Parsování aritmetického výrazu.

Bezkontextová gramatika. Lexikální analýza. Syntaktická analýza. Sémantická analýza.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

Obsah přednášky

1. Jazyk

Každý přirozený *jazyk* založen na *slovníku*.

Obsahuje všechna *slova* v jazyce používaná.

Posloupnost slov tvoří *věty*.

Tvoření vět podléhá pravidlům, věta vytvořena správně/špatně.

```
Psi štěkají. vs. Štěkají psi.
Kočky mňoukají. vs. Mňoukají kočky.
```

Pravidla definovaná *syntaxí*.

Soubor pravidel definující množinu korektních vět.

U přirozených jazyků větší volnost co je/není správná věta.

Struktura věty důležitá pro její význam, *sémantiku*.

Věta gramaticky správná + rozumíme ji.

U přirozených jazyků více významů, u formálních nelze.

2. Příklad

Popis syntaxe:

```
<Veta> → <Podmět> <Přísudek>
<Podmět> → <psi | kočky>
<Přísudek> → <štěkají | mňoukají>
```

Tento zápis definuje pravidla:

P_1: Věta může být tvořena `Podmětem` následovaným `Přísudkem`.

P_2: Věta začíná `Podmětem`.

P_3: Podmět může být `psi` nebo `kočky`.

P_4: Přísudek může být `štěkají` nebo `mňoukají`.

$P_1 - P_4$: Definují množinu vět (konečná) jako $\forall$ kombinace podmětu/přísudku:

```
psi štěkají
psi mňoukají
kočky štěkají
kočky mňoukají
```

Neterminály (slovní druhy, abstraktní): `Podmět, Přísudek.`

Počáteční neterminál: `Podmět.`

Terminály, tzv. tokeny, zástupci (konkrétní): `psi, kočky, štěkají, mňoukají.`

3. Chomského hierarchie

Autor Noam Chomsky, 1956.

1. *Gramatiky typu 0*
   Rekurzivně spočetné jazyky, L_0.
   Neomezená kritéria generování, zpracovávány Turingovým strojem.
2. *Gramatiky typu 1*
   Kontextové jazyky, L_1,

$$aX\beta \to a\gamma\beta, \quad X \in N, \alpha, \beta, \gamma \in N \cup T.$$

3. *Gramatiky typu 2*
   Bezkontextové gramatiky, L_2, obecnější

$$X \to \alpha, \quad X \in N, \alpha \in N \cup T.$$

   Významná skupina, většina programovacích jazyků.
4. *Gramatiky typu 3*
   Regulární jazyky L_3, nejjednodušší jazyky.

$$X \to a|aY, \quad a \in T, \quad X, Y \in N.$$

3. Bezkontextová gramatika (CFG)

Gramatika $\mathcal{G}$ (bezkontextová, CFG) definována jako uspořádána čtveřice

$$\mathcal{G} = (N, T, S, P), \qquad N \cap T = \emptyset, P \subseteq N \times \{N \cup T\},$$

kde:

N konečná množina neterminálních symbolů: $A, B, ..., Z$, (syntaktické kategorie),

T konečná množina neterminálních symbolů: $a, b, ..., z, 0, ..., 9$, (dávají větě smysl),

S, $S \in N$, startovní (počáteční) neterminál,

P konečná množina přepisovacích (produkčních) pravidel.

Řetězec: kombinace terminálů a neterminálů, $\alpha, ..., \omega$.

Přepisovací (derivační) pravidlo P (přepiš A na α s využitím P)

$$P : A \to \alpha, \qquad A \in N, \alpha \in N \cup T.$$

Bezkontextová gramatika: způsob úplné definice jazyka, slouží pro generování slov.
Lze ji vyjádřit většinu syntaktických struktur programovacích jazyků.

Dělení CFG:

- nerekurzivní: generuje konečný počet slov.
- rekurzivní: generuje nekonečný počet slov.

4. Ukázka CFG, nerekurzivní

Příklad 1: CFG, nerekurzivní.

Gramatika $\mathcal{G} = (N, T, S, P)$:

$A \equiv$ `Podmět`, $B \equiv$ `Přísudek`, $C \equiv$ `Věta`.

$a \equiv$ `psi`, $b \equiv$ `kočky`.

$c \equiv$ `štěkají`, $d \equiv$ `mňoukají`.

Logické operátory: $x|y$ (OR), xy (AND), $\overline{x}$ (NOT).

Pak, $\mathcal{G} = (N, T, S, P)$,

$$N = \{A, B\},$$
$$T = \{a, b, c, d\},$$
$$S = \{A, B\},$$
$$P = \{P_1, P_2, P_3, P_4\}.$$

Pravidla $P_1 - P_4$, vyhodnocování $\rightarrow$:

$P_{1,2} : S \rightarrow AB$, (Větu tvoří Podmět a Přísudek, začíná podmětem).

$P_3 : A \rightarrow a|b$, (Podmětem jsou psi nebo kočky).

$P_4 : B \rightarrow c|d$, (Přísudkem jsou štěkají nebo mňoukají).

Modifikace: věta začíná podmětem nebo přísudkem:

$$P_{1,2} : S \rightarrow AB|BA.$$

5. Ukázka CFG, rekurzivní, přirozené jazyky

Pokus o definici syntaxe anglického jazyka, Noam Chomsky (1957).

Příklad 2: CFG, rekurzivní, zjednodušená varianta.

```
<sentence>      → <noun phrase> <verb phrase>
<noun phrase>   → <adjective> <noun phrase> |
                  <adjective> <singular noun>
<verb phrase>   → <singular verb> <adverb>
<adjective>     → <a | the | little | young>
<singular noun> → <boy | girl>
<singular verb> → <ran | walked>
<adverb>        → <quickly | slowly>
```

Příklady:

```
a little boy ran quickly.
a young boy walked quickly.
the young girl ran slowly.
```

Přirozený jazyk tak bohatý, že nelze vytvořit úplný CFG.

Přirozeně rekurzivní, nekonečná slovní zásoba.

6. Přepis do $\mathcal{G}$

Pak, $\mathcal{G} = (N, T, S, P)$,

$$
\begin{aligned}
N &= \{NP, VP, AD, A, SN, SV\},\\
T &= \{a, b, c, d, e, f, g, h, i, j\},\\
S &= \{NP, VP\},\\
P &= \{P_1, P_2, P_3, P_4, P_5, P_6, P_7\}.
\end{aligned}
$$

Pravidla:

$$
\begin{aligned}
P_1 &: S \to NP\ VP,\\
P_2 &: NP \to AD\ NP \mid AD\ SN,\\
P_3 &: VP \to SV\ A,\\
P_4 &: AD \to \{a|b|c|d\},\\
P_5 &: SN \to \{e|f\},\\
P_6 &: SV \to \{g|h\},\\
P_7 &: A \to \{i|j\}.
\end{aligned}
$$

7. Ukázka rekurzivní CFG

Definujme prázdné slovo ϵ, tzv. *vypouštěcí gramatika*.

Přidejme logické operátory.

Příklad 3: Rekuzivní CFG:

Gramatika $\mathcal{G} = (N, T, S, P)$: $N = \{S\}$, $T = \{0, 1\}$, $P = \{P_1, P_2\}$.

Dvě pravidla, první generuje prázdný vstup, druhé rekurzivní.

$$P_1 : S \to \epsilon,$$
$$P_2 : S \to 0S1.$$

Pokud levá strana stejná, lze přepsat do tvaru

$$S \to \epsilon | 0S1.$$

Pak

$$S \Rightarrow \epsilon,$$
$$S \Rightarrow 0S1 \Rightarrow 01,$$
$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011,$$
$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000111.$$

Gramatika $\mathcal{G}$ generuje posloupnost

$$S \Rightarrow \{0^n 1^n\}.$$

Posloupnosti lze prodlužovat až do délky n: vznikají derivace délky n.

8. Ukázka rekurzivní CFG

Příklad 4: Generování palindromů

Gramatika $\mathcal{G} = (N, T, S, P)$:$N = \{S\}, T = \{a, b\}, P = \{P_1, P_2, P_3\}$.

Palindrom: zrcadlove symetrický řetězec:

```
aba, bab, abba, baab, ababa, babab,...
```

Pokud S představuje a nebo b, doplníme zleva/zprava stejným znakem:

```
aSa, bSb,   S=b, S=a.
```

Pokud S je již řetězec z předchozí iterace, analogie

```
ababa = aSa, kde S = bab
babab = bSb, kde S = aba
```

3 pravidla: 1 nerekurzivní, 2 rekurzivní

$$P_1 : S \to \epsilon,$$
$$P_2 : S \to aSa,$$
$$P_3 : S \to bSb,$$

se stejnou levou stranou lze přepsat do tvaru

$$S \to \epsilon | aSa | bSb.$$

Pak

$$S \Rightarrow \epsilon,$$
$$S \Rightarrow aSa \Rightarrow aa,$$
$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba.$$

Posloupnosti lze prodlužovat až do délky n: vznikají derivace délky n.

9. Derivace délky *n*

Derivace délky n je posloupnost $\beta_0, \beta_1, ..., \beta_n$, kde $\beta_i \in N \cup T$ a pro všechna i, $1 \leq i \leq n$,

$$\beta_{i-1} \Rightarrow \beta_i.$$

Definice říká, že pravidlem P ze slova β_0 lze odvodit slovo β_1, z něj slovo β_2, atd, což lze neomezeně opakovat

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow ... \Rightarrow \beta_{n-1} \Rightarrow \beta_n.$$

Příklad: *Gramatika* $S \rightarrow \epsilon|aSa|bSb$*:*

$$aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow ababSbaba \Rightarrow ...,$$
$$(\beta_0) \Rightarrow (\beta_1) \Rightarrow (\beta_2) \Rightarrow (\beta_3) \Rightarrow ...$$

Skutečnost, že pro nějaké $\alpha = \beta_0$ a $\alpha' = \beta_n$ existuje derivace

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow ... \Rightarrow \beta_{n-1} \Rightarrow \beta_n,$$

zapisujeme

$$\alpha' \Rightarrow^n \alpha, \text{resp. } \alpha' \Rightarrow^* \alpha.$$

pro nějaké $n \in N$.

Říkáme, že α generuje α'.

10. Jazyk $L(\mathcal{G})$

Větná forma α:

Větná forma α, $\alpha \in N \cup T$, je taková derivace délky n

$$S \Rightarrow^* \alpha,$$

kde S je počáteční neterminál. Vzniká odvozením z počátečního neterminálu pro nějaké n za použití pravidel P, např.

$$\alpha = abSba.$$

Slovo w (word):

Slovo w, $w \in T$, je taková derivace délky n

$$S \Rightarrow^* w,$$

kde S je počáteční neterminál. Z α vznikne dosazením za neterminály, např.

$$w = ababa.$$

Jazyk $L(\mathcal{G})$:

Jazyk $L(\mathcal{G})$ generovaný gramatikou $\mathcal{G} = (N, T, S, P)$ je množina všech slov w v abecedě T, kterou lze odvodit z počátečního neterminálu S pomocí pravidel P

$$\mathcal{L}(\mathcal{G}) = \{w \in T, S \Rightarrow^* w\}.$$

Jazyk $L(\mathcal{G})$ je bezkontextový, pokud $\mathcal{G} = (N, T, S, P)$ je bezkontextová.

Tvoří ho pouze terminální slova odvozená od S.

11. Ukázka jazyka $L(\mathcal{G})$

Vytvoření gramatiky $\mathcal{G} = (N, T, S, P)$ generující jazyk

$$L = \left\{ a^n b^n, n \geq 0 \right\}.$$

Pak $N = \{S\}$, $T = \{a, b\}$, $P = \{P_1\}$, kde

$$S \to \epsilon | aSb.$$

$$S \Rightarrow \epsilon,$$
$$S \Rightarrow aSb \Rightarrow ab,$$
$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb,$$
$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb.$$

Větné formy $L(\mathcal{G})$

$$\alpha = \{\epsilon, aSb, aaSbb, aaaSbbb, ...\}.$$

Jednotlivá slova $L(\mathcal{G})$

$$w = \{ab, aabb, aaabbb, ...\}.$$

12. Syntaktický/derivační strom (Parse Tree)

Grafické vyjádření struktury věty w v $\mathcal{G} = (N, T, S, P)$.

$\mathcal{T}(\mathcal{G})$ tvořen uzly $v_0, ..., v_k$, kořenový strom.

Uzly v_i mají různé stupně.

Vlastnosti derivačního stromu $\mathcal{T}(\mathcal{G})$:

1. Kořenem $\mathcal{T}(\mathcal{G})$ úvodní neterminál S: $v_0 = S$.
2. Každý uzel v_i představován prvkem z množiny $N \cup T$.
3. Pokud má vrchol v_i s ohodnocením $\alpha \in N \cup T$ potomky $\alpha_1, \alpha_2, ..., \alpha_k$, existuje derivační pravidlo v P.

