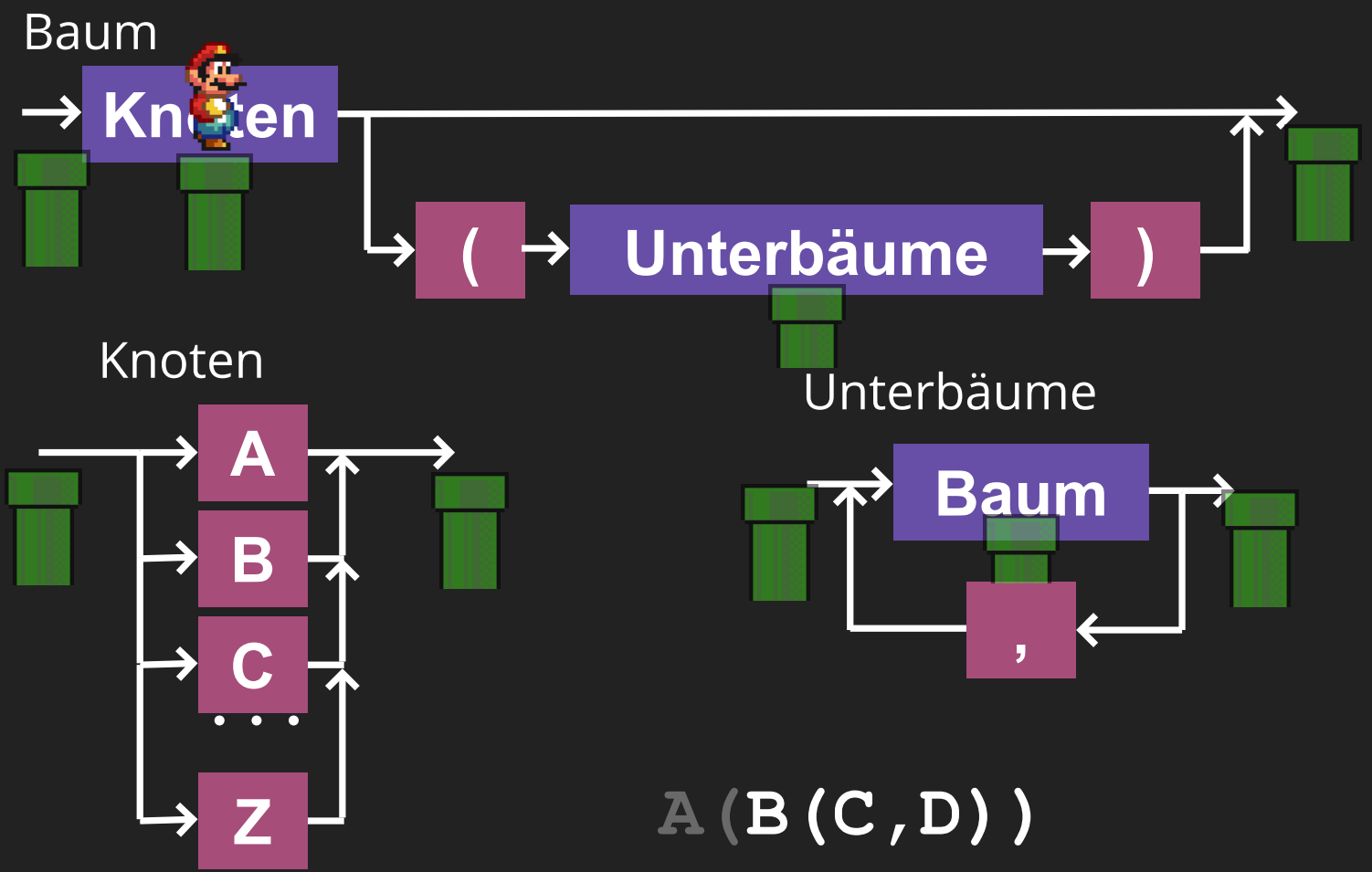




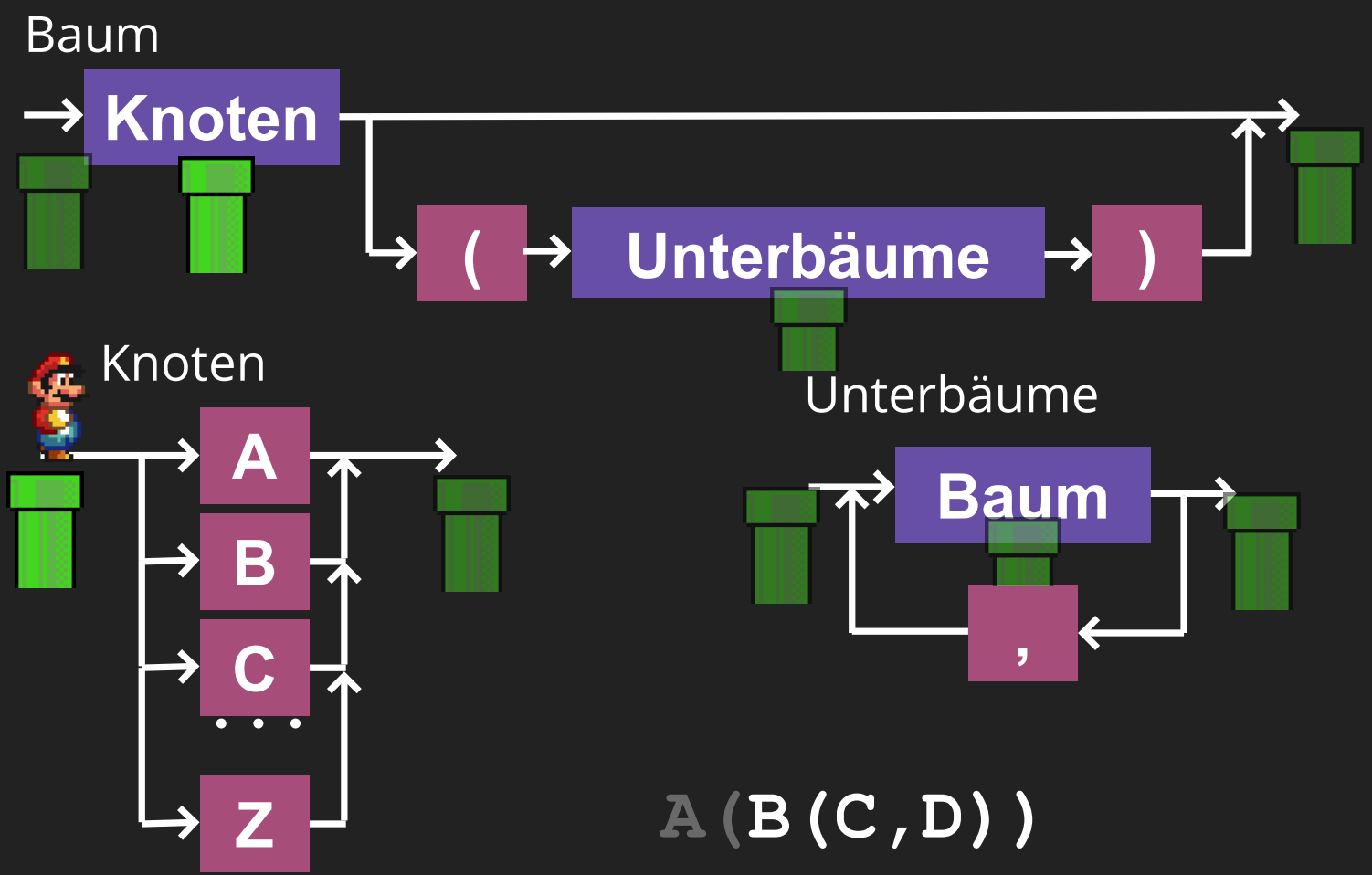



```
A(B(C,D))
```

Syntaxdiagramme

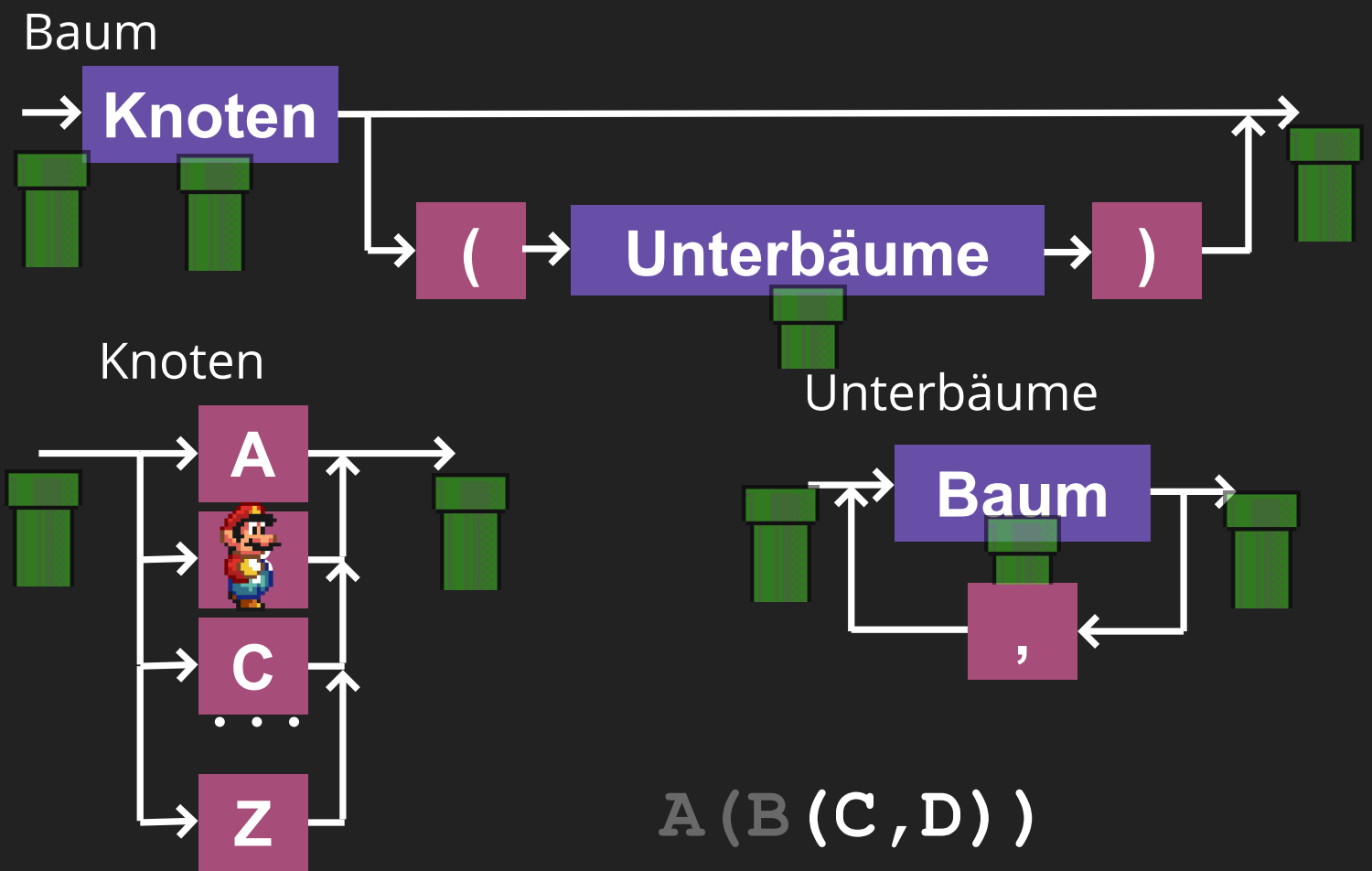


Syntaxdiagramme



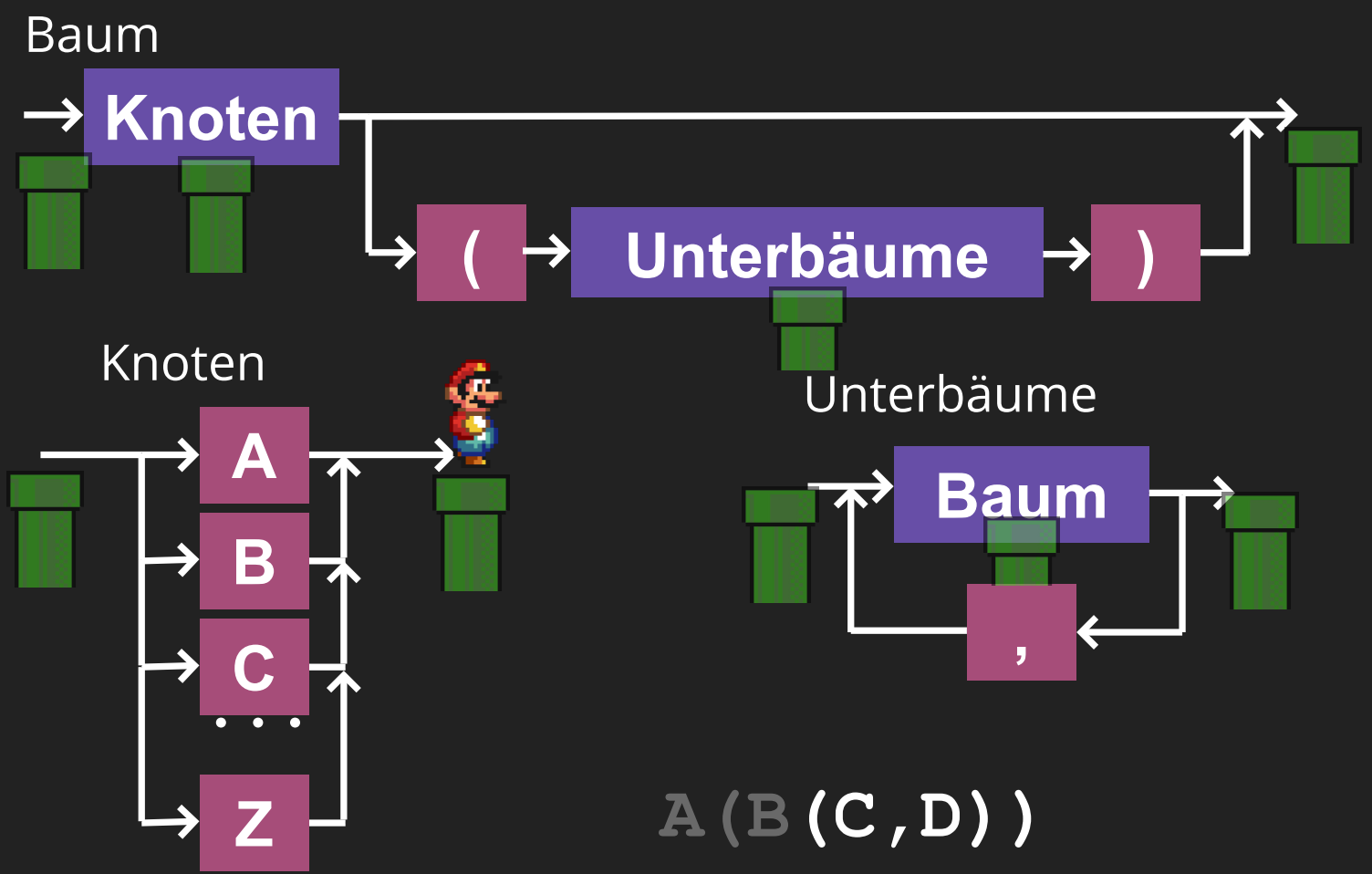

```
A(B(C,D))
```

Syntaxdiagramme



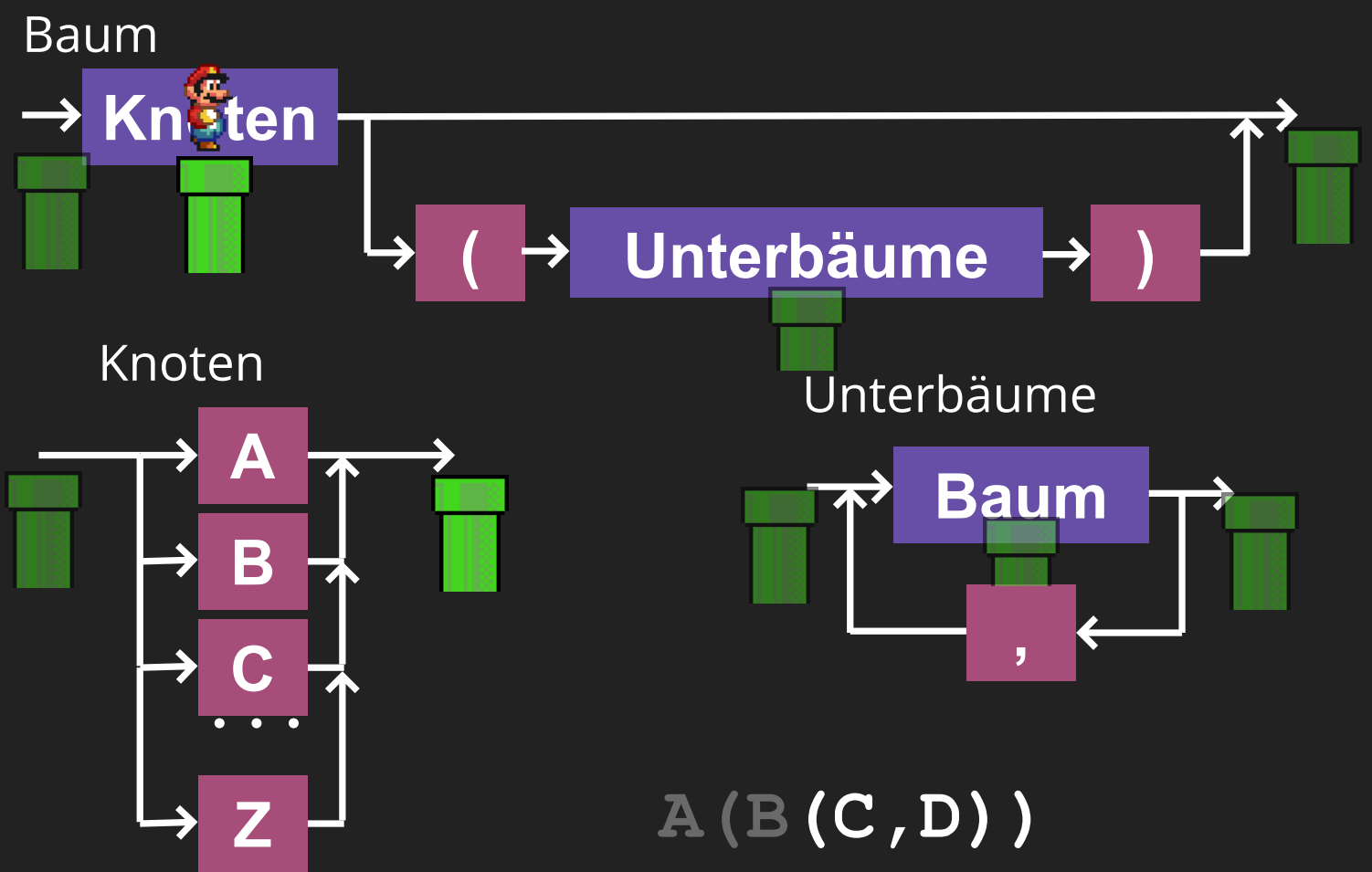

```
A(B(C,D))
```

Syntaxdiagramme



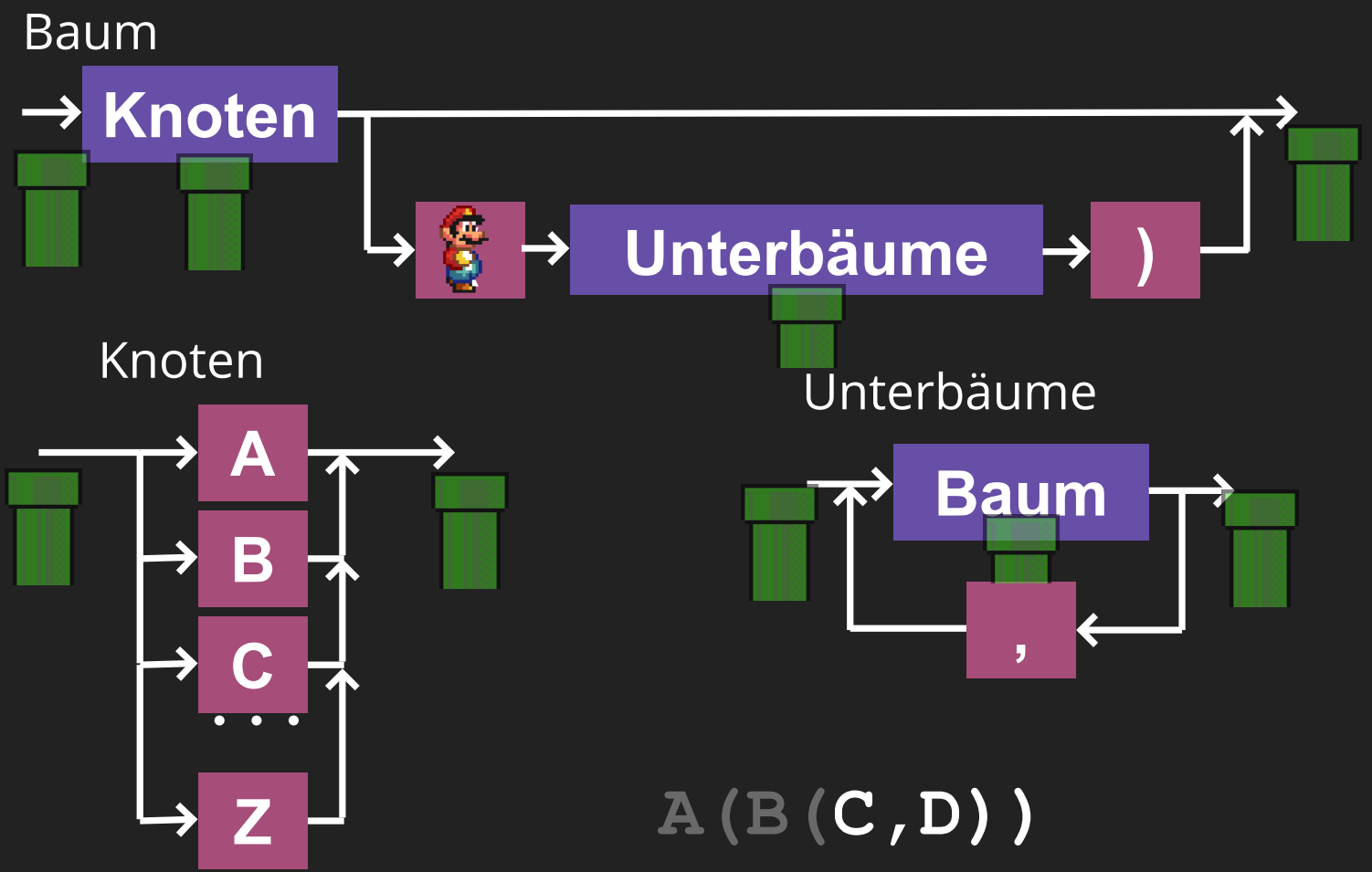

```
A(B(C,D))
```

Syntaxdiagramme



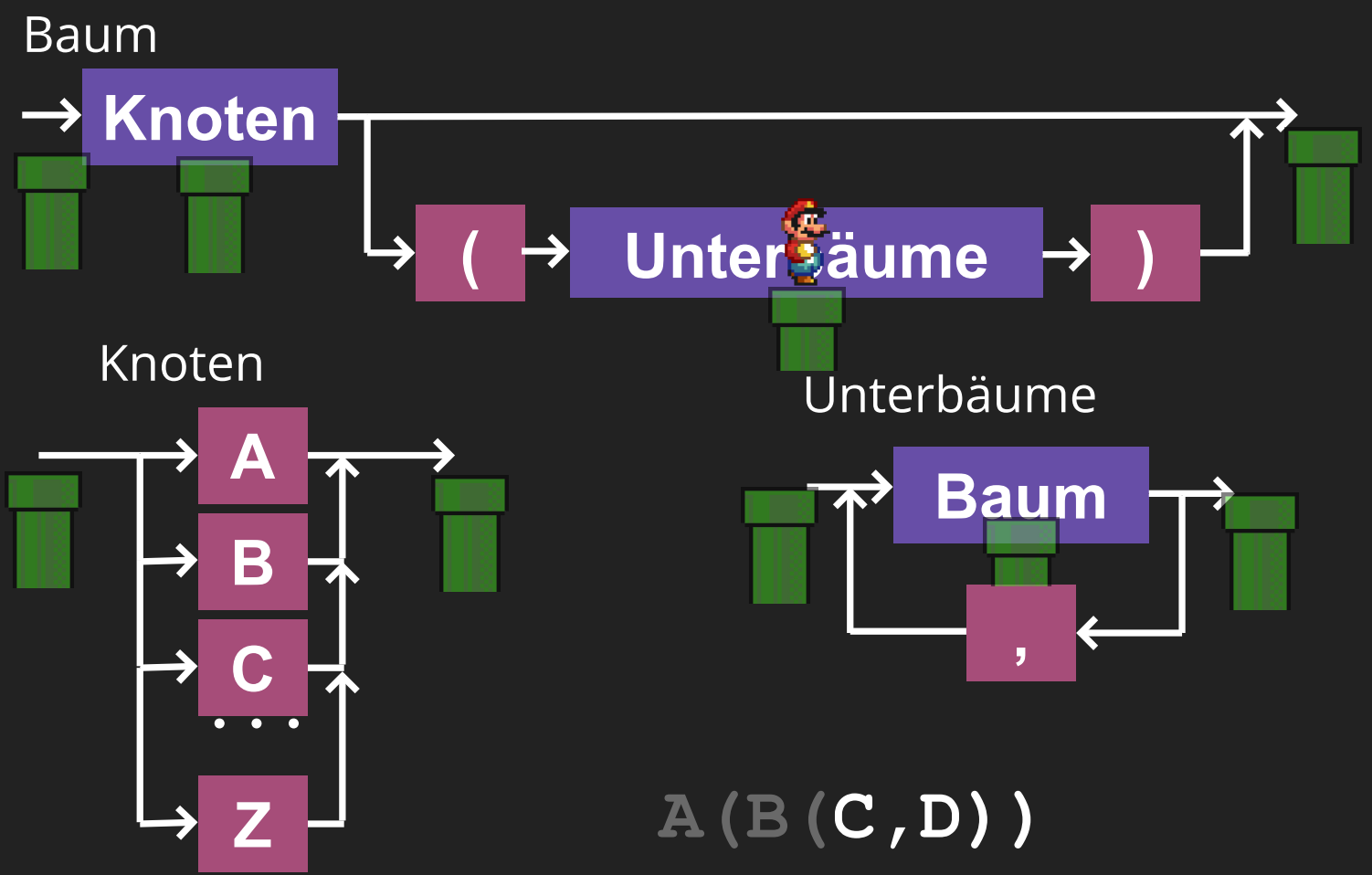

```
A(B(C,D))
```

Syntaxdiagramme



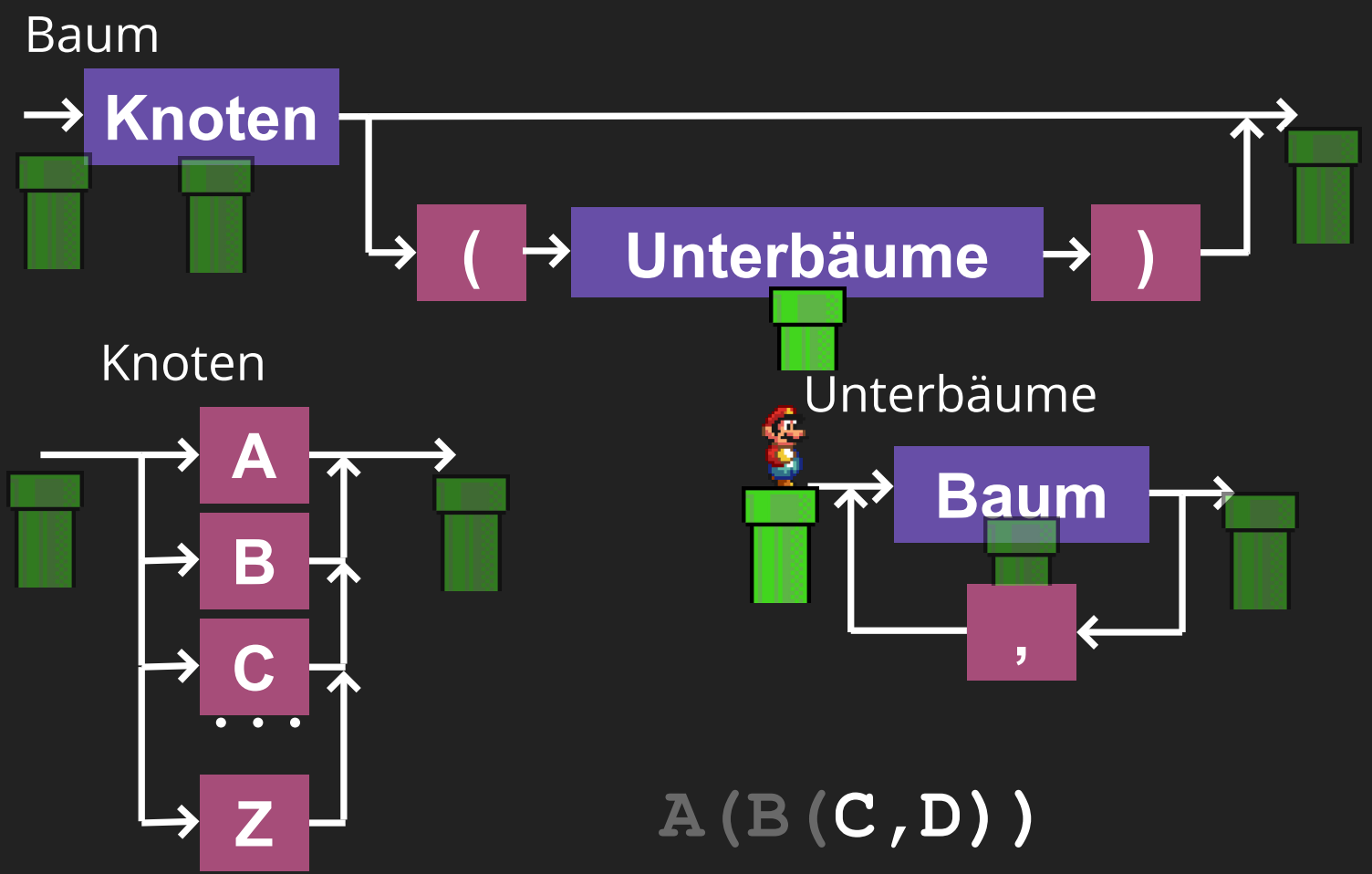

```
A(B(C,D))
