

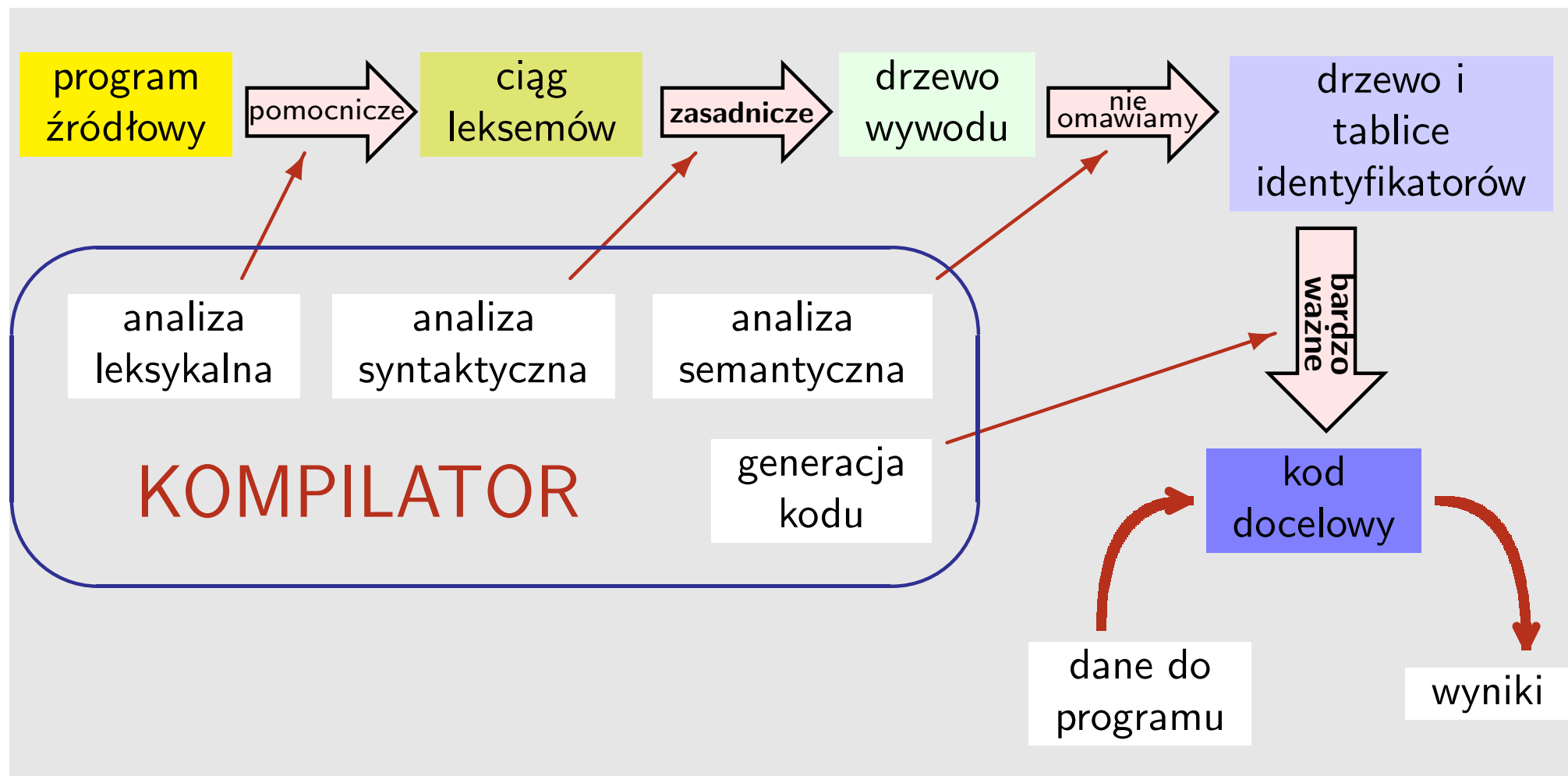
Stefan Sokołowski

JĘZYKI FORMALNE I METODY KOMPILACJI

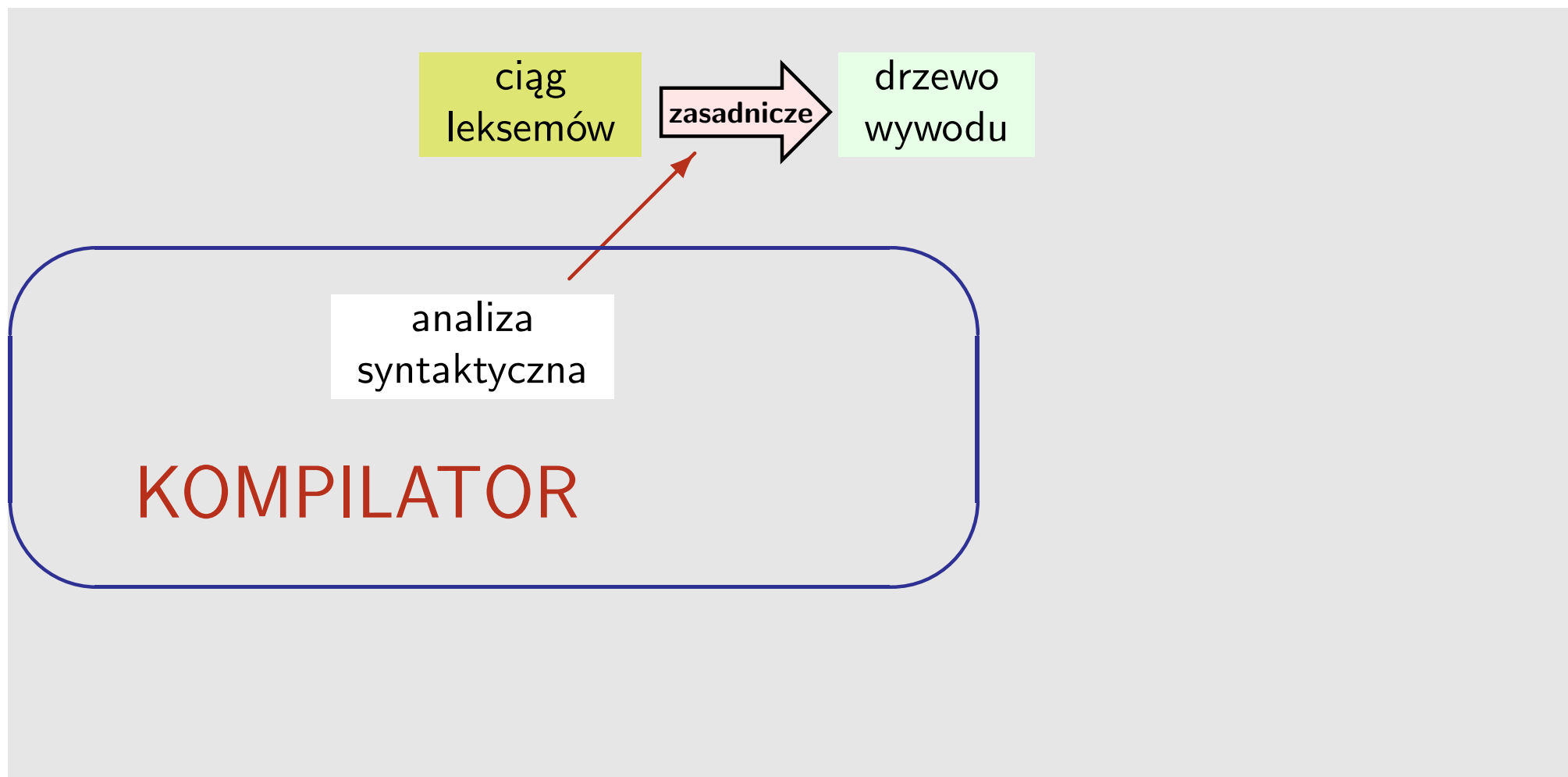
wykład 7

**Inst. Informatyki Stosowanej, ANS
Elbląg, 2025/2026**

Uproszczony schemat kompilacji



Uproszczony schemat kompilacji



Różne metody analizy syntaktycznej

Rozbiór słowa

— budowanie jego drzewa wyvodu w danej gramatyce **bezkontekstowej**.

Różne metody analizy syntaktycznej

Rozbiór słowa

— budowanie jego drzewa wyvodu w danej gramatyce **bezkontekstowej**.

Synonimy: rozbiór = analiza syntaktyczna = parsing.

Różne metody analizy syntaktycznej

Rozbiór słowa

— budowanie jego drzewa wywodu w danej gramatyce **bezkontekstowej**.

Synonimy: rozbiór = analiza syntaktyczna = parsing.

Parser — program (procedura, podprogram) dokonujący rozbioru.

Różne metody analizy syntaktycznej

Rozbiór słowa

— budowanie jego drzewa wyvodu w danej gramatyce **bezkontekstowej**.

Synonimy: rozbiór = analiza syntaktyczna = parsing.

Parser — program (procedura, podprogram) dokonujący rozbioru.

Budowy drzewa wyvodu można dokonywać

- albo od strony **korzenia** (rozbiór *zstępujący*),
- albo od strony **liści** (rozbiór *wstępujący*).

Różne metody analizy syntaktycznej

Rozbiór słowa

— budowanie jego drzewa wyvodu w danej gramatyce **bezkontekstowej**.

Synonimy: rozbiór = analiza syntaktyczna = parsing.

Parser — program (procedura, podprogram) dokonujący rozbioru.

Budowy drzewa wyvodu można dokonywać

- albo od strony **korzenia** (rozbiór *zstępujący*),
- albo od strony **liści** (rozbiór *wstępujący*).

Jest wiele metod rozbioru danego słowa; różnią się

- zakresem stosowalności (do jakich rodzajów gramatyk bezkontekstowych się nadają)

Różne metody analizy syntaktycznej

Rozbiór słowa

— budowanie jego drzewa wyvodu w danej gramatyce **bezkontekstowej**.

Synonimy: rozbiór = analiza syntaktyczna = parsing.

Parser — program (procedura, podprogram) dokonujący rozbioru.

Budowy drzewa wyvodu można dokonywać

- albo od strony **korzenia** (rozbiór *zstępujący*),
- albo od strony **liści** (rozbiór *wstępujący*).

Jest wiele metod rozbioru danego słowa; różnią się

- zakresem stosowalności (do jakich rodzajów gramatyk bezkontekstowych się nadają),
- oraz efektywnością (jak szybko działają i ile potrzebują pamięci).

Różne metody analizy syntaktycznej

Rozbiór słowa

— budowanie jego drzewa wywodu w danej gramatyce **bezkontekstowej**.

Synonimy: rozbiór = analiza syntaktyczna = parsing.

Parser — program (procedura, podprogram) dokonujący rozbioru.

Budowy drzewa wywodu można dokonywać

- albo od strony **korzenia** (rozbiór *zstępujący*),
- albo od strony **liści** (rozbiór *wstępujący*).

Jest wiele metod rozbioru danego słowa; różnią się

- zakresem stosowalności (do jakich rodzajów gramatyk bezkontekstowych się nadają),
- oraz efektywnością (jak szybko działają i ile potrzebują pamięci).

Programy do automatycznego tworzenia parserów potrafią na ogół dobrać optymalny (albo przynajmniej niezły) parser do danej gramatyki.

Koszty

- Do każdej **jednoznacznej** gramatyki bezkontekstowej można zbudować parser, działający w czasie $\mathcal{O}(n^3)$, gdzie n jest długością ciągu terminali na wejściu.

Koszty

- Do każdej **jednoznacznej** gramatyki bezkontekstowej można zbudować parser, działający w czasie $\mathcal{O}(n^3)$, gdzie n jest długością ciągu terminali na wejściu.
- Ta złożoność wynika z działania przez próbowanie i wycofywanie się w przypadku złego dopasowania; drzewo wywodu jest wielokrotnie budowane i niszczone.

Koszty

- Do każdej **jednoznacznej** gramatyki bezkontekstowej można zbudować parser, działający w czasie $\mathcal{O}(n^3)$, gdzie n jest długością ciągu terminali na wejściu.
- Ta złożoność wynika z działania przez próbowanie i wycofywanie się w przypadku złego dopasowania; drzewo wywodu jest wielokrotnie budowane i niszczone.
- Niezbyt bolesne ograniczenia na gramatyki umożliwiają istnienie parsera, działającego w czasie $\mathcal{O}(n)$, czyli znacznie szybciej.