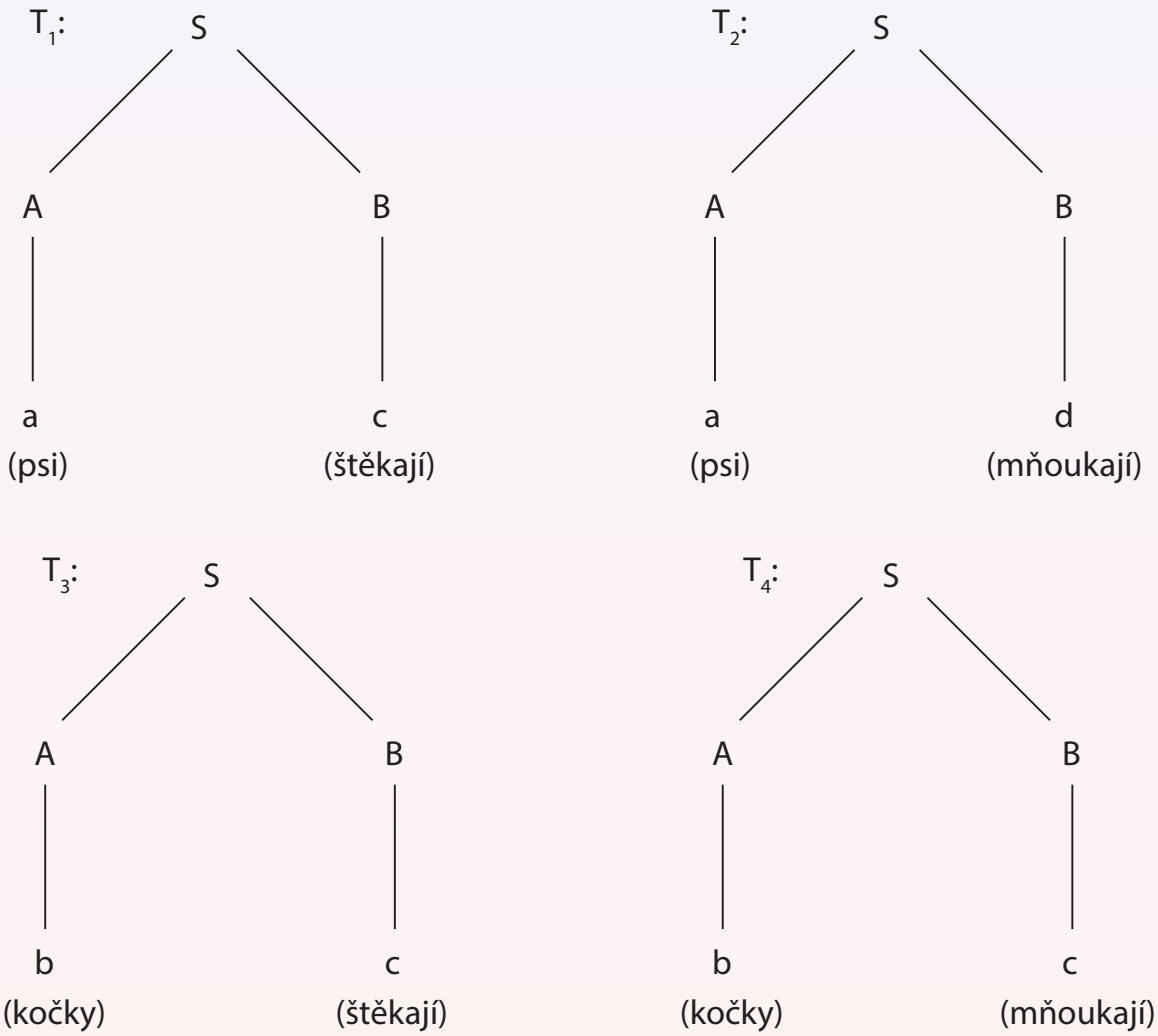
$$\alpha \rightarrow \alpha_1 \alpha_2 .... \alpha_k.$$

4. Listy $\mathcal{T}(\mathcal{G})$ jsou tvořeny terminály T (slovy) nebo ϵ.

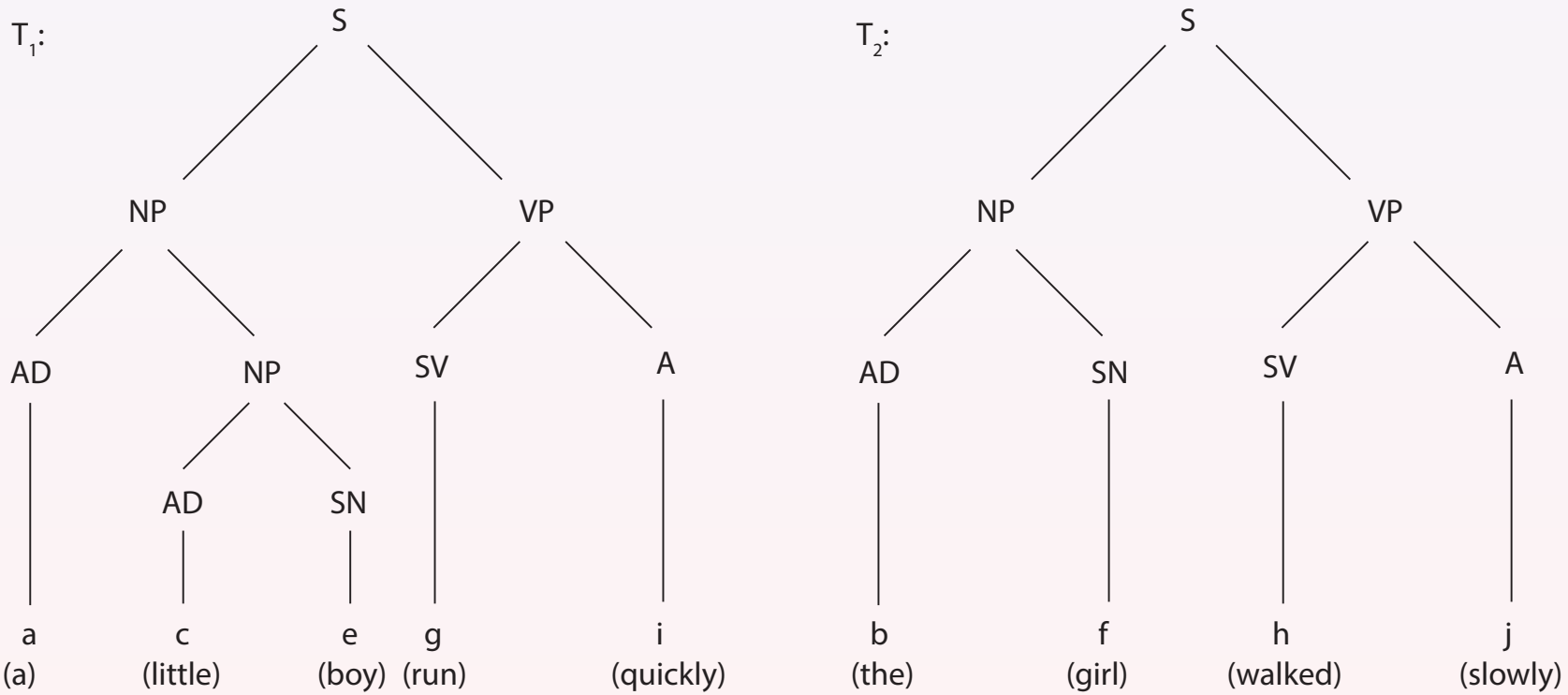
Věta w tvořena listy $T(\mathcal{G})$ čtenými z leva do prava.

Každému odvození věty w v $\mathcal{G}$ odpovídá jeden strom $T(\mathcal{G})$.

13. Derivační strom, Příklad 1



14. Derivační strom, Příklad 2



15. Druhy derivací

Pro $\mathcal{G} = (N, T, S, P)$ a $\alpha, \beta \in N \cup T$ obecná derivace

$$\alpha \Rightarrow \beta.$$

Levá derivace:

Pokud existují $u \in T$ a $\rho \in N \cup T$ a pravidlo $P : A \to \gamma$, $A \in N$,takové

$$\alpha = uA\rho \text{ a } \beta = u\gamma\rho,$$

pak α lze přepsat na β levým přepsáním (derivací)

$$\alpha \underset{L}{\Rightarrow} \beta.$$

Pravá derivace:

Pokud existují $v \in T$ a $\lambda \in N \cup T$ a pravidlo $P : A \to \gamma$, $A \in N$,takové

$$\alpha = \lambda Av \text{ a } \beta = \lambda\gamma v,$$

pak α lze přepsat na β pravým přepsáním (derivací)

$$\alpha \underset{R}{\Rightarrow} \beta.$$

Při *levé derivaci* nahrazujeme v každém kroku nejlevější neterminál.

Při *pravé derivaci* nahrazujeme v každém kroku nejpravější neterminál.

Derivovat můžeme i jejich kombinacemi: smíšené derivace.

Pokud $\exists$ více než jedno levé nebo více než jedno pravé odvození: $\mathcal{G}$ **nejednoznačná**.

16. Ukázka L a P derivace (1)

Gramatika $\mathcal{G} = (N, T, S, P)$, kde $N = \{AB\}$, $T = \{a, b, c\}$, $P = \{P_1, P_2, P_3\}$, kde

$$S \to AB,$$
$$A \to aAb|ab,$$
$$B \to cB|c,$$

odvození slova

$$w = aabbcc.$$

Levá derivace, nahrazujeme nejlevější neterminál

$$S \underset{L}{\Rightarrow} AB \underset{L}{\Rightarrow} aAbB \underset{L}{\Rightarrow} aabbB \underset{L}{\Rightarrow} aabbcB \underset{L}{\Rightarrow} aabbcc.$$

Pravá derivace, nahrazujeme nejpravější neterminál

$$S \underset{R}{\Rightarrow} AB \underset{R}{\Rightarrow} AcB \underset{R}{\Rightarrow} Acc \underset{R}{\Rightarrow} aAbcc \underset{R}{\Rightarrow} aabbcc.$$

Obecná derivace, střídáme nejlevější a nejpravější neterminál

$$S \underset{L}{\Rightarrow} AB \underset{L}{\Rightarrow} aAbB \underset{R}{\Rightarrow} aAbcB \underset{L}{\Rightarrow} aabbcB \underset{R}{\Rightarrow} aabbcc.$$

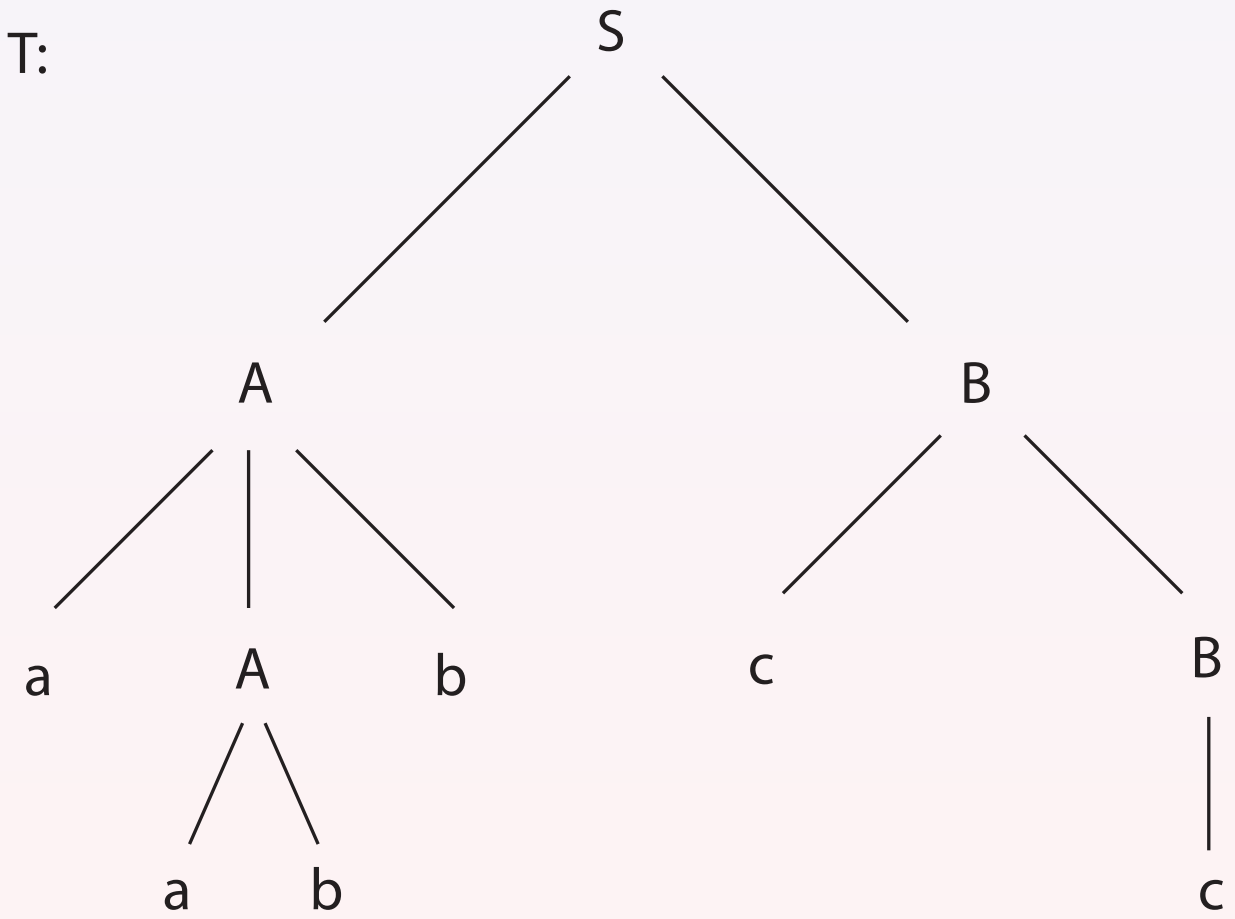
Existuje 1 levá a 1 pravá derivace, $\mathcal{G}$ jednoznačná, 1 derivační strom.

Poznámka:

Každé pravidlo lze použít vícekrát.

Terminály / neterminály lze aplikovat v libovolném pořadí.