```

Syntaxdiagramme



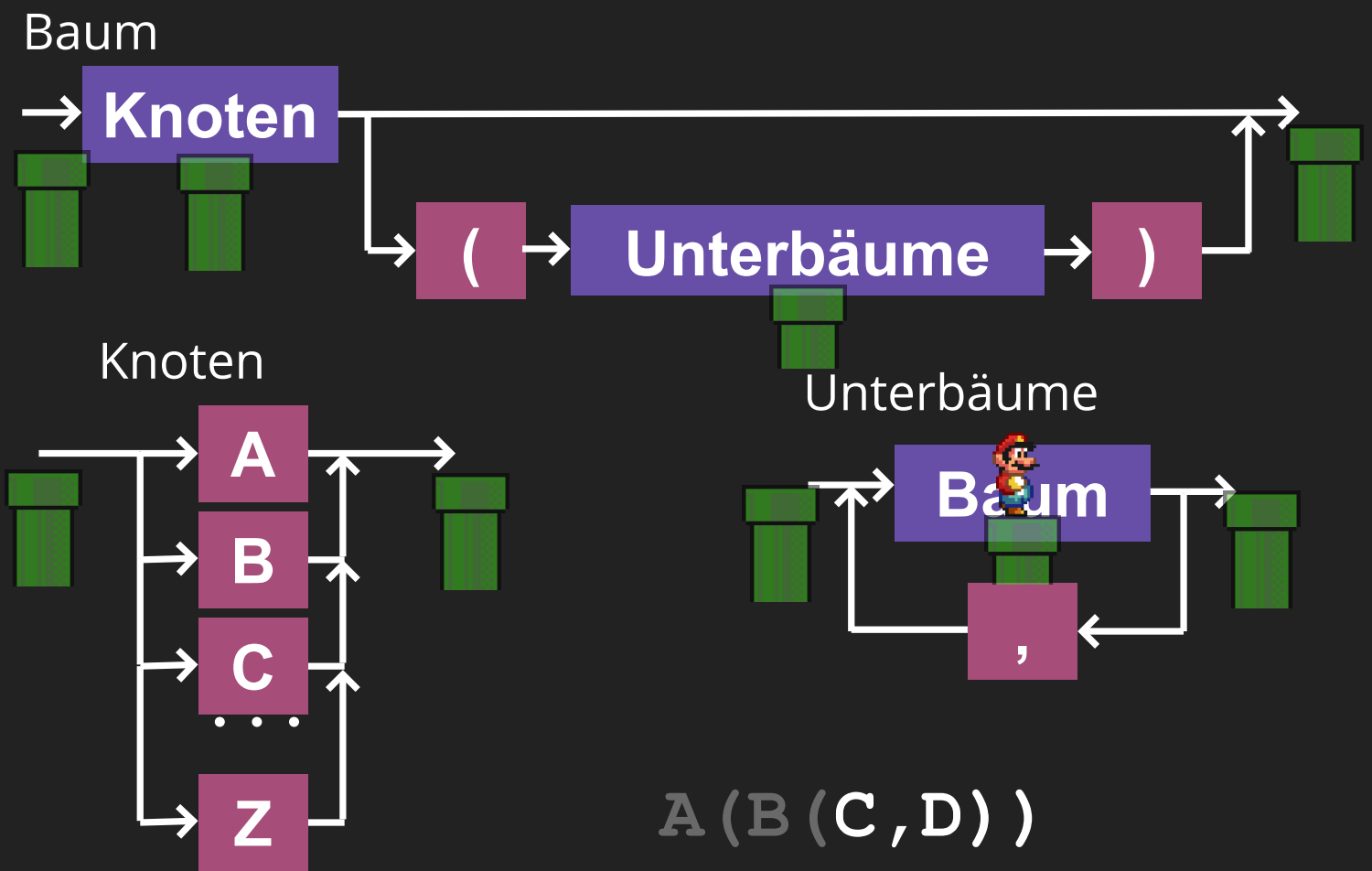

```
A(B(C,D))
```

Syntaxdiagramme



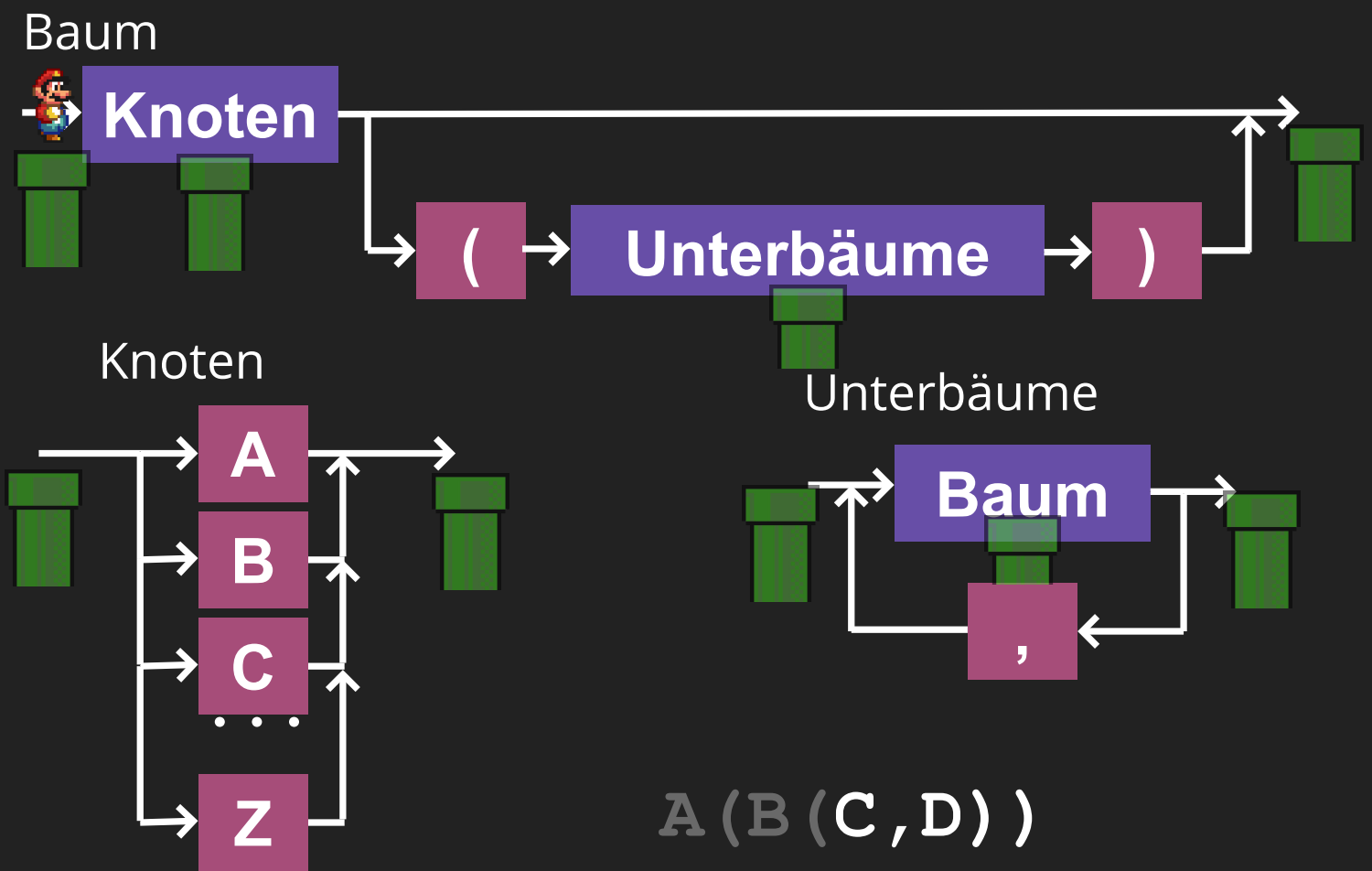

```
A(B(C,D))
```

Syntaxdiagramme



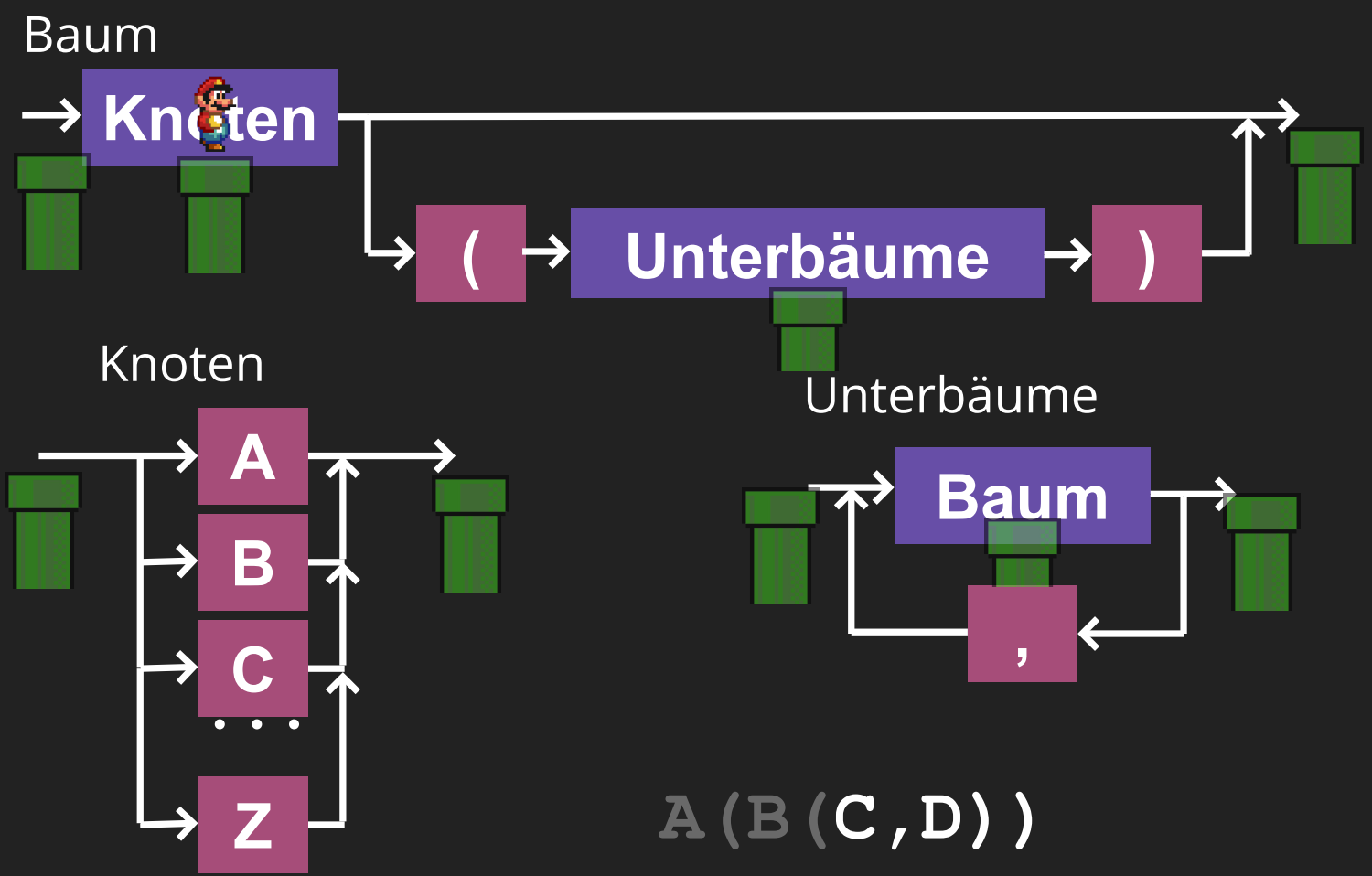

```
A(B(C,D))
```

Syntaxdiagramme

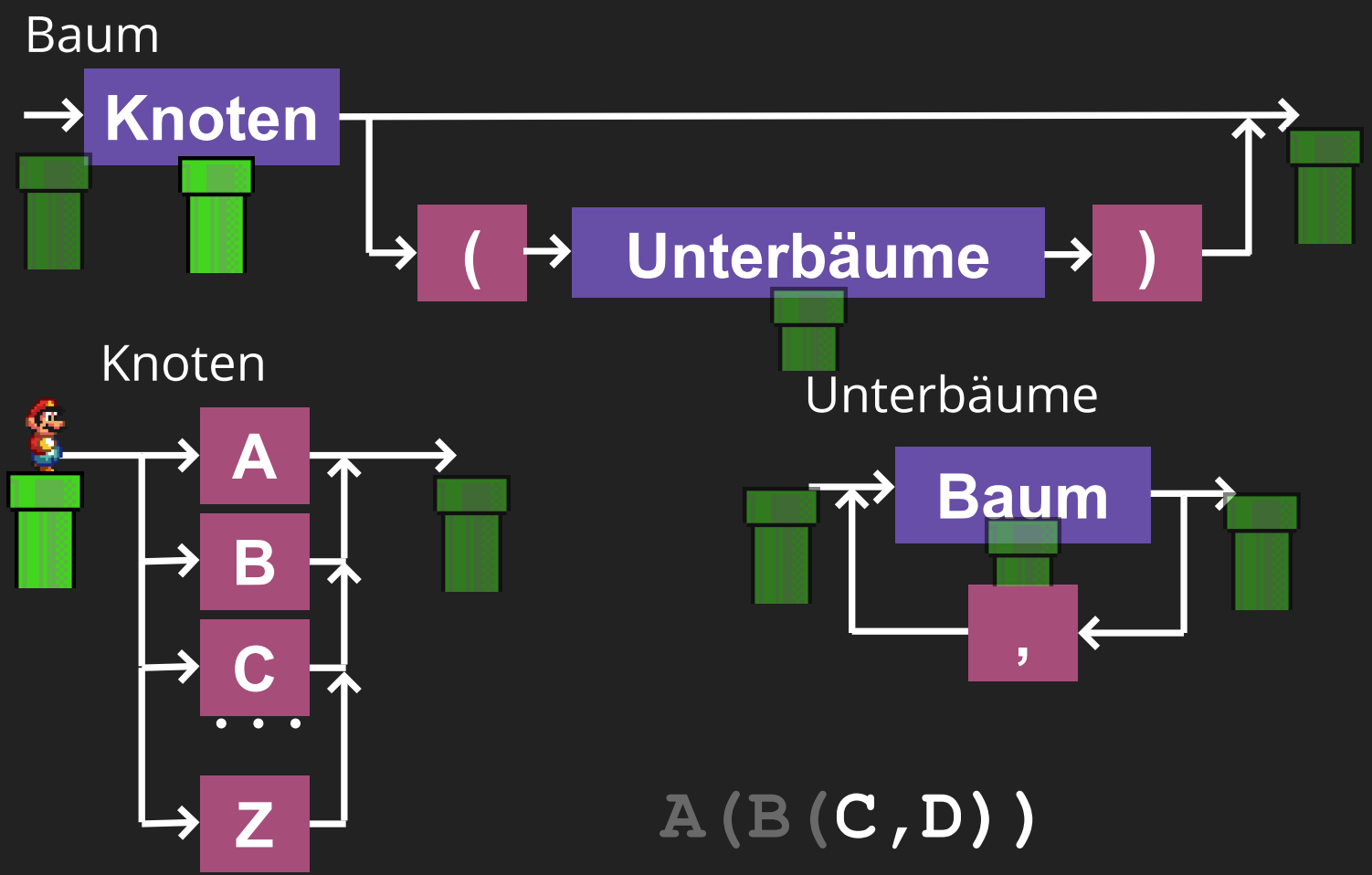



```
A(B(C,D))
```

Syntaxdiagramme

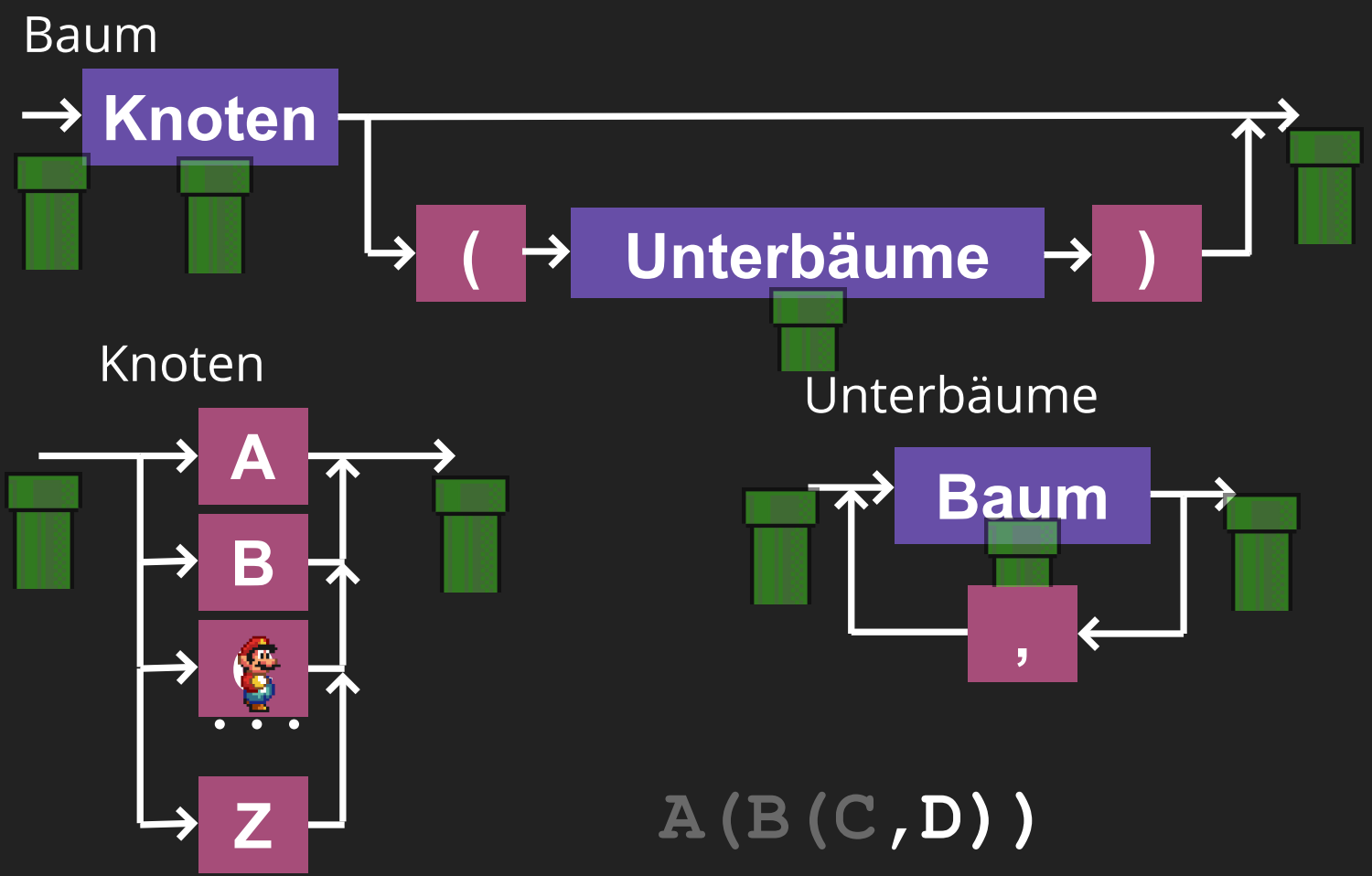


Syntaxdiagramme



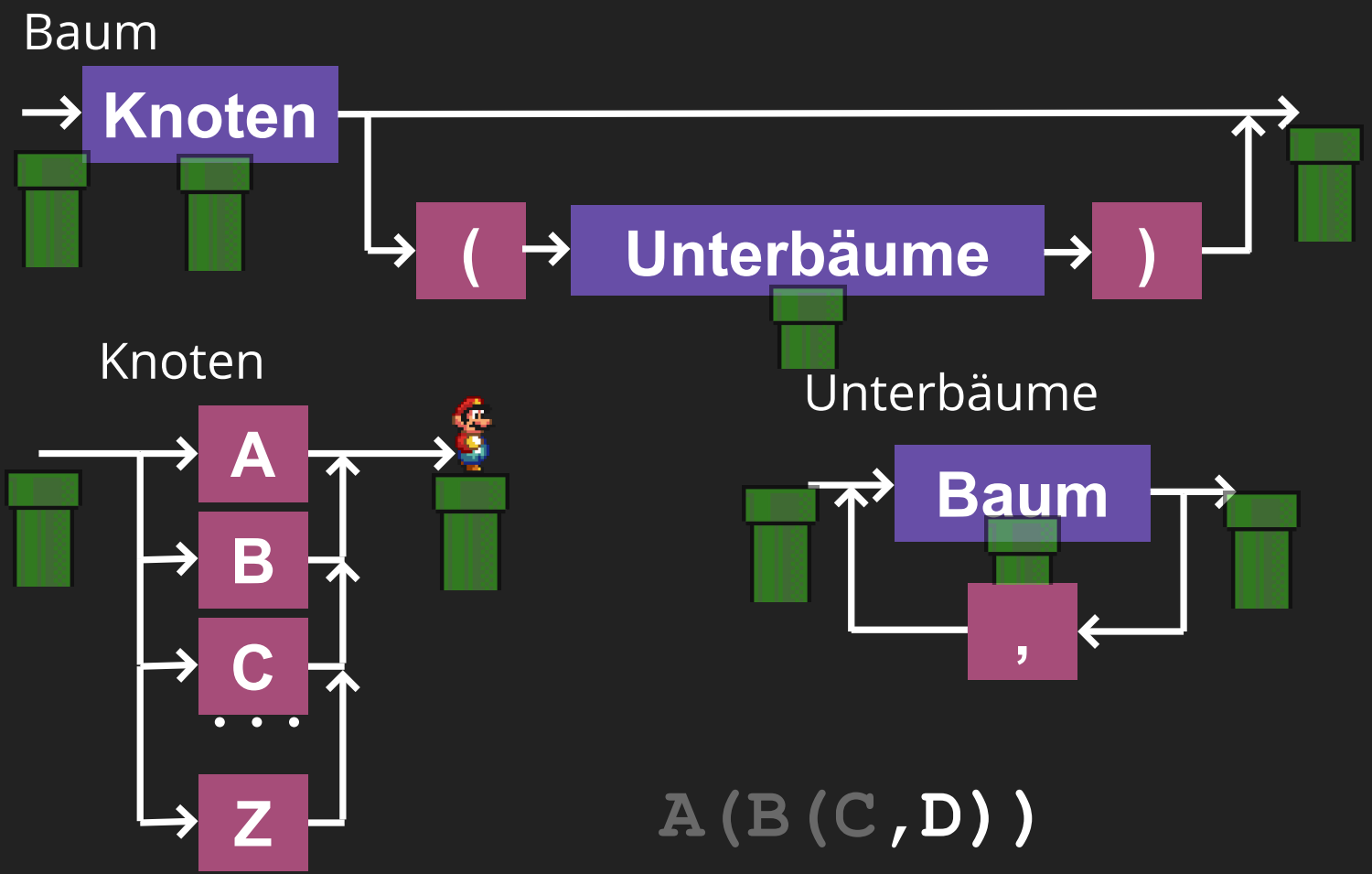

```
A(B(C,D))
```

Syntaxdiagramme



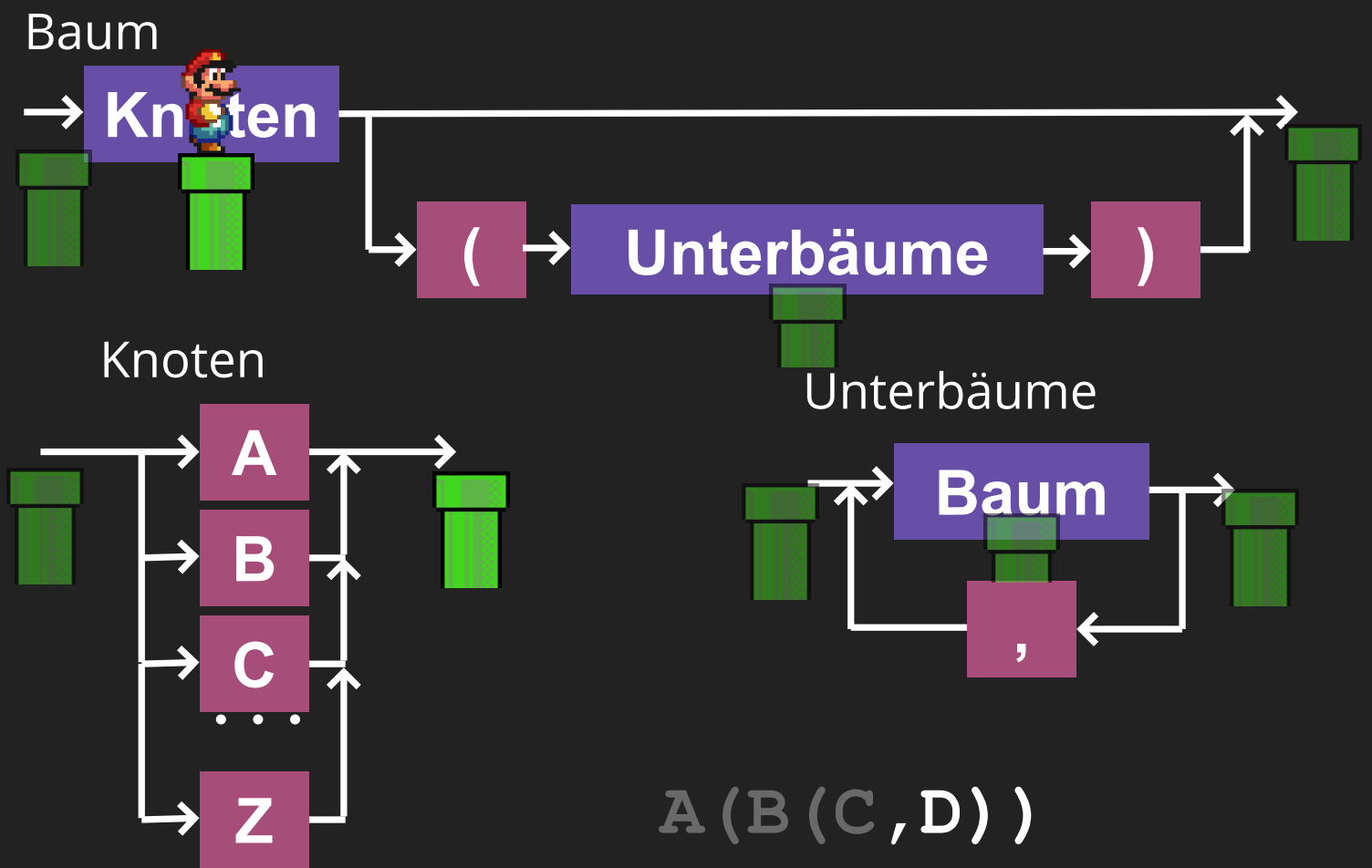

```
A(B(C,D))
```

Syntaxdiagramme



