Stefan Sokołowski

# JĘZYKI FORMALNE I METODY KOMPILACJI

**wykład 13**

**Inst. Informatyki Stosowanej, ANS**
**Elbląg, 2024/2025**



## Automatyczna konstrukcja translatorów

**GRAMATYKA $\longrightarrow$ PARSER**
— to da się zrobić automatycznie

**GENEROWANIE PRZEKŁADU**
— z tym jest większy problem, bo skąd system automatyczny ma wiedzieć, jakie znaczenie chcemy nadać składni?

Nadajemy znaczenie każdej **produkcji** z gramatyki osobno — piszemy, co ma się dziać, kiedy parser dokonuje odpowiadającej jej redukcji.

**narzędzie: GRAMATYKA *ATRYBUTOWA***

## Gramatyki atrybutowe

***Gramatyka atrybutowa*** — gramatyka bezkontekstowa wzbogacona o

- ***atrybuty*** symboli w drzewie wywodu (terminali i nieterminali); atrybuty mogą mieć wartości;
- ***funkcje obliczające*** wartości atrybutów.

**atrybut *syntetyzowany*:**
z dzieci do rodziców

**atrybut *dziedziczony*:**
z rodziców do dzieci



## Gramatyki atrybutowe

**Przykład:** język $\left\{ a^n b^n c^n \,\middle|\, n \geqslant 0 \right\}$ —
nie daje się opisać gramatyką bezkontekstową;
daje się opisać gramatyką atrybutową.

$$
\begin{array}{rcl}
\langle \text{słowo} \rangle & ::= & \langle \text{część\_a} \rangle \langle \text{część\_b} \rangle \langle \text{część\_c} \rangle \\
\langle \text{część\_a} \rangle & ::= & \lambda \\
 & | & \langle \text{część\_a} \rangle \text{a} \\
\langle \text{część\_b} \rangle & ::= & \lambda \\
 & | & \langle \text{część\_b} \rangle \text{b} \\
\langle \text{część\_c} \rangle & ::= & \lambda \\
 & | & \langle \text{część\_c} \rangle \text{c}
\end{array}
$$

Nie ma jak sprawdzić, czy długości tych trzech części są równe.

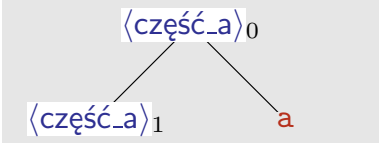
# Gramatyki atrybutowe

Do nieterminali dodajemy atrybut **syntetyzowany** (od dzieci do rodziców) *ile* liczący litery w poddrzewie poniżej tego nieterminalu:

$\langle \text{część\_a} \rangle \longrightarrow \lambda$


$\langle \text{część\_a} \rangle .ile \leftarrow 0$

$\langle \text{część\_a} \rangle \longrightarrow \langle \text{część\_a} \rangle \text{a}$

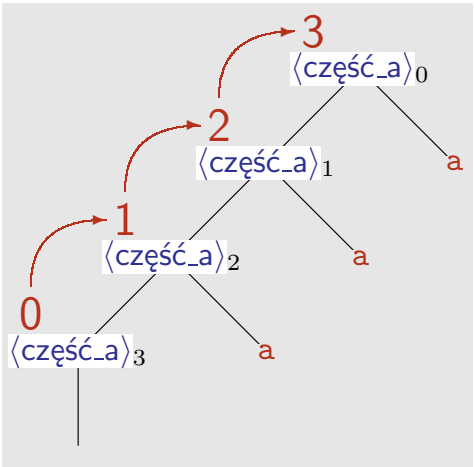



$\langle \text{część\_a} \rangle_0 .ile \leftarrow \langle \text{część\_a} \rangle_1 .ile + 1$



# Gramatyki atrybutowe

Funkcje obliczające powodują propagowanie wartości atrybutu po drzewie wywodu:



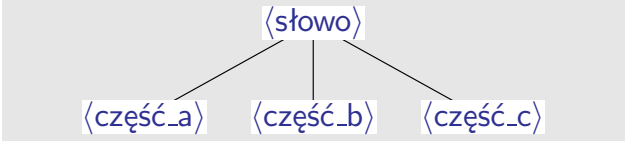

$\langle \text{część\_a} \rangle_0 .ile \leftarrow \langle \text{część\_a} \rangle_1 .ile + 1$

$\langle \text{część\_a} \rangle .ile \leftarrow 0$

# Gramatyki atrybutowe

W korzeniu predykat sprawdzający, czy długości równe:

⟨słowo⟩ → ⟨część_a⟩⟨część_b⟩⟨część_c⟩




Czy
⟨część_a⟩.ile = ⟨część_b⟩.ile
= ⟨część_c⟩.ile ?