17. Derivační strom: L derivace



18. Ukázka L a P derivace (2)

Gramatika $\mathcal{G} = (N, T, S, P)$, kde $N = \{S, +, *\}$, $T = \{a, b, c\}$, $P = \{P_1\}$, kde

$$S \to S + S | S * S | a | b | c.$$

Odvození slova

$$w = a + b * c.$$

L derivace, existují 2 různé

$$S \underset{L}{\Rightarrow} S + S \underset{L}{\Rightarrow} a + S \underset{L}{\Rightarrow} a + S * S \underset{L}{\Rightarrow} a + b * S \underset{L}{\Rightarrow} a + b * c,$$
$$S \underset{L}{\Rightarrow} S * S \underset{L}{\Rightarrow} S + S * S \underset{L}{\Rightarrow} a + S * S \underset{L}{\Rightarrow} a + b * S \underset{L}{\Rightarrow} a + b * c.$$

P derivace, existují 2 různé

$$S \underset{R}{\Rightarrow} S * S \underset{R}{\Rightarrow} S * c \underset{R}{\Rightarrow} S + S * c \underset{R}{\Rightarrow} a + S * c \underset{R}{\Rightarrow} a + b * c,$$
$$S \underset{R}{\Rightarrow} S + S \underset{R}{\Rightarrow} S + S * S \underset{R}{\Rightarrow} S + S * c \underset{R}{\Rightarrow} S + b * c \underset{R}{\Rightarrow} a + b * c.$$

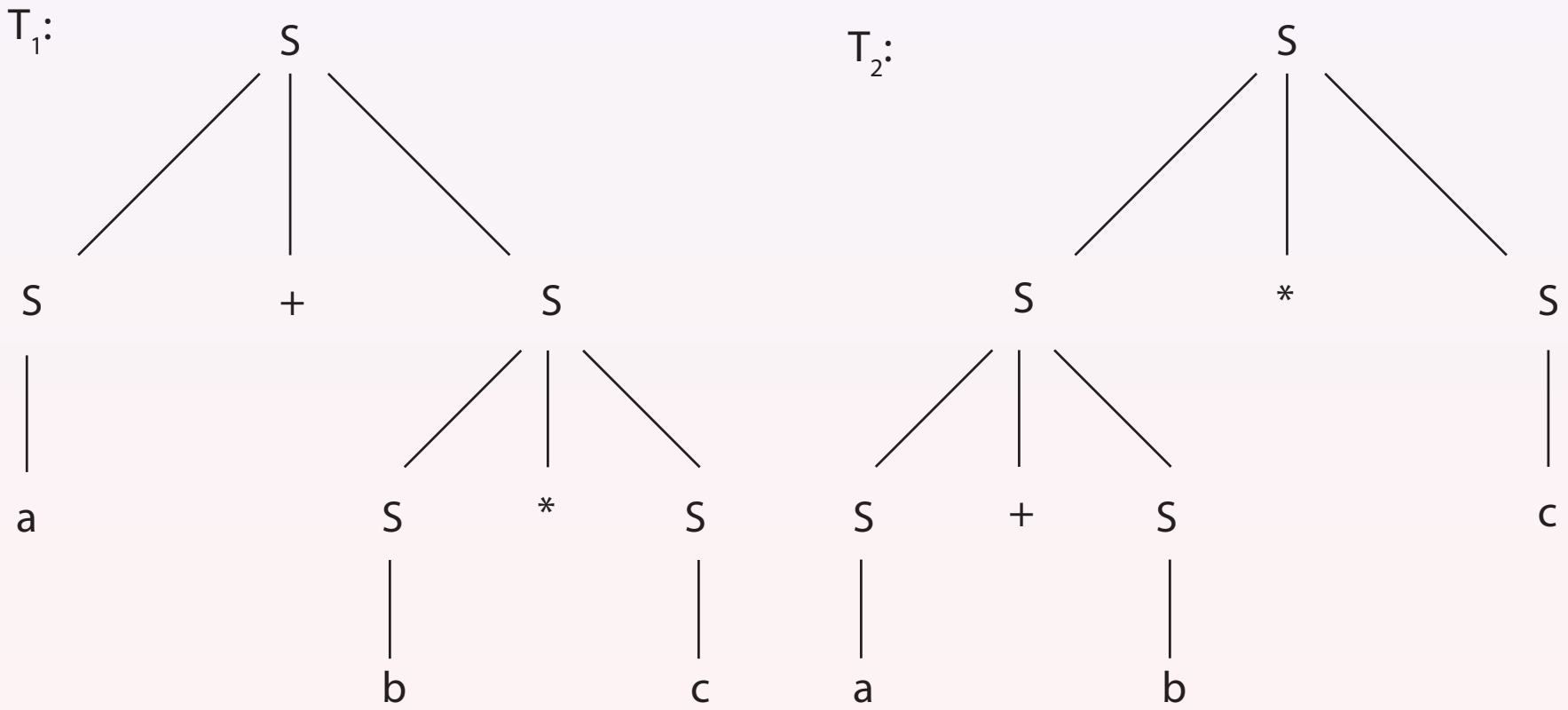
Obecná derivace, střídáme nejlevější a nejpravější terminál

$$S \underset{L}{\Rightarrow} S + S \underset{R}{\Rightarrow} S + S * S \underset{L}{\Rightarrow} a + S * S \underset{R}{\Rightarrow} a + S * c \underset{L}{\Rightarrow} a + b * c.$$

Dvě různé L derivace + dvě různé P derivace, $\mathcal{G}$ **není jednoznačná**!
Oběma L derivacím odpovídají *různé* derivační stromy.

U programovacích jazyků víceznačnost nepřípustná!

19. Derivační strom: obě L derivace



20. Návrh gramatiky: aritmetický výraz, v1

Výpočet hodnoty výrazu:

$$w = a + b * c, \quad a = 1, b = 3, c = 5.$$

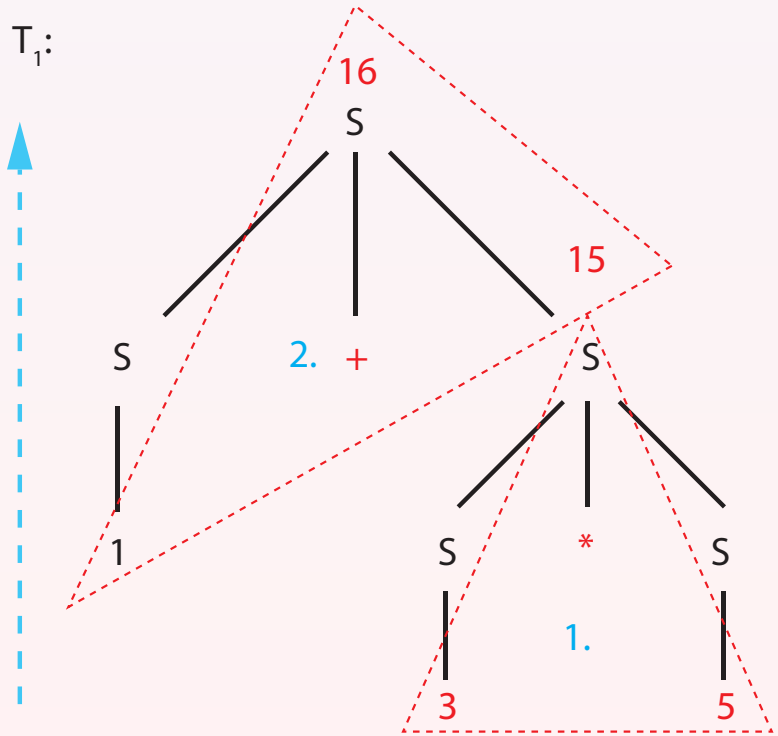
Použití modifikované gramatiky z předcházejícího příklad: $\mathcal{G} = (N, T, S, P)$, kde $N = \{S, +, *\}$, $T = \{1, 3, 5\}$, $P = \{P_1\}$, kde

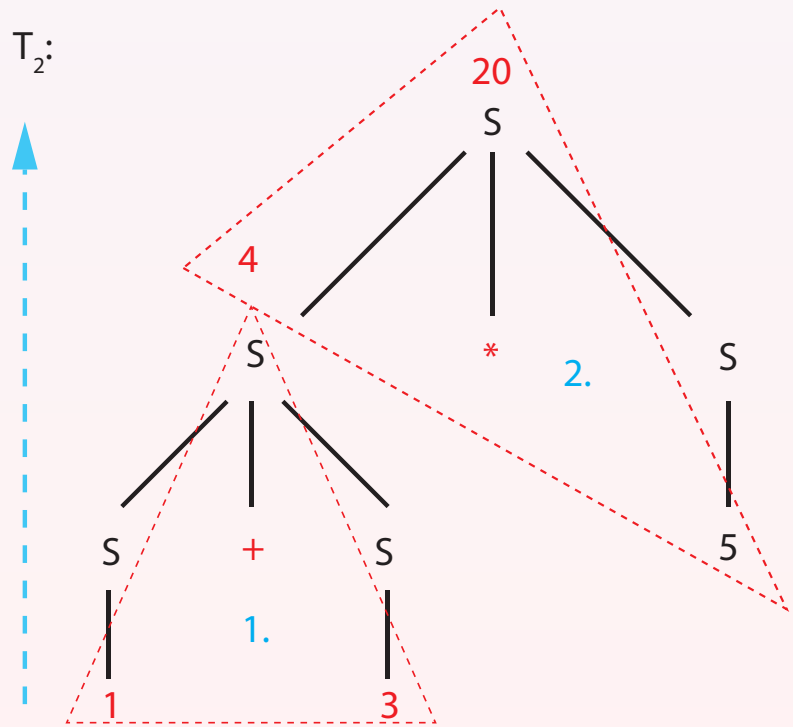
$$S \to S + S | S * S | 1 | 3 | 5.$$

Strom vyhodnocujeme od listů, problémem nejednoznačnost výpočtu x:

$$x = 1 + (3 * 5), \text{ nebo } x = (1 + 3) * 5.$$

Takovou gramatiku nelze použít (chybí priorita operací), které vyhodnocení je správné? (Levé)





21. Podrobnější pohled na výraz

Výraz $\approx$ Expression.

Požadavek: podpora artimetických operací: +, -, *, /.

Různá priorita operací: (*, /) > (+, -): čím vyšší priorita, tím později derivujeme!

Dodatečná změna priority: ().

Number (N):

Číslo, atomická struktura.

Factor (F):

Number nebo *Expression* ohraničený závorkami.

Nejvyšší priorita.

Term (T):

Posloupnost *Factorů* oddělená *, / nebo *Factor*.

Nižší priorita než *Factor*.

Expression (E):

Posloupnost *Termů* oddělená +, - nebo Term.

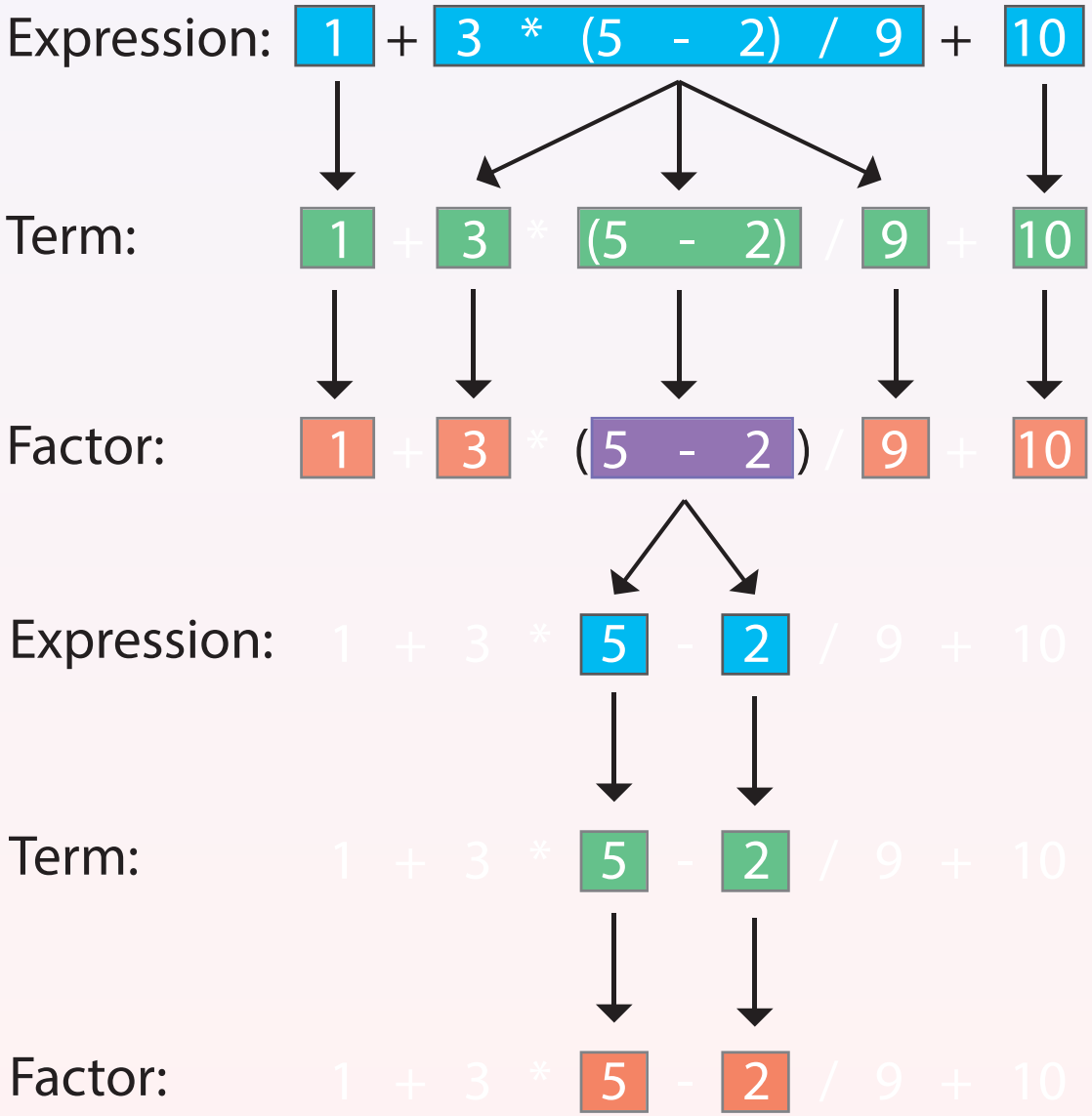
Nejnižší priorita.