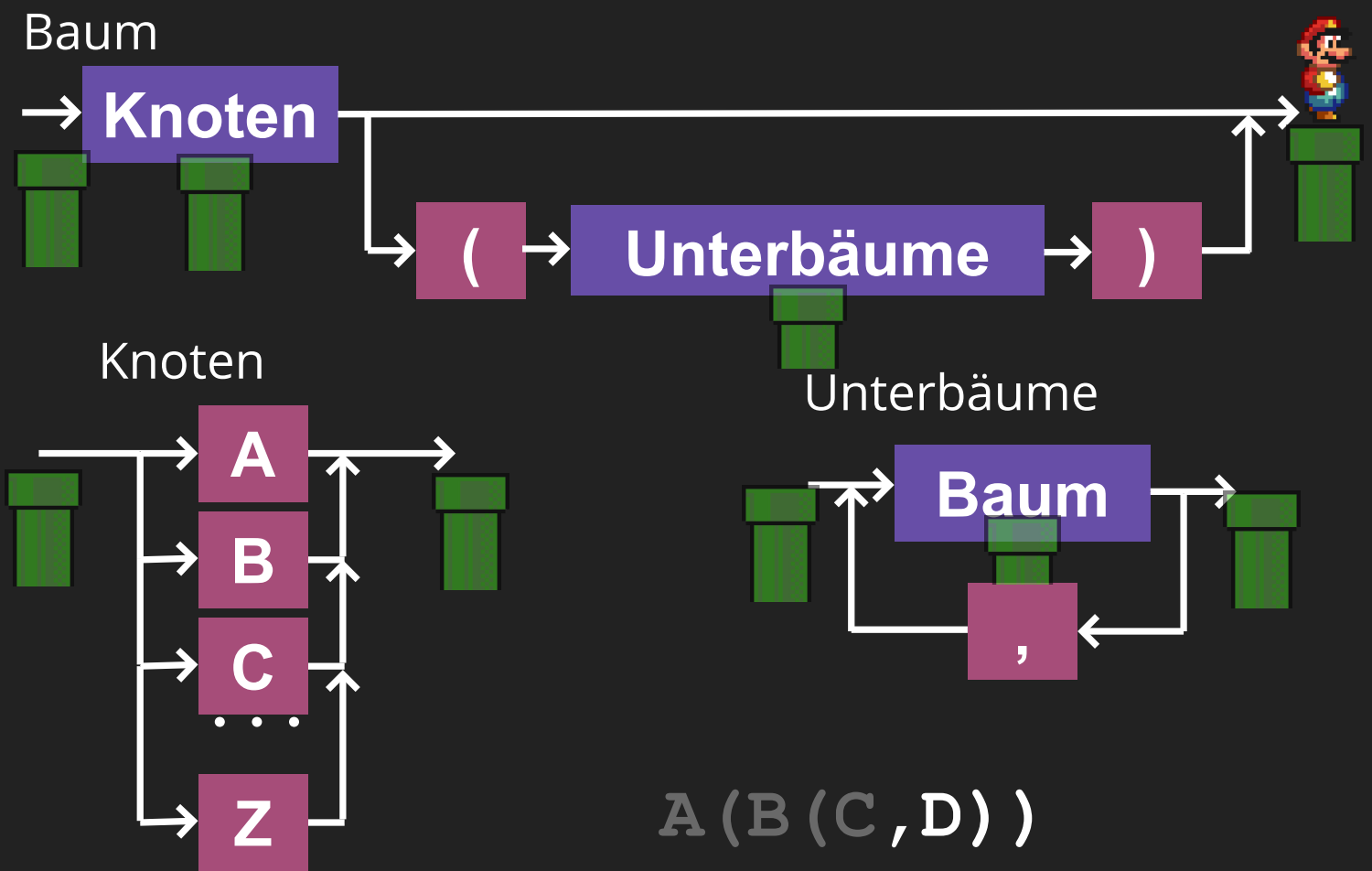

```
A(B(C,D))
```

Syntaxdiagramme



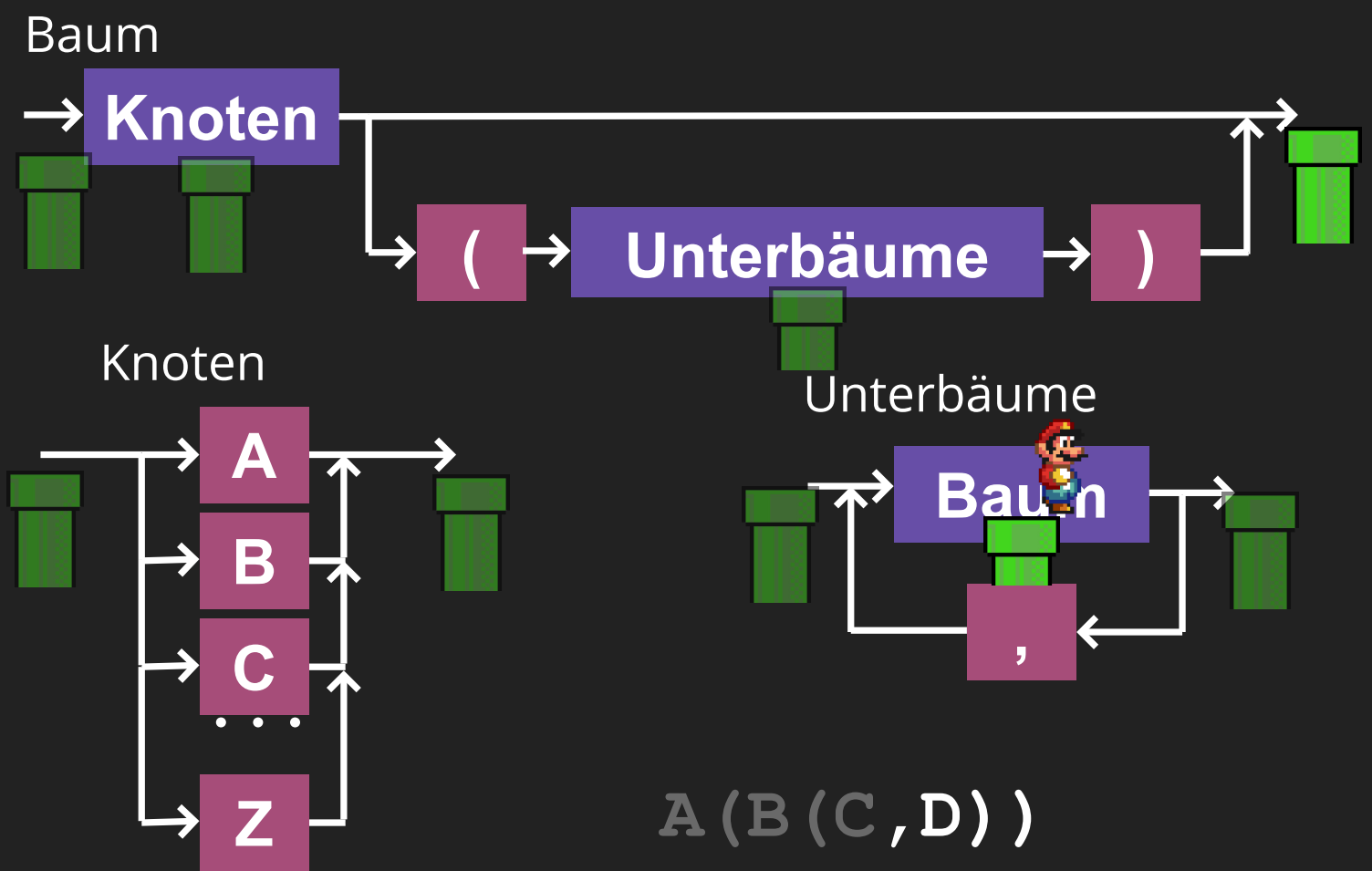

```
A(B(C,D))
```

Syntaxdiagramme



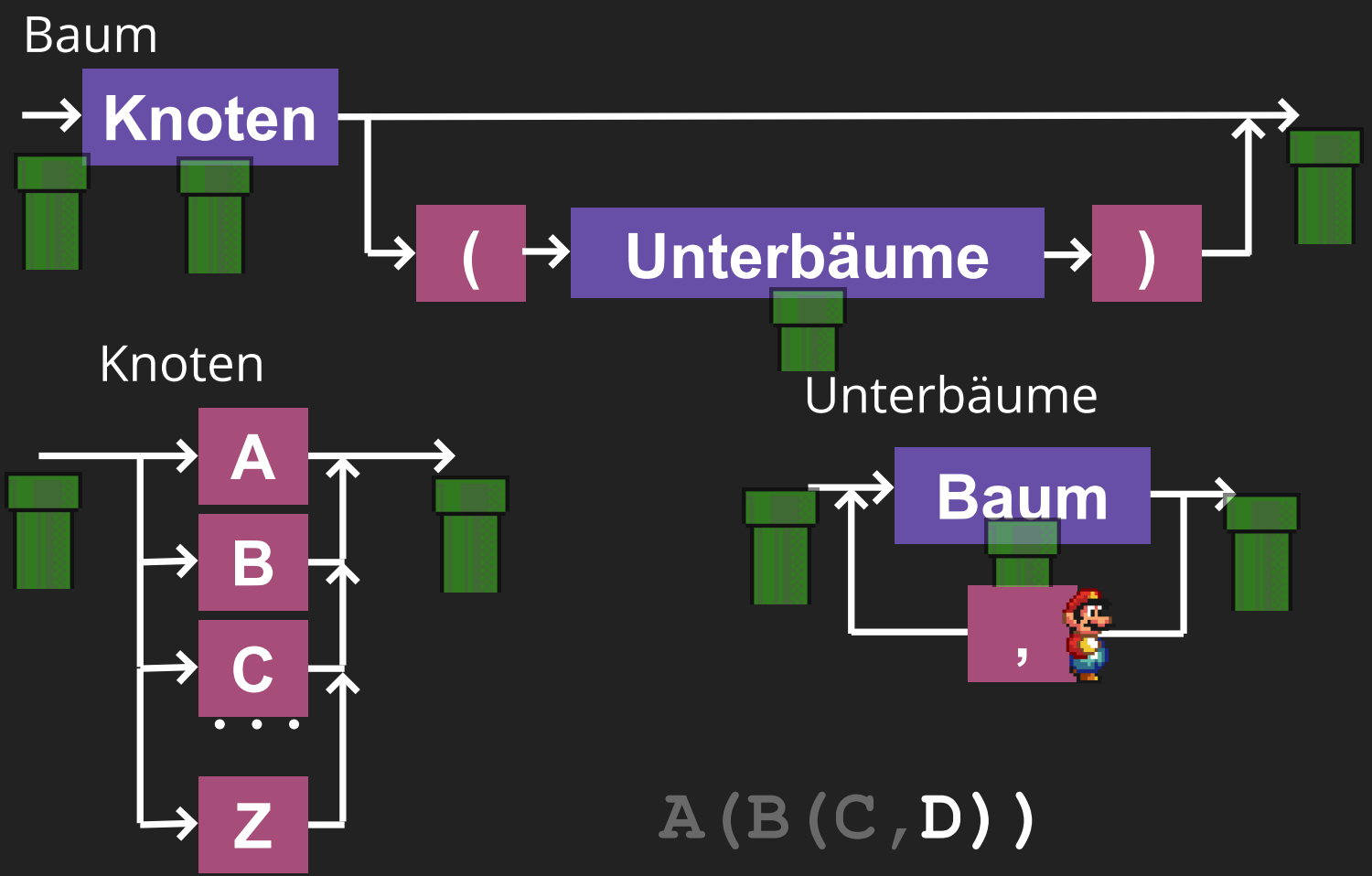

```
A(B(C,D))
```

Syntaxdiagramme



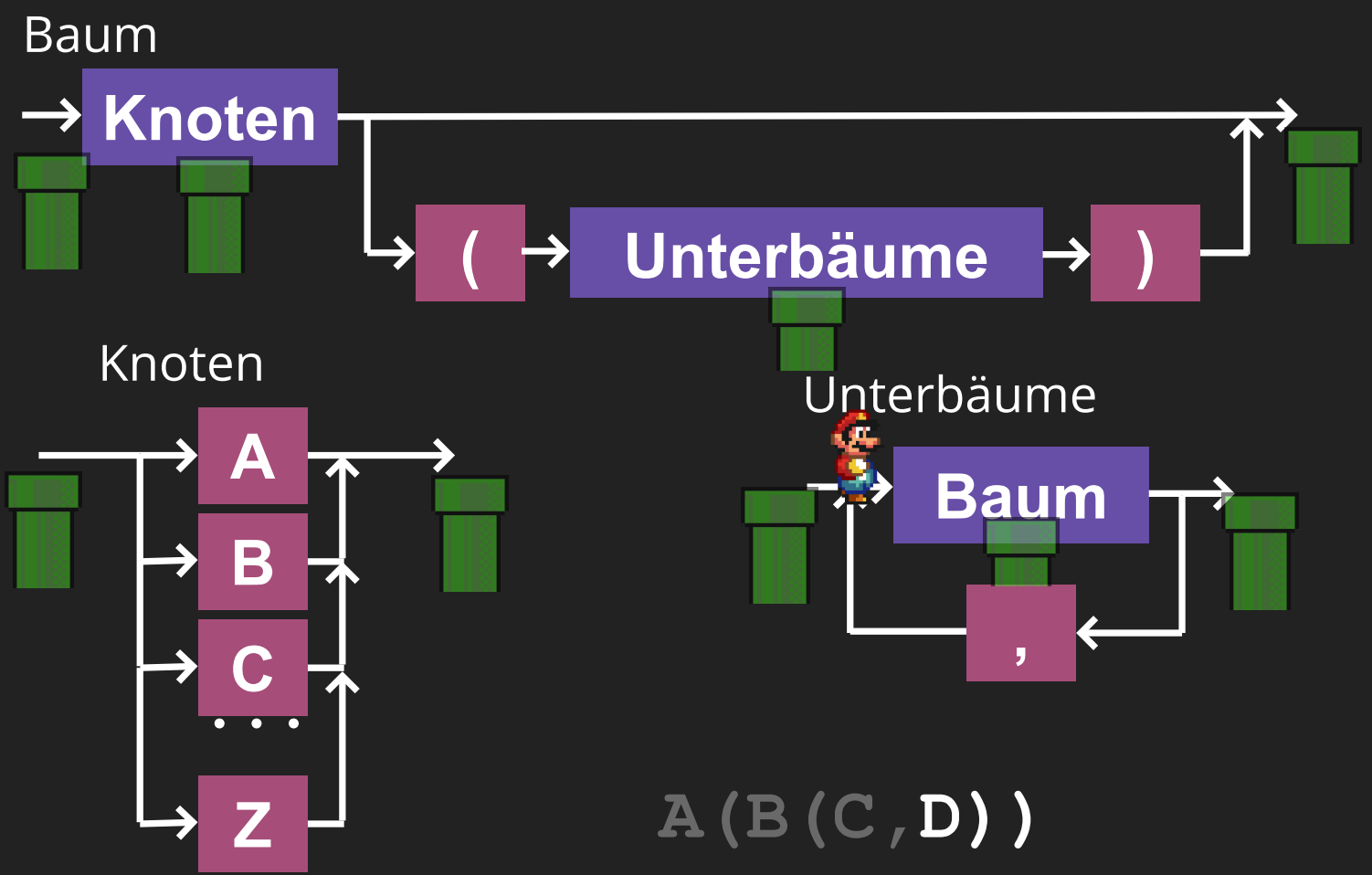

```
A(B(C,D))
```

Syntaxdiagramme



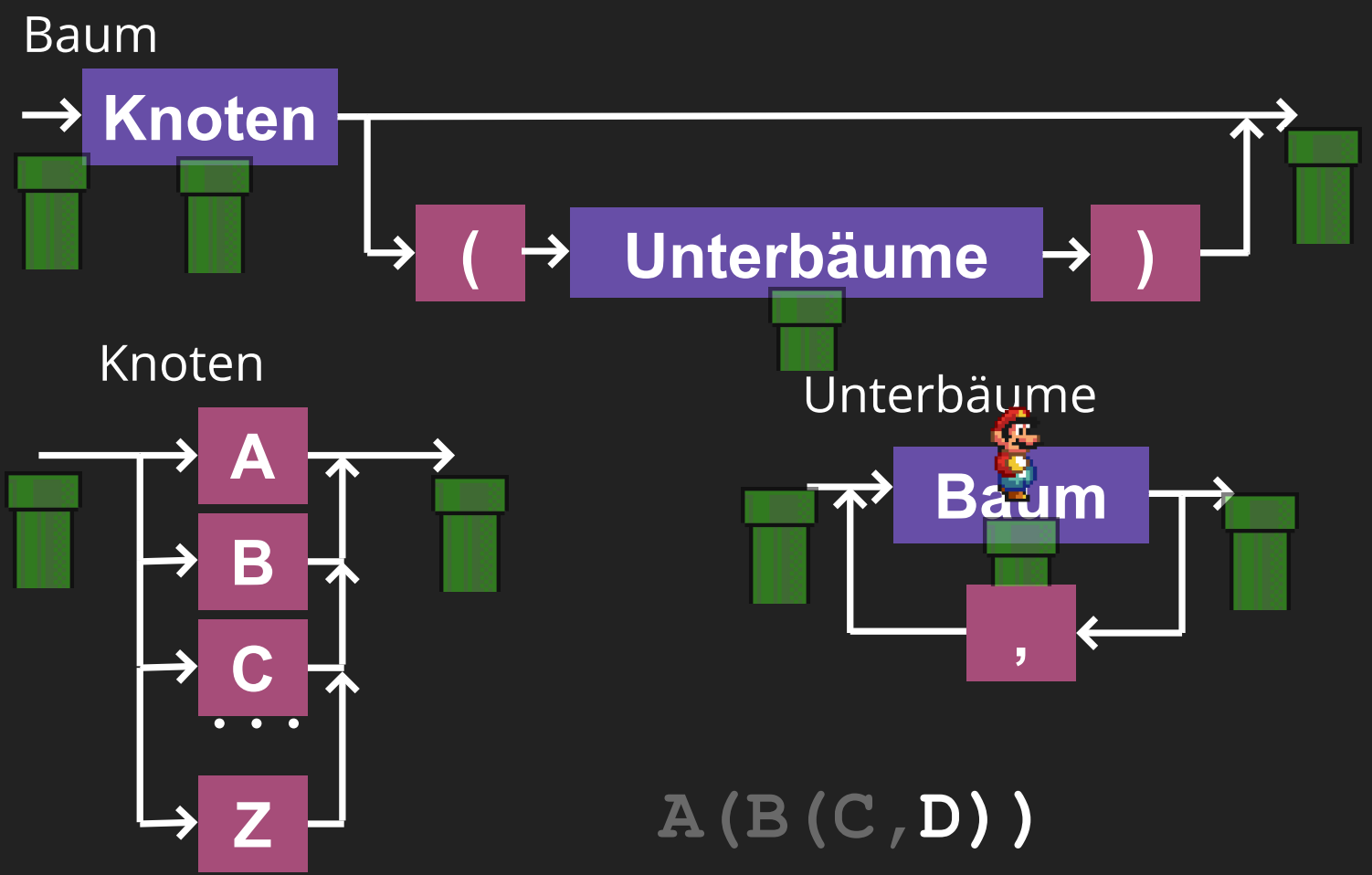

```
A(B(C,D))
```

Syntaxdiagramme



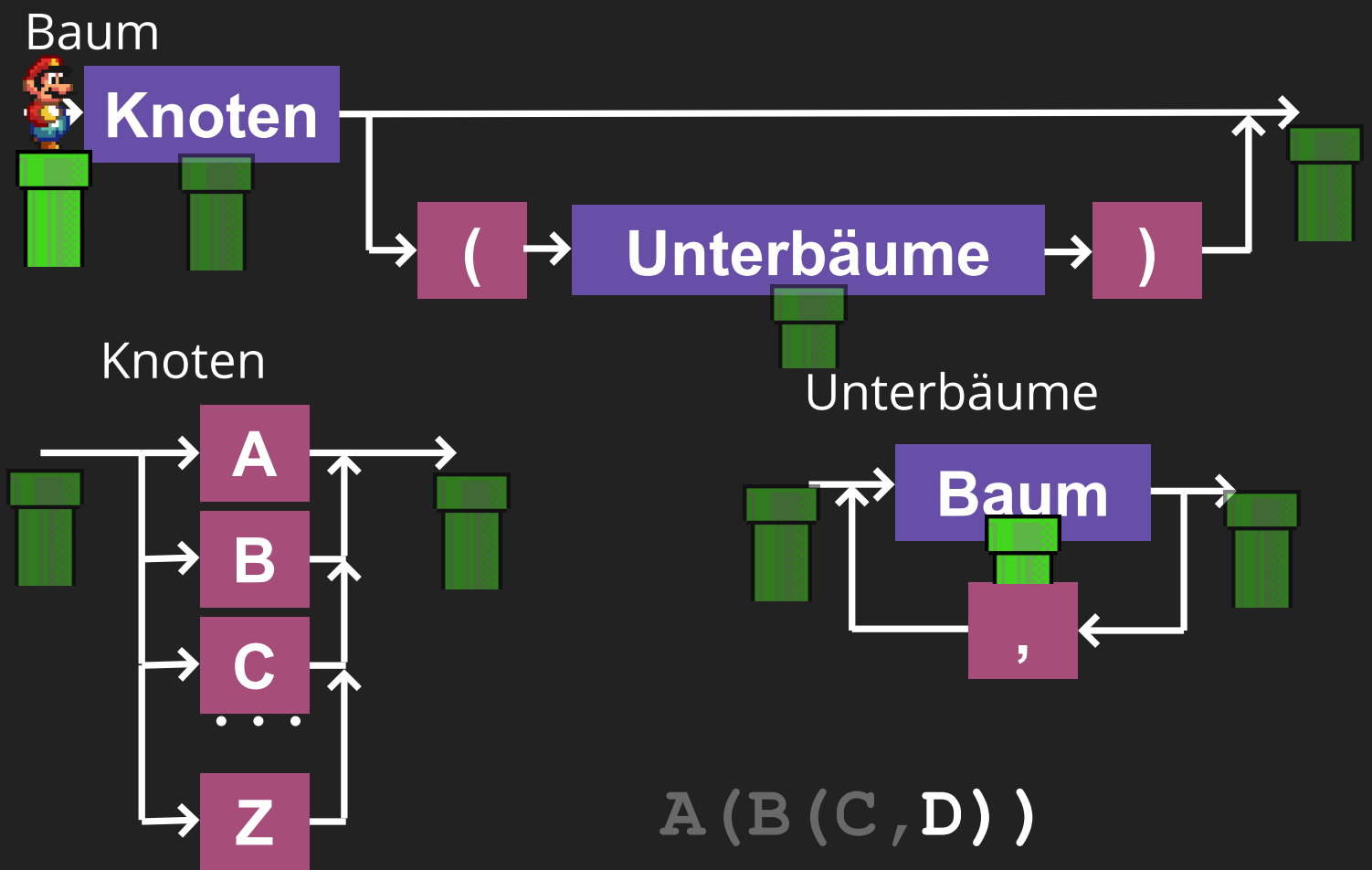

```
A(B(C,D))
```

Syntaxdiagramme



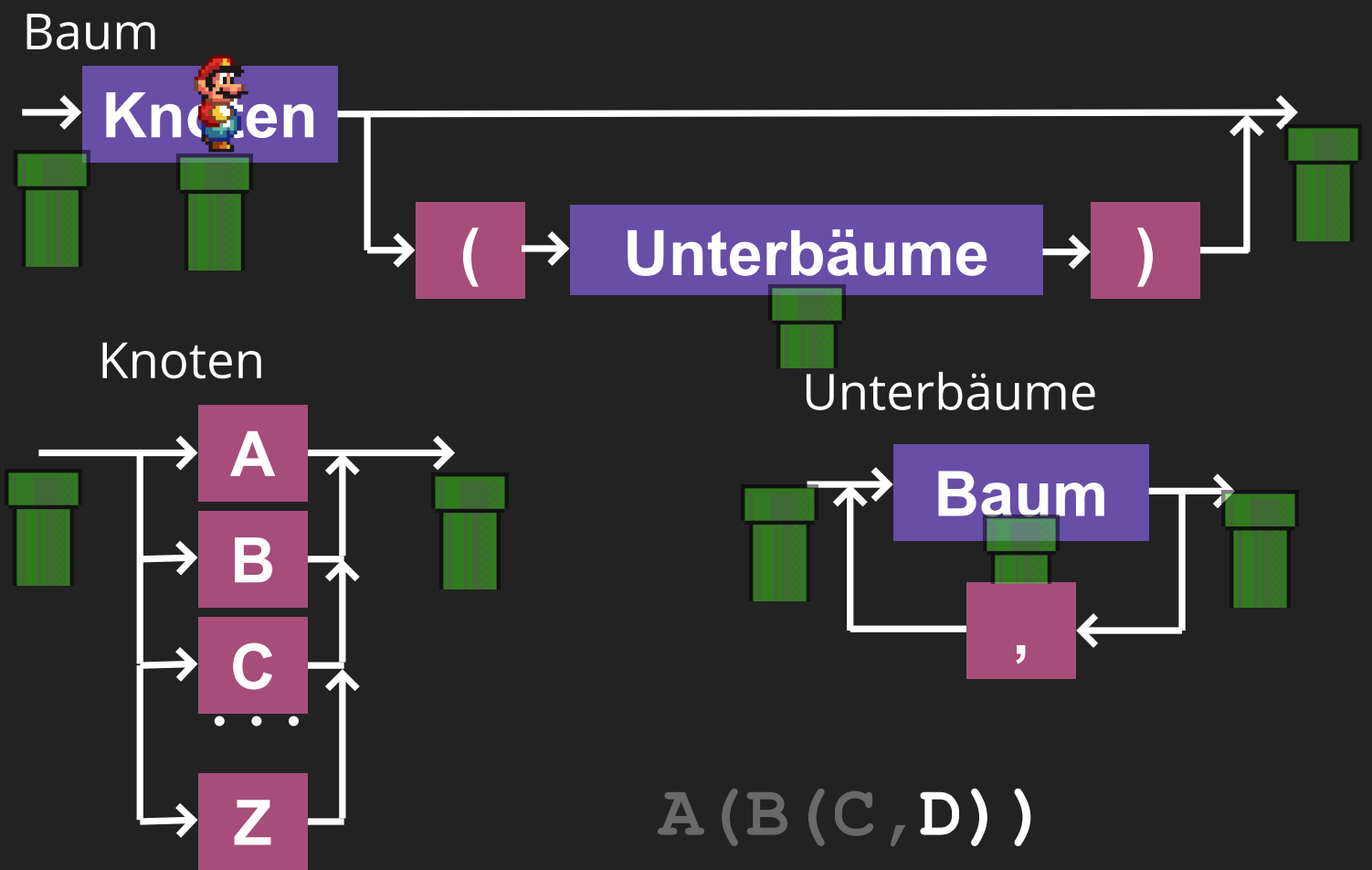

```
A(B(C,D))
```