---

B
I
S
O
N

## Kompilator kompilatorów

Prekursor:
**YACC** — *Yet Another Compiler Compiler*

Wersja GNU (działa na `unixlabie`):
**BISON** — `www.gnu.org/software/bison`
— stosuje atrybuty syntetyzowane (od liści do korzenia)



# Działanie BISONa — rozpoznawanie $a^n b^n c^n$

```
slowo:  czesc_a czesc_b czesc_c '.'
    { if ($1 == $2 && $1 == $3)  printf("\n  NALEZY.\n\n");
      else  printf("\n  NIE NALEZY.\n\n");
      exit(1);
    } ;
```

```
czesc_a:  /* puste */   { $$ = 0; }
        |  czesc_a 'a'   { $$ = $1+1; } ;

czesc_b:  /* puste */   { $$ = 0; }
        |  czesc_b 'b'   { $$ = $1+1; } ;

czesc_c:  /* puste */   { $$ = 0; }
        |  czesc_c 'c'   { $$ = $1+1; } ;
```

— gramatyka

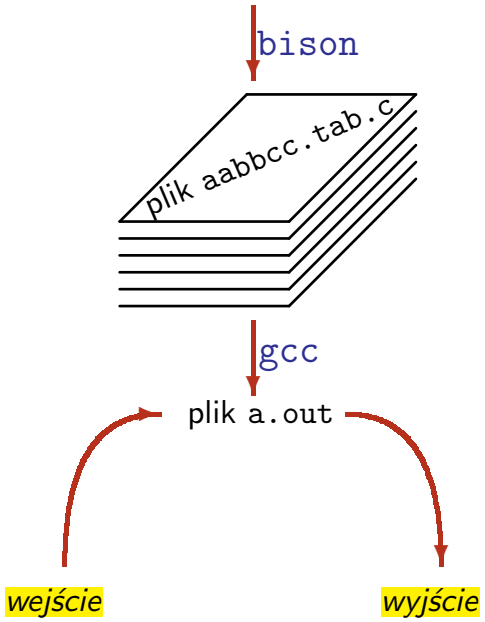
— działania na atrybutach

iis.ans-elblag.pl/~stefan/Dydaktyka/JezForm/Wyklady/13-bizony/1-aabbcc.y

# Działanie BISONa — przetwarzanie

plik z gramatyką aabbcc.y :

```
slowo: czesc_a czesc_b czesc_c '.' { ... }
czesc_a: /* puste */ { ... }
...
```






# Przykład: kalkulator

⟨program⟩ ::= ⟨komendy⟩ .

⟨komendy⟩ ::= (puste)
| ⟨komendy⟩ ⟨komenda⟩

⟨komenda⟩ ::= ⟨zmienna⟩ = ⟨wyrażenie⟩ ; ← { pamięć[$1] = $3; }
| ⟨wyrażenie⟩ ;

⟨wyrażenie⟩ ::= ⟨składnik⟩ | - ⟨składnik⟩
| ⟨wyrażenie⟩ + ⟨składnik⟩ | ⟨wyrażenie⟩ - ⟨składnik⟩

⟨składnik⟩ ::= ⟨potega⟩ | ⟨składnik⟩ * ⟨potega⟩
| ⟨składnik⟩ / ⟨potega⟩

⟨potega⟩ ::= ⟨podstawa⟩ | ⟨podstawa⟩ ^ ⟨potega⟩

⟨podstawa⟩ ::= ⟨liczba⟩ | ⟨zmienna⟩ | ( ⟨wyrażenie⟩ )

⟨liczba⟩ ::= ...