Expression složen z *Termů*, *Term* z *Factorů*.

Factor složen z *Numberu* nebo *Expressionu*.

Hierarchická struktura: $E \rightarrow T \rightarrow F \rightarrow E|N$.

22. Ukázka dekompozice výrazu



23. Návrh gramatiky: aritmetický výraz, v2

Operátory + * s rozdílnou prioritou.

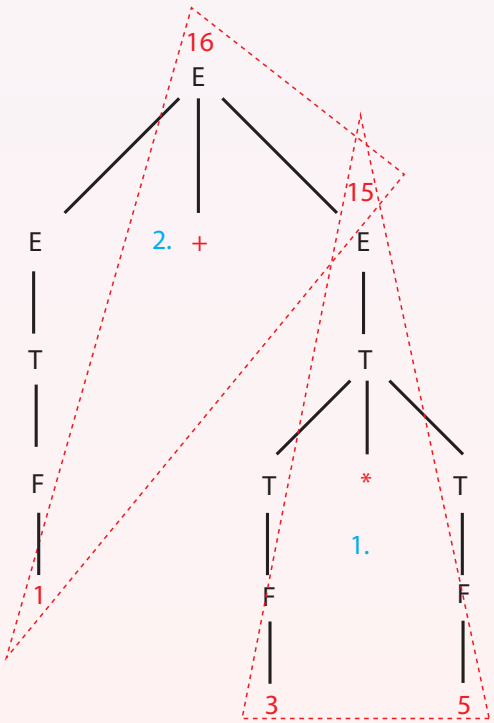
Výpočet hodnoty výrazu:

$$w = 1 + 3 * 5.$$

Modifikovaná gramatika dodržující prioritu

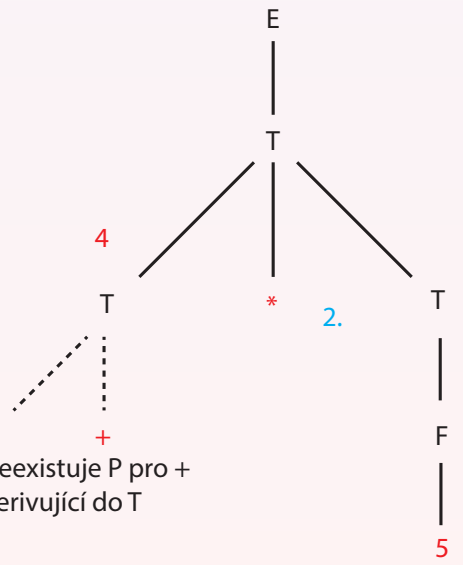
$$E \to E + E | T$$
$$T \to T * T | F,$$
$$F \to a | b | c | (E).$$

T_1: 1 + 3 * 5



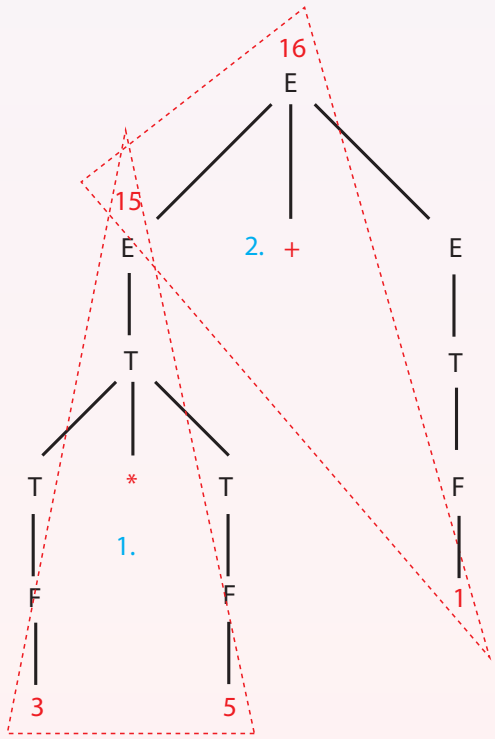
T_2: 1 + 3 * 5

Neexistuje



Neexistuje P pro + derivující do T

T_3: 3 * 5+1



Gramatika jednoznačná, podporuje prioritu operací. Je OK?

24. Návrh gramatiky: aritmetický výraz, v3

Přidáme operátor -: +, -, *

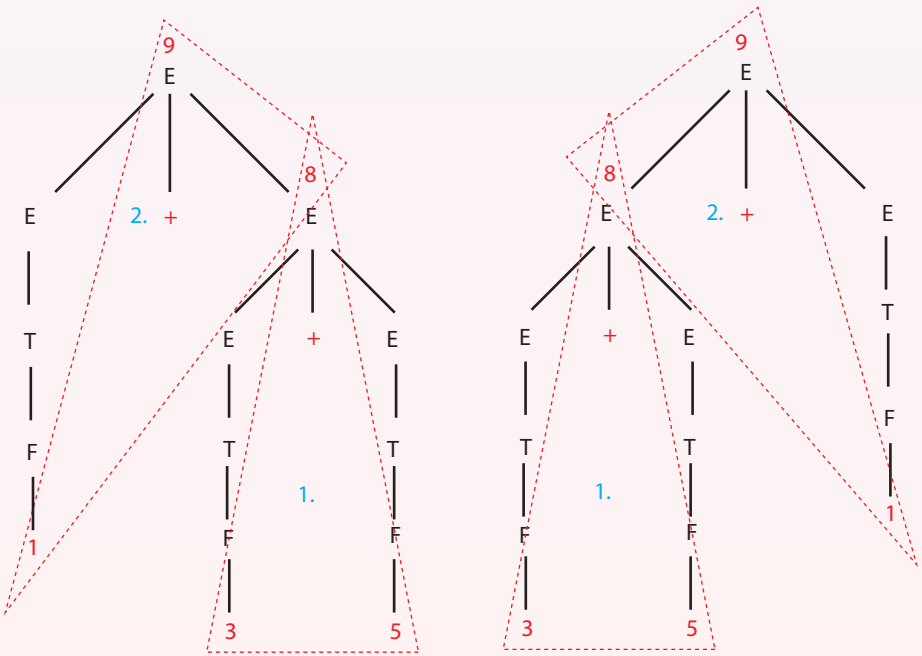
Výpočet hodnoty výrazu, operátory se stejnou prioritou:

$$w = 1 + 3 + 5,$$
$$w = 1 - 3 - 5.$$

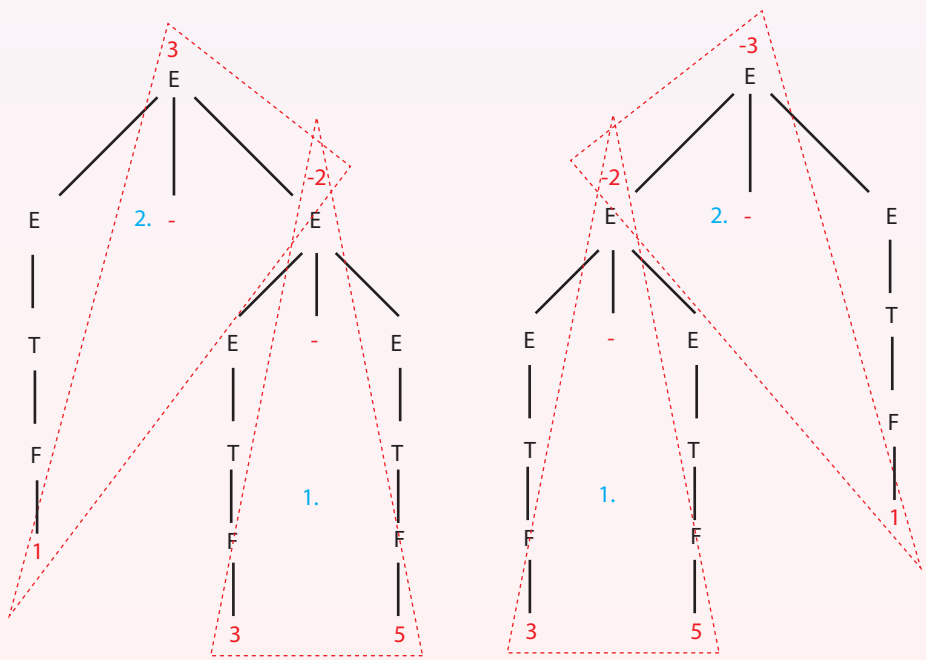
Modifikovaná gramatika dodržuje prioritu, co L/P-asociativní operace?

$$E \to E + E | E - E | T,$$
$$T \to T * T | F,$$
$$F \to a | b | c | (E).$$

P-asociativita: 1 + 3 + 5 = 1 + (3 + 5) = (3 + 5) + 1 → OK.



P-asociativita: 1 - 3 - 5 ≠ 1 - (3 -5) ≠ (3 - 5) - 1 → KO.



Gramatika podporuje P-asociativní operace: + *

Gramatika nepodporuje L-asociativní operace: - /

25. Rekurzivita gramatiky

$\mathcal{G} = (N, T, S, P)$, je *rekurzivní*, pokud pro nějaké $A \in N$, $\alpha, \beta \in N \cup T$ existuje derivace

$$A \Rightarrow^* \alpha A \beta.$$

Pokud $\alpha = \epsilon$, pak $\mathcal{G}$ je *rekurzivní zleva*: A derivuje sekvenci začínající A

$$A \Rightarrow^* A\beta.$$

Pokud $\beta = \epsilon$, pak $\mathcal{G}$ je *rekurzivní zprava*: A derivuje sekvenci končící A

$$A \Rightarrow^* \alpha A.$$

Rekurzivní gramatika je vždy **víceznačná**.
Problém při konstrukci syntaktického analyzátoru.
Snaha zbavovat se pravidel rekuzivních zleva i zprava

$$A \Rightarrow^* A \oplus A,$$

nejsou L-asociativní. Dále se budeme zbavovat i rekurze.

L/P-asociativita:

$$a \oplus b \oplus c = (a \oplus b) \oplus c,$$
$$a \oplus b \oplus c = a \oplus (b + c).$$

Operace -, / jsou L-asociativní (P-asociativitu již umíme)

$$1 - 3 - 5 = -7 = (1 - 3) - 5 \neq 1 - (3 - 5) = 3,$$
$$1/3/5 = 1/15 = (1/3)/5 \neq 1/(3/5) = 5/3.$$

26. Návrh gramatiky: aritmetický výraz, v4

Podpora operátorů: +, -, *, /.

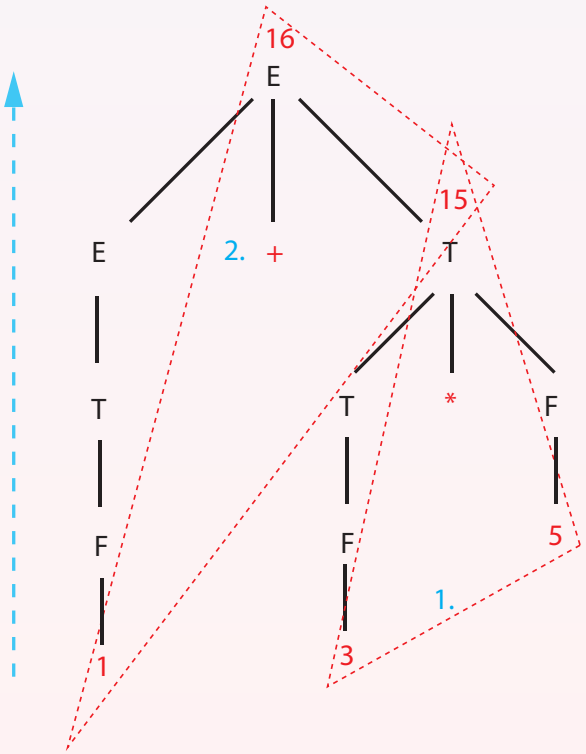
Návrh gramatiky $\mathcal{G} = (N, T, S, P)$,

$$E \to E + T | E - T | T,$$
$$T \to T * F | T / F | F,$$
$$F \to a | b | c | (E).$$

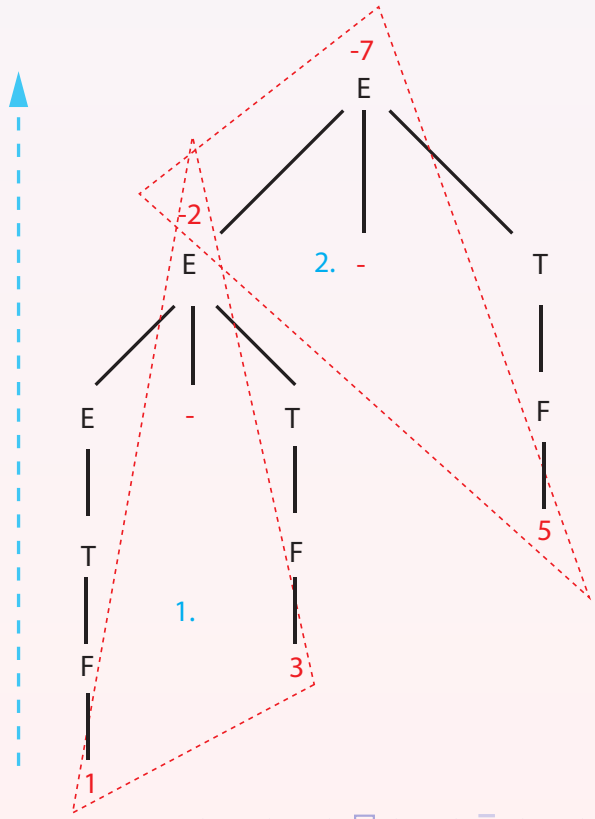
Ukázka L-asociativity: $E \Rightarrow E - T \Rightarrow (E - T) - T \Rightarrow ([E - T] - T) - T$.