Koszty

- Do każdej **jednoznacznej** gramatyki bezkontekstowej można zbudować parser, działający w czasie $\mathcal{O}(n^3)$, gdzie n jest długością ciągu terminali na wejściu.
- Ta złożoność wynika z działania przez próbowanie i wycofywanie się w przypadku złego dopasowania; drzewo wywodu jest wielokrotnie budowane i niszczone.
- Niezbyt bolesne ograniczenia na gramatyki umożliwiają istnienie parsera, działającego w czasie $\mathcal{O}(n)$, czyli znacznie szybciej.
- **Wszystkie** praktycznie stosowane parsery działają w czasie $\mathcal{O}(n)$.

Analiza *top-down* — rekursywny parser zstępujący

Analiza *top-down* — rekursywny parser zstępujący

Przykład: (prościutki)

$$\begin{aligned}\langle \text{wyrażenie} \rangle &::= \langle \text{liczba} \rangle \mid + \langle \text{liczba} \rangle \langle \text{liczba} \rangle \\ \langle \text{liczba} \rangle &::= 0 \mid 1 \mid 2\end{aligned}$$

Analiza *top-down* — rekursywny parser zstępujący

Przykład: (prościutki)

$$\begin{aligned}\langle \text{wyrażenie} \rangle &::= \langle \text{liczba} \rangle \mid + \langle \text{liczba} \rangle \langle \text{liczba} \rangle \\ \langle \text{liczba} \rangle &::= 0 \mid 1 \mid 2\end{aligned}$$

Parser (dwie funkcje)

Analiza *top-down* — rekursywny parser zstępujący

Przykład: (prościutki)

$$\begin{aligned} \langle \text{wyrażenie} \rangle &::= \langle \text{liczba} \rangle \mid + \langle \text{liczba} \rangle \langle \text{liczba} \rangle \\ \langle \text{liczba} \rangle &::= 0 \mid 1 \mid 2 \end{aligned}$$

Parser (dwie funkcje):

```
funkcja liczba:
    if (wejście ∈ {0, 1, 2})
        OK;
    else BLAD;
```

Analiza *top-down* — rekursywny parser zstępujący

Przykład: (prościutki)

$$\begin{aligned} \langle \text{wyrażenie} \rangle &::= \langle \text{liczba} \rangle \mid + \langle \text{liczba} \rangle \langle \text{liczba} \rangle \\ \langle \text{liczba} \rangle &::= 0 \mid 1 \mid 2 \end{aligned}$$

Parser (dwie funkcje):

```
funkcja liczba:
    if (wejście ∈ {0, 1, 2})
        OK;
    else BLAD;
```

```
funkcja wyrażenie:
    if (liczba()) OK;
```

Analiza *top-down* — rekursywny parser zstępujący

Przykład: (prościutki)

$$\begin{aligned} \langle \text{wyrażenie} \rangle &::= \langle \text{liczba} \rangle \mid + \langle \text{liczba} \rangle \langle \text{liczba} \rangle \\ \langle \text{liczba} \rangle &::= 0 \mid 1 \mid 2 \end{aligned}$$

Parser (dwie funkcje):

```
funkcja liczba:
    if (wejście ∈ {0,1,2})
        OK;
    else BLAD;
```

```
funkcja wyrażenie:
    if (liczba()) OK;
    else
        if (wejście == '+') {

        } else BLAD;
```

Analiza *top-down* — rekursywny parser zstępujący

Przykład: (prościutki)

$$\langle \text{wyrażenie} \rangle ::= \langle \text{liczba} \rangle \mid + \langle \text{liczba} \rangle \langle \text{liczba} \rangle$$
$$\langle \text{liczba} \rangle ::= 0 \mid 1 \mid 2$$

Parser (dwie funkcje):

```
funkcja liczba:
    if (wejście ∈ {0,1,2})
        OK;
    else BLAD;
```

```
funkcja wyrażenie:
    if (liczba()) OK;
    else
        if (wejście == '+') {
            if (liczba())

                else BLAD;
        } else BLAD;
```

Analiza *top-down* — rekursywny parser zstępujący

Przykład: (prościutki)

$$\begin{aligned} \langle \text{wyrażenie} \rangle &::= \langle \text{liczba} \rangle \mid + \langle \text{liczba} \rangle \langle \text{liczba} \rangle \\ \langle \text{liczba} \rangle &::= 0 \mid 1 \mid 2 \end{aligned}$$

Parser (dwie funkcje):

```
funkcja liczba:
    if (wejście ∈ {0,1,2})
        OK;
    else BLAD;
```

```
funkcja wyrażenie:
    if (liczba()) OK;
    else
        if (wejście == '+') {
            if (liczba())
                if (liczba()) OK;
                else BLAD;
            else BLAD;
        } else BLAD;
```

Analiza *top-down* — rekursywny parser zstępujący

Przykład: (prościutki)

$$\langle \text{wyrażenie} \rangle ::= \langle \text{liczba} \rangle \mid + \langle \text{liczba} \rangle \langle \text{liczba} \rangle$$

$$\langle \text{liczba} \rangle ::= 0 \mid 1 \mid 2$$

Parser (dwie funkcje):

```
funkcja liczba:
    if (wejście ∈ {0, 1, 2})
        OK;
    else BLAD;
```

```
funkcja wyrażenie:
    if (liczba()) OK;
    else
        if (wejście == '+') {
            if (liczba())
                if (liczba()) OK;
            else BLAD;
        } else BLAD;
```

Analiza *top-down* — rekursywny parser zstępujący

Przykład: (rekursywny)

$$\begin{aligned}\langle \text{wyrażenie} \rangle &::= \langle \text{liczba} \rangle \mid + \langle \text{wyrażenie} \rangle \langle \text{wyrażenie} \rangle \\ \langle \text{liczba} \rangle &::= 0 \mid 1 \mid 2\end{aligned}$$

Analiza *top-down* — rekursywny parser zstępujący

Przykład: (rekursywny)

$$\begin{aligned} \langle \text{wyrażenie} \rangle &::= \langle \text{liczba} \rangle \mid + \langle \text{wyrażenie} \rangle \langle \text{wyrażenie} \rangle \\ \langle \text{liczba} \rangle &::= 0 \mid 1 \mid 2 \end{aligned}$$

Parser (dwie funkcje):

```
funkcja liczba:
    if (wejście ∈ {0,1,2})
        OK;
    else BLAD;
```

```
funkcja wyrażenie:
    if (liczba()) OK;
    else
        if (wejście == '+') {
            if (wyrażenie())
                if (wyrażenie()) OK;
            else BLAD;
        } else BLAD;
```

Analiza *top-down* — rekursywny parser zstępujący

Przykład: (rekursywny)

$$\langle \text{wyrażenie} \rangle ::= \langle \text{liczba} \rangle \mid + \langle \text{wyrażenie} \rangle \langle \text{wyrażenie} \rangle$$

$$\langle \text{liczba} \rangle ::= 0 \mid 1 \mid 2$$

Parser (dwie funkcje):

```
funkcja liczba:
    if (wejście ∈ {0,1,2})
        OK;
    else BLAD;
```

```
funkcja wyrażenie:
    if (liczba()) OK;
    else
        if (wejście == '+') {
            if (wyrażenie())
                if (wyrażenie()) OK;
            else BLAD;
        } else BLAD;
```

Analiza *top-down* — rekursywny parser zstępujący

Ogólne zasady (**bardzo nieformalnie**)

Analiza *top-down* — rekursywny parser zstępujący

Ogólne zasady (**bardzo nieformalnie**):

- po jednej funkcji na każdy nieterminal w gramatyce;

Analiza *top-down* — rekursywny parser zstępujący

Ogólne zasady (**bardzo nieformalnie**):

- po jednej funkcji na każdy nieterminal w gramatyce;
- ta funkcja zwraca informację, czy **się udało**, czy **błąd**;

Analiza *top-down* — rekursywny parser zstępujący

Ogólne zasady (**bardzo nieformalnie**):

- po jednej funkcji na każdy nieterminal w gramatyce;
- ta funkcja zwraca informację, czy **się udało**, czy **błąd**;
- ciało tej funkcji pisane wg produkcji z tego nieterminalu

Analiza *top-down* — rekursywny parser zstępujący

Ogólne zasady (**bardzo nieformalnie**):

- po jednej funkcji na każdy nieterminal w gramatyce;
- ta funkcja zwraca informację, czy **się udało**, czy **błąd**;
- ciało tej funkcji pisane wg produkcji z tego nieterminalu:
 - napotkanemu *nieterminalowi* w produkcji odpowiada:
`if (nieterminal())`

Analiza *top-down* — rekursywny parser zstępujący

Ogólne zasady (**bardzo nieformalnie**):

- po jednej funkcji na każdy nieterminal w gramatyce;
- ta funkcja zwraca informację, czy **się udało**, czy **błąd**;
- ciało tej funkcji pisane wg produkcji z tego nieterminalu:
 - napotkanemu *nieterminalowi* w produkcji odpowiada:
`if (nieterminal())`
 - napotkanemu *terminalowi* w produkcji odpowiada:
`if (wejście == terminal) { czyt_wejście(); ...`

Analiza *top-down* — rekursywny parser zstępujący

Ogólne zasady (**bardzo nieformalnie**):

- po jednej funkcji na każdy nieterminal w gramatyce;
- ta funkcja zwraca informację, czy **się udało**, czy **błąd**;
- ciało tej funkcji pisane wg produkcji z tego nieterminalu:
 - napotkanemu *nieterminalowi* w produkcji odpowiada:
`if (nieterminal())`
 - napotkanemu *terminalowi* w produkcji odpowiada:
`if (wejście == terminal) { czyt_wejście(); ...`
 - końcowi *nieostatniej* produkcji odpowiada: `OK; else ...`

Analiza *top-down* — rekursywny parser zstępujący

Ogólne zasady (**bardzo nieformalnie**):

- po jednej funkcji na każdy nieterminal w gramatyce;
 - ta funkcja zwraca informację, czy **się udało**, czy **błąd**;
 - ciało tej funkcji pisane wg produkcji z tego nieterminalu:
 - napotkanemu *nieterminalowi* w produkcji odpowiada:
`if (nieterminal())`
 - napotkanemu *terminalowi* w produkcji odpowiada:
`if (wejście == terminal) { czyt_wejście(); ...`
 - końcowi *nieostatniej* produkcji odpowiada: `OK; else ...`
 - końcowi *ostatniej* produkcji odpowiada: `OK;`
- oraz pozamykanie wszystkich niezamkniętych if-ów:
- `else BŁĄD;`

Analiza *top-down* — rekursywny parser zstępujący

Ogólne zasady (**bardzo nieformalnie**):

- po jednej funkcji na każdy nieterminal w gramatyce;
- ta funkcja zwraca informację, czy **się udało**, czy **błąd**;
- ciało tej funkcji pisane wg produkcji z tego nieterminalu:
 - napotkanemu *nieterminalowi* w produkcji odpowiada:
`if (nieterminal())`
 - napotkanemu *terminalowi* w produkcji odpowiada:
`if (wejście == terminal) { czyt_wejście(); ...`
 - końcowi *nieostatniej* produkcji odpowiada: `OK; else ...`
 - końcowi *ostatniej* produkcji odpowiada: `OK;`oraz pozamykanie wszystkich niezamkniętych if-ów:
`else BŁĄD;`
- funkcja musi jeszcze budować drzewo rozbioru...

Rekursywny parser zstępujący

Analiza *top-down* — zstępująca.

Rekursywny parser zstępujący

Analiza *top-down* — zstępująca. Zawiera po jednej funkcji rekursywnej dla każdego nieterminala.