Syntaxdiagramme



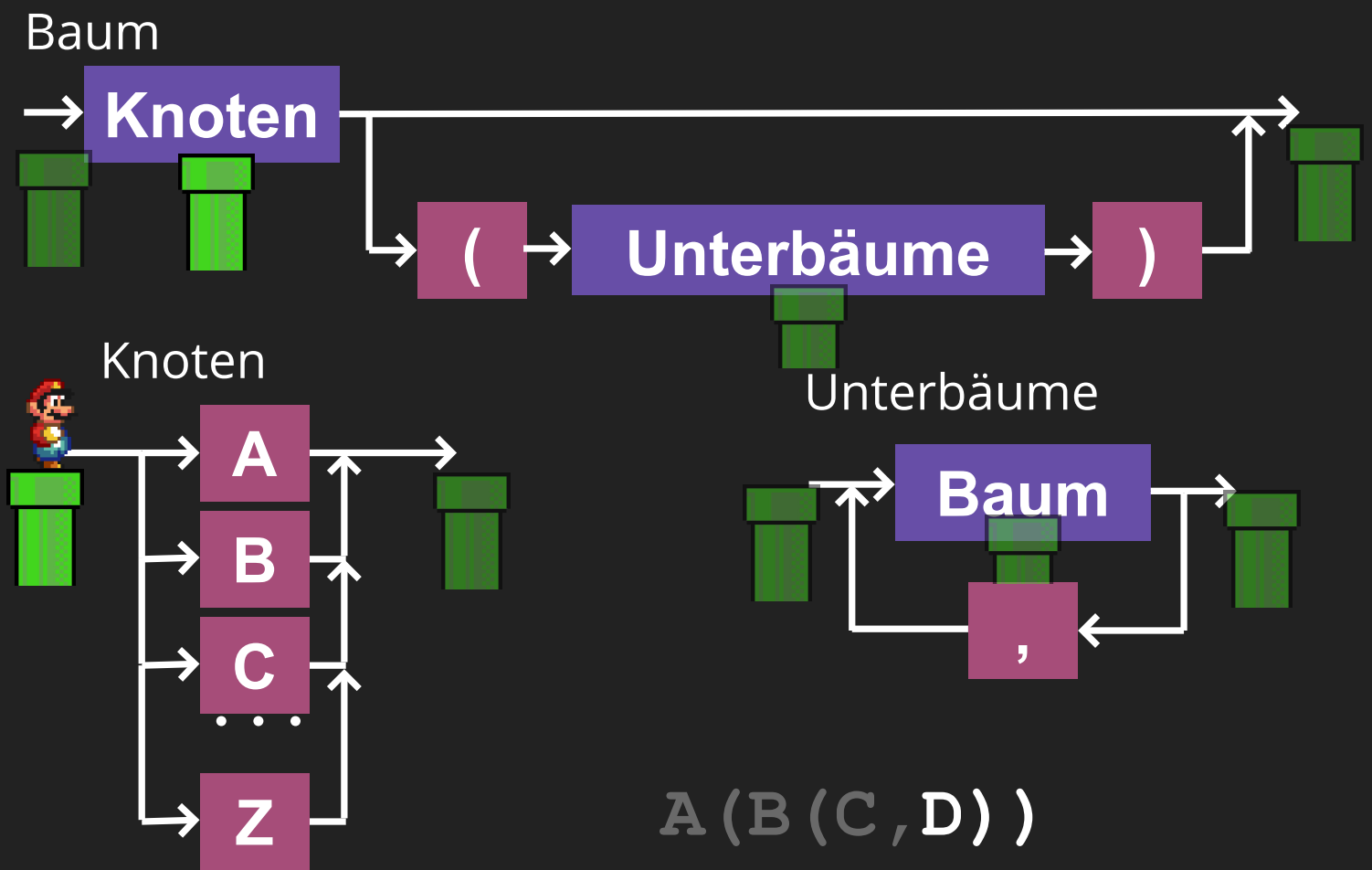

```
A(B(C,D))
```

Syntaxdiagramme



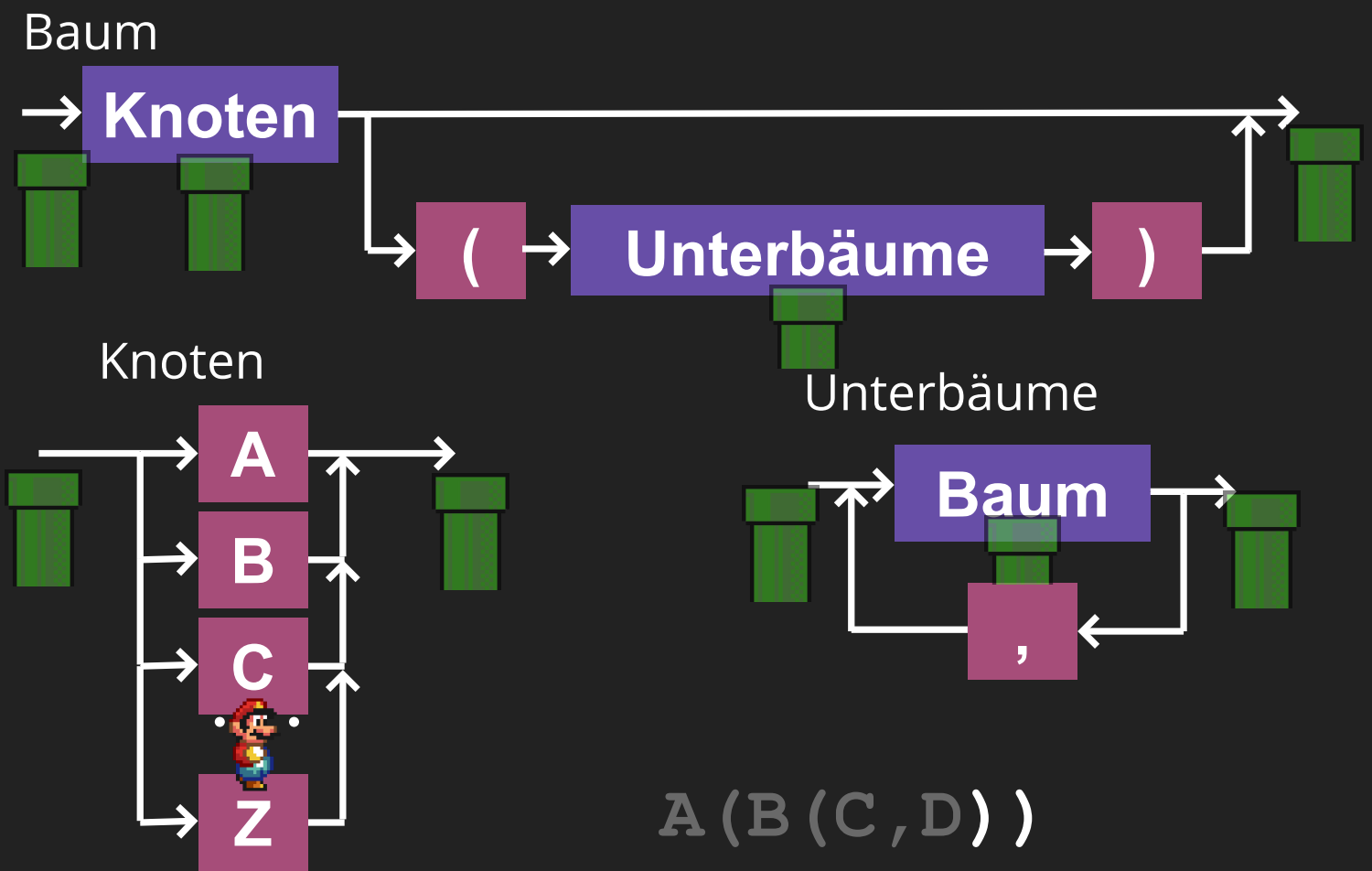

```
A(B(C,D))
```

Syntaxdiagramme



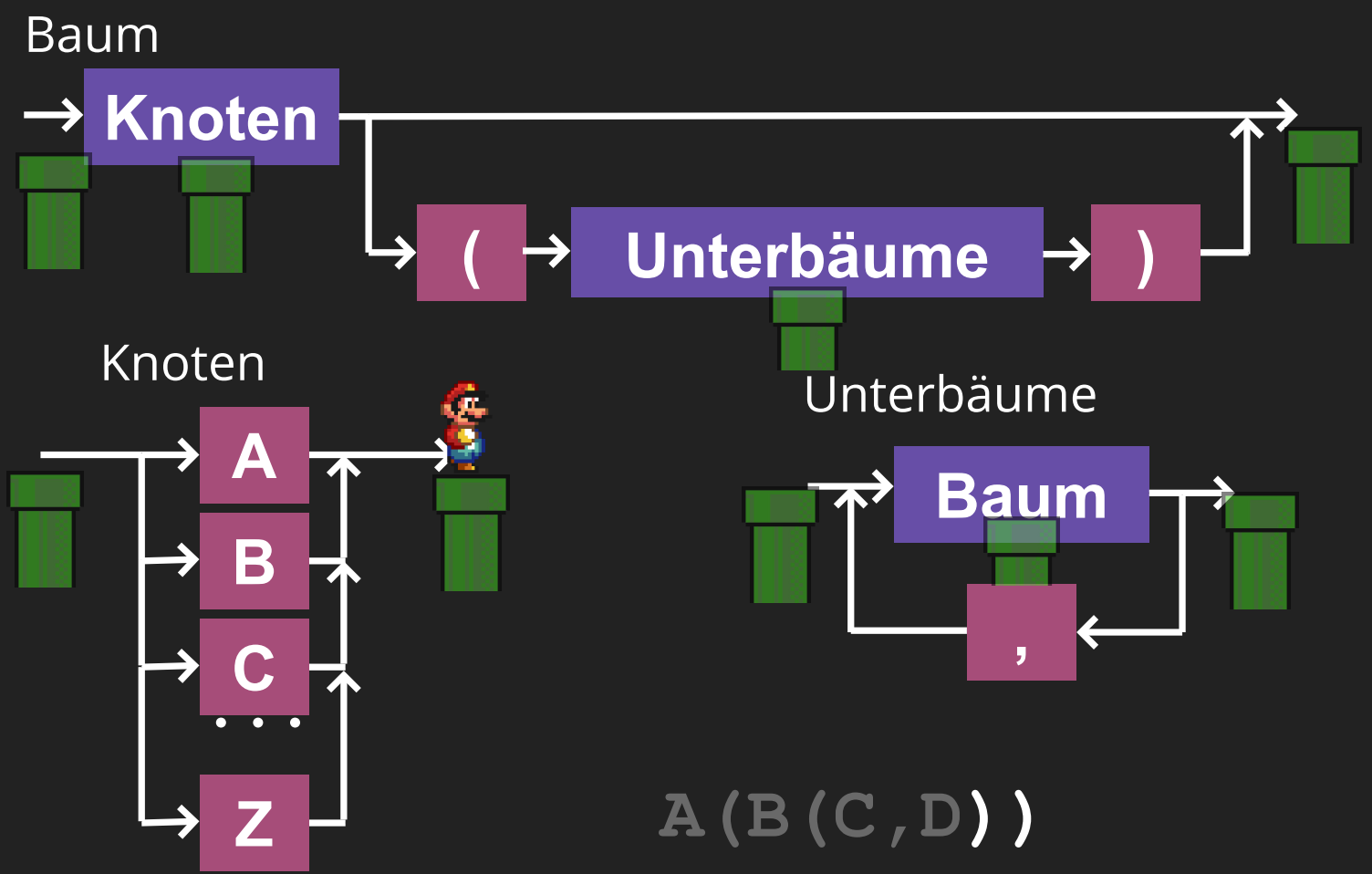

```
A(B(C,D))
```

Syntaxdiagramme

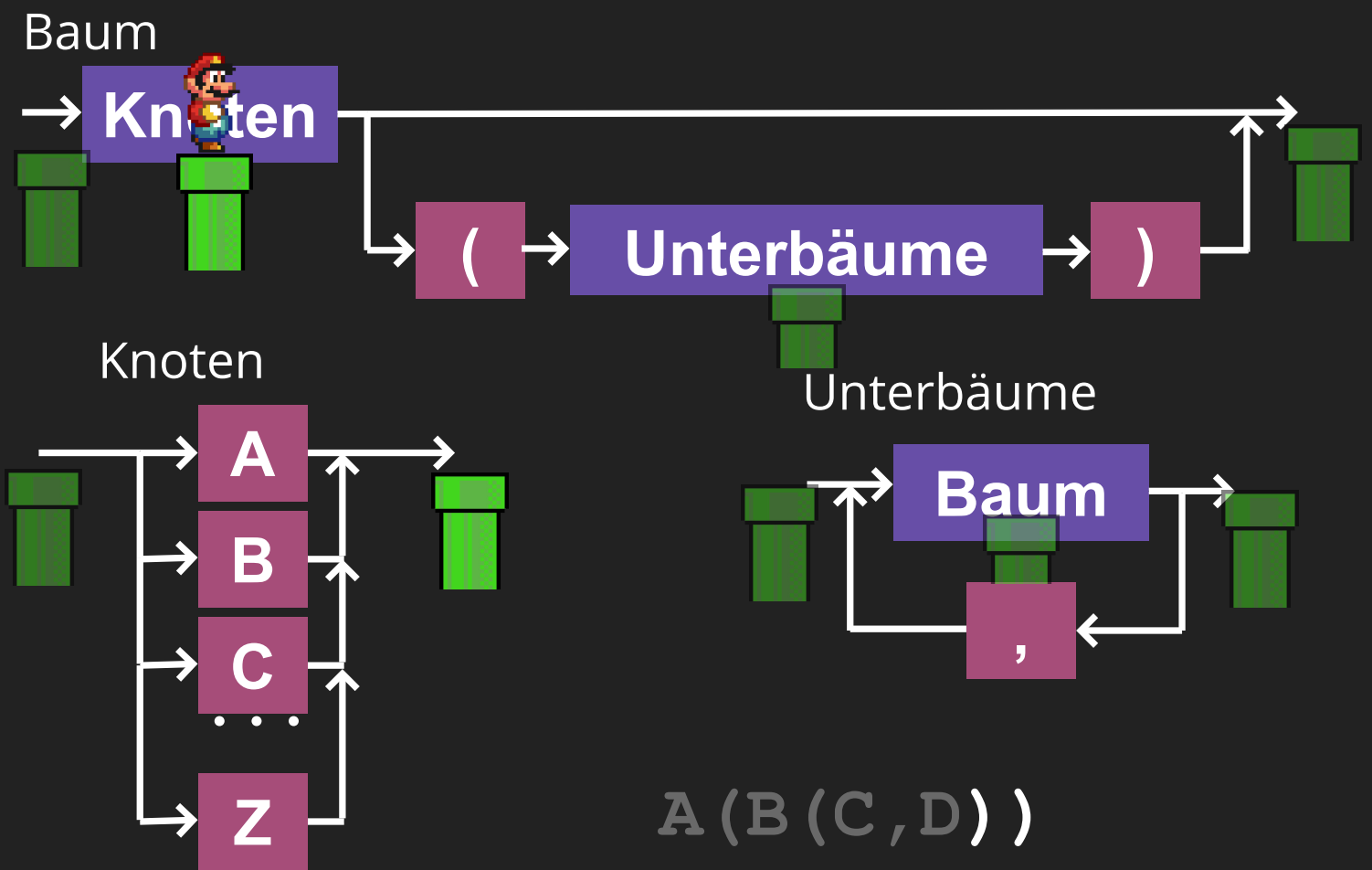



```
A(B(C,D))
```

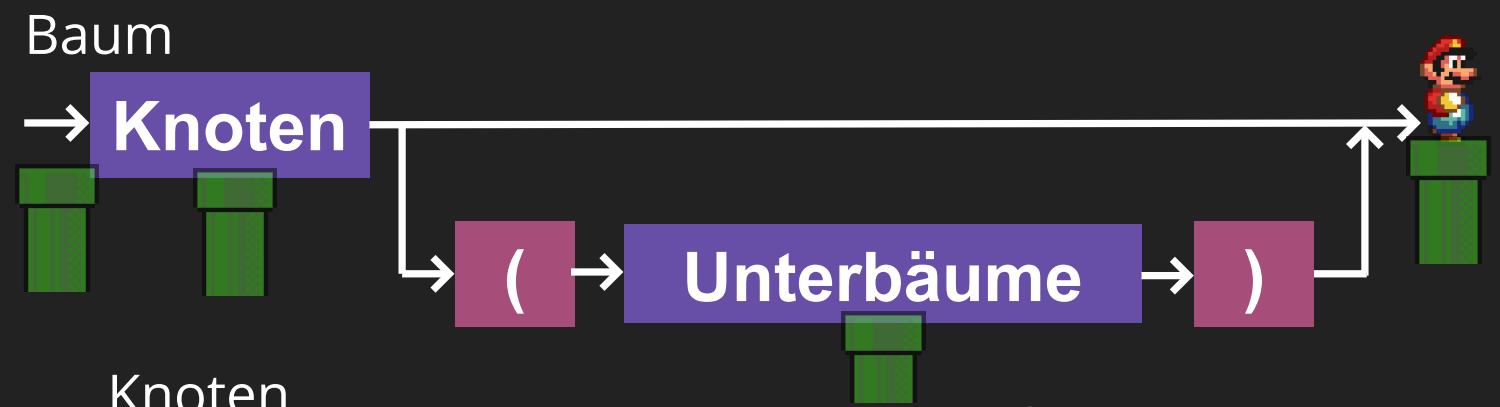
Syntaxdiagramme

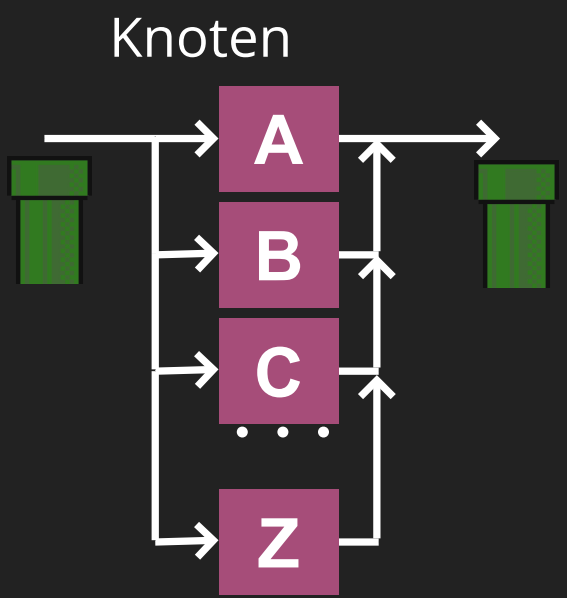


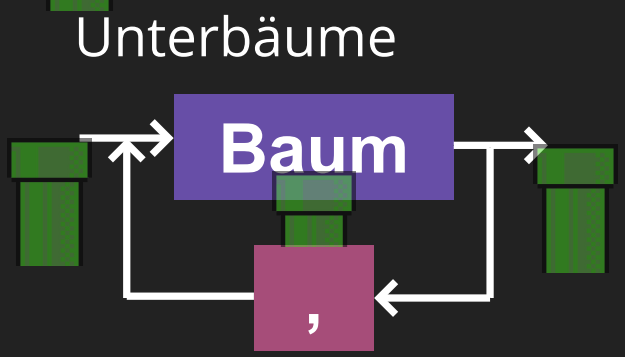
Syntaxdiagramme



Syntaxdiagramme

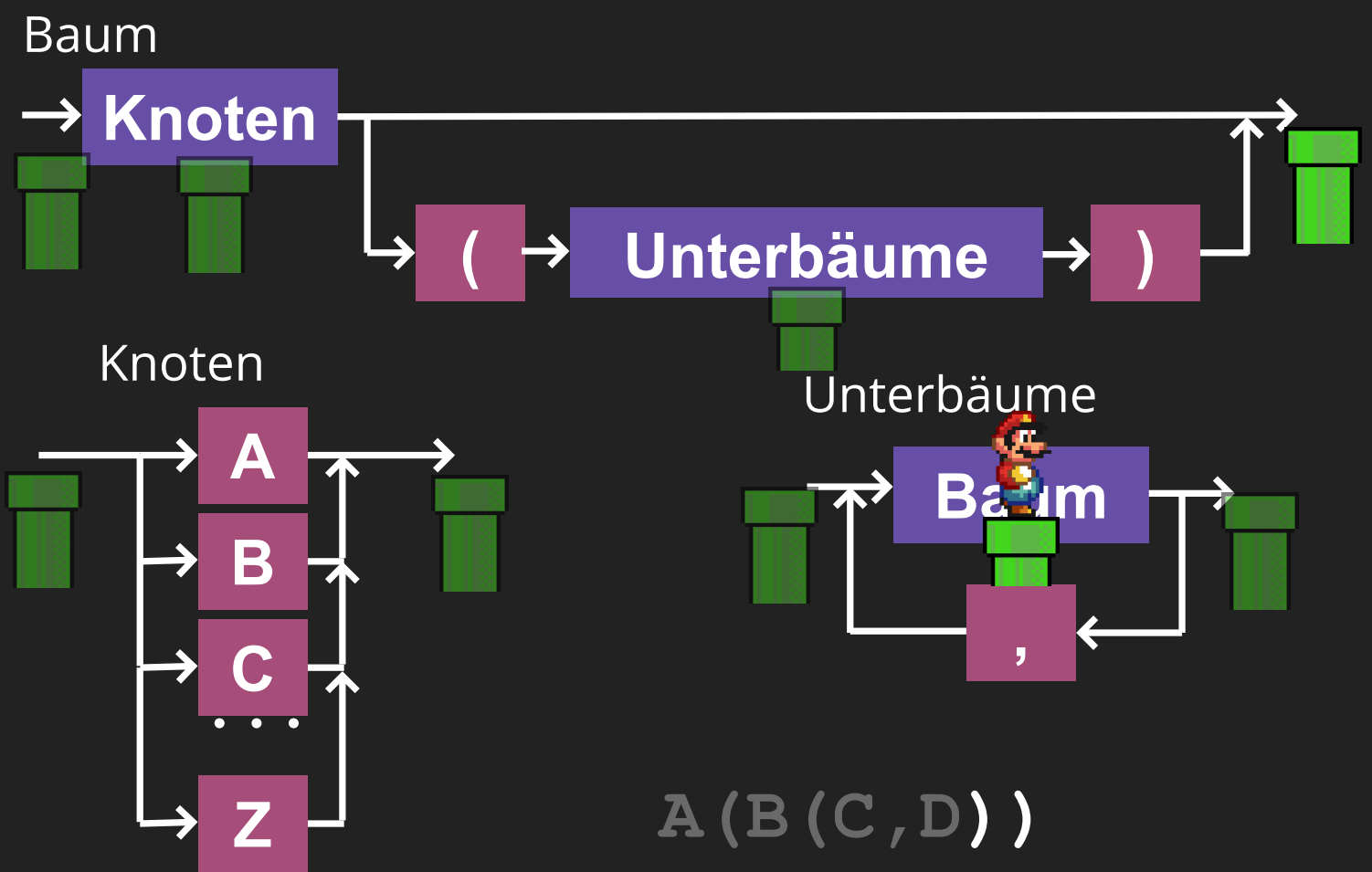






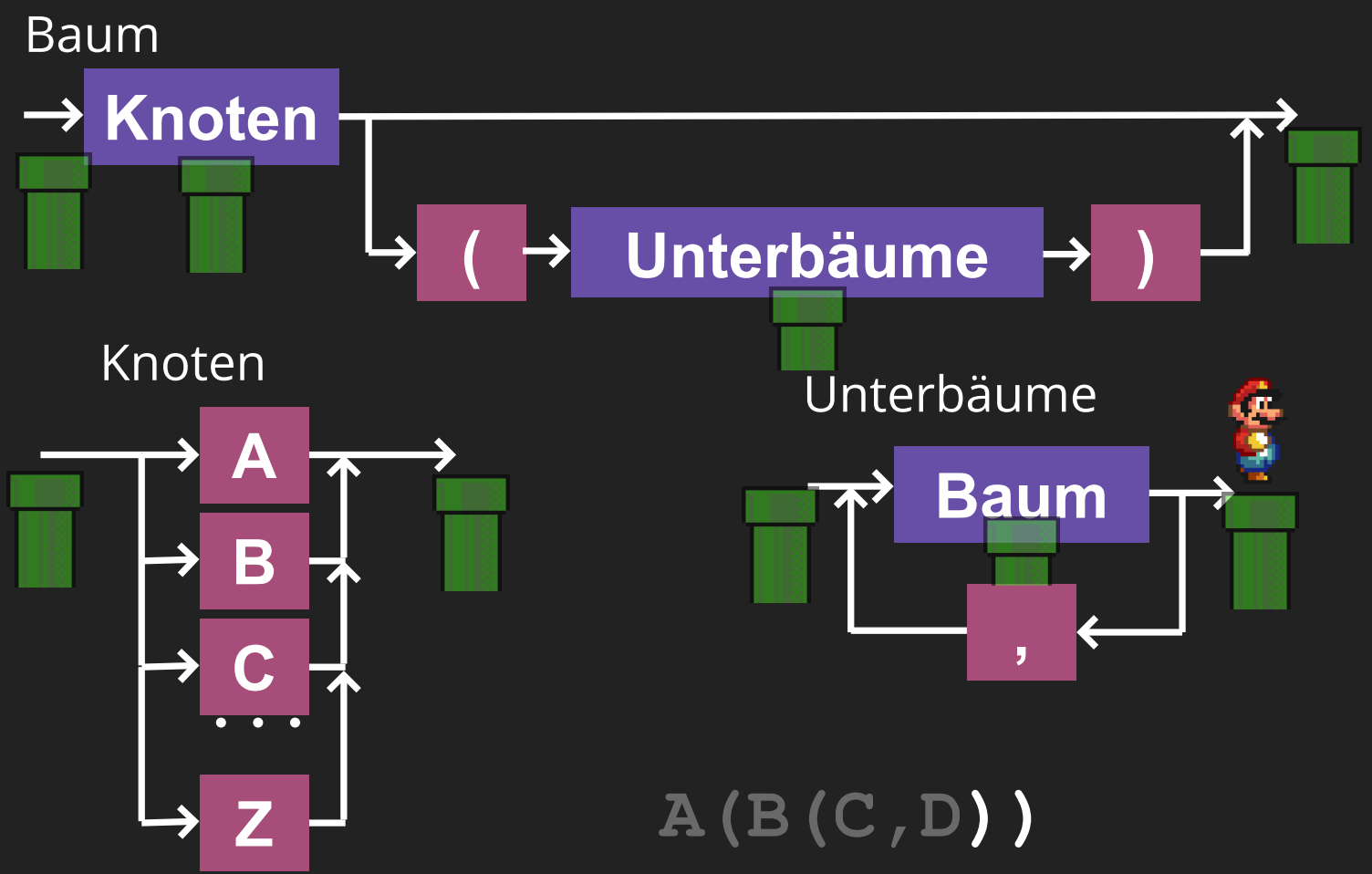