Podpora L-asociativity (-, /), P-asociativity (+, *), priority operátorů.

Priorita: 1 + 3 * 5 = 1 + (3 * 5) = 16 → OK.



L-asociativita: 1 - 3 - 5 = (1 - 3) - 5 = -7 → OK.



27. Návrh gramatiky, aritmetický výraz, v5

Úprava výše odvozené gramatiky o nové pravidlo, mocnina

```
a^b = a**b = pow(a,b) = a pow b.
```

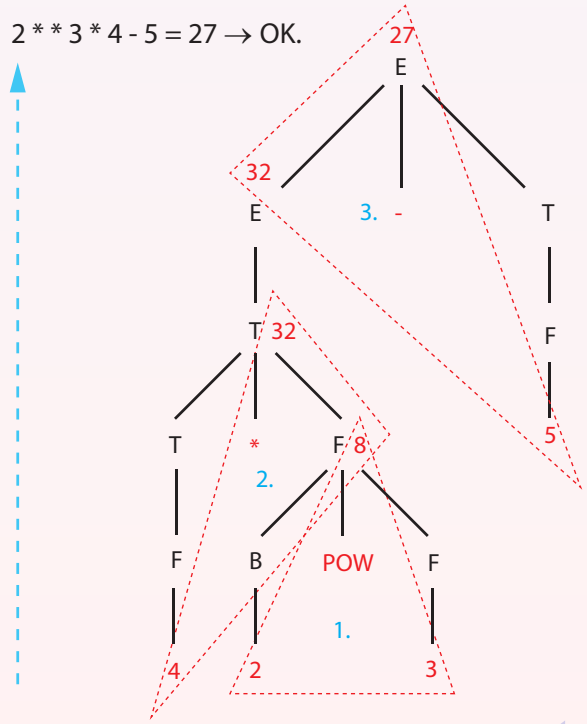
Tuto syntaxi používá Python.

Příklady:

```
2 ** 3 = 8,     2 ** 3 - 4 = 4,     2 ** 3 * 4 - 5 = 27,
```

Operátor `pow` má nejvyšší prioritu, derivační pravidlo co nejpozději: umístěno za *, /; argument může být složitější výraz (součin):

$$
\begin{aligned}
E &\to E + T | E - T | T, \\
T &\to T * F | T / F | F, \\
F &\to B \ POW \ F | B \\
B &\to a | b | c | (E).
\end{aligned}
$$



28. Překlad výrazu

Na vstupu nějaké množina slov w z gramatiky $\mathcal{G} = (N, T, S, P)$, (např. program v Pythonu).
Chceme zjistit, zda odpovídají $\mathcal{G}$, a zpracovat je.

Zjednodušený příklad:

```
w = 1 + 3 * 5 - 7
```

Jak spočítat hodnotu w vzhledem ke $\mathcal{G}$? $\Rightarrow$ překlad, vyhodnocení.

Tvořeno dvěma fázemi:

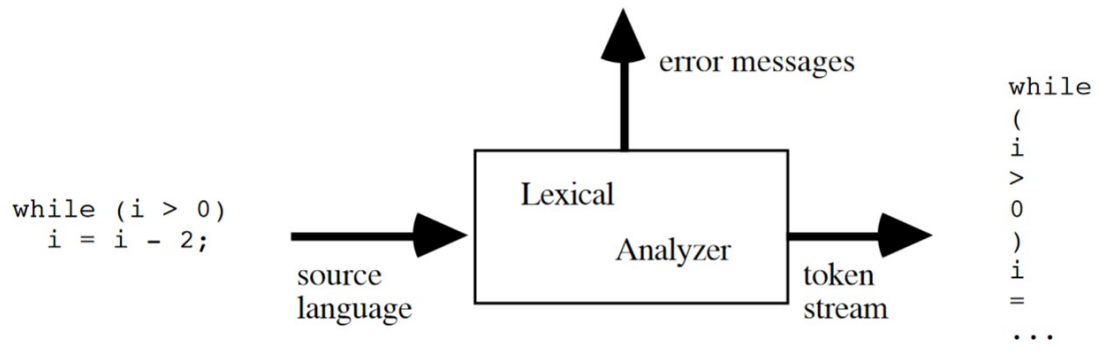
1. *Lexikografická analýza*
   Načtení w, konverze na lexikální symboly, tzv. *tokeny*.
   Tokeny atomické, generovány vzhledem k $\mathcal{G}$.
   Rozpoznání terminálů, netermiálů: čísla, klíčová slova, operátory.
2. *Syntaktická analýza*
   "Zpracování" gramatiky $\mathcal{G}$ - konstrukce derivačního stromu pro $\mathcal{G}$.
   Aplikace derivačních pravidel $\mathcal{G}$ na w.
   Varianty konstrukce derivačního stromu: shora-dolů / zdola-nahoru.
   Výsledek: zda $w \in \mathcal{G}$?
3. *Sémantická analýza*
   Vyhodnocení w vzhledem k $\mathcal{G}$.
   Může být i součástí syntaktické analýzy.

29. Lexikografická analýza

Vstupní posloupnost znaků zpracována v L->P směru: rozpoznání N/T, ale neporozumění pravidlům $P \in \mathcal{G}$.

Cílem zjednodušení syntaktického analyzátoru: korektní vstup.

Popis vstupních symbolů prostřednictvím *regulární gramatiky*.



Výsledkem LA:

- Odstranění redundantních znaků: mezery, komentáře.
- Konverze posloupnosti na tokeny.
- Normalizace symbolů: lower case <-> upper case.
- Hlášení o případných chybách: např. nerozpoznání proměnné, špatné komentáře.

Token

Neprázdná množina znaků, reprezentuje *syntaktické kategorie*.

- Identifikátory: řetězec písmen a čísel začínající písmenem
- Čísla: řetězec číslic.
- Klíčová slova: pro daný jazyk, např. if, else, for, atd.
- White space: tabelátory, konce řádek.

30. Realizace lexikografické analýzy

Nutná definice vstupní množiny tokenů dle výše uvedených kritérií.

LA rozeznává znaky příslušející tokenu: sestavení tokenu ze znaků.

Řetězce nemající oporu v jazyku: chyba.

2 typy implementací LA:

- *Ad hoc metoda*
  Sekvenční čtení znaků po jednom ze vstupu.
  Při načtení aktuálního znaku se rozhoduje, jak s nezpracovaným řetězcem naložit.
  Nutné vidět vpřed: např * vs **.
  Výhodou rychlost, customizace, používají některé kompilátory (gcc).
  Nevýhoda: vlastní implementace rozpoznávání, pro složité jazyky obtížnější.
- *Regulární výrazy + konečné automaty*
  Lexikální symboly lze vyjádřit regulárními jazyky.
  Regulární jazyky zpracovávány konečnými automaty.
  Nutné definovat pouze tokeny + pravidla při sestavení tokenu: regulární výrazy.
  Méně programování, univerzalita.
  Ne všechny progr. jazyky podporují regulární výrazy.
  Implementace automatu složitá.

LA prováděna specializovanými nástroji, tzv. skenery.

31. Lexikografická metoda Ad hoc, aritmetický výraz

S využitím regulárních výrazů:

```
def tokenize(str_in):
    split = '([+-/*])'                  #Splitting rule
    str_in.replace(' ','')             #Remove whitespaces
    tokens = re.split(split, str_in)   #Split input string by tokens
    return tokens
```

Bez použití regulárních výrazů:

```
def tokenize(str_in):
    tokens = [""]                                   #Empty tokens
    str_in.replace(' ','')                         #Remove whitespaces
    for c in str_in:
        if c.isdigit() and tokens[-1].isdigit():  #Current char is digit
            tokens[-1] = tokens[-1] + c           #Add digit to the last one
        else:
            tokens.append(c)                      #Add char to the tokens
    return tokens
```

Volání:

```
print(tokenize('((12 * 5) / 6 + (4-2))')
>>['', '(', '(', '12', ' ', '*', ' ', '5', ')', ' ', '/', ' ', '6', ' ', '+', ' ', '(', '4', '-', '2', ')
```

32. Syntaktická analýza

Založena na CFG.

Rekombinace tokenů získaných lexikální analýzou k vybudování derivačního stromu.

Provádí se pro konkrétní gramatiku $\mathcal{G} = (N, T, S, P)$ a w.

Na rozdíl od lexiografické analýzy již rozumíme derivačním pravidlům.

Cíle syntaktické analýzy:

- Ověření, zda slovo w může být generováno gramatikou $\mathcal{G}$.
- Konstrukce derivačního stromu (ne vždy).
- Zpracování w v $\mathcal{G}$ (ne vždy, často tzv. sémantická analýza).
- Nalezení syntaktických chyb.

SA analýza prováděna specializovanými nástroji, tzv. parsery.

Techniky parsování:

- *Top-down parsing*
  Od kořene k listům, obvyklejší.
- *Bottom-up parsing*
  Od listům ke kořeni, méně častá.

33. Top-down vs. bottom-up parsing

Top-down parsing (T-D):

Vstupní gramatika $\mathcal{G}$ je v LL tvaru: bez cyklů a levé rekurze.

- Zpracování začíná od kořene a pokračuje směrem k listům: shora dolů.
- Derivujeme tak dlouho, dokud listy derivačního stromu neodpovídají w.
- V uzlu A derivujeme potomky podle konkrétního $P_i \in P$.
- Nutná jednoznačnost.

Bottom-up parsing (B-U):

Vstupní gramatika $\mathcal{G}$ je v LR tvaru: bez cyklů a pravé rekurze.

- Zpracování začíná od listu a pokračuje směrem ke kořeni: zdola nahoru.
- Používány pravé derivace, hledáme pravý rozklad.

2 přístupy:

- *Deterministické zpracování:*
  Dodatečné informace o řetězci: nutno znát k znaků vstupního řetězce dopředu.
  $LL(k)$ gramatiky pro T-D a $LR(k)$ gramatiky pro B-U.
  V praxi používány LL(1) a LR(1) gramatiky: nutno znát následující znak.
- *Backtracking:*
  Aplikujeme pravidlo, pokud uvázneme, jdeme stromem vzhůru a aplikujeme jiná pravidla.
  V praxi se nepoužívá, neefektivní (prodlužuje kompilační čas).
  Netřeba znát dodatečné informace o řetězci.

Dále se budeme zabývat pouze T-D parsingem (deterministické řešení).

34. Levá a pravá derivace, připomenutí

Gramatika $\mathcal{G} = (N, T, S, P)$, kde $\mathcal{G} = (\{E, T, F\}, \{a, b, c, +, -*, /\}, P = \{P_1, ..., P_{10}\},)$

$$\begin{aligned} E &\to E + T | E - T | T, & 1|2|3, \\ T &\to T * F | T / F | F, & 4|5|6, \\ F &\to (E) | a | bc |, & 7|8|9|10. \end{aligned}$$

Odvození $w = (a + b) * c$.

Levá derivace, shora dolů, nejlevější $T \to T \cup N$

$$\begin{aligned} E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow (E) * F \Rightarrow (E + T) * F, \Rightarrow (T + T) * F \\ \Rightarrow (F + T) * F \Rightarrow (F + F) * F \Rightarrow (a + F) * F \Rightarrow (a + b) * F \Rightarrow (a + b) * c, \end{aligned}$$

použitá pravidla: 3, 4, 6, 7, 1, 3, 6, 6, 8, 9, 10.

Pravá derivace, shora dolů, nejpravější $T \to T \cup N$

$$\begin{aligned} E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow F * c \Rightarrow (E) * c \Rightarrow (E + T) * c, \Rightarrow (T + T) * c \\ \Rightarrow (T + F) * c \Rightarrow (F + F) * c \Rightarrow (F + b) * c \Rightarrow (a + b) * c, \end{aligned}$$

použitá pravidla: 3, 4, 6, 10, 7, 1, 3, 6, 6, 9, 7.

Pravá derivace, zdola nahoru, nejpravější $T \cup N \to T$

$$\begin{aligned} w \Rightarrow (a + b) * c \Rightarrow (F + b) * c \Rightarrow (F + F) * c \Rightarrow (T + T) * c, \\ \Rightarrow (E + T) * c \Rightarrow (E) * c \Rightarrow F * c \Rightarrow F * F \Rightarrow T * F \Rightarrow T \Rightarrow E. \end{aligned}$$

Pravidla aplikována v **opačném** směru: 7, 9, 6, 6, 3, 1, 7, 10, 6, 4, 3.

35. LL(1) gramatika

Prohledává vstup ve směru z Leva do prava.

Generuje Levé odvození.

Vidí o 1 token vpřed: které z alternativních pravidel použijeme.

Zpracovávána T-D parserem.

Vlastnosti $LL(1)$ gramatiky $\mathcal{G} = (N, T, S, P)$:

- $\nexists$ *levé rekurze*
  $\mathcal{G}$ nesmí obsahovat levou rekurzi.
  Odstranění substitučním pravidlem.
- $\nexists$ *prefixu*
  Prefix, první znak zleva, např. závorka.
  2 pravidla se společnou levou stranou v $\mathcal{G}$ nemohou mít stejný prefix.
  Odstranění levou faktorizací.

Nejčastěji používaná gramatika při návrhu parserů.

Využívána i u velkých řešení: kompilátory jazyků.

Před zpracováním T-D parserem nutno ověřit, zda $\mathcal{G}$ je $LL(1)$.

Pokud není, hrozí uváznutí (nekonečná rekurze).