Rekursywny parser zstępujący

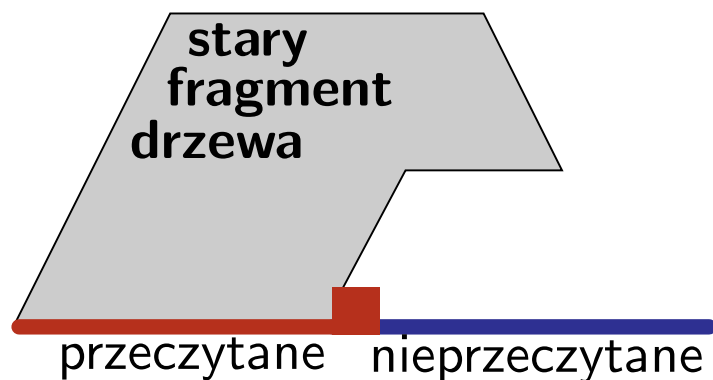
Analiza *top-down* — zstępująca. Zawiera po jednej funkcji rekursywnej dla każdego nieterminala. Funkcja dla nieterminala $\langle X \rangle$

- konstruuje nad leksemami z wejścia drzewo wywodu z nieterminalem $\langle X \rangle$ w korzeniu

Rekursywny parser zstępujący

Analiza *top-down* — zstępująca. Zawiera po jednej funkcji rekursywnej dla każdego nieterminala. Funkcja dla nieterminala $\langle X \rangle$

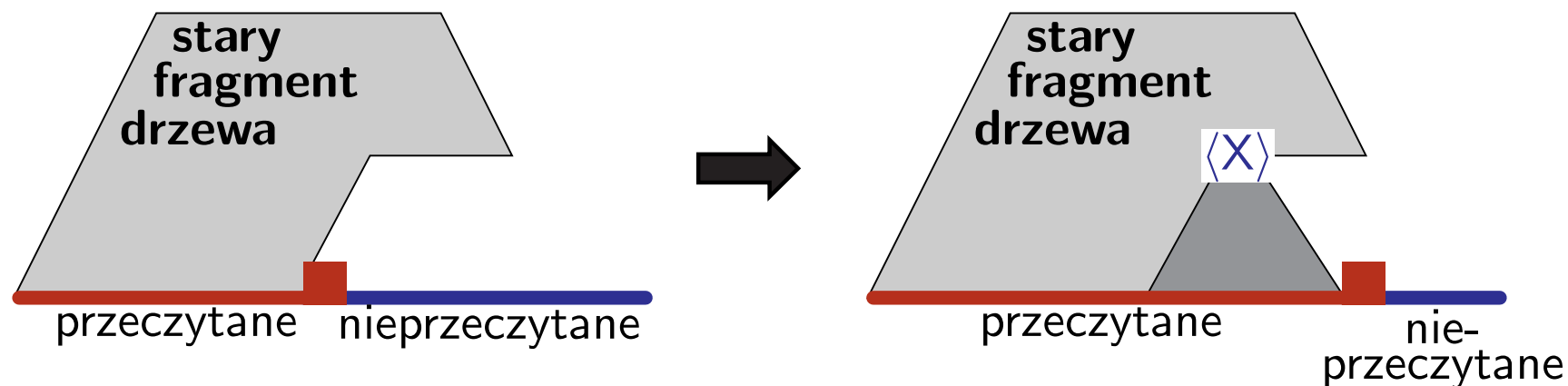
- konstruuje nad leksemami z wejścia drzewo wyvodu z nieterminalem $\langle X \rangle$ w korzeniu:



Rekursywny parser zstępujący

Analiza *top-down* — zstępująca. Zawiera po jednej funkcji rekursywnej dla każdego nieterminala. Funkcja dla nieterminala $\langle X \rangle$

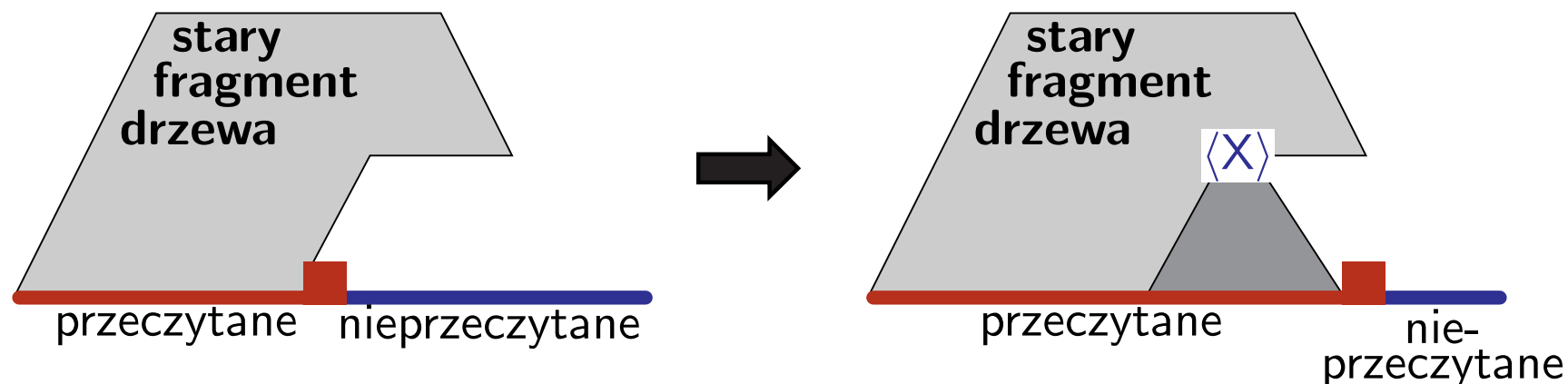
- konstruuje nad leksemami z wejścia drzewo wyvodu z nieterminalem $\langle X \rangle$ w korzeniu:



Rekursywny parser zstępujący

Analiza *top-down* — zstępująca. Zawiera po jednej funkcji rekursywnej dla każdego nieterminala. Funkcja dla nieterminala $\langle X \rangle$

- konstruuje nad leksemami z wejścia drzewo wywodu z nieterminalem $\langle X \rangle$ w korzeniu:

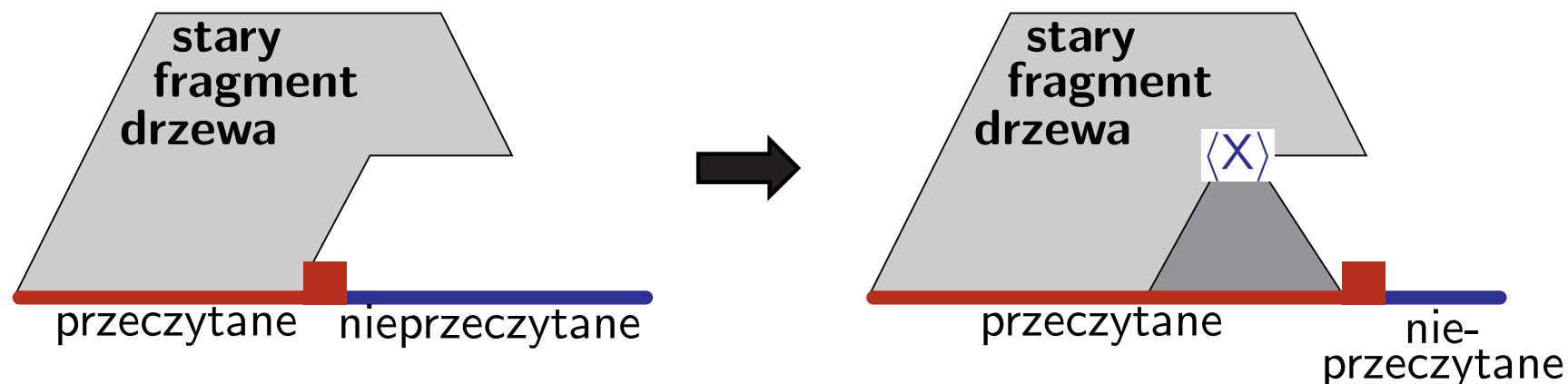


- przed — znany (wczytany) jest pierwszy liść (leksem) tego drzewa;
po — znany jest pierwszy leksem *spoza* korony tego drzewa;

Rekursywny parser zstępujący

Analiza *top-down* — zstępująca. Zawiera po jednej funkcji rekursywnej dla każdego nieterminala. Funkcja dla nieterminala $\langle X \rangle$

- konstruuje nad leksemami z wejścia drzewo wywodu z nieterminala $\langle X \rangle$ w korzeniu:



- przed — znany (wczytany) jest pierwszy liść (leksem) tego drzewa;
po — znany jest pierwszy leksem *spoza* korony tego drzewa;
- może zasygnalizować błąd i wtedy program ją wywołujący stara się znaleźć inną produkcję do zastosowania.

Rekursywny parser zstępujący

$$\langle B \rangle ::= w \langle B \rangle \mid w$$

Rekursywny parser zstępujący

$\langle B \rangle ::= w \langle B \rangle \mid w$

```
Boolean B(drzewo* drz) {
```

```
}
```

Rekursywny parser zstępujący

$\langle B \rangle ::= w \langle B \rangle \mid w$

KONSTRUOWANE
DRZEWO

```
Boolean B(drzewo* drz) {
```

```
}
```

Rekursywny parser zstępujący

$\langle B \rangle ::= w \langle B \rangle \mid w$

CZY SIĘ
UDAŁO

KONSTRUOWANE
DRZEWO

```
Boolean B(drzewo* drz) {
```

```
}
```

Rekursywny parser zstępujący

$$\langle B \rangle ::= w \langle B \rangle \mid w$$

CZY SIĘ UDAŁO

KONSTRUOWANE DRZEWO

```
Boolean B(drzewo* drz) {  
    drzewo drz1;  
    if (nowyleks('w'))  
  
        else return false;  
}
```

Rekursywny parser zstępujący

$$\langle B \rangle ::= \underline{w \langle B \rangle} \mid \underline{w}$$

CZY SIĘ
UDAŁO

KONSTRUOWANE
DRZEWO

```
Boolean B(drzewo* drz) {
    drzewo drz1;
    if (nowyleks('w'))
        if (B(&drz1)){

        }
    else {

    }
    else return false;
}
```

Rekursywny parser zstępujący

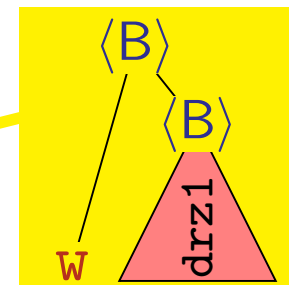
$$\langle B \rangle ::= \underline{w \langle B \rangle} \mid \underline{w}$$

CZY SIĘ
UDAŁO

KONSTRUOWANE
DRZEWO

```
Boolean B(drzewo* drz) {
    drzewo drz1;
    if (nowyleks('w'))
        if (B(&drz1)) {
            *drz = ...;
            return true;
        }
    else {

    }
    else return false;
}
```



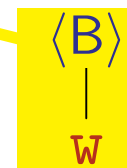
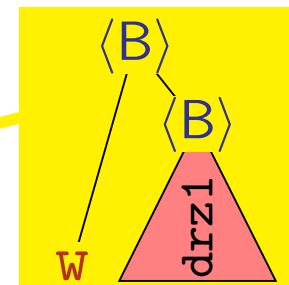
Rekursywny parser zstępujący

$$\langle B \rangle ::= \underline{w \langle B \rangle} \mid w$$

CZY SIĘ
UDAŁO

KONSTRUOWANE
DRZEWO

```
Boolean B(drzewo* drz) {
    drzewo drz1;
    if (nowyleks('w'))
        if (B(&drz1)){
            *drz = ...;
            return true;
        }
    else {
        *drz = ...;
        return true;
    }
    else return false;
}
```



Rekursywny parser zstępujący

$\langle A \rangle ::= y \langle A \rangle \langle B \rangle z \mid x$

```
*drz = ...;  
return true;
```

Rekursywny parser zstępujący

$\langle A \rangle ::= y \langle A \rangle \langle B \rangle z \mid x$

```
Boolean A(drzewo* drz) {  
    drzewo drz1, drz2;
```

```
}
```

Rekursywny parser zstępujący

$$\langle A \rangle ::= \underline{y \langle A \rangle \langle B \rangle z} \mid \underline{x}$$

```
Boolean A(drzewo* drz) {
    drzewo drz1, drz2;
    if (nowyleks('y'))

else
    if (nowyleks('x')) {

    }
    else return false;
}
```