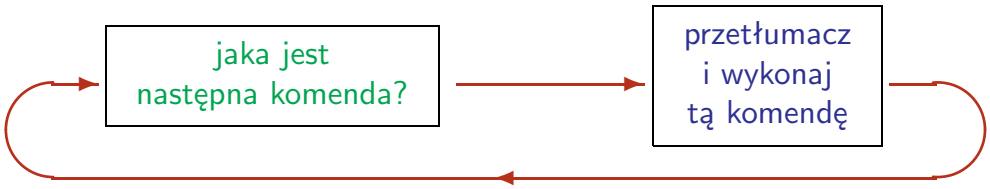
⟨zmienna⟩ ::= ... ← { $$ = numer zmiennej:
a == 0
b == 1
... }

{ $$ = pamięć[$1]; }

iis.ans-elblag.pl/~stefan/Dydaktyka/JezForm/Wyklady/13-bizony/3-pjo.y

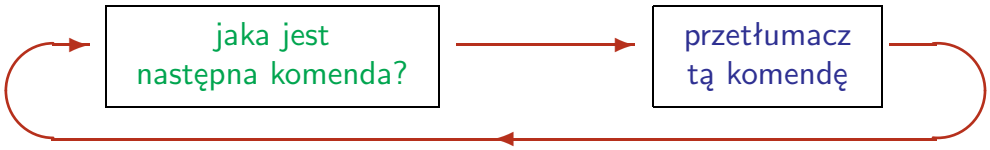
# Interpretacja a kompilacja

**Interpreter** — superprogram wykonujący poszczególne komendy programu jedna za drugą:




**Skutek:** wynik obliczenia.

**Kompilator** — superprogram tłumaczący cały program na język wewnętrzny, przygotowujący go do wykonania:




**Skutek:** program w języku wewnętrznym gotowy do wykonania..

Komendy wewnątrz pętli interpreter od nowa tłumaczy za każdym obrotem pętli; a kompilator tłumaczy raz, a potem przekład tylko wielokrotnie je wykonuje.



# Działanie BISONa — kompilacja

**Przykład:** „języczek"

⟨program⟩ ::= ⟨komendy⟩ .

⟨komendy⟩ ::= (puste)
| ⟨komendy⟩ ⟨komenda⟩

⟨komenda⟩ ::= ⟨zmienna⟩ = ⟨zmienna⟩ ;
| ⟨zmienna⟩ = ⟨zmienna⟩ + ⟨zmienna⟩ ;
| ⟨zmienna⟩ = ⟨zmienna⟩ − ⟨zmienna⟩ ;
| ⟨zmienna⟩ = ⟨liczba⟩ ;
| ⟨zmienna⟩ = ⟨zmienna⟩ + ⟨liczba⟩ ;
| ⟨zmienna⟩ = ⟨zmienna⟩ − ⟨liczba⟩ ;
| ( ⟨zmienna⟩ ) { ⟨komendy⟩ } (pętla)

⟨liczba⟩ ::= ...

⟨zmienna⟩ ::= ...

wyjaśnione dalej

# Działanie BISONa — kompilacja

Pętla: $(x)\ \{\ldots\}$ — dopóki $x \neq 0$, wykonuj $\ldots$

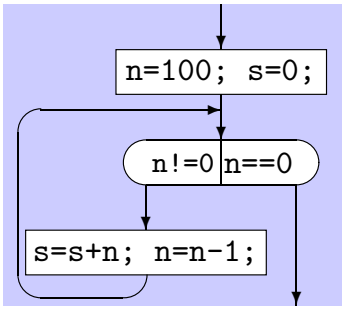
**Przykłady programów:**

```
n=100; s=0;
(n) {
  s=s+n; n=n-1;
}
.
```

dopóki $n \neq 0$

oznacza

```
n=100; s=0;
while (n!=0) {
  s=s+n; n=n-1;
}
```




— wylicza $s = \sum_{n=1}^{100} n$



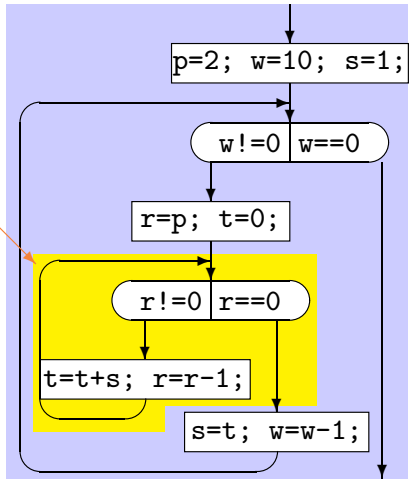
# Działanie BISONa — kompilacja

```
p=2; w=10; s=1;
(w) {
  r=p; t=0;
  (r) { t=t+s; r=r-1; }
  s=t; w=w-1;
}
.
```

oznacza

```
p=2; w=10; s=1;
while (w!=0) {
  r=p; t=0;
  while (r!=0) {
    t=t+s; r=r-1;
  }
  s=t; w=w-1;
}
```

pętla wewnętrzna




— wylicza $s = 2^{10}$