```
A(B(C,D))
```

Syntaxdiagramme

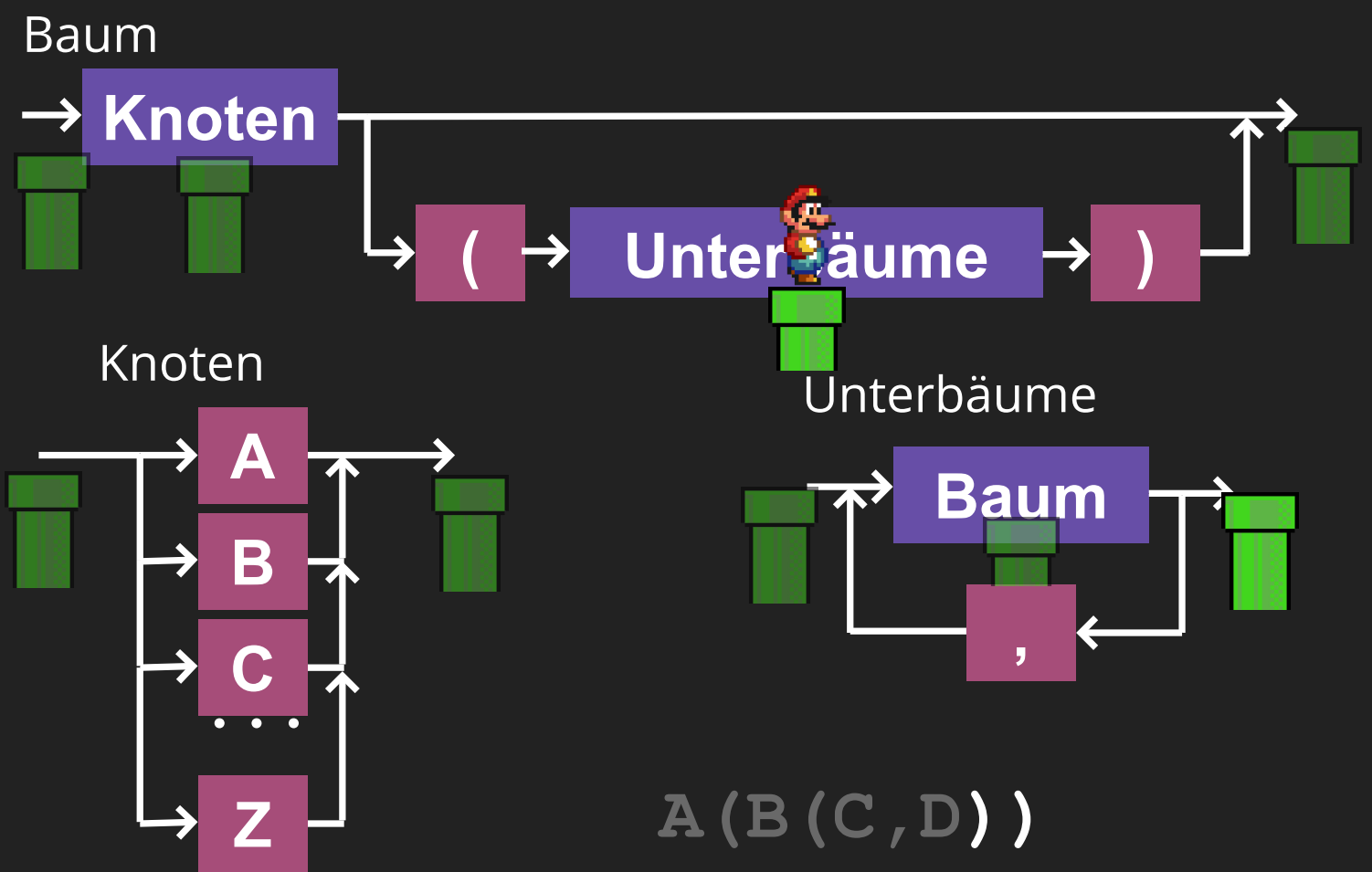



```
A(B(C,D))
```

Syntaxdiagramme

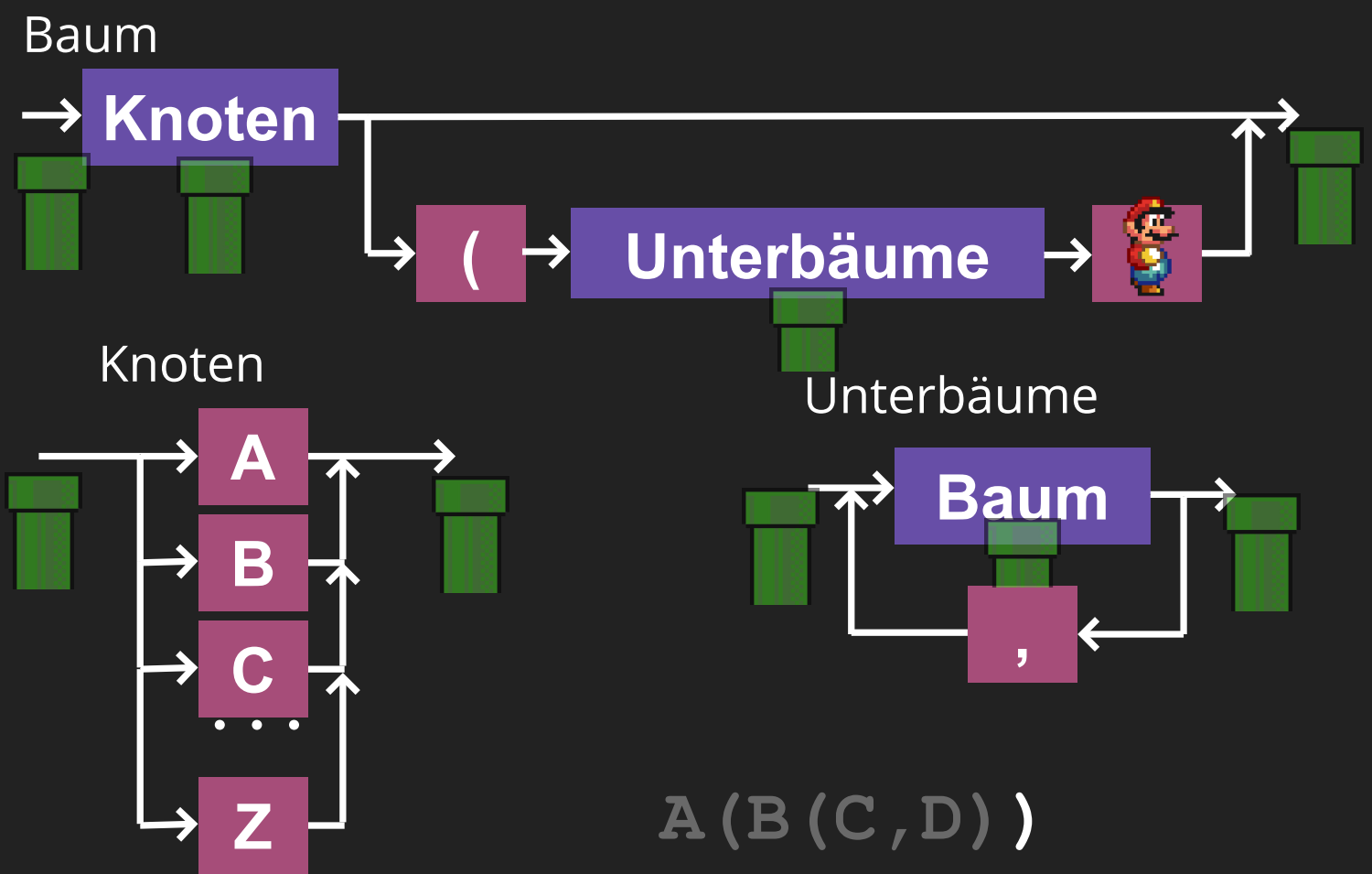


Syntaxdiagramme



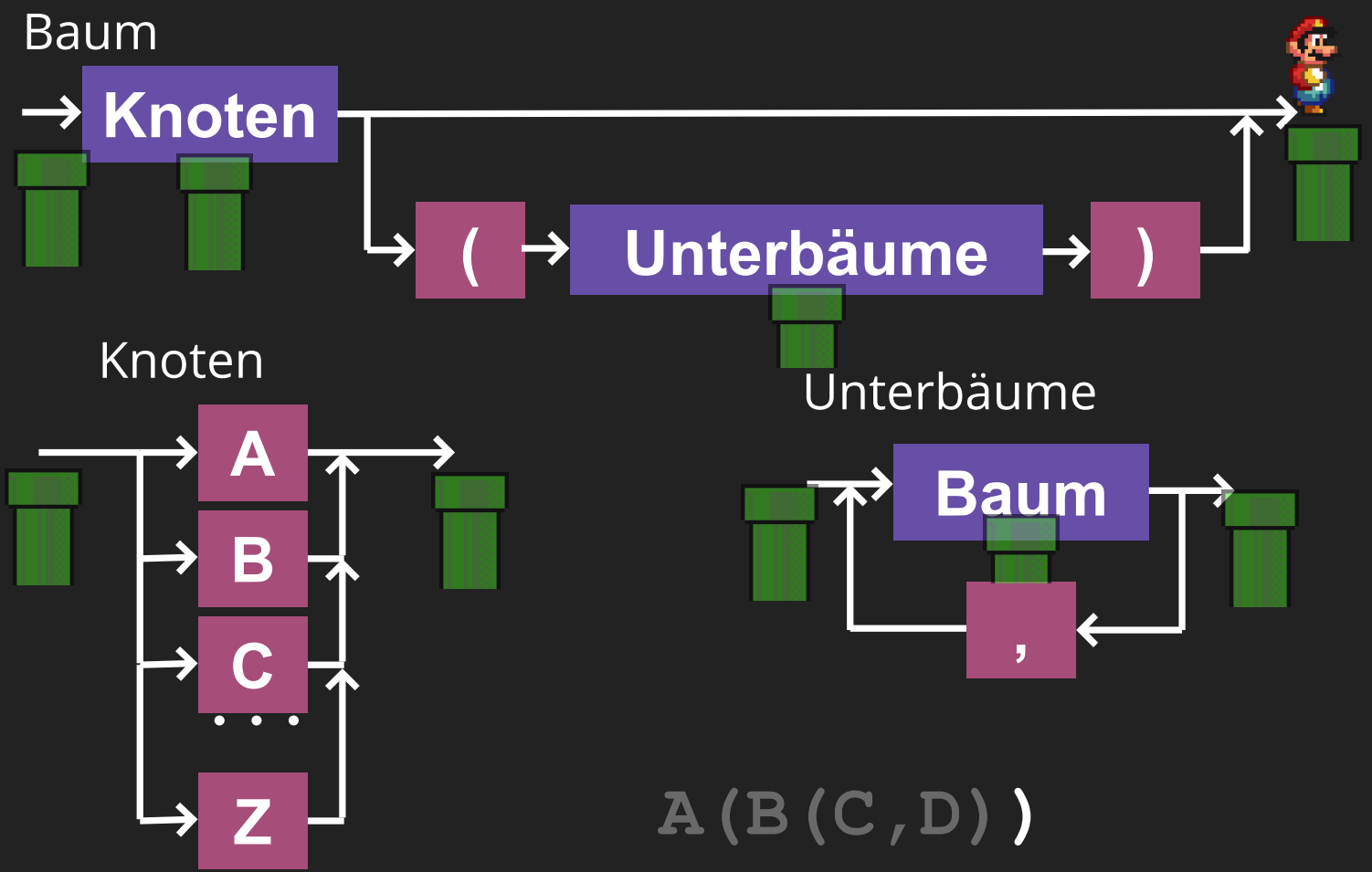

```
A(B(C,D))
```

Syntaxdiagramme



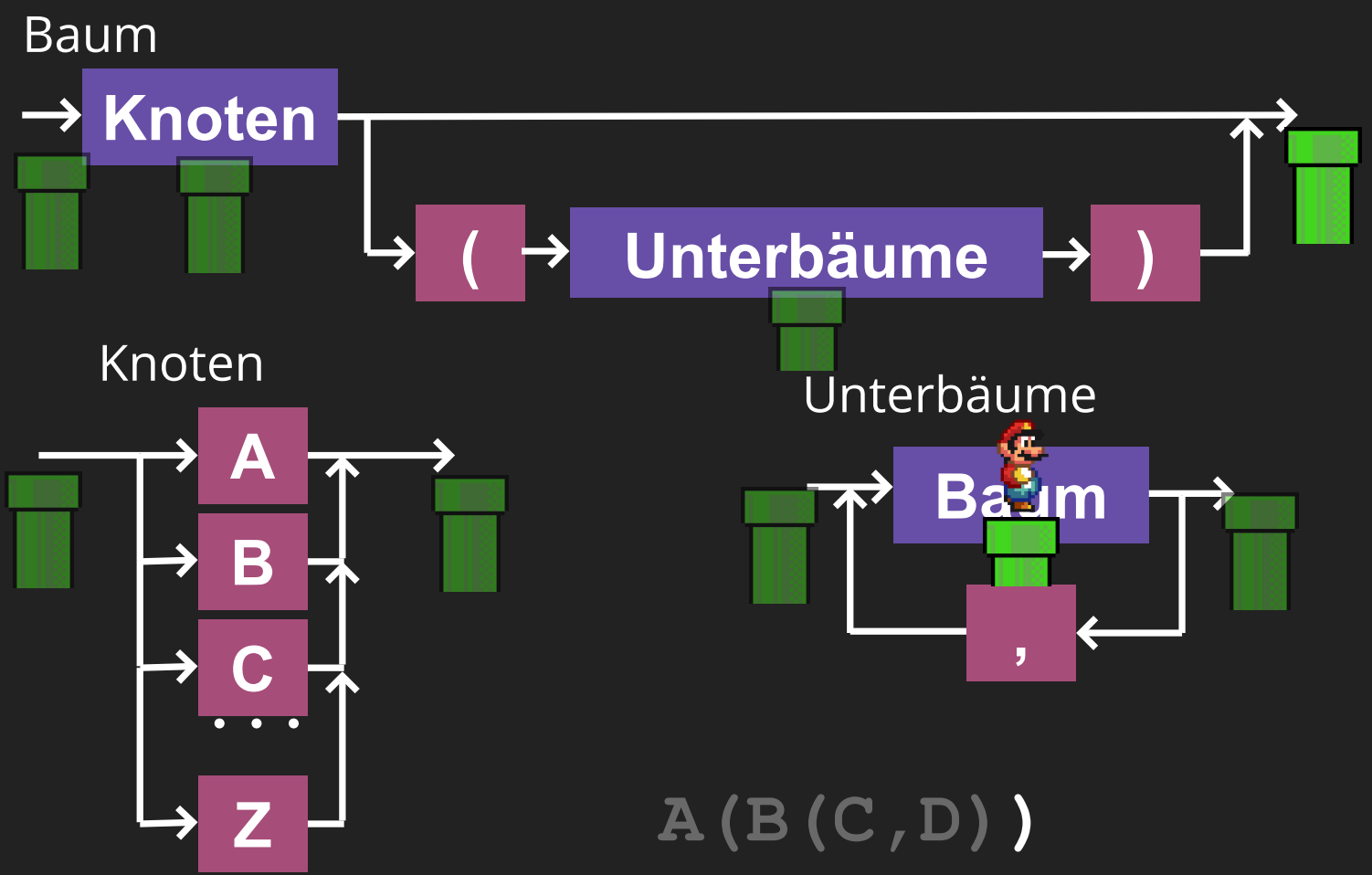

```
A(B(C,D))
```

Syntaxdiagramme



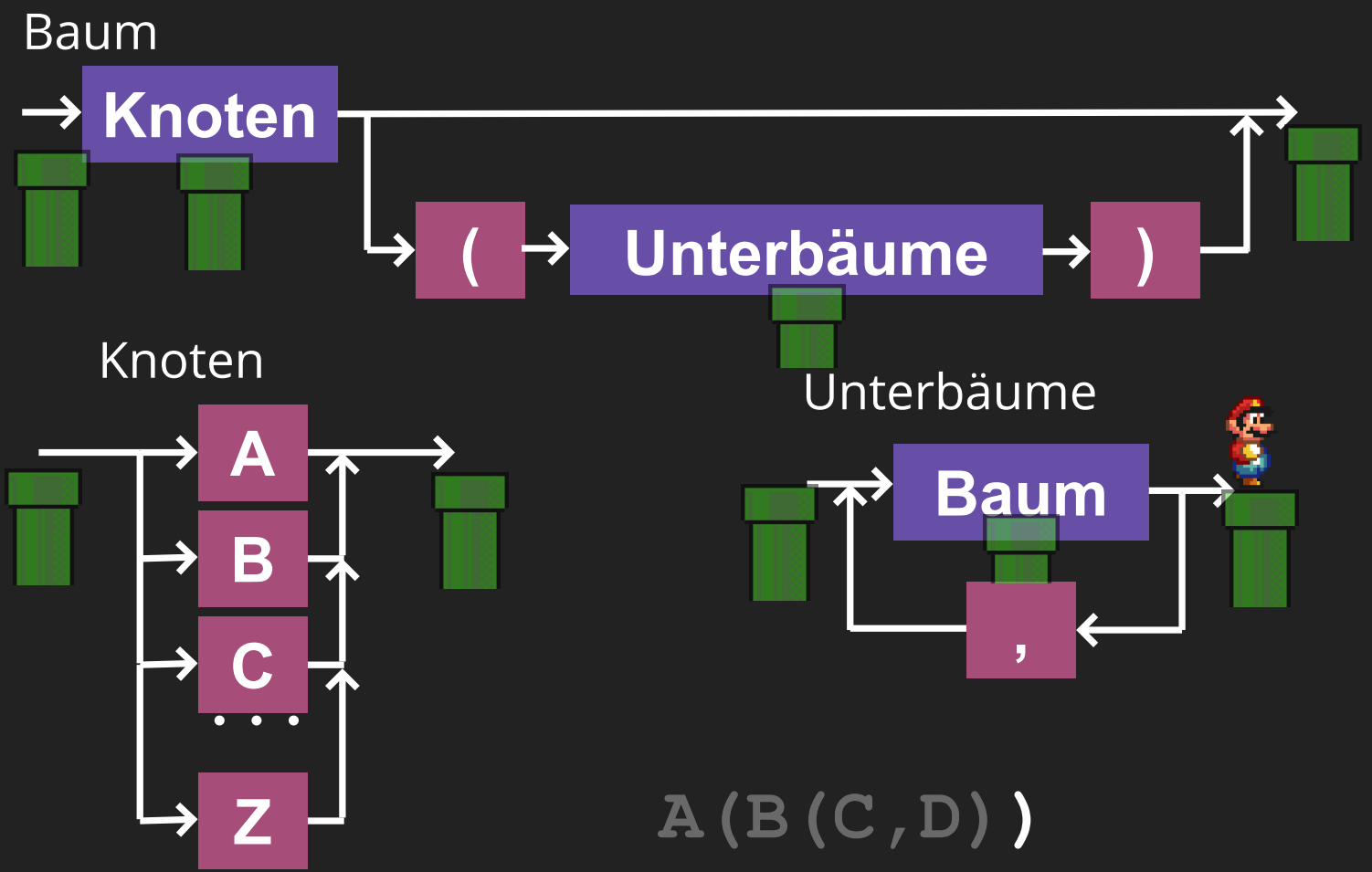

```
A(B(C,D))
```

Syntaxdiagramme



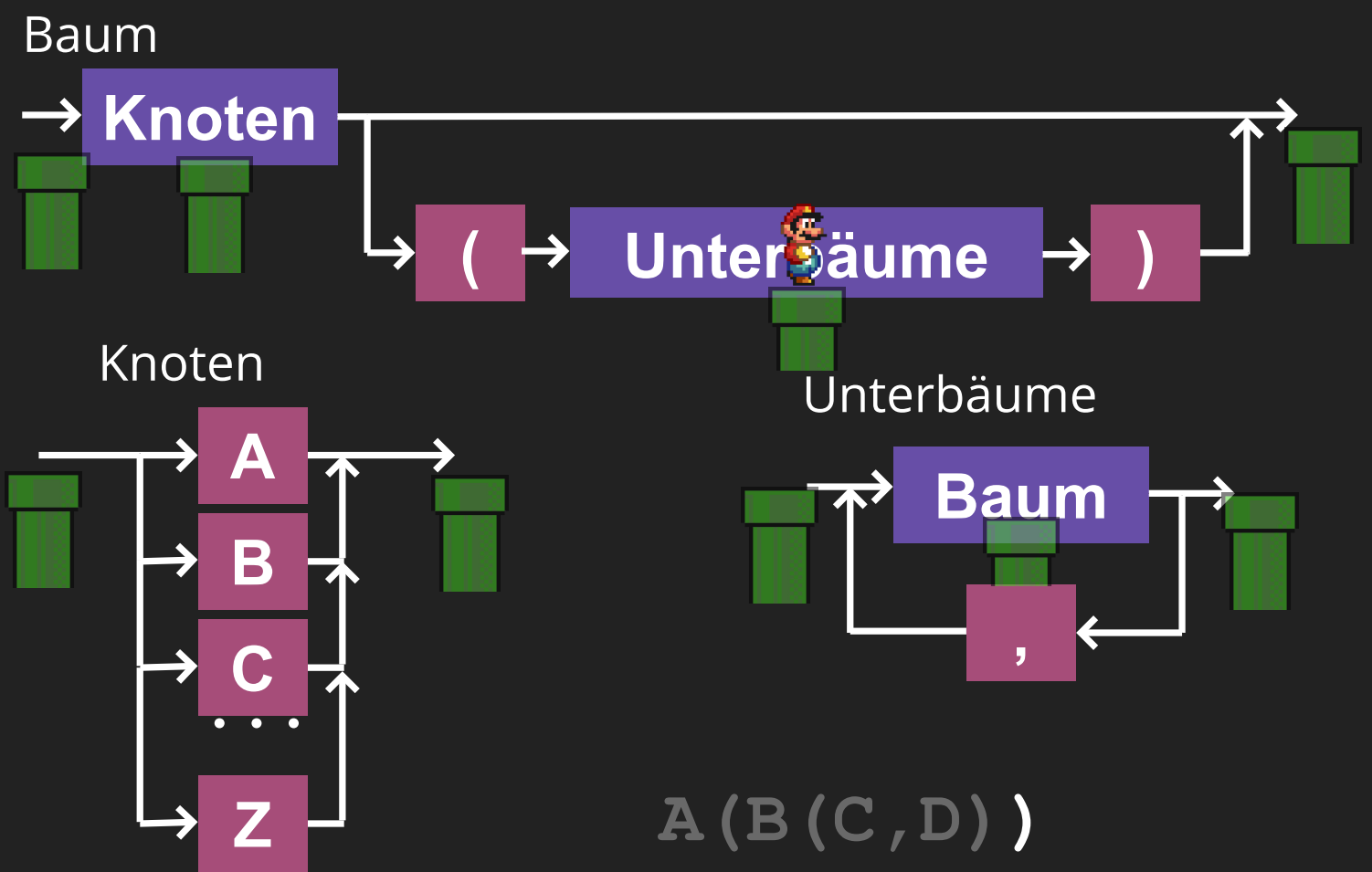
`A(B(C,D))`

Syntaxdiagramme

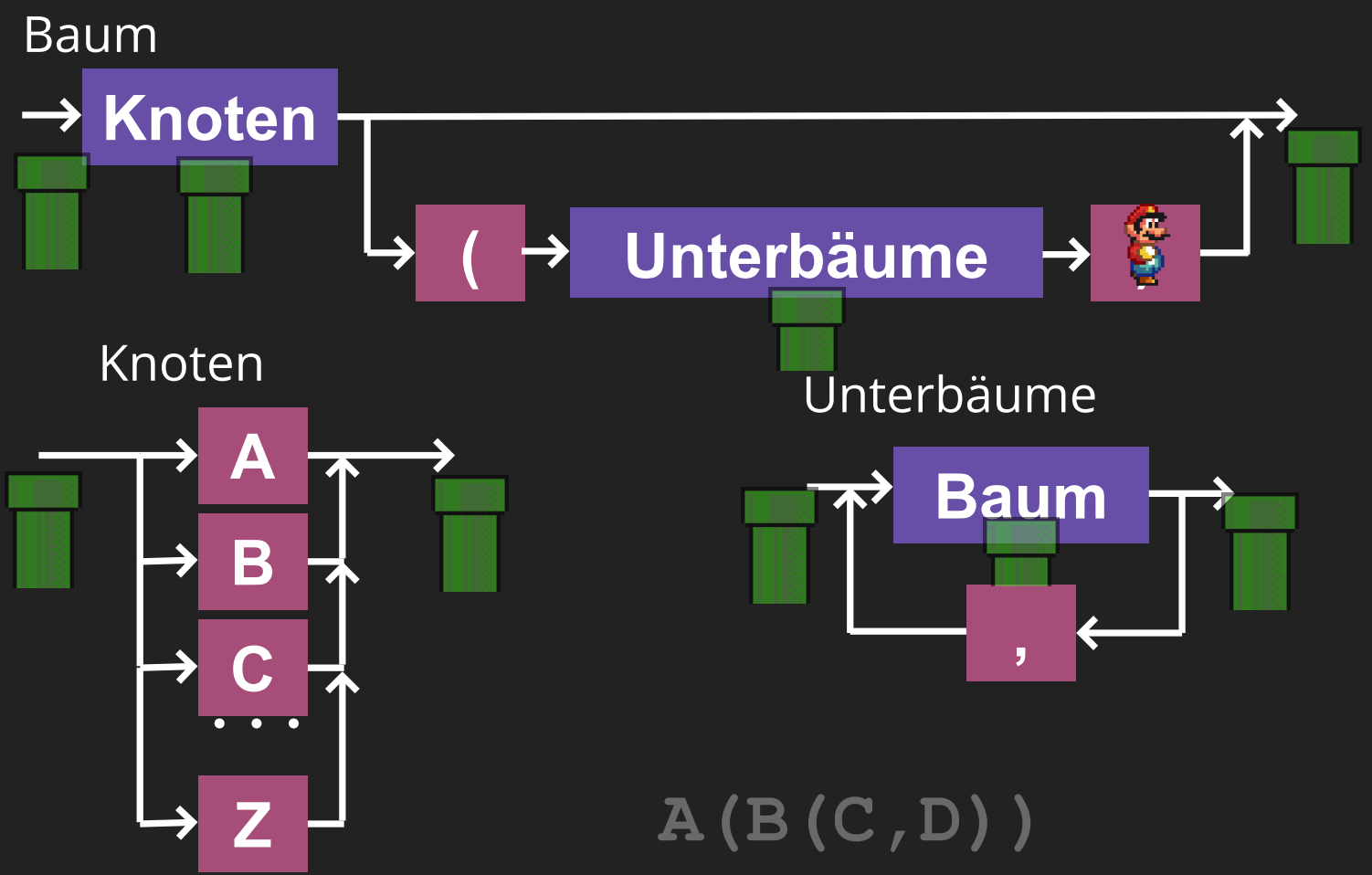



```