Rekursywny parser zstępujący

$$\langle A \rangle ::= \underline{y\langle A \rangle \langle B \rangle z} \mid \underline{x}$$

```
Boolean A(drzewo* drz) {
    drzewo drz1, drz2;
    if (nowyleks('y'))
        if (A(&drz1))

        else blad("brakuje nieterminala <A>");
    else
        if (nowyleks('x')) {

        }
    else return false;
}
```

Rekursywny parser zstępujący

$$\langle A \rangle ::= \underline{y\langle A \rangle \langle B \rangle z} \mid \underline{x}$$

```
Boolean A(drzewo* drz) {
    drzewo drz1, drz2;
    if (nowyleks('y'))
        if (A(&drz1))
            if (B(&drz2))

        else blad("brakuje nieterminala <B>");
    else blad("brakuje nieterminala <A>");
    else
        if (nowyleks('x')) {

        }
    else return false;
}
```

Rekursywny parser zstępujący

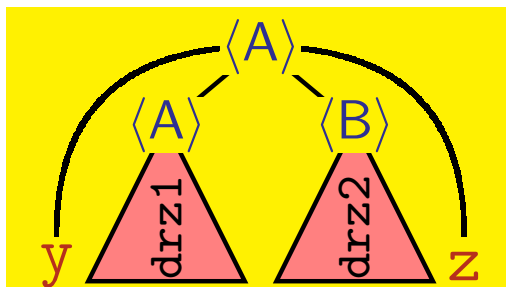
$$\langle A \rangle ::= \underline{y\langle A \rangle \langle B \rangle z} \mid \underline{x}$$

```
Boolean A(drzewo* drz) {
    drzewo drz1, drz2;
    if (nowyleks('y'))
        if (A(&drz1))
            if (B(&drz2))
                if (nowyleks('z')) {

                }
            else blad("brakuje terminala z");
            else blad("brakuje nieterminala <B>");
            else blad("brakuje nieterminala <A>");
        else
            if (nowyleks('x')) {

            }
        else return false;
    }
}
```

Rekursywny parser zstępujący

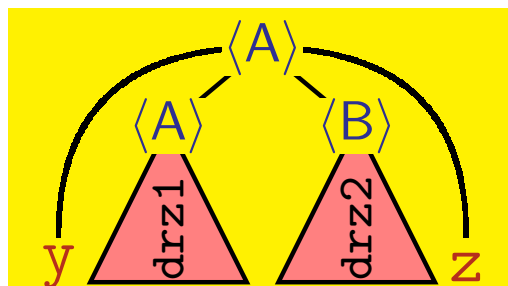
$$\langle A \rangle ::= \underline{y \langle A \rangle \langle B \rangle z} \mid \underline{x}$$


```
Boolean A(drzewo* drz) {
    drzewo drz1, drz2;
    if (nowyleks('y'))
        if (A(&drz1))
            if (B(&drz2))
                if (nowyleks('z')) {
                    *drz = ...;
                    return true;
                }
            else blad("brakuje terminala z");
        else blad("brakuje nieterminala <B>");
    else blad("brakuje nieterminala <A>");
    else
        if (nowyleks('x')) {

        }
    else return false;
}
```

Rekursywny parser zstępujący

$\langle A \rangle ::= y \langle A \rangle \langle B \rangle z \mid x$



```
Boolean A(drzewo* drz) {
    drzewo drz1, drz2;
    if (nowyleks('y'))
        if (A(&drz1))
            if (B(&drz2))
                if (nowyleks('z')) {
                    *drz = ...;
                    return true;
                }
            else blad("brakuje terminala z");
        else blad("brakuje nieterminala <B>");
    else blad("brakuje nieterminala <A>");
    else
        if (nowyleks('x')) {
            *drz = ...;
            return true;
        }
    else return false;
}
```

Rekursywny parser wstępujący

Funkcja `nowyleks(z)` sprawdza, czy na wejściu jest znak *z*

Rekursywny parser wstępujący

Funkcja `nowyleks(z)` sprawdza, czy na wejściu jest znak *z* i

- jeśli **jest**, to oddaje `true` i **sczytuje** ten znak z wejścia

Rekursywny parser wstępujący

Funkcja `nowyleks(z)` sprawdza, czy na wejściu jest znak *z* i

- jeśli **jest**, to oddaje `true` i **zaczytuje** ten znak z wejścia;
- jeśli **nie ma**, to oddaje `false` i **nie zmienia** wejścia; wtedy można sprawdzić, czy może na wejściu jest jakiś inny znak.

Rekursywny parser wstępujący

Funkcja `nowyleks(z)` sprawdza, czy na wejściu jest znak *z* i

- jeśli **jest**, to oddaje `true` i **sczytuje** ten znak z wejścia;
- jeśli **nie ma**, to oddaje `false` i **nie zmienia** wejścia; wtedy można sprawdzić, czy może na wejściu jest jakiś inny znak.

Znaku sczytanego nie można zwrócić na wejście!

Rekursywny parser wstępujący

Funkcja `nowyleks(z)` sprawdza, czy na wejściu jest znak *z* i

- jeśli **jest**, to oddaje `true` i **sczytuje** ten znak z wejścia;
- jeśli **nie ma**, to oddaje `false` i **nie zmienia** wejścia; wtedy można sprawdzić, czy może na wejściu jest jakiś inny znak.

Znaku sczytanego nie można zwrócić na wejście!

Jeśli czytanie z wejścia doszło do końca produkcji, to `return true;`.

Rekursywny parser wstępujący

Funkcja `nowyleks(z)` sprawdza, czy na wejściu jest znak *z* i

- jeśli **jest**, to oddaje `true` i **sczytuje** ten znak z wejścia;
- jeśli **nie ma**, to oddaje `false` i **nie zmienia** wejścia; wtedy można sprawdzić, czy może na wejściu jest jakiś inny znak.

Znaku sczytanego nie można zwrócić na wejście!

Jeśli czytanie z wejścia doszło do końca produkcji, to `return true;`.

Jeśli znak na wejściu **nie zgadza się** z produkcją, ale istnieje inna produkcja z tego samego nieterminalu, z którą się zgadza, to trzeba się na nią przestawić.

Rekursywny parser wstępujący

Funkcja `nowyleks(z)` sprawdza, czy na wejściu jest znak *z* i

- jeśli **jest**, to oddaje `true` i **sczytuje** ten znak z wejścia;
- jeśli **nie ma**, to oddaje `false` i **nie zmienia** wejścia; wtedy można sprawdzić, czy może na wejściu jest jakiś inny znak.

Znaku sczytanego nie można zwrócić na wejście!

Jeśli czytanie z wejścia doszło do końca produkcji, to `return true;`.

Jeśli znak na wejściu **nie zgadza się** z produkcją, ale istnieje inna produkcja z tego samego nieterminalu, z którą się zgadza, to trzeba się na nią przestawić.

Jeśli takiej produkcji nie ma, ale nic z wejścia nie sczytane, to `return false;` (błąd **miękki**).

Rekursywny parser wstępujący

Funkcja `nowyleks(z)` sprawdza, czy na wejściu jest znak *z* i

- jeśli **jest**, to oddaje `true` i **sczytuje** ten znak z wejścia;
- jeśli **nie ma**, to oddaje `false` i **nie zmienia** wejścia; wtedy można sprawdzić, czy może na wejściu jest jakiś inny znak.

Znaku sczytanego nie można zwrócić na wejście!

Jeśli czytanie z wejścia doszło do końca produkcji, to `return true;`.

Jeśli znak na wejściu **nie zgadza się** z produkcją, ale istnieje inna produkcja z tego samego nieterminalu, z którą się zgadza, to trzeba się na nią przestawić.

Jeśli takiej produkcji nie ma, ale nic z wejścia nie sczytane, to `return false;` (błąd **miękki**).

Jeśli takiej produkcji nie ma, ale już coś sczytane, to `bład(...);` (błąd **twardy**).

Rekursywny parser wstępujący

Przykład:

$$\langle S \rangle ::= a b \mid a c$$

Rekursywny parser wstępujący

Przykład:

$\langle S \rangle ::= a b \mid a c$

```
Boolean S(drzewo* drz) {  
    if (nowyleks('a'))  
        if (nowyleks('b')) {  
            (konstrukcja drz); return true;  
        }  
    else if (nowyleks('c')) {  
        (konstrukcja drz); return true;  
    }  
    else blad( ... );  
    else return false;  
}
```

← twardy

← miękki

Przekształcenie gramatyki dla parsera zstępującego

Przekształcenie gramatyki dla parsera zstępującego

Zmiana kolejności produkcji i „wyłączenie przed nawias”:

$\langle W \rangle$	$::=$	$\langle S \rangle$		$\langle S \rangle + \langle W \rangle$
				$\langle S \rangle - \langle W \rangle$
$\langle S \rangle$	$::=$	$\langle C \rangle$		$\langle C \rangle * \langle S \rangle$
				$\langle C \rangle / \langle S \rangle$
$\langle C \rangle$	$::=$	$\langle L \rangle$		$(\langle W \rangle)$
$\langle L \rangle$	$::=$	0		1
				$0 \langle L \rangle$
				$1 \langle L \rangle$

Przekształcenie gramatyki dla parsera zstępującego

Zmiana kolejności produkcji i „wyłączenie przed nawias”:

$\langle W \rangle$	$::=$	$\langle S \rangle$	$ $	$\langle S \rangle + \langle W \rangle$
			$ $	$\langle S \rangle - \langle W \rangle$
$\langle S \rangle$	$::=$	$\langle C \rangle$	$ $	$\langle C \rangle * \langle S \rangle$
			$ $	$\langle C \rangle / \langle S \rangle$
$\langle C \rangle$	$::=$	$\langle L \rangle$	$ $	$(\langle W \rangle)$
$\langle L \rangle$	$::=$	0	$ $	1
			$ $	$0 \langle L \rangle$
			$ $	$1 \langle L \rangle$

$\langle W \rangle$	$::=$	$\langle S \rangle$	$\{$	$+$	$\langle W \rangle$	$ $	$-$	$\langle W \rangle$	$ $	λ	$\}$
---------------------	-------	---------------------	------	-----	---------------------	-----	-----	---------------------	-----	-----------	------

Przekształcenie gramatyki dla parsera zstępującego

Zmiana kolejności produkcji i „wyłączenie przed nawias”:

$$\begin{aligned}
 \langle W \rangle &::= \langle S \rangle \mid \langle S \rangle + \langle W \rangle \\
 &\quad \mid \langle S \rangle - \langle W \rangle \\
 \langle S \rangle &::= \langle C \rangle \mid \langle C \rangle * \langle S \rangle \\
 &\quad \mid \langle C \rangle / \langle S \rangle \\
 \langle C \rangle &::= \langle L \rangle \mid (\langle W \rangle) \\
 \langle L \rangle &::= 0 \mid 1 \mid 0 \langle L \rangle \mid 1 \langle L \rangle
 \end{aligned}$$

$$\begin{aligned}
 \langle W \rangle &::= \langle S \rangle \{ + \langle W \rangle \mid - \langle W \rangle \mid \lambda \} \\
 \langle S \rangle &::= \langle C \rangle \{ * \langle S \rangle \mid / \langle S \rangle \mid \lambda \}
 \end{aligned}$$

Przekształcenie gramatyki dla parsera zstępującego

Zmiana kolejności produkcji i „wyłączenie przed nawias”:

$$\begin{aligned}
 \langle W \rangle &::= \langle S \rangle \mid \langle S \rangle + \langle W \rangle \\
 &\quad \mid \langle S \rangle - \langle W \rangle \\
 \langle S \rangle &::= \langle C \rangle \mid \langle C \rangle * \langle S \rangle \\
 &\quad \mid \langle C \rangle / \langle S \rangle \\
 \langle C \rangle &::= \langle L \rangle \mid (\langle W \rangle) \\
 \langle L \rangle &::= 0 \mid 1 \mid 0 \langle L \rangle \mid 1 \langle L \rangle
 \end{aligned}$$

$$\begin{aligned}
 \langle W \rangle &::= \langle S \rangle \{ + \langle W \rangle \mid - \langle W \rangle \mid \lambda \} \\
 \langle S \rangle &::= \langle C \rangle \{ * \langle S \rangle \mid / \langle S \rangle \mid \lambda \} \\
 \langle C \rangle &::= (\langle W \rangle) \mid \langle L \rangle
 \end{aligned}$$