36. Odstranění levé rekurze

Gramatika $\mathcal{G} = (N, T, S, P)$ popsaná pravidly

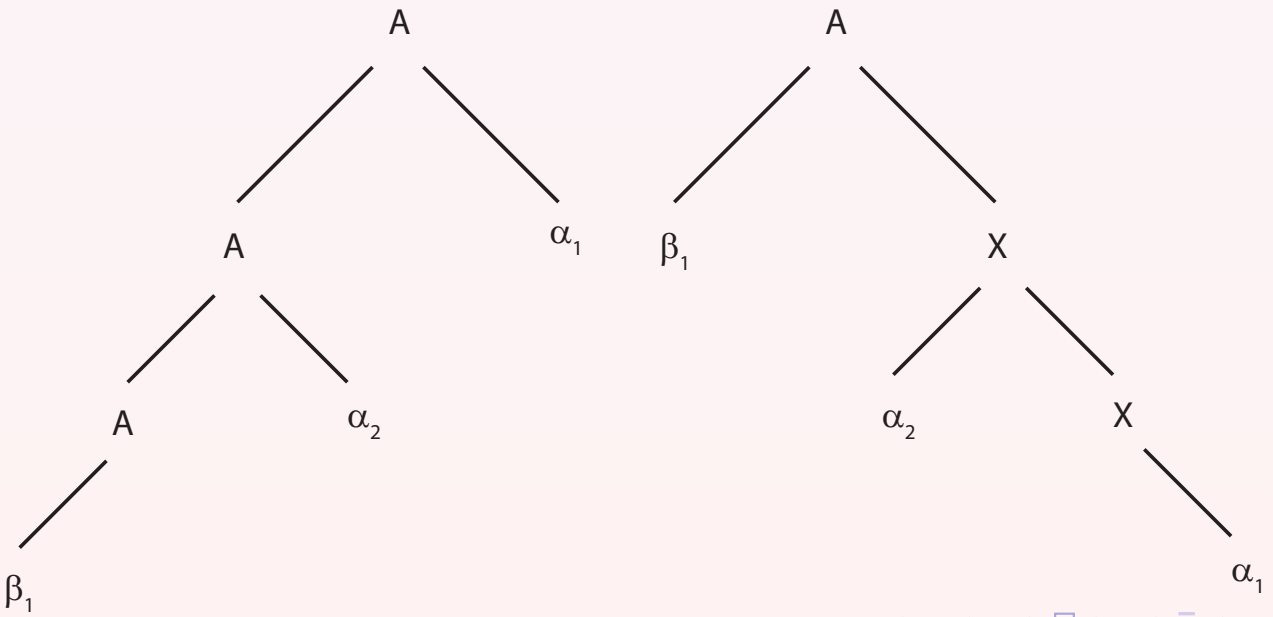
$$A \to A\alpha_1|A\alpha_2|...|A\alpha_m|\beta_1|\beta_2|...|\beta_m,$$

kde $A \in N$ a $\alpha \in T \cup N$, obsahuje levou rekurzi. $\Rightarrow$ Odstranění substitucí.
Ekvivalentní gramatika $\mathcal{G}' = (N \cup X, T, S, P')$ má tvar

$$A \to \beta_1 X|\beta_2 X|...|\beta_m X,$$
$$X \to \alpha_1 X|\alpha_2 X|...|\alpha_m X|\epsilon.$$

Obě gramatiky generují stejné stromy:

$$\mathcal{G} : A \to A\alpha_1|A\alpha_2|\beta_1|\beta_2|,$$
$$\mathcal{G}' : A \to \beta_1 X|\beta_2 X|, \alpha_1 X|\alpha_2 X.$$



37. Ukázka odstranění levé rekurze

Gramatika $\mathcal{G}$ obsahuje levou rekurzi

$$E \to E + T | E - T | T,$$
$$T \to T * F | T / F | F,$$
$$F \to a | bc | (E).$$

Pravidlo

$$E \to E + T | E - T | T,$$

obsahuje levou rekurzi, kde

$$\alpha_1 = +T, \alpha_2 = -T, \beta_1 = T, E' = E.$$

Pak

$$E \to TE',$$
$$E' \to +TE' | -TE' | \epsilon.$$

Pravidlo

$$T \to T * F | T / F | F,$$

obsahuje levou rekurzi

$$\alpha_1 = *F, \alpha_2 = /F, \beta_1 = F, T' = T.$$

Pak

$$T \to FT',$$
$$T' \to *FT' | /FT' | \epsilon.$$

38. Odstranění prefixu

Gramatika $\mathcal{G} = (N, T, S, P)$ obsahuje více pravidel se společnou levou stranou a prefixem α

$$A \to \alpha\beta_1|\alpha\beta_2|...,\alpha\beta_n$$

kde $\alpha, \beta \in N \cup T$.

Levá faktorizace vytvoří ekvivalentní gramatiku $\mathcal{G}'$

$$A \to \alpha X,$$
$$X \to \beta_1|\beta_2|...,\beta_n.$$

Jinak řečeno, vytkneme α.

Při znalosti jednoho tokenu vpřed nelze rozhodnout, které pravidlo aplikovat.

Příklad: řešení levé faktorizace v $\mathcal{G} = (N, T, S, P)$

$$E \to (E)|().$$

Protože

$$\alpha = (, \beta_1 = E), \beta_2 =)$$

pravidlo převedeme na tvar

$$E \to (X,$$
$$X \to E)|).$$

39. Prediktivní parser, princip

Předpokládáme, že $\mathcal{G}$ je LL(1), tj. bez prefixu a zleva nerekurzivní.

Prediktivní parser (PP) buduje strom shora dolů.

2 varianty PP:

- *Parser řízený tabulkou*
  Řídí se pravidly přechodové tabulky, množiny FIRST a FOLLOW.
  Univerzální, zpracuje libovolnou CFG, zásobníkový automat.
- *Rekurzivně sestupný parser.*
  Zásobník nahrazen rekurzivním voláním.
  Méně univerzální, nutno implementovat pro každou CFG.

Parser řízený tabulkou potřebuje 2 charakteristické množiny:

- $FIRST(\alpha)$
  Množina prvních terminálů derivačních pravidel.
  Pro jednu levou stranu existuje více pravých stran, jak pokračovat?
- $FOLLOW(A)$
  Množina terminálů, které jsou následníkem neterminálu.
  Důsledek použití ϵ, které je přiřazeno nějakému neterminálu.
  Aneb jak poté pokračovat?

Slouží ke konstrukcí *přechodové tabulky* ověřující, zda $\mathcal{G}$ je LL(1).

Pokud není, překlad zastaven.

40. Množina $FIRST(\alpha)$

U pravidel se společnou levou stranou, více pravých stran.
V takových případech není jasné, jak pokračovat dále.

$FIRST(\alpha)$ je množinou úvodních terminálů.

Příklad: Slovo $w = cad$, zpracováváme od D, použijeme E nebo F?

$$S \to cAd,$$
$$A \to a|bc,$$

Při zpracování a dvě pravidla pro A. Vybereme to, které začíná na a.

Výsledkem této operace je vždy terminál.

$$FIRST(\alpha) = \begin{cases} \epsilon, & \alpha \to \epsilon, \\ a, & \alpha \to a\beta, \\ FIRST(A), & \alpha \to A\beta, \epsilon \notin FIRST(A), \\ (FIRST(A) - \{\epsilon\}) \vee FIRST(\beta), & \alpha \to A\beta, \epsilon \in FIRST(A). \end{cases}$$

Pro $FIRST(A)$ za A dosazujeme z derivačních pravidel.

Při tvorbě přechodové tabulky počítán pouze pro neterminály.

41. Ukázka výpočtu $FIRST(\alpha)$

Gramatika $\mathcal{G} = (\{E, T, F\}, \{x, +, -*, /, (,)\}, P)$ dána pravidly

$$
\begin{aligned}
E &\to TE', && 1\\
E' &\to +TE' | -TE' | \epsilon. && 2|3|4\\
T &\to FT', && 5\\
T' &\to *FT' | /FT' | \epsilon. && 6|7|8\\
F &\to (E) | x. && 9|10
\end{aligned}
$$

Pak, množiny $FIRST(\alpha)$

$$
\begin{aligned}
FIRST(E') &= \{+, -, \epsilon\},\\
FIRST(T') &= \{*, /, \epsilon\},\\
FIRST(F) &= \{(, x\},\\
FIRST(T) &= FIRST(F) = \{(, x\},\\
FIRST(E) &= FIRST(T) = \{(, x\}.
\end{aligned}
$$

42. Množina $FOLLOW(\alpha)$

Množiná následujících terminálů.

Důsledek použití operace ϵ, která smaže nejlevější neterminál.
Jak dále pokračovat při levé derivaci?
Nestačí znalost aktuálního neterminálu, nutno znát následující, proto $LL(1)$.

Příklad: Slovo $w = ab$, odvodíme v gramatice

$$A \to aBb,$$
$$B \to \epsilon | c.$$

Aplikace $A \to aBb$, shoda na 1. znaku, a.
Hledáme pravidlo generující shodu v b, neexistuje derivace B generující b. Chyba gramatiky???
Existuje $B \to \epsilon$, to však lze použít jedině při znalosti následujícího znaku b.
Použití $FOLLOW(B)$ generující b.

$$FOLLOW(A) = \begin{cases} \$, & A = S, \\ FIRST(\beta), & B \to \alpha A \beta, \\ FOLLOW(B), & B \to \alpha A | \alpha A \beta, \beta \Rightarrow \epsilon. \end{cases}$$

Poslední 2 kroky opakovány, dokud se změní alespoň jedna z množin: aplikace pravidel.pravidlo

43. Ukázka výpočtu $FOLLOW(A)$

Gramatika $\mathcal{G} = (\{E, T, F\}, \{x, +, -*, /, (,)\}, P)$ dána pravidly

$$\begin{aligned} E &\to TE', && 1 \\ E' &\to +TE' | -TE' | \epsilon. && 2|3|4 \\ T &\to FT', && 5 \\ T' &\to *FT' | /FT' | \epsilon. && 6|7|8 \\ F &\to (E) | x. && 9|10 \end{aligned}$$

Pravidla (4), (8) pro $E' = \epsilon$ a $T' = \epsilon$ generují

$$E' \to +T | -T, \quad T' \to *F | /F$$

Pak, množiny $FOLLOW(A)$:

$$\begin{aligned} FOLLOW(E) &= \{\$,)\}, \text{nepotřebujeme,} \\ FOLLOW(E') &= FOLLOW(E) = \{\$,)\}, \\ FOLLOW(T) &= FIRST(E') = \{+, -,), \$\}, \text{nepotřebujeme,} \\ FOLLOW(T') &= FOLLOW(T) = \{+, -,), \$\}, \\ FOLLOW(F) &= FIRST(T') = \{*, /, +, -,), \$\}, \text{nepotřebujeme.} \end{aligned}$$

Pokud v řádku $FIRST(A)$ nefiguruje ϵ, nepočítáme $FOLLOW(A)$.

44. Přechodová tabulka

Zachycuje vztahy mezi všemi terminály a neterminály v gramatice.

Řádky tvoří N, sloupce T, buňky pravidla derivující T.

Slouží k ověření, zda je gramatika $LL(1)$.

Prediktivní parser řízen přechodovou tabulkou.

Algoritmus tvorby přechodové tabulky:

```
for ∀ P_i : A → α ∈ P:                  #Pro kazde pravidlo
    for ∀a ∈ FIRST(α):                  #Pro vsechny term. a gener. FIRST
        insert A → α to T[A, a]         #Pridej pravidlo do radku A,sloupce a
    if ϵ ∈ FIRST(α):                    #Pokud FIRST generuje ϵ
        for ∀b ∈ FOLLOW(A) :            #Pro vsechny term. a gener. FOLLOW
            insert A →
α to T[A, a] #Pridej pravidlo do radku A,sloupce a
```

Poznámka: $FOLLOW(X)$ počítáme, pokud $FIRST(X)$ generuje ϵ.

Místo pravidel do buněk zapisujeme jejich pořadové číslo.

Pokud buňka obsahuje více derivačních pravidel, gramatika nejednoznačná.

45. Ukázka přechodové tabulky

$E \to TE'$	1	$FIRST(E') = \{+, -, \epsilon\}$
$E' \to +TE' \mid -TE' \mid \epsilon$	2|3|4	$FIRST(T') = \{*, /, \epsilon\}$,
$T \to FT'$	5	$FIRST(F) = \{(, i\}$,
$T' \to *FT' \mid /FT' \mid \epsilon.$	6|7|8	$FIRST(T) = FIRST(F) = \{(, i\}$,
$E' \to (E) \mid x$	9|10	$FIRST(E) = FIRST(T) = \{(, i\}.$

$FOLLOW(E') = FOLLOW(E) = \{\$,)\}$,
$FOLLOW(T') = FOLLOW(T) = \{+, -,), \$\}.$

$FIRST(X)$:

V řádku X vyplňujeme sloupce, jejichž terminály obsaženy ve $FIRST(X)$.

Číslo v buňce odpovídá pořadovému číslu derivačního pravidla.

$FOLLOW(X)$:

Všímáme si pouze $FIRST(X)$ generujících ϵ. V řádku X vyplníme sloupce jejichž terminály obsaženy ve $FOLLOW(X)$.

Číslo v buňce odpovídá pořadovému číslu pravidla derivujícího ϵ.

N/T	x	$+$	$-$	$*$	$/$	$($	$)$	\$
E	1					1		
E'		2	3				4	4
T	5					5		
T'		8	8	6	7		8	8
F	10					9		

46. Prediktivní parser řízený tabulkou

Zásobníkový automat, využívá přechodovou tabulku.

Vstup: w, $\mathcal{G}$ popsaná množinami $N, T, F/F, P$.

Výstup: $w \in \mathcal{G}$?

Z S vybíráme prvek, pokud je neterminál, do S přidáme derivační pravidlo.

Pokud je terminál, kontrolujeme, zda je roven $w[i]$.