A(B(C,D))
```

Syntaxdiagramme

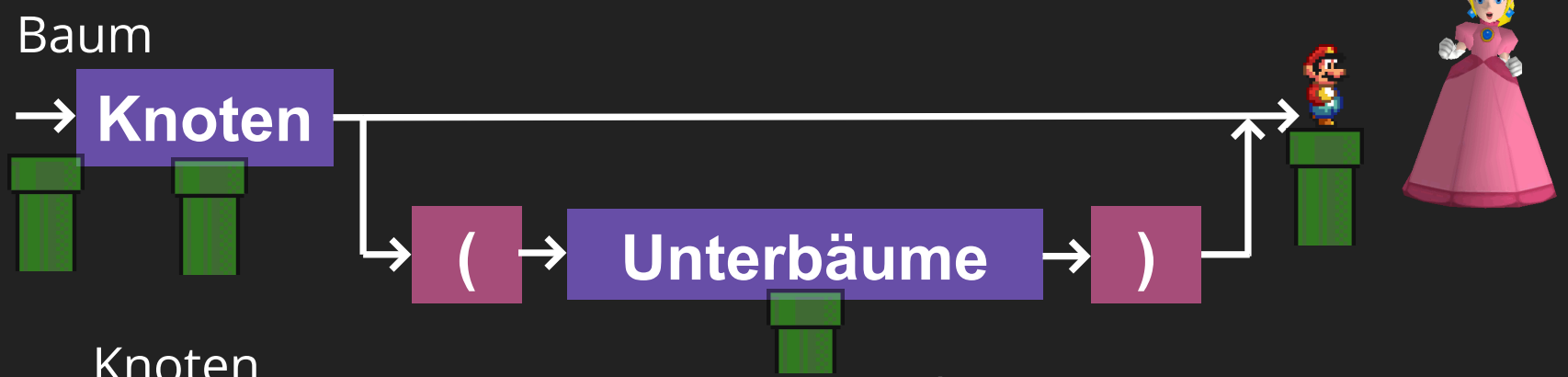


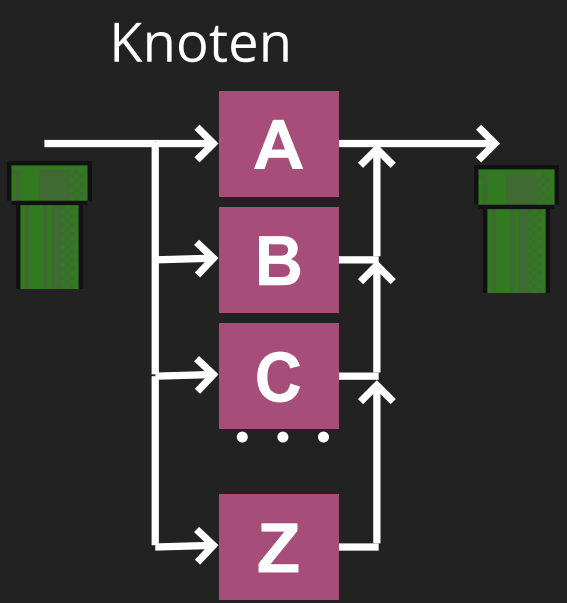
Syntaxdiagramme

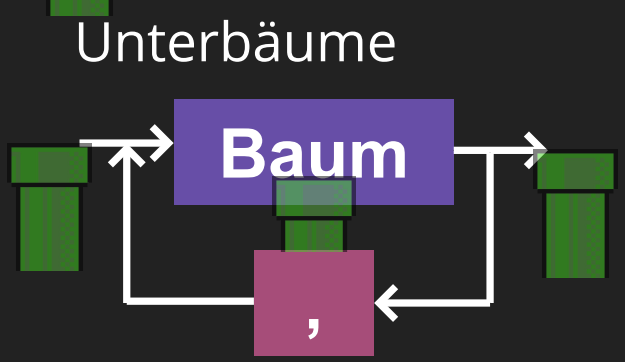



```
A(B(C,D))
```

Syntaxdiagramme

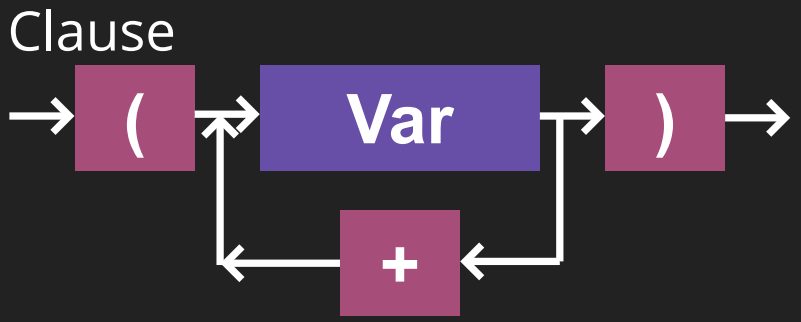







```
A(B(C,D))
```

Syntaxchecker




```
1  private static int parseClause(String kd, int offset) throws ParseException {
2      if (!checkNext('(', kd, offset)) {
3          throw new ParseException("expected '('", offset);
4      }
5      offset++;
6
7      while(true){
8          offset = parseVar(kd, offset);
9          if (!checkNext('+', kd, offset)) {
10             break;
11         }
12         offset++;
13     }
14
15     if (!checkNext(')', kd, offset)) {
16         throw new ParseException("expected ')'", offset);
17     }
18     offset++;
19
20     return offset;
21 }
```

parst '('

parst Var

falls kein '+' folgt, Schleife abbrechen

sonst offset erhöhen ('+' parsen)

parst ')'

neuen offset zurückgeben

Welches ist die korrekte Schleifeninvariante?

```
1 int j = 9;
2 int i = 0;
3
4 while(i < 9){
5       i++;
6       j--;
7 }
```