Przekształcenie gramatyki dla parsera zstępującego

Zmiana kolejności produkcji i „wyłączenie przed nawias”:

$$\begin{aligned}
 \langle W \rangle &::= \langle S \rangle \mid \langle S \rangle + \langle W \rangle \\
 &\quad \mid \langle S \rangle - \langle W \rangle \\
 \langle S \rangle &::= \langle C \rangle \mid \langle C \rangle * \langle S \rangle \\
 &\quad \mid \langle C \rangle / \langle S \rangle \\
 \langle C \rangle &::= \langle L \rangle \mid (\langle W \rangle) \\
 \langle L \rangle &::= 0 \mid 1 \mid 0 \langle L \rangle \mid 1 \langle L \rangle
 \end{aligned}$$

$$\begin{aligned}
 \langle W \rangle &::= \langle S \rangle \{ + \langle W \rangle \mid - \langle W \rangle \mid \lambda \} \\
 \langle S \rangle &::= \langle C \rangle \{ * \langle S \rangle \mid / \langle S \rangle \mid \lambda \} \\
 \langle C \rangle &::= (\langle W \rangle) \mid \langle L \rangle \\
 \langle L \rangle &::= 0 \{ \langle L \rangle \mid \lambda \} \mid 1 \{ \langle L \rangle \mid \lambda \}
 \end{aligned}$$

Programowanie parsera zstępującego

$$\langle W \rangle ::= \langle S \rangle \mid \langle S \rangle + \langle W \rangle \mid \langle S \rangle - \langle W \rangle$$

Programowanie parsera zstępującego

$$\langle W \rangle ::= \langle S \rangle \mid \langle S \rangle + \langle W \rangle \mid \langle S \rangle - \langle W \rangle$$
$$\langle W \rangle ::= \langle S \rangle \left\{ + \langle W \rangle \mid - \langle W \rangle \mid \lambda \right\}$$

Programowanie parsera zstępującego

$$\langle W \rangle ::= \langle S \rangle \mid \langle S \rangle + \langle W \rangle \mid \langle S \rangle - \langle W \rangle$$

$$\langle W \rangle ::= \langle S \rangle \{ + \langle W \rangle \mid - \langle W \rangle \mid \lambda \}$$

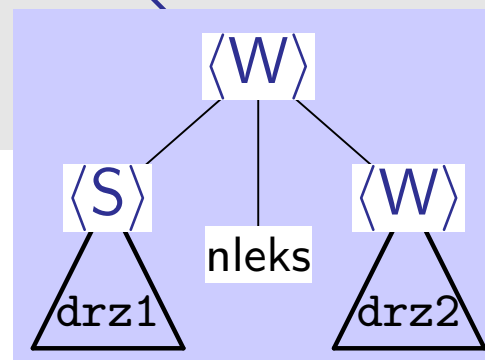
```
Boolean W(drzewo* drz) {
    drzewo drz1, drz2;
    if (S(&drz1))
        if (nowyleks('+', &nleks) || nowyleks('-', &nleks))
            if (W(&drz2)) { *drz = ...; return TRUE; }
            else blad("po + lub - nie ma <W>");
        else { *drz = ...; return TRUE; }
    else return FALSE;
}
```

Programowanie parsera zstępującego

$$\langle W \rangle ::= \langle S \rangle \mid \langle S \rangle + \langle W \rangle \mid \langle S \rangle - \langle W \rangle$$

$$\langle W \rangle ::= \langle S \rangle \{ + \langle W \rangle \mid - \langle W \rangle \mid \lambda \}$$

```
Boolean W(drzewo* drz) {
    drzewo drz1, drz2;
    if (S(&drz1))
        if (nowyleks('+', &nleks) || nowyleks('-', &nleks))
            if (W(&drz2)) { *drz = ...; return TRUE; }
            else blad("po + lub - nie ma <W>");
        else { *drz = ...; return TRUE; }
    else return FALSE;
}
```

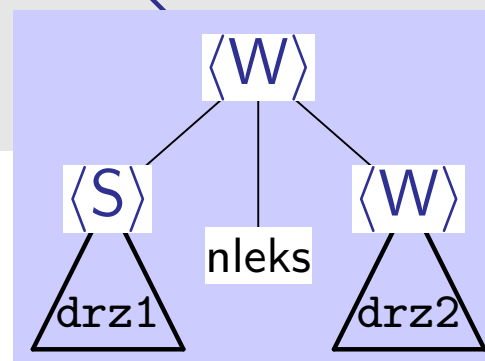
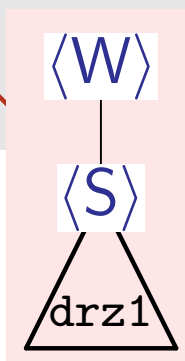


Programowanie parsera zstępującego

$$\langle W \rangle ::= \langle S \rangle \mid \langle S \rangle + \langle W \rangle \mid \langle S \rangle - \langle W \rangle$$

$$\langle W \rangle ::= \langle S \rangle \{ + \langle W \rangle \mid - \langle W \rangle \mid \lambda \}$$

```
Boolean W(drzewo* drz) {
    drzewo drz1, drz2;
    if (S(&drz1))
        if (nowyleks('+', &nleks) || nowyleks('-', &nleks))
            if (W(&drz2)) { *drz = ...; return TRUE; }
            else blad("po + lub - nie ma <W>");
        else { *drz = ...; return TRUE; }
    else return FALSE;
}
```

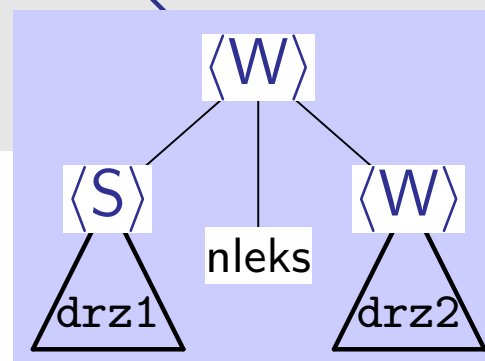
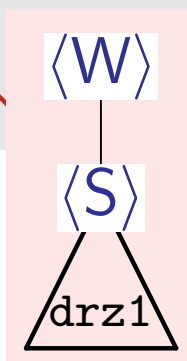


Programowanie parsera zstępującego

$$\langle W \rangle ::= \langle S \rangle \mid \langle S \rangle + \langle W \rangle \mid \langle S \rangle - \langle W \rangle$$

$$\langle W \rangle ::= \langle S \rangle \{ + \langle W \rangle \mid - \langle W \rangle \mid \lambda \}$$

```
Boolean W(drzewo* drz) {
    drzewo drz1, drz2;
    if (S(&drz1))
        if (nowyleks('+', &nleks) || nowyleks('-', &nleks))
            if (W(&drz2)) { *drz = ...; return TRUE; }
            else blad("po + lub - nie ma <W>");
        else { *drz = ...; return TRUE; }
    else return FALSE;
}
```



Uwaga: To jeszcze nie jest dokładnie to, co trzeba. Patrz dalej.

Parser zstępujący — problemy

Lewostronna rekursja w gramatyce powoduje nieskończone obliczenie.

Parser zstępujący — problemy

Lewostronna rekursja w gramatyce powoduje nieskończone obliczenie.

PRZYKŁAD: inna gramatyka: $\langle W \rangle ::= \langle S \rangle \mid \langle W \rangle + \langle S \rangle$

Parser zstępujący — problemy

Lewostronna rekursja w gramatyce powoduje nieskończone obliczenie.

PRZYKŁAD: inna gramatyka: $\langle W \rangle ::= \langle S \rangle \mid \langle W \rangle + \langle S \rangle$

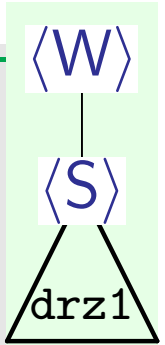
```
Boolean W(drzewo* drz) {  
    drzewo drz1, drz2;  
    if (S(&drz1)) { *drz = ...; return TRUE; }  
    else  
        if (W(&drz1)) {  
            if (nowyleks('+', &nleks))  
                if (S(&drz2)) { *drz = ...; return TRUE; }  
                else blad("po + brakuje <S>");  
            else return FALSE;  
        }  
    else return FALSE;  
}
```

Parser zstępujący — problemy

Lewostronna rekursja w gramatyce powoduje nieskończone obliczenie.

PRZYKŁAD: inna gramatyka: $\langle W \rangle ::= \langle S \rangle \mid \langle W \rangle + \langle S \rangle$

```
Boolean W(drzewo* drz) {
    drzewo drz1, drz2;
    if (S(&drz1)) { *drz = ...; return TRUE; }
    else
        if (W(&drz1)) {
            if (nowyleks('+', &nleks))
                if (S(&drz2)) { *drz = ...; return TRUE; }
                else blad("po + brakuje <S>");
            else return FALSE;
        }
    else return FALSE;
}
```

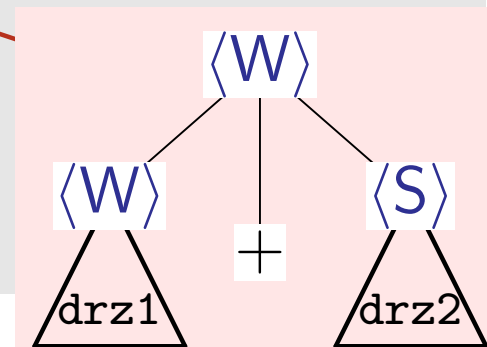
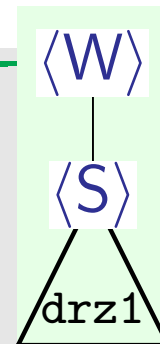


Parser zstępujący — problemy

Lewostronna rekursja w gramatyce powoduje nieskończone obliczenie.

PRZYKŁAD: inna gramatyka: $\langle W \rangle ::= \langle S \rangle \mid \langle W \rangle + \langle S \rangle$

```
Boolean W(drzewo* drz) {
    drzewo drz1, drz2;
    if (S(&drz1)) { *drz = ...; return TRUE; }
    else
        if (W(&drz1)) {
            if (nowyleks('+', &nleks))
                if (S(&drz2)) { *drz = ...; return TRUE; }
            else blad("po + brakuje <S>");
            else return FALSE;
        }
    else return FALSE;
}
```

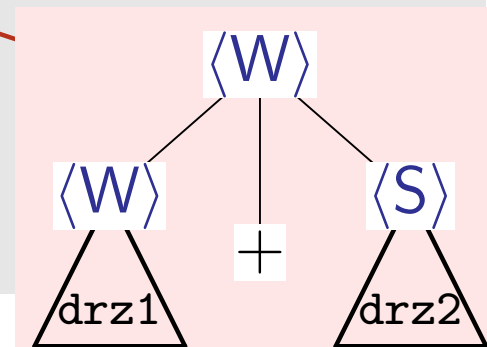
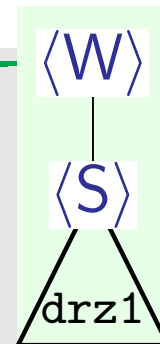


Parser zstępujący — problemy

Lewostronna rekursja w gramatyce powoduje nieskończone obliczenie.

PRZYKŁAD: inna gramatyka: $\langle W \rangle ::= \langle S \rangle \mid \langle W \rangle + \langle S \rangle$

```
Boolean W(drzewo* drz) {
    drzewo drz1, drz2;
    if (S(&drz1)) { *drz = ...; return TRUE; }
    else
        if (W(&drz1)) {
            if (nowyleks('+', &nleks))
                if (S(&drz2)) { *drz = ...; return TRUE; }
            else blad("po + brakuje <S>");
            else return FALSE;
        }
    else return FALSE;
}
```

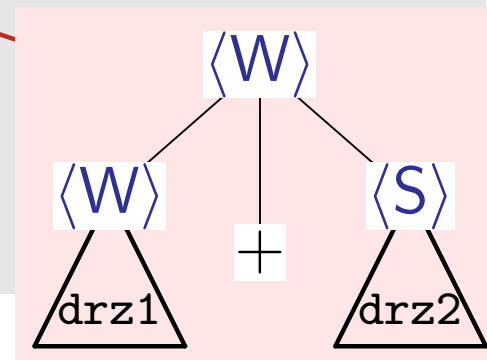
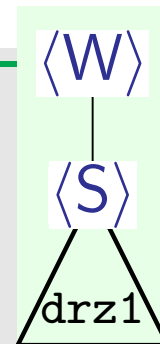


Parser zstępujący — problemy

Lewostronna rekursja w gramatyce powoduje nieskończone obliczenie.