Postup:

1. Inicializace zásobníku $S = \emptyset$, indexu $i \rightarrow 0$
2. Přidání 1. pravidla: $S \leftarrow P_1$.
3. Dokud S není prázdný, opakuj:
   1. Vyjmi prvek e z vrcholu zásobníku: $e \leftarrow \mathrm{pop}(S)$.
   2. Pokud $e \in N$, najdi v F pravidlo p v řádku e a sloupci $w[i]$, přidej ho do S: $S \leftarrow p$.
   3. Pokud $e \in T$, v případě $w[i] = e$ inkrementace $i \rightarrow i + 1$, jinak chyba.

Snadná implementace, lze použít pro libovolnou $\mathcal{G}$.

47. Prediktivní parser, implementace v jazyce Python

```
def parse(F, P, w):
    S = []                                  #Create empty S
    i = 0                                   #Token index
    S[0:0] = P[1]                           #Add first production rule
    while S:                                #Repeat until S is empty
        e = S.pop(0)                        #Get element on the top of S
        if e == 'eps':                      #e is eps, jump over
            continue
        if e in F:                          #e is nonterminal
            r = F.get(e)                    #Row of the production rule in F
            ip = r.get(w[i])                #Production rule in F
            if ip != None:                  #Production rule exists
                S[0:0] = P[ip]              #Add production rule to Se
            else:                           #w[i] not in N
                print('Error:' + w[i])      #Print exception and stop
                return
        else:                               #e is terminal
            if w[i] == e:                   #Both terminals are analogous
                i = i + 1                   #Move to the next token of w
            else:                           #Both terminals are different
                print ('Error: No T')       #Print exception and stop
                return
```

48. Datový model

Množiny *FIRST*(α), *FOLLOW*(A) ve spojové reprezentaci: použití slovníku.

Popsány pouze neprázdné buňky přechodové tabulky: Klíč = terminál, Hodnota = pravidlo.

```
F = {
     'E' :  {'x':1, '(':1},
     'E2' : {'+':2, '-':3, ')':4, '$':4},
     'T' :  {'x':5, '(':5},
     'T2' : {'+':8, '-':8, '*':6, '/':7, ')':8, '$':8},
     'F' :  {'x':10, '(':9},
    }
```

Množina pravidel v $\mathcal{G}$:

```
P =  {
      1:['T', 'E2'],
      2:['+', 'T', 'E2'],
      3:['-', 'T', 'E2'],
      4:['eps'],
      5:['F', 'T2'],
      6:['*', 'F', 'T2'],
      7:['/', 'F', 'T2'],
      8:['eps'],
      9:['(', 'E', ')'],
     10:['x']
    }
```

Pak:

```
w = ['(', 'x', '+', 'x', ')', '*', 'x', '$']
parse(F, P, w)
```

49. Ukázka činnosti zásobníkového automatu

Vstupní slovo:

```
w = ['(', 'x', '+', 'x', ')', '*', 'x', '$']
```

Činnost zásobníkového automatu pro w a $\mathcal{G}$:

Pro neterminál na vrcholu S postupně přidáváme pravidla dle přechodové tabulky: F/F.

Na vrchol S se postupně dostávají hledané terminály.

Za terminál již nedosazujeme, pouze odebíráme z vrcholu S.

Vyhodnocení w v $\mathcal{G}$: znázornění jednotlivých kroků v S.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
						F	x				$+$		F	x													
			$($		T	T'	T'	T'	ϵ		T	T	T'	T'	T'	ϵ											
			E	E	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	ϵ			$*$						
		F	$)$	$)$	$)$	$)$	$)$	$)$	$)$	$)$	$)$	$)$	$)$	$)$	$)$	$)$	$)$	$)$	$)$		F	F	x				
	T	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	T'	ϵ		
	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	E'	ϵ

Výsledek: $w \in L(\mathcal{G})$.

50. Sémantická analýza

Sémantika popisuje vztah mezi syntaxí a výpočetním modelem.

Definuje význam jednotlivých konstruktů jazyka a přiřazuje jim funkcionalitu.

Aneb co jednotlivá slova vlastně znamenají?

Sémantická analýza:

- Poslední fáze “překladu”, navazuje na syntaktickou analýzu.
- Zpracovává derivační strom $\mathcal{G}$ vzhledem k w, interpretace gramatických pravidel.
- Pracuje s významem w, provádí ji *sémantický analyzátor*.

Příklad: podmíněný příkaz, různá syntaxe, stejný význam.

```
if ( a < b)                 if a < b:
   x = x + 1;                   x = x + 1
else                        else:
   x = x - 1;                   x = x - 1
```

Syntaktická vs sémantická analýza:

SyA ověřuje, zda $w \in L(\mathcal{G})$ (je w platné?) chyby známy v době překladu.

SéA: interpretace w vzhledem k $\mathcal{G}$ (co w znamená?), chyby známy až za běhu.

Pro sémantickou analýzu konstruován *abstrakní syntaktický strom* (AST).

51. Abstraktní syntaktický strom (AST)

Kořenový strom, zjednodušení derivačního stromu.

Nezbytný pro sémantickou analýzu.

Používán v kompilátorech, umožňuje uchovat strukturu generovaného kódu.

Vlastnosti AST:

- Vynechává syntaxi pro popis problému redundantní: závorky, gramatická pravidla, ...
- Není konstruován pro určitý jazyk $L(\mathcal{G}) \Rightarrow$ *abstrakce*.
- Na rozdíl od derivačního stromu obsahuje pouze terminály: neumožňuje zpětnou rekonstrukci $\mathcal{G}$.
- Hustší, datově kompaktnější než derivační strom.
- Vnitřní uzel reprezentuje operátor, potomci jsou operandy.
- Jednoznačný pro každé w.

Konstrukce AST:

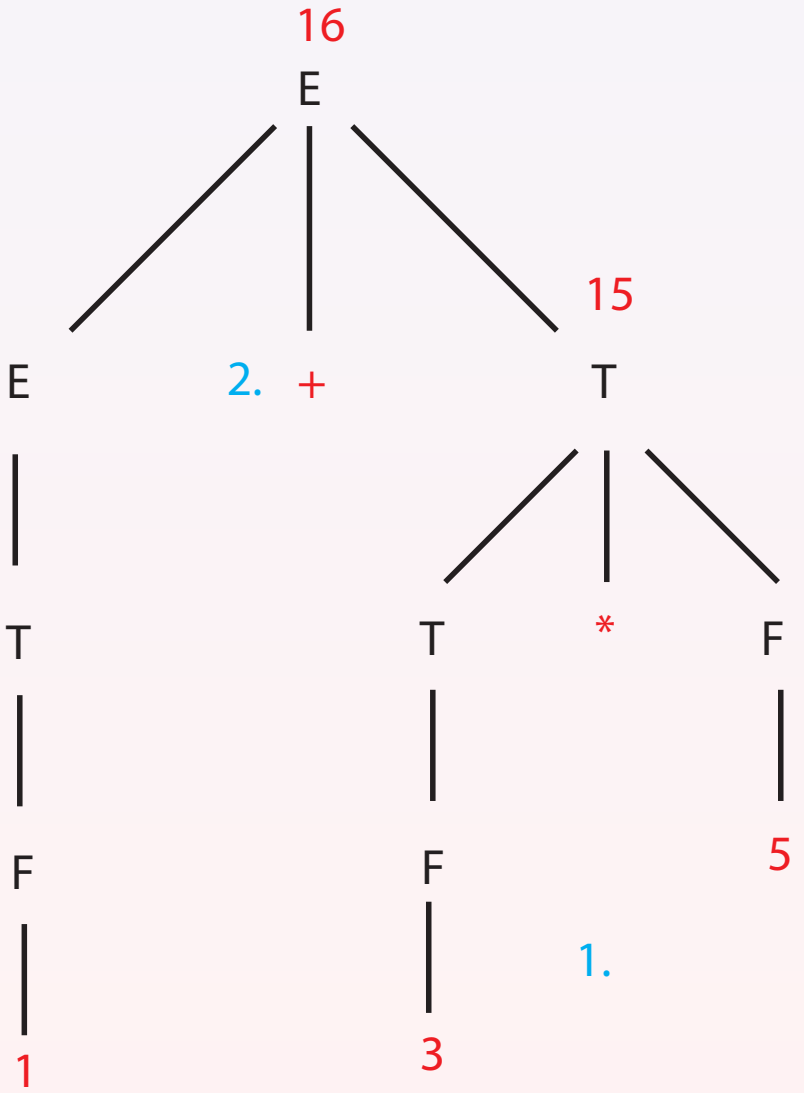
Transformace derivačního stromu obtížná, nepoužívá se.

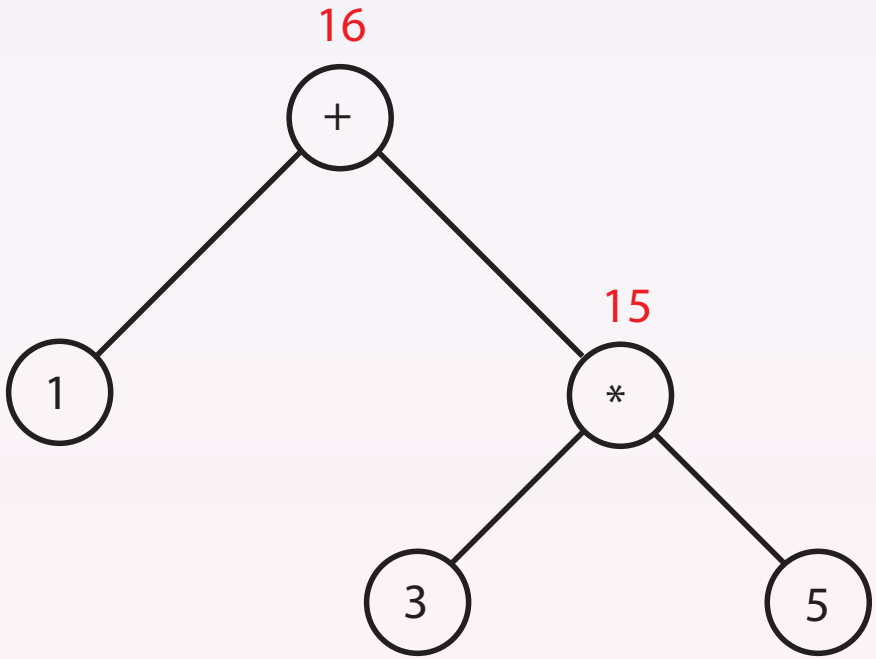
Zpravidla bottom->up konstrukce pro konkrétní $\mathcal{G}$.

Posun operátorů z listů do uzlů.

52. Derivační vs. abstraktní strom

w = 1 + 3 * 5 .





Funkční i pro w = 1 + (3 * 5)

53. Konstrukce AST pro $\mathcal{G}$

Vstupní gramatika

$$
\begin{aligned}
E &\to TE', \\
E' &\to +TE' | - TE' | \epsilon, \\
T &\to FT', \\
T' &\to *FT' | / FT' | \epsilon, \\
E' &\to (E) | x.
\end{aligned}
$$

Výpočet výrazu

$$w = 1 + 2 * (4 - 8).$$

Vlastnosti AST pro $\mathcal{G}$:

- Pro aritmetické operace AST binárním stromem, nebude vyvážený.
- Uzly stromu aritmetické operátory +, -, *, /.
- V kořeni operátor s nejmenší prioritou.
- Listy operandy (tj. čísla).
- Závorky netřeba, gramatika zachovává prioritu operací.

Konstrukce stromu zdola nahoru (bottom-up), tj. od uzlů ke kořenům.

Použití rekurze, naposledy vytvořen *kořen*, vrácen odkaz na kořen.

54. Datová reprezentace AST

Třída `BNode` binárního stromu, ukládá data, L a P potomka.

```
class BNode:
    def __init__(self, data, left, right):
        self.data = data
        self.left = left
        self.right = right
```

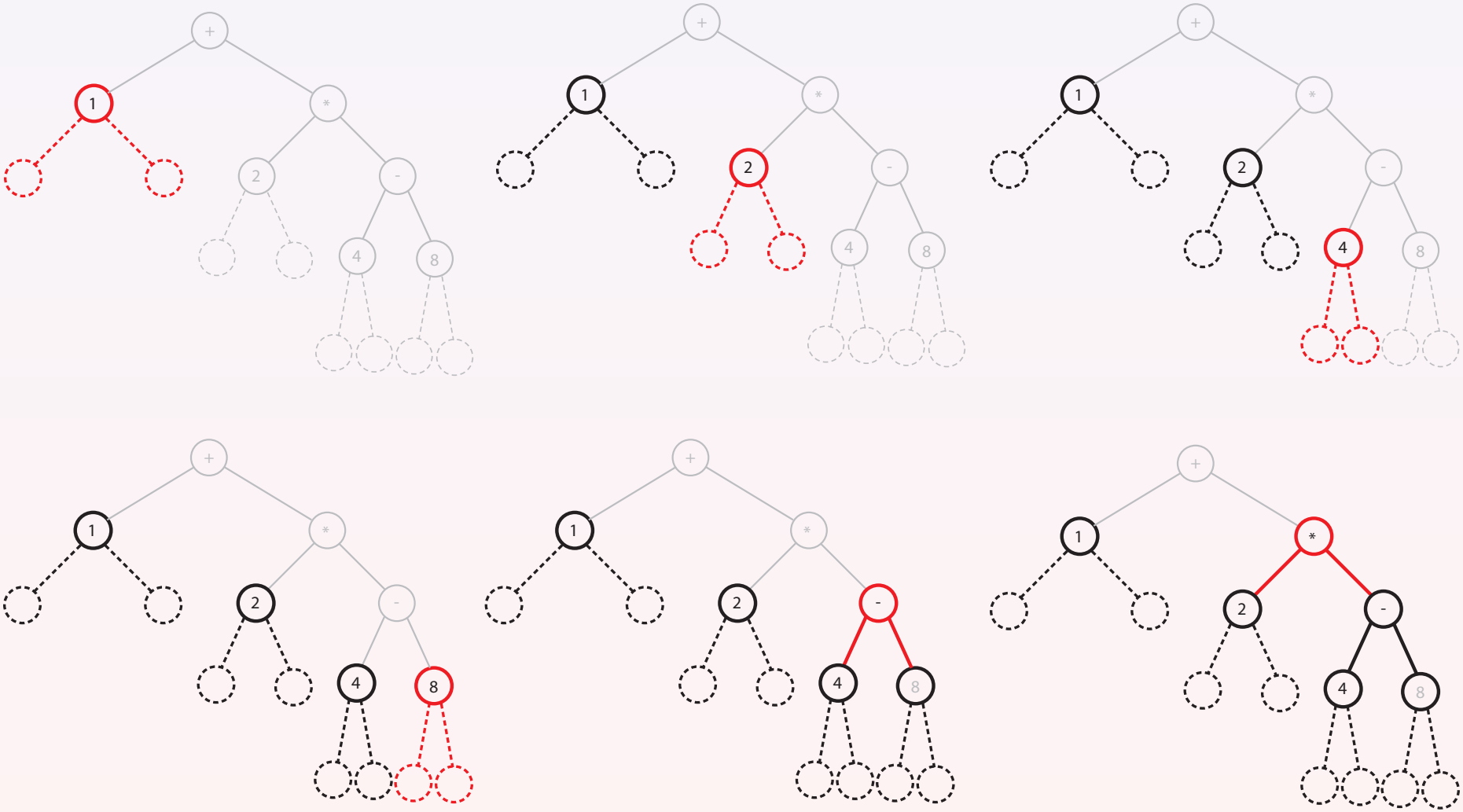
Třída `Parser` ukládá seznam tokenů a aktuální token

```
class Parser:
    def __init__ (self, tokens):
        self.i = 0                          #Index of current token in li
        self.tokens = tokens                #Initialize tokens
        self.cur_token = self.tokens[0]     #Initialialize current token
```