1. i == 9
2. j == 9
3. **i + j == 9**
4. i - j == 9

Welches ist die korrekte Schleifeninvariante?

```
1  int a = 0;
2  int b = 0;
3
4  while(b < 7){
5        b++;
6        a += 10;
7  }
```

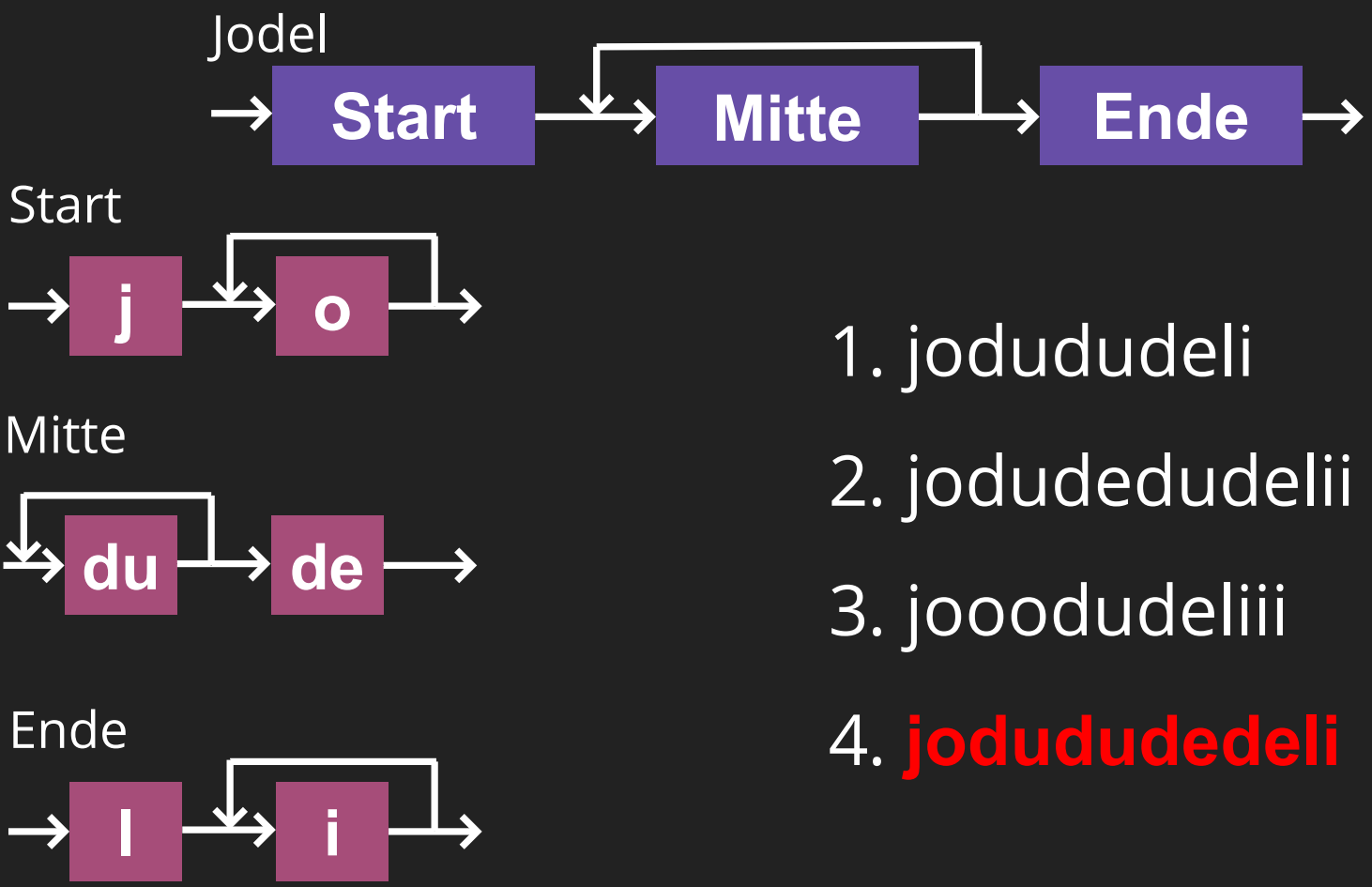
1. a == b + 10
2. **a == b * 10**
3. b == a - 10
4. b == 0

Welches ist die korrekte Schleifeninvariante?

```
1 int info = -1;
2 int II = 1;
3
4 while(info < 2){
5       info++;
6 }
```

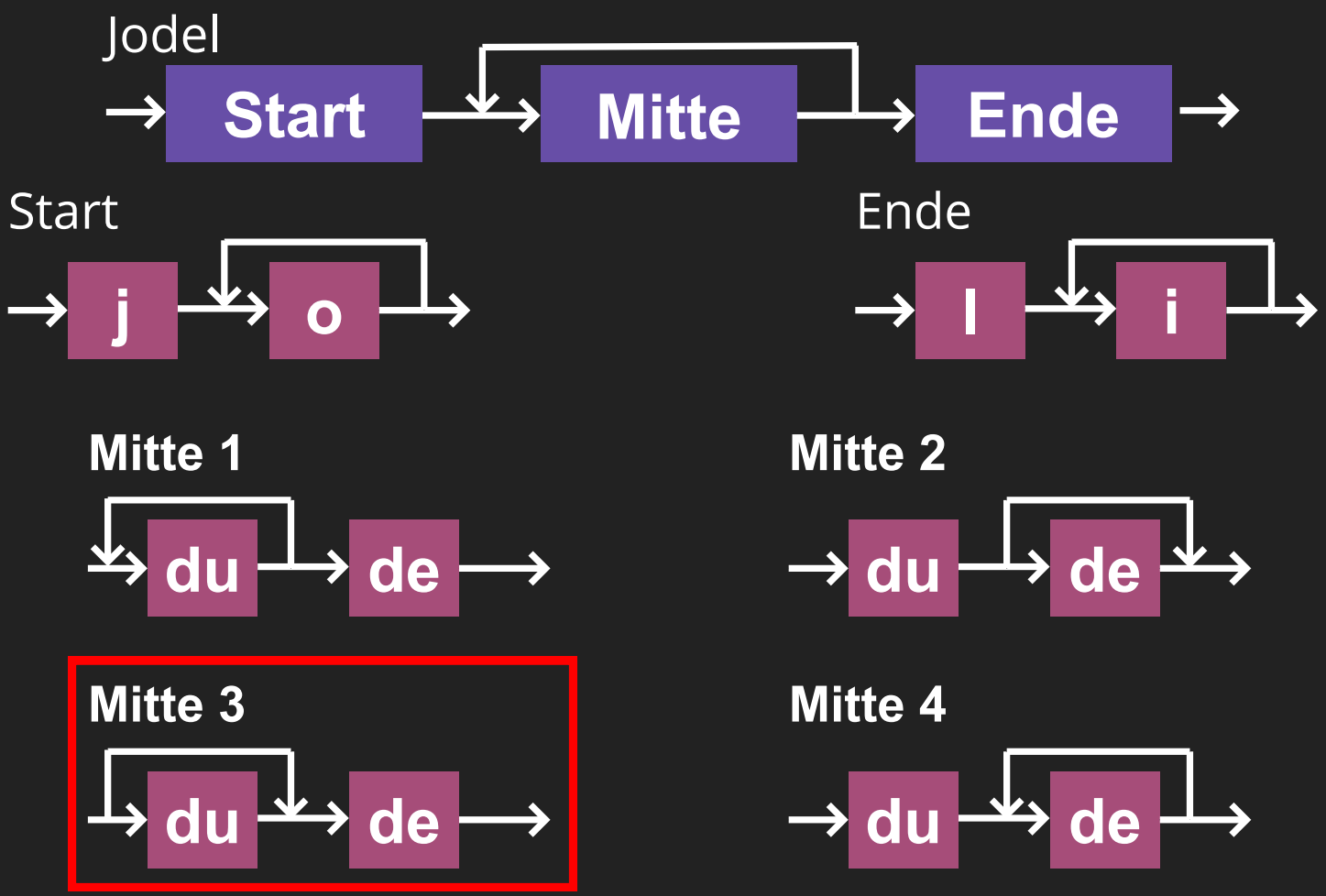
1. `info - II == -2`
2. `info + II == 0`
3. **`info*II <3`**
4. `info / II < 2`

Welcher Jodel ist falsch?



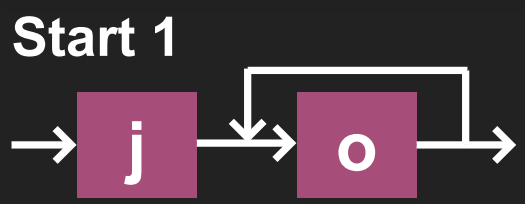
1. jodududeli
2. jodudedudelii
3. jooodudeliii
4. **jodududedeli**

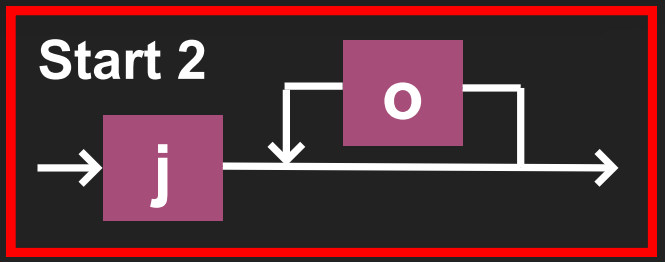
Mit welcher "Mitte" kann man "jodeli" darstellen?



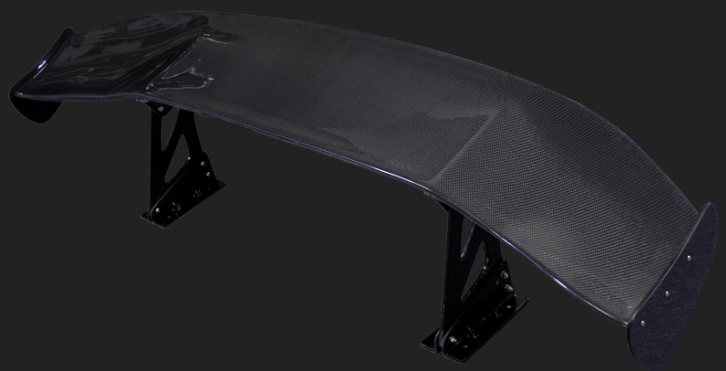
Welches Syntaxdiagramm wird von diesem Programm geprüft?

```java
private static int parseStart(String kd, int offset) throws ParseException{

    if(checkNext('j', kd, offset)){
        offset++;
    }
    else{
        throw ParseException("Expected \"j\" at "+offset);
    }

    while(checkNext('o', kd, offset)){
        offset++;
    }

    return offset;
}
```





Vorbesprechung



Programmverifikation

1. Findet eine Schleifeninvariante.
2. Beweist mittels der gefundenen Schleifeninvariante die Korrektheit der Implementierung.
3. Gilt nach der Änderung an Zeilen 5,6 noch die Schleifeninvariante? Ist der Beweis immer noch gültig?

```
1 static int f(int i, int j) {
2     int u = i;
3     int z = 0;
4     while (u > 0){
5         z = z + j;
6         u = u - 1;
7     }
8     return z;
9 }
```

```
1 static int f(int i, int j) {
2     int u = i;
3     int z = 0;
4     while (u > 0){
5         z = z;
6         u = u;
7     }
8     return z;
9 }
```

Tipp: Der Beweis ist recht ähnlich zu meinem Beispiel im BestOf

String und StringBuffer

Caesar-Verschlüsselung

- Buchstaben werden um 3 weitergesetzt.
  - Geheimniss $\rightarrow$Jhklpqlvv
- Chars können durch Integeroperationen nach ASCII modifiziert werden.
  - 'A' + 1 == 'B'
  - 'A' + '$' == 'e'

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

String und StringBuffer

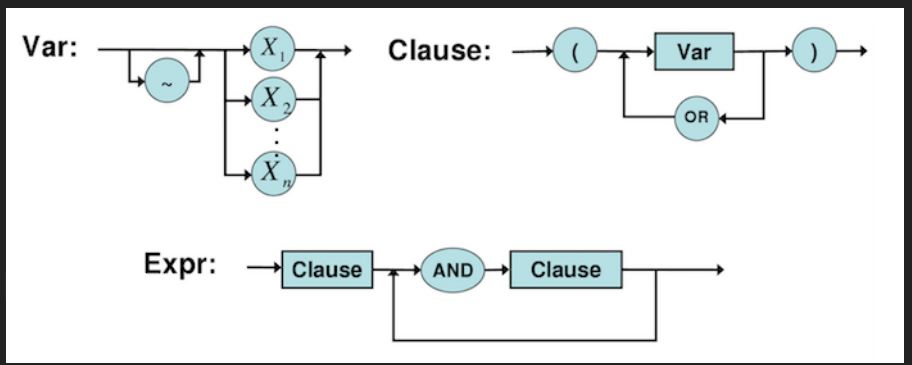
- Die Funktion encrypt ist bereits gegeben, implementiert analog dazu die Funktion decrypt.
- Verwendet StringBuffer in decrypt.
- Der Rückgabewert sollte aber immer noch String sein
  → .toString()

```java
public static String encrypt(String s) {
    String ret = "";
    for (int i = 0; i != s.length(); ++i) {
        ret = ret + (char) (s.charAt(i) + 3);
    }
    return ret;
}
```

```java
public static String decrypt(String s) {
    // TODO
}
```

Syntaxdiagramm

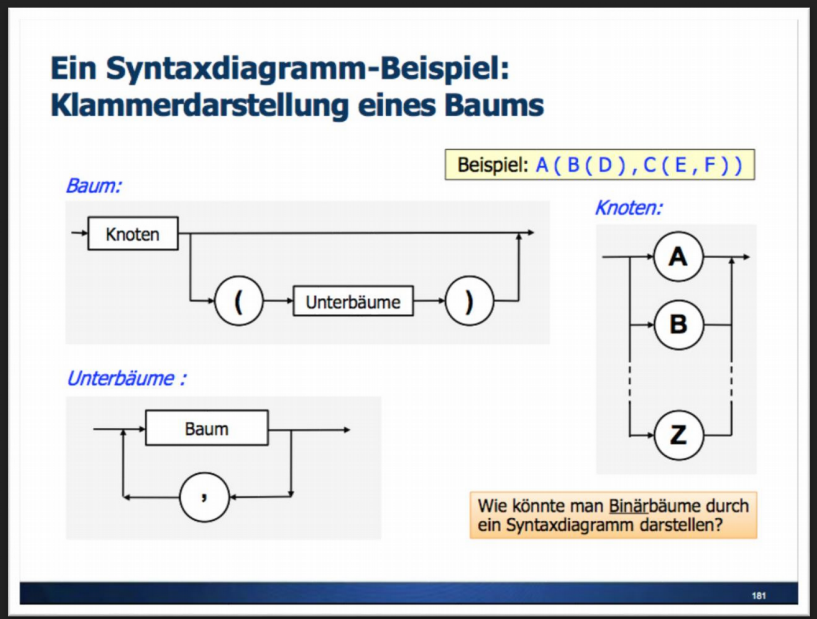
- Sind die folgenden Strings korrekte Clauses?
  1. X_2
  2. $(\sim X_1)$
  3. $\sim (X_1\ OR \sim X_2)$
  4. $(X_2)\ OR\ (\sim X_1\ OR\ X_2)$



- Sind die folgenden Strings ko
  Expressions?
  1. $(X_1\ OR\ X_2)\ AND\ (\sim X_2)$
  2. $(X_1)\ AND\ (\sim X_1\ OR\ \sim X_2)\ AND\ (X_2)$

Syntaxdiagramm für Bäume

- Ergänzen Sie das Syntaxdiagramm , sodass leere Bäume und leere Teilbäume generiert werden können.
- Ein leerer Baum bzw. Teilbaum soll dabei durch ein '-' repräsentiert werden.

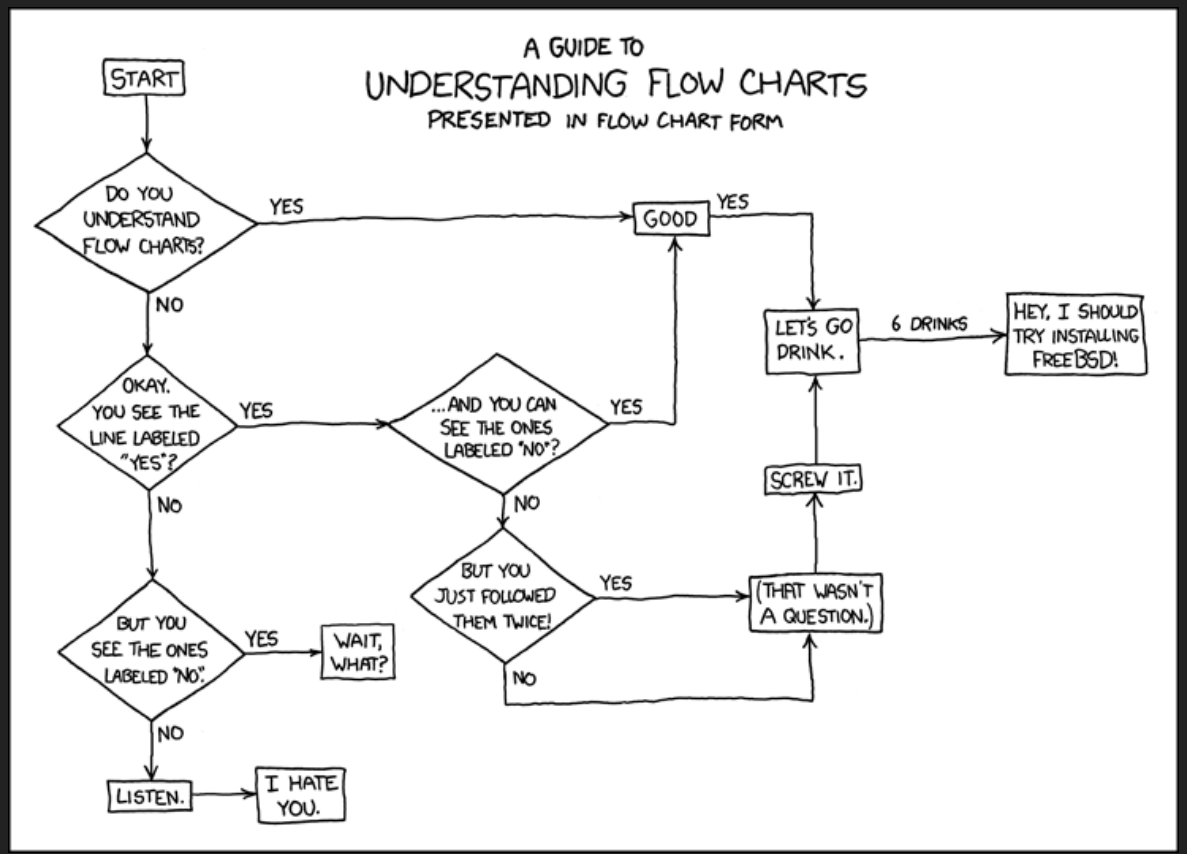


Syntaxchecker

- Implementiert einen Syntaxchecker für Wurzelbäume in der Klammerdarstellung.
- **Tipp**: Verwendet die folgenden Funktionen:

```
1 //checks if the char at position offset in the String kd is equal to c
2 Boolean checkNext(char c, String kd, int offset){ ... }
3 int parseTree(String kd, int offset){ ... }
4 int parseSubtree(String kd, int offset) { ... }
5 int parseNode(String kd, int offset){ ... }
```

Viel Spass!


A GUIDE TO
UNDERSTANDING FLOW CHARTS
PRESENTED IN FLOW CHART FORM
START
DO YOU UNDERSTAND FLOW CHARTS?
YES
GOOD
YES
NO
LET'S GO DRINK.
6 DRINKS
HEY, I SHOULD TRY INSTALLING FREEBSD!
OKAY. YOU SEE THE LINE LABELED "YES"?
YES
...AND YOU CAN SEE THE ONES LABELED "NO"?
YES
SCREW IT.
NO
NO
BUT YOU JUST FOLLOWED THEM TWICE!
YES
(THAT WASN'T A QUESTION.)
BUT YOU SEE THE ONES LABELED "NO".
YES
WAIT, WHAT?
NO
NO
LISTEN.
I HATE YOU.