PRZYKŁAD: inna gramatyka: $\langle W \rangle ::= \langle S \rangle \mid \langle W \rangle + \langle S \rangle$

```
Boolean W(drzewo* drz) {
    drzewo drz1, drz2;
    if (S(&drz1)) { *drz = ...; return TRUE; }
    else
        if (W(&drz1)) {
            if (nowyleks('+', &nleks))
                if (S(&drz2)) { *drz = ...; return TRUE; }
            else blad("po + brakuje <S>");
        }
    else return FALSE;
}
```



— wewnętrzne W (w ciele funkcji) wywoływane jest dla tego samego stanu wejścia, co zewnętrzne — **ślepa rekursja**.

Leczenie lewostronnej rekursji

Problem: produkcje gramatyczne postaci

$$\langle N \rangle ::= \dots \mid \langle N \rangle \dots \mid \dots$$

(czyli *lewostronnie rekursywne*) prowadzą do zapętlenia parsera zstępującego.

Leczenie lewostronnej rekursji

Problem: produkcje gramatyczne postaci

$$\langle N \rangle ::= \dots \mid \langle N \rangle \dots \mid \dots$$

(czyli *lewostronnie rekursywne*) prowadzą do zapętlenia parsera zstępującego. Takie produkcje potrzebne są dla często spotykanej w składni języków programowania konstrukcji „*ciąg ... grupowany do lewej*”.

Leczenie lewostronnej rekursji

Problem: produkcje gramatyczne postaci

$$\langle N \rangle ::= \dots \mid \langle N \rangle \dots \mid \dots$$

(czyli *lewostronnie rekursywne*) prowadzą do zapętlenia parsera zstępującego. Takie produkcje potrzebne są dla często spotykanej w składni języków programowania konstrukcji „*ciąg ... grupowany do lewej*”.

PRZYKŁAD:

$$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \mid \langle \text{cyfra} \rangle \langle \text{liczba} \rangle$$

— ciąg cyfr grupowany do prawej (nie ma problemu)

$$345 = 3_{\text{45}} = 3 \cdot 10^2 + 45$$

Leczenie lewostronnej rekursji

Problem: produkcje gramatyczne postaci

$$\langle N \rangle ::= \dots \mid \langle N \rangle \dots \mid \dots$$

(czyli *lewostronnie rekursywne*) prowadzą do zapętlenia parsera zstępującego. Takie produkcje potrzebne są dla często spotykanej w składni języków programowania konstrukcji „*ciąg ... grupowany do lewej*”.

PRZYKŁAD:

$$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \mid \langle \text{cyfra} \rangle \langle \text{liczba} \rangle$$

— ciąg cyfr grupowany do **prawej** (nie ma problemu)

$$345 = 3 \lfloor \textcolor{red}{45} \rfloor = 3 \cdot 10^2 + 45$$

$$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \mid \langle \text{liczba} \rangle \langle \text{cyfra} \rangle$$

— ciąg cyfr grupowany do **lewej** (jest problem)

$$345 = \lfloor \textcolor{red}{34} \rfloor 5 = 34 \cdot 10 + 5$$

Leczenie lewostronnej rekursji

Lewostronna rekursja (lub lewostronna rekursja wzajemna) w gramatyce — powoduje zapętlenie parsera zstępującego

Leczenie lewostronnej rekursji

Lewostronna rekursja (lub lewostronna rekursja wzajemna) w gramatyce — powoduje zapętlenie parsera zstępującego

Co na to można zrobić?

Leczenie lewostronnej rekursji

Lewostronna rekursja (lub lewostronna rekursja wzajemna) w gramatyce — powoduje zapętlenie parsera zstępującego

Co na to można zrobić?

Zastąpić

$\langle X \rangle ::= L \mid \langle X \rangle M$

przez

$\langle X \rangle ::= LM^*$

Leczenie lewostronnej rekursji

Lewostronna rekursja (lub lewostronna rekursja wzajemna) w gramatyce — powoduje zapętlenie parsera zstępującego

Co na to można zrobić?

Zastąpić

$\langle X \rangle ::= L \mid \langle X \rangle M$

przez

$\langle X \rangle ::= LM^*$

Nieformalnie:

M^* oznacza „ciąg M -ów grupowany do lewej”.

Leczenie lewostronnej rekursji

PRZYKŁAD:

zamiast

$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \mid \langle \text{liczba} \rangle \langle \text{cyfra} \rangle$

Leczenie lewostronnej rekursji

PRZYKŁAD:

zamiast

$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \mid \langle \text{liczba} \rangle \langle \text{cyfra} \rangle$

— **liczba** to cyfra albo **liczba** a po niej cyfra

Leczenie lewostronnej rekursji

PRZYKŁAD:

zamiast

$$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \mid \langle \text{liczba} \rangle \langle \text{cyfra} \rangle$$

— **liczba** to cyfra albo **liczba** a po niej cyfra

piszemy

$$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \langle \text{cyfra} \rangle^*$$

Leczenie lewostronnej rekursji

PRZYKŁAD:

zamiast

$$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \mid \langle \text{liczba} \rangle \langle \text{cyfra} \rangle$$

— **liczba** to cyfra albo **liczba** a po niej cyfra

piszemy

$$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \langle \text{cyfra} \rangle^*$$

— **liczba** to cyfra a po niej ciąg cyfr

Leczenie lewostronnej rekursji

PRZYKŁAD:

zamiast

$$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \mid \langle \text{liczba} \rangle \langle \text{cyfra} \rangle$$

— **liczba** to cyfra albo **liczba** a po niej cyfra

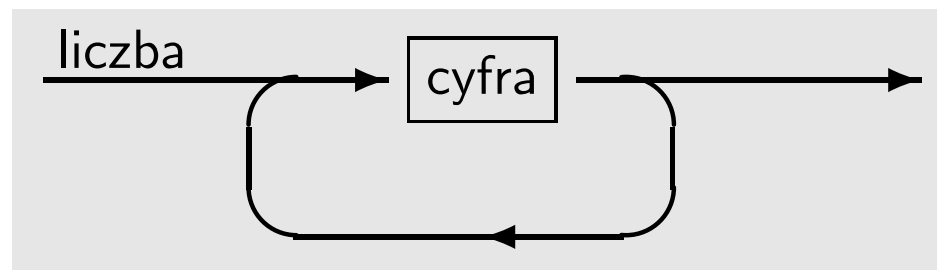
piszemy

$$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \langle \text{cyfra} \rangle^*$$

— **liczba** to cyfra a po niej ciąg cyfr

$$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \langle \text{cyfra} \rangle^*$$

W rysunkowej notacji Pascala:



Leczenie lewostronnej rekursji

PRZYKŁAD:

zamiast

$\langle \text{wyrażenie} \rangle$	$::=$	$\langle \text{składnik} \rangle$
		$\langle \text{wyrażenie} \rangle + \langle \text{składnik} \rangle$
		$\langle \text{wyrażenie} \rangle - \langle \text{składnik} \rangle$

Leczenie lewostronnej rekursji

PRZYKŁAD:

zamiast

$$\langle \text{wyrażenie} \rangle ::= \langle \text{składnik} \rangle$$

$$| \langle \text{wyrażenie} \rangle + \langle \text{składnik} \rangle$$

$$| \langle \text{wyrażenie} \rangle - \langle \text{składnik} \rangle$$

czyli $\left(\langle \text{wyrażenie} \rangle ::= \langle \text{składnik} \rangle \right.$
 $\left. | \langle \text{wyrażenie} \rangle \{ + \mid - \} \langle \text{składnik} \rangle \right)$

Leczenie lewostronnej rekursji

PRZYKŁAD:

zamiast

$$\langle \text{wyrażenie} \rangle ::= \langle \text{składnik} \rangle$$

$$\quad \quad \quad \langle \text{wyrażenie} \rangle + \langle \text{składnik} \rangle$$

$$\quad \quad \quad \langle \text{wyrażenie} \rangle - \langle \text{składnik} \rangle$$

czyli $\left(\begin{array}{l} \langle \text{wyrażenie} \rangle ::= \langle \text{składnik} \rangle \\ \quad \quad \quad \langle \text{wyrażenie} \rangle \{ + \mid - \} \langle \text{składnik} \rangle \end{array} \right)$

piszemy

$$\langle \text{wyrażenie} \rangle ::= \langle \text{składnik} \rangle \{ \{ + \mid - \} \langle \text{składnik} \rangle \}^*$$

Leczenie lewostronnej rekursji

PRZYKŁAD:

zamiast

$$\langle \text{wyrażenie} \rangle ::= \langle \text{składnik} \rangle$$

$$\quad \quad \quad \langle \text{wyrażenie} \rangle + \langle \text{składnik} \rangle$$

$$\quad \quad \quad \langle \text{wyrażenie} \rangle - \langle \text{składnik} \rangle$$

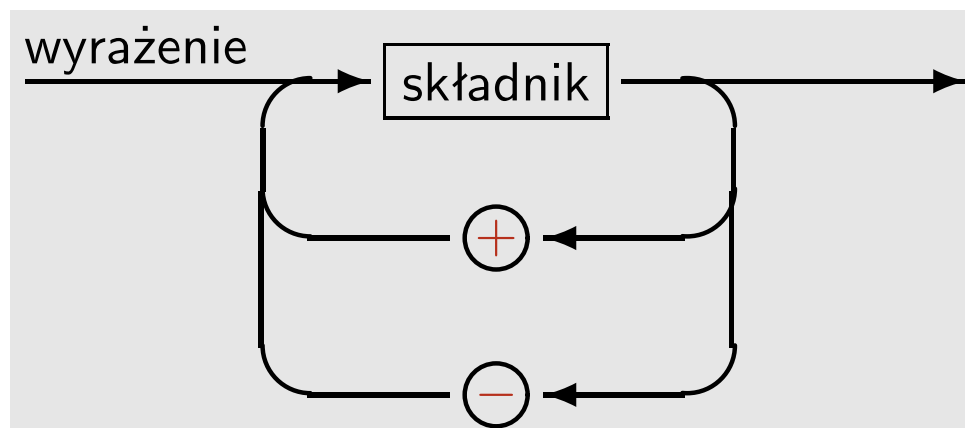
czyli $\left(\begin{array}{l} \langle \text{wyrażenie} \rangle ::= \langle \text{składnik} \rangle \\ \quad \quad \quad \langle \text{wyrażenie} \rangle \{ + \mid - \} \langle \text{składnik} \rangle \end{array} \right)$

piszemy

$$\langle \text{wyrażenie} \rangle ::= \langle \text{składnik} \rangle \{ \{ + \mid - \} \langle \text{składnik} \rangle \}^*$$

$$\langle \text{wyrażenie} \rangle ::= \langle \text{składnik} \rangle \{ \{ + \mid - \} \langle \text{składnik} \rangle \}^*$$

W rysunkowej notacji Pascala:



Leczenie lewostronnej rekursji

Gramatyka oryginalna
z lewostronną rekursją:

$$\langle W \rangle ::= \langle S \rangle \mid \langle W \rangle + \langle S \rangle$$

$$\mid \langle W \rangle - \langle S \rangle$$

$$\langle S \rangle ::= \langle C \rangle \mid \langle S \rangle * \langle C \rangle$$

$$\mid \langle S \rangle / \langle C \rangle$$

$$\langle C \rangle ::= \langle L \rangle \mid (\langle W \rangle)$$

$$\langle L \rangle ::= 0 \mid 1 \mid \langle L \rangle 0 \mid \langle L \rangle 1$$

Leczenie lewostronnej rekursji

Gramatyka oryginalna
z lewostronną rekursją:

$$\langle W \rangle ::= \langle S \rangle \mid \langle W \rangle + \langle S \rangle$$

$$\mid \langle W \rangle - \langle S \rangle$$

$$\langle S \rangle ::= \langle C \rangle \mid \langle S \rangle * \langle C \rangle$$

$$\mid \langle S \rangle / \langle C \rangle$$

$$\langle C \rangle ::= \langle L \rangle \mid (\langle W \rangle)$$

$$\langle L \rangle ::= 0 \mid 1 \mid \langle L \rangle 0 \mid \langle L \rangle 1$$