Umožňuje skok na následující token

```
def next(self):
     if self.i < len(self.tokens) - 1:          #Is there any valid token?
         self.i = self.i + 1                    #Increment token index
         self.cur_token = self.tokens[self.i]   #Actualize current token
     else:                                      #No more tokens
         self.cur_token = None
```

55. Ukázka konstrukce AST: $w = 1 + 2 * (4 - 8)$.



56. Výpočet *F*

Test, zda je token číslice.

Pokud ano, přidán jako nový list do AST, žádní potomci.

Pokud je závorkou, zpracujeme její obsah jako *E*.

Obě závorky přeskočeny.

```
def F(self):                                        # F | ( x )
    token = self.cur_token                          #Get current token
    if token.isdigit():                             #Token is number
        self.next()                                 #Move to the next token
        node = BNode(float(token), None, None)      #Return node, no parents
        node.print()
        return node
    elif token is '(':                              #Token is '(', skip
        self.next()                                 #Move to the next token
        node = self.E()                             #Get node of subtree: E
        self.next()                                 #Token is ')', skip
        return node                                 #Return node
```

57. Výpočet T

Nejprve zpracováno pravidlo

$$T \rightarrow FT'$$

pro L operand. Pokud detekováno *, /, řešeno 1 z pravidel (násobíme nebo dělíme)

$$T' \rightarrow *FT'|/FT',$$

hledán F pro P operand.

Následně substituce za T' do prvního vztahu.

Každé derivační pravidlo vpravo lze použít více než 1x, proto cyklus (podmínka nestačí).

```
def T(self):                                    #T * F | T / F | F
    node = self.F()                             #Get factor
    while self.cur_token in ('*', '/'):         #Process all tokens *, /
        token = self.cur_token
        self.next()                             #Move to the next token
        node = BNode(token, node, self.F())     #Create new node
        node.print()
    return node
```

58. Výpočet E

Analogický postup, nejprve zpracováno pravidlo

$$E \to +TE'$$

pro L operand. Pokud detekováno +, -, řešeno 1 z pravidel (sčítáme nebo odečítáme)

$$E' \to +TE' | -TE',$$

hledán T pro P operand.

Následně substituce za E' do prvního vztahu.

Každé derivační pravidlo vpravo lze použít více než 1x, proto cyklus (podmínka nestačí!).

```
def E(self):                                        # E + T | E - T | T
    node = self.T()                                 #Get term
    while self.cur_token in ('+', '-'):             #Process all tokens *, /
        token = self.cur_token
        self.next()                                 #Move to the next token
        node = BNode(token, node, self.T())         #Create new node
        node.print()
    return node
```

59. Interpretace AST

Interpretace binárního stromu metodou *postorder,* postup od kořene k listům:
Zpracován L podstrom, poté P podstrom, následně kořen.
Snadná implementace, použití dvojnásobné rekurze.

Operátory seřazeny dle priority vzestupně.
Pokud je v uzlu u operátor $\otimes$ (+, -, /, *):

- Rekurze na L podstrom $u.left$: levý operand.
- Rekurze na P podstrom $u.right$: pravý operand.
- Aplikace operátoru
$$res = u.left \otimes u.right.$$
- Podmínka ukončení rekurze: uzel neobsahuje $\otimes$, ale číslo.

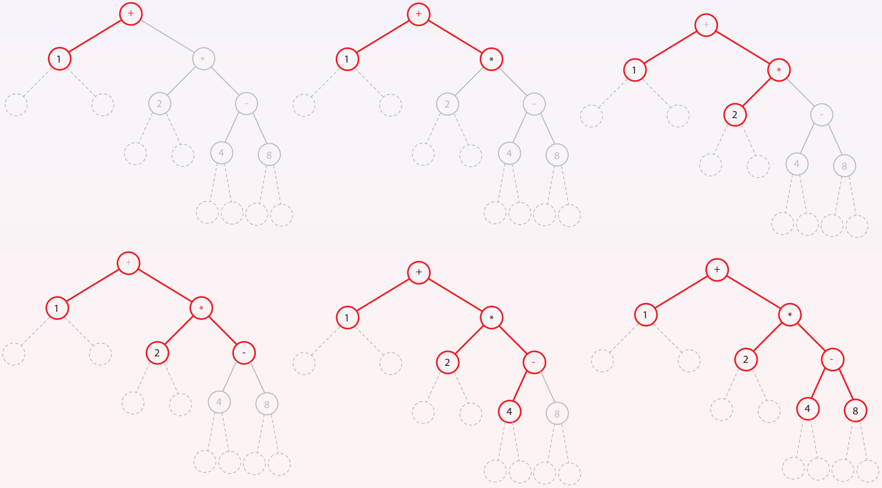
Přímý chod: od kořene dojdeme do listů.
Zpětný chod: od listů ke kořeni, výpočet hodnoty v uzlu z L, P potomků.

Ukázka pro operaci +:

```
def interpret(self, node):
    if node.data is '+':              #Node contains +
        lval = interpret(node.left)   #Proces left subtree
        rval = interpret(node.right)  #Proces right subtree
        res = lval + rval             #Add
        return res                    #Return res
    else:                             #Node contains number
        return node.data              #Stop recursion, return number
```

Lze zapsat efektivněji, bez pomocných proměnných.

60. Interpretace AST: přímý chod



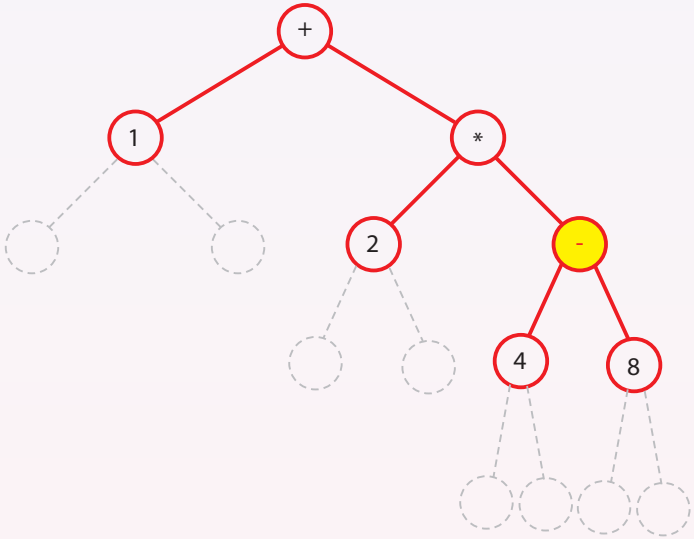
61. Interpretace AST: zpětný chod

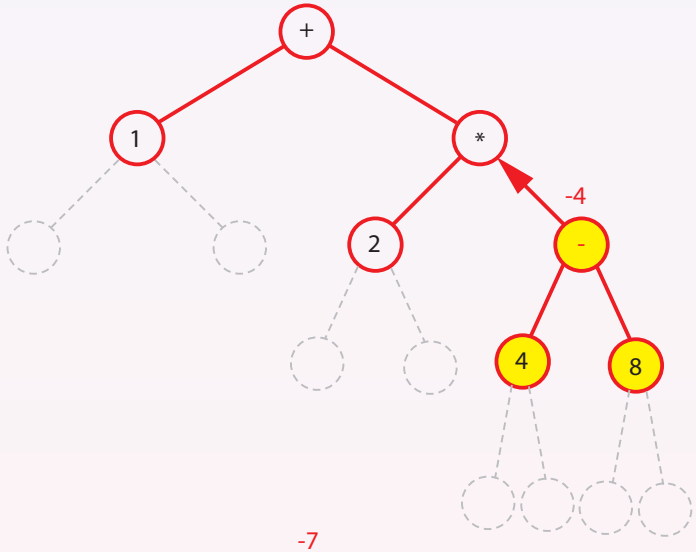
Procházení stromu bottom-up, od listů ke kořenu.

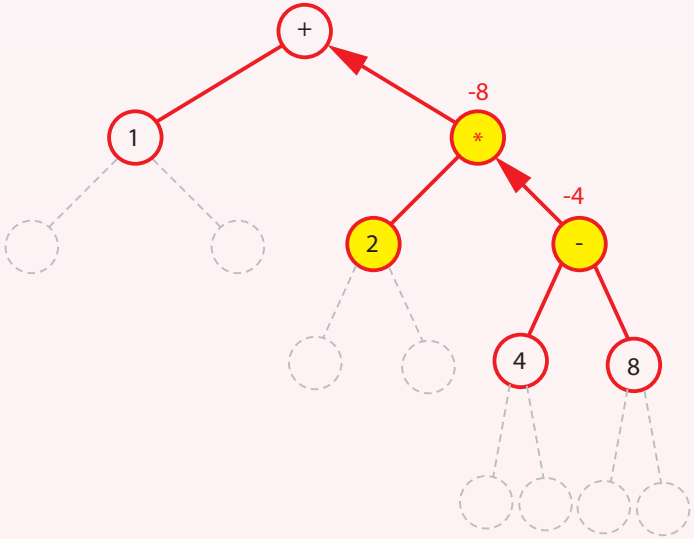
Začínáme v nejspodnějším uzlu obsahující operátor $\otimes$.

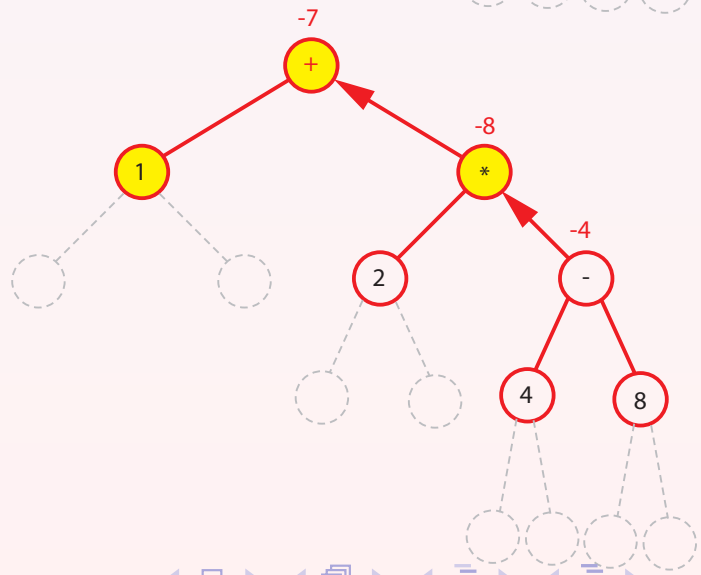
Dle $\otimes$ spočtena hodnota z jeho potomků.

Postup výše do dalšího uzlu obsahující operátor $\otimes$, atd.









62. Interpretace AST: implementace (1/2)

Procházení stromu bottom-up, od listů ke kořenu.

Pro každého rodiče spočtena hodnota z jeho potomků dle operátoru $\otimes$.

Operátory seřazeny dle priority.

Pokud uzel neobsahuje operátor, vrať hodnotu, ukončení rekurze.

```
def interpret(self, node):
    if node.data is '+':                    #Operator is +
        return self.interpret(node.left) +  #Process left subtree
               self.interpret(node.right)   #Process right subtree
    elif node.data is '-':                  #Operator is -
        return self.interpret(node.left) -  #Process left subtree
               self.interpret(node.right)   #Process right subtree
    elif node.data is '*':                  #Operator is *
        return self.interpret(node.left) *  #Process left subtree
               self.interpret(node.right)   #Process right subtree
    elif node.data is '/':                  #Operator is \
        return self.interpret(node.left) /  #Process left subtree
               self.interpret(node.right)   #Process right subtree
    else:                                   #Node has no child
        return node.data                    #Process number, stop recursio
```

63. Interpretace AST: implementace (2/2)

Ukázka:

```
p = Parser(['1','+','2','*', '(', '4', '-', '8', ')'])
root = p.parse()
res = p.interpret(root)
print(res)
```

Postup konstrukce uzlů:

```
None  1.0  None
None  2.0  None
None  4.0  None
None  8.0  None
4.0    -   8.0
2.0    *    -
1.0    +    *
```

Existuje i nerekurzivní implementace, využití zásobníku.