Gramatyka wyleczona
z iteracją:

$$\langle W \rangle ::= \langle S \rangle \left\{ \left\{ + \mid - \right\} \langle S \rangle \right\}^*$$

Leczenie lewostronnej rekursji

Gramatyka oryginalna
z lewostronną rekursją:

$$\langle W \rangle ::= \langle S \rangle \mid \langle W \rangle + \langle S \rangle$$

$$\mid \langle W \rangle - \langle S \rangle$$

$$\langle S \rangle ::= \langle C \rangle \mid \langle S \rangle * \langle C \rangle$$

$$\mid \langle S \rangle / \langle C \rangle$$

$$\langle C \rangle ::= \langle L \rangle \mid (\langle W \rangle)$$

$$\langle L \rangle ::= 0 \mid 1 \mid \langle L \rangle 0 \mid \langle L \rangle 1$$

Gramatyka wyleczona
z iteracją:

$$\langle W \rangle ::= \langle S \rangle \{ \{ + \mid - \} \langle S \rangle \}^*$$

$$\langle S \rangle ::= \langle C \rangle \{ \{ * \mid / \} \langle C \rangle \}^*$$

Leczenie lewostronnej rekursji

Gramatyka oryginalna
z lewostronną rekursją:

$$\langle W \rangle ::= \langle S \rangle \mid \langle W \rangle + \langle S \rangle$$

$$\mid \langle W \rangle - \langle S \rangle$$

$$\langle S \rangle ::= \langle C \rangle \mid \langle S \rangle * \langle C \rangle$$

$$\mid \langle S \rangle / \langle C \rangle$$

$$\langle C \rangle ::= \langle L \rangle \mid (\langle W \rangle)$$

$$\langle L \rangle ::= 0 \mid 1 \mid \langle L \rangle 0 \mid \langle L \rangle 1$$

Gramatyka wyleczona
z iteracją:

$$\langle W \rangle ::= \langle S \rangle \left\{ \left\{ + \mid - \right\} \langle S \rangle \right\}^*$$

$$\langle S \rangle ::= \langle C \rangle \left\{ \left\{ * \mid / \right\} \langle C \rangle \right\}^*$$

$$\langle C \rangle ::= (\langle W \rangle) \mid \langle L \rangle$$

Leczenie lewostronnej rekursji

Gramatyka oryginalna
z lewostronną rekursją:

$$\langle W \rangle ::= \langle S \rangle \mid \langle W \rangle + \langle S \rangle$$

$$\mid \langle W \rangle - \langle S \rangle$$

$$\langle S \rangle ::= \langle C \rangle \mid \langle S \rangle * \langle C \rangle$$

$$\mid \langle S \rangle / \langle C \rangle$$

$$\langle C \rangle ::= \langle L \rangle \mid (\langle W \rangle)$$

$$\langle L \rangle ::= 0 \mid 1 \mid \langle L \rangle 0 \mid \langle L \rangle 1$$

Gramatyka wyleczona
z iteracją:

$$\langle W \rangle ::= \langle S \rangle \left\{ \left\{ + \mid - \right\} \langle S \rangle \right\}^*$$

$$\langle S \rangle ::= \langle C \rangle \left\{ \left\{ * \mid / \right\} \langle C \rangle \right\}^*$$

$$\langle C \rangle ::= (\langle W \rangle) \mid \langle L \rangle$$

$$\langle L \rangle ::= \left\{ 0 \mid 1 \right\} \left\{ 0 \mid 1 \right\}^*$$

Leczenie lewostronnej rekursji

Z każdym nieterminalem związujemy funkcję rekursywną:

- jej **ciało** wg gramatyki **wyleczonej**
- **iteracji** odpowiada pętla **while**
- budowanie **drzewa** wg gramatyki **oryginalnej**

Leczenie lewostronnej rekursji

Z każdym nieterminalem związujemy funkcję rekursywną:

- jej **ciało** wg gramatyki **wyleczonej**
- **iteracji** odpowiada pętla **while**
- budowanie **drzewa** wg gramatyki **oryginalnej**

$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \mid \langle \text{liczba} \rangle \langle \text{cyfra} \rangle$

$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \langle \text{cyfra} \rangle^*$

Leczenie lewostronnej rekursji

Z każdym nieterminalem związujemy funkcję rekursywną:

- jej **ciało** wg gramatyki **wyleczonej**
- **iteracji** odpowiada pętla **while**
- budowanie **drzewa** wg gramatyki **oryginalnej**

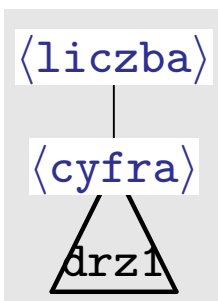
$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \mid \langle \text{liczba} \rangle \langle \text{cyfra} \rangle$

$\langle \text{liczba} \rangle ::= \langle \text{cyfra} \rangle \langle \text{cyfra} \rangle^*$

```
Boolean liczba (drzewo* drz) {
```

```
    if (cyfra(&drz1)) {
```

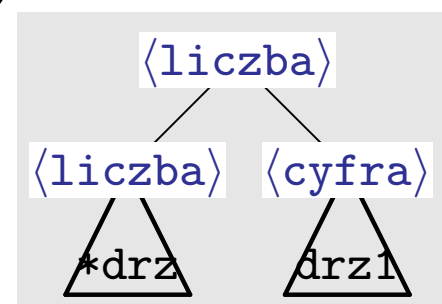
```
        *drz =
```



```
    ;
```

```
while (cyfra(&drz1))
```

```
    *drz =
```



```
    ;
```

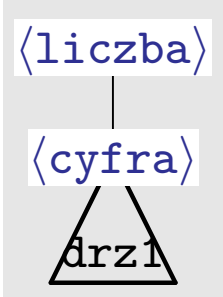
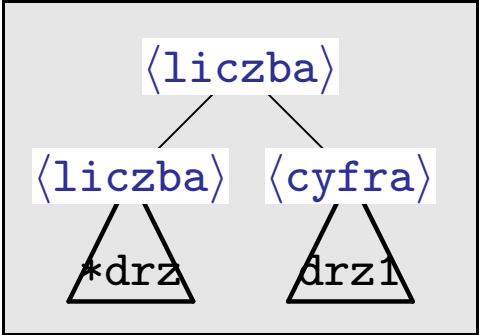
```
    return TRUE;
```

```
    }
    else return FALSE;
```

```
}
```

Leczenie lewostronnej rekursji

```

Boolean liczba (drzewo* drz) {
    if (cyfra(&drz1)) {
        *drz =
            
        while (cyfra(&drz1))
            *drz =
                
            ;
        return TRUE;
    }
    else return FALSE;
}

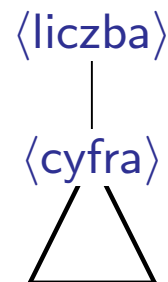
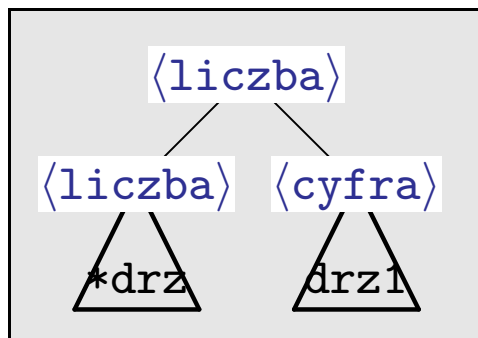
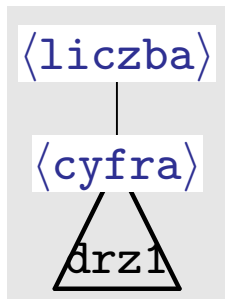
```

Leczenie lewostronnej rekursji

```

Boolean liczba (drzewo* drz) {
    if (cyfra(&drz1)) {
        *drz =
            while (cyfra(&drz1))
                *drz =
                    return TRUE;
    }
    else return FALSE;
}

```

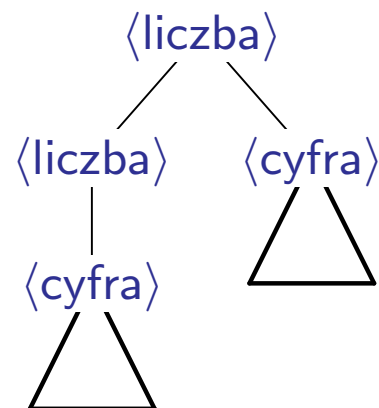
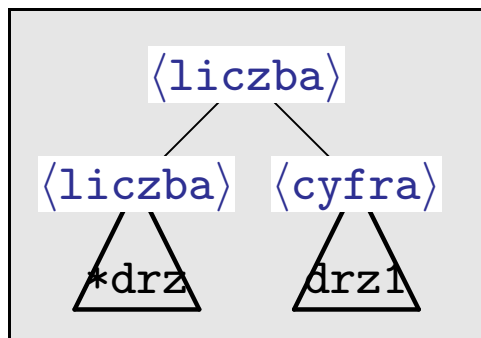
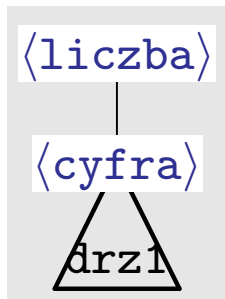


Leczenie lewostronnej rekursji

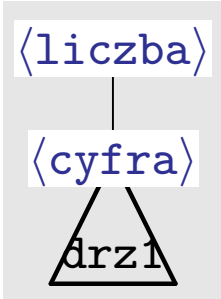
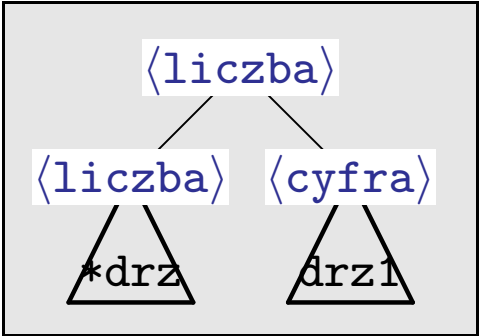
```

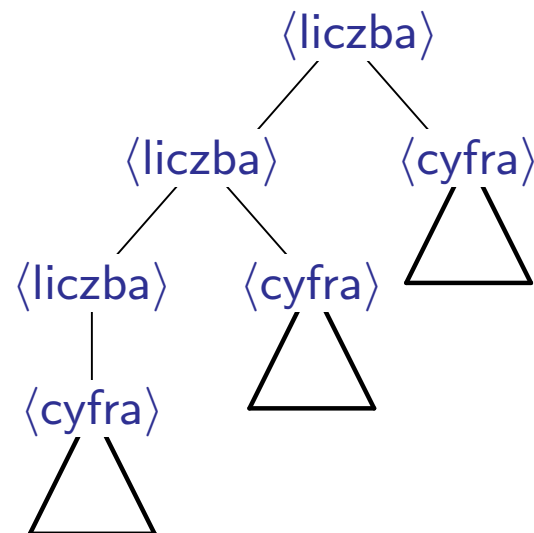
Boolean liczba (drzewo* drz) {
    if (cyfra(&drz1)) {
        *drz =
            while (cyfra(&drz1))
                *drz =
                    return TRUE;
    }
    else return FALSE;
}

```



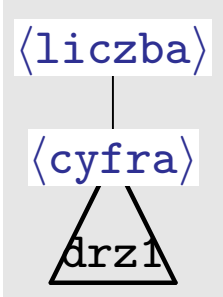
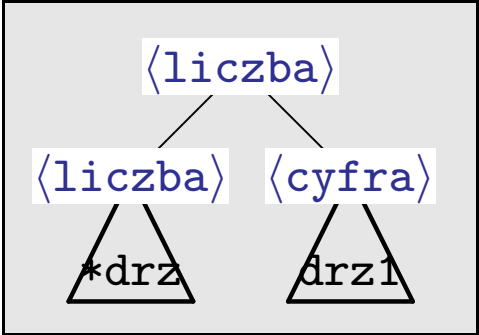
Leczenie lewostronnej rekursji

```
Boolean liczba (drzewo* drz) {
    if (cyfra(&drz1)) {
        *drz =
            
        while (cyfra(&drz1))
            *drz =
                
        return TRUE;
    }
    else return FALSE;
}
```

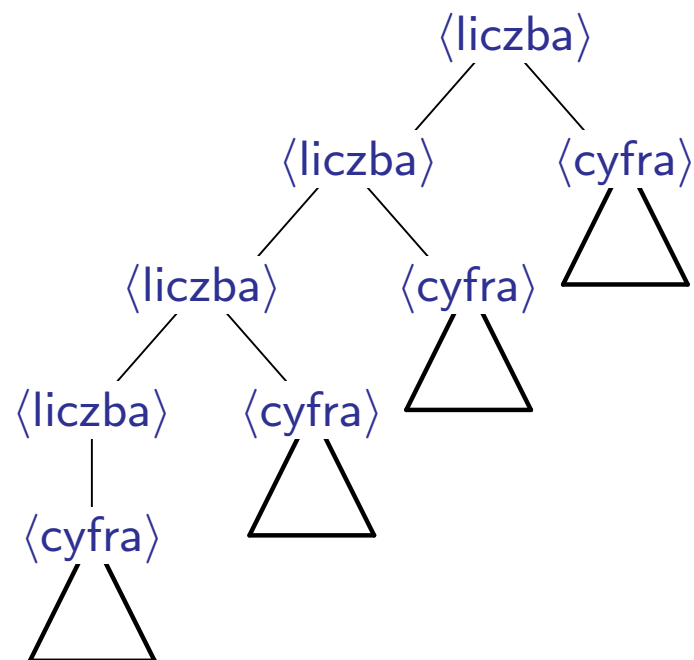


Leczenie lewostronnej rekursji

```

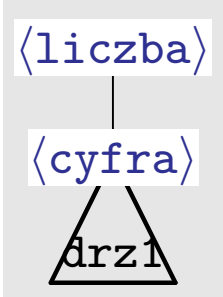
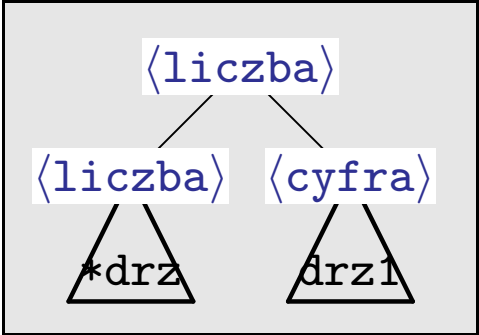
Boolean liczba (drzewo* drz) {
    if (cyfra(&drz1)) {
        *drz =
            
        while (cyfra(&drz1))
            *drz =
                
            ;
        return TRUE;
    }
    else return FALSE;
}

```

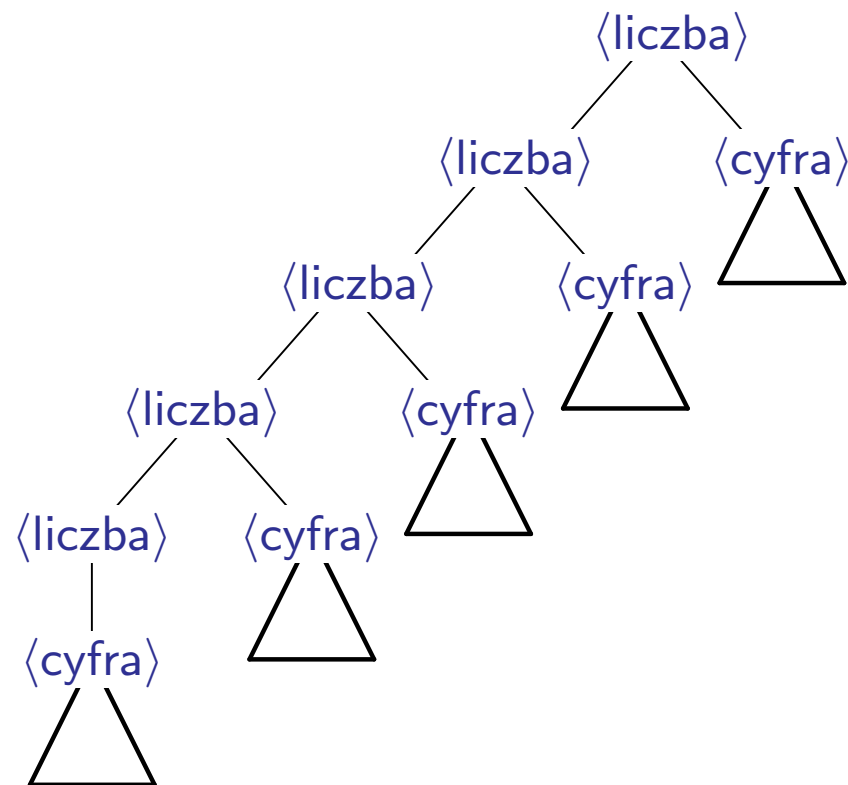


Leczenie lewostronnej rekursji

```

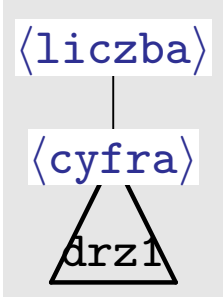
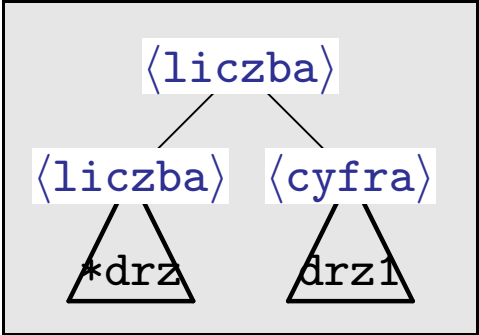
Boolean liczba (drzewo* drz) {
    if (cyfra(&drz1)) {
        *drz =
            
        while (cyfra(&drz1))
            *drz =
                
            ;
        return TRUE;
    }
    else return FALSE;
}

```

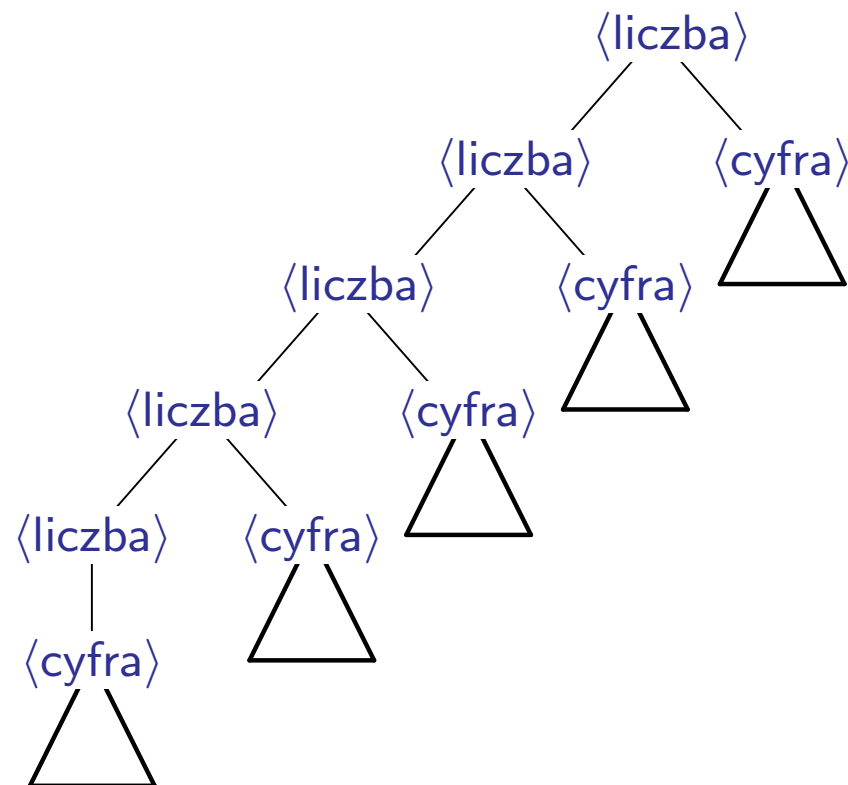


Leczenie lewostronnej rekursji

```

Boolean liczba (drzewo* drz) {
    if (cyfra(&drz1)) {
        *drz =
            
        while (cyfra(&drz1))
            *drz =
                
            ;
        return TRUE;
    }
    else return FALSE;
}

```



Parser zstępujący rekursywny — kiedy nie działa?

PRZYKŁAD:

$$\begin{aligned}
 \langle \text{miara} \rangle &::= \langle \text{odległość} \rangle \mid \langle \text{waga} \rangle \\
 \langle \text{odległość} \rangle &::= \{ 0 \mid \dots \mid 9 \} \{ 0 \mid \dots \mid 9 \}^* \text{ m} \\
 \langle \text{waga} \rangle &::= \{ 0 \mid \dots \mid 9 \} \{ 0 \mid \dots \mid 9 \}^* \text{ kg}
 \end{aligned}$$

Parser zstępujący rekursywny — kiedy nie działa?

PRZYKŁAD:

$$\begin{aligned}
 \langle \text{miara} \rangle &::= \langle \text{odległość} \rangle \mid \langle \text{waga} \rangle \\
 \langle \text{odległość} \rangle &::= \{ 0 \mid \dots \mid 9 \}^* \text{ m} \\
 \langle \text{waga} \rangle &::= \{ 0 \mid \dots \mid 9 \}^* \text{ kg}
 \end{aligned}$$

$$\langle \text{miara} \rangle \Rightarrow \langle \text{odległość} \rangle \Rightarrow^* 11111\text{m}$$

$$\langle \text{miara} \rangle \Rightarrow \langle \text{waga} \rangle \Rightarrow^* 11111\text{kg}$$

Parser zstępujący rekursywny — kiedy nie działa?

PRZYKŁAD:

$$\begin{aligned}
 \langle \text{miara} \rangle &::= \langle \text{odległość} \rangle \mid \langle \text{waga} \rangle \\
 \langle \text{odległość} \rangle &::= \left\{ 0 \mid \dots \mid 9 \right\} \left\{ 0 \mid \dots \mid 9 \right\}^* \text{m} \\
 \langle \text{waga} \rangle &::= \left\{ 0 \mid \dots \mid 9 \right\} \left\{ 0 \mid \dots \mid 9 \right\}^* \text{kg}
 \end{aligned}$$

$$\langle \text{miara} \rangle \Rightarrow \langle \text{odległość} \rangle \Rightarrow^* 11111\text{m}$$

$$\langle \text{miara} \rangle \Rightarrow \langle \text{waga} \rangle \Rightarrow^* 11111\text{kg}$$

Do którego nieterminala redukować? — nie da się postanowić na podstawie pierwszej cyfry, bo nie wiadomo jeszcze, czy na końcu będzie **m** czy **kg**.

Parser zstępujący rekursywny — kiedy nie działa?

PRZYKŁAD:

$$\begin{aligned}
 \langle \text{miara} \rangle &::= \langle \text{odległość} \rangle \mid \langle \text{waga} \rangle \\
 \langle \text{odległość} \rangle &::= \left\{ \begin{array}{c|c|c} 0 & \dots & 9 \end{array} \right\} \left\{ \begin{array}{c|c|c} 0 & \dots & 9 \end{array} \right\}^* m \\
 \langle \text{waga} \rangle &::= \left\{ \begin{array}{c|c|c} 0 & \dots & 9 \end{array} \right\} \left\{ \begin{array}{c|c|c} 0 & \dots & 9 \end{array} \right\}^* \text{kg}
 \end{aligned}$$

$$\langle \text{miara} \rangle \Rightarrow \langle \text{odległość} \rangle \Rightarrow^* 11111m$$

$$\langle \text{miara} \rangle \Rightarrow \langle \text{waga} \rangle \Rightarrow^* 11111\text{kg}$$

Do którego nieterminala redukować? — nie da się postanowić na podstawie pierwszej cyfry, bo nie wiadomo jeszcze, czy na końcu będzie m czy kg .

Parseery zstępujące nadają się nie do wszystkich gramatyk bezkontekstowych; nadają się tylko do t.zw. gramatyk **LL(1)**.